

Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

From High-Usability Cross-Device Wireframe-Based Storyboards to Component-Oriented Responsive-Design User Interfaces

Stefanie Grewenig

Masterarbeit im Elitestudiengang Software Engineering



SOFTWARE ENGINEERING

Elite Graduate Program



Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

From High-Usability Cross-Device Wireframe-Based Storyboards to Component-Oriented Responsive-Design User Interfaces

Matrikelnummer: 1228728
Beginn der Arbeit: 02. September 2013
Abgabe der Arbeit: 03. März 2014
Erstgutachter: Prof. Dr. Alexander Knapp
Zweitgutachter: Prof. Dr. Bernhard Bauer
Betreuer: Dipl.-Inf. Univ. Ralf S. Engelschall



SOFTWARE ENGINEERING
Elite Graduate Program

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 3. März 2014

Stefanie Grewenig

Abstract

In recent years, the usage of wireframes to draw an initial specification for a user interface has become increasingly common. Even though they have great benefits in terms of rapidly sketching the state of a system and documenting basic user interactions, they lack in capturing key aspects of a user interface design, such as the relation between a task and a particular wireframe. Existing model-based approaches that target those problems are often not practical due to heavy-weight and hard to maintain models. Typically they are closed environments and do not integrate well in the software engineering process.

In this thesis we introduce Wireframe-Driven User Interface Design (WDD), which provides a low barrier of entry for the use in everyday software projects. It builds on top of a powerful model that interrelates a series of declarative models, such as user tasks, data objects and wireframes.

With Wireframe2Artefact, we provide a model-based software tool that enables designers and engineers to rapidly specify a user interface without being hung up on modelling details. Wireframe2Artefact is built on a powerful XML-based meta-level modelling language. In a case study, we demonstrate how our tool can be used to specify and evaluate the user interface of the P1-Timesheet time account app developed at msg systems ag.

Contents

List of Figures	xiii
1. Introduction	1
1.1. Objective and Motivation	1
1.2. State of the Art	1
1.3. Contribution	2
1.4. Structure of this Thesis	2
2. User Interface Design	3
2.1. Initial UI Design	3
2.1.1. Task Analysis	3
2.1.2. Domain Modelling	4
2.1.3. Creating Low-Fidelity Prototypes	4
2.2. Refining UI Designs	4
2.2.1. Storyboarding	5
2.2.2. Analyzing the Flow of Events	5
2.3. Implementing the UI Design	5
3. Requirements and Related Work	7
3.1. Requirements	7
3.1.1. Specifying UI Designs	7
3.1.2. Views on UI Designs	8
3.1.3. Estimation of UI Designs	8
3.1.4. Specifying responsive UI Designs	8
3.2. Related Work	9
3.2.1. Classic Interface Design Tools	9
3.2.2. Model-Based Interface Design Approaches	10
3.2.3. User Interface Description Languages	11
3.2.4. Shortcomings	12
4. Wireframe-Driven UI Design	15
4.1. Overview	15
4.2. User Interface Model	16
4.2.1. Task Model	16
4.2.2. Data Model	17
4.2.3. Wireframe Model	17
4.2.4. Example	19
4.3. Storyboard View	22
4.3.1. Definition	22
4.3.2. Approach	22
4.3.3. Equality	24

4.3.4. Example	24
4.4. Dataflow View	25
4.4.1. Definition	25
4.4.2. Approach	25
4.4.3. Equality	26
4.4.4. Example	27
4.5. Architecture View	28
4.5.1. Definition	28
4.5.2. Approach	28
4.5.3. Equality	31
4.5.4. Example	32
4.6. Evaluation of UI Designs	35
4.6.1. Example	35
5. Tool: Wireframe2Artefact	39
5.1. Overview	39
5.2. User Interface DSL	39
5.3. Features	41
5.4. Architecture	43
5.5. Current Limitations	44
6. Case Study	47
6.1. P1-Timesheet App	47
6.2. Specification of the UI	47
6.3. Comparison of the Component Trees	49
6.4. Lessons Learned	50
6.5. Full example	51
7. Towards responsive User Interfaces	53
7.1. Overview	53
7.2. Assisting responsive User Design	53
7.3. Adapting the front-end architecture derivation	54
7.3.1. Similarity Definition for Elements and Components	54
7.3.2. Handling shared Components	54
8. Further Work and Conclusion	57
8.1. Conclusion	57
8.2. Future work	58
A. List of Wireframe Element Types	61
B. Wireframe2Artefact Screenshots	63
C. P1-Timesheet	67
C.1. Storyboards	67
C.2. Data Flow Diagrams	69
D. RelaxNG Schemas	71
D.1. RelaxNG Schema of the Balsamiq Model	71
D.2. RelaxNG Schema of the Wireframe Task Model	73

D.3. RelaxNG Schema of the Wireframe Data Model	73
D.4. RelaxNG Schema of the Wireframe Model	74
D.4.1. RelaxNG Schema of the Wireframe Attributes	78
D.4.2. RelaxNG Schema of the Wireframe Container Elements	79
D.4.3. RelaxNG Schema of the Wireframe Input Elements	80
D.4.4. RelaxNG Schema of the Wireframe Layout Elements	83
D.4.5. RelaxNG Schema of the Wireframe Media Elements	84
D.4.6. RelaxNG Schema of the Wireframe Mobile Elements	85
D.4.7. RelaxNG Schema of the Wireframe Text Elements	85
D.4.8. RelaxNG Schema of the Wireframe Annotation Elements	86

List of Figures

3.1. UI Design	10
3.2. Result of XIML example	11
3.3. USIXML Model Collection [Lim+05]	12
4.1. UI Model	15
4.2. User Interface Model	16
4.3. Elements Submodel	17
4.4. Interaction model	18
4.5. my to-do app - wireframes	20
4.6. Storyboard Submodel	22
4.7. Dataflow Submodel	25
4.8. my to-do app - dataflow	27
4.9. Component Tree Submodel	28
4.10. "my to-do app" - after pre-processing	32
4.11. my to-do app - Component Tree	34
5.1. Screenshot of the wireframe model tab	42
5.2. BMML to WXML	43
5.3. Grid sketched with Balsamiq	43
6.1. Component Tree	50
6.2. Component Tree derived by Wireframe2Artefact	50
B.1. Wireframe Tab	63
B.2. Task Tab	63
B.3. Data Tab	64
B.4. Storyboard Tab	64
B.5. Data Flow Tab	65
B.6. Component Tree Tab	65
C.1. Login Storyboard	67
C.2. Create Entry Storyboard	68
C.3. Copy Entry Storyboard	68
C.4. Login Data Flow Diagram	69
C.5. Create Entry Data Flow Diagram	69
C.6. Copy Entry Data Flow Diagram	69

1. Introduction

We start with a brief introduction of the challenges during the user interfaces design process. (section 1.1). Next we summarize the state of the art (section 1.2), followed by our contribution (section 1.3) to the field of user interface design. Last we outline the structure of this thesis (section 1.4).

1.1. Objective and Motivation

“If I can't picture it, I can't understand it.” — Albert Einstein

According to most software engineering approaches, wireframes are a widely spread mechanism to deliver the initial specification for a user interface. Even though the term wireframe varies from organization to organization, it typically covers the following three aspects. First, it shows the main content in a screen. Second, it shows the structure of the information displayed on the screen and third, it briefly describes how the user interacts with the system.

The wireframing process has proven to be very effective, especially when gathering and communicating requirements between designers and developers. Nevertheless they usually fail in answering key questions such as: how wireframes are involved in accomplishing a particular task; what data is represented through a widget on the screen.

The answer to these questions is crucial to know in order to sufficiently specify and implement the user interface. Usually the developer overcomes those challenges by experience or by distilling the information from loosely connected documents, such as: domain model and description of the user tasks. This practice is no sustainable solution, especially when designing complex user interfaces. Consequently an approach is needed to master those challenges and provide the possibility to capture all aspects of a user interface design.

1.2. State of the Art

A number of methods and tools to describe, create and maintain user interface have been adopted by the industry. They require different skill sets, starting from knowledge about basic user interface design terminology up to expert modelling experience. It is not surprising that the resulting designs range from simple images up to fine-grained formal models. Whereas, most user interface design tools offer the designer ample support to rapidly manage and layout widgets on a screen. They lack in providing a possibility to capture information other than the graphical representation of the user interface. In strong contrast, model-based approaches capture most relevant aspects of user interface design, but fall short to provide a solution to effortlessly create and maintain those models. They come with a high barrier of entry and are usually closed development units and do not integrate well with common software engineering methods.

Current user interface design approaches and environments lack the ability to capture all relevant aspects of user interface design without restricting, limiting or complicating the design process.

1.3. Contribution

This thesis proposes the Wireframe-Driven user interface Design approach (WDD) which introduces three key innovations to user interface design:

Universal Model We offer a powerful model that tightly connects all relevant aspects of everyday user interface design to provide a sufficient representation of a user interface. It interrelates a series of declarative models, such as user tasks, domain model and wireframes to provide a formal representation of a user interface. It provides a low barrier of entry for the use in everyday software projects and allows rapid user interface design without being bogged down by modelling details. Specifying a user interface can be done very intuitively by manipulating concepts such as tasks, data objects, widgets and their interactions.

Multiple Views Our approach emphasizes the collaboration of different professions as well as continuous end-user involvement. We provide tailored views on the user interface model to make it possible for different professions to communicate effectively. Even though our approach does not focus on code generation of the user interface, we provide an educated guess for the frontend architecture of a user interface design.

Design Evaluation We offer a set of metrics that focuses on the impacts a user interface design has on software construction and the overall development process. Furthermore, designers and developers can assure that all tasks and data objects are embodied in the user interface design and visualized through one or more wireframes.

1.4. Structure of this Thesis

To accomplish this goal, this thesis will first explain the basic concepts it is built on, such as the user interface and a component-oriented architecture (chapter 2). We will then try to identify common scenarios and derive applicable requirements that we think are crucial to efficiently generate and evaluate user interface designs in the real world (section 3.1), before examining the existing solutions both academic and commercial (section 3.2). We give a detailed explanation of our approach (chapter 4) and its implementation (chapter 5) before demonstrating its usage by comparing and analyzing our outputs against the actual implementation and specification of a time account application, P1-Timesheet, developed at msg systems ag (chapter 6). We will then propose basic concepts to extend our approach to work with responsive user interface designs (chapter 7). The thesis closes with a short conclusion and hints on possible further research topics (chapter 8).

2. User Interface Design

This chapter aims to give a broad overview of the concepts this thesis is built on and establishing a basic vocabulary to use for the remainder of this thesis. We start with a brief explanation of the initial user interface design (2.1) and what information and artefacts are needed during this process. Since user interface design has no widely accepted definition, we define this process covering the activities from start of design to a detailed description of the user interface, which can be used for implementation. Such a detailed description can be defined in one or more artefacts, e.g. formal models, class diagrams, etc.. We give a glimpse at the methodologies used to refine user interface designs (2.2) and close with a brief introduction of common heuristics used to modularize a user interface in a component-oriented fashion (2.3).

2.1. Initial UI Design

Specifying and documenting the user interface starts at early stages of the software development process and is typically part of the requirements engineering phase [Bra02], [HJD10]. Different from the software development process, activities of the user interface design process vary widely from organization to organization. Despite all differences, almost every user interface design effort will at some point do the following[RF13]:

1. Analyzing and specifying a set of user tasks
2. Modeling domain objects and their relations
3. Sketching the user interface

In the following section we introduce those concepts in detail.

2.1.1. Task Analysis

Eliciting user tasks is one of the first steps when building a software system and is usually part of the requirements engineering process. It consists of many activities, such as analysis, specification and validation. Different from the software development process, activities of the requirement engineering process vary, depending on the type of system being developed. There are plenty of concepts and tools on how to gather and document the user tasks. Classic software engineering methods tend to prefer use-cases to document their requirements. Software projects relying on agile principles opt for user stories.

For the course of this thesis we classify a user task as followed (see table 2.1): Our definition of the term user task is based on the terminology of the activity theory [PhD13], proposing the concept of activities, actions and operations. An activity is based on motives, in requirements engineering

	Activity Theory	WDD
Motive	Activity	
Goal	Action	Task
Condition	Operation	Step Interaction

Table 2.1.: Task Ontology

we want to understand the users' motivation to interact with the system. The action is defined by its goal and a set of individual steps. These steps have to be performed in order to complete an activity. Other than the activity, an action is independent from the users' motivation. An operation¹ represents the execution level of an action and does not rely on a goal.

In short, a task is described through a set of steps, which explain how the user can fulfill the task. An interaction can be seen as the click on a button and is bound to a condition, e.g. state of the button is not allowed to be disabled in order to click the button.

2.1.2. Domain Modelling

Domain analysis, as seen from a software engineering perspective is defined as the following [Pri90]: The process by which information used in developing software systems is identified, captured and organized.

The industry adopted a variety of concepts and tools to describe and document domain models during early stages of the software engineering process. Common methods are: UML class diagrams, entity-relationship diagrams or data dictionaries. During the course of this thesis we will refer to the domain model as a model that captures the core data objects and their relations, needed when creating a new system.

2.1.3. Creating Low-Fidelity Prototypes

Low-fidelity prototypes can be used to describe any human-computer interaction. They focus on the visible information and the flow of events and functionality. Usually they lack in typography, color and graphics and do not feature the look and feel of a system [RF13], [SB10]. Nowadays the following methods are widely spread to create low fidelity prototypes: Content inventories, sticky notes, wireframes, paper prototypes or mockups. Prototypes are generally created by business analysts, interaction designers, usability experts, developers and the user themselves.

In this thesis we focus on wireframes, which typically have the following three features: Basic UI elements, simple interaction design and layout options. Approaches and tools that ensure these features are common and widely spread. Wireframes have proven effective in communicating requirements [PMM05], nevertheless they are not intended to be a full specification of a user interface. Designers typically produce wireframes based on the information gathered with other methodologies, such as task analysis or domain modelling.

2.2. Refining UI Designs

The level of abstraction of a user interface design depends on multiple factors: Complexity of the user interface, experience of UIs designer and engineers, importance of usability or simply on the projects size and costs. One objective of refining a user interface is to capture and document the interrelation of the information of an initial user interface design. These can be done by creating storyboards and visualizing the users' interactions within a task. A description of the flow of events describes the interrelation between domain objects and the users' interactions with the system. The following section give a brief introduction to both concepts: Storyboarding and documenting the flow of events.

¹Note that the term operation will be further used in another context (chapter 4).

2.2.1. Storyboarding

There is no widely accepted definition of the term storyboarding in terms of user interface design. In our thesis we refer to the following definition [SB10]. The storyboard is an ordered set of sketches (wireframes) or screenshots (demonstrators). It serves to visualize the main forms of the event chain defined by the actions of the user.

In other words, the designer sketches various states of the user interface. These sketches are then assembled into a storyboard, representing the state changes of the user interface during a task. In the storyboarding context, these sketches of the UI are called keyframes. Transitions between those sketches are called actions.

Other than wireframing tools, storyboarding tools especially made for the purpose of creating and refining user interface designs are rather uncommon. Moreover, refining a user interface through a storyboard is usually done manually, by describing a task through an ordered subset of wireframes and interactions.

2.2.2. Analyzing the Flow of Events

When designing a user interface, it is not only important to specify and document how the user interacts with the system. The activity of analyzing the flow of events is generally done during early stages of the engineering process. The industry adopted a wide range of methodologies to do so. Sequence diagrams have proven to offer a great form to document the flow of events[Fow04]. A sequence diagram can be based on a user task[AI07]. It represents the user interaction with the system as well as the systems' internal communication.

In other words: high-level sequence diagrams represent a user task through an ordered set of operations on a set of data objects.

2.3. Implementing the UI Design

When it comes to designing the architecture of a software system, the functionality is separated from the frontend and the code is cut into handy chunks to handle the complexity and allow reusability. When identifying components from the frontend of an application we typically want maximum reusability of user interface elements in terms of human recognition and from a technical implementation point of view. In order to achieve this goal we need to understand what elements are displayed and when. This information can be gathered by a set of wireframes describing the visible elements of a user interface.

The approach proposed by [Engc] has proven to have great benefits, when modularizing and describing the architecture of a user interface in a so called "component tree". The goal of the component tree, as the name implies, is to assemble the containment hierarchy of a user interface through arranging the components of a user interface in a tree.

In summary, a component tree represents the elements of a set of wireframes grouped into components and organized in a hierarchical structure.

3. Requirements and Related Work

To assess and compare the state of the art, this chapter will define a set of scenarios we want to focus on. Based on these scenarios we derive applicable requirements and scope (section 3.1). The following section aims at giving a broad overview of the current state of the user interface design approaches and related areas (section 3.2). The last section closes with the validation against our requirements and summarizes the shortcomings.

3.1. Requirements

First, we define scenarios in which wireframe-driven software design can be applied. These scenarios target common problems designers, engineers and domain experts are faced with in all stages of the user interface design. They are not meant to cover all imaginable scenarios, but to define a scope for this thesis. Based on these scenarios, we derive requirements to define a context for the conceptual approach as well as the corresponding tool support.

3.1.1. Specifying UI Designs

In early stages of the development the conception of the UI follows a similar approach: based on user tasks and domain objects the designer sketches the UI. These high-level sketches help to draw an initial specification of a UI without being lost in details. Nevertheless simple sketches fall short in providing any notation or concept to enhance the initial specification. They usually cannot answer key questions, such as how a wireframe is involved in accomplishing a certain task or what domain objects are involved in it. Answering those questions is usually done by assembling wireframes into storyboards or creating data flow diagrams. Maintaining and synchronizing those artefacts with the initial specification is time consuming and requires heavy refactoring work, especially when designing a complex user interface. We need an approach that captures the interrelation between the concepts of a wireframes, domain objects and user tasks in one user interface model.

Requirement 1 *The approach must support and guide the process of user interface design based on a powerful model*

Requirement 2 *The approach must allow the user to capture the relation between wireframes, user tasks and domain objects*

Specifying a model is relatively difficult, due to a heavy model maintenance effort and a steep learning curve. Whereas sketching a user interface is low in costs and no special skills are needed to participate during the sketching process.

Requirement 3 *The tooling must hide the model from the user*

Requirement 4 *The tooling must provide an easy and self-descriptive way to specify and document user interface, without learning a modelling language*

Furthermore, model-based environments are usually closed development units. To integrate well with the software engineering process our tooling should enable starting and/or continuing the specification of a user interface using multiple methods or tools.

Requirement 5 *The tooling should provide a mechanism to use the output of other independent tools*

Requirement 6 *The tooling should provide a mechanism to provide suitable output which can be used by other tools*

Moreover, the approach should be valid whether the user interface design has been created incremental in different sprints in an agile project or all at once in a sequential design process, as in the waterfall model.

Requirement 7 *The approach must be independent from the software engineering method*

Requirement 8 *The approach must allow incremental prototyping*

3.1.2. Views on UI Designs

One objective of these wireframes is to communicate requirements between domain experts, designers and software engineers. Domain experts, who usually do not have a background in sketching a user interface, describe a user interface in terms of tasks, roles and domain objects. Whereas a human-computer-interaction experts treat the features of a system as a black box and focus on the navigation through the user interface and the organization of widgets and windows. In contrast, software engineers will not touch the graphic design of a user interface, but want to classify tasks into features and estimate size and complexity of a sketched user interface. In order for these three to communicate effectively it is crucial to provide tailored views on the user interface for each professional.

Requirement 9 *The approach must provide views for each profession*

3.1.3. Estimation of UI Designs

When talking about user interfaces and usability it is common to create multiple alternative designs, which differ from each other by the wireframes. A decision for a design is typically based on: Evaluation of usability experts, usability constraints and design criteria. With common approaches it is impossible to compare the designs by the impacts they have on the system: Size and complexity of the frontend architecture or the flow of data during a task. By providing a set of metrics based on the user interface design we provide an abstract view on the design. In addition we provide a way to compare different designs by metrics other than usability constraints and design criteria.

Requirement 10 *The approach must define a set of metrics to estimate the size and complexity of the user interface design*

3.1.4. Specifying responsive UI Designs

Nowadays user interfaces are developed to run on multiple types of devices, such as phones, tablets or desktops.

Requirement 11 *The approach must allow specifying responsive user interface designs*

3.2. Related Work

A number of papers have attempted to cover different tools and techniques to allow powerful user interface design. This section aims at giving a broad overview of the existing concepts and techniques. As there is no widely accepted definition of user interface design methodologies we classified the fairly wide number of methodologies into classic interface design tools, model-based approaches and user interface description languages. We will focus on tools and approaches that are widely spread and actively used in academia or industry.

3.2.1. Classic Interface Design Tools

Classically early interface design tools summarize tools which are designed to specify the user interface design through sketching the state of a user interface.

Tools

Balsamiq Mockup (Balsamiq)¹ is a web-based wireframing tool which allows the designer to arrange pre-built widgets using a WYSIWYG editor. It can be used to design either mobile or desktop applications. Moreover the designer is able to express basic user interactions by linking Balsamiq files. The resulting wireframes can either be exported as image files or as Balsamiq files. The Balsamiq file (BMML) is defined as an XML-based language and offers representation units, such as controls and properties.

WireframeSketcher² like Balsamiq, is a wireframing-tool with pre-defined widgets and a WYSIWYG editor. Additionally the designer is able to organize wireframes into interactive storyboards. The application is available as desktop version as well as a plug-in for any Eclipse IDE.

OmniGraffle³ is a multi-purpose drawing tool which provides a set of stencils created for the purpose of designing wireframes. Although it is not designed for the specific purpose of creating and maintaining user interfaces, it is commonly used in the industry. Like the other drawing/wireframing tools, it offers a drag-and-drop, WYSIWYG editor. It does not provide any additional features to describe user interactions or data objects.

Evaluation

The tools described are a great way to design and maintain wireframes. Focusing on this, they offer a tremendous set of pre-built widgets, which can be very handy. Using a graphical editor is understandable for almost everyone involved in the software engineering process (Requirement 4). It allows the user to rapidly sketch a first draft of user interface design. Some even offer the possibility to describe basic user interactions. But this high-level description of the user interface has deficiencies. These tools allow manipulating their pre-built widgets, but are completely separated from any notations of the task that the wireframe implements, as well as any notation of what domain objects are visualized by a widget. Even though some of them allow the specification of basic interactions, no concept or notation is provided to specify the relation between interactions and the operations triggered by them. The WireframeSketcher allows the creation of storyboards based on the design wireframes. Nevertheless these storyboards have to be created manually and

¹<http://balsamiq.com/>

²<http://wireframesketcher.com/>

³<http://www.omnigroup.com/omnigraffle>

cannot be enhanced with additional information. Consequently, they are not suitable to describe all relevant aspects of a user interface design, which we proposed in Requirement 2.

3.2.2. Model-Based Interface Design Approaches

Originating from model-driven software development, models can also be used to create the initial specification for a user interface. In this approach the engineer defines a high level model of the user interface based on concepts such as tasks, users and objects [PMM05], [Ang97], [PE02], [Lim+05]. The tool will then support the interface design by guiding the process or even producing deliverables, such as wireframes.

Tools and Methods

Mobi-D (Model-Based Interface Designer) [Ang97] and its underlying concept was developed at Stanford University. It provides a meta-level modeling language, which defines the elements of the user interface, as well as the components, structure and relations within interface models. The purpose of Mobi-D is to emphasize the communication between the end-user and the interface developer, in order to allow the end-user to participate in more aspects of the development. Furthermore, it focuses on capturing the relations between user tasks, domain models and representation units (widgets, components). Mobi-D achieves this by providing an iterative, user-centered development cycle: First, the end-user enters a textual description of the user task. Based on the key terms the tool elicits from the task description, the system guides the end-user in editing and refining the key terms into a structured task description. The developer uses this description to create a user task model and a domain model. Furthermore, the tool guides the developer during the modeling process and even recommends widgets for each task. The user is able to override these recommendations if necessary.

The UI Pilot [PMM05], like Mobi-D was developed at Stanford University. It has a similar set of features, but provides a more advanced tool⁴ for the modeling process. The approach also focuses on specifying and manipulating concepts instead of manipulating the widgets of a user interface. The tool is based on a subset of the eXtensible Interface Markup Language (XIML)[PE02]. We are going to explain the XIML language in section 3.2.3. The user interface design process can be described in three main steps: First, the designer identifies user tasks that the interface must enable. Second, data objects that are to be displayed or captured through the interface must be identified and third, identifying the various user types that the target application will support. Abstract wireframes are then designed by linking the user tasks to a wireframe. The tool depends on a web service which recommends widgets for the abstract wireframes. The resulting wireframes can be exported either as Rich-Text Format document or as XIML document.

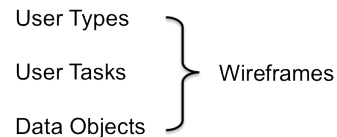


Figure 3.1.: UI Design

Evaluation

Model-based interface designing is a great way to focus on system-centered (Requirement 2) terms such as tasks, users and objects instead focusing on engineering terms like windows, widgets and screens. They have the advantage that they eliminate all distracting design choices and concentrate

⁴<https://www.redwhale.com/products/uipilot/about.html>

on the negotiation of an abstract wireframe. But the design recommendations of the tool follow a deterministic algorithm and will not be as creative as the design of a usability expert. Model-based tools rely on powerful models (Requirement 1) in their underlying frameworks. Both approaches are based on the concept of creating wireframes and the underlying models are hidden from the user (Requirement 3). While the developers claim that the approach is well suitable for an initial user interface design specification (Requirement 8), they usually rely on a defined set of user tasks to start the designing of wireframes. Conclusively it is not possible to define wireframes independent from other artefacts such as the user tasks or the data objects (Requirement 5). Although they emphasize end-user involvement and provide different views for the end-user and developer, they do not distinguish between other professions related to the user interface design process (Requirement 9).

3.2.3. User Interface Description Languages

Over the years, the research community has developed quite a number of different syntax and semantics to describe the structure and/or behavior of the user interface of a complex software system. Usually these languages focus on components and their functionality.

Languages

XIML (eXtensible Interface Markup Language) [PE02] is a powerful language to define the representation and manipulation of interaction data. It has been successfully applied as basis for software tools such as the UI Pilot.[PMM05] XIML is an XML-based language and includes the following representational units: components, relations and attributes. Interface elements are categorized into interface components: user tasks, domain objects, user types, representation elements and dialog elements. These elements can be linked by a relation. Attributes describe properties or features of a specific element. Listing 3.1 shows a basic example of the XIML language⁵, the corresponding sample result can be found in fig. 3.2. Moreover the XIML language claims to supports multi-platform interface development (Requirement 11).

```

1 <el eltype="txt" x="23" y="18" datatype="static" dataval="Hello, World!" font="Verdana"
  color="0x000000" size="30" />
2 <el eltype="line" x="20" y="60" x2="R-20" y2="60" c="0x000000" a="100" t="1" />
3 <el eltype="rect" x="30" y="75" w="140" h="25" c="0x000000" a="50" r="10" />
4 <el eltype="bord" x="50" y="90" w="170" h="25" c="0x000000" a="50" t="3" r="5" />

```

Listing 3.1: "XIML Example"

Hello, World!




Figure 3.2.: Result of XIML example

⁵<http://ximl.com/>

UsiXML (User Interface eXtensible Markup Language) [Lim+05] The UsiXML languages is an XML-compliant mark-up language. Designers specify user interfaces at different levels of abstraction: Task and Concepts (T&C), Abstract User Interface (AUI), Concrete User Interface (CUI) and the Final User Interface (FUI). The T&C level is the most abstract level and defines the user interface in terms of tasks and data objects (classes). At the FUI level, the user interface is described through code (markup, programming, declarative). In fig. 3.3 a simplified version of the UsiXML model collection is shown. The taskModel describes the tasks the user performs within the user interface and the corresponding interactions. The description of the classes of data objects manipulated by a user while interacting with the system is specified in the domainModel. Semantic relations between models are defined in the mappingModel. The contextModel itself consists of multiple models: user model, platform model and an environment model. Hence, a user interface model defined with UsiXML is defined by one or more of the model components described above.

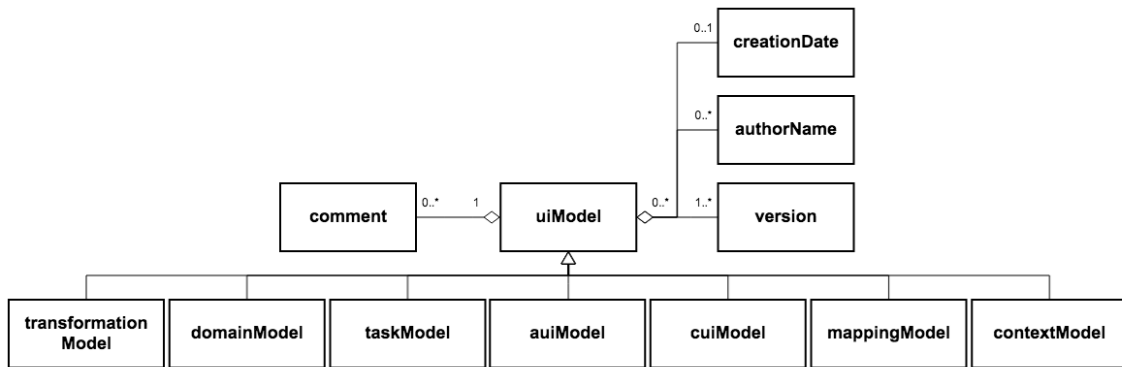


Figure 3.3.: USiXML Model Collection [Lim+05]

Evaluation

User interface description languages are well suitable to describe almost every user interface. Relying on an XML-based notation is understandable for almost everyone in the software industry. These languages are extremely expressive (Requirement 1) and flexible, and allow the user to specify the user interface down to very fine grained design specifications. But such power has its drawbacks: It suffers from massive complexity, leading to specifications that are not easy to read or to understand (Requirement 4). The structure of the languages and the concepts behind them forces the user to think and work like writing code instead of designing the user interface. These lower levels of abstraction might also lead to slow down the design process and also increase maintenance to keep the specification up to date. The focus of one integral model makes it harder to reuse existing documents and integrate well in the software engineering methods (Requirement 7). Although they claim to provide different views for each level of abstraction of the user interface design, those views are usually from a development point of view and do not emphasize effective communication between different professions (Requirement 9).

3.2.4. Shortcomings

As shown above, research in this area has been going on for a long time, but commercial software developers are still reluctant towards model-based user interface design and user interface descrip-

tion languages that offer more than the description of layouts and widgets. When analyzing the approaches and tools, we found the following: The tools are often stuck in theory without regard to the application in industry. These approaches are typically hard to learn, understand or implement in a real world situation. Others simply solved other problems: Classic early interface design approaches have their place for the use-cases they have been envisioned for, but do not support any options for specifying more than the visible elements of the user interface and their basic interaction. We found that The UI Pilot tool and its underlying user interface description language XIML are the closest fit to the requirements we defined. Nevertheless, they still have weaknesses with respect to our use-cases that we will try to address in this thesis.

4. Wireframe-Driven UI Design

In this chapter we will describe the Wireframe-Driven UI Design approach in detail. We start with a short overview (section 4.1) of the ideas our approach is based on before diving into a detailed look on our underlying user interface model (section 4.2). In the following sections we explain our approach to slice a storyboard (section 4.3) and a dataflow view (section 4.4) of the given model. In section 4.5 we explain the derivation of our component-oriented architecture proposal. We close with a set of metrics defined to evaluate and estimate the size and complexity of a given user interface model (section 4.6).

4.1. Overview

Reflecting the previous chapter, model-based approaches offer great possibilities to represent and interrelate all relevant aspects of a user interface design. Nevertheless they have their drawbacks when it comes to effortlessly specifying an initial design. Their main focus lies on task and domain models, which eliminates or restricts the creative process of designing a representation for a user interface design.

In this chapter we are going to introduce our user interface model that combines the benefits of model-driven approaches and the concepts of rapid wireframing methods. Our model interrelates as series of declarative models: user task model, data model and wireframe model (Requirement 1, Requirement 2), with the wireframes being the foundation of our user interface design model. Our approach offers domain experts or end-users to specify a user interface in terms of tasks, roles and domain objects. In parallel, the design expert focuses on finding scenarios and sketching the user interface. The designer then enhances the wireframe model by the information provided by the domain expert. This is done by linking tasks and domain objects to the wireframe model as described in section 4.2.3. By bringing together the principles of user-centered design approaches and a developer perspective on a user interface design, we are able to answer key questions, such as:

- What data is represented through a widget?
- How is a wireframe involved in accomplishing a user task?
- What data is involved in representing a wireframe or a certain widget?

Based on this information we can slice a set of views from the given model (Requirement 9): We offer a storyboard and the dataflow for a given task. Even though our approach does not focus on code generation, we propose an approach on how to derive a component-oriented architecture (component tree), based on a given wireframe model. Furthermore, we defined a set of metrics to assist designers and engineers in the process of estimating the size and complexity of user interface design (Requirement 10).

Moreover our model enables the designer to specify responsive user interfaces (Requirement 11).

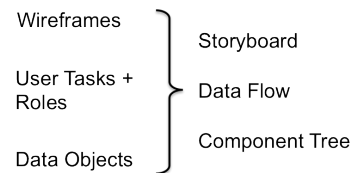


Figure 4.1.: UI Model

The user interface model can be thought of as a set of wireframes enhanced by the information given by the domain expert. The approach does not restrict or limit the process of defining the model and can be easily embedded in most software engineering methods (Requirement 7, Requirement 8).

4.2. User Interface Model

In this section we introduce our user interface model. In doubt, we chose to sacrifice expressiveness for simplicity, creating a model that works very well for common use cases instead of creating a more global model that suffers from massive complexity. Figure 4.2 shows the core of the model our approach is based on. Since an entire user interface model can be very involving when presenting at the same time, we decided to split the model into three sub-models: task model, data model, wireframe model. With each sub-model different aspects of the user interface are specified.

To be able to reason about the relations we specified in the class diagram (fig. 4.2) we use the following notation: For each relation, between two classes A, B we specify two functions. Based on the multiplicities of the relation, its type is either $f : A \rightarrow B$ or $f : A \rightarrow 2^B$. Moreover, every class specifies a set with the same name. For unary functions, let $a.f$ be equal to $f(a)$.

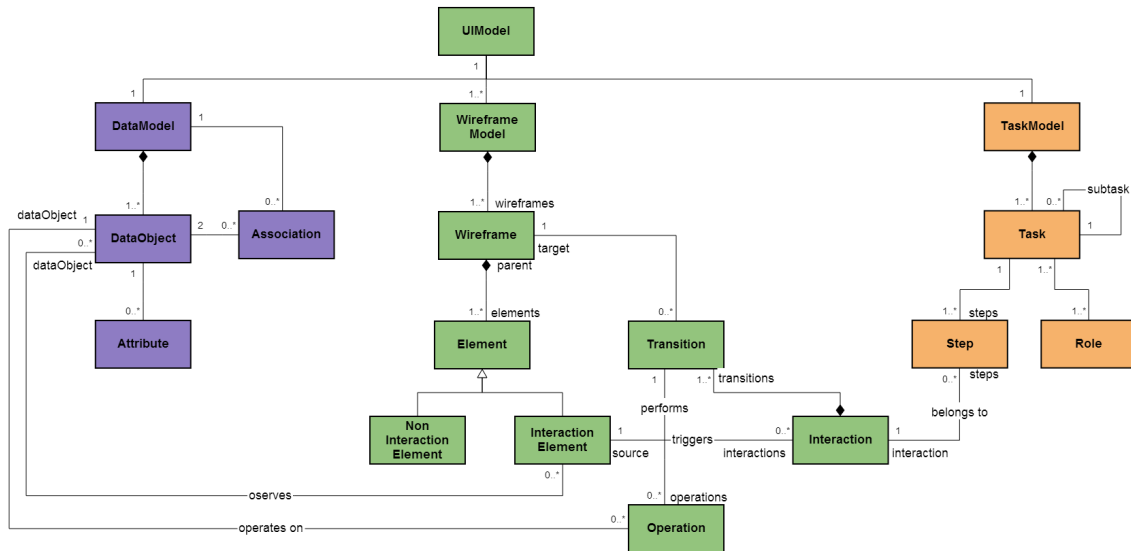


Figure 4.2.: User Interface Model

4.2.1. Task Model

There are a number of methodologies and software tools built for task elicitation and documentation. To address most of them we defined a simple yet sufficient task model, which focuses on the core aspects of a task: define steps and assign roles. As seen in fig. 4.2 a task can be defined by steps or a task itself, so called subtasks. Furthermore a task is defined through a set of ordered steps and a set of roles. We choose the concept of roles to determine whether a user is allowed to perform a task or not. As defined by the class diagram, we have the following sets and functions:

$$roles : Task \rightarrow 2^{Role}$$

$$steps : Task \rightarrow 2^{Step}$$

Note that the steps of a task are indexed.

4.2.2. Data Model

As mentioned before, there are multiple ways to define the data model of a system at an early stage of the development lifecycle. We defined a model which combines the core aspects described by common methods. This model most resembles the traditional UML class diagram in a typical lines-and-boxes data diagram. Data objects are specified by their name and attributes. Moreover two data objects can be connected through an association, which is refined by its multiplicities and a label.

$$dataObjects : DataModel \rightarrow 2^{DataObject}$$

4.2.3. Wireframe Model

We designed our user interface model to contain multiple wireframe models, with each wireframe model grouping a set of wireframes designed for a specific device. A more detailed view on our wireframe sub-model can be found in appendix D.4.

$$wireframes : WireframeModel \rightarrow 2^{Wireframe}$$

$$elements : Wireframe \rightarrow 2^{Element}$$

Wireframe Elements

To describe the elements that can be found in a wireframe, we need a model which satisfies the needs of everyday user interfaces in terms of windows, widgets and dialogs. Additionally we need to provide a notation for the concepts provided by common wireframing tools, such as browser windows, phone frames and annotations.

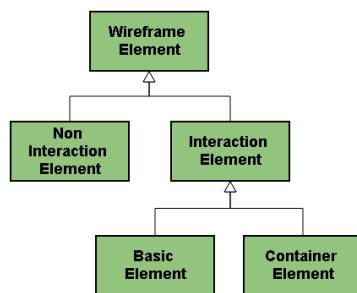


Figure 4.3.: Elements Submodel

Thus, we distinguish between interaction elements and non-interaction elements. In general everything that is visible on a screen, e.g. buttons, labels, panels and so on will be referred to as an interaction element.¹ A non-interaction element is an element that is required for the wireframing process only and will not be taken into account in the actual implementation, e.g. an annotation like a sticky note. These elements are usually used to document additional informations during the wireframing process. As the name implies a non-interaction element, in contrast to an interaction element, cannot trigger interactions. An example for an interaction is hovering or clicking on a button, checking a checkbox and so on. Examples for the different types of each element category are: Button, iPhone, Sticky Note. Altogether, our wireframe model provides over 50 different element types to ensure an equal wireframing experience as given by standard wireframing tools, such as Balsamiq.

¹Except the frame element. An example for a frame element is a browser window or a phone. These interaction elements can have behaviour and trigger interactions, e.g. the swipe interaction on a smartphone. Even though they are interaction elements, they will not be taken into account during implementation.

A full list of the element types provided can be found in appendix A and in the implementation on the enclosed CD.

During the course of this chapter it is not necessary to describe all elements specified in the wireframe model; for simplicity we refer to an element as defined below. Moreover, when referring to an element, we refer to an interaction element unless said otherwise.

parent : Element \rightarrow Wireframe

Additionally we need the following relations, which are not described in the class diagram:

area : Element \rightarrow Area

style : Element \rightarrow Style

type : Element \rightarrow Type

Interactions

An interaction describes the users communication with the system. To provide a powerful interaction concept, we inherited basic concepts of the interactive sketching notation proposed by [Jak09], [Lin99] and [LM95]. An interaction is specified by the attributes listed below:



Figure 4.4.: Interaction model

Set of transitions A transition consists of a target and a condition. The condition has to be met in order to trigger the transition. All conditions have to be mutually exclusive.

Trigger Defines whether the transition is triggered by a click, double-click, mouse-over etc. For different devices there are different triggers, e.g. a multi-touch device can have swipe or zoom.

Duration Defines the approximated time it takes the transition to complete. A good indicator for the duration time of a transition can be found in the execution time of the operations performed when the transition is triggered.

Delay Defines the time it takes to start the transition. The delay is meant to cover usability and design aspects of a user interaction.

Effect Defines the effect of the transition, for example slide or fade-in. The effects are comparable to an HTML5 canvas animation² for example.

As described in fig. 4.2, an interaction is triggered by an element in a wireframe. This can be represented by the following relation.

interactions : Element \rightarrow 2^{Interaction}

source : Interaction \rightarrow Element

²<http://slides.html5rocks.com>

Furthermore, an interaction consists of a set of transitions, with each transition having a condition and referring to a target wireframe:

$$transitions : Interaction \rightarrow 2^{Transition}$$

$$condition : Transition \rightarrow Condition$$

$$target : Transition \rightarrow Wireframe$$

To bridge the gap between the wireframe model and the data model we introduced the concept of operations. An operation, as the name implies, describes a systems internal operation which will be performed when an interaction has been triggered. We focused on the standard CRUD (create, read, update, delete) operations. Moreover, we specified a single transition of an interaction to trigger multiple operations. In order to interrelate a set of operations to an interaction we defined the following relation ³:

$$Type = \{CREATE, READ, UPDATE, DELETE\}$$

$$type : Operation \rightarrow Type$$

$$dataObject : Operation \rightarrow DataObject$$

$$operations : Transition \rightarrow Operation$$

Furthermore read operations can be triggered by an element itself. This would be comparable to an element observing a data object and can be defined as the following relation:

$$dataObject : Element \rightarrow DataObject$$

Analog to the relation between a wireframe model and a data model, we need to define the relation between a wireframe model and a given task model. This can be done by describing the relation between the step of a task to an interaction.

$$steps : Interaction \rightarrow 2^{Step}$$

$$interaction : Step \rightarrow Interaction$$

4.2.4. Example

In the following sections we to introduce a simple app called "my to-do app" to explain our approach. The "my to-do app" allows the user to login and create and maintain to-dos. In the following sections we are going to specify the model for the "my to-do app":

Task Model

Even though we could easily think of more than one task for an app like this, through the course of this example we will focus on adding a new to-do to the list:

$$task_{addTodo}.steps = \{step_{login}, step_{createNewTodo}, step_{saveTodo}\}$$

$$step_{login} = \text{"Enter username, password and click the login button"}$$

$$step_{createNewTodo} = \text{"Click on the create button"}$$

$$step_{saveTodo} = \text{"Enter name, date and click on the save button"}$$

We do not distinguish between different user roles: $task_{addTodo}.roles = \{User\}$.

³We define the operations of the relation $transition.operations$ to be ordered.

Data Model

The *dataModel*, showing the apps data and their relations:

$$dataModel.datatobjects = \{user, todoList, todo\}$$

In this example we focused on the data objects and did not specify any relations between them.

Wireframe Model

Our wireframe model consist of three wireframes: *wireframe_login* showing the state of the system when starting the app (fig. 4.5, A). And *wireframe_todoList* (B) showing the users' to-do list. The last wireframe *wireframe_newToDo* (C) displays the form to enter a new to-do. Although each wireframe consists of multiple elements (fig. 4.5), during the course of this example we are not going to specify all of them. Last, we define three simple interactions according to the steps in *task_addToDo*. The

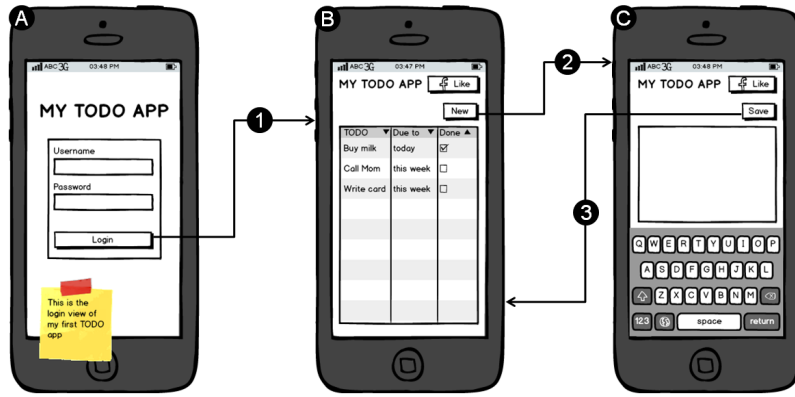


Figure 4.5.: my to-do app - wireframes

first interaction describes how the user logs into the "my to-do app" by clicking the login button in *wireframe_login* (fig. 4.5, 1).

$$interaction_{clickLogin}.source = loginBtn \wedge loginBtn \in wireframe_{login}.elements$$

$$interaction_{clickLogin}.transitions = \{transition_{login}\}$$

$$transition_{login}.target = \{wireframe_{todoList}\}$$

$$transition_{login}.condition = "username and password valid"$$

In order to verify the users' identity and show the list of to-dos we need to define the following to operations:

$$transition_{login}.operations = \{operation_{getUser}, operation_{getList}\}$$

$$operation_{getUser}.dataObject = User, operation_{getUser}.type = READ$$

$$operation_{getList}.dataObject = TodoList, operation_{getList}.type = READ$$

We describe the reference to the first step of our task as:

$$interaction_{clickLogin}.steps = \{step_{login}\}$$

The second interaction is defined, so that the user will switch from (B) to (C) when clicking on the "new" button:

$$interaction_{clickNew}.source = newBtn \wedge newBtn \in wireframe_{login}.elements$$

$$interaction_{clickNew}.transitions = \{transition_{new}\}$$

$$transition_{new}.target = \{wireframe_{newToDo}\}$$

$$transition_{new}.condition = TRUE$$

Furthermore, the reference to the second step id defined by:

$$interaction_{clickNew}.steps = \{step_{createNewToDo}\}$$

Lastly the user enters a name and time for the to-do in the form and saves the new to-do:

$$interaction_{saveToDo}.source = saveBtn \wedge saveBtn \in wireframe_{login}.elements$$

$$interaction_{saveToDo}.transitions = \{transition_{save}, transition_{couldNotSave}\}$$

If the input is valid the system will create a new to-do and update the to-do list:

$$transition_{save}.target = \{wireframe_{todoList}\}$$

$$transition_{save}.condition = "input valid"$$

$$transition_{save}.operations = \{operation_{createToDo}, operation_{updateList}\}$$

$$operation_{createToDo}.dataObject = ToDo, operation_{createToDo}.type = CREATE$$

$$operation_{updateList}.dataObject = ToDoList, operation_{updateList}.type = UPDATE$$

Otherwise the system will stay on the same screen and not perform any operations:

$$transition_{couldNotSave}.target = \{wireframe_{newToDo}\}$$

$$transition_{couldNotSave}.condition = "input invalid"$$

This interaction belongs to the interaction described in step three:

$$interaction_{saveToDo}.steps = \{step_{saveToDo}\}$$

4.3. Storyboard View

In this section we propose an approach to slice a storyboard from the information provided by the taskmodel and a wireframe model. A storyboard is a view on the UI design used to describe the system users' behavior during a single task. In the previous section we introduce the user interface model (section 4.2) and defined a relation between interactions and steps. Based on this information, we are able to derive a storyboard for each task. We start by providing a definition of the terms storyboard, keyframe and action. We explain the derivation of the storyboards and the information it depends on (section 4.3.3). We close with the storyboard sliced from the "my to-do app" introduced in the previous section (section 4.2.4).

4.3.1. Definition

The storyboard illustrates the important steps of the user experience based on a task. The structure of a storyboard as described in section 2.2.1 consist of two main parts: keyframes and action, with keyframes connected through actions. In our case, a storyboard is determined by a directed rooted graph of transitions that belong to the same task, which is specified by the relation we described in section 4.2.3: $interaction : Step \rightarrow Interaction$

In the context of the storyboard view, we define the nodes of the storyboard graph as wireframes, which are connected through transitions:

$$Keyframe \subseteq Wireframe$$

$$Action \subseteq Transition$$

$$connects : Keyframe \times Keyframe \rightarrow Action$$

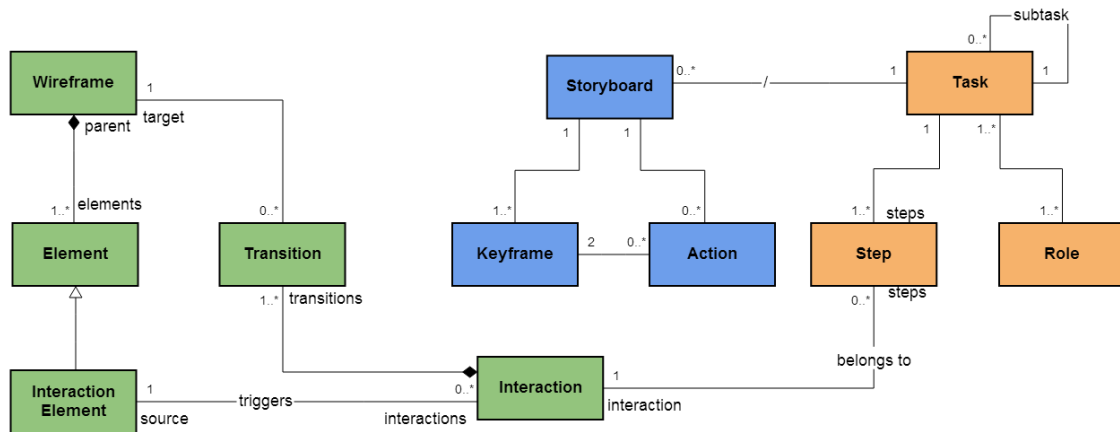


Figure 4.6.: Storyboard Submodel

4.3.2. Approach

For each task in a given task model we build a storyboard as described below:

Defining the nodes

Let the set of keyframes of a storyboard $keyframes : Storyboard \rightarrow Keyframe$ of a given $task$ be defined as:

$$\begin{aligned} Storyboard.keyframes = \{ & interaction.source, transition.target | \\ & \exists step \in task.steps \\ & \wedge interaction = step.interaction \\ & \wedge transition \in interaction.transitions \} \end{aligned}$$

Building the storyboard

Algorithm 1 determines the $connects : Keyframe \times Keyframe \rightarrow Action$ relation, in order to build the storyboard graph.

Algorithm 1: Defining the actions

Input: Task $task$

```

1 begin
2   foreach step in task.steps do
3     interaction = step.interaction;
4     foreach transition in interaction do
5       connects(interaction.source.parent, transition.target) = transition
6     end
7   end
8 end
```

Invalid Storyboards

In case the user specifies an unconnected storyboard, the resulting graph will also be unconnected and the storyboard will be invalid. For example:

$$interaction_1.transitions = \{transition_1\}, interaction_1.source = w_1$$

$$transition_1.target = w_2$$

$$interaction_2.transitions = \{transition_2\}, interaction_2.source = w_3$$

$$transition_2.target = w_4$$

$$step_1.interaction = interaction_1, step_2.interaction = interaction_2$$

The resulting output can be seen below:



4.3.3. Equality

As described above (section 4.3.1) the storyboard of a task depends on the $steps : Interaction \rightarrow 2^{Step}$ relation and the transitions of the interactions that belong to a task. By construction, it is independent from the operations linked to its interactions. Two storyboards of a given task are equal if they refer to the same graph (section 4.3.1). They are invariant against changes to the wireframes, as long as no interactions are edited.

4.3.4. Example

In the previous section (4.2.4) we introduced the "my to-do app" and specified its user interface model. The keyframes, and their relations for $task_{addTodo}$ can be seen below:

$Keyframes = Wireframes$

$Actions = Transitions$

$connects(wireframe_{login}, wireframe_{todoList}) = transition_{login}$

$connects(wireframe_{todoList}, wireframe_{newTodo}) = transition_{new}$

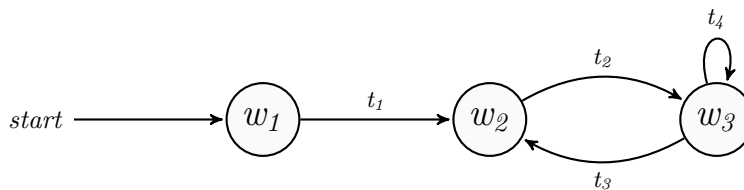
$connects(wireframe_{newTodo}, wireframe_{todoList}) = transition_{save}$

$connects(wireframe_{newTodo}, wireframe_{newTodo}) = transition_{couldNotSave}$

The resulting transition graph can be seen below:

$w_1 = wireframe_{login}, w_2 = wireframe_{todoList}, w_3 = wireframe_{newTodo}$

$t_1 = transition_{login}, t_2 = transition_{new}, t_3 = transition_{save}, t_4 = transition_{couldNotSave}$



4.4. Dataflow View

The behavior of a defined task can be described as a data flow diagram. We choose to describe our dataflow diagram as a simplified UML sequence diagram. Nevertheless, it does not substitute a proper sequence diagram, neither is it sufficient for implementation. This dataflow diagram should be used by engineers and architects to get an overview of the data flow described by the wireframes, without looking at the wireframes and tasks itself. In section 4.4.1 we introduce a basic vocabulary for this section used to explain our approach (section 4.4.2). Last, we show the derived data flow diagram for the "my to-do app" (section 4.4.4).

4.4.1. Definition

Our data flow diagram captures the behavior of a single task. The data diagram shows a number of data objects and the operations that are passed between those objects. In terms of the data flow diagram section, we will refer to the set of sample objects as participants. The set of operations passed between them will be described as methods.

$$\text{Participant} \subseteq \text{DataObject} \cup \{ \text{USER}, \text{SYSTEM} \}$$

$$\text{source} : \text{Method} \rightarrow \text{Participant}$$

$$\text{target} : \text{Method} \rightarrow \text{Participant}$$

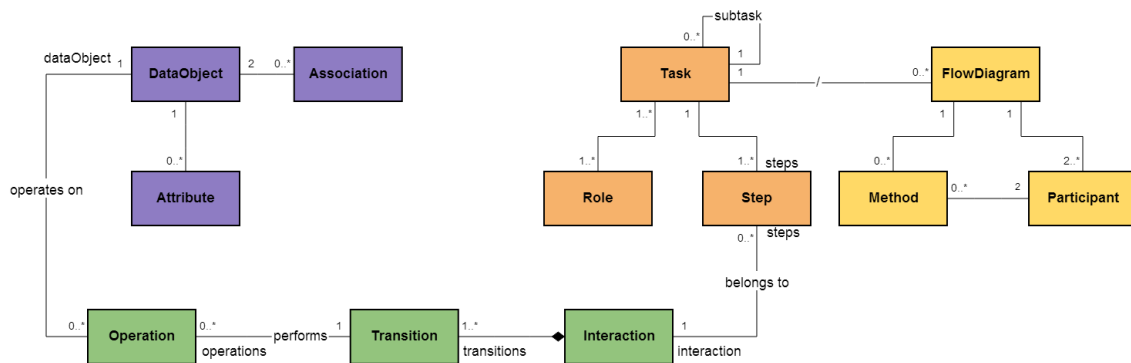


Figure 4.7.: Dataflow Submodel

4.4.2. Approach

As mentioned before, a data flow diagram is defined by participants and methods. This section gives a broad overview on the approach we used to define the participants and methods of a given task.

Defining the Participants

The following equation specifies how we derive the participants of a data flow diagram $participants : FlowDiagram \rightarrow Participant$ that belong to a particular $task$:

$$\begin{aligned}
 dataflow.participants = \{ & p | \exists step, \in task.steps \\
 & \wedge interaction = steps.interaction \\
 & \wedge transition \in interaction.transitions \\
 & \wedge p = transition.dataObject \} \cup \{USER, SYSTEM\}
 \end{aligned}$$

Defining the methods

Algorithm 2 defines an ordered set of methods⁴ which are passed between the participants in a data flow diagram based on a given $task$.

Algorithm 2: Defining the methods of the data flow diagram

Input: Task $task$

```

1 begin
2   foreach step in task.steps do
3     interaction = step.interaction;
4     Let method be a new method;
5     method.source = USER;
6     method.target = SYSTEM;
7     Method = Method  $\cup$  method;
8     foreach transition in interaction.transitions do
9       // create new alternative for each transition, with the transitions condition as
10      alternative condition
11      foreach operation in transition.operations do
12        Let method be a new method;
13        method.source = SYSTEM;
14        method.target = operation.dataObject;
15        Method = Method  $\cup$  method;
16      end
17    end
18  end
19 end

```

4.4.3. Equality

The data flow diagram of a given task depends on the interactions, transitions and operations linked to steps of the task. By construction dataflow models are invariant against changes to the wireframes, as long as no interactions and their corresponding operations are edited.

⁴Note that the order of the methods can be derived from the indexed steps.

4.4.4. Example

Figure 4.8 shows the dataflow diagram derived from the "my to-do app" model. To improve the understandability, we visualized the data flow diagram in an UML sequence diagram style.

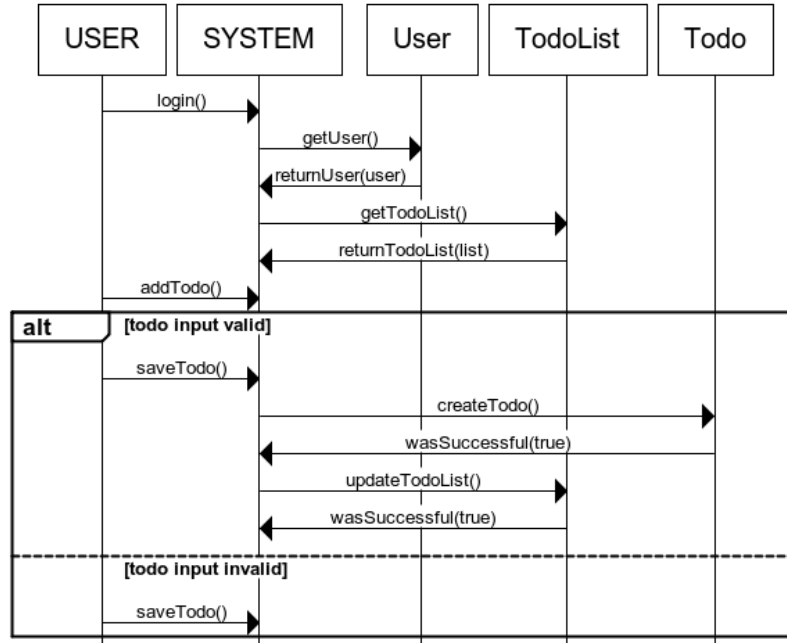


Figure 4.8.: my to-do app - dataflow

4.5. Architecture View

As introduced in section 2.3 we are applying hierarchical decomposition onto wireframes, using in particular the Model View Controller / Component Tree (MVC/CT) architecture pattern proposed by [Engb]. After giving a brief definition of the terms component and component tree (section 4.5.1) we explain our approach and the assumptions we made in detail (section 4.5.2). A detailed introduction to user interface decomposition and the (MVC/CT) can be found in [Vaa]. We close this section with the component tree of the "my to-do app" (section 4.5.4). All algorithms proposed in this section are valid for a user interface model with exactly one wireframe model. A brief explanation on how to handle responsive user interfaces can be found in chapter 7.

4.5.1. Definition

A component tree describes the architecture of a UI as a hierarchical structure of components. We define the component tree to consists of components, which are either composite-components or widget-components (see fig. 4.9). A widget-component, as the name implies, groups a set of widgets, here interaction-elements to a component. A composite-component can be thought of as an orchestration panel that logically groups a set of widget-components.

$$\text{WidgetComponent} \subseteq 2^{2^{\text{Element}}}$$

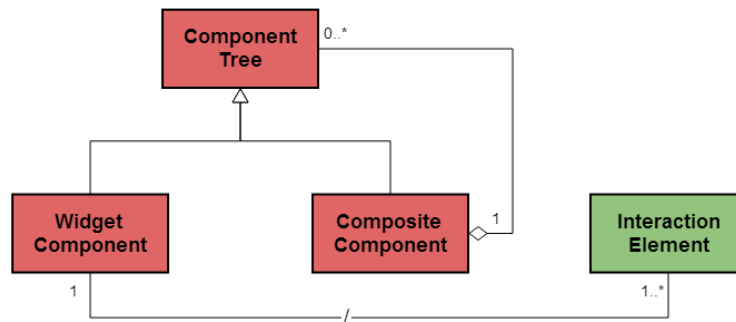


Figure 4.9.: Component Tree Submodel

4.5.2. Approach

This section explains the approach used to derive the component tree in detail. Our goal is to group the elements of a wireframe model to widget-components, as defined above (section 4.5.1). In order to do so, we made a few assumptions, based on best practices when modularizing a UI Design:

- Elements in different wireframes which are similar in size, position, style, etc. are most likely to represent the same element. We call these elements shared elements.
- Wireframes which have no shared elements visualize different views in an application, e.g. login and home screen of an application. Wireframes which share a lot of similar elements are most likely to represent one view of the application.

- Elements which are shared by most wireframes are usually ubiquitous elements in the application, e.g. the buttons in a header or footer of the application.
- Elements are grouped to widget-components, so that no two components in one wireframe visually overlap.

Based on these assumptions, the generation of a component tree of a single wireframe model can be summarized in three simple steps:

1. Pre-processing of the wireframes to be able to compare the wireframes to each other
2. Finding elements that are similar in multiple wireframes and representing the wireframes and their similar elements as a graph
3. Grouping elements to widget-components and building the component tree by decomposing the graph

In the following sections we are going to explain those steps in detail.

Pre-processing

Even though the component tree relies heavily on the wireframe model, we still need to preprocess the wireframes to remove unnecessary information. Wireframes of a given wireframe model are taken into account, according to the following constraints:

- $W_{oneFrame}$ = set of wireframes with exactly one frame ⁵
- $W_{notEmpty}$ = set of all wireframes with at least one interaction-element inside the frame

Furthermore, we are focusing on elements, that fulfill the following constraints:

- $E_{insideFrame}$ = set of elements that are inside the frame
- $E_{interactionElement}$ = set of elements from type interaction-element
- $E_{notFrame}$ = set of elements other from type frame

Let the set of *Wireframes* and *Elements* be:

$$Wireframes = w_{oneFrame} \cap w_{notEmpty}$$

$$Elements = e_{insideFrame} \cap e_{interactionElement} \cap e_{notFrame}$$

The next step of the pre-processing groups nested or overlapping elements and defines them as one single group element.⁶

Although there may be interactions specified in the wireframe model, by construction the derivation of the component tree is independent from interactions, as well as the operations and steps linked to them (see fig. 4.9).

⁵As mentioned before, a frame is an element, e.g. phone or browser window, used to improve understanding during the wireframe process.

⁶Note that, by construction, no two elements of a single wireframe overlap after the pre-processing step.

Finding shared elements

A wireframe can be thought of as a canvas holding all elements necessary to describe the state of a user interface. The designer can position and scale the frame and its elements freely in each wireframe. We refer to the size and position of an element as the size and position relative to its frame.

When creating wireframes, the designer usually sketches similar elements multiple times, for example the buttons in the header of an application. The challenge is to identify these elements across different wireframes. As there is no global definition of similar elements, we defined a basic definition of similarity which has to be adapted for every type of element⁷. Necessary but not sufficient conditions for two elements $e_1, e_2 \in \text{Wireframes} \wedge e_1.\text{parent} \neq e_2.\text{parent}$ to be similar ($x \sim_{elem} y$) are:

- $e_1.\text{area} = e_2.\text{area}$
- $e_1.\text{style} = e_2.\text{style}$
- $e_1.\text{type} = e_2.\text{type}$

Generating the shared elements graph

For further processing, we describe the relation between wireframes and similar elements with the help of a pseudo-graph. A pseudo-graph is a graph which is permitted to have multiple edges, edges with the same source and target nodes. Additionally it is allowed to have multiple loops on the same node. Formally, our shared element pseudo-graph G is an ordered pair $G = (V, A)$ with:

- $V = \text{set of vertices} = W_{oneFrame} \cup W_{notEmpty}$
- $A = \text{a multiset of unordered pairs of vertices, so called arcs, in this case, two nodes are connected via one or more arcs if they share one or more elements}^8$. To define A we use the following algorithm:

Algorithm 3: Defining the set of arcs

```

Data:  $SharedElements = Elements \setminus \sim_{elem}$ 
1 begin
2   foreach  $sharedElement$  in  $SharedElements$  do
3     if  $sharedElement = \{e\}$  then
4        $A = A \cup (e.\text{parent}, e.\text{parent}, sharedElement)$ 
5     else
6       foreach  $e_1, e_2$  in  $sharedElement$  do
7         if  $e_1 \neq e_2$  then
8            $A = A \cup (e_1.\text{parent}, e_2.\text{parent}, sharedElement)$ 
9         end
10      end
11    end
12  end
13 end

```

⁷For details see the source code on the enclosed CD.

⁸Two wireframes w_1, w_2 share the same elements e_1, e_2 if $e_1 \sim_{elem} e_2 \wedge e_1 \in w_1$ and $e_2 \in w_2$.

Decomposing the shared-elements graph

The shared elements graph G gives an overview of the number of shared elements in a wireframe model. A strongly connected graph determines that we have less widget-components and the wireframes are mostly compositions of these components. The resulting component tree will have a small depth with less composite-widgets and proportionally more widget-components. If we have a large number of unconnected sub-graphs, we are going to have more widget-components and each wireframe in the model shows different elements. The resulting component tree architecture has a greater depth and more composite-components.

During the decomposition of the graph, we are going to remove edges of the graph. The basic idea behind the decomposition is to identify elements which are shared by most or all of the wireframes and group them to widget-components.

The following recursive algorithm 4 shows how we are going to decompose the graph and reduce the number of edges in every iteration. Definitions on bi-connectivity and cut points used in the algorithm can be found in [Gro04].

Grouping elements to widget-components

As described in section 2.3 our goal is to achieve maximum reusability of components without suffering from disproportional complexity. The following equation describes in detail how we group elements to widget-components, based on the assumption we made before (section 4.5.2): No two components in the same wireframe are allowed to intersect.

Let $intersects(area_1, area_2)$ determines whether or not $area_1$ and $area_2$ intersect.
 $getBoundingBox(component)$, be the bounding box of the elements in $component$.

$$\begin{aligned} components = \{ & c \subseteq candidates \wedge \forall otherElem \in otherElements \\ & \neg intersects(otherElem.area, getBoundingBox(c)) \\ & \wedge \forall otherComp \in otherComponents \\ & \neg intersects(getBoundingBox(otherComp), getBoundingBox(c)) \\ & \wedge |c| \text{ is maximum} \} \end{aligned}$$

4.5.3. Equality

In fig. 4.9 we can see that the component tree is independent from the task model and the data model. Moreover, by construction, our component tree is invariant against the following: all wireframes that can be built by compositions of elements of other wireframes do not affect the derived component tree. For example, adding or removing copies of the same wireframe in one model does not change the resulting component tree. Formally, we define the following equality relation: Let G_i be the shared-elements-graph of $wireframeModel_i$ as described in section 4.5.2. Let \sim_{graph} be defined as:

$$G_1 = \langle V_1, A_1 \rangle \sim_{graph} G_2 = \langle V_2, A_2 \rangle \Leftrightarrow \forall v \in V_2 \setminus V_1. \exists S \subseteq V_1. elements(\{v\}) \sim_{elemSet} elements(S)$$

Let $elements(v)$ be the solution to the following:

$$\{e | \exists v_2, elements.e \in elements \wedge (v, v_2, elements) \in A\}$$

Let $elements(\{v_1, \dots, v_n\})$ be the solution to the following:

$$\bigcup elements(v_i) \wedge 1 \leq i \leq n$$

Let A, B sets of elements,

$$A \sim_{elemSet} B \Leftrightarrow \exists a \in A \wedge b \in B. a \sim_{elem} b$$

By construction, we can see that:

$$Tree(G_1) = Tree(G_2) \Leftrightarrow G_1 \sim_{graph} G_2$$

4.5.4. Example

In the following we are going to explain our algorithms step by steps with the "my to-do app" example.

Pre-processing

After the pre-processing step, we are left with 10 non intersecting elements (see fig. 4.10).



Figure 4.10.: "my to-do app" - after pre-processing

Finding shared elements

We can see that $e_3 \sim_{elem} e_7 \wedge e_4 \sim_{elem} e_8$. The similarity definition does not apply to any other elements in the given wireframes.

Decomposing the Graph and Building Components

Based on the shared elements we found, we can define the sets *Elements* and *SharedElements* as followed:

$$Elements = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$$

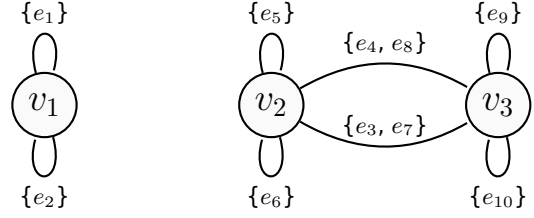
$$SharedElements = \{\{e_1\}, \{e_2\}, \{e_3, e_7\}, \{e_4, e_8\}, \{e_9\}, \{e_{10}\}\}$$

The initial shared-elements-graph G_1 we build based on the infoemtaion of the shared elements is unconnected (see Iteration: 1). To be able to derive the component tree, the algorithm to decompose

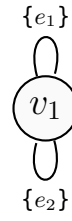
the graph (see) is called with G_1 and the *ROOT* composite component. In the following, we explain in detail how our algorithm works. Each iteration shows the input graph on the right hand side. On the left hand side we refer to the line numbers of the decompose graph algorithm and provide detailed information on the intermediate results.

Iteration: 1

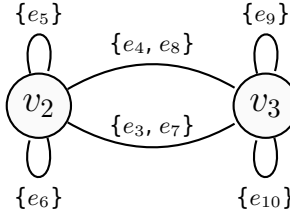
line 21: G_1 is unconnected.
 line 22: $Subgraphs = \{G_{11}, G_{22}\}$
 line 25-26: $decomposeGraph(G_{11}, ROOT)$
 (see Iteration 1.1)
 line 24-29: $CComponents = \{ROOT, Comp\}$
 $parent(Comp) = ROOT,$
 $decomposeGraph(G_{12}, Comp)$
 (see Iteration 1.2)

**Iteration: 1.1**

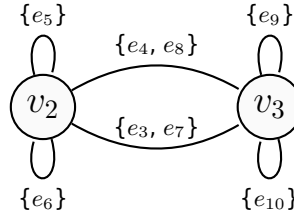
line 5: G_{11} has only loop edges
 line 6: v_1
 line 7-10: $buildComponents(\{\{e_1\}, \{e_2\}\}, \emptyset, \emptyset)$
 line 11: $WComponents = \{Login\}$
 $Login = \{\{e_1\},$
 $\{e_2\}\}, parent(Login) = ROOT$

**Iteration: 1.2**

line 27-28: $CComponents = \{ROOT, Comp\},$
 $parent(Comp) = ROOT$
 line 29: $decomposeGraph(G_{12}, Comp)$
 (see iteration 1.2.1)

**Iteration: 1.2.1**

line 18: G_{121} is connected
 line 19: $G_{split} = splitGraph(G_{12})$
 $gE = \{\{e_4, e_8\}, \{e_3, e_7\}\}$
 $oE = \{\{e_5\}, \{e_6\}, \{e_9\}, \{e_{10}\}\}$
 $buildComponents(gE, oE, \emptyset)$
 $WComponents = \{Login, Header\}$
 $Header = \{\{e_4, e_8\}, \{e_3, e_7\}\},$
 $parent(Header) = Comp$
 line 20: $decomposeGraph(G_{split}, Comp)$
 (see Iteration 1.2.1.1)



Iteration: 1.2.1.1

line 5: G_{1211} has only loop edges

line 6: v_2

line 7-10: $buildComponents(\{\{e_5\}, \{e_6\}\}, \emptyset, \{Header\})$

line 11: $WComponents = \{Login, Header, TodoList\}$

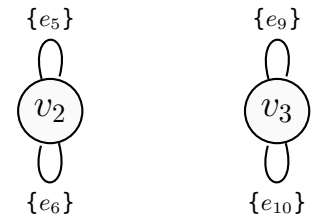
$TodoList = \{\{e_5\}, \{e_6\}\},$
 $parent(TodoList) = Comp$

line 6: v_3

line 7-10: $buildComponents(\{\{e_9\}, \{e_{10}\}\}, \emptyset, \{Header\})$

line 11: $WComponents = \{Login, Header, TodoList, NewTodo\}$

$TodoList = \{\{e_9\}, \{e_{10}\}\},$
 $parent(TodoList) = Comp$



Building the component tree

The resulting component tree (see fig. 4.11) consists of *composite – component* = $\{ROOT, Comp\}$ and *widget – components* = $\{Login, Header, TodoList, NewTodo\}$.

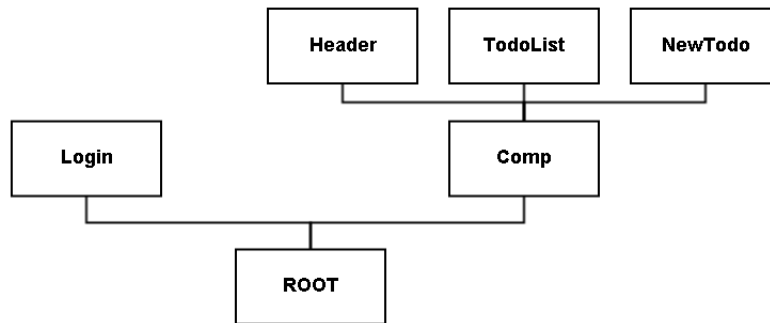


Figure 4.11.: my to-do app - Component Tree

4.6. Evaluation of UI Designs

In this section, we present a set of metrics to analyze and evaluate a user interface design according to its size and complexity. Other than the heuristic evaluation of a user interface [Nie94], we focus on the illustration of the impact on software construction and the overall development process. With these metrics we can assure that all tasks and data objects are embodied in the user interface design and visualized through one or more wireframes.

Components overview per Task This overview enables the developer to make more knowledgeable decisions and gain better insight into the effect each interface element, or the component has on a given task.

Data overview per task Providing an overview of the data objects created, read, updated or deleted during a task can assist the developer when estimating the cost and complexity when implementing a task related feature.

Usage overview Determines whether a data object specified in the domain model, is created, read, updated or deleted. It enables the developer to quickly identify unused data objects during the designing process.

Access Overview By listing the roles related to the operations they are able to perform in a given design, the developer can immediately identifying any permission violations.

Components per wireframe Estimating size and complexity is an ongoing issue in all software engineering methods. Providing extended feedback on the components used to implement a certain wireframe can assist during this process.

Observed data objects per wireframe Giving an overview of the elements that observe one or more data objects in allows the developer to estimate the complexity of a sketched wireframes. Wireframes that do not contain elements that trigger interactions which perform or observe data objects cost less time and resources during implementation.

Reusability of widget-components Highly reusable components are desirable, they reduce the overall lines of code and improve maintainability. Having a continuous feedback on the usage of the generated components facilitates cost and complexity estimation of a user interface design.

4.6.1. Example

Based on the views we derived in the previous sections, we can calculate the following metrics for our example app, the "my to-do app".

Data overview for: *task_addTodo*

Data Object	CREATE	READ	UPDATE	DELETE
<i>user</i>	0	1	0	0
<i>todoList</i>	0	1	1	0
<i>todo</i>	1	0	0	0

Component overview for: *task_addTodo*

Composite-components *ROOT*
 Widget-Components *Login, Header, TodoList, NewTodo*

Usage Overview

Unused \emptyset
 Unchanged *user*

Access Overview

Role	<i>user</i>	<i>todoList</i>	<i>todo</i>
<i>user</i>	<i>READ</i>	<i>READ, UPDATE</i>	<i>CREATE</i>

Number of component per wireframe

Wireframe	Components
<i>wireframe_login</i>	<i>Login</i>
<i>wireframe_todoList</i>	<i>Header, TodoList</i>
<i>wireframe_newTodo</i>	<i>Header, NewTodo</i>

Observed data objects per wireframe

Wireframe	Data Objects
<i>wireframe_login</i>	\emptyset
<i>wireframe_todoList</i>	\emptyset
<i>wireframe_newTodo</i>	\emptyset

Reusability of widget-components

Component	Used abs.	Used perc.
<i>Login</i>	1	0.25
<i>Header</i>	2	0.5
<i>TodoList</i>	1	0.25
<i>NewTodo</i>	1	0.25

Algorithm 4: Decompose shared-elements-graph

Input: Shared elements graph $G = (V, A)$, current parent node $parentComponent$

Data: $getElements(v_1, \dots, v_n) = \bigcup \{elements \mid \forall 1 \leq i, j \leq n. \langle v_i, v_j, elements \rangle \in A\}$
 $WComponents$ is the set of widget-components, which are defined as a subset of $SharedElements$.
 $CComponents$ is the set of composite-components, identified by markers.
 $parent : (Components \cup CComponents) \rightarrow CComponents$, initially empty
 $getComponents(v_1, \dots, v_n) = \{component \mid \forall elements \in component. component \in WComponents \wedge elements \in getElements(v_1, \dots, v_n)\}$

```

1 begin
2   if  $A = \emptyset$  then
3     | return;
4   end
5   if  $\forall \langle v_1, v_2, elements \rangle \in A. v_1 = v_2$  then
6     | // graph has only loop edges
7     | foreach  $v$  in  $V$  do
8       | // build widget component for each vertex
9       |  $componentCandidates = \{e \mid e = A(v, v)\}$ ;
10      |  $otherElements = \emptyset$ ;
11      |  $otherComponents = getComponents(v)$ ;
12      |  $components = buildComponents(componentCandidates, otherElements,$ 
13      |  $otherComponents)$ ;
14      | // add components to set of widget components
15      |  $WComponents = components \cup WComponents$ ;
16      | foreach  $c$  in  $components$  do
17        | // set parent for each component
18        |  $parent(c) = parentComponent$ ;
19      | end
20    end
21    return;
22  end
23  if  $G$  is connected then
24    |  $G = splitGraph(G, parentComponent)$ ;
25    |  $decomposeGraph(G, parentComponent)$ ;
26  else
27    |  $Subgraphs =$  set of connected components;
28    | foreach  $G_s = (V_s, A_s)$  in  $Subgraphs$  do
29      | if  $|V_s| = 1$  then
30        | // graph has only loop edges
31        |  $decomposeGraph(G_s, parentComponent)$ ;
32      | else
33        | // create new composite-component  $composite$  with a new marker
34        |  $CComponents = CComponents \cup \{composite\}$ ;
35        |  $parent(composite) = parentComponent$ ;
36        |  $decomposeGraph(G, composite)$ ;
37      | end
38    end
39  end
40 end

```

Algorithm 5: Split Graph

Input: Shared elements graph $G = (V, A)$, current parent node $parentComponent$
Output: Shared-elements-graph $G = (V, A)$

```

1 begin
  // let global elements be the set of elements that all wireframes share
2   $globalElements = \{e | \forall v_1, v_2 \in V. e \in A(v_1, v_2)\};$ 
3   $otherElements = getElements(V) \setminus globalElements;$ 
4   $otherComponents = getAllComponents(V);$ 
5   $components = buildComponents(globalElements, otherElements, otherComponents);$ 
  // add components to set of widget components
6   $WComponents = components \cup WComponents;$ 
7  foreach  $c$  in  $components$  do
  | // set parent for each component
  |  $parent(c) = parentComponent;$ 
8  end
9  end
  // remove cutpoint elements from Graph
10  $A = A \setminus \{\langle v_1, v_2, elements \rangle \mid \langle v_1, v_2, elements \rangle \in A \wedge elements \in globalElements\};$ 
11 if  $G$  is connected then
12   if  $G$  is biconnected then
  | // no heuristic for further decomposition
  |  $component = getElements(V);$ 
  |  $WComponents = component \cup WComponents;$ 
  |  $parent(component) = parentComponent;$ 
  |  $A = \emptyset;$ 
  | return  $G;$ 
13
14
15
16
17   else
18      $cutpoints = \text{set of cut points of } G;$ 
19     // Let  $cutpointElements$  be the set of all elements between all cut points
20      $cutpointElements =$ 
      $\cup \{elements \mid \forall v_1, v_2 \in cutpoints. (A(v_1, v_2) = elements \wedge v_1 \neq v_2)\};$ 
     // build widget components by cut point elements
21      $vertices = \{v \mid \exists \langle v, v_1, elements \rangle \in A. elements \in cutpointElements\}$ 
      $otherElements = getElements(vertices);$ 
22      $otherComponents = getComponents(vertices);$ 
23      $components = buildComponents(cutpointElements, otherElements,$ 
      $otherComponents);$ 
     // add components to set of widget components
24      $WComponents = components \cup WComponents;$ 
25     foreach  $c$  in  $components$  do
  | // set parent for each component
  |  $parent(c) = parentComponent;$ 
26     end
27     // remove cut point elements from Graph
28      $A = A \setminus \{\langle v_1, v_2, elements \rangle \mid \langle v_1, v_2, elements \rangle \in A \wedge elements \in cutpointElements\};$ 
29     return  $G;$ 
30   end
31 end
32 return  $G;$ 
33 end

```

5. Tool: Wireframe2Artefact

In this section, we present Wireframe2Artefact, a model-based software tool that supports user interface design. We start with a brief overview of the design principles our tool is based on (section 5.1). Then we introduce the XML-based underlying meta-level modelling languages (section 5.2). This is followed by a detailed explanation of the features provided by Wireframe2Artefact (section 5.3). Next we give a brief overview of the architecture of the tool and used technologies (section 5.4). In the last section we summarize the current limitations (section 5.5).

5.1. Overview

Wireframe2Artefact is a software tool that allows the designer to represent all relevant aspects of a user interface by capturing the relation between wireframes, the user tasks and its data objects.

A central activity of any user interface design is the specification of a user task model. It requires close collaboration between domain experts, interface designers and engineers. Wireframe2Artefact involves the end user directly in the development of the user interface design model. The user describes the task, in terms of tasks, steps and roles. The designer then identifies key interactions and connects the task model to the wireframe model by refining the interactions. Wireframe2Artefact is then able to generate storyboards for each task. These storyboards can be used to communicate and evaluate requirements and help to reduce the chance of misunderstood requirements. The user can visualize how each interaction relates to a task.

In parallel, a data model can be specified and linked to the wireframe model, by adding operations that are triggered by an interaction. Wireframe2Artefact is then able to generate a data flow diagram for each task specified in the task model. These simple connections help the developer to identify what data objects are represented through a certain element in a wireframe.

Based on the wireframe model, Wireframe2Artefact provides an educated guess on the frontend architecture by deriving a component tree. Our tool does not emphasize the generation of code, but the complete user interface design is available in a declarative form and can be reused in other applications to do so.

The clear connection between representation of the user interface, the user tasks and the data model, makes it easy to track the way the design relates to the constructed interface model. The user does not need to know the structure of our XML-based underlying modelling language in order to use our tool. We provide a low barrier of entry and an open environment that allows to cooperate with existing tools in the industry.

5.2. User Interface DSL

In this section we introduce our XML-based user interface modelling language, based on the model we proposed in section 4.2. To support reuse (Requirement 5, Requirement 6) of the artefacts specified for the user interface model, we decided to split the user interface language into three connected languages:

- WXML (Wireframe eXtensible Markup Language), to specify the wireframes, interactions and operations
- TXML (Task eXtensible Markup Language), to specify the tasks, steps and roles
- DXML (Data eXtensible Markup Language), to specify data objects and their associations

The schema for our languages is defined in RELAX NG (REgular LAnguage for XML Next Generation) ¹, a schema language for XML. An example of the schema definition of a button in a wireframe can be found in 5.1.

```
1 button-elem = element button {
2     BasicInteractionElement,
3     attribute pointy {direction-attr}?,
4     attribute switch {"on" | "off"}?,
5     attribute selectedState {selectedState-attr},
6     menu-elem?,
7     label-elem?
8 }
```

Listing 5.1: "RELAX NG Button schema"

With the help of trang², a multi-format schema converter based on RELAX NG, we converted our RELAX NG schema into an XML schema. This can be done with a simple command line call (listing 5.2).

```
1 java -jar trang.jar foo.rnc > bar.xsd
```

Listing 5.2: "Trang command line call"

Trang aims to preserve all aspects of the schema that may be significant to a human reader, listing 5.3 shows an example of the converted schema snippet we introduced before (listing 5.1).

```
1 <xs:element name="button">
2   <xs:complexType>
3     <xs:complexContent>
4       <xs:extension base="BasicInteractionElement">
5         <xs:sequence>
6           <xs:element minOccurs="0" ref="menu"/>
7           <xs:element minOccurs="0" ref="label"/>
8         </xs:sequence>
9         <xs:attribute name="pointy" type="direction-attr"/>
10        <xs:attribute name="switch">
11          <xs:simpleType>
12            <xs:restriction base="xs:token">
13              <xs:enumeration value="on"/>
14              <xs:enumeration value="off"/>
15            </xs:restriction>
16          </xs:simpleType>
17        </xs:attribute>
18        <xs:attribute name="selectedState" use="required" type="selectedState-attr"/>
19      </xs:extension>
20    </xs:complexContent>
21  </xs:complexType>
22 </xs:element>
```

Listing 5.3: "XSD Button schema"

¹<http://relaxng.org/>

²<https://code.google.com/p/jing-trang/>

Analog to the XML schema, we defined schemas for the task model (TXML) and the data model (DXML). All schemas have been implemented as described in section 4.2.

5.3. Features

Wireframe2Artefact offers a limited, but yet sufficient set of features to design a user interface and capture the interrelation between the specified models. It offers a separate view for each model in order to sufficiently represent all information. The following sections give an overview of the currently implemented features of Wireframe2Artefact:

Load Models

Our reference tool does not offer the possibility to specify the input models within the tool. The user starts the design process by loading a set of data. He is allowed to load the following:

- mandatory: a set of BMML files or a single WXML file to specify the wireframe model
- recommended: a TASKXML file to specify the task model
- recommended: a DATAXML file to define the data model

Afterwards the files are read, converted to our underlying user interface model and post-processed (for further details see section 5.4) by the system.

Model Views

For better intelligibility and a clear distinction between the concepts: wireframes, tasks and data objects, a tab for each sub-model is provided by Wireframe2Artefact:

Wireframe Model Tab (fig. 5.1) Finding a suitable and self-descriptive visualization for the wireframe model was one of the obstacles we had to overcome. The wireframe model shows the state of a system, while at the same time describes the users' interaction with the system. As mentioned before, most wireframing tools, such as Balsamiq and WireframeSketcher offer the user the possibility to define these interactions, but provide no features to visualize the connections between the wireframes within their tool. We solved this problem by displaying the wireframe model as a so called "navigation map" (a). The navigation map is a directed graph, with each wireframe represented as a node in the navigation map. An edge is described by an interaction triggered by an element in a wireframe. The source of the edge is defined by the parent of the source element. The target is defined by the interactions target wireframe. By clicking on an edge, our tool provides detailed information on the interaction (b), such as trigger, duration, delay and effect. When selecting a node, detailed information of the selected wireframe is shown (c).³

Task Model Tab (fig. B.2) End-users or domain experts typically have no background in modelling languages and aim to describe and read a task as informal as possible. Our task tab collects all specified task in a list, by selecting a task, detailed information, such as name, description, steps and roles are displayed.

³Note that it is only possible to display the images of a wireframe when the images has been uploaded with the model. Furthermore, they have to have the same file name as the corresponding BMML file. Using Balsamiq this can be done by the "export all Mockups to PNG" feature.

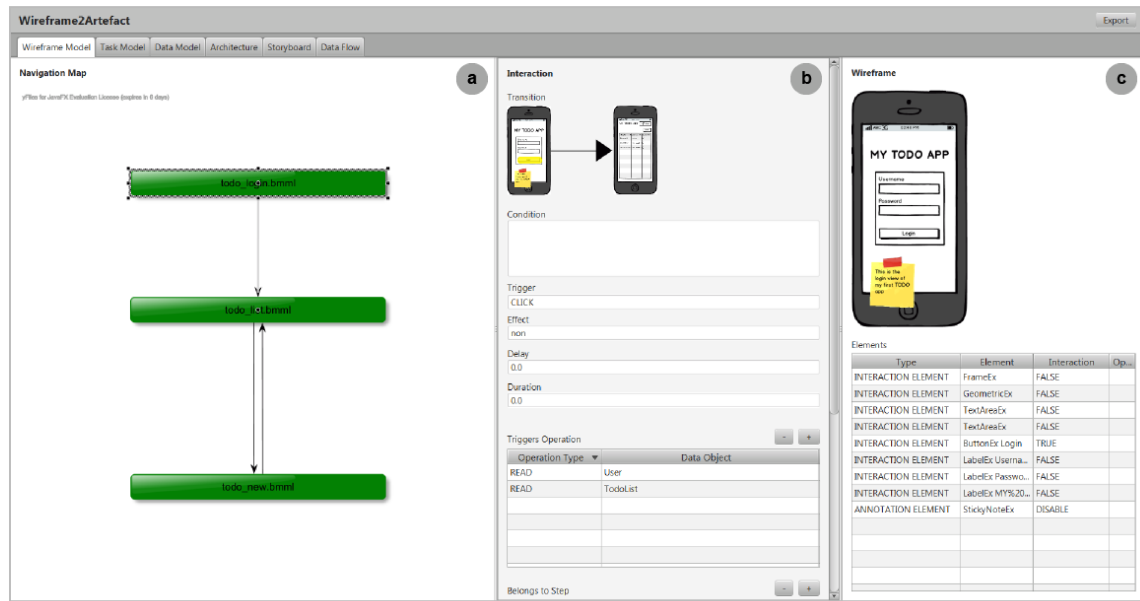


Figure 5.1.: Screenshot of the wireframe model tab

Data Model Tab (fig. B.3) One of our requirements was to hide the model from the user in order to avoid being slowed down by modelling details. Hence, we choose a simplified UML class diagram notation to visualize our data model. This way we ensure self-descriptiveness and a low barrier of entry as claimed in the requirements section (Requirement 3, Requirement 4).

Refine interactions

Reflecting the previous chapters, we defined the wireframe model as the foundation of our approach. In order to define the user interface design the designer links the user tasks and the data objects to the wireframe model by refining the transitions of the wireframe model. Both can be done in the wireframe tab of our tool. To add an operation to an interaction, the user clicks on the '+' button above the operation table (fig. 5.1, a). He then chooses an operation type (CRUD) and a data object (as defined in the data model) from a combobox.

Our implementation is currently limited to one transition per interaction; hence the user refines interactions instead of transitions. The user can add and remove steps that belong to an interaction. Furthermore, the user is able to add and remove operations that are triggered by an interaction.

Generate Views

Wireframe2Artefact offers a tab for each view that can be generated: Storyboards, data flow diagrams and component tree. Using the toolbar in each tab, the user can generate the desired view. A screenshot of each view can be found in fig. B.4, fig. B.5 and fig. B.6.

5.4. Architecture

Wireframe2Artefact is implemented in JavaFx. The core of the system consists of four main components: Model Converter, Storyboard Generator, Data Flow Generator, and Component Tree Generator. The model converter is responsible to read and convert BMML files into a single WXML file. The generators derive the corresponding view from the loaded user interface model. We use JAXB (Java Architecture for XML Binding) to map our Java classes to our user interface model languages (WXML, TXML, DXML).

BMML to WXML converter

Balsamiq stores each wireframe in a custom XML-based file format, the BMML format. Figure 5.2 shows how we convert multiple BMML files into one WXML file. In order to convert a BMML wireframe object into a WXML wireframe object with JAXB, we defined the schema of the BMML language (appendix D.1). The model converter then reads a folder with BMML files and the defined schema. Afterwards JAXB unmarshalls the BMML files into java objects and we are able to convert the objects. Last, we are able to marshal the java objects into our WXML format and write a WXML file.

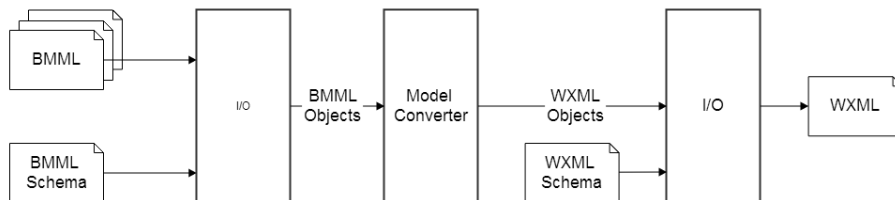


Figure 5.2.: BMML to WXML

Name (job title)	▲ Age ◆	Nickname	Employee ▼
Giacomo Guilizzoni Founder & CEO	37	Peldi	<input type="radio"/>
Marco Botton Tuttofare	34		<input checked="" type="checkbox"/>
Mariah Maclachlan Better Half	37	Patata	<input type="checkbox"/>
Valerie Liberty Head Chef	:)	Val	<input checked="" type="checkbox"/>
Guido Jack Guilizzoni	6	The Guids	<input type="checkbox"/>

Figure 5.3.: Grid sketched with Balsamiq

Even though the BMML format is quite simple, we dealt with some challenges, for example: A grid defined in BMML (see listing 5.4) stores all information in the `<text/>` tag (see 5.4). The conversion of these URL encoded, coma separated string became a chore in and on itself.⁴ The corresponding representation of the grid in WXML can be found in fig. 5.3.

⁴Wireframe2Artefact does not support the conversion of every possible input yet.


```

1 <control controlId="0" controlTypeID="com.balsamiq.mockups::DataGrid" x="60" y="20" w="-1"
  h="-1" measuredW="340" measuredH="195" zOrder="0" locked="false" isInGroup="-1">
2   <controlProperties>
3     <text>Name%5Cr%28job%20title%29%20%5E%2C%20Age%20%5Ev%2C%20Nickname%2C%20Employee
      %20v%0AGiacomo%20Guilizzoni%5CrFounder%20%26%20CE0%2C%2037%2C%20Peldi%2C%20%28
      o%29%0AMarco%20Botton%5CrTuttofare%2C%2034%2C%20%2C%20%5Bx%5D%0AMariah%20
      Maclachlan%5CrBetter%20Half%2C%2037%2C%20Patata%2C%20%5B-%5D%0AValerie%20
      Liberty%5CrHead%20Chef%2C%20%3A%29%2C%20Val%2C%20%5Bx%5D%0AGuido%20Jack%20
      Guilizzoni%2C%206%2C%20The%20Guids%2C%20%5B%5D%0A%7B64L%2C%200R%2C%2035%2C%200
4     </text>
5   </controlProperties>
6 </control>

```

Listing 5.4: "Grid in BMML"

```

1 <table hasHeader="true" rowHeight="30" state="default" xPos="560" yPos="62" width="340"
  height="195" zOrder="5">
2   <row selected="false">
3     <cell>
4       <label>
5         <text bold="false" undelined="false" strikethrough="false" italic="false"
          fontSize="13" orientation="default" color="black" opacity="1.0" align="
          tr">Name\n(job title)
6         </text>
7       </label>
8     </cell>
9     ...
10  </row>
11  ....
12 </table>

```

Listing 5.5: "Grid in WXML"

Generators

The generators of Wireframe2Artefact have been implemented as described in chapter 4. We used JGraphT⁵, a free Java graph library that provides an ample set of graph-theory algorithms for the decomposition of the shared elements graph (algorithm 4).

5.5. Current Limitations

Being a proof of concept, the current implementation of Wireframe2Artefact still lacks some functionality that would be desirable for real-world projects. This section mentions some issues we are aware of and which do not require major conceptual work to be implemented. We discuss more complex open issues that would require substantial research or conceptual improvements in chapter 7 and in section 8.2.

Extended Balsamiq Parser Wireframe2Artefact is able to read and convert the basic elements described in a set of BMML files to our underlying model (WXML). We are still lacking some functionalities in our model converter. An example is the conversion of the styles defined in BMML.

⁵<http://jgraph.org/>

These features are crucial to allow lossless conversion from a WXML file back into BMML files. Furthermore, the BMML file specification allows the user to define custom properties for each element (control). Balsamiq does not use these properties, but stores them in the BMML file. This allows us to losslessly convert our interaction and operation information specified in the WXML file. The conversion from WXML to BMML can be implemented analog to the conversion from BMML to WXML.

Tool Support The industry has brought up an extensive number of tools to target particular needs of the user interface design. In order to target the needs of a real-world software project, it would be desirable to offer extensive tool support for:

- Rapid wireframing tools other than Balsamiq
- UML or UML drawing tools to simplify the specification of the data model through class diagrams or entity relationship diagrams
- Tools to specify use-cases or user stories to simplify the modelling of user tasks

This could be targeted by providing a mechanism to exchange documents of outside tools without losing the specified relations. This mechanism could be realized analog to the mechanism we proposed for the Balsamiq wireframing tool.

Performance Optimization During the implementation of Wireframe2Artefact we did not focus on performance issues. Every time the information of the input models changes, all artefacts (storyboards, data flow diagrams and component tree) are derived. We recommend to use the information of the equality relations (section 4.3.3, section 4.4.3, section 4.5.3) to determine whether a component tree, storyboard or data flow diagram has to be updated or not.

Extend Functionality Our reference implementation focuses on the most important tasks, such as linking steps and operations to interactions. Concerning the usability of this tool, it would be handy to have a more extended set of functions:

- Editing of the tasks model as well as the data model within the tool
- Balsamiq adapter to allow "editing" Balsamiq wireframes within the tool
- Graphic representation of the data flow model

6. Case Study

In this chapter, we will put Wireframe2Artifact to the test and show that it is useful for real-world applications. First, we will specify the user interface with Wireframe2Artefact and second compare the component tree derived by the tool to the actual implementation of the project. Our example is a time account web-application which has been developed at msg systems ag.

This application is well suitable for our case study, because it has a significant size, but its user interface and functionalities are still easy to understand.

6.1. P1-Timesheet App

P1-Timesheet is a time account web-application designed and developed at msg systems ag. The development of the P1-Timesheet application started in summer 2013. Version 1.0.0 has been released in February 2014 and is intended to be used by approximately 2000 employees on a daily basis at the end of 2014. Its component-based frontend architecture is implemented using the componentjs framework [Enga]¹, based on the approach presented by [Engc].

It is used to manage working hours and holidays of the employees and its main functionality can be split down to the following activities: The user has to login to the system in order to use the app. He can browse through his bookings, which show booked working hours and remaining holidays. He can switch between different views of his bookings, either displayed in a list, a table or a daily view. Furthermore, the app enables the user to add new entries to the booking list or copy an existing entry.

When the user logs in, the app verifies the users name and password and loads all data necessary to display the users booking list, so that navigating through the app does not require loading any additional data. When creating or copying an entry of the bookings, a new entry is created and the bookings updated.

6.2. Specification of the UI

The user interface development process of the P1-Timesheet app did not follow any particular approach. Most notably, they sketched an extensive set of 65 wireframes, which are very detailed and highly interactive. All wireframes have been created with the Balsamiq wireframing tool. For our evaluation we are going to use reduced subset of the specified wireframes, due to a high number of redundant and obsolete wireframes. Furthermore, they documented the initial user tasks in a small use-case diagram. The main data objects and their relations have been documented as an UML class diagram. The full example and all artefacts mentioned above can be found on the CD enclosed to this thesis.

It is important to know that they did not create any storyboards or sequence diagrams during the UI design process. Even though they sketched the wireframes based on the tasks they documented, they did not document the relation between wireframes and tasks.

¹<http://componentjs.com/>

This is an example how beneficial Wireframe2Artefact can be in the development of a user interface. Even though, the developers knew how each wireframe was related to a particular task, they had no possibilities to document or synchronize this relationship within the wireframes themselves.

In the following, we are going to show how the specification of the user interface looks after loading the original wireframes, tasks and data objects. Furthermore we are going to enhance the wireframes by refining the interactions according to the information provided by the project manager.²

We start with a snippet of the user tasks, which can be seen in listing 6.1. It shows the login task and its two simple steps in TXML. We did focus on the three major task, login, create and copy and entry during this evaluation, because they promised to be the most demanding in terms of their corresponding interactions. Nevertheless, the designer is free to choose what interactions should or should not be refined. Specifying only parts of the user interface does not have any effect on the resulting output.

```

1 <task>
2   <id>4a5c1ca4-6617-4184-a505-8c1c14bd5fe4</id>
3   <title>Login</title>
4   <description>The user wants to login to the system.</description>
5   <role name="User"/>
6   <step id="e7268d88-5f1c-489d-a13c-4f60dcd6eeba" name="Enter password and username"/>
7   <step id="0bdf0a7d-5a63-4683-837f-bd2923875cac" name="Click the 'Login' button"/>
8 </task>

```

Listing 6.1: "P1-Timesheet Task Model"

The following snippet (listing 6.2) shows a data object defined in DXML and matches the "User" class in the original P1-Timesheet UML class diagram³.

```

1 <data>
2   <id>092f69c1-a356-498f-a2d1-8b4b25b8d031</id>
3   <name>User</name>
4   <attribute name="givenname"/>
5   <attribute name="surname"/>
6   <attribute name="email"/>
7   <attribute name="monthdata"/>
8   <attribute name="pspelements"/>
9 </data>

```

Listing 6.2: "P1-Timesheet Task Model"

The next snippet (6.3) shows the specification for the login button of the application, before refining the interaction of the button. With Wireframe2Artefact we refined the interactions as the last snippet (6.4) shows the specification of the same button after refining the interaction. As mentioned before, the user is verified and all data is loaded after the login, which is documented through <operation/> and <step/>.

```

1 <button selectedState="default" state="default" xPos="363" yPos="485" width="81" height="
2   27" zOrder="1">
3   <interaction trigger="" duration="" delay="" effect="">
4     <transition>
5       <target>booking_moth.bmml</target>
6     </transition>
7   </interaction>
8   <Label state="default" xPos="363" yPos="485" width="81" height="27" zOrder="1">

```

²Note that user interface was not designed using the Wireframe2Artefact during the P1 project. This Evaluation has been made after the release of P1 and is based on the documents provided by the project manager.

³The class diagram and full specification with all data objects and their relations can be found on the enclosed CD.

```

8     <text bold="false" undelined="false" strikethrough="false" italic="false" fontSize="13
9       " color="black" opacity="1.0" align="center">Anmelden</text>
10  </label>
    </button>

```

Listing 6.3: "Snippet of P1-Timesheet Specification"

```

1 <button selectedState="default" state="default" xPos="363" yPos="485" width="81" height="
2   27" zOrder="1">
3   <interaction trigger="click" duration="0.0" delay="0.0" effect="non">
4     <transition>
5       <condition>username and password valid</condition>
6       <target>booking_moth.bmml</target>
7     </transition>
8     <step>0bdf0a7d-5a63-4683-837f-bd2923875cac</step>
9     <operation type="read">092f69c1-a356-498f-a2d1-8b4b25b8d031</operation>
10    <operation type="read">f10ad57c-d9f7-4325-b994-90fa0e89cf56</operation>
11    <operation type="read">a5bba82a-b14e-4391-983b-21b9c98a1547</operation>
12  </interaction>
13  <label state="default" xPos="363" yPos="485" width="81" height="27" zOrder="1">
14    <text bold="false" undelined="false" strikethrough="false" italic="false" fontSize="13
15      " color="black" opacity="1.0" align="center">Anmelden</text>
    </label>
  </button>

```

Listing 6.4: "Snippet of P1-Timesheet Specification"

Based on this specification, we are now able to derive the storyboards and the data flow diagrams for the three tasks we defined. The fact that no storyboards or data flow diagrams have been documented during the P1-Timesheet user interface development, makes it impossible to compare our output. A full set of the derived storyboards and data flow diagrams can be found in appendix C.

6.3. Comparison of the Component Trees

In the previous section we showed how the user interface of the app was defined with Wireframe2Artefact. We spotted the differences between the wireframes with and without refined interactions. Based on this specification we are able to generate the component tree and compare it to the implemented architecture of the app. In fig. 6.1 we can see a simplified version of the component tree of the P1-Timesheet application. It consists of four composite components: Root, Panel, BookingList and Page. Furthermore, it has nine widget components.

The component tree derived by Wireframe2Artefact (fig. 6.2) consists of 12 widget components and two composite components, including the root composite component. We can see that our approach derived slightly more components compared to the original component tree. In order to compare our component trees we named widget-component that group the same elements equal. Which means, the "Login" component in the original component tree groups the same elements as in "Login" component of the derived component tree.

We can see that our approach grouped most widget-components similar to the widget-components in the original tree. One of the most significant differences is the "Header" widget component. In the original tree, the header consists of three buttons and a label. The label text and size varies, depending on the booking view, which is displayed. Our approach could not identify these labels as similar elements (as described in section 4.5.2) and therefore split the header into a button group and the particular labels. Furthermore, we found an empty and invisible button in one of the wireframes, which is represented through the "EmptyButton" widget-component. Another difference can be

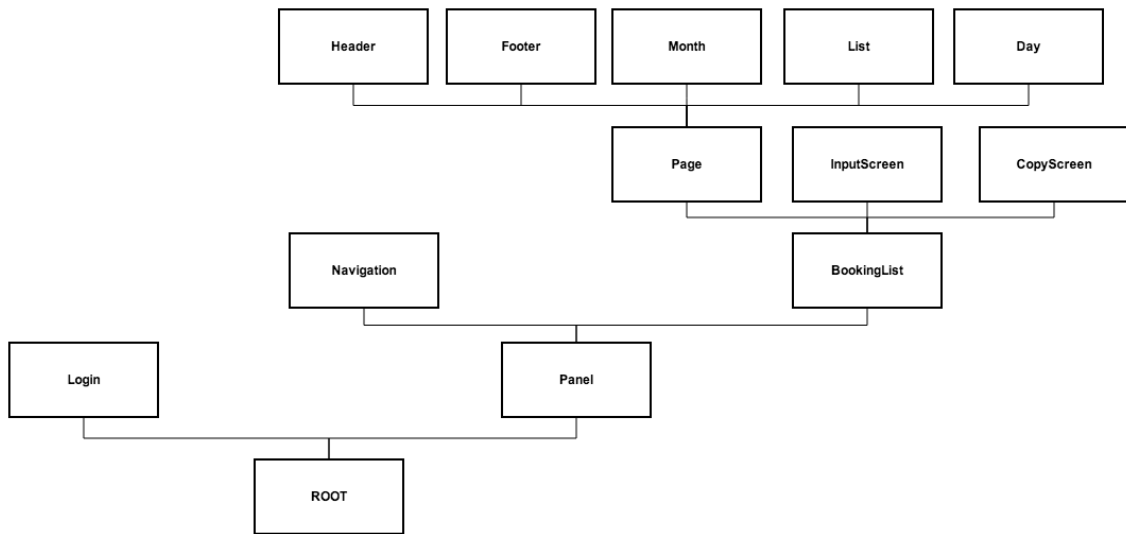


Figure 6.1.: Component Tree

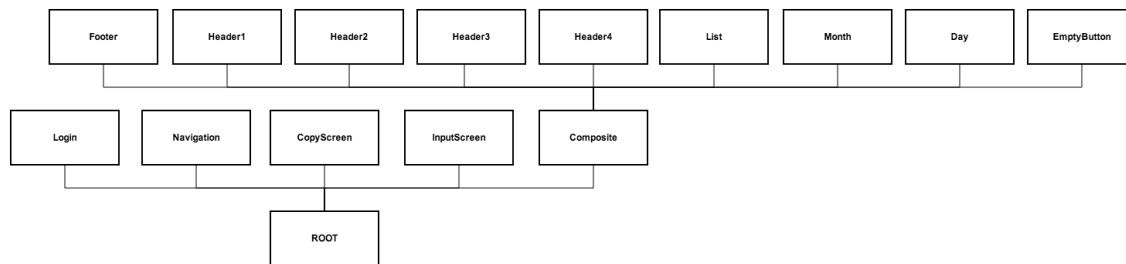


Figure 6.2.: Component Tree derived by Wireframe2Artefact

seen in the structure of the tree. The original tree has more composite components, compared to the derived tree. By construction our approach creates as few composite-components as possible. Whereas a software architect groups widget-components based on their logical cohesion in terms of the domain.

This comparison has shown that most of the widget components have been identified as implemented in the original tree. Nevertheless, designing architecture is a highly creative process and relies heavily on the experience of architect. Therefore did not focus on code generation during this thesis. Our main goal is it to provide engineers and developers a different view on the user interface, to be able to make more knowledgeable decisions when designing the architecture of a user interface design.

6.4. Lessons Learned

It is always highly interesting how such a tool performs when faced with a real-world project. The following sections describes some of the most notably findings we had during the specification of the user interface, as well as during the comparison of the component tree.

Usability Issues Even though Wireframe2Artefact is a prototype implementation, refining the interactions was very intuitive and was done very quickly according to the steps of the defined tasks. Furthermore, we figured that the navigation map (section 5.3) has proven very helpful to understand the possible user interactions across all wireframes. It was especially useful to determine whether an interaction was missing or pointed to the wrong target.

Expressiveness of the Model The expressiveness of our underlying model offered more than sufficient expressiveness to describe the user interface design for the P1-Timesheet app. We did not find any aspects that could not have been described with our modelling language. All input wireframes (BMML) have been losslessly converted into our underlying modelling language (WXML)⁴.

Usefulness of the Output The previous section has shown that our tool was able to derive a component tree very similar to the component tree of the actual implementation. Even though our tool had some difficulties with the structure of the tree, the result would still be beneficial.

6.5. Full example

The full example can be found on the CD enclosed to this thesis.

⁴As mentioned in section 5.5, our model converter does not losslessly convert all Balsamiq files yet. It was tailored in order to convert the wireframes of this case study.

7. Towards responsive User Interfaces

In this chapter we give an overview and a draft for a possible solution when designing responsive user interfaces. We start with an overview of the problems we face when creating and maintaining user interface designs (section 7.1). We then focus on two particular challenges - providing assistance during the design process (section 7.2) and adapting the derivation of the component tree (section 7.3).

7.1. Overview

Most notably, nowadays user interfaces are developed via the popular HTML5 technology stack for even running on multiple types of devices, such as phones, tablets or desktops in parallel. Those interfaces are usually designed separately either mobile-first or desktop-first or a hybrid method. Designing and maintaining those responsive user interfaces challenges most user interface design approaches. Even though our approach and its underlying model (section 7.2) allows specifying more than one wireframe model for a given user interface design, we are faced with difficulties:

How can we assist maintaining multiple wireframe models in one UI design? When dealing with responsive user interfaces, we are confronted with specifying and maintaining a set of wireframe models, each consisting of a set of wireframes. This is significantly more work for a designer, as he has to specify interactions and their related operations and steps for each wireframe model separately. This is especially time consuming when the defined wireframe models of the different devices look and behave similar. A draft for a possible solution is described in section 7.2.

How can we derive one component tree for multiple wireframe models? Furthermore, the derivation of a component tree has to be adapted in order to work well with responsive user interfaces. The current approach offers a very basic solution for this problem, by simply deriving a component tree for each wireframe model (section 4.5.2). This violates our requirement to maximize reusability of components and is therefore not a sufficient solution. Hence, we need to adapt the derivation of the component tree in order to master this problem. In section 7.3 we briefly explain how a possible solution could look like.

7.2. Assisting responsive User Design

One possibility to target this challenge is by enhancing the model with the possibility to interrelate different wireframe models. This could be done before or after the derivation of the component tree, by providing a mechanism to:

- Define a relation between the components derived from different wireframe models
- Define a relation between wireframes and elements of different wireframe models

If done by the designer, especially the second approach would be time consuming and destruct the initial concept of rapid prototyping. Therefore it is crucial to provide a mechanism to support the designer when specifying and maintaining user interfaces

7.3. Adapting the front-end architecture derivation

Despite the fact that different device types usually require different visuals, we want maximum reusability of UI dialogs between different devices. Hence, we have to adapt our methodology to allow combining responsive design with the underlying component-orientation. In section 4.5.2 we proposed an approach to derive a component tree. One of the first steps was building the shared-elements graph by identifying similar elements and grouping them to widget-components. This leads to the following questions:

How can we identify similar elements of different wireframe models? By construction, the similarity definition is mainly based on the size and position of an element relative to its frame (e.g. browser window, phone, etc.). Even though different devices share similar elements in the proper meaning of the word, their relative position and size diverges due to different frame sizes. The naive way to solve this problem is to create a mutually exclusive sub-tree for each device. The parent composite-component of the sub-trees will then detect which device is used and decide which sub-tree will be visible. This approach is not satisfying due to the lack of reusability of similar components across devices. In section 7.3.1 we briefly propose an idea of how a solution to this problem could look like.

How can we handle shared components? Another problem when handling multiple wireframe models within one user interface design is by specifying the components they have in common. Assume we have a have identified similar elements and are able to group them into components as described in section 4.5.2. We are faced with some difficulties: The component is specified by a set: $component = \{ \{e_n, e_m, \dots, e_o\}, \dots, \{e_i, \dots, e_j\} \}$ where $e_n \sim_{elem} e_m, e_m \sim_{elem} e_o, e_n \sim_{elem} e_o$ and so on. Consequently an arbitrary element of each subset can be used as a proxy for the representation of an element in a component. If we adopt the definition of similarity to allow elements with different size and position, the proxy has to be chosen depending on the device. A brief idea how to target this problem can be found in section 7.3.2.

7.3.1. Similarity Definition for Elements and Components

Although we have different wireframe models, they are all linked to the same task and data model. A mechanism to identify similar widgets other than through their position and size could be found by those links:

Imagine having two elements of different wireframe models with two interactions, both interactions performing the same operation and referencing to the same step. This implies that those elements behave equally in the terms of user experience. Assuming that the designer followed basic usability guidelines when sketching the wireframes, we can conclude that those elements can be considered similar. Based on this information we can adapt the \sim_{elem} relation.

7.3.2. Handling shared Components

The problem of finding a proxy element based on the device type could be solved by providing a notation for the proxy in the component tree model. A possible solution could be defined like this:

instead of the widget-component knowing its position and size, the parent composite-component needs to know these information. The composite-component decides, based on the device, how the widget-component will be layouted. In short, the parent composite-component of each shared widget-component holds one reference per device type (wireframe model) to a proxy element.

8. Further Work and Conclusion

In the following sections we will summarize our findings for the wireframe-driven user interface design approach and its benefits for real world projects. In the last section we discuss potential research topics to build on top of the foundation laid in this thesis.

8.1. Conclusion

This thesis proposed an approach for user interface design built on one model that interrelates the concepts of wireframes, user tasks, domain objects and user roles.

We first set out to examine two forms of user interface design approaches - wireframing approaches and model-based approaches - and evaluate their suitability to identify points to improve. As previously stated, what differentiates model-based user interface design from rapidly wireframing the user interface is the possibility to capture all relevant aspects of a user interface in a structured and well-defined manner. These include the aspect that user interface designs need to capture the user tasks, as well as the corresponding roles and domain objects. In order to compare these approaches, we carried out four scenarios of user interface design which cover the process, starting from the initial sketches up to fine grained models. Based on these scenarios we defined a set of requirements for concept and tooling that can be adopted in the industry. From this examination, it has been shown that existing model-based approaches provide great ways to specify all kinds of user interface models, but fall short in effortlessly creating and maintaining those models. Furthermore, the creative process of sketching an initial user interface to gather and discuss requirements is neglected by those approaches. We figured that there is also a very high expectation of the design process itself; it must be suitable for early stages in the development circle, integrate well with common software engineering methods and provide a low barrier of entry.

Thus, we provide an approach that combines the benefits of rapid wireframing and model-based approaches. Our approach is built on a universal model that consists of three tightly connected sub-models: wireframes, user tasks and data objects. We designed the wireframes as the foundation of our model, to emphasize rapidly sketching a UI during early stages of the development. We interrelate these sub-models by introducing interactions and operations. Based on this user interface model, we were able to slice different views and provide storyboards and data flow diagrams for each task. Furthermore, we give an educated guess for a component-oriented frontend architecture.

To validate our approach and support our claims that it is feasible in real-world projects, we implemented Wireframe2Artefact. Its underlying user interface model has been implemented and split into three connected XML-based user interface languages (DXML, TXML, WXML). Wireframe2Artefact allows the user to specify a user interface design. Furthermore, our tool derives storyboards, data flow diagrams and a component tree from a given user interface to enable designers, domain experts and engineers to effectively communicate the details of a design. While still a lot of work has to be done, we find Wireframe2Artefact highly usable for the scenarios we described. Our case study has shown that our underlying model is suitable for common user interface designs of real-world projects.

8.2. Future work

This thesis has thrown up many questions that need for further investigation. One particular question deals with the design of responsive user interfaces and how we can assist the designer to maintain consistency across different devices. In chapter 7 we gave a brief introduction to the problems and how possible solutions could look like.

Moreover, the integration of our tool into the development process would be an interesting topic to asses. In this thesis we made simple recommendations on how designers, developers and domain experts could collaborate using our approach. More information of the design process and its activities would help us to establish a better understanding of the collaboration between different professionals.

In terms of deriving the views, our approach just scratched at the surface. Further research should therefore concentrate on the algorithm and informations needed to slice the different view. A more powerful specification for the concept of operations seems desirable. In this thesis we focused on the CRUD operations on data objects. Allowing the user to specify more advanced operations and introducing concepts of custom methods with parameters and return values is desirable. This information would help as to establish a greater degree of accuracy on the data flow diagrams.

Considerably more work will also need to be done to generate working code for a given user interface model. Even though we provided an educated guess on the component tree of a user interface, the communication flow of the components was not intended to be in the scope of this thesis.

Bibliography

- [AI07] Jesús M. Almendros-Jiménez and Luis Iribarne. “Describing Use-Case Relationships with Sequence Diagrams”. In: *Comput. J.* 50.1 (2007), pp. 116–128. ISSN: 0010-4620. DOI: [10.1093/comjnl/bxl053](https://doi.org/10.1093/comjnl/bxl053). URL: <http://dx.doi.org/10.1093/comjnl/bxl053>.
- [Ang97] Angel R. Puerta. “A Model-Based Interface Development Environment”. In: *IEEE Software* 14,4 (July/August (1997), pp. 41–47.
- [Bra02] Ian Bray. *An Introduction to Requirements Engineering* -. Amsterdam: Pearson Education, 2002. ISBN: 978-0-201-76792-6.
- [Enga] Ralf S. Engelschall. *ComponentJS Features*. URL: <http://componentjs.com/features.html> (visited on 03/02/2014).
- [Engb] Ralf S. Engelschall. *User Interface Component Tree Architecture Pattern*. URL: <http://engelschall.com/go/EnTR-04:2013.12>.
- [Engc] Ralf S. Engelschall. *User Interface Composition*. URL: <http://engelschall.com/go/EnTR-03:2013.12>.
- [Fow04] Martin Fowler. *UML distilled: A brief guide to the standard object modeling language*. 3rd ed. The Addison-Wesley object technology series. Boston: Addison-Wesley, 2004. ISBN: 9780321193681.
- [Gro04] Jonathan L. Gross. *Handbook of graph theory*. Discrete mathematics and its applications. Boca Raton and FL: CRC Press, 2004. ISBN: 1584880902.
- [HJD10] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering* -. 3. Aufl. Berlin, Heidelberg: Springer, 2010. ISBN: 978-1-849-96405-0.
- [Jak09] Jakub Linowski. *Interactive Sketching Notation*. Ed. by Wireframe Magazine. 2009. URL: <http://wireframes.linowski.ca/2009/10/interactive-sketching-notation-v0-1/>.
- [Lim+05] Quentin Limbourg et al. “USIXML: A Language Supporting Multi-path Development of User Interfaces”. In: *Engineering Human Computer Interaction and Interactive Systems*. Ed. by Rémi Bastide, Philippe Palanque, and Jörg Roth. Vol. 3425. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 200–220. ISBN: 978-3-540-26097-4. DOI: [10.1007/11431879_12](https://doi.org/10.1007/11431879_12). URL: http://dx.doi.org/10.1007/11431879_12.
- [Lin99] James Lin. “A Visual Language for a Sketch-based UI Prototyping Tool”. In: *CHI '99 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '99. Pittsburgh, Pennsylvania: ACM, 1999, pp. 298–299. ISBN: 1-58113-158-5. DOI: [10.1145/632716.632899](https://doi.org/10.1145/632716.632899). URL: <http://doi.acm.org/10.1145/632716.632899>.
- [LM95] James A. Landay and Brad A. Myers. “Interactive Sketching for the Early Stages of User Interface Design”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '95. New York, NY, and USA: ACM Press/Addison-Wesley Publishing Co, 1995, pp. 43–50. ISBN: 0-201-84705-1. DOI: [10.1145/223904.223910](https://doi.org/10.1145/223904.223910). URL: <http://dx.doi.org/10.1145/223904.223910>.

- [Nie94] Jakob Nielsen. “Usability Inspection Methods”. In: *Conference Companion on Human Factors in Computing Systems*. CHI '94. Boston, Massachusetts, USA: ACM, 1994, pp. 413–414. ISBN: 0-89791-651-4. DOI: [10.1145/259963.260531](https://doi.org/10.1145/259963.260531). URL: <http://doi.acm.org/10.1145/259963.260531>.
- [PE02] Angel Puerta and Jacob Eisenstein. “XIML: A Common Representation for Interaction Data”. In: *Proceedings of the 7th International Conference on Intelligent User Interfaces*. IUI '02. New York, NY, and USA: ACM, 2002, pp. 214–215. ISBN: 1-58113-459-2. DOI: [10.1145/502716.502763](https://doi.org/10.1145/502716.502763). URL: <http://doi.acm.org/10.1145/502716.502763>.
- [PhD13] Tobias PhD. Komischke. *Activity Theory & Hierarchical Task Analysis: The Power Couple for Effective UX Analysis*. 2013. URL: <http://d3.infragistics.com/wp-content/uploads/2013/04/Activity-Theory-and-Hierarchical-Task-Analysis.pdf>.
- [PMM05] Angel Puerta, Michael Micheletti, and Alan Mak. “The UI Pilot: A Model-based Tool to Guide Early Interface Design”. In: *Proceedings of the 10th International Conference on Intelligent User Interfaces*. IUI '05. New York, NY, and USA: ACM, 2005, pp. 215–222. ISBN: 1-58113-894-6. DOI: [10.1145/1040830.1040877](https://doi.org/10.1145/1040830.1040877). URL: <http://doi.acm.org/10.1145/1040830.1040877>.
- [Pri90] Rubén Prieto-Díaz. “Domain Analysis: An Introduction”. In: *SIGSOFT Softw. Eng. Notes* 15.2 (Apr. 1990), pp. 47–54. ISSN: 0163-5948. DOI: [10.1145/382296.382703](https://doi.org/10.1145/382296.382703). URL: <http://doi.acm.org/10.1145/382296.382703>.
- [RF13] Michael Richter and Markus D. Flückiger. *Usability Engineering kompakt: Benutzbare Produkte gezielt entwickeln*. 3. Aufl. IT kompakt. Berlin: Springer Vieweg, 2013. ISBN: 9783642348310.
- [SB10] Florian Sarodnick and Henning Brau. *Methoden der Usability Evaluation: Wissenschaftliche Grundlagen und praktische Anwendung*. 2., überarb. u. aktualis. Aufl. Wirtschaftspsychologie in Anwendung. Bern: Verlag Hans Huber, 2010. ISBN: 9783456848839.
- [Vaa] Christian Vaas. *Pattern Guideline and Constraint Validation of Runtime Communication in User Interface Component Architectures*. URL: <http://architecture-dissertation.com/publication/MasterThesis-Vaas.pdf> (visited on 03/02/2014).

A. List of Wireframe Element Types

The following list shows all element types of our user interface model. Further details on each element type can be found in the schemas in appendix D or in the implementation of the schemas in the source code (see enclosed CD).

Accordion	Group	Radio Button
Alert	Icon	Radio Button Group
Annotation	Label	Row Element
Arrow	Link Bar	Rule
Image	Link	Scroll Bar
Bread Crumbs	List	Search Field
Button	List Item	Slider
Callout	Media	Splitter
Chart	Menu Bar	Sticky Note
Check Box	Menu	Tabbar
Check Box Group	Menu Item	Tab
Collapsible Panel	Modal Overlay	Table
Color Picker	Numeric Stepper	Text Area
Combo Box	Panel	Tooltip
Cover Flow	Paragraph	Tree
Curly Brace	Phone Keyboard	Lines
Date Chooser	Phone Menu	Window
Frame	Phone Picker	
Geometric	Progress Bar	

B. Wireframe2Artefact Screenshots

The following figures show screenshots of each tab of Wireframe2Artefact. In all screenshots the "my to-do app" is used as an example input.

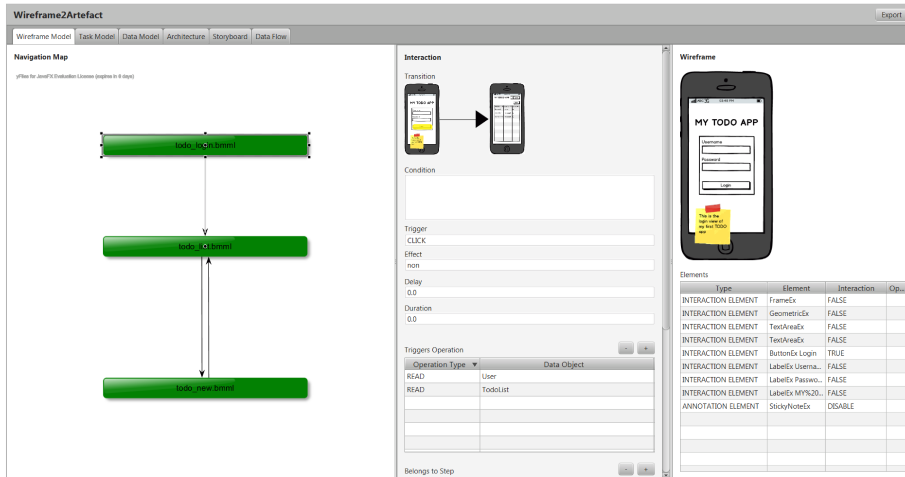


Figure B.1.: Wireframe Tab

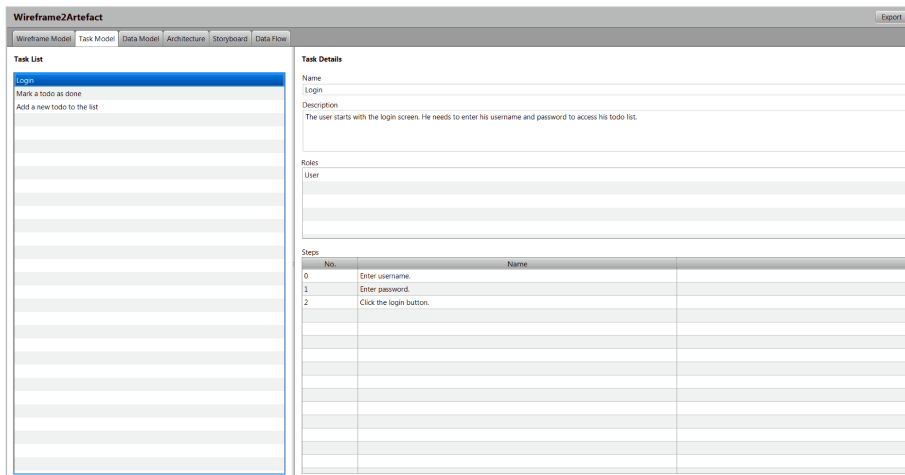


Figure B.2.: Task Tab

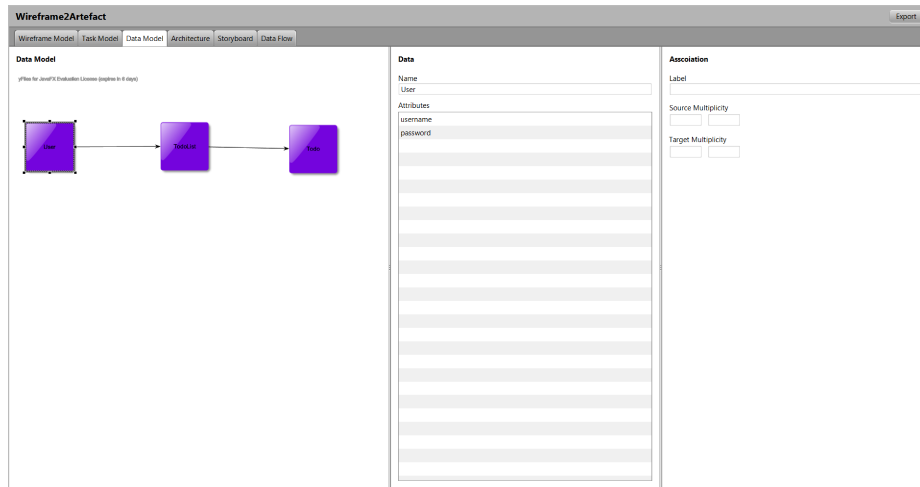


Figure B.3.: Data Tab

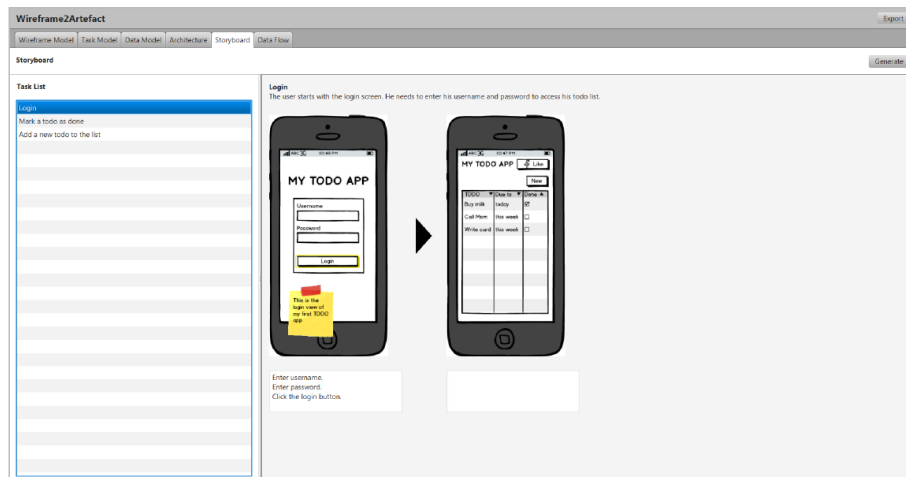


Figure B.4.: Storyboard Tab

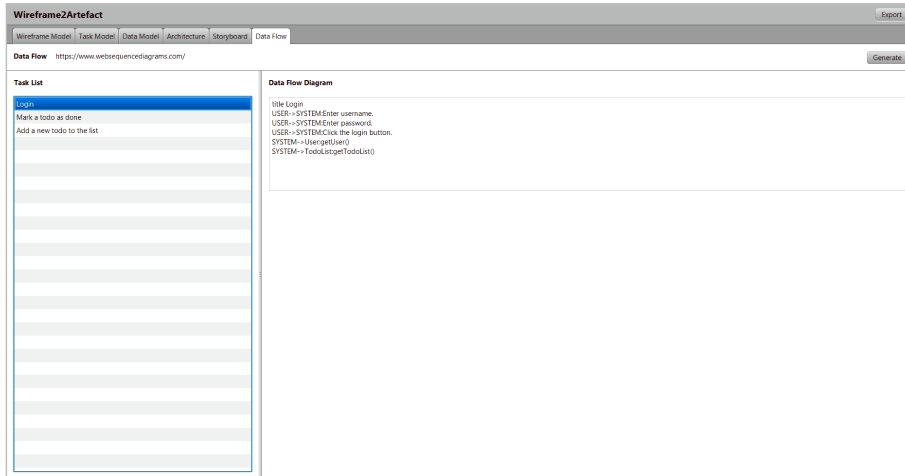


Figure B.5.: Data Flow Tab

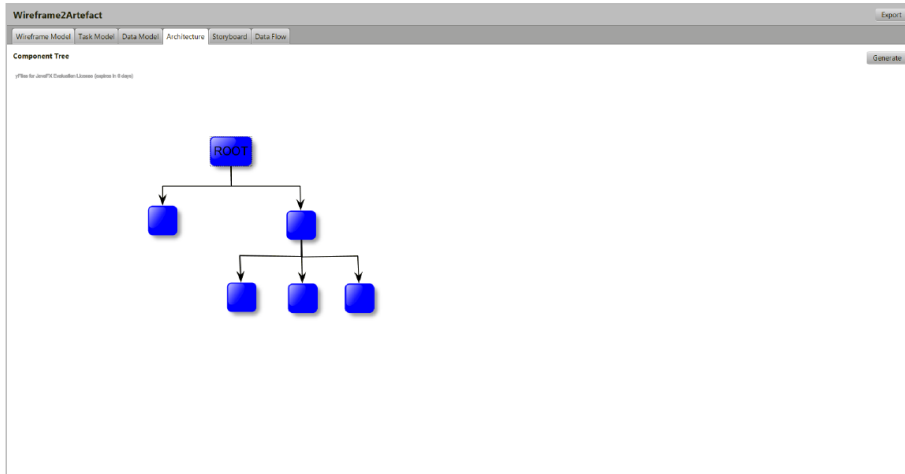


Figure B.6.: Component Tree Tab

C. P1-Timesheet

In chapter 6 we introduced the P1-Timesheet app developed at msg systems ag. Furthermore, we specified its UI with Wireframe2Artefact and derived storyboards and data flow diagrams for each task: login, create entry and copy entry. The following section shows a screenshot for each storyboard derived by our tool. The next sections shows the derived data flow diagrams.

C.1. Storyboards

The following figures show a screenshot of the storyboard derived by Wireframe2Artefact.

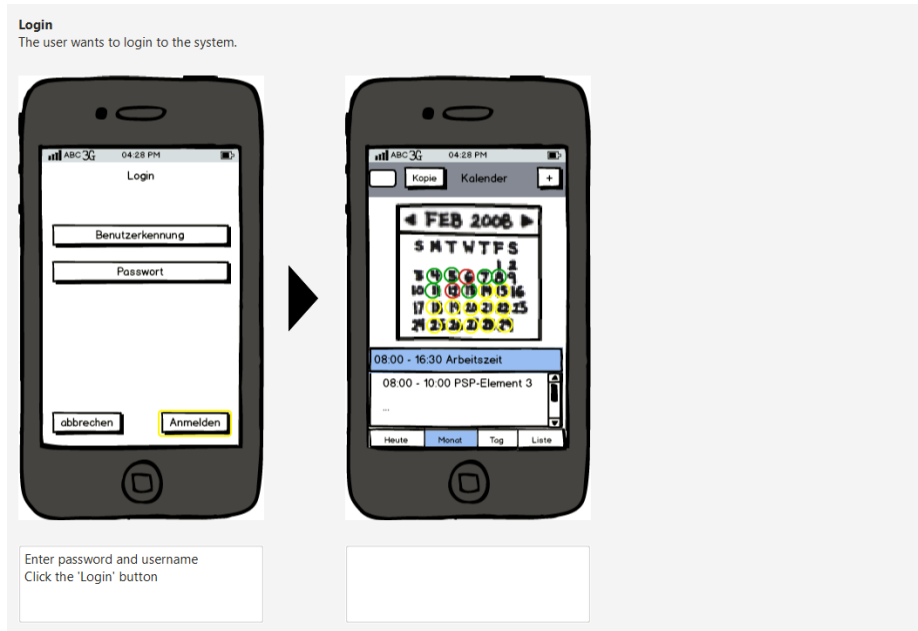


Figure C.1.: Login Storyboard

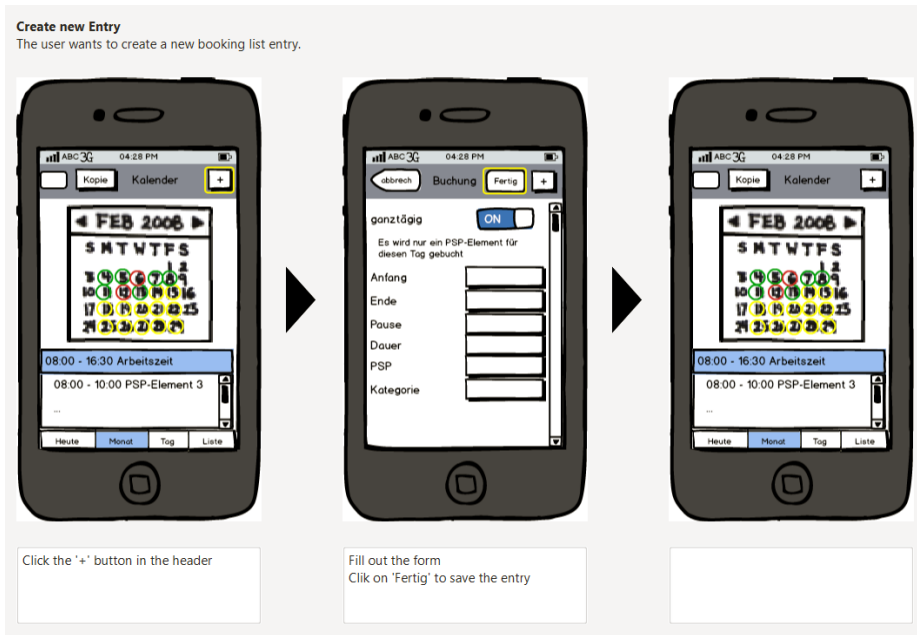


Figure C.2.: Create Entry Storyboard

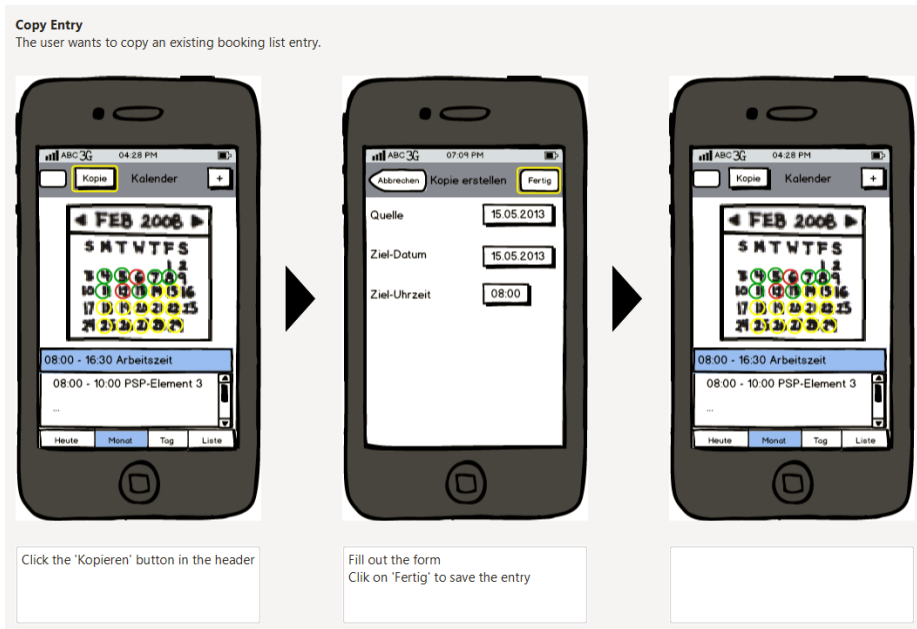


Figure C.3.: Copy Entry Storyboard

C.2. Data Flow Diagrams

This section shows the data flow diagrams of the P1-Timesheet app as derived by our tool. The left hand side, shows the textual output of our tool. The right hand side shows the corresponding diagram, which was generated by a web application called "web sequence diagrams"¹.

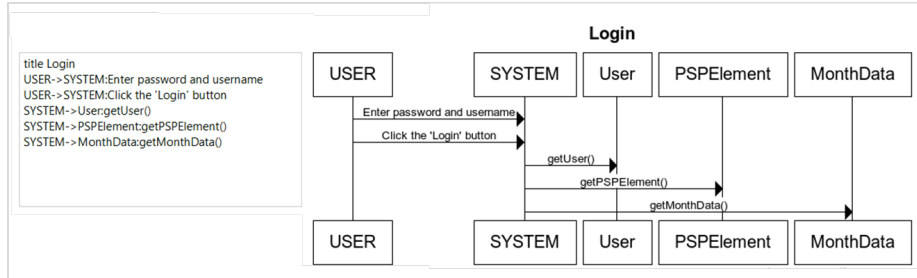


Figure C.4.: Login Data Flow Diagram

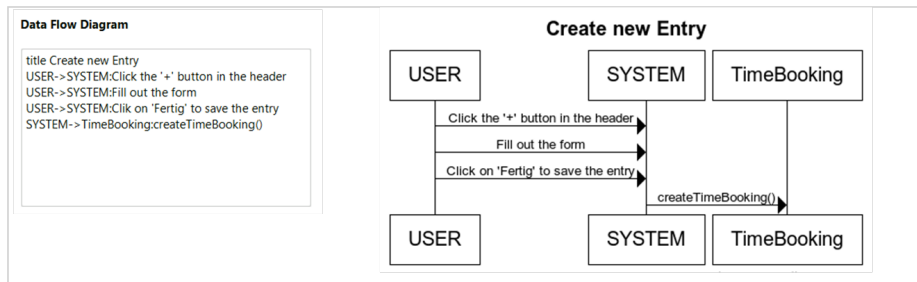


Figure C.5.: Create Entry Data Flow Diagram

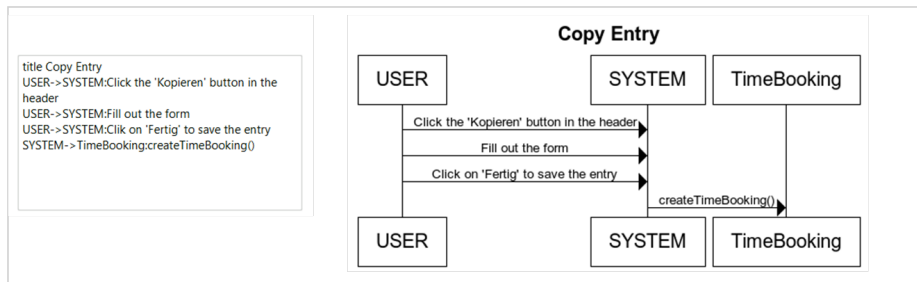


Figure C.6.: Copy Entry Data Flow Diagram

¹<https://www.websequencediagrams.com/>

D. RelaxNG Schemas

As mentioned in section 5.2, Wireframe2Artefacts underlying framework is based on three XML-based meta-level modelling languages (TXML, DXML, WXML). The corresponding RelaxNG schemas for each language can be found in the following sections.

D.1. RelaxNG Schema of the Balsamiq Model

The following schema describes the BMML file format. We created this specification based on the poorly documented BMML file format specification on the Balsamiq Webpage¹.

```
1 grammar {
2   start = element mockup {mockup-elem}
3
4   mockup-elem =
5     attribute version {xsd:string},
6     attribute skin {"sketch" | "wireframe"},
7     attribute fontFace {"Balsamiq Sans" | "Sans"},
8     attribute measuredW {xsd:integer},
9     attribute measuredH {xsd:integer},
10    attribute mockupW {xsd:integer},
11    attribute mockupH {xsd:integer},
12    element controls {controls-elem} ?
13
14   controls-elem =
15     element control {control-elem} *
16
17   control-elem =
18     attribute controlID {xsd:nonNegativeInteger},
19     attribute controlId {controlType-attr},
20     attribute x {xsd:integer},
21     attribute y {xsd:integer},
22     attribute w {xsd:integer},
23     attribute h {xsd:integer},
24     attribute measuredW {xsd:integer},
25     attribute measuredH {xsd:integer},
26     attribute zOrder {xsd:nonNegativeInteger},
27     attribute locked {xsd:boolean},
28     attribute isInGroup {isInGroup-attr},
29     element controlProperties {controlProperty-elem}?,
30     element groupChildrenDescriptors {groupChildrenDescriptors-elem} ?
31
32   groupChildrenDescriptors-elem =
33     element control {control-elem} +
34
35   controlProperty-elem =
36     element alternateRowColor {xsd:integer}?,
37     element model {"iPhone4" | "iPhone5"}?,
38     element rowHeight {xsd:integer}?,
39     element hasHeader {xsd:boolean}?,
```

¹<http://support.balsamiq.com/customer/portal/articles/111834>

```

40     element vLines {xsd:boolean}?,
41     element hLines {xsd:boolean}?,
42     element backgroundAlpha {xsd:float {pattern = "(0\.[0-9][0-9]*)|1"}}?,
43     element bgPattern {"allWhite" | "allBlack" | "topOnly" | "topBlueBottom" | "
      topBlackBottom"}?,
44     element bgTransparent {xsd:boolean}?,
45     element topBar {xsd:boolean} ?,
46     element align {"right" | "left" | "center"} ?,
47     element bold {xsd:boolean} ?,
48     element italic {xsd:boolean} ?,
49     element underline {xsd:boolean} ?,
50     element size {xsd:integer} ?,
51     element borderStyle {"none" | "square" | "squareBreakline" | "roundedSolid" | "
      roundedDotted"} ?,
52     element borderColor {xsd:integer} ?,
53     element close {xsd:boolean} ?,
54     element minimize {xsd:boolean} ?,
55     element maximizeRestore {xsd:boolean} ?,
56     element dragger {xsd:boolean} ?,
57     element topheight {xsd:integer} ?,
58     element bottomheight {xsd:integer} ?,
59     element color {xsd:integer} ?,
60     element crop {crop-content} ?,
61     element customData {xsd:string} ?,
62     element customID {xsd:string} ?,
63     element filter {xsd:boolean} ?,
64     element direction {"left" | "center" | "right" | "bottom" | "top"} ?,
65     element href {href-content} ?,
66     element hrefs {hrefs-content} ?,
67     element icon {icon-content} ?,
68     element indeterminate {xsd:boolean} ?,
69     element scrollbarValue {xsd:integer {pattern = "[0-9][1-9][0-9]|100"}} ?,
70     element verticalScrollbar {xsd:boolean} ?,
71     element horizontalScrollbar {xsd:boolean} ?,
72     element labelPosition {"bottom" | "right"} ?,
73     element leftArrow {xsd:boolean} ?,
74     element rightArrow {xsd:boolean} ?,
75     element curvature {xsd:integer {pattern = "-1|0|1"}}?,
76     element map {xsd:string} ?,
77     element onOffState {"on" | "off"} ?,
78     element orientation {"portrait" | "landscape"} ?,
79     element position {"left" | "right"} ?,
80     element selectedIndex {xsd:integer} ?,
81     element src {xsd:string} ?,
82     element state {"up" | "selected" | "focused" | "disabled" | "disabledSelected" |
      "indeterminate" | "disabledIndeterminate"} ?,
83     element tabHPosition {"left" | "right"}?,
84     element tabVPosition {"top" | "bottom"}?,
85     element tooltipDirection {"NW" | "N" | "NE" | "E" | "SE" | "S" | "SW" | "W"} ?,
86     element value {xsd:integer {pattern = "[0-9][1-9][0-9]|100"}} ?,
87     element shape {"circle" | "rectangle" | "roundRect" | "dottedRect" | "diamond" | "
      star" | "triangle" | "parallelogram" | "rectangle%20breakline%20bottom%20" | "
      rectangle%20breakline%20top"} ?,
88     element shapeRotation {"0" | "90" | "180" | "270"} ?,
89     element override {override-elem} *,
90     element text {xsd:string} ?,
91     element stroke {"solid" | "dashed" | "dotted" | "breakBottom" | "breakTop" | "
      break"} ?,
92     element menuIcon {xsd:boolean} ?,
93     element textOrientation {xsd:string} ?
94

```



```

1  grammar {
2      start = element dataModel {DataModel}
3
4      DataModel =
5          element data {Data}*,
6          element association {Association}*
7
8      Association =
9          element source {xsd:string},
10         element target {xsd:string},
11         element label {xsd:string}?,
12         element sourceMultiplicity {Multiplicity}?,
13         element targetMultiplicity {Multiplicity}?
14
15     Data =
16         element id {xsd:string},
17         element name {xsd:string},
18         element attribute {Attribute}*
19
20     Attribute =
21         attribute name {xsd:string}
22
23     Multiplicity =
24         list {xsd:nonNegativeInteger, xsd:nonNegativeInteger}
25 }

```

Listing D.3: "RelaxNG Schema for the Wireframe Data Model"

D.4. RelaxNG Schema of the Wireframe Model

To improve understandability, we split the schema documents into nine sub-schemas. The following sections show these sub-schemas.

```

1  grammar {
2      include "wireframe-annotation-elems.rnc"
3      include "wireframe-attr.rnc"
4      include "wireframe-container-elems.rnc"
5      include "wireframe-input-elems.rnc"
6      include "wireframe-layout-elems.rnc"
7      include "wireframe-media-elems.rnc"
8      include "wireframe-mobile-elems.rnc"
9      include "wireframe-text-elems.rnc"
10
11     start = element WireframeModel {Wireframe*}
12
13     Wireframe = element wireframe {
14         attribute id {xsd:string},
15         attribute width {xsd:int},
16         attribute height {xsd:int},
17         attribute filename {xsd:string},
18         ( accordion-elem |
19           annotation-elem |
20           arrow-elem |
21           chart-elem |
22           breadCrumbs-elem |
23           callout-elem |
24           geometric-elem |
25           collapsiblePanel-elem |

```

```

26         coverFlow-elem |
27         label-elem |
28         link-elem |
29         linkBar-elem |
30         paragraph-elem |
31         tooltip-elem |
32         phonePicker-elem |
33         phoneKeyboard-elem |
34         phoneMenu-elem |
35         button-elem |
36         checkBox-elem |
37         checkBoxGroup-elem |
38         colorPicker-elem |
39         comboBox-elem |
40         slider-elem |
41         list-elem |
42         menu-elem |
43         menuBar-elem |
44         radioButton-elem |
45         radioButtonGroup-elem |
46         searchField-elem |
47         dateChooser-elem |
48         textArea-elem |
49         tree-elem |
50         alert-elem |
51         numericStepper-elem |
52         table-elem |
53         scrollbar-elem |
54         splitter-elem |
55         modalOverlay-elem |
56         panel-elem |
57         window-elem |
58         tabBar-elem |
59         image-elem |
60         icon-elem |
61         progressBar-elem |
62         media-elem |
63         curlyBrace-elem |
64         stickyNote-elem |
65         frame-elem |
66         group-elem |
67         rule-elem
68     )*
69 }
70
71
72 group-elem = element group {
73     BasicInteractionElement,
74     attribute groupName {xsd:string}?,
75     (
76         accordion-elem |
77         annotation-elem |
78         arrow-elem |
79         chart-elem |
80         breadCrumbs-elem |
81         callout-elem |
82         geometric-elem |
83         collapsiblePanel-elem |
84         coverFlow-elem |
85         label-elem |
86         link-elem |

```



```
87     paragraph-elem |
88     tooltip-elem |
89     phonePicker-elem |
90     phoneKeyboard-elem |
91     phoneMenu-elem |
92     button-elem |
93     checkBox-elem |
94     checkBoxGroup-elem |
95     colorPicker-elem |
96     comboBox-elem |
97     slider-elem |
98     list-elem |
99     menu-elem |
100    menuBar-elem |
101    radioButton-elem |
102    radioButtonGroup-elem |
103    searchField-elem |
104    dateChooser-elem |
105    textArea-elem |
106    tree-elem |
107    alert-elem |
108    numericStepper-elem |
109    table-elem |
110    scrollbar-elem |
111    splitter-elem |
112    modalOverlay-elem |
113    panel-elem |
114    window-elem |
115    tabBar-elem |
116    image-elem |
117    icon-elem |
118    progressBar-elem |
119    media-elem |
120    curlyBrace-elem |
121    stickyNote-elem |
122    frame-elem |
123    group-elem |
124    rule-elem
125  )*
126 }
127
128 CustomProperty =
129     attribute id {xsd:string},
130     xsd:string
131
132
133 Interaction =
134     attribute trigger {trigger-attr},
135     attribute duration {xsd:float}?,
136     attribute delay {xsd:float}?,
137     attribute effect {effect-attr}?,
138     element transition {transition-elem},
139     element step {Step}*,
140     element operation {Operation}*
141
142 transition-elem =
143     element condition {xsd:string}?,
144     element target {xsd:string}
145
146 Step =
147     xsd:string
```

```
148
149 Operation =
150     attribute type {operationType-attr},
151     xsd:string
152
153
154 border-elem =
155     attribute width {xsd:int},
156     attribute stroke {stroke-attr},
157     attribute radius {xsd:int},
158     attribute color {color-attr},
159     attribute opacity {opacity-attr},
160     empty
161
162
163 style-elem =
164     attribute backgroundColor {color-attr}?,
165     attribute opacity {opacity-attr},
166     attribute margin {xsd:int}?,
167     attribute padding {xsd:int}?,
168     element backgroundImage {backgroundImage-elem} ?,
169     element border {border-elem}?
170
171
172 backgroundImage-elem =
173     attribute backgroundRepeat {"repeat-x" | "repeat-y" | "no-repeat" | "repeat"},
174     xsd:string
175
176 mobileDvice-elem =
177     attribute mobileDeviceType {mobileDeviceType-attr},
178     attribute model {model-attr} ?,
179     attribute topBar {xsd:boolean} ?,
180     attribute backgroundPattern {backgroundPattern-attr}?,
181     empty
182
183
184 otherDevice-elem =
185     attribute deviceType {deviceType-attr},
186     empty
187
188 WireframeElement =
189     attribute xPos {xsd:int}?,
190     attribute yPos {xsd:int}?,
191     attribute width {xsd:int}?,
192     attribute height {xsd:int}?,
193     attribute zOrder {xsd:int}?,
194     element style {style-elem}?,
195     element customProperty {CustomProperty}*
196
197 InteractionElement =
198     WireframeElement,
199     element interaction {Interaction}?,
200     attribute state {state-attr}?
201
202 NonInteractionElement =
203     WireframeElement
204
205
206 BasicInteractionElement =
207     InteractionElement,
208     element operation {Operation}*

```

```

209
210
211     ContainerInteractionElement =
212         InteractionElement,
213         element operation {Operation}*,
214         attribute xPosInner {xsd:int}?,
215         attribute yPosInner {xsd:int}?,
216         attribute widthInner {xsd:int}?,
217         attribute heightInner {xsd:int}?,
218         attribute hScrollInner {scroll-attr}?,
219         attribute vScrollInner {scroll-attr}?
220     }

```

Listing D.4: "RelaxNG Schema for the Wireframe Model"

D.4.1. RelaxNG Schema of the Wireframe Attributes

```

1     color-attr = xsd:string
2
3     operationType-attr = "create" | "read" | "update" | "delete"
4
5     opacity-attr = xsd:float {pattern = "(0.[0-9][0.9]|1.00)"}
6
7     curvation-attr = "convex" | "straight" | "concarve"
8
9     scroll-attr = false-attr | scrollValue-attr
10
11     scrollValue-attr = xsd:int {pattern="[0-9]|[1-9][0-9]|100"}
12
13     state-attr = "default" | "disabled" | "focused"
14
15     selectedState-attr = "selected" | "indeterminate" | "default"
16
17     stroke-attr = "solid" | "dashed" | "dotted" | "breaklineBottom" | "breaklineTop" | "
18         break"
19
20     align-attr = "tl" | "tr" | "bl" | "br" | "center" | "middle"
21
22     orientation-attr = "horizontal" | "vertical"
23
24     labelOrientation-attr = "topBottom" | "bottomTop" | "default" | "slanted"
25
26     direction-attr = "left" | "right" | "top" | "down"
27
28     tooltipDirection-attr = "NW" | "N" | "NE" | "E" | "SE" | "S" | "SW" | "W"
29
30     iconSize-attr = xsd:int
31
32     rotation-attr = xsd:int {pattern="
33         [\-|+]?([0-9]|[1-9][1-9]|1[0-9][0-9]|2[0-9][0-9]|3[0-5][0-9]|360)"}
34
35     mobileDeviceType-attr = "iPhone" | "iPad" | "android" | "windowsPhone" | text
36
37     deviceType-attr = "browser" | "other" | text
38
39     model-attr = "4" | "5" | text
40
41     backgroundPattern-attr = "allWhite" | "allBlack" | "topBlueBottom" | "topBlackBottom"
42         | "topOnly" | text

```

```

40
41 deviceOrientation-attr = "portrait" | "landscape"
42
43 trigger-attr = "touch" | "doubleTouch" | "drag" | "swipeLeft" | "swipeRight" | "
    swipeUp" | "swipeDown" | "pinchOpen" | "pinchClose" | "click" | "doubleClick" | "
    rightClick" | "hover"
44
45 false-attr = xsd:boolean - ("true" | "1")
46
47 true-attr = xsd:boolean - ("false" | "0")
48
49 effect-attr = xsd:string
50
51 shape-attr = "circle" | "rectangle" | "roundRectangle" | "diamond" | "star" | "
    triangle" | "parallelogram"
52
53 annotationType-attr = "crossOut" | "scratchOut" | "redX" | "tagCloud"
54
55 mediaType-attr = "map" | "videoPlayer" | "playbackControls" | "volumeSlider" | "
    webcam" | "formattingToolbar" | "hotspot" | "calendar"

```

Listing D.5: "RelaxNG Schema for the Wireframe Attributes"

D.4.2. RelaxNG Schema of the Wireframe Container Elements

```

1
2 accordion-elem = element accordion{
3     ContainerInteractionElement,
4     element accordionItems {accordionItem-elem +}
5 }
6
7 accordionItem-elem = element accordionItem {
8     BasicInteractionElement,
9     attribute selected {xsd:boolean},
10    label-elem?
11 }
12
13
14 panel-elem = element panel {
15     ContainerInteractionElement,
16     attribute name {xsd:string}
17 }
18
19 window-elem = element window{
20     ContainerInteractionElement,
21     attribute topHeight {xsd:int} ?,
22     attribute bottomHeight {xsd:int} ?,
23     attribute closeable {xsd:boolean},
24     attribute minimizeable {xsd:boolean},
25     attribute maximizeable {xsd:boolean},
26     attribute resizeable {xsd:boolean},
27     element title {text-elem}?
28 }
29
30 tabbar-elem = element tabbar{
31     ContainerInteractionElement,
32     attribute tabPosition {direction-attr},
33     element tab {tab-elem}
34 }

```

```

35
36     tab-elem =
37         BasicInteractionElement,
38         attribute selected {xsd:boolean},
39         label-elem?
40
41     collapsiblePanel-elem = element collapsiblePanel {
42         ContainerInteractionElement,
43         attribute trigger {trigger-attr},
44         collapsiblePanelHead-elem
45     }
46
47     collapsiblePanelHead-elem = element head{
48         BasicInteractionElement,
49         attribute isOpen {xsd:boolean},
50         label-elem?
51     }
52
53
54     frame-elem = element frame {
55         ContainerInteractionElement,
56         attribute orientation {deviceOrientation-attr},
57         attribute frameType {mobileDvice-elem | otherDevice-elem}?
58     }

```

Listing D.6: "RelaxNG Schema for the Wireframe Container Elements"

D.4.3. RelaxNG Schema of the Wireframe Input Elements

```

1     button-elem = element button {
2         BasicInteractionElement,
3         attribute pointy {direction-attr}?,
4         attribute switch {"on" | "off"}?,
5         attribute selectedState {selectedState-attr},
6         menu-elem?,
7         label-elem?
8     }
9
10
11     menu-elem = element menu {
12         BasicInteractionElement,
13         attribute menuState {"open" | "closed"},
14         ( element splitter {empty} |
15           element menuItem {menuItem-elem}
16         )+
17     }
18
19     menuItem-elem =
20         BasicInteractionElement,
21         attribute selected {xsd:boolean},
22         ( label-elem |
23           checkBox-elem |
24           radioButton-elem |
25           button-elem |
26           panel-elem
27         )+
28
29     checkBox-elem = element checkBox {
30         BasicInteractionElement,

```

```
31     attribute selectedState {selectedState-attr},
32     label-elem?
33 }
34
35 checkBoxGroup-elem = element checkBoxGroup{
36     BasicInteractionElement,
37     checkBox-elem+
38 }
39
40
41 colorPicker-elem = element colorPicker{
42     BasicInteractionElement,
43     attribute pickedColor {color-attr} ?
44 }
45
46
47 comboBox-elem = element comboBox{
48     BasicInteractionElement,
49     attribute comboBoxState {"open" | "closed"},
50     label-elem+
51 }
52
53
54 slider-elem = element slider{
55     BasicInteractionElement,
56     attribute value {scrollValue-attr},
57     attribute orientation {orientation-attr}
58 }
59
60
61 list-elem = element list{
62     BasicInteractionElement,
63     attribute evenColor {color-attr},
64     attribute oddColor {color-attr},
65     attribute hasHeader {xsd:boolean},
66     attribute rowHeight {xsd:int},
67     attribute vScroll {scroll-attr} ?,
68     attribute hScroll {scroll-attr} ?,
69     element listItem {listItem-elem}?
70 }
71
72
73 listItem-elem =
74     attribute selected {xsd:boolean},
75     label-elem*,
76     checkBox-elem*,
77     radioButton-elem*,
78     button-elem*,
79     panel-elem*
80
81 menuBar-elem = element menuBar{
82     BasicInteractionElement,
83     attribute orientation {orientation-attr},
84     button-elem+
85 }
86
87 radioButton-elem = element radioButton {
88     BasicInteractionElement,
89     attribute selectedState {selectedState-attr},
90     label-elem
91 }
```

```
92
93
94 radioButtonGroup-elem = element radioButtonGroup {
95     BasicInteractionElement,
96     radioButton-elem+
97 }
98
99
100 searchField-elem = element searchField {
101     BasicInteractionElement,
102     label-elem ?,
103     element text {text-elem}?
104 }
105
106 textArea-elem = element textArea{
107     BasicInteractionElement,
108     attribute hScroll {scroll-attr}?,
109     attribute vScroll {scroll-attr}?,
110     label-elem?,
111     element text {text-elem}?
112 }
113
114
115 dateChooser-elem = element dateChooser{
116     BasicInteractionElement,
117     element pickedDate {xsd:string} ?
118 }
119
120 tree-elem = element tree{
121     BasicInteractionElement,
122     element root {treeItem-elem}
123 }
124
125
126 treeItem-elem =
127     BasicInteractionElement,
128     attribute selectedState {selectedState-attr},
129     attribute treeItemState {"open" | "closed"},
130     ( element treeItem {treeItem-elem} |
131         label-elem |
132         checkBox-elem |
133         radioButton-elem |
134         button-elem |
135         panel-elem
136     )+
137
138
139 alert-elem = element alert {
140     BasicInteractionElement,
141     attribute isModal {xsd:boolean}?,
142     element headline {text-elem}?,
143     element message {text-elem}?,
144     button-elem+
145 }
146
147 numericStepper-elem = element numericStepper {
148     BasicInteractionElement,
149     attribute pickedValue {xsd:int} ?,
150     element stepperItem {text-elem}+
151 }
152
```

```

153
154 table-elem = element table{
155     BasicInteractionElement,
156     attribute evenColor {color-attr},
157     attribute oddColor {color-attr},
158     attribute hasHeader {xsd:boolean},
159     attribute rowHeight {xsd:int},
160     element verticalLines {tableLine-elem} ?,
161     element horizontalLines {tableLine-elem} ?,
162     element hScroll {scroll-attr} ?,
163     element vScroll {scroll-attr} ?,
164     element row {row-elem}+
165 }
166
167 tableLine-elem =
168     attribute width {xsd:int},
169     attribute stroke {stroke-attr},
170     attribute color {color-attr},
171     attribute opacity {opacity-attr},
172     empty
173
174 row-elem =
175     attribute selected {xsd:boolean},
176     element cell {cell-elem}*
177
178 cell-elem =
179     ( label-elem |
180       checkBox-elem |
181       radioButton-elem |
182       button-elem |
183       panel-elem
184     )+

```

Listing D.7: "RelaxNG Schema for the Wireframe Input Elements"

D.4.4. RelaxNG Schema of the Wireframe Layout Elements

```

1
2 geometric-elem = element geometric {
3     BasicInteractionElement,
4     attribute shape {shape-attr},
5     element text {text-elem}?
6 }
7
8 rule-elem = element rule{
9     BasicInteractionElement,
10    attribute stroke {stroke-attr},
11    attribute orientation {orientation-attr}
12 }
13
14
15 scrollbar-elem = element scrollbar{
16     BasicInteractionElement,
17     attribute value {scrollValue-attr},
18     attribute orientation {orientation-attr}
19 }
20
21
22

```



```
23 splitter-elem = element splitter{
24     BasicInteractionElement,
25     attribute orientation {orientation-attr}
26 }
27
28 modalOverlay-elem = element modalOverlay {
29     BasicInteractionElement,
30     attribute color {xsd:string}
31 }
```

Listing D.8: "RelaxNG Schema for the Wireframe Layout Elements"

D.4.5. RelaxNG Schema of the Wireframe Media Elements

```
1     chart-elem = element chart {
2         BasicInteractionElement,
3         attribute chartType {"barChart" | "columnChart" | "lineChart" | "pieChart" }
4     }
5
6     coverFlow-elem = element coverFlow {
7         BasicInteractionElement,
8         attribute hScroll {scroll-attr}?,
9         element selectedImage {xsd:int}?,
10        element coverFlowImage {coverFlowImage-elem}+
11    }
12
13    coverFlowImage-elem =
14        element title {text-elem} ?,
15        element description {text-elem} ?,
16        element imagePath {xsd:string}
17
18
19    image-elem = element image {
20        BasicInteractionElement,
21        element imagePath {xsd:string}
22    }
23
24
25    icon-elem = element icon {
26        BasicInteractionElement,
27        attribute size {iconSize-attr},
28        attribute rotation {rotation-attr},
29        element imagePath {xsd:string}
30    }
31
32
33    progressBar-elem = element progressBar{
34        BasicInteractionElement,
35        attribute progressBarStyle {"standard" | "indeterminate"}
36    }
37
38
39    media-elem = element media{
40        BasicInteractionElement,
41        attribute mediaType {mediaType-attr}
42    }
```

Listing D.9: "RelaxNG Schema for the Wireframe Media Elements"

D.4.6. RelaxNG Schema of the Wireframe Mobile Elements

```

1  phonePicker-elem = element phonePicker {
2      BasicInteractionElement,
3      attribute orientation {deviceOrientation-attr},
4      element column {phonePickerColumn-elem}+
5  }
6
7  phonePickerColumn-elem =
8      BasicInteractionElement,
9      attribute selected {xsd:boolean}?,
10     element columnItem {label-elem}+
11
12  phoneKeyboard-elem = element phoneKeyboard {
13      BasicInteractionElement,
14      attribute orientation {deviceOrientation-attr}
15  }
16
17  phoneMenu-elem = element phoneMenu {
18      BasicInteractionElement,
19      attribute orientation {deviceOrientation-attr},
20      element phoneMenuItem {phoneMenuItem-elem}+
21  }
22
23  phoneMenuItem-elem =
24      element draggable {xsd:boolean},
25      element deletable {xsd:boolean},
26      element submenu {xsd:boolean},
27      element selected {xsd:boolean} ?,
28      ( label-elem |
29          button-elem |
30          checkBox-elem |
31          radioButton-elem |
32          panel-elem
33      ) +

```

Listing D.10: "RelaxNG Schema for the Wireframe Mobile Elements"

D.4.7. RelaxNG Schema of the Wireframe Text Elements

```

1  breadCrumbs-elem = element breadCrumbs {
2      BasicInteractionElement,
3      element breadcrumbItem {text-elem}+
4  }
5
6  label-elem = element label {
7      BasicInteractionElement,
8      attribute labelAlignment {direction-attr},
9      attribute orientation {labelOrientation-attr},
10     icon-elem?,
11     element text {text-elem}
12 }
13
14 link-elem = element link {
15     BasicInteractionElement,
16     attribute visited {xsd:boolean},
17     element text {text-elem}
18 }
19

```

```

20     linkBar-elem = element linkBar {
21         BasicInteractionElement,
22         attribute linkColor {color-attr},
23         attribute splitterColor {color-attr},
24         element link {link-elem}+
25     }
26
27     paragraph-elem = element paragraph {
28         BasicInteractionElement,
29         element text {text-elem}?
30     }
31
32     tooltip-elem = element tooltip {
33         BasicInteractionElement,
34         attribute toolTipTrigger {trigger-attr} ?,
35         attribute tooltipDirection {tooltipDirection-attr},
36         element text {text-elem}?
37     }
38
39
40     callout-elem = element callout {
41         BasicInteractionElement,
42         label-elem?
43     }
44
45     text-elem =
46         attribute bold {xsd:boolean},
47         attribute undelined {xsd:boolean},
48         attribute strikethrough {xsd:boolean},
49         attribute italic {xsd:boolean},
50         attribute fontSize {xsd:int},
51         attribute orientation {labelOrientation-attr},
52         attribute color {color-attr},
53         attribute opacity {opacity-attr},
54         attribute align {align-attr},
55         xsd:string

```

Listing D.11: "RelaxNG Schema for the Wireframe Text Elements"

D.4.8. RelaxNG Schema of the Wireframe Annotation Elements

```

1     arrow-elem = element arrow {
2         NonInteractionElement,
3         attribute direction {direction-attr},
4         attribute curvation {curvation-attr},
5         attribute stroke {stroke-attr},
6         attribute leftArrow {xsd:boolean},
7         attribute rightArrow {xsd:boolean}
8     }
9
10    curlyBrace-elem = element curlyBrace {
11        NonInteractionElement,
12        attribute direction {direction-attr},
13        attribute orientation {orientation-attr},
14        element text {text-elem}?
15    }
16
17    stickyNote-elem = element stickyNote{
18        NonInteractionElement,

```

```
19|     element text {text-elem}?
20|   }
21|
22|   annotation-elem = element annotation {
23|     NonInteractionElement,
24|     attribute annotationType {annotationType-attr}
25|   }
```

Listing D.12: "RelaxNG Schema for the Wireframe Annotation Elements"