UNIVERSIDAD DE MURCIA

FACULTAD DE INFORMÁTICA

Model-Driven Modernisation of Legacy Graphical User Interfaces

Modernización Dirigida por Modelos de Interfaces Gráficas de Usuario

D. Óscar Sánchez Ramón

2014

Model-Driven Modernisation of Legacy Graphical User Interfaces

A dissertation presented by
Óscar Sánchez Ramón
and supervised by
J. García Molina & J. Sánchez Cuadrado

In partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the subject of Computer Science

University of Murcia
October 2014

Modernización Dirigida por Modelos de Interfaces Gráficas de Usuario

Resumen extendido de la tesis

Motivación

Actualmente numerosas empresas abordan la migración de los sistemas heredados (*legacy systems*) que disponen, con el fin de adaptarlos a nuevas tecnologías de software que ofrecen mejores características (por ejemplo, mayor facilidad de mantenimiento o mejor experiencia de usuario). Las interfaces gráficas de usuario (*Graphical User Interfaces, GUIs*) constituyen un elemento importante en dichas migraciones, dado que son el medio que los usuarios utilizan para interaccionar con el sistema. Además, la aparición en los últimos años de una gran variedad de dispositivos capaces de ejecutar aplicaciones (tabletas, teléfonos y televisiones inteligentes, etc.) ha repercutido en que el diseño de las interfaces de usuario se convierta en un reto mayor.

Un ejemplo típico de sistemas heredados son las aplicaciones creadas con entornos RAD (*Rapid Application Development*), tales como Oracle Forms y Microsoft Visual Basic, que gozaron de gran aceptación en los noventa. Nuestro trabajo se centrará en este tipo de aplicaciones, a las que nos referiremos como aplicaciones RAD. Éstas ofrecían un paradigma de programación centrado en la GUI que permitía la creación de ventanas en un tiempo reducido. Sin embargo, las aplicaciones RAD poseían dos características fundamentales que reflejan prácticas desaconsejadas en ingeniería del software. La primera característica es que la posición de los controles (tales como cajas de texto o etiquetas) estaba expresada con coordenadas. Esto constituye una mala práctica porque el cambio de posición de un control puede implicar la modificación de la posición de otros. Además, las interfaces expresadas con coordenadas sólo están optimizadas para una resolución y tamaño de ventana determinados (no se adaptan al tamaño de éstas), por lo que no se muestran adecuadamente cuando redimensionamos las ventanas o cuando se ejecuta la aplicación en dispositivos con pantallas de diferentes dimensiones. Por el contrario, en la actualidad se usan gestores de *layout* (*layout managers*) como *FlowLayout* o *BorderLayout* en Java Swing, que permiten adaptar el contenido a las dimensiones de la ventana.

La segunda característica consiste en que el código de los manejadores de eventos de la interfaz frecuentemente mezcla diferentes aspectos, desde aspectos arquitecturales como la lógica de negocio o la presentación, hasta aspectos funcionales como la validación de los formularios o el flujo de navegación entre las vistas de la aplicación. En la actualidad habitualmente se utilizan frameworks de desarrollo que fomentan la separación de aspectos porque facilitan en gran medida el mantenimiento y la extensibilidad de las aplicaciones, en contraste con las aplicaciones RAD que eran más difíciles de mantener.

En relación con la primera característica y su tratamiento en una migración, existen diversidad de trabajos que versan sobre ingeniería inversa de GUIs [1] [2], sin embargo en muchos de ellos la migración de las ventanas se limita a detectar controles y traducir dichos controles al toolkit de la tecnología destino. Especial relevancia tienen los trabajos [3] [4] y [5], que presentan tres enfoques que prestan atención al *layout* de las vistas (ventanas, páginas web, etc.) y que extraen un modelo que representa dicho *layout*. La desventaja fundamental de estos enfoques es que obtienen una única representación del *layout* (por ejemplo, usando el *GridBagLayout* de Java Swing), con lo cual, la generación de interfaces utilizando otros tipos de *layout* (por ejemplo, las capas flotantes de CSS) no es una labor directa.

Con respecto a la segunda característica, podemos encontrar varios trabajos centrados en el análisis de código de la GUI [6] [7] [8]. La gran mayoría se centra en extraer las transiciones que se producen entre las distintas vistas de la aplicación, que normalmente se representan con algún tipo de máquina de estados, con el objetivo de utilizar esta información para en el ámbito de la comprensión de programas (*program comprehension*) o para realizar pruebas unitarias.

Objetivo

Nuestro objetivo consiste en facilitar la migración de aplicaciones RAD a través de la creación de un framework de migración de GUIs de sistemas RAD heredados. El framework está destinado fundamentalmente a inferir el *layout* de la aplicación original y separar los aspectos que se encuentran entremezclados en los manejadores de eventos.

El análisis de varias aplicaciones creadas con entornos RAD y el estudio del trabajo relacionado condujeron a la extracción de una serie de requisitos que orientó el diseño de la solución, y que son los siguientes:

(R1) *Extracción explícita de información.* Es necesario obtener una representación explícita de alto nivel la información de la interfaz de usuario.

(R2) *Modularidad.* Es deseable fragmentar el proceso de reingeniería en etapas más sencillas para favorecer su mantenimiento.

(R3) *Automatización.* El proceso debe ser automatizado en la medida de lo posible.

(R4) *Independencia del y origen y el destino.* Debe ser posible extender el proceso y su reutilización con distintas tecnologías de origen/destino con un esfuerzo relativamente reducido.

(R5) *Asemejar la estructura visual y lógica.* La estructura lógica de las vistas, esto es, cómo están contenidos unos controles en otros, debe coincidir con la estructura que un usuario percibe al observar la vista.

(R6) *Representación de alto nivel.* El *layout* de la vista debe expresarse con construcciones de alto nivel, como por ejemplo los gestores de *layout* de Java Swing, que controlan la disposición espacial de componentes en una ventana.

(R7) *Tolerancia a controles desalineados.* La solución debe manejar la situación en que los controles se encuentren levemente desalineados.

(R8) *Soluciones alternativas.* Un mismo *layout* puede lograrse con varias combinaciones distintas de gestores de *layout*, y sería deseable que los desarrolladores pudieran conocer esas alternativas.

(R9) layout *configurable.* El conjunto de gestores de *layout* a utilizar debe ser parametrizable.

(R10) *Abstracción de código.* El código se debe abstraer para facilitar su análisis. Dado que el código de los manejadores de eventos responde a una serie de patrones recurrentes, sería interesante detectar esos patrones para abstraer el código.

(R11) *Categorización de código.* Es necesario que sea posible identificar los distintos aspectos arquitecturales del código de la aplicación, esto es, el código de la lógica de negocio, de los controladores y de la interfaz de usuario.

(R12) *Identificación de las interacciones y flujos de navegación.* La solución deber permitir además identificar otros aspectos, como las interacciones que existen entre los controles (por ejemplo, que al marcar una casilla de verificación se permita editar un determinado campo de texto) o el flujo de navegación entre las distintas vistas de la aplicación.

Desarrollo de la arquitectura del framework

Nuestro framework ha sido construido aplicando la Ingeniería del Software Dirigida por Modelos (*Model-Driven Engineering, MDE*) que se caracteriza por utilizar modelos a varios niveles de abstracción para representar diversos aspectos del sistema, con el fin de obtener una automatización en el proceso de desarrollo. En nuestro caso, MDE aporta a nuestra solución dos principales beneficios : la representación de aspectos del sistema heredado mediante modelos y metamodelos, y la automatización del proceso y modularidad de la solución por medio de cadenas de transformaciones que incluyen transformaciones modelo-a-modelo, modelo-a-código y código-a-modelos.

La arquitectura de modelos que hemos diseñado incluye dos modelos que independizan la solución de la tecnología origen (el modelo GUI normalizado y el modelo de comportamiento RAD), y una serie de modelos de interfaz de usuario concreta (*CUI, Concrete User Interface*) que aportan independencia de la tecnología destino. Hemos definido varios modelos de CUI, de modo que cada uno de ellos trata un aspecto diferente (aquí no nos referimos a aspectos arquitecturales), con lo que se fomenta la separación de aspectos. Los modelos CUI implementados son:

- *Modelo de estructura*: muestra la estructura lógica de las vistas, esto es, muestra las partes distinguibles de las vistas y los controles que contienen.
- *Modelo de* layout: representa la disposición espacial de los controles que contiene la vista en términos de gestores de *layout*.
- *Modelo de separación de aspectos*: expresa el código de los manejadores de eventos mediante patrones de código y etiqueta dicho código con el aspecto arquitectural al que corresponde (lógica de negocio, GUI, o controlador).
- *Modelo de interacciones*: expresa las dependencias entre los controles de la interfaz, así como el flujo de navegación que existe en las diferentes vistas de la aplicación.

Inferencia del *layout*

La inferencia del *layout* de las vistas consta de tres fases: i) extracción de regiones, ii) representación de relaciones espaciales relativas, y iii) descubrimiento del *layout* de alto nivel. Se han desarrollado dos versiones del proceso de inferencia del *layout*. En la primera versión se abordaron las tres fases mencionadas, siendo la última de ellas implementada mediante una aproximación heurística. En la segunda versión se sustituyó el algoritmo de la tercera fase por un algoritmo exploratorio, más sofisticado que en la primera versión, lo que conllevó también a realizar modificaciones en la segunda fase.

En las aplicaciones RAD pueden existir controles simples (no contenedores, como los botones) que no se encuentren contenidos en controles contenedores (por ejemplo, paneles), sino que se encuentren solapados con estos. El proceso de extracción de regiones (primera fase de la inferencia del *layout*) en primer lugar hace explícita esta relación de contención entre los controles. Para lograr esto, se crea una región para cada control, y para aquellos controles que visualmente tienen borde y contienen a otros controles simples, se añaden las regiones de estos últimos a la región del control que los contiene visualmente. En segundo lugar, la extracción de regiones evita que existan controles simples al mismo nivel que controles contenedores. Para ello, crea regiones nuevas que contienen las regiones de aquellos controles no contenedores que están al mismo nivel que controles contenedores. Al final de este proceso se tiene la vista organizada en un árbol de regiones, donde la estructura lógica concuerda con la estructura visual.

La segunda fase de la estrategia de inferencia del *layout* es la representación de relaciones espaciales relativas a partir de la información de las regiones. En esencia trata de expresar las relaciones entre controles contiguos por medio de un grafo de posiciones relativas, donde los vértices son los controles y las aristas son las relaciones espaciales. La implementación de este grafo ha variado entre la primera y la segunda versión del proceso. En la primera versión se representa explícitamente la posición entre dos controles mediante las relaciones arriba, abajo, izquierda, derecha, y una distancia significativa entre los controles se representa por medio de vértices especiales denominados huecos. En la segunda versión se optó por representar la posición entre dos controles por medio de dos intervalos Allen [9], uno para el eje X y otro para el eje Y, y la distancia entre los nodos se mide en niveles discretos que se calculan dinámicamente aplicando técnicas de agrupamiento (*clustering*).

La tercera fase obtiene el diseño expresado por medio de una composición de gestores de *layout*. En la primera versión se implementó un algorítmico heurístico basado en el encaje de patrones. Se definió un patrón para cada tipo de gestor de *layout*, así como una función de idoneidad que, aplicada a un conjunto de nodos del grafo de posiciones relativas, devuelve el porcentaje de nodos encajados en el patrón. El modo de funcionamiento es el siguiente: para cada grafo de posiciones relativas que proviene de una región contenedora se aplican las funciones de idoneidad de todos los gestores de *layout*, y se aplica el patrón asociado a aquella función que obtiene un valor más alto. Este algoritmo tiene un inconveniente de especial relevancia: no permite detectar patrones anidados, con lo cual, las vistas que tienen un diseño complejo en muchas ocasiones no serán reconocidas correctamente.

La segunda versión del descubrimiento de alto nivel utiliza un algoritmo exploratorio que se basa en el encaje de patrones y la reescritura del grafo de posiciones relativas. Cada gestor de *layout* tiene un patrón asociado. El algoritmo en primer lugar genera todas las secuencias de gestores de *layout* posibles, e intenta llegar a una solución aplicando cada secuencia. Para cada secuencia, se aplican los patrones sobre el grafo en el orden indicado por ésta, de modo que cuando un patrón encaja en un subgrafo, éste se reemplaza por un único nodo. Se continúa aplicando el proceso de encaje de patrones y reescritura del grafo hasta que queda un único nodo, lo que denota que hemos alcanzado una solución. Si sucede que tras un número adecuado iteraciones no se han producido cambios en el grafo, entonces se detiene la búsqueda pues no es posible hallar una solución con esa secuencia. Cada solución obtenida es evaluada por una función de idoneidad que nos indica cómo de buena es la solución hallada. Al final del proceso se tiene un modelo que indica una serie de posibles *layouts* para cada contenedor de la vista, y también nos indica cuál es el mejor *layout* de acuerdo con la función de idoneidad. El conjunto de gestores de *layout* utilizados en la solución es configurable, con lo que es posible limitar o extender el mismo según las características de la tecnología destino.

Desarrollo del enfoque de análisis de manejadores de eventos

Hemos desarrollado una solución para separar los aspectos que se encuentran mezclados en los manejadores de eventos. Concretamente abordamos la separación de los aspectos arquitecturales de la aplicación (lógica de negocio, controlador e interfaz de usuario), así como la extracción de las interacciones que existen entre los controles y entre las vistas de la GUI.

Para alcanzar este objetivo realizamos una fase de abstracción del código previa a la separación de aspectos. La abstracción consiste en representar el código fuente de los manejadores de eventos en términos de primitivas que expresan patrones de código comunes en las aplicaciones RAD. Por ejemplo, Oracle Forms utiliza el lenguaje PL/SQL para implementar los manejadores de eventos, y en este lenguaje se puede hacer uso de cursores para el acceso a base de datos. Nosotros simplificamos dichas instrucciones de apertura y lectura del cursor explícito con una primitiva que indique una lectura de base de datos. Algunas de las primitivas que hemos definido son: lectura de base de datos, escritura en un control o invocación a una función de lógica de negocio. El código expresado de este modo es más sencillo de analizar que el código fuente.

El código representado por medio de primitivas es entonces analizado para separar los aspectos arquitecturales, obteniéndose el modelo de separación de aspectos. Para tal fin, las primitivas se dividen en bloques básicos [10] que se estructuran formando un grafo de control de flujo. Cada bloque básico a su vez se divide en fragmentos, que son conjuntos de instrucciones relacionadas que pertenecen al mismo aspecto (lógica de negocio, controlador o GUI), y que por tanto deben ser migradas conjuntamente. Los fragmentos se obtienen analizando el tipo de las primitivas y las variables de entrada y salida que poseen. Gracias a que las primitivas guardan referencias al código original, es posible utilizar el grafo de flujo de fragmentos para clasificar el código original y guiar la migración a una arquitectura de capas.

Las primitivas también se utilizan en la identificación de interacciones entre los controles y entre las vistas. Se analiza recursivamente el flujo de control de las primitivas para extraer: i) los controles que generan los eventos, ii) las condiciones en las cuáles se disparan los eventos, iii) los controles en los que se produce un efecto, y iv) el efecto producido sobre éstos. Por ejemplo, seleccionar una opción determinada de una lista desplegable puede producir que se habilite un formulario que antes no se mostraba. Con esta información se construye un grafo multi-nivel donde los vértices son los controles y las vistas, y las aristas son las interacciones entre ellos. El grafo es multi-nivel porque un vértice que represente una vista contendrá a su vez el grafo formado por los controles que forman parte de ella. Este grafo puede ser de utilidad para documentar el sistema, generar artefactos que describan el flujo de navegación entre las vistas, o detectar llamadas asíncronas en un entorno web con Ajax.

Evaluación

Las dos versiones de la solución de inferencia del *layout* han sido evaluadas. En la primera versión se realizó mediante un caso de estudio de migración de dos aplicaciones Oracle Forms a Java. El proceso de evaluación básicamente consistió en generar automáticamente el código Java y analizar manualmente las ventanas obtenidas. Particularmente se midió el porcentaje de partes distinguibles que habían sido colocadas correctamente, así como el porcentaje de controles situados en el lugar correcto. En el posicionamiento de partes se obtuvo una tasa de éxito del 96% y 97% en cada una de las aplicaciones, y el porcentaje de controles correctos fue de 87% y 95% en cada una. El caso de estudio reveló varias limitaciones de la primera versión del enfoque, siendo particularmente destacable la incapacidad para detectar *layouts* complejos (que no pueden ser expresados con un único gestor de *layout*).

La segunda versión se diseñó para paliar las limitaciones de la primera versión. En este caso, la aproximación se testeó en un escenario diferente a la migración, concretamente la generación de una nueva interfaz web a partir de esbozos (*wireframes*) creados con alguna herramienta para tal efecto. La evaluación se llevó a cabo con profesionales de las TICs que siguieron el siguiente proceso: leer una breve documentación de la aplicación propuesta, realizar los esbozos de la GUI, generar automáticamente el código, analizar los resultados y rellenar un cuestionario. El 85% de los participantes indicaron que las vistas se habían generado totalmente o en gran medida como ellos esperaban, el 65% estaban totalmente o parcialmente de acuerdo en que las ventanas generadas podían usarse en aplicaciones reales, y el 90% estuvieron de acuerdo en que la herramienta es útil. Las características de nuestra solución que incidieron negativamente en el resultado fueron dos: i) la configuración de los parámetros del algoritmo, que en algunos casos era vital para obtener el resultado adecuado, y ii) la función de idoneidad, que obtenía buenas soluciones en cuanto al número de gestores de *layout* empleados, pero no siempre obtenía la mejor solución desde el punto de vista visual.

Para comparar la segunda versión con la primera se evaluó el nuevo algoritmo con una de las aplicaciones del caso de estudio de Oracle Forms, obteniéndose un 99% de acierto en la organización de las partes y un 97% en el posicionamiento de controles. El hecho de aplicar el enfoque de inferencia del *layout* en dos escenarios diferentes nos sirve para demostrar que la solución es aplicable en cualquier caso en que se disponga de una interfaz donde los controles se posicionan con coordenadas.

La evaluación de la separación de aspectos estructurales de los manejadores de eventos se llevó a cabo con un caso de estudio de migración de una aplicación Oracle Forms a una arquitectura cliente-servidor de 2 capas, donde la capa de presentación se implementaba en el navegador, y la lógica de negocio permanecía en el servidor y se exhibía al cliente mediante un servicio REST. Este caso de estudio nos permitió evaluar también el enfoque de abstracción de código, en el que el 96% del código fue encajado en alguno de los patrones definidos, y se obtuvo una tasa de código correctamente transformado en primitivas del 83%. La tasa de error del 17% fue ocasionada por ciertos elementos del código PL/SQL que no se tratan en la implementación actual, como las excepciones, y otras funciones específicas de Oracle Forms que no se traducen correctamente. Con respecto a la separación de aspectos, se obtuvo un 86% de código correctamente clasificado, lo que demuestra que ésta es altamente dependiente del éxito del proceso de abstracción del código.

Conclusiones

La arquitectura MDE que hemos desarrollado nos ha permitido solventar los requisitos R1, R2, R3 y R4. Concretamente la representación explícita de la información (R1) se ha logrado por medio de metamodelos, la modularidad (R2) y la automatización (R3) se han conseguido mediante cadenas de transformaciones, y la independencia del origen y el destino (R4) se ha obtenido gracias a los metamodelos diseñados para tal efecto.

El requisito de asemejar la estructura lógica y visual (R5) se cubre mediante el modelo de regiones. La representación de alto nivel (R6) se logra mediante el modelo de *layout*. La tolerancia a controles desalineados (R7), las soluciones alternativas (R8) y el requisito de diseño configurable (R9) se ha conseguido implementando un algoritmo de inferencia parametrizable. Cabe destacar que no se han encontrado trabajos que planteen una solución a los requisitos R8 y R9, dado que los trabajos existentes presentan algoritmos ad-hoc para generar *layouts* compuestos por un gestor de *layout* [3] [4] [5].

La abstracción de código (requisito R10) se ha logrado mediante el modelo de primitivas de comportamiento abstracto, la categorización de código (R11) se ha conseguido a través del grafo de flujo de fragmentos de código (modelos de separación de aspectos), y el requisito de identificación de interacciones y flujos de navegación ha sido obtenido por el modelo de interacciones. No hemos hallado ningún trabajo relacionado que utilice una representación similar para abstraer código. Con respecto al requisito R11, los trabajos existentes separan la aplicación en capas [11], pero requieren asistencia del desarrollador, mientras que en nuestra solución este proceso ha sido automatizado.

Contribuciones

Las contribuciones de esta tesis son fundamentalmente tres. La primera es una arquitectura de modelos que puede ser utilizada para migrar aplicaciones RAD. Esta arquitectura posee una serie de características (reusabilidad, extensibilidad, mantenibilidad) muy útiles para la migración. Además, como parte de esa arquitectura destacamos el diseño del modelo CUI, que favorece la separación de aspectos en el desarrollo de una GUI . La segunda aportación es la estrategia de inferencia del *layout*, de la cual se proponen dos versiones. El enfoque propuesto permite inferir diversas opciones de *layout* en base a un conjunto de gestores de *layout* parametrizable, y que puede ser utilizado no solo en un escenario de migración sino también de ingeniería directa, como la generación de código a partir de wireframes de la GUI. La tercera contribución es la solución de análisis de código de los manejadores de eventos para separar los diferentes aspectos que se encuentran mezclados en el código, tanto arquitecturales como otros tales como las interacciones entre los controles de la vista.

Agradecimientos

Muchas veces he soñado con este momento. Siempre me he preguntado cómo me sentiría en este instante, que significa el final de una etapa para mí. Son varios años de trabajo, mucho esfuerzo condensado en un documento, y mucha gente que de una manera u otra me ha apoyado y me ha ayudado a seguir adelante.

Quiero empezar dedicando unas palabras de agradecimiento a mis padres Juan y Soledad, que siempre han velado porque me centrara en los estudios y nunca me faltase de nada. Gracias también a mis hermanos Juan Miguel y Marisol que siempre me han apoyado y me han demostrado que están ahí, y a Laura, Dani, Álvaro y Héctor, que endulzan nuestra familia con su inocencia y alegría.

Jesús García Molina y Jesús Sánchez Cuadrado, mis directores de tesis y amigos, han sido piezas clave para superar con éxito esta odisea. Jesús García me acogió en su grupo allá en 2006, y me enseñó que los modelos no solo desfilan por las pasarelas. Años más tarde, Jesús Sánchez aceptó unirse al carro de las interfaces de usuario y se unió a Jesús García para guiarme por el tortuoso e incierto mundo de la investigación. ¡La de veces que habré maldecido RubyTL!... (y que posteriormente he alabado). A ambos les debo mi formación, y les agradezco el esfuerzo y tiempo que han invertido en mí.

En mi camino de investigación se han cruzado muchos compañeros que han dejado huella. Empecé trabajando en el grupo de investigación desarrollando *wrappers* de código con Javier Cánovas, que se sentaba en la mesa contigua, y tantas veces me ha escuchado y soportado. En aquel momento integraban también el laboratorio Jesús Sánchez, Fernando Molina, Francisco Javier Lucas, Joaquín Lasheras, Miguel Ángel Martínez, y posteriormente llegaron Espinazo, Javier Bermúdez, Jesús Perera y Juanma. Me vienen a la memoria el descubrimiento del Musicovery, el secuestro del peluche, la escena del electricista, la escala Cuadrado, las JISBD en Gijón... Gracias a todos por los buenos ratos que pasamos, en los que me enseñásteis otra forma de 'investigar'.

De mi estancia en Bélgica en 2012 guardo gratos recuerdos. Pese a vivir la primavera más nublada que había visto en mi vida, mis compañeros de laboratorio François Beauvens y Jérémie Melchior, que se pasaban los lunes discutiendo del Madrid y el Barça, me hacían más llevaderas las frías mañanas de Louvain-La-Neuve. Allí conocí también a Vivian, Ugo, Diana, Cinthya, Diogo, Sophie, Nesrine, Mathieu, Edu y otros tantos que me han demostrado que tengo amigos distribuidos por el mundo. Quiero hacer una mención distinguida a mi supervisor en Bélgica, Jean Vanderdonckt, que sin conocerme prácticamente de nada me otorgó la posibilidad de realizar la estancia.

No quiero olvidarme tampoco de mis amigos Edu, Anabel, Pablo, Laura, Daniel, Carras, la peña La Jarra, los monitores del campamento, y de los últimos visitantes del laboratorio, Saad, Manal y Sofia. Ellos han sufrido mis inquietudes y preocupaciones, y han sido de un modo u otro, testigos de mis logros y mis fallos durante el transcurso del doctorado.

A todas y cada una de las personas citadas, gracias.

Model-Driven Modernisation of Legacy Graphical User Interfaces

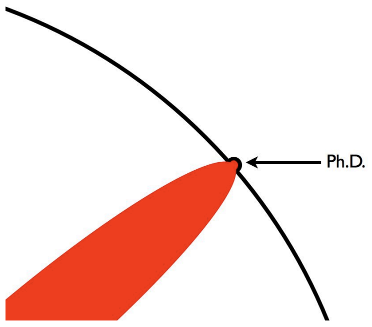
Abstract

Businesses are more and more modernising the legacy systems they developed with Rapid Application Development (RAD) environments, so that they can benefit from new platforms and technologies. As a part of these systems, Graphical User Interfaces (GUIs) pose an important concern, since they are what users actually see and manipulate. When facing the modernisation of GUIs of applications developed with RAD environments, developers must deal with two non-trivial issues. The first issue is that the GUI layout is implicitly provided by the position of the GUI elements (i.e. coordinates). However, taking advantage of current features of GUI technologies often requires an explicit, high-level layout model. The second issue is that developers must deal with event handling code that typically mixes concerns such as GUI and business logic. In addition, tackling a manual migration of the GUI of a legacy system, i.e., re-programming the GUI, is time-consuming and costly for businesses.

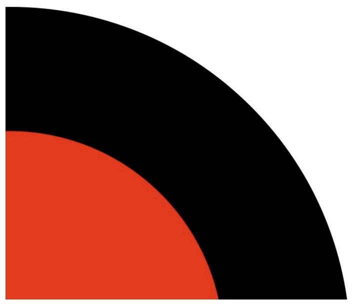
This thesis is intended to address these issues by means of an MDE architecture that automates the migration of the GUI of applications created with RAD environments. To deal with the first issue we propose an approach to discover the layout that is implicit in widget coordinates. The underlying idea is to move from a coordinate-based positioning system to a representation based on relative positions among widgets, and then use this representation to infer the layout in terms of layout managers. Two versions of this approach have been developed: a greedy solution and a more sophisticated solution based on an exploratory algorithm. To deal with the second issue we have devised a reverse engineering approach to analyse event handlers of RAD-based applications. In our solution, event handling code is transformed into an intermediate representation that captures the high-level behaviour of the code. From this representation, separation of concerns is facilitated. Particularly it has allowed us to achieve the separation of architectural concerns from the original code, and the identification of interactions among widgets. All the generated models in the reverse engineering process have been integrated into a Concrete User Interface (CUI) model that represents the different aspects that are embraced by a GUI.

The two layout inference proposals and the event handler analysis have been tested with real applications that were developed in Oracle Forms. The exploratory version of the layout inference approach was in addition tested with wireframes, which poses a different context in which the layout inference problem is also useful.

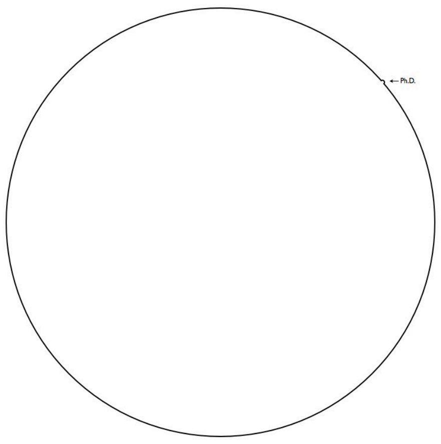
[...] You push at the boundary for a few years.
Until one day, the boundary gives way.
And, that dent you've made is called a Ph.D.



Of course, the world looks different to you now:



So, don't forget the bigger picture.



Keep pushing.[1]

[1]http://matt.might.net/articles/phd-school-in-pictures/

Contents

Listing of figures

List of Tables

Straight ahead of him, nobody can go very far...

Antoine de Saint-Exupéry, The Little Prince
(Suggested by Daniel Medina)

1 Introduction

Graphical User Interfaces (GUIs) represent a crucial part of software systems as they are what users actually see and manipulate to interact with them. Therefore, the design and implementation of GUIs is an issue that must not be neglected and developers typically devote a great effort to build application GUIs. Lately there has been a significant growth of types of devices that can run applications (smartphones, tablets, televisions, and so forth), each one having different screen sizes, resolutions and even interaction modalities (e.g., tactile screens). GUI technologies have also evolved to offer new possibilities that improve the user experience, particularly in the web setting with the emergence of HTML5 and Ajax (Asynchronous JavaScript And XML). This variety of devices and sophistication in technologies has brought that GUI design is now more challenging than ever. In fact, companies spend large amounts of money of their budget creating interfaces that must be functional, appealing and, at the same time, usable, because they are aware that this is key to succeed in their business. The challenge of creating quality GUIs does not only concern to the development of new applications, but is also faced at present by companies that are migrating their legacy applications to modern technologies as they offer a better user experience.

Software modernisation refers to understanding and evolving existing software assets to maintain their business value. A legacy system is modernised when maintenance is not enough to achieve the desired improvements (e.g., new capabilities or greater maintainability) and that system must be extensively changed. Software migration is a form of modernisation that involves moving an application, as a whole or a part of it, from the platform on which is currently operating to a target platform that provides better features. A migration can be done in a disciplined way by applying a software reengineering process that consists of three stages: reverse engineering the legacy system to obtain a representation of the system at a higher abstraction level, restructuring these representations according to the new architecture, and finally creating code of the new system from the restructured information [12] [13]. Reverse engineering techniques are therefore essential to understand and obtain representations at a high level of abstraction when a reengineering process is applied.

GUI migration has been typically regarded as a straightforward research topic, in which the only concern is to establish mappings between widgets of the source and target technologies. However, dealing with current technologies and devices requires a thorough analysis of the user interface so that it can be suitably reengineered. This analysis affects both the structural and behavioural aspects of a GUI, and sophisticated reverse engineering algorithms must be designed to cope with it.

Model Driven Software Engineering (MDSE or simply MDE) has emerged as a new area of software engineering that emphasizes the systematic use of models in the software lifecycle in order to improve its productivity and software quality aspects such as maintainability and interoperability. MDE techniques, e.g. metamodeling and model transformations, allow tackling the complexity of software by raising its abstraction and automation levels [14]. These techniques are useful not only for developing new software applications [15] [16] but also for reengineering legacy systems [17] [18] and dynamically configuring running systems [19]. In the latest years, MDE techniques have been applied to a variety of modernisation scenarios, especially in the migration of applications [20] [21] and some MDE tools have been created [22] [23] [24]. A notable effort is the Architecture-Driven Modernization (ADM) initiative [17], which was launched in 2003 and is targeted at offering a set of standard metamodels for representing information that is frequently implicated in modernisation. Although MDE is increasingly gaining acceptance in the software community [25], "the adoption of this approach has been surprisingly slow" [26] and there is still a need for successful experiences of using MDE in real projects.

The purpose of this thesis is to bring together the fields of Reengineering, Reverse Engineering, Model Driven Engineering and Graphical User Interfaces (GUIs) in order to encompass them all and create a solution for migrating GUIs of legacy systems to modern frameworks and technologies. In particular, we have designed and implemented a solution for migrating applications created with Rapid Application Development (RAD) environments, but the proposed approach is applicable to other legacy systems sharing the same requirements we have considered for RAD-based applications.

The rest of this chapter is organised as follows: first, the motivation of the work is presented; then, the goals of this thesis are outlined; afterwards, the development of the solution is explained and the main contributions of the thesis are enumerated; finally, the contents of the rest of this manuscript are summarised.

1.1 Motivation

Most information systems dating from the 90's were built using RAD environments. The RAD paradigm appeared in the early 90's as a response to the non-agile development processes that existed [27], and a number of Integrated Development Environments (IDEs) supporting fourth generation languages (4GLs) for the RAD paradigm also appeared. Oracle Forms, Visual Basic or Delphi are well-known examples of RAD environments. These IDEs provided a programming paradigm centered on the application GUI, allowing developers to create initial prototypes rapidly and reducing development time by facilitating GUI design and coupling data access to graphical components. However, the gaining of productivity is achieved at the expense of reducing the software quality. Next, we discuss two features of applications that have been created with a RAD environment (hereinafter referred as *RAD applications*), which negatively affect the software quality: the use of coordinates and the tangling of concerns.

In RAD environments, the position of widgets was expressed in terms of abolute or relative coordinates (normally pixels), so the windows created with them were optimised just for a certain size. Nonetheless, this is a bad practice since the interfaces are difficult to maintain. Let us consider a GUI defined by coordinates and a change consisting of adding a new widget. That change may lead developers to shift the coordinates of other widgets.

Furthermore, designing user interfaces for a fixed resolution and screen format is no longer admissible. With the popularisation of smartphones and tablets, there has been a explosive growth of devices that can run graphical applications (either natively or by means of a web browser).

Therefore, applications can be executed on a variety of devices with different features such as screen size, computing capacity or modality (e.g. tactile or voice) that produce different user experiences. Developers have now to meet the challenge of implementing GUIs that can be accessed via different devices with different screen features. As a result, in the last few years, flexible interfaces (non-fixed layouts) have gained in popularity due to the fact that designing different interfaces for the same application but targeted at different devices is impractical. Layout managers came up in the late nineties to overcome the weaknesses of coordinated-based GUIs by offering a mechanism to locate widgets in such a way that they are adapted to their container elements.

On the other hand, in RAD environments, event handlers (which were sometimes included in the same file as the GUI definition) usually contained code belonging to several aspects of the application. For example, an event handler could accomplish the validation of a form and if it succeeds, then perform some calculations by applying some business rules and finally write the calculated data in a database by itself. This tangling of aspects is nowadays considered as a bad practice since it has a negative impact on software maintenance and reuse. Moreover, RAD developers often implemented event handlers which were attached to widgets that accessed the database and at the same time manipulated the GUI. This makes migration difficult, in particular to web platforms, since database code cannot be executed in the client side.

RAD environments have been used to develop a great number of desktop applications as part of information systems, many of them being still in production. However, the evolution of these applications is hindered in the long term because of the two aforementioned issues: fixed GUIs (non-adaptable GUIs) and tangling of aspects in the GUI code. This has motivated a large number of businesses to manually migrate their RAD legacy systems to new platforms (typically Web platforms), which better meet their needs of extensibility, maintainability or distribution, among others. Another reason for this migration is that some vendors are increasingly ceasing support in favour of other platforms.

As pointed out in [11], migrating a legacy business application to a new technology necessitates tackling three main aspects: data access, business logic and graphical user interface (GUI). Besides, migration would be facilitated by tools that help to discover architectural concerns that are only implicit (and mixed together) in the source code, such as database access, navigation flow, validation or exception handling. Figure 1.1 shows many of the aspects that are tangled in a legacy GUI.

Figure 1.1: Tag cloud of the blended elements in with a legacy GUI.

To our knowledge, just a few works have dealt with the migration of RAD-based legacy systems [28, 29], and they regard GUI migration as a straightforward task which is addressed by mapping GUI components between the source and target views. However, dealing with current technologies and devices requires a thorough analysis of the user interface so that it can be suitably reengineered. Notably, there are two main types of artefacts involved in a GUI migration: GUI definitions and event handlers. We will refer throughout this document to *GUI definition* as the software artefact or set of them that describe the widgets that compose the view, their location and their graphical properties, which are normally generated by a GUI builder. Reverse engineering the layout of the user interface (i.e. obtaining an explicit model from the spatial relationships among widgets) is crucial to migrate the GUI of a RAD application to modern GUI toolkits. However, works about migration of RAD applications reveal that layout inference is often neglected. In fact, just a few works have reported a restructuring of coordinate-based GUIs to views where the layout is managed by the toolkit [3] [4] [5]. In contrast, there is a variety of works coping with static or dynamic analysis of event handlers in order to obtain a state machine or a similar representation of the flow of windows and events, which is mostly used for testing or program comprehension purposes [7] [8] [30] [31]. Nevertheless, we have not found reverse engineering literature dealing with the comprehension and automated migration of event handlers in the context of RAD applications.

1.2 Problem statement

The hypothesis we intend to demonstrate in this thesis is the following: *We claim that the migration of a legacy GUI should consider the recognition of the graphical structures that compose the layout of the original application, and should also separate the concerns that are blended into event handlers. Then, it is possible to develop algorithms and techniques to uncover the GUI layout and disentangle the application concerns. Furthermore, we believe that MDE is a paradigm that facilitates the achievement of this goal since it has some features, namely metamodelling and model transformations, which ease the development of an automated solution.* When we will refer to *legacy systems* throughout the thesis, we will specifically refer to the applications created mostly during the 90's with the aid of RAD environments and Fourth Generation Languages (4GLs), such as Oracle Forms 6, Delphi 5, or Visual Basic 6. The *legacy system* term embraces much more platforms than RAD environments, however we will restrict the term to such context, for which we have analysed and tested some applications. Nevertheless, our proposal may be used in other scenarios, for instance, the layout inference process can be applied to the generation of final GUIs from mockups as we will see in Chapter 6.

Three high-level goals are derived from the previous statement, namely:

(G13) **Design an MDE architecture for migrating legacy GUIs**. We need to create a solution to tackle the migration of legacy systems to modern technologies, which will be settled on MDE because it provides the foundations to explicitly represent the information extracted (by means of models), and to automate the generation of these models (by means of model transformation chains). There will be models that represent the information of every GUI aspect considered, like the layout. All these models will be described by a metamodel, and the mappings between two related models will be defined by a model transformation. The construction of a full-fledged MDE solution will involve the use of several MDE tools such as model injectors (for transforming text into models), model transformation languages (to define mappings between models) and template languages (to generate code from models).

(G14) **Separate and make explicit the information of GUI definitions**. It is important to separate and make explicit the information contained in the GUI definitions (the definition of the views). Specifically, separate the logical structure of the views, the style and the layout. In a legacy application the layout is implicitly expressed in coordinates.

Representing that layout by means of high-level elements such as layout managers is a particularly challenging problem that must be addressed to achieve a good-quality migration. Then, data structures (metamodels) for representing GUIs and layout inference algorithms must be part of the solution.

(G15) **Separate and make explicit the information of event handlers**. As noted above, the code of the event handlers usually tangles several concerns, ranging from view manipulation (e.g., enabling/disabling form fields), navigation to other views, form validation, and so on. It would be desirable to uncouple them all to promote the evolution of the system. To this aim, considering the distinctive nature of RAD applications may lead to better results that addressing the problem from a general point of view. Then, performing some kind of preprocessing of the code of the event handlers before dealing with the separation of concerns will be helpful, for example, obtaining a representation that summarises the meaning of snippets of code and extracts the variables of each type.

1.3 Development

In the late nineties many companies began to migrate their Oracle Forms applications to modern platforms like JavaEE or .NET. The ModelUM group started in 2009 a research project to investigate to what extent an MDE-based solution could automate such migrations [32]. This pilot project was carried out in collaboration with the Sinergia IT software company, and the main aim was to develop a framework for automating the migration of the GUI and the data access layers. Some research problems related to the GUI migration arose at the early stages of the project, which settled the objectives of this thesis.

In the first place we performed a literature review to know the state of the art about reverse engineering and reengineering of the GUI of legacy systems. We inspected applications in Oracle Forms 6, Delphi 5 and Visual Basic 6 to know the features that typify the RAD environments, and we also analysed several User Interface Description Languages (UIDL) and Concrete User Interface (CUI) models, with the intention of using a technology-independent representation of the GUI. These languages and models represented the layout in a simple way, and they were not focused on separation of concerns, so we finally decided to create our own CUI representation and we defined a first version of our architecture.

The inspection of views of different RAD applications revealed that widgets were always placed by means of coordinates expressed in pixels or other fixed units, whereas in modern technologies the use of fixed measures is not recommended but some sort of layout managing system is advised. Then we developed a reverse engineering approach to deal with the inference of the layout, which was presented in the ASE conference [33]. An extension of this contribution, which described the approach in detail and the validation accomplished, was published in the ASE journal [34]. Before developing the approach, we did a literature review for searching works that performed some sort of layout inference from coordinates and we only found one relevant approach [3] that was not easily extended to different layout managing systems. Our solution was tested with two real case studies with positive results.

Then we moved to the analysis of event handlers. Since the analysis of code is a totally different area, we performed a new literature review and we learned the foundations from the existing approaches. We realised of the specific features of the event handlers of RAD applications and we decided to profit from them by identifying code constructions that were frequently found. We develop a program comprehension approach to disentangle the different concerns that are mixed in the code of the event handlers. The cornerstone of that approach was a model representing the behaviour of the code in an abstract way so the later reverse engineering tasks were facilitated. The work resulted in contributions in the WCRE [35], JISBD [36] and UIDL [37] conferences.

In 2012 the PhD candidate did a 9-month research stay in Louvain-La-Neuve (Belgium), in the LILab group, which is a renowned team in Human-Computer Interaction (HCI) led by Jean Vanderdonckt. During the stay, a tool for analysing web pages into UsiXML [38] specifications was developed, and we cooperated in a work presented in the RCIS'13 conference [39]. This work made us reconsider the layout inference solution to implement some improvements.

Back to ModelUM, we started to work on the idea of applying our layout inference solution to generate GUIs from wireframes. Then we considered the approach developed during the stay in Belgium to overcome some of the limitations detected in the first version of our layout inference algorithm. Therefore, the last period of the research was devoted to design and implement a new version of the layout inference approach and develop a tool to automatically generate GUIs from wireframes. The new version of the approach was tested with a real wireframing tool, and an article that described our work was submitted to the IST journal [40], which is under review at present.

1.4 Outline

The structure of the rest of this document is as follows:

- **Chapter 2** introduces the background needed for a better understanding of this thesis. It comprises basic concepts of software modernisation, the features of legacy GUIs, scenarios in which extracting information of the GUI is useful, and the principles of the MDE paradigm.

- **Chapter 3** analyses the state of the art in three areas, namely layout inference, code analysis of event handlers, and MDE approaches for reengineering GUIs. For the first two areas, some dimensions will be defined to compare the works and a discussion in each area will present the lacks and weaknesses of up-to-date approaches for reverse engineering legacy GUIs.

- **Chapter 4** outlines our proposal for migrating legacy GUIs. It describes the overall challenges we have found when addressing the problem, and we identify the requirements that we believe that a proper solution should have. We will also present the general architecture of the solution, which includes models specially designed to deal with each concern, and we will foresee how do we cope with each one of the elicited requirements in the solution.

- **Chapter 5** explains the first approach we devised to tackle the layout inference of legacy GUIs. We will expound the data structures and algorithms involved in the solution. Finally we will present the validation of the approach through a case study of the migration of Oracle Forms applications to the Java platform. Besides the evaluation of the approach, the case study has served to disclose the limitations of the current solution and draw conclusions.

- **Chapter 6** expounds a second version of the layout inference algorithm. The chapter starts stating the reasons that motivated a new version and shows the changes that have been accomplished in the solution to incorporate the new requirements. The main change affects the high-level layout inference algorithm, which is now an exploratory algorithm based on graph rewriting and pattern matching, which will be explained in depth. A new case study about reengineering of wireframes will be introduced, which will be used to test the new approach. We also include a brief evaluation to compare both approaches in the context of the migration of legacy GUIs, concretely Oracle Forms windows.

- **Chapter 7** focuses on the part of the reengineering architecture that is devoted to the analysis of event handlers. We will present the metamodel (i.e., data structure) we have built to represent the code in a concise manner, and the pattern recognition algorithm designed to extract that representation. We put into practice this metamodel in two cases: the separation of the code in layers (business logic, the controller and the GUI code), and the identification of the interactions among widgets and among views. The separation in layers will be also tested with a case study to migrate Oracle Form event handlers to an Ajax application.

- **Chapter 8** concludes this thesis by analysing the level of achievement of the goals we presented in Chapter 1 and the requirements enumerated in Chapter 4. Our solutions are contrasted with the related work and a discussion about the benefits and disadvantages with regard to those works is included, which leads to the future work proposal. Finally, the results of this thesis in terms of publications and projects are enumerated.

The world is full of obvious things which nobody by any chance ever observes.

Mark Haddon, The Curious Incident of the Dog in the Night-Time
(Suggested by Javier Cánovas)

2 Background

This chapter introduces the background needed for a better understanding of this thesis, which consists of: the basic concepts in the area of Software Modernisation, some common notions about GUIs, the particular features of the GUI of legacy systems, GUI modernisation scenarios in which an inference process is useful, and the foundations of Model Driven Engineering such as metamodelling and model transformations, and their applicability to software modernisation.

2.1 Software modernisation

Modernisation is a form of software evolution of legacy systems which involves deeper and more extensive changes than maintenance, but in which the system still has some business value that is preserved [41]. A modernisation process is applied when the desired properties of a legacy system cannot be achieved by means of maintenance. Two kinds of modernisation are distinguished: *white-box* and *black-box*. In the former the internal details of the system must be understood and some significant changes in the system structure are required (e.g. code restructuring). In the latter, the analysis of legacy systems is based on their input and output, e.g., a wrapper is a commonly used technique to achieve black-box modernisation. *Migration* is a kind of modernisation in which an entire source application or a part of it is moved to a different technology, for instance a source code translation or a database engine change.

Reengineering is also a form of modernisation that applies software engineering practices to an existing system to meet new requirements [41]. Tilley and Smith [12] define *reengineering* as *"the systematic transformation of an existing system into a new form to realise quality improvements in operation, system capability, functionality, performance, or evolvability at a lower cost, schedule, or risk to the customer"*. A reengineering process can be applied in three stages [13]. Firstly, a *reverse engineering* stage analyses the existing system and extracts knowledge which is represented at different abstraction levels. A second stage *restructures* these abstract representations in order to establish a mapping between the existing system and the target system. Finally, a *forward engineering* stage is applied to obtain the artefacts of the new system from the output of the restructuring stage. As the horseshoe model [42] illustrates (see Figure 2.1), the reverse engineering process can be applied in several steps which form a transformation chain. That chain is intended to increase the level of abstraction of the extracted knowledge so it achieves an architectural representation of the system. Then restructuring and forward engineering can be applied at different abstraction levels for any of the obtained representations to derive artefacts of the new system.

Reverse engineering is an essential activity in a reengineering process which is based on code and data comprehension techniques. Chikofsky and Cross [13] define *reverse engineering* as *"the process of analyzing a subject system to i) identify the system's components and their interrelationships, and ii) create representations of the system in another form or at a higher level of abstraction"*. Reverse engineering techniques are commonly classified in two major groups [43]: *static analysis* is based on the inspection of the application artefacts (normally source code), and *dynamic analysis* examines the state of a running application. Each type of technique has its limitations: with static analysis it is difficult to have good coverage of highly dynamic applications, while dynamic analysis faces problems with guaranteeing that generated models fully capture the behavior of the system. A third technique is *hybrid analysis*, which joins both static and dynamic analysis to take the best of each procedure.

There are numerous forms of reengineering [41]. A *platform migration* typically combines several of these forms, for instance, source code transformation, program modularisation, and data

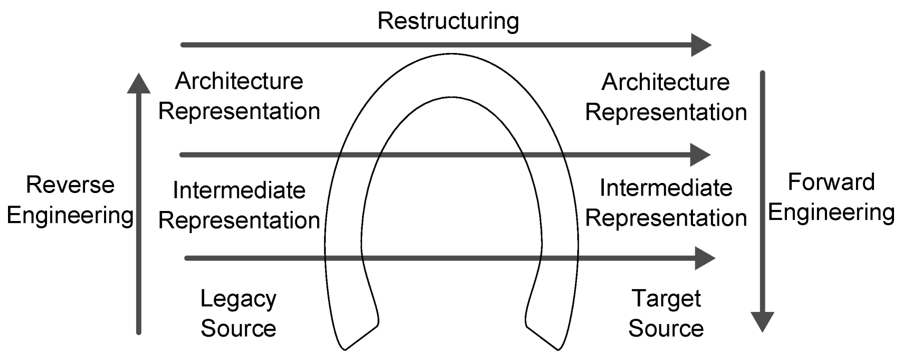

Restructuring
Architecture Representation
Architecture Representation
Reverse Engineering
Intermediate Representation
Intermediate Representation
Forward Engineering
Legacy Source
Target Source

Figure 2.1: The Horseshoe model

reengineering can be involved in a RAD-to-Java platform migration. *Revamping* is connected with the modernisation of user interfaces, in which only the user interface is changed to improve some aspects like usability. These days, with increasingly high interest in the Internet, the most popular form of revamping is adding a web interface to legacy systems. In the past a very common practice was replacing a text interface with a graphical user interface. One of the methods for this kind of revamping was *screen scraping*, that is, a black-box method, in which an application (usually an existing component) is 'redirected' from a console screen into a graphical frame of web interface [41]. This method is relatively cheap and results of a modernisation are well visible. Nevertheless the UI is just a wrapper on the old system which remains unchanged, so adding new functionalities or further maintenance is still very difficult, because system extensibility has not been improved. When these improvements are needed, a white-blox approach should be applied to move the GUI legacy code to the target platform, for instance, when an Oracle Forms application is converted into a Java Server Faces (JSF) one.

2.2 Graphical User Interfaces (GUI)

A *User Interface (UI)* is the part of a software/hardware system that is designed to interact with users. A *Graphical User Interface* (GUI) is a UI that takes advantage of computer graphics to facilitate the interaction with users. Before the popularisation of touch devices such interaction has been typically performed by means of a cursor on the screen that is controlled by a mouse, which lets the user select graphical elements such as menu items or buttons. User interfaces have a static component which is related to the presentation of the information (i.e., the structure, the layout, the usability, the accessibility or the aesthetics), and a dynamic part that is associated with the behaviour when the user interacts with it (i.e. the events that are triggered and perform actions and/or changes in the interface).

A *GUI toolkit (or widget toolkit)* is a library that supports building GUIs for a particular programming language and sometimes is tied to a framework or operating system. For instance, Gtk+ for desktop applications in C/C++ under Windows/Linux/Mac, or the Java Android SDK for mobile applications in Android. Each toolkit provides different features for the static and dynamic aspects of the GUI.

We will use the term *view* to refer to the graphics displayed on device screens. Common examples of views are *windows* in desktop applications, *web pages* in web applications, and *views* in mobile applications. The elements displayed in views are *widgets, controls or visual components* (e.g., buttons or combo boxes). The term *widget* will be used plenty of times throughout this document. There are different kinds of widgets, and every widget is characterised by a type, a set of graphical properties such as background colour or font type, and status properties such as visibility (if the widget is visible) or editability (if the widget can be edited). In general, widget types are commonly classified according to their purpose: entering data (e.g., text fields), showing information (e.g., data grids) or interacting with the system (e.g., buttons). There are also widgets (like panels) that are used to structure views, in such a way that buttons or text fields are contained in panels (similarly to a Composite pattern). In this sense, views are containers too and they are actually the topmost components in the aggregation hierarchy of the GUI elements, which is sometimes referred as *GUI tree*. Figure 2.2 shows an example view for recording user data which contains *NameLabel*, *NameBox*, *PaymentFrame* and some other widgets, and *PaymentFrame* is in turn the container of *CardLabel*, *CardCombo*, *DiscountLabel*, and *DiscountCheck*. A part of the GUI tree of this view is shown in Figure 2.3.

The *layout* of a graphical user interface is the spatial distribution of the elements in the views of the application. There are GUI toolkits that define explicit components for laying out content (e.g., the hbox and vbox in ZK [44]), while in other cases the layout is defined by properties (e.g., float in CSS [45]) or assigning predefined layout types to certain groups of widgets (e.g., Java AWT [46] layouts). The latter are commonly known as *layout managers*. They are software components that automatically lay out the widgets on a view based on relative relations and restrictions that are inherent to the layout type and partly specified by the programmer.

In every modern GUI technology, the GUI behaviour is implemented by an event-driven ap-

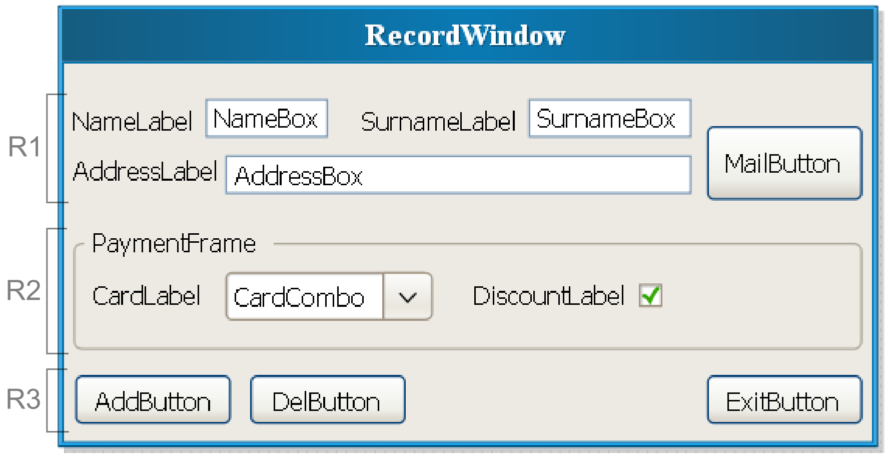


Figure 2.2: Example view for entering personal information. Widgets are placed with explicit coordinates.

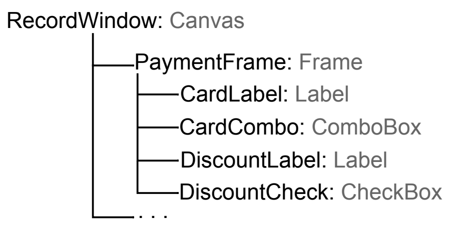


Figure 2.3: An excerpt of the GUI tree for the window in Figure 2.2.

proach. Each widget is able to trigger some types of events under certain conditions. For instance, typical types of events for a button are *click* (the button has been pushed) and *hover* (the cursor is over the button), and common types of events available for text boxes are *change* (the content of the text field has been modified) and *focus* (the text field has been selected and is ready for writing). Different types of widgets can trigger the same event types, but not all the types of events are available for all the types of widgets. For example, buttons and text fields can trigger the *hover* event, but the *change* event makes no sense for buttons. Note that the set of events supported by widgets is not standard, but each GUI toolkit may implement a different one. Widgets can be attached actions that are implemented by programming code (i.e., event handlers) that are executed every time a certain event happens on the widget. These actions can provide some application functionality, modify the aspect of the current view, or change the view, among others. In short, an event is featured by three elements: i) a widget, ii) an event type, and iii) an event handler that deals with it.

2.2.1 Visual GUI features

Widgets are not randomly distributed on the screen but they form some sort of design (layout) that deeply affects the readability and usability of the GUI. The layout is probably the most complex element of the visual part of a GUI, as it cannot be defined by a single value or a list of values, but it is the result of applying several features on different widgets or groups of them. We have identified several features that characterise the layout. Next we comment on them.

- **Visual structure.**

  It is related to the human perception about the widget arrangement, and is a key feature to allow adapting the content of a view to different text or screen sizes. Knowing the visual structure requires analysing the positions of all the elements in the view to recognise the 'shapes' they form and how they are visually grouped. A horizontal flow of widgets or a grid of elements are examples of visual structures. Note that different layout arrangements may produce similar visual structures that can be equally valid for the same view.

  For example, in the login view shown in Figure 6.11, *nameLabel* and *nameField* form a line, *passwordLabel* and *passwordField* form a second line, and the *ok* and *cancel* buttons form a third line. Another layout possibility would be to put *nameLabel* and *passwordLabel* in one column, and the rest of widgets in another column.

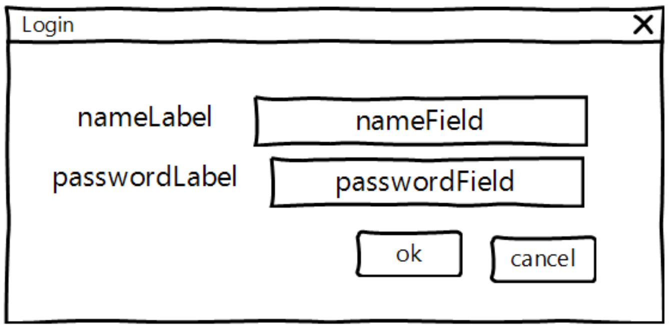


Figure 2.4: Login window created with WireframeSketcher.

Sometimes there are widgets that are surrounded by a rectangle because they are related to the same topic, or there are groups of widgets that are visually distant to other groups. In these cases, the groups should be identified and handled as a unit if compared to the rest of elements.

- **Sizing**

  The size of the widgets is another feature that must be considered. Sizes can be expressed in absolute units like pixels, or in relative units, for example in percentages regarding the container element. It is advisable to always use relative units so the measures are independent of the concrete screen of the device.

- **Spacing**

  The spacing between the widgets in the view is also relevant. We must distinguish between the gaps and the margins. We call *gaps* to the spacing between the single widgets (e.g. the separation between a label and a text field). *Margins* are the distances between the single widgets and their container. Note that gaps and margins are either *horizontal* or *vertical*, depending on the axis in which they are observed. Like sizes, gaps and margins can be expressed in absolute or relative units, though the latter are preferred.

- **Alignment**

  The alignment is either horizontal or vertical, and it is defined for a widget with respect to other widgets, or defined for a widget regarding its container. For instance, in Figure 6.11 the widgets *nameField*, *passwordField* and *cancel* are aligned to the right with regard to each other.

Looking at these three widgets carefully we can see that they are not perfectly aligned, though it seems that the intention is that they are aligned. Therefore, when dealing with the layout, it would be interesting to accept some degree of misalignment, i.e., the analysis of the positions of the elements must be flexible. In addition, in cases it may happen that the area taken by a widget slightly overlaps other widgets, and it is neccessary to deal with some small overlapping.

2.2.2 Legacy GUI features

The GUI of a legacy system commonly has some features that are not present in modern GUI technologies. Some of them are discouraged practices in software engineering that are no longer implemented. We have studied the GUI definition and code of three different RAD environments, namely Oracle Forms 6, Visual Basic 6 and Delphi 5. Next table summarises the main features of the studied environments.

	Oracle Forms 6	**Microsoft Visual Basic 6**	**Borland Delphi 5**
Year	1996	1998	1999
Implicit layout	Yes	Yes	Yes
Proprietary units	Yes (points)	Yes (twips)	No (pixels)
Clustering elements (containers)	Canvases, Frames, Rectangles, ...	Frames only	Panels, GroupBoxes, RadioGroups...
Container overlapping	Yes	Not compulsory	Not compulsory
Widget set	20 standard	20 standard, +30 complex controls	34 standard, +50 complex controls
Widget-database links	Yes	Yes (ADO)	Yes (ADO)
Table widget	Multirecord text-fields	ADODC	TDBGrid
Code mixes aspects	Yes	Yes	Yes
GUI definition format	binary	property=value	property=value
Event handler format	PL/SQL triggers (binary)	Visual Basic subroutines, mixed with GUI definition	Delphi methods

Table 2.1: GUI features of three different RAD environments

Based on the mentioned table, we list some features that can be frequently found in GUI definitions that have been built with RAD environments:

- **Implicit layout**. The position of widgets is stated by means of a pair of coordinates that are relative to the main window or another container, and rarely, relative to another widget (e.g., a label to its text box). The size (width and height) of a widget is also given explicitly by the RAD environment. This means that, for example, when a window is resized the widgets are not resized or rearranged accordingly. As it can be seen, the three studied RAD environments have an implicit layout. In cases, these technologies do not use standard units like pixels or centimetres, but proprietary units. For example, in Visual Basic 6 the default measurement unit is the *twip*, which is 1/20 of a typographical point (1/1440 of an inch). Twips are screen-independent units, they were created to avoid the disadvantages of fixed units like pixels, but they are no commonly found in modern IDEs.

- **Clustering elements**. There are special widgets which are intended to group and/or highlight semantically-related widgets. In particular, we distinguish between elements that arrange a window in parts (in some legacy environments they can also be reused between windows), and elements that are used to highlight a set of widgets in close proximity, frequently by means of a border.

- **Overlapping**. Widgets are often loosely contained in their container, that is, they are overlapped with the container instead of having explicit containment relationships. A container could also be overlapped with another container. This means that a container may not have any children widget in the GUI tree, although there may be some widgets that would (visually) be expected to be contained. In Visual Basic 6 and Delphi 5, containers and widgets may be overlapped, but in Oracle Forms 6 this overlapping is unavoidable. For example, in relation to the view in Figure 2.2, Figure 2.5(a) shows a fragment of the original GUI tree created by a RAD environment like Oracle Forms, and Figure 2.5(b) shows the expected GUI tree. In that view it can be seen that *PaymentFrame* surrounds *CardLabel, CardCombo, DiscountLabel* and *DiscountCheck* (the checkbox next to *DiscountLabel*), but these widgets are only visually contained in the frame, that is, their parent element in the model is not *PaymentFrame*, but rather *RecordWindow* (see Figure 2.5(a)). We could expect that the GUI tree would be like Figure 2.5(b).

- **Widget set**. RAD environments as well as modern frameworks share a common set of standard widgets, such as text boxes, buttons, combo boxes, tables, and so forth. However, some environments like Delphi 5 include technology-dependant widgets that may not have an equivalent in other environments. Developers sometimes wanted to use

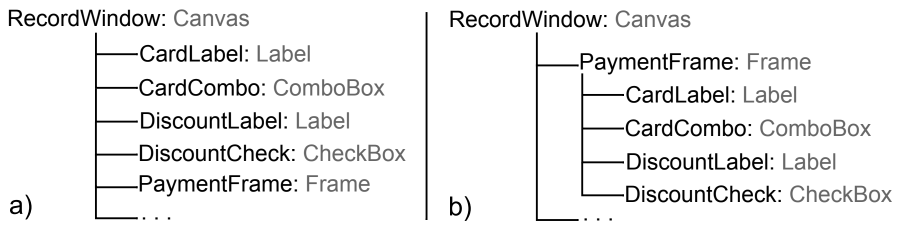


Figure 2.5: (a) Fragment of the original GUI tree. (b) The expected GUI tree.

complex widgets that were not available in the GUI technology in which they were programming and they did a bit of a trick by emulating those complex widgets by means of a composition of the available widgets. For example, a calendar (which is nowadays a common component) was typically emulated in Oracle Forms by means of a grid of buttons (see Figure 2.6). Another example is a table with a scrollbar, in which the parts of the scrollbar were emulated by buttons.

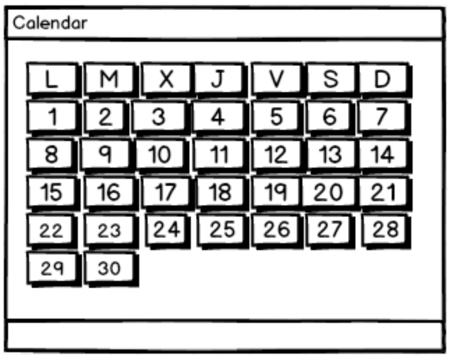


Figure 2.6: A calendar component emulated by a grid of buttons.

- **Widget-database links**. Sometimes widgets are tied to table columns in database tables. In Oracle Forms, Visual Basic and Delphi, the property sheets of widgets include some properties to indicate that information. Particularly, in Visual Basic and Delphi, widgets contain datasource and data field properties. The former is configured by means of an ADO control that indicates the connection string and the data table, and the latter is the column name in the database table. Oracle Forms does not use ADO, but there are rather similar properties to indicate the database connection.

The code of event handlers in legacy systems also has some characteristics that are not typically found in modern applications. Next we list some of them.

- **Tangling of concerns**. Code managing the GUI is mixed with business logic and database access. There is no clear separation among the different concerns of the application. For example, the event handler shown in Figure 2.7b takes the value of *ABE_IMPP*, divides it by the euro exchange value obtained from the database, and places the result in *ABE_IMPE*. As it can be seen, the database access and the GUI are tightly tied.

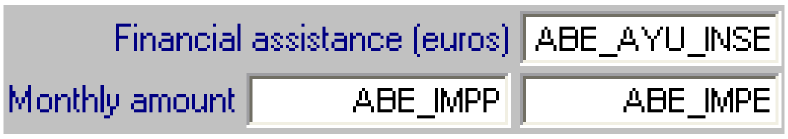
- **Simple behaviour**. It does not perform complex algorithms or calculations. Event handlers are hardly ever complex, which is caused by the fact that complex functionality is typically implemented in separate functions or stored procedures that are called by the handlers.

- **Restricted looping**. Loops are only used to iterate over database tables. This is a consequence of the previous point, since algorithms used to solve problems are programmed in procedures. Loops are only found when using collections or using sentences to iterate over database rows.

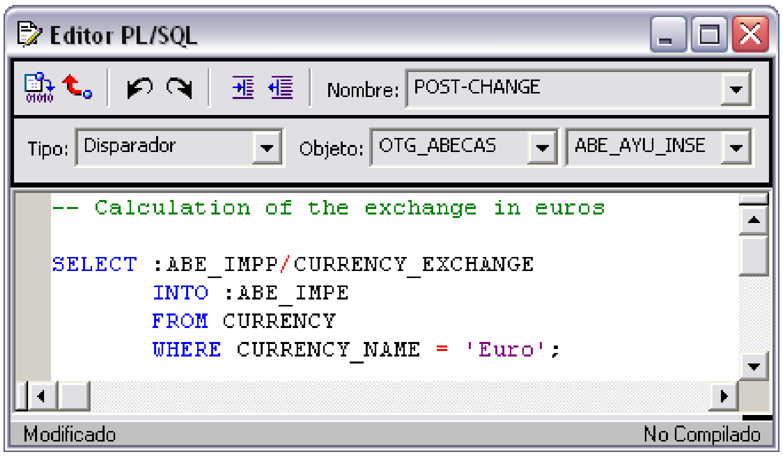
- **Conditional paths**. Several levels of nested conditional statements are common, where conditions check values from the GUI or the database. Actions such as updating the GUI or modifying the database are normally performed in the most inner blocks.

- **Idiom-based programming**. Applications usually repeat a series of idioms. Some of them are specific of each RAD environment, while others are conventions dependant on the company. Querying a value from the database and placing it in a text field after some kind of modification is a recurrent pattern carried out in event handlers of legacy applications, as it is done in the example code of Figure 2.7b. Another example is shown in Figure 2.8, and excerpt of an event handler in Delphi 5. The code checks whether a task is active before deleting it, and if the task is active, then aborts the deletion operation. Checking if a value exists in a database before performing an operation is also a common pattern.

- **Similar programming abstractions**. Although each legacy environment has its own programming language to write event handling code, most of them provide similar constructs. As it was seen in Figure 2.7b, in Oracle Forms 6 simple database access can be performed with implicit PL/SQL cursors, and in Delphi 5 it can be accomplished through a *TADOQuery* object.



(a) Example window fragment

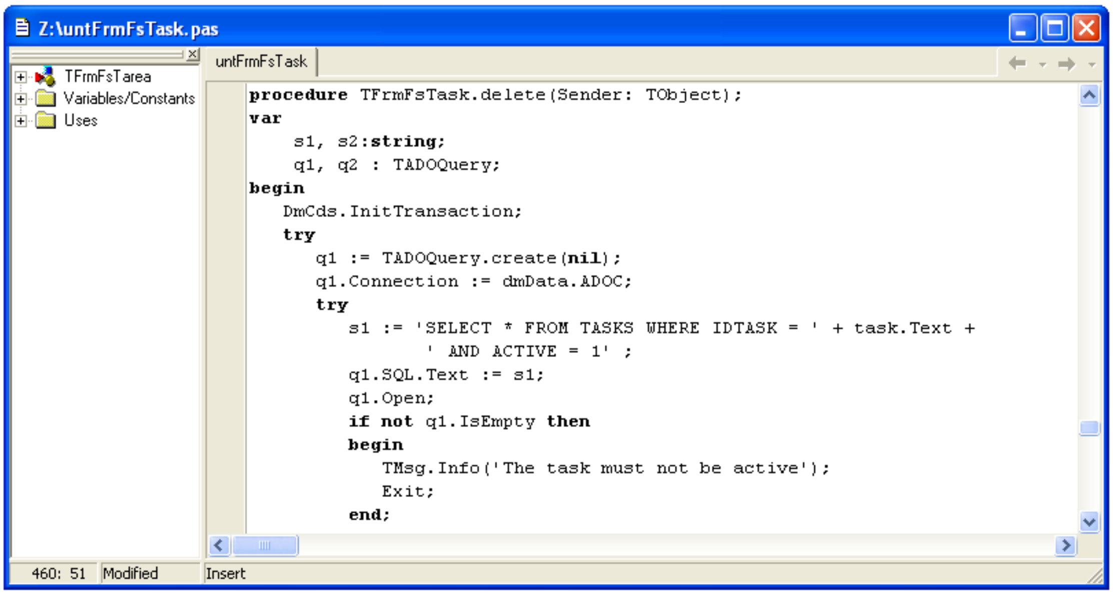


```
-- Calculation of the exchange in euros

SELECT :ABE_IMPP/CURRENCY_EXCHANGE
       INTO :ABE_IMPE
       FROM CURRENCY
       WHERE CURRENCY_NAME = 'Euro';
```

(b) POST_CHANGE Event handler associated with ABE_AYU_INSE (PL/SQL)

Figure 2.7: Example of mixing of concerns in an Oracle Forms application



```
procedure TFrmFsTask.delete(Sender: TObject);
var
    s1, s2:string;
    q1, q2 : TADOQuery;
begin
   DmCds.InitTransaction;
   try
      q1 := TADOQuery.create(nil);
      q1.Connection := dmData.ADOC;
      try
         s1 := 'SELECT * FROM TASKS WHERE IDTASK = ' + task.Text +
               ' AND ACTIVE = 1' ;
         q1.SQL.Text := s1;
         q1.Open;
         if not q1.IsEmpty then
         begin
            TMsg.Info('The task must not be active');
            Exit;
         end;
```

Figure 2.8: Fragment of a Delphi 5 event handler that checks if a task is active before deleting it.

2.2.3 Use scenarios of GUI reverse engineering

Inferring information of the GUI such as the layout or the aspects involved in code, and representing it explicitly, is useful in a variety of cases. Next, we briefly comment on several scenarios in which GUI reverse engineering activity would enable GUI reengineering and other types of activities to be performed:

- **Revamping**. As we have already mentioned, this is the case in which the business logic of the legacy system is reused, and only the views are changed. Frequently, this scenario involves wrapping[47] the legacy code in order to be able to access from the code of the new GUI technology. A few changes are performed on event handlers just to adapt them to the new views. A particular case of revamping is the *layout-preserving migration*, which takes place when a migration project have a requirement which specifies that the original GUI layout must be preserved in the target application due to users are averse to change.

- **GUI testing**. There are different strategies to accomplish GUI testing. An strategy is to generate a mock application with the views original application which tracks user input in order to generate test cases [48]. Another strategy consists of symbollically executing the code to generate test inputs [6]. Other works instrument event handler code to record user interactions which are later analysed [49].

- **GUI adaption**. Migrating to a new GUI technology requires taking advantage of the target technology's features (e.g. usability standards, high-level layout models of modern GUI toolkits, etc.). Deep changes in views and event handlers are usually required in this scenario. A particular case of this category would be the migration to technologies with constraints related to the screen size, such as mobile devices.

- **Quality improvement**. Perfective maintenance tasks may be required to improve the system quality, such as the detection of usability issues, non-visible widget removal, GUI resizing and beautification [3], separation of concerns [11], code refactoring [50] or death code removal.

- **Forward engineering**. Forward engineering approaches to develop new systems can also benefit from GUI reverse engineering. In software development methodologies, GUI designs are validated in the early stages of development with a mockup (Figure 6.11 shows a plain mockup), which is a GUI representation that is created before the final product so stakeholders can check it. The same approach used in reverse engineering an existing system can be applied to the development of a new one just by taking mock-ups as source artefacts. Then, final GUIs for different platforms or technologies can be generated from the GUI representations.

2.3 Model Driven Engineering (MDE)

Model-Driven Software Engineering (MDSE or simply MDE) is an emerging area of Software Engineering which addresses the systematic use of models to improve the software productivity. Models can be used in the different stages of the software lifecycle to raise the abstraction level and automate development tasks. There exist several MDE paradigms such as Model Driven Architecture (MDA) [15] or Domain-Specific Development [16] [51] which share the same four basic principles [52]: (i) models are used to represent aspects of a software system at some abstraction level; (ii) they are expressed using DSLs (a.k.a. modelling languages) (iii) that are built by applying metamodelling techniques and (iv) model transformations provide automation in the software development process.

2.3.1 Metamodelling

A *metamodel* is a model that describes the concepts and relationships of a certain domain. A metamodel is commonly defined by means of an object-oriented conceptual model expressed in a metamodelling language such as *Ecore* [53] or *MOF* [54]. A metamodelling language is in turn described by a model called *meta-metamodel*, therefore, a metamodel is an instance of a meta-metamodel and a model is an instance of a metamodel.

Metamodelling languages generally provide four main constructs to express metamodels: classes (normally referred as metaclasses) for representing domain concepts, attributes for representing properties of a domain concept, association relationships (e.g., aggregations and references) between pairs of classes to represent connections between domain concepts, and inheritance between child metaclasses and their parent metaclasses for representing specialisation between domain concepts. In the following chapters we will use metamodels to describe the data structures involved in the proposed solution.

2.3.2 Domain-Specific Languages (DSLs)

In contrast to General Purpose Languages (GPLs), *Domain-Specific Languages (DSLs)* are languages that are defined to solve problems in a specific domain. In the MDE context, the *DSL* and *modelling language* terms are commonly used to refer to the languages used to build models, which are usually created by applying metamodelling, that is, the language allows creating models whose structure is determined by a metamodel.

A DSL consists of three basic elements: abstract syntax, concrete syntax and semantics. The *abstract syntax* describes the set of language concepts and their relationships, along with the rules to combine them. Metamodelling provides a good foundation for this component, and it is the most widespread formalism in MDE but other formalisms have also been used over the years, such as grammars for programming languages and DTD/XML schemas for XML documents. The *concrete syntax* defines the notation of the DSL, which can be textual or graphical (or a combination of both). The semantics defines the behavior of the DSL; there are several approaches for defining it [55], but it is typically provided by building a translator (i.e., a compiler) to another language that already has a well-defined semantics (e.g., a programming language) or an interpreter.

An example of graphical DSL for creating quick designs of GUIs are mockup tools (e.g., *Balsamiq* [56]), as they conform to a formalism (metamodels or DTD/XMLSchema in most cases), they have a graphical notation (widgets) and they have the semantics of the GUI toolkits for which the GUI code can be generated.

DSLs have been used since the early years of programming, however, MDE has substantially increased the interest in them. Most MDE solutions involve the definition of one or more DSLs in order for users to create the models that are required. When MDE is applied in reengineering legacy systems, concrete syntaxes are not needed for the metamodels that represent the information gathered in that process if such information is not intended to be understood by users. Actually, in our case we have not defined a concrete syntax for any of the metamodels we will present, but models (i.e. instances of metamodels) have been directly manipulated by model transformations, which we introduce next.

2.3.3 Model transformations

Model transformations allow automating the conversion of models between different levels of abstraction. An MDE solution usually consists of a model transformation chain that generates the desired software artefacts from the source models. Three kinds of model transformations are commonly used: model-to-model (M2M), model-to-text (M2T) and text-to-model (T2M).

M2M transformations generate a target model from a source model by establishing mappings between the elements defined in their metamodels. One or more models can be the input and output of a M2M transformation. M2M transformations are used in a transformation chain as intermediate stages that reduce the semantic gap between the source and target representations.

The complexity of model transformations mainly depends on the abstraction level of the metamodels to which the models conform. The most frequently used M2M transformation languages (e.g., *QVT* [57], *ATL* [58], *ETL* [59]) have a hybrid nature since M2M transformations can be very complex to be expressed only by using declarative constructs [60]. These languages allow transformations to be imperatively implemented by using different techniques: i) imperative constructs can be used in declarative rules (e.g, ATL and ETL), ii) a declarative language is combined with an imperative one (e.g., QVT Relations and QVT operational), or iii) the language is designed as a DSL embedded into a general purpose language (e.g., *RubyTL* [61] into Ruby). Using model transformations to solve reverse engineering problems is an example of scenario where a high degree of processing of information is required and the complexity of transformations can become very high. A survey on model transformation languages can be found in [62].

M2T transformations generate textual information (e.g. source code) from an input model. M2T transformations produce the target artefacts at the last stage of the chain. *MOF2Text* [63] and *XPand* [64] are some of the most widely used M2T model transformation languages.

Finally, *T2M transformations* (also called *injectors*) are used to extract models of the source artefacts of an existing system, and are mainly used in software modernisation to obtain the initial model to be reverse engineered. Hence, they are less frequently used than M2M and M2T. Among the tools for extracting models from code we remark *MoDisco* [23] that implements parsers (called *discoverers*) for Java and other languages, the XML injector of the *Eclipse Modeling Framework (EMF)* [53] that obtains Ecore models from XML schemas, and *Gra2MoL* [22], which is a textual DSL especially designed to define T2M transformations when the source artefact consists of text that conforms to a grammar, by establishing mappings between that source grammar and a target metamodel.

2.3.4 Model-Driven Modernisation (MDM)

MDE is increasingly gaining acceptance, mainly because of it is being successfully used in building new software systems (forward engineering) [47] [65]. But MDE techniques, such as metamodelling and model transformations, are also useful to evolve existing systems, as they can help to reduce the software maintenance and modernisation costs by automating many basic activities in software evolution processes. In this setting, *Model-Driven Modernisation (MDM)*[1] has emerged as an MDE approach to be applied in the software modernisation scenario. Several experiences of applying MDM have been recently published [66] [67] [20], which have showed how MDE techniques facilitate the obtainment of representations that have an abstraction level higher than source code, and how modernisation tasks can be automated, e.g., providing metrics to analyse the impact of the changes or automatically generating software artefacts of the evolved system.

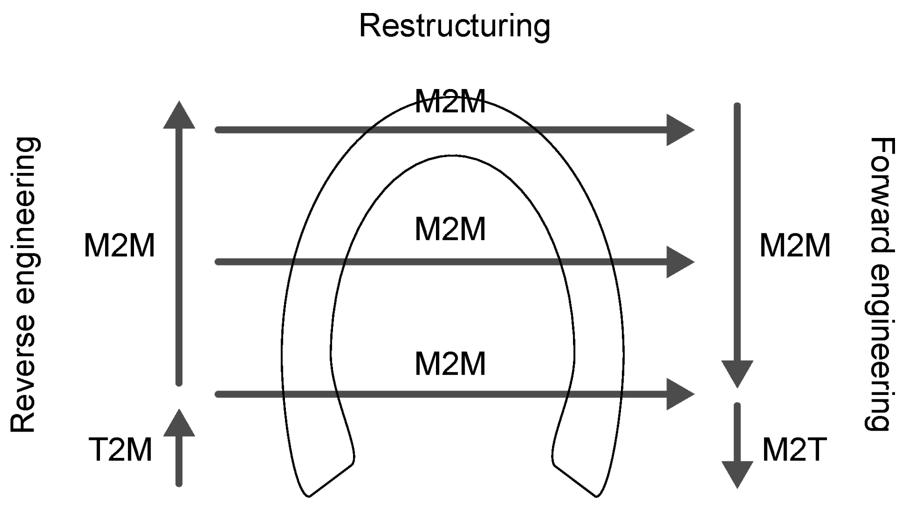


Figure 2.9: MDE applied to reengineering

In the MDM context, reengineering is accomplished by applying model transformations in each of three stages of the process (see Figure 2.9). *Reverse engineering* gets models from the source artefacts which are not just a model representation of the code, but they provide a higher abstraction level. Frequently, this step is tackled by a T2M transformation that gets a low-level

[1]*Model-Driven Reengineering (MDR)* is an approach related to MDM that advocates the use of models in reengineering.

representation of the code (and is therefore dependent on the type of source artefact), followed by one or more M2M transformations that get more abstract representations. A crucial aspect is the definition of the metamodels that are appropriate to represent the knowledge collected in each step of the transformation chain. *Model-Driven Reverse Engineering (MDRE)* [68] [69] is a common term referred to the use of MDE in the reverse engineering stage. In the restructuring stage the models are transformed into other ones that conform to some aspects of the target architecture, which is accomplished by one or more M2M transformations. Finally, the *forward engineering* stage takes the models obtained in the restructuring stage and generates artefacts of the new system, which can be performed by a M2T transformation. If there is a wide semantic gap between the models obtained after the restructuring stage and the target code, a M2M transformation chain finished by a M2T transformation is frequently advised.

To increase the interest in applying MDE to modernise legacy systems, OMG launched the *Architecture Driven Modernisation (ADM)* initiative in 2003 [17], whose objective is to develop a set of standard metamodels for common tasks in software modernisation in order to facilitate the interoperability among tools. Several modernisation scenarios in which ADM metamodels have prove to bring benefits are described in [18] [70].

Among these metamodels, *Knowledge Discovery Metamodel (KDM)* [71] plays a main role due to it is targeted at representing application code at different abstraction levels, from GPL statements to business rules. It is, therefore, an arguably large metamodel structured in four layers, namely *Infrastructure*, *Program elements*, *Resource*, and *Abstractions*. The *Abstract Syntax Tree Metamodel (ASTM)* is a metamodel that complements KDM and is devised to represent code in the *Abstract Syntax Tree (AST)* form. In [66] a detailed explanation on how to use KDM and ASTM to model PL/SQL code can be found, as well as a case study for gathering software metrics is presented. Other ADM metamodels are *Software Metrics Metamodel (SMM)* for representing metrics, and *Automated Function Point (AFP)* for automating the extraction of function points. Up to the present time, the impact of the ADM standards has been very limited, mainly due to the complexity of KDM [66] and few works that illustrate real case studies have been published.

In [24] some MDM tools that have been recently developed are presented, among which MoDisco has received greater attention. MoDisco[23] is an extensible open source MDRE framework to develop model-driven tools to support use-cases of existing software modernisation. MoDisco aims at supporting the description, understanding and transformation of existing sofware by providing four elements: i) metamodel implementations like relational database, KDM and

JavaSE metamodels, ii) discoverers to automaticaly inject models of these systems such as a discoverer from Java code to KDM models, iii) generic tools to understand and transform complex models created out of existing systems, and iv) use cases illustrating how MoDisco can support modernisation processes.

We ourselves feel that what we are doing is just a drop in the ocean. But the ocean would be less because of that missing drop.

Mother Teresa of Calcutta
(Suggested by Jesús García Molina)

3 State of the art

Our work tackles the problem of reverse engineering the GUI of legacy systems, concretely two aspects, namely layout and behaviour. To cope with it we have used MDE techniques. Consequently, the analysis of the state of the art has been classified in three sections: layout recognition approaches, behaviour extraction approaches, and MDE approaches for representing GUIs.

3.1 Analysis of layout recognition approaches

In this section we will present some works related to GUI layout inference. Three works are of special relevance for our work, which are [3] [4] [5], since they deal with the extraction of a layout expressed in coordinates and they deserve a section each one to analyse them in detail. Other works related to reverse engineering layout and structure of GUIs will be summarised in a single section, as they are not as close to the topic as the former ones.

We have identified a set of dimensions which are useful to classify layout inference approaches. The three aforementioned works will be categorised according to the following dimensions:

1. **Source/target independence**: whether the proposed approach is generic, i.e. it is independent of the source and target technology.

2. **Tested source technology**: the technology or type of tool which was originally used to create the GUI definitions in the case studies of the approach. For example, a RAD environment such as Oracle Forms, or a wireframing tool like Balsamiq.

3. **Tested target technology**: the platform and toolkit in which the final GUI is created in the case studies of the approach. For instance, the ZK web framework, or the Java Swing toolkit for desktop applications.

4. **Reverse engineered information**: the kind of information that is extracted in the GUI reverse engineered process. Different approaches may describe a user interface by using different types of information, for example, the sizes of the widgets or how the widgets are contained in other widgets (containment hierarchy).

5. **Layout model**: the data structure devised to explicitly represent all the information that has been extracted from the original GUI. A layout model based on combining horizontal and vertical elements (HVLayout) is one simple example. This representation is a cornerstone in the approach since any forward engineering approach to generate a final GUI will use this representation.

6. **Algorithm type**: the algorithmic strategy involved in the discovery of the layout, such as backtracking or heuristics, and/or the theoretical basis to solve the problem, e.g. linear programming.

7. **Implementation technology**: the technological basis used to implement the approach, for instance, an MDE-based approach.

8. **Automation degree**: wether the approach is totally automated or mostly automated with user intervention in many cases (semi-automated).

Next we analyse the three approaches that are closely related to ours.

3.1.1 Lutteroth

Lutteroth [3] claims that most GUIs are specified in the form of source code, which hard-codes information relating to the layout of graphical controls. He points out that hard-coded GUIs

lack in dynamic layout as the position and size of the elements are expressed in pixels, and that this representation is very low-level and makes GUIs hard to maintain. He suggests a reverse engineering approach that is able to recover a higher-level layout representation called the Auckland Layout Model (ALM).
The author argues that GUIs using pixel units have many disadvantages. GUIs can be executed in different devices with different resolutions, and even the visible part of the GUI is modified when the window is resized. He claims that, in those cases, pixel-based GUIs do not guarantee a correct display. Moreover, when the content of a widget changes, the size of the widget has to be manually re-defined, and when some widgets are added or removed, it is likely that other widgets have to be manually modified. All these adjustments do not automatically happen in a pixel-based GUI.
The ALM is a mathematical model that captures the invariants of a GUI by using linear programming. An invariant is a condition to be satisfied, e.g., the width of a widget must be less than the width of the panel it contains it. Those invariants are used as constraints in an optimisation process that results in the calculation of an adapted layout whenever circumstances change (e.g., the dimension of the window is altered). ALM offers different layers of abstraction on top of bare linear programming (which is very low-level) that make it possible to specify the invariants of typical GUIs more conveniently.
ALM allows developers to define linear constraints in terms of tabstops and areas:

1. A *tabstop* represents a position in the coordinate system of a GUI. All positions and sizes in a layout are defined symbolically using tabstops as variables. Tabstops form a grid in which all the controls are aligned.

2. An *area* is a rectangular portion of space defined by the tabstops of the upper-left corner and the lower-right corner, the control that occupies the space, and the preferred, minimum and maximum sizes of the space.

Heuristics are applied for choosing the preferred, maximum and minimum size of the area depending on the control. For example, buttons do not normally change their size when they are resized, whereas text areas commonly take the extra space of the window. Two types of constraints can be specified: hard constraints and soft constraints. Hard constraints have to be always satisfied, and soft constraints may not be satisfied fully if circumstances do not permit so.

The input of the reverse engineering process is a hard-coded GUI, and the output is a set of areas containing the children controls and a set of linear constraints (equations/inequations with the tabstops as variables). From the point of view of the developer, a layout manager is provided, that resolves the linear constraints and adapts the layout accordingly. There is an implementation of the layout manager for C#, so developers can use this layout manager to lay out containers such as *Form* elements.
The reverse engineering algorithm uses some criteria to beautify the recovered layout, namely:

1. Controls can be slightly misplaced when creating the GUI. The algorithm can correct these misplacements by introducing some additional constraints.

2. Margins are standardised. Distances between controls or between controls and borders are adjusted so they are similar.

3. Sizes are standardised. For example, make the controls in the same column have the same height.

4. Keep rows/columns of similar controls in a certain proportion of other rows/columns.

5. Use real world units such as centimetres to make GUIs be rendered consistently on different screen resolutions.

3.1.2 Rivero et al.

In [4] authors state that mockups have become a very popular artefact to capture GUI requirements in agile methods, but most development approaches use them informally without providing ways to reuse them in development processes. They bet on taking advantage of mockups during software development to automate the creation of GUIs, and they propose a model-driven approach for importing mockups and transforming them into a technology-dependant model that can be used to generate code for web technologies.
They have set their approach in the context of a WebTDD process though they claim that it can also be used with RUP-based processes or Extreme Programming. The approach can be seen in Figure 3.1. For each mockup tool, a parser needs to be created (step 1). Then, the controls are rearranged as explained below (step 2) and the Abstract Mockup model is obtained (step 3), which helps to abstract mockups in a tool-independent way. This model can be used to

derive UI class stubs or models implemented with a concrete technology. For each concrete technology of interest, a code generator must be constructed (step 4).

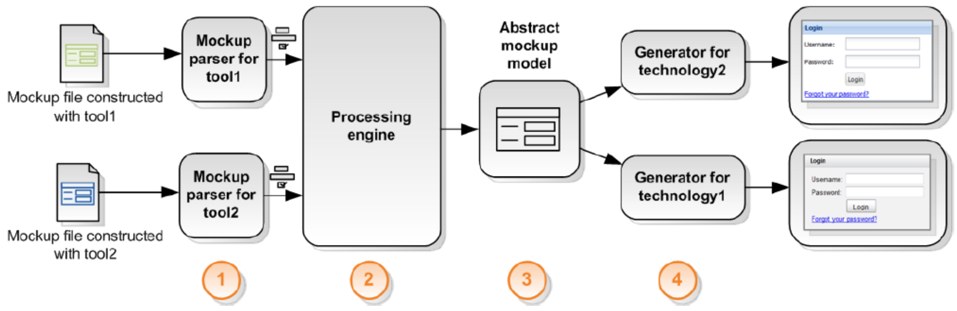


Figure 3.1: Schema of the Rivero et al. approach (extracted from [4]).

Unlike common UI frameworks, mockup tools do not generally provide ways of defining UI control composition, but all the controls are at the same level (controls are not contained in other controls). The Abstract Mockup metamodel takes this issue into account in order to derive complete UI specifications for concrete technologies. The mockup parsers scan the UI specifications looking for controls and storing their properties (e.g., position or size), and they also detect clusters of controls, so each cluster represents a set of components in a unique graphic space (e.g., a page, a window or another grouping concept). Then, the Processing engine creates a hierarchy of controls as follows: if a control is graphically contained in another one and the first one is a composite control (i.e., a panel), the second one is added as a child of the first one.

Because of the myriad of different web technologies, an absolute positioning scheme is not sufficient to model a UI in a platform-independent way. To avoid this problem, the Processing engine arranges components in a platform-independent layout. Particularly a *GridBag* layout similar to the Java Swing layout manager of the same name has been implemented. This layout manager arranges components in the same way it is done in HTML tables and it was selected because authors consider that it is richer and more flexible than others. The algorithm to obtain a *GridBag* layout starts by placing all the components in a single cell, and iteratively divides it so creating a grid of cells. In every iteration, a new column or row is created, and the algorithm stops when every cell is occupied by at most one widget. There may be widgets that take more than one cell, e.g. a text field t that occupies the space of two cells ($t.colspan = 2$).

Since their approach can be used in iterative processes in agile methodologies, UI evolution is

an important concern. Between two iterations, existing UI controls can be possibly modified, which could entail a problem if the automatically generated UI component identifiers change from the previous iteration. The solution proposed is to indirectly reference UI components by means of an identifier translation function (*reference translator*), which maps logical identifiers of UI components to real identifiers assigned by the code generator. Therefore, every time it is required to access to a control, the reference translator is used. Then, that problem can be solved by correcting the real identifiers in the reference translator between iterations.
The proposed architecture is extensible, given that a developer can take the framework and extend it. In order to add a new mockup tool, a parser that returns a collection of control clusters must be implemented. With the aim of adding a new target UI technology a code generator must be implemented. The framework provides some helper classes (e.g. indentation for code generators) and uses object oriented patterns such as the abstract factory or visitor pattern to make extension easy.
As a proof of concept, authors have tested the approach with different mockup tools (Pencil, GUI Design Studio and Balsamiq) and target web technologies (YUI and Ext JS).

3.1.3 Sinha and Karim

A recent work by N. Sinha and R. Karim [5] proposes a model-based approach to compile mockups to flexible web interfaces. The authors refer to *flexible layout* as a layout that is *fluid* (when the window is resized the content scales accordingly) and *elastic* (the content resizes on changes in font sizes).
Two phases are defined in the process of obtaining high-quality web pages from mockup editors (see Figure 3.2). The first phase is to infer the right page layout, i.e. the vertical/horizontal flow of content that preserves the relative sizes and alignment of individual elements. The second phase is to encode the inferred layout in a HTML page faithfully.

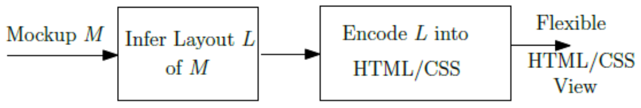


Figure 3.2: Sinha and Karim approach (extracted from [5]).

A mockup is defined as a collection of rectangular objects (boxes), each box having its visual properties (e.g., size or colour). Given that a native web application is laid out with HTML/CSS

boxes, the authors propose a box-based layout. They suggest two box-based layouts: grid layout (a unique grid with $n \times m$ cells) and HVBox layout (hierarchy of horizontal/vertical boxes), and they claim that HVBox layout is preferred since grids result in fine-grained layouts which have additional overhead. They made the decision of inferring HVBox layout from mockups.
In order to infer the box hierarchy their approach employs a combinatorial search, which is inspired on the explore-fail-learn paradigm used in constraint solving problems. The algorithm starts with the single boxes and applies a bottom-up approach to merge pairs of boxes until a solution is reached. When a pair of merging boxes intersect other boxes, the configuration is discarded since it will not reach a valid solution. After obtaining the layout tree, nodes that have children of the same type (vertical or horizontal) are compacted.
HVBox layout is not natively supported in HTML/CSS, therefore, the boxes must be encoded to create the desired layout. They have a set of modular rules to encode the layout in HTML/CSS such as rules to pre-compute the offset and height/width of an element relative to its parent(enclosing) box, rules to compute the size and margin in percentages of the width of the parent (height is left unconstrained), or rules to mark HTML tags to be *float*.
The authors mention the following four additional implementation considerations:

- *Rounding*: prevent that rounding errors during margin and size calculations cause that a child content overflows its parent.

- *User guidance*: the mockup may be ambigous and not fully capture the designer intent. There may be multiple valid merge choice sequences and therefore multiple feasible layouts. Consequently the algorithm may not obtain the desired layout. The tool allows users to guide the algorithm by indicating which boxes can be merged or not in a configuration file.

- *Browser incompatibilities*: the pages may not be displayed correctly in browsers that do not implement CSS 2.1 completely.

- *Overlapping boxes*: the framework discards overlapping boxes before inferring layout.

The approach has been tested with a mockup builder called Maqetta for a set of web pages constructed by the authors which follow common design patterns extracted from the web. They have also verified its correctness in some up-to-date web browsers. Tests resulted in high-quality replicas of the original mockups in most cases. Sometimes undesired boxes were merged together and user guidance was required, and in other cases fine-grained tweaks were required to fix the layout.

3.1.4 Other approaches

In this section we will show some other works that do not strictly deal with layout inference but they are somewhat related to the topic.
A well-known example of GUI builder with code generation facilities for the NetBeans IDE is *Matisse* [72]. It is a full-fledged design tool that supports the user in the GUI design and which generates code that perfectly fits the design. The generated code is based on the *GroupLayout*, a layout manager which was intentionally introduced to work with IDEs. The tool automatically generates code for Java Swing, particularly based on the *GroupLayout*, and is tied to the NetBeans IDE.
An approach with which to migrate Windows applications to Visual Basic .NET can be found in [1]. Its aim is to replicate the GUI's look & feel by means of mapping runtime objects to .NET objects, so explicit layout recovery is not tackled.
In [73], the authors propose a pixel-based approach based on real-time interpretation of the GUI to identify the hierarchical model of complex widgets. This information is then used to modify an existing GUI (e.g. to translate the text of the widgets) with independence of the interface implementation.
VAQUISTA [74] is a tool which performs the reverse engineering of web pages into XIML [75] models according to flexible heuristics, and requires user interaction during the reverse engineering process. In this case, the source are web pages written in HTML 4 which were laid out with tables, and the tool maps each table cell to a target element, so the table layout is replicated.
In [76] an approach for extracting the web content structure based on the visual representation is proposed, which simulates how users understand web layout structure based on their visual perception. The approach is tightly based on the nature of the HTML code and cannot be applied to coordinated-based interfaces.
Some other related works propose the reengineering of web pages, particularly to adapt them to mobile devices. The following two works fall into this area. In [77] an approach with which to structure web pages in a two level hierarchy is presented, in such a way that if a user selects a part of the web page, this part will be displayed with the screen size like a zoom-in. In [78], a solution

for generating dynamic web migratory interfaces is explained. The authors rely on the analysis of HTML tags in order to split the original web pages in regions that are transformed into web pages with hyperlinks between them. It is worth noting that UI reengineering approaches for web pages work on DOM trees, which are tree-based representations of the HTML code, in which the GUI structure is already explicitly expressed by means of HTML tags.

3.1.5 Discussion

We have presented several works related to reverse engineering of GUIs, and we have focused on three of them that deal with layout inference, which are summarised in Table 3.2. Next we will contrast these approaches and we will indicate desirable features of a layout inference solution. In two of the proposals (Rivero et al., Sinha and Karim) the source technology is a mockup and the target technology is a web technology, whereas in Lutteroth the source technology is a GUI programmed with object oriented code and the target is a desktop toolkit. Two of the approaches (Lutteroth and Rivero et al.) are general, i.e., they can be used with any pair of source/target technologies, and the work of Sinha and Karim is tightly tied to the web target platform. It is clear that a generic solution (not tied to source/target technologies) is desirable. Since hard-coded GUIs and mockups have implicit layouts expressed in pixel coordinates, the same approach could be used for both cases.
With regard to the extracted information, we can see that all these approaches collect some common data (sizes, margins) but they recreate the layout based on different information: Lutteroth uses constraints; Rivero et al. identify the widgets in each grid; and Sinha and Karim extract HTML boxes. We believe that a good layout inference approach should extract all the information we presented in Chapter 2.2.1 explicitly. Granted, some information can be used in place of other one to obtain a similar visual appearance. For example, Lutteroth extracts information about constraints and margins, but it does not get explicit information about the alignment between widgets, so one widget below another one both having the same left margin may look aligned though the layout manager does not explicitly know that they are aligned. Having explicit information about alignment and other features of the source GUI can ease the forward engineering step and led to better adapted layouts.
In Sinha and Karim and Rivero et al., the layout model that is the result of the reverse engineering process is a concrete layout manager model that can be found in numerous GUI frameworks (particularly GridBagLayout and HVFlow) . In contrast, Lutteroth obtains a model with in-

Approach	Lutteroth	Rivero et al.	Sinha and Karim
Source/target independence	Yes [a]	Yes	No (target must be web)
Tested source technology	Hard-coded GUIs (C#)	Mockups (Pencil and others)	Mockups (Maqetta)
Tested target technology	Desktop toolkit	Web (YUI, Ext JS)	Web (HTML/CSS)
Information extracted	Positions, margins, sizes	Containment hierarchy, layout structure	Boxes, margins, sizes
Layout model	ALM (constraint model)	GridBagLayout	HVFlow
Algorithm type	Linear programming, heuristics	Heuristics	Exploratory
Implementation technology	Programming language	Model-based approach	Modular rules
Automation degree	Automated	Automated	Automated

Table 3.1: Summary of layout inference approaches

[a]Requires implementing the layout manager in every target technology

formation about widget constraints. Given that nowadays most GUI frameworks offer layout managers, representing the design of the GUI in terms of layout managers will make the forward engineering step much more easier than using other models such as the ALM model. The proof of concept of Lutteroth generates a C# GUI, which involved the creation of a layout manager in C# to deal with the linear constraints, so in case of using his solution with another target technology, programming the layout manager would be required. This is likely to be a more complex solution than mapping a predefined layout manager (e.g., GridBagLayout in Rivero et al.) to the set of layout managers of the target technology.
The representation used to define the GUI structure (the layout model) has a great impact in the forward engineering step of the process. It must be flexible enough to represent any design, but at the same time it must be close to the well-known existing layout managers in order to make the mapping to other GUI toolkit easy. The works of Sinha and Karim and Rivero et al. rely on single concrete layout managers so the whole reverse engineering process is aimed at generating a design using a certain layout. However, when designing a GUI (either programming or with visual builders) developers do not normally use a single layout manager but a composition of them. Due to this reason, we believe that the layout model should contemplate a set of generic layout managers in such a way that a layout is defined by using the layout managers that are more suitable for the concrete GUI. Moreover, it would also be desirable that the set of layout managers used in the layout model is parameterised. The rationale is to avoid emulating them or implementing new layout managers if they are not available in the target technology.
There is a variety of algorithmic techniques that can be used in the inference approach (linear programming and heuristics in Lutteroth, heuristics in Rivero et al. and an exploratory algorithm in Sinha and Karim), and any of them can be equally valid. The implementation technology may have some importance in the overall solution. Rivero et al. proposes a model-based approach to implement the solution, whereas the others use imperative or object-oriented programming. We think that a model-based approach endows the implementation with additional benefits to implementing good-quality solutions over classical programming. For instance, transformation chains offer a straightforward solution to obtain source/target independence. MDE also brings other benefits such as automation, thanks to the model transformations.
In short, we believe that a good layout inference solution should:

- be source/target independent
- provide explicit information for every layout feature

- use a layout model made up of a variety of layout managers to facilitate the layout definition, which can be selected by developers
- be implemented using a paradigm (e.g., MDE) that provides architectural benefits such as extensibility.

3.2 Analysis of behaviour extraction approaches

In this section we comment on some works which perform some kind of reverse engineering of the GUI behaviour. We will emphasise three of them that we considered more interesting to accomplish the separation of concerns in legacy GUIs, though in the discussion we will take into account the nine works that are mentioned throughout this section, as they can be compared by using the same criteria. Works presenting solutions for code analysis that are not focused on the GUI but other concerns (e.g., business rules) such as [79] and [80], which present C++ static analysis solutions to generate UML models, have been excluded from this discussion.
We will classify each work according to the following four criteria:

1. **Source artefacts**: the source artefacts that are the input of the analysis process (including programming languages and UI toolkits used). For example, Gtk C++ files.
2. **Extracted information**: the output of the analysis. For instance, a state machine model representing the flow of events.
3. **Goal**: the purpose for which the information extracted by the analysis is going to be used.
4. **Analysis type**: It can be static (the source code is analysed statically), dynamic (it analyses information that is collected when executing the code in some way), or hybrid (uses both static and dynamic analysis).

3.2.1 Memon (GUIRipping)

In [7] an approach called *GUIRipping* to reverse engineer a runtime GUI into three models is described, namely *GUI forest*, an *event-flow model* and an *integration tree*[1]. These models, which

[1]The author later refers to all the aforementioned models as an *event-flow model*

we will explain next, are intended to be used to automatically generate test cases. The approach has been implemented in a tool called *GUIRipper*.
The *GUI forest* is a representation that indicates for each window which other windows are opened if performing an event in the former. Two windows are distinguised: *modal windows* and *modeless windows*. The former once invoked monopolise the GUI interaction, whereas the latter do not restrict the user focus.
The author defines a *component* as a modal window together with the modeless windows that have been directly or indirectly invoked from the former. In an *event-flow graph* for a specific component (a modal dialog), the vertices represents all the events in the component. The outgoing directed edges from a vertice represent which vertices can be reached from that vertex (i.e., which events can be performed immediately after the event associated with that vertex). The types of events identified are five:

- *Restricted-focus events*: open modal windows.
- *Unrestricted-focus events*: open modeless windows.
- *Termination events*: close modal windows.
- *Menu-open events*: open menus.
- *System-interaction events*: interact with the underlying software to perform some action.

The *integration tree* is constructed to show the invocation relationships among components (modal dialogs) in a GUI. It is obtained by integrating the information of the GUI forest and the event-flow model. This decomposition of the GUI makes the testing process intuitive for the test designer because he can focus on a specific part of the GUI.
GUIRipper firstly obtains the GUI forest by performing a depth-first traversal of the hierarchical structure of the GUI. The runtime GUI is analysed (e.g., using the Windows API in case of a Windows application) to get the top-level windows, the executable widgets (widgets that invoke other GUI windows), and the windows that are opened by performing events on executable widgets. During the traversal of the GUI, the event type is also determined by using low-level calls. After the automating ripping process has finished, manual inspection is required since some information cannot be extracted by the GUIRipper.
The event-flow model can be used in the definition of *event-space exploration strategies* for automated model-based testing, particularly: i) goal-directed search for model checking, ii) graph-

exploration for test-case generation, iii) operator execution for test-oracle creation. The author delves into these strategies for several scenarios in [81].

3.2.2 Heckel et al.

In [11] a methodology to deal with the evolution of legacy systems to three-tier architectures and Service Oriented Architectures (SOA) is proposed. This methodology is based on the Horseshoe Model introduced in Section 2 and consists of three steps, namely reverse engineering, redesign, and forward engineering, preceded by a preparatory step of code annotation, which can be seen in Figure 3.3.

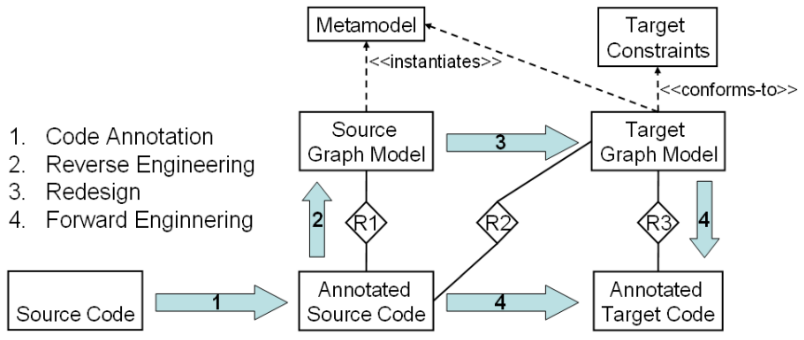


Figure 3.3: Approach of Heckel et al. (extracted from [11]).

The source code elements (packages, classes, methods, or code fragments) are annotated by code categories (step 1) with respect to their architectural function in the target system, e.g., like GUI, application logic or data. Annotations are manually written by developers in the original source code in the form of comments, and they are propagated through the code by categorisation rules defined at the level of abstract syntax trees, so it is not needed for developers to annotate all the source code elements.
From the annotated source code, a graph model is created (step 2), whose level of detail depends on the annotation. The graph model is a reduced Abstract Syntax Tree (AST) representation where the nodes are packages, classes, methods, parameters and variables, and additionally *CodeBlocks* to represent groups of statements, and the edges represent the order of the nodes. Moreover, there is an node type to represent the categories of a code element. Then, all the contiguous statements that are annotated in the same way are grouped in the same *CodeBlock* node, and associated a category. This step is a straightforward translation of the relevant part

of the code into its graph-based representation. The relation between the original (annotated) source code and the graph model (relation *R1*) is kept to support traceability.
During the redesign phase (step 3) the source graph model is restructured to reflect the association between code fragments and target architectural elements. Code categories guide the automation of the transformation process. This transformation is specified by graph transformation rules aimed at performing code refactoring. The relation with the original source code is kept (relation *R2*) in order to support the code generation.
The target code is either generated from the target graph model and the original source code or obtained through the use of refactorings at the code level (step 4). The result of this step is the annotated code of the new system written in the target language.

3.2.3 Morgado et al. (ReGUI)

This work [8] presents a dynamic reverse engineering approach and a tool (*ReGUI*) aimed at diminishing the effort of producing visual and formal representations of the GUI, which enables verification of properties or can serve as the input of Model-Based GUI Testing techniques.

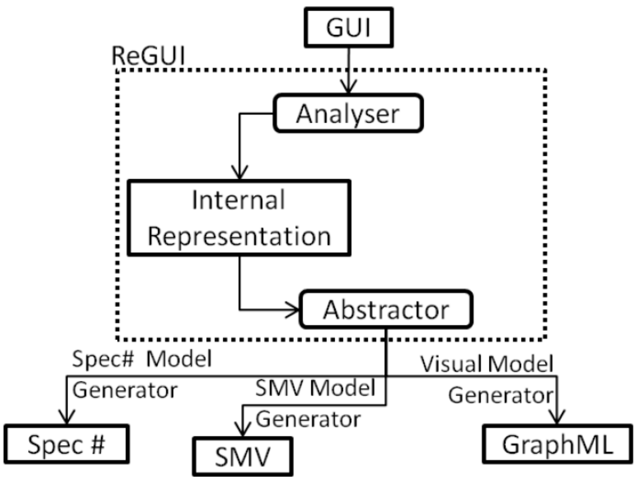


Figure 3.4: Approach of Morgado et al. (ReGUI) (extracted from [8]).

The approach, which is depicted in Figure 3.4, has two main components: the *analyser* and the *abstractor*. The analyser component uses UI Automation, the accessibility framework for the Microsoft Windows operating systems supporting Windows Presentation Foundation. With this framework, the runtime instances of a Windows application can be explored. During the

exploration process, every menu option is navigated to extract its initial state (i.e., enabled or disabled), and each menu option is triggered to verify what windows are opened because of that interaction and also see if the state of any element has changed.
The analyser extracts some information about the GUI elements and their interactions. Particularly, the analyser distinguises two GUI elements: *Windows*, which can be modal or modeless, and *Controls*, which can be menu items or other controls. The interactions between the GUI elements can be of five different types: *Open*, a window is opened; *Close*, a window is closed; *Expansion*, new controls become accessible (e.g., the expansion of a menu); *Update*, one or more properties of one or more GUI elements are updated; *Skip*, nothing happens.
The abstractor component generates different views on the extracted information, which are:

- *ReGUI tree*: represents the different aspects of the structure of the GUI (e.g., the containment hierarchy of a menu).
- *Navigation graph*: stores information about which user actions must be performed in order to open the different windows of the application.
- *Window graph*: is a subset of the information represented in the navigation graph that describes the windows that may be opened in the application.
- *Disabled graph*: its purpose is to show which nodes are accessible but disabled at the beginning of the execution.
- *Dependency graph*: A dependency between two elements means that interacting with the former modifies the value of a property in the latter. This representation shows all the dependencies among controls.

Apart from these views which can be used to inspect the GUI, an Spec# model and an Symbolic Model Verification (SMV) model can be generated. Spec# is a formal specification language that can be used as input to Spec Explorer [82], an automatic model-based testing tool for test generation. An SMV model can be used in combination of model checking techniques to verify properties, which is useful, for example, in usability analysis.

3.2.4 Other approaches

We will summarise other works that analyse UI behaviour and are less relevant to the purpose of separating concerns. First we will comment on two static analysis approaches [2] [30], and then we will oversee four dynamic analysis approaches [83] [49] [6] [31].

In [2] a static analysis for GUIs is presented, which extracts information about the GUI out of the source code. It is targeted at applications written in programming languages such as C/C++ using GUI libraries such as GTK [84] or Qt [85]. The goal is to extract, from the source code, the widget hierarchies forming the windows together with the widget attributes and event handlers. GUI detection is accomplished to determine which types, variables, functions and files are relevant to the GUI. Then ISSA (Interprocedural Static Single Assignment) form is used to detect the widget hierarchy, and also determine the widget attributes and event handlers. After detecting the GUI and obtaining the widget hierarchy, a window graph is created. In this graph nodes are given by windows and indicate that an event raised in the first window can create or show the second window. Edges are labelled hence with events or sets of events. In order to create the outgoing edges for the nodes, the algorithm inspects all the event handlers for the events issued by members of the hierarchy of widgets of the window. An event handler gives rise to an edge if the handler itself or some function directly or transitively called by it creates or shows a window, and if no window is created or shown along the control-flow path in between.

The work presented in [30] proposes an approach to obtain state machines of the transitions between windows based on source code written in Java. The approach is implemented by three tools: *FileParser*, which parses a particular code file, *ASTAnalyser* that slices the Abstract Syntax Tree (AST) obtained by FileParser, and *Graph* which generates metadata files with the state machines. The approach uses Strategic Programming and Program Slicing to isolate the parts of the code which are related to the GUI, in order to make the approach easily retargetable to different programming languages and GUI toolkits. The state machine representation they propose is a graph where states represent windows and transitions include: i) the internal state of the window (it is useful for example to detect windows complexity), ii) the user action that triggers the event, and iii) the condition that must be hold for the transition to occur.

Stroulia et al. [83] propose a method for migrating Text-based User Interfaces (TUIs) in the context of the CelLEST project. These TUIs are part of legacy distributed systems in which there are terminals that interact with a mainframe by means of a communication protocol. Its novelty lies in that it models the system dynamic behavior based on traces of the user interaction with the system, instead of focusing on the system code structure. The reverse engineering phase is based on the analysis of the dynamic traces generated by real user interaction. In order to obtain traces, they propose using an emulator that provides users with a text-based interface that mimics the original hardware terminals used to access the host system, on which the legacy application resides, by implementing the protocol of communication between the host and the emulator user interface. The emulator is instrumented so that it also records the interaction between the legacy application and its users. A trace recorded by this emulator consists of a sequence of snapshots of the screens forwarded by the legacy application to the user's terminal. Between every two snapshots, the user keystrokes are recorded. The result is a model of the TUI behaviour represented as a directed state-transition graph. The graph nodes correspond to the distinct interface screens, which are identified by clustering all the screen snapshots, contained in the recorded trace according to their visual similarity. Each edge of the graph corresponds to an action that can be taken, i.e., a command that can be executed when the source-screen node is visible to the user and leads to the destination-screen node.

A GUI test generation approach based on symbolic execution is presented in [6]. The GUI testing framework (named Barad) generates values for data widgets and enables a systematic approach that uniformly addresses the data-flow as well as the event-flow for white-box testing of a GUI application. The approach is applied to Java event handlers. Firstly, the event handler bytecode is instrumented, i.e. it is modified to execute a custom code after every sentence. During the instrumentation, they generate an inline version (with branching statements removed) of the program with primitives, strings, and conditional instructions replaced with the corresponding symbolic values. Then, the code is symbolically executed. Basically, symbolic execution uses symbolic values instead of actual data, and represents the values of program variables as symbolic expressions. The symbolic execution is performed by applying a chronological backtracking that visites all the branches of the program. For a branch to be explored, the set of constraints of the states must be satisfied. When an entire branch has been executed, the test case for that branch is generated, and the program state (the set of values of the variables) is restored. After test cases have been generated, some heuristics to reduce the test suite are applied. The resulting suite maximises the code coverage while minimising the number of tests needed to systematically check the GUI.

In [49] the authors present a reverse engineering approach for abstracting Finite State Machines representing the client-side behaviour offered by Rich Internet Applications (RIAs). The reverse engineering process consists of two activities: extraction and abstraction. During the extraction activity, the user interacts with the RIA in a controlled environment and the sequences of events are registered. The abstraction activity is composed of three tasks: RIA Transition Graph building, Clustering, and Concept assignment. The first task builds the Transition Graph from the traces stored in the extraction activity. This graph models the flow of RIA views that

were generated. The second task analyses the Transition Graph and clusters the nodes and edges that are equivalent. The Finite State Machine models the event listeners that are associated with DOM elements of a web page, which can be: user events listeners, time event listeners (due to the occurrence of timeout conditions) and HTTP response event listeners (due to receptions of responses to some HTTP request). It also models the transitions between web pages and the events that caused those transitions. These events can be associated to web page requests (traditional HTTP requests) or XmlHttpRequests (asynchronous Ajax requests).
Mesbah et al. [31] describe a technique for crawling Ajax-based applications through automatic dynamic analysis of user interface state changes in web browsers. The analysis process infers a state machine that models the navigational paths within an Ajax application, which can be used in program comprehension, analysis and testing. The analysis works in the following way. Firstly, the Controller traverses the web page to find clickable elements, which are elements that have event listeners and can cause a state transition. For each element, the crawler instructs the Robot to fill in the form fields and fire events on the elements in the browser. When the events are triggered in the clickable elements, changes in the DOM tree are produced. Then the DOM Analyzer compares the current DOM tree and the previous one by using some heuristics. If a state change is detected, a new state is created and added to the state machine. If a similar state is recognised, that state is used for adding a new edge (no new state is created). The algorithm uses backtracking to recursively traverse all the code branches until all the code is executed. When applying backtracking, the DOM tree has to be set to a previous state. This is achieved by using the browser history if the Ajax application has support for it, or reproducing the event sequence from the initial state in contrary case.

3.2.5 Discussion

We have reviewed some of the most relevant approaches up to date about reverse engineering and reengineering of UI behaviour. Now we will make some reflections about these works.
First of all, we see that the majority of the works (6 out of 9) coincide in representing the behaviour by means of some sort of state machine (transition graph) where the states represent views and the transitions represent the events that trigger the changes. The granularity of the states and events represented differs between the different works. For instance, in [83] events represent transitions between complete views, so the state machine is used as a model of the navigation among them. In contrast, in [31] events represent changes in parts of a view, as it

Approach	Source artefacts	Extracted information	Goal	Analysis type
Memon et al.	Runtime GUI (Java/Windows)	Transition graph	Testing	Dynamic
Heckel et al.	Annotated code (Java)	AST-like graph with code categories	Migrate to 3-tiers	Static
Morgado et al.	Runtime GUI (Windows)	Interaction model	PC, verif. properties	Dynamic
Staiger	GTK/Qt code (C/C++)	Widget hierarchy, transition graph	Maintenance	Static
Silva et al.	Java code	Transition graph	PC., testing	Static
Stroulia et al.	TUI runtime traces	Transition graph	Migration to the web	Dynamic
Ganov et al.	Java bytecode	Symbolic tree, test suite	Test generation	Dynamic
Amalfitano et al.	Instrumented RIA	Transition graph	Maintenance, testing	Dynamic
Mesbah et al.	Ajax web applications	Transition graph	PC, analysis, testing	Dynamic

Table 3.2: Summary of the behaviour extraction approaches (*PC* stands for *Program Comprehension*)

happens in Ajax applications, which has a much smaller granularity level than in the previous work. In [8] several models that focus on specific behaviour are even created, such as a model to know which elements that are disabled at the beginning are accesible after a sequence of events. Therefore, depending on the purpose of the reverse engineering or reengineering, different information represented in the form of a state machine may be useful.

With regard to the goal of the reverse engineering, most of the works are aimed at perform testing or program comprehension (7 out of 9), and a few works (2 out of 9) are targeted at generating a new system. In [11] the separation of legacy applications in layers in order to generate web applications is proposed, and the idea of abstracting the source code in a model that guides the generation of the new system is introduced. It is worth remarking that different from the rest of the works, it addresses a separation of concerns, particularly from the point of view of the architecture of the application (business logic, UI, data access). In that work, the reverse engineering is assisted by the developer, that must tag the code parts so the tool knows which layer the code belongs to. This procedure is useful, but developers must spend time in inspecting the whole code by hand.

Most approaches (6 out of 9) are based on dynamic analysis while the rest apply a static one. This is due to it is easier to determine which views are displayed from other views with dynamic analysis. In general, static and dynamic analysis provide us with different kinds of information: static analysis can access to all the code (which can be executed or not) so the information of all the possible states of the application is available, whereas dynamic analysis can obtain data about every state that is reached by execution. Moreover, when no source code is available, dynamic analysis is the only option. An scenario in which static analysis is not enough to obtain proper information is the reverse engineering of Ajax applications [31], and in that case, also dynamic analysis is required. On the other hand, static analysis can access to all the code, which is necessary to accomplish a faithful migration of the code. In addition, static analysis is faster and easier to perform than dynamic analysis that implies executing the code and maybe redeploying the application or running the source runtime platform (e.g., the Oracle Forms runtime environment).

To sum up the aforementioned approaches, we reckon that an the extraction of the behaviour of the GUI aimed at migration should:

- separate the different concerns that are tangled in the code of event handlers, but different from [11], marking code by hand should be avoided.

- represent the transtition between views and dependencies between widgets by means of a state-machine-like representation, as there is a wide consensus about that.
- static analysis is desirable if source code is available, given that we need the whole information about the GUI and the runtime information is not enough.

3.3 GUI representation approaches

This section is devoted to describe well-known metamodels (KDM, IFML) and User Interface Description Languages (UIDLs) that can be used to represent user interfaces. We will also introduce the Cameleon framework, though it is neither a metamodel nor a UIDL, it establishes different abstraction levels that are desirable for modelling user interfaces and it is used by many UIDLs. Since these approaches are rather heterogeneous, we are not going to classify them as we did in the previous sections, but we will restrict ourselves to describe them and put some examples.

3.3.1 Knowledge Discovery Metamodel (KDM)

Section 2.3.4 introduced KDM as the core element of the ADM initiative. KDM is a metamodel aimed at representing software systems at different levels of abstraction which range from program elements to business rules. KDM is intended to facilitate the interoperability between software modernisation tools, as a common representation for software artefacts.
It is a very large metamodel that is composed of twelve packages organised in four layers: *Infrastructure, Program elements, Runtime resources* and *Abstractions* (see Figure 3.5). Each package defines a set of metamodel elements whose purpose is to represent a certain independent facet of knowledge related to existing software systems. The packages defined in the specification are:

- **Core** and **Kdm**: define common elements that constitute the infrastructure for other packages.
- **Source**: enumerates the artefacts of the existing software system and defines the mechanism of traceability links between the KDM elements and their original representation.

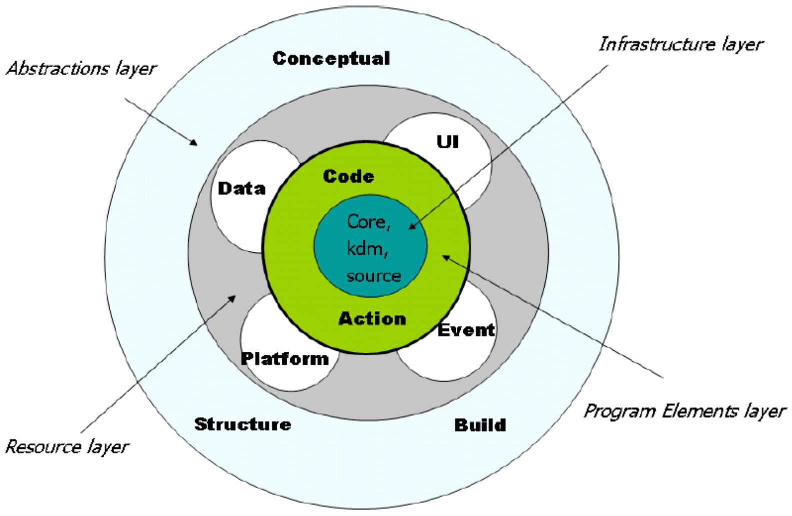


Figure 3.5: KDM layers and packages (extracted from [71]).

- **Code**. It is focused on representing common program elements supported by various programming languages, such as data types, data items, classes, procedures, macros, prototypes, and templates, and several basic structural relationships between them.

- **Action**: Along with the *Code* package, it represents the implementation level assets of the existing software system. This package is focused on behaviour descriptions and control and data-flow relationships determined by them.

- **Platform**: defines a set of elements whose purpose is to represent the runtime operating environments of existing software systems.

- **UI**: represents facets of information related to user interfaces, including their composition, their sequence of operations, and their relationships to the existing software systems.

- **Event**: it specifies the high-level behaviour of applications, in particular event-driven state transitions.

- **Data**: it is used to describe the organisation of data in the existing software system.

- **Structure**: it is aimed at representing architectural components of existing software systems, such as subsystems, layers, packages, etc. and define traceability of these elements to other KDM facts for the same system.

- **Conceptual**: it provides constructs for creating a conceptual model during the analysis phase of knowledge discovery from existing code.

- **Build**: represents the facts involved in the build process of the given software system (including but not limited to the engineering transformations of the "source code" to "executables").

From the point of view of the migration of graphical user interfaces, four of these packages can be useful: the *Code*, *Action*, *UI* and *Event* packages. The *Code* and *Action* packages can be used together to represent programming code with independence of the specific programming language. The UI package was conceived to represent the elements and behaviour of the GUIs. Next we will deep into the UI package to analyse its usefulness. Figures 3.6, 3.7 and 3.8 compose the UI package. As there are a lot of dependencies among several the packages, we will only briefly comment on those metaclasses that are relevant for us.

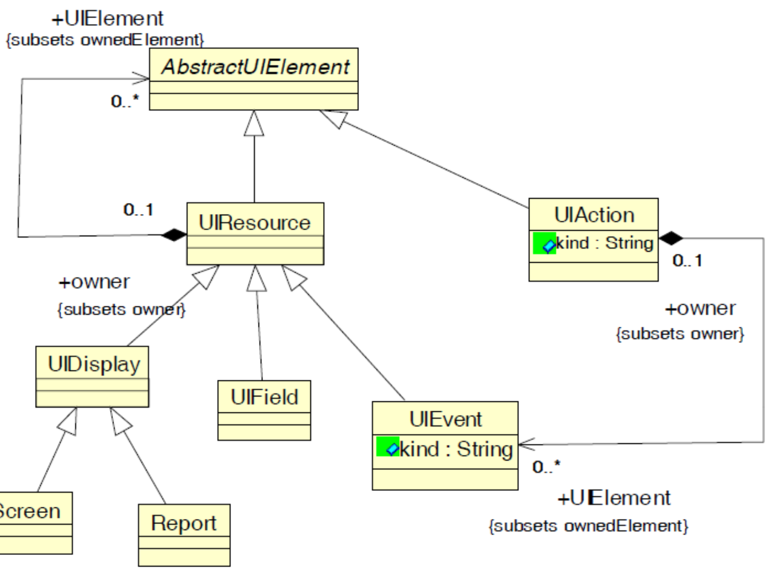


Figure 3.6: KDM metamodel. UI package (UIResources)(extracted from [71]).

Figure 3.6 shows the *UIResources* that can be defined: *Screens, Reports, UIFields, UIEvents. Screens* are units of display in an application, such as windows or web pages, and *Reports* are printed units of display, like a printed report. *UIField* is a generic element to represent any field in a *Screen* or *Report*, such as a text field or a combo box. *UIEvents* can be declared and 'be' associated with a *UIAction*; a *UIAction* can have associated zero or more events (e.g., a *UIAction* called 'navigate' can be triggered by many *UIEvents* such as 'click' or 'select'). Note that UI resources can contain other *UIResources* (e.g., a *Screen* can contains *UIFields* that in turn contain *UIEvents*).

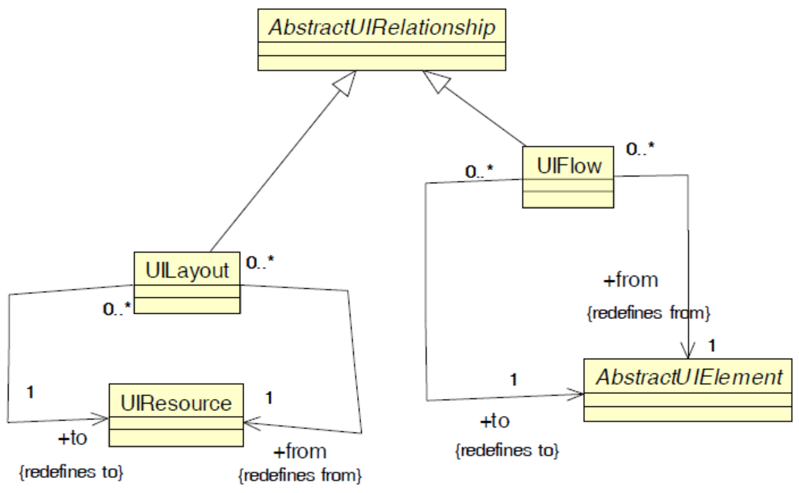


Figure 3.7: KDM metamodel. UI package (UIRelations)(extracted from [71]).

In Figure 3.7 there are two generic relationships, *UILayout* and *UIFlow*. *UILayout* indicates the layout of a *UIResource*, and *UIFlow* allows defining the flow of Screens (without indicating the event that originated it).

The diagram of Figure 3.8 defines several relationships between a *UIResource* and *ActionElement*. The latter is defined in the *Actions* package and refer to a block of code. These relationships represent the effect of a block of code in the UIResources: *Displays* allows a UIResource to be shown, *ReadsUI* takes the value of a *UIField*, *WritesUI* puts a value in a *UIField*, *DisplaysImage* shows an image, and *ManagesUI* represents other accesses to the *UIResources*.

As it can be seen, the *UI* package can be used to represent the logical structure of views, the spatial relationships among the UI elements (layout), and the events associated with them. However, the specification just offers a few generic concepts for them. For example, related to the logical structure it defines *Screen* as a container and *UIField* as a generic widget, and related to the layout of the elements it defines a generic layout (*UILayout*).

Finally, the *Event* package, is not aimed at expressing the event flow of the GUI (which is actually addressed in the UI package), but is aimed at describing the behaviour of the entire system as a state machine. It could be used somehow to express the behaviour of the UI, though it was not

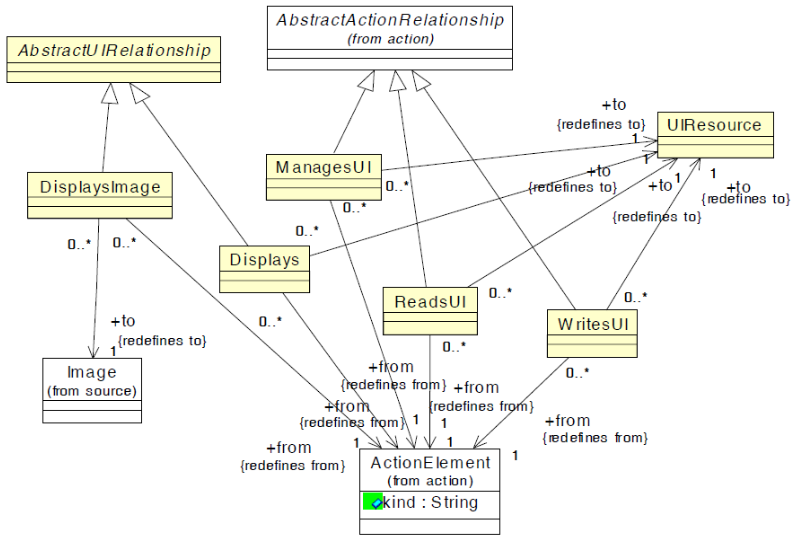


Figure 3.8: KDM metamodel. UI package (UIActions) (extracted from [71]).

conceived to that goal.

3.3.2 Interaction Flow Modeling Language (IFML)

The *Interaction Flow Modeling Language (IFML)*[86] has been recently adopted (March, 2013) as an OMG specification for building visual models of user interactions and front-end behavior in software systems. As indicated in [87], IFML can be seen as the consolidation of the Web Modelling Language (WebML) [88] defined and patented about 15 years ago as a conceptual model for data-intensive web applications. In fact, WebRatio, which has been supporting WebML over the years, is now adopting IFML as official notation.
The objective of IFML is to provide system architects, software engineers, and software developers with tools for the definition of Interaction Flow Models that describe the principal dimensions of an application front-end: the view part of the application, made of containers and view components; the objects that embody the state of the application and the business logic actions that can be executed; the binding of view components to data objects and events; the control logic that determines the sequence of actions to be executed after an event occurrence; and the distribution of control, data and business logic at the different tiers of the architecture.
An IFML diagram consists of one or more top-level view containers. Each view container can be internally structured in a hierarchy of sub-containers. The child view containers nested within a parent view container can be displayed simultaneously or in mutual exclusion. A view container

can contain view components, which denote the publication of content or interface elements for data entry (e.g., input forms). A view component can have input and output parameters. A view container and a view component can be associated with events, to denote that they support the user's interaction. Events are rendered as interactors, which depend on the specific platform and therefore are not modeled in IFML but produced by the PIM to PSM transformation rules. The effect of an event is represented by an interaction flow connection, which connects the event to the view container or component affected by the event. The interaction flow expresses a change of state of the user interface: the occurrence of the event causes a transition of state that produces a change in the user interface. An event can also cause the triggering of an action, which is executed prior to updating the state of the user interface. An input-output dependency between view elements (view containers and view components) or between view elements and actions is denoted by parameter bindings associated with navigation flows (interaction flows for navigating between view elements).

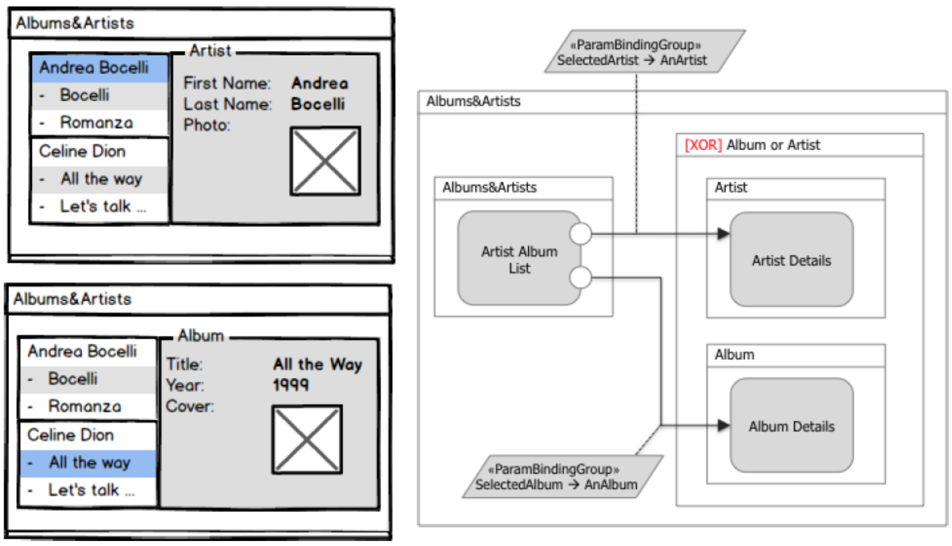


Figure 3.9: Example of user interface (left) and corresponding IFML model (right) (Extracted from [86]).

The left part of Figure 3.9 shows two states of the same view, and the right part represents the IFML diagram. In the example there is one top-level container (Albums&Artists) that comprises three view containers: one with a list of artists and of their albums, one with the details of an artist, and one with the details of an album. The latter two view containers are mutually exclusive, so if a user selects an artist, the details of that artist are displayed, or if the user selects an album, the details of the album are displayed.

3.3.3 Cameleon framework

The Cameleon framework [89] is a model-based approach devised to cover the design, maintenance and evolution of a multi-target user interface. This framework does not describe concrete metamodels but recommends an architecture of models and the way it can be used to deal with forward engineering and reengineering of user interfaces. The overall architecture is shown in Figure 3.10 (arrows indicate which models originate other ones). Three types of models are differentiated: *ontological*, *archetypal* and *observed*.

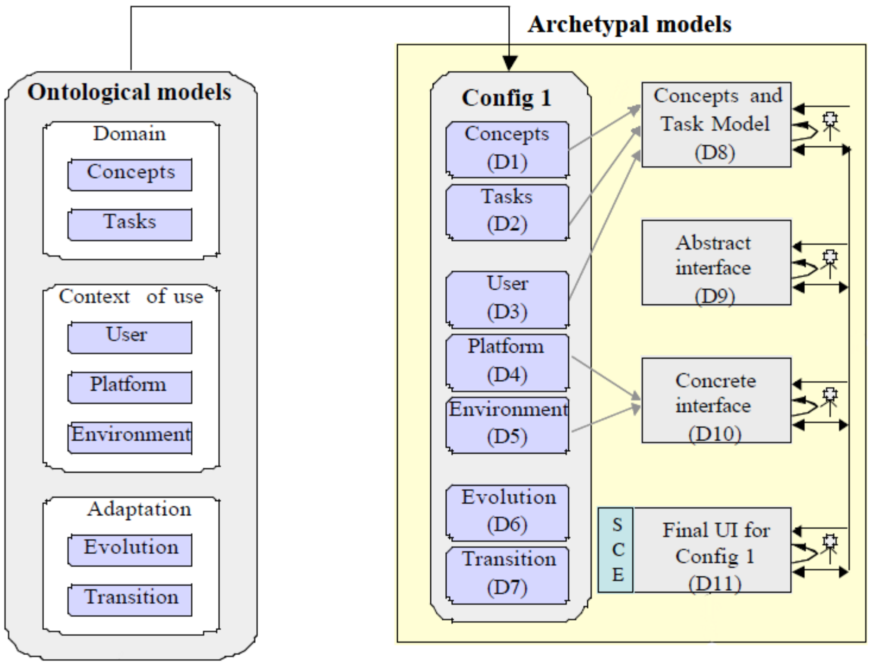


Figure 3.10: Cameleon framework (extracted from [89]).

- The **ontological models** (left side of the figure) are metamodels of the concepts (and their relationships) involved in a multi-target UI. These models are instantiated into archetypal and/or observed models, which depend on the domain and the interactive system being developed.

- **Archetypal models** are declarative models that serve as input to the design of a particular interactive system. They are instances of the ontological models for a specific target.
- **Observed models** are executable models that support the adaptation process at run-time. They have been omitted in Figure 3.10 because they are out of the scope of our work and will not be explained.

The types of ontological models are:

- **Domain Models**: cover the domain concepts and users tasks. Domain concepts denote the entities that users manipulate in their tasks. Tasks refer to the activities users undertake in order to reach their goals with the system.
- **Context of use Models**: describe the context of use in terms of the user, the platform and the environment.
- **Adaptation Models** specify the reaction to adopt when the context of use changes. It includes information about the new UI to switch to, and the particular transition UI to be used during the adaptation process.

The ontological models are independent of any domain and interactive systems, and define key dimensions for a given retargeting. On the contrary, archetypal models are instances of the ontological models in a specific context (a specific domain, platform, etc.). The information of the archetypal models is used to express a UI at four levels of abstraction, from the task specification to the running interface:

- **Task and Concepts level**. It corresponds to the Computational-Independent Model (CIM) in MDA [15] and considers: (a) the logical activities (tasks) that need to be performed in order to reach the user goals and (b) the domain objects manipulated by these tasks. Often tasks are represented hierarchically along with indications of the temporal relations among them and their associated attributes. This level uses the information of the Concepts, Tasks and User models.
- **Abstract User Interface (AUI)**. Corresponding to the Platform-Independent Model (PIM) in MDA, is an expression of the UI in terms of *interaction spaces* (or *presentation units*), independently of which interactors are available and even independently of the

modality of interaction (graphical, vocal, haptic, etc.). An *interaction space* is a grouping unit that supports the execution of a set of logically connected tasks.

- **Concrete User Interface (CUI)**. It corresponds to the Platform-Specific Model (PSM) in MDA. It is an expression of the UI in terms of *Concrete interactors*, that depend on the type of platform and media available and has a number of attributes that define more concretely how it should be perceived by the user. *Concrete interactors* are, in fact, an abstraction of actual UI components generally included in toolkits. The CUI model uses the information of the Platform and Environment models.

- **Final User Interface (FUI)**. It is related to the code level in MDA and consists of source code, in any programming language or mark-up language (e.g. Java, HTML5, VoiceXML, X+V, ...). It can then be interpreted or compiled. A given piece of code will not always be rendered on the same manner depending on the software environment (virtual machine, browser, etc.). For this reason, Cameleon considers two sublevels of the FUI: the source code and the running interface.

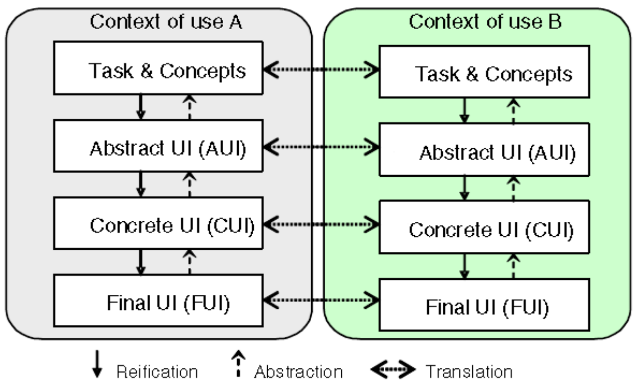


Figure 3.11: Abstraction, reification and translation in the Cameleon framework (extracted from [89]).

When using Cameleon in a development, three different paths can be followed, namely *reification*, *abstraction* and *translation*, which are depicted by downward, upward and bidirectional arrows in Figure 3.11. Reification is the transformation of a description (or of a set of descriptions) into another one that has a less abstract than the former. Abstraction is the transformation of a description into another one whose semantic content and scope are higher than the content and scope of the initial description content (i.e., is more abstract). In the context of reverse engineering, abstraction is the elicitation of descriptions that are more abstract than the artefacts that serve as input to this process. Finally, a *translation* shifts the interface from one type of platform to another, or more generally, from one context to another (e.g., a legacy UI migration).

3.3.4 User Interface Description Languages (UIDLs)

UIDLs are DSLs for defining user interfaces. In the following subsections we present three of the most widespread UIDLs: UsiXML [38], Maria [90] and XAML [91]. Some other examples of UIDLs are: User Interface Markup Language (UIML) [92], eXtensible Interface Markup Language (XIML) [75], eXtensible Interaction Scenario Language (XISL) [93] and XML User Interface Language (XUL) [94].

3.3.4.1 UsiXML

User Interface eXtensible Markup Language (UsiXML) is a DSL used in Human-Computer Interaction (HCI) and Software Engineering (SE) in order to describe any user interface of any interactive application independently of any implementation technology. The language is able to represent user interfaces which vary on the context of use (in which the user is carrying out her interactive task), the device or the computing platform (on which the user is working), the language (used by the user), the organisation (to which the user belongs), the user profile or the interaction modalities (e.g., graphical, vocal, tactile or haptics).
UsiXML has following features which are interesting in GUI migrations:

- *Model-driven*: it is defined according to the principles of MDE. Metamodels are expressed in MOF and OWL 2.0 Full [95].

- *Multi-level of abstraction*: it is compliant with the four levels of abstraction of the Cameleon framework (as it is shown in Figure 3.12). It provides a metamodel for Abstract User Interfaces, and a metamodel for Concrete User Interfaces which can be used for different modalities (graphical, vocal, haptic, etc.). The Task level is based on Concur Task Trees (CTT) [96] and the domain is expressed with UML class and object diagrams [97].

- *Complete lifecycle support*: it provides means for conceptual modeling of task, domain abstract user interface, concrete user interface, and contexts of use as defined in the Cameleon framework. In addition, it covers transformation, mapping, adaptation, and interactor modeling, so all the paths of reengineering (reverse engineering, restructuring, and forward engineering) can be tackled by means of the UsiXML metamodels and tools.

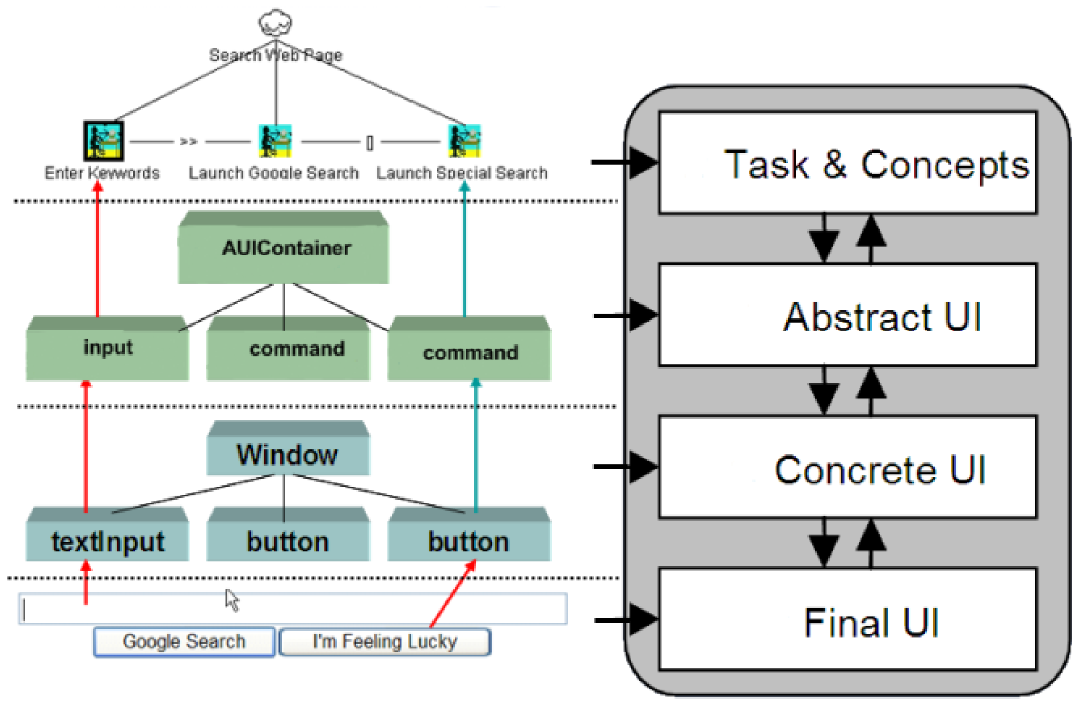


Figure 3.12: UsiXML models conforming to Cameleon (extracted from [38]).

Figure 3.12 shows an example of the four levels of the Cameleon framework in UsiXML. In the bottom part of the figure we see an HTML form with a text field and two buttons to perform a search. These controls are represented in the CUI level with independence of the concrete technology (HTML). The AUI level abstracts the elements of the CUI model so they are independent of the modality (GUI controlled by keyboard and mouse). Finally, the Task level captures the sequence of tasks to perform a search, this is, write some keywords in the text field and then click on the one of the two buttons.

There is a variety of tools supporting UsiXML for creating the models (IdealXML, KnowiXML), obtaining UsiXML models from code (ReversiXML) or other representations (SketchiXML, VisiXML, TransformiXML, etc.) or generating new systems (FormiXML, GrafiXML, FlashiXML, etc.).

3.3.4.2 Maria

MARIA, Model-based lAnguage foR Interactive Applications [90] is a universal, declarative, multiple abstraction-level, XML-based language for modelling interactive applications in ubiquitous environments. The language inherits the modular approach of its predecessor, TERESA XML [98], with one language for the abstract description and then a number of platform-dependent languages that refine the abstract one depending on the interaction resources considered.
Some features of the language that are relevant for our purposes are:

- *Model-driven*: the language has been described by means of MOF metamodels.
- *Multi-level of abstraction*: MARIA conforms to the Cameleon framework and defines metamodels for the four abstraction levels: defines an Abstract description metamodel, a few Concrete description metamodels for the desktop, mobile, vocal and multimodal platforms, and relies on CTT for the Task level.
- *Events at abstract and concrete levels*: an event model has been introduced at different abstract/concrete levels of abstractions. The introduction of an event model allows for specifying how the user interface responds to events triggered by the user.
- *Extended Dialog Model*. The dialog model contains constructs for specifying the dynamic behaviour of a presentation, specifying what events can be triggered at a given time. The dialog expressions are connected using CTT operators in order to define their temporal relationships.
- *Continuous update of fields*. It is possible to specify that a given field should be periodically updated invoking an external function (i.e., it supports Ajax scripts). This can be defined at the abstract level and detailed at the concrete level.
- *Dynamic Set of User Interface Elements*. The language contains constructs for specifying partial presentation updates (dynamically changing the content of entire groupings) and the possibility to specify a conditional navigation between presentations. This is useful for supporting Ajax techniques.

The Maria language is supported by the Maria tool.

3.3.4.3 XAML

Extensible Application Markup Language (XAML) is a markup language developed by Microsoft for declarative programming of user interfaces in the .NET framework. XAML is used extensively in .NET Framework 3.0 and.NET Framework 4.0 technologies, particularly in the Windows Presentation Foundation (WPF) [99], Silverlight, Windows Workflow Foundation (WF), Windows Runtime XAML Framework and Windows Store apps. In WPF, XAML forms a user interface markup language to define UI elements, data binding, events, and other features. In WF contexts, XAML is used to describe potentially long-running declarative logic, such as those created by process modeling tools and rules systems.
The scope of this language is more ambitious than that of most user interface markup languages, since program logic and styles are also embedded in the XAML document. Functionally, it can be seen as a combination of XUL, SVG, CSS, and JavaScript into a single XML schema. XAML directly represents the instantiation of objects in a specific set of backing types defined in assemblies (.NET libraries). This is unlike most other markup languages, which are typically an interpreted language without such a direct tie to a backing type system.
XAML is supported by the Microsoft environments such as Visual Studio and can also be used to generate desktop applications, Silverlight applications, Windows Phone apps and Windows Store apps among others.

3.3.5 Discussion

We have presented the different approaches we analysed for representing GUIs. We found though different disadvantages that led us to eventually discard them and define our own metamodels. Next we enumerate the reasons for this decision.
KDM is a complex metamodel which can be used to model an entire software system. The *Code* and *Action* packages, which are the most extensive of KDM, can be used to represent programming code in a generic fashion. However, given that KDM intends to be language-independent, some of its packages are too generic to be useful as-is, and need to be extended in some way. For instance, we could see that the UI package does not offer a widget or layout classification, so if distinguishing the different widget types or layouts is needed, extending KDM is required. KDM itself offers an extension mechanism, but as mentioned in [66] it is poor in practice and using it means losing the interoperability among tools, which is one of the presumed benefits of KDM. Moreover, using such a large metamodel like KDM involves a lot of unnecessary com-

plexity which most of the times does not pay off (e.g., model transformations become far more complex than using a simple metamodel). On the other hand, representing event handlers is awkward in KDM, because it is possible to define which events are triggered by each widget, but we cannot specify the code that is executed in each case.
With respect to IFML, it allows expressing the events and the effect they produce in the GUI, and it can be considered to model the behaviour of the GUI in a technology-independent fashion. Since it has recently appeared, it was not considered in our solution.
Regarding the UIDLs, UsiXML and Maria are technology-independent languages which have been designed to cope with multi-modal UIs in ubiquitous environments. A forte of UsiXML is that it has several graphical DSLs supporting the creation of the different models, and an interesting feature of Maria is that it includes elements to deal with Ajax applications. Both offer a wide widget hierarchy, but the layout representation is somewhat limited. For instance, UsiXML 2.0 just offers a generic *TableLayout* (similar to HTML tables) to represent layout. For this reason, these UIDLs are not suitable to be used in the reverse engineering stage as intermediate representations to manipulate the data. On the contrary, both, UsiXML and MariaXML could be used to represent a generic GUI at a CUI level (i.e., the technology-independent level, which is the abstraction level in which our reverse engineering proposal is enclosed). On the other hand, XAML is a UIDL which is devised to work with Windows frameworks, and includes information that is dependent of those frameworks. Moreover, it has a complex specification due to it mixes different kinds of information, and in fact some people are critical of this design, as many standards (Javascript, CSS, etc.) exist for doing these things. For all these reasons, XAML is not suitable for representing generic GUIs.

Fall in love with the problem, don't fall in love with the solution.

Paul Graham
(Suggested by Jérémie Melchior)

4 Overview

So far we have set the context of this thesis. In the introductory chapter we motivated and stated the problem that is tackled. There we introduced two main issues to be addressed when migrating RAD applications to modern platforms: coordinate-based layouts and tangling of concerns. These two issues were explained in more detail in Section 2.2.2, where we discussed the main features of legacy RAD applications that were summarised in Table 2.1.
Now we will clearly define the requirements that we expect in our solution, and we will outline the generic architecture of the solution. This chapter serves as a brief guide of the entire work and summarises the solution that we will describe in detail in the following chapters.

4.1 Goal

Our main goal is to develop a migration framework for GUIs of legacy systems built with RAD environments, in order to migrate them to modern platforms and/or different GUI frameworks in such a way that the implementation of the new system follows common best practices. As stated in Section 1.2, to reach this objective we have identified three high level goals: to define an architecture (goal *G1*) for the framework to be developed, which should separate and make explicit the different aspects involved in the GUI of RAD applications, and that should deal with the layout and event handlers (goals *G2* and *G3*). This implies addressing the two aforementioned issues: coordinate-based layouts and tangling of concerns. We propose to apply static analysis on the GUI-related artefacts of the source legacy systems in order to extract relevant information for implementing the migrations. Particularly, we intend to analyse the view definitions in order to extract a layout definition, and analyse the code of event handlers to obtain an abstract representation that let us to achieve separation of concerns and get other useful information such as the navigation flow. Granted, a requisite to apply our solution is that source code is available.

From the examination of GUIs of RAD applications expounded in Section 2.2.2 and the study of the state of the art presented in Chapter 3 we have elicited a set of requirements for our solution, which can be organised in three groups: general requirements, requirements specific to the layout inference, and requirements of the analysis of event handlers. Hence, our solution is driven by the following general requirements:

(R1) **Explicit GUI information**. A high-level representation of the GUI must be discovered, i.e., metadata concerning the GUI. The metadata should be of interest for migrating the GUI to different platforms and GUI frameworks. It must be possible to analyse and automatically transform this metadata.

(R2) **Modularity**. Owing to the wide semantic gap between RAD environments and current technologies, it would be desirable to split the reengineering process into simpler stages to make it maintainable. In addition, a solution split in decoupled stages would facilitate extension (for instance, to add new processing stages) and reusability in different projects.

(R3) **Automation**. We are interested in automating it as much as possible so that it can be easily applied to a large number of applications with minimum effort. Ideally, the process would end up in a generation task that would produce artefacts that could be seamlessly integrated in the new system.

(R4) **Source and target independence**. The reengineering process should be easy to reuse with different technologies (source independence). Furthermore, it must be extensible, so that new target platforms can be added without changing the reverse engineering and restructuring stages (target independence).

With respect to the coordinate-based layout issue, we expect to fulfill the following requirements:

(R5) **Matching between the visual and logical structure**. The logical structure of the views (the GUI tree, i.e., the nesting of the widgets) must mirror the visual structure that users perceive when they see those views.

(R6) **High-level layout representation**. The layout of the views must be represented in terms of high-level structures that ensure a proper visualisation in different screen sizes and resolutions, such as the well-known layout managers that are used at present in a myriad of GUI frameworks.

(R7) **Misalignment tolerance**. The solution proposed must take into account that some graphical editors (e.g., RAD editors) do not include alignment guidelines and therefore some minor misalignments can occur if developers are not careful. Then, the solution must allow certain degree of imprecision when recognising widget location.

(R8) **Alternative solutions**. It would be useful that the layout inference solution could output different ranked alternatives in order to know which options are better according to some criteria. Then, if the solution marked as 'best' does not produce the desired results, developers could inspect the other options and choose a different alternative.

(R9) **Configurable layout set**. Developers should be able to choose which layout managers can be used for laying out views. This can be useful if some layout types are not available in the target toolkit or if using certain types can result in awkward or unexpected design.

Regarding the issue of tangling of concerns, we intend to endow our solution with the following features:

(R10) **Code abstraction**. Code should be abstracted so it is possible to understand what it does (how it works). This abstraction consists of moving the code representation ('how to do it') to the intention of the code ('what it does'). For example, opening a database cursor is a recurrent pattern in PL/SQL, which typically requires several instructions. From the reverse engineering and restructuring point of view it is useful to know that these statements just perform a database access. Raising the abstraction level in this way facilitates later processing.

(R11) **Code categorisation**. Related to the previous requirement, the solution must provide automated categorisation of pieces of code, so separation of concerns in the source system can be enabled. In this way, it should be possible to differentiate between statements related to the GUI, the control or to business logic in order to structure the new system in multiple tiers (n-tier architecture). In [11] this is regarded as an important activity to disentangle spaguetti code. Furthermore, sub-concerns of the GUI such as validation rules should be also detectable.

(R12) **Explicit interaction and navigation flows**. The solution must be able to explicitly represent the interactions that exist between widgets and the transitions that may occur between views. For example, this is useful for migrating to many modern frameworks that provide a means to declaratively express the navigation flow (e.g., Java Server Faces).

These requirements cover the ones that we extracted in the discussion of the state of the art for layout recognition approaches and behaviour extraction approaches, as it can be seen in Table 4.1.

Source/target independence	**R4**
Provide explicit information	**R1**
Layout model with a variety of layout managers	**R6, R9**
Use of implementation paradigm with architectural benefits	**R2, R3, R4**
Separate tangled concerns	**R11**
Represent transitions and dependencies with State Machine	**R12**

Table 4.1: Relationships between the requirements and the discussion of the state of the art.

In Table 4.2 we also show how these requirements try to address some of the bad practices that are present in the applications created with RAD environments.

Implicit layout	**R1, R6**
Overlapping	**R5**
Widget-database links	**R11**

Table 4.2: Requirements that cover bad practices in RAD environments.

4.2 Architecture of the solution

In this section we will present the architecture of the framework we have devised for migrating legacy GUIs, which will deal with the aforementioned requirements, and which is called *GUIZMO (GUI to MOdels)*. The MDE paradigm presented in Section 2.3 provides mechanisms (mainly metamodels and transformations) and benefits which fit the architectural requirements *R1, R2, R3 and R4*. Therefore, we decided to implement our solution with an MDE-based architecture because it is suitable to cover these requirements.
The summary of this section is as follows. Firstly we present the Concrete User Interface (CUI) model that we use to represent GUIs. Secondly, we will show how the CUI model is integrated in the context of the migration of legacy GUIs by means of the MDE-based architecture. Finally, we will indicate how the requirements we have listed are fulfilled by GUIZMO.

4.2.1 The Concrete User Interface model

According to the Cameleon reference framework introduced in Section 3.3.3, a *CUI model* is a technology-independant representation of a GUI that can be seen as an abstraction of the Final User Interface (FUI). Actually, we have not devised a single CUI model, but a series of models arranged in a star. As it can be seen in Figure 4.1, the CUI model has a base model representing the structure of a GUI and different models connected to it that represent aspects of that GUI that are interesting to cope with in a migration. In this thesis we have dealt with three aspects: layout, event concerns and interactions. Validation and style are other aspects that should be considered in a good-quality migration, but they have not been addressed in this thesis. Next we will outline all of these models.
The *Structure model* is the pivot of the CUI model. It describes the logical structure of views, that is, the hierarchy of widgets that compose the views. This hierarchy must be aligned with what the user sees in the screen. Moreover, it must include support for internationalisation (i18n) and has to be backed up by a *Resource model* (omitted in the figures) that contains the paths to the actual resources (images, icons, language files, etc.). The rest of the models reference the widgets defined in this one.
The spatial arrangement of the GUI is represented by the *Layout model*. The layout is made up with a composition of high-level layout components, such as layout managers (e.g., Flow layout or Grid layout). The composition should ensure that the view will be displayed properly under

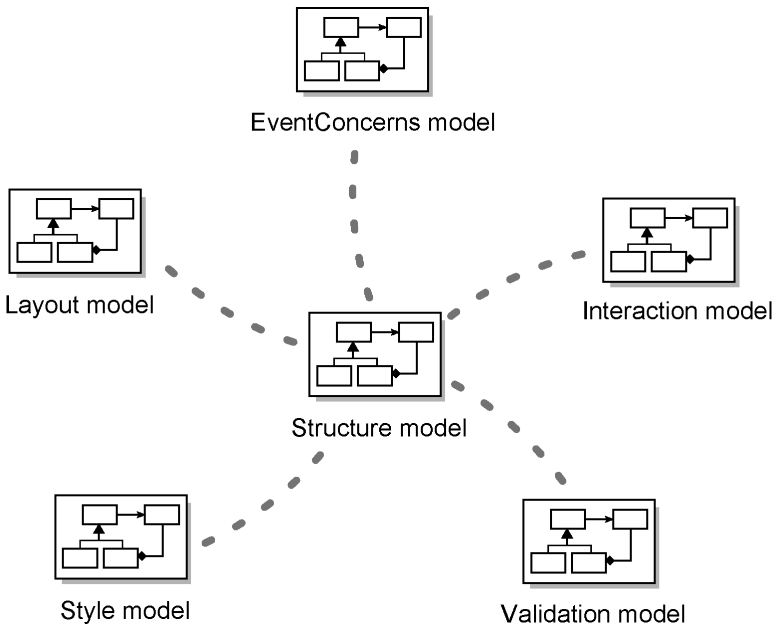


Figure 4.1: Concrete User Interface models in our solution

different screen sizes and resolutions, and when the views are resized.

The *Style model* defines the look and feel of the views, that is, background and foreground colours, font types and sizes, border types and so forth. With this model, styles (groups of visual properties) would be defined, and inherited from other styles in order to promote reuse (somewhat similar to CSS philosophy).

The code of the event handlers is represented in the *EventConcerns model* in a language-independent fashion. This model presents an abstraction of the code where groups of sentences of the original code that match some pattern are replaced by application primitives that express the semantics of the code. Moreover, the code fragments are tagged with the concern (view, controller, business logic) they are related to, and they are also structured in a control-flow graph.

The information of the *Interaction model* is twofold: it specifies the dependencies among views, and also the dependencies among the widgets contained in these views. It represents the navigation flows of the application by means of a Finite State Machine in which the states are the different views of the application and the transitions are the events that let them happen. For each view, the dependencies among widgets are represented through a dependency graph, in which dependencies are expressed with an event-condition-action schema (similar to transitions between views). For example, selecting a specific checkbox triggers an event that enables

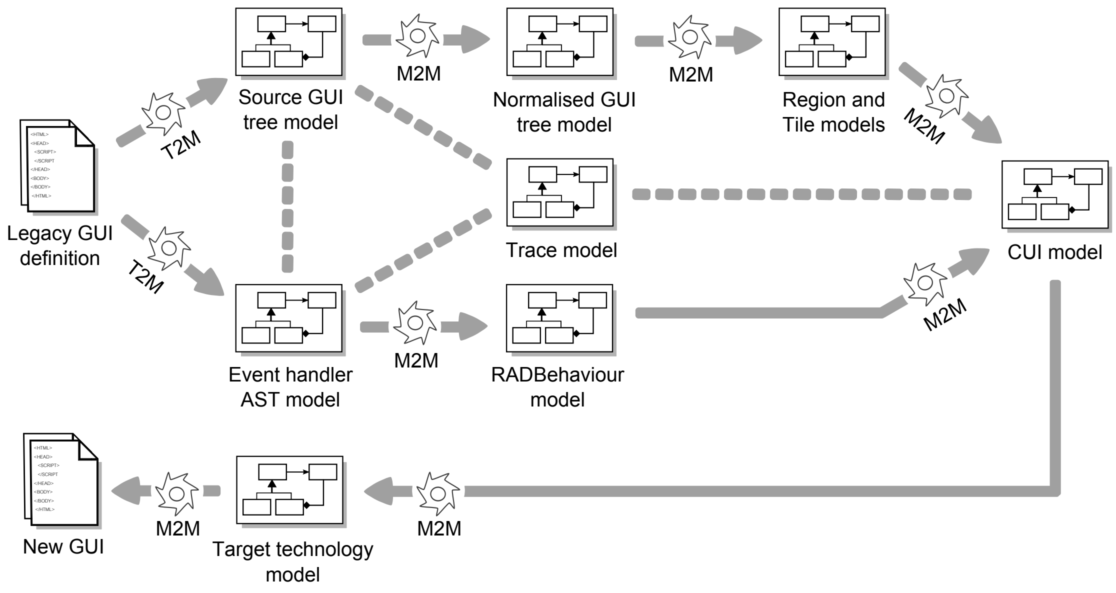


Figure 4.2: Architecture of the solution (GUI2MO framework)

a certain text field.

The validation rules of the data introduced in forms are represented in the *Validation model*. It also considers the notification of the validation errors to the user. For instance, rules such as two password fields must match or that a text field must have a valid e-mail format would be specified.

4.2.2 Overview of the migration architecture

Figure 4.2 presents the general MDE-based architecture of the solution we propose for migrating GUIs of legacy systems. Arrows indicate a model transformation that is applied (the type is also indicated: T2M, M2M, or M2T), and dashed lines represent dependencies between models. The architecture has been simplified, as some specific models and dependencies have been omited, to ease the comprehension of the global idea.

The input of the process is the set of artefacts that describe the GUI in a specific technology. In some RAD environments such as Oracle Forms 6i the GUI definition and the code of the event handlers are included in the same files, whereas in others such as Delphi 5 they are separated in different files. From the source artefacts two types of models are injected: the Source GUI tree and the Event handler AST. The *Source GUI tree model* represents the logical structure of the view, that is, how the widgets are organised in the view and their visual properties, including the position in coordinates. The *Event handler AST model* is the Abstract Syntax Tree (AST) representation of the code of the event handlers, which depends on the programming language. Some legacy tools have some peculiarities when representing the GUI tree that are not found in current GUI frameworks. For example, in Oracle Forms there is not a component for representing a grid of data. Instead, some of the components such as text fields or buttons have an attribute that indicates whether the component will be repeated several times. Another peculiarity is that positions are not expressed in pixels by default, but in proprietary measures, so a conversion to pixels or other standard unit is required. With the aim to avoid these features that are too specific, the Source GUI tree model is transformed into a *Normalised GUI tree model* that makes uniform the input of later reverse engineering algorithms so they can be reused with independence of the legacy technology.

Legacy management applications were frequently used to display data in forms that are backed up by a relational database. In this kind of applications, many actions were recurrent, such as displaying a data table after selecting a row in another data table (master-detail pattern). The *RADBehaviour model* is an abstraction of the source code that captures the behaviour of event handlers in terms of simple primitives which are common in RAD environments, such as read data from a database or write some data in the GUI controls. Even though it is not its primary purpose, this model also serves us to make the rest of the analysis independent of the language in which the event handlers were written.

It is important to note that the T2M to create the Source GUI tree model, the T2M that injects the Event handler AST model, the M2M to derive the Normalised GUI tree model, and the M2M to create the RADBehaviour model, must be implemented for every new source technology, but the rest of the process is reused.

The Normalised GUI tree model is the basis on which we apply some reverse engineering algorithms to obtain the Structure model, the Layout model and the Style model, which are the 'visible' parts of the CUI (what users see in the screen). The transformation of the Normalised GUI tree model into the CUI is not performed in a single step but in several steps which are supported by some intermediate models (*Region and Tile models*) that we will present later. On the other hand, the RADBehaviour model is used to derive the part of the CUI that is related to the interaction (Interaction model) and the behaviour of the interface (EventConcerns model).

It usually happens that when raising the abstraction level, an amount of details that appear in the source artefacts are missed because they are too specific of the source technology. However, this information that is intentionally lost in the abstraction process may be useful when generating the Final User Interface, for example to know what is the source of the generated elements in case there were bugs or unexpected results. In these cases, a traceability mechanism can be used to ensure that all the information is available when generating the Final User Interface. The *Trace model* serve us for this purpose by linking the Structure model with the Source GUI Tree model and the EventConcerns model with the Event Handler AST model.

Once all the models conforming to the CUI and Trace metamodels have been generated, a model transformation can be applied to move the GUI information expressed in the CUI to a particular technology or a different UIDL, and from that model, generate the Final User Interface of the new system. In the following chapters we will focus on the different parts of the architecture presented in 4.2, and we will describe all the metamodels and transformations that are implied in the process.

4.2.3 Requirement implementation

Table 4.3 summarises how the requirements elicited in Section 4.1 are fulfilled in the GUIZMO arquitecture.

R1: Explicit GUI information	Metamodels
R2: Modularity	Model transformation chain
R3: Automation	Model transformation chain
R4: Source independence	Normalised GUI tree model, RADBehaviour model
R4: Target independence	CUI model
R5: Logical/visual structure matching	Structure model
R6: High-level layout representation	Layout model
R7: Misalignment tolerance	Layout inference algorithm
R8: Alternative solutions	Layout inference algorithm
R9: Configurable layout set	Layout inference algorithm
R10: Code abstraction	RADBehaviour model
R11: Code categorisation	EventConcerns model
R12: Explicit interaction and navigation	Interaction model

Table 4.3: Implementation of the requirements

Using MDE endowes the solution with some benefits. The use of metamodels serve us to explicitly represent the metadata that have been gathered from the source system (*R1*). The model transformation chain splits the whole migration process in smaller steps, then promoting mod-

ularity (*R2*). Moreover, the transformation chain is executed automatically, which fulfils requirement *R3*. In the transformation chain we introduced the *Normalised GUI Tree* and *RAD-Behaviour* models that makes the approach independent of the source RAD technology, and the CUI model that makes it independent of the target technology (*R4*).
Matching the logical and visual structures is supported by the Region model and finally achieved in the Structure model (*R5*). The Layout model is used to represent the layout of the GUI by means of high-level constructions (*R6*). Our reverse engineering algorithms are designed to deal with certain misalignment tolerance (*R7*) and output alternative layout compositions (*R8*). Moreover, they are configurable so the layout sets (i.e., the different types of layouts) to use can be specified (*R9*), as well as some other parameters such as the comparison margin used to obtain misalignment tolerance.
The solution raises the abstraction level of the source code by means of the RADBehaviour model (*R10*). Then, that model is used to categorise the code according to the tier to which it belongs (*R11*). Other useful information such as the interaction and navigation flows are extracted from the RADBehaviour model and represented in the Interaction model (*R12*).

Design is not just what it looks like and feels like. Design is how it works.

Steve Jobs
(Suggested by Pablo Gómez)

5 Layout inference: greedy approach

Layout is the sizing, spacing, and placement of content within a window, which is a key aspect in GUI design, as explained in Section 2.2. In this chapter, we explore the concerns involved in discovering layout relationships among user interface elements. We focus on GUIs built with legacy IDEs, particularly RAD environments, in which the layout is implicitly represented by means of the explicit position of widgets. The solution we propose can be reused with any GUI that matches the features of RAD environments related to the implicit layout, and in fact, in the next chapter we will apply the same schema to the wireframing tools.

Figure 5.1 shows the whole architecture of the reengineering approach proposed in this thesis, which was described in Section 4.2.2. In this chapter we will focus on the parts of the reengineering process used to obtain the Structure and Layout models (highlighted in black), which are the main static components of the CUI model, and we will see how to use them to generate a new GUI.

Basically, the layout inference process obtains an explicit Layout model in three stages: region identification, positioning system change, and high-level structure detection. Several meta-

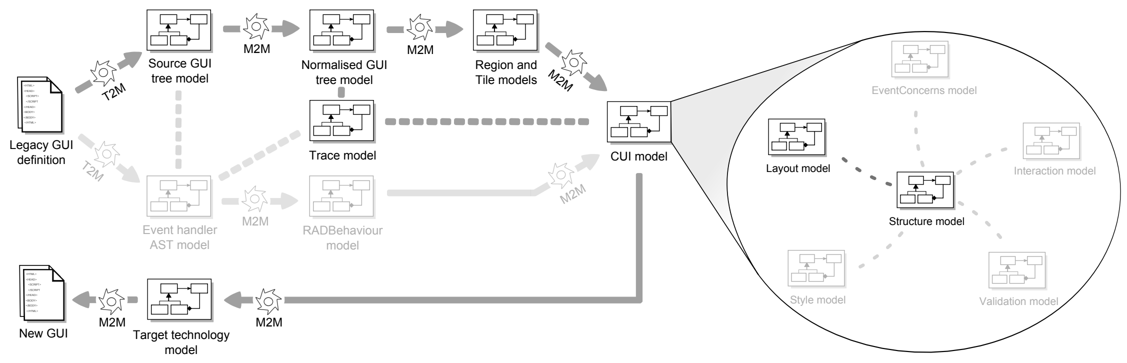


Figure 5.1: Part of the architecture explained in this chapter.

models and algorithms have been defined to implement this process, which will be explained in detail in this chapter. We will put this process in the context of Oracle Forms 6, which is a concrete RAD environment, and we will evaluate the algorithms by reverse engineering two real legacy applications, built by two different companies, and consisting of 57 and 107 windows of different types, respectively.

5.1 MDE architecture for layout inference

Figure 5.2 represents the transformation chain that we have devised to deal with the reengineering of RAD GUIs. Note that this figure depicts the same elements that are highlighted in Figure 5.1 in addition to the Region and Tile models which were represented in one box in the former diagram. The steps involved in the reverse engineering part (layout inference process) have been highlighted. Boxes stand for models and solid arrows depict M2M transformations, and dashed arrows mean model dependencies (i.e., a model refers to elements of another model).

The actual input of the layout inference process is the Normalised GUI Tree model. This model conforms to the so called Normalised GUI Tree metamodel which generalises concepts that are common to GUIs built with legacy tools. It is a kind of normalisation model that is intended to make the rest of the reverse engineering process independent of the source technology. On the basis of that model, we apply an algorithm to identify distinguisable parts in the views of the legacy GUI, thus obtaining the Region model. The Tile model, which is obtained from the analysis of the Normalised model and the Region model, is a representation of the GUI views that

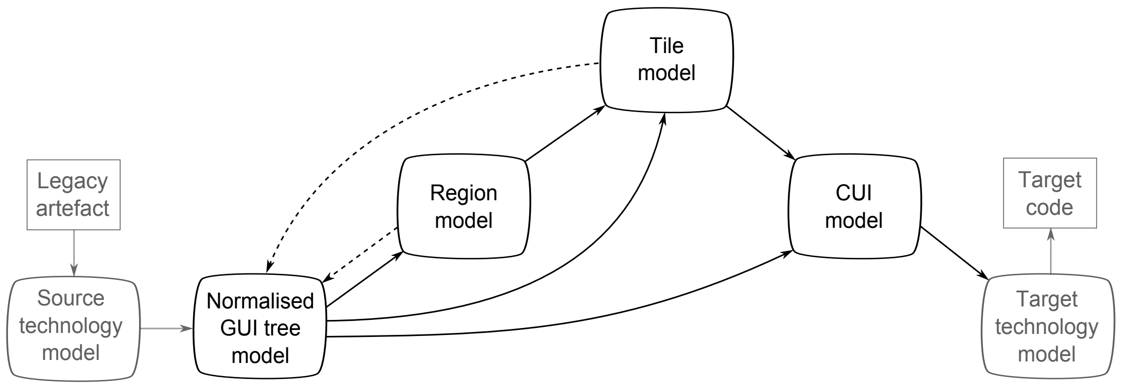


Figure 5.2: Model-based architecture used to migrate legacy GUIs.

includes a positioning system of the widgets based on relative positions among them. The Tile and Normalised models are the input of the transformation that generates the CUI model and the Trace model (omitted in Figure 5.2). The latter simply keeps links between the CUI model and the Source GUI Tree model, and can be obtained by traversing the models by means of the backward references. Although in our case studies we have not made use of the Trace model, it can be useful in the later restructuring or forward engineering phases, for example, if we performed some modifications in the CUI model and the whole transformation chain needed to be re-run without overwriting the changes (GUI evolution) [4]. It is worth remarking that this architecture integrates the Structure and the Layout model in the same software artefact (the CUI model).

5.2 Reverse engineering metamodels

In this section we describe the Normalised GUI Tree metamodel (from now on, Normalised metamodel) and CUI metamodels that were presented in the previous section. Contrary to Source and Target Technology models, Normalised and CUI models are independent of a concrete technology.

Normalised GUI Tree metamodel

The commonalities of the GUIs built with RAD environments are described by means of this metamodel. It is a generic representation which allows the GUI of source RAD applications to be expressed in terms of features which are typically provided by RAD environments, such as widgets positioned with coordinates (which form an implicit layout) and a hierarchy of common widgets.

In essence, the Normalised metamodel represents a GUI definition as follows. There are two types of *Widgets*: *Containers* (e.g., *PlainPanels*) that nest other *Widgets*, and *SingleWidgets*(e.g., *TextBoxes*), that cannot contain other *Widgets*. A *View* represents the area of the screen that displays the part of the GUI that a user sees at a particular moment, such as a desktop application window. Both *Views* and *Containers* can nest *Widgets*, and the complete hierarchical structure of a *View* formed by *Containers* and *SingleWidgets* is called *GUI tree*. From here on, the term *View* will be used to refer to the metaclass and the term *view* will be used as a general concept.

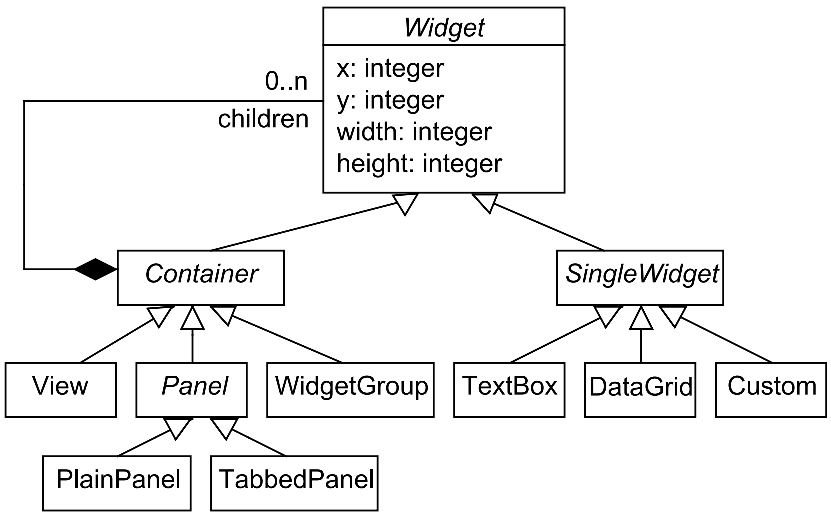


Figure 5.3: Excerpt of the Normalised metamodel.

The design of the Normalised metamodel has been driven by the identification of common features of legacy-tool-based GUIs that were listed in Section 2.2.2. An excerpt of the metamodel is shown in Figure 5.3. These features and their representation in the metamodel are outlined as follows:

- **Implicit layout**. The position of *Widgets* is stated by means of coordinates that are relative to the main window or another container. In the metamodel this is conveyed by the *Widget* metaclass which has *x* and *y* attributes (a coordinate), and an explicit *width* and *height*.

- **Clustering elements**. There are special widgets which are intended to group and/or highlight semantically-related widgets. These widgets are represented as subtypes of *Container* in the metamodel. We distinguish between *Panel*s that are elements that arrange a window in parts (in some legacy environments they can also be reused between windows), and *WidgetGroup*s that are used to highlight a set of widgets in close proximity, frequently by means of a border.

- **Overlapping**. Widgets are often loosely contained in their container, that is, they are overlapped with the container instead of having explicit containment relationships. A container could also be overlapped with another container. This means that a *Container* may not have any widget in the *children* reference, although there may be some widgets that would (visually) be expected to be contained.

- **Standard widgets**. Legacy environments share a common set of standard widgets, such as text boxes, buttons, combo boxes, tables, and so forth. They are represented in the metamodel with metaclasses inheriting from *SingleWidget*.

- **Technology-dependent widgets**. Source technology-dependent widgets (e.g. an ActiveX control) cannot be represented in the Normalised metamodel (which is technology-independent), and cannot therefore be part of the subsequent reverse engineering. We propose two alternatives to deal with this issue: i) the metamodel provides a special widget (*Custom*), that allows the reverse engineering process to deal with them, and developers are in charge of giving them a proper meaning in a later reengineering stage, ii) some specific widgets can be emulated by one or more standard widgets from the metamodel. For instance, an Oracle Forms multirecord is a group of single widgets (e.g. text boxes) arranged in a tabular form, which can be mapped into a *DataGrid*. This mapping is typically carried out in the normalisation stage (i.e., the transformation of a Source Technology model into a Normalised model).

As stated previously, a Normalised model is derived from a Source Technology model by means of a M2M transformation. Given that the Normalised metamodel does not establish tight restrictions regarding the arrangement of widgets, defining this M2M transformation in order to translate Source Technology metamodel concepts into Normalised metamodel concepts is normally straightforward.

CUI metamodel

In our solution, CUI models conform to the metamodel shown in Figure 5.4, in which the layout is explicitly modelled with compositions of high-level concepts which are present in most GUI frameworks, such as flows of elements, grids, and so forth. It is worth noting that the Structure model and the Layout model have been merged in a single CUI model.

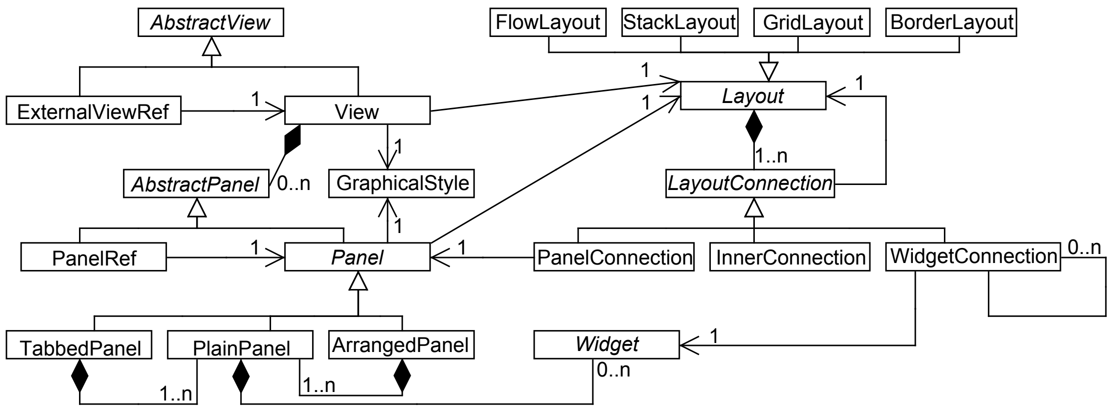


Figure 5.4: Simplified CUI metamodel.

*View*s are composed of *AbstractPanel*s (i.e. *Panel*s and *PanelRef*s), which are reusable parts of the GUI, in such a way that a panel could be used in several views. *Panel*s can contain subpanels or widgets. Views and panels have a graphical style (that defines the font type and background colour, for example) and a layout that describes how the subpanels or widgets are arranged. The layout is expressed in terms of hierarchies of high-level arrangements (e.g. *FlowLayout, StackLayout,* etc.), and has connections (*LayoutConnection*) that indicate which subpanels (*PanelConnection*) or widgets (*WidgetConnection*) are arranged according to it. *InnerConnection*s do not refer to any panel or widget and are used to create a layout tree structure. A *WidgetConnection* can be related to other *WidgetConnection*s, which is used to express dependencies between widgets (e.g. associate a text field and a label).

It is worth noting that the metamodel supports the separation between three concepts: the panel as a reusable part of a view, its graphical style and the layout of the subpanels or widgets that it contains. This metamodel also covers some other aspects of a GUI, such as support for internationalisation.

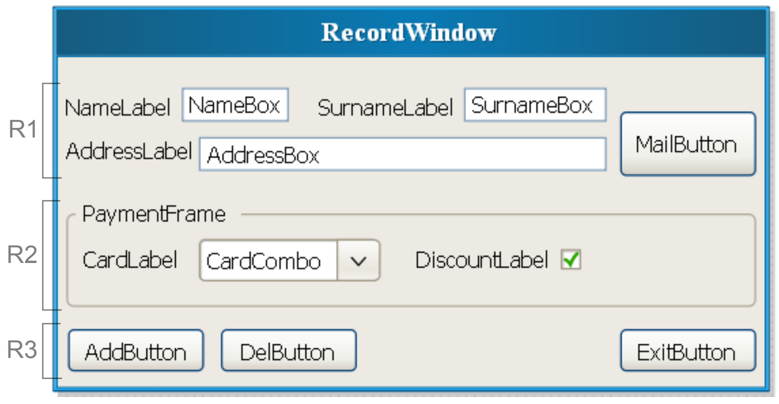


Figure 5.5: Example view for entering personal information. (Same window as Figure 2.2).

5.3 Challenges in layout reverse engineering

As we indicated in Section 2.2.2, in RAD applications the layout is implicitly defined by the position of the elements, which are expressed in terms of coordinates. Our aim is to capture the visual arrangement of the elements in such a way that both replicating the layout and redesigning it for a different technology is easy. Transforming an implicit, coordinate-based layout (represented by the metamodel in Figure 5.3) into an explicit, high-level layout (represented by the metamodel in Figure 5.4) poses the following challenges.

(L1) **Region identification**. A view can be seen as a composition of parts or regions (perhaps implicit) which provides the widgets of the view with a structure. Reverse engineering the structure of a view by identifying regions is necessary for layout redesign. In the example of Figure 5.5 we can make out three regions in the window. Region *R2* contains the widgets that are surrounded by the *PaymentFrame* frame, region *R1* is composed of the widgets above the frame, and region *R3* includes the widgets below the frame (note that *R1* and *R3* are implicit).

(L2) **Explicit containment**. As explained in Section 2.2.2, in some cases elements are not actually contained in a container, but are overlapped. Matching the containment hierarchy and the visual structure of the layout greatly simplifies the reverse engineering and restructuring algorithms, and it is thus necessary to establish explicit containment relationships.

(L3) **Widget structure recognition**. While region identification aims to recognize those parts of which the view is structured, widget structure recognition is focused on how widgets that are spatially-close to each other are arranged. For example, the widgets inside the *PaymentFrame* form a line. Widgets are often not perfectly aligned, so heuristics are needed. To continue with the example, *NameLabel, NameBox, SurnameLabel* and *SurnameBox* could form a line, but it is not clear whether *MailButton* would be considered as a component of this line.

(L4) **Coordinate abstraction**. As already mentioned, a coordinate-based positioning system is not desirable, and thus an alternative means to represent relationships between elements is needed. For example, it would be desirable to know that *NameLabel* is above *AddressLabel* and on the left of *NameBox*.

(L5) **Alignment and spacing detection**. The widget structure is tunned by means of the alignment and spacing (gaps and margins) assigned to the widgets. With the term *hole* we will refer to an area of a remarkable size that does not contain widgets but is surrounded by them, i.e., a gap of a considerable size. In the example view, there is a hole between *DelButton* and *ExitButton*. It is necessary to capture the alignment, gaps and margins if a similar layout is to be reproduced in a different technology.

Challenges *L1* and *L2* are related to the fulfilment of the requirement *R1* (matching the visual and logical structure), and challenges *L3*, *L4* and *L5* are related to the requirement *R2* (high-level layout representation). The following sections show the algorithms that deal with these issues.

5.4 Detecting regions and containers

This stage is intended to tackle issues L1 and L2 commented on above (namely, region identification and explicit containment). Here, a Region model is automatically derived from a Normalised model.

A *Region model* is a model that annotates a Normalised model in order to make visual containment relationships between widgets explicit. A Region model represents a tree of regions that conforms to the metamodel shown in Figure 5.6. It has a unique metaclass called *Region*, which has the two pairs of coordinates that define a rectangular area, and the *children* reference to the

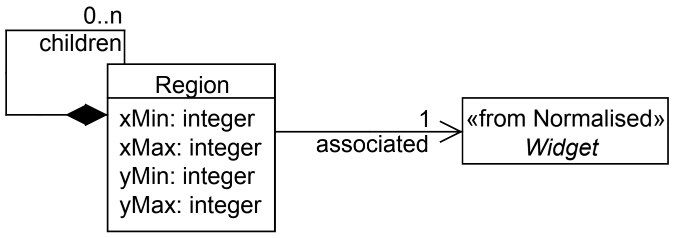


Figure 5.6: Region metamodel.

sub-regions contained in it. Note that *Region* elements are annotations for the *Widget*s of a Normalised model. Region models have three main features: i) each *Widget* is associated with a *Region* defined by two pairs of coordinates, ii) *Container*s and *SingleWidget*s must not exist at the same level (i.e. a region that annotates a *Container* cannot be a sibling of a region that annotates a *SingleWidget*), and iii) overlapped regions are not permitted.

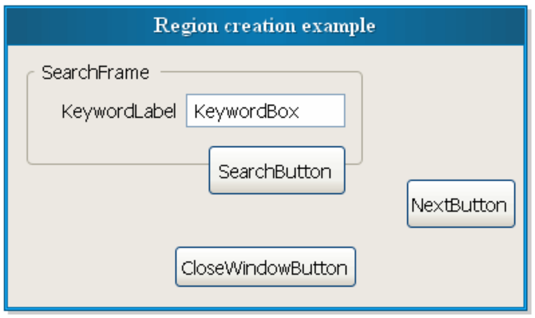
SingleWidgets are prevented from being at the same level as *Containers* as a means to structure the GUI in a uniform manner, so that views are divided into parts which are disjointed and complementary. Each view therefore contains several separate regions (which can in turn contain more regions or widgets), and each widget belongs to a unique region. The goal of this design decision is twofold. On the one hand, we believe that conceptually a UI is composed of related parts like a puzzle in such a way that there are no widgets outside of a part. On the other hand, it makes the structure of the UI uniform and simplifies the later algorithms.

A precondition of the algorithm used to create the regions is that the border of a *Container* must never cross the border of another *Container*. Our framework has a previous phase that checks whether frame border overlapping occurs. If this occurs, then the reverse engineering process is stopped and a message is shown to the developper so he can fix the GUI manually (although, in our experience this situation rarely arises).

In the algorithm we distinguish between three types of regions: *widget regions, base regions* and *extra regions*. A *widget region* is a region associated with a widget. The term *base region* is used to refer to a region that is associated with a container. *Extra regions* are artificial regions which are created to contain widgets that are not included in a *base region*. Note that *base regions* and *extra regions* will contain subregions, unlike *widget regions*. We will explain the region detection algorithm with the ad-hoc example window in Figure 5.7. The algorithm used to create the

Algorithm 1 Region creation algorithm.

```
 1: for all $view$ do
 2:     $r_0 \leftarrow createRegion(view)$
 3:     for all $w \in getWidgets(view)$ do                ▷ Gets contained widgets
 4:         $r_1 \leftarrow createRegion(w)$
 5:         $addChild(r_0, r_1)$
 6:     end for
 7:
 8:     for all $r_1, r_2 \in children(r_0)$ do
 9:         if $r_1 \neq r_2 \wedge contains(r_1, r_2)$ then
10:             if $\nexists r_3 \neq r_2 \neq r_0.(contains(r_3, r_2)) \vee$
                    $\forall r_3 \neq r_2 \neq r_0.(contains(r_3, r_2) \rightarrow contains(r_3, r_1))$ then
11:                 $addChild(r_1, r_2)$
12:             end if
13:         end if
14:     end for
15:
16:     $createExtraRegions(r_0)$
17: end for
```



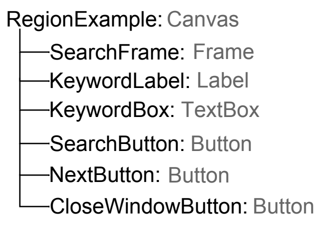


Figure 5.7: Left: example window for the region detection. Right: the logical structure of the widgets.

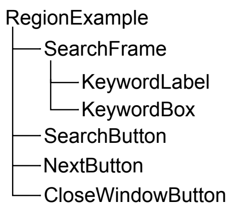


Figure 5.8: Structure of the regions after step 2 for the example in Figure 5.7.

Region model (Algorithm 1) is summarised in the following steps:

1. Create a region for every *Widget* (lines 2 to 6). r_0 is a base region associated with the window, which is the root of the region tree. r_1 is a (widget or base) region associated with w, which can be a single widget or a container. $add(r_0, r_1)$ means that r_1 is set as a child of r_0. The area of a new region is derived from the (x, y) coordinates, the width and the height of the *Widget*. For example, in Figure 5.7, a base region is created for each one of the containers (the *RegionExample* window and *SearchFrame*), and a widget region is created for each single widget (*KeywordLabel, KeywordBox, SearchButton, NextButton, CloseWindowButton*).

2. Create a tree structure by nesting the regions according to the visual containment relationships (lines 8 to 14). The expression $contains(r_1, r_2)$ is true if the coordinates of r_2 are inside the rectangle defined by the coordinates of r_1. For each pair of regions, r_1 and r_2, we make r_2 a child of r_1 if r_1 contains r_2 and one of the following conditions is true: i) there is not a different region r_3 containing r_2 (r_2 is a direct child of r_1), ii) there is another region r_3 containing r_2 but it also contains r_1 (r_2 is a direct child of r_1 which in turn is a direct child of r_3). The evolution of the example window after this step can be seen in Figure 5.8: *SearchFrame* now contains *KeywordLabel* and *KeywordBox*. At the end of this step, there can be widget regions which are siblings of base regions in the Region model. Following with the example we can see that *SearchButton, NextButton CloseWindowButton* are siblings of *SearchFrame*.

3. Create extra regions to prevent *SingleWidget*s from being at the same level (siblings) as the *Container*s. The algorithm iterates once over every widget region that is a sibling of either a base region or an extra region (at the beginning there are only base regions). For each widget region we have three possible cases:

    - *Case A*: the widget is not partly contained in any existing base or extra region (i.e. the widget does not cross the bounds of any base or extra region), so a new extra region is therefore created for the widget region. The new region takes the maximum area available without interfering with the other regions. Following with the example, we assume that we have already dealt with *KeywordLabel* and *KeywordBox*, and now is the turn of *CloseWindowButton*. As this widget is not contained in the unique base region *R1* (see left part of Figure 5.9), a new extra region *R2* is created for it (right part of Figure 5.9).

- *Case B*: the widget region is partly contained in a base region. In this case the size of that base region is increased to enable it to cover the area occupied by the widget region, and the widget is added to it. Augmenting the size of the base region may cause that the base region overlaps some extra regions, and the overlapped extra regions are therefore shrunk to avoid the overlapping. Continuing with the example, let us make the algorithm iterate over *SearchButton* which is partly contained in the region *R1* associated with *SearchFrame* (left part of Figure 5.10), so we augment the base region to fully contain *SearchButton* (right part of Figure 5.10). This implies that the region *R2* is shrunk. If a widget is partly contained in more than one sibling base region, then the widget is included in only one base region, and in this case the widget is shrunk to fit into that base region. We have not found this case yet in practice.
- *Case C*: the widget is partly contained in an extra region. It is necessary to reduce the extra region that partly contains the widget so that the widget no longer enters its area anymore. In addition, a new extra region to contain the widget is created. Going back to the example (see Figure 5.11), the algorithm iterates over *NextButton*. As the widget crosses the bounds of the extra region *R2*, this region is resized to exclude the widget. Hence, a new extra region *R3* is created to contain the new widget without interfering with any of the already created regions.

The cases are evaluated in the following order: case B, case C, case A. Note that in the example we have iterated over the widgets in a way that it facilitates the explanation of the cases, though other orders are also possible. The different orders will end up in regions that may differ in their coordinates but that group the widgets in the same way.

5.5 Uncovering relative positions

The objective of this second stage is to make the layout independent of the coordinate-based system. This deals with issues L3, L4 and L5 mentioned previously (namely, widget structure

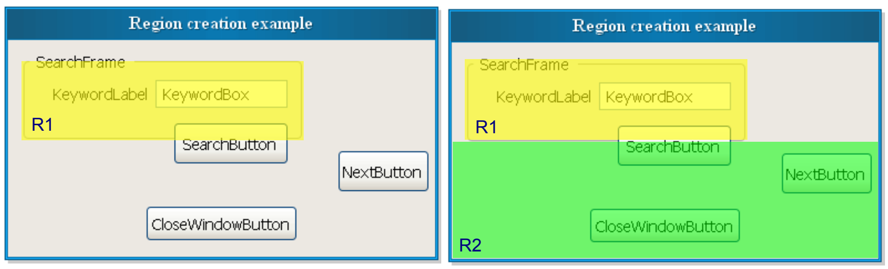


Figure 5.9: Case A. Left: example window with a base region *R1*. Right: a new extra region *R2* created to contain *CloseWindowButton*.

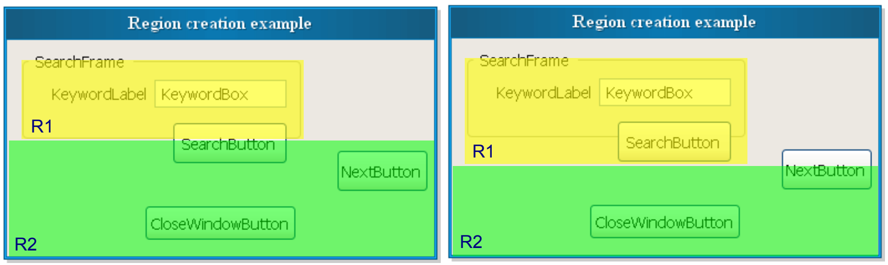


Figure 5.10: Case B. Left: example window with a base region *R1* and an extra region *R2*. Right: the base region *R1* is augmented to include *SearchButton* completely and the extra region *R2* is diminished.

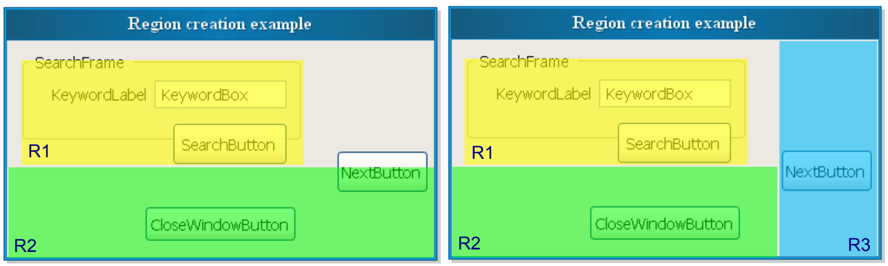


Figure 5.11: Case C. Left: example window with a base region *R1* and an extra region *R2*. Right: a new extra region *R3* is created to contain *NextButton*, and the region *R2* is diminished.

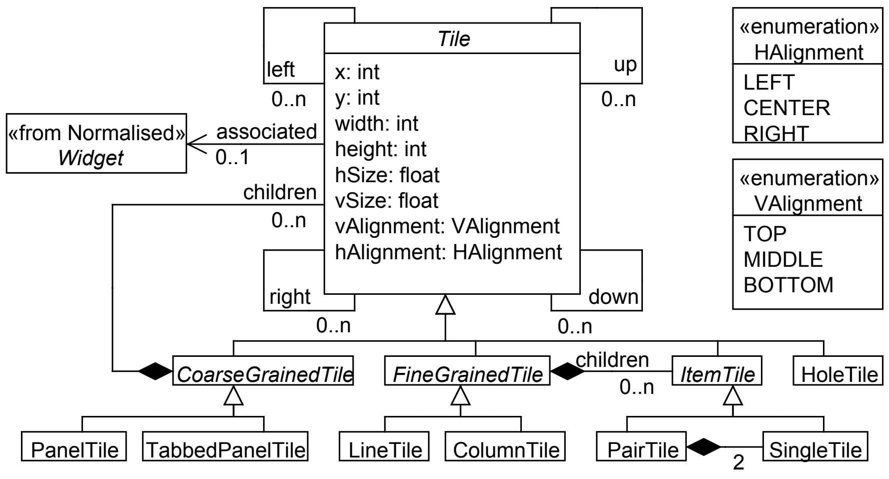


Figure 5.12: Tile metamodel.

recognition, coordinate abstraction, and alignment and spacing detection). The input of this stage is a Region model, and a Tile model is automatically generated.

Tile models are mainly focused on representing how widgets and containers are arranged, in terms of relative positions among them. We define a *tile* as a part of a view with spatial relationships with other neighbouring parts. For example, a certain tile could have another tile above it and a different tile below. This positioning system is useful for the later identification of high-level layout patterns, as will be shown in Section 5.6. Tile models also refine Region models by identifying sub-structures inside regions, such as groups of widgets that form a line.

The Tile metamodel is shown in Figure 5.12. The main concept is that of *Tile*. Every *Tile* is associated with the *Widget* from which it originated, if one exists (i.e. some tiles originated from extra regions). Such references to the Normalised model are propagated from the Region model. There are four zero-to-many relationships between tiles, which are used to relate the tiles spatially, namely *right, left, up, down*. We use *hSize* and *vSize* to measure the percentage of the width and height that is taken up by that tile in the view with regard to the width and height of the container tile. Tiles also include information about the area they take up by means of *x, y, width, height*. A tile can also be aligned with regard to its container tile, and *hAlignment* and *vAlignment* are used for this purpose. We distinguish four kinds of tiles:

- **Coarse-grained tiles**: these tiles arrange a view in parts which can be visually distinguished. Each tile represents a block of related widgets which are in the same area and are likely to contain widgets to perform system actions (e.g., the bottom buttons in Figure 5.5), or data concerning a topic such as "payment details". All base and extra regions are mapped to this kind of tile. For instance, in Figure 5.5 the regions *R1*, *R2*, and *R3* are mapped to *PanelTiles*.

- **Fine-grained tiles**: these tiles arrange a set of widgets that are spatially close and have a certain spatial structure, such as a horizontal line (*LineTile*) or a vertical column (*ColumnTile*). Fine-grained tiles are aggregated inside coarse-grained tiles. To continue with the example, *NameLabel, NameBox, SurnameLabel* and *SurnameBox* are all mapped together to a *LineTile*.
- **Item tiles**: they are associated with single widgets (*SingleTile*) and pairs (*PairTile*) of related widgets such as a text box (e.g. *NameBox*) and its associated label (e.g. *NameLabel*). *Item tiles* are contained in *Fine-grained tiles*.
- **Hole tiles**: these tiles represent a portion of the view of notable size which has no widgets, such as the space between *DelButton* and *ExitButton* in Figure 5.5.

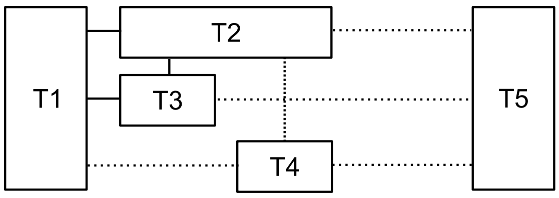


Figure 5.13: Adjacency example

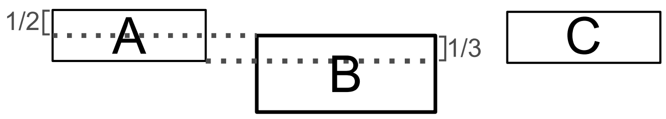


Figure 5.14: Horizontal intersection value example

Next, we establish some of the concepts which allow spatial relationships between tiles to be detected. Figure 5.13 is used to illustrate these concepts. All the following concepts are defined over tiles, but since we have the (X,Y) coordinates, the width and height of both regions and tiles, the concepts are applicable to regions too.

We will define *adjacency* as a criterion with which to decide whether two tiles of the same kind are spatially related (for example, that a coarse-grained tile T1 is on the left of another coarse-grained tile T2). Our definition of adjacency is based on the concept of *sharing*. A pair of tiles is *vertically sharing* if the intersection of the projections of both tiles on the X axis is not empty, i.e. the x-range of both tiles is overlapped. Likewise, a pair of tiles is *horizontally sharing* if the intersection of the projections of both tiles on the Y axis is not empty, i.e. the y-range of both tiles is overlapped. As is observed in Figure 5.13, *T2* and *T3* are vertically sharing, and T2 and T4 are also vertically sharing, but T3 and T4 are not.

The introduced definitions of sharing are too strict because they consider that overlapped projections always reflect horizontal lines or vertical columns. For instance, in Figure 5.14 *A*, *B* and *C* may (or may not) form a line, because they are not perfectly aligned. This can be addressed by modifying the sharing definition to be more tolerant, and we introduce the *intersection value* with this aim. We define the *vertical intersection value* as the percentage of width that a pair of tiles have in common. It is calculated as the intersection of the x-ranges of the pair of tiles divided by the minimum width of both tiles. Similarly we define the *horizontal intersection value* between a pair of tiles as the percentage of height that a pair of tiles have in common, which is calculated as the intersection of the y-ranges of the pair of tiles divided by the minimum height of both tiles. Figure 5.14 shows how this function is applied. The percentage of the height that tile *A* has in common with tile *B* regarding tile *A* is 0.5, while the value is 0.33 as regards tile *B*. The result is therefore the maximum value, that is 0.5. Note that a pair of tiles that are horizontally sharing always have a positive horizontal intersection value (similarly with the vertically sharing). The sharing can be redefined (for horizontal sharing as well as vertical sharing) as follows: a pair of tiles are sharing if the intersection value is greater than a threshold which represents the tolerance level, currently set to 0.5.

Based on the concept of sharing, we can now define *adjacency*. A tile t_1 is *vertically adjacent* to another tile t_2 if and only if both tiles are vertically sharing and there is no tile t_3 between t_1 and t_2. Likewise, a tile t_1 is *horizontally adjacent* to another tile t_2 if and only if both tiles are horizontally sharing and there is no tile t_3 between t_1 and t_2. There is a precondition that the tiles t_1 and t_2 must not be overlapped (in our case this is enforced by the Normalised model). To continue with the example in Figure 5.13, we can see that *T2* and *T3* are vertically adjacent, and *T1* and *T4* are horizontally adjacent, among others.

The *up, down, left, right* relationships of the tiles are defined based on the adjacency as follows. For a tile t_1 it is true that $t_1.right = \{t_2\}$ and $t_2.left = \{t_1\}$ if t_1 and t_2 are horizontally adjacent and t_2 is to the right of t_1. The *down, left, right* relationships are defined in a similar way. Note that when one type of relationship is established for a tile, the opposite type is also set. In the example shown in Figure 5.13, we have the following relationships for T1, T2 and T4:

$$T1.right = \{T2, T3, T4\}$$
$$T2.left = \{T1\}; T2.right = \{T5\}; T2.down = \{T3, T4\}$$
$$T4.up = \{T2\}; T4.right = \{T5\}; T4.left = \{T1\}$$

As can be seen, $T2.down = \{T3, T4\}$. However, there is a blank space between T2 and T4 that is not captured with the concept of adjacency. Thus, there is some layout information that is lost due to blank spaces being ignored.

In order to tackle this issue, the first step is to set a criterion with which to decide whether two vertically/horizontally adjacent tiles are not sufficiently close, but there is a significant blank space between them. We define that a pair of widgets is *horizontally close* if the percentage of the horizontal distance between the pair, with regard to the container width is smaller than a particular value. A pair of widgets is *vertically close* if the percentage of the vertical distance between the pair, with regard to the container height is smaller than a particular value. It is currently set at 20%. In the example shown in Figure 5.13, when using this criterion we have that *T2* and *T3* are adjacent and close, whereas *T2* and *T4* are adjacent but not close.

When a blank space is detected, two complementary approaches are used to represent it. The first one is to specify that some tiles are aligned with regard to the container tile. To continue with the example, *T1*, *T2* and *T3* are aligned on the left, *T5* is aligned on the right, and *T4* is aligned in the bottom-center. There can be several adjacent tiles with the same alignment, which does not mean that all these tiles have to be attached to the bounds of the container. For instance, *T1* and *T2* are both aligned to the left but actually *T2* is on the right of *T1*. The alignment solution has the disadvantage that there may be blank spaces that are not represented.

The second approach is to include *HoleTile*s which represent blank spaces in the layout, thus signifying that an arbitrary distance between tiles must be maintained. These kind of tiles have dimensions that are specified as a proportion between the empty space and the width or height of the container. Since they are not exclusive solutions, both have been implemented in order to facilitate the obtaining of an accurate high-level layout in the CUI model. The *hAlignment* and

vAlignment attributes were introduced for the first alternative and the *HoleTile* metaclass for the second one.
Next, the algorithm that takes a Region model and generates a Tile model is presented. Some auxiliary functions are not explained, but their names denote what they do. The algorithm is organised in four phases: i) creating the tiles, ii) establishing *up, down, left, right* relationships between tiles, iii) setting the spatial alignment of the tiles with regard to the container tiles, and iv) creating hole tiles to represent blank spaces. Each phase will be explained separately.

PHASE 1. The first phase (see Algorithm 2) generates tiles based on regions. The algorithm traverses the Region model recursively from the root region. It has two parameters: i) the container region (base or extra region) to be traversed, and ii) the parent tile which will contain the created tiles. For each container region (the parameter) to which the procedure *CreateTiles* is applied, it creates a coarse-grained tile (line 5), and for each widget region that is a child of the parameter region, it creates an item tile (lines 10 to 13).
Fine-grained tiles are generated for the content of container regions which include widget regions (lines 7 to 15). This is done by using a clustering algorithm that is applied to the container region in order to identify structures of widget regions (line 24). The clustering algorithm makes a first attempt to group widgets in horizontal lines or columns (horizontal lines have priority over columns) based on the vertical/horizontal sharing. As it has already been said, a pair of regions are sharing if their intersection value is higher than a threshold (set by default at 0.5), and will therefore be classified in the same group. In cases it happens that some widget regions have such a big height that they are horizontally close to widget regions in more than one line, that is, they can belong to different lines of widgets (e.g. tile *T1* in Figure 5.13). Similarly, some widget regions may be so wide that they are vertically close to widget regions in more than one column. In order to avoid this, we create new groups for those regions that are classified in more than one group (lines 26–31). Finally, we check that adjacent regions inside the groups are vertically/horizontally close, and if this is not the case, then the group is split (lines 32–36).

PHASE 2. The second phase of the tile creation algorithm establishes the *up, down, left, right* relationships between adjacent tiles. For each ordered pair of tiles (t_1, t_2) which are children of the same coarse or fine-grained tile, $t_1.up \leftarrow t_2$ and $t_2.down \leftarrow t_1$ if: i) they are vertically

Algorithm 2 Tile creation algorithm. Phase 1: Mapping and clustering.

```
 1: $root \leftarrow getRootRegion()$
 2: $createTiles(root, \emptyset)$
 3:
 4: procedure CREATETILES($region, parentTile$)
 5:     $coarseTile \leftarrow createCoarseGrainedTile(region)$
 6:     if $containsWidgetRegions(region)$ then                    ▷ All children are widget regions
 7:         $groups \leftarrow clusterWidgets(region)$
 8:         for all $group \in groups$ do
 9:             $fineTile \leftarrow createFineGrainedTile(group)$
10:             for all $itemRegion \in group$ do
11:                 $itemTile \leftarrow createItemTile(itemRegion)$
12:                 $add(fineTile, itemTile)$
13:             end for
14:             $add(coarseTile, fineTile)$
15:         end for
16:     else                                                    ▷ All children are container regions
17:         for all $childRegion \in children(region)$ do
18:             $createTiles(childRegion, coarseTile)$
19:         end for
20:     end if
21:     $addChild(parentTile, coarseTile)$
22: end procedure
23:
24: function CLUSTERWIDGETS($region$)                                 ▷ Clustering algorithm
25:     $G \leftarrow detectGroups(children(region))$                ▷ Uses horizontal/vertical sharing
26:     for all $G_1, G_2 \in G.(G_1 \cap G_2 \neq \emptyset)$ do
27:         $G_{new} \leftarrow G_1 \cap G_2$
28:         $remove(G_1, G_{new})$
29:         $remove(G_2, G_{new})$
30:         $add(G, G_{new})$
31:     end for
32:     for all $G_1 \in G$ do
33:         if $\exists r_1, r_2 \in G_1.(areAdjacent(r_1, r_2) \wedge notClose(r_1, r_2))$ then
                                                                ▷ Uses horizontally/vertically close
34:             $splitGroup(G_1)$
35:         end if
36:     end for
37:     return $G$
38: end function
```

adjacent, ii) they are vertically close and iii) t_2 is above t_1. The *left, right* sets are obtained in the same manner.

Algorithm 3 Tile creation algorithm. Phase 3: Alignment.

```
 1: for all $t_0 \in \mathcal{T}_{coarse} \cup \mathcal{T}_{fine}$ do          ▷ $t_0$ is a coarse-grained or fine-grained tile
 2:     $HAlignedSeq = \{\}$
 3:     $OrderedTiles \leftarrow topologicalSort(children(t_0))$
                                        ▷ Topological sort from up to down and left to right
 4:     for all $t_1 \in OrderedTiles$ do
 5:         /* For simplicity we are only considering the horizontal alignment */
 6:         $add(HAlignedSeq, t_1)$
 7:         if $t_1.right = \{\}$ then
 8:             $xMinPercent \leftarrow first(HAlignedSeq).x / t_0.width$
 9:             $xMaxPercent \leftarrow$
                    $(last(HAlignedSeq).x + last(HAlignedSeq).width) / t_0.width$
10:             if $xMinPercent \leq Lower\_threshold$ then
11:                 for all $t_2 \in HAlignedSeq$ do $t_2.hAlignment \leftarrow LEFT$
12:             else if $xMaxPercent \geq Upper\_Threshold$ then
13:                 for all $t_2 \in HAlignedSeq$ do $t_2.hAlignment \leftarrow RIGHT$
14:             else
15:                 for all $t_2 \in HAlignedSeq$ do $t_2.hAlignment \leftarrow CENTER$
16:             end if
17:             $HAlignedSeq = \{\}$
18:         end if
19:     end for
20: end for
```

Phase 3. The third phase (see Algorithm 3) is in charge of aligning tiles with regard to their container tile. The idea behind this algorithm is based on the following two principles: i) if a tile is very close to the boundaries of its container tile, then the tile is aligned with regard to them, and ii) if several tiles are next to each other, then all of them have the same alignment. For instance, let us assume that in Figure 5.13 the tiles *T1*, *T2*, *T4* and *T5* are very close to the boundaries of the container tile. Therefore *T1* is aligned to the left because it is close to the left boundary, and *T2* and *T3* are aligned to the left because they are on the right of *T1* which is aligned to the left.

In the algorithm the tiles are iterated in a topological order (lines 4–19), which is computed from the directed graph that results from taking into account only the *right* and *down* relations of the tiles. We add each tile to the current alignment group (line 6) and when there are no more adjacent close tiles on the right (line 7), then we assign an alignment type to each one of the tiles in the group (lines 8 to 18). If the most-left tile of the group (the first tile) is close to the left boundary, the alignment is *LEFT* (line 11). If the most-right tile of the group (the last tile) is close to the right boundary, the alignment is *RIGHT* (line 13). If none of the previous cases is applicable, then the alignment is set to *CENTER*.

Phase 4. The last phase of the algorithm identifies significant blank spaces in the view, and creates hole tiles for them. For each pair of tiles that are children of a coarse or fine-grained tile, if the tiles are adjacent and are not close, then we create a hole tile. This new hole tile is placed between t_1, t_2 and the *up, down, left, right* relationships of both tiles are modified. These properties are also initialised for the hole tile according to its relative positioning regarding the t_1 and t_2 tiles. Finally the new hole tile is added to the parent tile.

5.6 High-level layout

At this stage, information about the relationships among elements of the GUI has been gathered. However, it is convenient to take a further step forward in the way in which the layout is represented in the Tile model to make it more similar to the layout managers provided by modern GUI frameworks. To this end, the CUI metamodel introduced in section 5.2 defines explicit high-level layouts such as grids (*GridLayout*) or stacks of elements (*StackLayout*). For example, if we had a sequence of tiles sorted vertically (each tile below another one), we would explicitly "mark" those tiles as forming a stack layout. The layout types which we use are included in common GUI frameworks such as Java Swing, as well as in diagram editors and other domains [100, 101].
CUI models are generated from Normalised models by using the information provided by the Tile model, in the form of annotations. The algorithm that creates CUI models from Tile models is split into three phases:

1. *Create the structure tree.* The widgets in the Normalised model are mapped to CUI widgets, and the tree structure of the widgets of the CUI model is created according to the containment relationships detected in the Region identification stage. With this aim, the

tile model is traversed in a recursive manner, and the following actions are performed according to the tile type: if the tile is a coarse-grained tile, it creates a *Panel*, adds it to its container *View* or *Panel*, and continues with the tile children; if the tile is a fine-grained tile, it simply navigates its children; if it is a single tile, it creates a widget for it and adds it to the container *Panel*.

2. *Create the layout tree*. In order to get the high-level layout tree, the Tile model is traversed recursively. For each coarse-grained tile we apply several fitness functions on its children and the layout type whose fitness function returns the greatest value is selected. The fitness functions return a number between 0 and 1 that represents the estimated percentage of tiles that fit the layout out of the tiles in the group. A new layout of the selected type is created by applying a heuristic associated with the layout type. In the case of fine-grained tiles, *LineTiles* are directly mapped to *FlowLayouts*, and *ColumnTiles* are directly mapped to *StackLayouts*.

3. *Link both trees*. It links the GUI and layout trees, by selecting the layout for each container and the container of each child connection of each layout.

The tree structure of the layout tree in step 2 is achieved by means of the *LayoutConnection*s. Each new layout that is created is nested in the parent *LayoutConnection*. Depending on the type of children tiles, different *LayoutConnections* will be created: *PanelConnection* if the child is a coarse-grained tile (it is associated with a *Panel* in the step 3), *InnerConnection* if the child is a fine-grained tile, and *WidgetConnection* if the child is a item tile (it will be associated with a *Widget* in the step 3). Then, the same process is applied for each children coarse or fine-grained tile with the *LayoutConnection* as a parameter.
As can be noticed from step 2, we have a set of layout types and each of them has an associated heuristic and a fitness function. The heuristics select a starting tile and navigates its *left, right, up, down* references in an attempt to discover whether related tiles form a high-level layout. In general, several alternative layouts can be found to obtain a similar GUI from a visualisation point of view. In order to decide which layout best fits a group of tiles, the fitness functions are calculated for the group, and the layout heuristic whose fitness function is maximum is applied. It may happen that two or more functions return the highest values. In this case, the best layout is selected according to the following priority criterion: *FlowLayout, StackLayout, GridLayout, BorderLayout, VHLayout, HVLayout.* Next we will detail each type of layout, as well as the heuristics and fitness functions.

FlowLayout and StackLayout

A *FlowLayout* is a set of tiles arranged in a row (horizontal line). Similarly, a *StackLayout* is a set of tiles arranged in a column (vertical line). The tiles are contiguous, i.e. there cannot be a big separation between a pair of tiles. If the layout defines some kind of alignment (*horizontalAlignment* and *verticalAlignment*), all the widgets to which the layout is applied are aligned in that way.

The heuristic for the *FlowLayout* takes the top-left tile and navigates the tiles to the right until there are no more tiles. When there are several tiles to the right of a tile, only the uppermost tile is selected. For the *StackLayout*, the heuristic starts with the top-left tile and navigates to the bottom until there are no more tiles. When there are several tiles below a tile, only the leftmost is selected. As it has already been said, these heuristics are only applied to the content of coarse-grained tiles, since fine-grained tiles are directly mapped.

The fitness function for the *FlowLayout* obtains the percentage of tiles that can be navigated from left to right (starting with the most top-left tile). The fitness function for the *StackLayout* obtains the percentage of tiles that can be navigated from top to bottom (starting with the most top-left tile). In these functions *HoleTiles* are considered to be tiles that have not been navigated and then they reduce the fitness value.

Let us focus on the Figure 5.5 to show some layout examples. We can find a *FlowLayout* in the region *R2* composed of *CardLabel*, *CardCombo*, *DiscountLabel* and *DiscountCheck*. A *StackLayout* is formed by the three regions *R1*, *R2*, *R3*. In the region *R3* we could see a non-perfect match of the *FlowLayout* heuristic. Assuming that *AddButton* and *DelButton* form a fine-grained tile and *ExitButton* forms another fine-grained tile, the fitness function would return 0.66. This value is caused by the hole that exists between both tiles (2 fine-grained tiles / 2 fine-grained tiles + 1 hole).

GridLayout

This is a set of tiles arranged in a grid of *n* rows $\times$ *m* columns. The number of rows and columns may be different, but all the rows (and columns) must have the same number of tiles.

In this case the heuristic starts with the top-left tile and navigates the group of tiles from left to right and from top to bottom in a tabular way.

The fitness function returns the percentage of tiles that can be matched by a rectangular grid. It starts with the top-left tile and counts the number of tiles of the biggest grid possible. *HoleTiles*

are not counted (they reduce the fitness value).
When some tiles fit a *FlowLayout* or *StackLayout*, then they also fit a *GridLayout*. For this reason, *FlowLayout* and *StackLayout* have a higher priority than *GridLayout*. There are no *GridLayouts* in the example introduced in Figure 5.5.

BorderLayout

This layout divides the container into five parts: left, right, top, bottom and center. The heuristic selects at most one tile for each one of the five given parts as follows. A tile *t* will be: in the top part if $t.vAlignment = TOP$, in the left part if $t.hAlignment = LEFT$, in the center part if $t.hAlignment = CENTER$, in the right part if $t.hAlignment = RIGHT$, and in the bottom part if $t.vAlignment = BOTTOM$. In addition, for a tile to match a part it must keep some relations with the rest of the tiles (e.g. the left tile must be below the top tile, on the left of the center tile, and above the bottom tile).
The fitness function evaluates the tiles that can fit any of the five areas predefined by a *BorderLayout*. If there is more than one tile that can fit one part, these "excess" tiles are penalised. In contrast to other layouts, a *HoleTile* is not penalised but permitted. Note that because the *FlowLayout* and *StackLayout* have a higher priority, a *BorderLayout* with emtpy parts (i.e. *HoleTiles*) that matches *FlowLayout* or *StackLayout* will be never selected. For instance, in Figure 5.5, the regions *R1*, *R2* and *R3* could be considered as a *BorderLayout* with top, center and bottom parts, but they are detected as a *StackLayout*.
In Figure 5.5, we can find an example of *BorderLayout* in the region *R3*. In that region, the widgets *AddButton* and *DelButton* are grouped in a fine-grained tile and *ExitButton* is another fine-grained tile. Thus, *AddButton* and *DelButton* are the left part and *ExitButton* is the right part of the *BorderLayout* (there are only two parts). In this case the fitness function associated with the BorderLayout returns 1 (i.e. the maximum value), so we can see that the hole has not been penalised.

HVLayout and VHLayout

An *HVLayout* is a *FlowLayout* composed of *StackLayouts*. A *VHLayout* is a *StackLayout* composed of *FlowLayouts*. An *HVLayout* can have a different number of elements in each column while in a *GridLayout* all the columns must have the same number of rows. Similarly *VHLayout* is not restricted to have the same number of elements in the lines (rows) as in a *GridLayout*.

The *HVLayout* heuristic obtains the group of tiles that have no upper tiles. From the top-left tile it navigates the tiles from the top to the bottom until there are no more tiles below, and it thus obtains the first column. The tile from the upper tiles that is next to the top-left tile is then selected and navigated to the bottom until it obtains a second column which will be to the right of the first column. This process is repeated while new columns on the right of existing ones can be found. The heuristic penalises *HoleTiles*. The heuristic for *VHLayout* is similar to the *HVLayout* heuristic but in this case it searches for rows until there are no more rows below the previous one.

VHLayout and *HVLayout* are more general than the others and may fit in most cases, in fact *VHLayout* is the most common layout found in legacy applications. On the other hand they are less specific and do not capture the visual design as well as other layouts such as *GridLayout* and *BorderLayout*. Because of this, *GridLayout* and *BorderLayout* have a higher priority than *VHLayout* and *HVLayout*, but a lower priority than *FlowLayout* and *StackLayout* since the latter are more specific.

In the example window in Figure 5.5, we can see a *VHLayout* in region *R1*, where there are two lines of widgets.

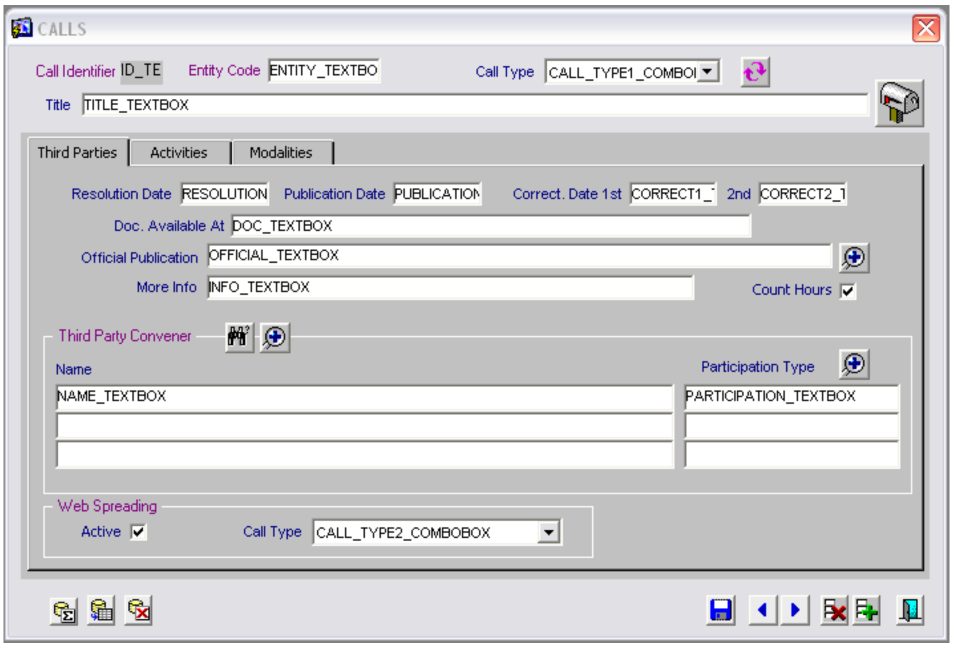
Unknown

If the maximum value returned by all the fitness functions is below a certain threshold (it has been set to 0.65, which means that equals or more than 65% of the elements in the group must fit the layout), then an *UnknownLayout* is created, which is a special layout that indicates that the layout of the group must be determined by the developer.

5.7 Detailed example

This section illustrates our GUI reengineering approach by applying it to an example in detail. A typical window from the case study presented in Section 5.8 has been selected and translated into English. This shown in Figure 5.15, and will be used throughout this section to guide the explanation.

The window is used to manage grant calls. The upper part of the window contains some administrative information about the call, such as the title, the identifier and the type. There is also a button to refresh the data and a button to send the call data to an administrator by e-mail.


CALLS

Call Identifier: ID_TE
Entity Code: ENTITY_TEXTBO
Call Type: CALL_TYPE1_COMBOI
Title: TITLE_TEXTBOX

Third Parties | Activities | Modalities

Resolution Date: RESOLUTION
Publication Date: PUBLICATION
Correct. Date 1st: CORRECT1_
2nd: CORRECT2_1
Doc. Available At: DOC_TEXTBOX
Official Publication: OFFICIAL_TEXTBOX
More Info: INFO_TEXTBOX
Count Hours ✓

Third Party Convener

Name	Participation Type
NAME_TEXTBOX	PARTICIPATION_TEXTBOX

Web Spreading
Active ✓
Call Type: CALL_TYPE2_COMBOBOX

Figure 5.15: Example window

The middle of the window contains a tabbed panel, which provides more information about the calls. The *Third Parties* tab contains general call information such as the resolution date, the date of the publication and later corrections, or where it is published. The *Third Party Convener* frame specifies which companies are involved in the call and the type of participation. There is also a functionality with which to search for companies by means of buttons. The *Web Spreading* frame contains information related to the on-line publication of the call. The lower part of the window contains several buttons which are used to add, delete or update the data, in addition to other functions such as quit the application.

Each step of the migration chain will be described in the following sections.

5.7.1 Injection of Forms models

The first step consists of obtaining models of the user interface from the source system. Oracle provides a tool with which to export FMB files to XML files and this tool has been used to obtain a definition of the application GUI in XML files. As explained in Section 5.9.1 EMF was used to automatically obtain models that mirror the information contained in the Oracle Forms files.

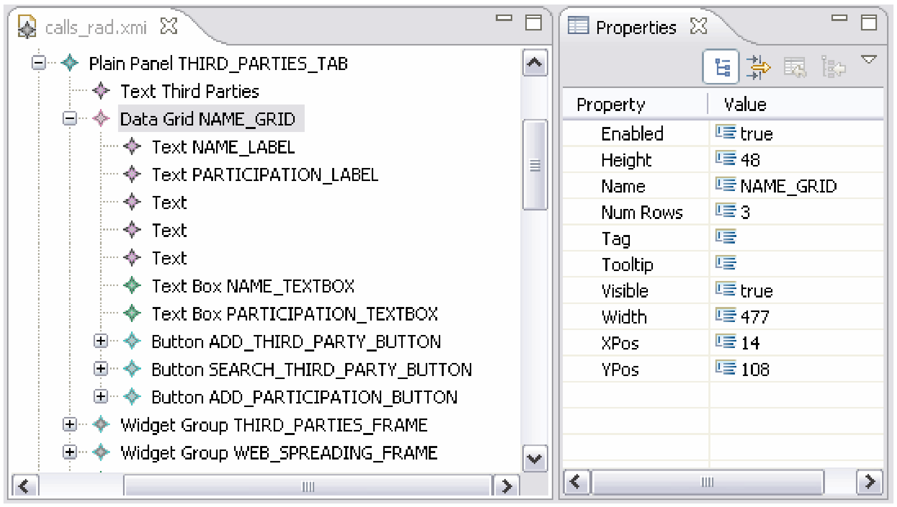
5.7.2 Mapping Oracle Forms to RAD models

The model obtained in the previous step is mapped to a RAD model, thus enabling the reverse engineering algorithms to be applied (as explained in Section 5.9.2). The RAD model generated is basically a model which contains panels and widgets that have mostly the same structure as the canvases and widgets in the Forms model. Two problems arise when attempting to normalise the example window to our RAD representation, which were explained in Section 5.9.2: the position of prompts (labels) and how to migrate the multi-record that appears inside the *Third Party Convener* frame.

For the first problem, we have an auxiliary module that calculates the relative coordinates of the labels based on: i) certain attributes of the prompt such as alignment, attachment edge, attachment offset, alignment offset, ii) certain attributes of the font such as: font name, font size, font spacing, and iii) the height and width of the text displayed.

With regard to the second problem, in Oracle Forms several instances (multi-record) of the same widget type can occur, while in current GUI technologies is usually represented with a table widget. This is the case of the two text fields that appear inside the *Third Party Convener* frame. The issue here is to decide, appart from the two text fields and their associated labels, which widget must belong to the table. It seems clear that the button above the *PARTICIPATION_TEXTBOX* is related to the table, but this is not so clear for the two buttons that are on the frame line. This problem has been solved by grouping all the widgets that belong to the same datablock in the same table, since these datablocks contain multi-record widgets. Another option, would be to consider the percentage of the widget surface that is visually contained in a frame that includes multi-record widgets. Thus, if 50 per cent or more of the area of a widget is contained in a frame, it will be included in that frame. This could be used to arrange widgets that overlap one or more frames, although the latter is not a common case. In order to generate the coordinates and size of the new table in the RAD model, the related buttons are ignored since they can be scattered in the window (not neccesarily next to the multi-record), and only the area occupied by the multi-records is considered.

Figure 5.16 shows an excerpt of the resulting RAD model, which shows how the widgets in the *Third Party Convener* frame have been transformed. As can be seen, the prompts associated with the text fields are the titles of the columns, and the widgets are the types of the columns. Since buttons do not have an associated prompt (i.e., there is no label next to a button), there is no header for the button columns. It is worth noting that the *NAME_GRID* table is not contained



```
Plain Panel THIRD_PARTIES_TAB
    Text Third Parties
    Data Grid NAME_GRID
        Text NAME_LABEL
        Text PARTICIPATION_LABEL
        Text
        Text
        Text
        Text Box NAME_TEXTBOX
        Text Box PARTICIPATION_TEXTBOX
        Button ADD_THIRD_PARTY_BUTTON
        Button SEARCH_THIRD_PARTY_BUTTON
        Button ADD_PARTICIPATION_BUTTON
    Widget Group THIRD_PARTIES_FRAME
    Widget Group WEB_SPREADING_FRAME
```

Property	Value
Enabled	true
Height	48
Name	NAME_GRID
Num Rows	3
Tag	
Tooltip	
Visible	true
Width	477
XPos	14
YPos	108

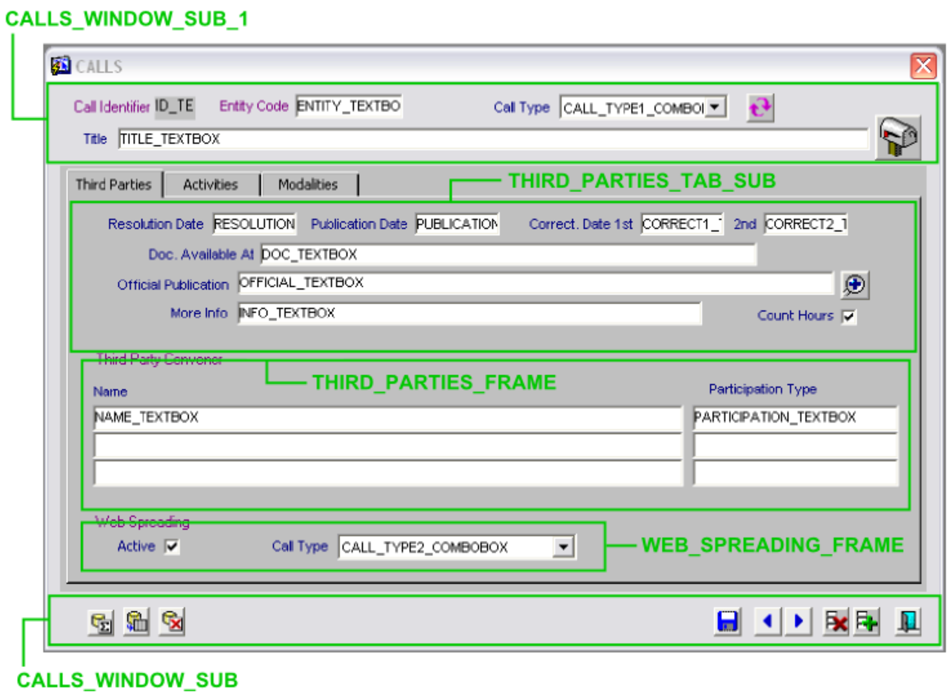
Figure 5.16: Excerpt of the RAD Model for the example window in Figure 5.15

in the *THIRD_PARTIES_FRAME* but that they are siblings, despite the fact that a parent-child relationship exists between them.

5.7.3 Identification of the regions

Using the RAD model as a starting point, we apply the M2M transformation that implements our algorithm in order to identify regions. The main regions identified are shown in Figure 5.17 (note that we have removed the buttons from the *Third Party Convener* frame since now they are assumed to belong to the table called *NAME_GRID*). The *CALLS_WINDOW_SUB_1* and *CALLS_WINDOW_SUB* regions are created in order to prevent widgets such as *TITLE_-TEXTBOX* from being at the same level of the tabbed panel. For the two new groups of widgets, the area that is enclosed by the regions is limited by the bounds of the tabbed panel and the bounds of the window itself. The transformation also creates a region based on the two frames that appear in the *Third Parties* tab. Some widgets are inside the *Third Parties* tab and are not contained in any of the frames, and a new region is therefore created in order to avoid this situation, as occurred previously. In all cases, the widgets are modeled as regions inside the corresponding container region.

This phase of the reverse engineering process not only identifies regions, but also corrects the containment of the regions so that they match the visual aspect. The regions are nested as they are visually displayed, as can be seen in the tree view of the Region model in Figure 5.18, which



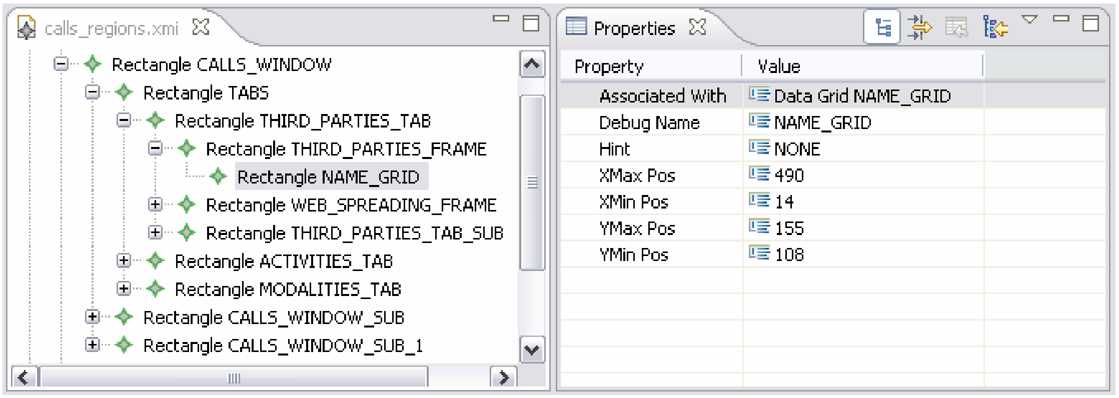
Call Identifier ID_TE
Entity Code ENTITY_TEXTBO
Call Type CALL_TYPE1_COMBOI
Title TITLE_TEXTBOX
Resolution Date RESOLUTION
Publication Date PUBLICATION
Correct. Date 1st CORRECT1_
2nd CORRECT2_1
Doc. Available At DOC_TEXTBOX
Official Publication OFFICIAL_TEXTBOX
More Info INFO_TEXTBOX
Count Hours ✓
Name NAME_TEXTBOX
Participation Type PARTICIPATION_TEXTBOX
Active ✓
Call Type CALL_TYPE2_COMBOBOX

Figure 5.17: Some regions identified for the example window in Figure 5.15.

has been obtained using the EMF tree editor (this editor reflects the containment relationships that exists between the elements in the model). It is also possible to appreciate that the *NAME_GRID* region is not a sibling of *THIRD_PARTIES_FRAME*, but is nested into it. Note that in the region model, all the elements are placed by means of coordinates. If the coordinate systems between Figure 5.16 and Figure 5.18 are compared, a slight difference will be noted. In the RAD Model the area occupied by a widget is represented by the X and Y coordinates of the upper-left corner and the width and height, whereas in the Region Model the same area is defined by the X and Y coordinates of the upper-left corner and the coordinates of the lower-right corner. Although both are equivalent, the second means to represent the area allows the number of operations in the algorithms to be reduced.

5.7.4 Recovering the low-level layout

In this phase of the process, further refinement of the regions is performed and the coordinate-based positioning system is replaced with spatial relationships among the elements (i.e., *tiles*), bearing in mind that there may be parts without widgets (previously referred to as *holes*).


calls_regions.xmi

- Rectangle CALLS_WINDOW
  - Rectangle TABS
    - Rectangle THIRD_PARTIES_TAB
      - Rectangle THIRD_PARTIES_FRAME
        - Rectangle NAME_GRID
      - Rectangle WEB_SPREADING_FRAME
      - Rectangle THIRD_PARTIES_TAB_SUB
    - Rectangle ACTIVITIES_TAB
    - Rectangle MODALITIES_TAB
  - Rectangle CALLS_WINDOW_SUB
  - Rectangle CALLS_WINDOW_SUB_1

Properties

Property	Value
Associated With	Data Grid NAME_GRID
Debug Name	NAME_GRID
Hint	NONE
XMax Pos	490
XMin Pos	14
YMax Pos	155
YMin Pos	108

Figure 5.18: Excerpt of the Region Model for the example window in Figure 5.15.

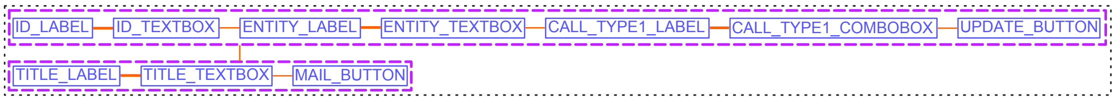
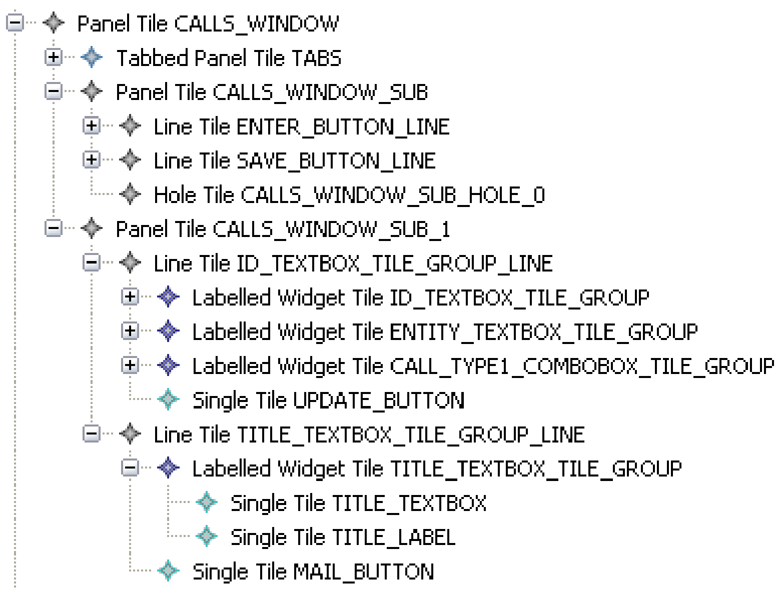


Figure 5.19: Representation of the tiles in the upper part of the window

We shall now analyse the *CALLS_WINDOW_SUB_1* region in Figure 5.17 which is located at the top of the window. Our algorithm generates a *PanelTile* based on this region, and it infers that the region is composed of a sequence of two horizontal lines (*LineTiles*). The tiles identified are shown in Figure 5.19. The rectangle drawn with a black dotted-line represents a *PanelTile*, the two drawn with purple dashed-lines represent *LineTiles* and the boxes drawn with blue solid-lines are *SingleTiles*.

The orange connectors (with a thick or thin line) between the boxes represent relationships. For example, the horizontal connector between *ID_LABEL* and *ID_TEXTBOX* signifies that the first one is on the left of the second one and the second one is on the right of the first one. In addition, thick connectors mean the widgets are close together. More particularly, every widget and its associated label are close together, and we have therefore depicted this association with thick orange connectors, which in the Tile model representation will be encapsulated in a *LabelledTile*. Figure 5.20 shows an excerpt of the Tile model which shows how the tiles are nested.

Two details in this example are notable. The first is that there is a certain distance between *ENTITY_TEXTBOX* and *CALL_TYPE1_LABEL* but they still belong to the same line. This is a consequence of the default configuration of the approach, since if there is not a relatively



```
Panel Tile CALLS_WINDOW
  Tabbed Panel Tile TABS
  Panel Tile CALLS_WINDOW_SUB
    Line Tile ENTER_BUTTON_LINE
    Line Tile SAVE_BUTTON_LINE
    Hole Tile CALLS_WINDOW_SUB_HOLE_0
  Panel Tile CALLS_WINDOW_SUB_1
    Line Tile ID_TEXTBOX_TILE_GROUP_LINE
      Labelled Widget Tile ID_TEXTBOX_TILE_GROUP
      Labelled Widget Tile ENTITY_TEXTBOX_TILE_GROUP
      Labelled Widget Tile CALL_TYPE1_COMBOBOX_TILE_GROUP
      Single Tile UPDATE_BUTTON
    Line Tile TITLE_TEXTBOX_TILE_GROUP_LINE
      Labelled Widget Tile TITLE_TEXTBOX_TILE_GROUP
        Single Tile TITLE_TEXTBOX
        Single Tile TITLE_LABEL
      Single Tile MAIL_BUTTON
```

Figure 5.20: Excerpt of the Tile Model for the example window in Figure 5.15.

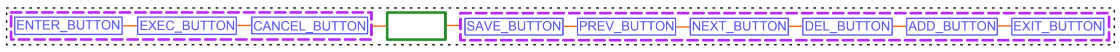


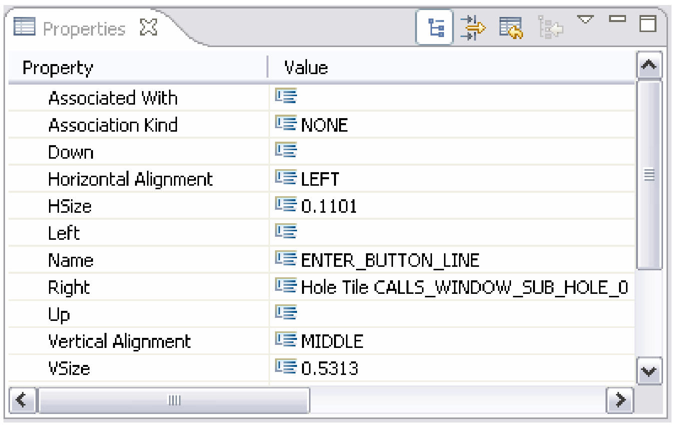
Figure 5.21: Representation of the tiles in the lower part of the window

wide gap between two consecutive widgets (which is set to 40 pixels for the applications created with Oracle Forms), they are included in the same line. In our case the desired result was to keep just once single line in order to ensure that the default configuration was suitable, but the framework configuration files could have been tuned if another different partitioning had been required.

The second detail is that the *MAIL_BUTTON* could have been separated from the second line since it could still be part of the first line, but given that the horizontal intersection with the second line is complete, and the horizontal intersection with the first line is only partial, it is included in the second line.

We shall now shift the focus to the lower part of the window, to the *CALLS_WINDOW_SUB* region in Figure 5.17. The representation of the tiles identified for this region is depicted in Figure 5.21.

In this case, it can be observed that there is a great distance between the two groups of widgets, and our approach has detected a hole between the tiles of buttons (depicted as a green rectangle). It is also worth noting that the first tile is aligned to the left and the second tile is aligned



Property	Value
Associated With	
Association Kind	NONE
Down	
Horizontal Alignment	LEFT
HSize	0.1101
Left	
Name	ENTER_BUTTON_LINE
Right	Hole Tile CALLS_WINDOW_SUB_HOLE_0
Up	
Vertical Alignment	MIDDLE
VSize	0.5313

Figure 5.22: Properties of the lower-left tile of buttons

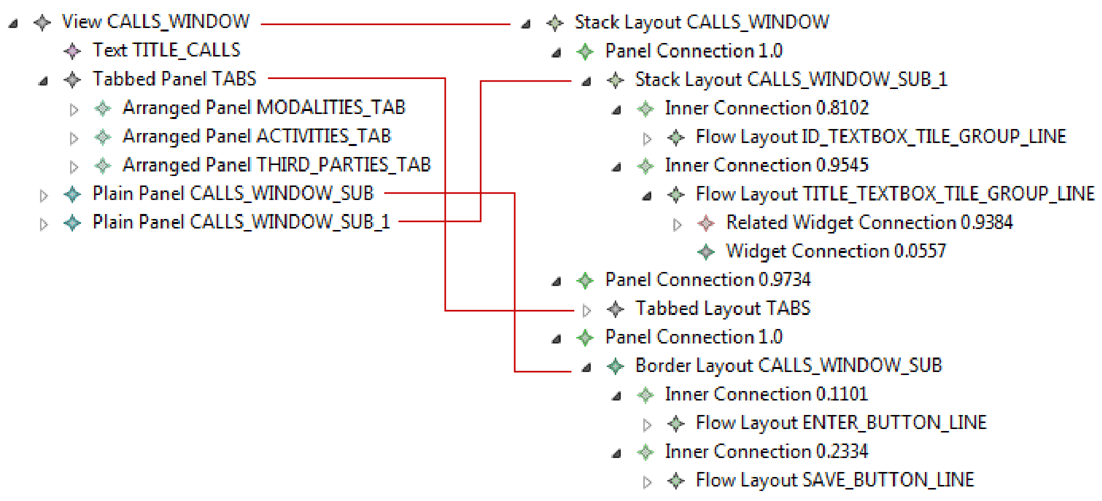
to the right.

Figure 5.22 includes a fragment of the property sheet of the lower-left group of buttons (*ENTER_BUTTON_LINE* tile). It is worth highlighting some attributes: the *hSize* and *vSize* attributes that represents the percentage of space taken up by a widget, *left*, *right*, *up*, *down* that maintain the relationships among the tiles, and the *horizontalAlignment* and *verticalAlignment*.

5.7.5 Recovery of the high level layout

Figure 5.23 shows a fragment of the resultant CUI model which has been split into two parts. The left part specifies the structure of the window and the right part details the layout of the window. It is worth noting that the order of the model elements in the left part is arbitrary whereas in the right part the order is part of the layout information. As can be seen, the overall layout of the window is a *StackLayout* since the three main regions in the window are arranged in a vertical sequence. This is the layout selected since the *StackLayout* fitness function for the set of tiles {*CALLS_WINDOW_SUB_1*, TABS, *CALLS_WINDOW_SUB*} (which can be seen in Figure 5.20) returns 1, given that all the tiles are visited if we start from the upper tile (*CALLS_WINDOW_SUB_1*) and we navigate them from top to bottom. It is also worth noting that the StackLayout has a higher priority than other layouts (such as VHLayout), and this is why it has been selected.

In the upper part (*CALLS_WINDOW_SUB_1*) we also have a *StackLayout* composed of two horizontal flows (*FlowLayout*). The two horizontal flows are composed of single widgets (*WidgetConnection*) and pairs composed of a label and its associated widget (*RelatedWidgetConnec-*



```
View CALLS_WINDOW
  Text TITLE_CALLS
  Tabbed Panel TABS
    Arranged Panel MODALITIES_TAB
    Arranged Panel ACTIVITIES_TAB
    Arranged Panel THIRD_PARTIES_TAB
  Plain Panel CALLS_WINDOW_SUB
  Plain Panel CALLS_WINDOW_SUB_1
```

```
Stack Layout CALLS_WINDOW
  Panel Connection 1.0
    Stack Layout CALLS_WINDOW_SUB_1
      Inner Connection 0.8102
        Flow Layout ID_TEXTBOX_TILE_GROUP_LINE
      Inner Connection 0.9545
        Flow Layout TITLE_TEXTBOX_TILE_GROUP_LINE
          Related Widget Connection 0.9384
          Widget Connection 0.0557
  Panel Connection 0.9734
    Tabbed Layout TABS
  Panel Connection 1.0
    Border Layout CALLS_WINDOW_SUB
      Inner Connection 0.1101
        Flow Layout ENTER_BUTTON_LINE
      Inner Connection 0.2334
        Flow Layout SAVE_BUTTON_LINE
```

Figure 5.23: Excerpt of the CUI Model for the example window split into two parts

tion). The layout for the middle part of the window (*THIRD_PARTIES_TAB*), which has been omitted to make the model more readable, is very similar to the previous one, i.e. it is a *StackLayout* in which each region is a *StackLayout* of *FlowLayout*.

The lower part of the window (*CALLS_WINDOW_SUB*) is composed of a tile of buttons aligned to the left, an empty tile in the middle and another tile of buttons aligned to the right. As occurs in all the cases, the fitness functions are calculated for the set of tiles and the heuristic with the highest fitness value and highest priority is selected. In this case the layout selected is *BorderLayout* since the function returns 1 and none of the fitness functions with higher priority return such a high value.

The connection elements in the CUI model (such as *InnerConnection* or *WidgetConnection*) are used to maintain information about the alignment and the amount of space occupied by a portion of the GUI (for example, a layout or a concrete widget). For each *InnerConnection* the alignment of the nested layout with regard to the container layout is maintained, and in addition to the percentage of vertical and horizontal space that is occupied by that layout. The values that are shown in Figure 5.23 next to the *InnerConnections* actually represent the horizontal space, i.e. the *hSize* attribute. This information is useful in order to generate precise layouts.

In some cases some widgets are not aligned by their container but are aligned to other widgets. For example, in Figure 5.15, *OFFICIAL_TEXTBOX* and *INFO_TEXTBOX* are both aligned with regard to the left edge of the boxes. This issue is not addressed in the current version of the

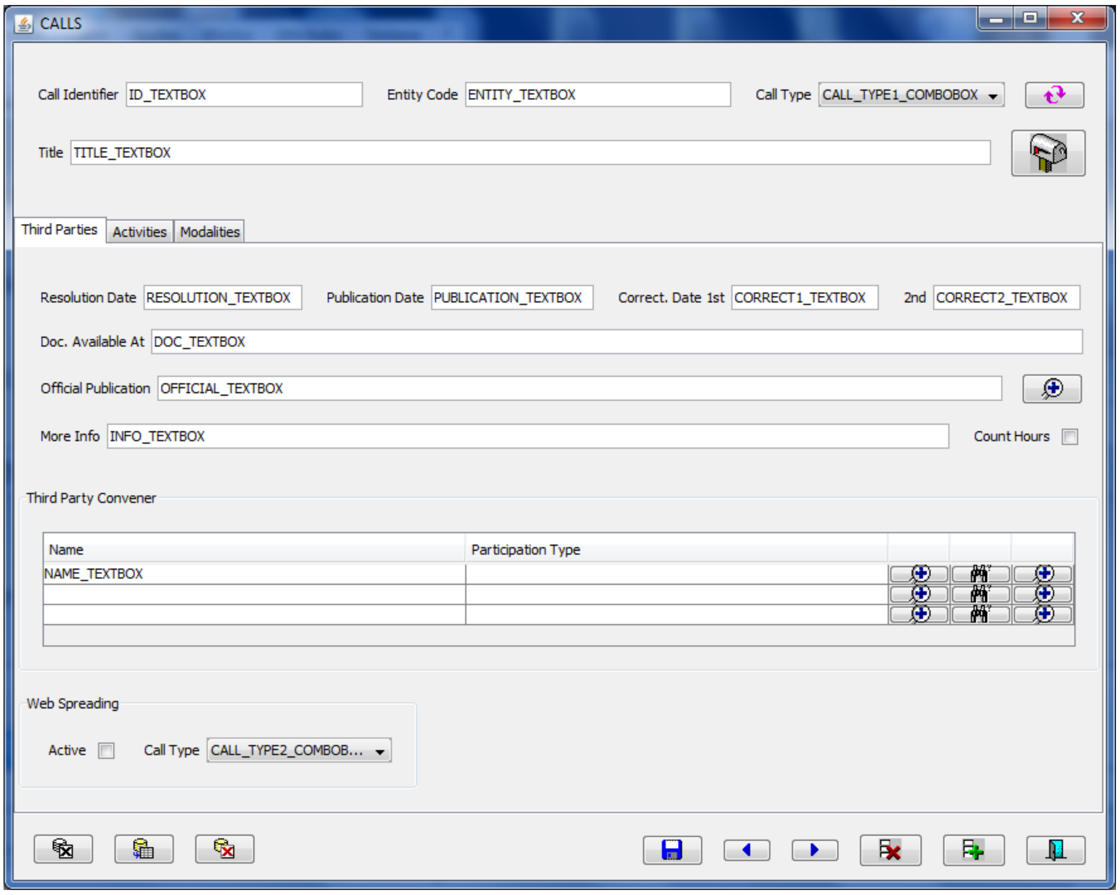


Figure 5.24: The example window shown in Figure 5.15 migrated to Java Swing

framework.

5.7.6 Generation of Java Swing code

Once the structure and layout of the window in the CUI model have been captured it is possible to take advantage of this information in order to automate some restructuring and forward engineering tasks. In particular, the example window in Figure 5.15 has been migrated to Java Swing. The new Java Swing window is shown in Figure 5.24.

5.8 Case study: from Oracle Forms to Java

In order to evaluate our approach and demonstrate its applicability we have applied our prototype to the GUI of two real applications in two different domains created by two different companies.
The case study A is a business management application which is intended to be used to manage the research projects and grants that are assigned to the research groups of a Spanish university. It is composed of 107 windows, which indicates a medium-high complexity. The application was developed by different developers of the same company and the conventions concerning the style of the forms were not particularly strict, signifying that there is a variety of form styles.
The case study B is a business management application targeted at being used by a department of the Regional Government to deal with budgets, income sources, expenses, investment projects and human resources. The application consisting of 57 windows has a medium complexity. Though this application was also programmed by different people, the windows follow a more strict style (imposed by the company) than in the case study A.
Both applications were developed in Oracle Forms 6 and both applications needed to be migrated to the Java platform.

5.8.1 Methodology

When recovering the visual appearance of a window, it frequently occurs that different layouts applied to the same widgets could result in a window with a similar visual appearance. The evaluation of our approach cannot therefore be accomplished by simply comparing the layout produced by our tool with an expected layout visually. Instead, the following steps are performed for each window:

1. The original window is manually analysed by a member of our team (different to the developer of the tool) in such a way that certain data concerning the following criteria are registered:

    - *Window parts*. Identify the parts of the window and register the relationships among them. A *part* is defined as a group of widgets that form a distinguished area of the window. A part is a set of close widgets which:
        – is visually highlighted by means, for example, of a surrounding frame or is enclosed in a coloured rectangle.

    - is distant to other groups of widgets.
    - has other parts around it to which the widgets do not belong.

  Note that in our approach, parts are normally represented with coarse-grained tiles, although this is irrelevant to the person in charge of performing the evaluation.

  - *Relationships among widgets*. The structure of the widgets within each part is identified: the position of every widget regarding the others and the part, and the alignment which exists among them and with regard to the part.

  The rationale for identifying parts is to have a layout-independent notion of the coarse-grained structure of the windows, while the relationships among the widgets are related to the fine-grained structure of the window.

2. The complete reverse engineering transformation chain is executed for the given window to obtain a CUI model. This CUI model is used to execute an additional generation step in order to obtain a Java Swing GUI, which uses the layout discovered.

3. The GUI generated is now assessed by the same person and using the same criteria as in step 1, and is compared with the data gathered from the original window. The CUI model is also analysed in order to avoid that mistakes in the Java Swing generator could mislead the evaluation, since in some cases the generated GUI had layout mistakes because of bugs in the Swing template. Two main metrics are obtained in the evaluation process for each window:

   - *Parts laid out OK*. For a part to be correct, it must contain the same widgets and it must be located in the same place as the original window.
   - *Widgets laid out OK*. The widgets within each part are analysed, by counting which widgets are located in the right place with regard to the container part and other widgets, also taking into account their alignment.

The criteria used to assess both the original and the generated windows are obviously subjective. In order to reduce the inconsistencies between the results, the evaluation of all the windows has been performed by the same person. 15% of the windows (with a range of complexity) were also evaluated by a second member of our team to check whether his evaluation matches to a great extent with the one carried out by the main evaluator. This aims at ensuring that the main evaluator has not introduced a strong systematic bias.

5.8.2 Evaluation results

The results of the evaluation of the two case studies are summarised in Table 5.1 and Table 5.2. In order to show the scalability of our approach we have classified the windows used in the evaluation into three groups, according to the number of widgets involved. As can be observed, there are a large number of small windows in both cases (63.55% for the case study A and 66.67% for the case study B) which are used as dialogs, for example, to perform searches based on certain criteria. Almost 20% of the windows in the case study A are large (an average of 86 widgets/window), and commonly use tabbed panels to arrange the widgets (an average of 4.24 canvases/window). In contrast, the case study B contains more medium-size windows (28.07%) and a few large windows (5.26%).

	Large (>60)	Medium (20 - 60)	Small (<20)	Total
Total amount of windows	21	18	68	107
Windows of each type (out of the total)	19.63%	16.82%	63.55%	100%
Total canvases	89	19	69	177
Canvas/window average	4.24	1.06	1.01	1.65
Widgets/window average	86.00	36.43	8.10	28.15
Parts/window average	10.18	3.14	1.70	3.61
Parts laid out OK	83.24%	98.06%	100.00%	96.38%
Widgets laid out OK	87.14%	85.61%	88.10%	87.50%

Table 5.1: Evaluation results for the case study A.

	Large (>60)	Medium (20 - 60)	Small (<20)	Total
Total amount of windows	3	16	38	57
Windows of each type (out of the total)	5.26%	28.07%	66.67%	100%
Total canvases	6	24	38	68
Canvas/window average	2.00	1.50	1.00	1.19
Widgets/window average	65.33	37.31	7.03	18.60
Parts/window average	5.00	4.31	2.42	3.09
Parts laid out OK	89.00%	94.75%	100.00%	97.95%
Widgets laid out OK	95.87%	89.98%	97.80%	95.51%

Table 5.2: Evaluation results for the case study B.

Figures 5.25 and 5.26 show the dispersion of the success rate (as a percentage) of our approach when identifying parts for both case studies, and Figures 5.27 and 5.28 represent the dispersion of the success rate when placing widgets. The plots also include a regression curve which expresses the tendency of the percentages when the number of elements increases. Various conclusions can be drawn from these results.

In general, the accuracy of the coarse-grained layout detection (parts) is 100% when there are few widgets and it drops when the number of widgets increases. In the case study A there are a few outliers (below an accuracy of 40%) that correspond to a special kind of layout that we have not considered (this will be commented on in the description of the non-regular layout detection in Section 5.8.3). In the case study B we have a high success rate (almost 98%) because the windows are better structured and the visible parts are normally surrounded by borders (frames). The errors in this case are mainly due to frames which have been emulated by forming a rectangle with four single graphical lines. This feature is not supported at present, which leads to unidentified regions. In both case studies, the recognition of parts is higher than 80% in the majority of occasions, which could be considered as an acceptable rate.

With regard to the fine-grained layout detection (widgets laid out properly), in the case study A there are several windows whose accuracy is below 80%, particularly those with less than 20 widgets. We have observed that they normally correspond to dialog windows, in which the developers place many widgets very close together in order to make the most of the available space in the dialog. We have also observed that, in the case study application, buttons are sometimes situated in a particular place simply because there is some free space there. In these cases, a small refactoring of the generated layout will lead to a cleaner GUI. In the case study B we can see that most of the windows have a certain error rate that stems from the fact that some widgets are missing because they are not well-recognised in the current implementation (it is a problem of detecting the widgets to generate the Normalised model). In some of the windows, mainly in the medium-size windows, we have also a slightly higher error rate since the layouts obtained do not properly reflect the holes detected (the *unidentified holes* problem will be explained in Section 5.8.3).

The plots show that there is not a considerable variation in the success rate neither for detecting regions nor for placing widgets in the window when the number of widgets increases, so it seems that our approach is scalable for reasonably large GUIs (in the case study A it has been applied to windows with 160 widgets per window). The rationale behind this result is that large GUIs are typically arranged in parts, either using explicit markup elements (e.g., frames) or using implicit separators such as blank spaces.

Comparing both case studies, the best results are obtained in the case study B, mainly because the windows follow a more strict style than in the case study A. The success rate of the layout of

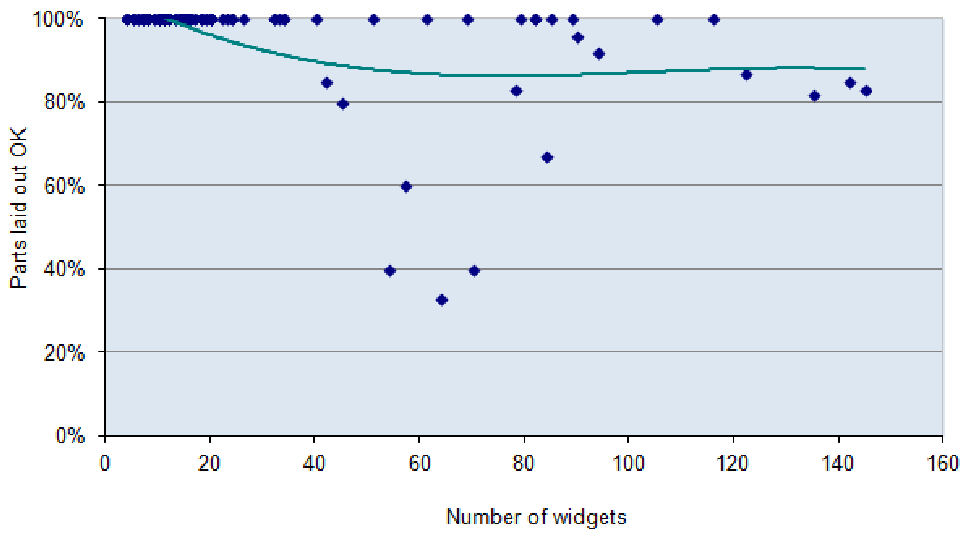


Figure 5.25: Scatter plot that represents the accuracy of part detection for the case study A.

the parts in case study B is higher (6% better for large windows) mainly because the parts are surrounded by frames. In general, our approach works better when explicit markup elements are used. The improvement of the widget layout in the case study B (8% better for large windows) is because the widgets are not scattered but conform to more or less common layouts. Considering both measures together, we can claim that our approach has an acceptable accuracy rate. It is important to note that in any case, the CUI model obtained after the discovery process can be edited either to fix errors or to refactor the GUI.

5.8.3 Limitations of the approach

This evaluation has allowed us to identify a set of limitations that are not currently dealt with by our approach, and which may lead to inaccuracies in the layout recovery process.

Missing parts identification. Parts that are not explicitly limited by frames, panels or the boundaries of the window itself are not identified as regions or coarse-grained tiles. For example, in Figure 5.29 it is possible to visually identify two parts in the window because

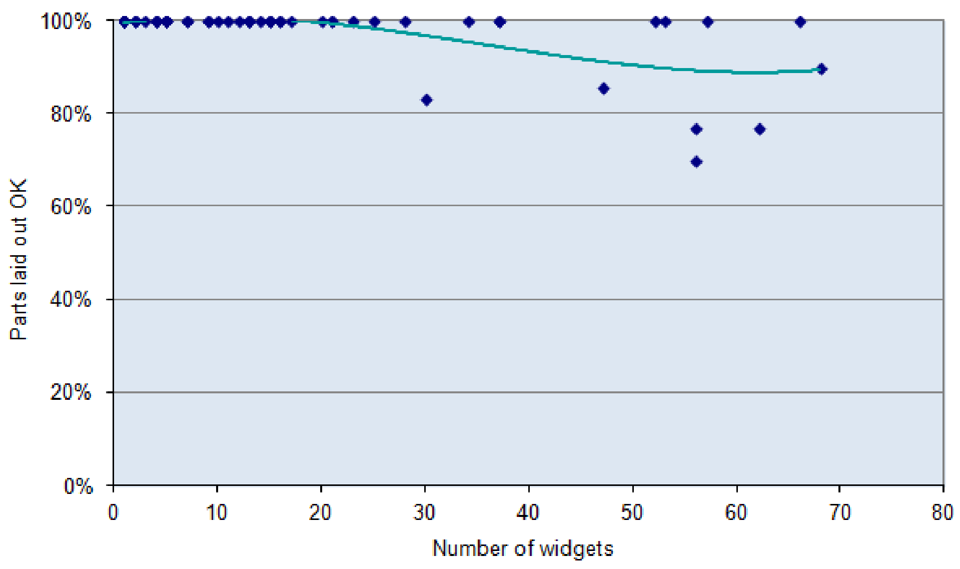


Figure 5.26: Scatter plot that represents the accuracy of part detection for the case study B.

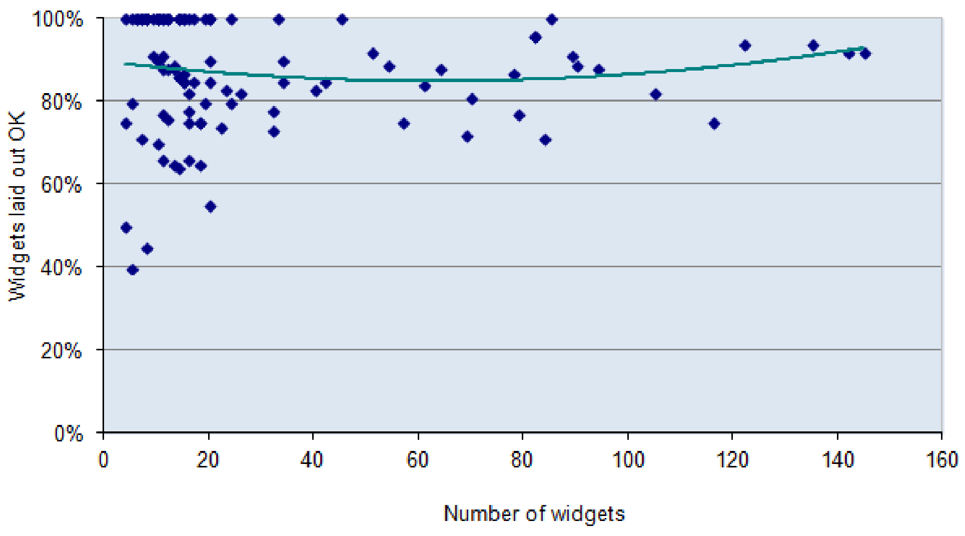


Figure 5.27: Scatter plot that represents the accuracy of widget placement for the case study A.

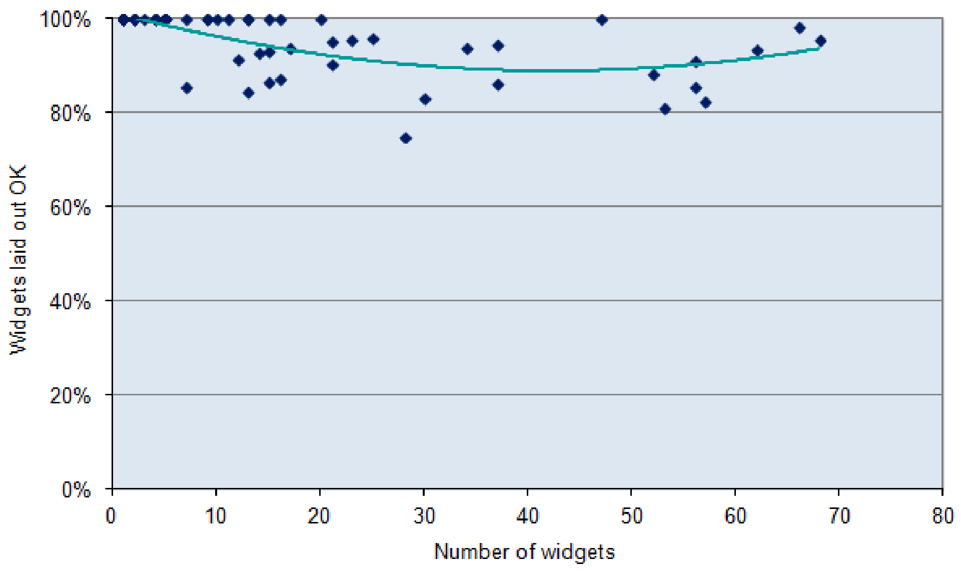


Figure 5.28: Scatter plot that represents the accuracy of widget placement for the case study B.

of the distance between the elements. The first is the upper part which is composed of the labels and text boxes, and the second is the lower part which is composed of buttons. In this case, our algorithm will create just one coarse-grained tile, made up of three line tiles. However, this inaccuracy does not always make the final layout incorrect, since the content of the region could be laid out correctly, as is the case of the example.

Non-regular layout detection. It sometimes occurs that there are widget arrangements that do not have a regular structure, and they cannot therefore be easily represented with a layout system. For example, the window in Figure 5.30 shows a layout that is not properly detected. There are two main problems: (1) our algorithms identify only one part, so they are not able to split the window into one part for the input and the buttons, and another part for the checkboxes, and (2) “Option 5”is not aligned with any column. In this case, an algorithm that is focused on small groups of widgets and creates a composite layout might attain better results.

Widget alignment with other widgets. Our approach uses a Tile model to arrange structures of widgets in terms of tiles that can be nested and it is possible to specify that the tiles inside a tile are aligned to the left, right, top or bottom. Nevertheless, in some cases we

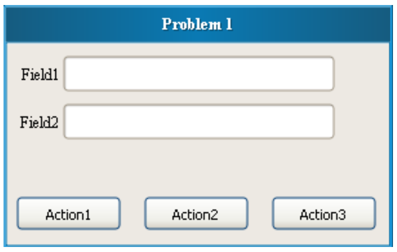


Figure 5.29: Missing part identification problem

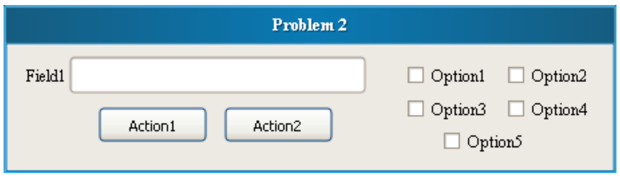


Figure 5.30: Non-regular layout detection problem

have found that widgets are aligned with regard to other widgets (rather than with regard to their parent container). In order to implement this feature which was not considered in the design of the solution, it is necessary to align the tiles with regard to their sibling tiles. In our evaluation, this situation occurred in a small percentage of windows, but in some cases the current implementation generated a good layout because the sibling tiles that are aligned are both aligned with regard to the same parent tile.

Unidentified holes. Hole recognition is addressed in the algorithm that generates the Tile model, and depends on the parameters that specify the minimum distance between tiles (seen as a percentage of the width/height occupied by the tile with regard to its parent). If the parameters are set in such a way that a small distance is captured as a hole, there will be a lot of holes and the high-level layout algorithm (that generates the CUI) will not know how to deal with them. We therefore prefer to set the parameters in such a way that only notable holes are captured. This implies that some holes are not identified, but this hardly ever occurs.

On the whole we can state that our approach has an accuracy of 96% when laying out parts and an accuracy of 87% for widgets inside the parts. Simple layouts fit the arrangement of the widgets in most cases, especially the stack of flows layout. However, the problems mentioned above should be tackled if higher success rates are to be considered. These issues will be addressed in future work.

5.9 Implementation

This section briefly presents the tools involved in and some of the implementation details of our GUI reengineering framework (see highlighted parts in Figure 5.31), focusing particularly on Oracle Forms as the legacy technology, and Java Swing as the target technology.

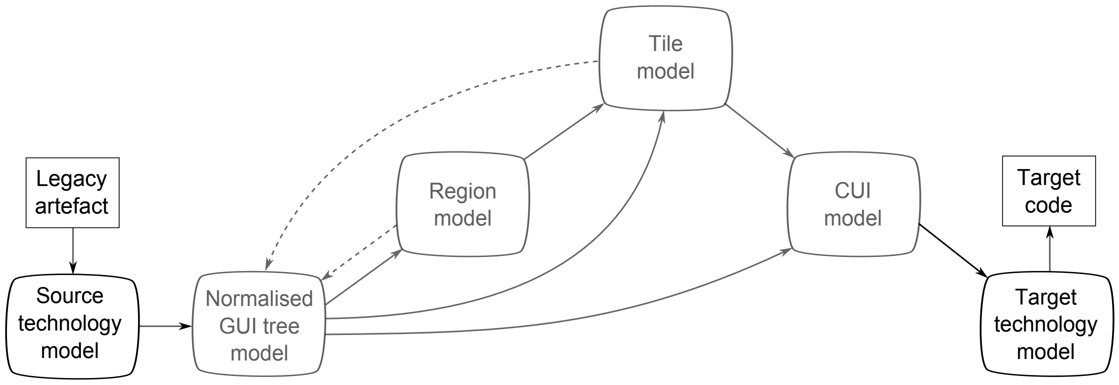


Figure 5.31: Model-based architecture used to migrate legacy GUIs.

The framework has been implemented on top of the Eclipse platform, and is based on the Eclipse Modeling Framework (*EMF*) [53]. The metamodelling language that has been selected to represent the models and metamodels is Ecore. The workflow of the reengineering process is defined and managed by a task management tool called Rake [102], a sort of Make for Ruby.

5.9.1 Injection

Let us first consider the injection step (from legacy artefacts to Source Technology models) shown in Figure 5.31. An injector is required for every legacy technology for which we want to migrate applications. It is worth noting that this step is particularly dependent on the source artefact format and the export facilities of the legacy environment. Some environments such as Delphi and Visual Basic use plain text files to store the GUI specification. Oracle Forms, however, uses a binary format (FMB files), but there is an export facility that generates XML files conforming to an XML schema which is available in the Oracle Developer Suite.

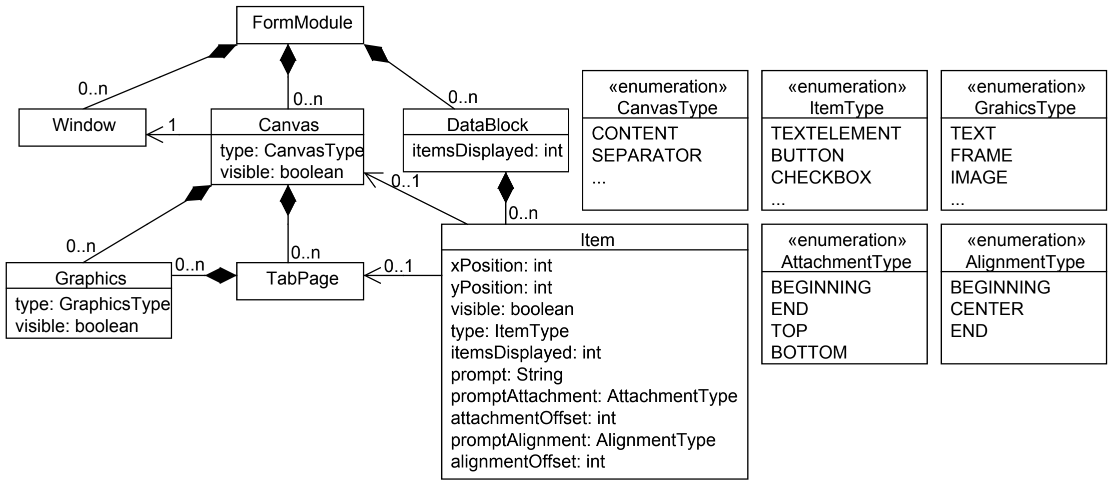


Figure 5.32: Excerpt of the Oracle Forms metamodel.

In our case, we have built an injector for Oracle Forms on the basis of the aforementioned XML schema. This has been done by using *EMF* which, given an XML schema, automatically generates a metamodel and the injector that takes XML files and creates models conforming to this metamodel. The Oracle Forms metamodel automatically derived by EMF mirrors the structure of the XML schema provided by Oracle, as is shown in Figure 5.32.
The following points summarise the structure of this metamodel.

- A *form* (*FormModule*) in Oracle Forms is a set of *Windows* with its related business logic expressed in PL/SQL triggers. The code is extracted from the XML files in a separate process, which is not within the scope of this paper.

- A *Window* can show one or several *Canvas*es, which are the panels on which the widgets are displayed. There is a special type of *Canvas* called *SEPARATOR* which can contain *TabPage*s.

- A *Canvas* is a surface that is used to display *Graphic*s and *Item*s. *Graphic*s are graphic decorators such as fixed text (*TEXT*), or graphical frames (*FRAME*). *Item*s are widgets such as buttons or text fields, which are distinguished by the *type* property. Contrary to what might be expected, *Canvas*es contain *Graphic*s but not *Item*s. *Item*s are contained in *DataBlock*s and they are associated to zero or one *Canvas*es. An *Item* will be displayed if it is associated with a *Canvas*, its *width* and *height* are greater than zero, and it has *visible* set to *true*.

- A *DataBlock* is a logical group of widgets that are often associated with columns of the same table in the database.
- The coordinates of *Item*s and *Graphic*s are relative to the *Canvas* that displays them, whereas *Canvas*es are located with absolute coordinates. In the metamodel there are no explicit relationships to specify whether a graphical frame contains other widgets or graphical frames, or whether two canvases are overlapped.
- Moreover, the *itemsDisplayed* property of *Item* indicates the number of instances of the same kind of widget that are shown. This feature is referred to as *multi-record items*, and can be regarded as a form of data grid.
- It is possible to specify a *prompt* for an *Item*, i.e. a text that is associated with the *Item*. The coordinates of the *prompt* elements are defined with regard to the associated *Item*. The *prompt* can therefore be on the left (*promptAttachment=BEGINNING*), on the right (*promptAttachment=END*), above (*promptAttachment=TOP*), or below (*promptAttachment=BOTTOM*) the *Item*, and it can also be aligned to the left (*promptAlignment= BEGINNING*), in the middle (*promptAlignment=CENTER*) or to the right (*promptAlignment=END*) of the *Item*.

5.9.2 Mapping Oracle Forms to Normalised models

Once the source artefacts related to the GUI have been injected into a model, the latter must be transformed into a Normalised model that represents the same GUI but is independent of the source technology (step from Source Technology models to Normalised models in Figure 5.2). The Normalised model can be considered as a normalised form of the source artefacts.
The Forms to Normalised transformation is, in general, fairly straightforward, since there is a direct mapping between the source technology metamodel elements and the Normalised metamodel elements. This mapping is summarised in Table 5.3.
However, Oracle Forms has some specific features which are not found in other legacy applications such as Delphi or Visual Basic. We shall now discuss two specific features that hinder the Forms-to-Normalised transformation, which are prompts and multi-record items.

Forms	Normalised
Window	View
Canvas (type=CONTENT)	PlainPanel
Canvas (type=SEPARATOR)	TabbedPanel
TabPage	PlainPanel
Graphics (type=TEXT)	Label
Graphics (type=FRAME)	WidgetGroup
Item (type=TEXTELEMENT)	TextBox
Item (type=BUTTON)	Button
Item (prompt)	Label
Item (itemsDisplayed >1)	DataGrid

Table 5.3: Forms to Normalised mappings.

In some legacy environments, there is a kind of widget that is frequently called *Label* which is a piece of static text that can be placed anywhere in a window. In contrast, Oracle Forms includes a similar widget, but it also offers another possibility, which is to use a *Prompt* element which is associated with a widget. The location of the *Prompt* can be expressed with regard to different reference points, which always refer to the associated widget. Specifically, the *Prompt* can be above, below, on the left or on the right of the widget and aligned to the beginning, the middle, or the end of the widget, which results in twelve possibilities. In order to calculate its coordinates it is necessary to obtain the width or height of the text of the *Prompt*, which is not easy since it depends on both the font type and the font size. Moreover, Forms by default does not express coordinates in pixels, but in proprietary measures. This implies that *Prompt* coordinates can contain a small error owing to the width/height calculation and the conversion between measures. In our case, we did not find a coordinate error greater than 8 pixels.

Another specific feature is multi-record widgets, that is, a widget that is replicated a number of times. For example, let us assume a window that must show some aspects of people's personal data. In this case, it will be necessary to have some text fields to display the name, surname and other data, and one multi-record text field could therefore be used for the name, another for the surname and so on. In current GUI technology we use data tables for this purpose. Since multi-record widgets can be scattered on the canvas and show different kinds of information, there is the challenge of deciding when certain multi-record widgets must be in the same table (i.e., each multi-record widget is a column of the table). The criterion used to group widgets in tables is the following: we group multi-record widgets that are closer than a fixed value where no non-multi-record widget exists between them. Moreover, when buttons that belong to the same datablock as multi-record widgets exist, they are also included in the table. Since this is a heuristic to group widgets in tables, developers might need to modify the Normalised models in order to correctly rearrange widgets in tables.

Finally, the transformation from the source technology to Normalised performs some clean up tasks. In particular, it marks the elements that are not visible and checks that widgets do not overlap. A GUI frequently includes non-visible widgets which are intended to store values that are used in transactions, so they never appear in the interface. The elements that are not visible are marked in the Normalised model, so the layout algorithms ignore them. Overlapped widgets are sometimes found in applications. Developers can use overlapping to show different information with different widgets that are displayed and hidden by means of programming. Since this is not a good practice and widget overlapping hinders layout detection algorithms, overlapping is detected, and developers must fix this to ensure that the rest of the process continues properly.

5.9.3 Reverse engineering

With regard to the reverse engineering stage of Figure 5.2, the algorithms presented in Sections 5.4, 5.5, 5.6 have been implemented as a chain of M2M transformations. To this end we have chosen the RubyTL [61] language. RubyTL is a rule-based M2M transformation language embedded in Ruby which is integrated in the AGE environment [103]. It provides powerful query facilities, in addition to a modularity mechanism, called phasing, that has facilitated the implementation and modularisation of the solution [104].

5.9.4 Forward engineering

Restructuring and forward engineering tasks are made possible with the CUI model obtained in the reverse engineering step. As part of our prototype we have implemented a generator from CUI models to Java Swing, using the Textplate code-generation language integrated into the AGE environment.

The transformation is relatively straightforward, since the CUI model represents the layout information explicitly. This also allows the original legacy GUI to be recreated using features that are only available in the target technology. For instance, the proportion of the space that is occupied by a widget or layout is used to generate resizable windows that preserve the original

proportions. When a window is resized, its content (i.e. the widgets) must be resized accordingly. However we do not wish to resize all the widgets, but only those widgets that can contain or display more information if they are resized. Therefore, some widgets will be resized while others will maintain a fixed size. Widgets such as *JLabels* and *JButtons* will have a fixed size, which will be their preferred size. Widgets such as *JTextField* or *JTable* will have a variable size, which will depend on the size of the window.
It is interesting to note that further advantage can be taken of the information expressed explicitly in the CUI in order to generate a GUI in the most suitable manner for a target technology. For example, in our CUI it is possible to represent some relationships among widgets (think, for example, of a widget and its associated label) by means of the association between *WidgetConnection*s in the CUI metamodel presented in Figure 5.4. This information can be used to define gaps between pairs of widgets, and the gap between related widgets can be made narrower than the gap between unrelated widgets.

5.10 Conclusions

In this chapter we have presented a first approach to recover the implicit layout of the GUIs of RAD applications, whose layout is implicitly represented by widget positioning. The work has been focused on legacy applications created with RAD environments, though it could be easily adapted to other environments sharing the same features. Actually, the layout inference approach can be applied to non-legacy environments, for example to generate final GUIs from views created with wireframing tools, as we will show in the next chapter. As a result, a framework for migrating GUIs implemented with legacy environments has been built. This framework has been evaluated with two real-world Oracle Forms applications by migrating a sum of 164 windows to Java Swing.
The results evidence that the solution devised can be used to infer the layout of the applications to a great extent. However, the algorithms fail to accurately detect the layout when the window is not arranged in parts (surrounded by a border) or the widgets are placed in such a way that their structure cannot be easily described by a common layout. In these cases, manual tuning of the CUI model would be required. Fortunately, the windows of RAD applications often follow some style patterns that contribute to make the GUI more comprehensible and usable, and for this reason our approach succeeds in most of the cases, as it has been shown.
Although the *Non-regular layout detection* problem may not be crucial for RAD applications be-

cause simpler layout patterns are commonly found, it is considerably important to deal with layout inference of views in general. For example, a view like Figure 5.30 is fairly common in current desktop applications, but our approach fails to detect the layout because it does not exactly fit any of the predefined layout types. However, that layout could be described by a composition of *FlowLayout* and *StackLayout*. Therefore, in order to widen the scope of our approach, the layout inference algoritm should be able to recognise layout patterns that are nested in other patterns (layout composition).

The proposed architecture satisfies some of the requirements stated in Section 4.1. Firstly, explicit information about the layout of the graphical elements in the user interface is condensed in CUI models (requirement *R1*), thus allowing automatic restructuring and forward engineering processes and tools to be applied to these models.

Secondly, resolving the layout abstraction by means of a transformation chain allows the problem solution (algorithms) to be split into smaller modules (transformations) which can be developed and evolved independently (requirement *R2*), since the metamodels act like the contracts of the modules. The transformations can be chained and executed sequentially to achieve automation, signifying that the most complex part of the process (the generation of the CUI models from the Normalised models) is performed automatically (requirement *R3*). It can be observed that the application of the MDE principles results in a more maintainable solution.

Thirdly, as we have a Normalised model as an input of the reverse engineering process and a CUI model as an output, we achieve source and target platform independence (requirement *R4*). Thus, only the corresponding injector plus the transformation to derive Normalised models are required to support a new legacy technology. Note that the reverse engineering algorithms can be applied independently of the source and target technologies. Reusability and extensibility are thus also promoted in our approach.

Fourthly, matching the visual and logical structure (requirement *R5*) has been achieved by addressing challenges *L1* and *L2*. The model transformation from the Normalised to the Region model is in charge of dealing with these challenges.

Fifthly, the CUI model contains the high-level layout representation that fulfils the requirement *R6*. With the purpose of getting this representation, the positions of the widgets have been turned into spatial relations among them (challenge *L4*), information about spacing and alignment has been gathered in relative units (challenge *L5*) and the structure that widgets form has been recognised and abstracted in high-level layout types (challenge *L3*).

Lastly, misalignment tolerance (requirement *R7*) is achieved by means of the *sharing* concept, that introduces a tolerance margin to let the comparison of tiles be flexible. The current implementation does not cover the generation of different alternative solutions (requirement *R8*) and the inference algorithm has not been designed to support a configurable layout set (requirement *R9*).

Some similarities between our work and the works ([3] [4] [5]) about layout inference that were presented in Section 3.1 can be found. Concretely, like in [5] we created a model-based architecture to make the approach independent of any source and target technology (requirement *R4*), and we both dealt with the problem of matching the visual and logical structure of the views (requirement *R5*). In [3], the author detected the problem of small misalignment, which has been taken into account in our solution too (requirement *R7*). Similar to them, our approach needs to be manually tweaked in many cases, particularly when the heuristics fail in detecting a complex layout, which makes the approach semi-automated.

On the other hand, all these three related works are designed to represent the layout with only a specific layout type. The [3] and [5] approaches generate code for a fixed layout manager, and in [4], user can select the target layout manager, but different layout managers cannot be used at the same time. Different from them, our approach deals not with a single layout manager, but a set of them. However, in our approach we do not cover the *R9* requirement since as it is possible to indicate a subset of layout managers to use from the layout set available. In the next table we classify our solution as we did with the state of the art.

Tested source technology	Oracle Forms
Tested target technology	Java Swing
Source/target independence	Yes
Information extracted	layout composition, alignment, margins, holes, sizes
Layout model	Layout metamodel
Algorithm type	Heuristics
Implementation technology	MDE
Automation degree	Automated

Table 5.4: Classification of the approach of this chapter

It is only with the heart that one can see rightly; what is essential is invisible to the eye.

Antoine de Saint-Exupéry, The Little Prince
(Suggested by Jesús Sánchez)

6 Layout inference revisited: exploratory approach

In the previous chapter we tackled the layout inference of GUIs of legacy systems, particularly RAD-based applications. Although the results of the evaluation were satisfactory for these applications, the approach has some limitations, which were discussed in Section 5.10. The main drawback was that the heuristics cannot be composed and they are too simple to fit any layout. Moreover, it lacks of some features that would be desirable in a general solution for layout inference and which were introduced in Section 4.1, such as offering alternative solutions (requirement *R8*) and configuration of the layout set (requirement *R9*). To overcome these limitations we have devised a new version of the layout inference algorithm that replaces the last part of the layout inference process (the high-level structure detection based on heuristics) by a more complex algorithm that would be able to recognise the layout of any static view.

The CUI and Tile metamodels were also modified in order to improve the approach. The CUI model was split into two separate models (Structure and Layout models) in order to promote the separation of concerns and ease the evolution of each concern. Although the original Tile representation could have been reused, some changes were performed on the Tile metamodel in order to represent gaps in a more useful way and to facilitate the manipulation of Tile models

to the high-level layout inference algorithm.
Consequently, in this chapter we will show the changes we have made in the architecture introduced in the previous chapter (Figure 5.2) to acommodate the new requirements, and we will explain the data structures (metamodels) and the algorithm we have devised to cope with layout detection. We will finish with the case study for evaluating the approach and the conclusions drawn from the experience.
We will demonstrate that the layout inference process that we proposed for legacy applications can also be used in other scenarios such as the generation of the final GUI of a new system based on wireframes. To be more precise, our layout inference solution can now be reused with in any scenario in which we have files defining views in terms of coordinates.

6.1 MDE architecture for layout inference (revisited)

In the first approach to infer the layout of GUIs, the CUI metamodel integrated the Structure and Layout metamodels introduced in Section 4.2.1. When designing the second approach, we realised that a separation of the different aspects of the GUI would favour the evolution of these metamodels. As a result, the schema depicted in Figure 5.2 was modified and turned into Figure 6.1. We have followed the same notation as in the previous chapter and the elements involved in the layout inference part have been highlighted.

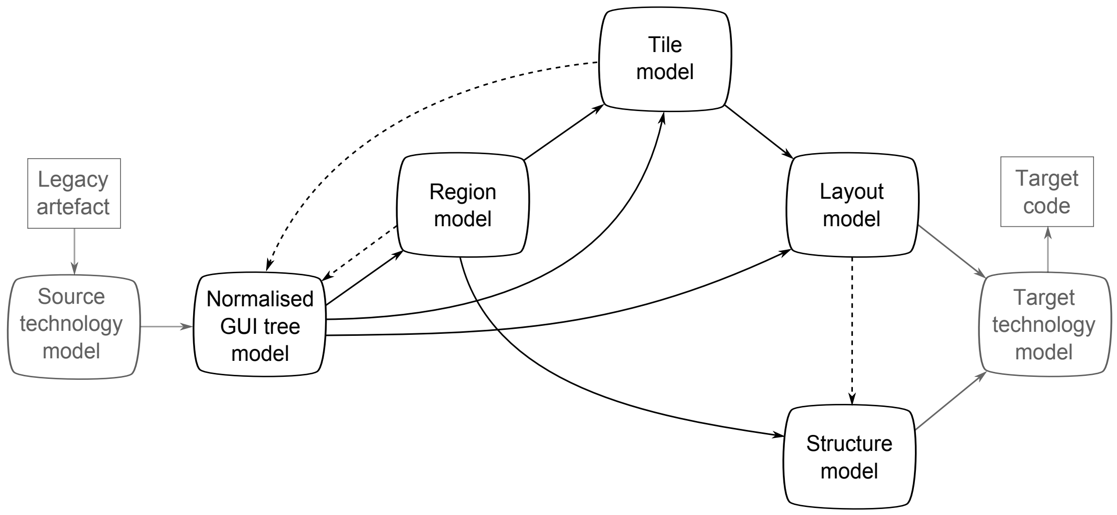


Figure 6.1: Model-based architecture used to migrate legacy GUIs.

Basically, the separation between the Structure and Layout models entails that the Structure model is derived from the Region model and the Normalised model (these models were described in the previous chapter). The Trace model, which has been omitted in Figure 6.1 for clarity reasons, is also generated at the same time that the Structure model. The Layout model now contains references to the Structure model, and in fact, the traces stored in the Trace model are queried in order to establish those links between the Layout model and the Structure model.

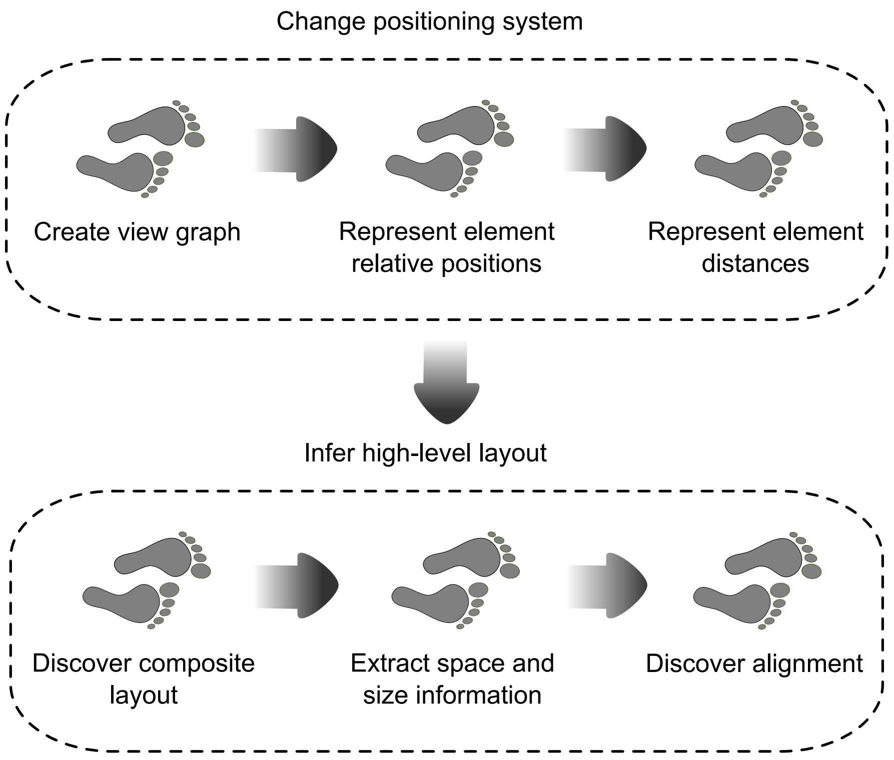

Change positioning system
Create view graph
Represent element relative positions
Represent element distances
Infer high-level layout
Discover composite layout
Extract space and size information
Discover alignment

Figure 6.2: Steps to explicitly infer the layout information.

The layout inference process, which generates a Layout model from a Structure model, consists of two main stages that are composed of three steps each one, which have been depicted in Figure 6.2. The first stage changes the positioning system from coordinates to relative positions between the elements. It corresponds to the uncovering of relative positions of the first version, which was explained in Section 5.5, and it keeps the basics of Tile models, though in the new version some information has been represented in a different fashion, notably relative positions are described with Allen intervals. The outcome of the first stage is a graph representation of the view with relative positions (i.e., a Tile model). The second stage takes that graph and applies a pattern matching algorithm that reduces the graph in every iteration (the matched nodes are replaced by a single node). When all the nodes will be matched then we will have obtained a tree of layout managers (the Layout model that was presented in Figure 6.5). This second stage corresponds to the high-level layout inference of the first version, which was explained in Section 5.6, being this new version more sophisticated than the former version. These two stages will be detailed in Sections 6.3 and 6.4, after describing the input and output metamodels of that process.

6.2 Reverse engineering metamodels

Next we will present the Structure and Layout metamodels, which determine respectively the input and output of the reverse engineering process. In Figure 6.3 we have demarcated the parts of the CUI metamodel described in Section5.2 that correspond to the Structure and Layout metamodels presented in this chapter. A Structure model is the input to generate a Tile model, and the output is a Layout model which is connected to the original Structure model. We will also explain how a Structure model is generated from the Normalised and Region models. The Structure and Layout metamodels are not just the result of splitting the CUI metamodel into two metamodels, but they have been redesigned to better meet the essentials of these metamodels.

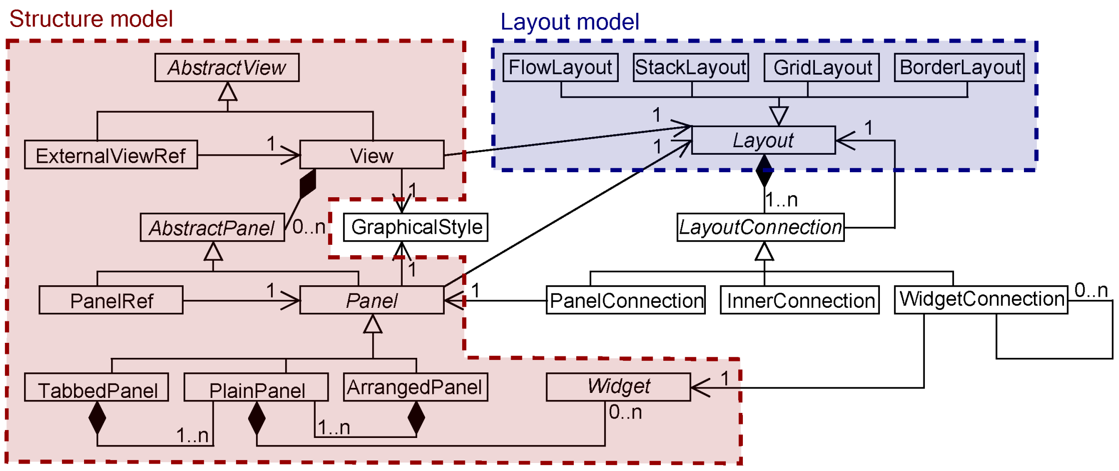


Figure 6.3: Relation between the CUI and the Structure and Layout metamodels.

6.2.1 Structure metamodel

The Structure metamodel has been devised to clearly represent the logical hierarchical structure of the views (the GUI tree) in a technology-independent fashion. A significant excerpt of the Structure metamodel can be seen in Figure 6.4. Attributes and role names can be easily guessed by the reader so they have been omitted in the metamodel to make it clearer.

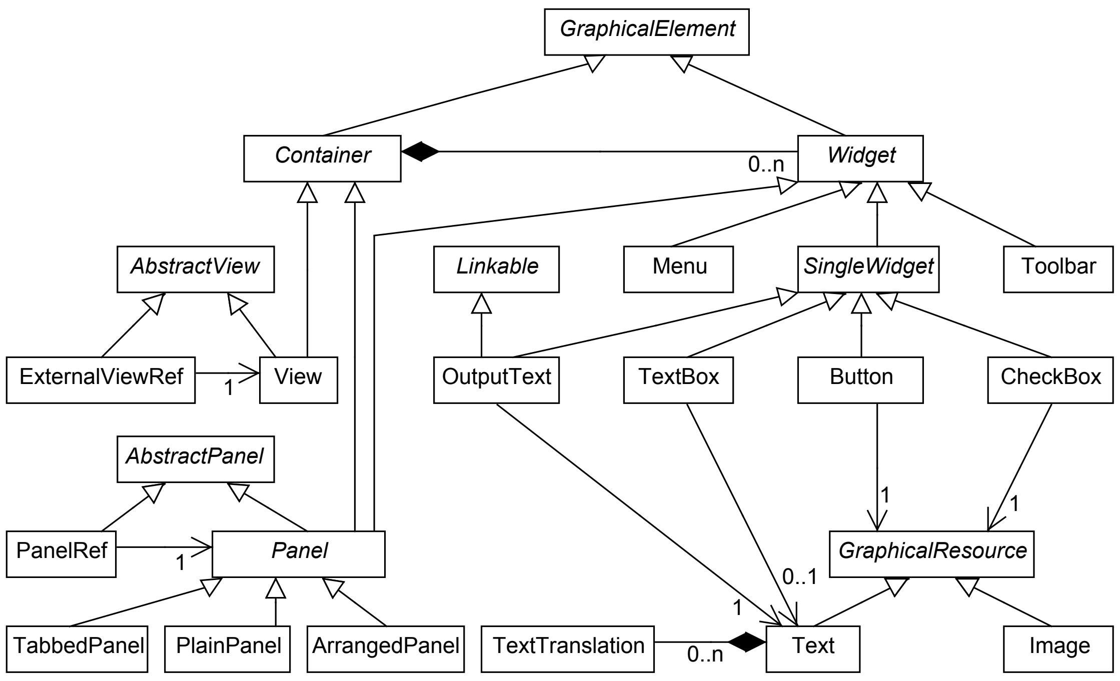


Figure 6.4: Structure metamodel.

According to our metamodel, a GUI is composed of a set of *Views*, such as mobile phone views, desktop windows or web pages. *View*s are composed of *Widget*s. A *Panel* is a special *Widget* that represents a visually distinguished part of the view. For example, a set of widgets surrounded by a border form a *PlainPanel*. It can be seen that *View*s and *Panel*s are *Container*s and that *Widget* and *Container* inherit from *GraphicalElement*.

Three types of *Widgets* are supported: *SingleWidget, Menu* and *Toolbar* (the elements that compose the two latter have been omited). There are many kinds of *SingleWidgets,* such as *TextBox, Button* or *CheckBox* each one having different features. For example *TextBox* has a *Text, Button* has a *GraphicalResource* (the graphics that are displayed on a button), and an *OutputText* can be *Linkable* (i.e., it is a hyperlink). For *Text* resources, internationalisation and localisation information (*TextTranslation*) can be provided. There are some constraints imposed on this metamodel, namely: i) *Menus* and *Toolbars* are restricted to be associated with *Views*, ii) *TabbedPanels*, *ArrangedPanels* can only contain *Panels*, iii) PlainPanels must only contain *SingleWidgets*, and iv) *Views* cannot nest *Panels* and *SingleWidgets* at the same level (this contraint has been propagated from the Region model).

6.2.2 Layout metamodel

The *Layout metamodel* defines the design of the views, that is, how the widgets are spatially arranged in the views. This design is expressed in terms of high-level constructions (particularly, layout managers) similar to Java Swing layout managers, which is a better representation system than others such as coordinates or positioning based on boxes (such as HTML). The Layout model can be used to derive a good quality GUI. This model conforms to the metamodel that can be seen in Figure 6.5.

In this metamodel, *LayoutElements* can be *ElementNodes* or *Layouts*. An *ElementNode* represents a *Widget* (either *Container* or *SingleWidget*) that is managed by a *Layout*. Actually, the *refNode* reference indirectly connects an *ElementNode* with the associated *Widget* in the Structure model.

Layouts are arranged hierarchically, so the layout of an element can be a composition of layouts. The set of predefined layouts currently supported is: *FlowLayout*, *BorderLayout*, *GridLayout*, *FormLayout* and *CustomLayout*. A *FlowLayout* is a horizontal flow or a vertical flow depending on the *type* attribute. A *BorderLayout* is a layout that places the content in five areas: *top*, *bottom*, *left*, *right*, *center*. Not all the five areas of the *BorderLayout* have to be occupied. A *GridLayout* arranges the elements in a grid of $numRows \times numCols$ cells of equal size (only the *numCols* attribute is stored). A *FormLayout* is a more complex layout that is applied to rows of elements where some of the elements are vertically aligned more or less in the same way. This layout contains (*rows* reference) a list of vertical *FlowLayouts*. In addition, it defines some *AlignedColumns* in such a way that every element must belong to just one *AlignedColumn*. There are three references from *AlignedColumn* to *ElementNode*: *lnodes* represents the elements aligned to the left bound of the column, *rnodes* is the same for the right bound and *nodes* include all the elements contained in the column (aligned or not). *CustomLayout* is used in case the GUI layout cannot be composed by a composition of the predefined layouts.

A *Layout* also includes some other attributes that are useful to tune the design determined by

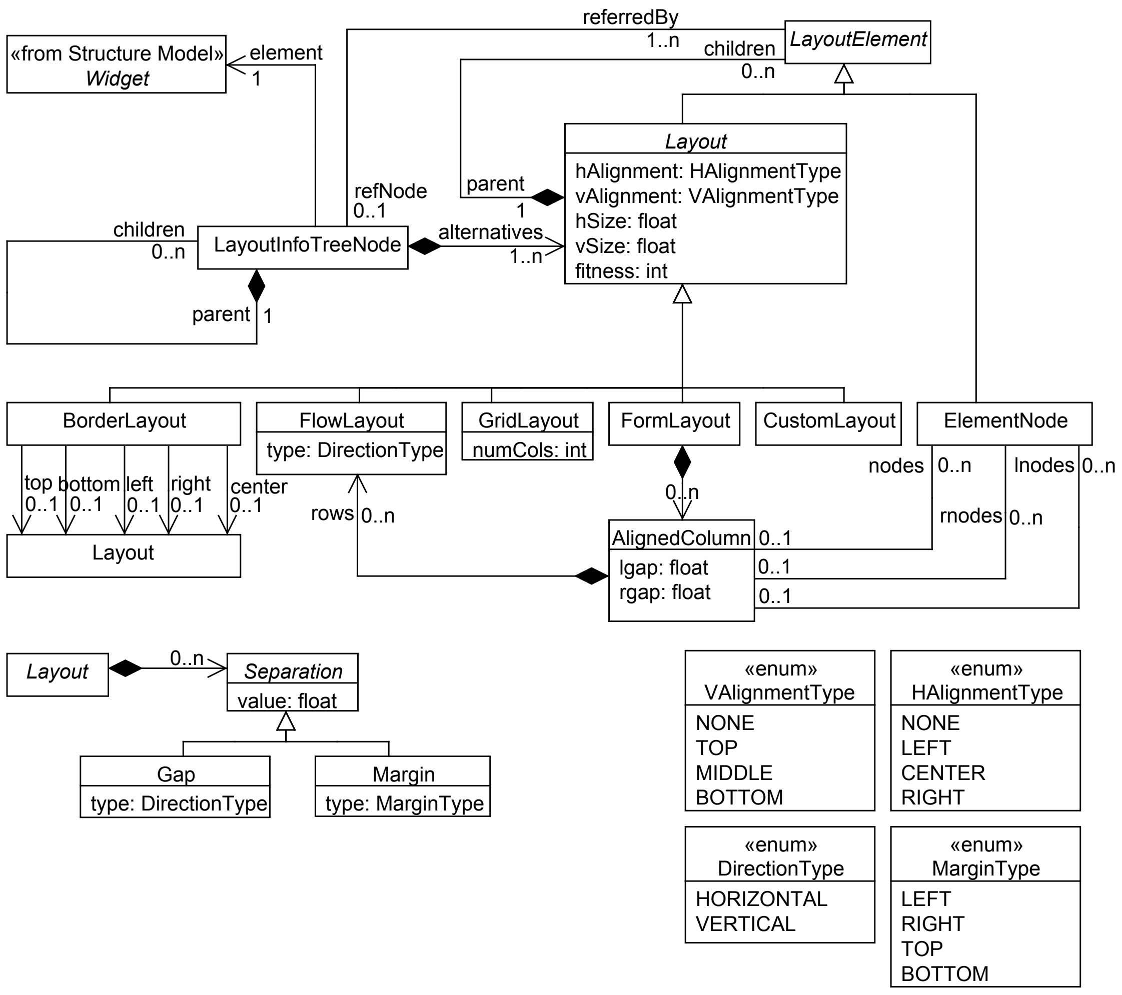


Figure 6.5: Layout metamodel.

the predefined set of layouts. The *hAlignment* and *vAlignment* attributes are used to indicate how a layout is horizontally and vertically aligned. We have two different cases: i) when the *Layout* is associated with a *Container*, the alignment is relative to that Container; ii) when the *Layout* is part of a more complex layout (it is nested in another *Layout*), then the alignment is relative to the enclosing *Layout*. We refer to a *Layout* that is part of a complex *Layout* as *intermediate layout*.

The *hSize* and *vSize* attributes are the percentages of the horizontal and vertical space (respectively) taken by the element, with regard to the *Container*. Adjacent elements are commonly separated by horizontal or vertical *Gaps* (empty space), and *Margins* represent the distance of a layout to the bounds of the *Container*. It is worth remarking that *Margins* are only applica-

ble to the *Layouts* associated with a *Container*, not to the children *Layouts* of these ones. For both, *Gaps* and *Margins*, the distances are measured with percentages of horizontal or vertical distances with regard to the *Container*.
Each *Widget* of the Structure model is reproduced in the Layout model by a *LayoutInfoTreeNode* which contains the reference to it (*element* reference). There may be different visual structure compositions that can result in a similar layout perceived by users. Therefore, there may be different layout alternatives to lay out the same *Container*, which means that every *LayoutInfoTreeNode* that is associated to a *Container* will have a set of possible layouts (*alternatives*). Each *Layout* contains a reference (*refNode*) to the *LayoutInfoTreeNode* to which the layout is applied, except for intermediate layouts that are not linked to any *Widget*.
Each *Layout* includes the *fitness* attribute that serves to compare the alternatives among them and know which ones are better than others. The *fitness* attribute takes a value between 0 and 1. The closest to 1, the better a solution is. Fitness values are meant to be used to compare different alternatives for the same *Container*, and should not be used to compare different *Container* since these values are arbitrary.

6.3 Changing the positioning system

As an intermediate step in the transition from absolute coordinates to a layout representation using layout managers, we use a representation of the GUI using a relative positioning system based on the spatial relations among the widgets, i.e., the *Tile metamodel*. This representation is the basis for the layout inference algorithm. We have performed some changes in the original Tile metamodel presented in Section 6.3.1 with the aim of adapting the data structure to the new layout inference algorithm, and to improve the representation of the distances between the elements.
The creation of the Tile metamodel (step 1) will be presented in Section 6.3.1, the relative positioning (step 2) will be explained in Section 6.3.2, and Section 6.3.3 will delve into the representation of the distance between elements (step 3).

6.3.1 Creating the view graph

The means we propose to represent a view is a nested, attributed, relational acyclic directed graph, that is, a digraph without cycles where the nodes can be digraphs and the nodes as well as the edges have attributes. The data structure that defines these graphs is the metamodel presented in Figure 6.6. This representation is focused on making positions between elements explicit, which is very useful to detect layout patterns.

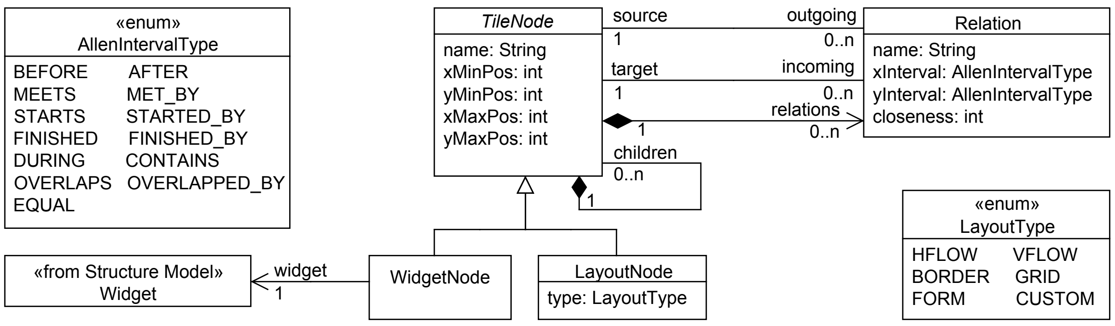


Figure 6.6: Tile metamodel (new version)

The model contains two main classes, *TileNode* and *Relation*. A *TileNode* represents a rectangular area of a view that contains a widget or a group of widgets. As can be seen, *TileNodes* contain information about the coordinates of the area they take. Although our inference algorithm is applied to the Tile model as a graph, coordinates are still required for calculating some attributes such as margins of a widget with respect to the container.

A *WidgetNode* is a *TileNode* that represents the area of a widget (either *SingleWidget* or *Container*) and contains the reference to that *Widget* in the Structure model. A *LayoutNode* is a *TileNode* that represents the area of a group of widgets that are laid out by using a certain layout *type*. As we will see later, at the beginning of the layout inference process we have a graph that only contains *WidgetNodes* and at the end of the process (after applying rewriting) we have a graph composed of *WidgetNodes* and *LayoutNodes*. From now on, we will indistinctly refer to *TileNodes* as *tiles* or *nodes*.

Relation represents a spatial relation between two *TileNodes* (*source* and *target*) by means of three attributes. The first attribute is the Allen interval for the X-axis (*xInterval*), the second attribute is the Allen interval for the Y-axis (*yInterval*) and the third parameter is the closeness level between a pair of connected *TileNodes* (*closeness*).

The Allen intervals [9] can be used to express the spatial relations for a pair of segments in one dimension. They provide us with two interesting pieces of information. Firstly, they serve us to represent the relative positioning of nodes (e.g. if a node is on the right or below another node). Secondly, they also capture the alignment of one node with respect to another one.

Allen intervals will be explained in detail in Section 6.3.2.
To represent distances between widgets we do not use absolute distances measured in pixels, but we calculate the so called *closeness level*, which is a means to classify widget distances in groups. The distances that belong to the same group are more or less similar. This will be used by our algoritms to prioritise close widgets over farther widgets. Please note that in the Layout model one attribute is enough to represent this feature (*closeness*), because we do not allow widget overlapping, and thus it measures the distance in only one axis according to the Allen intervals. We will discuss more about the meaning of the closeness levels in Section 6.3.3.
The Tile model is created as follows. A *WidgetNode* is created for each *Widget* of the Structure model. *WidgetNodes* keep a reference to the *Widget* from which they are created, and a copy of the coordinates of the element. The containment hierarchy of the Structure model is therefore replicated when creating the Tile model. For instance, if there is a *View* which aggregates three *children Panels* in the Structure model, then there will be a *WidgetNode* containing three *children WidgetNodes* in the Tile model. For each pair of adjacent *WidgetNodes*, a *Relation* is created. Let us recall a definition of the previous chapter: A tile t_1 is *adjacent* to another tile t_2 if and only if i) the projection on the X-axis or Y-axis of both tiles is overlapped and ii) there not exists a tile t_3 between t_1 and t_2. For each *Relation* created, the Allen intervals for the X (*xInterval*) and Y (*yInterval*) axis are calculated, and the a closeness level (*closeness* attribute) is assigned to it.

6.3.2 Representing widget relative positions

In this section we explain how the Allen interval algebra [9] has been used to express the relative positions between the widgets and how they are obtained.
Figure 6.7 shows all the intervals and their meaning. For example, if *A MEETS B* it means that the segment A is before the segment B and the end of A 'touches' the beginning of B (i.e., there is no blank space between them). Note that all the intervals (except for *EQUALS*) have an opossite interval, e.g., if *A MEETS B* it implies that *B MET_BY A*. In order to represent the spatial relations of 2D objects we need two intervals, one interval for the projection of the node on the X-axis and another one for the projection of the node on the Y-axis.
Note that *Relations* in Tile models are directed, i.e., they distinguish between the source the target node of the *Relation*. Given that the Allen intervals are defined on ordered pairs of segments, the pair *(source, target)* indicates how to interpret the intervals. For instance, let $r(t_1, t_2)$ be a relation between *TileNodes* t_1 and t_2 and $r.xInterval = Before$ means that t_1 is before t_2 with

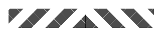
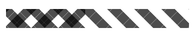
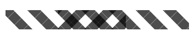
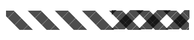
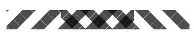
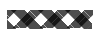
Allen interval	Meaning	Opposite
BEFORE		AFTER
MEETS		MET_BY
STARTS		STARTED_BY
DURING		CONTAINS
FINISHES		FINISHED_BY
OVERLAPS		OVERLAPPED_BY
EQUALS		-

Figure 6.7: Allen intervals

regard to the projections on the X-axis. The pairs of *TileNodes* in a *Relation* are ordered in the following way: the target *TileNode* is always on the right or below the source *TileNode*. As it was already said, *TileNodes* are arranged in a hierarchy so a *TileNode* can contain some other *TileNodes* and *Relations*.

Each *Relation* has then two Allen intervals: An Allen interval for the X axis (*xInterval*) that is based on the comparison between the y-coordinates of both *TileNodes*, and an Allen interval for the Y axis (*yInterval*) that is based on the comparison of the x-coordinates. As already indicated, one Allen interval allows representing relative positions between pairs of segments (1 dimension). Then, with two Allen intervals it is possible to represent the relative position of two widgets (represented as boxes) in a bidimensional space. The comparisons of the positions to calculate the Allen intervals are carried out with some margin m that is parameterised, so for a pair of widgets w_1 and w_2, $w_1.x = w_2.x$ if $w_1.x \in (w_2.x - m, w_2.x + m)$. By default the margin of the comparisons has been set to 10 pixels. It allows avoiding the negative effect of misalignment, which results in some flexibility when placing widgets onto the canvas for creating quick GUIs.

Figure 6.8 shows the Allen intervals for the *Relation* between the *passwordField* and *cancel* tiles extracted from the view example in Figure 6.11. Given that the projection of *passwordField* in the Y-axis precedes the projection of *cancel*, then the *yInterval* is *Before*. Regarding the X-axis, the projection of *cancel* exceeds the end of *passwordField* in 10 pixels. If we were strict (the comparison margin would be set in 0 pixels), the Allen interval that describes the relative position

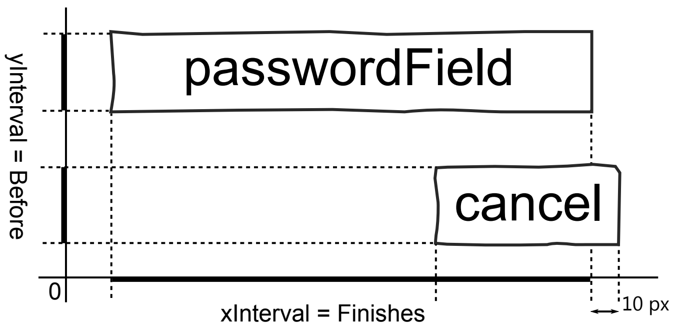


Figure 6.8: Allen interval example for a pair of widgets

of both projections would be *Overlaps*. However, as far as we have set the comparison margin to 10 pixels, the excess is not significant, so both projections can be considered as they end at the same point. Therefore, in this case *xInterval* is *Finishes*.

6.3.3 Representing widget distances

This is the step 3 in Figure 6.2. The closeness levels provide *Relations* with meaningful distances. The levels are obtained by taking the distance measured in pixels and mapping it to a finite set of values (levels, like 1 for very close, 2 for close and so on).

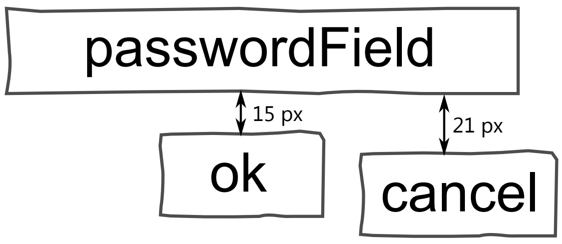


Figure 6.9: Problem when setting fixed limits for the closeness levels.

A simple way to classify distances in levels is to set fixed ranges. For instance, Figure 6.9 shows a portion of a view with three widgets (extracted from Figure 6.11). If we establish that a distance between 1 and 20 pixels is mapped to a *Very_Close* level, and a distance between 21 and 40 pixels is mapped to *Close*, then we would have that the closeness level between *passwordField* and *ok* (*Very_Close*) would be different to the closeness level between *passwordField* and *cancel* (*Close*). Since the diference of distances between *passwordField-ok* and *passwordField-cancel* is not significantly different when a user sees the view, then they should be tagged with similar levels.

As it can be deduced from the example, the classification of the distances should not be accomplished using tight limits (absolute distances) but variable limits that depend on the data set. In this sense, a group of nodes that are more or less at the same distance should always be in the same group, and the closeness level defines a partitioning of the nodes in groups according to the distance.

In order to address this shortcoming we apply a clustering algorithm, which performs a dynamic partition of the set of distances in the view. The partitioning of the distances is then used to classify the relations. Algorithm 4 details this process. We will use the simple example shown in Figure 6.10(a) during the explanation of the algorithm, which shows four widgets, a, b, c, d and the horizontal distances between them.

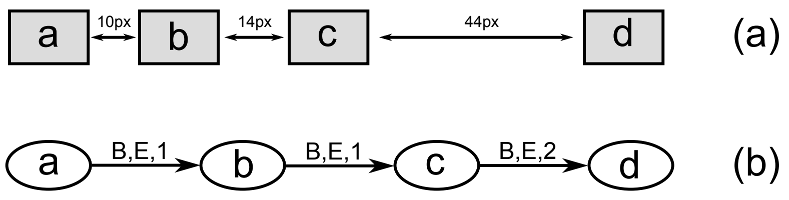


Figure 6.10: Closeness assignment example. (a) Widgets and distances between them. (b) Result graph.

Firstly the algorithm gets all the distances (vertical and horizontal) of the relations and creates a single cluster with these distances (lines 2 to 5). In this case, $BestPartition = \{\{10, 14, 44\}\}$. We use the population standard deviation (σ) of the distances to measure whether the cluster is homogeneous enough, i.e. the distances in the cluster can be considered similar (to a certain degree). If the standard deviation of the initial cluster is greater than the maximum closeness cluster deviation (*maxDev*, which by default is 15), then we have to split the cluster (line 7). In the example, $\sigma = 15.17 > maxDev$, so we have to split the distances in clusters.

In order to perform the clustering of distances, we have selected the k-means algorithm [105] (line 12), with the euclidean distance as similarity function. Given that k-means is a divisive algorithm, the number of clusters must be passed as a parameter. However, we do not know the number of clusters a priori. Therefore, we apply the k-means algorithm several times (lines 9 to 20), increasing the number of clusters in each iteration (line 10) until the stop condition. This condition is that the standard deviation of every cluster is less than *maxDev* (line 17).

Algorithm 4 Closeness assignment algorithm

```
 1: procedure AssignCloseness($Relations, maxDev$)
 2:     $AllDistances \leftarrow getAllDistances(Relations)$
 3:     $nClusters \leftarrow 1$
 4:     $Cluster \leftarrow AllDistances$
 5:     $BestPartition \leftarrow \{Cluster\}$
 6:
 7:     if $\sigma_{Cluster} > maxDev$ then
 8:         $partitionOK \leftarrow false$
 9:         while $\neg partitionOK$ do
10:             $nClusters \leftarrow nClusters + 1$
11:             for $i \leftarrow 1, Num\_Iterations$ do
12:                 $Clusters \leftarrow kMeans(AllDistances, nClusters)$
13:                 if $isBestPartition(Clusters, BestPartition,$
                            $SumOfSquaredErrors())$ then
14:                     $BestPartition \leftarrow Clusters$
15:                 end if
16:             end for
17:             if $\forall C \in BestPartition, \sigma_C \leq maxDev$ then
18:                 $partitionOK \leftarrow true$
19:             end if
20:         end while
21:     end if
22:
23:     $SortedPartition \leftarrow sort(BestPartition)$
24:     $closeness \leftarrow 1$
25:     $PartitionMap \leftarrow \{\}$
26:     for all $Cluster \in SortedPartition$ do
27:         $range \leftarrow getRange(Cluster)$
28:         $PartitionMap[range] \leftarrow closeness$
29:         $closeness \leftarrow closeness + 1$
30:     end for
31:
32:     for all $relation \in Relations$ do
33:         $d \leftarrow getDistance(relation)$
34:         $relation.closeness \leftarrow PartitionMap[d]$
35:     end for
36: end procedure
```

Because k-means is a heuristic algorithm, it is very fast, but it could fall into a local maximum. In order to get a better clustering, the algorithm is executed multiple times (lines 11 to 16) with different random starting conditions. By default the number of iterations, *Num_Iterations* variable, is 20, and we keep the best solution according to the intra-cluster homogeneity criterion, which is the sum of the squared errors (line 13). Following with the example, the k-means algorithm is applied with $nClusters = 2$ and the output is: $BestPartition = \{\{10, 14\}, \{44\}\}$, and we have that $\sigma_{\{10,14\}} = 2 < maxDev \wedge \sigma_{\{44\}} = 0 < maxDev$, so the clustering loop stops.
After obtaining the clusters, we sort the clusters and we assign a numerical tag to each one (lines 23 to 30). The lesser the values (distances) of the cluster, the lesser the numerical value of the tag (the lower distance group is tagged with *1*). For each cluster, we set a minimum and a maximum value in pixels (lines 27). In the example, *PartitionMap* maps each range to a closeness level: $(-\infty, 14] = 1$ and $[15, +\infty) = 2$.
When all this process has been accomplished, we iterate over the *Relations* and for each one and we use *PartitionMap* to know which closeness level must be assigned to the *Relation* by comparing the distance with the ranges. Figure 6.10(b), shows the Tile model fragment of the widgets in Figure 6.10(a), being the closeness level the numeric parameter of the edges (*Relations*).

6.3.4 Tile model example

The graph in Figure 6.12 is the Tile model derived from the example window shown in Figure 6.11. Tiles (nodes) have been represented with ellipses that include the name of the widget and relations (edges) have been represented with arrows with three attributes: the Allen interval for the X axis (*xAllenInterval*), the Allen interval for the Y axis (*yAllenInterval*), and the *closenessLevel*. The coordinates and dimension of the nodes have been omitted.

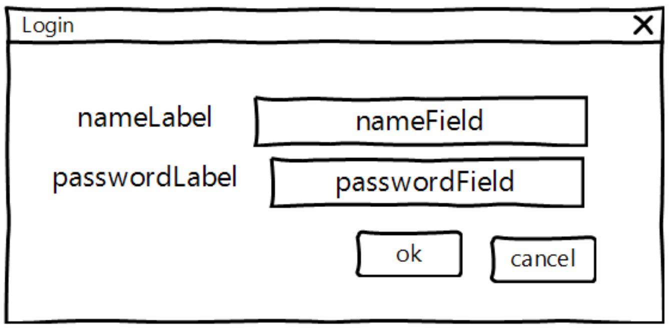


Figure 6.11: Login window created with WireframeSketcher.

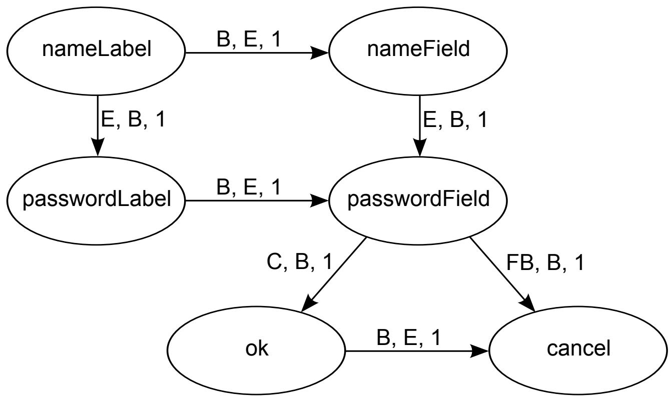


Figure 6.12: Graph representation of the login window example. *B=BEFORE, E=EQUALS, C=CONTAINS, FB=FINISHED_BY*

Since all the distances between the nodes are more or less similar, the clustering algorithm groups all the distances in just one cluster. This unique group is assigned the closeness level 1. As we mentioned in Section 6.3.2, the comparisons of the positions take into account some margin. This is the reason why the *xAllenInterval* of the relation between *nameField* and *passwordField* is *EQUALS* though the projection of the coordinates in the X axis for both widgets is not exactly the same.

6.4 Infering a high-level layout

The Tile model is the basis to apply the layout inference algorithm we have devised (steps 4, 5 and 6 in Figure 6.2 are encompassed by this algorithm). It is a backtracking algorithm based on graph rewriting. The main idea consists of matching a predefined set of layout patterns against the graph (the Tile model) until all the nodes have been matched and replaced by the corresponding layout node, so rewriting finishes when there is only one node left (the root layout). The layout patterns are applied in all the possible orders, so obtaining several solutions that are evaluated in order to see how good or bad the solution is.

The algorithm generates all the possible permutations of the layout patterns and checks for each sequence if we can meet a solution by applying the layout patterns in the order specified by the sequence. Every time there is a pattern match, all the matched nodes are replaced just by one node, and the pattern matching continues with the resultant reduced graph. A solution sequence is a composition of layouts that covers all the nodes of the graph, this is, when only one node remains. Each different solution that is found is assessed by a fitness function. The best solution will be the solution with the highest fitness value.

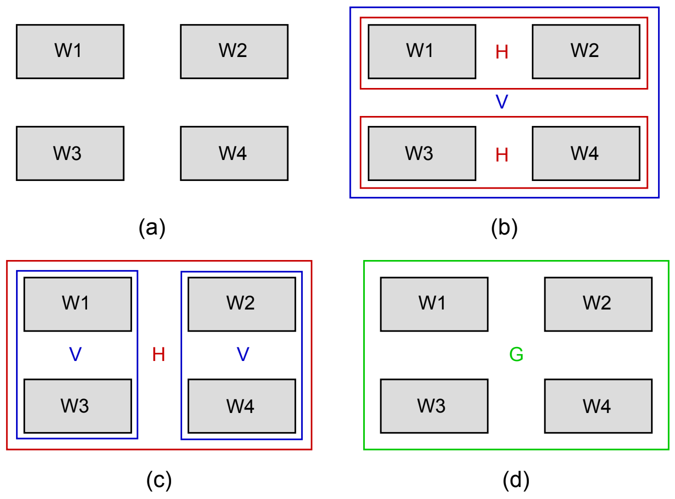


Figure 6.13: Pattern matching example on four widgets

In order to show the logic behind the algorithm with a simple example, let us suppose we have four widgets $W1$, $W2$, $W3$, $W4$ that are spatially distributed as it is shown in Figure 6.13(a). There are several possibilities to arrange these widgets depending on the order in which the layout patterns are applied. If we start looking for horizontal flows of elements, we find two matches: *W1*-*W2* and *W3*-*W4*. Then, a vertical flow composed of the two previous matches can be applied (see Figure 6.13(b)). If we had started looking for vertical sequences, the pairs *W1*-*W3* and *W2*-*W4* would have matched, and then these two matches would make a horizontal matching (see Figure 6.13(c)). Finally, if we had looked for a grid pattern of 2×2 elements, all the nodes would have fit in just one match (see Figure 6.13(d)).
This approach has the advantage of offering a list of alternative solutions, which could be interesting to know different implementation options and choose the desired layout. Before explaining the algorithm in deep in Section 6.4.2, the following section describes the predefined set of layout patterns that can be used in the algorithm.

6.4.1 The layout patterns

In this section we describe the layout patterns used to detect the predefined layout types that were introduced in Section 6.2.2.

- **(Horizontal / Vertical)** ***FlowLayout***: selects a sequence of nodes that are connected by only one outgoing edge with the *xInterval* / *yInterval* equals to *BEFORE* or *MEETS*.

- ***BorderLayout***: looks for subgraphs that match the five areas of a star topology: top, bottom, left, right, center. Not all the five areas have to be identified. The currently supported patterns for the *BorderLayout* can be seen in Figure 6.14. In order to detect the areas, not only the edges of the graph are taken into account, but also the relative distances to the container. For the *top*, *bottom*, *left* and *right* areas, it must not be a distance lower than a certain value (15% by default) from the container bounds. When detecting a BorderLayout with only the top-bottom areas or left-right areas, the relative distance between the areas must be greater than a value (20% by default) regarding the container bounds.

- ***GridLayout***: searches recursively for subgraphs connected among them so they form a rectangular grid topology of $n \times m$ nodes. Firstly it attemps to match a 2×2 square (the smallest allowed grid). Then it tries to expand the rectangle by recursively matching the nodes to the right and below the square. In the end the match is the biggest rectangular grid that it is possible to match from the 2×2 square. There is a constraint that the nodes inside the grid cannot contain edges whose target node is outside the grid, only the border nodes of the grid are allowed to have connections to the nodes outside the grid. Additionally, for a *GridLayout* to be matched the closeness level of all the edges has to be the same.

- ***FormLayout***: it is a pattern devised to arrange *SingleWidgets*, not *Containers*. The pattern firstly detects a vertical *FlowLayout* composed of a list of (more than one) horizontal *FlowLayouts*. Secondly, it has to be checked that at least two of the elements are vertically aligned. The widgets shown in the example of Figure 6.11 match the *FormLayout*. This pattern searches for *alignment marks*, which are imaginary vertical lines to which some of the widgets are aligned. In the example, we have an alignment mark between *passwordLabel* and *nameField*, and another one on the right border of the *cancel* button. These alignment marks are used later to define the bounds of the *AlignedColumns* (see Layout model in Figure 6.5). For example, *nameLabel* and *passwordLabel* would form an *AlignedColumn*, and the rest of widgets would form another one. Not all the widget types are allowed in a *FormLayout*, but only widgets typically found in a form (e.g.,*ComboBoxes* and *CheckBoxes* are allowed, but not *ImageContainers*).

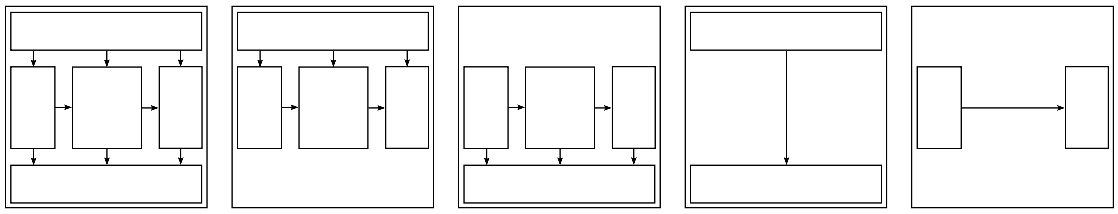


Figure 6.14: Border layout supported patterns.

There is another layout defined in the metamodel, the *CustomLayout*. As we indicated in Section 6.2.2, this is not actually a layout, but it means that no combination of the selected layout patterns can be applied to the original graph to reach a solution. For example, the distribution of widgets shown in Figure 6.15(a) and 6.15(b) does not fit any combination of the aforementioned layout patterns, so a *CustomLayout* will be generated in these cases. When a *CustomLayout* is obtained, developers are responsible for programming the layout by hand.

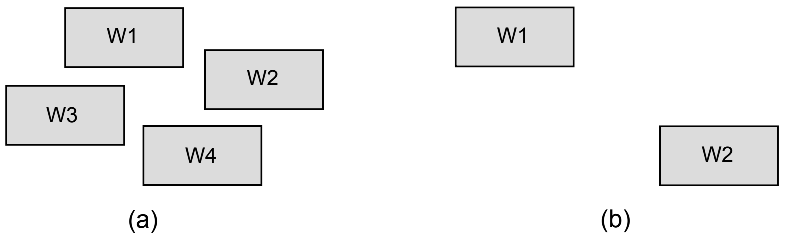


Figure 6.15: Examples of widgets that do not match any pattern

It can be seen that some patterns are more likely to be used when arranging containers, such as the *BorderLayout*, whereas other layouts such as the *FormLayout* are devised to work with single widgets. *FlowLayout* (vertical or horizontal) is the most general layout and can be used for both, containers and single widgets. *GridLayouts* can also work for both cases.

The patterns have not been defined by means of a graph grammar but they are hardcoded because some patterns (such as the *Grid* pattern) cannot be expressed by a context-free graph grammar, and therefore they cannot be easily managed by graph transformation tools.

6.4.2 Layout inference algorithm

In this section we present the algorithm to discover the structure of the layout (step 4 in Figure 6.2) in terms of the layout managers defined in the previous subsection by means of patterns. The algorithm generates an instance of the Layout model depicted in Figure 6.5. In Section 6.4.3 a complete example describing a step-by-step execution of the algorithm is given.
The layout inference is presented in Algorithm 5 (function *InferLayout*). The function is executed for every *WidgetNode* associated with a *Container*. It receives three inputs: a *WidgetNode* which is associated with a *Container* (*cNode*), the set of identifiers of the predefined layouts to use (*layoutSet*) and the number of closeness levels that appear in the relations of the graph (*nCLevels*). It is important to remark that a *WidgetNode* associated with a *Container* represents a graph, and contains the *WidgetNodes* included in that *Container*.

Global initialisation (lines 2 to 4)

The *solutions* set stores the different alternative graphs that steam from the rewriting process, and represents the possible visual structures of the container associated with *cNode*. The *solSequences* set is used to store the sequences of layout patterns that have been applied to obtain each solution stored in *solutions*. Each layout type is given an integer identifier, so that a sequence of applied patterns is just represented as a sequence of integers. Thus, the rationale of the *solSequences* set is to allow fast comparison of solutions, instead of comparing the solution graphs (graph isomorphism problem).
generatePermutations() in line 4 generates all the $n!$ possible permutations for the layout identifiers, being n the number of predefined layouts used.

Iterate over the permutations (lines 6 to 11)

The algorithm iterates over all the permutations (line 6) searching for solutions, so there will be at most $n!$ solutions for a graph (in practice there will be only a few solutions).
Some initialisations are carried out between lines 7 and 11. *currentSolSeq* is used to store the current solution sequence, and is initialised to an empty sequence. Since the algorithm needs to modify the graph represented by *cNode*, that graph is deeply cloned in each iteration (i.e., for each permutation), so the algorithm works on that copy. The graph will be reduced each time that there is a pattern match on a subgraph, that is, each subgraph that matches the pattern

Algorithm 5 Layout inference algorithm

```
1: function InferLayout($(cNode, LayoutSet, nCLevels)$): $Solutions$
2:     $Solutions \leftarrow \{\}$
3:     $SolSequences \leftarrow \{\}$
4:     $Permutations \leftarrow generatePermutations(LayoutSet)$
5:
6:     for all $permutation \in Permutations$ do
7:         $CurrentSolSeq \leftarrow \{\}$
8:         $graph \leftarrow copyGraph(cNode)$
9:         $closenessLimit \leftarrow 1$
10:        $remainingNodes \leftarrow graph.order$
11:        $loops \leftarrow 0$
12:
13:        while $remainingNodes > 1 \wedge$
                   $loops < LayoutSet.size * nCLevels$ do
14:            $pattern \leftarrow getNextLayoutPattern(permutation)$
15:            $Matches \leftarrow match(graph, pattern, closenessLimit)$
16:
17:            if $\neg isEmpty(Matches)$ then
18:                for all $match \in Matches$ do
19:                    $mergeNodes(graph, match, pattern)$
20:                end for
21:                $add(currentSolSeq, pattern)$
22:                $resetSequence(permutation)$
23:                $loops \leftarrow (closenessLevel - 1) * LayoutSet.size$
24:                $remainingNodes \leftarrow graph.order$
25:            else
26:                $loops \leftarrow loops + 1$
27:                if $loops \bmod LayoutSet.size = 0$ then
                       $closenessLimit \leftarrow closenessLimit + 1$
28:                end if
29:            end if
30:        end while
31:
32:        if $remainingNodes > 1$ then return 'No solution'
33:        else
34:            if $\neg contains(SolSequences, currentSolSeq)$ then
35:                $layout \leftarrow createLayout(graph)$
36:                if $\neg contains(Solutions, layout)$ then
37:                    $layout.fitness \leftarrow fitness(layout)$
38:                    $add(SolSequences, currentSolSeq)$
39:                    $add(Solutions, layout)$
40:                end if
41:            end if
42:        end if
43:
44:    end for
45: end function
```

is replaced by a layout node that contains the subgraph. The *closenessLimit* is initially set to the lowest level. *remainingNodes* is assigned the number of nodes of the graph (i.e. the graph *order*). *loops* indicates the number of loops without applying any pattern to the graph, i.e. loops without changes.

Matching patterns (lines 13 to 15)

The loop in line 13 is in charge of applying pattern matchings on the current graph in order to look for solutions. There are two conditions that must be satisfied to continue iterating. The first condition is that *remainingNodes* is greater than one. We have already said that the pattern matching engine progressively reduces the size of the graph, until the whole graph is transformed in a single layout node. Therefore, if *remainingNodes* is one it means that we have found a solution. Otherwise, not all the nodes have been matched and a solution has not been reached so far. The second condition keeps the loop running while there are pairs of (pattern, closenessLevel) that have not been tried (the number of loops without changes is equals to the number of predefined layouts used multiplied by the number of closeness levels used). This is the maximum number of iterations that are required to perform a pattern match. Consequently, it is a stop condition to avoid a infinite loop when no solution can be reached.
Given a permutation of layout types (*layoutSet*), *getNextLayoutPattern()* (line 14) iterates over the permutation and returns the next layout type, in such a way that when there are no more layout types left it restarts the cycle from the beginning. The invocation of the *pattern matching engine* is represented by the function *match()* (line 15).
The pattern matching engine iterates over the graph nodes looking for matches of a given pattern. The pattern is matched against the graph starting from every node (because the starting node of a match cannot be determined beforehand). That leads to match subgraphs that are contained in other matches. In that case in which there are nested matches, we keep the largest one (that nests the other submatches). Single-node matches are discarded.
The *closenessLimit* is included in the pattern matching engine call so only the edges with a closeness level equals or less than the limit can match a pattern (the rest of edges are ignored). Note that it makes a partition of the graph in connected components, so each connected component is a subgraph of the original graph where all the edges have a closeness level equals or less than the limit.

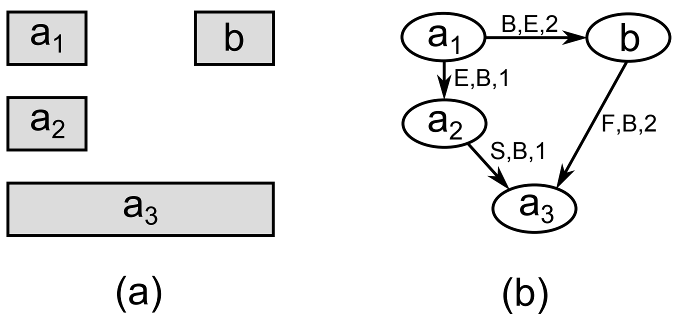


Figure 6.16: Example of non-valid match for the Vertical Flow Layout pattern.

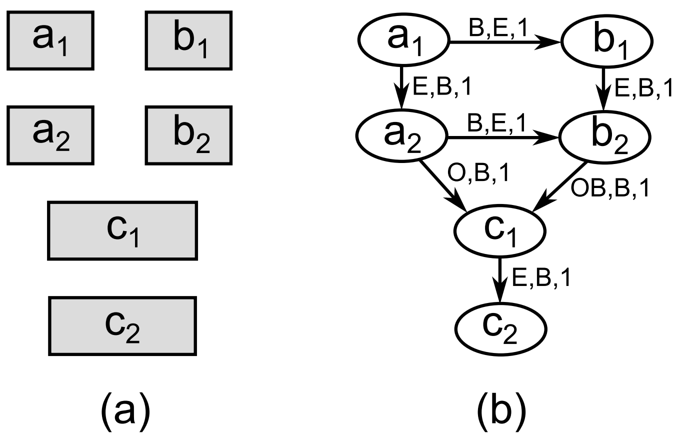


Figure 6.17: Example of match split for the Vertical Flow Layout pattern.

Not all the matches performed by the matching engine are valid. There are two constraints that must be ensured: i) the area delimited by the matched nodes does not enter the area occupied by other node outside the match, and ii) there are no nodes that are shared by different matches. To explain the first constraint, let us consider the graph in Figure 6.16(b) that corresponds to the layout of widgets represented in Figure 6.16(a). In this example, if the matching engine tries to match a vertical column of nodes (Vertical Flow Layout), it would perform the following match: $M_1 = \{a_1, a_2, a_3\}$. The edges $e_1(a_1, b)$ and $e_2(b, a_3)$ are tagged with level 2, which is a higher closeness level than the edges $e_3(a_1, a_2)$ and $e_4(a_2, a_3)$ (level 1), and this leads to the pattern matching engine to ignore the edges $e_1(a_1, b)$ and $e_2(b, a_3)$, and thus b could not be matched anymore. As it can be seen, the rectangular area composed by the nodes of the match would enter the area taken by b. In order to avoid the conflict, the match is discarded.

The second constraint ensures that we get disjoint matches. When we have nodes that are shared by two or more matches, we convert the shared nodes in a new match, and we remove these nodes from the rest of matches, so obtaining two or more new matches. For instance, if we match a vertical column of nodes (Vertical Flow Layout) against the graph presented in Figure 6.17(b) that reflects the layout of Figure 6.17(a), then we will have two matches: $M_1 = \{a_1, a_2, c_1, c_2\}$ and $M_2 = \{b_1, b_2, c_1, c_2\}$. However, $\{c_1, c_2\}$ are conflicting nodes since they are shared by both matches. Therefore, we split the two matches in three matches, namely $M'_1 = \{a_1, a_2\}$, $M'_2 = \{b_1, b_2\}$ and $M'_3 = \{c_1, c_2\}$. When we split some matches, we have to check that every submatch still fits the layout pattern, otherwise it is discarded.

Matches found case (lines 17 to 22)

If there are matches for a pattern on the current graph (line 17), then the nodes of the match are merged into one node. The *mergeNodes()* (line 19) works as follows:

- A new node of type *LayoutNode* is created, which will represent the joining of the match. The new node is marked with the layout type (*type* attribute) that has been applied.
- All the matched nodes are removed from the original graph and included in the new node as children. The coordinates of the new node represent the area that contains all its children.
- All the edges between a pair of matched nodes (which are now children of the new node) are kept.
- All the edges from a non-matched node that starts or ends in a matched node now refer to the new node.
- If there are more than two edges between the new node and other non-matched node, the edges are replaced by a new edge. The Allen intervals are recalculated. The closeness level is the minimum level of the replaced edges.

After the reduction of the graph, the layout pattern applied is registered in the current solution sequence (line 21).
The permutation is reset (line 22) so the next layout to try will be the first one of the permutation again. This is needed to match the same pattern again over the rewritten graph. Hence, the *loop* variable is updated so that the iteration starts over the current closeness level.

No matches found case (lines 25 to 30)

If there are no layout pattern matches, the number of iterations without changes is increased (line 26). The condition in line 27 expresses that every k iterations without changes, being k the number of layout types used, the current closeness level limit (*closenessLimit*) is increased in one level. This will lead to that in the next iterations the layout patterns will be less strict about the distances between the elements. The *remainingNodes* variable is updated (line 24) with the order of the graph, considering that it is the number of *TileNodes* without parent.

Checking solutions (lines 32 to 42)

If the inner loop (lines 13 to 30) finishes and the number of remaining nodes is greater than one (line 32), then there is no solution. Otherwise, a solution has been found. If the solution found is different from all the solutions stored up to that point (line 34), then we may have a new solution. However, we cannot be sure whether the solution is new or not because two different solution sequences (made of layout pattern identifiers) may lead to the same graph when they contain common patterns in different orders. In this case, the corresponding layout tree is created (*createLayout() function* in line 35), and the layout trees are compared. In this way, only new solutions are stored (line 39). Please note that in a great number of cases the same solution sequence is reached by different permutations, so the *solSequences* set is an optimisation for fast comparison, which is useful to avoid many tree comparisons.

Creating the new layout (line 35)

The *createLayout()* function creates the layout structure for a *graph* that reflects the hierarchical structure of the layout that is going to be created. To make the explanation clearer we sometimes say '*Widgets*' or '*Container*' when we actually refer to 'the *LayoutElement* associated with that *Widget* or *Container*'.
For some of the layouts defined in the Layout model, specific attributes must be initialised. For instance, for a *BorderLayout* the nodes that correspond to the predefined areas (top, bottom, left, right, center) are set by analysing the incoming and outcoming relations of that node. A *FormLayout* also has its own attributes. The nodes that compose a *FormLayout* are analysed to identify the vertical alignment marks, which are distances in the X-axis that coincide (with some margin of tolerance) with the left or right bound of at least two nodes. A mark is represented as percentage of the relative distance to the left bound of the *Container*. When the alignment marks

have been detected, the nodes can be classified in the *AlignedColumns* defined by contiguous alignment marks.
For each *Layout*, the spacing and sizing properties are set (step 5 in Figure 6.2). *hSize* and *vSize* are the horizontal and vertical percentages of space occupied by the *Widgets* compared to their *Container*. *Gaps* (either horizontal or vertical) are created for each pair of adjacent *Widgets*. *Margins* are calculated for the *Layouts* (intermediate layouts or not) that are children of a *Layout* associated with a *Container*.
The *createLayout()* function also represents the alignment in an explicit manner (step 6 in Figure 6.2). *hAlignment* and *vAlignment* represent the horizontal and vertical alignment regarding the area of the enclosing layout, i.e. the minimum area which is large enough to contain all the widgets of the parent layout. For *Layouts* or *ElementNodes* nested in a horizontal *FlowLayout*, only *vAlignment* is set (the horizontal position is controlled by the layout manager) except in the case that there is a horizontal *FlowLayout* inside another one, then *hAlignment* is also set. Similarly, for *Layouts* or *ElementNodes* nested in a vertical *FlowLayout*, only *hAlignment* is set but in the case of a vertical *FlowLayout* nested in another one. The value to decide if the bound of an element (top, bottom, left or right) is aligned is 15% by default. For example, if we have a *HorizontalFlowLayout* associated with a container whose width is 100 pixels, and it contains an *ElementNode* associated to a label with coordinates (10, 20), then $hAlignment = LEFT$ for the *ElementNode* because $10/100 < 0.15$.

Assessing the new layout (line 37)

Every new solution is assessed by a fitness function (line 37) and assigned a fitness value. The meaning of our fitness function is that a higher value (close to one) denotes a better solution that a lower value (close to zero). The fitness value is calculated with the following formula:

$$fitness = \frac{n}{\sum_{i=1}^{n} w_i * (d_{max} - d_i + 1)}$$

Where n is the total number of layouts in the solution represented by the layout tree, w_i is the weight of the i-th layout, d_{max} is the depth of the layout tree and d_i is the depth of the i-th layout. The weight of a layout is obtained as follows:

- For each *FlowLayout* or *FormLayout* we add 2, but in the case of a *FlowLayout* nested in a *FormLayout* which is ignored (because it is part of the *FormLayout* and should not be counted twice).

- For each *GridLayout* or *BorderLayout* we add 1.

The fitness function gives a better score to more specific layouts (border and grid) over the flow and form layouts which are more general. It is also remarkable that deeper layouts get a worse fitness value than shallow layouts, because we prefer balanced layout trees (wider and less deep).

6.4.3 Layout inference example

Now we shall show an example on how the layout inference algorithm works based on the Tile model (i.e., graph) depicted in Figure 6.12. For the sake of simplicity, we will only use the following subset of the predefined layouts: horizontal flow layout, vertical flow layout and form layout, but the procedure would be applied in the same manner with more layout types. In this example there is only one closeness level, which means that all the distances between widgets are considered as similar. The algorithm will generate over $3! = 6$ permutations of the layouts, which are shown next (*HFlow* means horizontal flow layout, *VFlow* means vertical flow layout, and *Form* means form layout).

Iteration #1: HFlow, VFlow, Form	
Pattern to try	HFlow

Matches found	1: {*nameLabel, nameField*}
	2: {*passwordLabel, passwordField*}
	3: {*ok, cancel*}

Reduced graph	*(see step 1 in Figure 6.18)*
Graph order	3 nodes
Restart layout sequence	
Layout to try	HFlow
Matches found	None
Layout to try	VFlow
Matches found	1: {*name_merged, password_merged,* *ok_cancel_merged*}
Reduced graph	*(see step 2 in Figure 6.18)*
Graph order	1 node
New solution found: $sol1 = \{HFlow, VFlow\}$ $Fitness = \frac{4}{10}$ ((1 + 3) / (1 VFlow * 2 * 2 + 3 HFlow * 2 * 1))	
Solutions	sol1

Iteration #2: HFlow, Form, VFlow
Similar to Iteration #1 (no match for the *Form* pattern
Solution found: $sol2 = sol1 \rightarrow Discardsol2$

Iteration #3: VFlow, HFlow, Form	
Pattern to try	VFlow
Matches found	1: {*nameLabel, passwordLabel*}, 2: {*nameField, passwordField*}
Reduced graph	*(see step 1 in Figure 6.19)*
Graph order	4 nodes
Restart layout sequence	
Layout to try	VFlow
Matches found	None
Layout to try	HFlow
Matches found	1: {*name_pass_label_merged,* *name_pass_field_merged*}, 2: {*ok, cancel*}
Reduced graph	*(see step 2 in Figure 6.19)*
Graph order	2 nodes
Restart layout sequence	

Iteration #3 *(Continuation)*	
Pattern to try	VFlow
Matches found	1: {*name_pass_merged,* *ok_cancel_merged*}
New solution found: $sol3 = \{VFlow, HFlow, VFlow\}$ $Fitness = \frac{5}{18}$ ((1+2+2)/(1 VFlow * 2 * 3 + 2 HFlow * 2 * 2 + 2 VFlow * 2 * 1))	
Solutions	sol1, sol3

Iteration #4: VFlow, Form, HFlow
Similar to Iteration #3 (no match for the *Form* pattern)
Solution found: $sol4 = sol3 \rightarrow Discardsol4$

Iteration #5: Form, HFlow, VFlow	
Pattern to try	Form
Matches found	1: {*nameLabel, nameField, passwordLabel, passwordField, ok, cancel*}
Graph order	1 node
New solution found: $sol5 = \{Form\}$ $Fitness = \frac{1}{2}$ (1 / (1 Form * 2))	
Solutions	sol1, sol3, sol5

Iteration #6: Form, VFlow, HFlow
Identical to Iteration #5   Solution found: $sol6 = sol5 \rightarrow Discardsol6$

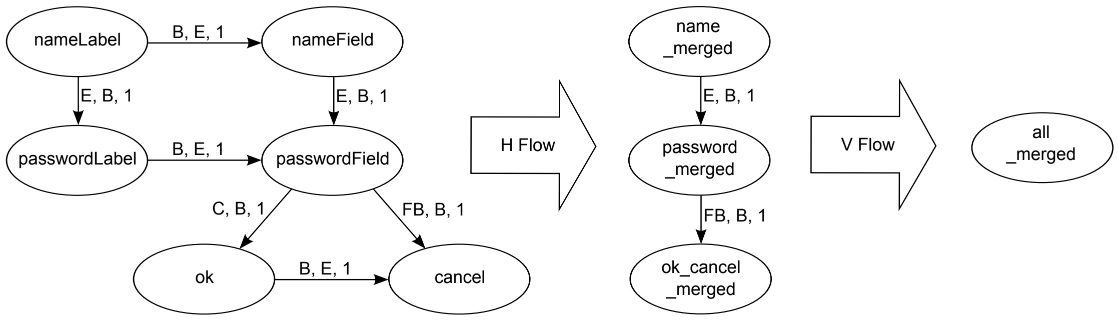


Figure 6.18: Inference example. Permutation {*HFlow, VFlow, Form*} applied to the graph in Figure 6.12.

The algorithm returns the solution set: $\{sol1, sol3, sol5\}$. The best solution is *sol5* because it has the highest fitness value.

There are some remarkable details about the layouts created. We will comment on the best solution. It is a *FormLayout* with two *AlignedColumns* (columns *c1* and *c2* in Figure 6.20). The first *AlignedColumn* contains *nameLabel* and *passwordLabel* aligned to both left and right (i.e., justified). The second *AlignedColumn* includes *nameField* and *passwordField* that aligned to both left and right, and *ok* and *cancel* that are aligned to the right.

The *FormLayout* has *hAlignment=CENTER* and *vAlignment=CENTER* because the group of widgets laid out are centered in their container (the window). The last Vertical *FlowLayout*

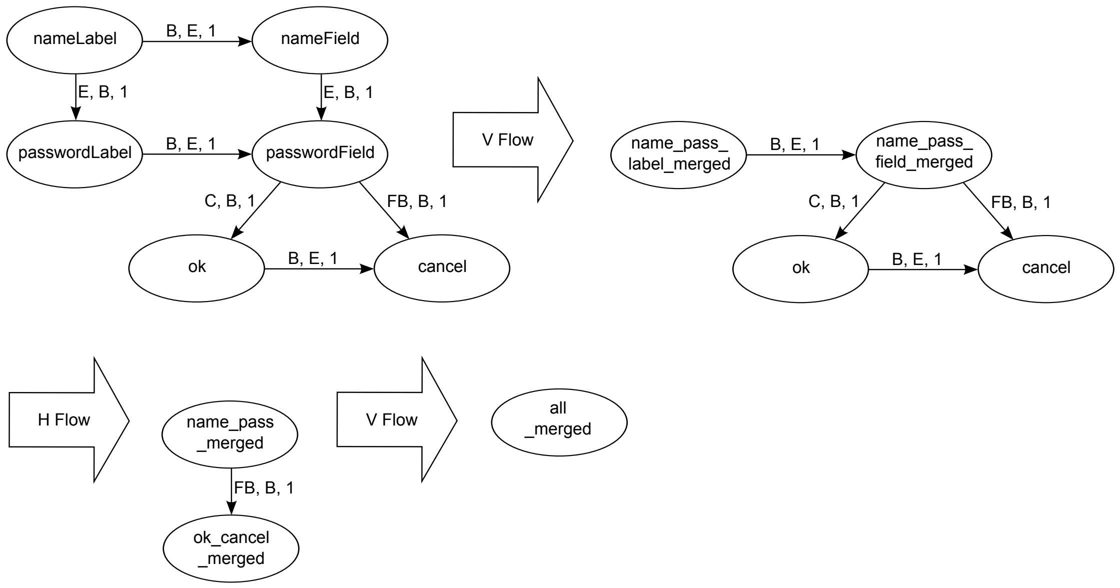


Figure 6.19: Inference example. Permutation {*VFlow, HFlow, Form*} applied to the graph in Figure 6.12.

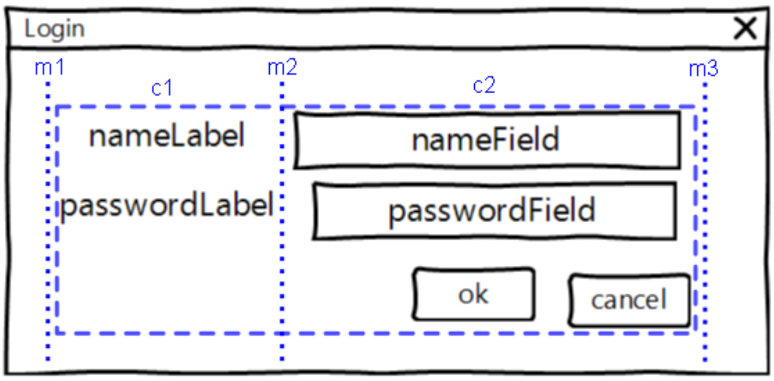


Figure 6.20: Alignment columns for the Login window.

inside the *FormLayout* (i.e., the *ok* and *cancel* widgets) has *vAlignment=RIGHT*, because both widgets as a whole are aligned to the right part of the area delimited by *FormLayout* (i.e., the area of all the widgets inside the window).

6.4.4 Performance evaluation

In this section we will show the results of the performance analysis of the layout inference algorithm that we have carried out. We have generated several views containing an increasing number of widgets. The widgets of a view are arranged in groups, and each group conforms to a layout type supported by our algorithm. The groups are randomly placed but close to other groups (so there are not significant distances between widgets), with the additional constraint that a group cannot overlap another one. Then, we have measured the execution time of the inference algorithm. Aftwerwards, the same process is repeated but this time the same views are arranged in several containers, to emulate the common scenario when developers design GUIs. Besides, the analysis has been carried out for three to five layout types to show the impact of the number of layout types.

Figure 6.21 shows the execution (in seconds) in the case that there are no containers in the view (the view itself is the only container) and all the widgets are close. This is the worst case as the algorithm has to deal with a single graph with all the widgets. Figure 6.22 shows the result when the view is split in containers, with each container consisting of up to 20 widgets. This is an average case of the algorithm. The tests have been run in an Intel Core i5 with 4GB RAM.

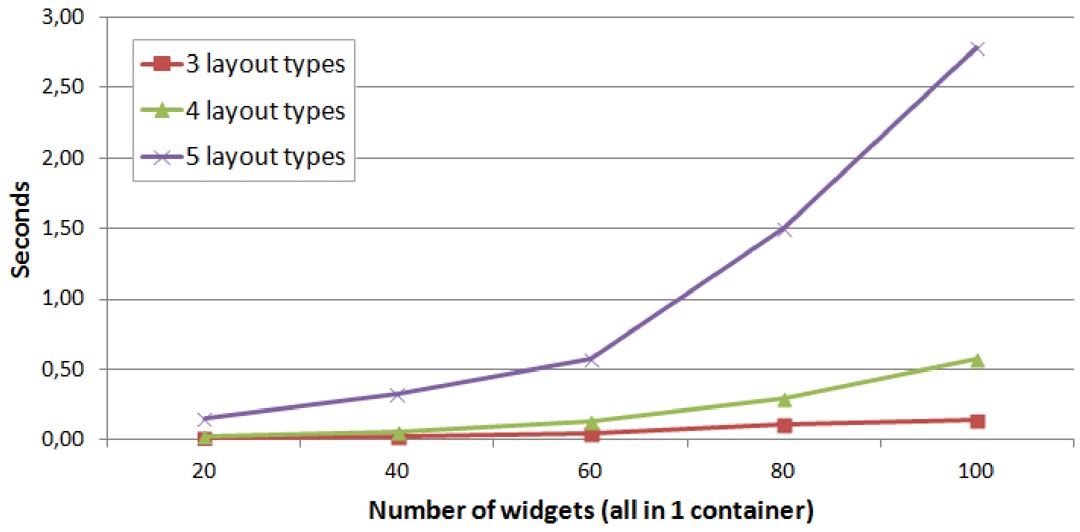


Figure 6.21: Execution time for widgets in a single container.

Comparing both charts we can see that the layout inference applied to a view split in containers (Figure 6.22) obtains significative better results than using no containers at all (Figure 6.21). When a view is split in containers and the number of widgets is augmented, the execution time is linearly increased. On the other hand, when a view is not split in containers the execution

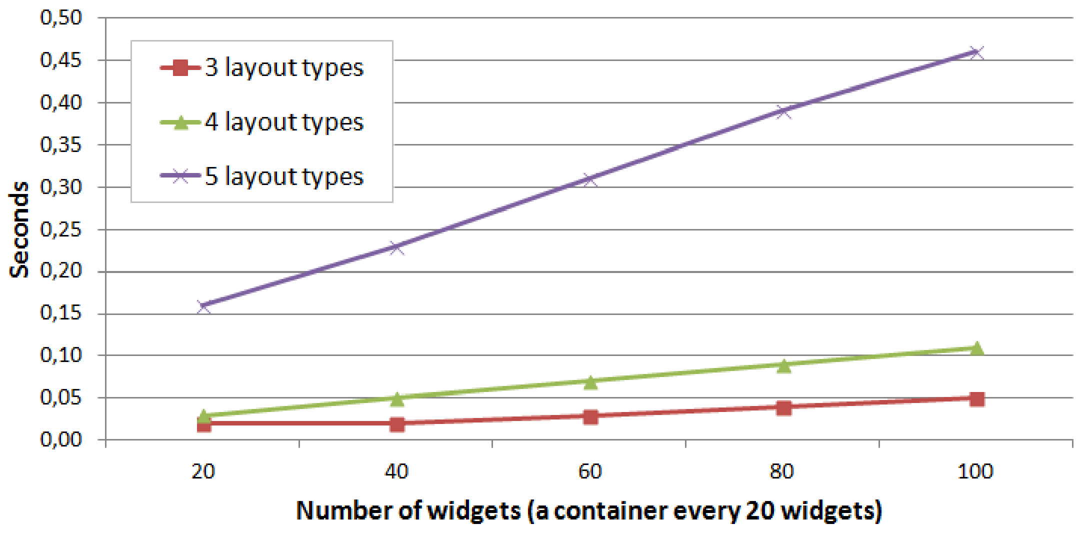


Figure 6.22: Execution time for widgets arranged in containers (a container every 20 widgets).

time increases in a polynomial or even exponential way. It comes as no suprise since the pattern matching is applied on smaller graphs with only one container that contains a large graph. For example, if applying the algorithm to a view of 100 nodes arranged in 5 containers of 20 nodes each one, then the algorithm analyses 5 graphs of 20 nodes each one, and one graph with 5 nodes (i.e., the graph that relates the 5 containers among each other). Note that in the vast majority of cases, views are arranged in containers, so the chart displayed in Figure 6.22 is more realistic than Figure 6.21.

As it can be seen in both graphs, applying the inference algorithm with more layout types seems to have an exponential impact on the execution time, which is logical since the algorithm iterates as often as the number of permutations ($n!$) of n layout types. For three layout types, we used HFlow, VFlow and Form layouts, for four layout types we used the same three layouts and either Grid or Border layout (different tests with each one), and for five layout types we used HFlow, VFlow, Form, Grid and Border layouts. We decide to include the Form layout in all cases because its pattern matching is the most complex one, whereas for the other layout types the complexity is more or less similar.

The results show that the actual execution time of the inference algorithm is reasonably acceptable (beyond its algorithmic complexity). For interactive applications that require on-the-fly layout inference, using the algorithm with more than 5 layout types may be a little slow (5 lay-

out types and 100 widgets takes 2,78 seconds in the worst case and 0,46 seconds in the average case). However, for non-interactive applications such as GUI migration, that works on batch mode, the algorithm is practical as it obtains an admissible execution time with even more than five layout types.
The current implementation of the algorithm has little optimisations, so in the future we expect a substantial drop in the execution time when some optimisations are done (e.g., pruning the search tree by detecting already visited sequences).

6.5 Case study: from Wireframes to fluid web interfaces

This section presents a case study that has been carried out to put into practice the layout inference approach of the previous section. The goal of the case study is to automatically generate final GUI code from wireframes created with the WireframeSketcher [106] tool (it could have been equally applied to sketching and mockup-tools without loss of generality). The transformation of a wireframe into a final user interface requires layout inference in order to generate the source code for a particular platform and GUI toolkit. The actual implementation used in this case study is presented in Section 6.6.

6.5.1 Context of the case study

Designing GUIs is a crucial and complex task in software application development, which involves dealing with aspects such as functionality, accessibility and usability. An iterative process is normally applied in GUI design, in which several representations of the GUI at a different detail level are built, so users and developers can experiment and discuss about the structure and behaviour of the GUI. Frequently three representations are used: sketches, wireframes and mockups. Sketches are rapid, freehand drawings that show an initial design idea on the interface. Wireframes reflect how the contents are distributed in the screen (i.e., the layout of the widgets that represent the content). Mockups refine wireframes by adding details like colours or images.
Wireframing tools commonly provide specific editors for creating wireframes and mockups. There are also tools that can automatically generate final GUI code from wireframes or mockups. For example, Reify [107] is a tool that generates web interfaces from wireframes created with the Balsamiq [56] wireframing tool. However, wireframe generation tools have significant limitations at present, as they are limited to certain types of layouts [4], they are only applicable

on certain platforms such as web interfaces with CSS [5], or they are still in an immature state (like Reify).

Wireframes do not have an explicit notion of layout, but widgets are dragged from a palette and placed into a particular position (which is sometimes almost arbitrary) on a canvas. Wireframes therefore only provide a *coordinate-based layout*. The transformation of wireframes into final GUIs in modern platforms with explicit layout facilities poses the challenge of uncovering the implicit structure of the GUI in order to obtain an explicit representation of the layout. Although we have only used wireframes in the case study, the approach is also applicable to mockups.

On the other hand, in 2010 Ethan Marcotte coined the term *Responsive Web Design* [108] to a design philosophy aimed at crafting sites to provide an optimal viewing experience. Three basic principles make up this philosophy: i) define fluid grids, ii) define flexible images, and iii) use CSS 3 media queries to change the style depending on the screen dimension. Since then, responsive interfaces have become popular, as well as fluid layouts (not necessarily grids).

In the case study, adaptive web interfaces (interfaces with a fluid layout) were generated by using ZK [44]. We did not addressed a full-fledged responsive UI design because that implies the use of algorithms to rearrange the content. As our Layout model explicitly captures explicit information about the layout, this rearrangement is possible. Thus, this case study can be considered as a first step towards generating responsive web interfaces.

6.5.2 Evaluation of the approach

We have conducted an experiment with users to validate our approach. 20 people working on the IT sector (with different roles such as web developer or software analyst) have been prompted to design a series of wireframes and then apply our layout inference tool to generate the source code of the final GUI.

6.5.2.1 Methodology

The methodology we have used is the following. Firstly, each person was provided with an explanatory document that he or she should read carefully. The document explains the utility of wireframes, gives some instructions on how to use both the wireframing tool we have chosen (WireframeSketcher) and our layout inference tool, and explains the task to accomplish. After reading the document, each participant should accomplish the design of 5 screens for an on-line bookstore application. These 5 screens intend to be typical views of a web application that involves common design patterns such as master-detail or registration form. Particularly, these are the 5 screens we demanded (the name of the view is indicated before the colon):

- *best*: it displays information about the best-sellers.
- *cart*: a shopping-cart view that shows the current state of the cart.
- *detail*: it shows detailed information about a selected book.
- *search*: allows searching for some criteria and see the results of the query.
- *user*: it lets users create a new user account.

When the user finished the screens, he or she was encouraged to apply our tool to generate the code of the final GUI and execute it to see how the view looks like. Participants must not modify the default values for the tool parameters (see Figure 6.37) in the first execution, but can be altered if the result was not what the user expected at first.
From the wireframes designed by the users we assess the approach in two ways. Firstly, each user was requested to fill in a questionnaire about the experience, where the user could indicate how good the generated view was and they could express whether the generated layout matched the idea he or she had in mind. This was intented to know how useful is the tool from the developer point of view. Secondly, we demanded the participants to submit the wireframes they created, in order to perform a quantitative analysis of the result of the layout inference.

6.5.2.2 Quantitative results

Screen	**best**	**cart**	**detail**	**search**	**user**	**TOTAL**
Visual resemblance	96.2%	97.2%	97.2%	99.1%	95.6%	97.4%
Parameter changes	30.0%	33.3%	30.0%	26.7%	36.7%	31.3%
Average of layouts (best solution)	5.0	4.2	4.9	3.4	2.4	4.0
Average of alternatives per view	2.6	4.1	3.3	3.1	4.2	3.4
Layout resizing	77.8%	84.9%	75.6%	90.5%	93.4%	84.4%

Table 6.1: Evaluation results.

Table 6.1 shows the results of the evaluation (classified by view). *Visual resemblance* measures how good the generated GUI resembles the original wireframe (i.e., the accurary of the generated GUI). We count the number of generated widgets located in the same place as the corresponding widget in the original view. Our tool intentionally compares widgets with some degree of flexibility, so minor misalignments are allowed, and widgets that are clearly misplaced or misaligned with regard to the original view are counted as errors. In global, there is a high degree of accuracy (97%) of the generated windows, and there is no significative difference between the different types of views. This high accuracy is partly because if, in the first try, the GUI does not ressemble the original wireframe we change the algorithm parameters in order to improve the result, as it is explained below.

Parameter changes expresses the percentage of views designed by the users that required changes in the default values of the parameters to get a reasonable good GUI (high *visual resemblance*). As it can be seen, in many cases (31% of the views) parameters needed to be tuned and there are not remarkable differences between the different types of views. From the end-user perspective, this means that he or she would get a good enough GUI without tuning the algorithm in the 70% of the cases. In the next subsection we will explain a current limitation of the approach related to the maximum closeness cluster deviation parameter.

The *average of layouts* of the best solution counts the number of layout managers used in the best solution (i.e., the solution which the highest fitness value). On average a composition of 4 layout managers are required to completely define the layout of the views. A low average of layouts indicates that the best solution does not use unnecessary layouts but just the required layouts (i.e., it is efficient in most cases).

The *average of alternatives per view* represents the number of different layout compositions that are offered for each view on average. In our case we have an average of 3.4 alternatives per view.

The last row of the table (*layout resizing*) indicates whether the final GUI generated for the best solution is resized appropriately when tested. There are different alternative solutions for a given view that at first glance may seem equally valid, but they look completely different when the view is resized. A good layout solution arranges the widgets in such a way that, when they are resized, they seem alright.

We have an 84,4% of success rate related to view resizing, which means that there are around 15% of views for which the fitness function fails (i.e., it does not always select the best option for resizing). We have a slightly higher success rate for the *search* and *user* types of views, because for these views most people used more or less standard form-like designs which fit our

FormLayout instead of using complex combinations of other layout types. Using more complex fitness functions that not only take into account the number of layouts involved could result in improvements in the *best*, *cart* and *detail* types of views. We will deep into the limitations of the current fitness function in the next subsection.

6.5.2.3 User assessment

Users filled in a questionnaire that included five questions that summarise their experience. The questions were: 'Are the generated views as I expected?', 'Are the margins, gaps and alignment correct?', 'When resizing the windows, are the widgets resized appropriately?', 'Could the generated windows be used in a real application?', and 'Is the layout inference tool useful?'. They were graded by using a 5-point Likert scale, and the results for the questions are shown in Figures 6.23 to 6.27 respectively.

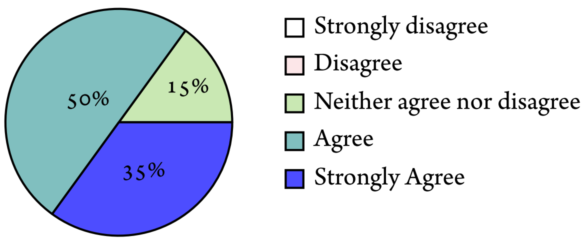


Figure 6.23: Are the generated views as I expected?

The vast majority of the users (85%) agree or totally agree that at first sight, the generated views resemble the original ones (see Figure 6.24). This results are in line with the assessment that we tackled by manually inspecting the models and views, but we could have expected a higher rate in this question. The reason because users did not give a better mark to this question is due to almost none of the users (only 10% of them) changed the default parameters, so the algorithm did not always showed a very good result. If users had tuned the parameters, better score would have probably been achieved.

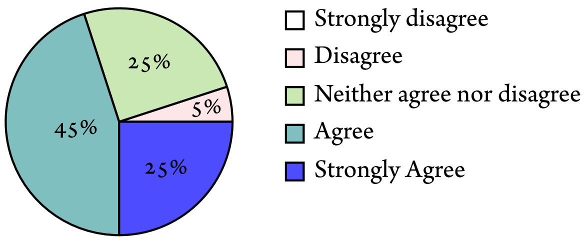


Figure 6.24: Are the margins, gaps and alignment correct?

The results of 'Are the margins, gaps and alignment correct?' (Figure 6.24) have a certain degree of similitude with the results obtained in the previous question, since that views that are more or less similar to the original ones must have correct margins, gaps, and alignment.

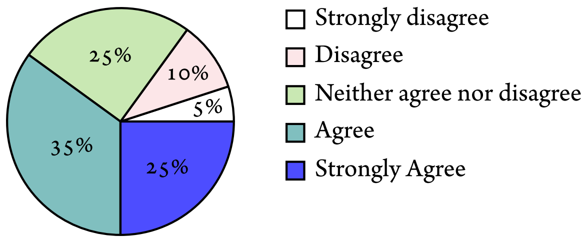


Figure 6.25: When resizing the windows, are the widgets resized appropriately?

With respect to resizing, 60% of the users think (agree or totally agree) that the resizing behaviour is more or less suitable (see Figure 6.25). As we indicated, the resizing behaviour is not always suitable, which is partly related to the weakness of the fitness function, which we will explain in detail in the next subsection. The difference between the score of the quantitative evaluation (84% of success) and the score given by users is mainly due to the fact that users did not tune the parameters, so they found weird resizing in many cases.

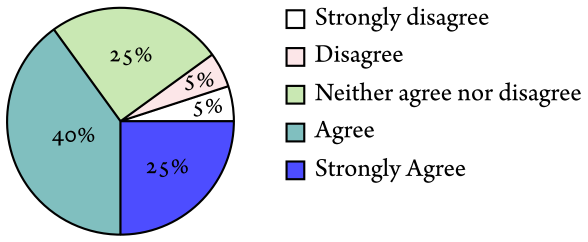


Figure 6.26: Could the generated windows be used in a real application?

With regard to the question 'Could the generated windows be used in a real application?' 65% of the users agree, but others have some reservations, mainly due to that resizing fails in some cases and users have to tune parameters that they do not feel that are easy to change.

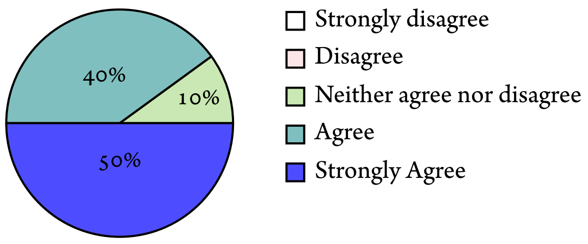


Figure 6.27: Is the layout inference tool useful?

50% of the users find the tool extremely useful, and 40% think that the approach is useful (Figure 6.27). Despite the current limitations of the approach, developers think that the tool is useful because they can reuse wireframes and save time when implementing the GUI of the application.

6.5.2.4 Approach limitations

The current implementation of the approach has two limitations at present. The first limitation is related to the maximum closeness cluster deviation parameter, and the second limitation is the implementation of the fitness function.

Widget distance clustering. As we have already said, the maximum cluster deviation parameter represents the maximum standard deviation of a group of distances that is admissible for them so they all can be considered similar. This parameter is not only used to perform flexible comparisons, but also drives the pattern matching phase by indicating what relations among the elements can be matched.
For example, in the fragment of the *Detail* view in Figure 6.28 there are two clearly different areas in the view, the form on the left and the right part composed of the *Cover* image and the *Enlarge* button, thus there should be a layout for each part, and a layout that 'glues' both parts. The designer that created the view left some empty space in the middle of the view on purpose, so both parts can be distinguished. Given that the 'close' or 'far' concepts are subjective (they depend on the human perception), we need the maximum closeness deviation parameter to be able to decide which widgets are close or far.

But sometimes, the closeness level between a pair of widgets may confuse the inference process. In the example of Figure 6.28, the relation between the *Description* label and the *Author* label will have a closeness level higher (i.e., they are significantly distant) than the *Description* label and the *Description* text area, and higher than the *Description* text area and the *Author* text area. While this is strictly correct, the inference process should consider these relations as equally close because they are part of a form and the user that created the view meant it to be one form, not two separated parts. In such cases, the maximum closeness deviation parameter may require being carefully tuned in order to get a good result.

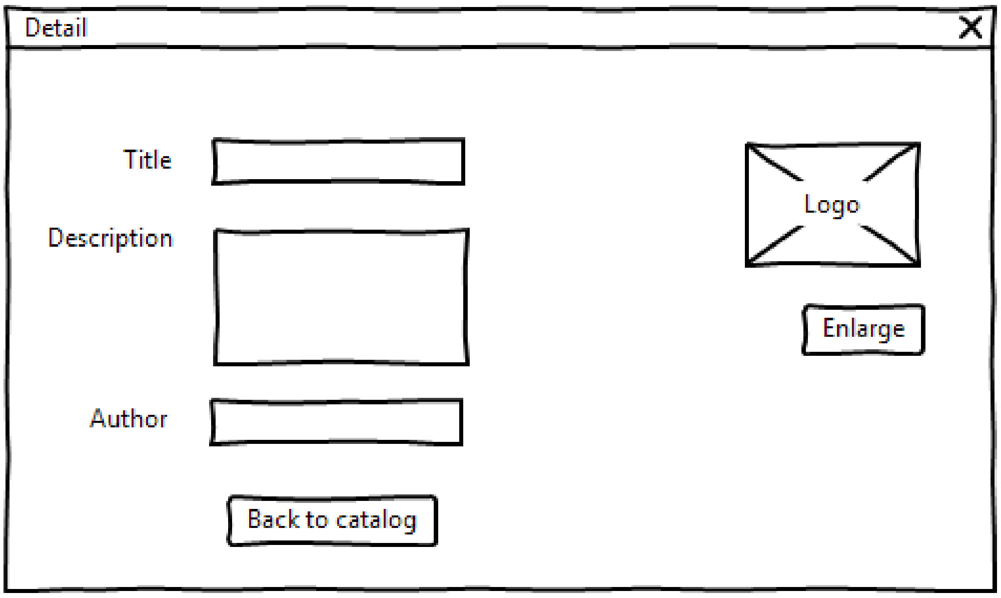


Figure 6.28: Example of the closeness problem.

Fitness function implementation. Figure 6.29 shows a simplified *Best* view. Let us assume that we only want to use the (vertical/horizontal) FlowLayout. This view can be laid out in two different ways (two layout alternatives), which are:

- *Alternative 1*: a horizontal FlowLayout composed of 4 vertical FlowLayouts (with 3 widgets each one).
- *Alternative 2*: a vertical FlowLayout composed of 3 horizontal FlowLayouts (with 4 widgets each one).

Figure 6.30 shows the generated view for the first alternative, and Figure 6.31 shows that view after resizing. In the same manner, Figure 6.32 shows the second alternative, and Figure 6.33 shows the resized view. It can be seen that the alternative 1 generates nice views since the aspect ratio of widgets and distances are kept, but on the contrary, alternative 2 leads to unaesthetical views when they are resized. These differences in the appearance of the GUI are due to the way that the information about the layout is expressed in each alternative. In the first alternative, we can specify that each widget is centered with regard to its column, but in the second way, we have no direct means to specify the relative distances between the widgets in each row. We can see that there are better alternatives than others (particularly, alternative 1 is better than alternative 2).

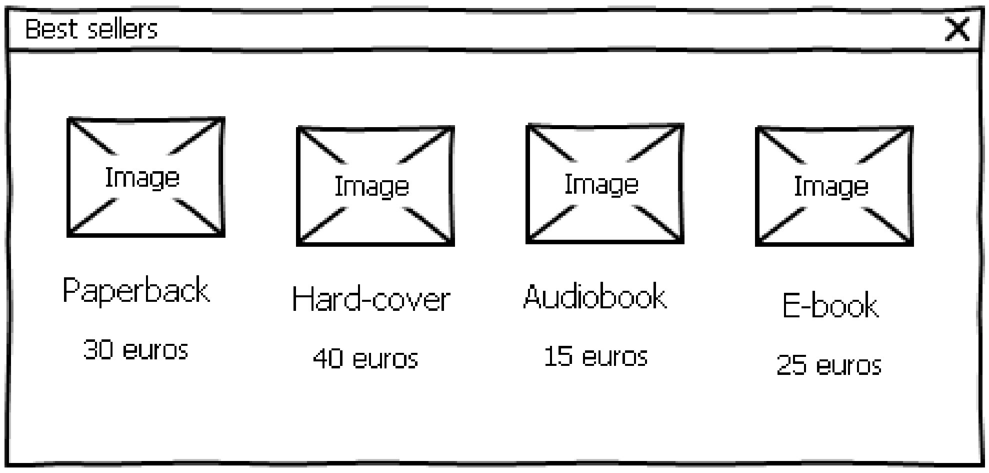


Figure 6.29: Example window.

Therefore, we need to develop more complex fitness functions that not only take into account efficient layout compositions (in the sense of reducing the number of layouts nested) but also aesthetic criteria. Particularly, considering the homogeneity of the content of the layouts in combination with the current fitness function could lead to better results.

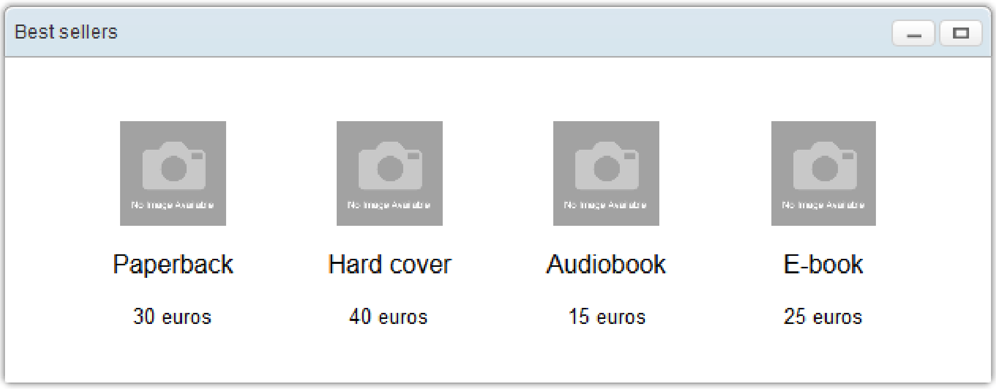


Figure 6.30: Example horizontal-vertical flow.

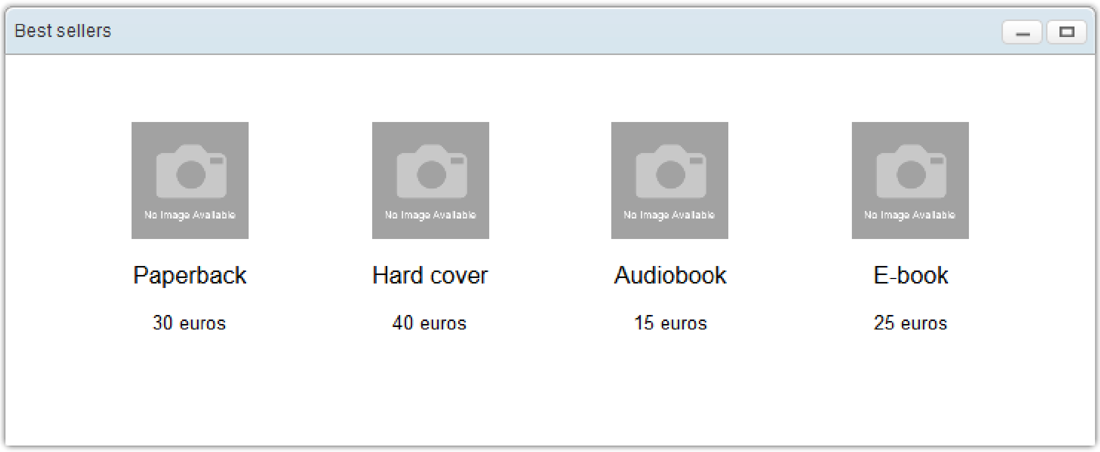


Figure 6.31: Example horizontal-vertical flow resized.

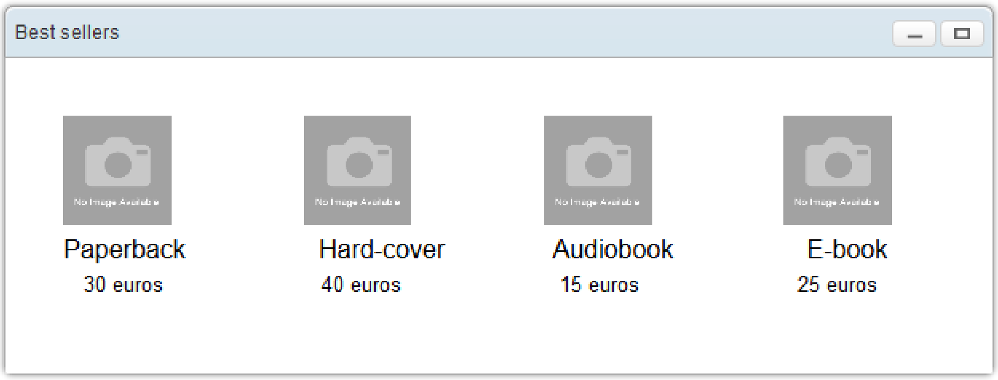


Figure 6.32: Example vertical-horizontal flow.

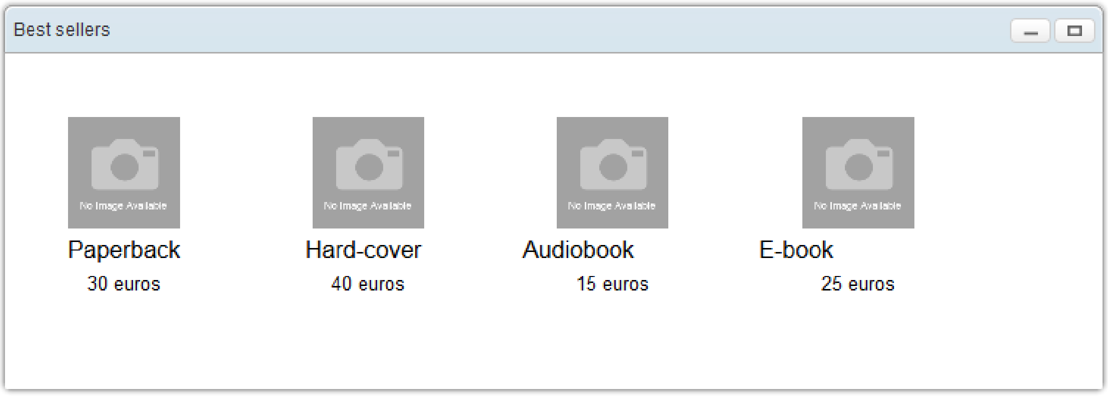


Figure 6.33: Example vertical-horizontal flow resized.

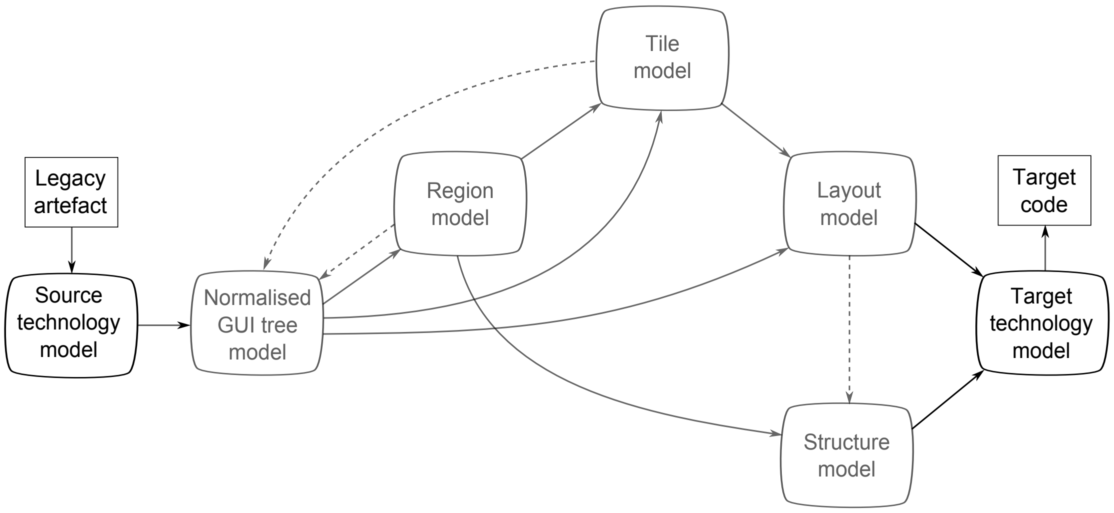


Figure 6.34: Parts of the MDE architecture related to the Wireframes to ZK case study.

6.6 Implementation

We have implemented a tool that supports the case study and includes an implementation of the proposed algorithm. The tool provides all the necessary elements to transform a wireframe created with WireframeSketcher to a fluid user interface with the ZK framework. It has been implemented in Java using the Eclipse Modeling Framework (EMF) [53], using Ecore to define the metamodels. M2M and M2T transformations have been programmed in Java using the Dynamic EMF API.

Next we will show a few details about the implementation of the tool. Given that the main part has been thoroughly explained in the previous sections, we will briefly comment on the parts of the approach that are dependant of the case study and which can be used as a guide for implementing other scenarios. Concretely, these parts are (see highlighted parts in Figure 6.34): the obtainment of Normalised models from the models provided by WireframeSketcher, and the generation of web interfaces with ZK from the Layout and Structure models. We will also outline the transformation to get Structure models. Finally, we will briefly present the interface of the tool.

6.6.1 Mapping WireframeSketcher to Normalised models

WireframeSketcher [106] is a tool to create wireframes and mockups for desktop, web and mobile applications, which can be run on the Eclipse platform. WireframeSketcher generates wireframes as models conforming to a metamodel that is provided with the tool, which is partially shown in Figure 6.35. If we used a different wireframing tool that does not represent wireframes as models, implementing an ad-hoc injector or using injection tools such as Gra2MoL [22] or MoDisco [23] would be required.

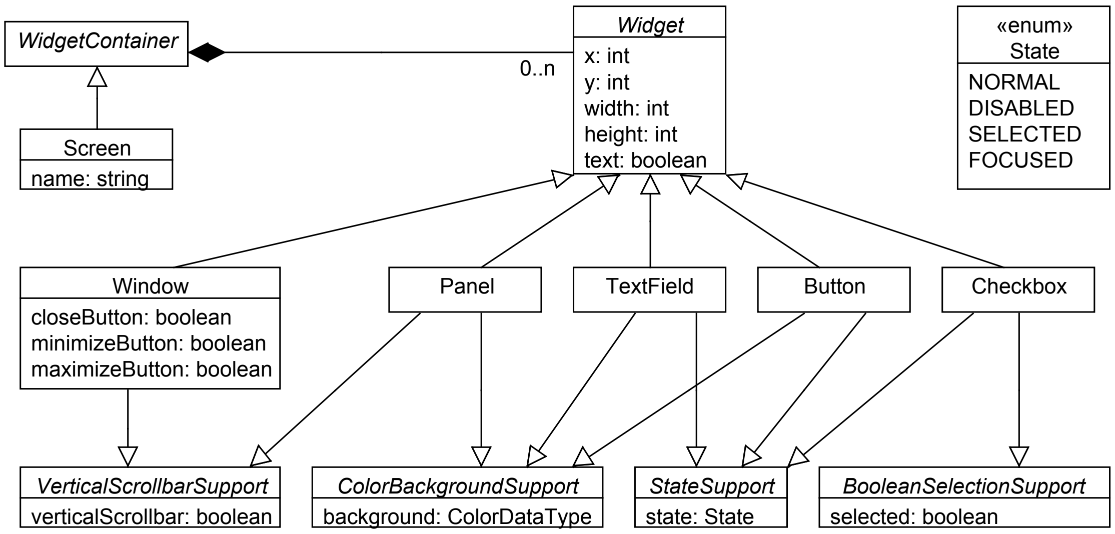


Figure 6.35: Excerpt of the WireframeSketcher metamodel.

The metamodel is relatively simple. *Screens* are the canvases where users design the views of the new applications. There are different types of *Widgets*, every type of *Widget* having a different set of properties (depending on the *Support* metaclass that they inherit, e.g., a *TextField* has the

State and *ColorBackground* attributes). It is worth remarking two details. Firstly, all the widgets are placed with absolute coordinates. Secondly, neither *Windows* nor *Panels* are *WidgetContainers*, so the *Widgets* in the *Screen* will be overlapped. These two features were also found in the RAD applications. Therefore, the solution that we make for wireframes can be reused for RAD applications and vice-versa.

To make the rest of the layout inference process independent of the source wireframing tool artefacts the source WireframeSketcher models are transformed into *Normalised models*. The model transformation basically maps the WireframeSketcher widgets to generic widgets (for most of them there is a 1-to-1 mapping), which is a straightforward task. Additionally, the transformation performs two actions: reduces the area of a label to the area that is actually occupied by the text of the label, and checks that it does not exist two widgets (*SingleWidgets*, such as combo boxes or text fields) that are visually overlapped. If there are overlapping widgets, the process stops and the user is notified about the conflict so he or she can manually solve it and continue the process.

6.6.2 Mapping Normalised models to Structure models

The M2M transformation that obtains the Structure model from the Normalised and Region models is relatively simple, and works as follows. The region hierarchy is navigated and replicated in the Structure model by means of *Panels*. The leaves of the hierarchy are the *SingleWidgets* that are mapped to the *SingleWidgets* in the Structure model. Toolbars and menus, which are not explicitly represented in the Region model because they belong to the *View* regions, are generated after them. For the *SingleWidgets* containing text or images, the corresponding *GraphicalResource* is created and linked to it. For each *Container* and *SingleWidget*, the graphical attributes that are common to most of the current GUI toolkits are mapped. Normally, only a few attributes are left outside this mapping, because are too specific or not related to the presentation layer; for example, the property that links a widget with a column in a database table.

6.6.3 Generation of the web interface

The Layout model is the result of the layout inference process, and makes GUI restructuring and code generation possible. In the case study we have transformed the Structure and Layout models into a ZK model to generate ZK views.

ZK is a UI framework to build web and mobile applications, which implements the Model-View-Controller (MVC) and Model-View-ViewModel (MVVM) design patterns. ZK views run on application servers that are compliant with the Java Servlet and Java Server Pages specifications, like Apache Tomcat [109]. The Model and Controller parts of the ZK application are written in Java, whereas views are created using Java or by means of a readable XML-formatted language called ZUML (ZK User Interface Markup Language). ZUML allows creating fluid layouts by using different layout managers and the *hflex*/*vflex* attributes that indicate the ratio of the total width/height that the element should take.
We have mapped the layouts defined in the Layout model to the ZK layouts. For example, the *BorderLayout* is mapped to a *BorderLayout* in ZK, and the *FormLayout* is transformed into a *TableLayout*. The *hSize* and *vSize* attributes of the Layout model are used to calculate the *hflex* and *vflex* attributes. *hAlignment* and *vAlignment* in our Layout model express the alignment, which in ZK is handled with the *pack* and *align* attributes inside boxes. Margins and gaps can be specified in ZK views by means of the CSS model box (margin and padding). Figure 6.36 shows the ZK view generated for the original wireframe created with WireframeSketcher (Figure 6.11).

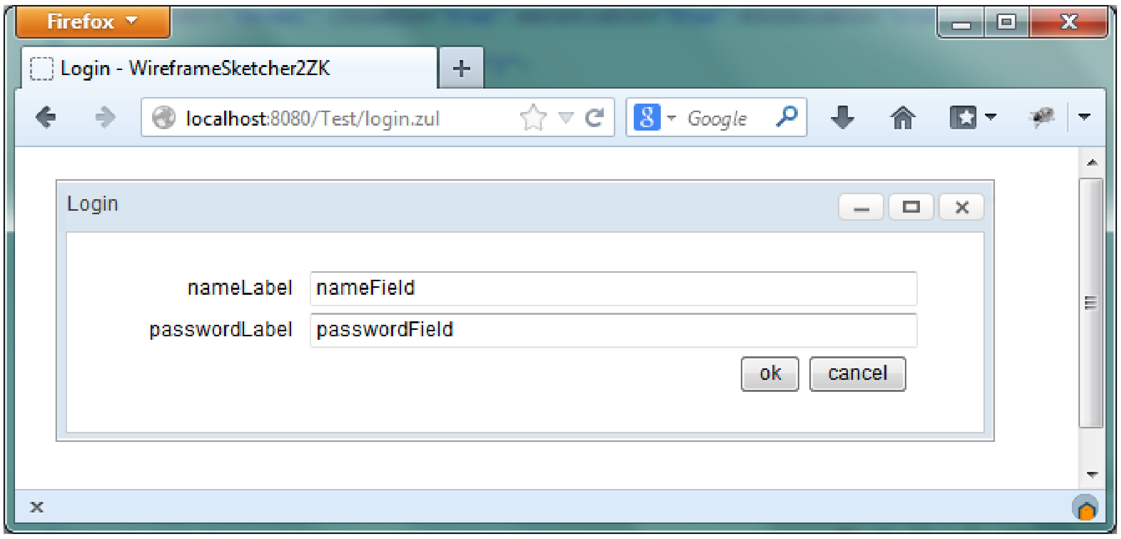


Figure 6.36: The login window generated in ZK.

In our case study we decided to generate code for a concrete UI framework. However, developers can take advantage of the CUI model (i.e. the Structure and the Layout model) in other manners. For instance, the CUI model could be mapped into another User Interface Description Language (UIDL) such as UsiXML [38] (this can be considered as restructuring) in order to take advantage of the existing tools, for example, their code generators for mobile applications.

6.6.4 The tool

The tool consists of a plugin for the Eclipse IDE that is integrated with *WireframeSketcher* tool (which is also an Eclipse plugin). The plugin offers two facilities:

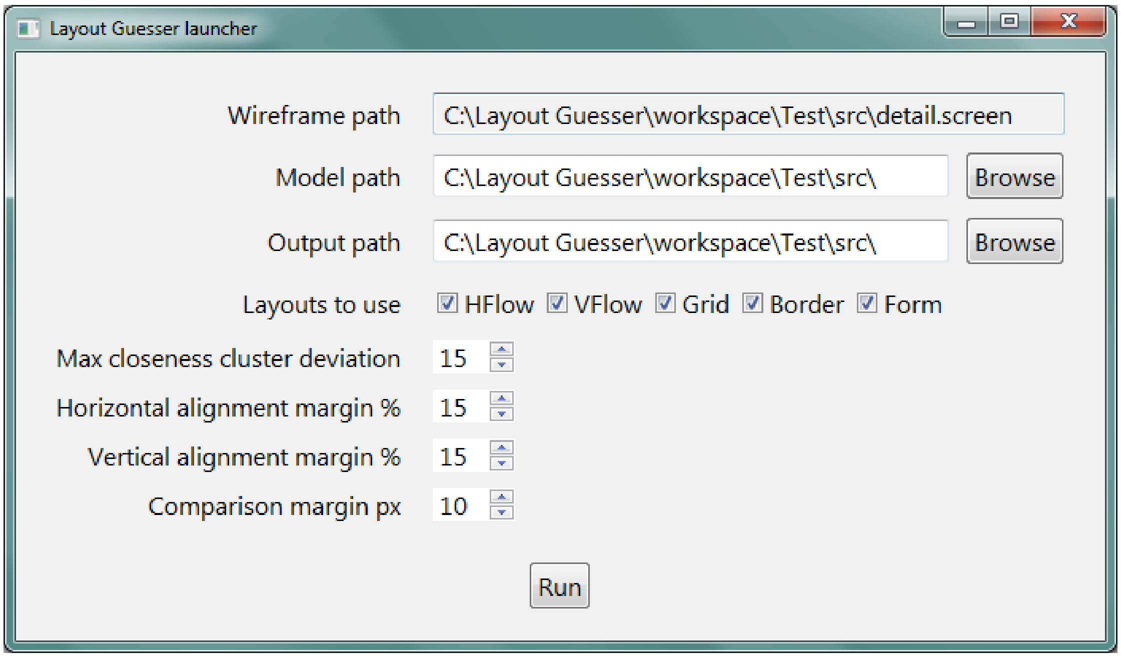
- Generate the Structure and Layout models from the wireframes created with WireframeSketcher (*.screen* files).
- Generate ZK code (ZUML web pages) from the Structure and Layout model files. In the future the tool will offer the possibility to generate code for different toolkits.

The execution of the layout inference module depends on several parameters that can be seen in Figure 6.37:

- The layout types to use in the layout inference algorithm.
- The maximum allowed standard deviation of the distances in every cluster. It affect how the distances are clustered to obtain the closeness levels.
- The horizontal/vertical alignment margin, which is expressed as a percentage of the container widget. It has a result in the alignment comparisons. For example, for a widget to be considered as aligned to the right, the percentage of the distance between the right bound of the widget and the container must be equals or less than that value.
- The comparison margin in pixels is used to give some flexibility when performing comparisons between widgets (e.g. when detecting if two adjacent widgets are aligned to the left).

6.7 Comparison of the greedy and exploratory approaches

We have compared the two layout inference strategies by means of applying the second algorithm to 30 of the 107 Oracle Forms windows that compose the application of the case study A


Layout Guesser launcher

Wireframe path: C:\Layout Guesser\workspace\Test\src\detail.screen

Model path: C:\Layout Guesser\workspace\Test\src\

Output path: C:\Layout Guesser\workspace\Test\src\

Layouts to use: ☑ HFlow ☑ VFlow ☑ Grid ☑ Border ☑ Form

Max closeness cluster deviation: 15

Horizontal alignment margin %: 15

Vertical alignment margin %: 15

Comparison margin px: 10

Figure 6.37: Layout inference parameters.

explained in Section 5.8.2. Concretely we have chosen 6 of large complexity, 5 of medium complexity and 19 of small complexity, so the ratio of windows in the original application are kept. We opted for the case study A because it got worse results than case study B. Table 6.2 shows the percentages of regions and widgets that have been correctly placed in both approaches (*v1* and *v2* are used to denote the versions 1 and 2 of the algorithm respectively).

	Large (>60)	Medium (20 - 60)	Small (<20)	Total
Windows of each type (out of the total)	19.63%	16.82%	63.55%	100%
Parts laid out OK (v1)	83.24%	98.06%	100.00%	96.38%
Parts laid out OK (v2)	95.24%	100.00%	100.00%	99.07%
Widgets laid out OK (v1)	87.14%	85.61%	88.10%	87.50%
Widgets laid out OK (v2)	91.25%	94.12%	98.00%	97.29%

Table 6.2: Evaluation results for the case study A.

In relation to the layout of the parts, there has been a significant improvement for large windows (12%), while in the case of medium windows the 100% have been laid out ok (98% with the first version). We used in the second approach the same algorithm for region detection as we used in the first approach, so the regions detected for the windows are the same. However, the layout among distinguished parts has been ameliorated. As we already mentioned, it depends not only on the region detection but also the layout recognition inside regions, so the improvement is due to the latter. The small failure rate is caused by some bugs and limitations of the implementation. Particularly, the current implementation does not allow the text of the frames be replaced by widgets such as checkboxes, which spoils the region detection.

The percentages of widgets laid out in the second version have also improved considerably: 4.11% for large windows, 8.51% for medium-size windows and 10% for small windows. In the first version, many simple (small complexity) windows were not perfectly replicated because the widgets could only be aligned with regard to the container region (widget alignment problem), and gaps where not explicitly indicated (unidentified holes problem). In the second version, widgets can be aligned with respect to the containing layout and gaps are explicitly expressed. Moreover, the *FormLayout* considers horizontal alignment with also helps to indicate a suitable alignment, so widgets match to a great extent the appearance of the original window.

Regarding the widgets laid out in medium and large windows, in the first version the layout managers defined could not be composed so complex layouts could not be properly captured (non-regular layout detection problem). Additionally, developers used to make full use of the empty space in large windows, which led to even crammed and weird layouts which were awkwardly recognised by our first approach. Due to the second approach allows layout nesting and explores different combinations of elements, it is able to recognise more complex layouts that in the first approach. However, the improvement is not so high as it could be expected. The reason behind this fact is that, in many cases, the alternative solution that is chosen as the best alternative does not get a perfect layout (fitness function problem).

For instance, Figure 6.38 shows a window in the Oracle Forms designer (not in runtime), Figure 6.39 depicts the window that has been generated by the first approach from the former, and Figure 6.40 shows the window generated by the second approach. The layout of Figure 6.40 has been made up of FormLayout and FlowLayouts. In this case the buttons on the right have matched a vertical flow layout, which is the best option according to the fitness function though it is not, as it leads to misalignment with regard to the widgets on their left. On the other hand, in Figure 6.39 the buttons are placed side by side (in rows) because the whole window has been matched as a VHLayout (a stack of horizontal flow layouts). Hence, the heuristic search has failed to detect the right layout and the VHLayout has been matched, which do not completely fits the layout but it has the highest fitness value.

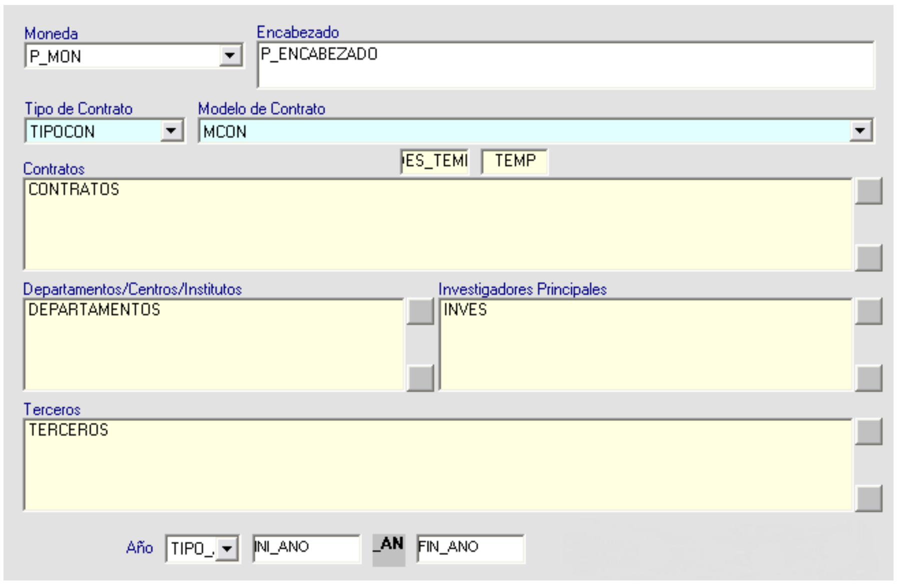


Figure 6.38: Example of an Oracle Forms window.

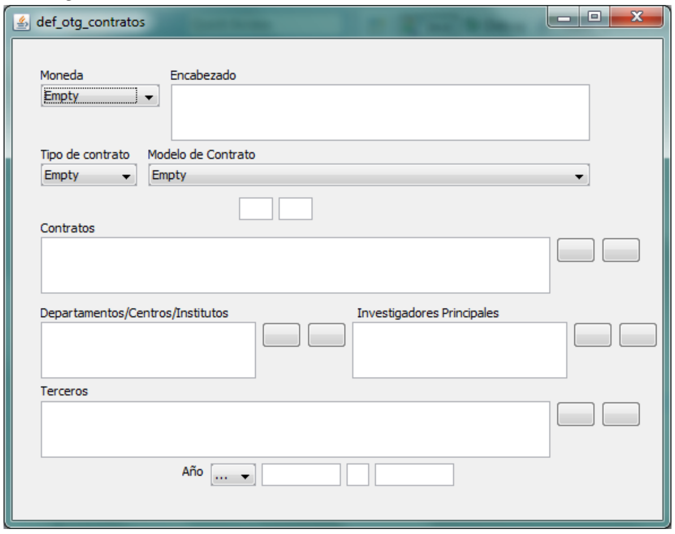


Figure 6.39: Generated window by the first approach for the Oracle Forms window.

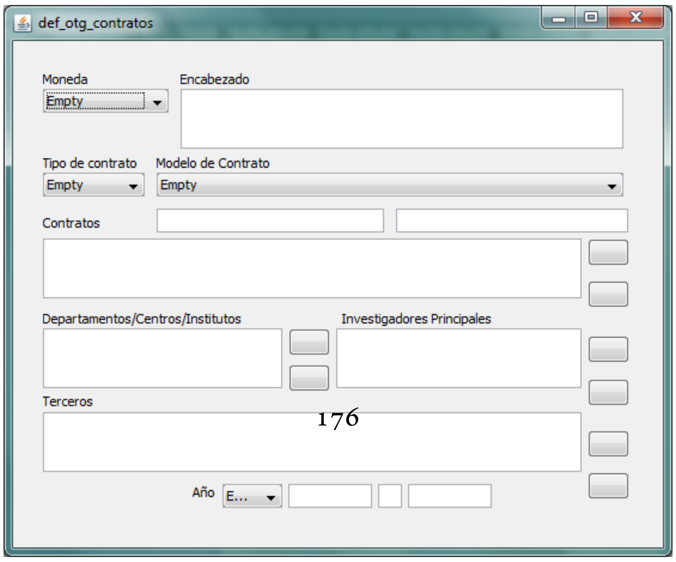


Figure 6.40: Generated window by the second approach for the Oracle Forms window.

In view of the results of the comparison of both approaches, we claim that the exploratory approach is better than the greedy approach. The latter obtained good results with RAD applications due to the type of windows that are found in these applications. However, the exploratory approach works better in RAD applications as well as in other context because of the deep analysis of the relations to return a layout composition, as we had hypothesised before implementing the exploratory approach.

6.8 Conclusions

In this chapter we have presented an algorithm and a data structure (the Tile metamodel) to reverse engineer GUIs with a coordinate-based layout in order to transform them into a representation based on layout managers, which can be used to generate a final GUI based on good practices. The solution proposed is aligned with the reengineering architecture proposed in the previous chapter. The algorithm calculates several alternative layout compositions, and also estimates which one is the best option. Moreover, it allows placing widgets with some degree of misalignment and allows selecting the set of layout managers which will be used in composing the layout. The layout inference algorithm proposed here is more sophisticated than the former and it achieves better results when considering views in general (not only views gathered from legacy systems).

We have also presented a case study to infer the layout of GUIs created with a wireframing tool to generate fluid web interfaces, which is supported by a tool integrated in the Eclipse IDE. The case study revealed that the current approach is somewhat limited in some cases by the maximum deviation parameter used in distance clustering, and that the fitness function needs to take into account aesthetic criteria such as homogeneity in order to get solutions that are better adapted to different screen dimensions. As a whole, results drawn from the evaluation show that our approach is able to perform a good layout inference in most cases (97% of the views are accurately reproduced, 84% of the views are appropriately resized) and 70% users are satisfied with the tool results.

Given that the architecture of the solution in this chapter is essentially the same as the one in the previous chapter, requirements *R1* to *R6* enumerated in Section 4.1 are fulfilled. More specifically, requirements *R1* to *R4* are the consequence of designing a suitable MDE architecture, *R5* is achieved by means of the Region model, and *R6* is covered by the Layout model that expresses the layout as a composition of layout managers.

The functions that compare the relative position of tiles include a parameter that specifies a margin to be considered, so the comparisons are flexible, thus achieving the misalignment tolerance (requirement *R7*). The inference algorithm explores the different combinations of a layout set, and stores all the solutions that it finds, hence it is capable of offering all the solutions found for a given layout set, then covering the requirements alternative solutions (*R8*) and configurable layout set (*R9*).

Contrasting our solution with the related works, we claim that up to date we have not found any work that addresses requirements *R8* and *R9*, but the layout inference algorithms are strictly designed to work with a unique layout type. Like our appproach, the ALM model [3] is able to detect misalignment by means of specifying additional constraints. The differences that we contrasted in the former chapter between our first approach and the related works is still valid in this new approach. In the next table we classify our solution as we did with the state of the art.

Source/target independence	Yes
Tested source technology	Wireframes (WireframeSketcher)
Tested target technology	Web (ZK)
Information extracted	layout composition, alignment, margins, gaps, sizes
Layout model	FlowLayout, BorderLayout, GridLayout, FormLayout
Algorithm type	Backtracking
Implementation technology	MDE
Automation degree	Automated

Table 6.3: Classification of the approach of this chapter

If I have a thousand ideas and only one turns out to be good, I am satisfied.

Alfred Nobel
(Suggested by Marisol Sánchez)

7 Event handler analysis

Migrating legacy code poses many challenges, such as the ones introduced in Section 1.1 for the case of RAD applications. Notably, disentangling the code of the GUI, control, and business logic layers so that the new system has a better separation of concerns, is an important issue. Besides, migration would be facilitated by tools that help to discover non-architectural concerns that are only implicit (and mixed together) in the source code, such as validation, navigation flow or exception handling.

In this chapter we describe the reverse engineering approach we propose for analysing the code of event handlers of RAD applications in order to be able to separate the concerns that are entangled. In Figure 7.1 we can see the part of the GUIZMO architecture we are going to explain in this chapter, as well as the models of the CUI that are involved.

We have defined a RAD environment-independent metamodel to represent the original code in a more abstract form, which is based on a set of primitive operations intended to describe common behaviour of RAD-based code. We will refer to this representation as the *RADBehaviour* metamodel. We have identified a set of programming idioms for Oracle Forms, and we have

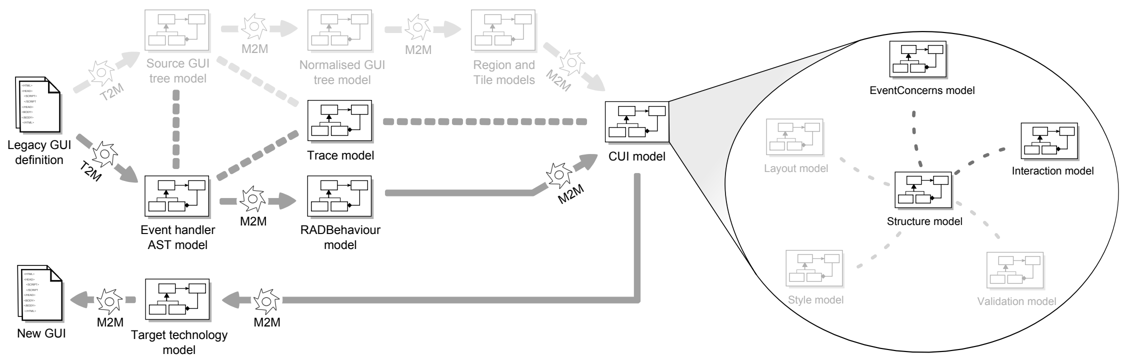


Figure 7.1: Part of the GUIZMO architecture explained in this chapter

implemented an M2M transformation that matches these idioms against Event handler AST model and represents the coincidences in the form of primitives in a RADBehaviour model. This model can be used to achieve the mentioned separation of architectural concerns (GUI, business logic and control), which is materialised in the *EventConcerns* model. We have built a prototype implementation to test this approach. It has been validated with a case study based on a real application written in Oracle Forms, which has been migrated to a client-server web application.

Futhermore, we have also used RADBehaviour models as a basis to identify dependencies among widgets in a view and among different views. We have defined a state-machine-like metamodel to represent these dependencies. This information can be useful, for instance, in a scenario of a migration of a legacy system to the web platform, to generate Ajax code that only refreshes the part of the view that is affected. It can also be used for documentation purposes and for declaratively defining the navigation flow and exception handling in some frameworks (e.g., Struts [110] or JSF [111]).

Consequently, this chapter pursues the goal of separating and making explicit the information of event handlers (*G3*), and covers the following requirements: code abstraction (*R10*), code categorisation (*R11*) and explicit interaction and navigation flows (*R12*). The subsequent section will present the concrete architecture we have devised to deal with event handlers and in the next sections we will explain the different parts of this architecture.

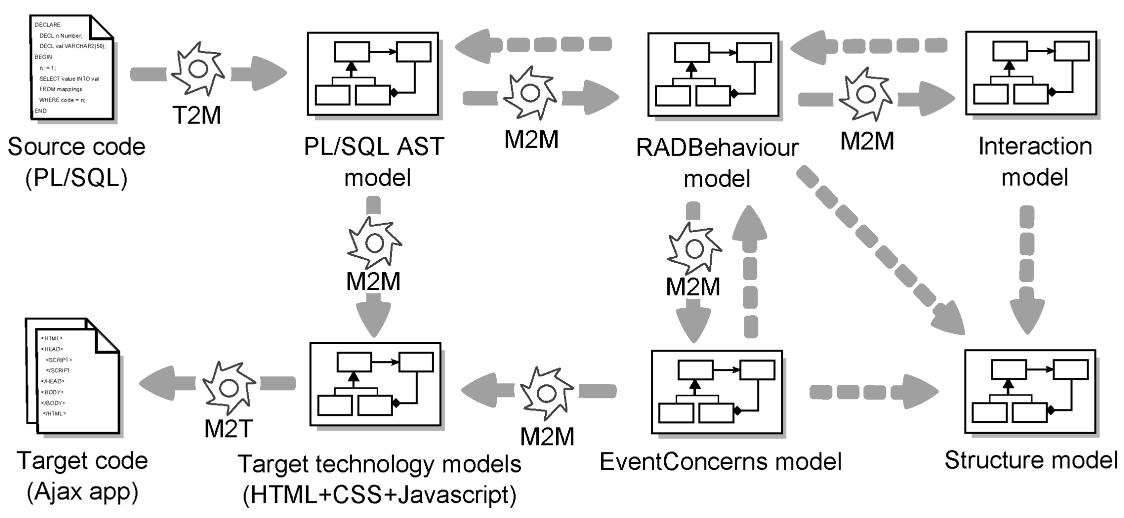


Figure 7.2: Model-based architecture for reengineering RAD-based applications. Solid lines mean transformations and dashed lines are model dependencies.

7.1 Architecture for analysing events

Our model-based architecture is shown in Figure 7.2, exemplified for Oracle Forms as source RAD technology and HTML/Javascript as client-side target platform. Anyway, the approach could be applied to a different RAD technology or target platform likewise. It is based on the Horseshoe model explained in Section 2.1 (rotate 90° to the left Figure 7.2 to see it more clearly), which provides a conceptual framework for the different stages involved in reengineering.

The first step of the approach is the injection of models from the source code of event handlers, in this case PL/SQL code. Given that there is available a grammar and the corresponding Abstract Syntax Tree (AST) metamodel for the PL/SQL language, we selected Gra2MoL (introduced in Section 2.3.3) to accomplish this injection step by writing a T2M transformation that implements the mapping between PL/SQL grammar elements and AST metamodel elements. As a result of applying Gra2MoL, an AST model representing the code of the application triggers (event handlers) is obtained. If we wanted to tackle a different RAD environment (e.g. Borland Delphi 5), we should write the corresponding injector (e.g., a Gra2MoL transformation if a grammar and an AST metamodel are available).

The reverse engineering step starts by transforming the AST model of the event handlers into an intermediate model, named *RADBehaviour*. This model captures the behaviour of the source code in terms of simple primitives which are common in RAD environments, such as read data from a database or write some data in the GUI controls. It is worth noting that the *RADBehaviour* representation is a RAD environment-independent abstraction of the source code. A different model transformation is needed for each RAD technology (e.g. Oracle Forms or Borland Delphi) in order to generate the *RADBehaviour*.

From this model, further reverse engineering can be performed to extract implicit information from the source system. *EventConcerns* is a model derived from *RADBehaviour*, which represents the source code with a kind of control flow graph made up of code fragments. A *code fragment* is a sequence of primitives related to the same category (UI, control, business logic). This is useful to achieve the separation of concerns in the target application. The *Interaction model* is another model obtained from *RADBehaviour* and its goal is twofold: i) define the navigation flow between application windows and ii) show how the events produced in the elements of the GUI (e.g. windows or widgets) affect other elements. This model makes explicit information that is hidden in the source code.

It is important to note that the intermediate models (*RADBehaviour*, *EventConcerns* and *Interaction*) contain cross-references to the model from which they have been derived, in order to trace back the original code when performing forward engineering. In addition, they keep some cross-references to the *Structure model* (e.g., to point to the widgets accessed by the code).

Based on the presented models, restructuring and forward engineering of the original system to a different architecture are possible. In our case, we have experimented regenerating the original application into an AJAX-based web architecture, with HTML/Javascript in the client-side and Java/JPA in the server-side, but other target platforms are possible.

All the M2M transformations in the architecture have been implemented with the RubyTL transformation language [61], and code generation has been performed with the facilities of the AGE environment [103].

The next section introduce the example that will be used through the chapter to support the explanations.

7.2 Running example

The running example is based on the Oracle Forms application for managing public grants that we used in Section 5.8 and that we will use as a case study in this chapter. For illustrative purposes we have simplified and translated into English one of the windows (Figure 7.3) from the

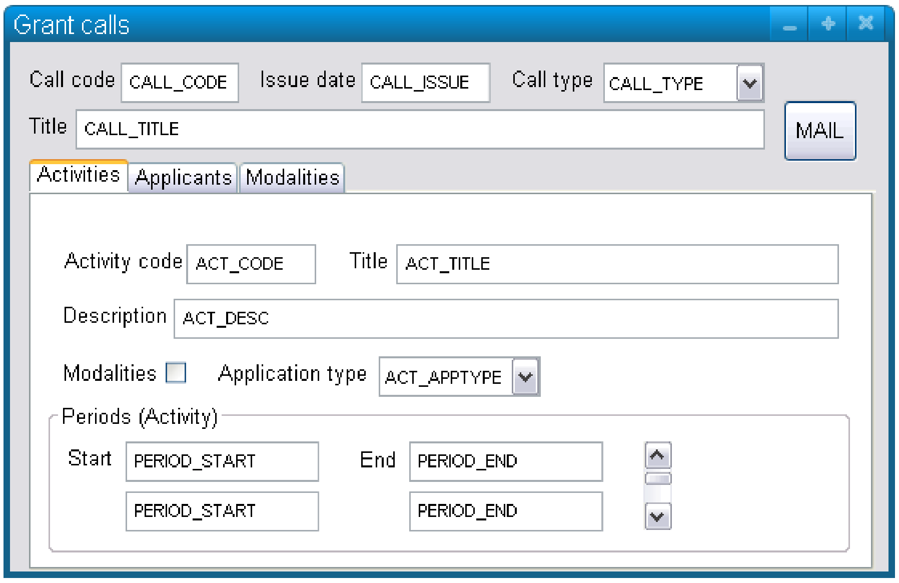


Figure 7.3: Grants example

original application.

In the upper part of the window there are several widgets to display general information about the grant, and a tabbed panel is shown below, where can be seen some information about the activities for which the grant is conceived, in addition to the periods when the grant must take place.

We will focus the example on a simplified event handler associated to the only checkbox in the window (named *ACT_MODALITIES*). This checkbox is used to indicate if an activity can have several modalities. When the checkbox is not checked, the *Modalities* tab must be disabled, but this can only be done if there are not periods for that activity stored in the database. The behaviour of the checkbox is defined in an event handler implemented as a PL/SQL trigger and can be seen in Figure 7.4.

The trigger works as follows. The code that is nested in the *IF* statement is executed if the checkbox *ACT_MODALITIES* is checked. It is worth noting that the *'Y'* value is not a predefined value but is specified in the checkbox properties. To check if there are periods for the current activity, a SQL query is used to store the number of periods in the *periods* variable. If there are periods, a pop-up with a message is displayed. Otherwise the tab page is enabled.

```
IF :ACT_MODALITIES = 'Y' THEN
  SELECT COUNT(*) INTO periods FROM CallPeriods
    WHERE activity = :ACT_CODE;
  IF periods != 0 THEN
    Show_alert('ModalitiesAlert');
    :ACT_MODALITIES := 'N';
  ELSE
    SET_TAB_PAGE_PROPERTY('TABS.MODALITIES',
      ENABLED, PROPERTY_TRUE);
  END IF;
END IF;
```

Figure 7.4: PL/SQL trigger for the checkbox change event

7.3 Representing event handling code

Code analysis is required when event handler migration is needed, which is a tough task. RAD environments are based on different programming languages, so if we intended to restructure several RAD legacy systems to generate new systems, we would have to deal with the source language in all the development phases of the new system.

A solution for this would be to have an intermediate representation that allowed restructuring and forward engineering phases to be independent of the source language. Moreover, programming languages sometimes perform the same taks differently. For example, Oracle Forms uses PL/SQL cursors to retrieve data from database, whereas in Borland Delphi these data are obtained by means of data sources. Therefore, an intermediate language would also be useful to normalise the actions that are done in programming languages.

Hence, we have defined a representation aimed at expressing event handlers in a more abstract form than just the AST model of the program. It acts as technology-independent pivot model, which allows transformations to ignore technology-specific details in the following steps of the reengineering process. It consists of primitive operations (referred as *primitives* from now on) that intend to represent a wide range of code written in event handlers.

In the following section we will first describe the *RADBehaviour* representation for event handling code. Next, we will explain how to obtain *RADBehaviour* models from the PL/SQL AST models. Finally, we will show a RADBehaviour model derived from the example introduced in Section 7.2.

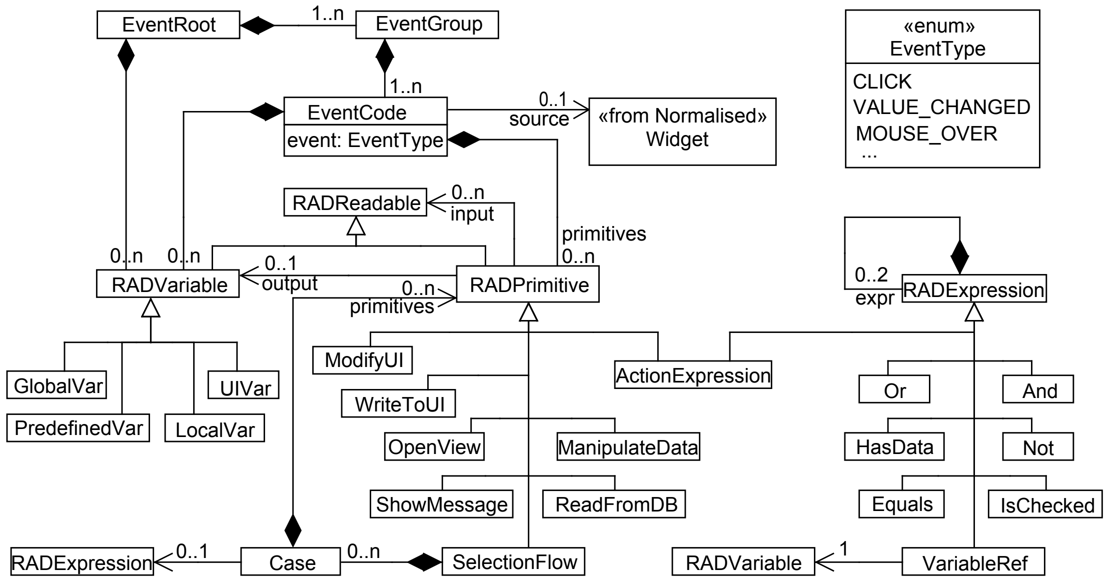


Figure 7.5: Excerpt of the RADBehaviour metamodel.

7.3.1 Metamodel description

The *RADBehaviour* metamodel is presented in Figure 7.5. Its main concept is *EventCode*, which is an abstract representation for the code of an event handler. *EventCode*s include information about the type of event and a reference to the widget that originated the event (it is also possible that an event occurs before the window is displayed and therefore has no widget associated). *EventCode*s are grouped into *EventGroup*s which represent the event handlers that are related to the same application window.

The behaviour of every *EventCode* is expressed in terms of a sequence of *RADPrimitive*s. A *RAD-Primitive* attempts to replace a statement or a set of statements of the original code, defining what they were intended for. The primitives we have identified are listed in Table 7.1. The *input* of a *RADPrimitive* can be another *RADPrimitive* or a variable, so primitives can be composed. The optional *output* must be a variable.

There are several types of variables. *UIVar* is a variable that represents the value contained in a widget. *LocalVar* is a user-defined temporary variable that is just visible in the *EventCode* scope. *GlobalVar* is a user-defined variable that is visible in all the event handlers throughout the application execution. *PredefinedVar* represents any technology-dependant variable that keeps

Table 7.1: RAD primitives

Primitive	Meaning
ReadFromUI	Reads a value from a widget
WriteToUI	Modifies a widget value
WriteToVar	Writes a value in a global or local variable
ReadFromDB	Reads some values from a database
WriteToDB	Writes some values to a database
ModifyUI	Modifies a widget graphical attribute
ManipulateData	Performs an operation on a primitive datatype
SelectionFlow	Selects an execution flow based on some conditions
ExecuteBL	Executes a (user-defined or stored) procedure
OpenView	Opens a specified window
ShowMessage	Opens a modal window (Pop-up)
Leave	Aborts event handler execution

the application status. The root element of the metamodel is *EventRoot,* which contains all the *EventGroups* and *RADVariables.*
Some *RADPrimitives,* such as *SelectionFlow,* need to specify conditions on their application. These conditions are expressed in terms of a simple expression language, whose base class is *RADExpression.* There are two types of expressions, typical expressions such as *Or, And, Equals,* and more complex expressions such as *HasData* that checks if a *RADVariable* has a value.

7.3.2 Deriving a RADBehaviour model

A *RADBehaviour model* is obtained through a M2M transformation that takes an AST model of the event handlers of a RAD application as input. The transformation matches code patterns and generates a *RADBehaviour model* that summarises the behaviour of the original code. As explained above, event handling code is usually repetitive, and there are some idioms that frequently appear. Conceptually, we have identified three types of idioms in RAD applications.

- **Programming language idioms**. They are facilities provided by the underlying RAD programming language to perform recurrent and/or specialized tasks. For instance, PL/SQL allows special versions of SQL DML statements (e.g., SELECT) to be used within regular imperative code, while Delphi uses data sources configured with queries.

- **Community idioms**. They refer to sequences of statements that are widely accepted by the corresponding community as a good way of doing a particular task. For instance, obtaining and traversing a database cursor. These idioms are typically found in technical documentation.

- **Business-dependant idioms**. These are idioms that are originated from the company conventions and practices. Available knowledge about the way of work in business can be expressed with patterns that could highly improve the reverse engineering process.

It is possible that several idioms match the same code snippet, so a priority criterion is followed to decide which of them must be selected. Business-dependant idioms have the highest priority, followed by the community idioms, while programming language idioms have the lowest priority. Note that some of the idioms can be composed of other idioms, and in this case the same priority criterion is followed.

The transformation to derive a *RADBehaviour model* must be implemented specifically for each different RAD environment. At present we have a M2M transformation that supports Oracle Forms PL/SQL. We have separated transformation modules to deal with each type of idiom independently, so that they can be extended or replaced seamlessly. This is particularly useful in the case of business-dependant idioms, that may need to be adapted for a specific company.

Figure 7.6 shows some mappings between PL/SQL idioms and *RADBehaviour* primitives (expressed with a textual notation only for illustrative purposes). We have followed this nomenclature: *x* and *y* are variables (*LocalVar* or *UIVar*), *s1* and *s2* are strings, *p1* and *p2* are predefined properties, *v1* and *v2* are specific values for these properties, *c1* and *c2* are table columns of a table *t*, *c* is a PL/SQL explicit cursor, and *d1* and *d2* are Forms datablocks (a logical group of widgets linked to the database). The meaning of each construct can be easily deduced from the notation.

Mappings *M1* to *M4* are *programming language idioms*, PL/SQL idioms in this case. If a variable name starts with ":" (mapping *M1*), then it refers to a widget value and it must be mapped to a *WriteToUI* primitive, otherwise it would be mapped to a *WriteToVar*. Mappings *M2* and *M3* are library functions to show a message box and to change tab page respectively. It is worth noting that the translation of *M3* consists of creating a ModifyUI action that refers to the corresponding CUI model, which was obtained in a previous transformation.

On the other hand, mappings *M5* and *M6* are *community idioms* as they are recommendations typically followed by PL/SQL programmers, in this case to fill a master/detail relationship and

	PL/SQL idioms	RAD Behaviour mappings
M1	:x := <function>;	**WriteToUI**(*output*=x, *input*=<function>)
M2	Show_alert(s1);	ShowMessage(*message*=s2) (*s2* is a message associated with the *s1* alert)

M3	Set_tab_page_property(x,p1,v1);	**ModifyUI**(input=x,
		property=p2,
		value=v2);
		(*p1*, *v1* are mapped to *p2*, *v2*)

M4	**SELECT** c1 **INTO** x
	FROM t
	WHERE c2 = y;
		WriteToVar(*output*=x,
		input=**ReadFromDB**(*input*=y,
		table=t,
		col=c1,
		cond=c2=y)

M5	**IF** <cond> **THEN**
	x := Find_relation(s1);
	Query_master_details(x, d2);
	END IF;
	(The trigger is in datablock *d1*,
	which is different from datablock *d2*)
		{
		WriteToUI(*output*=y,
		input=**ReadFromDB**(<dbdata>))
		}1..n
		(*y* is a widget of the *d2* datablock,
		<dbdata> is obtained from datablock *d1*
		and the properties of the relation *s1*)

M6	**OPEN** c;
	FETCH c **INTO** x;
	IF (c%FOUND) **THEN**
	<statements>
	END IF;
	CLOSE c;
		SelectionFlow
		Case(*condition*=**HasValue**(
		ReadFromDB(<dbdata>)))
		<primitives>
		(<dbdata> is obtained from the cursor declaration)

M7	x := Name_in (:SYSTEM.TRIGGER_ITEM || '_Value')	**WriteToVar**(*output*=x, *input*=**ReadFromUI**(*input*=y)) (*y* is a widget whose name is the name of the widget associated to the trigger plus '_Value')

Figure 7.6: PL/SQL to RADBehaviour mappings

to manipulate a database cursor respectively. These idioms have a coarser-grained granularity than the previous ones.

Mapping *M7* is a *business-dependant idiom* to write generic event handlers, which is based on Oracle Forms reflective facilities to manipulate the GUI. The *:SYSTEM.TRIGGER_ITEM* special variable contains the name of the widget that is the source of the event that has lead the event handler (trigger in PL/SQL terminology) to be executed. The *Name_in* function takes a widget name as a parameter and returns its value. Therefore, M7 is mapped to a WriteToVar whose input is the value of a widget whose name is the same as the one triggering the event plus "_Value". In this way, using naming conventions(for example having a widget *X* and a related widget *X_Value*) it is possible to write generic event handlers that can be executed for different widgets. Translating this idiom requires embedding the convention into a specific transformation. Besides, the outcome of the transformation is not a single reference to a widget, but it computes every possible widget that could be read (looking for widgets in the CUI model that match the pattern). This uncovers widget relationships that were implicitly specified in the source code.

Finally, some idioms and statements cannot be translated without additional information, because Oracle Forms allows the developer to declaratively specify some behaviour by means of property sheets, i.e., without writing code. For example, there is a function named *execute_query()* that executes a database query defined for the current data block, and fills in all the widgets that are related to this data block and are linked to database columns. Therefore, our model transformation also takes as input this information (gathered from a *Oracle Forms model*, not shown in the architecture diagram due to space reasons) in order to deal with this kind of functionality.

7.3.3 Example

The fragment shown in Figure 7.7 is the *RADBehaviour* representation of the PL/SQL code for the checkbox change event that was introduced in Figure 7.4. The same textual notation as in Figure 7.6 is used.

The outmost *IF* statement, whose condition is that the checkbox is checked, has been replaced for a *SelectionFlow*. We must remark that the *RADBehaviour model* captures explicitly the *IsChecked* condition, while in the original code this condition is not clearly expressed since the checked value (*'Y'* in our case) is not predefined, but defined by the programmer in the checkbox property sheet.

The *Select* statement that counts the number of periods has been replaced with a *WriteToVar*,

```
SelectionFlow
  Case(condition=IsChecked(UIVar(name=ACT_MODALITIES)))
    WriteToVar(output=LocalVar(name=periods),
               input=ReadFromDB(input=UIVar(name=ACT_CODE)
                                table=CallPeriods,
                                isCount=true))
    SelectionFlow
      Case(condition=HasData(LocalVar(name=periods))
        ShowMessage(msg="...")
        WriteToUI(output=UIVar(name=ACT_MODALITIES),
                  input=Literal(value=true))
      Case
        ModifyUI(input=TABS.MODALITIES,
                 property=enabled, value=true)
```

Figure 7.7: RADBehaviour example for the checkbox event

composed of a read from the database (*ReadFromDB*). The inner *IF* statement becomes a *SelectionFlow* which has two cases. The first case shows a message to the user if there are some periods left (*ShowMessage* is enclosed in a *WriteToVar* since a pop-up could allow the user to perform some actions) and sets the checkbox as checked. The second case modifies a predefined property (*enabled*) from the widget *TABS.MODALITIES*.

7.4 Separating concerns

As already explained, dealing with the migration of applications written with a RAD environment requires disentangling GUI, control and business logic. Therefore, our aim is to automatically categorise fragments of code where statements of each fragment are related to the same concern.

In order to achieve this goal, we have defined a metamodel (named *EventConcerns*) that represents fragments and their categories. It is obtained from a *RADBehaviour* model through a model transformation. This transformation is facilitated by the fact that we are not dealing directly with source code, for two main reasons: i) as the source code is represented with a few primitives we just need to check the type of the primitive and sometimes the variables that it uses, so limiting the number of cases that must be handled, ii) given that every primitive represents its input and output explicitly, establishing variable dependencies between primitives can be easily done. Next we will introduce the *EventConcerns* representation.

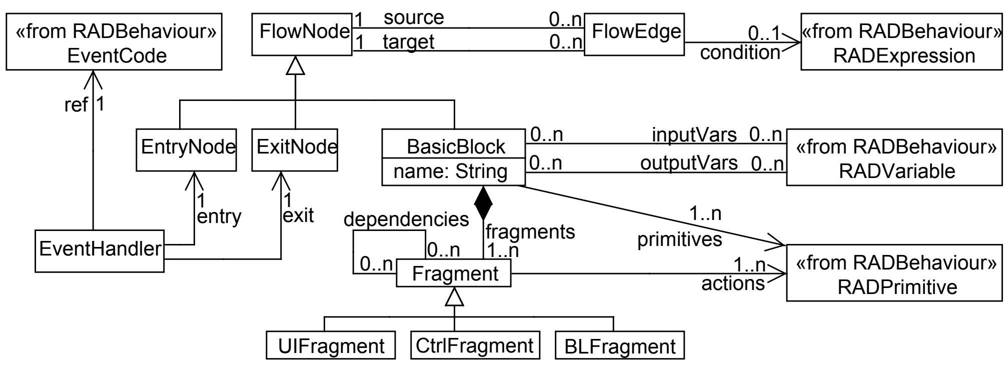


Figure 7.8: Excerpt of the EventConcerns metamodel

7.4.1 Metamodel description

In this representation each event handler is represented as a kind of control flow graph, where the nodes are basic blocks [10] and the edges are execution flows. Interestingly, basic blocks are composed of fragments, where a fragment is defined as a sequence of primitives classified in the same category. We have considered three categories: user interface, control and business logic.

The *EventConcerns* metamodel is shown in Figure 7.8 and defines control flow graphs that are composed of nodes (*FlowNodes*) connected by edges (*FlowEdges*). The types of nodes in the graph are: *BasicBlocks*, *EntryNode* that is a unique node that refers to the first basic block and *ExitNode* that is a unique node that represents the end of the execution flow. A basic block is composed of code fragments that can be of three types:

- *UIFragments* contain primitives that read some data from the interface, or perform a change in the GUI (e.g. show a pop-up, change the value of a text field or change the background colour of the widget that has got the focus).

- *BLFragments* represent code that performs some kind of calculation or information processing (which is commonly done by calling a function that implements the required functionality), or is code related to data persistence.

- *ControlFragments* are used to represent those primitives that are neither user interface nor business logic related and affect the status of the application. For example, set a user identifier in a global variable that is used throughout the user session.

It is worth noting that *Fragment*s keep references to *RADBehaviour* primitives (i.e., instances of the *RADPrimitive* metaclass). Also, for each basic block we keep the set of input variables (*inputVars*) and output variables (*outputVars*), which are obtained by joining the input and output (respectively) of the single primitives. This will be useful to identify variable dependencies among the fragments.

Moreover, each code fragment is given a significant name that is inferred from the statements of the block due to it can be useful later, for example to generate methods from the fragments.

The nodes of the graph are linked by means of edges (*FlowEdge*s) that allow us to navigate through the graph. When there are alternative paths from a node, each edge has a *condition* associated. We have another relationship for code fragments named *dependencies*, which is based on the idea that a fragment can depend on previous fragments. Particularly, when a fragment *f1* assigns a value to a variable that is read in another fragment *f2* that can be reached from *f1*, then *f2* depends on *f1*.

Figure 7.9 shows a graphical rendering of the *EventConcerns model* derived from the *RADBehaviour* model shown in Figure 7.7. In the example there are four basic blocks represented as rounded boxes with two compartments: a upper compartment that shows the descriptive name given to the block and a lower compartment that includes the sequence of fragments for that block. Fragments are represented with roundes boxes that indicate the type of fragment and a descriptive name.

7.4.2 Fragment identification

Next we explain how can we use the *RADBehaviour* representation to obtain *EventConcerns models*. We have split the transformation in several phases that are described next.

7.4.2.1 Creating a control flow graph of fragments

Algorithm 6 describes how to create a control flow of fragments for the primitives of an *EventCode*. It is based on the basic block partitioning algorithm that can be found in [10]. According to that algorithm, a **basic block (BB)** is a sequence of instructions which are executed from the first one to the last one without performing jumps. The first instruction of a basic block is called **leader**. Our algorithm has been split in two functions, namely *identifyBB()* and *identifyFragments()*, which we explain next.

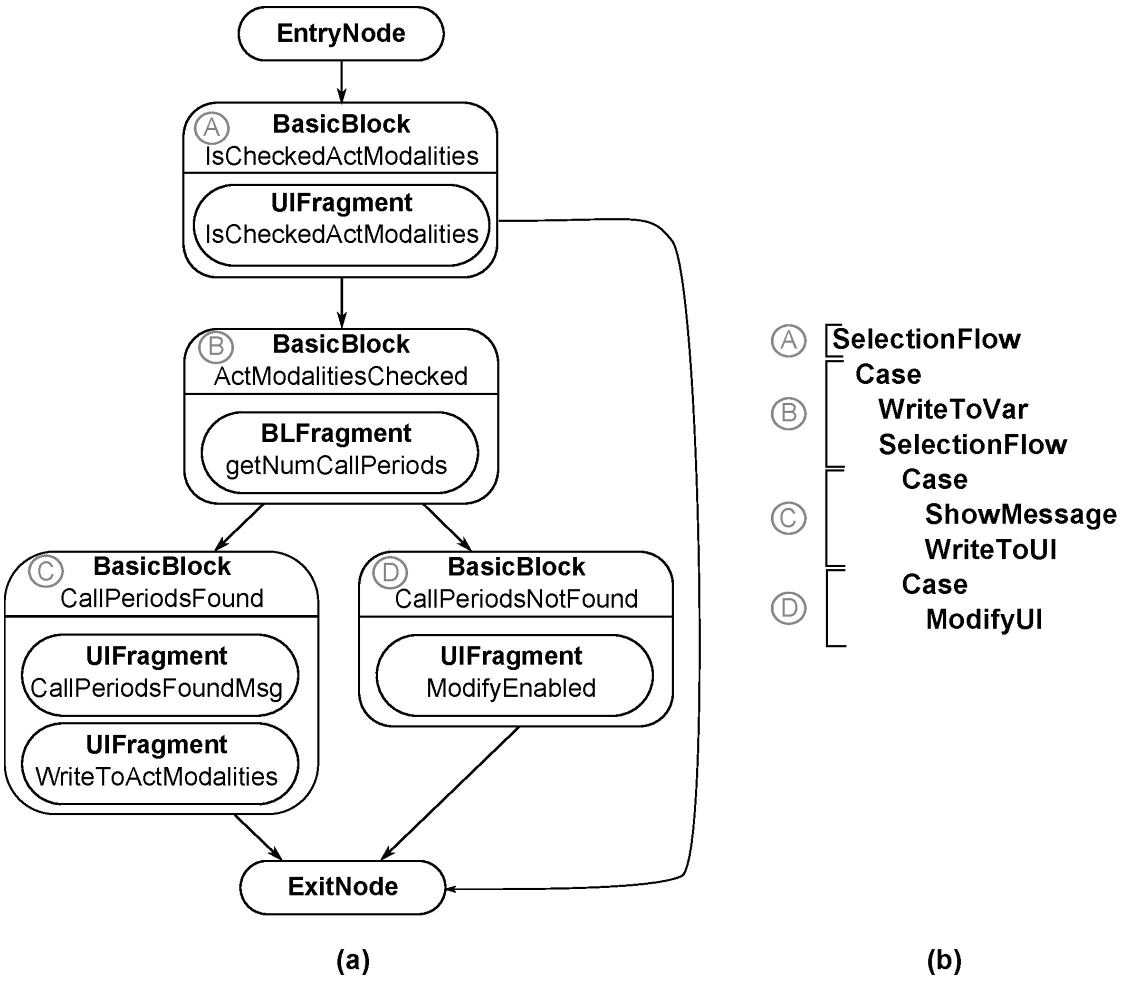


Figure 7.9: EventConcerns model derived from the model in Figure 7.7. Labels *A, B, C, D* are used to show the primitives that originate the basic blocks.

Lines 1 to 4 create an empty set of basic blocks, to which the entry node and the exit node are added. Then the *identifyBB()* function that identifies the basic blocks is called. This function receives the sequence of primitives for which basic blocks are going to be separated, and the set of basic blocks identified so far.

identifyBB() iterates over the primitives. If the primitive *p* is a leader (lines 8 to 13), then the primitives that compose the basic block are selected (line 9) as it is explained in [10] and a new basic block (*bb*) is created (line 10). This basic block is then connected with some of the already visited ones according to the control flow (line 11) . Then, the fragments of the basic block are identified (line 12) and the block is added to the set of basic blocks *BBSet*. If the primitive *p*

(p_1) **WriteToVar**(*output*=**LocalVar**(*name*=X),
input=**ReadFromUI**(*input*=**UIVar**(U))
(p_2) **WriteToVar**(output=**LocalVar**(*name*=Y),
input=**ReadFromUI**(*input*=**UIVar**(V))
(p_3) **WriteToUI**(*output*=**UIVar**(W),
input=**LocalVar**(*name*=X))

Figure 7.10: Fragment identification example

is a *SelectionFlow* (lines 15 to 19), then foreach case of the *SelectionFlow* we apply *identifyBB()* recursively (line 17). Note that it would be more efficient to create the edges of the control flow graph while distinguishing the types of primitives, but we have described the algorithm this way to make it easier to understand.

As we have said, for each basic block we apply *identifyFragments()* to separate the code. This function iterates over the primitives. If there are no fragments, it creates a fragment that contains p (lines 26 to 28). If there are fragments, *findFittingFragment()* tries to find an existing fragment that fits p (line 30). This function works as follows: i) it searches for a *fitting* fragment whose primitives have the same category as p; ii) if p has input variables, then at least one of these variables must be the output of the *fitting* fragment, or if p has an output variable, then this variable must be in the input of the *fitting* fragment; iii) the search starts from the fragment of the primitive that precedes p in order to get the closest fragment that fits. The category of a primitive is determined by its type and input variables. For example, a *WriteToVar* whose input is a *UIVar* will belong to the UI, but if we had *WriteToVar* whose input is a *ReadFromDB*, then the primitive will be tagged as a BL concern.

Let us show a simple example of how the *identifyFragments()* function works. In Figure 7.10 there are three primitives, p_1, p_2 and p_3. Assume that p_1 belongs to fragment f_1 and p_2 belongs to fragment f_2, so:

$$f_1.input = \{U\}; f_1.output = \{X\}$$
$$f_2.input = \{V\}; f_2.output = \{Y\}$$

In this setting the *findFittingFragment()* function would assign p_3 to fragment f_1 because the input of p_3 is contained in the output of f_1, and neither the input nor the output of p_3 appears in the input or output of f_2.

If there exists a *fitting* fragment (lines 31 to 33), then p is added to it, and the output variable of p is added to the output of *fitting*. If p does not fit any fragment (lines 34 to 36), a new fragment f is created and p is added to it. The type of the new fragment f (*UIFragment*, *BLFragment*, *CtrlFragment*) will be the type of the primitive. The output variable of p is added to the output of f, and f is inserted in the current *BasicBlock* according to the creation order.

Figure 7.9 shows the control flow graph that has been built based on the primitives of Figure 7.7. Note that block *C* includes two *UIFragments* due to there are no variable dependencies between the *ShowMessage* and the *WriteToUI*. Our algorithm is not optimum in the sense that it can generate several fragments for UI primitives that refer to the same widget. Anyway, it is not a real problem since contiguous fragments of the same type can be treated as if they belonged to the same fragment when generating code.

7.4.2.2 Giving a descriptive name to the fragments

This is an important step that is accomplished in the *identifyBB()* function of Algorithm 6, and which will be useful to generate code. Particularly, the name of the *BLFragments* can be used to generate the name of the business logic methods. Moreover, giving a meaningful name to the fragments allows capturing the semantics of a fragment code, which is useful as documentation of the original system. However, it is not always possible to infer a useful description for the fragment. The solution we propose to assign names to the fragments is based on heuristics. Next we describe four heuristics we have devised.

In many cases, *BLFragments* perform some operations on the database after reading the value of some widgets. In these cases, we give a name to the fragment by looking at the database access primitives and ignoring the rest of them. For example, the *ReadFromDB* primitive that appears in Figure 7.7 comes from the *SELECT* statement in Figure 7.4, and the name generated from this primitive is *getNumCallPeriods*, as can be seen in the *BLFragment* of block *B* in Figure 7.9 (note the infix *Num* that indicates that the operation returns a number). When we have that a *BLFragment* invokes a function or procedure, we take the first invocation as a name.

A *UIFragment* often refers to just one primitive, so in that cases we obtain the name based on the primitive. For example, a fragment with a *WriteToUI* primitive is named with *WriteToX* where *X* is the widget that is being written, for example the second *UIFragment* in block *C* is called *WriteToActModalities*.

Algorithm 6 Algorithm for identificating of basic blocks (BB) and fragments.

```
1: $BBSet \leftarrow \{\}$
2: $add(BBSet, createEntryNode())$
3: $add(BBSet, createExitNode())$
4: $identifyBB(Primitives, BBSet)$                              ▷ Primitives of an EventCode
5:
6: function IDENTIFYBB($Primitives, BBSet$)
7:     for all $p \in Primitives$ do
8:         if $isLeader(p, Primitives)$ then
9:             $BBPrimitives = getBBPrimitives(Primitives, p)$
10:            $bb \leftarrow createBB(BBPrimitives)$
11:            $createEdges(bb, BBSet)$
12:            $identifyFragments(bb)$
13:            $add(BBSet, bb)$                                  ▷ BBSet is modified in every use
14:        end if
15:        if $p.type = SelectionFlow$ then
16:            for all $c \in p.Cases$ do
17:                $identifyBB(c.Primitives, BBSet)$
18:            end for
19:        end if
20:    end for
21: end function
22:
23: function IDENTIFYFRAGMENTS($bb$)
24:    $bb.Fragments = \{\}$
25:    for all $p \in bb.Primitives$ do
26:        if $isEmpty(bb.Fragments)$ then
27:            $f \leftarrow createFragment(p)$
28:            $add(bb.Fragments, f)$
29:        else
30:            $fitting \leftarrow findFittingFragment(bb.Fragments, p)$
31:            if $\exists fitting$ then
32:                $add(fitting.Actions, p)$
33:                $add(fitting.OutputVars, p.Output)$
34:            else
35:                $f \leftarrow createFragment(p)$
36:                $add(bb.Fragments, f)$
37:            end if
38:        end if
39:    end for
40: end function
```

The name of a *BasicBlock* that starts with a *SelectionFlow* is the name of the condition of the *Case*, taking into account previous primitives that are referred by this condition. For example, in block *A* the condition of the *SelectionFlow* is an *IsChecked* expression that does not depend on previous primitives (actually the *SelectionFlow* is the first primitive), so the inferred name for the block is *IsCheckedActModalities*.
The name of a *BasicBlock* which is the first block in a branch uses the branch condition and previous primitives that are referred to this condition. For example, the name of the block *C* is *CallPeriodsFound*, which is derived from the condition and the *WriteToVar* that precedes the *SelectionFlow*.

7.4.2.3 Setting dependencies among fragments

Code fragments often depend on some values that where calculated or retrieved in other fragments which were executed before, so it is interesting to explicitly capture these relationships. To know the dependencies, we must identify the set of input and output variables for each fragment, which is easily done by using *input* and *output* attributes of the primitives. Then we set the dependencies according to this criterion: A fragment *f1* depends on another fragment *f2* if the input set for *f1* includes some variables from output set of the fragment *f2*.

7.5 Generating layered code

In this section we will outline the last part of the architecture proposed in Figure 7.2, that is, how a *EventConcerns model* can be used to generate a part of the new system.
Separating the different concerns of the legacy system allows us to migrate the application to a new platform and technology, especially to some web technologies where the separation between UI and business logic is imposed.
We have built a chain of M2M and M2T transformations that migrates PL/SQL event handlers to a heavy-client, two tier architecture, where the GUI is defined with HTML/Javascript/jQuery which invokes a REST service made up of business logic fragments. We have defined several metamodels to represent the target architecture, which comprise the several technologies involved: HTML and jQuery (Javascript) for the client side and Java for the server side.
The M2M transformation takes the AST model of the PL/SQL code and the EventConcerns model as input and outputs one or more models representing the target artefacts. This trans-

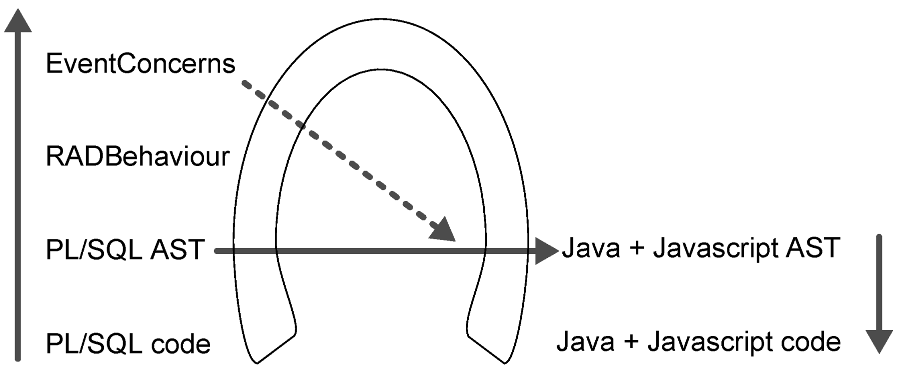


Figure 7.11: Horseshoe model applied to the separation of concerns

formation is performed between snippets of PL/SQL to either Javascript or Java. In order to decide which parts of the PL/SQL must be transformed to Javascript (UI), the EventConcerns model is queried. In fact, this transformation is explicitly parameterised [112] by the Event-Concerns model, as it is represented in Figure 7.11. It is the latter which actually drives the transformation in the sense that it is used by the transformation rules to disentangle the original code by changing and relocating the content of a fragment according to its category. The cross-references between a fragment and the *RADBehaviour model*, and between it and the PL/SQL AST model are essential to achieve this effect, as they allow navigating from the EventConcerns to the PL/SQL model.
Listings 7.1 and 7.2 show the translation of the original code of the running example (Figure 7.4). There are three important issues about the translation which are worth mentioning, namely:

- First of all, it is possible to some extent generate idiomatic code because the *RADBehaviour* model contains certain semantic information. For instance, line 2 checks whether a checkbox is checked or not in idiomatic jQuery.

- Secondly, the generated UI code in Javascript has the same shape as the original PL/SQL code, except business logic fragments, which are translated to a remote AJAX call. In Javascript a callback is executed when the result is available, so every fragment (UI or BL) which depends on the transformed logic fragment is put within such a callback (lines 7-14). Currently, we only support synchronous calls, but we intend to develop another transformation which will be able to perform asynchronous calls based on the dependencies among fragments.

- Finally, each business logic fragment is mapped to a Java method which connects to the database and performs the required logic. The input parameters of this method are the UI variables that the fragment depends on, and the returning value is a JSON object made up of the variables (UI or local) used by other fragments that depend on this fragment.

```
var periods;
if ( $('#act_modalities').is(':checked') ) {
  $.ajax({
    url: "getNumCallPeriods/" + $('#act_code').val(),
    dataType: "json",
    async   : false,
    success : function( result ) {
      periods = result.periods
      if ( periods !== 0 ) {
        alert('No␣periods');
        $('#act_modalities').attr('checked', true);
      } else {
        $('#act_modalities').tabs('enabled', 1);
      }
    }
  });
}
```

Listing 7.1: Event handling code rewritten in Javascript

```
@Produces("application/json")
public class Service extends BaseResource {
  private static EntityManager em = ...;

  @GET @Path("grants/getNumCallPeriods/{act_code}")
  public Representation getNumCallPeriods(
         @PathParam("act_code") String code) {
    Query q = em.createQuery("SELECT␣COUNT(*)␣FROM␣CallPeriods␣WHERE␣activity␣=␣:ACT_CODE
          ");
    q.setParameter("ACT_CODE", code);
    return new JSONObject().put("periods", q.getSingleResult());
  }
}
```

Listing 7.2: Entangled business logic moved to REST service

7.6 Capturing dependencies among the GUI elements

RADBehaviour contains implicit information about the dependencies that exist between the elements in the window, for example, which windows can be reached directly from certain window, or which widgets are affected by a change in the value of another widget. An *Interaction model* is a model derived from *RADBehaviour* that captures these kinds of interactions explicitly. It can be considered as a view of a RADBehaviour model, since they gather a subset of the information from the latter one and arranges it in such a way that is useful to perform some tasks of forward engineering and program comprehension. We distinguish two main uses for this model.

The first use is to exploit it as high level documentation, such as interaction diagrams. The information is presented in a readable way so it can be utilised as a guideline to lead a manual migration process. It could support a semi-automatic process in which event handler skeletons can be automatically generated, and some hints can be included as comments to ease the migration.

The second use consists of taking advantage of the representation to generate GUI-related artefacts. In particular we have identified three possible artefacts. The first possibility is to generate navigation flow files. This is, since the Interaction model makes explicit the navigation flow between application windows that is hidden in RADBehaviour models, it can be used to generate the page navigation configuration files for web applications such as Struts or Java Server Faces. The error pages (exception handling) can be also specified.

The second possibility in which the model is useful is for generating web interfaces without refreshing the whole page. Widget updates that do not require data (e.g., enabling a panel after checking a checkbox) can be performed with Javascript, and widgets demanding data from the server can be updated using Ajax. This task could be done with the RADBehaviour, but since in the Interaction model the events are expressed much clearer and simpler than in the source code, it is much easier to identify what type of implementation (server-side, Javascript, Ajax) is suitable for an event handler.

The third possibility is related to the widget interactions. A code generator for the event handlers could be derived from RADBehaviour. This generator would create one event handler for each one in the original code and every handler would modify the properties or fill in all the widgets that would be affected. However, a better design would be to have that every widget is in charge of updating itself, instead of other widgets can modify it. This would promote that widget functionality and dependencies are separated, what leads to a better maintenance, especially when complex widgets (e.g. tree views) are involved. In RADBehaviour dependencies between widgets are not clearly identified, whereas the Interaction model is focused on this aspect. Thus, a Interaction model directly shows which widgets are publishers of events

and which widgets are subscribers of event, so implement either the Observer or the Message Broker patterns is eased.
Note that the *Interaction model* makes some data more accesible than *RADBehaviour*, and can ease generating code for some frameworks. Both, *Interaction* and *RADBehaviour* can be used together to design the new system, given that each model is focused on different aspects of the source system.

7.6.1 Metamodel description

The *Interaction metamodel* is shown in Figure 7.12. Basically the metamodel represents a graph where each node can have a nested subgraph. There are two types of nodes: *GUIFragmentNodes* which represent windows or composable parts of windows (such as portlets), and *WidgetNodes* for representing widgets.
Every *Interaction* is related to the *source* node that originated the interaction, and the *target* nodes that are affected by the *Interaction*. An *Interaction* is produced when a certain event is triggered (*trigger* attribute) and some guard conditions (*condition* attribute) are fulfilled (all the conditions must be fulfilled as if there were join with *And*). In this case, some *actions* are performed on the target nodes. Note that the conditions are references to some conditions in the *RADBehaviour model* which originated the *Interaction model*, and the actions to be performed when an *Interaction* occurs are also defined in the *RADBehaviour model*.
These nodes can be connected through two types of *Interactions* which differ in the type of target. One type is for *Interactions* that cause a change in the window. An example of this type can be found when a user presses a button and this produces that the current window is closed and a different window is displayed. This is the interaction that has targets of type *OuterTarget*. The second type of *Interactions* is for expressing that a change in one widget has an effect in another widget. For example, a user introduces its name in a text widget and automatically another text widget is filled with a user identification number. *InnerTarget* is the type of the targets of these Interactions.

7.6.2 From RADBehaviour to the Interaction model

In this section we will outline the M2M transformation that takes the *RADBehaviour model* and gets an *Interaction model*. We will explain how we get the *GUINodes* and the *Interactions* by

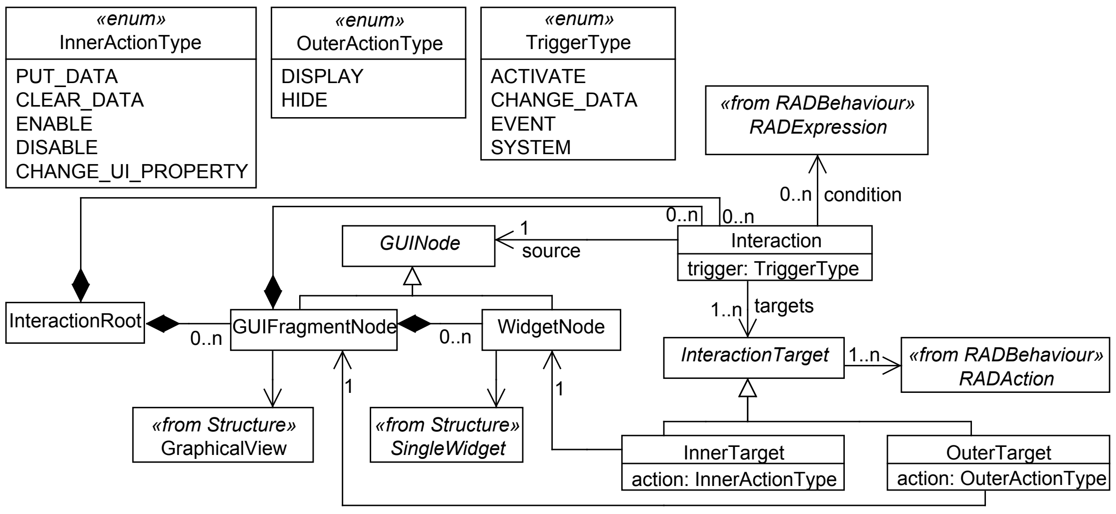


Figure 7.12: Interaction metamodel

means of Algorithm 7.

Lines 2 to 28 create *GUINodes* for widgets that are involved in event handlers (i.e., *EventCodes*). Concretely, nodes are created for the widgets which are the source of an *EventCode* (lines 3 to 9), and nodes are created for the widgets that appear in the primitives of an *EventCode* (lines 11 to 22). If the widget is a *View*, then a *GUIFragmentNode* is instantiated. Otherwise a *WidgetNode* is built. After creating all the nodes, *WidgetNodes* are nested in the corresponding *GUIFragmentNodes* (lines 24 to 27).

Lines 30 to 48 are intended to create links (*Interactions*) between nodes. An *effect primitive* is a type of primitive that may produce a change in a view, this is, *OpenView*, *ShowMessage*, *WriteToUI* and *ModifyUI*. An *effect block* is a basic block that contains at least one effect primitive. For each EventCode, the effect blocks are retrieved and iterated (line 31). Note that the effect primitives of an effect block are all executed in a block under the same conditions, i.e., either all of them or none of them are executed. Then, for each effect block, an interaction will be created, whose *source* widget will be the *source* of the event handler (line 33) and *conditions* will be a join (with the *And* operator) of all the conditions that wrap the basic block from the beginning of the event code (line 35). For example, in the primitive code of Figure 7.7, the wrapping conditions of the *ShowMessage* primitive are *IsChecked* and *HasData*.

For each effect primitive in an effect block (line 36), an *InteractionTarget* is created according to the type of primitive: if it is a *OpenView* or a *ShowMessage* primitive, an *OuterTarget* is created,

Algorithm 7 Algorithm to generate the Interaction model.

```
$NodeSet \leftarrow \{\}$
**for all** $e \in getAll(EventCode)$ **do**
    **if** $\neg created(e.source)$ **then**
        **if** $e.source.type = View$ **then**
            $NodeSet \leftarrow createGUIFragmentNode(e.source)$
        **else if** $e.source.type = SingleWidget$ **then**
            $NodeSet \leftarrow createWidgetNode(e.source)$
        **end if**
    **end if**

    **for all** $p \in e.Primitives$ **do**
        $Widgets \leftarrow getWidgets(p.input) \cup getWidgets(p.output)$
        **for all** $w \in Widgets$ **do**
            **if** $\neg created(w)$ **then**
                **if** $w.type = View$ **then**
                    $NodeSet \leftarrow createGUIFragmentNode(w)$
                **else if** $w.type = SingleWidget$ **then**
                    $NodeSet \leftarrow createWidgetNode(w)$
                **end if**
            **end if**
        **end for**
    **end for**

    **for all** $n \in NodeSet.(n.type = WidgetNode)$ **do**
        $container \leftarrow findContainer(n)$
        $add(n, container)$
    **end for**
**end for**

**for all** $e \in getAll(EventCode)$ **do**
    **for all** $block \in getEffectBlocks(e)$ **do**
        $i \leftarrow createInteraction()$
        $i.source \leftarrow e.source$
        $i.trigger \leftarrow mapEvent(e.event)$
        $i.Conditions \leftarrow getConditions(block)$
        **for all** $p \in getEffectPrimitives(block)$ **do**
            **if** $p.type = OpenView \lor p.type = ShowMessage$ **then**
                $t \leftarrow createOuterTarget()$
            **else if** $p.type = WriteToUI \lor p.type = ModifyUI$ **then**
                $t \leftarrow createInnerTarget()$
            **end if**
            $t.target \leftarrow getTarget(p)$
            $t.action \leftarrow mapAction(p)$
            $t.Primitives \leftarrow findDependencies(p)$
            $i.target \leftarrow t$
        **end for**
    **end for**
**end for**
```

which represents a change in the flow of views; if the primitive is a *WriteToUI* or a *ModifyUI* then an *OuterTarget* is created, which represents a modification in the current view. The *findDependencies* function in line 44 gets all the primitives that are placed before the given primitive and affects the result of this primitive. For example, if we apply *findDependencies(p)* and there is a *WriteToVar* primitive prior to p that writes a value in a variable X that is used in p, then the *WriteToVar* is added to the result. Finally, the *target* is added to the *Interaction* (line 45)

7.6.3 Example

In Figure 7.13 we show an *Interaction model* for the event handlers associated to the window presented in Figure 7.3. The graphical notation which we have used is the following. Rounded-boxes with two compartments represent application windows, for example *GRANT_CALLS-_WINDOW* or *MAIL_WINDOW*. Rounded-boxes without compartments represent widgets, for example *CALL_CODE* or *ACT_MODALITIES*. Arrows represent interactions with the following notation: *event [condition] / actions*. For example, from *CALL_CODE* there is an interaction to *PERIOD_START* and PERIOD_END. In this case the event is *Change* (is a predefined event), the condition is *HasData(CON_CODE)*, and it has two actions: *display*s a value in *PERIOD_START* and *display*s a value in *PERIOD_END*. In some cases an interaction is performed when a window is displayed, so in these cases the arrow starts in the window, such as the arrows that start in *GRANT_CALLS_WINDOW* and ends in *MODALITIES*.

In the upper-right part of the diagram we can see two interactions whose source is the checkbox *ACT_MODALITIES* and are related to RADBehaviour example shown in 7.7. Each interaction is related to a *Case* of the nested *SelectionFlow*. From the first *Case*, an interaction with two targets has been generated. One target is generated from the *WriteToUI* primitive, aims at the *MODALITIES* widget and is tagged with the *PutData* action. The other target is generated from the *ShowMessage* primitive, aims at the *POP-UP* generic widget and is tagged with the *Display* action. Note that both targets belong to the same interaction, so they have in common the event that produces the interaction (a *Change* of data in a widget our case), and the guard condition that has been obtained from the join of the conditions of the *SelectionFlows* in which the *effect primitives* are nested. The second interaction also takes place when there is a change in the value of the *ACT_MODALITIES* widget (and the condition guard is fulfilled), and produces a *ChangeUIProperty* in *MODALITIES*, which is the unique target of this interaction.

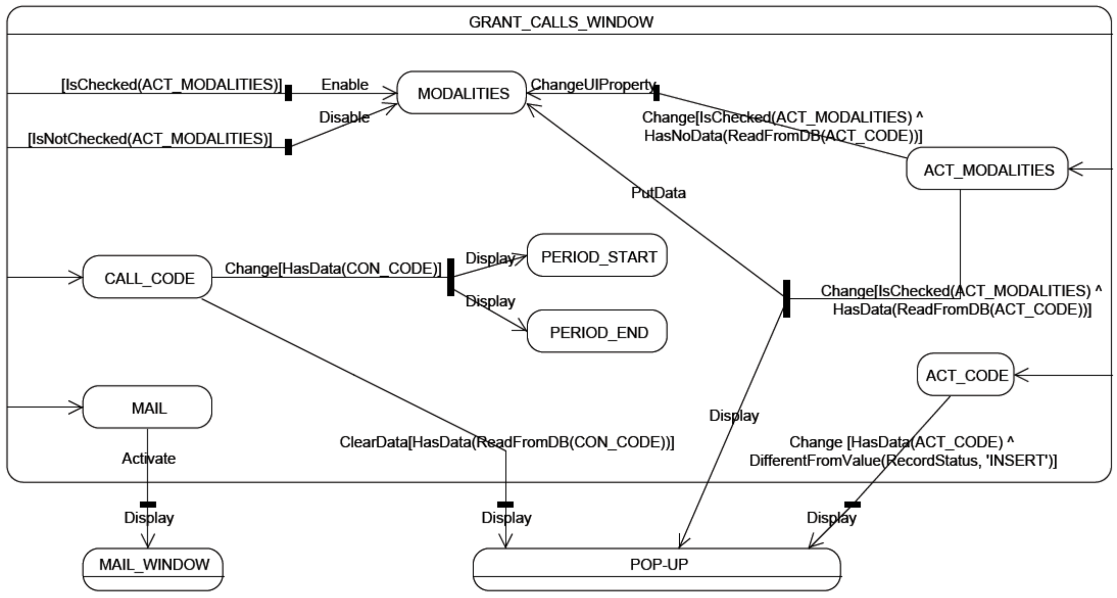

GRANT_CALLS_WINDOW
[IsChecked(ACT_MODALITIES)]
Enable
MODALITIES
ChangeUIProperty
[IsNotChecked(ACT_MODALITIES)]
Disable
Change[IsChecked(ACT_MODALITIES) ^
HasNoData(ReadFromDB(ACT_CODE))]
ACT_MODALITIES
PutData
CALL_CODE
Change[HasData(CON_CODE)]
Display
PERIOD_START
Display
PERIOD_END
Change[IsChecked(ACT_MODALITIES) ^
HasData(ReadFromDB(ACT_CODE))]
ACT_CODE
MAIL
Display
Activate
ClearData[HasData(ReadFromDB(CON_CODE))]
Change [HasData(ACT_CODE) ^
DifferentFromValue(RecordStatus, 'INSERT')]
Display
Display
Display
MAIL_WINDOW
POP-UP

Figure 7.13: Interaction model for the event handlers of the window shown in Figure 7.3

7.7 Evaluation of the approach

In order to assess the utility of our approach we have performed a case study reusing the Oracle Forms application for managing research projects that was introduced in Section 5.8. Around 11,000 lines of code (LOC) were evaluated (comments are not counted), what indicates a medium-high complexity.

We have executed the complete reverse engineering process for the application and we have manually inspected the models in order to count the LOC[1] correctly matched and classified. For the *RADBehaviour* model, we count the LOC that have been successfully matched, comparing the idioms matched with the expected ones. For the *EventConcerns model* we count the LOCs that have been classified in each category, in order to assess the amount of code that our approach is able to relocate. The extraction of interactions has been tested with only a few windows so it cannot be considered a reliable evaluation but a proof of concept, and therefore it will not be commented in this section. Despite that, the evaluation of the code abstraction give us an idea of the correctness of the extraction of interactions since it strongly depends on how good the RADBehaviour model represents the semantics of the source code, and therefore it is

[1]Tokens like *begin* or *end*, and variable declarations are not counted.

Table 7.2: RADBehaviour evaluation

LOC of idioms matched / total LOC	95.65%
LOC mapped OK / total LOC	83.04%
LOC of matched programming idioms / total LOC	36.67%
LOC of matched community idioms / total LOC	56.45%
LOC of matched business idioms / total LOC	6.88%

expected that the correctness of both models will be rather similar.

7.7.1 Evaluation of the code abstraction

The results of the assessment of the code abstraction algorithm are shown in Table 7.2. *LOC of idioms matched* is the percentage of LOCs out of the total that have been matched with some idiom. However, not all LOCs that are matched are mapped properly, so *LOC mapped OK* is a measure of the amount of code whose behaviour has been captured right.

As can be seen, almost all LOCs match some idiom. This is a consequence of having fine-grained idioms (programming language idioms) that match almost everything that coarse-grained idioms (community and business idioms) cannot. This avoids the need for writing idioms for every built-in function, and it offers a migration option when some coarse-grained idioms have not been identified. For example, there is a built-in function that copies a value to a given variable if the current value of the variable is NULL. Since we do not have a specific mapping for this function, it is automatically transformed into a *ExecuteBL*, which is a wrong mapping. When there are statements that are not matched, they are notified to the user.

As can be seen, almost 17% of LOC are mismatched. In our case, the majority of the fails are due to the fact that we do not deal with PL/SQL exceptions, and because of some specific Forms functions that are not mapped properly. We can conclude that the set of primitives identified in *RADBehaviour* is enough to capture the basic behaviour of the application.

The second part of Table 7.2 shows the percentage of each type of idiom that has been matched out of total of correct matches. This reinforces the idea that RAD applications are programmed based on a set of more or less fixed idioms that are used throughout the code given that approximately 63% of the code are coarse-grained idioms (i.e., community and business idioms).

Table 7.3: EventConcerns evaluation

LOC classified OK	86.10%
LOC classified as BL	15.87%
LOC classified as Ctrl	4.80%
LOC classified as UI	79.33%

7.7.2 Evaluation of the separation of concerns

Table 7.3 shows the amount of LOCs that has been classified in each category (user interface, control and business logic). *LOC classified OK* shows the number of LOCs out of the total that have been categorised properly. Interestingly, the success percentage in this case (86.10)% is slightly higher than the percentage of code well mapped when obtaining the *RADBehaviour* (83.04%, *LOC mapped OK* in Table 7.2). This is due to the fact that some original statements are mapped to wrong primitives, but by chance they belong to the right category, so they are classified correctly. However, this may lead to generate a wrong piece of target code (i.e., around 3% of the generated code is wrong). We are looking into ways of detecting this corner case.
It can be seen that a certain amount of the code (20.67%) should be relocated to achieve separation of concerns, what shows that RAD applications are tightly coupled, and that our approach facilitates identifying fragments related to each concern and automatically relocating them.
With regard to code classified as UI (79.33%), it is translated in a straightforward manner to the new application. However, we have estimated that around 18% out of UI code is in charge of performing interactions among widgets or performing a change in the navigation flow of the application, and could be moved to a different module if we intended to decouple the interactions among widgets. Based on the *RADBehaviour* representation it is possible to identify those widgets interacting with other GUI elements, so enabling further separation of concerns.

7.8 Conclusions

In this chapter an approach to reverse engineer event handlers of applications developed with RAD environments has been presented. The aim is to separate the different concerns that are tangled in the event handlers of those applications, that is, the *G3* goal that was presented in Section 1.2. As a result of the reverse engineering process we have obtained two kinds of models: the EventConcerns model and the Interactions model. The former is used to separate the architectural concerns (e.g., MVC layers) of an application and then improve the quality of the code in the new system. The latter serves to separate the navigation flow and widget interaction concerns, and concretely can be used to: generate navigation flow descriptions (e.g., for JSF), detect asynchronous updates if the application is migrated to the web platform, or for graphically documentating the source system.

With the aim of getting the EventConcerns and Interaction models, source code had to be analysed. However, analysing code of a programming language is a tough task, since there are a lot of different ways to perform the same effect. For example, changing the order of independent statements or introducing local variables to store temporary results of functions or database queries are two ways of modifying the source code while preserving its semantics. Usually standalone statements are meaningless, but they are part of more complex structures that have a concrete purpose (which developers use to indicate with code comments). Therefore using a simpler representation (RADBehaviour) that makes explicit the intention of portions of code greatly eases the manipulation of code and simplifies the anlysis. Given the specific nature of event handlers of RAD applications that usually perform the same tasks in a more or less similar fashion, pattern matching is able to detect complex code structure (idioms) in most of the code (96%), with a success rate of 83%.

The RADBehaviour representation greatly facilitated the achievement of the EventConcerns and Interaction models since it was much simpler to analyse that the AST tree of the PL/SQL language. We assessed the separation of architectural concerns with a migration of the Oracle Forms application that we used in previous chapters to the web platform with Ajax, obtaining a 86% of accuracy. A lesson learned in the case study was that when accomplishing a migration, raising the abstraction level of the code may be useful, but the AST representation of the source code is still required in many cases, as the abstract representation supresses details that are needed to perform a complete migration of the source code. Hence, traces to the source artefacts must be kept to perform this.

In relation to the requirements of Section 4.1, requirement *R10* can be tackled by means of the RADBehaviour model, requirement *R11* is covered by the EventConcerns model and *R12* is achieved by the Interaction model. We have not found related work that deals with event handlers with an abstraction of the code such as our RADBehaviour model. We believe that some kind of preprocessing such as our RADBehaviour is needed to shorten the gap between the code and its semantics. Regarding to the separation of code concerns, in [11] authors addressed this separation by marking code by hand. Different to them, our approach is fully automated and extracts the category of code snippets by applying pattern matching, as we stated in requirement *R3* (automation). The Interaction model was inspired in many works about GUI analysis that extracted some kind of state machine from the code [7] [8] [31]. Our Interaction model is somewhat similar to the Orchestration model presented in [113][2] and the main difference with our work is that we intend to be generic and therefore we do not include information about grouping widgets in Ajax pages.

Table 7.4 shows the classification of our work as we did with the works in the state of the art. The type of code analysis that we have applied is static analysis, given that the source PL/SQL code could be extracted from the binary artefacts. However, RAD environments have the possibility of using reflection (e.g., the *Name_in()* function in Oracle Forms), which some developers use to create code that can be copied and pasted in different event handlers, so dynamic analysis would be required to fully analyse reflective calls. Nevertheless, to our experience static analysis was enough to extract most of the behaviour of the legacy application, but there may be applications which use reflection in such a way that is not possible to analyse statically. In those cases a hybrid analysis would be the best option.

Source artefacts	Legacy code (PL/SQL)
Information extracted	EventConcerns and Interaction models
Goal	Migration, quality improvement, documentation
Analysis type	Static

Table 7.4: Classification of the approach of this chapter

[2]This work was excluded from the state of the art since it is not a reverse engineering approach.

If you can meet with Triumph and Disaster and treat those two imposters just the same.

Rudyard Kipling
(Suggested by Fernando Molina)

8 Conclusions

User interfaces are an important part of software systems. Nowadays users do not only expect from an application that some functionality is available, but many other qualities. For example, in e-commerce applications it is vital for the user interface to be appealing to attract people's attention and encourage them to purchase, and at the same time making it accessible from different devices (e.g., smartphones) without diminishing the user experience. Modern GUI frameworks technologies, for example the combination of web frameworks such as jQuery [114] and JSF [111], support programmers in the challenging task of implementing GUIs by offering powerful graphical options and including some code facilities to improve the maintainability and extensibility of the application. However, a great deal of applications that were created in the past do not take advantage of the new GUI technologies that enhance interaction and system quality, which pushes companies to address their migration. This thesis has been aimed at providing a model-driven reverse engineering approach that supports the migration of RAD applications to modern platforms and technologies.

This chapter finalises this manuscript. Some conclusions and reflections will be distilled, future work will be outlined and the results in terms of publications, contracts, projects and research stays will be succintly presented. Next, we will show a discussion about the level of achievement of the goals of Chapter 1, the fulfilment of requirements in Chapter 4 and the originality regarding the related work.

8.1 Discussion

In the introductory chapter we indicated three goals: create an MDE architecture for migrating legacy GUIs (*G1*), separate and make explicit the information of GUI definitions (*G2*) and event handlers (*G3*). The solution architecture we devised was presented in Chapter 4. In the next subsection we will discuss about how these goals have been achieved and we will put into relation with other works.

8.1.1 Goal 1: Architecture for migrating legacy GUIs

We have profited from the benefits provided by MDE to meet the *R1*, *R2*, *R3*, and *R4* requirements. Models have been useful to explicitly represent the information that is discovered in the reverse engineering stage (requirement *R1*), which are described by the many metamodels we have created, like the Structure metamodel, the Layout metamodel and so forth. All the models that form our CUI explicitly represent information of different aspects of the GUI. MDE brings an additional benefit, which is that models can be serialised using the XMI standard [115], so this information is available for different projects or for third-party entities that want to profit from it. For instance, our CUI models can be transformed into an existing UIDL description and then be used by code generators, documentation or GUI testing tools which are available for that UIDL.

An MDE architecture (i.e., a model transformation chain) significantly promotes modularity (*R2* requirement), what implies simplicity, reusability and extensibility, since the input and output models of each stage can be used as extension points. For instance, we can reuse the part of the architecture that obtains the Structure model in a solution to distribute a legacy GUI among several devices. In this setting, the Structure model would be the extension point that would be used to integrate our approach into an existing solution, wich would have a limited impact on the latter.

Automation (requirement *R3*) is achieved thanks to the chains of model transformations. In our solution the target artefacts are automatically generated. However, there are three cases in which developers need to modify either the input or the output. The first case happens when the execution stops due to the preconditions of the source artefacts are not fulfilled. For example, if there are widgets that are highly overlapped (slight overlappings are supported), then the program stops and a developer has to modify the input and execute the transformation again. In the second case the target artefact is generated, but it lacks of elements that have been omitted because the tool does not know how to deal with some elements. This is the case when a unusual widget is used or a fragment of code is not recognised. These flaws can be repaired by completing the metamodels and creating exhaustive pattern catalogs which cover not only the common cases but all the possibilities. The third case also consists of a successful execution, but the result is not what user expected, for example, the layout generated is not the best option according to the developer, so he or she has to tune the algorithm parameters or directly modify some models (the Layout model) and launch again the last part of the transformation chain. We conclude that requirement *R3* is mostly fulfilled, but not completely.

The architecture was designed to provide independence of the source technology by means of the Normalised and RADBehaviour models, and target independence by means of the CUI model (requirement *R4*). In fact, we have proved the source independence by reusing the same layout inference approach for two different sources, which are Oracle Forms windows and wireframes created with WireframeSketcher. Note that both of them have a rather different nature, since Oracle Forms windows belong to a legacy application and are encoded in XML format, while WireframeSketcher wireframes are created during the analysis stage of development and are actually Ecore models. Likewise, the target independence has been proved by generating code for Java Swing which is a Java desktop toolkit and ZK which is a web toolkit. Nonetheless, the information of the CUI model is not enough to migrate event handlers to new platforms, since complete information of the Event Handler AST model is needed to generate the Target Technology model. It is worth remarking that the Event concerns model or the RADBehaviour model are guides that can lead the generation of the target code, but they lack of information about the code that is tied to the source technology and therefore has not been represented in neither the RADBehaviour model nor the Event concerns model.

Given that we claim that our solution is source and target independent, it is worth commenting on the amount of effort needed to change the source or target technology in our architecture. Replacing the source technology entails programming the transformation to the Normalised model and transformation to the RADBehaviour (where the idioms are hard-coded). The transformation of the source models into the Normalised models is usually relatively straightforward. On the other hand, we believe that there is a trade-off between automation and accuracy of the

RADBehaviour model. That is, if we transform the source code models into KDM, then we can automate the pattern matching process for any language so reducing the effort of developers, but as far as patterns are somehow dependent on the source language, language-specific patterns could not be matched, thus obtaining more general primitives in the RADBehaviour model (there is a loss of semantics). On the other hand, replacing the target technology implies transforming the CUI model into a Target Technology model. whose complexity depends on the features of the target technology. Although the CUI model contains a lot of explicit and useful information, the transformation may be complicated due to technology quirks, so complexity is inversely proportional to the experience of the developer in the target technology.

There are other works that take advantage of the MDE benefits, like explicit information [90], modularity and automation [21], and independence from source or target technologies [4], and our model-driven architecture takes profit from all these features at the same time. Although there are approaches that gather information about a specific aspect of the GUI (e.g., interactions in [113]), we have not found any proposal that defines a CUI model that identifies the different aspects and integrates them into one model with a modular approach, with each model representing a GUI dimension that is linked to a base model (the Structure model).

Table 8.1 summarises the requirements of goal G1 and the limitations that reduce their fulfilment.

Requirement	**Solution**	**Fulfilment**	**Limitation**
R1: Explicit GUI information	Metamodels	Total	None
R2: Modularity	Model transformation chain	Total	None
R3: Automation	Model transformation chain	High	Fitness function
R4: Source independence	Normalised GUI tree model RADBehaviour model	Total	None
R4: Target independence	CUI model	Total	None

Table 8.1: Fulfilment of the requirements of goal G1

8.1.2 Goal 2: Analysis of GUI definitions for migration

Requirements *R5*, *R6*, *R7*, *R8* and *R9* are related to this second goal. The Region model identifies regions in views and let us achieve the matching of the visual and logical structure of views (requirement *R5*). Explicit containment is perfectly addressed and region identification is correctly solved when the regions are surrounded with borders, but when there are groups of widgets that are spatially separated (without borders), distinct regions are not created. Nevertheless, the layout inference algorithm is not affected by the 'imperfect' region detection as it is able to differentiate regions by itself thanks to the closeness level mechanism.

Several approaches in the literature have dealt with the segmentation of web pages in order to migrate them to a mobile web interface [77] [78]. Many of these works are based on the VIPS [76] page segmentation algorithm, which is an algorithm that performs a partition of the web page based on the type of HTML tags. When the web page has been segmentated, these works propose different visualisation alternatives in the mobile device: display relevant segments of the web page, show an snapshot of the interface so the user can select a part that is then zoomed-in, among others. We have not found any approach that explicitly presents a solution for matching the visual structure from a legacy GUI. Our approach is the equivalent to VIPS but with coordenate-based GUIs, and can be therefore used in the migration of legacy GUIs to mobile interfaces. Another use is the distribution of the GUI in different devices.

Regarding the high-level layout (requirement *R6*), the quality of the resulting layout is affected by the parameters of the algorithm and the implementation of the fitness function, though with the default configuration, the result is acceptable in most cases. As we clearly stated when explaining the Widget distance clustering problem in Chapter 6, the automated assignment of the closeness level is tricky and sometimes closeness levels may confuse the pattern matching algorithm. The effects of the fitness function are mainly visible when testing the flexibility of the layout (e.g., resizing a generated view and verifying). In some cases the best fitness value entails a final GUI that is not resized as a developer could expect (Fitness function implementation problem).

The misalignment tolerance (requirement *R7*) is fulfilled by including a margin when comparing coordinates to create tiles. A more or less similar approach was proposed in [3] to give flexibility to coordinate constraints. When there are widgets that are too close, the margin is automatically cut down to prevent it from spoiling the inference. Given that the same margins are applied to all the distances, sometimes it may happen that a high misalignment tolerance changes the position or the alignment of other elements. As a result, slightly misaligned widgets can be corrected with small margins, but large margins rarely work well.

The layout inference algorithm outputs all the possible layout compositions it has been able to create (requirement *R8*). If the algorithm selects a layout composition that is not desired by the developer, he or she can inspect the model and mark the layout that he or she prefers. Then, the last part of the transformation chain can be executed to generate the code with that choice.

The types of layouts to match against views is totally configurable (requirement *R9*). Furthermore, new layout types can be incorporated. The design of the tool allows the extension with new types easily, To add a new layout type, it is neccessary to implement a class that matches the layout pattern on the layout graph, implement a class that creates the layout type instances in the Layout model, and maybe, extend the fitness function.

As far as we know, there are no works that detect a composite layout based on a configurable set of layout types. The existing approaches such as [4] and [5] define algorithms that are tightly tied to specific types of layouts. There is neither an approach that outputs several ranked alternative solutions.

Table 8.2 summarises the requirements and limitations for goal G2.

Requirement	**Solution**	**Fulfilment**	**Limitation**
R5: Logical/visual structure matching	Structure model	High	Non-surrounded parts
R6: High-level layout representation	Layout model	Medium/High	Parameters and fitness function
R7: Misalignment tolerance	Layout inference algorithm	High	Small misalignments
R8: Alternative solutions	Layout inference algorithm	Total	None
R9: Configurable layout set	Layout inference algorithm	Total	None

Table 8.2: Fulfilment of the requirements of goal G2

8.1.3 Goal 3: Analysis of the code of event handlers for migration

The solution that has been implemented for event handler analysis fulfils requirements *R10*, *R11* and *R12*. The RADBehaviour model is useful because it explicitly represents simple information about sentences or groups of sentences in the code of event handlers. The key of this model is the ability of the devised primitives to represent the relations between the elements in the code of RAD applications (widgets, database fields, code functions, local variables, etc.) in a simple fashion. A reverse engineering task that for example needs to detect where the values of widgets are set, will find the RADBehaviour model much easier to analyse than the Event Handler AST model. A problem of the pattern matching approach is that the source code can contain statements that offer some behaviour that cannot be locally translated to primitives but if affects the entire application. For example, in Oracle Forms there is a function to know the status of a form, which is automatically managed by the environment. Emulating that status entails a large amount of code in all the views to handle it. Moreover, in some environments it is possible to declaratively specify some functionality. For instance, in Oracle Forms a developer can declare that views contain a 'go forward' and 'go back' buttons to see the next and previous bulks of results of a data table respectively. Technology-dependant functions may be tricky to abstract and sometimes they can only be transformed into function calls, which is not very useful as it does not add any semantics about that code. Fortunately, our approach is in most cases successful since event handler code frequently repeats the identified patterns which have a definite behaviour that generally is not tied to the source technology.

In relation to code categorisation and representation of the interactions and navigation (requirements *R11* and *R12*), the algorithms basically analyse the type of primitives in the RADBehaviour model as well as their inputs and outputs, which makes our solution heavily dependant the primitives. Hence, the separation of architectural concerns and the identification of interactions are performed alright as far as the source code is correctly represented in the form of primitives. On the whole, most of the code analysed was successfully separated in concerns, and the widget and view dependencies were found.

In the state of the art we listed many approaches that analyse event handler code. Most of them obtain some kind of state machine that is used for software testing or program comprehension. Our contribution in this sense is that we integrate a state machine that represents the navigation flow of views with a state machine for each view that expresses dependencies among their widgets. There are also approaches that propose semi-automated solutions for separating the code in layers [11] whereas our proposal is fully automated. On the other hand, as far as we know, there is no intermediate representation particularly designed for event handlers. That representation has demonstrated being helpful as it greatly facilitated other reverse engineering tasks (concern separation, interaction identification) related to the analysis of event handlers of RAD applications. Table 8.3 sums up the requirements and limitations of the G3 goal.

Requirement	**Solution**	**Fulfilment**	**Limitation**
R10: Code abstraction	RADBehaviour model	Medium-High	Technology-dependant statements
R11: Code categorisation	Event concerns model	High	Depends on the RADBehaviour model
R12: Explicit interaction and navigation	Interaction model	High	Depends on the RADBehaviour model

Table 8.3: Fulfilment of the requirements of goal G3

8.2 Contributions

8.2.1 First contribution: MDE-based migration architecture

In the course of our thesis we have made several contributions. Firstly, we have designed and implemented an MDE architecture to perform migrations of RAD GUIs. The design has been focused on separating the different aspects of a GUI and reducing the complexity of the problems by splitting them in smaller subproblems that are chained, which promotes modularity. As a part of that architecture, we have defined two metamodels that make the input of the process independent of the source legacy technology (the Normalised metamodel for the GUI definition and the RADBehaviour metamodel for the code of event handlers), and a set of related metamodels (Structure, Layout, EventConcerns, Interactions) that represent each one of the GUI aspects and provides independence of the target technology.
The main publication we have produced regarding this topic is the following (the publications mentioned in the other two contributions also deal with this topic).

- Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, *Architecture for Reverse Engineering Graphical User Interfaces of Legacy Systems*. In proceedings of the Software Support for User Interface Description Language (UIDL'11), in conjunction with the 13th IFIP TC13 Conference on Human-Computer Interaction (Interact 2011), Lisbon (Portugal), 2011.

8.2.2 Second contribution: Layout inference approach

Secondly, we have proposed a set of data structures (i.e., the Tile and Layout metamodels) and algorithms (i.e., model transformations) to reverse engineering a GUI definition in which the layout is expressed in coordinates to a layout described by a composition of a set of layout managers. The solution includes the identification of the visual parts of views (supported by the Region model) with the aim of generating a representation that matches the visual perception of the user, which is materialised in the Structure model. Actually we have proposed two versions of the approach: a greedy version which uses heuristics to detect the layout, and an exploratory version that uses a backtracking algorithm to identify possible solutions. Tools for supporting each version of the layout inference were developed.[1] The first tool migrates Oracle Forms GUIs

[1]The tools can be downloaded from http://modelum.es/trac/guizmo

to Java Swing, and the second tool transforms wireframes created with WireframeSketcher to web interfaces in ZK.
The most important publications we have produced regarding this topic are:

- Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, *Model-driven reverse engineering of legacy graphical user interfaces*. Journal of Automated Software Engineering, April 2014, Volume 21, Issue 2, pages 147-186.
  Impact factor: 1.400 (28/105, 2nd quartile in JCR/Computer Science/Software Engineering)

- Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, *Model-driven reverse engineering of legacy graphical user interfaces*. In the proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10), Antwerp (Belgium), 2010.
  Acceptance rate: 34%

- Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, Jean Vanderdonckt *A layout inference algorithm for graphical user interfaces*. Information and Software Technology. Under review.
  Impact factor: 1.522 (23/105, 1st quartile in JCR/Computer Science/Software Engineering)

- Óscar Sánchez Ramón, Jean Vanderdonckt, Jesús García Molina, *Re-Engineering Graphical User Interfaces from their Resource Files with UsiResourcer*. In proceedigns of the 7th International Conference on Research Challenges in Information Science (RCIS'13), Paris (France), 2013.
  Acceptance rate: 26%

8.2.3 Third contribution: Event handler analysis approach

Thirdly, we have described a set of data structures and algorithms to reverse engineer the code of event handlers in order to separate the many concerns that are involved in this kind of code. We have devised the RADBehaviour metamodel that takes advantage of the features of RAD environments and represents fragments of code by primitives that express the semantics behind the code. This representation is the basis of other static code analysis, which we have applied

to develop two tasks. The first one is to separate the code in three basic layers (business logic, controller and GUI) that are common in nowadays frameworks. The second one is to identify the interactions that exist among widgets (e.g., enable a text field when a checkbox is selected) and also the changes in the navigation flow (e.g., when a user press certain button then another view is displayed). We have also developed a tool that puts the approach into practice. It takes event handlers written in PL/SQL and generates code for a web application in which the GUI and control are targeted to a web client (javascript code) and the business logic is executed in the server side (Java code).
The publications we have produced regarding this topic are:

- Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, *Reverse Engineering of Event Handlers of RAD-Based Applications*. In proceedings of the 18th Working Conference on Reverse Engineering (WCRE'11), Limerick (Ireland), 2011.
  Acceptance rate: 26%

8.3 Future work

We have arranged the future work into several categories according to the part of the architecture that it is affected.

8.3.1 CUI metamodel

There are several metamodels that were conceived to take part in the CUI metamodel, but have not been implemented at present. Particularly the Style metamodel and the Validation metamodel. The Style metamodel is intended to define all the information about the appearance of the widgets: colours, fonts, sizes, etc. Styles (groups of graphical attributes) could be structured hierarchically, hence some styles would inherit from others (similar to CSS), and every widget would be assigned one or more styles. When analysing the widgets of a view, one style would be created for the first widget, and every widget that had different values for the graphical attributes would lead to a division of the already created styles. When all the widgets would be analysed, the process would output a style tree and the style nodes would be assigned to widgets. A challenge of this approach would be to maximise the styles and to avoid having a large number of styles with only one or two attributes.

The Validation metamodel would let developers specify all the information about form validation, such as validation rules, widgets associated with those rules and the error messages for each failed rule. Validation in legacy systems may be specified in the GUI definition, or defined in event handlers. In the latter case, matching portions of code that perform validation and assigning them a predefined category (e.g., checking that an e-mail has the proper format) may be a challenging task.

The Interaction and Structure metamodels we have defined in the previous chapters can also be extended to support new aspects. The Interaction metamodel could include information about error handling, which can be seen as a particular case of navigation flow in which the transition to the error view is triggered by an internal event (raised by certain system conditions). The Structure metamodel can be extended to represent complex widgets that are compositions of single widgets. The reason that motivates this design decision is that some legacy systems usually lack of certain widgets that are common at present in GUI toolkits, for instance, calendars. Then, developers used to make up complex widgets based on combinations of simple widgets, e.g., a calendar can be implemented as a grid of buttons (a button for each day of the month). The model transformation that generates the Structure model can be enriched to detect those complex widget, so the target technology transformation would avoid migrating each widget in an isolated manner, which is better practice as there are available mappings for them in modern toolkits.

IFML, which is gaining in acceptance, can be used in place of our CUI models. IFML lets us define view components, view containers, events, interaction between components and between user and components, and the referenced data at the different tiers of the architecture. It also promotes separation of aspects, so it can particularly replace the Structure and Interaction models, and to some extent, the Validation and EventConcerns models. The integration could be done by setting IFML as the central model, and then make the Layout and Style models reference the IFML model. If this change would be accomplished in the CUI model, most of the current reverse engineered information such as the Region and RADBehaviour models would be still valid.

8.3.2 Region identification

Sometimes there are groups of widgets that are spatially distant from other groups and are clearly perceived as regions. However, these regions will not be detected if they are not surrounded by a border, according to Algorithm 1. Then, an improvement of the region detection algorithm would be to consider such regions. It can be accomplished with the information of the tiles. Since the distance between tiles is expressed with closeness levels, we could group all the tiles with the same closeness level C_1 and look for the groups of tiles that are separated from the rest by means of a closeness level C_2 being $C_2 > C_1$.

Region identification can have many uses. In the context of a distributed GUI, each region could be launched in a different device. Another use would be the identification of GUI clones, i.e., portions of views that are duplicated, so developers could apply refactoring to improve the maintenance of the application.

8.3.3 High-level layout inference

There are several features that can be upgraded to get better results. Firstly, the fitness function can include metrics about human perception so the layouts are assessed more accurately. For instance, it would be interesting to know if a group of widgets is perceived as a horizontal block, vertical block or square block. This metrics should let us choose a better layout composition that is properly adapted to different screen sizes.

The widget distance clustering problem can be sometimes confusing for the layout inference algorithm. A possibility would be to handle some 'far' distances as 'short' when the involved widgets take part in a form. This is not trivial because we have to identify which widgets belong to a form previously.

Insomuch as the layout inference algorithm is basically a graph pattern matching problem, we could use pattern matching tools such as GrGen [116] or Viatra [117]. These tools surpass our pattern matching engine in two aspects: i) pattern matching is more efficient than our approach which has not been optimised, and ii) patterns can be declaratively specified, which is easier than hard-code them in Java classes. At the beginning we attempted to use GrGen, but some layout patterns like the GridLayout pattern were difficult to define with this tool then we quitted. Now we could deep into this area to find a way to represent layout patterns so we can take advantage of these pattern matching tools.

The algorithm has some parameters (maximum cluster deviation, comparison margin, etc.) that must tunned in many cases. It would be desirable an automated optimisation of those parameters. Since we do not know how good or bad is a configuration of the parameters a statistical approach could be developed to automatically tune the parameters based on the distances among the tiles.

RAD applications does not only allow to access to database information from the code of event handlers, but they commonly offer the possibility of linking widgets to database fields by setting some properties of widgets. Generating the code to decouple these widgets from database would be required if we intended to address a full-fledged migration of the GUI, although from the point of view of research it does not pose a real challenge.

Since the outbreak of different devices (laptops, tablets, etc.) having access to the Internet, responsive web design has strongly gained in followers. Generating responsive designs is a potential use of our layout inference solution. In order to generate responsive designs, common style rules about proportions and heuristics should be applied to generate the CSS 3 rules for different screen sizes. Another option would be to generate code based on an existing frontend framework that offers support for responsive designs, like Bootstrap [118].

8.3.4 Event handler code abstraction

In the current solution, code patterns of event handlers are hard-coded in the transformation. A better approach would be to have several repositories of patterns, and use a DSL for defining new patterns that are added to any of them. There would be different types of repositories: programming-language idioms, community idioms and business-dependant idioms, so the first and second ones could be reused among applications, even they could be shared with third parties.

Case studies for different RAD applications are needed to strongly prove that repetitive idioms can be found in other RAD environments like Delphi 5 (though the current idioms are based on the analysis of aplications in different RAD environments). These case studies would also serve to demonstrate that these idioms can be captured with the RADBehaviour primitives we have defined, and that these primitives are enough to represent the behaviour of any RAD application. Furthermore, the evaluation of the case studies should be more complete and systematic like the ones we accomplished for the layout inference approaches.

The algorithm that generates the fragments is not optimum regarding the number of fragments. In its current state, it may generate fragments of the same type that have not variables in common, and which can be safely placed in the same fragment.

On the other hand, we believe that we can take advantage of the RADBehaviour model in many ways beyond the separation of concerns, for instance, it can be used to accomplish code refactoring tasks such as death code cleaning. Considering it as a concise representation of the semantics that lie on the code, it could be used in clone detection. As a first approach, we could look for fragments in the model that are repeated throughout the event handlers and then use a deeper analysis to check if they are actually clones.

8.3.5 Identification of widget dependencies

The algorithm that identifies widget dependencies and generates the Interaction model should be assessed in a real case study to prove that it can capture all the interactions that take place in the GUI based on the RADBehaviour model. The identification of widget dependencies can be particularly useful to turn legacy applications into Rich Internet Applications (RIA). Two ways to proceed are posible: use the dependency model to identify widgets that can be asynchronously updated via Ajax and then replicate the look and feel and navigation flow of the original application, or use the dependency model to restructure the views according to development patterns, for example to convert it into a single-page application (this would require a restructuring of many of the CUI models such as the Structure and Layout models).

8.4 Publications related to the thesis

8.4.1 Journals with impact factor

- Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, *Model-driven reverse engineering of legacy graphical user interfaces*. Journal of Automated Software Engineering, April 2014, Volume 21, Issue 2, pages 147-186.
  Impact factor: 1.400 (28/105, 2nd quartile in JCR/Computer Science/Software Engineering)

- Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, Jean Vanderdonckt *A layout inference algorithm for graphical user interfaces.* Information and Software Technology. Under review.
  Impact factor: 1.522 (23/105, 1st quartile in JCR/Computer Science/Software Engineering)

8.4.2 Renowned international conferences

- Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, *Model-driven reverse engineering of legacy graphical user interfaces.* In the proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10), Antwerp (Belgium), 2010.
  Acceptance rate: 34%

- Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, *Reverse Engineering of Event Handlers of RAD-Based Applications.* In proceedings of the 18th Working Conference on Reverse Engineering (WCRE'11), Limerick (Ireland), 2011.
  Acceptance rate: 26%

- Óscar Sánchez Ramón, Jean Vanderdonckt, Jesús García Molina, *Re-Engineering Graphical User Interfaces from their Resource Files with UsiResourcer.* In proceedigns of the 7th International Conference on Research Challenges in Information Science (RCIS'13), Paris (France), 2013.
  Acceptance rate: 26%

8.4.3 Other journals

- Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, Jean Vanderdonckt *Generación de Interfaces de Usuario a partir de Wireframes.* Novática, Revista de la Asociación de Técnicos en Informática (Spain), Nov-Dec 2013, N°226, pages 24-29.

8.4.4 Other international and national conferences and workshops

- Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, *Ingeniería inversa de eventos GUI en aplicaciones RAD mediante MDD.* In proceedings of the VII Taller de Desarrollo de Software Dirigido por Modelos (DSDM'10), Valencia (Spain), 2010.

- Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, *Architecture for Reverse Engineering Graphical User Interfaces of Legacy Systems.* In proceedings of the Software Support for User Interface Description Language (UIDL'11), in conjunction with the 13th IFIP TC13 Conference on Human-Computer Interaction (Interact 2011), Lisbon (Portugal), 2011.

- Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, Jean Vanderdonckt, *GUI Layout Inference based on Wireframes*. In proceedings of the Interacción'13, Madrid (Spain), 2013.

- Óscar Sánchez Ramón, Francisco Javier Bermúdez Ruiz, Jesús García Molina, *Experiencias de Modernización de Software con DSDM*. In proceedings of the XVIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'13), Madrid (Spain), 2013.

8.5 Other publications in the MDE area

Along these years we have mainly worked in the migration of GUIs, but we have applied MDE in other software development areas and the results obtained have been disseminated in international journals and conferences. Particularly, we have publicated two works (WOSIS'09 and JUCS) about security requirement engineering, in the context of a collaboration with the GIS group of the University of Murcia, and two works (PMDE'13 and SPE) related to business process modelling and enactment, which stem from the development of a migration tool in which our GUI migration solutions were integrated. Next we sum up all these publications.

8.5.1 Journals with impact factor

- Óscar Sánchez Ramón, Fernando Molina Molina, Jesús J. García Molina, Ambrosio Toval Álvarez *A Generative Architecture for Model-Driven Security*. Journal of Universal Computer Science, 2009, volume 15, issue 15, pages 2957-2980.
  Impact factor: 0.762 (54/100, 3rd quartile in JCR/Computer Science/Theory and Methods)

- Francisco Javier Bermúdez Ruiz, Óscar Sánchez Ramón, Jesús García Molina, *A model-driven tool to support the definition and enactment of migration processes*. Software Practice and Experience. Under review.
  Impact factor: 1.008 (51/105, 2nd quartile in JCR/Computer Science/Software Engineering)

8.5.2 International conferences and workshops

- Óscar Sánchez, Fernando Molina, Jesús García Molina, Ambrosio Toval, *A model driven approach for generating code from security requirements*. In proceedings of the 7th International Workshop on Security in Information Systems (WOSIS'09), Milan, 2009.

- Francisco Javier Bermúdez Ruiz, Óscar Sánchez Ramón, Jesús García Molina, *Definition of processes for MDE-based migrations*. In proceedings of the 3rd Workshop on Process-based approaches for Model-Driven Engineering (PMDE'13), Montpellier (France), 2013.

8.6 Projects that are related to this thesis

- **"MOMO: Un Entorno de Modernización de Software Dirigida por Modelos en Escenarios de Migración de Plataformas (Ref. 08797/PI/08)"**. Granted by the Fundación Séneca (Regional plan of Science and Technology 2007-2010). From 2009-01-01 until 2010-12-31. In this project we designed the first approach for inferring the layout of the Oracle Forms windows.

- **"Impulso de la Investigación en Tecnologías del Desarrollo de Software (Un Entorno para el Desarrollo y Modernización Basado en Modelos: Forms-ADF) (Ref. CARM 129/2009)"**. Granted by the Consejería de Universidades, Empresas e Investigación. From 2009-06-04 until 2010-12-31. The goal of this project was the definition of a software environment for the migration of Oracle Forms applications to ADF. We used the results obtained in the previous project in order to implement the layout inference engine.

- **"GUIZMO: Un framework para la modernización basada en modelos de interfaces de usuario"**. Granted by the Fundación Séneca (Research Projects Funds). From 2011-01-01 until 2014-12-31. In this project we tackled the development of a model-driven framework for analysing the code of event handlers in order to separate the concerns that are tangled. Moreover, during this project we created a tooling to assist the automatic generation of web interfaces from wireframes.

8.7 Contracts supporting this thesis

- **"Automatización del Desarrollo de Software con Arquitecturas Generativas (Auto-GSA)"**. Granted by the Technological Center of the TICs (CENTIC). From 2009-04-29 until 2010-01-15.

- **"Impulso de la Investigación en Tecnologías del Desarrollo de Software (Un Entorno para el Desarrollo y Modernización Basado en Modelos: Forms-ADF) (Ref. CARM 129/2009)"**. Granted by the Consejería de Universidades, Empresas e Investigación. From 2009-07-13 until 2010-12-31.

- **"GUIZMO: Un Framework para Modernización Basada en Modelos de Interfaces de Usuario"**. Granted by the Fundación Séneca of the Region of Murcia. From 2011-01-01 until 2011-12-31.

- **"Reverse Engineering of Graphical User Interfaces (in the context of the UsiXML European Project)"**. Granted by the Université Catholique de Louvain. From 2012-01-01 until 2012-09-30.

- **"Construcción de una Plataforma para la Migración de Interfaces RAD"**. Granted by the Consejería de Universidades, Empresas e Investigación. From 2012-11-06 until 2013-12-31.

8.8 Research stays

- **Research Stay in the Université Catholique de Louvain (Belgium)**, during 9 months, in the Human-Computer Interaction Laboratory (LiLab). We were working in a Java tool that reversed engineering web pages (HTML 4/5 and CSS 2/3) and generated our CUI model, which was in turn transformed into a UsiXML CUI from which UsiXML definitions were generated. The work was the seed of the advanced layout inference approach. During the stay we reviewed and contributed to the CUI model of UsiXML, and we also collaborated in a work about reverse engineering of GUIs that resulted in [39].

8.9 Transfer of technology

- **"Herramienta orientada a la migración basada en modelos"**. Granted by the Ministerio de Industria, Turismo y Comercio. CDTI project granted to the Sinergia IT (Deusto Group) software company. From 2010-01-01 until 2011-12-31. This project was aimed at the creation of a tooling to assist the automatic migration of Oracle Forms applications to a Java platform. Our research group collaborated with the Sinergia company to accomplish research tasks in the context of this project.

- **"Use of the prototype for migrating Oracle Forms applications"**. Based on the results of this thesis, particularly the migration tool from Oracle Forms to Java Swing, the Open Canarias company developed a prototype of a migration tool from Oracle Forms to JSF 2.0 during the last few months of 2013. This company reused as-is the reverse engineering process and toolchain that obtains the CUI model, and extended it to derive KDM UI models and then generate JSF code from them. The prototype of the layout inference approach we had created was the cornerstone of a series of case studies of Oracle Forms application migrations, which resulted in a software requirements specification for a full-fledged industrial solution. By the end of 2014 a pilot project to apply this solution in the context of a major public institution will be carried out, which will let the company validate and assess the viability of the solution regarding a concrete problem.

References

[1] John Gerdes, Jr. User interface migration of microsoft windows applications. *Journal of Software Maintenance and Evolution*, 21(3):171–187, 2009.

[2] Stefan Staiger. Reverse engineering of graphical user interfaces using static analyses. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 189–198, 2007.

[3] Christof Lutteroth. Automated reverse engineering of hard-coded gui layouts. In *Ninth Australasian User Interface Conference (AUIC 2008)*, volume 76, pages 65–73, 2008.

[4] José Matías Rivero, Gustavo Rossi, Julián Grigera, Juan Burella, Esteban Robles Luna, and Silvia Gordillo. From mockups to user interface models: an extensible model driven approach. In *Proceedings of the 10th international conference on Current trends in web engineering*, ICWE'10, pages 13–24, 2010.

[5] Nishant Sinha and Rezwana Karim. Compiling mockups to flexible uis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 312–322, 2013.

[6] Svetoslav R. Ganov, Chip Killmar, Sarfraz Khurshid, and Dewayne E. Perry. Test generation for graphical user interfaces based on symbolic execution. In *Proceedings of the 3rd International Workshop on Automation of Software Test*, AST '08, pages 33–40, 2008.

[7] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, November 2003.

[8] Inês Coimbra Morgado, Ana CR Paiva, and João Pascoal Faria. Dynamic reverse engineering of graphical user interfaces. *International Journal On Advances in Software*, 5(3 and 4):224–236, 2012.

[9] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN 0-201-10088-6.

[11] Reiko Heckel, Rui Correia, Carlos M. P. Matos, Mohammad El-Ramly, Georgios Koutsoukos, and Luis Filipe Andrade. Architectural transformations: From legacy to three-tier and services. In *Software Evolution*, pages 139–170. 2008.

[12] S. R. Tilley and D. B. Smith. Perspectives on legacy system reengineering. Technical report, Software Engineering Institute, Carnegie Mellon University, 1995.

[13] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[14] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 1st edition, 2012. ISBN 1608458822, 9781608458820.

[15] Object Management Group (OMG). *MDA Guide Version 1.0.1*. http://www.omg.org /mda, 2003.

[16] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008. ISBN 978-0-470-03666-2.

[17] Object Management Group (OMG). Architecture-Driven Modernization. http://adm.omg.org/, .

[18] William M. Ulrich and Philip Newcomb. *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. Morgan Kaufmann Publishers Inc., 2010. ISBN 0123749131, 9780123749130.

[19] Gordon Blair, Nelly Bencomo, and Robert B. France. Models run.time. *Computer*, 42 (10):22–27, 2009.

[20] Thijs Reus et al. Harvesting software systems for mda-based reengineering. In *Proceedings of the Second European conference on Model Driven Architecture: foundations and Applications*, ECMDA-FA'06, pages 213–225, 2006.

[21] F. Fleurey et al. Model-driven engineering for software migration in a large industrial context. In *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007*, pages 482–497, 2007.

[22] Javier Luis Cánovas Izquierdo and Jesús García Molina. A domain specific language for extracting models in software modernization. In *ECMDA-FA '09: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, pages 82–97, 2009.

[23] Hugo Brunelière et al. MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the Automated Software Engineering*, pages 173–174, 2010.

[24] Ricardo Pérez-Castillo, Ignacio García Rodríguez de Guzmán, Mario Piattini, and Christof Ebert. Reengineering technologies. *IEEE Software*, 28(6):13–17, 2011.

[25] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.

[26] Bran Selic. What will it take? a view on adoption of model-based methods in practice. *Software and System Modeling*, 11(4):513–526, October 2012.

[27] James Martin. *Rapid application development*. Macmillan Publishing Co., Inc., 1991.

[28] John V. Harrison and Wie Ming Lim. Automated reverse engineering of legacy 4gl information system applications using the itoc workbench. In *Proceedings of the 10th Conference on Advanced Information Systems Engineering (CAiSE'98)*, pages 8–12, 1998.

[29] Luis Filipe Andrade, João Gouveia, Miguel Antunes, Mohammad El-Ramly, and Georgios Koutsoukos. Forms2net - migrating oracle forms to microsoft .net. In *GTTSE*, pages 261–277, 2006.

[30] José Campos, João Alexandre Saraiva, Carlos Silva, and J.C. Silva. *GUIsurfer: A Reverse Engineering Framework for User Interface Software*, chapter 2, pages 31–54. InTech, 2012.

[31] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):3:1–3:30, March 2012.

[32] Sinergia tecnológica (oesia group) and modelum (university of murcia). herramienta orientada a la migración basada en modelos. CDTI project, Ministry of Industry, Turism and Comerce. 2010-2011.

[33] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 147–150, 2010.

[34] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering*, 21(2):147–186, 2014.

[35] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Reverse engineering of event handlers of rad-based applications. In *Working Conference on Reverse Engineering (WCRE)*, pages 293–302, 2011.

[36] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Ingeniería inversa de eventos gui en aplicaciones rad mediante mdd. In *VII Taller de Desarrollo de Software Dirigido por Modelos (DSDM10)*, 2010.

[37] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Architecture for reverse engineering graphical user interfaces of legacy systems. In *Software Support for User Interface Description Language (UIDL11)*, 2011.

[38] Quentin Limbourg and Jean Vanderdonckt. Usixml: A user interface description language supporting multiple levels of independence. In *ICWE Workshops*, pages 325–338, 2004.

[39] Óscar Sánchez Ramón, Jean Vanderdonckt, and Jesús García Molina. Re-engineering graphical user interfaces from their resource files with usiresourcer. In *Seventh IEEE International Conference on Research Challenges in Information Science (RCIS)*, pages 1–12, 2013.

[40] Information and software technology. http://www.journals.elsevier.com/information-and-software-technology/.

[41] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321118847.

[42] Rick Kazman, Steven G. Woods, and S. Jeromy Carrière. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, pages 154–163, 1998.

[43] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and challenges in software reverse engineering. *Communications of the ACM*, 54(4):142–151, April 2011.

[44] ZK framework. http://www.zkoss.org/.

[45] W3C. Cascading Style Sheets (CSS) Level 3. http://www.w3.org/TR/CSS/.

[46] Java Abstract Window Toolkit (AWT). http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html.

[47] Jesús Sánchez Cuadrado, Javier Luis Cánovas Izquierdo, and Jesús García Molina. Applying model-driven engineering in small software enterprises. *Science of Computer Programming*, 89, Part B(0):176–198, 2014. Special issue on Success Stories in Model Driven Engineering.

[48] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, pages 260–269, 2003.

[49] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Reverse engineering finite state machines from rich internet applications. *2013 20th Working Conference on Reverse Engineering (WCRE)*, 0:69–73, 2008.

[50] *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-48567-2.

[51] Tony Clark, Andy Evans, Paul Sammut, and James Willans. *Applied Metamodelling - A Foundation for Language Driven Development*. Ceteva, second edition, 2004.

[52] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 1st edition, 2012. ISBN 1608458822, 9781608458820.

[53] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321331885.

[54] Object Management Group (OMG). The Meta-Object Facility (MOF). http://www.omg.org/mof/, .

[55] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, 2008. ISBN 0321553454, 9780321553454.

[56] Balsamiq mockups. http://balsamiq.com/products/mockups/.

[57] Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). http://www.omg.org/spec/QVT/.

[58] AtlanMod Transformation Language (ATL). http://www.eclipse.org/atl/.

[59] Epsilon. http://www.eclipse.org/epsilon/.

[60] T. gardner and c. griffin and j. koehler and r. hauser. review of omg mof 2.0 query/views/transformations submissions and recommendations towards final standard. http://www.omg.org/docs/ad/03-08-02.pdf. 2003.

[61] Jesús Sánchez Cuadrado, Jesús García Molina, and Marcos Menárguez. RubyTL: A practical, extensible transformation language. In *2nd European Conference on Model-Driven Architecture*, volume 4066 of *LNCS*, pages 158–172. Springer, 2006.

[62] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, July 2006. ISSN 0018-8670.

[63] Object Management Group (OMG). MOF Model to Text Transformation Language (MOFM2T), 1.0. http://www.omg.org/spec/MOFM2T/1.0/, .

[64] Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Eclipsecon Summit Europe 2006*, 2006.

[65] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, 2014.

[66] Javier Luis Cánovas Izquierdo and J García Molina. An architecture-driven modernization tool for calculating metrics. *IEEE Software*, 27(4):37–43, 2010.

[67] William M. Ulrich and Philip Newcomb. *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. Morgan Kaufmann Publishers Inc., 2010. ISBN 0123749131, 9780123749130.

[68] S. Rugaber and K. Stirewalt. Model-driven reverse engineering. *IEEE Software*, 21(4): 45–53, July 2004.

[69] Jean-Marie Favre. Foundations of Model (Driven) (Reverse) Engineering : Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, Dagstuhl Seminar Proceedings, 2005.

[70] Object Management Group (OMG). Architecture-Driven Modernization (ADM) Task Force: Overview, Scenarios & Roadmap. http://www.omg.org/adm/TF-1_Ulrich_ADM-PTF.pdf, .

[71] OMG. *Knowledge Discovery Meta-Model (KDM) v1.0*. http://www.omg.org/spec/KDM/1.0/, 2008.

[72] Netbeans. Java Swing GUI Builder (Matisse). http://www.netbeans.org/ features/-java/swing.html.

[73] Morgan Dixon, Daniel Leventhal, and James Fogarty. Content and hierarchy in pixel-based methods for reverse engineering interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 969–978, 2011.

[74] Jean Vanderdonckt, Laurent Bouillon, and Nathalie Souchon. Flexible reverse engineering of web pages with vaquista. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, pages 241–248, 2001.

[75] Angel Puerta and Jacob Eisenstein. Ximl: a common representation for interaction data. In *IUI '02: Proceedings of the 7th international conference on Intelligent user interfaces*, pages 214–215, 2002.

[76] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. Vips: a vision-based page segmentation algorithm. Technical report, Microsoft Research, 2003.

[77] Yu Chen, Wei-Ying Ma, and HongJiang Zhang. Detecting web page structure for adaptive viewing on small form factor devices. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 225–233, 2003.

[78] Renata Bandelloni, Giulio Mori, and Fabio Paternò. Dynamic generation of web migratory interfaces. In *MobileHCI '05: Proceedings of the 7th international conference on Human computer interaction with mobile devices & services*, pages 83–90, 2005.

[79] P. Tonella and A. Potrich. Reverse engineering of the interaction diagrams from c++ code. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 159–168, 2003.

[80] A. Sutton and J. Maletic. Mappings for accurately reverse engineering uml class models from c++. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 175–184, 2005.

[81] Atif M. Memon. An event-flow model of gui-based applications for testing: Research articles. *Software Testing Verification and Reliability*, 17(3):137–157, 2007.

[82] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Formal methods and testing. chapter Model-based Testing of Object-oriented Reactive Systems with Spec Explorer, pages 39–76. 2008.

[83] E. Stroulia, M. El-Ramly, P. Iglinski, and P. Sorenson. User interface reverse engineering in support of interface migration to the web. *Automated Software Engg.*, 10(3):271–301, July 2003.

[84] The Gimp Toolkit (GTK+). http://www.gtk.org/.

[85] Qt project. http://qt-project.org/.

[86] Object Management Group (OMG). Interaction Flow Modeling Language(IFML). http://www.ifml.org/.

[87] Brambilla Marco and Stefano Butti. Quince años de desarrollo industrial dirigido por modelos de aplicaciones front-end: desde webml hasta webratio e ifml. *Novática*, (228): 36–44, 2014.

[88] WebRatio. Web Modeling Language (WebML). http://www.webml.org/ webml/-page1.do.

[89] Gaelle Calvary, Joelle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, June 2003.

[90] Fabio Paterno', Carmen Santoro, and Lucio Davide Spano. Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. Comput.-Hum. Interact.*, 16(4):19:1–19:30, November 2009.

[91] Lori A MacVittie. *XAML in a Nutshell (In a Nutshell (O'Reilly))*. O'Reilly Media, Inc., 2006. ISBN 0596526733.

[92] Oasis. User Interface Markup Language (UIML). http://docs.oasis-open.org/ uiml/v4.0/cd01/uiml-4.0-cd01.html.

[93] Kouichi Katsurada, Yusaku Nakamura, Hirobumi Yamada, and Tsuneo Nitta. XISL: A Language for Describing Multimodal Interaction Scenarios. In *Proceedings of the 5th International Conference on Multimodal Interfaces*, ICMI '03, pages 281–284, 2003.

[94] Mozilla developer network. xml user interface language (xul). https://developer.mozilla.org /en-US/docs/Mozilla/Tech/XUL.

[95] W3c. web ontology language(owl). http://www.w3.org/standards/techs/owl.

[96] W3C. Concur Task Trees (CTT). http://www.w3.org/2012/02/ctt/.

[97] Object Management Group (OMG). Unified Modeling Language(UML). http://www.omg.org/spec/UML/.

[98] Silvia Berti, Francesco Correani, Fabio Paternò, and Carmen Santoro. The teresa xml language for the description of interactive systems at multiple abstraction. In *Leveles, Proceedings Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages*, pages 103–110, 2004.

[99] Microsoft Developer Network. Windows Presentation Foundation (WPF). http://msdn.microsoft.com/es-es/library/ms754130%28v=vs.110%29.aspx.

[100] Charles Jacobs, Wilmot Li, Evan Schrier, David Bargeron, and David Salesin. Adaptive grid-based document layout. *ACM Trans. Graph.*, 22(3):838–847, 2003.

[101] Weijiang Li and Hiroyuki Kurata. A grid layout algorithm for automatic drawing of biochemical networks. *Bioinformatics*, 21(9):2036–2042, 2005.

[102] Rake. http://www.rake.org/.

[103] Jesús Sánchez Cuadrado and Jesús García Molina. Building domain-specific languages for model-driven development. *IEEE Software*, 24(5):48–55, 2007.

[104] Jesús Sánchez Cuadrado and Jesús García Molina. Modularization of model transformations through a phasing mechanism. *Software and System Modeling*, 8(3):325–345, 2009.

[105] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[106] Wireframesketcher. http://wireframesketcher.com.

[107] Reify. http://www.smartclient.com/product/reify.jsp.

[108] E. Marcotte. *Responsive Web Design*. A Book Appart, 2011. ISBN 978-0-9844425-7-7.

[109] Apache Tomcat. http://tomcat.apache.org/.

[110] Apache struts. http://struts.apache.org/.

[111] JSR 372: JavaServer Faces (JSF 2.3) Specification. https://jcp.org/ en/jsr/detail?id=372.

[112] Jesús Sánchez Cuadrado, Orlando Ávila García, Javier Canovas, and Adolfo Sánchez-Barbudo Herrera. Parametrización de las transformaciones horizontales en el modelo de herradura. In *Jornadas de Ingeniería del Software y Bases de Datos*, 2012.

[113] Sandy Pérez, Oscar Díaz, Santiago Meliá, and Jaime Gómez. Facing interaction-rich rias: The orchestration model. In *Proceedings of the 2008 Eighth International Conference on Web Engineering*, pages 24–37, 2008.

[114] jQuery. http://jquery.com/.

[115] Object Management Group (OMG). XML Metadata Interchange(XMI). http://www.omg.org/spec/XMI/.

[116] Grgen .net. http://www.grgen.net.

[117] Viatra2 (visual automated model transformations) framework. http://eclipse.org /viatra2/.

[118] Bootstrap. http://getbootstrap.com/.

Colophon

NOW THAT EVERYTHING HAS FINISHED, I look back on these years and I recall a quote from a film that I read somewhere:
"Beginnings are scary and endings are usually sad, but it's what's in the middle that counts."