DIPLOMA THESIS

A Graphical Editor for Structural Screen Models

Submitted at the
Faculty of Electrical Engineering and Information Technology,
Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals Master of Sciences)

under supervision of

Univ.Prof. Dipl.-Ing. Dr.techn. Hermann Kaindl
Univ.Ass. Dipl.-Ing. Dr.techn. Roman Popp
Univ.Ass. Dipl.-Ing. Dr.techn. David Raneburger, BSc

by

Alexander Armbruster
Matr.Nr. 0125349
Anzengrubergasse 12/26, 1050 Wien

September 12, 2014

Kurzfassung

Die Usability von vollautomatisch generierten Graphical User Interfaces (GUIs) ist im allgemeinen nicht befriedigend. Semi-automatisches Generieren von GUIs bedeutet, dass der Designer manuelle Anpassungen vornehmen muss, um die Usability des generierten GUI zu verbessern. Um den Aufwand dafür gering zu halten, sind entsprechende Tools Voraussetzung. Die Unified Communication Platform (UCP) unterstützt semi-automatische Generierung eines GUI anhand eines Diskurs-basierten Kommunikationsmodelles. UCP transformiert dieses in ein Screen-basiertes GUI-Modell – das Structural Screen Model. Dieses Modell ist am Concrete User Interface (CUI) level und UCP stellt einen Baum-Editor zur Verfügung, welcher zwar die `Widget`-Hierarchie des Strukturellen Screen Modells darstellt, jedoch Layout- und Style-Informationen nicht graphisch visualisiert.

Der Graphical Screen Model Editor (GSME) ermöglicht das Visualisieren und Anpassen des automatisch generierten GUI auch von nicht-Programmierern (z.B., Designern). Insbesondere visualisiert der GSME das Structural Screen Model (i.e., im CUI level) mit dem angegebenen Layout und mit dem Style von einer definierten CSS-Datei. Zusätzlich ermöglicht der GSME Verbesserungen des GUI durch direkte Layout und Style-Anpassungen am Structural Screen Model, sowie spezielle Layout-Anpassungen, welche notwendige Zwischenschritte automatisieren.

Wir entwickelten den GSME mit Hilfe des Eclipse Modeling Projects (EMPs). Das `Structural UI Meta-Model` ist das zugrundeliegende Meta-Modell des graphischen Editors. Erweitert und angepasst haben wir diesen Editor, mittels Templates und zusätzlichen Java-Klassen um eine vollständige Neugenerierung zu ermöglichen.

Abstract

The usability of fully-automatically generated Graphical User Interfaces (GUIs) is typically in need of improvement. Semi-automatic GUI-generation, i.e, keeping the designer in the loop, allows for improving the usability through manual customizations and requires adequate tool support to keep the development effort low. The Unified Communication Platform (UCP) supports semi-automatic GUI generation from high-level interaction models. In particular, UCP automatically transforms Discourse-based Communication Models into a screen-based GUI model – the Structural Screen Model. This model is on the Concrete User Interface (CUI) level and UCP provides a Tree Editor that does not graphically visualize layout and style information, but displays the widget-hierarchy of the Structural Screen Model.

The Graphical Screen Model Editor (GSME) facilitates GUI customization for non-programming experts (e.g., designers) through visualizing the automatically generated GUI. In particular, the GSME visualizes the Structural Screen Model (i.e., on CUI level) with its layout information and with the style information from the specified Cascading Style Sheet (CSS)-file. Furthermore, it supports layout and style customizations through direct manipulation of the Structural Screen Model and advanced layout features that facilitate customization through partly automating moving and resizing a widget.

We developed the GSME based on the Eclipse Modeling Project (EMP). The `Structural UI meta-model` is the underlying model for the generation of the graphical editor. Adaptations and extensions of the GSME are persisted in templates and additional Java-classes, to support the re-generation of the graphical editor.

Acknowledgements

Nicht das Beginnen wird belohnt, sondern einzig und allein das Durchhalten. Schlussendlich diese Worte zu schreiben ist die wahre Belohnung meines Weges durch das Studium.

Ganz besonders bedanken möchte ich mich bei meinem Betreuer und Freund David Raneburger, der mich während des Entwickelns und ganz besonders während des Schreibens dieser Diplomarbeit bestens betreut hat und sich sogar an freien Tagen Zeit für meine Anliegen genommen hat. Vielen Dank David, du warst wirklich eine unglaubliche Hilfe.

Univ.Prof. Dipl.-Ing. Dr.techn. Hermann Kaindl möchte ich für die vielen wertvollen Tipps und Anregungen, für sein genaues Korrekturlesen, sowie für die schnellen Feedbacks danken. Nach der letzten Iteration kann ich diese Arbeit guten Gewissens drucken und binden lassen.

Meinen Studienkollegen und Freunden danke ich für die (teils nächtlichen) gemeinsamen Lernexzesse und überhaupt für die gemeinsame Zeit durch die verschiedensten Lebenssituationen.

Meinen Freunden, die mich während dieser – nicht immer einfachen – Zeit begleitet und unterstützt haben, möchte ich danken.

Ein herzliches Danke auch an meine Familie, dass sie mir das Studieren ermöglicht hat und mich so gut es ging unterstützt und motiviert hat.

Nicht zuletzt möchte ich meiner Freundin Judith danken, für ihre Geduld und Unterstützung, welche sie mir auch in den nächsten Wochen noch entgegenbringen wird.

Table of Contents

Abbreviations

ANM	Action-Notification Model
CRF	CAMELEON Reference Framework
CSS	Cascading Style Sheet
CUI	Concrete User Interface
DoD	Domain-of-Discourse Model
EMF	Eclipse Modeling Framework
EMP	Eclipse Modeling Project
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GSME	Graphical Screen Model Editor
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
MVC	Model View Controller
OCL	Object Constraint Language
RCP	Rich Client Platform
RST	Rhetorical Structure Theory
SWT	Standard Widget Toolkit
UCP:UI	Unified Communication Platform UI Generation Framework
UCP	Unified Communication Platform
UI	User Interface
UML	Unified Modeling Language

1 Introduction and Motivation

1.1 Motivation

Graphical User Interface (GUI) development is time-consuming and error prone, so automating this process is desirable. Model-driven software development provides the means to automatically generate GUIs. Generating GUIs from high-level models potentially saves time and effort, but currently has the drawback that the usability of such automatically generated User Interface (UI)s is typically rather low [MPV11]. A major reason for the low usability is that high-level interaction models do not specify any details on layout and style of the final GUI. This information is completed based on heuristics during the transformation process, which does typically not lead to the result desired by the designer.

One way to remedy this problem is semi-automated GUI generation, which keeps the designer in the loop and allows her to customize the resulting GUI through additional manual input. Inclusion of the human designer requires adequate tool support, to keep the development effort low.

The Unified Communication Platform (UCP) supports fully automated GUI generation for different devices [PRK13], but the usability of the fully automatically generated GUIs is still not satisfying [RWP+13]. Its transformation process allows for customizing a screen-based GUI model (i.e., the Screen Model) before the final source code is generated. However, UCP only provides a tree editor for the Screen Model, which makes layout and style customizations a rather tedious task [Ran14].

1.2 Approach

The proposed approach, to include the human designer, is to provide a Graphical Screen Model Editor (GSME) that supports layout and style customizations through direct manipulation. Such an editor has the additional benefit that it also supports "perfect fidelity prototyping" [FPR+07] through providing a graphical visualization of each application screen without requiring the corresponding application logic (i.e., application back-end).

UCP is based on the Eclipse Modeling Project (EMP), therefore we decided to use the Graphical Modeling Framework (GMF) which is part of the EMP. Development of the GSME has been separated in two major steps.

The first step was the visualization of the Screen Model in a diagram. This was designed to show a graphical representation of the Screens using further information from device-dependent and application-specific Cascading Style Sheet (CSS).

For the second step, we extended the GSME with a set of layout- and style-manipulations which are an intersection of the Screen Model with CSS possibilities and the possibilities provided by GMF. Additionally, we introduced a set of useful manipulations which partly automate previously required manual adaptations and allow the designer to achieve the desired result with less effort.

Layout manipulations update the model-file directly, so the tree editor can show them immediately. Style-manipulations are stored in a separate CSS-file, this enables their re-usability.

1.3 Outline

Chapter 2 presents background information on the Unified Unified Communication Platform UI Generation Framework (UCP:UI) whose Structural Screen Model provides the basis for the GSME. It introduces a bike rental scenario as a running example, which is used to illustrate the approach and the capabilities of the GSME in the subsequent chapters. Finally, it presents background information on the Eclipse Graphical Modeling Framework[1], which was used to implement the GSME.

Chapter 3 lists the requirements for the GSME. This chapter also introduces the GSME Workbench GUI, providing detailed information about the supported layout and style customizations and their initiation. Subsequently we present the GSME features that were implemented to support these customizations, including interesting aspects about the implementation. The satisfaction of each requirement is presented at the end of this chapter.

Chapter 4 presents the results of this work, by comparing visualization and customization of the Structural Screen Model using the GSME and using the Tree Editor.Furthermore, we provide the results of computational performance measurements of two aspects, the loading time of a `Screen` and the execution time of certain features.

Chapter 5 discusses the limitations and alternative possibilities. Additionally, it presents ideas and improvements for future work.

Chapter 6 provides the conclusion of this work.

Appendix Chapter A provides a "getting started document" for facilitating the application of the GSME.

Appendix Chapter B depicts the `Structural UI Meta-model`, which is the base-model of this work.

Appendix Chapter C shows the detailed command-execution of an `AutoSize Request`, which belongs to an example in Section 3.3 (i.e., Resizing by use of the GSME Pop-up bar)

[1] `http://www.eclipse.org/modeling/gmp/`

2 Background

This chapter provides background information on the UCP GUI Generation Framework and the Eclipse GMF which is relevant for subsequent chapters. Section 2.1 focuses on the process of GUI-generation as supported by the UCP and its models and tasks. Section 2.2 provides an overview about the GMF, its dependency to the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF), and details about the architecture which are relevant for this work.

2.1 The Unified Communication Platform User Interface Generation

The UCP supports automated generation of a GUI based on a Discourse-based Communication Model.

We use a simple Bike Rental Application[1] as a running example throughout this diploma thesis. This application supports the following use cases: Register, Login, Rent a bike, Return a bike.

This section presents an overview about the GUI-generation process in Subsection 2.1.1 focusing on details of its input, the Discourse-based Communication Model (Subsection 2.1.2), and on the Screen Model presented in Subsection 2.1.3.

2.1.1 The GUI Generation Process

The GUI Generation Process supports the (semi-) automatic transformation of a Discourse-based Communication Model to Source Code. Figure 2.1 shows a simplified version of this process, where the tasks (rounded rectangles) perform transformations between the models/artifacts (colored rectangles), which are on different levels of abstractions. Input is the `Discourse-based Communication Model`, which is on the Tasks and Concepts level of the CAMELEON Reference Framework (CRF) [CCT+03]. The task `Model2Model Transformation` includes automated device tailoring, which considers device characteristics (e.g., screen-size). It transforms the `Discourse-based Communication Model` into the `Screen Model`, which is device dependent but still independent from the target-toolkit. The abstraction level of the `Screen Model` is Concrete UI. The task `Model2Code transformation` transforms the `Screen Model` into the `Source`

[1] `http://ucp.ict.tuwien.ac.at/UI/BikeRental`

Code (e.g., Hypertext Markup Language (HTML)), which is on the Final UI level. For a more detailed description of the GUI generation process, see [RPK+11, Ran14].

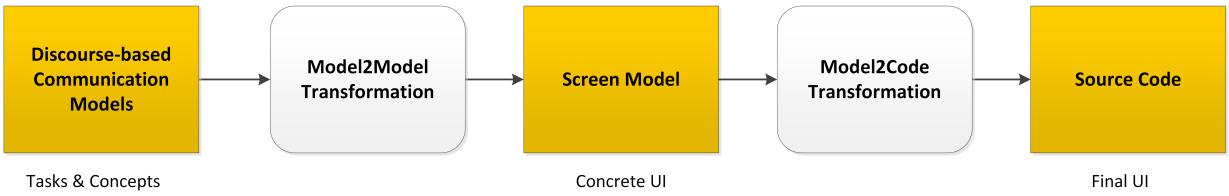


Figure 2.1: Simplified GUI Generation Process

2.1.2 Discourse-based Communication Model

The Discourse-based Communication Model represents the communicative interaction between two parties and consists of three models [Pop12].

The first one is the `Domain-of-Discourse Model (DoD)`, which specifies the objects of conversation (i.e., the objects that the two communicating parties can "talk about") and their relations.

The second one is the `Action-Notification Model (ANM)`, which specifies the operations that can be performed by a communication party [Pop12].

The third one is the `Discourse Model`, which defines classes of possible discourses using `Communicative Acts` as basic units. `Communicative Acts` specify the information exchanged between the communication parties, based on DoD and ANM. `Adjacency Pairs` model typical turn-takings (e.g., question-answer) and consist of an opening `Communicative Act` (e.g., a Request) and up to two optional closing Communicative Acts (e.g., Accept or Reject). `Adjacency Pairs` are linked through Discourse Relations, which are partly based on the Rhetorical Structure Theory (RST) [MT88]. In addition, there are Procedural Constructs like `Condition` or the more complex `IfUntil`, which specify the flow of interaction.

Figure 2.2 shows the Discourse Model of the `Login` use case of our `BikeRental` example. The diamond-shapes represent the `Adjacency Pairs`. Rounded rectangles represent the `Communicative Acts`, their fill-color shows the associated uttering communication party (`System` or `User`). `Alternative` is a Discourse Relation that allows for all branches to be executed concurrently (if assigned to the `User`), in this case register and login. Its left `Nucleus` consists of an `Adjacency Pair` with the opening Offer `Communicative Act O1`, where the `System` offers the `User` to register. The right `Nucleus` consists of a `Title` Relation, which also allows for all sub-branches to be executed concurrently. In this case, the left sub-branch is conditional due to the `Condition` Relation. The `Adjacency Pair` in the condition's `Then`-branch, has just an opening Informing `Communicative Act I2`, which conditionally informs the `User` about a failed login. The right sub-branch of the `Title` (i.e., its `Satellite`) consists of an `Adjacency Pair`, where the `System` asks for the login-credentials, modeled through the `OpenQuestion (OQ5)-Answer (AN5) Adjacency-Pair`. The opening `Communicative Acts` specify the so-called `propositional content`, which references concepts from the ANM and the DoD and defines the interface between `User` and `System`, for more details see [Pop12].

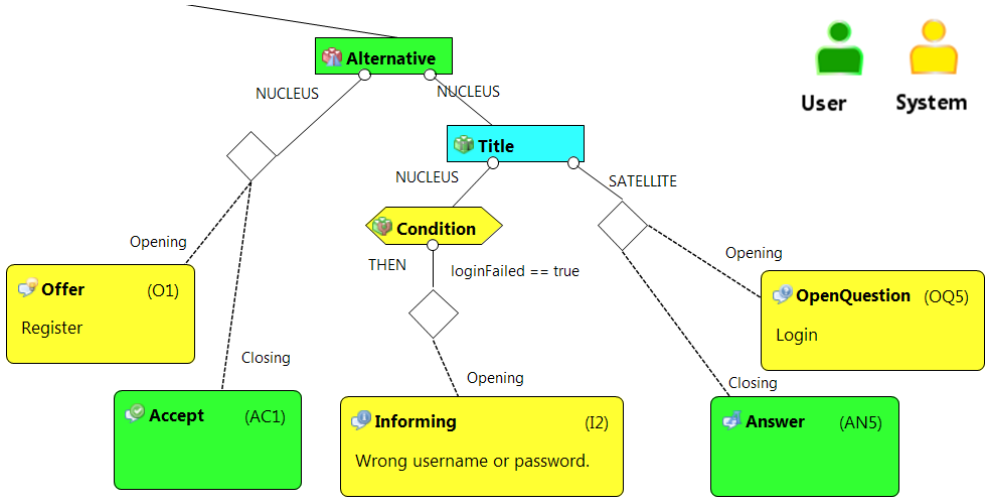


Figure 2.2: Discourse Model: Login of our running example (Example: BikeRentalExtended [Ran14])

2.1.3 The Screen Model

The Screen Model provides a screen-based GUI specification on CUI level [Ran14]. It consist of two parts, the `Behavior Screen Model` and the `Structural Screen Model`. The Behavior Screen Model is represented as a Unified Modeling Language (UML) state-machine, which defines all possible sequences of the Screens. The Structural Screen Model provides the basis for this work. It defines the concrete Screens with all the graphical as well as additional meta-information (e.g., reference to the corresponding element in the DoD, reference to the rule which created a specific `Widget`).

The `Structural UI meta-model` defines all the graphical Elements (i.e., `Widgets`) with their properties and associations, which are available for creating the Screen Model. One specialization of `Widget` is `Container` like `Screen`, `Panel`, `List`, etc., which can contain other `Widgets`. `Buttons`, `Labels`, `TextBoxes`, etc. cannot contain other `Widgets`, they are the leaves of the model-tree.

Figure 2.3 shows the Structural Screen Model of the login use case in a tree view. The root of a Structural Screen Model is a `Frame` containing information about the screen-size (i.e., resolution). `Screens` are the direct children of a `Frame`, and are optimized for the given resolution [Ran14].

An important attribute of `Widget` is `Layout Data`, which defines the position within its parent (i.e., `Container`). There are different specializations of `LayoutData`, the `XYLayoutData` and `GridLayoutData`. This work concentrates on the `GridLayoutData`, because it was the only one used in the examples and references when this work started. `Containers` need the additional attribute `Layout Manager`, which must fit with the kind of `LayoutData` of its children (i.e., containing `Widgets`). The `FlowLayout`, as an exception, does not need additional layout-information from the children. Their width and height, which are properties of the `Widget`, are sufficient for this trivial layout.

Figure 2.3 also shows optional `Style`-properties (e.g., heading). The value is an ID which references a predefined style in a separate CSS-file. The path of corresponding CSS-files is also specified in the Structural Screen Model.

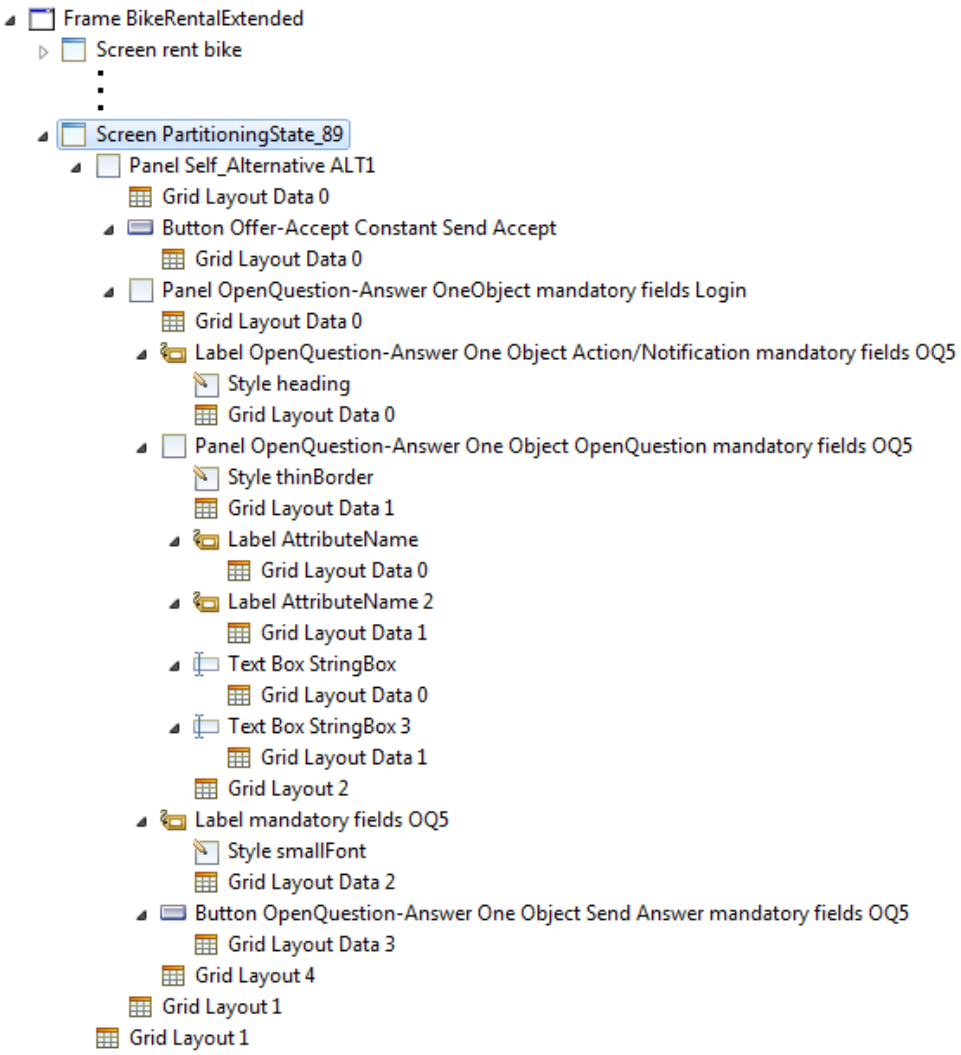


Figure 2.3: Example of a Structural Screen Model

This simple example allows identifying the references between the Structural Screen Model in Figure 2.3 and the Discourse Model in Figure 2.2. For example, the `Button Offer-Accept Constant Send Accept` in the Screen Model represents the `left Nucleus` in the Discourse Model. The `Panel OpenQuestion-Answer OneObject mandatory fields login` represents the `right Nucleus` of the `Alternative`.

2.1.4 UCP-Tool Support

The UCP-tools are packed in a Rich Client Platform (RCP)-Application, which is based on the Eclipse Integrated Development Environment (IDE). Eclipse is mostly written in Java. It consists of a small Runtime-Kernel and a configurable amount of plug-ins, like build-tools, compilers, text editor, resource explorer, problems view, console etc.

The EMF is the most important plug-in for UCP. EMF supports model-based software development. It provides an `Ecore-metamodel` and a graphical editor to create a custom model. The EMF also supports the automated generation of a simple tree editor.

In particular, UCP provides a wizard that facilitates the creation of new projects [Š14], graphical editors for the Discourse Model [FKP+09] and the ANM, tree-based editors for the transformation rules and the Structural Screen Model [PRK13]. The DoD is an `Ecore Model`. So, the graphical editor provided by EMF is sufficient. UCP applies model-to-model transformations in the context of automated GUI-tailoring [Ran14] and model-to-code transformations for automated source code generation (see Figure 2.1). UCP supports generating the GUI source code fully automatically [Ran08]. UCP also generates parts of the source-code for the back-end, which runs on a Web Server as the other interaction party, the `System` [PKR13]. Further details about tool support can be found in [PRK13].

2.2 Eclipse Graphical Modeling Framework

This section introduces the architecture of Graphical Modeling Framework (GMF)/Graphical Editing Framework (GEF) and provides information about the generation[2] process of the graphical editor in Subsection 2.2.1. In Subsection 2.2.2, we provide an overview of the Eclipse Workbench, which provides the basis for the GSME Workbench.

2.2.1 GMF/GEF Architecture

GMF is part of the `Eclipse Graphical Modeling Project`[3], which consists of the parts `GMF-Tooling` and `GMF-Runtime`. While `GMF-Tooling` provides a model-driven approach to generate standardized Eclipse graphical editors, `GMF-Runtime` is an application framework for creating graphical editors using Eclipse Modeling Framework (EMF) and GEF. GEF provides a set of libraries to develop graphical representations of a given meta-model[4].

Figure 2.4 illustrates the Model View Controller (MVC)-architecture of GMF, where the `Controller`, as well as the user-interaction-interface with `Events`, `Requests` and `Commands` can be considered as part of the GEF. Changes to the instantiation of a specific meta-model concept can only be done via the `Controller`. There is exactly one `Controller`, the so-called `EditPart`, per instantiation. Such an `EditPart` is responsible for synchronizing the `View` (i.e., notation model) and its model (instantiation).

Persistence of the `Meta-Model` is done by EMF. GMF produces a separate diagram-file in addition to the model-file for persisting the `Notation Model` with properties like position, size, format, annotations and a reference to the corresponding model.

Figure 2.5 depicts the prerequisites for generating a graphical editor, provided as `Diagram Plug-in`. The `Meta-Model` must be provided as an `Ecore Model`. This can be created with the graphical `Ecore-Editor`, which is part of the EMF. The `Graphical Definition Model (.gmfgraph)` defines the notational representation (i.e., `Figures`) for the concepts specified in the meta-model. A textual Tree Editor with the Eclipse Properties-View is provided for its creation by EMF. The `Tooling Definition Model` defines the tools in the Palette-View, which supports creating new `Nodes` (i.e., a `Figure` with a corresponding `EditPart` and the meta-model) and `Links` between `Nodes`, see [Gro09]. For creating this model, the EMF provides another Tree-Editor with the Eclipse Properties-View. The `Mapping Model` defines the mapping between the

[2] In this section, the term *generation* means generation of the source-code for the graphical editor.
[3] `http://www.eclipse.org/modeling/gmp/`
[4] This model is denoted as *domain* or *semantic* model in publications on GMF.

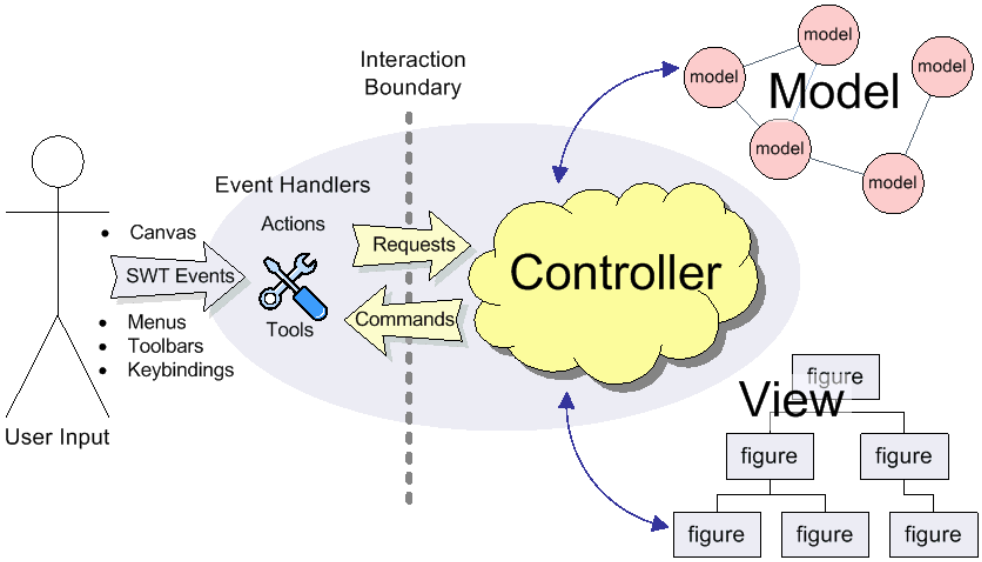


Figure 2.4: Architecture of GMF/GEF[Fou]

meta-model and the notation model, as well as the structure. EMF provides a wizard, which helps creating the `Mapping Model` using these three models. After all four models are provided, the `Generator Model` can be created automatically. Some adjustments on this model influence the final diagram plug-in (e.g., diagram root-element, package-prefix). Creating this plug-in, is also an automated step providing the option to build an RCP-application.

The created Diagram Plug-in (i.e., such an automatically generated graphical editor) provides common features as a basis:

- Selection and Zooming of all objects in the diagram-view
- Displaying a grid in the diagram-view
- A Diagram-Assistant providing Popup-Bars for creating new `nodes` and Connection-Handles to create `links`
- Core- and Appearance-Properties-View (e.g., to edit style-properties like font-size)
- An Appearance Toolbar to edit appearance properties (e.g., font-size)
- A Menu, providing Preview, Print and Export as an image

The GMF is built on top of EMF, it supports automated generation of the source-code for a basic graphical editor, which can be easily adapted or extended. At this point, we define the terms *adaptation* and *extension*, which are used for adapting the editor with templates and extending the editor via Extension Points. The term *customization* is reserved for manual changes during the semi-automatic generation of GUIs from the Discourse-based Communication Models.

The basic diagram plug-in does not support the consideration of explicitly specified layout data, because such editors are typically used for visualizing models with no information about the layout (e.g., UML class-diagram). Visualizing layout information is a core-task of the GSME, which means that the automatically generated editor needs to be adapted. Such adaptations are

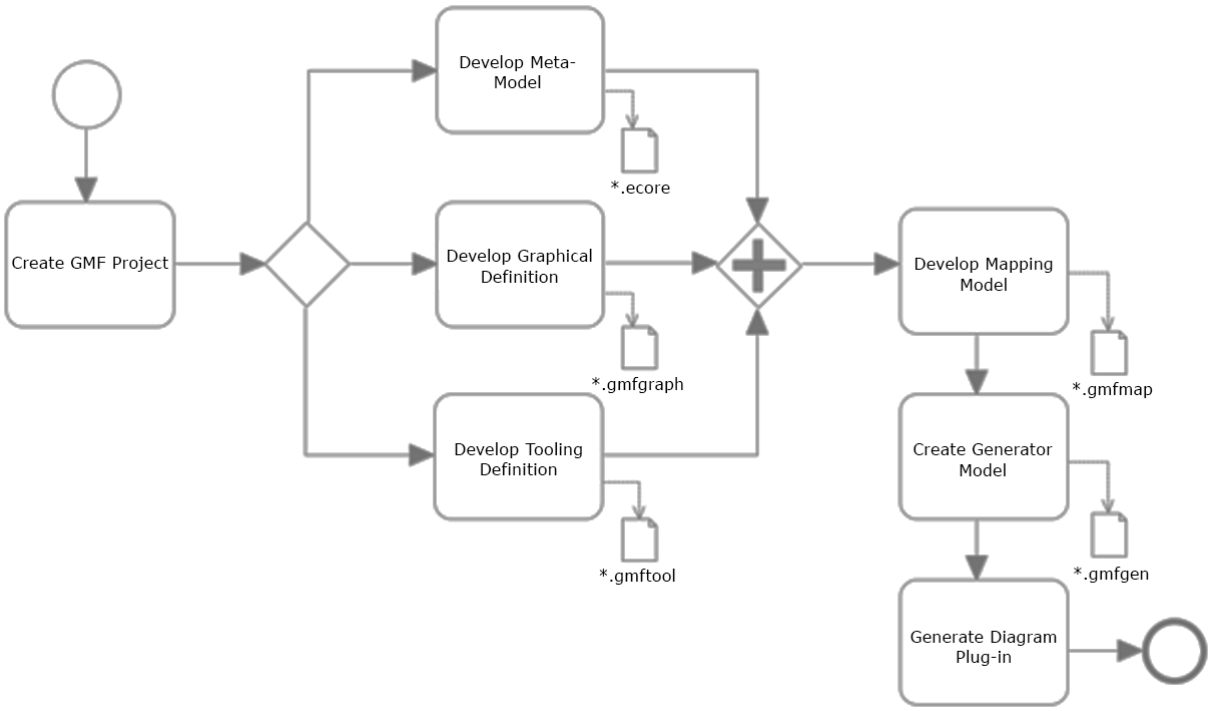


Figure 2.5: Generating a Diagram Plug-in with GMF, copied from [Com]

typically performed on source-code level. The problem is that they are lost in case of re-generation of the graphical editor, after altering the meta-model.

There are several ways to adapt the graphical editor in a persistent way. The simplest possibility is adaptation of the generated source-files. If the sources are regenerated, all manual changes are overwritten, unless the altered methods are annotated with `@generated not`.

Another way for making adaptations persistent, is offered through templates (e.g., Xtend, Xpand2). GMF uses templates for the generation of the source-files and supports using template-files. This adaption is applied at generation-time and allows for changing many similar classes/source-files (e.g., `EditParts` of `Widgets`).

Eclipse supports Extensions and Extension-Points, which can be defined by plug-ins and contributed by plug-ins. Extension-Points provide a loosely coupled way for a modular extension of a plug-in (e.g., adding menu-entries). An extending plug-in is typically an additional Eclipse-project, which the GMF generation-process does not alter.

In order to change the behavior of a `Widget` when the user interacts with it (e.g., hover, select, double-click), the GEF-architecture provides a pluggable contribution with so-called `EditPolicies` [Van04]. An existing `EditPolicy` can be assigned to a specific *role* (e.g., `PRIMARY_DRAG_ROLE`) and installed to an `EditPart`. Figure 2.6 shows the typical communication chain in GEF. A `Request Creator` (e.g., a drag-drop handler) creates a `Request`, which is forwarded to the corresponding `EditPart`. The `EditPart` delegates the `Request` to all installed `EditPolicies`. If an `EditPolicy` supports the `Request`, an appropriate `Command` is returned, otherwise `null`. Different `EditPolicies` can support the same `Request`. All returned `Commands` are chained together and executed sequentially. If no `EditPolicy` of a given `EditPart` supports the `Request`, the resulting `Command` is `null`, which means the `EditPart` does not support it and

the mouse-pointer shows a "not-allowed" icon to the user. A `Command`, which alters the model, can be interpreted as the upper blue arrow from the `Controller` to the `Model` in Figure 2.4.

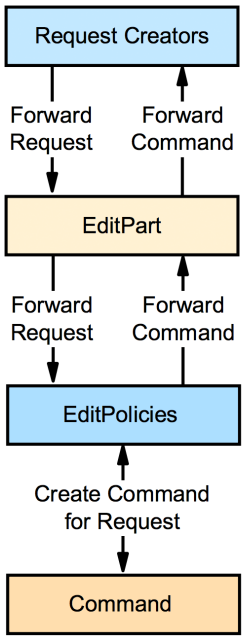


Figure 2.6: Communication Chain, copied from [Van04]

The `EditParts` (i.e., `Controller` of the MVC-architecture) implement a *NotificationListener*, from which the method *notifyChanged(Notification notification)* is called, when the `EditPart` must update its `View` due to a corresponding *semantic change within the model*, which is visualized by the blue arrow in Figure 2.4 pointing from the `Model` to the `Controller`. A notification is also fired, when an `EditPart` must update its model due to a *notational change in the diagram*. This notification is depicted as the upper blue arrow pointing from the `View` to the `Controller`.

2.2.2 Eclipse Workbench

The Eclipse Workbench provides many perspectives and views to facilitate different kinds of software development. This subsection introduces the most common visual parts. Figure 2.7 depicts an overview of the Eclipse IDE. The green-framed Views, referenced by the number in the box, are:

1 – The Toolbar Menu, which provides shortcuts for often used actions (e.g., New, Run, Debug, Search).

2 – The Project Explorer, which lets the user navigate through the projects, create/open files and folders. Alternatively, a Resource Explorer can be used, which provides similar navigation.

3 – The Editor, which visualizes the corresponding content. There are different Editors for different kinds of Data.

4 – The Outline View, which shows an outline of the Editor content (e.g., overview). This can also be a graphical representation.

5 – The Properties View, which provides details of the item that is selected in the Editor or in the Project Explorer.

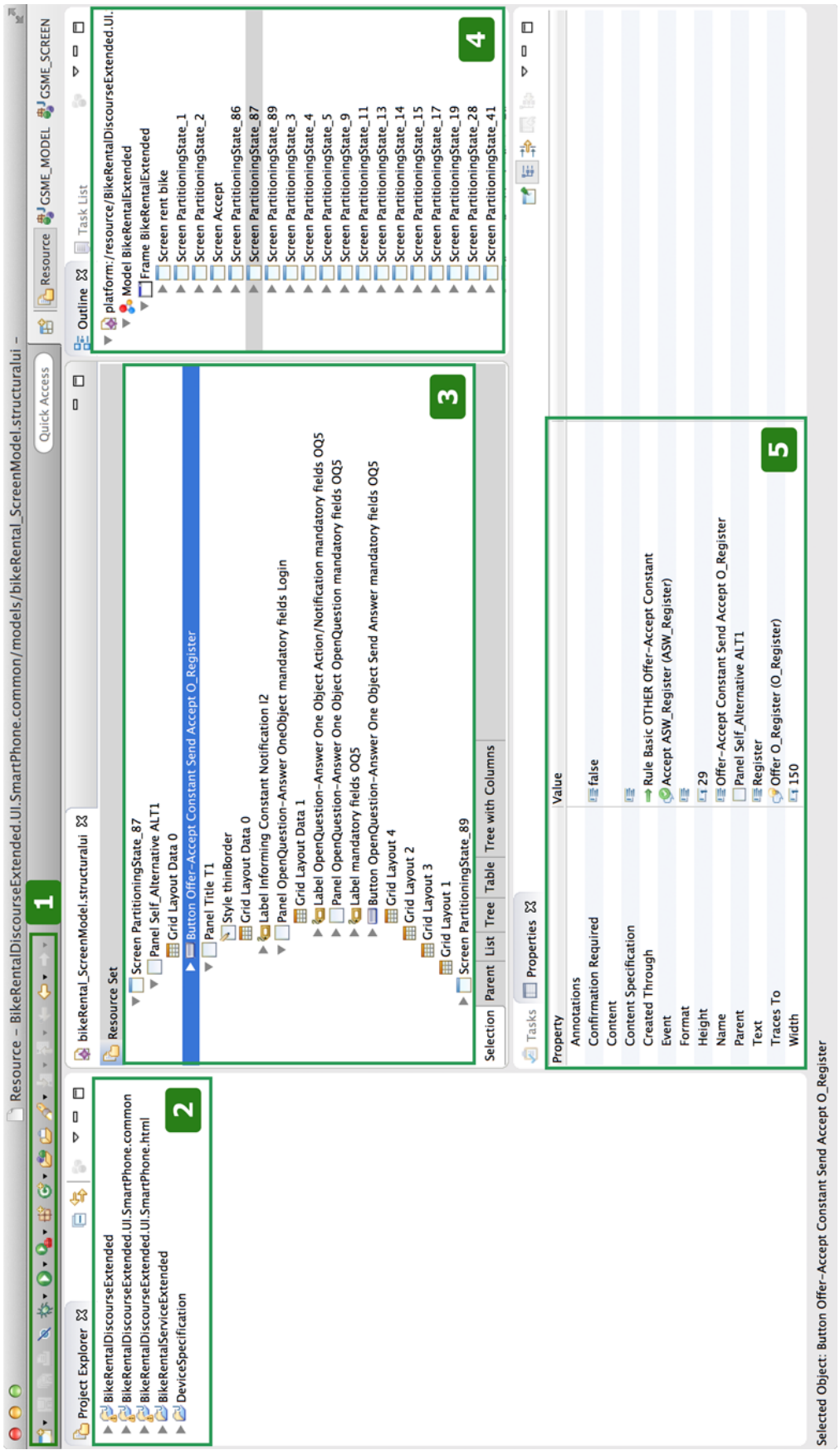


Figure 2.7: Eclipse Workbench

3 Graphical Screen Model Editor

The Graphical Screen Model Editor (GSME) visualizes the `Structural Screen Model` and supports its customization. The GSME consists of two diagram plug-ins and an additional plug-in that implements a custom toolbar (i.e., the GSME Toolbar) and handles its actions. The first diagram plug-in provides an overview of all Screens specified in a given `Structural Screen Model` and the second diagram plug-in provides a `Screen`-based visualization of the automatically generated GUI. Furthermore, the GSME supports several customizations.

Section 3.1 presents the requirements for the GSME, followed by argumentations about the choice of the used architecture. Section 3.2 presents an overview of the GSME Workbench, which hosts the diagram plug-ins, and details about the initiation of the supported customizations. Section 3.3 presents the software design and illustrates the implementation of the customizations (i.e., features) supported by the GSME.

3.1 Requirements for the GSME

Following, we itemize the requirements and present the rationale for the used architecture in the GSME.

RQT1 – The GSME shall generate a graphical representation of `Widgets` (i.e., a diagram) for *.structuralui files.

RQT2 – The GSME shall use the layout information specified in the Screen Model for the graphical representation.

RQT3 – The GSME shall use the style information specified in referenced CSS-files for the graphical representation.

RQT4 – The GSME shall use only information specified in the Structural Screen Model, including referenced information (e.g., style-references), for the graphical representation (i.e., without information from an application back-end).

RQT5 – The GSME shall make manipulations (e.g., layout, size, text, color, etc.) in the diagram persistent in either the *.structuralui file or an additional CSS-file.

RQT6 – The GSME shall provide support for restricting/constraining the allowed manipulations (e.g., a `Widget` can only be laid out anew within its `Container`).

RQT7 – The GSME shall be compatible with Eclipse Validation Framework Live Constraints.

RQT8 – The GSME shall be regenerate-able (i.e., all code modifications shall be persistent, for new generation).

RQT9 – The GSME should support realistic figures for displaying the `Widgets` to make the diagrams more intuitive.

RQT10 – The GSME should provide an interface for reporting layout customizations.

Based on these requirements, we compared three different alternatives for implementing the GSME. The first alternative was to build the GSME from scratch. This would have been the most flexible way, but also the most expensive one. The second alternative was to extend an existing tool (e.g., the Eclipse WindowBuilder[1]) and use it as a basis for the GSME. For that, intensive research of the specific tool and its output would have been necessary. A remaining uncertainty, about the extent of the changes on the tool, keeps the expense difficult to estimate. The third alternative was to use the Eclipse GMF to create the graphical diagrams. GMF supports visualizing EMF-models (e.g., the Screen Model) and provides basic editor functionality. The GMF is known to have a steep learning curve, but compared with the other alternatives, it potentially allows for an efficient implementation.

The rationale for using the Eclipse Modeling Project with the EMF and the GMF, was also driven by the already existing UCP, which hosts the GSME as a collection of plug-ins. The UCP already contained other graphical editors (e.g., the Discourse Model Editor), which are based on the EMF/GMF [FKP+09]. So, the EMF and GMF plug-ins were already part of UCP and the know-how was already available in the development team and in the project.

3.2 GSME Workbench GUI

The GSME Workbench provides the GUI for the GSME. Figure 3.1 depicts an overview of the GSME Workbench. The green rectangles are described next, referenced by their numbers in the boxes:

1 – The Appearance Toolbar provides a standard set of style modifications (e.g., fonts and colors).

2 – The Package Explorer shows the file-structure of all projects (i.e., plug-ins).

Figure 3.1 shows the project-structure of our Bike Rental Application running example. The `BikeRentalDiscourseExtended` plug-in contains the `Discourse-Based Communication Model`, automatically generated launch-configurations and optional custom data like images, rules and style-sheets.

The `BikeRentalDiscourseExtended.UI.SmartPhone.common` plug-in is an automatically generated plug-in, which contains the Screen Model. It is used by the GSME as a basis and for persisting changes. The files in the `models` folder are described next. The `bikeRental_ScreenModel_gsme.css` is a Cascading Style Sheet (CSS), created by the GSME, which is used to persist the style-changes of the widgets. The `bikeRental_ScreenModel.pstm` is a device-independent behavior Model, which is not to be modified and only provided for debugging reasons.

[1] `http://www.eclipse.org/windowbuilder/`

Figure 3.1: GSME Workbench showing the overview diagram (3)

The `bikeRental_ScreenModel.structuralui` is the `Structural Screen Model`, which is described in detail in 2.1.3 above. The file `bikeRental_ScreenModel.structuralui_diagram` was generated by the GSME.

This generation must be triggered manually, with "initialize diagram file" from the context-menu of the `Structural Screen Model`-file (accessible through a right-button mouse click on the corresponding Screen Model file). The `bikeRental_WIMPUI.pstm` represents the `Behavior Screen Model`. The model `bikeRental.structuralui` and the `bikeRental_ScreenModel.pstm` are intermediate models, which are weaved together resulting in the Screen Model, see [RPK+11].

The `BikeRentalDiscourseExtended.UI.SmartPhone.html` plug-in is created automatically, using the Screen Model. The task `Model2Code transformation` (Figure 2.1) generates the source-code of the GUI, which is persisted in this plug-in. The source code consists of HTML-, CSS- and JavaScript-files.

The `BikeRentalServiceExtended` plug-in contains the application back-end.

The `DeviceSpecification` plug-in contains default Application-Tailored Device Specifications [KRF+09] with the corresponding CSS, which are used by UCP for tailoring the GUI automatically.

3 – The Editor View is presented by an Editor plug-in, which visualizes the corresponding content (e.g., a Screen Model or a Screen). Figure 3.1 shows the GSME diagram plug-in, presenting an overview of the `Frame` (shown orange) with its `Screens` (shown blue), arranged in a matrix with three columns and as many rows as necessary to display all `Screens`. If there is more than one `Frame`, the `Frames` are displayed below each other. The purpose of this plug-in is to allow for opening a particular `Screen` using the second diagram plug-in, by double-clicking the corresponding `Screen` representation. The `Screens` have automatically generated screen-names, which are difficult to assign to a dedicated `Screen` for the designer. So, we show additional `Screen`-information (i.e., dimensions, number and nesting-depth of contained `Widgets`) in the overview diagram.

4 – The Palette presents elementary tools, e.g., Selection, Zooming and Annotation.

5 – The Outline View provides an overview of the editor view's content. The actually visible content of the editor view is a highlighted segment, which is indicated by the blue semitransparent rectangle. This can also be moved, to change the visible content of the editor.

6 – The Tabbed Property View in Figure 3.1 shows the `Core Properties`-tab, which represents the properties of the `Widget` that is selected in the Editor-View – in our case, the `Login-Screen` (PartitioningState_87) of our running example. Let us introduce the properties of the `Login-Screen`. `Annotations` are supported by every Ecore-Object (the Ecore-class is the very super-type of the `Widget` class). `Annotations` are used, for example, to pass information to the UCP-Layouter [RPV12]. The `Behavior State` links to the `Behavior Screen Model`. The `Content` defines information about dynamic data to display, unknown at design-time (e.g., a specification of a DoD-Model concept). The `Content Specification` is an Object Constraint Language (OCL) expression, evaluated on the object in the `Traces To`-property (which references to a `Communicative Act` of the `Discourse Model`). The result is stored in the `Content`-property. `Created Through` shows the id of the transformation rule that created a specific `Widget`. `Height` and `Width` define the dimension

of a `Widget`. The property `Name` presents a generated string that identifies the `Widget`. The `Parent` shows the type and the name of the `Widget's` parent (i.e., `Container`).

GMF supports tabbed Property Views, the `Appearance Property Sheet` can be selected by clicking on its caption. It contains the same properties as the `Appearance Toolbar`.

Figure 3.2 depicts the GSME-toolbar and the `Screen`-based diagram.

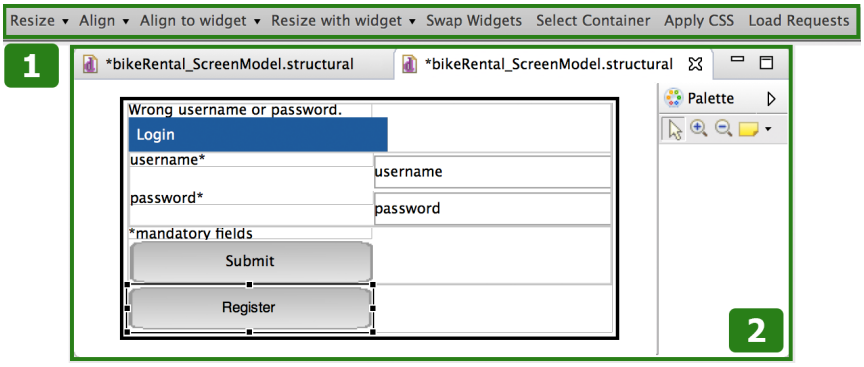


Figure 3.2: The GSME-Toolbar and the `Screen`-based diagram of our running example

1 – The GSME-toolbar lets the user initiate features for customizing the size and position of a `Widget` (depending on other `Widgets`). Furthermore, it allows for selecting the `Container` of a `Widget`, applying the CSS-styles on the selected `Widget` and all containing `Widgets` recursively. It also supports loading of previously initiated Changes (i.e., `Requests`) [Ran14]. A detailed description of the menu-items is presented below.

2 – The Screen-based diagram presents the detailed preview of our running example's login-screen (in the black box). It is the same screen as shown in the tree-view in Figure 2.3. This preview already visualizes the layout as well as the graphical style (e.g., font, font-size, font-color, background-color) from the given CSS and allows the designer to get a good idea of what the final UI will look like.

3.2.1 Layout Customizations

Layout customizations change the *size* or *position* of `Widgets`, constrained by the `Layout` of the `Container` (i.e., parent), by other `Widgets`, having the same parent (i.e., siblings) and by contained `Widgets` (i.e., children). Table 3.1 summarizes these constraints.

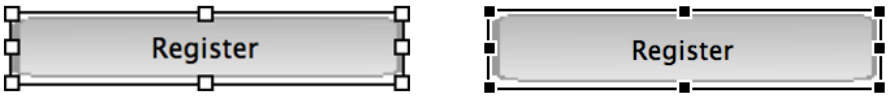
Table 3.1: Layout constraints

1	`Containers`	must wrap their children
2	`Widgets`	must be within their parents
3	`Widgets`	must not overlap with siblings

Layout customizations can be initiated in several ways, the most intuitive are direct manipulations on the `Widget`. Before a customization can be applied on a `Widget`, it must be selected (with the exception of using the GSME-Toolbar). Following, a short description of every customization and their initiation is presented. They are split in resize and move customizations.

Selection of a Widget

A `Widget` can be selected by a single left mouse-click, which is reflected by a surrounding thin rectangle with so-called *resize-handles*, see Figure 3.3 (a).



(a) Resize handles (b) Resize handles, last selection

Figure 3.3: Resize handles

Selection of multiple Widgets

Multiple selection is possible by pressing *CMD* on a MAC (or *CTRL* on a PC) while selecting the `Widgets`. The last selected `Widget` shows a small difference in its resize handles (Figure 3.3 (b)). Some `Resize with Widget` and `Align to Widget` customizations support changes on multiple `Widgets` in one step, using the last selected `Widget` as a `reference Widget`.

Selection of invisible Containers

Some `Widgets` have a `Container` with a size equal to the `Widgets`-size. Moving or Resizing such a `Widget`, may show the *unsupported operation icon* (Figure 3.4) decorating the mouse-pointer because of a violated constraint from Table 3.1 with this *invisible* `Container`. Preceding manipulation on this `Container` is necessary, which requires its selection. Direct selection of such an invisible `Container` is not feasible, so the GSME Toolbar offers the menu-entry `Select Container` (Figure 3.2 (1)).

Figure 3.4: The *unsupported operation icon*

3.2.1.1 Resize Customizations

Resize customizations include changing width and height of a `Widget`.

Directly Resizing

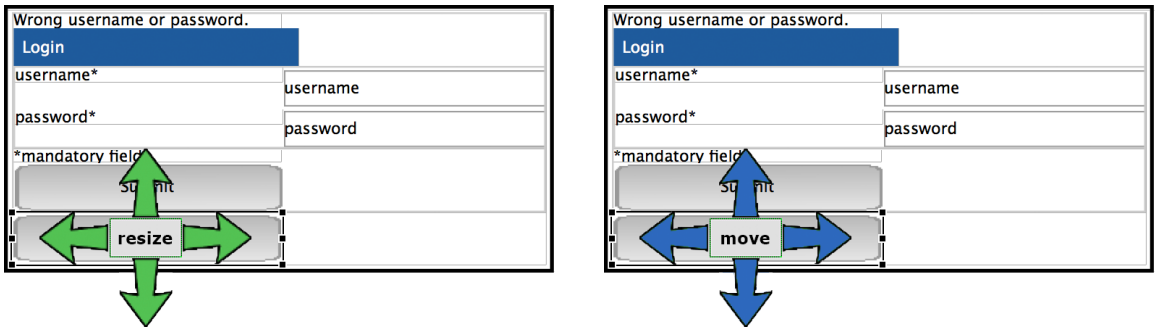
Direct resizing means to grab the `Widget` on a *resize-handle* and drag it to the desired size. This change of size is represented graphically with a gray rectangle, as well as numerically, showing the actual *width* and *height* during the operation. Figure 3.5 (a) shows the feedback during the resize-operation. If the designer tries to resize the `Widget` exceeding the available space, an *unsupported operation icon* decorates the mouse-pointer. The designer is responsible to free the space by moving or resizing other `Widgets` or `Containers`, which can be a tedious task if the `Widgets` are deeply nested.

(a) direct resize (b) direct move

Figure 3.5: Feedback during a direct manipulation

Resizing by use of the GSME Pop-up bar

This feature automates the tedious task of moving or resizing other `Widgets` or `Containers` described above. The designer just selects the `Widget` and the direction in which to resize (i.e., extend the `Widget` by 10 pixels), the GSME automatically frees the required space, by moving siblings and resizing parents recursively. This GSME Pop-up bar can be accessed by pressing *Shift* while hovering the `Widget` (i.e., moving the mouse-pointer over the corresponding `Widget`). The appearing pop-up bar shows the name of the operation (i.e., "resize") in the center, surrounded by green arrows in each direction (Figure 3.6 (a)). By clicking on an arrow, the GSME stretches the `Widget` by 10 pixels in the selected direction. Resizing to the left or top direction, performs a move and resize operation concurrently, because changing the left-top corner position of a `Widget` means a move operation.



(a) Resizing (b) Moving

Figure 3.6: Customizations through the GSME Pop-up bar

Resizing by use of the Core Properties View

Resizing a `Widget` can be done by changing the `width` and/or `height` values of the corresponding properties in the Properties View. After applying a new value, it is immediately verified by the Validation Framework [Sch10], regarding the constraints from Table 3.1. Figure 3.7 shows the result of changing the height of a `Button` from the Properties View. The new height of the `Widget` is invalid, because of a violated constraint from Table 3.1 (e.g., the `Widget` would overlap a sibling).

Resizing by use of the Toolbar Menu

Figure 3.2 (1) depicts the GSME Toolbar, which provides two advanced *resize-features*.

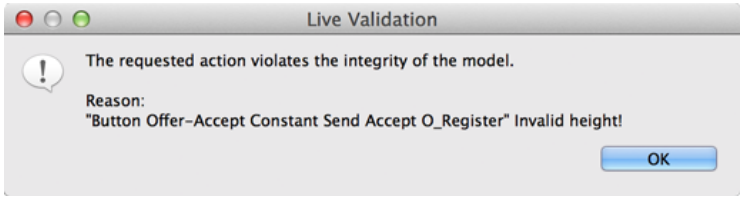
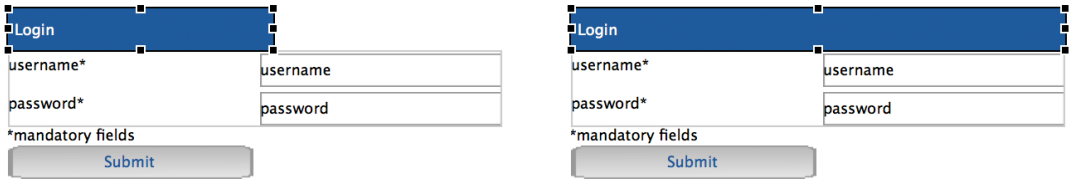


Figure 3.7: Feedback from the Validation Framework

Resize is the first *resize-feature*, which extends the selected `Widget` in a specified direction. The `Resize` menu-entry hides a sub-menu for `MAX`, `TOP`, `RIGHT`, `BOTTOM` and `LEFT`. These are the directions in which the selected `Widget` can be extended, until it touches a sibling or its container. `MAX` extends the `Widget` in all four directions.

Figure 3.8 presents an example, where the designer wants to extend the title "Login" (i.e., a styled `Label`) to grab the whole width of its `Container`. The designer selects the `Label` to resize Figure 3.8 (a), then she opens the `Resize` sub-menu and selects the entry `Right` to extend the `Label` to the most right valid position. Figure 3.8 (b) shows the resulting `Screen` with the extended `Label`.



(a) Selection (b) Resize

Figure 3.8: Example: `Resize` via the GSME-Toolbar

Resize with Widget is the second *resize-feature*, which changes the size of a `Widget` in a specified plane (`HORIZONTAL`, `VERTICAL` or `ALL`), depending on the size of a `reference Widget`. This menu-entry hides a sub-menu for selecting one of these planes. To use this feature, it is necessary to select two or more `Widgets`. The last selected `Widget` is used as the reference, whereas all previously selected `Widgets` are resized according to the size of the `reference Widget` in the respective plane(s). If there is not enough empty space for the resize operation, the GSME tries to free the space by moving and resizing other `Widgets` (i.e., obstacles). The GSME does not handle the case, if a `Widget` to resize is an obstacle for other `Widgets`. Figure 3.9 shows the corresponding error.

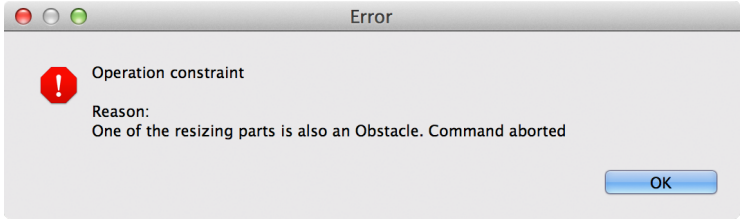
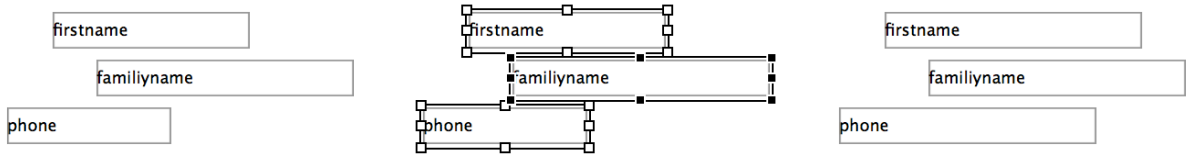


Figure 3.9: Error if an obstacle is a `Widget` to resize

Figure 3.10 (a) shows an example, where the designer wants the three `TextBoxes` to have equal widths, the same as the one in the middle. The corresponding multiple selection is depicted in Figure 3.10 (b), where the `reference TextBox` can be identified through the black *resize-handles*. After folding out the sub-menu of `Resize with Widget`, the designer selects the `HORIZONTAL` plane. Figure 3.10 (c) shows the result of this example, where all three `Widgets` have the same width.



(a) Initial layout (b) Selection (c) Resize

Figure 3.10: Example: `Resize with Widget` via the GSME-Toolbar

3.2.1.2 Move Customizations

Move customizations include changing the position of a `Widget` (specified through the x and y coordinates of the `Widget's` upper left corner).

Directly Moving

Moving a `Widget` can be done directly, by grabbing the whole `Widget` (not the *resize-handles*) and dragging it to the desired position. Another possibility to move the `Widget`, is to use the *cursor buttons* on the keyboard to move it by 1 pixel with each click. The visual feedback for moving is shown in Figure 3.5 (b), where the numerical values reflect the top-left coordinates x and y, relative to the `Widget's Container`. The designer is not allowed to move the `Widget` overlapping another one or outside its `Container`. This is displayed by the *Unsupported Operation Icon*, which decorates the mouse-pointer. The designer is responsible to free the space manually in advance.

Moving by use of the GSME Pop-up bar

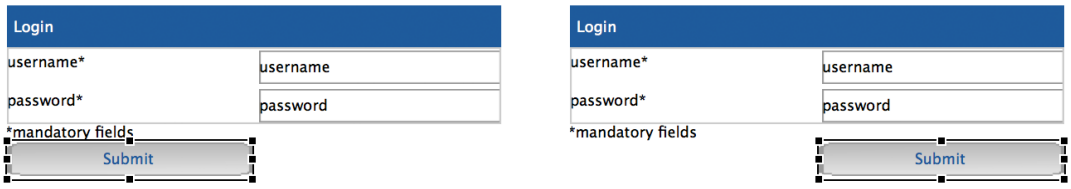
This feature moves a `Widget` by 10 pixels in the selected direction, freeing the required space for the movement automatically. The GSME Pop-up bar for moving can be accessed by pressing *CTRL* while hovering the `Widget`, see Figure 3.6 (b).

Moving by use of the Toolbar Menu

Figure 3.2 (1) depicts the GSME Toolbar, which provides two advanced *align-features* and the feature `Swap Widgets`, which is related to the move features.

Align is the first one, which moves the selected part as far as possible (until it touches a sibling or the border of the parent) in the selected direction. The sub-menu provides the directions `TOP`, `RIGHT`, `BOTTOM` and `LEFT`.

Figure 3.11 presents an example, where the designer wants to align the `Button` "Submit" to the right of its `Container`. She selects the `Button` to align (Figure 3.11 (a)) then she opens the `Align` sub-menu and selects the entry `Right` to move the `Button` to the right most position. Figure 3.11 (b) shows the result.

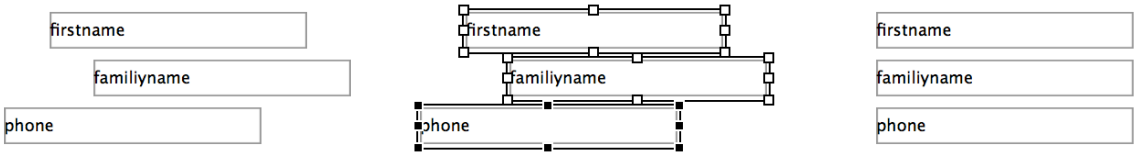


(a) Select (b) Align

Figure 3.11: Example: `Align` via the GSME-Toolbar

Align to Widget is the second *align-feature*, which automatically aligns a specified edge of the selected `Widget` to that of a `reference Widget`. This feature frees the space if necessary. The menu-item of this feature offers the edges `TOP`, `RIGHT`, `BOTTOM` and `LEFT`. The last selected `Widget` is the `reference Widget`.

Figure 3.12 (a) shows an example, where the designer wants the three `TextBoxes` to be aligned to the left edge of the bottom `TextBox`. The corresponding selection is depicted in Figure 3.12 (b), where the `reference TextBox` can be identified through the black *resize-handles*. After folding out the sub-menu of `Align to Widget`, the designer selects the `LEFT` edge. Figure 3.12 (c) shows the result of this example, where all three `Widgets` are aligned.



(a) Initial layout (b) Selection (c) Align

Figure 3.12: Example: `Align to Widget` via the GSME-Toolbar

Swap Widgets is a useful feature, which changes the position of two selected `Widgets`. This feature requires the selection of exactly two `Widgets`, which must have the same `Container`. The best result can be achieved, if the `Widgets` have the same size, so that no automated move or resize operations are necessary to free required space. This would result in a, maybe undesired, change of the layout. The size of the `Widgets` to swap is not changed.

3.2.2 Style Customizations

Style-customizations change the appearance of a `Widget` (e.g., border, color, font). Supported style-customizations are a subset of the styles specified in CSS 2.1, but not all of them can be reflected by the graphical editor, see Table 3.2. For example, if the selected border-style is inset, the GSME shows a solid bordered `Widget`. if the selected value fore vertical text alignment (`vertical-align`), is *baseline*, the GSME shows the text bottom-aligned. The supported values of `vertical-align` is just a subset of its CSS specification. The values *sub*, *percentage* and *length* are not available.

Table 3.2: Style properties that can not be displayed correctly by the `Screen`-based diagram

Style-Property	Value	displayed
border-style	inset	solid
	groove	solid
	outset	solid
	double	solid
	ridge	solid
Text: vertical-align	baseline	bottom
	sub	n.a.
	text-top	top
	text-bottom	bottom
	percentage	n.a.
	length	n.a.

The following is divided in the initiation of the change from our custom CSS-Property-page, the standard appearance property-page and the appearance toolbar.

3.2.2.1 CSS Property-Page

The CSS Property-Page in Figure 3.13 shows the supported style-customizations. In addition to the standard appearance styles, the GSME supports CSS-styles like text-alignment (horizontal and vertical), margin, and padding.

Some widget-style combinations are not supported by Web-Browsers, so they are also prohibited by the GSME. Table 3.3 shows these combinations, e.g., a `TextBox` has a defined border and a background-color, which cannot be changed. A `ComboBox` does not even show styles like font-size or font-color.

3.2.2.2 Appearance Property-Page and Toolbar

Figure 3.14 shows the appearance property-page, which is a standard Eclipse plug-in. It supports common style changes like fonts, colors and the border-style. The `Appearance Toolbar` (Figure 3.1(1)) provides the same possibilities to change the styles. Prohibited styles are grayed out.

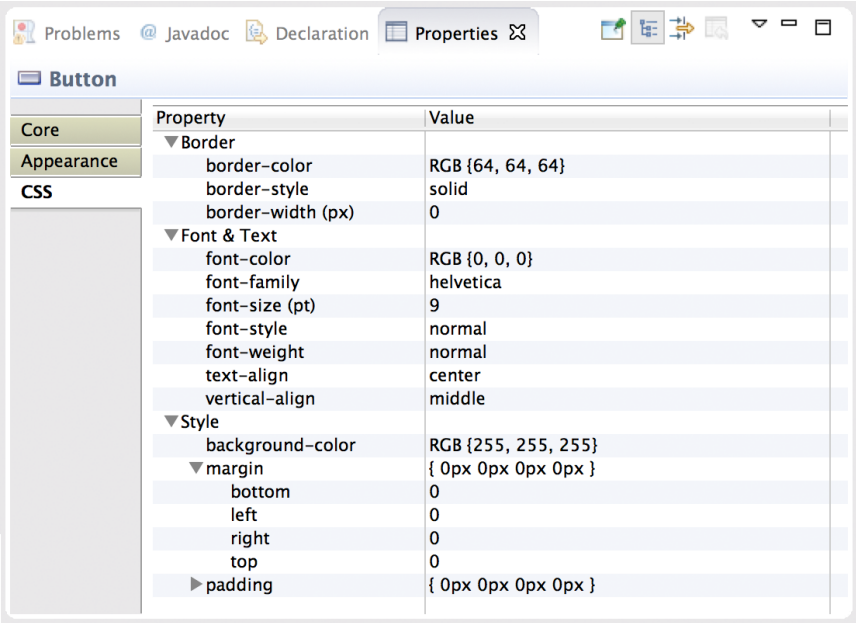


Figure 3.13: Property-Page CSS

Table 3.3: Widget-Style combinations not supported by Web-Browsers

Widget	Style
TextBox	border background-color dimension
ComboBox	border background-color dimension font-size font-style font-weight font-color
Button	border background-color

3.2.2.3 "Apply CSS" on the GSME-Toolbar

The GSME-Toolbar offers the menu-entry `Apply CSS`, which updates the actual styles of the selected `Widget` and its children recursively. In the development phase, it was an important menu-entry for testing and debugging purposes.

3.3 Software Design

The software design of the GSME is based on the design of EMF and GMF projects. The GSME consists of several plug-ins, containing the sources of the domain-model, the tree-view, the diagrams as well as a custom plug-in.

Figure 3.14: Property-Page `Appearance`

E1 – **The org.ontoucp.structuralui.model** plug-in contains the automatically generated sources of the `Structural UI Meta-model` (see Chapter B) as well as manually added helper-classes.

E2 –**The org.ontoucp.structuralui.edit** plug-in contains an `ItemProvider` for every `Structural UI Meta-model` element. `ItemProviders` are adapters, used to support the presentation of the model objects in the viewers.

E3 –**The org.ontoucp.structuralui.editor** plug-in consists of the sources of a fully functional Eclipse Editor (i.e., `Tree-View`) and a wizard for creating a new model-file (i.e., graphical Screen Model).

G1 – **The org.ontoucp.structuralui.digest.diagram** plug-in contains the sources of the overview-diagram depicted in Figure 3.1.

G2 – **The org.ontoucp.structuralui.diagram** plug-in contains the core part of this work, the `Screen`-based diagram. It contains the generated diagram-basis, templates to adapt this basis and further source-files for extending the generated diagram-view.

C1 – **The org.ontoucp.structuralui.diagram.custom** plug-in contains the toolbar-menu and the corresponding event-handling.

The Plug-ins E1 to E3 are generated by the EMF. Adaptations of these plug-ins, have been done directly in the files by adding the annotation *@generated not* and in separate files. If the `Structural UI meta-model` is changed (e.g., by adding a new type of `Widget`), these plug-ins must be re-generated, whereas the adaptations are kept.

The Plug-ins G1 and G2 are generated by the GMF. Adaptations have been done by adding templates, which define super-classes (e.g., for all the `EditParts` of `Containers`) and `Widget`-specific methods. These templates are used at diagram-generation-time, to create the source-code of the `EditParts`. Additional adaptations have been done by adding files. After changing the `Structural UI meta-model` and re-generating the plug-ins E1 to E3, the latter described plug-ins G1 and G2 must also be re-generated to reflect the changes in the diagrams. All adaptations are kept, but in case of adding a `Widget`, adaptations in a template-file to show an individual diagram-figure is necessary.

The plug-in C1 is created manually, it is not touched by a diagram-generation process.

3.3.1 Layout Features

This subsection shows details about the implementation of customizations, which are supported by the `Screen`-based graphical editor. The separation between *layout* and *style* continues in this section, although it cannot be split strictly, because layout-relevant data is not only persisted in the model, but also in the CSS-file. This is necessary to provide sufficient information for reflecting the layout-customizations in the final GUI. In 3.3.1 we describe details about supported layout-features and their implementation. In 3.3.2 we provide information about the style-features and how they are implemented.

3.3.1.1 Supported LayoutManagers

A `Layout` consists of a `LayoutManager` as a property of a container and `LayoutData` as a property of its children (e.g., `Widgets` in the domain of our Structural Screen Model).

The Screen Model uses a `GridLayout` to define position and size of the `Widgets`. The `GridLayoutManager` consists of the number of *columns* and *rows* and the lists *colWidth* and *rowHeight*, describing the size of the cells in pixels. `GridLayoutData` contains *column* and *row*, which reflect the position of the child. The values in *colSpan* and *rowSpan* reflect the used space in the granularity of a column or a row, which provide information to calculate the `Widget's` size. There are a few additional parameters in the `GridLayoutData`, which are omitted in this work, because they where not used [Ran14].

An XYLayout consists of an XYLayoutManager as a property of the container and XYLayoutData as a property of its children. XYLayoutData consists of the x and y values, which reflect the coordinates of the position in pixels, relative to its parent (i.e., container).

The GSME diagram uses an XYLayout with one pixel as the smallest unit to create the layout for a given `Screen` (i.e., to draw the figures, which represent the `Widgets` of the Structural UI model). Using the XYLayout in the GSME instead of, a grid-based layout, was driven by the following rationale:

- The position of a specific figure (i.e., `Widget`) can be determined directly without the need of additional calculations.
- The XYLayoutData allows for easy layout verification.
- The `Structural UI meta-model` (see Appendix B) also supports an `XYLayout` to layout the `Widgets` within their `Containers`.
- The `XYLayout` is a common denominator, where all other layouts of the `Structural UI meta-model` can directly be calculated and vice versa.
- The diagram-canvas uses *draw2d* to visualize figures. *Draw2d* also supports a GridLayout, but this is not compliant with the more powerful `GridLayout` specified in the `Structural UI meta-model`.

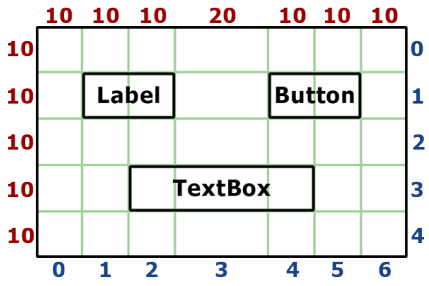
For visualizing the `Widgets` (i.e., drawing the corresponding figures) in the `Screen`-based diagram, it is necessary to calculate the positions (i.e., the XYLayoutData) from the `GridLayout`.

To persist a layout customization from the `Screen`-based diagram into the Structural Screen Model, the new `GridLayout` of the `Widgets` must be recalculated from the XYLayoutData and the size of the corresponding figures (and the size of their parent-figures) from the diagram. This calculation is described below:

- Store all x-values (x and $x + w$) of all direct children in a `List` (*colWidths*), avoiding duplicates.
- Add the values 0 and the `Container.`*width*, if not existing.
- Sort the values in ascending order.
- Calculate the *column* and *colSpan* values as follows. Iterate over the *colWidths* and compare with the left-bounds of the children. On a match, store the *column* and compare the next values of *colWidths* with the right bound of that child, incrementing the *colSpan* for every non-match. If the x-value from the *colWidths* equals the right bound ($x+w$) of the `Widget`, the *colSpan* is determined. Store the values in the `GridLayoutData` of the corresponding `Widget`.
- Calculate the distances between the values in the *colWidths* and store them in the `GridLayoutManager.`*colWidths*.
- Store the number of columns in the `GridLayoutManager.`*columns*.

The row-values can be determined analogously from the y-values, which result in the complete `GridLayout`.

An example is depicted in Figure 3.15, where the left side depicts the laid out `Widgets` including information about both layouts. The green grid visualizes the granularity of the `GridLayout`, but we use the same grid for the `XYLayout` and show the corresponding x and y values in red. The blue values enumerate the columns and rows of the `GridLayout`. The tables in Figure 3.15 present the corresponding `LayoutManagers` and `LayoutDatas` of this simple example.



XYLayoutManager	Container
width	80
height	50

XYLayoutData	Label	Button	TextBox
X	10	50	20
Y	20	10	30
width	20	20	40
height	10	10	10

GridLayoutManager	Container						
cols	7						
rows	5						
colWidths	10	10	10	20	10	10	10
rowHeights	10	10	10	10	10		

GridLayoutData	Label	Button	TextBox
column	1	4	2
row	1	1	3
colSpan	2	2	3
rowSpan	1	1	1

Figure 3.15: The relation between `XYLayout` and `GridLayout`

The `Structural UI Meta-model` also specifies an `XYLayout`, which can be used by the `Structural Screen Model`. The GSME is not prepared for that, because it was not implemented in the UCP, but it could be a trivial extension to implement support for it.

3.3.1.2 Resize Features

Following, the implementation of the resize-features is presented. They are structured according to their types of initiation. Details about the initiation are shown in 3.2.1.1.

Direct Resizing

This feature implements resizing a `Widget` through direct manipulation The implementation of this feature is based on the communication chain in Figure 2.6 above, in this case the `Request Creator` is the drag-handler, which creates a `ChangeBoundsRequest`. It is a standard `Request` from GMF, which is used to alter size and position. We extended the `XYLayoutEditPolicy` to check the constraints from Table 3.1, calculate the resulting layout and return the appropriate Commands. The `ContainerXYLayoutEditPolicy` of the `Widget's` parent accepts this `ChangeBoundsRequest` and the new `LayoutManager` and the `LayoutData` of its children are calculated. These new values are stored in the corresponding `Commands` (`SetWidgetBounds` and `SetContainerBounds`), which are returned and executed on the `Command Stack`. After execution, the corresponding `EditParts` get notified about the changes, which initiate refreshing the visuals with the new values.

A second part of this feature is the resize feedback, which is shown during direct resizing (Figure 3.5 (a)). For that, we created the `FeedbackEditPolicy`, which receives the `ChangeBoundsRequests` and shows a rectangle with the new size and position, overlaying the `Widget` on a so called *FeedbackLayer*. To show the corresponding feedback above the `Widget`, a coordinate transformations to absolute and calculations to translate and resize the shape according to the `Request`, are necessary.

Resizing by use of the GSME Pop-up bar

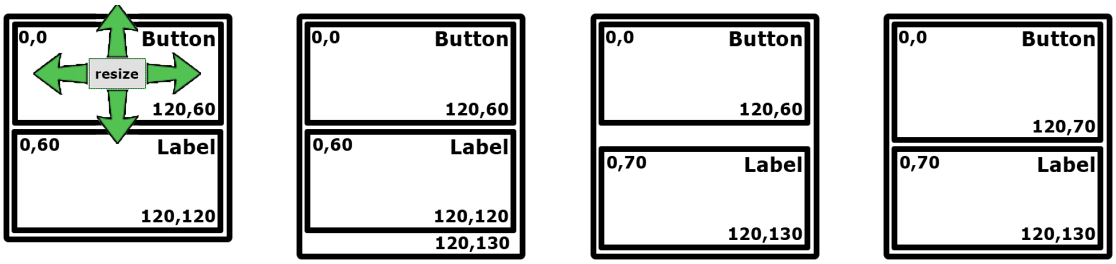
This advanced feature implements the customization Paragraph Resizing by use of the GSME Pop-up bar including many automatic steps involved. Let us illustrate this feature using a simple example.

The initial layout in Figure 3.16 (a) shows a `Panel` with the size of 120x120 pixels and its children, a `Button` and a `Label`. In this example, the designer wants to increase the height of the `Button` by 10 pixels, but there is no space between this `Button` and the `Label` below. Also the `Container` is not big enough for the new size.

If the designer did it by direct resizing, she would start with extending the `Panel` (b), moving the `Label` (c) and finally stretching the `Button` (d). The GSME starts with trying to stretch the `Button` by creating an `AutoSize-Request`, detects a violated constraint before execution, and solves the issue by creating the next `Request`, and so forth. The execution of the `Commands` is done in the opposite sequence (i.e., the same sequence as the designers').

The green arrow "down" of the GSME Pop-up bar sends the `Autosize-Request RQ1` *extend the height by 10* to the `EditPart` of the `Button`. The `EditPart` queries all its `EditPolicies`, until the `AutoSizeEditPolicy` accepts the `Request` and constructs the `Command` (compare Figure 2.6) for updating the model of this `Button`.

First of all, the `AutoSizeEditPolicy` determines the validity of the `Request`. On success, the requested changes are checked against the constraints defined in Table 3.1:



(a) initial layout (b) extending parent (c) moving sibling (d) extending `Button`

Figure 3.16: *Resizing* a `Widget` using the GSME Pop-up bar

1 – A Container must wrap its children: if the `Widget` is a `Container`, its size must not be smaller than the space required by its children with the given layout. If this constraint is not satisfied, the `Request` cannot be handled at all. In our case, the `Widget` is a `Button`, which cannot contain other `Widgets`.

2 – A Widget must be within its parent: the new bounds (i.e., a rectangle with the upper left edge on x, y and the width and height of the `Widget`) must not exceed the size of its `Container`. If this is not fulfilled, the `Container` must be resized to satisfy `constraint 1`. For that, a new `AutoSize-Request` must be created and handled by the `EditPart` of the `Container`. The execution of the previous `Request` must wait, until this is executed and the models have been updated. In our example, the new bounds of the `Button` and the bounds of the `Label` do not exceed the size of the `Panel`, so no additional `AutoSize-Request` is necessary.

3 – A Widget must not overlap with siblings: this constraint determines, if the layout changes would lead to an overlapping of `Widgets` within the `Container`, which compromises the Screen Model. In that case, the siblings of the `Widget` being resized, are expected to free the space by moving in the direction with the smallest necessary movement. A new `AutoSize-Request` is created to achieve this. In our example, the `Request RQ2` *increment the y-value by 10* is created for the `Label`. The initial `Request` must wait until the models are updated, and is stored in the HashMap `extendedData` with the key "nextRequest" of the newly created `Request`.

The `AutoSize-Request RQ2` for moving the `Label` by 10 pixels down, must also go through the procedure described above. It would violate `constraint 2`, because the new bounds of the `Label` exceed the size of the `Panel`. Another `AutoSize-Request RQ3` (i.e., *extend the height by 10*) is created with the previous one stored in its HashMap. This `Request` is successfully checked against the presented constraints, i.e., all necessary `Requests` for resizing the `Button` have been created.

Before we continue with the description of creating the corresponding `Commands`, we explain a small difference in the communication chain, shown in Figure 2.6. It matches exactly with the first `Request`, created by the GSME Pop-up bar until it is forwarded to the `EditPolicy`. Subsequent `Requests` are created directly within our `AutoSizeEditPolicy`, while checking the constraints in a recursive manner. The sequence of the `Requests` is inverted, because the necessary change resulting from a constraint, is a prerequisite for the previous `Request`. The returned `Command` (Figure 2.6) corresponds to the most recently created `Request`. Figure 3.17 depicts the created `AutoSize-Requests` in the execution order, containing the next `Request` as described above.

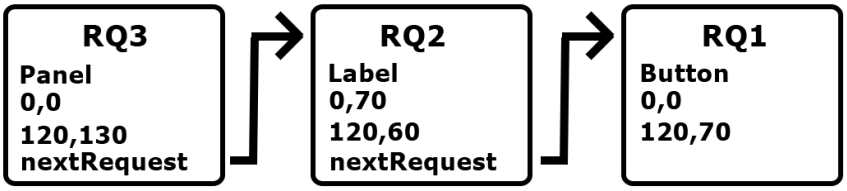


Figure 3.17: Inverted order of the chained Requests

In the next step, the GSME has to recalculate the `LayoutData` of the `Widget` to alter, the `LayoutData` of its siblings and the `LayoutManager` of their `Container`. Our example does not show a `Container` for the `Panel`, we define it now as the `Screen`, to describe the following steps. Figure 3.18 depicts the `Commands` (e.g., `SetWidgetBounds`) resulting from the previously created `Requests`. Every solid-lined box represents a `Command`, the `Commands` in a dashed-lined rectangle (e.g., RQ3) are chained together and put on the `Command-Stack` at once. The left column of Figure 3.18, labeled as RQ3, shows the `Commands` to update the `LayoutData` (blue) of the `Panel`, the `LayoutManager` (green) of the `Screen` and the `ScheduleNextCommand`, which carries the `Request RQ2` and the parameter *waitingFor* from the type `EditPart`. This is the `EditPart` (i.e, `Screen`), whose model must have been updated with the layout changes before `RQ2` is allowed to be handled. We usually use the `Container`, where we update the `LayoutManager` as the *waitingFor* part.

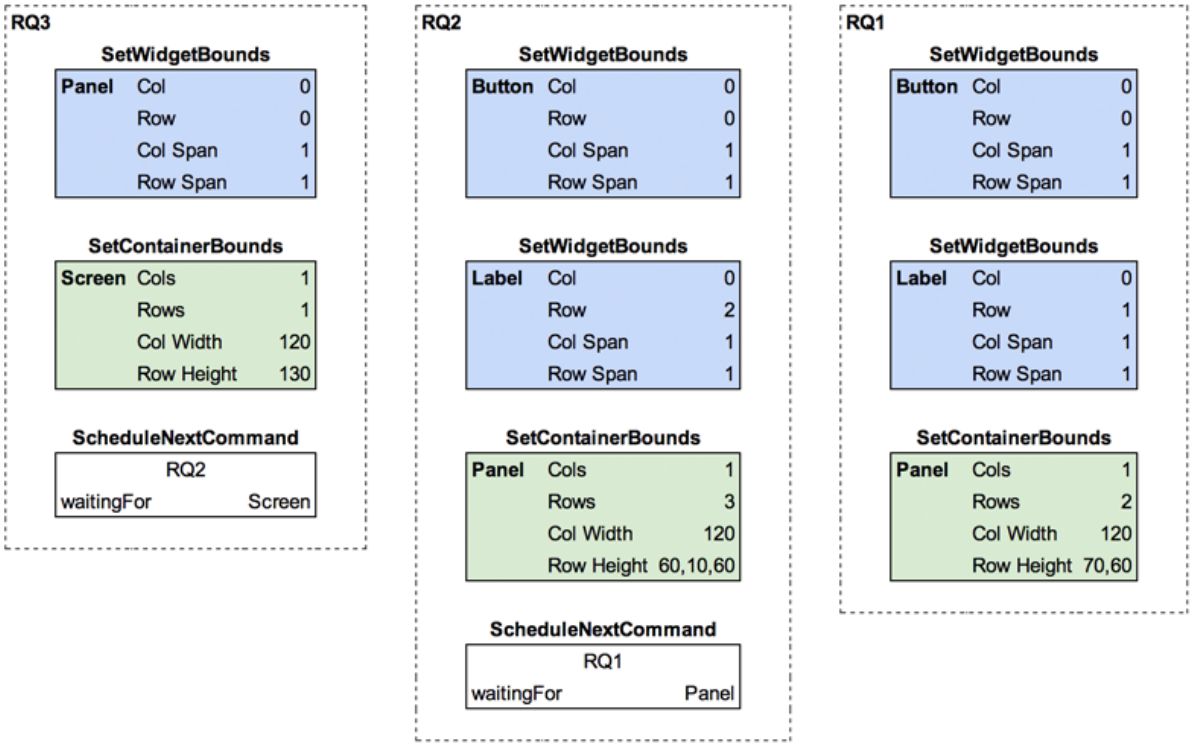


Figure 3.18: `Commands` resulting from the `Requests`

Figure 3.19 shows a UML Sequence Diagram executing the chained `Commands` from `RQ3`, which are returned from the `AutoSizeEditPolicy`. The `EditPart` puts them on the `CommandStack` for execution. The `SetWidgetBoundsCommand` and the `SetContainerBoundsCommand` update

the model directly, whereas the `ScheduleNextCommand` does not alter the model. It uses the method `addPendingRequest` of the `CommandScheduler` to store the *waitingFor* `EditPart` with the `Request`. The data is stored in the `HashMap<EditPart,Request>` *pendingRequests*, with the `EditPart` as the key and the `Request` as its value. Back to our example, after calling `addPendingRequest`, the HashMap contains the entry `<Screen, RQ2>`. When a model is altered by a `Command`, it notifies the `EditPart` about the changes. The `EditPart` of our `Panel` gets informed about changed `LayoutData` and its corresponding changes of the `Width` and `Height` attributes. This initiates an update of the `Panel's View`, reflecting the new values. The `EditPart` of our `Screen` gets notified about changes in the `LayoutManager`, refreshes its visuals and notifies the `CommandScheduler` with *notifyEditPartFinished* about the update. The `CommandScheduler` requests the stored value for the key *Screen* from the *pendingRequests*, which returns the `Request` `RQ2`. The `Commands` of `RQ2` are retrieved from the `AutoSizeEditPolicy`, which returns the middle column of Figure 3.18. During the execution, a `ProgressMonitor` is shown with details about the `Command` being executed.

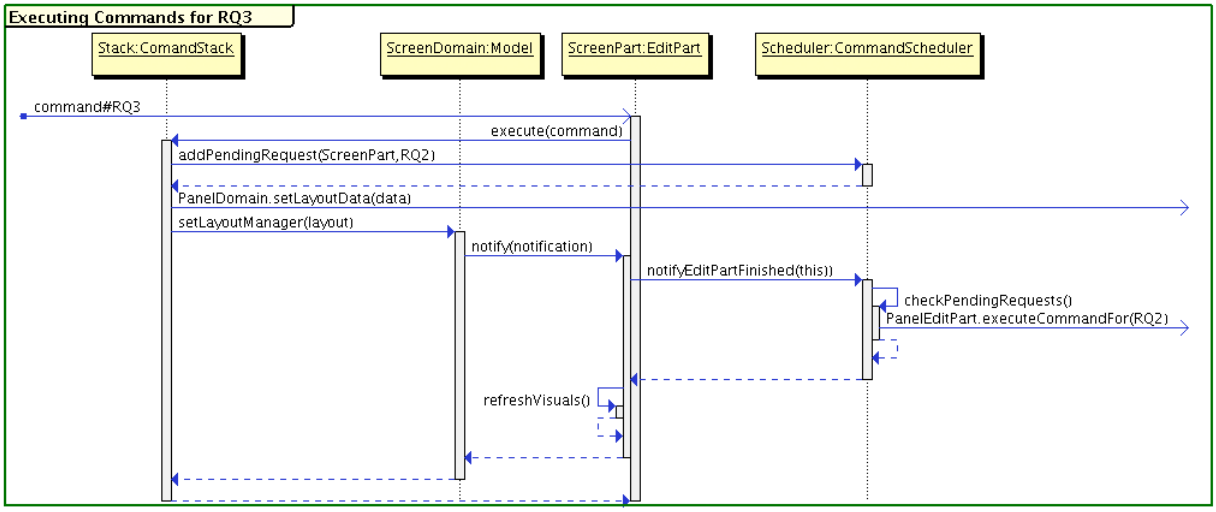


Figure 3.19: Executing the `Commands` of `Request` RQ3

Termination Conditions. The recursive creation of the `AutoSize-Requests` to free the required space is either terminated by the fact that there is enough space for the current `Request` or the conditions itemized below.

The given fixed size of the Frame is defined by the device-specification. If the required space of a given `Request` exceeds the size of the `Frame`, it is terminated by `constraint 1` of Table 3.1.

The SkipZone is a rectangle, which is stored in an `AutoSize-Request`. With every move or resize, the `SkipZone` grows by calculating the bounds of a new rectangle that contains the `SkipZone` and the new bounds of the `Widget`. This new rectangle is the new `SkipZone`. In subsequent `Requests`, the `Widget` is not allowed to be placed overlapping its `SkipZone`. This avoids the potentially infinite swapping of two `Widgets` to free space for each other.

An Obstacle is a `Widget` blocking the `Request` and must, therefore, be moved or resized, to free space for the given `Request`. This `Obstacle` must not be a `Widget` being involved in the

given `Request`. This case would lead to layout-changes that require a different algorithm to calculate the necessary `AutoSize-Requests`.

These conditions are detected before executing the initial `Command`. Undoing the executed `Commands` is also supported, to go back to the initial layout.

Resizing by use of the Core Properties View

Changes of the `Width` or `Height` from the `Core Properties View` are checked immediately after applying the new value. This check is performed by a so-called *validation provider*, which is based on the Eclipse Validation Framework. Strictly speaking, such a *validation provider* performs a verification. It contains the Adapter-class, extended from `AbstractModelConstraint`, which must override the abstract method validate(IValidationContext ctx). This parameter contains information about the selected `Widget` and the changed property with the new value. With this information, the GSME checks the constraints from Table 3.1 by creating a new `AutoSize-Request` and checking the integrity of the Screen Model with the resulting layout. If it fails, the dialog presented in Figure 3.7 is shown and the values are reset to the previous ones.

If the integrity check is positive, the `EditParts` of the involved `Widgets` get notified to redraw their visual representation.

Resizing by use of the Toolbar Menu

The result and so the implementation of the two following *resize-features* is quite different. The first one "Resize" uses the surrounding `Widgets` and bounds to calculate the resulting size (i.e., new size is defined by the first violated constraint), while the second one "Resize with Widget" creates a `Request` with a predefined size and frees the space if necessary.

Resize – The implementation of this feature is very simple. The GSME extends the bounds of the `Widget` by one pixel and checks the constraints from Table 3.1 in a loop, until a constraint-check fails. The last valid bounds are used to create a `ChangeBoundsRequest`. Our `ContainerXYLayoutEditPolicy` is used to check the constraints of the resulting layout and return the appropriate `Commands`. These are a `SetContainerBounds` for the parent and `SetWidgetBounds` for the `Widgets` (i.e., all children of the parent), similar to the ones presented above. This feature does not free space, it just fills empty space by extending the selected `Widget`. When the sub-menu `MAX` is selected, extending in the single directions follows a sequence (i.e., priority), which is *left, right, top and bottom.*

Resize depending on a Reference Widget – This feature requires a multiple selection of at least 2 `Widgets`. The initially created `Request` is an `AutoSize-Request` to resize the first to the $n-1$ selected `Widgets` to the size of the n^{th} selected (i.e., the `reference Widget`). If there is not enough available space for the resize operations, the GSME tries to free the space with subsequent `AutoSize-Requests` as described above. If there are many selected `Widgets` to resize (i.e., extend), they might mutually prevent a successful operation. The reason is the *Obstacle* presented in Subparagraph Termination Conditions on page 31.

3.3.1.3 Move Features

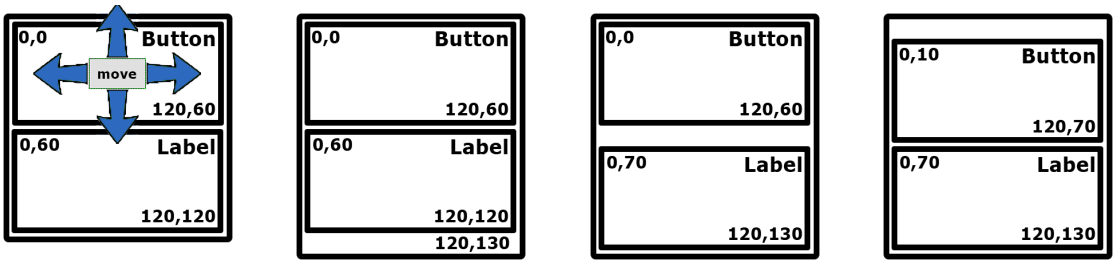
Moving with direct manipulation

The implementation of this feature is similar to the one presented in Paragraph Direct Resizing. A minor difference is in the implementation of the `FeedbackEditPolicy`, where the coordinates x and y are displayed, i.e., the left top position relative to the moving `Widget's` container.

Moving by use of the GSME Pop-up bar

A detailed description of the resizing-counterpart of this feature can be found in Paragraph Resizing by use of the GSME Pop-up bar. We used a simple example with just a few `Widgets`. Following, we provide a very similar example, where just the initial `Request` differs.

Example Moving by use of the GSME Pop-up bar Figure 3.20 (a) shows the initial layout of our example, with the GSME Pop-up bar. By clicking on the blue arrow "down", the `AutoSize-Request` *move down by 10 pixels* is created, and delegated to the `EditPart` of the `Button`. The subsequently created `AutoSize-Requests` are equal to RQ2 and RQ3 of the example above.



(a) initial layout (b) extending parent (c) moving sibling (d) moving `Button`

Figure 3.20: *Moving* a `Widget` using the GSME Pop-up bar

Moving by use of the Toolbar Menu

The GSME Toolbar (Figure 3.2)(1) consists of three `move-features`, where `Align` moves the `Widget` to the next obstacle in the specified direction, using the empty space. `Align to Widget` moves the `Widget's` edge to that of a reference, freeing the space if necessary. `Swap Widgets` exchanges the position of two widgets.

Align – The implementation of this feature is very similar to its *Resize*-counterpart from Paragraph Resizing by use of the Toolbar Menu. The `Widget's` bounds are moved by 1 pixel in the specified direction in a loop, until a constraint from Table 3.1 fails. It is moved, until it "touches" another `Widget` or the `Container`.

Align to Widget – The counterpart of this feature is *Resize depending on a Reference Widget*. Multiple selection is necessary, to provide a moving `Widget` and the `reference Widget`. After selecting an edge to align to, an `AutoSize-Request` is created, which contains all

moving Widgets an their new bounds. The GSME checks the resulting layout against the constraints in Table 3.1 and tries to free the required space if necessary.

Swap Widgets – To use this feature, it is necessary to select exactly two Widgets, which must have a common container. Widgets are not allowed to be "reparented", this would violate the integrity of the Screen Model. First of all, these prerequisites are checked. On success, the AutoSize-Request is created, containing the Widgets and their new bounds. By selecting two Widgets, the new bounds of each other are known. Depending on the position and size of the Widgets, it is possible that a termination condition, presented in Subparagraph Termination Conditions on page 31, cancels the execution in advance.

3.3.2 Style Features

The implementations of the different style features have individual sources (i.e., Request Creators, see Figure 2.6). Some features, i.e., *font-color*, can be changed on three places: via the CSS Property-Page, via the appearance Property-Page, and via the appearance Toolbar. The created PropertyChangeRequests to retrieve the Commands are equal. *NotationalListeners* notify the corresponding EditPart about the changed property, which persists the new value and updates the Views.

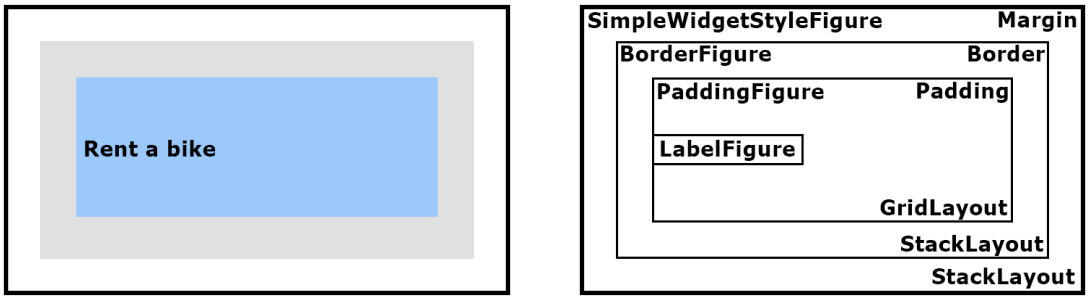
3.3.2.1 The Views

In the GMF diagram, the Widgets are represented through Figures (see Figure 2.4), which visualize the Screen Model with the supported CSS-styles (Figure 3.13).

Figure 3.21 (a) depicts the structure of a simple Label, where the outer frame represents its Container. The corresponding Figure is the SimpleWidgetStyleFigure, which implements the interface *IWidgetFigure* declaring a set of methods, used by their EditParts. The SimpleWidgetStyleFigure defines the Margin (i.e., the space outside the border) of the Label. It uses a StackLayout, which allows stacking a child on the parent (or on top of the previously added child). It adds the BorderFigure as its child, which reflects the CSS-border (i.e., border-width, border-style and border-color). The BorderFigure also uses the StackLayout to layout its child, the PaddingFigure. This figure keeps the specified space inside the border free (i.e., padding) and is filled with the background-color of the Label. Furthermore, it is responsible to layout the "real label" (i.e., the text) depending on the text-alignment in the horizontal and vertical direction. We use a GridLayout with the corresponding properties to achieve this. The LabelFigure extends the *WrappingLabel*, which automatically handles displaying of long text and supports CSS-fonts (i.e., font-color, font-family, font-size, font-style and font-weight).

The example in Figure 3.21 (b) shows a Label with the text "Rent a bike". The text is horizontally aligned to the left, and vertically aligned to the middle. There is no padding specified, the background-color is light blue. A thick, solid, light-gray border is visible and a margin in all directions is shown.

There are several Figures for the different Widgets, but their structure is very similar. E.g., the Button uses a SVGFigure instead of the BorderFigure, which shows the image of a typical button.



(a) Visualized `Label` including style-attributes

(b) Detailed layout of the `Label`

Figure 3.21: Visual representation of a `Label` through its `Figure`

3.3.2.2 CSS Property-Page

This property page (Figure 3.13) provides an overview about all supported style-customizations. It consists of several classes, where the *CSSPropertySource* implementing the *IPropertySource* does most of the work. It defines the property descriptors, which can be seen as the lines in the property page. Property descriptors can be instantiated from different specializations (e.g., *TextPropertyDescriptor*, *ComboBoxPropertyDescriptor*). For the properties `Margin` and `Padding`, a *PropertyDescriptor* containing a nested *PropertySource*, the *CSSArrayPropertySource* with a property descriptor for every direction (`bottom, left, right` and `top`), was created.

Changes of a style-property creates a `PropertyChangeRequest`, which notifies the corresponding `EditPart` to handle the update.

3.3.2.3 Appearance Property-Page and Toolbar

These two possibilities for changing the styles, create `PropertyChangeRequests` directly. The corresponding `EditPart` gets notified and handles the update.

3.4 Requirements Satisfaction

In the following, we evaluate the satisfaction of the Requirements that are presented in Section 3.1.

RQT1 – The GSME shall generate a graphical representation of `Widgets` (i.e., a diagram) for *.structuralui files.

This Requirement is fulfilled through both diagrams of the GSME. The Overview diagram presents all the `Screens` and the `Screen`-based diagram presents the content of the `Screens` (i.e., all their `Widgets`). The information is used from the Screen Model (i.e., the *.structuralui-file).

RQT2 – The GSME shall use the layout information specified in the Screen Model for the graphical representation.

This Requirement is fulfilled, because the `Screen`-based diagram of the GSME uses the `LayoutManager` of the `Container` and the `LayoutData` of all `Widgets` to provide a layout-correct graphical representation of the `Screens`.

RQT3 – The GSME shall use the style information specified in referenced CSS-files for the graphical representation.

This Requirement is fulfilled, because the GSME decorates the `Widgets` using a referenced set of CSS-styles, where the id of the CSS-style is annotated in the model, and the definition of the style is specified in a CSS-file (its location and name is part of the Screen Models content).

RQT4 – The GSME shall use only information specified in the Structural Screen Model, including referenced information (e.g., style-references), for the graphical representation (i.e., without information from an application back-end).

This Requirement is fulfilled, because the GSME visualizes the `Widgets` without the need of an application back-end. Static data that is defined in the Screen Model, is displayed in the `Widgets`. Dynamic data that is usually provided by the back-end in combination with previous user-input, is replaced with other information to help the designer identifying the `Widget`.

RQT5 – The GSME shall make manipulations (e.g., layout, size, text, color, etc.) in the diagram persistent in either the *.structuralui file or an additional CSS-file.

This Requirement is fulfilled, because using the features, presented in Section 3.3, may alter layout and style of the Structural Screen Model. Layout-changes are persisted directly in the Screen Model and style-changes are persisted in a newly created CSS-file.

RQT6 – The GSME shall provide support for restricting/constraining the allowed manipulations (e.g., a `Widget` can only be laid out anew within its `Container`).

This Requirement is fulfilled, because the GSME restricts manipulations, layout-constraints are defined in Table 3.1. Style constraints depend on the type of `Widget`. They are specified in Table 3.3.

RQT7 – The GSME shall be compatible with Eclipse Validation Framework Live Constraints.

This Requirement is fulfilled, because due to the choice of using GEF/GMF, the Eclipse Validation Framework is supported. the GSME uses its Live Validation when the designer alters the width or height value in the properties view.

RQT8 – The GSME shall be regenerate-able (i.e., all code modifications shall be persistent, for new generation).

This Requirement is fulfilled, because adaptations on the GSME diagrams are persisted in severals ways, like described in Section 3.3. Therefore, the code of the GSME is robust against re-generation.

RQT9 – The GSME should support custom figures for displaying the `Widgets` to make the diagrams more “fancy”.

This Requirement is fulfilled, because the `Widgets` have individual figures. `ComboBoxes` and `Buttons` are given a realistic representation, to get an immediate impression about the resulting `Screen`.

RQT10 – The GSME should provide an interface for reporting layout customizations.

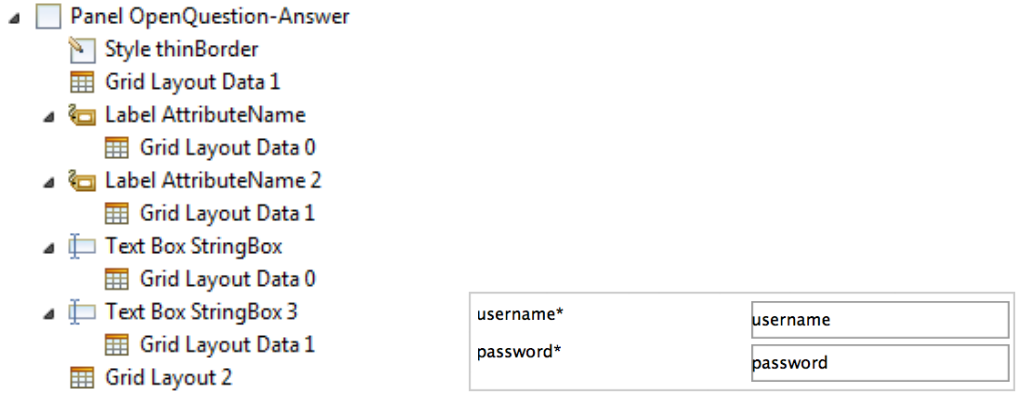
This Requirement is fulfilled, because an interface to persist model-changes was implemented, where the source of initiation (e.g., Pop-up bar, Toolbar), the trigger (manually or automatically) and the serialized Request-Object is stored in a `ChangeLog`.

4 Results

The main result of this diploma thesis is the prototypical implementation of the GSME in form of Eclipse plug-ins, which satisfies all requirements specified in Section 3.1. In particular, we present a visualization example and a customization example for a comparison between the GSME and the Tree Editor. Furthermore, we present the results of computational performance measurements of certain features of the GSME.

GUI visualization and customization: GSME vs. Tree Editor

Figure 4.1 shows the visualization of a Structural Screen Model excerpt using the GSME and the Tree Editor. This figure shows that the GSME visualizes the *Widgets* graphically with corresponding figures, the *Layout* and the *Style* in contrast to the Tree Editor. Thus, the GSME provides a visual impression of the final GUI, even without an existing back-end.



(a) Tree-view (b) GSME-diagram

Figure 4.1: Snippet of the Login-Screen of our running example

Visualizing the Structural Screen Model through the GSME means to visualize all the information the Tree Editor shows (the Structural Screen Model containing the `Widget`-hierarchy, layout information and style-ids). Additionally, the GSME visualizes the style-properties specified in a given CSS-file that are referenced through the style-ids. All visualized data is either static and specified in the Screen Model, or dynamic data, which will be provided by the application back-end at run-time. Such dynamic data is visualized based on information specified in the `Text`-attribute

of the Widget to visualize (e.g., the TextBox labeled "username" in Figure 4.1 (b)). If no such information is available, a place-holder (i.e., <...>) is inserted and the Widget can be identified through its properties, visualized by the Property-View. Visualization and Customization of this intermediate Structural Screen Model, allows for developing the GUI in iterations even without the need for an application back-end (see [RPK+14, RKP+14]). Furthermore, the GSME allows for decoupling the development of the GUI and the application back-end, which can in principle be developed in parallel.

In addition, the GSME facilitates *layout customizations*. Direct layout manipulations can be performed using the Tree Editor, too. Trivial manipulations that do not change the LayoutManager of the Container can be done in one step, even from the Tree Editor. An example is depicted in Figure 3.11, where a Button is laid out in the left column of its parent. It can be moved to the empty right column, which has the same size. From the Tree Editor, the Button can be moved by altering the *column*-attribute of the Button's GridLayoutData from 0 to 1. Using the GSME, there are several ways to move the Button. It can be done by direct manipulation with the mouse or the cursor-keys, or by the GSME Pop-Up bar. The most convenient way is to use the Align-feature of the GSME Toolbar.

Usually, a layout-change requires recalculation of the Container's LayoutManager and the LayoutData of all its children (see 3.3.1.1), because the empty target-space often differs from the required space. Using the Tree Editor, this layout-calculation has to be done manually and the resulting values have to be put in the corresponding fields of the LayoutManager and the LayoutData. This task is difficult and error-prone. The GSME facilitates such customizations through automating the recalculation of the layout.

Style customizations cannot be done by the Tree Editor at all, it just shows a reference-ID (e.g., thinBorder in Figure 4.1) for which the style is specified in one of the predefined CSS-files. The GSME reflects the specified style of the Widgets and supports their customization directly from the properties-view.

Computational performance of the GSME

We measured two types of computational performance. First, we measure the loading time required for visualizing (i.e., rendering) a given Screen. Second, we measure the execution time of resize-features. These measurements have been performed with a MacBook Pro, 2.6 Ghz dual-core Intel Core i5, 8 GB 1600 MHz memory, 256 GB PCIe-based flash storage.

The overview-diagram of the GSME allows for the selection of a Screen for visualizing its content in the Screen-based diagram. This rendering of a Screen (i.e., the creation of the Screen-based diagram) takes a certain amount of *loading time*. We define this *loading time* from initiating the task "open the Screen" until the new diagram with the corresponding Screen is completely visible to the user.

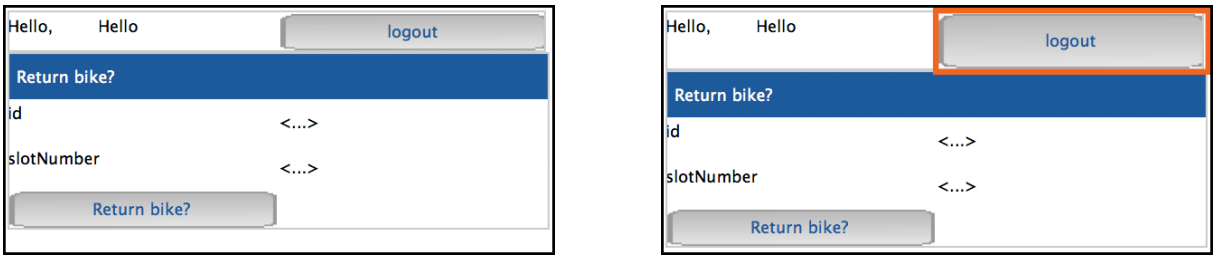
Table 4.1 presents three examples of Screens with different content-properties (i.e., nesting-depth and number of Widgets) and their loading times. The nesting-depth reflects the number of hierarchy levels.

The Screen *PartitioningState_1* (depicted in Figure 4.2 (a)) contains 13 Widgets in 3 hierarchy levels. *PartitioningState_87* (depicted in Figure 4.3 (a)) also contains 13 Widgets, but the maximum nesting-depth of this Screen is 5. The loading time of both Screens takes 7 seconds each. The more complex Screen *PartitioningState_13* visualized in Figure 4.4 (a) contains 36 Widgets

Table 4.1: Screen loading time

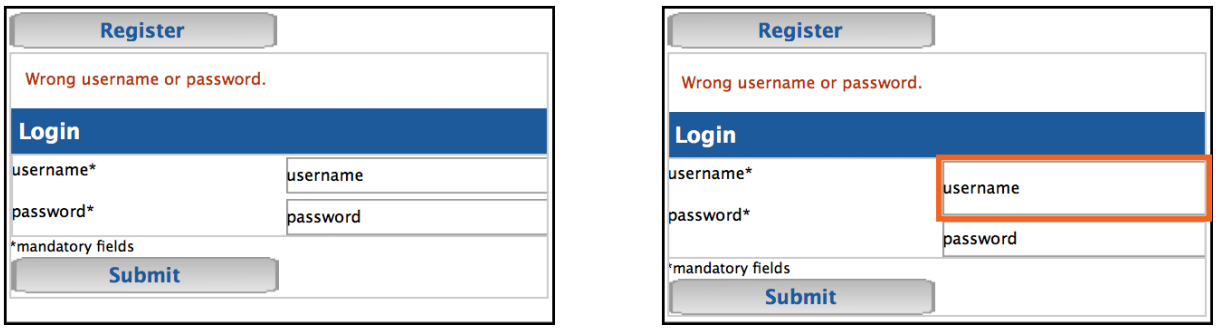
Screen	Maximum nesting-depth	Number of Widgets	Loading time
PartitioningState_1 (Figure 4.2 (a))	3	13	7s
PartitioningState_87 (Figure 4.3 (a))	5	13	7s
PartitioningState_13 (Figure 4.4 (a))	5	36	11s

and a maximum nesting-depth of 5 (like *PartitioningState_87*). This `Screen` needs 11 seconds to be rendered. These results indicate that the loading time depends more on the number of `Widgets` than on the maximum nesting-depth.



(a) Initial Screen (b) Customized screen after executed resize

Figure 4.2: `Screen` PartitioningState_1 of our running example

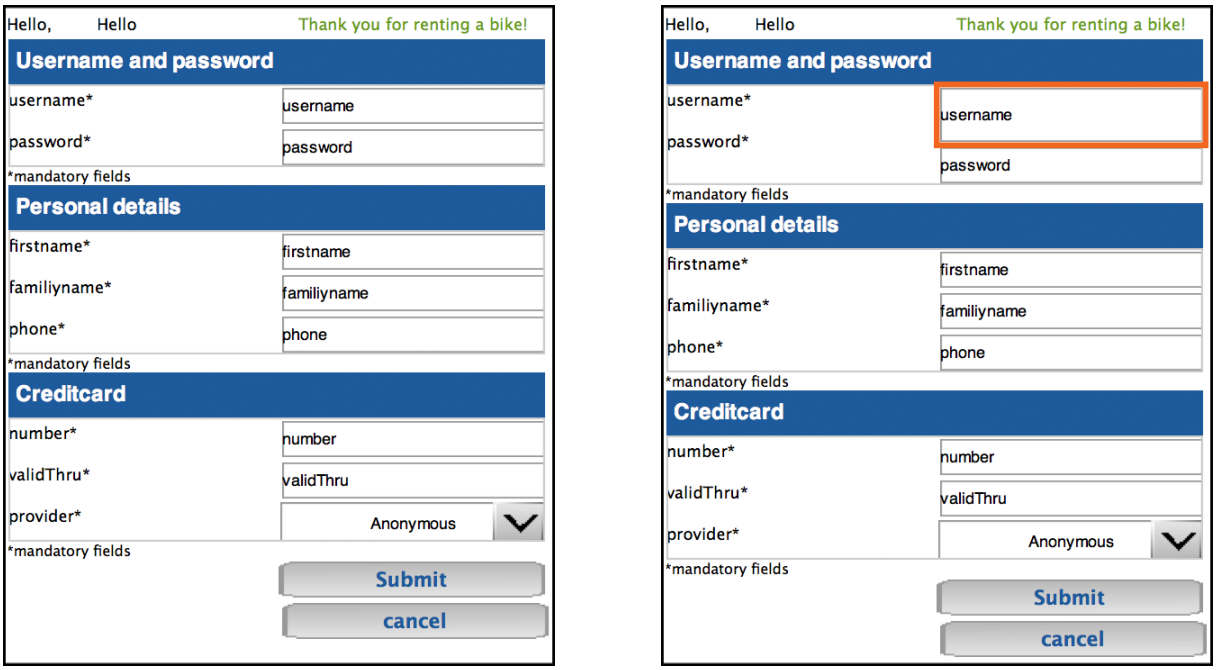


(a) Initial Screen (b) Customized screen after executed resize

Figure 4.3: `Screen` PartitioningState_87 of our running example

The GSME Pop-up bar provides access to the move and resize features, which can be used for layout customizations. These features automate the task "free the required space" if necessary. A simple example is presented in Paragraph Resizing by use of the GSME Pop-up bar on page 28, where one manual step with the GSME Pop-up bar triggers two additional steps to free the required space, which are automated by the GSME. The *execution time* for executing such a feature is defined from initiating the feature by the mouse-click, until the diagram shows the desired layout changes (e.g., the `Widget` with the new height).

Table 4.2 shows three examples of *resizing a Widget*, which need a different number of additional operations (i.e., move and resize) to free the required space. Figure 4.2 (a) visualizes the initial layout of the `Screen` *PartitioningState_1* and (b) shows the customized layout after executing the feature *increasing the height*. To increase the height of the `Button` in *PartitioningState_1*, one



(a) Initial Screen

(b) Customized screen after executed resize

Figure 4.4: `Screen` PartitioningState_13 of our running example

move operation and 3 resize operations are necessary, which take 1.8 seconds for execution.

Table 4.2: Execution time comparison

Screen	Nesting-depth	Operations		Execution time
		Move	Resize	
PartitioningState_1 (Figure 4.2)	3	1	3	1.8s
PartitioningState_87 (Figure 4.3)	5	3	5	3.9s
PartitioningState_13 (Figure 4.4)	5	6	5	6.3s

Increasing the height of the `TextBox` in *PartitioningState_87*, which is depicted in Figure 4.3 (a), takes 3.9 seconds and requires 8 operations. The resulting layout is presented in Figure 4.3 (b).

Applying the feature *increasing the TextBoxes height* on the more complex `Screen` *PartitioningState_13*, in Figure 4.4 (a), needs 11 operations and 6.3 seconds. Figure 4.4 (b) shows the layout after executed resizing.

5 Discussion & Future Work

This section discusses limitations of the current GSME implementation and highlights potential topics for future work.

Improving and extending the GSME

The GSME facilitates a preview of the final GUI without the need of an existing back-end application. A limitation of this preview is that it does not offer dynamic data, because this data can only be provided by the back-end application. However, the GSME offers static data (e.g., the title, labels, default-values), which is typically sufficient to provide a realistic preview of the final GUI.

Another limitation of the GSME is that it currently does not support the simulation of the GUI behavior, which was actually out of the scope of this diploma thesis. Simulating the GUI behavior would be a beneficial extension of the already existing GSME as it would allow for an early evaluation of the application. In principle, information about possible sequences and their initiation is specified in the `Behavior Screen Model` and already available. Pressing a `Button` can trigger opening a new diagram with the corresponding `Screen`. For that, a mode-switch could be implemented, to select between *editing* and *simulation* mode. One further limitation of the current implementation of the GSME in this context is that it takes a few seconds to render a `Screen` in a diagram view (see Chapter 4).

A potential solution for this performance problem would be to use Standard Widget Toolkit (SWT) directly, without the overhead of GEF/GMF. SWT is a library for creating GUIs in Java. The visualization of the `Widgets` could possibly be implemented more efficiently, but this approach would require additional effort for basic implementations like listeners to support drag and drop, providing information about the dragged part, the position, creating a feedback-figure, etc., to facilitate customizations. Additionally, the `GridLayout` of SWT uses different properties to specify the layout than the `GridLayout` of the Structural Screen Model. This means that layout specified in the Structural Screen Model needs to be mapped to an SWT layout (e.g., SWT GridLayout or XYLayout). The classes for transforming the Structural Screen Model `GridLayout` into an XYLayout, developed and implemented in this work, could be used in this solution as well.

Currently, the implementation of the layout transformation, considering the layout-constraints, is persisted in separate files of the plug-in `org.ontoucp.structuralui.diagram`. A similar calculation is used in several parts of the Unified Communication Platform (UCP). In future work,

these functions could be combined into a common part of the UCP, to facilitate its maintenance. For example, these calculations could be moved directly into the implementation of the Structural UI Meta-Model, because they just need the `LayoutManagers` of `Containers` and the `LayoutData` of the `Widgets`.

The computational performance of the GSME move and resize features depends on the number of operations (i.e., intermediate `Requests`), which are needed to free the required space. The execution time of these features consists of the time needed for the iterative constraint-checks, the creation of `Requests` and their `Commands`, and the execution of the `Commands` with updating the visuals (i.e., `Figures`) for every intermediate `Request`. The computational time could be reduced by calculating the finally resulting layout and creating one single `Request` that performs all the layout changes in one execution.

Customization persistence

As stated in [Ran14], the problem of customizing the `Structural Screen Model` on the CUI-level is that these customizations are lost in case of regenerating the `Screen Model`, because the previous Structural Screen Model and the CSS-file are overwritten. Thus all customization performed through the GSME are lost. Customizations can be made persistent if they are specified in the form of a transformation rule [KFK09, RPK13], because such customizations are applied while generating the `Screen Model` (see Figure 2.1). To additionally make customizations performed through the GSME persistent, such customizations need to be saved explicitly and re-applied again after the `Screen Model` has been re-generated. We already created a *Changes Model* to store the customizations with the corresponding `Requests` in a separate file [Ran14], which supports developing the GUI in full iterations, in principle. The drawback of this solution is that such customizations cannot be taken into account by UCP's automated device tailoring approach [Ran14], because they are applied only after the tailored `Screen Model` has been generated.

Customization propagation

GUI `Widgets` that trigger the same functionality are potentially contained in more than one `Screen` (e.g., a log-out `Button`). Customizations of such `Widgets` typically have effects on more than one `Screen`, the same customizations have to be repeated manually several times. A solution for this problem would be to create and store such a customization in a dedicated transformation rule, instead of storing the `Requests` in our Changes Model. Such a transformation rule should be created automatically based on customizations performed through direct manipulations. Additionally, the designer should be enabled to select the "scope" of such transformation rules. The scope could, for example, be either a given `Screen` or all `Screens` of a given Structural Screen Model.

Tool use

So far, the GSME has only been used by expert users (i.e., the author of this thesis and his supervisors). A usability evaluation of the tool was out of the scope of this diploma thesis. However, we created a "getting started" document (see Chapter A) to provide an introduction to the graphical diagrams, their features and how to use the GSME. This document is intended to facilitate using the GSME for novice users in the future.

6 Conclusion

In the course of this diploma thesis we developed the tool GSME, which intends to support the designer to improve the usability of automatically generated GUIs. Such improvements alter the Structural Screen Model (on the CUI level) directly, which is an intermediate output of the generation process. In particular, the GSME provides an overview of the `Frames` with all `Screens`, which allows for selecting a specific `Screen` to be visualized. Furthermore, the GSME facilitates the visualization of the `Screens`, including the layout and style information without the need of an application back-end. Additionally, it allows the designer to customize the layout and the style through direct manipulation and additionally facilitates customization through advanced features.

A Getting Started

A.1 Preface

The Graphical Screen Model Editor (GSME) is a collection of plug-ins, containing two graphical editors (i.e., diagrams).

The first diagram is the overview-diagram, which provides an overview of the Screen Model. It depicts the Frames with their containing Screens with the purpose to select a Screen. The content of the selected Screen is visualized with the second diagram, i.e., the *StructuralUI-Diagram* (or Screen-based diagram), which is a WYSIWYG editor providing a representation of the automatically generated Screen Model. Furthermore, it is a toolkit supporting a set of customizations.

A.2 Create a Screen Model Diagram

Initialize Diagram File

In the Package Explorer, select the previously created Screen Model "yyy_ScreenModel.structuralui". In the corresponding context-menu, you can find "Initialize model-diagram diagram file" (see Figure A.1). This command renders the *Overview Diagram* showing all screens of the model.

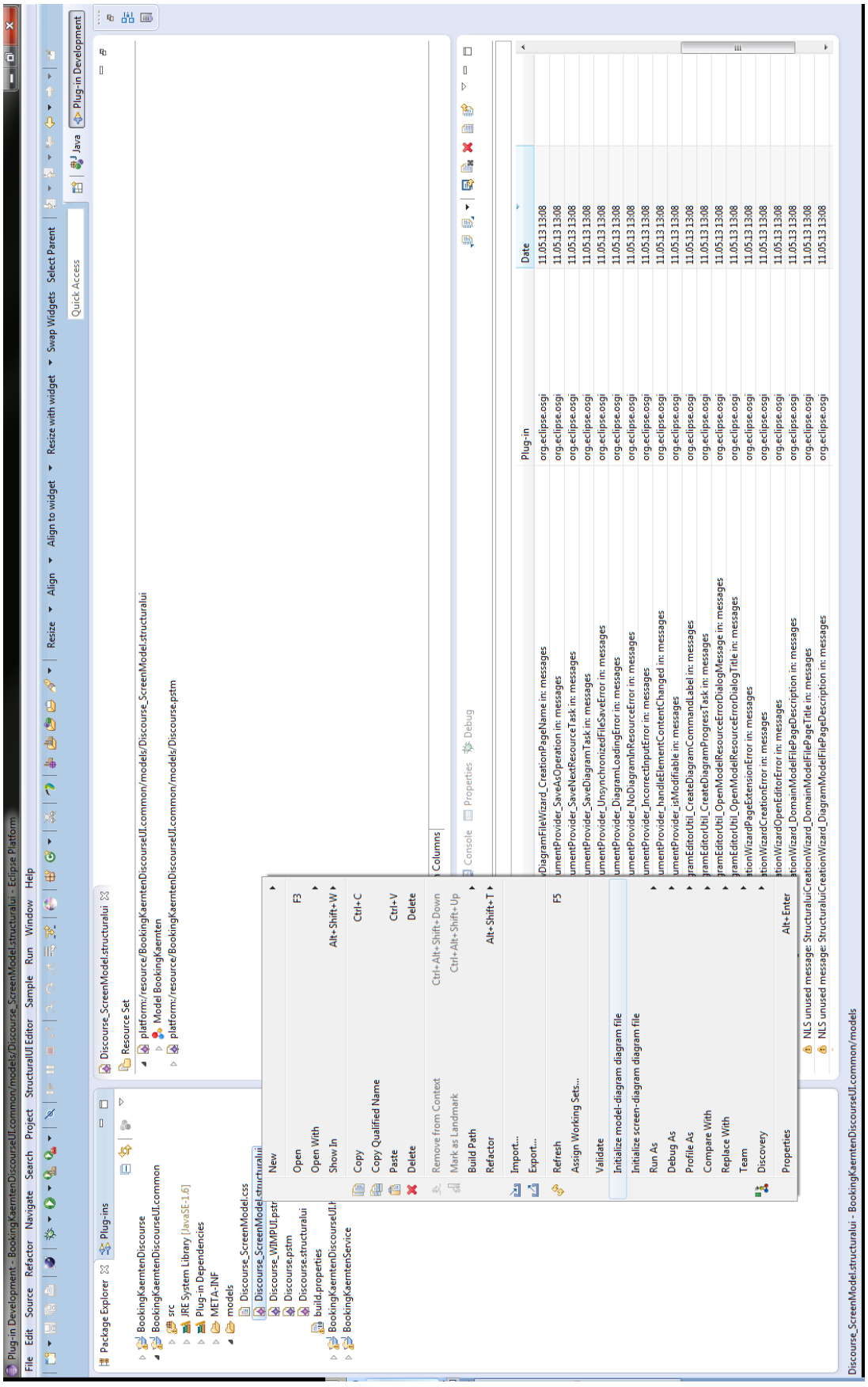


Figure A.1: Context menu: initialize model-diagram file

Overview Diagram

On the top of Figure A.2, you can see the name of the `Frame`. The boxes below represent the `Screens` in a table with three columns and as many rows as necessary to show all `Screens` within this `Frame`. If more than one `Frame` exists, further `Frames` are displayed in separate blocks below.

Double-clicking a `Screen`, visualizes the selected `Screen` with the *StructuralUI* diagram.

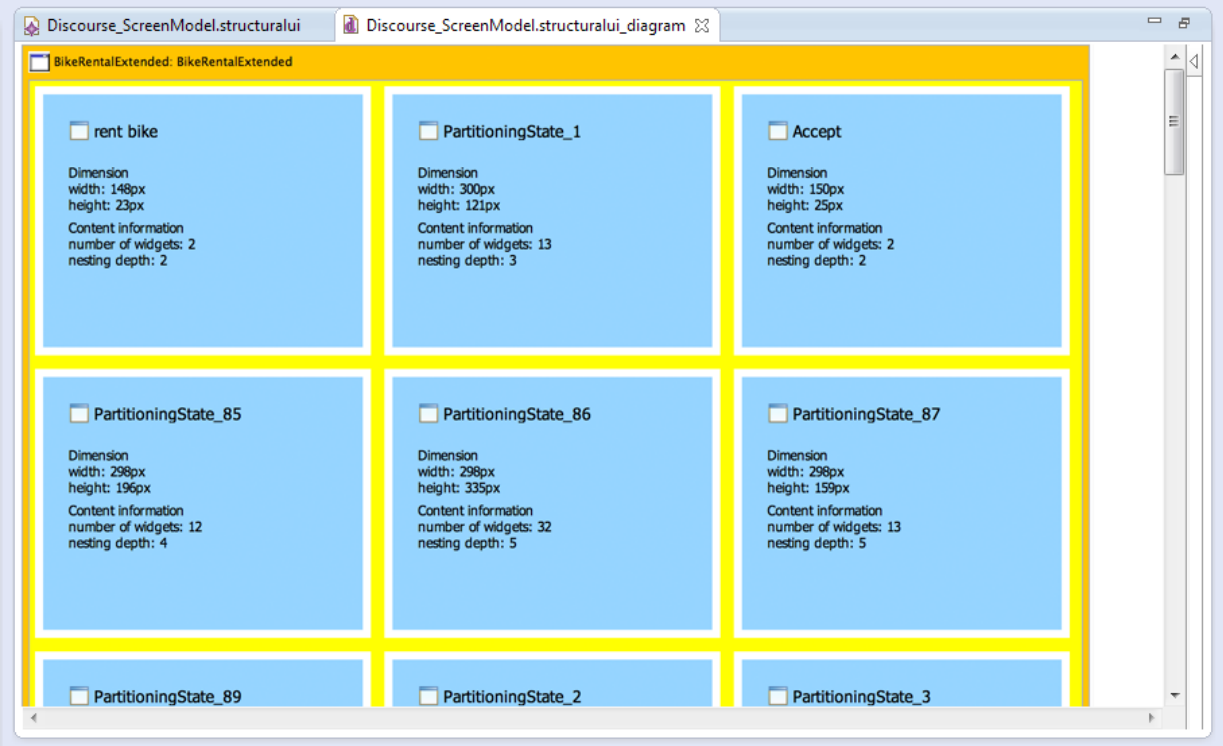


Figure A.2: Overview diagram

StructuralUI Diagram

This is the main part of the `GSME`, which visualizes a graphical representation of a single `Screen`, see Figure A.3. It includes all layout information from the `Structural Screen Model` and the style information from registered `CSS`-files.

Layout customizations are stored back in the *.structuralui-file, overwriting the initial data. Additional helper-information about the layout are stored in a newly created CSS-file, which also contains the style customizations. This file is stored in the model-folder with the name modelname_gsme.css.

A.3 Layout Customizations

Definition of Customizations

The `GSME` provides a set of layout customizations, which can be applied on `Widgets` within a `Screen`. A customization is defined to be manually initiated, either directly on the `Figure` (i.e.,

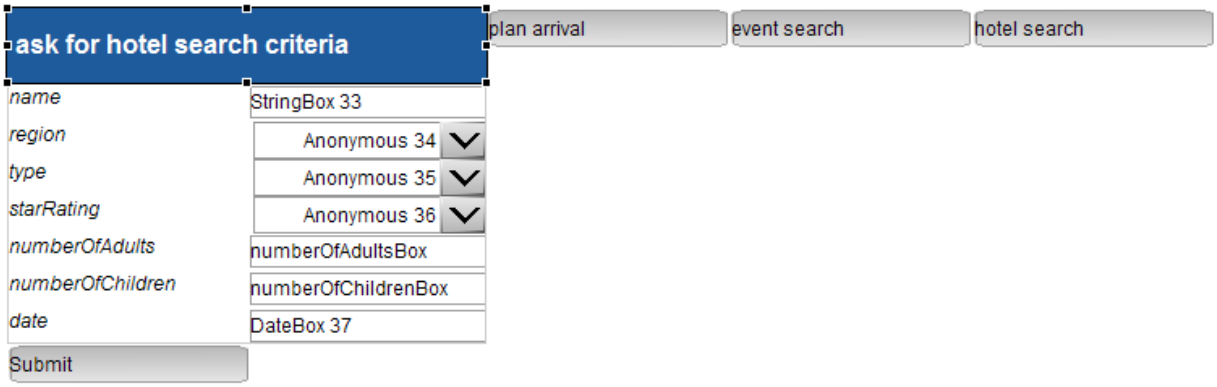


Figure A.3: StructuralUI diagram

the graphical representation of a `Widget`) or via the menu (i.e., toolbar or pop-up bar). An automatically generated `Request` (i.e, an `AutoSizeRequest`) is initiated by another `Request` or a customization and a certain condition. This condition could be "lack of space for the desired resize-operation". The resulting `AutoSizeRequest` could then be: "extend the container". As you might have guessed, this could result in a series of recursive operations.

Direct Resize

When a `Figure` is selected, it is wrapped by so-called "resize-handles" as depicted in the example of a `Label` in Figure A.3. If you drag such a handle, you can resize this `Figure`, but restricted by the constraints in Table A.1.

Table A.1: Constraints

Constraining part	Constraint	
1	`Containers`	must wrap their children
2	`Widgets`	must be within their parents
3	`Widgets`	must not overlap with siblings

Direct Move

A `Figure` can either be moved by the cursor-keys, or by dragging the whole part (not the resize-handles) using the mouse. For this operation the constraints from Table A.1 also apply.

Some `Widgets` have a `Container` with a size equal to the `Widget's` size. You may notice a strange behavior because of this "invisible" `Container`. To resize this `Container` before resizing the `widget`, you can select it either from the context-menu, or even more conveniently from the toolbar-menu, which is described next.

Toolbar

The `GSME Tool-bar`, depicted in Figure A.4, provides several advanced features. These are described in Table A.2.

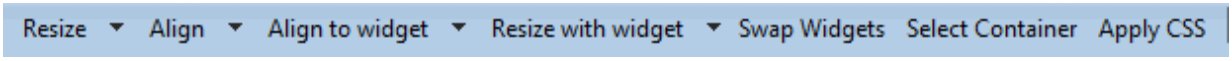


Figure A.4: `GSME` Toolbar

Table A.2: Toolbar Menu

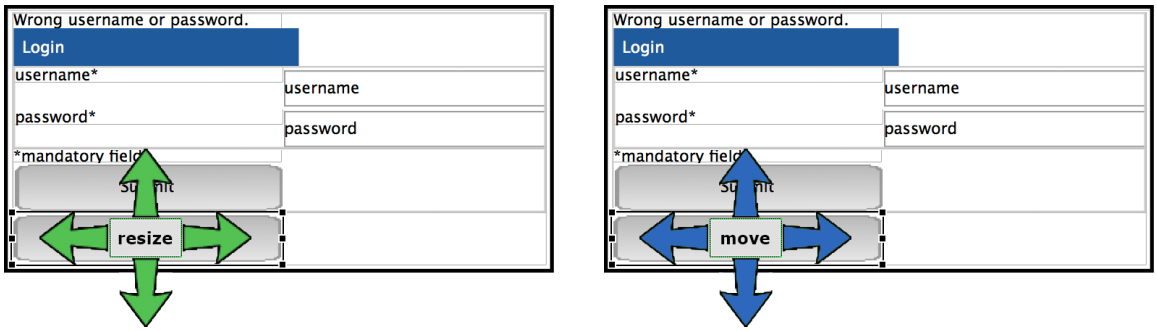
Menu	Submenu	Selection	Description
Resize	Max Top Right Bottom Left	one	Extending the size of the selected Widget to the maximum possible in the selected direction. "Max" stretches the size in all directions.
Align	Top Right Bottom Left	one	Moving the selected Widget to the maximum possible position in the selected direction.
Align to Widget	Top Right Bottom Left	several	The last selected `Widget` is the `Reference Widget`. All others are aligned to match with the selected edge of this reference.
Resize with Widget	All Horizontal Vertical	at least two	The last selected `Widget` is the `Reference Widget`. All others are resized according to match its size in the selected plane.
Swap Widgets	-	exactly 2	The two `Widgets` change their places if possible. This is just possible if they have a common ancestor and if there is enough space. The best result is achieved, if the `Widgets` have the same size.
Select Container	-	one	This is very useful to select an "invisible `Container`" whose size is equal to that of its single child-`Widget`.
Apply CSS	-	several	This applies the loaded CSS-files to the selected `Widget` and all its children, e.g., selecting the `Screen` will apply the styles to all `Widgets` within this `Screen`.

Pop-up Bar

`GMF` provides a pop-up bar, usually to create new parts. In our case, creating new `Widgets` would not be very useful, so we used the pop-up bar for another advanced feature. Figure A.5 shows the two types of the `GSME` Pop-up bar. By pressing *CTRL* or *SHIFT* and moving the mouse-pointer over a `Widget` (i.e., *hovering*), the corresponding `GSME` Pop-up bar appears. The type of operation is displayed in the center of the selected part, surrounded by colored arrows to initiate the operation.

The main advantage of modifying a layout in this way, is the recursive freeing of space to make the desired operation possible. This is achieved by resizing the container and moving siblings.

There are also some drawbacks on this kind of customization:



(a) Resizing: *SHIFT*+hovering the `Widget`

(b) Moving: *CTRL*+hovering the *Widget*

Figure A.5: The GSME Pop-up bar

- Move- and resize-delta is a fixed value of 10px.
- `Containers` don't support a pop-up bar (`Panels, ListPanels, etc.`)

A.4 Style Customizations using the Properties View

Core

In the property sheet `Core`, you get information about the selected `Widget`. Some of the values are allowed to be altered (e.g., width, height, format and text). Altering height and width of a `Widget` is constrained as described Table A.1. A message-box appears if the altered value violates the model.

Appearance

The property sheet `Appearance` is a standard property sheet, where you can change `Fonts` and `Colors` of selected `Figures`. The *Lines and Arrows* section can be used to change the border width of a `Figure`. For these operations, we recommend to use the property sheet `CSS`, which is described next.

CSS

The property sheet `CSS` shows all styles that are allowed to be modified. They are persisted in the `CSS`-file.

Attention: Not all `CSS`-styles match with all widget-types (see Table A.3)
Not all defined `CSS`-rule-values are available in `GSME`

Table A.3: Widget-Style combinations not supported

Widget	Style
TextBox	border background-color dimension*
ComboBox	border background-color dimension* font-size font-style font-weight font-color
Button	border background-color

* Increasing the dimension of the `Widget` only increases the space around the `Widget`. The `Widget's` size does not change.

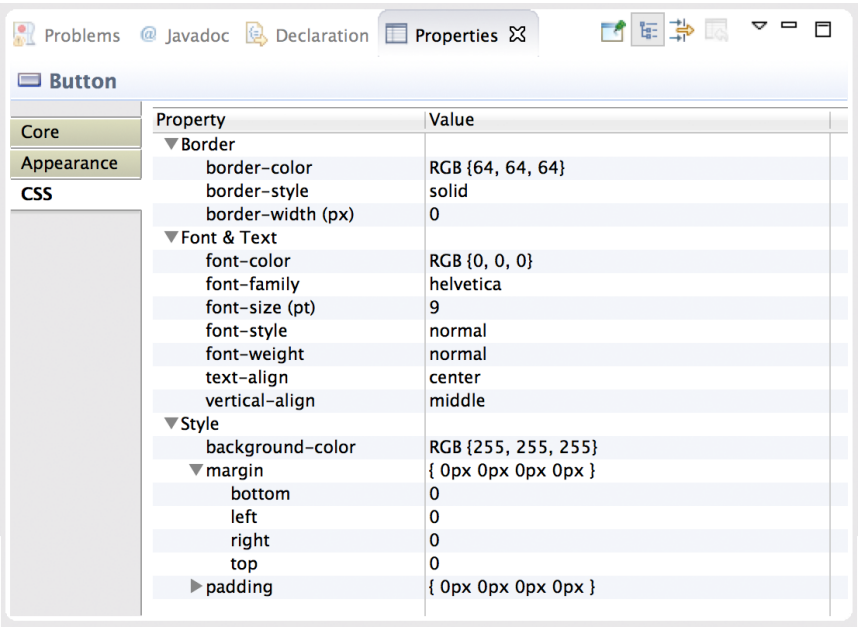


Figure A.6: Property-Page CSS

B The Structural UI Meta-Model

The Structural UI meta-model is the meta-model for the Structural Screen Model. It defines the types of `Widgets` and their properties and relations. It also defines the types of `LayoutManagers` and their corresponding `LayoutData`, which are available in the Structural Screen Model.

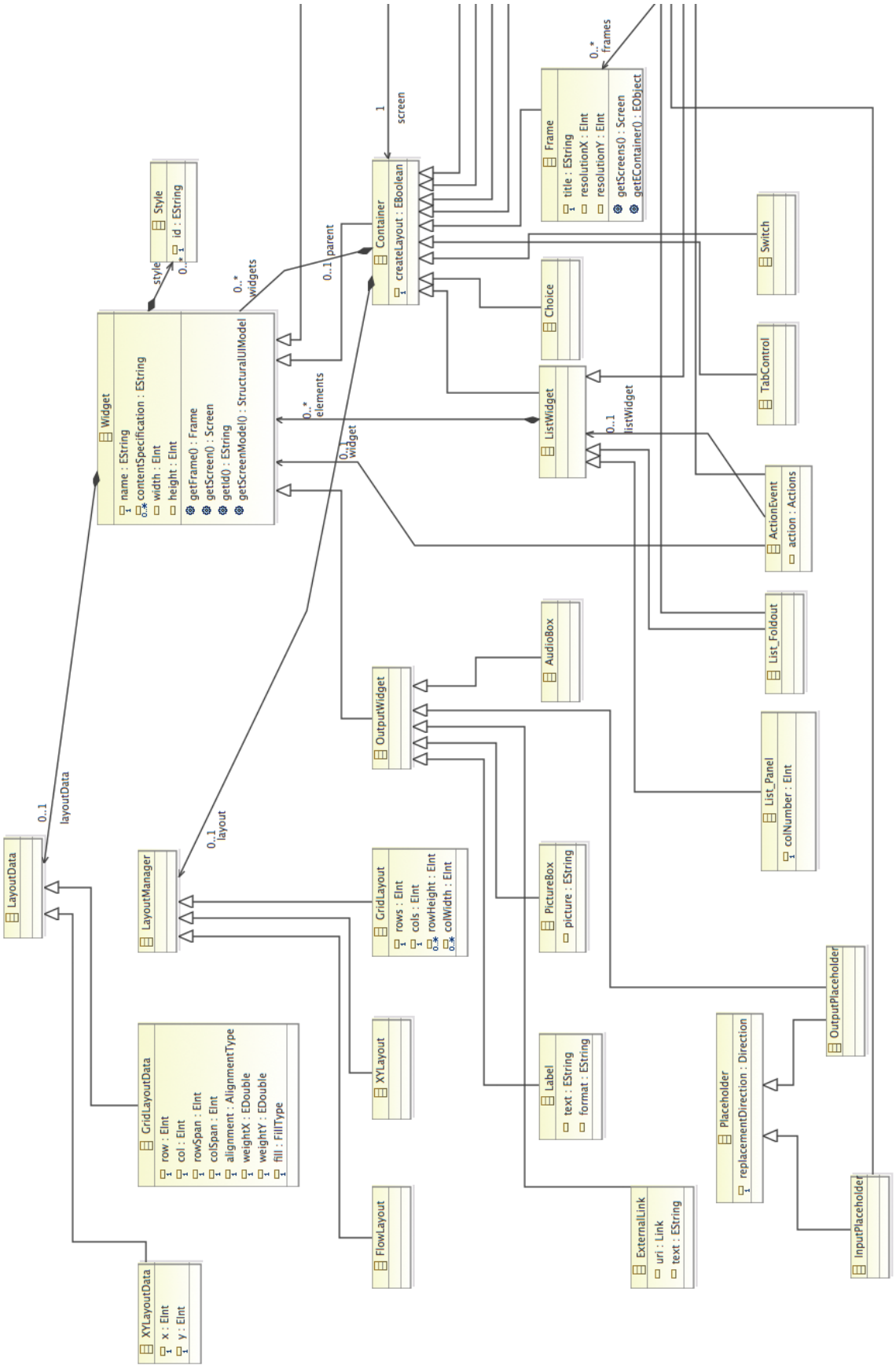


Figure B.1: `StructuralUI Meta Model` page 1/2

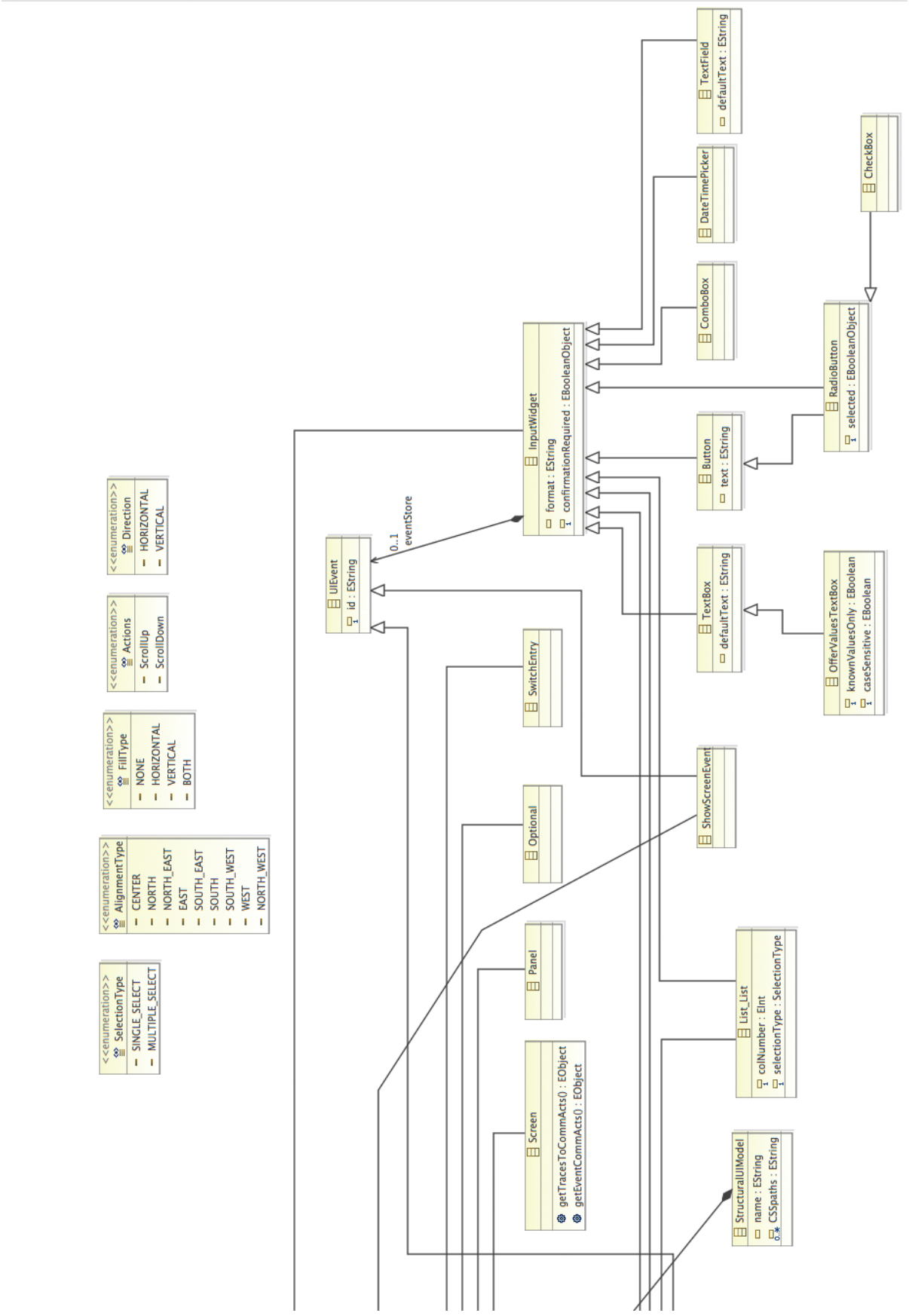


Figure B.2: `StructuralUI Meta Model` page 2/2

C Sequence Diagram of the Command Execution “Stretching the Button”

Figure C.1 presents the full sequence diagram showing the creation of the `AutoSize-Requests`, their corresponding commands and their detailed execution. In particular, we present the details of the command execution *Stretching the Button* from the example in Paragraph Resizing by use of the GSME Pop-up bar on page 28.

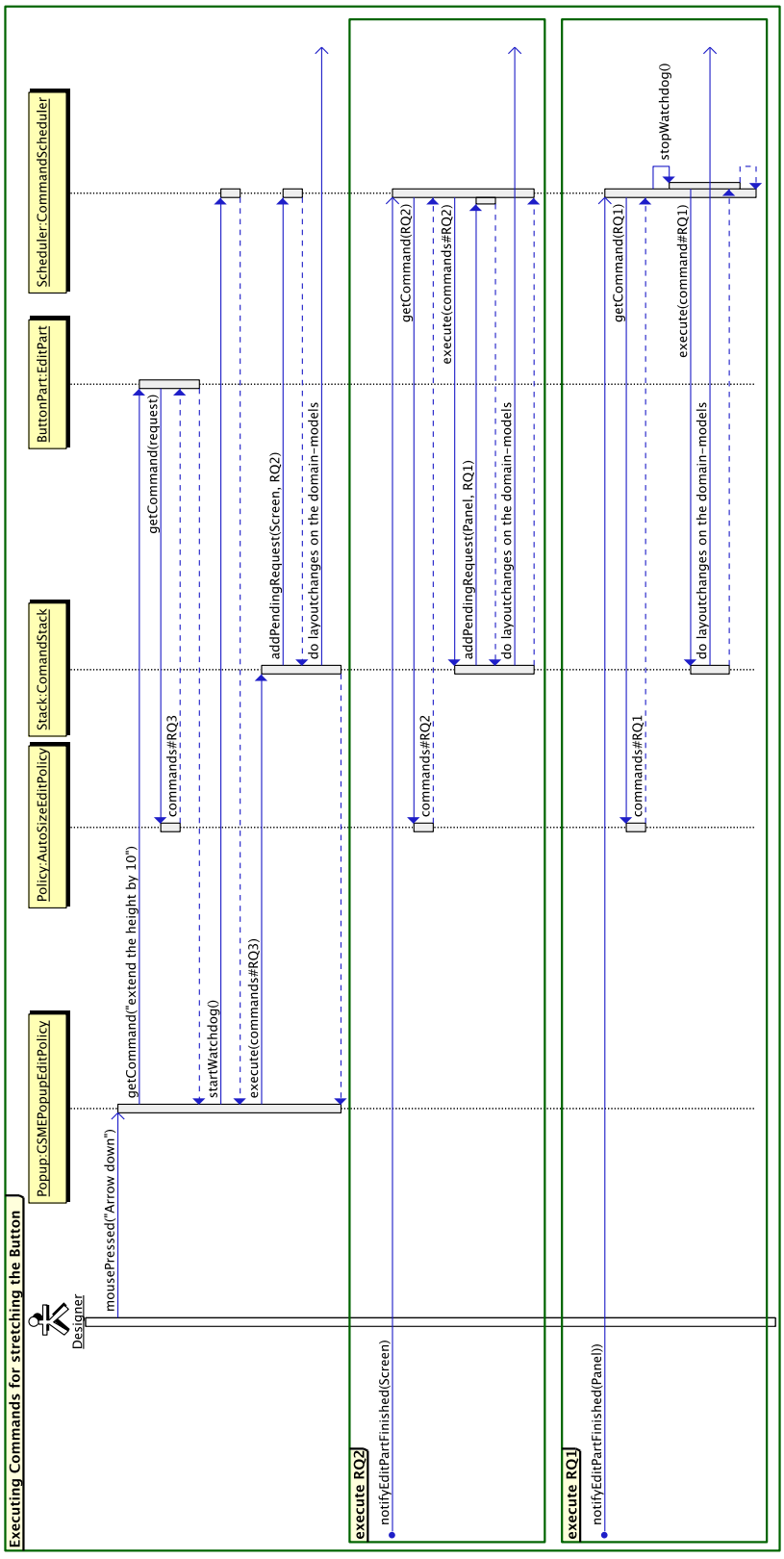


Figure C.1: Sequence diagram: Executing an `AutoSize-Request`

Literature

[CCT+03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003.

[Com] Eclipse Community. *GMF Tutorial Part1*. Eclipse Foundation. last visit 2014/06.

[FKP+09] Jürgen Falb, Sevan Kavaldjian, Roman Popp, David Raneburger, Edin Arnautovic, and Hermann Kaindl. Fully automatic user interface generation from discourse models. In *Proceedings of the 13th International Conference on Intelligent User Interfaces (IUI '09)*, pages 475–476. ACM Press: New York, NY, 2009.

[Fou] Eclipse Foundation. *GEF Developer Guide*. Eclipse Foundation. last visit 2014/06.

[FPR+07] Jürgen Falb, Roman Popp, Thomas Röck, Helmut Jelinek, Edin Arnautovic, and Hermann Kaindl. UI prototyping for multiple devices through specifying interaction design. In *Proceedings of the 11th IFIP TC 13 International Conference on Human-Computer Interaction (INTERACT 2007)*, pages 136–149, Rio de Janeiro, Brazil, September 2007. Springer.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, 2009.

[KFK09] Sevan Kavaldjian, Jürgen Falb, and Hermann Kaindl. Generating content presentation according to purpose. In *Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics (SMC 2009)*, San Antonio, TX, USA, Oct. 2009.

[KRF+09] Sevan Kavaldjian, David Raneburger, Jürgen Falb, Hermann Kaindl, and Dominik Ertl. Semi-automatic user interface generation considering pointing granularity. In *Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics (SMC 2009)*, San Antonio, TX, USA, Oct. 2009.

[MPV11] Gerrit Meixner, Fabio Paternò, and Jean Vanderdonckt. Past, present, and future of model-based user interface development. *i-com*, 10(3):2–10, November 2011.

[MT88] W. C. Mann and S.A. Thompson. Rhetorical Structure Theory: Toward a functional theory of text organization. *Text*, 8(3):243–281, 1988.

[PKR13] Roman Popp, Hermann Kaindl, and David Raneburger. Connecting interaction models and application logic for model-driven generation of Web-based graphical user interfaces. In *Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC 2013)*, 2013.

[Pop12] Roman Popp. A unified solution for service-oriented architecture and user interface generation through discourse-based communication models. Doctoral dissertation, Vienna University of Technology, Vienna, Austria, 2012.

[PRK13] Roman Popp, David Raneburger, and Hermann Kaindl. Tool support for automated

multi-device GUI generation from discourse-based communication models. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*, New York, NY, USA, 2013. ACM.

[Ran08] David Raneburger. Automated graphical user interface generation based on an abstract user interface specification. Master's thesis, Vienna University of Technology, Vienna, Austria, 2008.

[Ran14] David Raneburger. Interactive model-driven generation of graphical user interfaces for multiple devices. Doctoral dissertation, Vienna University of Technology, 2014.

[RKP+14] David Raneburger, Hermann Kaindl, Roman Popp, Vedran Šajatović, and Alexander Armbruster. A process for facilitating interaction design through automated GUI generation. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14)*, 2014.

[RPK+11] David Raneburger, Roman Popp, Hermann Kaindl, Jürgen Falb, and Dominik Ertl. Automated Generation of Device-Specific WIMP UIs: Weaving of Structural and Behavioral Models. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, pages 41–46, New York, NY, USA, 2011. ACM.

[RPK13] David Raneburger, Roman Popp, and Hermann Kaindl. Model-driven transformation for optimizing PSMs: A case study of rule design for multi-device GUI generation. In *Proceedings of the 8th International Joint Conference on Software Technologies (ICSOFT'13)*. SciTePress, July 2013.

[RPK+14] David Raneburger, Roman Popp, Hermann Kaindl, Alexander Armbruster, and Vedran Šajatović. An iterative and incremental process for interaction design through automated GUI generation. In *Proceedings of the 16th International Conference on Human-Computer Interaction*, 2014.

[RPV12] David Raneburger, Roman Popp, and Jean Vanderdonckt. An automated layout approach for model-driven WIMP-UI generation. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 91–100, New York, NY, USA, 2012. ACM.

[RWP+13] David Raneburger, Barbara Weixelbaumer, Roman Popp, Jürgen Falb, Nicole Mirnig, Astrid Weiss, Manfred Tscheligi, and Brigitte Ratzer. A case study in automated GUI generation for multiple devices. In *Proceedings of the 11th IEEE AFRICON Conference*, pages 1212–1217, 2013.

[Sch10] Alexander Schörkhuber. Integritätsprüfung von Diskursmodellen, Transformationsregeln und strukturellen Modellen von graphischen User Interfaces. Master's thesis, Technische Universität Wien, Fakultät für Elektrotechnik und Informationstechnik, Institut für Computertechnik, E384, 2010.

[Van04] Bill Moore; David Dean; Anna Gerber; Gunnar Wagenknecht; Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, first edition, February 2004. This edition applies to Version: 2.1.1 of the Eclipse Platform, Version 1.1.0 of the Eclipse Modeling Framework (EMF), and Version 2.1.1 of the Graphical Editing Framework (GEF) on Microsoft Windows.

[Š14] Vedran Šajatović. Improved tool support for model-driven development of interactive applications in UCP. Master's thesis, Vienna University of Technology, 2014.