Engelschall
Technical Report
EnTR-03:2013.12

User Interface Composition

Specification, Functionality Classification, Hierarchical Composition, Model-View-Controller Roles and Component-Orientation

User Interface Composition

Specification, Functionality Classification, Hierarchical Composition, Model-View-Controller Roles and Component-Orientation

Ralf S. Engelschall, rse@engelschall.com, http://engelschall.com

Abstract

User Interfaces (UI) are complex, inherently hierarchical structures. They can be implemented under run-time via a composition hierarchy of UI Fragments, which are derived under design-time from Wireframe-based Storyboards through hierarchical decomposition. The resulting implementation, by implementing the UI Fragments with the help of Model-View-Controller based architecture patterns, then also follows the Component Orientation architecture paradigm.

Keywords

user interface, ontology, hierarchy, composition, component, composite, widget

1 Motivation

Independent of the used technology, User Interfaces (UI) of applications usually have a very high overall complexity in their implementation. To master complexities in general, two approaches are known to be very useful: applying the architecture principle *Logical Separation* and applying the architecture paradigm *Component Orientation*. We show how we can leverage from those also in the particular context of UIs with the help of the Hierarchical Composition process. Additionally, it is vital to have a common terminology and understanding of all the involved aspects.

2 Methodology

User Interfaces (UI) are inherently hierarchical structures. As such, it makes sense to both comprehend and implement them with a stringent hierarchical approach and applying the architecture principle *Logical Separation* (aka *Separation of Concern*) by hierarchically assembling the UI from UI fragments. For this, it is necessary to understand how to first decompose the (usually *Wireframe* based) specification of an UI into a hierarchy of UI fragments, implement each fragment as a separate component and then re-compose the UI under runtime again.

Additionally, for implementing UI fragments the architecture pattern Model-View-Controller (MVC) is usually preferred. This triad of model, view and controller roles is taken into account, too. All ingredients are named and defined and this way form an UI taxonomy. Finally, the relationships between the ingredients are defined and this way (together with the taxonomy) form an UI ontology.

Title:	**User Interface Composition**
Series:	Engelschall Technical Report (EnTR)
ID:	EnTR-03:2013.12
URI:	http://engelschall.com/go/EnTR-03:2013.12
ISBN:	978-3-944645-03-2

2.1 User Interface Specification

An *User Interface* (UI), during the Analysis phase of the Software Engineering process, is usually specified through a *Storyboard:* the visual surface of an application as a whole, defined with the help of one or more *Wireframes*. A *Wireframe* is a high-level sketch-like drawing of an UI *Panel* or *Dialog* (see below) and is comprised of one or more *Wireframe Areas*.

A *Wireframe Area* is the mid-level visual area of a *Wireframe*, usually functionally corresponding to a *Dialog* (see below) or a *Container*, *Control* or *Visual* (see below) and it is in turn comprised of one or more *Wireframe Elements*. A *Wireframe Element* is the low-level visual element of a *Wireframe Area*, consisting of text and/or geometrical graphics primitives.

The set of *Wireframes* in a *Storyboard* are interlinked through *Interactions*, i.e., user actions starting on a *Wireframe Area* (usually corresponding to a *Control*), causing arbitrary domain-specific functionality to run and ending with the appearance of another *Wireframe*. Additionally, the interactions can also be grouped and ordered into interaction paths, which correspond to domain-specific tasks.

2.2 User Interface Fragment Functional Classification

As UIs are complex structures, it is reasonable to break them down into a set of *UI Fragments* (see below) and classify *UI Fragments* into *Composites* and *Widgets*. A *Composite* is a high-level *UI Fragment*, which is either an orchestrating *Panel* or interacting *Dialog*. A *Widget* is a mid-level *UI Fragment*, which is either an orchestrating *Container*, an interacting *Control* or a non-interacting *Visual*.

A *Panel* is a *Composite* which is mainly orchestrating multiple contained *UI Fragments*. A *Dialog* is a *Composite* which is mainly interacting with the user through contained *Wigets*. A *Container* is an active Widget, which is mainly logically grouping other *UI Fragments*. A *Control* is an active *Widget*, which is mainly interacting with the user through input mechanisms like keyboard, mouse, touch-screen, etc. A *Visual* is a passive *Widget*, which is just showing content textually and/or graphically.

2.3 Hierarchical Composition

To being able to hierarchically compose an UI under run-time, we first have to hierarchically decompose its specification under design-time. For this, we start at the *Storyboard* level. The *Storyboard* corresponds to the root node of the composition hierarchy and leads to a root *User Interface (UI)* node.

Then we take *Wireframes* and *Wireframe Areas* and derive *UI Fragments*, i.e., high-level visual UI parts, consisting of other nested *UI Fragments* and *UI Elements*. *UI Elements* in turn are low-level visual UI parts, consisting of text and/or geometrical graphics primitives.

The crux of the hierarchical decomposition process is in two major creative decisions: First, when to choose a a Composite or a Widget flavor for an UI Fragment. Second, when to use an all-in-one UI Fragment and when to use a finer sub-hierarchy of UI Fragments. Both decisions are highly ambiguous and depend on personal preferences, domain-specific relationships and even technical constraints.

The key rules are: First, a Composite is usually always non-reusable and hence a singleton in the composition hierarchy, while a Widget intentionally is reusable and potentially occurs multiple times in the composition hierarchy. Second, a reasonable balance between all-in-one "god composites" (which are hard to maintain) and fine-granular composite sub-trees (which can cause noticeable UI communication overhead) has to be chosen. Third, the largest *Wireframe Areas* which occur in multiple *Wireframes* are good candidates for UI Fragments.

The result is a composition hierarchy with the *User Interface (UI)* as a whole at the root, then a tree of *UI Fragments* as intermediate nodes and finally primitive *UI Element* nodes nodes at the leaves.

2.4 Component Tree

Component Orientation is a major architecture paradigm which implements especially the important architecture principles *Logical Separation* (separation of concerns between the components of a solution), *Structural Modularity* (splitting of a solution into manageable structural components) and *Encapsulated Complexity* (complex related aspects of a solution are encapsulated into a single responsible component).

As such it is the perfect vehicle to master the inherent complexity of the UI implementation. In this context all *Composites* and *Widgets* are implemented as separate *Components* and the composition hierarchy is represented as a component tree.

A *Component* is an Object-Oriented (OO) grouping of data and behaviour, wrapping a *Backing Object*. Usually in the form of a generic functionality provided by a framework. A *Backing Object* is an OO grouping of data and behaviour, backing a *Component*. Usually in the form of domain-specific functionality provided by the application.

A *Component* can also host a so-called Shadow Tree, rooted at another *Component*.

2.5 Model-View-Controller (MVC)

Model-View-Controller (MVC) is a well-known — but most of the time very less strictly applied — architecture pattern for implementing *UI Components*. Independent of the particular MVC flavor, there will be always a *Triad* of *Model*, *View* and *Controller* roles a *Component* plays when implementing *Composites* and *Widgets*.

Just notice that in practice, because of the architecture principles *Avoided Redundancy* and *Contextual Adequacy*, for implementing a *Widget*, we usually leave out the *Controller Component*, because it is usually provided by a parent *Composite*. Similarly, for implementing a *Composite*, we often leave out the *Model* and *View Components*, because they are usually provided by child a *Widget*.

3 Related Work

The idea of applying *Component Orientation* to the problem domain of User Interfaces is not new [Batory & O'Malley 1992]. It was also described by [Haft & Olleck 2007] [Haft 2009] and successfully applied to their *Quasar Client* architecture.

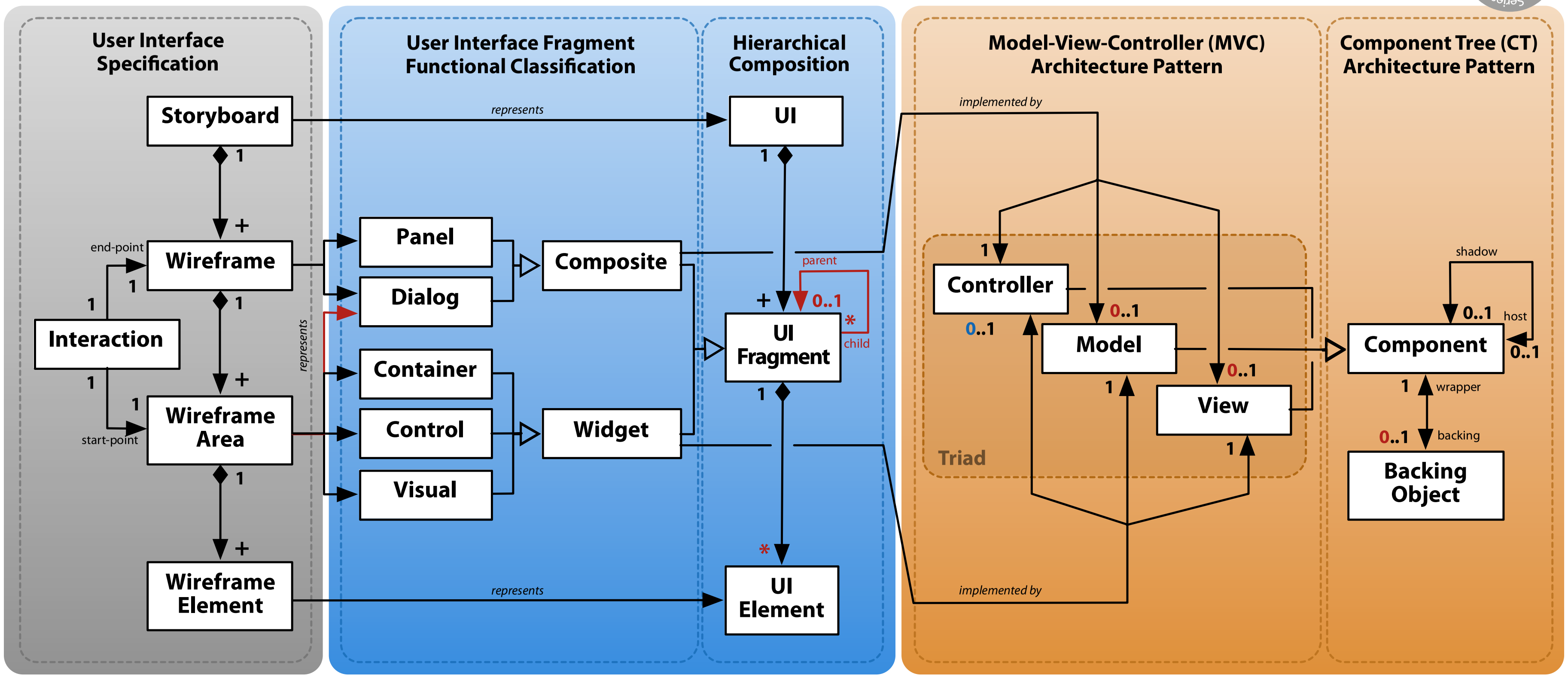
4 Acknowledgement

Thanks to Martin Haft (sd&m Research, 2008) for detailled first-hand information about *Quasar Client* and its corresponding *Component-Oriented Client-Architecture*, which was one of the main inspirations for this methodology.

5 References

- Martin Haft, Bernd Olleck: *Komponentenbasierte Client-Architektur*, Informatik Spektrum 30, p. 143, March 2007.
- Martin Haft, *Quasar-Client-Architectures,* Version 1.02, June 2009.
- Don Batory, Sean O'Malley: *The Design and Implementation of Hierarchical Software Systems With Reusable Components*, ACM Transactions on Software Engr. and Methodology, October 1992.
- Wikipedia: *Ontology (information science)*, http://en.wikipedia.org/wiki/Ontology_%28information_science%29
- Martin Fowler: *GUI Architectures*, http://martinfowler.com/eaaDev/uiArchs.html, July 2006
- Steve Burbeck: *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC),* http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html, 1987

User Interface **Ontology**



Storyboard: Visual specification of an application as a whole, defined with the help of one or more **Wireframes**.

Wireframe: High-level visual specification of a **Panel** or **Dialog Composite**, intentionally drawn in sketch-style.

Composite: High-level **UI Fragment**; either an orchestrating **Panel** or interacting **Dialog**.

Widget: Mid-level **UI Fragment**; either an orchestrating **Container**, an interacting **Control** or a non-interacting **Visual**.

User Interface (UI): Visual presentation and interaction surface of an application as a whole, consisting of one or more **UI Fragments**.

UI Fragment: High-level visual **UI** part, consisting of other nested **UI Fragments** and **UI Elements**.

UI Element: Low-level visual **UI** part, consisting of text and/or geometrical graphics primitives.

Controller: Active **Component**, dedicated to perform presentation provisioning to a **Model** and presentation actioning from a **Model**.

Model: Passive **Component**, dedicated to host (and perform logical operations on) values (parameters, commands, states, data and events) to serve a **View**.

View: Active **Component**, dedicated to displaying and interacting with a view mask, based on a bi-directional binding to values in a **Model**.

Interaction: User action on a **Wireframe Area**, causing arbitrary domain-specific functionality to run and resulting in the appearance of another **Wireframe**.

Wireframe Area: Mid-level visual area of a **Wireframe**, functionally corresponding to a **Dialog Composite** or a **Container**, **Control** or **Visual Widget**.

Wireframe Element: Low-level visual element of a **Wireframe Area**, consisting of sketch-styled text and/or geometrical graphics primitives.

Panel: **Composite**, mainly orchestrating multiple contained **UI Fragments**.

Dialog: **Composite**, mainly interacting with the user through contained **Widgets**.

Container: Active **Widget**, mainly logically grouping other **UI Fragments**.

Control: Active **Widget**, mainly interacting with the user through input mechanisms like keyboard, mouse, touch-screen, etc.

Visual: Passive **Widget**, just showing content textually and/or graphically.

Component: Object-oriented grouping of data and behavior, wrapping a **Backing Object**; usually in the form of generic functionality provided by a framework.

Backing Object: Object-oriented grouping of data and behavior, backing a **Component**; usually in the form of domain-specific functionality provided by the application.

Series: Engelschall Technical Report (EnTR)
http://engelschall.com/go/EnTR

Document: EnTR-03:2013.12
http://engelschall.com/go/EnTR-03:2013.12
ISBN 978-3-944645-03-2

Last Modification: 2013-12-31

Author: Ralf S. Engelschall
Weblinger Weg 28, 85221 Dachau, GERMANY
rse@engelschall.com
http://engelschall.com