

TRABAJO FIN DE GRADO

Una arquitectura basada en modelos para el análisis y la corrección de la usabilidad de interfaces de usuario

Alumno

José Manuel Cruz Zapata

Directores

Jesús J. García Molina

Diego Sevilla Ruiz



Julio de 2014

Índice general

1. Resumen	5
2. Extended Abstract	7
3. Introducción	11
4. Fundamentos	15
4.1. Usabilidad de una GUI	15
4.2. Introducción a MDE	17
4.3. Tecnologías utilizadas	21
4.3.1. Epsilon	21
4.3.1.1. EOL	22
4.3.1.2. ETL	23
4.3.1.3. EGL	25
4.3.2. EMFText	26
5. Definición del problema	29
6. Visión general de la arquitectura propuesta	33
7. Metamodelos de interfaz: Definición y transformaciones	37
7.1. Metamodelo GUI	37
7.2. Metamodelo de usabilidad	40
7.3. Del modelo GUI al modelo de usabilidad	45
7.3.1. Inicio de la transformación	45
7.3.2. Generación de Propiedades	46
7.3.3. Análisis de la separación entre etiquetas y campos de entrada	49
7.3.4. Conversión de elementos ComboBox y Listas	51
8. El DSL UsabilityRules	53
8.1. Metamodelo UsabilityRules	53
8.1.1. Notación textual	56
8.1.2. Ejemplos de definición de reglas	58

Índice general

9. Análisis y corrección de defectos	65
9.1. Metamodelo de defectos	65
9.1.1. Generación de los informes	66
9.1.1.1. Generación del informe de análisis	67
9.1.1.2. Generación del informe de corrección	68
9.2. Análisis	69
9.2.1. Generación de la transformación de análisis	71
9.3. Corrección	82
10. Validación	87
10.1. Caso de estudio	87
10.2. Resultados	94
10.3. Discusión	103
11. Conclusiones y vías futuras	105
Bibliografía	107
A. Estructura y ejecución del proyecto	109
B. Catálogo de reglas	111

1 Resumen

La usabilidad es un aspecto clave en el diseño de interfaces gráficas de usuario (GUI) y su evaluación es fundamental tanto durante el proceso de desarrollo como una vez puesta en marcha la aplicación. Para ello, es habitual contar con expertos en usabilidad o bien se proporciona una formación al respecto al propio equipo de diseño. Dado que esta tarea exige un esfuerzo considerable, su automatización permitiría un ahorro significativo de recursos e incluso podría abarcar tareas de corrección además de la detección de defectos. Esta automatización sólo puede ser parcial dado que hay defectos cuya detección o corrección requieren la intervención humana por su naturaleza (por ejemplo, los relativos a la importancia o frecuencia de uso de un componente o el significado de un texto).

En este documento presentamos una herramienta de análisis y corrección automática de la usabilidad de interfaces gráficas, basada en una arquitectura diseñada de acuerdo con los principios de la Ingeniería del Software Dirigida por Modelos (MDE), en las que las GUI y reglas de usabilidad se representan a un alto nivel de abstracción. Como prueba de concepto de esta arquitectura se ha implementado y probado su parte central para un pequeño catálogo de defectos. Nuestra solución se caracteriza por la independencia de la tecnología subyacente y la capacidad de soportar catálogos de reglas de usabilidad creados por el propio usuario.

A lo largo del documento se definen los metamodelos utilizados y se describen en detalle las transformaciones modelo a modelo y modelo a texto que automatizan el proceso implementando las tareas de análisis y corrección de defectos de usabilidad. Incluimos un apartado de validación donde se verifica el correcto funcionamiento de la arquitectura definida mediante una sencilla GUI con defectos comunes de usabilidad.

Concluimos con unas palabras que resumen la contribución de este trabajo y que hacen especial hincapié en las vías de futuro del proyecto, que incluyen la finalización de la arquitectura y la optimización de alguno de los procesos expuestos.

2 Extended Abstract

Usability has become a key aspect in the design of graphical user interfaces (GUI) and its evaluation is essential during the development process and once the application is delivered. Among the usability evaluation techniques there are several options, some of them targeting early phases within the design process, others more centered in the user, others which are more formal, etc., but for all of them it is necessary to have the support of usability experts or either providing training in usability to some members of the development team. Since this task requires some significant effort, its automation would suppose notable resource savings and it might even include correction tasks besides defect detection. This type of technique would fall into the category of checklist usability evaluation, that is to say, a set of usability rules which are applied on the GUI elements in order to detect usability defects (e.g. the number of fonts used in a window is greater than a threshold value or a label widget is misaligned with a data input field)..

In this document we present a tool for the automatic analysis and correction of a graphic user interface usability, based on an architecture designed following the principles of Model-Driven Engineering (MDE), representing GUI and usability rules at a high level of abstraction. As a proof of concept of this architecture its central part has been implemented and tested for a small defects catalogue. Our system's main goals are underlying technology's independence and the support of usability rules catalogues defined by the user himself.

With these requirements in mind, we have designed an architecture based on a model transformation chain. We have defined four metamodels to represent the information managed in the analysis and correction process: UsabilityRules, UsabilityDefects, GUI and UsabilityGUI. GUI metamodel is a simplified representation of the graphic user interface concept, and includes the most common GUI components as well as its main properties. It includes the concepts of container and widget and containers are recursively composed of widgets or other containers, so that GUI model can represent the typical hierarchical structure of GUIs. UsabilityGUI metamodel is defined as an intermediate representation aimed to facilitate the application of usability rules on the interface. This is achieved by transforming the component-centered representation of the GUI metamodel into a property-centered representation. UsabilityGUI metamodel has four main characteristics: i) preserves the hierarchical structure of GUIs, ii) includes new concepts such as a **Box** which represents the dimension and position of every component, iii) gives great importance to certain types of GUI components such as **Label**

2. *Extended Abstract*

and Input, and iv) dispenses with the rest of information available in GUI metamodel but not necessary for the application of usability rules. UsabilityDefects metamodel is the one that represents the usability defects found in the GUI and it is the result of both analysis and correction processes. Its purpose is to be displayed in a readable format to the user, so that he can check the usability defects found in the GUI and/or the result of the correction process. It must be a simple and flexible metamodel able to adapt to the (slightly different) needs of both analysis and correction processes. UsabilityRules metamodel represents the usability rules catalogue that will be applied to the GUI. It considers essential properties such as usability rule's name, description, importance or type of GUI component to be applied to, but also includes properties that contain blocks of code that will determine the way the usability rule works. The *raison d'être* of this kind of properties resides in the scheme adopted for the processes of analysis and correction. UsabilityRules metamodel also defines the concept of rank, key in our understanding of the usability rule, that determines the values that will be considered correct and that will be compared with the ones obtained after the application of the usability rule to each GUI component. Just as we have designed the metamodels involved in our system, we define the necessary M2M and M2T transformations to work with those metamodels. GUI-UsabilityGUI transformation is a direct conversion from one representation of an interface to another, but the transformations corresponding to analysis and correction are of a higher complexity. Our architecture is defined in such a way that both of these transformations are in turn generated by a previous M2T transformation which takes as its input the usability rules model. In this way, we design such a scheme that for each usability rule a transformation rule will be generated, being this second rule the one that is applied to the interface resulting in the creation of a usability defect if the corresponding component don't comply with the usability rule. This scheme implies that each usability rule must have been associated with certain blocks of code that will eventually be used to build the corresponding M2M transformation rule. In particular, each usability rule has three code properties: restriction, calculation and correction, to limit the components to which the rule is applied and determine the way that calculation and correction processes will be carried out. This approach implies that the person willing to create his own usability rules catalogue must know the EOL language defined by Epsilon to perform the right queries on the GUI model, but at the same time it provides a great flexibility because the code responsible for the rule's behavior is almost completely in the hands of its creator.

The complete architecture includes the injection of concrete interfaces in the system, allowing the user to provide as an input GUIs defined using a particular technology such as Qt. In order to do that an additional transformation is necessary to transform the original model in a model that conforms our GUI metamodel, and in the same way it becomes necessary the definition of the inverse transformation, one able to transform the GUI model back to its original representation. This mechanism would allow us to begin the process with a GUI built in Qt, detect its usability defects and, once the correction is completed, obtain the original Qt GUI corrected. Nevertheless, this part of the architecture remains as future work since it does not belong to the system we

introduce in this document, which takes a model conforming our GUI metamodel as an input and, in case of correction, returns this very same model as output.

To begin with the implementation of this system we started by studying material related to GUI usability, which lead us to the definition of a reduced catalogue of generic, non-subjective usability rules, meaning that the rules can be applied to any kind of interface and can be easily defined programmatically. We have then analyzed the available technologies within MDE to find the ones that better suited our needs, finally opting for the framework Epsilon with its languages ETL and EGL for M2M and M2T transformations respectively, as well as EMFText for the elaboration of textual DSLs.

Once we had laid the foundations of the project, we began the implementation of the metamodels used in our architecture, as well as the conceived M2M and M2T transformations. The GUI-UsabilityGUI transformation includes advanced aspects such as the label-input association, accomplished through the analysis of the positions of the components, and other important design decisions such as the way to represent components like Combo Boxes and lists. The UsabilityRules DSL contains not only the metamodel defining its abstract syntax, but also the concrete syntax defined with EMFText, with the aim of making it easy for users who are not familiar with model managing to develop their own usability rules catalogue. Using this DSL through the editor generated by EMFText, we defined the complete usability rules catalogue as it was conceived in a previous step of the process, creating every rule conforming the UsabilityRules metamodel and including the codes corresponding to restriction, calculation, and correction for each usability rule. The catalogue contains rules of all kinds, applied to different GUI components and of distinct complexity. We have included usability rules related to the position of a component, its color, the characteristics of its text and some other more specific properties, such as tabfocus or tooltip. The usability catalogue creation was one of the most complex tasks of the project, and among the eighteen usability rules finally defined we can find some complicated calculation codes and different ways of understanding the concept of rank, thus showing the power and flexibility of both the Epsilon EOL language used for the code and the UsabilityRules metamodel defined by us. The catalogue we have created includes the correction code only in some cases; in others only the calculation code has been defined, due to the complexity of the correction process or the subjectivity inherent to some of the rules, which makes it hard to make the correction of the usability defect automatic. This usability rules model is the one we use to generate the M2T transformations that iterate through the catalogue generating, for each usability rule, a transformation rule. Throughout this process the blocks of code corresponding to restriction, calculation, and correction are copied from the usability rule to the transformation file. The concept of *rank* has an important role, since it determines the type of value returned by the calculation function, and its values form the condition that checks whether the component GUI is considered faulty or not. During this process we also generate some auxiliary functions, like the ones that allow us to obtain the RGB representation of a color from its hexadecimal code. Once we have defined the M2T transformations, its execution will automatically generate the M2M transformations in charge of the analysis and correction. Both processes take as an input the UsabilityGUI

2. *Extended Abstract*

representation of the interface and generate a defects model. In the case of correction, the code defined in each usability rule (and copied into the M2M transformation file) will alter the input model modifying some of the properties and thus correcting the usability defects found. In both cases we add some communication features to inform the user, especially during the correction, where we ask for user input to decide the way the process is going to be carried out: either we automatically correct every usability defect found in the GUI or we request user confirmation for the correction of each of them. Once analysis and/or correction is completed, the resulting defects model is converted into a readable HTML report for the user through a M2 T transformation, listing every usability defect found in the GUI in a table displaying its main properties and, in the case of correction, showing an attribute that tells if the defect has been corrected or not. The reports are the end of the process, a process that takes a GUI model and a usability catalogue as input and performs, through a chain of transformations and intermediate representations, the usability analysis and/or correction of the GUI.

Once we have completed the implementation of the architecture, we carry out a validation process by testing the system with some example GUIs containing several usability defects meant to be detected and, in some cases, corrected. In particular, we design a test case with a simple interface containing a single window with some of the most common GUI components such as labels, input fields, images or buttons, and we include in this interface a big enough set of usability defects. We validate the system through the analysis and correction of this faulty GUI, studying the results we obtain from both processes. This turns out to be useful not only in checking the appropriate performance of the system, but also in understanding some of its main features, as well as its strengths and weaknesses. We observed how the correction of certain usability defects can cause the emergence of new ones, starting a chain reaction that can usually be resolved with one more execution of the correction process. This is important to keep in mind in case of more complex interfaces. We also observed that, even though the usability rules catalogue defined is big enough to take into account a great variety of defects, not all the rules have a correction code associated and in some cases it becomes necessary for the user to take part in this correction process to completely remove every usability defect found in the GUI.

We conclude this document with some words that sum up the contribution of this work and emphasize on the areas of future work, including the completion of the architecture through the injection of specific interface technologies and the optimization of the process by linking analysis and correction together.

3 Introducción

Una aplicación debe ser fácil de usar y aprender, esto es usable, para alcanzar el éxito con los usuarios a los que se destina. Por ello, la usabilidad es uno de los aspectos fundamentales a tener en cuenta en el diseño de interfaces de usuario y constituye un campo de estudio de la ingeniería del software. Las interfaces gráficas de usuario (GUI) de las aplicaciones son construidas por equipos que no sólo incluyen desarrolladores de software sino también profesionales de áreas como diseño gráfico o interacción persona-ordenador. A lo largo de los años ha surgido un conjunto de principios, técnicas y reglas que guían la construcción de GUI usables. La evaluación de la usabilidad es una práctica habitual durante y tras el diseño de una GUI, es realizada normalmente de forma manual y requiere un esfuerzo considerable. Por tanto, la automatización de las pruebas de usabilidad supondría un beneficio importante ya que disminuiría el tiempo dedicado a la tarea de evaluación. Esta automatización requiere construir herramientas capaces de analizar una interfaz de usuario en base a unas reglas conocidas de usabilidad para detectar defectos y siempre que sea posible corregirlos, todo ello de manera automática.

En la última década la Ingeniería de Software Dirigida por Modelos (Model Driven Software Engineering, MDSE o simplemente MDE) se ha consolidado como uno de los principales paradigmas de desarrollo de software que favorece la automatización [8, 11]. MDE propone un uso sistemático de los modelos en la construcción de software a través de dos técnicas principales: metamodelado y transformaciones de modelos [2, 6]. MDE ha mostrado su utilidad no sólo para la construcción de nuevas aplicaciones sino también en la modernización o reingeniería de software.

En este trabajo se propone una arquitectura dirigida por modelos para el análisis y corrección de defectos de usabilidad. Se ha diseñado y construido una herramienta basada en la arquitectura propuesta que realmente supone un primer paso en un trabajo de investigación sobre aplicación de MDE en la evaluación de usabilidad. Este trabajo fin de grado supone una prueba de concepto de la arquitectura ideada.

La herramienta que presentamos muestra cómo los modelos posibilitan representar reglas de usabilidad y GUI a un alto nivel de abstracción y facilitan obtener una solución independiente de la plataforma y fácilmente extensible. También hemos podido comprobar que las transformaciones de modelos son un mecanismo potente para automatizar el proceso de evaluación de usabilidad.

Para abordar este trabajo se definieron las siguientes etapas:

1. Etapa de estudio

3. Introducción

Se realizó la contextualización del problema, es decir, la lectura de documentación, trabajos, libros, etc. sobre temas relacionados con MDE y la usabilidad de interfaces GUI. Dado que no había recibido formación sobre MDE (se imparte en una asignatura del máster), los tutores me impartieron varios seminarios de introducción a los conceptos y a las tecnologías. Durante el estudio de las tecnologías a utilizar se contemplaron otras opciones además de las escogidas finalmente para la implementación y que también fueron estudiadas como la implementación de las transformaciones modelo a modelo con el lenguaje RubyTL y con Java/EMF, aunque finalmente elegimos ETL/Epsilon.

2. Etapa de definición del problema

Se confeccionó el catálogo de reglas de usabilidad a partir del gran número de reglas y normas de usabilidad existentes y se diseñó la arquitectura del sistema, valorando qué metamodelos y transformaciones eran necesarios para componer el sistema.

3. Etapa de diseño de los metamodelos

Se especificaron todos los metamodelos involucrados en la solución, analizando las necesidades de la arquitectura en cada punto.

4. Etapa de implementación de las transformaciones

Se definieron las transformaciones modelo a modelo y modelo a texto que forman parte de nuestra arquitectura, y que comprenden transformaciones sencillas como la generación de los informes y transformaciones bastante más complejas, como las encargadas del análisis y la corrección de la usabilidad.

5. Etapa de validación

Durante esta etapa se probó el funcionamiento del sistema mediante el análisis y la corrección de interfaces de ejemplo con determinados defectos de usabilidad.

6. Etapa de documentación

En esta etapa se realizó toda la documentación del trabajo realizado.

En la Figura 3.1 podemos ver a través de un diagrama de Gantt la distribución en el tiempo de cada una de estas tareas, desde el comienzo del proyecto a principios de febrero hasta completar la etapa de documentación a finales de julio.

Durante la realización del trabajo se ha utilizado un repositorio Git para mantener el proyecto y la coordinación/comunicación con los tutores se ha realizado a través de reuniones periódicas y por correo electrónico.

La organización de este documento guarda cierto paralelismo con el proceso seguido durante la implementación del sistema, y es la siguiente. La Sección 4 introduce los conceptos básicos de MDE y la noción de usabilidad, y presenta las herramientas usadas: Epsilon y EMFText. La Sección 5 define brevemente el problema abordado y la arquitectura de la solución propuesta se presenta en la Sección 6. La Sección 7 expone los

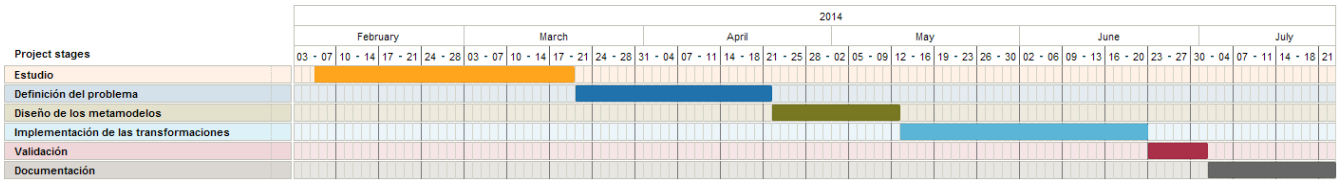


Figura 3.1.: Diagrama de Gantt de las etapas del proyecto.

metamodelos utilizados para representar la interfaz a lo largo del proceso y las transformaciones involucradas. En la Sección 8 definimos el DSL correspondiente a las reglas de usabilidad, presentando las notaciones abstracta y concreta. La Sección 9 describe los procesos de análisis y corrección de la usabilidad. En la Sección 10 mostramos el proceso de validación del sistema a través de un caso de prueba. En la Sección 11 se encuentran las conclusiones y trabajo futuro. Por último, el Anexo A incluye información relativa a la forma en que se ha estructurado el proyecto adjunto a este documento así como las instrucciones para su ejecución, y el Anexo B contiene el catálogo completo de reglas de usabilidad que hemos definido.

4 Fundamentos

En este apartado introduciremos la noción de usabilidad de una GUI y los conceptos básicos que caracterizan a la Ingeniería del Software Dirigida por Modelos y que son necesarios para comprender la descripción del sistema desarrollado que se ofrece en los siguientes capítulos. Asimismo haremos una breve introducción de las tecnologías utilizadas: Epsilon y EMFText.

4.1 Usabilidad de una GUI

Podemos definir una interfaz de usuario como la parte de una aplicación que el usuario ve y con la que interactúa. Dentro del concepto general de interfaz, una GUI (Graphical User Interface) es un tipo de interfaz que permite la interacción del usuario con el sistema a través de iconos gráficos e indicadores visuales, al contrario de lo que ocurre con otros tipos de interfaces basadas en texto como las interfaces CLI (Command-Line Interface). El uso de elementos visuales facilita a los usuarios la interacción con la aplicación, y la combinación más común de dichos elementos en una GUI es el paradigma WIMP (Window, Icon, Menu, Pointing device) que presenta la información organizada en ventanas y representada mediante iconos.

Una ventana es un área de la pantalla que muestra información relacionada con una aplicación, siendo su contenido presentado de forma independiente al resto de la pantalla. La ventana es el elemento estructural principal de una GUI y decimos que es un tipo de contenedor porque contiene al resto de elementos de la interfaz de una aplicación. Dentro de una ventana pueden existir otros contenedores como paneles o marcos (*frames*) que del mismo modo contienen otros elementos, formando de esta manera una estructura jerárquica tal que una GUI puede ser representada por un árbol de componentes donde existen contenedores y contenido, que a su vez puede estar formado por otros contenedores o componentes primitivos (comúnmente referidos como widgets o controles).

Los widgets o controles son componentes software con los que el usuario interactúa a través de su manipulación directa para leer o editar información dentro de una aplicación. Cada widget facilita un tipo concreto de interacción usuario-ordenador, de tal manera que existen widgets para mostrar colecciones de ítems relacionados (como listas), para el inicio de acciones y procesos (botones y menús), para la navegación (enlaces, pestañas y

4. Fundamentos

barras de desplazamiento) o para la representación y manipulación de datos (etiquetas, check boxes, radio buttons,...).

El diseño de la GUI de una aplicación es un proceso complejo que involucra a profesionales de distintas áreas: diseñadores gráficos, programadores, expertos en comunicación persona-ordenador, etc. Normalmente se aplican procesos iterativos e incrementales que parten de bocetos y acaban con la GUI implementada con una determinada tecnología.

En el diseño de interfaces de usuario, el término *usabilidad* se refiere a la claridad y la elegancia con que se diseña la interacción con el usuario, de tal forma que el sistema sea fácil de utilizar y fácil de aprender, permitiendo al usuario centrarse en la tarea y no en la propia aplicación. El término usabilidad está estrechamente relacionado con el de accesibilidad, cualidad que persigue el diseño de interfaces que sean usables por el rango más amplio de personas, en el rango más amplio de situaciones.

En el diseño de interfaces de usuario la usabilidad es un requisito esencial y su estudio ha cobrado mucha importancia en los últimos años [9]. Son muchas las disciplinas y métodos que proponen un diseño centrado en el usuario que asegure el cumplimiento de los principios básicos de la usabilidad en una interfaz: facilidad de aprendizaje, facilidad de uso, flexibilidad y robustez. Una GUI usable reduce los costes de aprendizaje y esfuerzos del usuario, optimiza los costes de rediseño y mantenimiento, aumenta la satisfacción del usuario y mejora la imagen y el prestigio del sistema, entre muchos otros beneficios.

En [9] se aborda en detalle y rigor la usabilidad de GUI. La elicitación de los requisitos de usabilidad es un aspecto crucial del diseño de una GUI usable y permite establecer las pruebas y medidas para evaluarla. El estándar ISO 9421 establece cuáles son las medidas en cuanto a la eficacia, eficiencia y satisfacción del usuario. Entre ellas podemos encontrar: tiempo de aprendizaje, velocidad de realización de tareas, porcentaje de errores de los usuarios, retención con el paso del tiempo y satisfacción subjetiva. La gran diversidad de los posibles usuarios en cuestiones tales como cultura, formación, destrezas y personalidad convierten en un verdadero desafío la construcción de GUI usables. Sin embargo, a lo largo de los años han ido surgiendo guías y recomendaciones (por ejemplo, para la navegación, disposición de elementos en la ventana, facilitar la entrada de datos, capturar la atención del usuario), principios básicos y teorías (por ejemplo, patrones de construcción de GUIs). En nuestro trabajo nos hemos centrado en las guías y recomendaciones que son reglas concretas a aplicar en el diseño de una GUI usable, con el objetivo de automatizar el análisis que comprueba si se satisfacen y la corrección si es necesaria.

Tanto durante el diseño de una GUI como en fases posteriores, la usabilidad se evalúa mediante técnicas que guían el proceso o corrigen los defectos de una interfaz ya diseñada. Los fundamentos y conocimientos relacionados con la usabilidad suelen presentarse en forma de reglas o normas de usabilidad que una interfaz debe satisfacer, abarcando aspectos muy variados dentro de la interfaz y pudiendo referirse a componentes y situaciones concretos o abstractos o bien estar orientados a tipos específicos de interfaz como las interfaces web (para las que existe una gran base de información relativa a la usabilidad). Estas reglas de usabilidad guían los procesos de evaluación, bien de una forma directa o bien como principios que un experto en usabilidad tiene en cuenta durante el análisis de una interfaz.

Las técnicas de evaluación de la usabilidad más comunes pueden dividirse en cuatro categorías principales [9]:

- Técnicas de Sondeo: permiten investigar sobre la satisfacción de los usuarios observando si están conformes con el diseño actual o se deberían plantear cambios. Las técnicas de sondeo incluyen métodos de evaluación como las encuestas de contexto, las sesiones capturadas, las encuestas personales o los logs auto-reportados.
- Técnicas de Inspección: en este tipo de técnicas un usuario experto realiza la revisión del sistema sin que medie ninguna persona de la plantilla encargada del desarrollo de la interfaz. Dentro de esta categoría se incluyen técnicas de evaluación como la evaluación heurística, las inspecciones formales o los checklist de usabilidad.
- Técnicas de Testing: según el tipo de interactividad que buscamos en nuestro sistema, se diseña un test y se selecciona un grupo de usuarios para que ejecuten todas las pruebas del test, analizando posteriormente los datos obtenidos. Incluye técnicas como el método de co-descubrimiento o la medida de rendimiento.
- Técnicas Auxiliares: son técnicas que no se engloban en ninguna de las categorías anteriores pero que ayudan igualmente al diseño de una buena interfaz de usuario, permitiendo medir la usabilidad de la misma. Por ejemplo, el prototipado o el uso de diagramas de afinidad.

En todos los casos anteriores interviene un experto en usabilidad, ya sea interno o externo a la empresa, o bien el propio equipo de diseño, familiarizado con la propia GUI y con los principios de usabilidad de interfaces. Las reglas de usabilidad guían el proceso y orientan a los evaluadores, pero no es común la aplicación automatizada de las mismas pues en la mayoría de los casos las reglas de usabilidad trabajan con aspectos abstractos y altamente subjetivos de una GUI. Los principios de usabilidad anteriormente mencionados (facilidad de aprendizaje y uso, flexibilidad y robustez) se organizan en reglas más concretas pero muchas veces igual de subjetivas, y esto supone un reto para todo sistema de evaluación automática de la usabilidad.

El sistema que aquí presentamos es un ejemplo de un proceso automatizado de evaluación de la usabilidad mediante *checklist*, es decir, una lista de reglas de usabilidad que el sistema debe cumplir o respetar para ser considerado usable.

4.2 Introducción a MDE

La Ingeniería del Software Dirigida por Modelos (*Model-Driven Software Engineering*, MDSE o simplemente MDE) es un área de la Ingeniería del Software que ha despertado un interés creciente a lo largo de los años transcurridos del siglo XXI. MDE se ocupa de la utilización sistemática de los modelos en las diferentes etapas de los procesos de producción de software con el fin de mejorar la productividad a través de un incremento

4. Fundamentos

en los niveles de abstracción y automatización. Una solución MDE se basa en la aplicación de transformaciones sobre modelos. Estos modelos son expresados por un lenguaje de modelado que se define por medio de un metamodelo. Por tanto, podemos considerar que los cuatro elementos clave de MDE son: *modelo*, *metamodelo*, *transformación de modelos* y *lenguaje de modelado*.

La Figura 4.1 ilustra la relación entre modelo y metamodelo a través de un paralelismo entre los conceptos de programa y gramática. De la misma forma que un programa es conforme a una gramática, un modelo es conforme a un metamodelo que determina su estructura. Además, del mismo modo que una gramática es expresada por una notación especial, como es la EBNF, también son necesarios lenguajes de metamodelado para expresar los metamodelos, uno de los más extendidos es Ecore que es el elemento central del framework de meta modelado EMF (Eclipse Modeling Framework) de Eclipse. La Figura 4.1 expresa, por tanto, que un modelo UML, por ejemplo un diagrama de clases, es conforme a un metamodelo de UML que ha sido definido con Ecore y se establece el paralelismo con un programa Java que es conforme a la gramática de Java que se ha expresado con el lenguaje EBNF.

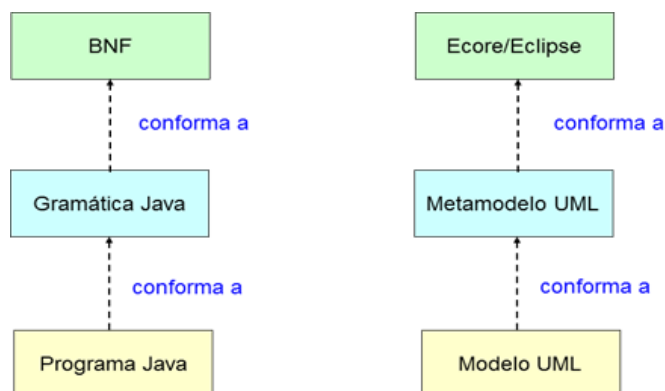


Figura 4.1.: Comparación entre metamodelos y gramáticas.

Como veremos con los metamodelos creados en nuestro proyecto, Ecore proporciona los conceptos básicos del modelado conceptual basado en objetos para crear el metamodelo de un dominio: los conceptos son representados por clases, las propiedades de los conceptos por atributos de la clase, las relaciones entre conceptos por medio de asociaciones entre clases (referencias y agregación) y la especialización de conceptos por medio de herencia de clases. Un metamodelo también incluye un conjunto de reglas en OCL [10, 1] o lenguaje similar que establece restricciones a ser satisfechas por un modelo bien formado.

Cada metamodelo define un lenguaje de modelado asociado, de la misma forma que un lenguaje de programación es definido por una gramática. En el caso de los lenguajes de modelado se diferencia entre la sintaxis abstracta (metamodelo que expresa los conceptos del lenguaje y relaciones entre ellos) y la sintaxis concreta que expresa la notación del lenguaje. Todos los modelos especificados con un lenguaje de modelado deberán ser una

instanciación correcta de su metamodelo. Se suele utilizar el término “conforma con” para expresar estas relaciones de instanciación. Por tanto, la estructura de un modelo es determinado por la del metamodelo al que conforma (del que es una instancia).

Los lenguajes de modelado pueden ser gráficos, textuales o híbridos según la naturaleza de la representación que permiten crear. Existen dos grandes categorías de lenguajes de modelado:

- Lenguajes específicos del dominio (DSL) que son lenguajes diseñados específicamente para expresar modelos para un dominio, contexto o empresa. Un ejemplo sería un DSL diseñado por un fabricante de *smartphones* para expresar modelos con los que generar código para una plataforma de creación de aplicaciones móviles.
- Lenguajes de modelado de propósito general que son lenguajes diseñados para expresar modelos para cualquier dominio y contexto. Un ejemplo típico sería UML.

Los DSLs como alternativa a los lenguajes de programación de propósito general (GPL) han existido desde los primeros días de la programación. Ejemplos de DSLs utilizados desde hace muchos años son SQL, make, Excel y VHDL. Sin embargo, con la aparición de MDE ha crecido de forma considerable el interés por ellos. Mientras que tradicionalmente los DSLs se han construido con la tecnología (Grammarware) usada para crear GPLs, en el contexto de MDE se están creando con la propia tecnología MDE (Modelware) y han aparecido varias herramientas que automatizan la creación de DSLs, por ejemplo GMF¹ para DSLs gráficos y Xtext² y EMFText³ para DSLs textuales.

MDE se puede usar en diferentes escenarios [2], como son desarrollo de nuevas aplicaciones (ingeniería directa) o tareas de reingeniería o modernización de software (ingeniería inversa). En todos ellos la solución MDE que automatiza una tarea de desarrollo de software se implementa como una cadena de transformaciones en la que dado un modelo fuente se generan artefactos software de una aplicación (por ejemplo código fuente o archivos XML) o se modifican artefactos de una aplicación existente. En nuestro caso, se trata de un proceso de modernización en el que se mejora la usabilidad de una interfaz existente y en el que por tanto es preciso llevar a cabo una ingeniería inversa de la interfaz, la cual se lleva a cabo a través de una cadena de transformaciones.

Una cadena de transformaciones involucra normalmente transformaciones modelo-a-modelo (M2M) y transformaciones modelo-a-texto (M2T). La Figura 4.2 muestra un ejemplo de cadena de transformación en la que dado un modelo inicial se generan artefactos de una aplicación. El objetivo de las transformaciones M2M es reducir el salto semántico desde el metamodelo al que conforma el modelo origen al código final generado mediante el uso de modelos intermedios. Por tanto, cuando este salto semántico es pequeño la cadena puede reducirse a una única transformación M2T.

En el caso de aplicar MDE en un proceso de reingeniería o modernización de software es preciso aplicar también un tercer tipo de transformación de modelos: transformaciones

¹<http://www.eclipse.org/gmf-tooling/>.

²<http://www.eclipse.org/Xtext/>.

³<http://www.emf-text.org/index.php/EMFText>.

4. Fundamentos

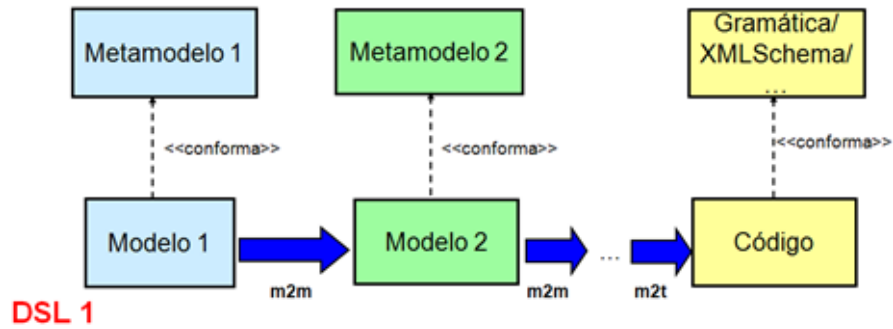


Figura 4.2.: Cadena de transformación de modelos en ingeniería directa.

texto-a-modelo (T2M), las cuales obtiene un modelo a partir de un documento textual (por ejemplo, código fuente). Este proceso se denomina comúnmente inyección de modelos. Por ejemplo, la Figura 4.3 ilustra una cadena en la que un documento XML que representa una GUI es inyectado en un modelo y entonces se aplicaría sobre él un proceso de ingeniería inversa que en una o más etapas (transformaciones M2M) obtendría algún conocimiento en forma de modelos (por ejemplo, defectos de usabilidad) a partir del cual se podrían obtener nuevos artefactos software (de nuevo documentos XML que representan la GUI corregida).

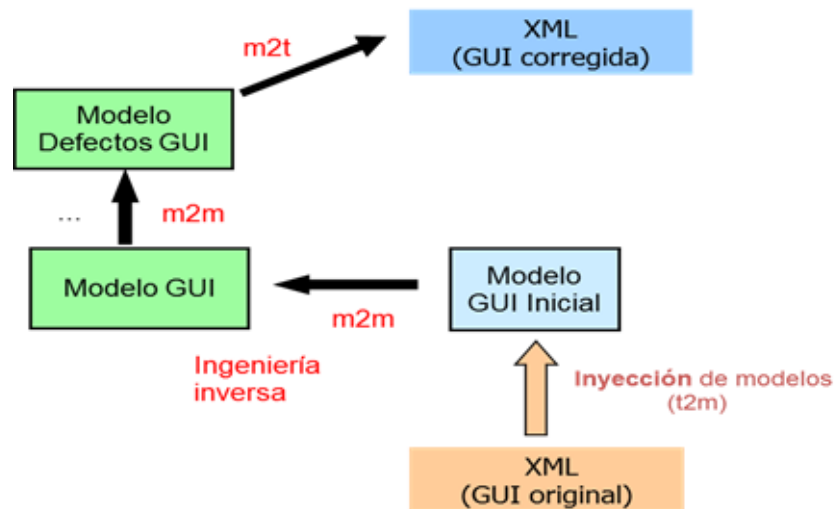


Figura 4.3.: Cadena de transformación de modelos en ingeniería inversa.

Las transformaciones se pueden implementar en un lenguaje de programación como Java haciendo uso de un API de manejo de modelos como el API de EMF, pero se suelen escribir con alguno de los muchos DSLs que se han definido en los últimos años para

tal fin, entre los que destacan: ATL⁴ y QVT⁵ para transformaciones M2M; Acceleo⁶, y Mofscript⁷ para transformaciones M2T, y Gra2MoL [4] y los inyectores de Modisco [3] para transformaciones T2M.

Los lenguajes de transformaciones modelo-a-modelo proporcionan construcciones para definir correspondencias entre elementos de un metamodelo fuente y otro destino, de modo que la ejecución de la transformación pueda generar un modelo destino a partir de uno de entrada. Los lenguajes de transformación modelo-a-texto proporcionan construcciones que permiten recorrer un modelo de acuerdo a la estructura definida en el metamodelo y expresar qué texto se debe generar de acuerdo a los elementos del metamodelo accedidos.

Debe notarse que aunque distintos, todos los elementos del Modelware (espacio tecnológico MDE) son modelos: un metamodelo es un modelo con un objetivo específico y una transformación también es un modelo cuando se escribe con un lenguaje definido con un metamodelo. Esta unificación de conceptos permite tratar de una forma homogénea todos los elementos involucrados en un proyecto y es una de las principales propiedades de MDE.

El lector puede ampliar detalles sobre MDE en [2] que ofrece una visión completa de todos los conceptos, incluso de algunos temas avanzados. Una introducción a MDE en español y más centrada en las herramientas que en los conceptos puede encontrarse en [6].

4.3 Tecnologías utilizadas

4.3.1 Epsilon

Epsilon (Extensible Platform of Integrated Languages for mOdel maNagement) es una plataforma para la construcción de lenguajes consistentes e interoperables específicos para tareas de gestión de modelos como transformación de modelos, generación de código, comparación de modelos, mezcla, refactoring y validación. Actualmente Epsilon proporciona los siguientes lenguajes:

- *Epsilon Object Language (EOL)*

Lenguaje principal de Epsilon sobre el que se construyen el resto de lenguajes.

- *Epsilon Validation Language (EVL)*

Lenguaje destinado a la validación de modelos.

- *Epsilon Transformation Language (ETL)*

Lenguaje para la implementación de transformaciones modelo a modelo.

⁴<http://www.eclipse.org/atl/>.

⁵<http://www.omg.org/spec/QVT/1.1/>.

⁶<http://www.eclipse.org/acceleo/>.

⁷<http://www.eclipse.org/gmt/mofscript/>.

4. Fundamentos

- *Epsilon Comparison Language (ECL)*

Lenguaje para la comparación de modelos, es decir, la búsqueda de elementos similares entre modelos.

- *Epsilon Merging Language (EML)*

Lenguaje para la combinación de dos o más modelos.

- *Epsilon Wizard Language (EWL)*

Lenguaje para la definición y ejecución de transformaciones de actualización en modelos de diversos metamodelos.

- *Epsilon Generation Language (EGL)*

Lenguaje para la implementación de transformaciones modelo a texto.

Para cada uno de ellos, Epsilon proporciona herramientas de desarrollo basadas en Eclipse y un intérprete para la ejecución de programas escritos con ellos. Epsilon también proporciona un conjunto de tareas ANT para la creación de flujos de trabajo.

En este apartado nos centraremos en los lenguajes EOL, ETL y EGL que son los utilizados en el desarrollo de nuestro sistema.

4.3.11 EOL

El objetivo principal de EOL es proporcionar un conjunto común y reutilizable de facilidades para la gestión de modelos, sobre el cual se puedan implementar los lenguajes específicos. Es decir, es la base del conjunto de lenguajes que ofrece Epsilon e imprescindible para trabajar con cualquiera de ellos.

EOL ofrece todos los mecanismos que caracterizan a un lenguaje de alto nivel: definición de operaciones, tipos de datos, operadores (aritméticos, lógicos, de comparación, ...), expresiones (declaración de variables, asignaciones, condicionales, iteraciones, ...) y otros aspectos más avanzados como propiedades extendidas o la entrada de datos por parte del usuario. Mencionaremos aquí brevemente algunas de las características más importantes del lenguaje.

La Figura 4.4 muestra el sistema de tipos de EOL. A destacar la existencia de cuatro tipos de colección, además de `Map`, y la distinción únicamente de dos tipos de datos numéricos: `Integer` y `Real`. Todos los tipos tienen una serie de métodos comunes, de entre los que destacamos métodos de comprobación de tipos (`isTypeOf`, `isKindOf`), métodos de comprobación de propiedades (`hasProperty`) y métodos para la conversión entre tipos (`asString` o `asSet`, entre otros). Asimismo, destaca el soporte de iteradores sobre conjuntos con métodos como `select`, `collect` o `sortBy`, de gran potencia y utilidad para el trabajo con colecciones.

Por último, indicar que EOL permite la definición de operaciones especificando un contexto, es decir, un tipo de dato, de tal forma que las operaciones que así se definan podrán ser invocadas sobre ese tipo de dato como si se hubieran definido de forma nativa.

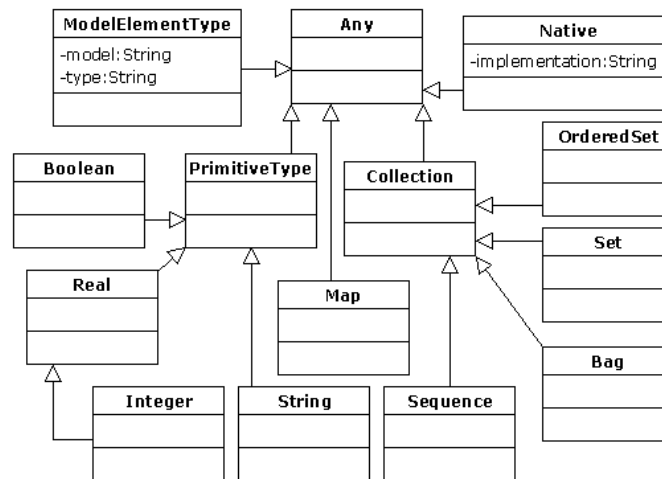


Figura 4.4.: Sistema de tipos de EOL.

En el siguiente extracto de código vemos un ejemplo del funcionamiento de este tipo de operaciones:

```

"1".test();
1.test();

operation String test() {
    (self + " is a string").println();
}

operation Integer test() {
    (self + "is an integer").println();
}
  
```

Dos operaciones de igual nombre `test` se definen sobre los tipos `String` e `Integer` respectivamente. Al principio del código vemos cómo dichas operaciones pueden ser invocadas sobre uno u otro tipo de dato como lo haríamos con cualquiera de las otras operaciones definidas de forma nativa para los tipos `String` e `Integer`.

Nosotros hemos utilizado EOL dentro de todas las transformaciones que hemos implementado, tanto modelo a modelo como modelo a texto. En particular, los extractos de código asociados a cada una de las reglas de usabilidad del catálogo son código EOL, y éste es el lenguaje que uno debe aprender para confeccionar sus propias reglas de usabilidad.

4.3.12 ETL

ETL es el lenguaje definido por Epsilon para las transformaciones modelo a modelo.

ETL ha sido diseñado como un lenguaje híbrido que soporta la definición de reglas declarativas pero también hereda las características imperativas de EOL para manejar transformaciones complejas. Existe un consenso sobre la necesidad de que los lenguajes

4. Fundamentos

de transformación modelo a modelo sean híbridos para soportar la definición de transformaciones complejas difíciles de implementar con reglas declarativas.

ETL puede convertir un número arbitrario de modelos de entrada en un número arbitrario de modelos de salida, y sus reglas de transformación siguen la siguiente sintaxis:

```
(@abstract)?
(@lazy)?
(@greedy)?
rule <name>
    transform <sourceParameterName>:<sourceParameterType>
    to <rightParameterName>:<rightParameterType>
        (, <rightParameterName>:<rightParameterType>)*
    (extends <ruleName>(<ruleName>)*)? {

        (guard (:expression)|({statementBlock}))?

        statement+
    }
}
```

Existen tres tipos de reglas: **abstract**, **lazy** y **greedy**.

Una regla **lazy** es una regla que sólo es ejecutada cuando se invoca desde otra regla de transformación. Si no se etiqueta una regla como **lazy**, por omisión se considerará que esa regla se ejecuta directamente sin necesidad de ser invocada.

Una regla **abstract** es una regla que nunca es ejecutada por sí misma, sólo como parte de la ejecución de otra regla que hereda de ella.

Una regla **greedy** es una regla que se aplica para todos los elementos cuyo tipo es el tipo indicado en su parte **transform** o bien uno de sus subtipos.

El nombre de la regla sigue a la palabra clave **rule** y los parámetros de entrada y salida se definen tras las palabras clave **transform** y **to**. Estos parámetros están formados por un par nombre de variable y metacalse del metamodelo que corresponde a un elemento origen o destino de la transformación.

Además, la regla puede definir una lista opcional de reglas separadas por comas de las que hereda la regla actual mediante la palabra clave **extends**. La herencia de reglas facilita la construcción de los ficheros de transformación, permitiendo poner en común parte de la definición. Una regla padre se ejecutará cada vez que se invoque una de sus hijas y antes de que ésta se ejecute.

Entre corchetes, la regla puede especificar de forma opcional una guarda, bien con una expresión EOL precedida por dos puntos (:) o bien como un bloque de sentencias entre corchetes (útil para guardas complejas).

Por último, el cuerpo de la regla se especifica como una secuencia de sentencias EOL.

Además, ETL proporciona la posibilidad de definir dos bloques de código, **pre** y **post**, que serán ejecutados antes y después de la transformación respectivamente.

```
(pre|post) <name> {
    statement+
}
```

ETL dispone de un método **equivalents** que, aplicado sobre un elemento de origen, inspecciona el árbol de trazabilidad manejado internamente e invoca si es necesario las

reglas de transformación aplicables sobre el elemento de origen para calcular sus correspondientes elementos en el modelo destino. Habitualmente, la invocación del método `equivalents` sobre un elemento va seguida de la asignación o adición de los elementos obtenidos a una propiedad u otro elemento del modelo destino.

Para hacer las transformaciones más legibles, ETL dispone del operador especial de asignación (`:=`), que equivale a la invocación del método `equivalents` sobre el elemento de la parte izquierda y la asignación de los valores obtenidos sobre el elemento o propiedad de la parte derecha. Este tipo de asignaciones disparará la ejecución de nuevas reglas de transformación por lo que, en definitiva, son las que guían el proceso completo de transformación modelo a modelo.

4.3.1.3 EGL

EGL [7] es el lenguaje ofrecido por Epsilon para las transformaciones modelo a texto.

EGL es un generador de código basado en plantillas que proporciona una serie de características que simplifican la generación de texto a partir de modelos, incluyendo: un sofisticado motor de mezclado, algoritmos de formateado, un sistema extensible de plantillas y mecanismos de trazabilidad.

Un programa EGL se compone de una o más secciones que pueden ser estáticas o dinámicas. El contenido de las secciones estáticas se vuelca directamente en el texto generado. El contenido de las secciones dinámicas se ejecuta y se utiliza para obtener el texto que se genera. Dentro de una sección dinámica incluiremos código EOL que recorre los modelos y obtiene información de ellos para llevarlo a la salida.

La sintaxis concreta de EGL guarda muchas semejanzas con la de otros lenguajes basados en plantillas como PHP. Se utiliza el par de etiquetas [% y %] para delimitar las secciones dinámicas. Cualquier texto que no se sitúe dentro de estas etiquetas se considera dentro de una sección estática y es por tanto volcado directamente en el texto generado.

La expresión [%=expr %] se utiliza como un atajo para [% out.print(expr); %], lo que supone agregar `expr` a la salida generada por la transformación.

El siguiente extracto de código representa una sencilla plantilla EGL que ilustra el uso de las secciones estáticas y dinámicas:

```
[% for (i in Sequence{1..5}) { %]
    i is [%=i%]
[% } %]
```

El texto generado a partir de la ejecución de la plantilla EGL anterior sería el siguiente:

```
i is 1
i is 2
i is 3
i is 4
i is 5
```

4. Fundamentos

4.3.2 EMFText

EMFText es una herramienta que permite definir la sintaxis textual de lenguajes descritos mediante un metamodelo Ecore. EMFText hace posible la definición de DSLs textuales de una forma rápida y sin la necesidad de aprender nuevas tecnologías ni conceptos.

A partir de la sintaxis textual y del metamodelo con la sintaxis abstracta de un DSL, EMFText es capaz de generar un editor en Eclipse que permite la edición de modelos siguiendo la sintaxis textual definida, con marcado de código y corrección de sintaxis, así como un inyector que nos permite convertir el texto escrito mediante el editor en un modelo serializado en formato XML.

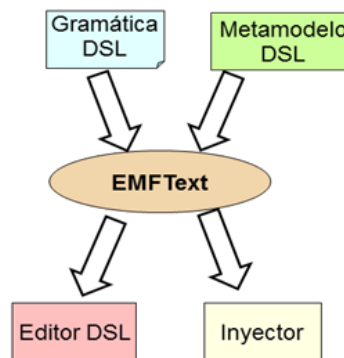


Figura 4.5.: Arquitectura EMFText.

La especificación de la sintaxis textual mediante EMFText consta de cinco secciones:

- En la primera sección se define la extensión que tendrá el fichero con la representación textual, se establece el metamodelo correspondiente y la entidad de inicio del metamodelo.
- En la segunda sección se pueden configurar determinadas opciones relativas a la generación de código.
- En la tercera sección se definen los tipos de token usados por el lenguaje.
- En la cuarta sección se especifican los estilos para cada token, permitiendo personalizar el marcado de código que realizará el editor.
- En la quinta y última sección se definen las reglas sintácticas del lenguaje.

En la definición de las reglas utilizadas para la especificación de la sintaxis intervienen una serie de conceptos que ilustraremos con un simple DSL para el metamodelo de la Figura 4.6 que representa un tipo de grafo [5]. Abajo se muestran las reglas sintácticas para este metamodelo y vemos que tenemos una regla para cada metaclassa excepto `NamedElement` que es abstracta y no tendrá notación asociada.

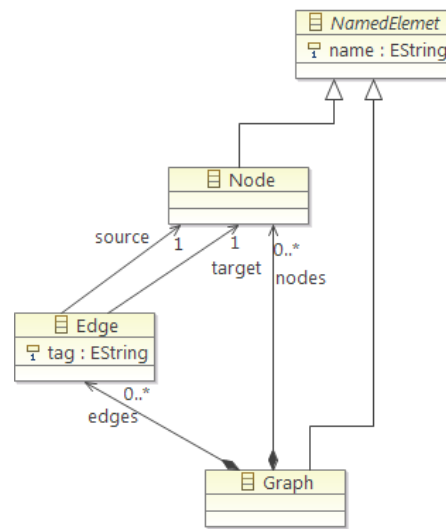


Figura 4.6.: Metamodelo de un sencillo grafo.

```

SYNTAXDEF graph
FOR <http://www.modelum.es/graph>
START Graph

TOKENS {
  DEFINE COMMENT $'/'/'/'/('~('\'n'|\'r'|\'u{ffff}')*)$;
}

TOKENSTYLES {
  "Graph" COLOR #7F0055, BOLD;
}

RULES {
  Graph ::= "Graph" name[] "{" "Nodes" "{" nodes ("," nodes)* "}" "Edges" "{"
    edges* "}" "}" ;
  Node ::= name[] ;
  Edge ::= source[] "->" target[] "(" tag[] ")" ;
}
  
```

Los conceptos involucrados en la definición de la sintaxis son:

- Regla: cada regla está asociada con una clase del metamodelo y define la representación sintáctica del elemento representado por dicha metaclass, así como de sus atributos y referencias. En el ejemplo, se definen reglas para las clases `Graph`, `Node` y `Edge` que componen el metamodelo.
- Palabras clave: las palabras clave son elementos puramente sintácticos que se utilizan principalmente para estructurar y marcar expresiones del lenguaje. En las reglas anteriores podemos observar palabras clave como `"Graph"`, `"Nodes"`, `"Edges"`.

4. Fundamentos

- **Atributo terminal:** se utilizan en los cuerpos de las reglas para especificar la representación sintáctica de los atributos de la correspondiente metaclase. Se representan con el nombre del atributo seguido de corchetes, dentro de los cuales puede haber un token que especifica la sintaxis permitida para los valores del atributo, o un prefijo y un sufijo que rodearán el valor del atributo. En las reglas anteriores observamos los atributos **name** tanto para el grafo como para el nodo, así como los atributos **source**, **target** y **tag** correspondientes a la metaclase **Edge**.

A partir de la definición de una gramática, EMText genera el parser y el editor. Entonces podríamos escribir una especificación como la de abajo y obtendríamos el correspondiente modelo conforme al metamodelo de la Figura 4.6.

```
Graph MyGraph {  
    Nodes { A,B,C}  
    Edges {  
        A -> B (2)  
        B -> C (5)  
    }  
}
```

5 Definición del problema

La evaluación de la usabilidad de una interfaz gráfica de usuario es realizada normalmente de forma manual, como se señaló en la Sección 4.1, lo cual supone un gran esfuerzo y coste de tiempo. En este trabajo se explora la utilidad de las técnicas MDE para automatizar la detección y corrección de defectos de usabilidad en interfaces gráficas de usuario. Para ello se ha definido una arquitectura basada en modelos para el proceso de evaluación de la usabilidad de una interfaz. Esta arquitectura debe cumplir una serie de requisitos que serán tenidos en cuenta durante la construcción del sistema y que son los siguientes:

- Independencia de la tecnología fuente y destino
- Catálogo de reglas de usabilidad extensible
- Reglas de usabilidad genéricas (consecuencia del primer requisito)

La usabilidad abarca muchas características de una interfaz, incluyendo aspectos subjetivos, por esta razón y como es lógico no pretendemos una automatización completa del análisis de la usabilidad de una GUI dado que para ello es preciso considerar conocimientos y heurísticas de expertos que no son factibles o son muy complicados de representar y procesar algorítmicamente. En cambio, un sistema que automatice el análisis de la usabilidad puede abarcar aquellos defectos que pueden ser identificados mediante un procesamiento de la representación interna de la interfaz (por ejemplo, formato XML) y en nuestro caso hemos elegido un conjunto de defectos sencillos y comunes (por ejemplo, el número de diferentes tipografías usadas en una interfaz) como prueba de concepto. Según hemos podido investigar, nuestro trabajo es el primero que aborda el uso de MDE en la evaluación de la usabilidad y nuestro objetivo es arrancar un proyecto de investigación en este campo destinado a analizar las ventajas e inconvenientes de las técnicas MDE para este problema.

Hemos definido un pequeño catálogo que incluye dieciocho reglas de usabilidad y accesibilidad las cuales se han extraído del amplio conjunto de reglas que se aplican normalmente en las pruebas de usabilidad siguiendo los siguientes criterios:

- Ser genéricas, es decir, aplicables sobre cualquier tipo de interfaz
- No ser subjetivas con respecto a la identificación, si bien pueden serlo para la corrección

5. Definición del problema

- Considerar aspectos básicos como texto, color o posición
- Contemplar situaciones o condiciones comunes en una interfaz y no demasiado específicas

La Tabla 5.1 muestra de forma resumida el catálogo de reglas, indicando únicamente su nombre y una breve descripción. Para un listado completo y mucho más detallado del catálogo de reglas de usabilidad contemplado, se puede consultar el Anexo B.

Las reglas elegidas definen normas básicas como las relativas a la alineación o separación entre elementos, las relacionadas con el tamaño de fuente o la longitud del texto o las que tratan aspectos como el contraste o la paleta de colores. También se han incluido reglas relacionadas con propiedades como *tabfocus* y *tooltip* y se ha prestado especial atención a los elementos **Label** (etiqueta) e **Input** (campo de entrada), con los que se forman los formularios de entrada de información que se utilizan en la mayoría de aplicaciones.

En general, se ha tratado de contemplar un abanico lo suficientemente amplio de reglas de usabilidad, excluyendo aquellas demasiado abstractas o demasiado orientadas a un tipo específico de interfaz, con el objetivo de cumplir el principio de independencia de la tecnología. De esta forma, descartamos para nuestra solución reglas de usabilidad orientadas, por ejemplo, a dispositivos táctiles o a interfaces web. De la misma manera, no contemplamos reglas de usabilidad que hacen referencia a conceptos como la importancia de los elementos, la frecuencia con que se usan o el contenido semántico del texto. Son todos ellos aspectos altamente subjetivos, más apropiados para la percepción de un experto en usabilidad que para el análisis automatizado de un sistema software como el nuestro.

El listado presentado en la Tabla 5.1 introduce todas las reglas de usabilidad contempladas, pero no todas ellas tendrán asociado un proceso automatizado de corrección. Esto se debe, una vez más, a la subjetividad inherente a muchas de las reglas.

Es fácil, por ejemplo, detectar que en un elemento no se ha establecido la información de accesibilidad, pero es una cuestión muy diferente dar un valor a esta propiedad, pues para ello debemos saber cuál es la función del elemento dentro de la interfaz y poder explicarla de una manera comprensible en pocas palabras.

No obstante, la corrección automática asociada a problemas de posición, color o formato del texto puede ser muy útil. La rectificación de defectos de usabilidad como la mala alineación o la escasa separación de elementos puede funcionar muy bien y ahorrar trabajo al usuario, especialmente en interfaces de usuario con muchos elementos.

En definitiva, el objetivo es realizar un análisis básico de la usabilidad diseñado en torno a un catálogo de reglas elemental pero **ampliable**, permitiendo al usuario detectar una serie de problemas comunes de usabilidad en su interfaz y facilitando la **corrección automática** de aquellos defectos que lo permitan.

El sistema debe diseñarse primando ante todo la **generalidad** y la **independencia**, mediante la abstracción, del tipo concreto de interfaz subyacente.

La estructura de reglas de usabilidad debe contemplar el catálogo anteriormente presentado, permitiendo al mismo tiempo la definición, mediante un método lo más sencillo

Nombre	Definición
Tamaños de fuente	Comprueba el número de tamaños de fuente distintos utilizados en la interfaz
Tabfocus	Comprueba si la propiedad tabfocus ha sido establecida de una manera adecuada y consistente en los elementos de un contenedor
Paleta de colores frontal	Comprueba si el color frontal de un elemento se encuentra dentro de la paleta especificada
Paleta de colores de fondo	Comprueba si el color de fondo de un elemento se encuentra dentro de la paleta especificada
Contraste	Comprueba el contraste entre los colores frontal y de fondo de un elemento
Separación	Comprueba la separación entre un elemento y el resto de elementos dentro de su mismo contenedor
Tipos de fuente	Comprueba el número de tipos de fuente diferentes usados en la interfaz
Separación Etiqueta-Campo de entrada	Comprueba la separación entre un campo de entrada y su correspondiente etiqueta
Profundidad de menú	Comprueba la profundidad máxima de la estructura de un menú
Longitud de texto de etiqueta	Comprueba la longitud máxima del texto de una etiqueta
Tamaño de fuente	Comprueba el tamaño mínimo de fuente de cualquier texto
Alineación vertical	Comprueba si un elemento está alineado verticalmente con el resto de elementos de su contenedor
Alineación horizontal	Comprueba si un elemento está alineado horizontalmente con el resto de elementos de su contenedor
Número de elementos	Comprueba el número total de elementos en un contenedor
Etiquetado de campo de entrada	Comprueba si un campo de entrada tiene asociada una etiqueta
Información de accesibilidad	Comprueba si la información de accesibilidad se ha establecido para un elemento
Tooltip	Comprueba si el tooltip ha sido establecido para un elemento
Solapamiento	Comprueba si un elemento se solapa con otros elementos de su contenedor

Tabla 5.1.: Reglas de usabilidad.

5. Definición del problema

posible, de reglas de usabilidad adicionales.

Por último, el resultado de los procesos de análisis y posterior corrección **deberá mostrarse al usuario de forma clara y bien estructurada**, pues es fundamental involucrar al usuario en el proceso de manera que entienda los aspectos analizados y los posibles cambios realizados sobre su interfaz.

6 Visión general de la arquitectura propuesta

Una vez presentado el problema que se pretende abordar, esto es, la automatización de los procesos de detección y corrección de defectos de usabilidad en las interfaces de usuario, en este apartado se presentará la arquitectura basada en modelos que se ha diseñado como solución y que ha sido implementada por medio de la aplicación de técnicas MDE.

Nuestra solución consta de dos componentes principales, uno encargado de analizar la interfaz para la detección de defectos y otro que realiza automáticamente las correcciones de defectos. La Figura 6.1 muestra el esquema general del componente de análisis y la Figura 6.2 muestra el esquema general de la parte de corrección. Como muestran ambas figuras primero es necesario obtener (inyectar) un modelo que represente la GUI a analizar. Para ello es preciso inyectar modelos a partir de los formatos de exportación de la plataforma GUI. Con el fin de independizar la arquitectura definida de las plataformas GUI se ha definido un metamodelo GUI que permite representar de forma abstracta una GUI de cualquier plataforma. Por tanto, los modelos de la plataforma (por ejemplo, los modelos Qt) son transformados en modelos GUI como primer paso de nuestra solución. En el caso de la corrección, como muestra la Figura 6.2 el mismo modelo GUI suministrado como entrada, una vez corregidos los defectos, es utilizado para aplicar los cambios sobre la interfaz de entrada al proceso. Estas dos transformaciones, la que nos permite obtener el modelo GUI a partir de la interfaz de entrada y la transformación que realiza el proceso inverso, forman parte de la arquitectura general de análisis y corrección de la usabilidad pero no han sido implementadas como parte del sistema que proponemos en este documento. Su implementación se plantea como una de las principales vías de futuro de este proyecto, tal y como mencionamos en la Sección 11.

Las dos figuras anteriores muestran como una caja negra la parte central de la arquitectura que tiene que ver con la detección y corrección de defectos a partir del modelo GUI, y que es la que constituye el sistema que presentamos en este documento. Esta parte de la arquitectura se muestra en la Figura 6.3, aplicable tanto al proceso de análisis como al de corrección. Como se puede observar, en ambos casos el proceso se realiza en dos etapas. Primero se transforma el modelo GUI en un modelo Usability-GUI cuyo objetivo es facilitar las acciones a ser ejecutadas durante el análisis. Esta conversión se ha implementado mediante una sencilla transformación M2M. En segundo lugar, se

6. Visión general de la arquitectura propuesta

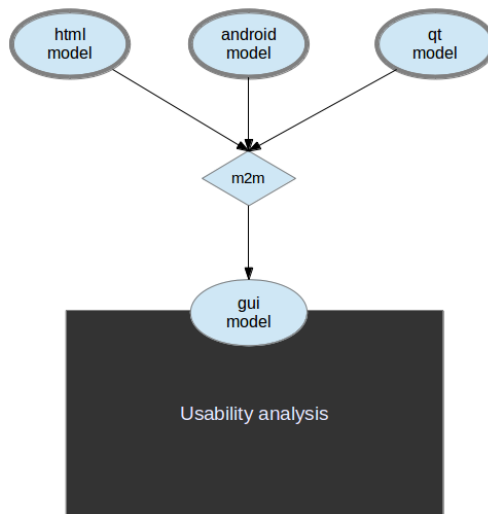


Figura 6.1.: Esquema general de análisis de la usabilidad.

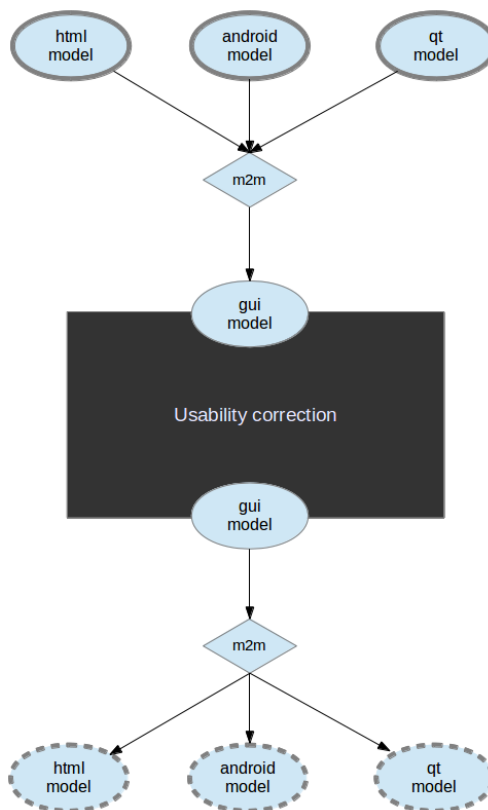


Figura 6.2.: Esquema general de corrección de la usabilidad.

realiza el análisis/corrección de usabilidad por medio de una transformación M2M que tiene como entrada el modelo Usability-GUI y un modelo de reglas de usabilidad y como salida el modelo de defectos. Esta tarea es la más compleja del proceso y conlleva aplicar las reglas de usabilidad del catálogo sobre el modelo Usability-GUI, tanto para la identificación de los defectos como para su corrección. En el caso de la corrección, el modelo GUI original se modifica para corregir los defectos de usabilidad. A partir del modelo de defectos obtenido se genera un informe de defectos por medio de una transformación M2T. Este informe describe los defectos indicando sus atributos y características esenciales y se presenta al usuario en forma de tabla en un documento HTML.

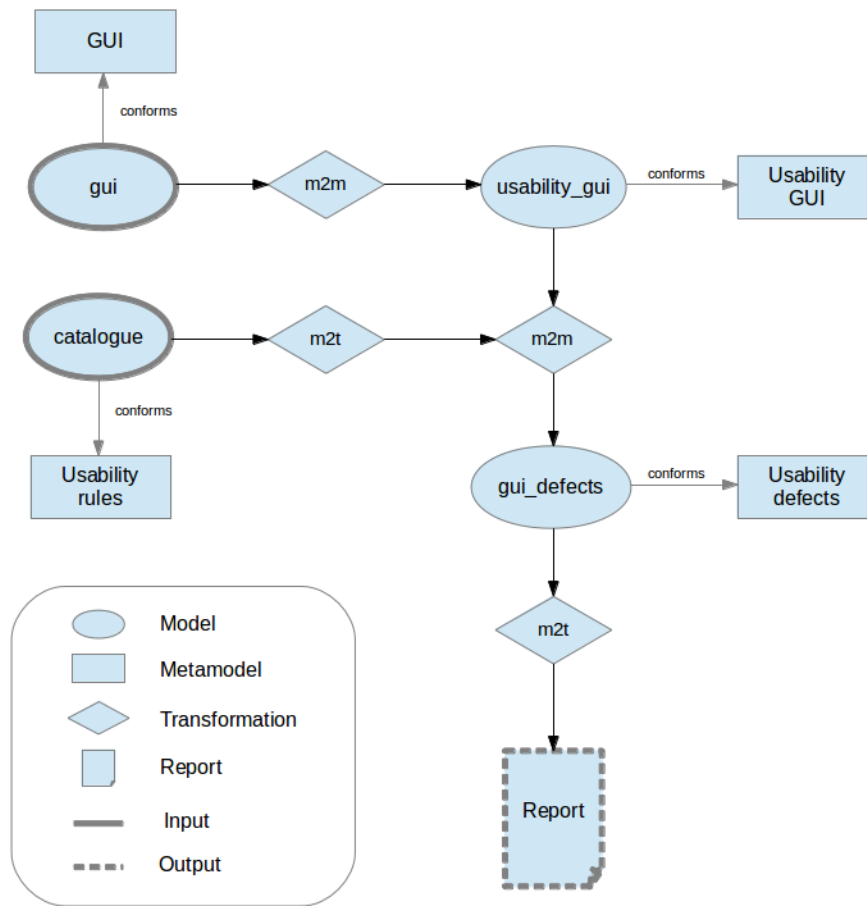


Figura 6.3.: Diagrama de la arquitectura propuesta.

El modelo de reglas de usabilidad no es fijo, sino que se ha creado un DSL (*Domain Specific Language*, Lenguaje específico del dominio) textual para permitir a los expertos definir sus propios modelos de reglas específicos de un determinado contexto (por ejemplo, una empresa que aplica sus propias reglas) o bien añadir reglas al modelo existente. Por tanto, el modelo de reglas es una entrada más del sistema, es decir, no es ni mucho

6. Visión general de la arquitectura propuesta

menos una parte estática del conjunto sino una variable más que el usuario puede establecer. Como veremos más adelante, además de conocer el DSL de definición de reglas, los expertos también deberán conocer el lenguaje EOL (Epsilon Object Language) usado para expresar cómo se detecta y corrige un defecto. En la Sección 8 se presentará el DSL de definición de reglas de usabilidad y se mostrarán ejemplos de uso.

Cabe destacar que, como vemos en la Figura 6.3, la transformación M2M que da lugar al proceso de análisis y corrección de defectos ha sido generada automáticamente por medio de una transformación M2T que tiene como entrada el modelo con el catálogo de reglas. Esta es una de las decisiones de diseño más interesantes de la arquitectura propuesta y será motivada y explicada en detalle en la Sección 9.

Con esto queda expuesta la arquitectura del sistema construido, que ha sido utilizado para implementar los procesos de análisis y corrección. En la Figura 6.3 se encuentran marcados los elementos que constituyen la entrada y salida. El sistema toma como entrada un modelo representando la interfaz y un conjunto de reglas, y genera un informe como salida. En el caso de la corrección, el informe también es generado pero además la interfaz se modifica, corrigiendo los errores detectados, por lo que el modelo de interfaz podría considerarse tanto entrada como salida del sistema.

La siguiente sección presentará los metamodelos GUI y Usability-GUI y a continuación dedicaremos dos secciones a describir en detalle los dos componentes de la solución.

7 Metamodelos de interfaz: Definición y transformaciones

En este capítulo se expondrán los diferentes metamodelos utilizados para la representación de una interfaz de usuario, analizando su estructura y su función dentro del sistema y estudiando las transformaciones que nos permitirán pasar de una representación a otra.

7.1 Metamodelo GUI

El punto de partida de todo sistema encargado de evaluar la usabilidad de una interfaz ha de ser la propia interfaz. La forma más sencilla de trabajar sobre una interfaz es tratar con el fichero fuente, el que define la interfaz en su formato original. No obstante, a menudo la representación de una interfaz está repleta de entidades de todo tipo (*layouts*, eventos, relaciones entre elementos, numerosos tipos de widgets y propiedades, etc.). Navegar en este conjunto de entidades interconectadas puede ser complejo y tedioso, más aún si lo que interesa se reduce a un conjunto de *widgets* y propiedades clave, siendo una gran parte del resto de la información irrelevante.

Por otro lado, tratar con la interfaz original tiene una desventaja adicional, y es que tu sistema sólo funcionará para ese tipo de interfaz. Aunque la mayoría de interfaces definen una serie de elementos y propiedades comunes, cada una tiene sus particularidades, su manera de representarlos e incluso nombres diferentes, con lo que el código programado específicamente para una interfaz Qt, por ejemplo, no serviría para el análisis de una interfaz web definida en HTML.

No obstante, podemos aprovechar la similitud entre los diferentes tipos de interfaz para definir una representación genérica que se adapte a la mayoría de ellas y nos permita realizar un análisis de la usabilidad independiente de la plataforma. Aquí es donde entra en juego el metamodelo GUI que usaremos para representar de forma genérica una interfaz gráfica de usuario.

Este metamodelo incluye los elementos más comunes en una GUI, de forma que un modelo GUI con la información que nos interesa pueda ser obtenido a partir de GUIs de cualquier plataforma. Un modelo GUI que conforme a este metamodelo representará una simplificación de la interfaz original, ya que el metamodelo GUI prescinde de muchos

7. Metamodelos de interfaz: Definición y transformaciones

conceptos y entidades que algunos tipos de interfaz contemplan pero que no son de interés para nuestro sistema.

Con los modelos GUI genéricos, solventamos los dos problemas que comentábamos al principio. Por un lado, conseguimos un modelo simplificado de una interfaz, que contendrá sólo la información relevante para nuestros propósitos, prescindiendo del resto. Por otro lado, tenemos una representación genérica de una interfaz que nos permite realizar un análisis de la usabilidad independiente de la plataforma.

El metamodelo GUI diseñado puede verse al completo en la Figura 7.1. Cualquiera que esté familiarizado con los conceptos básicos de GUIs observará que el metamodelo incluye los elementos principales de una GUI típica. El metamodelo se define en torno a dos conceptos principales: **Component** y **Property**. Un **Component** representa a los elementos o componentes que forman una ventana y que pueden ser simples (**Widget**) o compuestos (**Container**). Todo componente agrega una serie de **Properties** que representan propiedades comunes a todos ellos entre las que destacan:

- **Dimension**: ancho y alto del componente.
- **Color**: colores frontal y de fondo, representados por su código hexadecimal.
- **Position**: posición horizontal y vertical del componente.
- **Font**: características relativas a la fuente de texto.
- **Tabfocus**: propiedad relativa al orden en que el componente adquiere el foco.

Además, cada componente tiene dos atributos de tipo texto: *tooltip* y una información de accesibilidad. El *tooltip* es un texto que se muestra al usuario cuando detiene el ratón sobre el elemento en cuestión. Normalmente, se informará al usuario de la función del elemento. La información de accesibilidad es una descripción del elemento que suele incluirse para que los usuarios con problemas de visión puedan acceder a ella a través de un lector de pantalla.

Container no es más que un componente que contiene otros componentes. Se han considerado tres tipos comunes de contenedores: **Panel**, **Window** y **Frame**.

Ciertas tecnologías de interfaz ofrecen numerosos tipos de contenedores, que no son más que variantes de estos tres tipos principales (por ejemplo, un panel con pestañas o *Tabbed Panel*). A efectos prácticos, todas estas variantes pueden ser agrupadas y clasificadas en base a los tres tipos de contenedores definidos anteriormente.

En cuanto a los widgets también hemos considerado los más comunes y que son usados en la mayoría de aplicaciones: **Button**, **CheckBox**, **ComboBox**, **Image**, **Input**, **Label**, **List**, **Menu** y **Radio Button**.

Los widgets que pueden o deben tener un texto asociado tienen un atributo **text** para registrarlo. El tamaño y la fuente de este texto, como ya hemos visto, es una propiedad que posee todo elemento independientemente de que tenga un texto asociado o no. Esto se hace así ya que un **Frame**, por ejemplo, puede no tener ningún texto ni título pero al tener una fuente definida, todos los elementos que se creen en su interior heredarán dicha

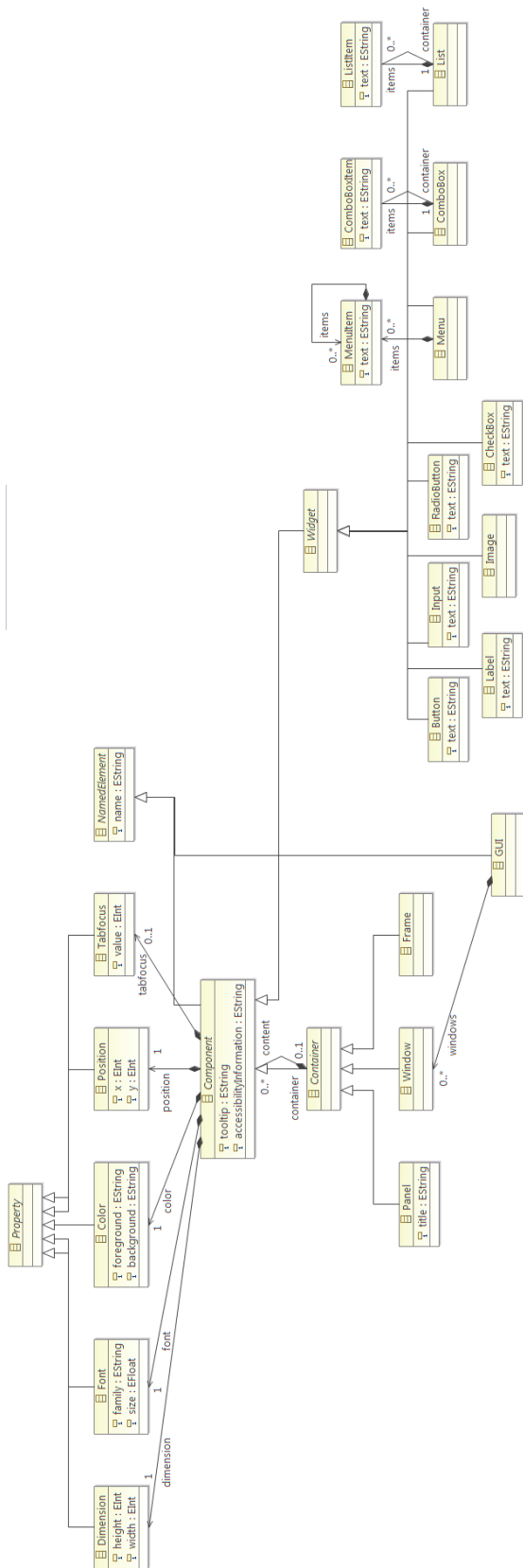


Figura 7.1.: Metamodelo GUI.

7. Metamodelos de interfaz: Definición y transformaciones

propiedad. La herencia de propiedades es algo muy común en casi todas las tecnologías que permiten la especificación de interfaces, e interviene en otras propiedades como el color.

Por otro lado, tanto `Menu` como `ComboBox` o `List` comparten una estructura similar, donde los ítems que contienen se definen como una sencilla entidad con un texto asociado y, en el caso de los menús, con una estructura anidada que permite que un ítem contenga, a su vez, un conjunto de ítems.

Nótese que el metamodelo no incluye determinados widgets (por ejemplo, un área de texto o tipos adicionales de botón). No obstante, consideramos que la lista contempla la mayoría de los elementos usados habitualmente. En cualquier caso, cualquier otro widget no tenido en cuenta aquí probablemente podría ser incluido en alguna de las clases definidas, al ser especializaciones o variaciones de alguno de los widgets básicos.

De nuevo, insistimos en que el fin último de este metamodelo es elaborar un esquema que permita representar de forma simplificada cualquier tipo de interfaz, y no era nuestra intención tener en cuenta todas las propiedades y componentes concebibles en una interfaz de usuario.

Finalmente, `GUI` es el elemento raíz del metamodelo y representa una interfaz completa que agrega el conjunto de ventanas que conforman la interfaz en cuestión y que contendrán, a su vez, contenedores y widgets.

Como ejemplo del modelo `GUI`, mostramos en la Figura 7.2 la equivalencia existente entre una sencilla interfaz definida en Qt y su representación conforme a nuestro metamodelo `GUI` en forma de un modelo de objetos UML. Recordemos que esta transformación es parte de la arquitectura pero no se ha implementado como parte de nuestro sistema actual.

7.2 Metamodelo de usabilidad

El metamodelo `GUI` definido representa una simplificación de la interfaz de usuario, pero mantiene la estructura y las entidades propias de un modelo de interfaz. A la hora de realizar un análisis de la usabilidad, puede resultar difícil analizar determinadas propiedades o características de un componente o de la interfaz en general, debido a la necesidad de navegar por la estructura jerárquica que forman los contenedores de una ventana. Por ejemplo, dada una regla de usabilidad relativa a la longitud o algún otro aspecto del texto de los elementos, deberemos buscar dentro de la interfaz todos aquellos elementos que tengan un texto y, cuando lo tengan, consultar las características del mismo dentro de la propiedad `Font`, de acuerdo al metamodelo `GUI` expuesto en el apartado anterior. En una búsqueda así deberían incluirse no solo widgets como un botón y una etiqueta sino que también deberían analizarse de forma individual cada uno de los ítems que componen, por ejemplo, un `ComboBox`, ya que tienen un texto asociado.

Este y otros casos similares son una muestra de las dificultades que uno puede encontrar cuando sigue un proceso de análisis de la usabilidad que en la mayoría de los casos está centrado en las propiedades de los elementos, cuando la representación de la interfaz

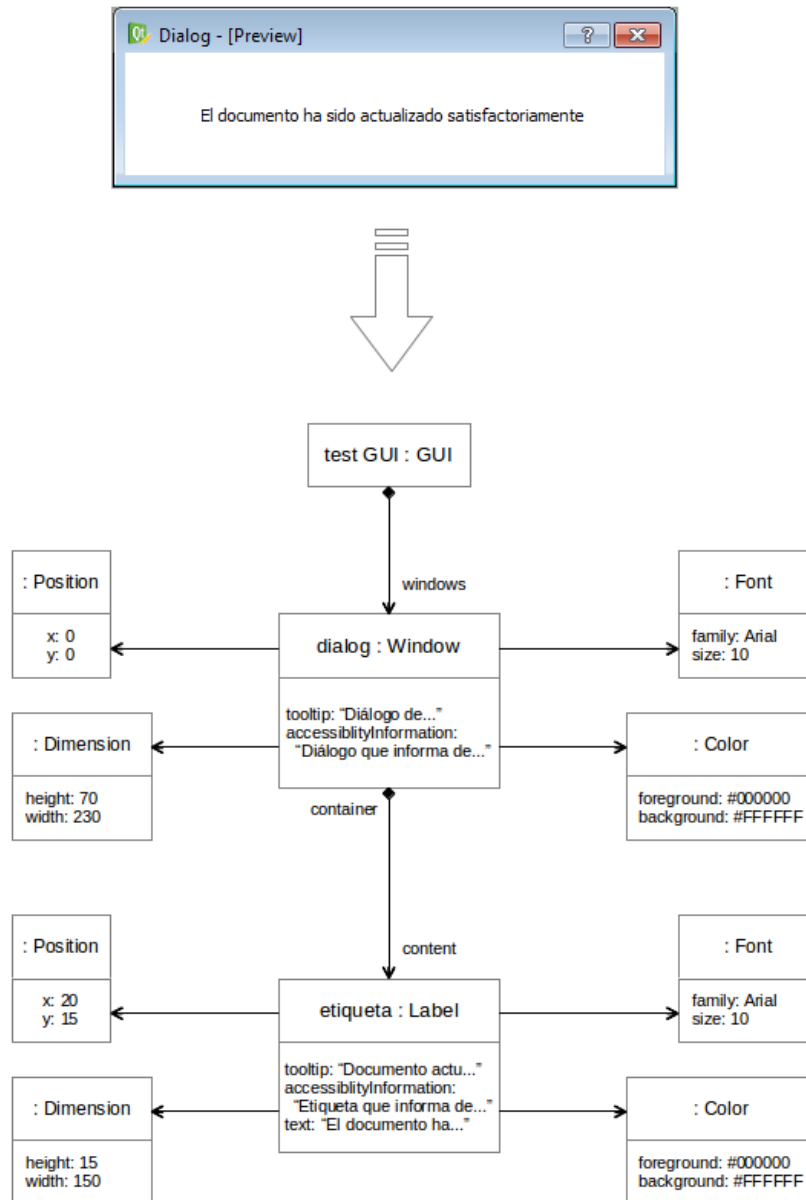


Figura 7.2.: Modelo GUI correspondiente a una interfaz definida en Qt.

7. Metamodelos de interfaz: Definición y transformaciones

viene dada por una estructura de componentes construida a través de contenedores y widgets contenidos en ellos.

Por tanto, para facilitar el proceso de análisis y aplicación de reglas de usabilidad se ha definido el metamodelo Usability-GUI cuyo propósito es reorganizar la representación jerárquica de una interfaz para ajustarla a nuestras necesidades, dando un mayor protagonismo a las propiedades de los elementos por encima de los elementos en sí. No obstante, la estructura elemental de contenedores y widgets debe mantenerse, pues es inherente a la interfaz y básica para su correcto recorrido. Muchas normas de usabilidad, por ejemplo, se aplicarán sobre los elementos situados en un mismo contenedor, ya sea un panel o una ventana, con lo que necesitamos conservar esta relación de pertenencia.

Así pues, el objetivo en este punto del sistema es diseñar un metamodelo que permita una representación de la interfaz orientada hacia las propiedades de los elementos pero que respete la estructura jerárquica que existe entre los componentes.

El metamodelo Usability-GUI que proponemos es mostrado en la Figura 7.3. Existen diferentes tipos de UsabilityElements siendo Role y Property los dos elementos principales.

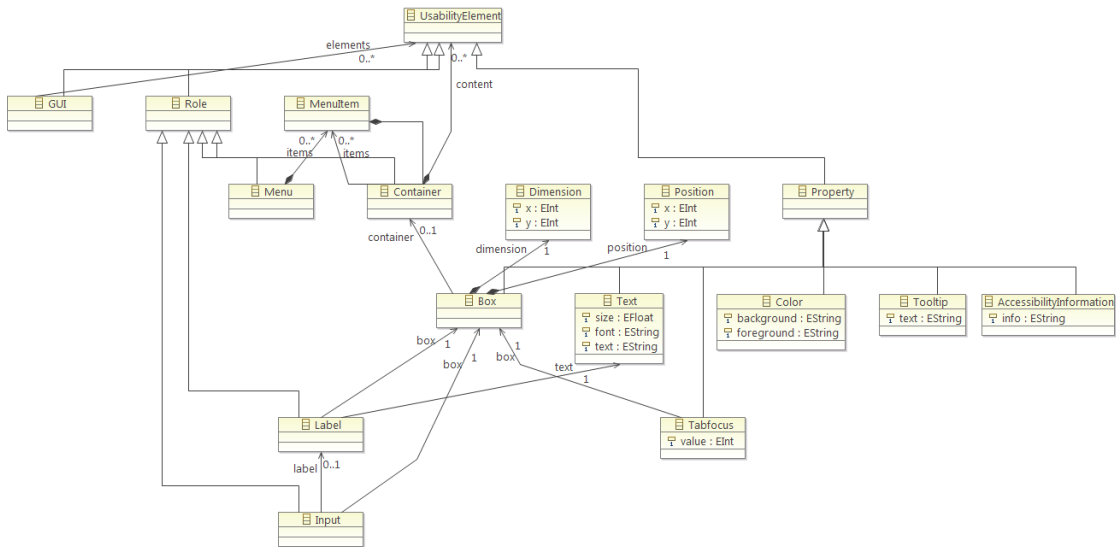


Figura 7.3.: Metamodelo de interfaz orientado a la usabilidad.

Una Property representa una propiedad de un componente y tenemos de diferentes tipos que corresponden en gran medida a las propiedades definidas en el metamodelo GUI. No obstante, se han realizado pequeños cambios para facilitar el análisis de usabilidad:

- Las entidades Dimension y Position han sido agrupadas en una única entidad llamada Box, que representa la caja o envoltente del componente. Muchas normas de usabilidad analizan la situación y los márgenes de los componentes de una interfaz, y para ello necesitan estudiar su posición y sus dimensiones. Este tipo

de reglas, de las que veremos algún que otro ejemplo más adelante, trabajarán únicamente con entidades de tipo `Box`.

- Las propiedades `Tooltip` y `AccessibilityInformation` son ahora entidades independientes, con un atributo de tipo texto que contiene la información correspondiente.
- Se ha añadido una entidad `Text` que representa un texto dentro de la interfaz de usuario, con su contenido y sus propiedades (tipo de fuente y tamaño del texto). Por cada texto definido en la interfaz, ya sea el contenido de una etiqueta, el título de un panel o uno de los ítems de una lista, el modelo tendrá una instancia de `Text`. Esto nos permitirá el análisis rápido y eficaz de todos los textos presentes en una interfaz determinada, simplificando una de las situaciones que comentábamos con anterioridad.

Utilizamos la clase `Role` para representar no propiedades sino roles de un componente dentro de una interfaz, es decir, papeles que desempeña un elemento y que es necesario conocer para un análisis efectivo de la usabilidad. Un rol será ligado a un determinado componente de usabilidad para indicar el papel que juega. Los tipos de rol definidos son: `Container`, `Menu`, `Label` e `Input`.

El rol `Container` nos permite mantener la estructura jerárquica de una interfaz. Para todo contenedor del metamodelo GUI (ya sea `Window`, `Panel` o `Frame`) existirá una entidad `Container` en el modelo. Este `Container` contendrá a su vez elementos `UsabilityElement` por cada uno de los widgets que contiene el contenedor de la GUI. Esto quiere decir que si un elemento `Button` viene representado por una entidad `Text`, otra de tipo `Color`, etc., su `Container` incluirá todas esas entidades.

Así, se mantiene la estructura jerárquica tan característica y necesaria en las interfaces de usuario. Este proceso de conversión y la forma de conservar las relaciones entre los elementos es algo que se verá con más detalle en el siguiente apartado.

`Menu` es un rol que se utiliza para identificar el componente que representa el menú de una ventana, y que es padre de la jerarquía de submenús e ítems. En este caso la estructura de un menú se conserva intacta con respecto a la representación que veíamos de este concepto en el metamodelo GUI. Con la distinción de un elemento como `Menu` se facilita la aplicación de reglas de usabilidad que incluyen alguna norma específica para los menús de una interfaz.

Por último, los roles `Label` e `Input` se han incluido dado que las etiquetas y campos de entrada son muy usados en las GUI, sobre todo en el caso de interfaces en las que prima la entrada de datos del usuario, y es necesario realizar comprobaciones de forma eficiente.

El correcto etiquetado de los campos de entrada es un principio fundamental de la usabilidad y numerosas reglas hacen referencia a estos elementos. En un modelo `Usability-GUI` cada elemento etiquetado con el rol `Input` tendrá asociado un elemento con el rol `Label` con el fin de facilitar la comprobación de ese tipo de reglas. Por ejemplo, la regla que establece la distancia correcta entre un campo de entrada y su etiqueta. Este ejemplo

7. Metamodelos de interfaz: Definición y transformaciones

de regla de usabilidad permite también ilustrar la razón de que un **Label** o **Input** tengan una referencia, a su vez, a un **Box**, pues la posición y dimensión de estos componentes son necesarios para calcular la distancia entre ellos.

Algo similar ocurre con **Tabfocus** o con la relación de un **Box** con su contenedor. Este tipo de referencias se mantienen porque son necesarias para el correcto desempeño de determinadas normas de usabilidad que han sido definidas.

En general, para la concepción de este metamodelo se ha partido en todo momento de las necesidades del proceso de análisis de la usabilidad en general y del conjunto de reglas contemplado en particular. Es decir, se ha diseñado de tal forma que la ejecución de las reglas de usabilidad expuestas en la Sección 5 (y cuya implementación explicaremos en la Sección 8) sea lo más sencilla posible.

Así, el metamodelo de usabilidad contiene única y exclusivamente la información necesaria para la aplicación de estas reglas, y prescinde del resto de características y datos sobrantes que pudiera contener el metamodelo GUI. La adición posterior de reglas de usabilidad diferentes podría motivar la actualización de este metamodelo, añadiendo entidades o relaciones de forma que se facilite su cálculo. No obstante, la estructura general de este metamodelo es la apropiada para la realización de un análisis de usabilidad. Sin embargo, cómo nuevas reglas de usabilidad que no hemos previsto en este proyecto encajen en este metamodelo es una validación pendiente que podría motivar algunos cambios en la estructura de las interfaces que es impuesta.

Cabe destacar que la creación de este metamodelo es un ejemplo de uso típico de las transformaciones M2M, esto es, la definición de modelos intermedios que permiten descomponer una transformación compleja (normalmente una M2T) en varios pasos intermedios más simples.

Un último detalle que no hemos comentado aún por no aparecer en el diagrama de la Figura 7.3 es la referencia existente entre los metamodelos GUI y Usability-GUI, concretamente entre **UsabilityElement** (metaclase de Usability-GUI) y **NamedElement** (metaclase del metamodelo GUI). Esta relación, que vemos representada en la Figura 7.4, es imprescindible para el correcto desempeño del análisis y la corrección de la usabilidad, y su importancia quedará clara conforme avancemos en la explicación de estos procesos.

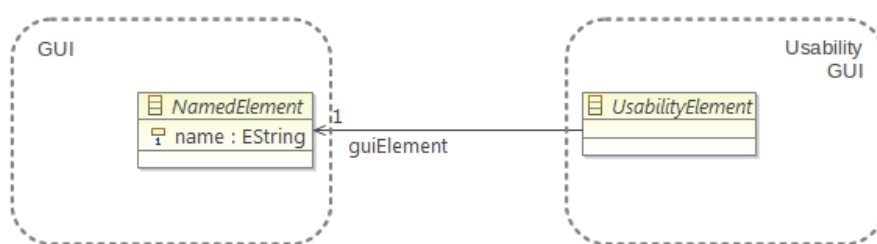


Figura 7.4.: Relación entre los metamodelos GUI y Usability-GUI.

Todo elemento dentro del metamodelo de usabilidad contiene una referencia a su elemento correspondiente dentro del metamodelo GUI. Esta trazabilidad nos será útil para acceder a información adicional del elemento original (por ejemplo, su nombre) que no

recoja el metamodelo de usabilidad, o para acceder directamente al elemento para su corrección.

La relación se establece entre las entidades `UsabilityElement` y `NamedElement`.

`UsabilityElement` es la clase principal del metamodelo de usabilidad y todas las entidades heredan de ella, de manera que al utilizar esta clase como origen de la referencia nos aseguramos de que todos los elementos del modelo deban tener dicha referencia.

Por otro lado, el hecho de establecer `NamedElement` como entidad destino de la referencia se debe a que el elemento raíz GUI también debe poder ser establecido como tal. Si limitásemos esta referencia al tipo `Component`, el elemento GUI del metamodelo de usabilidad no podría tener asociado su correspondiente elemento del metamodelo GUI, pues dicho elemento no es un componente (no se define como componente pues un elemento raíz GUI no tiene color, posición ni ninguna de las propiedades que caracterizan a un componente).

Así pues, para permitir que la referencia se establezca sobre un componente o sobre el elemento raíz GUI, definimos como tipo `NamedElement`, que engloba a ambos.

7.3 Del modelo GUI al modelo de usabilidad

Una vez definidos los metamodelos GUI y de usabilidad, estamos en condiciones de escribir la transformación M2M que generará un modelo de usabilidad a partir de un modelo GUI de entrada. Describiremos la correspondencia (*mapping*) entre elementos al mismo tiempo que expondremos el orden en que se ejecutan las reglas.

7.3.1 Inicio de la transformación

El punto de partida de esta transformación será la ejecución de la regla que establece el mapping entre el elemento raíz del metamodelo fuente (parte `transform` de la regla), en nuestro caso el elemento `GUI!GUI` y el elemento raíz del metamodelo de salida (parte `to` de la regla), en nuestro caso el elemento `UsabilityGUI!GUI`. Esta regla establece la relación entre el elemento `UsabilityGUI!GUI` creado y su elemento GUI origen y que los `UsabilityElement` que contendrá el elemento GUI generado se crearán a partir de las `windows` del elemento GUI fuente. El código de esta sencilla regla de transformación es el siguiente:

```
rule GUI
  transform gui : GUI!GUI
  to usability_gui : UsabilityGUI!GUI
  {
    usability_gui.guiElement = gui;
    usability_gui.elements ::= gui.windows;
  }

```

La ejecución de la segunda asignación disparará la ejecución de una regla que establezca un mapping entre `Window` (tipo de la expresión de la parte derecha de la asignación) y

7. Metamodelos de interfaz: Definición y transformaciones

`UsabilityElement` (tipo de la expresión de la parte izquierda). De la misma manera, la ejecución de esta regla provocará de forma recursiva el disparo del resto de reglas a partir de la ejecución de reglas que involucren a dos metaclases en vez de tipos simples. Los elementos de usabilidad generados con esta segunda asignación se agregarán al elemento GUI creado.

Esta primera regla es la única que se ejecutará directamente mientras que el resto de reglas son etiquetadas como `@lazy` (o bien son abstractas, como veremos a continuación).

7.3.2 Generación de Propiedades

Ya que los diferentes tipos de widgets tienen una gran parte de sus propiedades en común, lo ideal sería poder aprovechar esta situación para simplificar el código de la transformación. Para ello hacemos uso del mecanismo de herencia de reglas que ofrece el lenguaje ETL, definiendo una regla de transformación abstracta que determine la forma en que se realiza la conversión de un widget, y estableciendo después reglas adicionales que la extiendan añadiendo los atributos particulares de cada tipo de widget.

La regla que aglutina por tanto la mayor parte de la funcionalidad en la transformación de un componente cualquiera es la regla abstracta para `GUI!Component`, cuyo código es el siguiente:

```
@abstract
rule Component
  transform component : GUI!Component
  to color : UsabilityGUI!Color,
    tooltip : UsabilityGUI!Tooltip,
    accessibilityInfo : UsabilityGUI!AccessibilityInformation,
    box : UsabilityGUI!Box,
    tabfocus : UsabilityGUI!Tabfocus
  {
    color.background = component.color.background;
    color.foreground = component.color.foreground;
    color.guiElement = component;

    tooltip.text = component.tooltip;
    tooltip.guiElement = component;

    accessibilityInfo.info = component.accessibilityInformation;
    accessibilityInfo.guiElement = component;

    box.dimension ::= component.dimension;
    box.position ::= component.position;
    if (component.container.isDefined()) {
      box.container = component.container.equivalents()
        .selectOne(e|e.isInstanceOf(UsabilityGUI!Container));
    }
    box.guiElement = component;

    if (component.tabfocus.isDefined())
      tabfocus.value = component.tabfocus.value;
    else
      tabfocus.value = -1;
    tabfocus.box = box;
    tabfocus.guiElement = component;
  }
}
```

7.3. Del modelo GUI al modelo de usabilidad

Esta regla se ejecutará cuando se ejecute cualquier regla que herede de ella dado que es abstracta. Se puede observar que dado un `GUI!Component` se generarán cinco elementos que son de tipo `UsabilityGUI!Property` y que representan las propiedades: color, posición, dimensión, tooltip, información de accesibilidad y tabfocus.

Si observamos el código de la regla, podemos ver que la conversión es sencilla para la mayoría de las propiedades, pues los elementos origen y destino comparten los mismos atributos. Destacamos la asociación de un `Box` con su `Container`, que debe incluir una condición previa pues no todos los elementos tendrán siempre otro que los contenga (las ventanas que componen la interfaz no tienen un contenedor). Para ello hemos usado el método `equivalents` que nos proporciona ETL para consultar la traza de una transformación y que fue descrito en la Sección 4.3.1.2. Este método retorna los elementos que se han generado a partir del objeto sobre el que se aplica. En este caso se usa para obtener los elementos de usabilidad que se han generado para un contenedor de la GUI y la colección retornada se filtra para quedarnos sólo con aquellos que sean de tipo `UsabilityGUI!Container` (consultar en la Figura 7.4 la relación entre `Box` y `Container`).

Otro aspecto a destacar es que un elemento `Tabfocus` se genera siempre para todo componente GUI, independientemente de que esta propiedad haya sido realmente establecida en la interfaz original. Se toma esta decisión para simplificar la regla de transformación, y se establece un valor de 1 para identificar aquellos componentes que no tenían definido el tabfocus.

Por último, indicar que los mappings relativos a la dimensión y a la posición no son directos sino que disparan un par de reglas de transformación adicionales, cuyo código puede consultarse en el propio archivo de transformación dentro del proyecto pero que no se incluyen aquí por su simplicidad (la correspondencia entre las posiciones y dimensiones `x` e `y` es directa entre las entidades origen y destino).

Una vez definida la regla `Component`, gran parte del trabajo está hecho, pues la mayoría de propiedades y atributos de los componentes son genéricos y por tanto se contemplan en esta regla que acabamos de exponer. A partir de ella, se escribirán nuevas reglas que la extienden para cada uno de los contenedores y widgets.

Los contenedores son un ejemplo de la sencillez del código a partir de este punto. Se define de nuevo una regla abstracta para el tipo `GUI!Container` cuyo código es el siguiente:

```
@abstract
rule Container
  transform component : GUI!Container
  to color : UsabilityGUI!Color, tooltip : UsabilityGUI!Tooltip,
    accessibilityInfo : UsabilityGUI!AccessibilityInformation,
    box : UsabilityGUI!Box,
    tabfocus : UsabilityGUI!Tabfocus,
    container : UsabilityGUI!Container
  extends Component
  {
    container.guiElement = component;
    container.content ::= component.content;
  }
```

En la cabecera de la regla indicamos que hereda de la regla `Component`, y como destino

7. Metamodelos de interfaz: Definición y transformaciones

de la transformación (parte **to**) se ha añadido la metaclase `UsabilityGUI!Container`. En el cuerpo de la regla primero se asocia la referencia `container.guiElement` al contenedor de la GUI que es el elemento de entrada y en la segunda asignación se establece que el contenedor creado tendrá el mismo contenido (referencia `content`) que el contenedor origen en el modelo GUI. Esta segunda asignación disparará las reglas que convierten el contenido de un contenedor GUI en el contenido del correspondiente contenedor en el modelo `UsabilityGUI`.

Este proceso es común para todos los contenedores, y una vez definido la conversión de elementos concretos como `Window` o `Frame` es directa, de manera que su regla de transformación correspondiente, que hereda de la que acabamos de ver, queda vacía pues no es necesario añadir nada más.

En cuanto a la transformación de los widgets, de nuevo en la mayoría de los casos no será necesario añadir mucho más. En la mayoría de tipos de widgets, lo único que nos queda es la generación del elemento que representa al texto. Es el caso de `Button`, cuya regla de transformación podemos ver a continuación:

```
@lazy
rule Button
  transform component : GUI!Button
  to color : UsabilityGUI!Color, tooltip : UsabilityGUI!Tooltip,
    accessibilityInfo : UsabilityGUI!AccessibilityInformation,
    box : UsabilityGUI!Box,
    tabfocus : UsabilityGUI!Tabfocus,
    text : UsabilityGUI!Text
  extends Component
  {
    text.text = component.text;
    text.font = component.font.family;
    text.size = component.font.size;
    text.guiElement = component;
  }
```

Vemos que en la parte **to** se ha añadido la metaclase `Text` por lo que en el cuerpo de las reglas sólo debemos incluir asignaciones que establezcan qué valores tendrán los atributos y referencias de elementos `Text`.

Las reglas correspondientes a un `CheckBox` o a un `RadioButton` son exactamente iguales, pues ambas añaden únicamente un texto a las propiedades básicas de un componente.

Estos widgets son de los más sencillos, y nos pueden servir para entender mejor mediante un ejemplo el proceso que hemos explicado hasta ahora.

En la Figura 7.5 ilustramos con un esquema el proceso de conversión asociado a un botón, cuya regla de transformación acabamos de enseñar. En el modelo GUI, un botón es un elemento con varios atributos y con diversas propiedades asociadas. Para un widget como este, el proceso de transformación tendrá como resultado la generación de los elementos que vemos en la parte derecha del diagrama, correspondientes a las propiedades y atributos del elemento origen. Concretamente, se generarán las entidades `Color`, `Tabfocus`, `Tooltip`, `AccessibilityInformation` y `Box` (con sus correspondientes `Dimension` y `Position`), definidas en la regla abstracta `Component`, así como un elemento `Text` cuya creación se define en la regla de transformación definida para `Button` y que extiende a la regla `Component`. La equivalencia entre los atributos de estas entidades es directa, y se

7.3. Del modelo GUI al modelo de usabilidad

establecen entre ellos las relaciones necesarias, como la existente entre `Tabfocus` y `Box`.

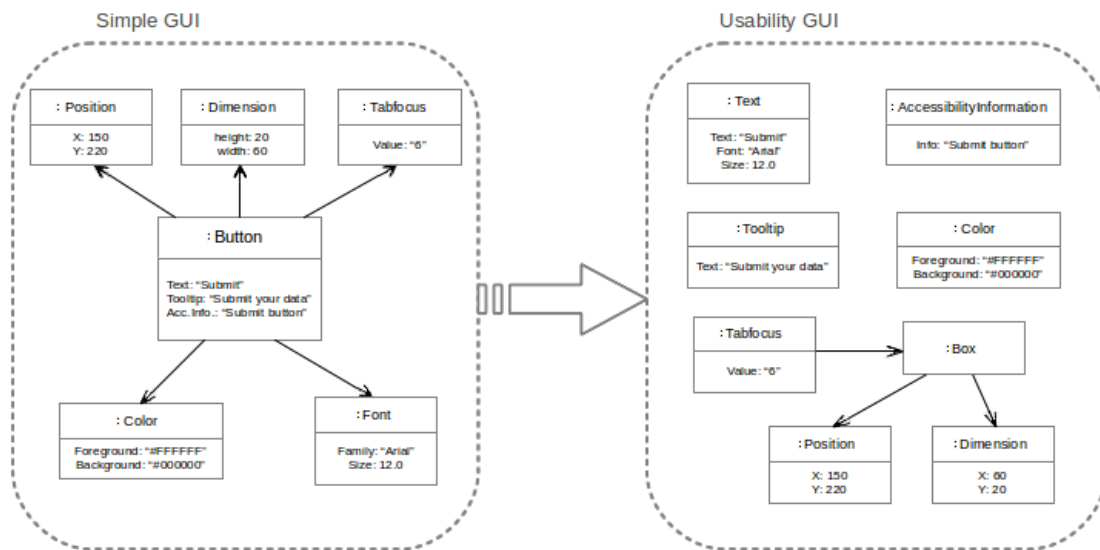


Figura 7.5.: Elementos del metamodelo Usability-GUI generados para un botón del metamodelo GUI.

Para este ejemplo se ha omitido por simplicidad la referencia al contenedor. Realmente, el botón tiene un elemento padre para el que se generarán las mismas entidades (salvo quizás `Text`) y una entidad adicional `Container` que englobaría todas las generadas para `Button`.

Una vez más, remitimos al lector a la Sección 10 de este documento, donde con un ejemplo completo puede observar mejor el comportamiento de este proceso, aunque con este sencillo caso pretendemos haber aclarado su funcionamiento.

7.3.3 Análisis de la separación entre etiquetas y campos de entrada

Otros widgets que no son botones, no obstante, tienen asociado un proceso de transformación más complejo que el que acabamos de ver. Es el caso de un elemento de tipo `Input`. La dificultad en este caso radica en la asociación `Input-Label`, que debe deducirse de la posición de los elementos dentro de la interfaz.

La referencia existente entre estos elementos fue expuesta en la Sección 7.2 y es importante para analizar la usabilidad de interfaces como formularios.

Por desgracia, es una información que en la mayoría de los casos no está disponible y tenemos que calcular por nuestra cuenta. Para averiguar qué etiqueta se corresponde con qué campo de entrada, estudiaremos la posición de ambos elementos. Si están lo suficientemente cerca, se asociarán. Este proceso que acabamos de resumir en unas pocas

7. Metamodelos de interfaz: Definición y transformaciones

palabras es en realidad algo más complejo. El código correspondiente se engloba dentro de la regla de transformación para Input y es el siguiente:

```
// Input-label association
// We search for the closest label, among the labels not assigned yet
// and close enough, on the top or on the left of the input
var selectedLabel =
  component.container.content.select(label|label.isTypeOf(GUI!Label)
    and not assignedLabels.includes(label)
    and
    ((component.position.y-(label.position.y+label.dimension.height)<=18
      and component.position.y-(label.position.y+label.dimension.height)>=-5
      and (component.position.x-label.position.x).abs()<=25)
    or
    ((component.position.y-label.position.y).abs()<=18
      and component.position.x-(label.position.x+label.dimension.width)<=100
      and component.position.x-(label.position.x+label.dimension.width)>=-10))
  ).sortBy(label|(component.position.x-(label.position.x+label.dimension.width)).abs()
    .min((component.position.y-(label.position.y+label.dimension.height)).abs())).first
    ();

if (selectedLabel <> null) {
  input.label =
    selectedLabel.equivalents().selectOne(e|e.isTypeOf(UsabilityGUI!Label));
  assignedLabels.add(selectedLabel);
}
```

Todo el proceso de búsqueda se engloba dentro de una única sentencia que explicamos a continuación. Se considera que una etiqueta está ligada a un campo de entrada si está situada encima de él o a su izquierda a una distancia menor que cierto valor umbral, como se muestra en la Figura 7.6. Las zonas marcadas con línea discontinua representan las áreas donde se considerará que una etiqueta puede estar asociada al campo de entrada. Se ha tratado de definir un margen lo suficientemente amplio para reconocer correctamente interfaces amplias con separaciones grandes entre sus elementos, pero no tanto como para establecer asociaciones entre elementos que en realidad sean inconexos.

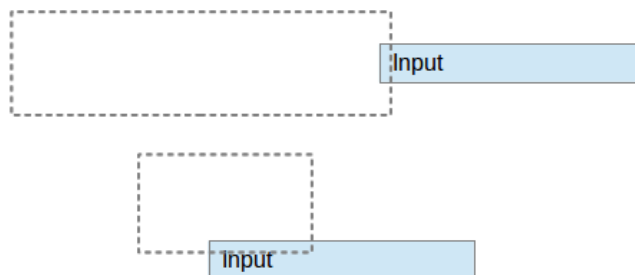


Figura 7.6.: Espacio de búsqueda de la etiqueta asociada a un campo de entrada.

Una vez localizadas todas las etiquetas candidatas a ser asociadas con el campo de entrada, las ordenaremos según su mínima distancia con el campo de entrada (distancia con la parte superior o izquierda). La etiqueta que más cerca se encuentre quedará asociada al campo de entrada. Si no se encontró ninguna etiqueta dentro del rango definido,

entonces el campo de entrada quedará sin relacionar (como veremos más adelante, este es uno de los errores de usabilidad contemplados).

Para evitar que una etiqueta sea asignada a más de un campo de entrada diferente, algo que podría suceder en interfaces donde la separación entre elementos es pequeña, utilizamos una estructura de datos adicional: una colección donde iremos almacenando las etiquetas que ya han sido asociadas a algún campo de entrada. Esta colección se declara dentro de un bloque de código `pre` que se ejecutará antes de iniciarse la transformación.

7.3.4 Conversión de elementos ComboBox y Listas

La conversión de los widgets `ComboBox` y `List` y sus correspondientes `ComboBoxItem` y `ListItem` tiene algunas peculiaridades.

Estos elementos son widgets pero al mismo tiempo contienen otros elementos, en este caso ítems, de manera que guardan cierta similitud con un contenedor. Es por esto que a la hora de representarlos en el modelo de usabilidad, para conservar esta relación de pertenencia entre un `comboBox` o lista y sus ítems, generamos para el primero un elemento `Container` que abarque las entidades generadas para sus ítems.

Las reglas de transformación correspondientes en el caso de un `ComboBox` son las siguientes:

```
@lazy
rule ComboBox
  transform component : GUI!ComboBox
  to color : UsabilityGUI!Color,
    tooltip : UsabilityGUI!Tooltip,
    accessibilityInfo : UsabilityGUI!AccessibilityInformation,
    box : UsabilityGUI!Box,
    tabfocus : UsabilityGUI!Tabfocus,
    container : UsabilityGUI!Container
  extends Component
  {
    container.guiElement = component;
    container.content ::= component.items;
  }
@lazy
rule ComboBoxItem
  transform component : GUI!ComboBoxItem
  to text : UsabilityGUI!Text
  {
    text.text = component.text;
    text.font = component.container.font.family;
    text.size = component.container.font.size;
    text.guiElement = component;
  }
```

En la regla `ComboBox` se asocia el `comboBox` de entrada al `guiElement` del contenedor generado y se agregan los ítems del `comboBox` al contenido (`container.content`) del contenedor generado. Cada ítem está representado únicamente por un elemento de tipo `Text`, ya que los ítems de un `comboBox` no son componentes por sí mismos, sino que su única característica propia es el texto que contienen. De esta forma, aun no siendo

7. Metamodelos de interfaz: Definición y transformaciones

contenedores generamos elementos de tipo **Container** para estos widgets por ser ésta la estructura que mejor se adapta a la situación. Las entidades de tipo **Text** que se generan para los ítems estarán contenidas en el elemento **Container**, tal y como vemos en la Figura 7.7.

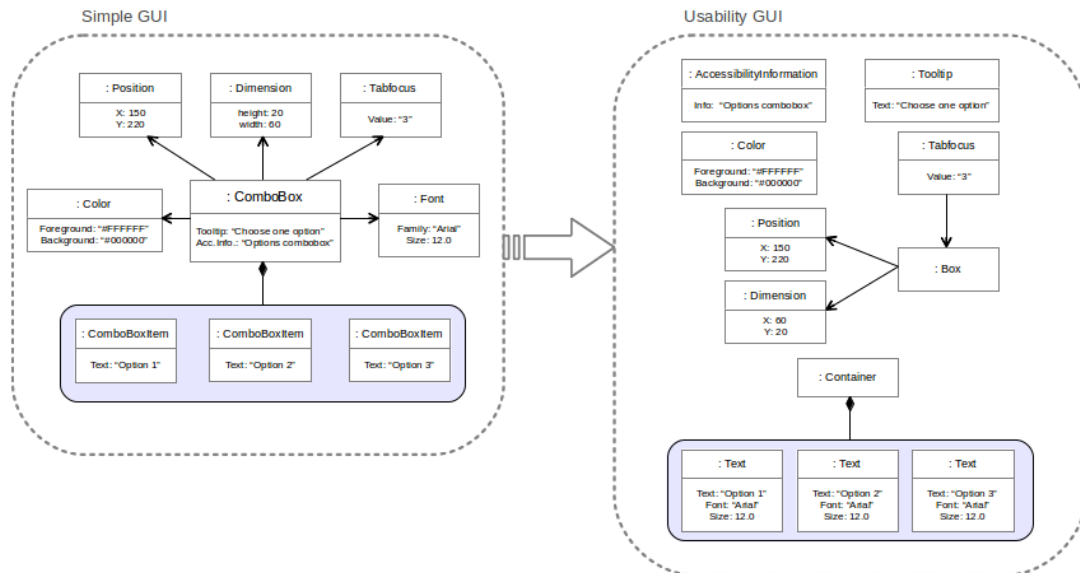


Figura 7.7.: Elementos del metamodelo Usability-GUI generados para un comboBox del metamodelo GUI.

8 El DSL UsabilityRules

Hemos definido un lenguaje específico del dominio textual para expresar las reglas de usabilidad por las razones expuestas en la Sección 6. Como indicamos en la Sección 4.2, un DSL está formado por tres elementos: el metamodelo que define su sintaxis abstracta, la sintaxis concreta y la semántica. Nosotros primero hemos definido el metamodelo y entonces hemos definido la notación textual utilizando la herramienta EMFText introducida en la Sección 4.3.2. La semántica de las reglas ha sido expresada por medio de código EOL que es interpretado en las transformaciones modelo a modelo creadas para la detección de defectos y la corrección.

8.1 Metamodelo UsabilityRules

En su definición hemos seguido un proceso de abstracción a partir del propio catálogo de reglas que hemos seleccionado para este proyecto y que fue presentado en la Sección 5. A partir de ellas hemos tratado de deducir una estructura que se adapte bien, agrupando las normas que tienen determinadas características en común pero permitiendo al mismo tiempo la extensibilidad. Con esto se ha pretendido dotar al catálogo de reglas de cierto orden y claridad, distinguiendo unas reglas de otras en base a alguna de sus propiedades, pero manteniendo la flexibilidad para que el usuario pueda definir cualquier tipo de comportamiento tanto en el análisis como en la corrección. El metamodelo se muestra en la Figura 8.1. Como se observa no contiene muchos conceptos.

Un catálogo (metaclase `Catalogue`) está formado por un conjunto de reglas (metaclase `Rule`). Cada regla tiene como propiedades básicas un nombre (`name`), una importancia (`importance`) que se representa como un valor un enumerado con tres posibles categorías, un tipo de elemento (`elementType`) que registra el tipo de elemento sobre el que se aplica, una descripción (`description`) que referencia a un elemento `Description` que contiene la definición y el mensaje que se mostrará cuando se detecte un error de usabilidad mediante dicha regla, y además está ligado a tres elementos `EOLCode` (`correction`, `calculation` y `restriction`) que especifican el comportamiento de la regla. Un `EOLCode` contiene un fragmento de código EOL directamente ejecutable.

Una regla de usabilidad se aplicará siempre sobre un tipo de elemento GUI. Esto es lo que se indica en el atributo `elementType`. Las reglas de usabilidad podrán aplicarse sobre widgets o propiedades concretas, como un `Label` o un `Text`, sobre un `Contai-`

8. El DSL UsabilityRules

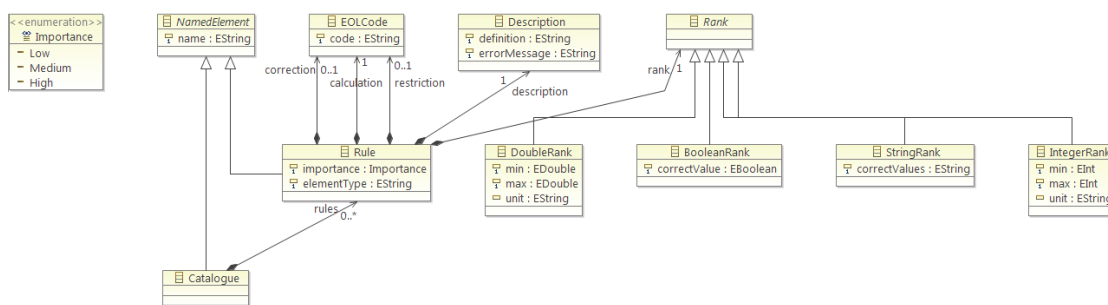


Figura 8.1.: Metamodelo Usability Rules.

ner cualquiera o incluso sobre la interfaz completa, en cuyo caso se indicará GUI como elementType.

El comportamiento de la regla viene determinado por los tres elementos EOLCode asociados a una regla:

- **Restriction:** es una restricción que sirve para reducir el conjunto de elementos (de tipo elementType) sobre el que se aplicará la regla de usabilidad.
- **Calculation:** es el bloque de código que analiza el elemento de entrada para comprobar si cumple o no la regla de usabilidad en cuestión.
- **Correction:** incluye las sentencias que modifican el modelo GUI original para corregir el defecto de usabilidad.

Así pues, a la hora de definir una regla de usabilidad es necesario escribir manualmente el código que determina cómo se realiza el cálculo y, en caso de ser necesario, cómo se realiza la corrección o qué restricciones se hacen sobre los elementos de entrada. Este código permitirá crear una función que contendrá el código definido en cada uno de estos elementos EOLCode, y que guiará los procesos de análisis y corrección.

El experto que crea una regla de usabilidad deberá tener en cuenta que:

- El elemento a analizar estará disponible en una variable de nombre e.
- El bloque de sentencias definido debe terminar con el retorno de un valor adecuado, tal que (los siguientes puntos se resumen en la Tabla 8.1):
 - La restricción deberá ser un bloque de código que retorne un valor true o false, entendiendo que aquellos elementos para los que la función de restricción devuelva true serán analizados por la regla.
 - El código de corrección únicamente modificará el modelo de interfaz y por tanto no será necesario que retorne valor alguno.
 - Para el proceso de cálculo se define una nueva entidad Rank que determina, entre otras cosas, el valor de retorno de la función correspondiente y que

EOLCode	Tipo de retorno
Restriction	Boolean
Calculation	Dependiente del Rank
Correction	-

Tabla 8.1.: Tipos de retorno de los EOLCode.

tiene una gran importancia en nuestro metamodelo de usabilidad. El concepto de rango nos permite realizar una distinción a nivel funcional de las reglas de usabilidad y facilita en parte el proceso de cálculo, como veremos en apartados posteriores. Distinguimos, por el momento, cuatro tipos de rango: `IntegerRank`, `DoubleRank`, `StringRank` y `BooleanRank`.

Los rangos no sólo determinan el valor que devolverá la función de cálculo sino que definen el intervalo de valores correctos, es decir, aquellos valores para los que se considera que el elemento analizado cumple la regla de usabilidad.

En el caso de los rangos de tipo numérico, se deben establecer los valores mínimo y máximo, así como una unidad (opcional). En el caso de los rangos `StringRank` y `BooleanRank`, se define el valor o conjunto de valores correctos mediante un String.

Esto quiere decir que si una regla de usabilidad tiene asociado, por ejemplo, un rango de tipo `IntegerRank`, su función de cálculo debe retornar un entero. Dicho valor entero se comparará con el rango de valores definido y, si se encuentra fuera, se detectará un defecto de usabilidad.

Así pues, el concepto de rango es fundamental en la definición de una regla de usabilidad pues guía la forma en que se realizará su aplicación sobre los elementos de una interfaz.

A la hora de definir el rango de valores correctos para una regla de usabilidad, es importante tener en cuenta que:

- El rango definido por los valores mínimo y máximo tanto en `IntegerRank` como en `DoubleRank` es un intervalo cerrado que incluye ambos extremos como valores correctos.
- Los valores correctos para un rango de tipo `StringRank` se indicarán como cadenas encerradas entre comillas simples y separadas por comas. En caso de que el valor correcto sea únicamente la cadena vacía, se deberá especificar explícitamente una cadena vacía encerrada entre comillas simples.

Como veremos más adelante, este metamodelo con los cuatro tipos de rango definidos son suficientes para representar de forma adecuada todas las reglas de usabilidad que hemos contemplado en nuestro catálogo. En la Sección 8.1.2 dentro de este capítulo veremos algunos ejemplos de definición de reglas, y en el Anexo B (además de en los

8. El DSL UsabilityRules

ficheros fuente) puede consultarse la implementación completa del catálogo. El lector podrá darse cuenta de cómo, bien entendida, la estructura definida es lo suficientemente flexible para permitir la creación de reglas de usabilidad de todo tipo, con comportamientos diferentes orientados a distintos tipos de propiedades y con un número variable de elementos involucrados.

En definitiva, ante el problema de cómo representar un catálogo amplio de reglas de usabilidad nos enfrentamos a una situación complicada, pues es un concepto difícil de abstraer con eficacia. Una forma de abordarlo podría haber sido la definición de una estructura parametrizable, con un metamodelo que sólo contemplase unos tipos muy concretos de normas de usabilidad permitiendo al usuario especificar, por ejemplo, el elemento sobre el que aplicarla y los valores correctos. Pero, además de la dificultad de dar con este tipo de reglas genéricas, esta solución adolece de ser demasiado rígida, pues no permitiríamos la definición de reglas que se apartasen de los tipos concretos contemplados en el metamodelo.

Dado que las reglas de usabilidad son muy variadas y dispares, es difícil dar con una solución que facilite su creación y al mismo tiempo soporte esta necesaria flexibilidad. Es por esto que no se pueden definir tipos más concretos de reglas o patrones parametrizables que faciliten la confección de un catálogo. El abanico es tan amplio y las reglas pueden llegar a ser tan complejas que la mejor solución es proporcionar un metamodelo que, como este, deje el comportamiento de la regla casi completamente en las manos de su creador, pero estructure los diferentes aspectos y bloques de código de la manera más clara y organizada posible, añadiendo eso sí ciertos conceptos como el rango que faciliten en la medida de lo posible la definición de las reglas de usabilidad.

8.1.1 Notación textual

Durante las primeras etapas de diseño de nuestra solución hemos utilizado el editor de modelos Ecore que proporciona Eclipse/EMF para crear modelos conforme al metamodelo UsabilityRules, el cual permite la rápida creación de modelos a partir de un metamodelo, ofreciendo un diálogo para establecer los valores de las diferentes propiedades y atributos. Realmente, el diseño de un modelo mediante el uso de este editor es, en general, cómodo, pero en nuestro caso particular presentaba algunas dificultades. Fundamentalmente, el hecho de que el editor no está preparado para atributos que pueden adoptar como valor una cadena multilínea. El cuadro de edición de propiedades sólo admite cadenas simples y no acepta saltos de línea, lo que hace imposible establecer directamente bloques formados por varias sentencias de código.

Este problema, unido al hecho de que el usuario o experto en usabilidad no debería por qué conocer el proceso de creación de un modelo ni lidiar con él, nos condujo a la necesidad de definir una notación específica para expresar textualmente las reglas de usabilidad e inyectar los modelos a partir de dicho texto. Hemos usado EMFText, brevemente introducido en la Sección 4.3.2, para definir la sintaxis textual. Con EMFText, la gramática del lenguaje se especifica mediante un conjunto de reglas sintácticas que anotan los elementos del metamodelo. A continuación se muestra el conjunto de reglas

para nuestro sencillo lenguaje de reglas.

```

RULES {
  Evaluator ::= "evaluator" "{" "name" ":" name["'", "'"] (rules)* "}";
  Rule ::= "rule" "{" "name" ":" name["'", "'"] "description" "{" description "}" "
    importance" ":" importance[Low:"Low", Medium:"Medium", High:"High"] "elementType"
    ":" elementType["'", "'"] ("restriction" ":" restriction)? "calculation" ":"
    calculation "rank" "{" rank "}" ("correction" ":" correction)? "}";
  EOCode ::= code["'", "'"];
  Description ::= "definition" ":" definition["'", "'"] "errorMessage" ":" errorMessage
    ["'", "'"];
  DoubleRank ::= "type" ":" "double" "min" ":" min[FLOAT] "max" ":" max[FLOAT] ("unit"
    ":" unit["'", "'"])?;
  BooleanRank ::= "type" ":" "boolean" "correctValue" ":" correctValue[];
  StringRank ::= "type" ":" "string" "correctValues" ":" correctValues["'", "'"];
  IntegerRank ::= "type" ":" "integer" "min" ":" min[INTEGER] "max" ":" max[INTEGER] ("
    unit" ":" unit["'", "'"])?;
}

```

Por lo general hemos tratado de seguir unas pautas de diseño tal que:

- Los bloques de código y los atributos se representan con su nombre seguido de dos puntos seguido de su valor, encerrando las cadenas entre comillas dobles.
- Las entidades compuestas por otras entidades o con varios atributos se representan por su nombre seguido de dos corchetes entre los que se encierra todo su contenido.

A continuación mostramos un ejemplo de definición de reglas para ilustrar la gramática del lenguaje.

En los ficheros fuente del proyecto, se puede consultar el catálogo completo de reglas de usabilidad definido en un archivo “evaluator.rules” que sigue la notación descrita por este DSL textual. Aquí mostramos una de las reglas que aparecen, relativa al atributo tooltip de un elemento:

```

rule {
  name: "Tooltip"
  description {
    definition: "Checks if tooltip text has been set for an element"
    errorMessage: "Tooltip text has not been set"
  }
  importance: Low
  elementType: "Tooltip"
  calculation: "return e.text.length() > 0;"
  rank {
    type: boolean
    correctValue: true
  }
}

```

Como vemos, los atributos como `name` o `elementType` y los bloques de código aparecen representados por su nombre y su valor separados por dos puntos.

En el caso de entidades como la descripción, el rango o la propia regla de usabilidad, se hace uso de los corchetes para delimitar el principio y fin de su definición.

En este sentido, la notación empleada no supone ninguna revolución y hace uso de una sintaxis común en los lenguajes de programación, con la que es probable que el usuario esté familiarizado.

8. El DSL UsabilityRules

Es importante darse cuenta de que, al estar los bloques de código delimitados por dobles comillas, no es posible utilizar este carácter dentro del propio código; el usuario debe tener la precaución de usar comillas simples en su lugar. No obstante, en caso de que el usuario cometa un error de este tipo no tardaría en darse cuenta ya que, al utilizar comillas dobles dentro del código, el editor lo interpretará como el fin del mismo e informará de un error de sintaxis en el documento (ver Figura 8.2).

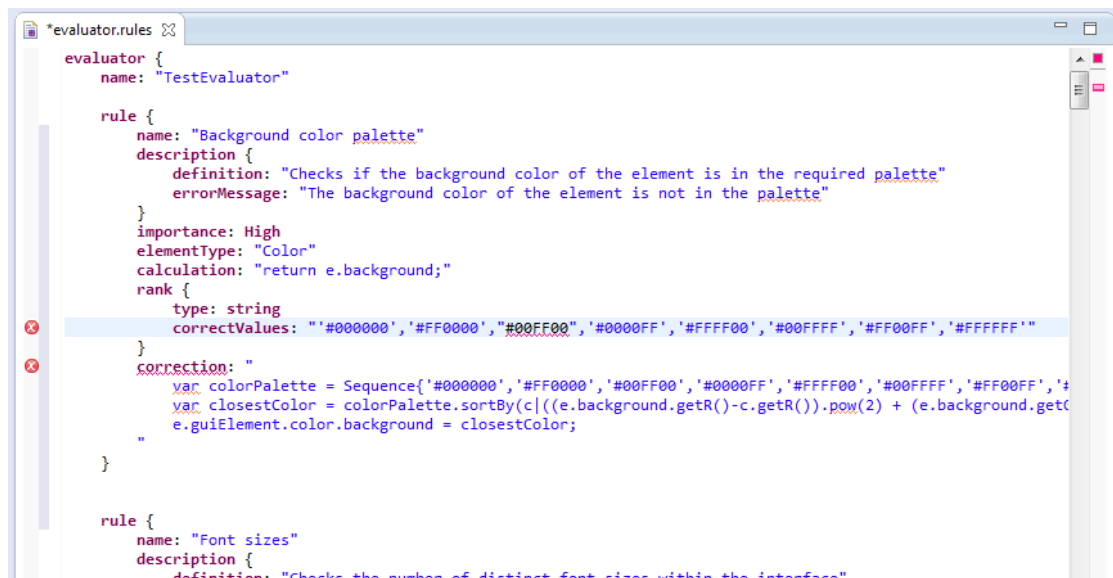


Figura 8.2.: Metamodelo Usability Rules.

Esta es una de las utilidades de la definición de un DSL textual con una herramienta como EMFText, a partir del metamodelo y la gramática se genera un editor del lenguaje y un inyector de modelos que convierte el texto escrito con el editor en un modelo (serializado en formato XMI) conforme al metamodelo. El editor generado proporciona algunas de las utilidades comunes de los editores de lenguajes de programación como el resaltado de sintaxis, comprobación de sintaxis, etc. facilitando la creación de un catálogo de reglas de usabilidad.

8.1.2 Ejemplos de definición de reglas

Mostraremos tres ejemplos de reglas definidas con el DSL UsabilityRules para ilustrar su uso. Los ejemplos han sido elegidos para presentar todas las posibilidades del lenguaje desde el punto de vista de la notación y de su semántica, y se explicará el comportamiento definido en los bloques de código. En el Anexo B se puede consultar la definición del catálogo completo de reglas.

Regla 'Tipos de fuente'

```
rule {
  name: "Font types"
```

```

description {
    definition: "Checks the number of distinct font types used in the interface"
    errorMessage: "Too many distinct font types"
}
importance: High
elementType: "GUI"
calculation: "return Text.allInstances().collect(t|t.font).asSet().size();"
rank {
    type: integer
    min: 1
    max: 2
    unit: "font types"
}
correction: "
    var maxTypes = 2;
    var validTypes : Set;
    for (t in Text.allInstances()) {
        if (validTypes.size() < maxTypes) {
            // We collect valid types until reaching the limit
            validTypes.add(t.font);
        }
        else if (not validTypes.includes(t.font)) {
            // We get the first font type
            var newType = validTypes.first();
            // We set the new size
            t.guiElement.font.family = newType;
        }
    }
"
}

```

Este primer ejemplo es una regla de usabilidad que mide el número de fuentes distintas de texto que se utilizan en toda la interfaz. Es una norma que busca la uniformidad dentro de una interfaz de usuario, no permitiendo más de dos tipos de fuente diferentes ya que esto tendría como resultado una apariencia más caótica y menos legible.

Se especifica que GUI es el elemento sobre el que se aplica la norma. Como ya explicamos anteriormente, cuando se quiere medir una característica de usabilidad propia de la interfaz en general se utiliza este elemento ya que es el elemento raíz del metamodelo.

La regla de usabilidad no define ninguna restricción, algo que por otro lado no tendría mucho sentido siendo una regla que se aplica sobre toda la interfaz y no sobre un grupo concreto de elementos.

El cálculo se reduce a una única sentencia de código en la que recorreremos todas las entidades de tipo `Text` almacenando sus tipos de fuente en una colección sin repetición, devolviendo finalmente su tamaño. Es un ejemplo de la potencia del lenguaje EOL y concretamente de los iteradores sobre conjuntos como `select` o `collect`.

Dado que la regla de usabilidad se refiere al número de fuentes utilizadas, es evidente que estamos ante un rango de tipo numérico. La sentencia de cálculo nos devolverá un entero que más tarde se cotejará con el rango numérico que aquí definimos. Permitimos uno o dos tipos de fuente diferentes, añadiendo el nombre de la unidad para mayor claridad.

Con esto ya se ha definido toda la información básica de la regla de usabilidad y el comportamiento necesario para el proceso de análisis.

8. El DSL UsabilityRules

Sólo nos queda el código opcional de corrección, que como vemos es ligeramente más complejo que el que teníamos para el cálculo.

El código está debidamente comentado y es fácil, aunque uno no esté familiarizado con el lenguaje EOL, entender lo que se está haciendo. Básicamente, recorreremos todos los textos definidos en la interfaz almacenando su tipo de fuente en una colección. Cuando alcanzamos el límite de tipografías diferentes, en nuestro caso dos, todos los textos analizados a partir de ese punto cuya tipografía no haya sido ya contemplada serán considerados defectuosos. A la hora de corregirlos, tomamos la decisión de sustituir su tipo de fuente por el primero encontrado. Nótese el uso de la referencia existente entre el modelo de usabilidad y el modelo gui, que nos permite acceder al elemento original de la interfaz (atributo `guiElement`) y modificarlo directamente.

Un proceso de corrección más complejo podría analizar cuál es la tipografía más utilizada hasta el momento y usarla para reemplazar los elementos defectuosos, o tener en cuenta las clases de fuente (serif o sans-serif) en este proceso de selección.

Regla 'Paleta de colores de fondo'

```
rule {
  name: "Background color palette"
  description {
    definition: "Checks if the background color of the element is in the required
      palette"
    errorMessage: "The background color of the element is not in the palette"
  }
  importance: High
  elementType: "Color"
  calculation: "return e.background;"
  rank {
    type: string
    correctValues:
      "'#000000', '#FF0000', '#00FF00', '#0000FF', '#FFFF00', '#00FFFF', '#FF00FF', '#
      FFFFFFF', 'E1E1E1'"
  }
  correction: ""
    var colorPalette = Sequence{'#000000', '#FF0000', '#00FF00', '#0000FF', '#FFFF00
    ', '#00FFFF', '#FF00FF', '#FFFFFF', 'E1E1E1'};
    var closestColor = colorPalette.sortBy(c|((e.background.getR()-c.getR()).pow(2)
      + (e.background.getG()-c.getG()).pow(2) + (e.background.getB()-c.getB()).
      pow(2)).pow(0.5)).first();
    e.guiElement.color.background = closestColor;
  }
}
```

La segunda regla de usabilidad que vamos a ver no es relativa al texto sino al color, otra de las características fundamentales de un elemento e importante para la usabilidad.

La regla está destinada a comprobar si el color de fondo de un elemento se encuentra dentro de una paleta determinada de colores. El concepto de paleta de colores es común en temas de usabilidad de interfaces. Es habitual que a la hora de configurar una interfaz de usuario se defina una paleta determinada de colores de forma que todos los colores utilizados en la definición de la interfaz pertenezcan a ella. Con esto se reduce el total de colores que pueden usarse o se limita a un subconjunto concreto de tonalidades. En este sentido, suele ocurrir que una empresa cuenta con una serie de colores representativos de su marca y quiere que su software respete este código de colores. También puede ser

útil para temas de accesibilidad, limitando los colores utilizados a un subconjunto de colores básicos que pueden representarse en las pantallas de todos los dispositivos.

En cualquier caso, la comprobación de la paleta de colores es una regla común de la usabilidad y su implementación, como veremos a continuación, puede realizarse de forma sencilla.

La norma se aplicará sobre los elementos de tipo `Color`. Una vez más, no contamos con ninguna restricción pues no hay excepciones en la aplicación de esta regla.

El cálculo es muy sencillo: simplemente obtenemos el color de fondo del elemento en su representación hexadecimal. Nótese la disponibilidad del elemento en cuestión bajo la variable de nombre `e`.

La regla define por tanto un rango de tipo `StringRank`, donde el valor calculado es el color de fondo y el rango de valores correctos es la paleta de colores.

Como ya hemos mencionado en anteriores ocasiones, debemos definir los valores posibles encerrados entre comillas simples. Para esta regla hemos definido una sencilla paleta formada por los colores básicos. Durante el futuro proceso de análisis se comprobará que el valor devuelto por la sentencia de cálculo se encuentra dentro del rango definido, detectando un error de usabilidad en caso contrario.

Esta regla es un ejemplo de la sencillez con que pueden definirse algunas normas de usabilidad haciendo uso del metamodelo de reglas definido. El mecanismo de rangos se encarga de casi todo el proceso, mediante la comprobación automática que se realiza durante el proceso de análisis y que veremos más adelante.

En cuanto a la corrección, se realiza en un par de sentencias que buscan el color dentro de la paleta más cercano al del elemento para su sustitución. El cálculo de las ‘distancias’ entre colores se realiza en base a sus valores RGB. Para ello, se cuenta con métodos de consulta que, aplicados sobre un `String` con la representación hexadecimal del color, devuelven el valor R, G ó B según corresponda. Estos métodos auxiliares son definidos durante el proceso de análisis dentro del fichero de transformación M2M en el que se incluyen las sentencias de cálculo y corrección, y es por ello que están disponibles y pueden ser utilizados para facilitar determinados procesos y el acceso a valores comunes relacionados con la usabilidad. Su definición será estudiada en la Sección 9.2.

Regla ‘Solapamiento’

```
rule {
  name: "Overlap"
  description {
    definition: "Checks if the element overlaps other elements in its container"
    errorMessage: "This element overlaps other elements in its container"
  }
  importance: High
  elementType: "Box"
  restriction: "return e.container.isDefined();"
  calculation: "return e.container.content.select(e2|e2.isTypeOf(Box) and e2<>e and
    not((e.position.x >= (e2.position.x+e2.dimension.x)) or ((e.position.x+e.
    dimension.x) <= e2.position.x) or (e.position.y >= (e2.position.y+e2.dimension
    .y)) or ((e.position.y+e.dimension.y) <= e2.position.y))).collect(e2|e2.
    guiElement.name).concat(',');"
  rank {
    type: string
    correctValues: ""
```

8. El DSL UsabilityRules

```
}
correction:      "
// We get the overlapped elements
var elements = e.container.content.select(e2|e2.isTypeOf(Box) and e2<>e and not((
    e.position.x >= (e2.position.x+e2.dimension.x)) or ((e.position.x+e.
        dimension.x) <= e2.position.x) or (e.position.y >= (e2.position.y+e2.
            dimension.y)) or ((e.position.y+e.dimension.y) <= e2.position.y)));

// For each of them:
for (other : Box in elements) {
    // We get the distances between the different edges
    var right = other.position.x - (e.position.x + e.dimension.x);
    var left = (other.position.x + other.dimension.x) - e.position.x;
    var bottom = other.position.y - (e.position.y + e.dimension.y);
    var top = (other.position.y + other.dimension.y) - e.position.y;

    // We get the minimum
    var min = Sequence{right, left, bottom, top}.sortBy(v|v.abs()).first();

    // We move the other element in that direction
    if (right == min or left == min) {
        other.position.x = other.position.x - min;
        other.guiElement.position.x = other.position.x;
    }
    else {
        other.position.y = other.position.y - min;
        other.guiElement.position.y = other.position.y;
    }
}
}
"
```

Para terminar analizaremos una tercera regla de usabilidad, esta vez relacionada con la posición de los elementos.

Es un ejemplo más complejo donde tanto el cálculo como la corrección son bloques de código algo más extensos y donde el uso que se hace de los rangos no es tan evidente.

La regla de usabilidad en cuestión comprueba que no existan elementos solapados. Es evidente que dos o más elementos que aparezcan superpuestos constituyen un fallo evidente de usabilidad, pues alguno de ellos (o ambos) podrían no apreciarse con claridad por parte del usuario, y en cualquier caso la interfaz da la impresión de estar mal estructurada.

Para comprobar el solapamiento de dos elementos debemos atender a su posición y a sus dimensiones. Por tanto, el tipo de elemento sobre el que aplicaremos esta regla de usabilidad es el elemento **Box**, definido especialmente en el metamodelo de usabilidad para casos como este.

Un aspecto muy importante que debemos considerar cuando se comprueban condiciones como el solapamiento es el contexto en el que se encuentran los elementos. No podemos aplicar una regla de usabilidad como esta sobre cualquier par de elementos, pues los elementos pueden pertenecer a ventanas o paneles diferentes y, por tanto, no tiene sentido comprobar sus posiciones ya que no se van a mostrar simultáneamente al usuario. Por esta razón, uno debe comprobar el solapamiento de un elemento con aquellos otros elementos que se encuentren en su mismo contenedor. Dos elementos que se hallen, por ejemplo, dentro del mismo panel sí serán mostrados al mismo tiempo y por

tanto la relación entre sus posiciones sí que es relevante.

Este es el motivo de que incluyamos en esta regla, a diferencia de las dos anteriores, una restricción que asegura que el elemento sobre el que se aplica la regla tiene definido un contenedor padre. Es necesario incluir la restricción ya que durante el cálculo buscaremos el resto de elementos de nuestro contenedor; si el elemento no está contenido en otro (como ocurre con las ventanas principales de la interfaz) entonces no tiene sentido ni es posible aplicar esta regla.

El código relativo al cálculo realiza un proceso del que ya hemos comentado algunos aspectos. La búsqueda se reducirá a los elementos dentro del mismo contenedor que el elemento que está siendo analizado. Lo que haremos será comprobar si alguno de ellos se solapa con él. Para lograrlo, los recorreremos uno a uno y comprobamos las diferencias entre las posiciones de cada par de elementos, teniendo en cuenta las dimensiones de ambos para calcular los márgenes y averiguar si existe solapamiento o no. La parte final de la sentencia recoge los nombres de los elementos superpuestos y los concatena, devolviendo un String.

Esta es la parte importante dentro de la concepción de esta regla de usabilidad, y lo que la diferencia en parte de las que hemos visto anteriormente. Inicialmente, podría pensarse que una regla de usabilidad que comprueba el solapamiento entre dos elementos debe aplicarse precisamente sobre esos dos elementos, y devolver un booleano que informe de la existencia o no de superposición. Este es quizás el enfoque más intuitivo de una norma como la que nos ocupa, pero presenta algunos inconvenientes. Para empezar, no está directamente soportado por nuestro metamodelo actual de reglas. Una regla de usabilidad, tal y como la hemos definido, se aplica siempre sobre un único elemento. Esto radica en parte en ciertas limitaciones del lenguaje de transformación modelo a modelo.

Además, con un enfoque así se generarían demasiados errores de usabilidad en una situación en la que un elemento se solapa con muchos otros, pues para cada par se generarían dos defectos.

Por fortuna, el metamodelo actual de reglas cuenta con herramientas lo suficientemente flexibles como para soportar la definición de reglas de usabilidad como esta, aunque sea mediante el uso de una técnica ligeramente menos intuitiva.

Lo que haremos será definir un rango de tipo `StringRank` donde el único valor correcto será la cadena vacía. En el proceso de cálculo, como ya hemos visto, obtenemos los nombres de los elementos que se solapan con el elemento analizado, concatenados formando una única cadena. Si no se ha encontrado ningún elemento, entonces el cálculo devolverá una cadena vacía y este valor, al estar dentro del rango, se considerará correcto. En caso contrario, se detectará un error de usabilidad y se dispondrá de los nombres de los elementos conflictivos, que el usuario podrá consultar para, si así lo desea, realizar posteriormente una corrección manual.

Esta forma de entender el rango de tipo `StringRank` es diferente a la que veíamos en el ejemplo anterior, donde el rango nos servía para definir los valores correctos que configuraban la paleta de colores. En este caso, el valor correcto es la cadena vacía y la función de cálculo es la que obtiene los elementos conflictivos.

Se trata de un ejemplo de cómo, bien entendida, la estructura de reglas definida es

8. *El DSL UsabilityRules*

capaz de soportar reglas de diferente tipo, incluidas reglas que, vistas desde otro enfoque, parecen no encajar con nuestro metamodelo.

Llegamos por último al código de corrección. Aunque es un bloque de código relativamente extenso para lo que hemos visto hasta ahora, no nos detendremos demasiado, pues tiene que ver con la gestión de las posiciones y dimensiones pero no introduce nuevos conceptos o técnicas que tengan que ver con el metamodelo de reglas o faciliten su entendimiento.

En resumidas cuentas, el código obtiene el solapamiento entre cada par de elementos y mueve uno de ellos en una dirección la distancia suficiente para eliminar dicha superposición, tratando de realizar el desplazamiento en la dirección que implique un menor movimiento.

Al terminar el proceso, los elementos quedan adyacentes (con una separación igual a cero) en la dirección en la que se superponían. Esta técnica de corrección soluciona de forma satisfactoria el solapamiento entre pares de elementos, incluidos aquellos casos en que existe un solapamiento encadenado entre más de dos elementos; pero no está asegurado que siempre funcione bien. En escenarios donde los elementos están muy juntos, el desplazamiento de un elemento realizado para corregir un solapamiento puede originar un nuevo solapamiento. En estos casos, será necesaria más de una pasada del corrector pues las primeras pueden originar nuevos fallos.

Este tipo de comportamiento puede estar presente en cualquiera de las normas que manipulan la posición de los elementos, de forma que la corrección de un solapamiento puede ocasionar no sólo otros solapamientos sino, por ejemplo, un fallo en la alineación horizontal o vertical. Aunque un número suficiente de ejecuciones del corrector suele terminar liberando la interfaz de errores de usabilidad, es importante tener en cuenta este tipo de comportamiento que, en ocasiones, puede darse en interfaces con muchos elementos y defectos de usabilidad relacionados. En casos así, muchas veces la corrección manual es más adecuada, pues el usuario cuenta con toda la información sobre los elementos defectuosos y puede planear con mayor efectividad la disposición que mejor soluciona el problema.

9 Análisis y corrección de defectos

En este capítulo vamos a explicar en detalle cómo se ha implementado el análisis y corrección de los defectos en las interfaces. El proceso de análisis aplica las reglas sobre el modelo de usabilidad y genera un modelo de defectos, el cual es utilizado para generar un informe al usuario, mientras que la corrección sigue un proceso similar donde, una vez realizada la detección de los defectos, se ofrece al usuario la posibilidad de corregirlos y se genera un informe con el resultado del proceso completo de corrección.

Comenzaremos con la descripción del metamodelo de defectos de usabilidad.

9.1 Metamodelo de defectos

El metamodelo `UsabilityDefects` representa los defectos detectados en las interfaces durante la etapa de análisis. Como muestra la Figura 9.1 se trata de un metamodelo muy sencillo que tiene un elemento raíz denominado `Evaluation` que agrega un conjunto de elementos `Defect`. La metaclase `Defect` representa un defecto y tiene varios atributos:

- `usabilityRule`: regla de usabilidad cuya aplicación generó el defecto de usabilidad
- `element`: el nombre del elemento asociado al defecto, es decir, el elemento sobre el que se aplicó la regla de usabilidad
- `description`: es una cadena que describe la naturaleza del defecto de usabilidad
- `importance`: se definen tres niveles de importancia, equivalentes a los establecidos en las reglas de usabilidad
- `value`: el valor obtenido tras la aplicación de la regla de usabilidad sobre el elemento analizado
- `correctValue`: valor o conjunto de valores considerados correctos

Dado que el único fin del modelo de defectos es presentar al usuario un informe sobre los defectos encontrados en un formato legible y no es utilizado en el resto del proceso de transformación, hemos tratado de mantener el metamodelo lo más sencillo posible.

9. Análisis y corrección de defectos

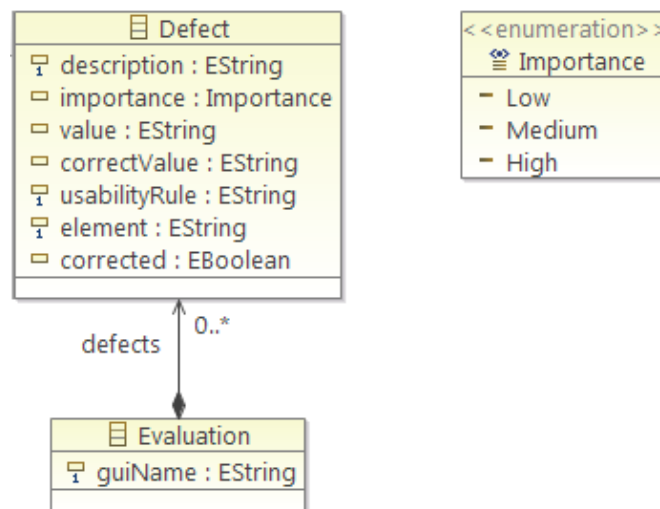


Figura 9.1.: Metamodelo Usability Defects.

Así, si queremos mostrar el elemento correspondiente a un defecto de usabilidad, por ejemplo, almacenamos su nombre en lugar de una referencia al propio elemento. Lo mismo ocurre con el resto de atributos, que son de tipos primitivos como cadenas o booleanos y que hemos introducido arriba.

El metamodelo se utiliza tanto durante el análisis como durante la corrección, y ese es el motivo de que ciertos atributos no sean obligatorios. Durante el proceso de análisis se extraerá más información que durante la corrección, donde los defectos ya han sido analizados y mostrados al usuario y ya sólo es necesaria la información fundamental.

El atributo booleano **corrected** está destinado al proceso de corrección y nos informará si un defecto de usabilidad ha sido corregido o no. Este atributo es fundamental para elaborar el informe de corrección presentado al usuario, cuya creación veremos a continuación.

9.1.1 Generación de los informes

Tanto análisis como corrección resultan en un modelo de defectos que conforma el metamodelo expuesto anteriormente y cuyo fin es ser presentado al usuario en un formato que sea entendible.

En ambos procesos se genera un modelo de defectos que es utilizado para generar un informe sobre los defectos destinado al usuario o experto en usabilidad. Decidimos generar estos informes como una tabla HTML, de modo que el usuario disponga de un documento que pueda ver desde su navegador.

Para convertir el modelo de defectos a este formato utilizamos una sencilla transformación modelo a texto haciendo uso del lenguaje EGL ofrecido por Epsilon, ya introducido brevemente en la Sección 4.3.1.3.

9.1.1.1 Generación del informe de análisis

Cada defecto tiene un conjunto de propiedades, cadenas de texto en su mayoría, que son las que debemos mostrar en el informe. Como estamos generando un documento HTML la mayor parte de la transformación, que puede consultarse en los archivos fuente bajo el nombre `Defects2AnalysisReport.egl`, es código HTML que será volcado tal cual en el fichero de salida.

En este apartado no analizaremos la estructura del documento HTML pero sí la generación de la tabla con los datos, pues es aquí donde entran en juego las consultas más importantes sobre el modelo de defectos.

El código HTML con extractos de EGL que da lugar a la creación de esta tabla es el siguiente:

```
<table id="usability-defects" summary="table containing the usability defects found in
the GUI" >
  <caption>
    Usability defects
  </caption>
  <thead>
    <tr>
      <th scope="col">Importance</th>
      <th scope="col">Usability rule</th>
      <th scope="col">Description</th>
      <th scope="col">Element</th>
      <th scope="col">Correct value</th>
      <th scope="col">Current value</th>
    </tr>
  </thead>

  <tfoot>
  </tfoot>

  <tbody>
    [% for (defect in Defect.allInstances()) { %]
    <tr>
      <td><span class="[%=defect.importance%]">[%=defect.importance%]</span></td>
      <td>[%=defect.usabilityRule%]</td>
      <td>[%=defect.description%]</td>
      <td>[%=defect.element%]</td>
      <td>[%=defect.correctValue%]</td>
      <td>[%=defect.value%]</td>
    </tr>
    [% } %]
  </tbody>
</table>
```

Las cabeceras de la tabla son código HTML puro que será volcado sobre el fichero de salida, pero a la hora de definir el cuerpo de la tabla es necesario incluir sentencias EGL que accedan a los datos del modelo de defectos.

El código encargado de esto es muy sencillo. Simplemente recorreremos todos los defectos del modelo mediante un bucle `for` e imprimimos en la columna adecuada cada una de sus propiedades. La correspondencia entre las columnas y los atributos del defecto es evidente; la iteración irá recorriendo todos los defectos del modelo e imprimiendo para cada uno de ellos una nueva fila con los valores de todos sus atributos como columnas.

9. Análisis y corrección de defectos

Lo único a destacar es el uso de clases CSS (estilos) para dotar de un sistema de colores a la importancia de un defecto (Low: amarillo, Medium: naranja, High: rojo). Las clases CSS han sido nombradas con el valor del enumerado con lo que este es el valor que imprimimos entre comillas para el atributo `class`.

Como se puede ver, la sencillez del lenguaje EGL y del metamodelo de defectos hace que la generación del informe no entrañe mayor dificultad. En la Figura 9.2 podemos ver un ejemplo de informe donde se listan algunos de los defectos de usabilidad encontrados sobre una interfaz llamada Test GUI.

Usability analysis					
Target GUI: Test GUI					
Some usability defects have been detected					
Importance	Usability rule	Description	Element	Correct value	Current value
Medium	Tabfocus	Tabfocus property has not been set correctly on the elements of the panel	formPanel	"	The element lastNameInput is higher than its predecessor.
High	Contrast	Low contrast between foreground and background colors	formButton	4.5-*	3.998476940343616
High	Separation	The element is too close to other elements in its container	formPanel	"	mainMenu
High	Separation	The element is too close to other elements in its container	mainMenu	"	formPanel
Medium	Accessibility information	Accessibility information has not been set	mainWindow	true	false
Medium	Accessibility information	Accessibility information has not been set	formPanel	true	false
Medium	Accessibility information	Accessibility information has not been set	nameLabel	true	false
Medium	Accessibility information	Accessibility information has not been set	nameInput	true	false
Medium	Accessibility information	Accessibility information has not been set	lastNameLabel	true	false

Figura 9.2.: Informe de análisis de la usabilidad.

9.1.12 Generación del informe de corrección

Si la generación del informe para el análisis no necesitó de una gran explicación, la corrección todavía requiere menos.

El fichero HTML es exactamente igual en su estructura al que vimos en el apartado anterior, solo que en este caso mostraremos una información más resumida de cada defecto y añadiremos, eso sí, el atributo que nos dice si ha sido corregido o no.

El código correspondiente a la generación de la tabla es el siguiente:

```
<table id="usability-defects" summary="table containing the usability defects found in  
the GUI and the result of the correction" >  
  <caption>  
    Usability defects correction  
  </caption>  
<thead>
```

```

    <tr>
      <th scope="col">Usability rule</th>
      <th scope="col">Description</th>
      <th scope="col">Element</th>
      <th scope="col">State</th>
    </tr>
  </thead>

  <tfoot>
  </tfoot>

  <tbody>
    [% for (defect in Defect.allInstances()) { %]
    <tr>
      <td>[%=defect.usabilityRule%]</td>
      <td>[%=defect.description%]</td>
      <td>[%=defect.element%]</td>
      <td><span class="correction-[%=defect.corrected%]">[% if
        % (defect.corrected)
          { %]Corrected[% }
        else { %]Not corrected[% } %]</span></td>
    </tr>
    [% } %]
  </tbody>
</table>

```

Como resultado del proceso de corrección, no nos interesa mostrar al usuario todos los detalles de cada uno de los defectos encontrados, pues se supone que esto es tarea del análisis. En su lugar, mostramos la información mínima para poder identificar cada defecto y distinguimos entre los errores que han sido corregidos y los que no. Este último dato es el característico del proceso de corrección y el que verdaderamente importa al usuario en este punto del sistema.

De forma similar a como ocurría con la importancia de un defecto, atributo que mostrábamos en el informe correspondiente al análisis, definimos aquí también un código de colores, en este caso para el atributo `corrected`, de tal forma que los errores que hayan sido corregidos lo indicarán utilizando el color verde y los que no, el rojo. La forma de implementarlo es parecida, nombrando las clases css según los valores posibles del atributo. Una sencilla sentencia `if else` es suficiente para realizar la distinción entre ambos tipos de defectos, mostrando el mensaje adecuado en cada caso.

En la Figura 9.3 podemos ver un extracto del informe generado para la corrección de la usabilidad en una interfaz de ejemplo.

9.2 Análisis

A continuación describiremos la implementación del proceso de análisis de una GUI para encontrar defectos de usabilidad.

En principio, el proceso de análisis consistiría en recorrer el modelo `UsabilityGUI`, que es generado para representar la interfaz objeto de la evaluación de usabilidad, para cada una de las reglas de usabilidad y comprobar cuáles de ellas no se satisfacen y provocan defectos en la interfaz. Implementar este proceso por medio de una única transformación

9. Análisis y corrección de defectos

Usability correction
Target GUI: Test GUI
Results of the usability correction process

Usability rule	Description	Element	State
Tabfocus	Tabfocus property has not been set correctly on the elements of the panel	formPanel	Not corrected
Contrast	Low contrast between foreground and background colors	formButton	Not corrected
Separation	The element is too close to other elements in its container	formPanel	Not corrected
Separation	The element is too close to other elements in its container	image02	Corrected
Separation	The element is too close to other elements in its container	image03	Corrected
Separation	The element is too close to other elements in its container	mainMenu	Not corrected
Accessibility information	Accessibility information has not been set	mainWindow	Not corrected
Accessibility information	Accessibility information has not been set	formPanel	Not corrected
Accessibility information	Accessibility information has not been set	nameLabel	Not corrected

Figura 9.3.: Informe de corrección de la usabilidad.

M2M que tenga como entrada el modelo UsabilityGUI y el modelo UsabilityRules y genere como salida un modelo de defectos es realmente complicado, dado que no se trata de un mapping entre metamodelos, sino de algo más complejo.

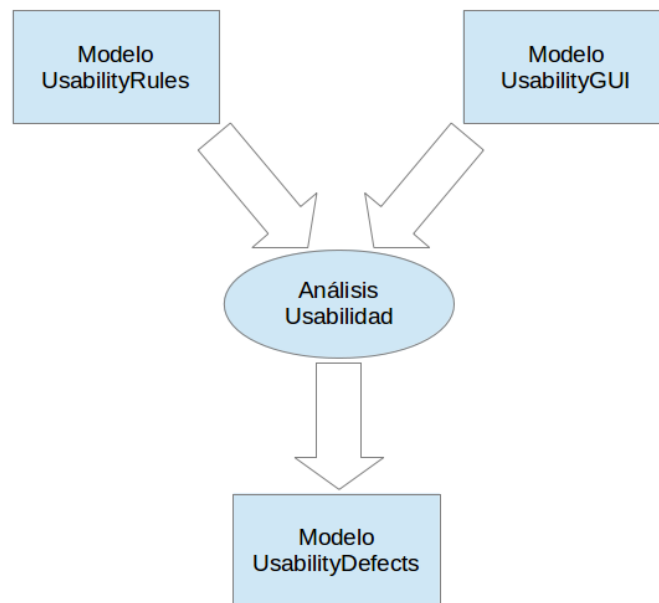


Figura 9.4.: Visión general del sistema de análisis de la usabilidad.

Nos hemos enfrentado al problema anterior del siguiente modo. Hemos generado una transformación M2M expresada en ETL por medio de una transformación M2T cuya entrada es el modelo de reglas de usabilidad. La transformación M2M generada es la que

tiene como entrada el modelo UsabilityGUI y se encarga de realizar el análisis generando el modelo de defectos. A continuación vamos a explicar estas dos transformaciones.

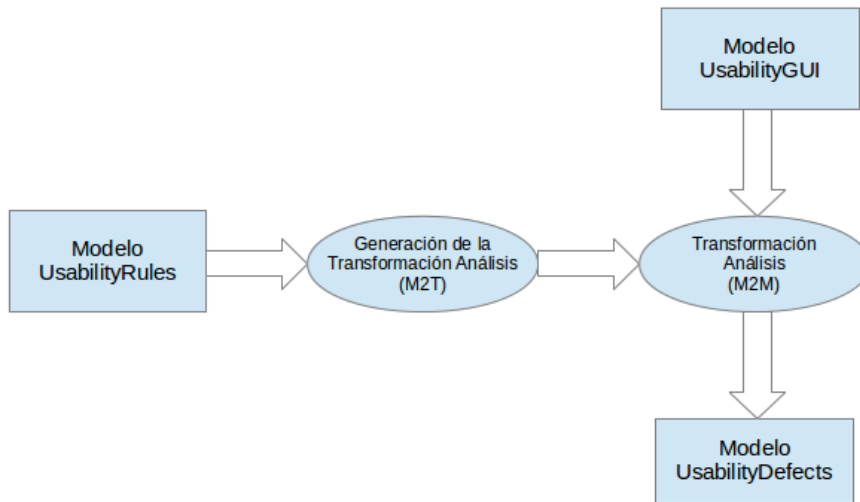


Figura 9.5.: Arquitectura del sistema de análisis de la usabilidad propuesto.

9.2.1 Generación de la transformación de análisis

La transformación M2T recorre todas las reglas de usabilidad en el modelo UsabilityRules. Para cada regla se generará una regla de la transformación M2M que se almacenará en el fichero de salida, en nuestro caso un fichero ETL (extensión `.etl`). A continuación mostramos la estructura de las reglas generadas ya que es común para todas las reglas de usabilidad.

```

operation restriction()
operation calculation()

rule Example
  transform e : elementType
  to defect : Defect {
    guard : restriction() and not (calculation() inside rank)

    << defect creation >>
  }

```

El nombre de cada regla es el nombre de la correspondiente regla de usabilidad que se aplica. El elemento origen es determinado por el atributo `elementType` de las reglas y el elemento destino es `Defect`. Hay una guarda (filtro) que comprueba si se debe generar el defecto.

Las reglas se aplicarán sobre los elementos indicados, siempre que cumplan la restricción y el valor obtenido en el cálculo no se encuentre dentro del rango definido. Si se dan estas condiciones, entonces el elemento se considera erróneo y se genera un defecto. Los detalles de su creación los veremos más adelante.

9. Análisis y corrección de defectos

El metamodelo de reglas de usabilidad cuenta con toda la información necesaria para definir estas reglas de transformación. La correspondencia entre una regla de usabilidad y su regla de transformación asociada puede apreciarse en la Figura 9.6. En el diagrama, por supuesto, el proceso aparece simplificado, pero al fin y al cabo este es el mecanismo que seguiremos, incluyendo cada trozo de código en su lugar correspondiente.

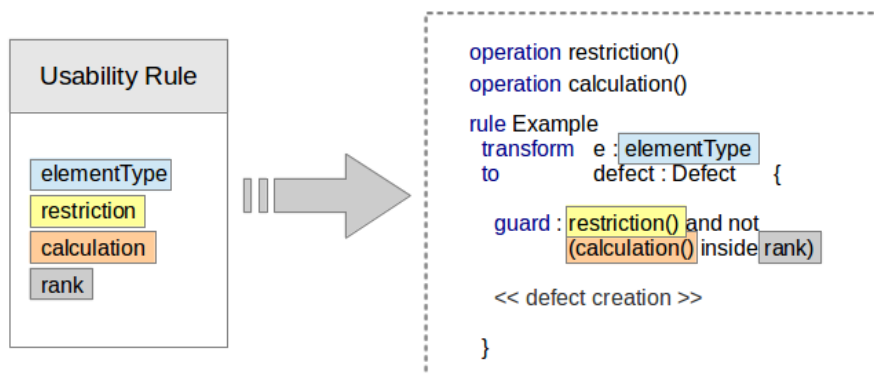


Figura 9.6.: Generación de una regla de transformación M2M a partir de una regla de usabilidad.

A continuación, presentaremos las reglas M2T que generan las reglas M2M con el formato anterior. Para empezar, y con el objetivo de simplificar el código, se definen dos operaciones: las relativas a la restricción y al cálculo, de la siguiente manera:

```
[%if(rule.restriction.isDefined()) { %]
operation [%=rule.name.replace(' ', '_')]_restriction(e : UsabilityGUI![%=rule.
    elementType%]) {
    [%=rule.restriction.code%]
}
[% } %]

operation [%=rule.name.replace(' ', '_')]_calculation(e : UsabilityGUI![%=rule.
    elementType%])
: [% switch (rule.rank.type().name) {
    case "IntegerRank" : out.print("Integer");
    case "DoubleRank" : out.print("Real");
    case "BooleanRank" : out.print("Boolean");
    case "StringRank" : out.print("String"); } %] {
    [%=rule.calculation.code%]
}
}
```

El lenguaje EGL, introducido en la Sección 4.3.1.3 de este documento, es muy sencillo y ya pudimos ver algunos ejemplos de su uso durante la explicación de la generación de los informes, en la Sección 9.1.1.1 y la Sección 9.1.1.2.

Las operaciones llevarán el nombre de la regla de usabilidad en cuestión seguido de `_restriction` o `_calculation`, según sea el caso. Esto garantiza que las funciones sean únicas dentro del fichero de transformación, siempre que todas las reglas de usabilidad tengan nombres diferentes.

La definición de la función de restricción es sencilla: toma como parámetro el elemento de la interfaz a analizar y devuelve un booleano, indicando si se cumple la restricción o no. El código se vuelca tal cual ha sido definido en la regla de usabilidad.

Lo mismo ocurre con el código de la función de cálculo, solo que en este caso el tipo de dato devuelto dependerá del rango definido en la regla. Un sencillo análisis de casos es suficiente para dar con la declaración correcta de la función.

Una vez definidas ambas funciones, estamos en condiciones de entender la generación completa de una regla de transformación. El código correspondiente es el que listamos a continuación:

```
@greedy
rule [%=rule.name.replace(' ', '_')]
  transform e : UsabilityGUI![%=rule.elementType%]
    to defect : AnalysisDefects!Defect {

      guard : [%if(rule.restriction.isDefined()) { %} [%=rule.name.replace(' ', '_')
        %]_restriction(e) and [% } %]
        not([%if (rule.rank.isKindOf(IntegerRank)) { %}
[%=rule.name.replace(' ', '_')]_calculation(e) >= [%=rule.rank.min%][%if (rule.rank.
  max <> -1) { %} and [%=rule.name.replace(' ', '_')]_calculation(e) <= [%=rule.
    rank.max%][% } %])
[%=rule.name.replace(' ', '_')]_calculation(e) >= [%=rule.rank.min%][%if (rule.rank.
  max <> -1) { %} and [%=rule.name.replace(' ', '_')]_calculation(e) <= [%=rule.
    rank.max%][% } %])
[% } else if (rule.rank.isKindOf(DoubleRank)) { %}
[%=rule.name.replace(' ', '_')]_calculation(e) >= [%=rule.rank.min%][%if (rule.rank.
  max <> -1) { %} and [%=rule.name.replace(' ', '_')]_calculation(e) <= [%=rule.
    rank.max%][% } %])
[% } else if (rule.rank.isKindOf(BooleanRank)) { %}
[%=rule.name.replace(' ', '_')]_calculation(e) == [%=rule.rank.correctValue%])
[% } else if (rule.rank.isKindOf(StringRank) and rule.rank.
  correctValues <> '""') { %}
Sequence{[%=rule.rank.correctValues%]}.includes([%=rule.name.replace(' ', '_')]_
  _calculation(e)))
[% } else { %}false]
[% } %]
defect.description = "[%=rule.description.errorMessage%]";
defect.importance = AnalysisDefects!Importance#[%=rule.importance%];
defect.usabilityRule = "[%=rule.name%]";
defect.value = [%=rule.name.replace(' ', '_')]_calculation(e).asString
  ()[%if (rule.rank.hasProperty("unit")) { %} + " [%=rule.rank.unit
    %]"[% } %];
defect.correctValue = [%if (rule.rank.isKindOf(IntegerRank)) { %}"[%=
  rule.rank.min%]-[%if (rule.rank.max == -1) { %}*[%} else { %}"[%=
    rule.rank.max%][% } %]"[%if (rule.rank.unit.isDefined()) { %} [%=
      rule.rank.unit%][% } %]";
[% } else if (rule.rank.isKindOf(DoubleRank)) { %}"[%=rule.rank.
  min%]-[%if (rule.rank.max == -1) { %}*[%} else { %}"[%=rule
    .rank.max%][% } %]"[%if (rule.rank.unit.isDefined()) { %}
    [%=rule.rank.unit%][% } %]";
[% } else if (rule.rank.isKindOf(BooleanRank)) { %}"[%=rule.rank
    .correctValue%]";
[% } else if (rule.rank.isKindOf(StringRank)) { %}"[%=rule.rank.
    correctValues%]";
[% } %]
defect.element = e.guiElement.name;

evaluation.defects.add(defect);
}
```

9. Análisis y corrección de defectos

Analizaremos este código para entender cómo se construyen, lo que también supone entender cómo se comportan las reglas de transformación que guían el proceso de análisis de la usabilidad.

Todas las reglas de transformación se definirán como **greedy**. Si en una regla **greedy** establecemos como tipo de elemento de entrada el padre de una jerarquía de clases, la regla se aplicará sobre todos los elementos que sean de ese tipo o uno de los subtipos que heredan de él. Esto hace que si quisiéramos, por ejemplo, aplicar una regla sobre todos los elementos del modelo de interfaz, pudiésemos indicar como tipo de elemento en la regla `UsabilityElement` y la regla se aplicaría sobre todos los subtipos. Aunque no contamos con ningún caso práctico en que este comportamiento sea útil, pensamos que es el más adecuado.

El nombre de la regla de transformación es el nombre de la regla de usabilidad, una vez normalizada la cadena para eliminar espacios.

Como ya vimos en la Figura 9.6, el tipo de elemento que tomamos como entrada es el indicado en la propia regla de usabilidad, y el elemento generado es siempre **Defect**.

Hasta aquí, la generación de la regla de transformación es bastante sencilla.

La guarda es un poco más compleja. Como explicamos anteriormente, debemos comprobar que el elemento cumpla la restricción y si el cálculo arroja un valor fuera del rango.

Comprobar la restricción es muy sencillo, pues la función ya ha sido definida y devuelve un booleano, con lo que simplemente tenemos que invocarla (teniendo en cuenta eso sí que su especificación dentro de una regla de usabilidad es opcional).

Para comprobar el rango son necesarias algunas líneas de código más, pero el proceso es también muy sencillo. Un análisis de casos nos ayuda a imprimir sobre el archivo las comprobaciones adecuadas según el tipo de rango definido en la regla de usabilidad:

- Si el rango es numérico (`Integer` o `Double`), comprobamos que el cálculo realizado sobre el elemento arroje un valor mayor o igual que el valor establecido como mínimo y menor o igual que el máximo (el intervalo es cerrado).
- Si el rango es de tipo booleano, el valor devuelto por la función de cálculo debe ser exactamente igual que el contemplado como correcto en la regla de usabilidad.
- Si el rango es de tipo `StringRank`, comprobaremos si la cadena obtenida del cálculo sobre el elemento se encuentra dentro del conjunto de cadenas definidas como correctas en la regla de usabilidad. El formato en que se deben establecer los valores correctos en este tipo de rango (entre comillas y separados por comas) nos permite la fácil creación de un conjunto de tipo `Sequence` que nos permite realizar la comprobación que necesitamos mediante el método `includes`.

Este análisis de casos realizado mediante EGL nos permite generar el texto adecuado según el tipo de rango de cada regla de usabilidad, y es una de las partes más complejas de esta transformación M2T.

Una vez establecida la guarda de la regla, nos queda su contenido, es decir, la creación del defecto de usabilidad. Iremos rellenando los atributos del defecto con los valores adecuados, tomando muchos de ellos de la regla de usabilidad.

El proceso puede verse resumido en la Figura 9.7. La descripción de un defecto de usabilidad, por ejemplo, se corresponde con lo que en el metamodelo de reglas llamábamos `errorMessage` o mensaje de error, una cadena que describe el motivo del fallo.

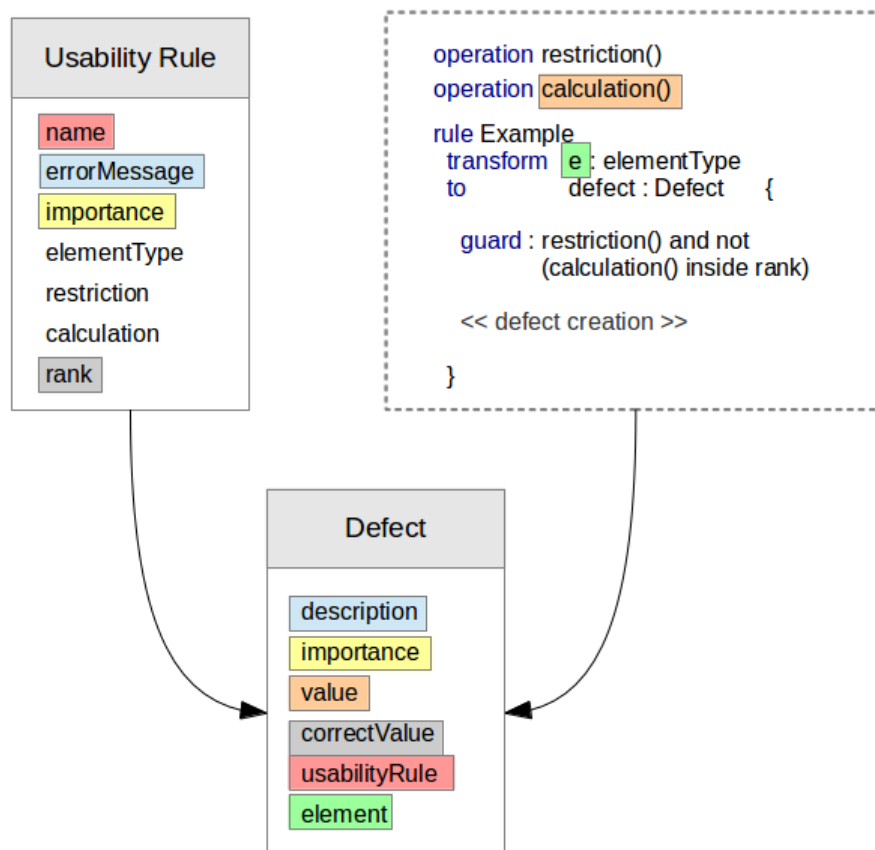


Figura 9.7.: Creación de un defecto a partir de una regla de usabilidad y su correspondiente regla de transformación M2M.

La importancia de un defecto es la importancia de su regla de usabilidad.

También almacenamos el nombre del elemento y el nombre de la propia regla de usabilidad, los atributos más identificativos de un defecto.

Por último, guardamos el valor obtenido de la función de cálculo invocándola una vez más, y rescatamos del rango los valores considerados correctos, para que posteriormente el usuario tenga información precisa de qué datos se esperaban y qué se ha obtenido.

Para imprimir los valores correctos debemos distinguir de nuevo entre los diferentes tipos de rango, ya que cada uno define unos atributos distintos y, como en el caso de los rangos numéricos, puede tener un formato específico (los rangos numéricos los

9. Análisis y corrección de defectos

representamos separando los valores mínimo y máximo por un guión y sustituyendo los -1 que representan el valor infinito por asteriscos). Una vez creado, el defecto es añadido al conjunto de defectos de usabilidad contenidos en el objeto raíz *Evaluation*.

Al final de este apartado retomaremos el proceso de creación de los defectos mostrando algunos ejemplos.

Este proceso que acabamos de ver se repite para cada una de las reglas de usabilidad del modelo de reglas de entrada, componiendo de esta manera un fichero de transformación modelo a modelo formado por tantas reglas de transformación como reglas de usabilidad hubiera en un principio.

Para entender bien cómo funciona la generación que acabamos de explicar, nada mejor que ver algunos ejemplos.

Esta es la regla de transformación generada para la regla de usabilidad 'Separación Etiqueta-Campo de entrada', que analiza la separación entre un campo de entrada y su etiqueta:

```
operation Label_input_separation_restriction(e : UsabilityGUI!Input) {
    return e.label.isDefined();
}

operation Label_input_separation_calculation(e : UsabilityGUI!Input) : Integer {
    return (e.box.position.x-(e.label.box.position.x+e.label.box.dimension.x))
        .max(e.box.position.y-(e.label.box.position.y+e.label.box.dimension.y));
}

@greedy
rule Label_input_separation
    transform e : UsabilityGUI!Input
        to defect : AnalysisDefects!Defect {

            guard : Label_input_separation_restriction(e) and
                    not(Label_input_separation_calculation(e) >= 0
                        and Label_input_separation_calculation(e) <= 10)

            defect.description = "The label is too far from its input field";
            defect.importance = AnalysisDefects!Importance#Medium;
            defect.usabilityRule = "Label input separation";
            defect.value = Label_input_separation_calculation(e).asString() + " px";
            defect.correctValue = "0-10 px";
            defect.element = e.guiElement.name;

            evaluation.defects.add(defect);
        }
    }
```

La regla define las operaciones de restricción y cálculo en base al código establecido por el usuario. Al tener asociado un rango numérico, se comprueba si la función de cálculo devuelve un valor que no es mayor que el mínimo ni menor que el máximo (en este caso 0 y 10 respectivamente).

Los nombres, descripciones e importancia de la regla son asignados a los atributos del elemento defecto generado, así como el rango de valores correcto adecuadamente formateado.

En la Figura 9.8 podemos ver con mayor claridad cómo es el proceso de generación de la regla de transformación a partir de la información disponible en el modelo de reglas de

usabilidad (para mayor claridad se toma la representación textual de la regla, siguiendo el DSL explicado en la Sección 8.1.1).

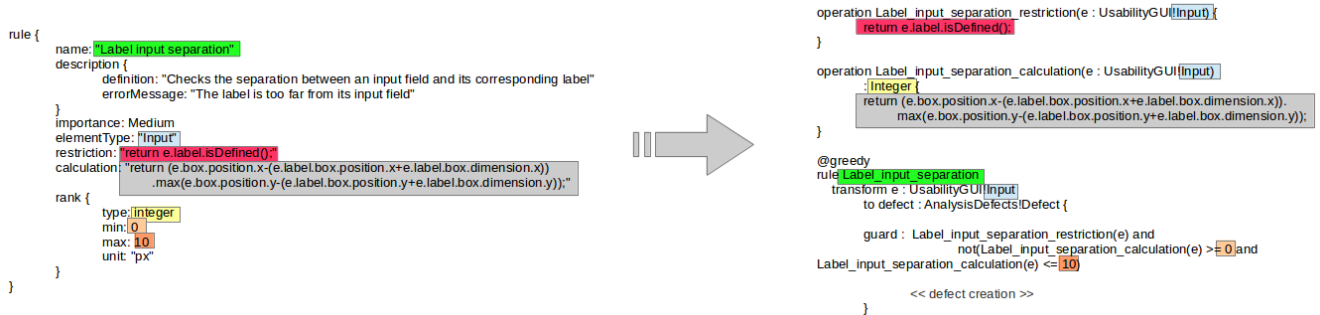


Figura 9.8.: Regla de transformación M2M generada a partir de la regla de usabilidad 'Separación Etiqueta-Campo de entrada'.

A continuación podemos ver otro ejemplo, en este caso correspondiente a la regla de usabilidad que ya hemos estudiado en otras ocasiones y que analiza el color de fondo de un elemento:

```

operation Background_color_palette_calculation(e : UsabilityGUI!Color)
: String {
  return e.background;
}

@greedy
rule Background_color_palette
transform e : UsabilityGUI!Color
to defect : AnalysisDefects!Defect {

  guard :
    not(Sequence{'#000000', '#FF0000', '#00FF00', '#0000FF', '#FFFF00', '#00FFFF', '#FF00FF', '#FFFFFF'}.includes(Background_color_palette_calculation(e)))

  defect.description = "The background color of the element is not in the palette";
  defect.importance = AnalysisDefects!Importance#High;
  defect.usabilityRule = "Background color palette";
  defect.value = Background_color_palette_calculation(e).asString();
  defect.correctValue = "'#000000', '#FF0000', '#00FF00', '#0000FF', '#FFFF00', '#00FFFF', '#FF00FF', '#FFFFFF'";
  defect.element = e.guiElement.name;

  evaluation.defects.add(defect);
}

```

En este caso no se cuenta con ninguna restricción, con lo que la función correspondiente no ha sido definida. Tenemos asociado un rango de tipo cadena, de forma que para averiguar si existe un error o no construimos un conjunto con los valores correctos y comprobamos si el valor obtenido para el elemento actual está en dicho conjunto o no. Sólo si no lo está tendremos una situación de error y por tanto se aplicará la transformación generando un nuevo defecto.

9. Análisis y corrección de defectos

El cuerpo de la regla es similar al anterior y sencillo de comprender.

Por último, mostraremos un tercer ejemplo de generación correspondiente a una sencilla regla de usabilidad relacionada con la propiedad tooltip de un elemento:

```
operation Tooltip_calculation(e : UsabilityGUI!Tooltip)
    : Boolean {
    return e.text.length() > 0;
}

@greedy
rule Tooltip
    transform e : UsabilityGUI!Tooltip
        to defect : AnalysisDefects!Defect {

        guard :
            not(Tooltip_calculation(e) == true)

        defect.description = "Tooltip text has not been set";
        defect.importance = AnalysisDefects!Importance#Low;
        defect.usabilityRule = "Tooltip";
        defect.value = Tooltip_calculation(e).asString();
        defect.correctValue = "true";
        defect.element = e.guiElement.name;

        evaluation.defects.add(defect);
    }
}
```

La regla de usabilidad es muy sencilla y así lo es también la regla de transformación generada a partir de ella. En este caso el rango asociado es de tipo booleano, con lo que la guarda y la impresión del valor correcto son muy simples.

Los ejemplos aquí citados así como la representación gráfica del proceso que hemos visto en la Figura 9.8 deberían ser suficientes para mostrar con claridad cómo funciona esta generación. La explicación dada debería ser suficiente para comprender no sólo cómo se genera la transformación M2M que realiza el análisis sino también cómo trabaja esta transformación. No obstante, si uno consulta el fichero de nombre Analysis.etl en el proyecto Eclipse que acompaña a esta Memoria, verá algunas funciones adicionales que todavía no hemos explicado. Estas operaciones aparecen en el fichero de generación Rules2Analysis.egl como texto plano para ser volcado directamente en la salida, ya que no contiene código EGL.

Se trata de funciones auxiliares que necesitamos en el fichero de transformación modelo a modelo y que son necesarias para su correcto desempeño o bien facilitan la definición de determinadas reglas de usabilidad. Algunas de ellas ya han sido mencionadas previamente en este documento.

Para empezar, si uno consulta la transformación M2M (aunque, por supuesto, también puede verlo en la transformación M2T que la genera) observará en primer lugar la definición de los siguientes dos bloques de código:

```
pre createEvaluation {
    var evaluation = new AnalysisDefects!Evaluation;
    evaluation.guiName = UsabilityGUI!GUI.allInstances().first().guiElement.name;
}

post evaluationDone {
```

```
System.user.inform("Usability analysis finalized. A report has been generated.");
}
```

Estos bloques de código son ejecutados al principio y al final de la transformación, respectivamente.

El bloque inicial se encarga de crear el elemento raíz *Evaluation*, mientras que el bloque último simplemente muestra un mensaje al usuario informando del fin del proceso de análisis. Este mecanismo de comunicación con el usuario que proporciona EOL tendrá un mayor protagonismo durante la corrección.

Bloques pre y post aparte, se definen una serie de funciones auxiliares cuyo fin es ampliar la funcionalidad del metamodelo orientado a la usabilidad. Se trata de operaciones y propiedades calculadas que no son lo suficientemente importantes para incluirlas en el metamodelo Usability-GUI pero sí para añadirlas en el fichero de transformación. Por ejemplo, las funciones relativas al código RGB de un color, del que ya hablamos algo en un apartado anterior. EOL nos permite ampliar la funcionalidad de cualquier tipo de dato a través de funciones que nosotros mismos definimos. De esta forma, hemos extendido la funcionalidad del tipo String añadiendo tres métodos de consulta que, suponiendo que el contenido del String es la representación hexadecimal de un color, nos devuelven sus correspondientes valores R, G o B:

```
operation String getR() : Integer {
    return self.substring(1,3).hexToInt();
}

operation String getG() : Integer {
    return self.substring(3,5).hexToInt();
}

operation String getB() : Integer {
    return self.substring(5,7).hexToInt();
}
```

Estos métodos utilizan una función adicional *hexToInt* que toma un String con la representación de un número en hexadecimal y devuelve el entero equivalente en base decimal.

La otra operación definida mediante este sistema es la relativa a la luminancia de un color, que se define como:

```
operation String getLuminance() : Real {
    var initialValues = Sequence{self.getR(), self.getG(), self.getB()};
    var finalValues = new Sequence;

    for (value : Integer in initialValues) {
        var finalValue = value.asReal()/255;
        if (finalValue <= 0.03928)
            finalValue = finalValue/12.92;
        else
            finalValue = ((finalValue+0.055)/1.055).pow(2.4);

        finalValues.add(finalValue);
    }

    return finalValues.first()*0.2126 + finalValues.second()*0.7152
}
```

9. Análisis y corrección de defectos

```
+ finalValues.third()*0.0722;  
}
```

Se trata de nuevo de un método añadido al tipo de dato String, y se encarga de calcular la luminancia de un color a partir de sus valores RGB y una fórmula determinada.

La definición de este tipo de funciones nos permite invocar sobre un String cualquiera de los métodos añadidos, como vemos en la propia función de luminancia cuando se invocan los métodos de consulta anteriores sobre un String (concretamente, sobre la instancia actual o `self`).

Se puede apreciar con estos ejemplos el tipo de funciones que uno puede declarar de esta forma en el fichero de transformación. Incluir el concepto de luminancia en el metamodelo orientado a la usabilidad es demasiado específico, pues no se considera esta una característica importante en un elemento de la interfaz. Sin embargo, es un concepto necesario para alguna regla de usabilidad como la que analiza el contraste. Por esta razón decidimos añadirla a este nivel y permitir por tanto su uso en los códigos de cálculo o corrección de las reglas.

Lo mismo ocurre con los métodos RGB. La representación que se hace de un color en el metamodelo de usabilidad utiliza el código hexadecimal, y no hay necesidad de complicar el metamodelo añadiendo, además, los valores RGB correspondientes. En su lugar, la obtención de esta información se incorpora en el fichero de transformación, con lo que tenemos esta funcionalidad disponible manteniendo el metamodelo de usabilidad con la representación mínima y necesaria de un color.

Una vez explicadas estas operaciones auxiliares, que podrían ampliarse en un futuro con la aparición de nuevas necesidades comunes en otras reglas de usabilidad, queda completamente descrita la transformación modelo a modelo generada. El archivo generado (Analysis.etl) puede consultarse dentro de la carpeta *transformations* del proyecto.

Entre las dieciocho reglas de transformación que incluye se encuentran las tres que hemos mostrado anteriormente a modo de ejemplo.

Ahora mostraremos un ejemplo de transformación de un modelo de usabilidad en un modelo defectos aplicando la transformación generada, aunque en la Sección 10 de este documento se verá un ejemplo más completo del análisis y corrección de una interfaz, que mostrará también los informes generados para el usuario pues en ellos se aprecian mejor los defectos de usabilidad encontrados.

En la Figura 9.9 vemos un ejemplo de modelo de defectos resultado de la ejecución del proceso de transformación modelo a modelo tomando una interfaz de ejemplo.

El modelo, abierto con el editor de modelos Ecore de Eclipse, muestra el elemento raíz **Evaluation** creado en el bloque de código **pre** del fichero de transformación.

Este elemento contiene una serie de defectos que son los creados a partir de los elementos que han cumplido la guarda de alguna de las reglas de transformación, es decir, que no han cumplido alguna de las reglas de usabilidad.

En la Figura 9.10 pueden verse los atributos de uno de estos defectos, creado por una de las reglas de transformación que vimos anteriormente (la relativa al texto tooltip de un elemento).

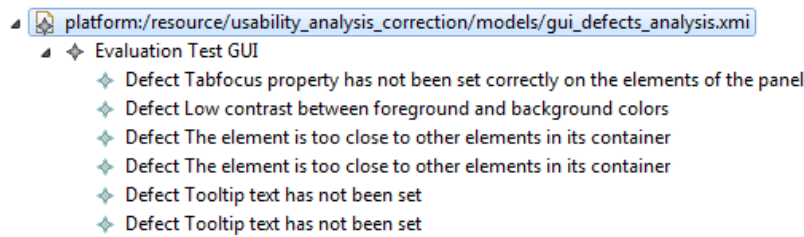


Figura 9.9.: Modelo de defectos generado como resultado del análisis de la usabilidad de una interfaz.

Property	Value
Corrected	false
Correct Value	true
Description	Tooltip text has not been set
Element	lastnameLabel
Importance	Low
Usability Rule	Tooltip
Value	false

Figura 9.10.: Atributos de uno de los defectos de usabilidad detectados.

En la imagen pueden apreciarse los valores tomados por cada una de las propiedades, tal y como son asignados en la regla de transformación.

En la Figura 9.11 puede verse la información detallada de otro defecto de usabilidad, en este caso relacionado con una regla de usabilidad que mide el contraste entre los colores de frente y fondo de un elemento.

Property	Value
Corrected	false
Correct Value	4.5-*
Description	Low contrast between foreground and backgrou...
Element	formButton
Importance	High
Usability Rule	Contrast
Value	3.998476940343616

Figura 9.11.: Atributos de uno de los defectos de usabilidad detectados.

La regla utiliza la función auxiliar que vimos con anterioridad para obtener la luminancia de un color, y obtiene el contraste como la relación entre la luminancia de ambos colores.

El resto de propiedades ha sido convenientemente establecido, incluyendo el formateo adecuado del rango de valores correctos (con la sustitución del valor -1 por un asterisco tal y como programamos en el fichero de transformación modelo a texto).

9.3 Corrección

Como ya adelantábamos, el proceso de corrección guarda muchos paralelismos con el análisis.

La estructura básica, tal y como la vimos en la Figura 9.5, se mantiene. Contamos igualmente con una transformación M2T que genera la transformación M2M encargada de la corrección. La única diferencia es que en este caso el modelo de interfaz puede considerarse tanto entrada como salida del sistema, pues el código de corrección modificará los elementos arreglando los defectos de usabilidad encontrados.

Las dos transformaciones involucradas en el proceso son muy parecidas a las que ya hemos estudiado para el análisis de la usabilidad. De hecho, para su creación lo que hicimos fue partir de la transformación M2T correspondiente al análisis y modificarla añadiendo o eliminando partes según convenía.

El esquema de las reglas de transformación M2M generadas será ahora el siguiente:

```
operation restriction()
operation calculation()
operation restriction()

rule Example
  transform e : elementType
  to defect : Defect {
    guard : restriction() and not (calculation() inside rank)

    << defect creation (simplified) >>

    << element correction >>
  }

```

A las operaciones de restricción y cálculo se suma ahora la de corrección, cuya generación es exactamente idéntica y se consigue con el siguiente código EGL:

```
[%if(rule.correction.isDefined()) { %]
operation [%=rule.name.replace(' ', '_')%]_correction(e : UsabilityGUI![%=rule.
  elementType%]) {
  [%=rule.correction.code%]
}
[% } %]
```

Una vez definida la función de corrección, generamos la regla de transformación. La cabecera y guarda de las reglas será la misma que ya vimos para el análisis, pero el cuerpo será ligeramente distinto. Mostramos a continuación únicamente el código de la transformación M2T que genera el cuerpo de la regla de transformación M2M:

```
defect.description = "[%=rule.description.errorMessage%]";
defect.usabilityRule = "[%=rule.name%]";
defect.element = e.guiElement.name;

[%if(rule.correction.isDefined()) { %]
if (autocorrect or System.user.confirm(e.guiElement.name + ": [%=rule.description.
  errorMessage%]. Do you want to correct this defect?", true)) {
  [%=rule.name.replace(' ', '_')%]_correction(e);
  defect.corrected = true;
}

```

```

else {
    defect.corrected = false;
}
[% } else { %]
    defect.corrected = false;
[% } %]

evaluation.defects.add(defect);

```

Como vemos, la diferencia radica en que la construcción del defecto se simplifica y la función de corrección es invocada.

Ya explicamos en la Sección 9.1 de este documento cómo el metamodelo de defectos se mantenía para los procesos de análisis y corrección pero en este último caso se tomaba una representación simplificada donde nos quedamos con la información básica y representativa de un defecto, añadiendo además el atributo **corrected**.

En el extracto de código anterior se establecen dichas propiedades básicas y, posteriormente, invocamos la función de corrección siempre y cuando se haya definido para esa regla de usabilidad. En el análisis de casos aparecen algunos conceptos que explicaremos a continuación, como la petición de confirmación por parte del usuario. Por ahora, basta comprender que la corrección sólo es invocada cuando se contempla para la regla de usabilidad y cuando el usuario así lo desea. En caso contrario, la corrección no se realiza y el atributo **corrected** queda con valor **false**.

La generación de reglas de transformación es por tanto muy similar a la realizada para el análisis, tal y como comentábamos. La creación del defecto se simplifica al mismo tiempo que añadimos la corrección, pero la estructura básica y la mayor parte del contenido de la regla permanece intacto, manteniendo las características estudiadas en el apartado anterior y ejemplificadas a través de diagramas como los de la Figura 9.6 y la Figura 9.8.

No obstante, incluiremos también aquí un único ejemplo de generación, para el que escogemos una vez más la regla de usabilidad 'Separación Etiqueta-Campo de entrada', estudiada con anterioridad.

Esta es la regla de transformación generada para la corrección:

```

operation Label_input_separation_restriction(e : UsabilityGUI!Input) {
    return e.label.isDefined();
}

operation Label_input_separation_calculation(e : UsabilityGUI!Input)
    : Integer {
    return (e.box.position.x-(e.label.box.position.x+e.label.box.dimension.x))
        .max(e.box.position.y-(e.label.box.position.y+e.label.box.dimension.y));
}

operation Label_input_separation_correction(e : UsabilityGUI!Input) {
    var horizontal =
        e.box.position.x-(e.label.box.position.x+e.label.box.dimension.x);
    var vertical =
        e.box.position.y-(e.label.box.position.y+e.label.box.dimension.y);

    if (horizontal > vertical) {
        e.label.box.position.x = e.label.box.position.x + (horizontal - 10);
    }
}

```

9. Análisis y corrección de defectos

```
        e.label.guiElement.position.x = e.label.box.position.x;
    }
    else {
        e.label.box.position.y = e.label.box.position.y + (vertical - 10);
        e.label.guiElement.position.y = e.label.box.position.y;
    }
}

@greedy
rule Label_input_separation
    transform e : UsabilityGUI!Input
        to defect : CorrectionDefects!Defect {

        guard : Label_input_separation_restriction(e) and
            not(Label_input_separation_calculation(e) >= 0
                and Label_input_separation_calculation(e) <= 10)

        defect.description = "The label is too far from its input field";
        defect.usabilityRule = "Label input separation";
        defect.element = e.guiElement.name;

        if (autocorrect or System.user.confirm(e.guiElement.name + ": The label is too far
            from its input field. Do you want to correct this defect?", true)) {
            Label_input_separation_correction(e);
            defect.corrected = true;
        }
        else {
            defect.corrected = false;
        }

        evaluation.defects.add(defect);
    }
}
```

Se sigue aplicando un proceso similar al de la Figura 9.8, con las diferencias que ya hemos comentado. Se genera la función de corrección que, como ocurre en la mayoría de ocasiones, es algo más compleja que las de restricción o cálculo. Después, la transformación M2T ha generado un análisis de casos en el que interviene la decisión del usuario a la hora de corregir o no el elemento en cuestión.

Veamos ahora en qué consiste esta interacción con el usuario durante el proceso de corrección. Si consultamos el fichero de transformación M2M, o bien la transformación M2T que lo genera, comprobaremos una vez más la presencia de los bloques de código **pre** y **post** así como las funciones auxiliares que ya explicamos en el apartado anterior.

En el bloque de código **post**, que se ejecuta una vez finalizada la transformación, mostramos de nuevo un mensaje informando de la terminación del proceso. El bloque **pre**, por el contrario, es un poco más complejo que su equivalente en el análisis. Su contenido es el siguiente:

```
pre createEvaluation {
    var evaluation = new CorrectionDefects!Evaluation;
    evaluation.guiName = UsabilityGUI!GUI.allInstances().first().guiElement.name;

    var autocorrect = true;
    var choices =
        Sequence{"Ask me what to correct", "Automatically correct everything"};
    var choice =
        System.user.choose("¿What do you want me to do?", choices, choices.get(0));
}
```

```

switch (choice) {
    case choices.get(0) : autocorrect = false;
}
}

```

Se sigue creando el elemento raíz de tipo **Evaluation**, pero se incluyen algunas sentencias que son las que engloban toda la funcionalidad de consulta al usuario. Básicamente, se ofrecen al usuario dos opciones y se almacena la elegida, pues guiará todo el proceso de corrección. Las alternativas entre las que puede escoger el usuario son:

- “Ask me what to correct”: se preguntará al usuario, para cada defecto de usabilidad encontrado, si desea corregirlo o no.
- “Automatically correct everything”: se corregirán automáticamente todos los errores de usabilidad para los que se haya definido una función de corrección.

Para este proceso hacemos uso del mecanismo que nos ofrece EOL y que mostrará al usuario un diálogo como el que vemos en la Figura 9.12, mostrando ambas opciones en una lista.

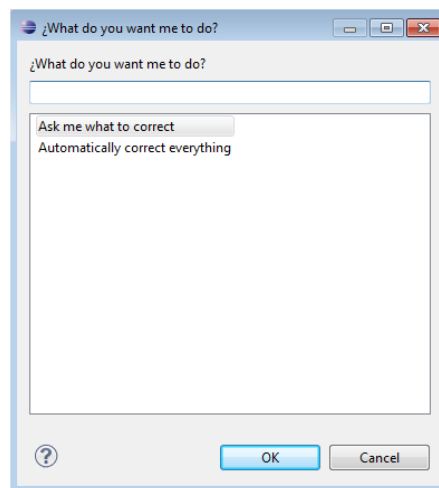


Figura 9.12.: Diálogo de entrada de datos por parte del usuario durante el proceso de corrección.

Este es el motivo de que la transformación M2T genere, para cada regla de transformación M2M, la comprobación que veíamos antes:

```

if (autocorrect or System.user.confirm(e.guiElement.name +
    ": Too many distinct font sizes. Do you want to correct this defect?", true))

```

Que será cierta si el usuario ha decidido que todos los errores se corrijan automáticamente o bien si confirma, en una nueva ventana de diálogo, que quiere que este defecto en concreto se arregle.

9. Análisis y corrección de defectos

La ejecución de la transformación modelo a modelo tiene como resultado un segundo modelo de defectos que conforma al mismo metamodelo pero que, como hemos visto, tiene menos información que el generado por el proceso de análisis. En la Figura 9.13 vemos el modelo de defectos obtenido tras la corrección de una interfaz de ejemplo.

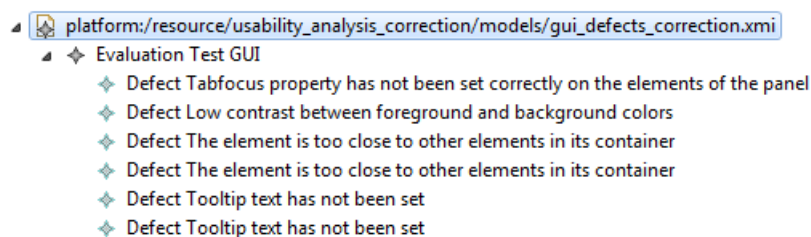


Figura 9.13.: Modelo de defectos generado durante la corrección de la usabilidad de una interfaz.

Es igual que el que mostramos en la Figura 9.9 pues los defectos hallados son los mismos. Pero si echamos un vistazo a las propiedades de uno de ellos (consultar Figura 9.14), vemos que algunos campos no han sido establecidos (otros, como la importancia, se establecen de forma automática) y que el campo **corrected** nos informa de si el defecto se ha corregido o no.

Property	Value
Corrected	false
Correct Value	
Description	Low contrast between foreground and background colors
Element	formButton
Importance	Low
Usability Rule	Contrast
Value	

Figura 9.14.: Atributos de uno de los defectos de usabilidad detectados.

El informe generado para la corrección mostrará de forma más clara cuál es la información que se tiene en cuenta para cada defecto en este punto del sistema. Su generación ya se expuso en la Sección 9.1.1.2. Un ejemplo más completo sobre una interfaz un poco mayor podrá verse en la Sección 10; se mostrarán tanto el informe de análisis como el de corrección, generados a partir de los modelos de defectos que se obtienen con la ejecución de las transformaciones modelo a modelo que acabamos de estudiar.

10 Validación

En este apartado se validará el sistema de análisis y corrección de la usabilidad mediante su aplicación sobre una sencilla interfaz de ejemplo. Se presentará primero dicha interfaz, mostrando sus características principales y los atributos que la hacen interesante como caso de prueba, así como el método que hemos seguido para su construcción. Después, se presentarán los resultados del análisis y corrección de la usabilidad de la interfaz, mostrando los modelos generados y los informes presentados al usuario. Para terminar, analizaremos el desempeño del sistema a través de los resultados obtenidos y discutiremos brevemente sobre la arquitectura propuesta basándonos en el caso de prueba ejecutado.

10.1 Caso de estudio

Para componer un caso de estudio a través del cual analizar el comportamiento de nuestro sistema, necesitamos una interfaz de usuario sobre la que realizar los procesos de análisis y corrección de la usabilidad.

Hemos optado por configurar nosotros mismos dicha interfaz, siguiendo para su creación los siguientes criterios:

- Debe ser sencilla, es decir, no incluir muchas ventanas ni elementos.
- Debe incluir componentes GUI de uso común.
- Debe poder contener los defectos de usabilidad más comunes de una interfaz que han sido incluidos en nuestro catálogo.

Para el desarrollo de la interfaz de usuario hemos utilizado Qt, que cuenta con una potente herramienta para el diseño de interfaces. La creación de interfaces de usuario con Qt es muy sencilla, pues dispone de una completa paleta de componentes GUI personalizables según un amplio conjunto de propiedades.

Respondiendo a la necesidad de simplicidad expuesta anteriormente, decidimos crear una interfaz de usuario formada por una única ventana. Esta ventana contará con un pequeño panel que contiene un formulario de datos y con un segundo panel, de tipo **Frame**, con una serie de imágenes en su interior. Además, incluye un sencillo menú (no operativo)

10. Validación

con el fin de ilustrar las reglas de usabilidad relativas a este tipo de componente. La apariencia de la interfaz puede consultarse en la Figura 10.1, tal y como es previsualizada por la propia herramienta Qt de diseño.

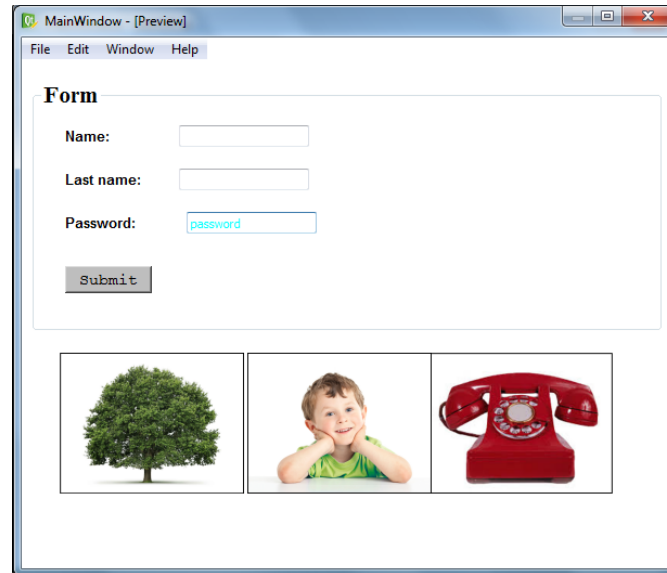


Figura 10.1.: Interfaz ejemplo para el proceso de validación.

La interfaz cuenta por tanto con un formulario en que se usan componentes GUI tan comunes como etiquetas, campos de entrada y botones, y con un contenedor con imágenes que nos permitirá estudiar algunas reglas de usabilidad relacionadas con la posición.

Si consultamos la Figura 10.1 podemos comprobar cómo, a simple vista, la interfaz creada para este caso de prueba contiene algunos defectos de usabilidad. Se puede apreciar fácilmente cómo los campos de entrada no están alineados o el solapamiento existente entre dos de las imágenes. Recordemos que nuestro objetivo en este apartado es el de mostrar con un ejemplo los procesos de análisis y corrección de la usabilidad y en consecuencia debemos partir de una interfaz de usuario defectuosa. Con la intención de que el caso de estudio sea lo más ilustrativo posible, decidimos incluir una cantidad elevada de defectos de usabilidad en la interfaz, abarcando una gran parte de las reglas de usabilidad expuestas en la Sección 5.

El listado completo de defectos de usabilidad que hemos incluido en nuestra interfaz es el siguiente:

- Se utilizan tres tamaños y tipos de fuente en total, para el título del formulario, las etiquetas y el botón, siendo el límite correcto en ambos casos dos.
- La propiedad `tabfocus` se establece sobre los campos de entrada y el botón siguiendo este orden: etiqueta 'Name', etiqueta 'Password', etiqueta 'Last name' y botón.

- El color de fondo del botón es #BEBEBE en su representación hexadecimal, color que no está definido en la paleta.
- El color frontal del campo de entrada relativo a la contraseña es un azul celeste, siendo el color de fondo blanco, lo que ocasiona un bajo contraste que hace difícil la lectura.
- La primera y segunda imagen están separadas únicamente por tres píxeles.
- La segunda y tercera imagen están solapadas.
- La etiqueta 'Password' está demasiado separada de su campo de entrada, a una distancia de 32px.
- El campo de entrada correspondiente a 'Password' no se encuentra correctamente alineado con los otros dos campos de entrada.
- El menú tiene cuatro niveles de profundidad, siendo el máximo tres.
- El texto tooltip no ha sido establecido para el campo de entrada correspondiente a 'Last name'.
- La información de accesibilidad no ha sido establecida para la segunda imagen.

En la Figura 10.2 puede verse de nuevo la interfaz de usuario de la que partiremos, con todos y cada uno de los defectos de usabilidad que acabamos de ver indicados sobre los propios componentes GUI, pues muchos de ellos no son apreciables directamente.

Se trata de un conjunto más que suficiente de defectos de usabilidad que nos permitirá poner a prueba nuestro sistema de análisis y corrección comprobando su desempeño sobre un caso de estudio real con defectos de usabilidad variados.

Qt nos facilita la creación de una interfaz, pero la interfaz que obtenemos de este modo no puede emplearse directamente como entrada de nuestro sistema. Es necesario un paso adicional en el que se obtiene, a partir de la interfaz Qt, el modelo que la representa conforme al metamodelo GUI presentado en la Sección 7.1.

Esta inyección del modelo GUI se realizaría en dos pasos. Primero se utilizaría el inyector proporcionado por EMF para convertir un documento XML en un modelo conforme a un metamodelo que corresponde al esquema de XML. A continuación se ejecutaría una transformación M2M para convertir este modelo en el modelo GUI. Como explicamos en la Sección 5 esta inyección no ha sido abordada en este trabajo y es parte de otro trabajo fin de grado. Por tanto, ha sido preciso crear manualmente el modelo GUI del ejemplo utilizando el editor de árbol de EMF que se genera para un metamodelo dado, como se muestra en las Figuras 10.3-10.6.

Nuestro metamodelo GUI es lo suficientemente intuitivo como para que este proceso pueda hacerse de una manera sencilla y comprensible. La estructura de contenedores, widgets y propiedades es fácilmente adaptable a nuestra interfaz y, a partir de su representación gráfica, podemos ir creando el modelo GUI correspondiente partiendo de

10. Validación

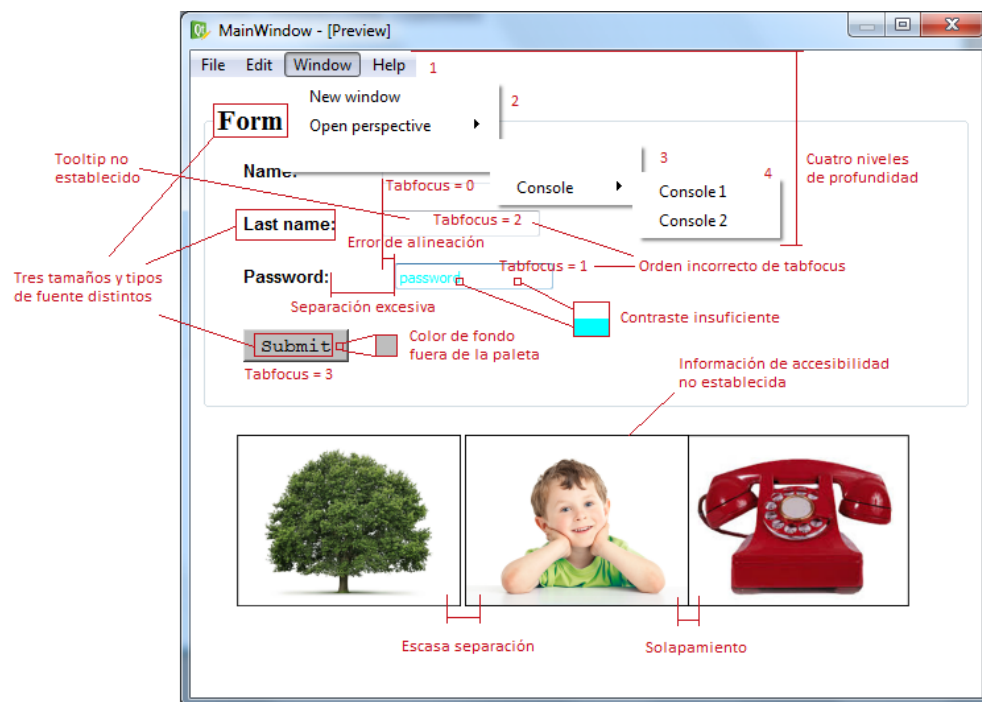


Figura 10.2.: Indicación de los errores de usabilidad de la interfaz de nuestro caso de estudio.

los contenedores y luego creando su contenido. Comenzaremos, por ejemplo, creando la ventana y sus propiedades, para luego crear los componentes que la ventana contiene directamente: el menú y los dos contenedores (Panel y Frame). Este primer paso de la creación del modelo GUI puede verse en la Figura 10.3.

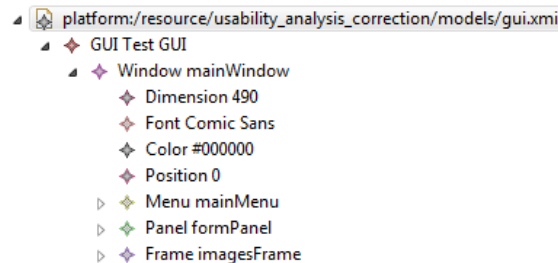


Figura 10.3.: Modelo GUI correspondiente a la interfaz. Contenido de la ventana principal.

A su vez, el menú y los contenedores que vemos en la Figura 10.3 tienen definidas sus propiedades así como los widgets o ítems de menú que los componen.

En la Figura 10.4 vemos la estructura completa del menú.

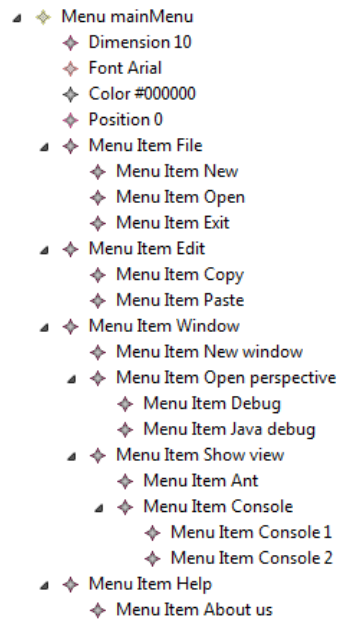


Figura 10.4.: Representación del menú según el metamodelo GUI.

En la Figura 10.5 puede verse la estructura de componentes GUI que contiene el panel con el formulario, así como las propiedades de uno de esos componentes (el campo de entrada *nameInput*). Cada componente contiene además, como atributos (no apreciables en la jerarquía de componentes de las figuras): la información de accesibilidad, el tooltip,

10. Validación

el nombre del elemento y una referencia a su contenedor, además de otros atributos opcionales como el texto que dependerán del tipo de componente.

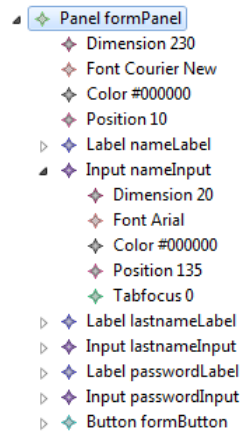


Figura 10.5.: Representación del panel con el formulario según el metamodelo GUI.

Por último, en la Figura 10.6 se puede apreciar la estructura jerárquica que componen el Frame y las imágenes que contiene.

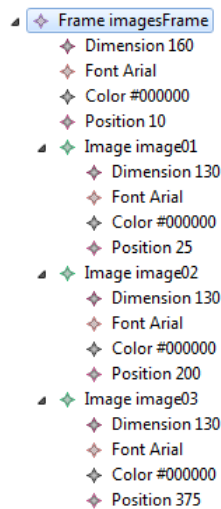


Figura 10.6.: Representación del marco con las imágenes según el metamodelo GUI.

El modelo GUI completo correspondiente a la interfaz Qt de la Figura 10.1 puede consultarse en los ficheros fuente del proyecto, en el archivo gui.xmi dentro de la carpeta *models*.

Consultando el modelo con el editor de modelos de Eclipse, podrán verse con detalle las propiedades y atributos de cada elemento, comprobando cómo han sido representados todos y cada uno de los defectos de usabilidad que veíamos en la Figura 10.2.

En la Figura 10.7, por ejemplo, podemos ver la propiedad **Color** del campo de entrada de la contraseña, donde se aprecian los valores establecidos para el color de fondo (blanco) y el color de frente (azul celeste), que resultan en un bajo contraste. De la misma forma, la Figura 10.8 muestra los atributos de la segunda imagen del **Frame**, donde podemos comprobar cómo la información de accesibilidad no ha sido establecida. Por último, la Figura 10.9 muestra la propiedad **Color** del botón del formulario, cuyo color de fondo no se encuentra entre los definidos en la paleta de colores. Estos defectos y el resto de defectos de usabilidad expuestos anteriormente y que podemos ver en la Figura 10.2 son los que nuestro sistema deberá detectar y, si es posible, corregir.

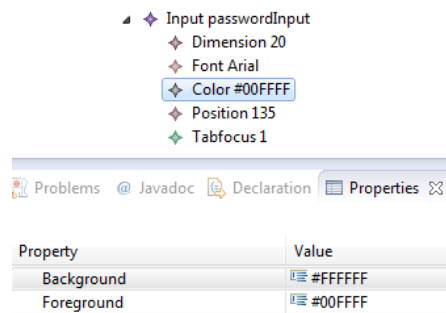


Figura 10.7.: Propiedad **Color** del componente GUI *passwordInput*.

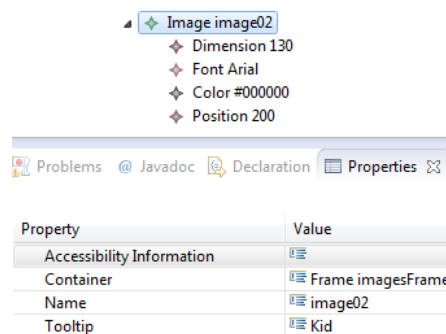


Figura 10.8.: Atributos del componente GUI *image02*.

Ejecutaremos nuestros procesos de análisis y corrección de la usabilidad sobre el modelo GUI generado a partir de la interfaz de ejemplo. Dicho modelo será establecido como modelo de entrada estableciendo su ruta dentro del proyecto en los ficheros Ant que lanzan los procesos de análisis y corrección.

El sistema deberá detectar todos y cada uno de los defectos que se deducen de los errores de usabilidad listados anteriormente y, una vez detectados, lanzaremos nuestro proceso de corrección, arreglando todos los defectos que tengan asociado un código de corrección.

Estudiaremos el resultado analizando cómo se han corregido los defectos de usabilidad

10. Validación

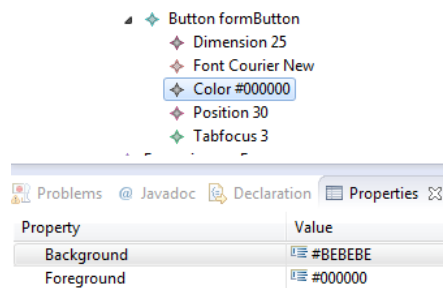


Figura 10.9.: Propiedad Color del componente GUI *formButton*.

y si estos se han eliminado completamente y de forma satisfactoria.

Analizaremos también los defectos que no puedan ser corregidos automáticamente y que requieren por tanto de una corrección manual.

10.2 Resultados

Una vez hemos construido el modelo GUI con nuestra interfaz de ejemplo y disponiendo del modelo de reglas expuesto en la Sección 8 estamos en condiciones de ejecutar los procesos de análisis y corrección, lanzando los ficheros Ant descritos en el Anexo A.

El resultado de la ejecución del análisis de la usabilidad es un modelo de defectos y su correspondiente informe en formato HTML. En la Figura 10.10 podemos ver el modelo completo de defectos de usabilidad resultado del análisis.

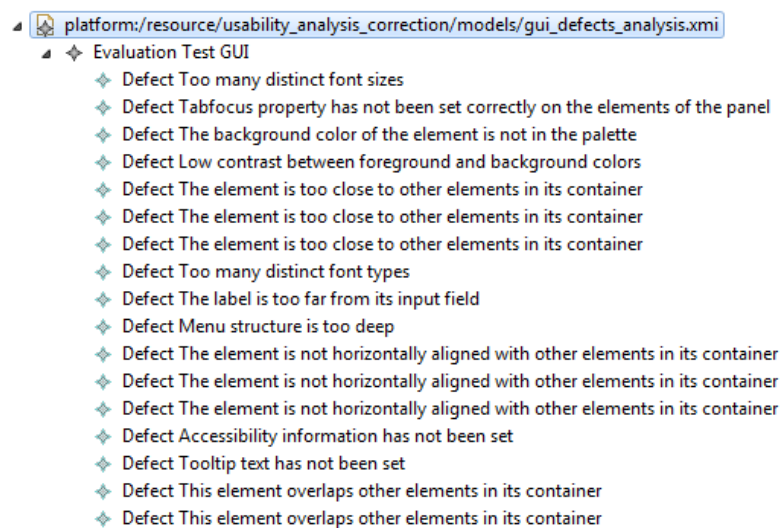


Figura 10.10.: Modelo de defectos resultado del análisis de la usabilidad.

Los defectos aparecen listados junto a su descripción, aunque no se muestran el resto

de atributos. Por esta razón, a partir de este punto mostraremos el resultado de los procesos de análisis y corrección a través de los informes generados, pues al fin y al cabo esa es su función dentro del sistema.

El informe generado a partir del proceso de análisis de la usabilidad es el que podemos ver en la Figura 10.11. En este informe aparecen representados todos los defectos de usabilidad encontrados en nuestra interfaz, que son los mismos defectos que acabamos de ver en el modelo de defectos de la Figura 10.10.

Analizando los defectos uno por uno es posible comprobar cómo todas las situaciones expuestas en el apartado anterior han sido detectadas y aparecen en el informe. En el caso de los defectos de usabilidad que involucran a más de un elemento, se detecta un error por cada uno de los elementos involucrados. Así, se detecta que los campos de entrada para nombre y apellido no están alineados con el de la contraseña y viceversa, originando en total tres defectos de usabilidad. Para la segunda y tercera imagen, entre las que existe solapamiento, también se detecta un error de separación pues, al estar superpuestas, la separación entre ambas está por debajo del límite y no se cumple la correspondiente regla de usabilidad.

Los atributos de cada defecto son más que suficientes para entender perfectamente qué es lo que ocurre en cada uno de los casos y por qué se ha considerado un fallo de usabilidad, mientras que la importancia nos permite establecer una jerarquía dentro de los defectos dando prioridad a unos sobre otros.

Este primer análisis de la usabilidad es la prueba más importante para determinar el correcto funcionamiento de esta parte de nuestro sistema, pues la interfaz que hemos utilizado como ejemplo cuenta con una amplia variedad de defectos que debían ser y que de hecho han sido detectados por el analizador, pero no será el único análisis que hagamos sobre la interfaz como veremos más adelante.

El siguiente paso una vez detectados todos los defectos de usabilidad es su corrección. Para ello ejecutamos el fichero Ant que lanza este proceso, y seleccionamos la opción 'Automatically correct everything' para que se corrijan automáticamente todos los defectos que el sistema pueda corregir.

Como existen determinados defectos de usabilidad para los que no se ha definido un proceso de corrección, no esperamos que tras este paso se eliminen todos los defectos encontrados en el análisis, pero sí una gran parte de ellos.

El informe generado tras la ejecución del proceso de corrección puede verse en la Figura 10.12 El sistema de corrección de la usabilidad realiza un rápido análisis, detectando los defectos también extraídos durante el análisis, y procede a la corrección de cada uno de ellos de forma automática (pues así se lo hemos indicado al comienzo de la ejecución). En el informe de corrección aparecen todos y cada uno de los defectos de usabilidad de nuestra interfaz, marcando los que han sido corregidos y los que no.

La mayoría de defectos de usabilidad han sido corregidos, pues las reglas de usabilidad correspondientes contemplaban esta parte del proceso, pero otros han quedado sin corregir y requerirán de un arreglo manual. Consultando el modelo GUI correspondiente a la interfaz podemos observar cómo los elementos defectuosos han sido corregidos, modificándose sus propiedades y atributos para ajustarse a los límites establecidos por

10. Validación

Usability analysis

Target GUI: Test GUI

Some usability defects have been detected

Importance	Usability rule	Description	Element	Correct value	Current value
Medium	Font sizes	Too many distinct font sizes	Test GUI	1–2 font sizes	3 font sizes
Medium	Tabfocus	Tabfocus property has not been set correctly on the elements of the panel	formPanel	"	The element lastnameInput is higher than its predecessor.
High	Background color palette	The background color of the element is not in the palette	formButton	'#000000','#FF0000','00FF00','0000FF','FFFF00','00FFFF','FF00FF','FFFFFF','E1E1E1'	#8E8E8E
High	Contrast	Low contrast between foreground and background colors	passwordInput	4,5–*	1.2538810511450746
High	Separation	The element is too close to other elements in its container	image01	"	image02
High	Separation	The element is too close to other elements in its container	image02	"	image01,image03
High	Separation	The element is too close to other elements in its container	image03	"	image02
High	Font types	Too many distinct font types	Test GUI	1–2 font types	3 font types
Medium	Label input separation	The label is too far from its input field	passwordInput	0–30 px	32 px
Medium	Menu depth	Menu structure is too deep	mainMenu	1–3 levels	4 levels
High	Horizontal align	The element is not horizontally aligned with other elements in its container	nameInput	"	passwordInput
High	Horizontal align	The element is not horizontally aligned with other elements in its container	lastnameInput	"	passwordInput
High	Horizontal align	The element is not horizontally aligned with other elements in its container	passwordInput	"	nameInput,lastnameInput
Medium	Accessibility information	Accessibility information has not been set	image02	true	false
Low	Tooltip	Tooltip text has not been set	lastnameInput	true	false
High	Overlap	This element overlaps other elements in its container	image02	"	image03
High	Overlap	This element overlaps other elements in its container	image03	"	image02

Figura 10.11.: Informe generado para el análisis de la usabilidad.

Usability correction

Target GUI: Test GUI

Results of the usability correction process

Usability rule	Description	Element	State
Font sizes	Too many distinct font sizes	Test GUI	Corrected
Tabfocus	Tabfocus property has not been set correctly on the elements of the panel	formPanel	Not corrected
Background color palette	The background color of the element is not in the palette	formButton	Corrected
Contrast	Low contrast between foreground and background colors	passwordInput	Not corrected
Separation	The element is too close to other elements in its container	image01	Corrected
Separation	The element is too close to other elements in its container	image02	Corrected
Separation	The element is too close to other elements in its container	image03	Corrected
Font types	Too many distinct font types	Test GUI	Corrected
Label input separation	The label is too far from its input field	passwordInput	Corrected
Menu depth	Menu structure is too deep	mainMenu	Not corrected
Horizontal align	The element is not horizontally aligned with other elements in its container	nameInput	Corrected
Horizontal align	The element is not horizontally aligned with other elements in its container	lastNameInput	Corrected
Horizontal align	The element is not horizontally aligned with other elements in its container	passwordInput	Corrected
Accessibility information	Accessibility information has not been set	image02	Not corrected
Tooltip	Tooltip text has not been set	lastNameInput	Not corrected
Overlap	This element overlaps other elements in its container	image02	Corrected
Overlap	This element overlaps other elements in its container	image03	Corrected

Figura 10.12.: Informe generado para la corrección de la usabilidad.

10. Validación

las reglas de usabilidad.

En la Figura 10.13 podemos ver, a modo de ejemplo, la propiedad Color del botón del formulario, uno de los errores de usabilidad de la interfaz. La propiedad, cuyo estado inicial veíamos en la Figura 10.9, ha sido modificada de tal forma que su color de fondo, cuyo antiguo valor #BEBEBE no pertenecía a la paleta, se ha convertido en #E1E1E1, valor que sí forma parte de la paleta de colores y que corresponde a una tonalidad de gris, el color más parecido al que tenía originalmente el botón.

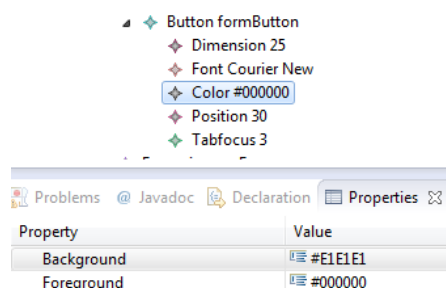


Figura 10.13.: Propiedad Color del componente GUI *formButton* tras la corrección.

Lo mismo ocurre con el resto de componentes, aunque el estudio de cómo se han modificado cada una de las propiedades y atributos se extendería demasiado. En su lugar, mostramos en la Figura 10.14 la representación de la interfaz tal y como queda después del proceso de corrección.

A simple vista, todos los defectos de usabilidad han sido corregidos. En la interfaz puede apreciarse cómo el tamaño y fuente del texto correspondiente al título del panel han sido corregidos adoptando las características del texto del botón, y limitando así el número total de tipografías y tamaños de fuente usados en la interfaz. También puede apreciarse cómo se ha corregido el color de fondo del botón, la separación y solapamiento entre las imágenes y la alineación entre los campos de entrada. No obstante, como veíamos en el informe de corrección, no todos los defectos de usabilidad han sido corregidos, pues algunos que no podemos apreciar directamente en la interfaz siguen existiendo. Además, aunque en la Figura 10.14 no se distingue con claridad, el proceso de corrección ha originado un nuevo defecto de usabilidad en la interfaz: la etiqueta correspondiente a la contraseña no se encuentra alineada con las otras dos etiquetas.

Un nuevo análisis de la usabilidad nos aclara los defectos que han sido corregidos y en consecuencia ya no aparecen en el informe, los defectos que permanecen y los que han surgido.

En la Figura 10.15 vemos el resultado de este segundo proceso de análisis.

Obtenemos los defectos de usabilidad que ya detectamos inicialmente y no fueron corregidos, pero también aparecen una serie de defectos nuevos que nos comunican una mala alineación entre una de las etiquetas y el resto (también interviene el botón, que tiene la misma alineación que las etiquetas).

Por lo demás, los defectos de usabilidad que obtuvimos en el primer informe y que corregimos no vuelven a aparecer, como era de esperar.

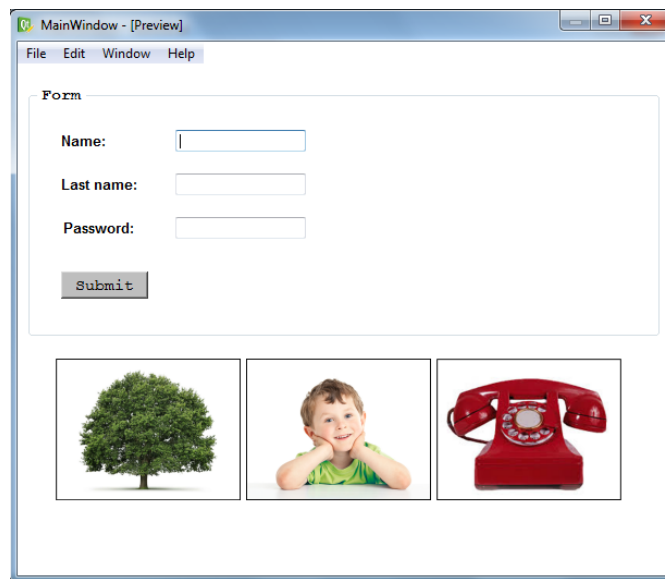


Figura 10.14.: Aspecto de la interfaz tras la primera fase de corrección.

Usability analysis

Target GUI: Test GUI

Some usability defects have been detected

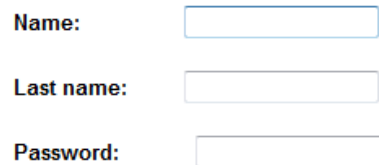
Importance	Usability rule	Description	Element	Correct value	Current value
Medium	Tabfocus	Tabfocus property has not been set correctly on the elements of the panel	formPanel	"	The element lastnameInput is higher than its predecessor.
High	Contrast	Low contrast between foreground and background colors	passwordInput	4.5~*	1.2538810511450746
Medium	Menu depth	Menu structure is too deep	mainMenu	1-3 levels	4 levels
High	Horizontal align	The element is not horizontally aligned with other elements in its container	nameLabel	"	passwordLabel
High	Horizontal align	The element is not horizontally aligned with other elements in its container	lastnameLabel	"	passwordLabel
High	Horizontal align	The element is not horizontally aligned with other elements in its container	passwordLabel	"	nameLabel,lastnameLabel,formButton
High	Horizontal align	The element is not horizontally aligned with other elements in its container	formButton	"	passwordLabel
Medium	Accessibility information	Accessibility information has not been set	image02	true	false
Low	Tooltip	Tooltip text has not been set	lastnameInput	true	false

Figura 10.15.: Informe del análisis de la usabilidad tras la primera corrección.

10. Validación

Estudiemos el proceso que ha dado lugar a la aparición del nuevo defecto de usabilidad.

En la Figura 10.16 vemos la situación de la que partíamos: un fallo de alineación entre los campos de entrada y una separación excesiva entre la etiqueta 'Password' y su campo de entrada.



Formulario de inicio con tres filas. La primera fila tiene la etiqueta 'Name:' a la izquierda y un campo de entrada a la derecha. La segunda fila tiene la etiqueta 'Last name:' a la izquierda y un campo de entrada a la derecha. La tercera fila tiene la etiqueta 'Password:' a la izquierda y un campo de entrada a la derecha. Hay un espacio excesivo entre la etiqueta 'Password' y su campo de entrada.

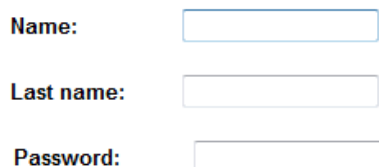
Figura 10.16.: Situación inicial de las etiquetas y campos de entrada del formulario.

El análisis de la usabilidad detectó los dos defectos existentes, y durante la corrección el sistema trató de solucionarlos.

El primer defecto de usabilidad detectado (por el orden en que han sido definidas las reglas de usabilidad) es el relativo a la distancia entre la etiqueta y su campo de entrada.

Para corregirlo, la etiqueta se desplaza acercándola 2px a su campo de entrada para que la distancia entre ambos, originalmente 32 px, se encuentre dentro del límite establecido de 30px.

La situación de ambos elementos tras la corrección de este defecto es la que vemos en la Figura 10.17, donde la etiqueta ha sido ligeramente desplazada.



Formulario de inicio con tres filas. La primera fila tiene la etiqueta 'Name:' a la izquierda y un campo de entrada a la derecha. La segunda fila tiene la etiqueta 'Last name:' a la izquierda y un campo de entrada a la derecha. La tercera fila tiene la etiqueta 'Password:' a la izquierda y un campo de entrada a la derecha. La etiqueta 'Name' ha sido desplazada ligeramente hacia el campo de entrada.

Figura 10.17.: Situación de las etiquetas y campos de entrada del formulario tras la corrección del primer defecto de usabilidad.

A continuación, y dentro del mismo proceso global de corrección, se procede al arreglo del segundo defecto de usabilidad: el relativo a la alineación de los campos de entrada.

Para ello se desplaza el campo de entrada mal alineado 7px en la dirección correcta. Los campos de entrada quedan correctamente alineados, tal y como vemos en la Figura 10.18, pero la etiqueta permanece desplazada y por tanto mal alineada con respecto al resto de etiquetas.

Esta la razón de que un segundo análisis de la usabilidad sobre la interfaz detecte nuevos defectos relacionados con la alineación de estos elementos del formulario.

Es necesaria por tanto una segunda corrección que elimine estos defectos, y logre que tanto etiquetas como campos de entrada queden correctamente alineados y con las distancias adecuadas.

Name:

Last name:

Password:

Figura 10.18.: Situación de las etiquetas y campos de entrada del formulario tras la corrección del segundo defecto de usabilidad.

Si lanzamos el proceso de corrección por segunda vez, obtenemos el informe que vemos en la Figura 10.19.

Usability correction
Target GUI: Test GUI
 Results of the usability correction process

Usability rule	Description	Element	State
Tabfocus	Tabfocus property has not been set correctly on the elements of the panel	formPanel	Not corrected
Contrast	Low contrast between foreground and background colors	passwordInput	Not corrected
Menu depth	Menu structure is too deep	mainMenu	Not corrected
Horizontal align	The element is not horizontally aligned with other elements in its container	nameLabel	Corrected
Horizontal align	The element is not horizontally aligned with other elements in its container	lastnameLabel	Corrected
Horizontal align	The element is not horizontally aligned with other elements in its container	passwordLabel	Corrected
Horizontal align	The element is not horizontally aligned with other elements in its container	formButton	Corrected
Accessibility information	Accessibility information has not been set	image02	Not corrected
Tooltip	Tooltip text has not been set	lastnameInput	Not corrected

Figura 10.19.: Informe para la segunda corrección de la usabilidad de la interfaz.

Los defectos cuyas reglas de usabilidad no tienen asociado un código de corrección siguen sin ser corregidos, pero la mala alineación entre las etiquetas se soluciona.

El resultado es una interfaz sin errores visibles de usabilidad, tal y como podemos ver en la Figura 10.20.

La alineación de los elementos del formulario ha sido corregida, y ya sólo quedan defectos de usabilidad que han de ser corregidos manualmente.

Un último análisis de la usabilidad nos confirma que, de entre los muchos defectos que contenía inicialmente la interfaz, solo restan aquellos que no han podido ser corregidos de forma automática por nuestro sistema (ver Figura 10.21).

10. Validación

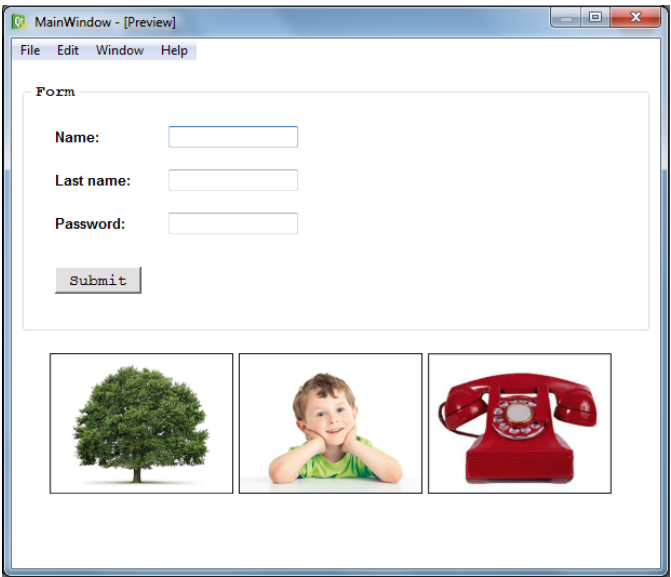


Figura 10.20.: Aspecto de la interfaz tras completar la corrección de la usabilidad.

Usability analysis					
Target GUI: Test GUI					
Some usability defects have been detected					
Importance	Usability rule	Description	Element	Correct value	Current value
Medium	Tabfocus	Tabfocus property has not been set correctly on the elements of the panel	formPanel	..	The element lastnameInput is higher than its predecessor.
High	Contrast	Low contrast between foreground and background colors	passwordInput	4.5-*	1.2538810511450746
Medium	Menu depth	Menu structure is too deep	mainMenu	1-3 levels	4 levels
Medium	Accessibility information	Accessibility information has not been set	image02	true	false
Low	Tooltip	Tooltip text has not been set	lastnameInput	true	false

Figura 10.21.: Informe para el análisis de la usabilidad tras completar la corrección.

10.3 Discusión

El caso de prueba que hemos estudiado a lo largo de este apartado es un sencillo pero efectivo ejemplo que nos permite apreciar cómo funcionan los procesos de análisis y corrección de la usabilidad definidos en nuestro sistema.

El amplio rango de reglas de usabilidad definido nos permite contemplar la mayoría de defectos de usabilidad que se dan comúnmente en las interfaces de usuario, muchos de los cuales se han mostrado a lo largo de este ejemplo: alineación y separación entre elementos, colores y contraste, tipografías, etc.

El sistema de análisis de la usabilidad ha probado ser lo suficientemente completo como para detectar todas estas situaciones, informando de ellas al usuario a través de un completo listado donde se muestran las características de cada defecto de usabilidad. No obstante, somos conscientes de la necesidad de realizar pruebas con más ejemplos de GUIs para completar la validación.

El sistema de corrección de la usabilidad se ha mostrado capaz de corregir eficientemente la mayoría de defectos de usabilidad detectados. No obstante, hemos podido comprobar cómo la corrección de alguno de estos defectos ha ocasionado, a su vez, la creación de otros diferentes. Esta situación es común cuando tratamos con defectos de usabilidad relacionados con la posición de los elementos, pues las reglas de usabilidad correspondientes están estrechamente relacionadas. En nuestro caso de estudio pudimos comprobar cómo la corrección de la separación entre dos elementos provocó un defecto en la alineación de uno de ellos. En cualquier caso, una segunda corrección de la usabilidad fue capaz de arreglar esta situación y eliminar completamente los defectos de usabilidad relacionados con la posición de etiquetas y campos de entrada.

La corrección fue satisfactoria para la sencilla interfaz utilizada como caso de prueba, pero es importante ser consciente de esta posible situación pues en interfaces más complejas los efectos en cadena pueden ser más complicados y, en ocasiones, puede ser más interesante una corrección manual por parte del usuario.

Otros defectos sí que requieren forzosamente de una corrección manual, pues no se ha definido el proceso de corrección correspondiente.

El último análisis de la usabilidad realizado sobre la interfaz mostraba todos estos errores para cuya corrección nuestro sistema no está capacitado. Entre ellos, contamos con defectos de usabilidad como la ausencia de información de accesibilidad o tooltip en un elemento o la profundidad de un menú.

En algunos casos, la corrección automática de estos defectos no es posible, como a la hora de establecer una información de accesibilidad que describa el contenido de una imagen. En otros casos, el proceso de corrección de un defecto es demasiado complejo o no garantiza un buen resultado, con lo que se prefiere dejar dicha corrección en manos del usuario.

Para eliminar los defectos de usabilidad, será preciso combinar las correcciones automáticas con las correcciones manuales realizadas por el usuario.

El objetivo de nuestro sistema es corregir de forma automática el mayor número de defectos posible y ofrecer un análisis completo y claro al usuario para que éste pueda

10. Validación

finalizar el proceso y eliminar por completo todo defecto de usabilidad de la interfaz. Con ello se reduce el esfuerzo del proceso y se mejora la calidad, el usuario se ve liberado de algunas correcciones y el sistema realiza un análisis automático que encuentra los defectos de usabilidad.

11 Conclusiones y vías futuras

En este trabajo se ha presentado un sistema para el análisis y la corrección automática de la usabilidad desarrollado siguiendo las directrices de MDE y caracterizado por la independencia de la tecnología subyacente y la extensibilidad. Se ha desarrollado una arquitectura basada en modelos y transformaciones M2M y M2T para la aplicación de un catálogo propio de reglas de usabilidad, y se han lanzado los procesos de análisis y corrección sobre una GUI de ejemplo con algunos de los defectos de usabilidad más comunes.

El beneficio concreto en cuanto a ahorro de tiempo de la solución propuesta dependerá del número de defectos considerados en el catálogo y no se ha hecho ninguna medición para la solución actual dado que el objetivo era realizar una prueba de concepto con un número limitado de defectos. No obstante, es evidente el beneficio de la automatización de la evaluación de usabilidad y en cuanto al uso de modelos y transformaciones de modelos frente a una solución tradicional los beneficios son también bien conocidos [12].

Como trabajo futuro queda completar la arquitectura propuesta a través de la inyección de interfaces concretas, en base a una tecnología de especificación de GUIs como podría ser Qt. Esto supondría la implementación de una transformación M2M adicional que nos permita obtener la representación conforme con nuestro metamodelo GUI de una interfaz Qt dada a partir de un modelo que conforme el metamodelo correspondiente a Qt. Además, sería necesario implementar la transformación M2M inversa, que convirtiera un modelo conforme al metamodelo GUI de nuevo en su correspondiente modelo Qt, transformación necesaria para trasladar las correcciones realizadas sobre la interfaz abstracta a la interfaz concreta. Ambas transformaciones completarían un sistema que tomaría como entrada una interfaz definida en Qt y devolvería un informe con sus defectos de usabilidad y, en caso de corrección, una interfaz Qt con los defectos corregidos. Una vez implementada esta parte del sistema, existiría la posibilidad de añadir soporte a tantos tipos de interfaz como se quiera, siguiendo el mismo proceso que acabamos de describir para Qt.

Otra línea futura de trabajo sería la optimización del sistema actual, juntando los procesos de análisis y corrección en un único proceso o bien mezclando ambas funcionalidades de tal forma que el resultado del proceso de análisis de la usabilidad sea la entrada del proceso de corrección. Los procesos ya no serían completamente independientes pero la nueva estructura nos permitiría aprovechar la labor de detección de defectos realizada durante el análisis para el posterior proceso de corrección. La corrección ya no requeriría

11. Conclusiones y vías futuras

de un nuevo proceso de extracción de defectos, sino que tomaría como entrada el modelo de defectos generado por un análisis anterior. La estrecha relación existente entre ambas funciones hace que un enfoque como este parezca más razonable pues resulta en un sistema más acoplado y eficiente donde no se repite funcionalidad.

Bibliografía

- [1] *Object Constraint Language (OCL), version 2.3.1.* 2012.
- [2] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.
- [3] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. MoDisco: A Model Driven Reverse Engineering Framework. *Information & Software Technology*, 56(8):1012–1032, 2014.
- [4] Javier Luis Cánovas Izquierdo and Jesús García Molina. Extracting Models from Source Code in Software Modernization. *Software and System Modeling*, 13(2):713–734, 2014.
- [5] Jesús García Molina and José Ramón Hoyos, editors. *Apuntes de la asignatura “Desarrollo De Software Dirigido por Modelos”*. Máster de Nuevas Tecnologías Informáticas, Facultad de Informática, Universidad de Murcia.
- [6] Jesús J.García Molina et al., editors. *Desarrollo de Software Dirigido por Modelos: Conceptos, Técnicas y Herramientas*. RA-MA, 2013.
- [7] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona Polack. The Epsilon Generation Language. *ECMDA-FA*, pages 1–16, 2008.
- [8] Bran Selic. What Will it Take? A View on Adoption of Model-Based Methods in Practice. *Software and Systems Modeling*, 11(4):513–526, 2012.
- [9] Ben Shneiderman and Catherine Plaisant, editors. *Diseño de interfaces de usuario*. Pearson, 2005.
- [10] Jos Warmer and Anneke Kleppe, editors. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.
- [11] Jon Whittle, John Hutchinson, and Mark Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85, 2014.
- [12] Óscar Sánchez Ramón, Javier Bermúdez Ruiz, and Jesús García Molina. Una Valoración de la Modernización de Software Dirigida por Modelos. In *JISBD*, 2013.

A Estructura y ejecución del proyecto

Adjunto a este documento presentamos el código fuente del sistema, que se divide en los siguientes proyectos:

- `usability_analysis_correction`: proyecto principal que contiene la mayor parte del sistema.
- `usability_rules.resource.rules`, `usability_rules.resource.rules.ui` y `org.emftext.commons antlr3_4_0`: proyectos generados por EMFText para la notación textual explicada en la Sección 8.1.1.

Nos centraremos en el primero de los proyectos, `usability_analysis_correction`, estructurado en las siguientes carpetas:

- `metamodels`: contiene los metamodelos definidos para reglas, interfaz y defectos. Incluye el metamodelo de Qt, correspondiente a una parte del sistema que no se ha llegado a implementar.
- `models`: contiene tanto los modelos de entrada del sistema (reglas e interfaz) como los generados a lo largo de los procesos de análisis y/o corrección. También incluye la definición del catálogo de reglas siguiendo el DSL textual.
- `reports`: dentro de esta carpeta se generan los informes tanto de análisis como de corrección.
- `transformations`: contiene todas las transformaciones M2M y M2T utilizadas a lo largo del sistema, siendo dos de ellas (Analysis y Correction) generadas durante los procesos de análisis y corrección respectivamente.

En la raíz del proyecto se encuentran los ficheros Ant de ejecución: `analysis-launch.xml` y `correction-launch.xml`.

Ambos tienen una estructura muy similar. Para el análisis, por ejemplo, se definen las siguientes tareas Ant:

A. Estructura y ejecución del proyecto

1. **loadModels**: se cargan todos los modelos que intervienen en el proceso, tanto los de entrada como los de salida, estableciendo mediante las variables booleanas **read** y **store** un tipo u otro.
2. **gui2usability**: tarea encargada de la transformación GUI-UsabilityGUI.
3. **analysis-generation**: tarea encargada de generar la transformación M2M que realiza el análisis.
4. **analysis**: tarea encargada de realizar el análisis mediante la ejecución de la transformación M2M, generando el modelo de defectos de usabilidad.
5. **analysis-report**: tarea encargada de la generación del informe a partir del modelo de defectos.

Para la corrección se define el mismo conjunto de tareas, con la salvedad de que, en el caso del modelo GUI, a la hora de cargarlo se establecen las variables **read** y **store** ambas a **true**, pues durante la corrección el modelo es tanto entrada como salida del sistema.

Se establecen dependencias entre las diferentes tareas Ant de manera que el orden en que estas son ejecutadas sea el orden en que las hemos definido anteriormente.

Para la ejecución de cualquiera de las dos tareas desde Eclipse, hay que seguir las indicaciones que se dan al principio de ambos ficheros, consideraciones que no son propias de nuestro sistema sino que deben seguirse para la ejecución de cualquier fichero Ant que defina tareas relativas a Epsilon.

B Catálogo de reglas

En las siguientes páginas se muestra el catálogo de reglas de usabilidad en formato tabla.

NAME		FONT SIZES
DEFINITION		Checks the number of distinct font sizes within the interface
ERROR MESSAGE		Too many distinct font sizes
IMPORTANCE		Medium
ELEMENT TYPE		GUI
RESTRICTION		
HOW TO CALCULATE		Takes all instances of class Text and put its font sizes in a set (no repetition collection). The number of distinct font sizes in the application is the size of this set.
CALCULATION		<pre>return Text.allInstances().collect(t t.size).asSet().size();</pre>
RANK	TYPE	Integer
	VALUES	1-2 font sizes
HOW TO CORRECT		We iterate through the Texts collecting the different font sizes until the limit is reached. From that point on, every extra font size will be replaced with the closer one in the valid sizes collection.
CORRECTION		<pre>var maxSizes = 2; var validSizes : Set; for (t in Text.allInstances()) { if (validSizes.size() < maxSizes) { // We collect valid sizes until reaching the limit validSizes.add(t.size); } else if (not validSizes.includes(t.size)) { // We get the closest valid size var newSize = validSizes.sortBy(s (t.size- s).abs()).first(); // We set the new size t.guiElement.font.size = newSize; } }</pre>
OTHER INFORMATION		

NAME	TABFOCUS
DEFINITION	Checks if the tabfocus property has been set in a proper, consistent way on the elements inside a panel
ERROR MESSAGE	Tabfocus property has not been set correctly on the elements of the panel
IMPORTANCE	Medium
ELEMENT TYPE	Container
RESTRICTION	
HOW TO CALCULATE	We get the elements inside the panel ordered by its tabfocus value. Then try to find inconsistencies (gaps, repetitions or incorrect order).
CALCULATION	<pre> // Tabfocus in the panel sorted by its value var orderedTabfocus = e.content.select(elem elem.isTypeOf(Tabfocus) and elem.value <> - 1).sortBy(elem elem.value); var errors = ""; // If there are no elements with tabfocus property, we finish if (orderedTabfocus.size() == 0) return errors; // The first tabfocus defined must be zero if (orderedTabfocus.first().value <> 0) errors = errors + "The first tabfocus defined must be zero. "; for (i in Sequence{1..orderedTabfocus.size()-1}) { var tabfocus = orderedTabfocus.at(i); var previousTabfocus = orderedTabfocus.at(i-1); // The tabfocus number must be incremental, no gaps if (tabfocus.value <> previousTabfocus.value + 1) { errors = errors + "The tabfocus number must be incremental, no gaps or repetitions. "; } // The element must be around the same vertical position or lower than its predecessor if (previousTabfocus.box.position.y - tabfocus.box.position.y > 5) { errors = errors + "The element " + tabfocus.guiElement.name + " is higher than its predecessor. "; } // If it's around the same vertical position, it must be on the right of its predecessor var verticalDifference = (tabfocus.box.position.y - previousTabfocus.box.position.y).abs(); if (verticalDifference <= 5 and tabfocus.box.position.x < previousTabfocus.box.position.x) { errors = errors + "The element " + tabfocus.guiElement.name + " is around the same vertical position but on the left of its predecessor"; } } return errors; </pre>

RANK	TYPE	String
	VALUES	"
HOW TO CORRECT		
CORRECTION		
OTHER INFORMATION		

NAME		FOREGROUND COLOR PALETTE
DEFINITION		Checks if the foreground color of the element is in the required palette
ERROR MESSAGE		The foreground color of the element is not in the palette
IMPORTANCE		High
ELEMENT TYPE		Color
RESTRICTION		
HOW TO CALCULATE		We just get the foreground color of the element. Rank check will determine if it's valid or not.
CALCULATION		<code>return e.foreground;</code>
RANK	TYPE	String
	VALUES	'#000000', '#FF0000', '#00FF00', '#0000FF', '#FFFF00', '#00FFFF', '#FF00FF', '#FFFFFF', '#E1E1E1'
HOW TO CORRECT		We get the closest color in the palette in terms of its r, g and b values, and set it as the foreground color of the element
CORRECTION		<pre> var colorPalette = Sequence{'#000000', '#FF0000', '#00FF00', '#0000FF', '#FFFF 00', '#00FFFF', '#FF00FF', '#FFFFFF', '#E1E1E1'}; var closestColor = colorPalette.sortBy(c ((e.foreground.getR()- c.getR()).pow(2) + (e.foreground.getG()- c.getG()).pow(2) + (e.foreground.getB()- c.getB()).pow(2)).pow(0.5)).first(); e.guiElement.color.foreground = closestColor; </pre>
OTHER INFORMATION		

NAME		BACKGROUND COLOR PALETTE
DEFINITION		Checks if the background color of the element is in the required palette
ERROR MESSAGE		The background color of the element is not in the palette
IMPORTANCE		High
ELEMENT TYPE		Color
RESTRICTION		
HOW TO CALCULATE		We just get the background color of the element. Rank check will determine if it's valid or not.
CALCULATION		<code>return e.background;</code>
RANK	TYPE	String
	VALUES	'#000000', '#FF0000', '#00FF00', '#0000FF', '#FFFF00', '#00FFFF', '#FF00FF', '#FFFFFF', '#E1E1E1'
HOW TO CORRECT		We get the closest color in the palette in terms of its r, g and b values, and set it as the foreground color of the element
CORRECTION		<pre> var colorPalette = Sequence{'#000000', '#FF0000', '#00FF00', '#0000FF', '#FFFF 00', '#00FFFF', '#FF00FF', '#FFFFFF', '#E1E1E1'}; var closestColor = colorPalette.sortBy(c ((e.background.getR()- c.getR()).pow(2) + (e.background.getG()- c.getG()).pow(2) + (e.background.getB()- c.getB()).pow(2)).pow(0.5)).first(); e.guiElement.color.background = closestColor; </pre>
OTHER INFORMATION		

NAME		CONTRAST
DEFINITION		Checks the contrast between an element's foreground and background color
ERROR MESSAGE		Low contrast between foreground and background colors
IMPORTANCE		High
ELEMENT TYPE		Color
RESTRICTION		
HOW TO CALCULATE		We get the division between the luminance of the foreground and the background colors of the element
CALCULATION		<pre>return (e.background.getLuminance() .max(e.foreground.getLumina nce())+0.05)/(e.background.getLuminance().min(e.foregro und.getLuminance())+0.05);</pre>
RANK	TYPE	Double
	VALUES	4.5-*
HOW TO CORRECT		
CORRECTION		
OTHER INFORMATION		<pre>function luminanace(r, g, b) { var a = [r,g,b].map(function(v) { v /= 255; return (v <= 0.03928) ? v / 12.92 : Math.pow(((v+0.055)/1.055), 2.4); }); return a[0] * 0.2126 + a[1] * 0.7152 + a[2] * 0.0722; }</pre>

NAME		SEPARATION
DEFINITION		Checks the separation between an element and the rest inside the same container
ERROR MESSAGE		The element is too close to other elements in its container
IMPORTANCE		High
ELEMENT TYPE		Box
RESTRICTION		<code>return e.container.isDefined();</code>
HOW TO CALCULATE		We iterate through the rest of the elements in the container and check if the separation is enough in at least one direction. We return the names of the elements that do not satisfy the restriction, separated by commas.
CALCULATION		<code>return e.container.content.select(e2 e2.isTypeOf(Box) and e2<>e and not((e.position.x - (e2.position.x+e2.dimension.x) >= 5) or (e2.position.x - (e.position.x+e.dimension.x) >= 5) or (e.position.y - (e2.position.y+e2.dimension.y) >= 5) or (e2.position.y - (e.position.y+e.dimension.y) >= 5))).collect(e2 e2.guiElement.name).concat(",");</code>
RANK	TYPE	String
	VALUES	"
HOW TO CORRECT		Get the distance between lefts-rights and tops-bottoms. We get the lower (in absolute value) and modify the element's position in that direction.
CORRECTION		<pre> var minDistance = 5; // We get the conflicting elements var elements = e.container.content.select(e2 e2.isTypeOf(Box) and e2<>e and not((e.position.x - (e2.position.x+e2.dimension.x) >= minDistance) or (e2.position.x - (e.position.x+e.dimension.x) >= minDistance) or (e.position.y - (e2.position.y+e2.dimension.y) >= minDistance) or (e2.position.y - (e.position.y+e.dimension.y) >= minDistance))); // For each of them: for (other : Box in elements) { // We get the distances between the different edges var right = other.position.x - (e.position.x + e.dimension.x); var left = (other.position.x + other.dimension.x) - e.position.x; var bottom = other.position.y - (e.position.y + e.dimension.y); var top = (other.position.y + other.dimension.y) - e.position.y; // We get the minimum var min = Sequence{right, left, bottom, top}.sortBy(v v.abs()).first(); // We move the other element in that direction switch (min) { case right : other.guiElement.position.x = other.position.x - min + minDistance; </pre>
118		

```
        case left : other.guiElement.position.x =
other.position.x - min - minDistance;
        case bottom : other.guiElement.position.y
= other.position.y - min + minDistance;
        case top : other.guiElement.position.y =
other.position.y - min - minDistance;
    }
    other.position.x = other.guiElement.position.x;
    other.position.y = other.guiElement.position.y;
}
```

OTHER INFORMATION

NAME		FONT TYPES
DEFINITION		Checks the number of distinct font types used in the interface
ERROR MESSAGE		Too many distinct font types
IMPORTANCE		High
ELEMENT TYPE		GUI
RESTRICTION		
HOW TO CALCULATE		Takes all instances of class Text and put its font types in a set (no repetition collection). The number of distinct font types in the application is the size of this set.
CALCULATION		<pre>return Text.allInstances().collect(t t.font).asSet().size();</pre>
RANK	TYPE	Integer
	VALUES	1-2 font types
HOW TO CORRECT		We iterate through the Texts collecting the different font types until the limit is reached. From that point on, every extra font type will be replaced with the first one in the valid types collection.
CORRECTION		<pre>var maxTypes = 2; var validTypes : Set; for (t in Text.allInstances()) { if (validTypes.size() < maxTypes) { // We collect valid types until reaching the limit validTypes.add(t.font); } else if (not validTypes.includes(t.font)) { // We get the first font type var newType = validTypes.first(); // We set the new size t.guiElement.font.family = newType; } }</pre>
OTHER INFORMATION		

NAME		LABEL-INPUT SEPARATION
DEFINITION		Checks the separation between an input field and its corresponding label
ERROR MESSAGE		The label is too far from its input field
IMPORTANCE		Medium
ELEMENT TYPE		Input
RESTRICTION		<code>return e.label.isDefined();</code>
HOW TO CALCULATE		We get the separation between an input field and its associated label (top or left separation)
CALCULATION		<code>return (e.box.position.x-(e.label.box.position.x+e.label.box.dimension.x)).max(e.box.position.y-(e.label.box.position.y+e.label.box.dimension.y));</code>
RANK	TYPE	Integer
	VALUES	0-30 px
HOW TO CORRECT		We get the label closer in the right direction (moving it lower or to the right)
CORRECTION		<pre> var horizontal = e.box.position.x- (e.label.box.position.x+e.label.box.dimension.x); var vertical = e.box.position.y- (e.label.box.position.y+e.label.box.dimension.y); if (horizontal > vertical) { e.label.box.position.x = e.label.box.position.x + (horizontal - 30); e.label.guiElement.position.x = e.label.box.position.x; } else { e.label.box.position.y = e.label.box.position.y + (vertical -30); e.label.guiElement.position.y = e.label.box.position.y; } </pre>
OTHER INFORMATION		

NAME		MENU DEPTH
DEFINITION		Checks the maximum depth of the menu structure
ERROR MESSAGE		Menu structure is too deep
IMPORTANCE		Medium
ELEMENT TYPE		Menu
RESTRICTION		
HOW TO CALCULATE		We iterate through the menu structure using a stack, keeping track of the depth at every moment to find out the maximum depth
CALCULATION		<pre> var stack = new Sequence; stack.add(e); var depth = 0; var maxDepth = 0; var count = 0; while(not stack.isEmpty()) { var last = stack.last(); stack.remove(last); // Extraction of an already explored father if (last.isKindOf(Real)) { depth = depth - 1; } else if (not last.items.isEmpty()) { // We replace a father by a number stack.add(count); stack.addAll(last.items); depth = depth + 1; if (depth > maxDepth) maxDepth = depth; } count = count + 1; } return maxDepth; </pre>
RANK	TYPE	Integer
	VALUES	1-3 levels
HOW TO CORRECT		
CORRECTION		
OTHER INFORMATION		

NAME		LABEL TEXT LENGTH
DEFINITION		Checks the length of the text in a label
ERROR MESSAGE		Label text is too long
IMPORTANCE		Medium
ELEMENT TYPE		Label
RESTRICTION		
HOW TO CALCULATE		We just get the label text's length
CALCULATION		<code>return e.text.text.length();</code>
RANK	TYPE	Integer
	VALUES	1-50 characters
HOW TO CORRECT		We take the 50 first characters of the label text and remove the rest
CORRECTION		<code>e.guiElement.text = e.text.text.substring(0, 49);</code>
OTHER INFORMATION		

NAME		FONT SIZE
DEFINITION		Checks the minimum font size for any text element
ERROR MESSAGE		Font size is too small
IMPORTANCE		High
ELEMENT TYPE		Text
RESTRICTION		
HOW TO CALCULATE		We just get the text's font size
CALCULATION		<code>return e.size;</code>
RANK	TYPE	Double
	VALUES	10-*
HOW TO CORRECT		<code>e.guiElement.font.size = 10.0;</code>
CORRECTION		
OTHER INFORMATION		

NAME		VERTICAL ALIGN
DEFINITION		Checks if an element is vertically aligned with the rest of elements in its container
ERROR MESSAGE		The element is not vertically aligned with other elements in its container
IMPORTANCE		High
ELEMENT TYPE		Box
RESTRICTION		<code>return e.container.isDefined();</code>
HOW TO CALCULATE		We iterate through the other elements in the container and check if the vertical positions are similar but not identical. We check both top and bottom positions. We return the names of the elements that do not satisfy the restriction, separated by commas.
CALCULATION		<pre>var margin = 10; return e.container.content.select(e2 e2.isTypeOf(Box) and e2<>e and ((e.position.y-e2.position.y).abs()>0 and (e.position.y-e2.position.y).abs()<margin) or (((e.position.y+e.dimension.y) - (e2.position.y+e2.dimension.y)).abs()>0 and ((e.position.y+e.dimension.y) - (e2.position.y+e2.dimension.y)).abs()<margin))).collect (e2 e2.guiElement.name).concat(",");</pre>
RANK	TYPE	String
	VALUES	"
HOW TO CORRECT		For every object not aligned with the current object, we get the difference between the two positions and move the other element in that direction.
CORRECTION		<pre>// We get the not aligned elements var margin = 10; var elements = e.container.content.select(e2 e2.isTypeOf(Box) and e2<>e and ((e.position.y-e2.position.y).abs()>0 and (e.position.y-e2.position.y).abs()<margin) or (((e.position.y+e.dimension.y) - (e2.position.y+e2.dimension.y)).abs()>0 and ((e.position.y+e.dimension.y) - (e2.position.y+e2.dimension.y)).abs()<margin))); // For each of them: for (other : Box in elements) { // We get the minimum distance between the vertical positions (top or bottom) var top = e.position.y-other.position.y; var bottom = (e.position.y+e.dimension.y) - (other.position.y+other.dimension.y); // We get the minimum var min = Sequence{top, bottom}.sortBy(v v.abs()).first(); // We move the other element in that direction other.position.y = other.position.y + min; other.guiElement.position.y = other.position.y; }</pre>
OTHER INFORMATION		

NAME		HORIZONTAL ALIGN
DEFINITION		Checks if an element is horizontally aligned with the rest of elements in its container
ERROR MESSAGE		The element is not horizontally aligned with other elements in its container
IMPORTANCE		High
ELEMENT TYPE		Box
RESTRICTION		<code>return e.container.isDefined();</code>
HOW TO CALCULATE		We iterate through the other elements in the container and check if the horizontal positions are similar but not identical. We check both left and right positions. We return the names of the elements that do not satisfy the restriction, separated by commas.
CALCULATION		<pre>var margin = 10; return e.container.content.select(e2 e2.isTypeOf(Box) and e2<>e and (((e.position.x-e2.position.x).abs())>0 and (e.position.x-e2.position.x).abs()<margin) or (((e.position.x+e.dimension.x) - (e2.position.x+e2.dimension.x)).abs())>0 and ((e.position.x+e.dimension.x) - (e2.position.x+e2.dimension.x)).abs()<margin))).collect (e2 e2.guiElement.name).concat(",");</pre>
RANK	TYPE	String
	VALUES	"
HOW TO CORRECT		For every object not aligned with the current object, we get the difference between the two positions and move the other element in that direction.
CORRECTION		<pre>// We get the not aligned elements var margin = 10; var elements = e.container.content.select(e2 e2.isTypeOf(Box) and e2<>e and (((e.position.x-e2.position.x).abs())>0 and (e.position.x-e2.position.x).abs()<margin) or (((e.position.x+e.dimension.x) - (e2.position.x+e2.dimension.x)).abs())>0 and ((e.position.x+e.dimension.x) - (e2.position.x+e2.dimension.x)).abs()<margin))); // For each of them: for (other : Box in elements) { // We get the minimum distance between the horizontal positions (left or right) var left = e.position.x-other.position.x; var right = (e.position.x+e.dimension.x) - (other.position.x+other.dimension.x); // We get the minimum var min = Sequence(right, left).sortBy(v v.abs()).first(); // We move the other element in that direction other.position.x = other.position.x + min; other.guiElement.position.x = other.position.x; }</pre>

NAME		NUMBER OF ELEMENTS
DEFINITION		Checks the total number of elements in a container
ERROR MESSAGE		Too many elements in this container
IMPORTANCE		Low
ELEMENT TYPE		Container
RESTRICTION		
HOW TO CALCULATE		We obtain the number of elements inside the container
CALCULATION		<code>return e.content.select(b b.isTypeOf(Box)).size();</code>
RANK	TYPE	Integer
	VALUES	0-12 elements
HOW TO CORRECT		
CORRECTION		
OTHER INFORMATION		

NAME		LABELED INPUT
DEFINITION		Checks if an input has its corresponding label
ERROR MESSAGE		This input does not have any label associated
IMPORTANCE		High
ELEMENT TYPE		Input
RESTRICTION		
HOW TO CALCULATE		We check if the input field has a label associated
CALCULATION		<code>return e.label.isDefined();</code>
RANK	TYPE	Boolean
	VALUES	true
HOW TO CORRECT		
CORRECTION		
OTHER INFORMATION		

NAME		ACCESSIBILITY INFORMATION
DEFINITION		Checks if accessibility information has been set for an element
ERROR MESSAGE		Accessibility information has not been set
IMPORTANCE		Medium
ELEMENT TYPE		AccessibilityInformation
RESTRICTION		
HOW TO CALCULATE		We check that the accessibility information has been set.
CALCULATION		<code>return e.info.length()>0;</code>
RANK	TYPE	Boolean
	VALUES	true
HOW TO CORRECT		
CORRECTION		
OTHER INFORMATION		

NAME		TOOLTIP
DEFINITION		Checks if tooltip text has been set for an element
ERROR MESSAGE		Tooltip text has not been set
IMPORTANCE		Low
ELEMENT TYPE		Tooltip
RESTRICTION		
HOW TO CALCULATE		We check that the tooltip property has been set.
CALCULATION		<code>return e.text.length() > 0;</code>
RANK	TYPE	Boolean
	VALUES	true
HOW TO CORRECT		
CORRECTION		
OTHER INFORMATION		

NAME		OVERLAP
DEFINITION		Checks if the element overlaps other elements in its container
ERROR MESSAGE		This element overlaps other elements in its container
IMPORTANCE		High
ELEMENT TYPE		Box
RESTRICTION		<code>return e.container.isDefined();</code>
HOW TO CALCULATE		We iterate through the other elements in the container. We check that each element is completely on the left, right, top or bottom. We return the names of the elements that do not satisfy the restriction, separated by commas.
CALCULATION		<code>return e.container.content.select(e2 e2.isTypeOf(Box) and e2<>e and not((e.position.x >= (e2.position.x+e2.dimension.x)) or ((e.position.x+e2.dimension.x) <= e2.position.x) or (e.position.y >= (e2.position.y+e2.dimension.y)) or ((e.position.y+e2.dimension.y) <= e2.position.y))).collect(e2 e2.guiElement.name).concat(',');</code>
RANK	TYPE	String
	VALUES	"
HOW TO CORRECT		Get the distance between lefts-rights and tops-bottoms. We get the lower (in absolute value) and modify the element's position in that direction.
CORRECTION		<pre>// We get the overlapped elements var elements = e.container.content.select(e2 e2.isTypeOf(Box) and e2<>e and not((e.position.x >= (e2.position.x+e2.dimension.x)) or ((e.position.x+e2.dimension.x) <= e2.position.x) or (e.position.y >= (e2.position.y+e2.dimension.y)) or ((e.position.y+e2.dimension.y) <= e2.position.y))); // For each of them: for (other : Box in elements) { // We get the distances between the different edges var right = other.position.x - (e.position.x + e.dimension.x); var left = (other.position.x + other.dimension.x) - e.position.x; var bottom = other.position.y - (e.position.y + e.dimension.y); var top = (other.position.y + other.dimension.y) - e.position.y; // We get the minimum var min = Sequence(right, left, bottom, top).sortBy(v v.abs()).first(); // We move the other element in that direction if (right == min or left == min) { other.position.x = other.position.x - min; other.guiElement.position.x = other.position.x; } else { other.position.y = other.position.y - min;</pre>

```
        other.guiElement.position.y =  
other.position.y;  
    }  
}
```

OTHER INFORMATION