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

Pattern Guideline and Constraint Validation of Run-time Communication in User Interface Component Architectures

Christian Vaas

Master's thesis in the Elite Graduate Program: Software Engineering

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

Pattern Guideline and Constraint Validation of Run-time Communication in User Interface Component Architectures

Matriculation number:	1228713
Started:	29. July 2013
Finished:	29. January 2014
First assessor:	Prof. Dr. Alexander Knapp
Second assessor:	Prof. Dr. Bernhard Bauer
Supervisor:	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.

I, hereby certify that this thesis has been written by me, that it is the record of work carried out by me and that I have not used anything else but the indicated sources and tools.

Augsburg, den 3. Februar 2014 Christian Vaas

Zusammenfassung

KONTEXT Während in den letzten Jahren die Leistungsfähigkeit unserer Endgeräte stetig stieg, bleibt die Netzwerk Bandbreite immer noch der Engpass. Aus diesem Grund ist das Übertragen ganzer Masken einer Benutzeroberfläche (UI) keine praktikable Lösung mehr. Somit geht der Trend zu Rich Client UIs, welche insbesondere auf HTML5 Technologien basieren.

MOTIVATION Bei der Entwicklung solcher UIs haben sogar kleine Anwendungen eine komplexe Code-Struktur und können zu Problemen bei der Softwarequalität führen. Diese Komplexität muss bewältigt werden, um akzeptablen Code in Hinsicht auf Wiederverwendbarkeit und Wartbarkeit liefern zu können.

ANSATZ Eine bewährte Lösung hinsichtlich der Architektur einer solchen Anwendung ist eine strikte hierarchische Dekomposition. Dabei wird eine UI auf eine Hierarchie von Oberflächenelementen abgebildet, die wiederum auf eine Hierarchie von Widgets, welche letztendlich auf eine Hierarchie von Darstellungselementen abgebildet wird.

HERAUSFORDERUNG Seit Jahrzehnten stellen UI-Toolkits die Widget-Hierarchie und Browser die Darstellungselement-Hierarchie bereit, doch die Architektur der Komponenten und deren Kommunikation wird oft vernachlässigt. Obwohl jüngste Entwicklungen einen Nutzen aus der Kombination von komponentenorientierter Architektur und der Trennung von Model und View Komponenten ziehen, besteht bei der UI-Entwicklung immer noch eine große Diskrepanz zwischen Theorie und Praxis.

LÖSUNG Heutzutage müssen Entwickler immer noch für jeden Einzelfall eine Lösung unter Berücksichtigung des verwendeten UI-Komponenten-Frameworks und dessen Besonderheiten erarbeiten. Ein Ansatz im Software Engineering Sinne fordert jedoch, dass dieser wiederkehrende Vorgang durch vorbereitete und wiederverwendbare Kommunikationsmuster abgelöst wird. Um nun die Konformität der Implementierung einer Anwendung mit eben jenen Mustern analytisch zu prüfen, wird zur Laufzeit ein regelbasierter Constraint Checker verwendet. Die Regeln für diese Überprüfung werden direkt von den Mustern abgeleitet und spiegeln somit das geforderte Verhalten der Anwendung wieder.

Abstract

CONTEXT Since years client devices become more and more powerful while the network bandwidth is still a bottleneck. Hence transferring User Interface (UI) masks is no longer an option. Thus, there is a strong trend towards Rich Client User Interfaces. Most notably, nowadays portable user interfaces are developed via the popular HTML5 technology stack.

MOTIVATION Nevertheless, we are faced with a software quality challenge: even small UIs, e.g., Mobile Apps, cause an inherently complex code structure. This has to be mastered in order to guarantee reasonable code reusability and maintainability.

APPROACH From an architecture point of view, the well-proven solution is a strict Hierarchical Composition: the UI is mapped onto a hierarchy of UI Composites, which in turn is mapped onto a hierarchy of Widgets, which in turn is mapped onto a hierarchy of Display Elements.

CHALLENGE While since decades UI toolkits address the Widget hierarchy and Browsers address the Display Elements, the architecture layer of UI Composites and their communication is still often neglected. Even though recent developments greatly leverage from the combination of Component-Orientation architecture paradigm and Model-View Separation architecture pattern, there is still a noticeable gap between theory and practice in UI implementation.

SOLUTION Developers currently still have to ad-hoc map from domain-specific use cases to features the underlying UI Composite framework provides. Software Engineering experience dictates that those recurring tasks should be addressed through pre-thought-out and reusable communication patters and their constructive mapping onto abstracted UI Composite communication features. In order to ensure the compliance of an application with the set of those patterns, an analytic check during run-time is performed using a rule-based constraint checker. Since patterns imply particular communication paths, the constraints are directly derived from those communications.

Contents

List of Figures

1. Introduction

1.1. Challenges

"Divide et impera" — Julius Caesar

According to the divide and conquer strategy, which was first used by Julius Caesar to rule the roman empire, one big problem should be cut into smaller pieces, because solving the whole problem at a time would be disproportionally difficult.

As this is a common approach in computer science, it is also used when building an application. Here, the application is divided into several components, that form the application's architecture. An architecture can be developed from various perspectives and at multiple levels of abstraction. Since the quality of these architectures highly depends on finding the correct points of intersection, we try to aid the architect during this process. Figure 1.1 displays a well known and simple looking user interface for watching videos.

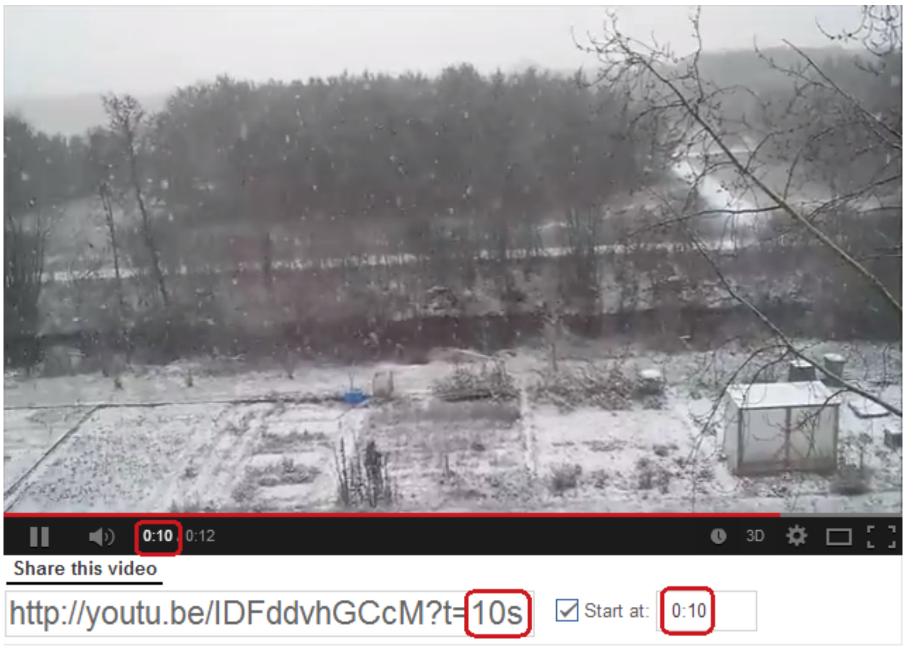


Figure 1.1.: Teasing example - YouTube^{TM}

Since the highlighted information is displayed in three different positions, the questions is: Which component inside our architecture is responsible to hold this data, while simultaneously providing it to the minimum possible and maximum necessary scope of components? With this being one of the easier questions that come up when trying to cut an application into components, we get a small preview on what to come. Throughout this thesis, we will answer this and even more mature questions occurring during the user interfaces development. Details about the used approach are described in Section 1.2.

The development of an architecture is only the beginning. Another question is: How can we guarantee strict adherence to an architecture throughout the implementation process and of course in the final version that is deployed? This question will also be answered shortly in the next section, alongside the enforcement of certain quality properties and behavior restrictions.

1.2. Approach

In software, everything and nothing is about quality. If a company just wants to have quick results and does not plan for the future, then it can completely ignore any kind of software quality. But if a minimal chance exists that the software has to be maintained or extended, then we should consider to invest in quality during the development process. Sure, there is a wide range, where quality can be injected into software, but as the previous section already indicated, we focus on the component architecture and to be even more precise: on the communication among the different components.

Our approach is to first examine already present applications and to extract sophisticated solutions for reoccurring problems. This leads to a pattern catalog, which contains information on the rationale, structure and especially the communication scheme of each pattern. Derived from the communication, we provide constraints that the implementation of an architecture has to adhere to, in order to guarantee high quality and a correct realization. These constraints can either be peephole constraints, that focus on a single communication occurrence between two components or temporal constraints, which allow statements about the sequence of such communication. Besides the architecture enforcement, we also give advise about checking coding guidelines and ensuring common interfaces among components, using our developed constraint Domain Specific Language (DSL).

Last, we introduce a tool to check these constraints at run-time. As we focus on web based Single-Page Applications (SPA), we install a proxy server that instruments the application code while it is loaded through it. This makes the instrumentation completely transparent to the developer. The instrumented code supplies our tool with the inter-component communication in real-time as well as with other application state information, e.g. the Component Tree (CT), and the state of each component. This communication traces, as we will introduce them later, are then checked by our tool and output to the user.

Now, before we start refining our approach, we set the context of this thesis in Chapter 2, including Model-View-Controller (MVC) and a component architecture as a common procedure to implement user interfaces. After explaining the concepts of component communication and run-time verification in Chapter 3, we proceed with the tasks of the different component traits, Chapter 4 used in the pattern & constraint catalog written down in Chapter 5. This catalog is completed with implementation quality constraints in Chapter 6. Afterwards, we show the application of the catalog in Chapter 7. Next, we look under the hood of our constraint checking algorithm in Chapter 8. At the end, we close with a short summary and a prospect for future work, in Chapter 9.

2. Context

This chapter introduces the concepts around which this thesis is built. We start with a brief explanation of the technique to hierarchically decompose user interfaces to identify their component architecture. Then we move to the user interface architecture ontology, which provides a common wording for the elements used in a user interface. This is followed by the Model-View-Controller principle adapted and tied to the hierarchical component architecture of Section 2.1. And finally we explain the features a component system, which is ComponentJS in our case, has to provide to allow a component based approach to build a user interface.

2.1. Hierarchical Decomposition of User Interfaces

The V-Model tells us that the first step is the requirements analysis phase, when we start building an application (see [RB06] for details). Afterwards, the system architecture has to be developed. During this process, the different components are identified and the application is cut into handy chunks alongside these components. This is a common approach for any type of software and quite feasible in almost all areas. The key is always to have a good heuristic for the component identification. Here, depending on the level of abstraction, different techniques are popular. In an early stage, a separation into several tiers can be suggested. The components of a client-sever application can be structured into presentation, logic and data responsibilities.

User interfaces are one shape of the presentation layer from the above example. Inside this layer, a finer granularity has to be applied and other indicators for the component finding have to be used. One way is hierarchical decomposition. Several publications ([HO07], [Engc], etc.) suggest this approach due to the inherent hierarchical character of a user interface. This hierarchy can be extracted by examining the visual containment of the different views [Pan97], making up the whole interface.

According to [Pan97] this approach is best to be combined with the Presentation Abstraction Control (PAC) paradigm, which introduces a hierarchy of PAC agents, whereby each agent is responsible for the implementation of a fraction of the requirements. Since PAC agents are dedicated to a high level of abstraction, they cannot represent the final component architecture but only show a first aggregation. That is why the established composition now has to be refined to the component level. This is done by preserving the hierarchy and decomposing the agents into MVC triads, [HO07] and [BM00]. The result is a layered architecture, where several MVC-triads are stacked on top of each other. In [CKP00], this is called Hierarchical Model View Controller (HMVC). As stated in [LVC89], this approach allows two ways of development separation: the implementation of the single display components on the one hand and the composition of them to the user interface design on the other hand. Of course, this only works as long as an agreed composition protocol is adhered to.

Eventually, as [HC95] also approves, we can say that a user interface provides enough space for functionality and requirements that leads to an explosion in complexity in the majority of cases. This can only be mastered by resorting to strong techniques like the hierarchical decomposition flanked by concepts like PAC and MVC.

2.2. User Interface Architecture Ontology

An ontology, describing and linking the elements used in the user interface architecture proposed by [Engd], on which many of the ideas of this thesis are based, is shown in Figure 2.1. As the image implies, the core of this architecture is the MVC triad. We have already mentioned this pattern in the previous section and it will accompany us throughout the whole thesis. On the right hand side, we see model, view and controller as the triad parts that are generalized to components, which can be nested arbitrarily. They are also linked to the elements of the hierarchical decomposition on the left, which may contain instances of themselves.

In the UI hierarchy the top level element is the UI as a whole, which is built up using several composites. A composite is a logical group of other composites or widgets and thus also aggregates MVC instances. The underlying widgets are the first abstraction of the display elements. They blend these primitive graphical representations like lines, text and geometric shapes together to more complex UI elements. Examples for widgets are buttons, static textual content or a scrollable container.

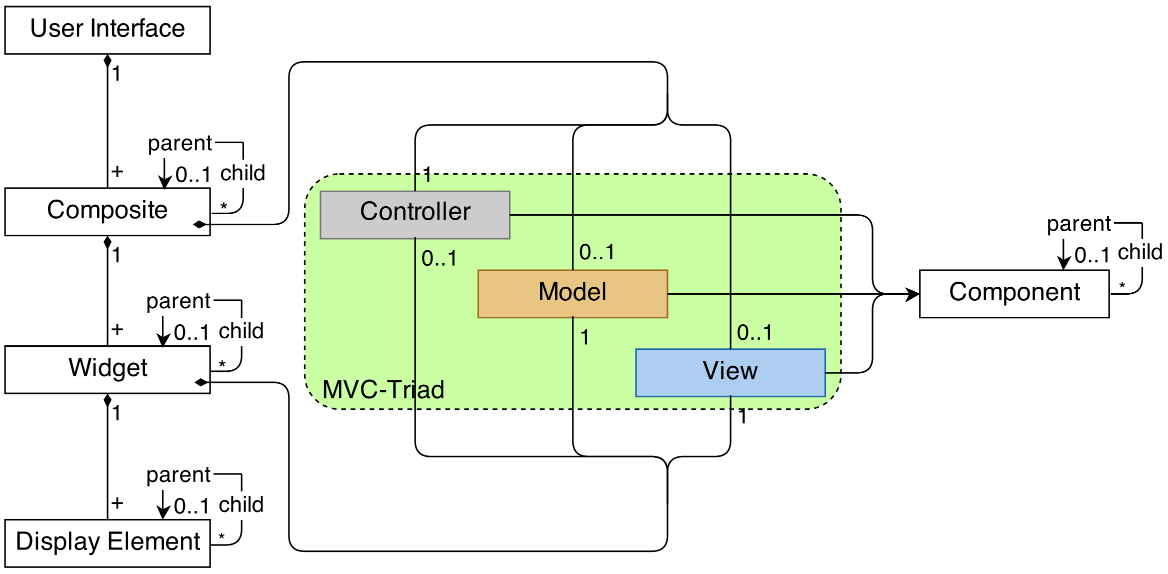


Figure 2.1.: User Interface Architecture Ontology (Excerpt of [Engd])

Like we already outlined in Section 2.1, the UI is inherently hierarchical. This is also reflected by the nestability of components on the one side and composites, widgets and display elements on the other side. As mentioned in the introduction, we focus on web applications. Hence, our display elements are mainly the nodes of the DOM tree. The result are three tree structures.

- The DOM tree, making up the HTML elements and thus the actual graphical representation

- The widget tree as an object mesh, which is the first abstraction of the DOM tree. This tree can be provided by a UI toolkit, e.g. jQuery UI or ExtJS.
- The Component Tree, representing the component architecture, which consists of model, view and controller instances

Of course, if any non-hierarchical technology ,e.g. Adobe Flash^{TM} is used for the display elements, then only the latter two trees emerge.

2.3. Model-View-Controller on a Component Tree

The patterns and constraints, proposed throughout this thesis are based on the MVC on a Component Tree architecture approach. Since both concepts can be very involving when presented at the same time, we start with the description of the MVC paradigm. Thereafter, we move to the component tree, which will later bear the MVC triads.

Model-View-Controller

In general, the MVC pattern is no new invention. But for the user interface component architecture by [Enge], they introduced a slightly modified version that unites the advantages of PAC, MVP and MVC. A comparison of the different approaches can be found in [Gre07]. To examine the three parts of the MVC pattern, we decompose it stepwise in a top-down fashion.

The first component is the view, shown in Figure 2.2. As it is the component whose testing is most expensive, it follows the suggestions from [Fowd], where as less code as possible is put into this impractical location. To achieve this, the application specific behavior is relocated to the controller and model [BM00]. Thus only the different bindings have to be set up inside the view. Technically, each binding is deployed using observer synchronization [Fowc], which utilizes the observer pattern [BD04]. This stands antithetic to flow synchronization [Fowa], that reduces the amount of synchronization operations, but lacks the possibility to have more than one view of a model.

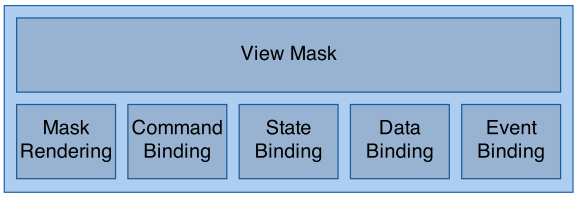


Figure 2.2.: View [Engc]

Another issue is the granularity of the observed data. If conceived too fine grained, it floods the application with events and when conceived too coarse, small changes cause updates to big areas, unnecessarily. For that reason the data is divided into five binding groups, each providing the appropriate richness of detail for the categories of tasks outlined in Table 2.1. Using this categorization of [Engc], all data required by the view is covered and each piece of data is as precisely observed as needed [Fowf]. It also enables us to plug any arbitrary view on top of the model component as long as it satisfies the interface formed by the underlying model. Of course there is no free lunch and using the observer pattern for the bindings has also downsides. The

Category	Explanation
Mask Rendering	Required to render the layout, e.g. device orientation.
Command Binding	Operations triggered by the controller that entail UI actions.
State Binding	Interface element states like a button being enabled or not. It can also contain domain states like data validity. Here an object containing the validation result is conceivable [Fowb].
Data Binding	Data meant to be displayed directly in the UI. It is also updated when changed in the UI.
Event Binding	Special data values, indicating the occurrence of an event in the UI, e.g. a button was clicked.

Table 2.1.: ComponentJS API methods

implicit control flow changes are hard to debug and orphaned observes may cause memory leaks.

After having understood the view component, we head over to the presentation model. The main task of the model is to store domain and state data. Five different types of data are displayed in Figure 2.3, which stand in conformity with the five bindings of the view. In compliance with [Pot], the model encapsulates the data to form a central position for persisting and sharing and thus clearly separates the view from the data management. When deciding which data belongs to the presentation model, it is important to have a strict separation of domain objects and presentation objects in mind. Only the latter are allowed to be stored in the model, whereas the former only used in the communication with the service facade. With all data inside the presentation model, it serves as an abstraction of the UI, which also enables Headless Testing for the user interface's architecture [Fowe]. The last box in Figure 2.3 is the presentation logic. In contrast to the model of the original MVC in Smalltalk-80 [Bur92], the presented model component is not completely passive but also contains logic that enforces cross value dependencies [Fow06].

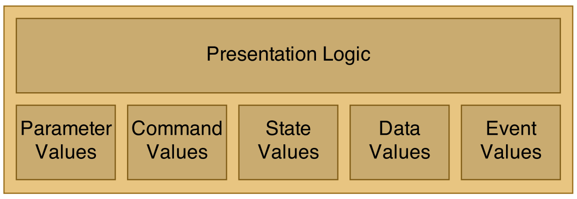


Figure 2.3.: Model [Engc]

An example is a mutual exclusive checked state of a checkbox group. If one checkbox is checked, then the presentation logic switches the enabled state of the other checkboxes to `false`. In addition to such pure data changes, the presentation logic can also handle events that lack any interaction with other components or the service facade and thus only affect internal data [Fowg].

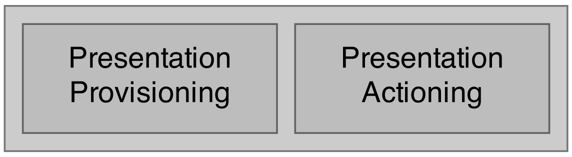


Figure 2.4.: Controller [Engc]

Last we examine the controller component as the gateway towards the remaining application and back-end. As we can see in Figure 2.4, it is split into two parts: the presentation provisioning and the presentation actioning. To provide the necessary data, the presentation provisioning directly accesses the service facade and pushes the information into the overlaying model. The presentation actioning is similar to the presentation logic we know from the model component. It also pulls complexity off the view to improve testability. The difference lies in the data affected by the logic, whereby the presentation logic is only allowed to touch internal data, the actioning has full access to the service facade and can thus trigger server actions [Fowg]. This clear distinction of presentation logic and presentation actioning is founded in the principle of Logical Separation.

Component Tree

The goal of the Component Tree is to provide a communication infrastructure among the different components of an application. In contrast to many approaches (e.g. [Inc]), which use a flat event bus to transfer information between components, the Component Tree explicitly reassembles the visual containment hierarchy of the particular UI. This ensures a natural scoping of the events, as we will see when inspecting the different delivery phases. The Component Tree is founded on an artificial root element, which hosts the remaining sub-trees of components. In the following, we have a look at the event delivery mechanism through this structure. It is divided into three phases inspired by the event handling within the DOM tree [Pix].

Capturing
: As the first phase, it is responsible to carry the event from the root component to the target component. Each component on the path, excluding the target component, can listen to the event while it is traversing the Component Tree.

Targeting
: Within this phase, the event is actually dispatched to the target component, which can now perform actions given a listener was registered beforehand.

Bubbling
: After hitting the target component, the event travels back to the root component. Here all components on the path again have the opportunity to catch this event. The target itself is excluded again.

Given these phases, the event system provides communication up- and downwards the Component Tree, starting and finishing at the root component. Even though it is simple and lean, it is still powerful enough to support the mechanisms required throughout this thesis, which are outlined 2.4.

2.4. Component System features

In this chapter, we focus on the main features of the component system (ComponentJS), which is used throughout this thesis. We will start with the event based component communication, followed by the property lookup mechanism and end with the component life-cycle. Before we can examine the individual communication methods, we need to have a look under the hood of the component system itself. As we have already mentioned in the previous section, the structure underlying the component architecture is shaped like a tree. It handles the communication among the components, whereby events are passed along it to transmit information from component to component.

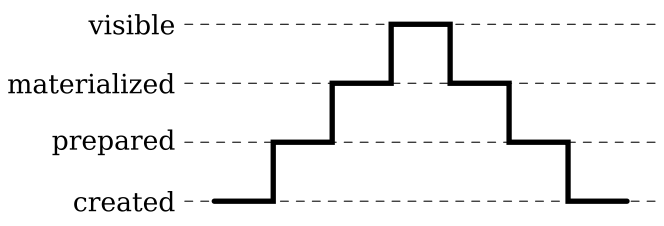
Every method, except the `publish` and `subscribe` pair is just a specialization of this mechanism. In addition to the usual event delivery technique, ComponentJS offers the possibility of marking events as *spreading*. This adds a new phase between the *targeting* and the *bubbling* phase, which is called *spreading* phase. In this phase, the event is additionally dispatched to all children of the target component.

Equipped with this powerful communication infrastructure, several convenience methods are provided to simplify the implementation process. Table 2.2 shows a brief listing of the methods used within this work, see the ComponentJS documentation [Enga] for an exhaustive list and the full Application Programming Interface (API) specification.

Method	Description
Model	Used to create a model accessible through the component system. Basis for the following two methods.
Value	Provides get and set access to a member of a model.
Observe	Monitor a model member for changes. A parameter *operation* can be set to the values “set”, “get” and “changed” in order to specify the moment when the method is triggered.
Register	Sets up a service endpoint which can be called by name including a list of parameters. It is then run synchronously and can thus provide a direct result value.
Call	Invokes a service method with given parameters and retrieves its result.
Socket	Prepares a socket for which a plug and unplug method is supplied and called when one of the two actions occurs, respectively.
Plug	Triggers the plug operation of a socket on a target component.
Link	Creates an artificial socket that bridges to a real socket on the same component or even a completely different component.
Publish	Releases an event while optionally piggybacking data onto it. The event follows the different delivery phases already mentioned.
Subscribe	Catches an event passing by a particular component. Can be flagged with *spreading* in order to also receive spreading events.
Property	Provides read and write access to a key-value like storage that is attached to a component instance.
State	Tells the component framework to switch a target component to a given state.

Table 2.2.: ComponentJS API methods

After these pure communication methods, we need to understand how the mentioned `property` method (2.2) is working. Initially, when called upon a component, an internal property lookup mechanism is started. The procedure inspects the key-value stores bound to each component, hereby the order is determined by performing a traversal,



visible Component is visible to the user.
materialized Component is rendered onto the DOM tree.
prepared Component is prepared and ready for rendering, i.e. its data is loaded.
created Component is created and attached to component tree.

Figure 2.5.: Component life-cycle [Engc]

which starts at the target component and passes to the root of the component tree. It is trying to resolve the property at each component, however the first successful retrieval aborts the traversal and returns the value.

The last concept we have to highlight before we are able to discuss the communication among components, is the life-cycle of components [HO07]. This is important due to the fact that we want to limit some of the above mentioned convenience methods to certain states, as we will see in Chapter 6. In addition to this, the deliberations in Chapter 3.4 require us to maintain the current state of the whole system at any time to do proper run-time verification.

In a common application, according to [Engb] and [HO07] the component states displayed in Figure 2.5 are sufficient to cover the behavior of a user interface. The picture on the left illustrates the transitions that are available in each state. As an example, a component that resides in the *materialized* state, can either advance its state to *visible* or lower its state to *prepared*. This transition order is invariant to all components within an application, thus it is not possible for a component to jump directly from the *created* to the *visible* state.

Now, if we consider a strict total order (2.1) over the component states S and the predicate *isParent* (2.2) that indicates whether a component C is a parent of another one. Then, we can set up condition (2.3) which forces all parents to be in no lower state than their child components. This rule follows the idea that showing nested interface elements implies that their parent components need to be visible, too.

$$\succ\ \subseteq S \times S : visible \succ materialized \succ prepared \succ created \tag{2.1}$$

$$isParent : C \times C \to \mathbb{B} \tag{2.2}$$

$$\forall x, y \in components : isParent(x, y) \Rightarrow state(x) \geq state(y) \tag{2.3}$$

This behavior it perfect to save a lot of mindless glue code, since it models the natural flow of a UI. However, it is a common pitfall when debugging applications as the implicit state changes are generated by an explicit change request that is preceded by transitions generated to preserve the invariant (2.3). From a developers point of view, this can be irritating at a first glance. In [Engc] this mechanism is described in detail.

3. User Interface Communication Concepts

3.1. User Interface Communication Ontology

Ontology

The ontology sets all parts participating in the creation process of a user interface in relation to each other, from an architectural point of view.

Taxonomy

Inside the taxonomy, a definition for all parts included in the user interface communication ontology is given. This guarantees a standardized way of communication accompanied by a common understanding of the precise meaning of each particular term.

User Interface
: Point where the human-machine interaction takes place. The interface enables the user to conduct his use cases and gives feedback conversely.

Use Case
: A task that needs to be performed in order to fulfill a requirement of the user of the application.

Architecture
: The set of concrete components, the relations between them and their attributes molding an application in its entirety.

Implementation
: The code realizing a certain user interface by following the designated architecture.

Architecture Principle
: Fundamental truth, rule, tenet or policy an architecture follows.

Reference Architecture
: Reusable proven architecture template, based on a set of patterns and following one or more tactics and strategies.

Pattern
: An approved solution for reoccurring problems, whereby the level of details comprises a handful of component communications.

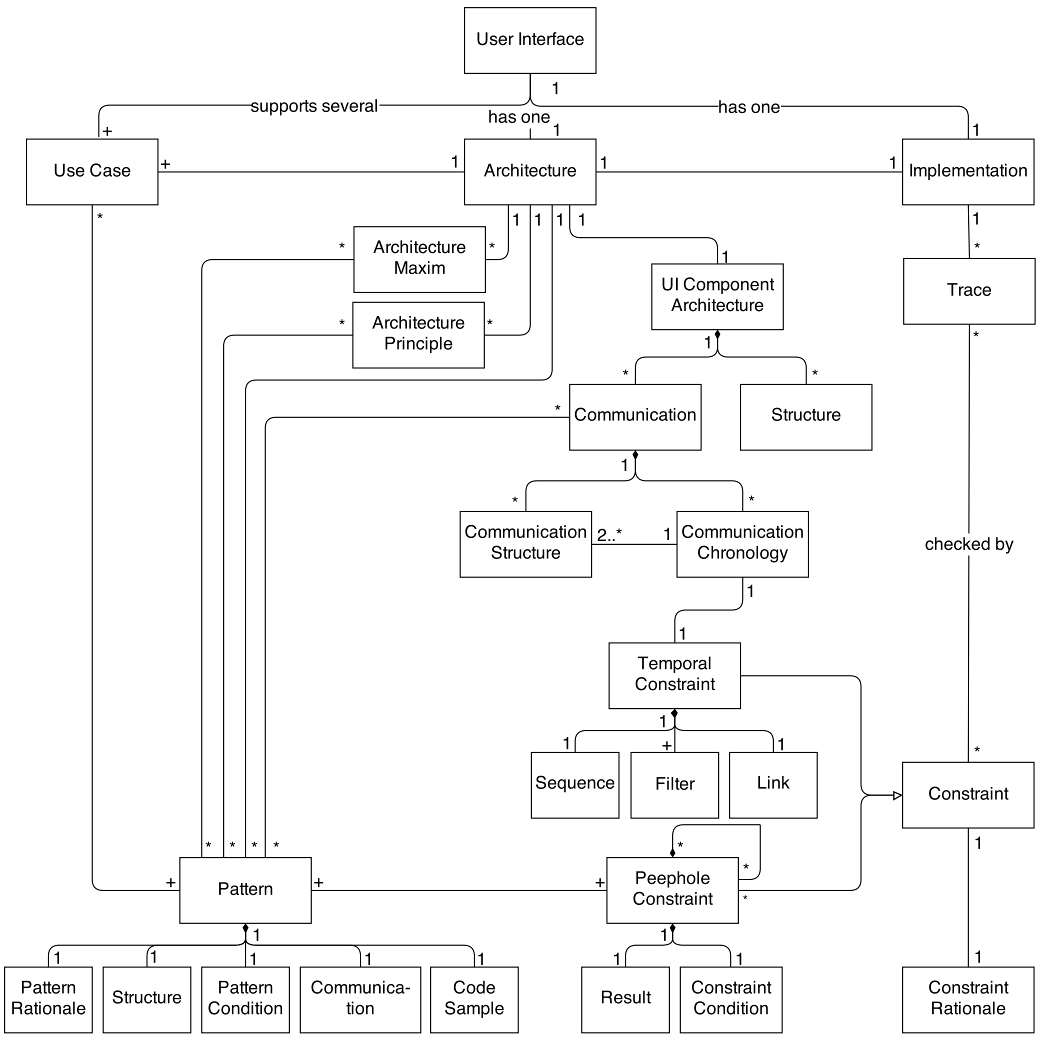


Figure 3.1.: User Interface Ontology

Pattern Rationale
: A textual description that outlines the reason for choosing and the intent of a certain pattern.

Communication
: The dynamic part of a pattern. It shows the collaboration paths between the components during run-time.

Structure
: A set of components that build up a component tree fragment and their assembly necessary to accomplish the goal of a pattern.

Pattern Condition
: A rule stating when the use of the pattern is indicated and appropriate.

Code Sample
Exemplary code snippet demonstrating an implementation of a particular pattern.

Trace
Documents the run-time communication between components, consisting of a source component, issuing an operation with particular parameters on an origin component.

Constraint
A formal specification partitioning the space of traces into forbidden and permitted ones.

Constraint Rationale
A textual description that outlines the reason for the existence of a constraint.

Constraint Condition
An expression, deciding whether a constraint is applicable or not.

Result
Indicates whether a given trace is permitted or forbidden.

Architecture Maxim
Fundamental, generally valid set of values and rules to guide the application architecture discipline.

Temporal Constraint
A formal specification for the behavior of an application during its run-time.

Sequence
The order of traces considered for a temporal constraint.

Communication
Messages, exchanged between the components of the application.

Communication Structure
Assembly of the components participating in a message exchange inside the application.

Communication Chronology
Sequence of messages between components in their temporal order.

3.2. Communication Patterns

A communication pattern provides a sophisticated solution for a common problem involving information exchange between several components within a user interface's application architecture. It describes the participants of the communication as well as the chronological order of the transmitted messages, since it may not only consist of one message but of several. After reviewing and examining various architectures, implementations and common use cases of user interfaces, we found a handful of patterns describing the recurring communication sequences among components. An example is the Master-Details use case (7.3) which was the first incident leading us, together with several other observations, e.g. in the Login use case (7.1), to the Common Ancestor pattern (5.8). Following this constructive technique we came up with the patterns presented in Chapter 5. The benefits of a pattern collection are obvious. The work of thinking about a solution has only to be done once. The found solution is proven, hence the developer can rely on it. And last, a pattern serves as a common communication term. The usage of patterns within the application development process can be structured into two phases.

Phase 1
The intelligibility of the user interface domain is improved, as the problem space can be inspected independent of a particular project. This proactive step is started, before the first project can make use of the pattern catalog and continues as long as the pattern catalog evolves and finally settles in its final version.

Phase 2
In this phase, a pattern comes to use to address problems in a particular project. It provides a distinct, swift and communicable solution including a reference implementation. Of course, it depends on the maturity level of the developer not to make inflationary use of patterns just for their own sake. That is why a problem from the functional domain has to be identified and matched with a resolving pattern, then validated and afterwards implemented.

3.3. Communication Constraints

A communication is represented by a tuple $\Gamma = \{(\epsilon, \sigma, t(\sigma), \theta, t(\theta), \omega, \rho)\}$. Component ϵ provides the time elapsed since the observation of the application has started. The variables σ, θ are elements of ς, which is the set of all components in an application's architecture. Function $t : x \to \tau$ donates the component trait, Chapter 4, $\tau = \{\text{view}, \text{model}, \text{controller}\}$ for a given component x. Furthermore, ω contains the transmitted operation with $\omega \in \{\text{value}, \text{publish}, \text{subscribe}, \text{call}, ...\}$ (see 2.4 for an exhaustive list). And last, variable ρ holds the parameters for operation ω.

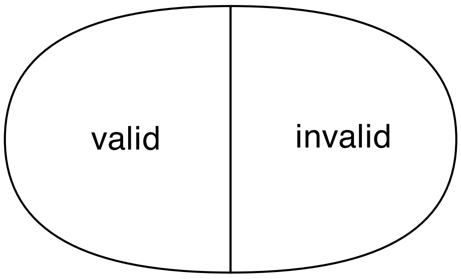


Figure 3.2.: Communication categories

The partition diagram in Figure 3.2 represents all tuples Γ exchanged within an application. These communications Γ can be partitioned into two categories: permitted, valid ones on the one hand and prohibited, invalid ones on the other hand. Within this thesis, we set up constraints Υ that specify the affiliation of each communication to these two groups. We can treat our tooling, described in Chapter 8 as a function

$c : \Upsilon, \Gamma \to \{\text{valid, invalid}\}$, of course the goal for the developer is to minimize the members of the invalid partition by aligning the code to the applied constraints. Since we trace the communication at run-time, the categorization is limited to be carried out during or after the development process and thus has a pure analytic character. The constraints that come to use are either derived from patterns as outlined in Chapter 5 or set up to enforce coding guidelines and other features, e.g. FAP, to increase software quality. Three examples for this can be found in Chapter 6.

In order to enable a natural and intuitive notation for these constraints, we developed a DSL. Additionally, we distinguish the constraints themselves into peephole and temporal constraints, the DSL for the two types differ slightly. In the following two sections, the grammar is displayed using a modified Backus-Naur Form.

Peephole Constraints

As the name already indicates, peephole constraints have only a limited field of view. They operate on single $\gamma \in \Gamma$ instances. This reduces their computation effort dramatically at cost of expressiveness. Their presence is still legit, as they are still sufficient for many scenarios. Peephole constraints follow an *implies* semantic, meaning that if the condition of a constraint holds, then the specified result is relevant. The grammar in Figure 3.3 describes the whole DSL for this type of constraints. One big advantage of our DSL and the resulting evaluation of the constraints is the compact notation that allows to specify whole categories of traces at once.

The top level element is a collection of sets of constraints labeled “start”. As we will see in Chapter 8, this is necessary to be able to have several constraint sets, for example a general one and a project specific one that are merged together. One remarkable subtlety in our grammar is the distinction between `expression` and `expression'`. It is needed to avoid infinite left recursion inside the parser that is generated for our language.

To specify a constraint, we start out with a unique id and a textual representation of the rationale. Through positioning information provided by “after” and “before” the constraint can be sorted in at an arbitrary position within the resulting constraint set. After that, we set up a condition which each incoming trace is checked against. If the result of that check is positive, then the specified result is applicable. If there is no result, then there have to be sub-constraints. These constraints are checked and their result is yielded.

When constructing a constraint set, the layout of it and the feedback generation for the developer is one key concern. One of the first ideas was to calculate the percentage of how much a trace matches a condition and to assume that the constraint with the most matching percentage should be considered. This idea was abandoned, since the statements generated by this approach would overexert the application developer and percentage evaluation of a condition is crucial. Our chosen approach arranges the constraints in such a fashion that they are first partitioned in the three basic component traits, see Chapter 4 for details. In case that no constraint with a condition evaluating to `true` is found, then all other constraints with result “PASS” can be considered as a potential solution and presented to the developer for the given trace. This strategy suits our needs best, since we derive the constraints directly from patterns. This approach leads us to constraints describing the permitted communications. Hence an intuitive way is to suggest these positive constraints in case a communication violates

```
start :=              constraintset?
constraintset :=      constraint+
constraint :=         "peephole−constraint" id "{" constraintBody "}"
constraintBody :=     after? before? rationale condition constraintset{1}
condition :=          "condition {" expression "}"
expression :=         "true" | "false" | term | "(" expression ")" expression′ |
                      "!" expression expression′
expression′ :=        "&&" expression expression′ | "||" expression expression′
term :=               field operation value | value operation function |
                      value operation field | field operation field |
                      function "(" params ")" operation value | function "(" params ")"
params :=             field ("," field)*
function :=           ([^"("]*)
value :=              "true" | "false" | "undefined" | """" ("\"" | [^"])* """" | "′" ("\′" | [^′])* "′" |
                      $([0 − 9] +  (.[0 − 9]+)? | "/" ("\/" | [^/])* "/"
field :=              id ("."id)*
operation :=          " == " | "! = " | " <= " | " >= " | " < " | " > " | " =∼ " | "! ∼ "
result :=             "FAIL_FINAL" | "PASS_FINAL" | "PASS" | "FAIL"
id :=                 "last" | "first" | ([a−zA−Z][a−zA−Z0−9_−]*)
before :=             "before" idseq
after :=              "after" idseq
iseq :=               id (","id)*
```

Figure 3.3.: Peephole Grammar

or can not be assigned to any constraint. Of course, negative constraints can still be formulated and are considered as results for communications. For further details about peephole constraint evaluation, see Chapter 8.4.

Temporal Constraints

Temporal constraints require a chronological order for the events they bear on. Timestamps provide this total order $\gamma_1 \prec_\epsilon \gamma_2 \mid \gamma_1,\ \gamma_2 \in \Gamma$ for all recorded communications Γ. The whole theory about temporal constraints is presented in Chapter 3.4. We can see in the tooling Chapter 8 that timestamp could also be omitted and replaced by a continuous id, since we program in a sequential manner and thus traces can not outpace others. However, if this property cannot be guaranteed, than the timestamps are vital for the analysis process. Furthermore, we also would have to wait until all traces have arrived and could then start checking our temporal constraints.

We do not establish any requirements on the order of the constraint within their set, since the rules are independent of each other and each of them is represented by an individual monitor, as we will see in the run-time verification Chapter 3.4.

Regarding the DSL dedicated to temporal constraints, some changes had to be made. The positioning keywords “before” and “after” are obsolete, since the order does not matter any more and have hence been removed. Additionally, the productions displayed in Figure 3.4 have been introduced to enable the formulation of temporal constraints. Last, it is no longer possible to nest constraints, because we have no semantic equivalent for temporal constraints. In terms of compact notation, the temporal constraint DSL is even more specialized than the one of for peephole constraints. Due to the usage of variables in the sequence section, as well as in the filter and link expressions, it is very flexible and avoids needless repetitions.

`sequence` :=	”sequence {” `id` (” << ” `id`)* ”}”
`filter` :=	”filter {” `expression` ”}”
`link` :=	”link {” `expression` ”}”

Figure 3.4.: Temporal Grammar

When specifying a constraint, we first have to label it with a unique id followed by a suitable rationale. After that, we specify the participant variables of the constraint, using the sequence section, e.g. $\alpha << \beta << \delta$. For each participant, a filter is set up to partition the space of all communications Γ into buckets, here $\Psi = \{\psi_\alpha, \psi_\beta, \psi_\delta\}$. During the observation of the application, each communication is sorted into the respective bucket. The special case is when one is assigned to the last bucket in this sequence ψ_δ. This triggers the evaluation of the “link” expression, which tries to find a feasible allocation for the used participant variables. If a solution is found, the monitor continues right away. Otherwise, an error is raised and the constraint fails for the whole run of the system under scrutiny, since it can never be healed in the aftermath.

3.4. Run-time Verification

Run-time verification is the task of checking whether an application satisfies a given set of constraints at run-time. This means, it solves the word problem [LC09], which is to decide a word’s membership in a formal language for candidates with a potentially infinite length, whereby only a finite prefix is given. Hence run-time verification can be compared with oracle-based testing, where an oracle specifies the valid runs of an application, this is done by the formal language in our case. In order to check a system, we derive a monitor for each constraint, which then observes the stream of traces. These monitors are usually not deployed in the final version of a system but during development time. Special case: If the monitors are deployed in the final product, the verification is testing the system forever.

In the context of this thesis, the goal of run-time verification is to check the validity of a given sequence of communication traces between different components of an applications architecture (3.1). This decision is done according to the constraints (3.1) that are established in Chapter 5. They are derived from the patterns identified in this thesis and thus intended to indicate whether the developer followed them or not.

Since the patterns described in Chapter 5 inherently identify the *allowed* communication traces for the participating components, it is obvious that the constraints also specify the *allowed* behavior of the system. Providing constraints for the prohibited behavior would result into a huge constraint set, which is simply *impractical* to be implemented. Hence we chose the approach to use the positive constraints to suggest enhancements for deviations in the behavior of the application.

Observation

The run of a system is defined by a possibly infinite sequence (word) of system states. This sequence is what can be used for further examination. In our particular setting, the system states are formed by the traces emitted by the application. Since observation can only be done as long as the developer is interacting with the application and thus triggering communication, the considered runs are always finite.

For the simple reason that it is often desirable to make a point about the time elapsed between traces, we froze the time of the occurrences of a communication to an integer property contained in our traces. As a result of that, constraints that claim hard real-time boundaries can be established as well.

Creating and destroying components inside the application causes communication traces to be emitted, too. Hence we can also maintain the current state of the Component Tree within our monitoring application. This enables us to visualize the current state as well as to inquire conditions on the Component Tree in our constraints. We provide an example for the usage of this component state information in Chapter 6.

Monitors

"A monitor is a device that reads a finite trace and yields a certain verdict." — [LC09]

Monitors as described in the definition by [LC09] are usually generated from formalized correctness properties. A common formalism is the Linear Temporal Logic (LTL) first proposed for the formal verification of computer programs by Amir Pnueli in 1977 [A P77].

This logic is suitable for run-time verification because it considers chronological state sequences. The operators provided by this logic are X, F, G, U and W with the semantics as defined in Table 3.1. Using these operators, propositions about states can be specified.

X ϕ	ϕ holds in the next state
F ϕ	ϕ holds finally in the future
G ϕ	ϕ holds globally
ϕ U ψ	ϕ holds until ψ holds
ϕ W ψ	ϕ holds until ψ holds unless ϕ holds globally

Table 3.1.: LTL semantics

After having defined the operators, we can distinguish finite prefixes of runs into *informative* and *uninformative*. Looking at Statement (3.1) we can say that only runs with at least a length of four traces are *informative*, because it specifies that the 4th state of a run should hold ϕ. All runs with less than four traces are considered *uninformative*.

$$XXX(\phi) \tag{3.1}$$

Getting back to monitors, according to [LC09] the following two characteristics have to be featured by them.

Impartiality
A monitor must not yield a particular verdict, if there is a continuation of a run leading to a different verdict.

Anticipation
At that point, when any continuation of a run leads to the same verdict, the monitor must return that verdict.

A verdict, in terms of monitors, is the result of an observed run, which is usually a member of the truth domain, see Definition (3.2).

$$\mathbb{B} = \{true, false\} \tag{3.2}$$
$$G(\beta \rightarrow F\alpha) \tag{3.3}$$

However, it is also possible that a monitor never yields a verdict for an expression. The LTL formula, shown in Equation (3.3), is an example for such a situation, as it takes the infinite character of a run into account, which can never be evaluated in practice. If we consider Expression (3.3), what should a monitor return in case when a β occurred and the application was terminated before an α was detected? In [LC09] a tweaked version of the LTL, the LTL$_3$ with a three-value semantic adding *inconclusive* to create the extended truth domain $\mathbb{B}'$ (3.4) is proposed to address this issue.

$$\mathbb{B}' = \{true, false, inconclusive\} \tag{3.4}$$

After having introduced the *inconclusive* verdict, LTL$_3$ is suitable to consider finite prefixes instead of infinite runs. With this in mind we can now have a look at the question: Which properties are actually *monitorable*? If we consider the previous Expression (3.3) again, we know that the verdict for this correctness property is *inconclusive* if an α is pending after the application has terminated. As a result of this observation, we say that (3.3) is not *monitorable*. In [LC09] an attempt is made to fix this problem by introducing two new verdicts *presumably true* and *presumably false* but this only shifts the problem and makes it still not applicable in our scenario. In order to overcome this gap, we decided to enable the developer and the monitoring tool, Chapter (8.1), to indicate the termination of a run by publishing a special *terminate* trace. As a result, we can also refer to the end of a run within our temporal constraints.

In the course of this thesis, we implement the capability to establish *happens before* relations, as a first step towards real-time verification in the user interface domain. An example is given in Formula (3.5). Here β must not hold before α, which is equivalent to α happens before β. Of course, we can also concatenate this expression to a chain of happens before relations with arbitrary length. Formula (3.6) shows a scenario where α needs to happen first and then β before γ.

$$\neg\beta\, U\, \alpha \tag{3.5}$$

$$\neg\gamma\, U\, (\neg\beta\, U\, \alpha) \tag{3.6}$$

Additionally, we provide a DSL that is more intuitive and user friendly to lower the hurdle for developers that are not yet familiar with LTL. Its grammar is listed in Chapter 3.3.

Application

To implement our constraint based run-time verification module, an architectural pattern called monitor-based run-time reflection or short run-time reflection (RR) was adapted and extended to meet our needs. Figure 3.5 is an adaption of that pattern presented in [LC09]. The original pattern consists of the logging, monitoring, diagnosis and mitigation component. To suit our scenario, we added the tracing and removed the mitigation component.

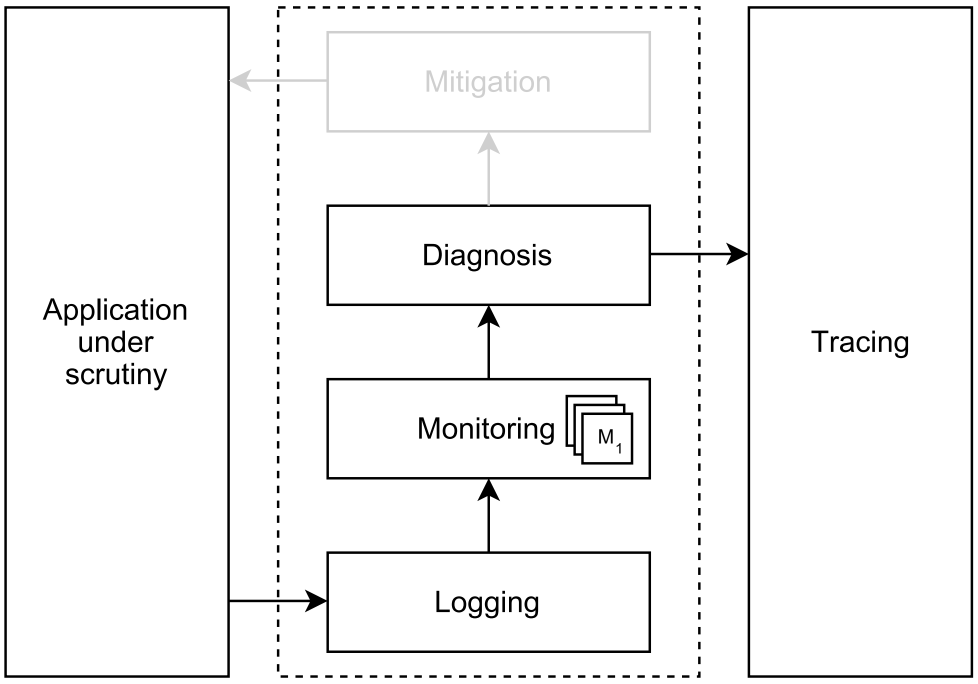


Figure 3.5.: Run-time Verification Scheme

Logging

The logging layer yields system events (traces) in a format supported by the monitoring layer. This sequence of emitted traces form the run of the system. We implemented this layer by hooking into API calls of the communication framework.

See Section 8.2 of the tooling Chapter 8 for more details. It is obvious that this technique only allows statements about the run until the current instant of time. That is why the range for our verdicts is $\mathbb{B}$.

Monitoring
The monitoring layer is built up by multiple monitors $M_1, M_2, ..., M_n$ that are derived from a number of formal specifications. Each of them produces verdicts based on the sequence of traces observed from the logging layer. It is desirable to do this in an incremental fashion, otherwise a verification during run-time is not feasible.

Diagnosis
To convert the output of the monitoring layer to a statement that can be read by the developer is the task of the diagnosis layer. It interprets the single verdicts and combines them with the given set of constraints. The result is then passed to the tracing user interface.

Mitigation
This layer is meant to reconfigure the system in case a failure occurs. As depicted in Figure 3.5, it is omitted because we don't want to interfere in the application but only observe it.

It is clear that monitors should influence the examined system as little as possible. That is why we equipped our implementation with online as well as offline monitoring abilities. To achieve this, we offer "post mortem" analysis, Section 8.4, via the ability to first direct an entire run to a file, which can then be loaded into the web interface, Section 8.1, for offline inspection. Even if it should be necessary to analyze the traces in real-time (online), the web interface can be viewed remotely as well as on the local machine. At a first glance this ability might seem to be a bit unsatisfactory. But as the whole analysis process is performed on the client side, the computation load can be easily outsourced to another machine leaving only the transmission of the traces on the machine with the system under scrutiny.

Which setting fits best has to be decided in accordance with the impact that is acceptable in the particular project.

4. Component Traits and Communication

An SPA is build by composing a large number of components. Each of these components plays its designated role and might even be instantiated several times. We discriminate the different roles into four categories or traits, as the behavior of a component is determined by the role it is playing.

Traits are very similar to the ones known from the Scala world [OSV10]. This means they can be stacked arbitrarily, enabling single components to indwell several traits at once. Choosing and grouping the right responsibilities for particular components is a hard task. That is why we give a guideline in Chapter 5, by providing elaborated and proven patterns for a variety of use cases. When we cut competences and assign them to components, we try not to mix up different responsibilities by putting them together into one component. Hence, having components that exhibit more than one trait might be a sign for a violation of the Logical Separation principle. The technical implementation of this is outlined in Chapter 8. However, we will already use this concept for further distinction in the constraints formulated in Chapter 5.

Note: When a component is acting as model and view at once, the properties of each type are combined.

4.1. View

View components shape the face of an application and are thus responsible for everything that is visible to the user.

But how exactly do the single components blend together to build up the user interface? We can simply imagine stacking components wielding the view trait on top of each other, starting at the root node, following the hierarchy of the Component Tree. The result of this traversal depicts the user interface in its entirety.

We denote components of this trait wit the `ComponentJS.marker.view` mark for further reference when setting up constraints.

Condition

The only points, where the component system touches the User Interface Toolkit (UI Toolkit) are the view components. These components are used to render everything, from the topmost dialogs of the application, down to layout components that combine several nested views into one comprehensive view.

Communication

Each view is tied to the underlying model via several bindings. Initialization values, which are usually not likely to change, are injected via the parameter binding (2.1). It can either be achieved by passing them into the constructor of the component —

considering that they are static and will never change — or by observing (see Table 2.2) designated model properties. An example for a static parameter might be the mode of a view, determining whether it is editable or not, whereas a dynamic parameter can be the orientation of the device displaying the application.

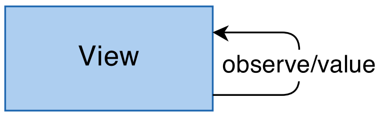


Figure 4.1.

The different states of the view, e.g. is a button enabled or not, are managed through the state binding (see Table 2.1). It is always implemented using model members, that are then observed, to ensure that the view always reflects the latest state of each user interface element.

For user interface events, the event binding (2.1) is used. For its implementation, two possibilities are conceivable. The first is to use the `publish` API method (2.2) to start an event bubbling up the Component Tree. The other option is to have designated model members, which are either observed by an underlying controller and/or model component. The purpose for this observation is explained in Chapter 4.3 and 4.2, respectively. The second alternative has to be preferred in most cases to stick to the principle of Factual Locality, considering that it makes the events only visible to observers.

At last, the view component has to be supplied with up-to-date data accomplished through the data binding (see Table 2.1), for depiction purpose. The data resides in the model component, see Chapter 4.2, making it obvious that the view can access it directly using the `value` API method (see Table 2.2), as well as watch it utilizing the `observe` API method.

Constraints

We set the constraint up as shown in Listing B.1 to assert the behavior explained in the communication section. At first, we make sure that the calls are performed on the component itself and that the component wields the view trait. After that, the core constraint is established. It limits the operations to `value` (2.2) and `observe` (2.2) calls.

4.2. Model

The model trait `ComponentJS.marker.model` holds all of the application's data necessary for the user to conduct his use cases. Most of the time, we use the term presentation model (2.3) due to the fact, that we deal with the user interface, which basically presents data to the user and lets him interact with it. We assign the contained data to four different categories:

- Domain specific application data
- UI state data

- UI parameter data
- Specialized event data

In order to improve intelligibility and to simplify debugging of an application, we define a specific naming convention. Hereby the data from the above categories is represented as outlined in the following micro grammar.

$$\texttt{type} := "data" \mid "state" \mid "param" \mid "event" \mid "cmd"$$
$$\texttt{identifier} := [a{-}zA{-}Z][a{-}zA{-}Z0{-}9\_{-}]^*$$
$$\texttt{name} := \texttt{type}\ ":"\ \texttt{identifier}$$

The `name` rule produces valid model labels and, when strictly followed, an application must not contain any model component that has differently termed members.

The prefixes “data”, “state” and “param” map one to one to the data categories. Both, “event” and “cmd” belong to the category of event data with the distinction that events are sent from overlying components towards the model, whereas commands arise from underlying components.

Condition

We have several possibilities to store data within an application: Use globally or locally scoped variables, member variables of component instances or outsource it to another storage layer. But when we want to make the data available within the Component Tree, which means that we can observe and manipulate it through component system’s API, we have to declare a model (see Table 2.2) and pull the required data into it.

In general, the model trait should be used whenever a component holds any kind of data, listed in the introduction of this chapter.

Communication

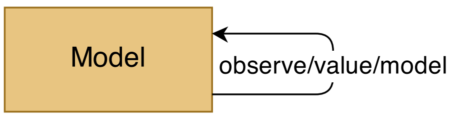


Figure 4.2.

First of all, the component has to perform the model definition by calling the `model` API method (2.2) onto itself. Afterwards, the model can observe its values to enforce its presentation logic (2.3) triggered by data/state changes or event occurrences. The presentation logic might cause changes for model values according to the dependencies between model members. This makes it vital for the component to do `value` API method (2.2) calls.

Constraints

As already seen in the constraint for the view trait (4.1), we enforce the naming convention in the constraint (B.2) for the model trait, as well. In addition, as inherent in the character of a model component, we of course have to permit `model` calls (2.2) onto the component itself.

4.3. Controller

A component with the trait `ComponentJS.marker.controller` is in most cases both interface to the rest of the Component Tree and coordinator for its sub-tree (2.3).

We can distinguish two perspectives when the controller acts in the interfacing role. In the one way, an underlying component sees the controller, whereas the sub-tree remains hidden behind it. This makes the controller act like a facade for the concealed components, from an ancestors point of view. Alternatively, from the child component's perspective, the controller is the gateway to the rest of the application. It can deliver information as well as expose sub-tree events to the underlying components.

In the coordinator role for its sub-tree, a controller can perform common ancestor communication to transmit data from one sibling component to another. This mechanism is described in detail in the pattern catalog in Chapter 5.8.

Condition

It is often advisable to hide a whole sub-tree from the remaining application when developing reusable or complex component formations. In these situations, the controller trait is the first choice. Of course, this is only possible, if the internal structure of such a component composition is not of particular interest for the instantiating components.

As already mentioned in the beginning, the controller is the interface for its sub-tree. This means that every communication from or into the sub-tree has to pass through the controller. Either via inherent hierarchy of the Component Tree or via methods directed to the controller. With that in mind, we can see the controller as an interface definition equally regulating the data going in and out of the sub-tree.

Communication

What the view trait, was for the visual representation and user interaction side, is the controller trait for the data representation and component interaction. One significant difference we already noticed when comparing Figure 4.3 to the figures of the other traits is the second component wielding the model trait. As we will see in the remainder of this section, the controller maintains a direct dependency on its child model component, rendering it nearly useless when used without.

First we want to have a closer look on the model-controller relation, as a base for further investigation of the tasks a controller has to carry out. We start with a question: Which data is exchanged between model and controller? Each controller has to deal with the four data categories as defined in the model trait. As we know, the general mechanism to read from and write data to model members is the `value` API method.

Hence these calls have to be performed by the controller upon the model component, in order to meet its presentation provisioning (2.3) duties.

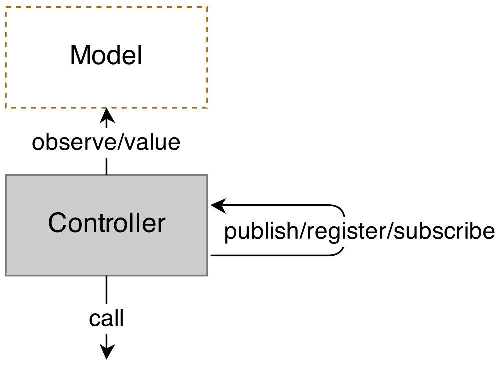


Figure 4.3.

These duties might also involve performing calls to the service facade to retrieve data, necessary for the other components within the sub-tree. In addition to the reading and writing of data, the controller also has to observe model properties to trigger its presentation actioning (2.3). The actioning can either be internal, meaning that one component raises an event and the controller delegates it to another sibling component or it can be an outgoing event that triggers an event to be published on the Component Tree. We notice, that it is also possible that a child component publishes an event instead of using model members. This happens when the particular component is not aware of the direct model the controller is observing. The last possible communication for a controller is that an overlying component wants to delegate or query it or its sub-tree, which is for example used in the Managing Widget pattern, see 5.5. That is why the controller might also call the `register` API method (see Table 2.2), targeting itself.

Constraints

Appendix B.3 shows the resulting constraints for the controller trait. All calls onto the controller itself, as described in the communication section, are permitted. Unfortunately, we can formulate the condition for the `observe` (2.2) and `value` (2.2) calls only to any model component that is a child of the controller component. Here, a more accurate term would be desirable for the future, see 9.2.

5. Communication Patterns & Constraints

The pattern catalog contains a brief explanation of each pattern, based on the pattern scheme of [Gam95]. We also link the patterns with their corresponding constraints, when new constraints are introduced by a particular pattern. For the "known uses" sections, we resort on existing applications, whose details can be found in the corresponding glossary entries.

The UI Toolkit Code Samples are HTML templates, that are parsed and populated using jQuery Markup, a plug-in for jQuery. Of course any other UI Toolkit could be substituted.

In this chapter, dashed elements stand for components provided by the environment of the particular pattern. The color of each component is aligned to the color schema in Chapter 4, along with the color used inside the component tree of the monitoring tool, Chapter 8. Arrows with filled heads are API calls whereas empty heads identify the components created by others. This notation is necessary, since the `create` API calls are always done on the creating component itself and would thus not provide any useful information.

In addition to the component types known from Chapter 4, a new dark grey colored stand-in type is introduced. Currently, these doubles can be replaced by only one other component of any type. In the future, with the concept of a shadow tree, which is an adaption of the eponymous paradigm proposed by the W3C in [W3C13], these surrogate components can be replaced with a whole sub-tree consisting of several components hidden inside one shadow component. This compromise guarantees that the patterns can also be applied after the introduction of a shadow tree in the near future.

5.1. Pattern 1: Lean Widget

In order to build a user interface (3.1), two domains have to be incorporated: The graphical and the component world. The graphical world is responsible for drawing the widgets, the component world, instead, handles the communication among them and their data objects.

Condition

This pattern is only indicated if the widget is *simple*. In this particular case, simple means that the amount of states and data that is required for the correct rendering of the widget consists only of a maximum of two elements.

Structure

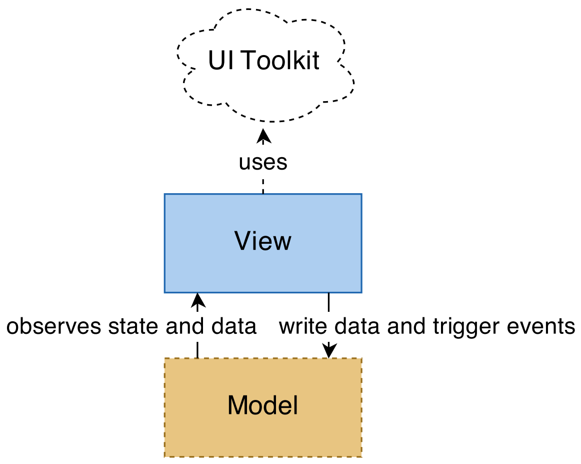


Figure 5.1.

The structure of this pattern consists of only one view component. Its collaborators are on the one hand the UI Toolkit, which is used to create a graphical representation, and on the other hand the model component, that is required by the view component to store its state, data and to trigger its events. Figure 5.1 outlines that model component and UI Toolkit are not part of the pattern itself and hence have to be provided by the components environment.

Communication

Communication paths for this component can be categorized into two sections: The data binding, see 2.1, and the event binding, see 2.1, domain. To achieve a bidirectional data binding, first, the observer pattern is applied to keep the displayed data in sync with the underlying model. Furthermore, the `value` mechanism is used to update the model with the latest data from the widget. The crucial part is to determine the data field of the underlying model, as the view needs to know where to store its data and which field of the model has to be observed. The same problem occurs for the event binding. To address this problem, a scoped property lookup (see Table 2.4) for the names of the fields is performed in order to enable value and event passing towards the model. Scoped, in this case, means that there is a dedicated version of the property, only visible for the particular instance of the view component. This property has to be provided by an underlying model, which is not part of this pattern.

Rationale

The pattern provides a connection between the UI Toolkit domain and the component system. It focuses on Loose Coupling between the view and the underlying structure to enable Headless Testing while simultaneously providing an abstraction of the deployed UI Toolkit for easy exchangeability.

Constraints

Allowed API methods for this pattern are `property` (2.2), `value` (2.2), `observe` (2.2) and `plug` (2.2). Hereby, the origin of these operations has to be the view component itself. See appendix B.4 for the corresponding DSL implementation.

Pros and Cons

Advantages

- The view component is highly reusable and can be instantiated several times across the application. This reduces the overall lines of code and improves maintainability.
- Reduces complexity of the Component Tree by replacing a full blown MVC triad with only one component, following the principle of Overall Simplicity.
- Less boilerplate code is necessary, as too simple components are omitted and do not need to be implemented.
- Enables Headless Testing by Loose Coupling of the view component with the underlying environment.

Disadvantage

- Although the number of components is reduced by this pattern, the granularity can still be too fine and condensing small widgets to a bigger component should be considered.
- The pattern tends to violate the principle of Factual Locality when the widget comprises of many states and pieces of data, since the data is stored somewhere outside the view component.

Alternatives

When it comes to widgets that are more intricate, i.e. they have to maintain several states and need not only one piece of data for the purpose of displaying their content, the Complex Widget (5.4) is the pattern of choice.

Code Sample

```
app.ui.lean_text_box = cs.clazz({
  mixin: [ cs.marker.view ],
   dynamics: {
    label: null
  },
  cons: function (label) {
    this.label = label
  },
  protos: {
```

```
10      render: function () {
11        var txt = $.markup('text-box', {
12          label: this.label
13        })
14        $('input[type=text]', txt).blur(function () {
15          var eventField = cs(this).property('blur')
16          cs(this).value(eventField, true)
17        })
18        $('input[type=text]', txt).change(function (event) {
19          var dataField = cs(this).property('data')
20          cs(this).value(dataField, event.target.value)
21        })
22        cs(this).observe({
23          name: cs(this).property('data'), spool: 'created',
24          func: function (ev, nVal) {
25            $('input[type=text]', txt).val(nVal)
26          }
27        })
28        cs(this).plug({
29          object: txt,
30          spool: 'created'
31        })
32      }
33    }
34  })
```

Listing 5.1: Basic text box widget

As a view is supposed to deal with the creation of user interface elements, the code displayed in Listing 5.1 handles the instantiation of the template HTML code in 11 and embeds it into the surrounding container via the socket (2.2) mechanism in line 28. The template (5.2) is populated with the member variables of the view component, that have been passed to its constructor. The used CSS classes are globally supplied and can be exchanged easily. For the communication of the view with its environment two kinds of bindings are necessary: The event binding, which is done in line 15 and the bidirectional data binding, that is linked up in line 19 and line 22 to 27, respectively.

```
<markup id="text-box">
    <li class="text-box item">
        {{label}} <input class="text-box" type="text"/>
    </li>
</markup>
```

Listing 5.2: Basic text box HTML template

Known Uses

ComponentJS Tracing
The button and text box widgets inside the toolbar of each tab follow this pattern.

Architecture Fundamentals
Depending on the complexity of each entry within the menu of this application, each could have been implemented using the Lean Widget pattern in combination with a shared model (5.2), providing the necessary data.

5.2. Pattern 2: Shared Model Component

Sometimes it is just exaggerated to have a model for a range of view components. In these cases a shared model component that contains the union of each single view's required data elements is a good solution.

Condition

When faced with a quite huge number of model-view dyads, with each model containing only a few members, it might be worth considering to collapse the models into one single comprehensive model. Another indicator are dependencies among the different models, where a common model leads to less unnecessary communication and a better manageable presentation logic. Separating these data into a shared model reduces communication overhead and improves the architecture principle of Exclusive Sovereignty, hence makes the application less vulnerable to inconsistencies.

Structure

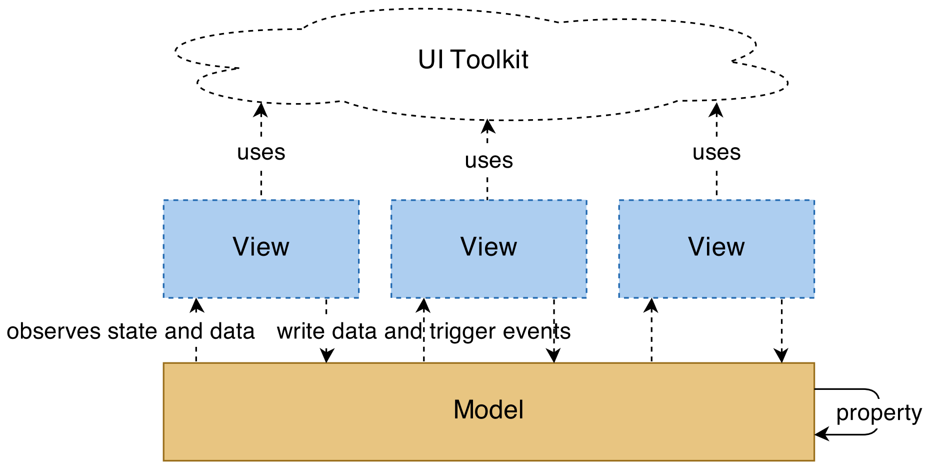


Figure 5.2.

As displayed in Figure 5.2, the pattern consists of one model component. We have two different peculiarities for this pattern. In one case, the different view components are aware of the names of the model members they want to access and can do this directly. In the other case, the views require a particular set of underlying model members and locate the ones designated for them using the property lookup (2.4) mechanism like explained in the Lean Widget pattern (5.1). Using the property lookup mechanism improves reusability. It is also an Inversion of Control (IoC) whereby the decision about the storage location of data is made by the model itself.

Communication

Each model member for the event binding (2.1), data binding (2.1) and parameter values (2.1) is stored in the model component. In both variations of this pattern, the values are accessed using the `value` (2.2) mechanism. An additional communication between views and model arises in the more generic case, it is used to look the address for the values up. This approach guarantees more reusability for the views but requires the model to maintain scoped properties (2.4) for all accessing components.

Rationale

Via providing a way to map data slots of a model seamlessly to the storage required by overlying components, this pattern offers the possibility to merge a bunch of tiny models into one model component. This reduces the implementation effort, the run-time overhead and the complexity of the application.

Constraints

Like outlined in appendix B.5, the model component has to be able to define the model for the data onto itself. Moreover, it has to provide scoped properties (2.2) to direct relying view components to the correct model members. This is essential, especially when the names of the model members required by the different views are ambiguous either by coincidence or because of the same view being instantiated multiple times.

Pros and Cons

Advantages

- Simplify several dedicated model components into one comprehensive component to reduce complexity, improving Overall Simplicity.
- Guarantees Exclusive Sovereignty over a set of coherent data and reduces redundancy.
- Supports views with a generic data interface and thus increases Constructional Reusability.

Disadvantage

- The pattern tends to become an anti-pattern. When pushed to the utmost, the model drifts towards the component tree's root, absorbing all models it passes through. This would result in one big omnipotent model, which breaks architecture principles like Factual Locality and Exclusive Sovereignty.

Alternatives

Due to the close relation to the Lean Widget pattern (5.1), the alternative for this pattern is the same. A complex widget (5.4) is advised, when the number of model members is too big and should thus be separated logically into different components.

Code Sample

```
 1 app.ui.shared_model = cs.clazz({
 2   mixin: [ cs.marker.model ],
 3   protos: {
 4     create: function () {
 5       cs(this).model({
 6         'data:items': { value: [], valid: '[{ label?: string,' +
 7           'id: string, data?:string, blur?:string }*]'}
 8       })
 9 
10       cs(this).observe({
11         name: 'data:items', spool: 'created',
12         func: function (ev, items) {
13           _.each(items, function (item) {
14             if (item.data)
15               cs(this).property({
16                 name: 'data', scope: 'view/' + item.id, value: item.data
17               })
18             if (item.blur)
19               cs(this).property({
20                 name: 'blur', scope: 'view/' + item.id, value: item.blur
21               })
22           })
23         }
24       })
25     }
26   }
27 })
```

Listing 5.3: Shared model for several basic text boxes 5.1

The shared model component provides a model member to store a set of configurations for its views, see line 5 in Listing 5.3. According to this configuration, scoped properties are provided in line 12 for each if them. Now comes a semantic variation point. The model members, identified by the names contained in the scoped properties do not necessarily have to reside in the shared model. They can also be provided from an underlying model as it is done in this code sample. In this case, the model is rather a router, than a dedicated model. Of course, it still has to hold the configuration information for each element, itself.

Known Uses

ComponentJS Tracing
The toolbar component provides a comprehensive model for all embedded controls.

cloudTeX
Header component and sharing component both access one common model to determine the current user.

5.3. Pattern 3: Sub-tree Information Delivery

Inherent with the hierarchical character of a Component Tree is the possibility to split it into individual sub-trees (2.3). This and the fact that data is often shared among these sub-trees, leads us to the Sub-tree Information Delivery pattern.

A quick example: Imagine an application that has two visual components, a selectable list of employees and a view to display the details of the selected employee. Of course, according to hierarchical decomposition (2.1), these two views have a common part in the Component Tree and one that is unique for each of them. That fork component, dividing the Component Tree into the common and the unique part is the point where the mediation between the list and the details view has to be located. What possibilities we have at hand and when to prefer which is the goal of this pattern.

Condition

The example in the introduction already highlights the problem that makes this pattern indispensable: Delivering data that is of interest for one or more sub-trees (2.3), outgoing from one (common) parent component. Two application scenarios are conceivable for this pattern. In the first one, a component P holds the Exclusive Sovereignty over a set of data. This data should be available for a variable amount of sub-trees. The second scenario contains the same component P that has at least two sub-trees, with one of them providing information to P that is afterwards transmitted to the other child components in a broadcast fashion. We notice, the information has to be distributed to the sub-trees of component P, in both cases. Hence, this need can be used as a clear indicator for the Sub-tree Information Delivery pattern.

Structure

The structure outlined in Figure 5.3 already implies the two variations of this pattern. One way is to use a component wielding the controller trait (4.3), whereas the other way is to use a model component (4.2). Furthermore, the illustrated component constellation is independent from the amount of components on the path between the controller respectively model to the root component of the sub-tree. This autonomy improves the architecture in terms of Loose Coupling since the only constraint to the participating trees is them being *real* sub-trees of the data providing component.

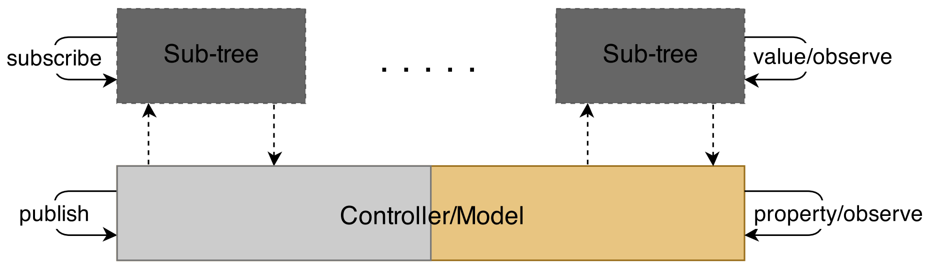


Figure 5.3.

Communication

At first we need to have a look at the relation between events and data. On the one hand, we can piggyback data on an event (2.2). On the other hand, we can use boolean model members to represent events by toggling their value. Hence we can say that events and data, i.e. model members, are equally powerful, whereby the data based approach has the downside that it needs a model (2.2) to provide event members. With this in mind, we can proceed to describe and distinguish the two variants.

First, we take a look at the option on the left side of Figure 5.3. Every sub-tree, interested in the particular event subscribes to it in spreading mode (2.4). The event is then fired by the controller upon occurring changes. The procedure in detail: An event is dropped at the controller component, which is then traveling towards the root of the Component Tree and triggering any observers it passes by.

The second option, shown on the right hand side of Figure 5.3, also implements the observer pattern (2.3), but now we use the event binding (2.1) approach. Here, the interested sub-trees can observe and modify the model members they are supplied with. To tweak this association we can use the property mechanism to lookup (see Table 2.4) the model members' addresses like already introduced in the Lean Widget pattern (5.1).

The main differences between the both approaches is the location where the observation is taking place as well as the usage of an intermediate component — the model — in the one case, whereas the other case copes without this extra component. At a first glance, the second variant seems to lose in a direct comparison. Thus we have to take a closer look on the disadvantages and advantages of each approach.

Spreading event

Advantages

- Guarantees that the sub-trees cannot modify the data, since they have no write access ensuring Exclusive Sovereignty.
- It is very good for communication triggered by an event outside the component system.
- Fulfills the User Interface Component Architecture.

Disadvantages

- Data is stored at several locations within the Component Tree, leading to redundancy that in turn can cause update anomalies.
- A spreading event travels through the whole sub-tree, causing additional communication overhead.
- Components that are created after an event occurred may be notified too late.

Model member

Advantages

- Each component can pull the required data right when it needs it.
- The data is redundancy free through Exclusive Sovereignty, in combination with Factual Locality.
- Globally scoped data can be easily accessed and observed, i.e. current logged in user, authorization information etc.

Disadvantages

- Disbands Loose Coupling and Strong Cohesion by introducing cross component dependencies.
- Breaks the User Interface Component Architecture by externalizing parts of the presentation.
- Difficult to understand due to long range dependencies.
- Unimpeded write access may break Exclusive Sovereignty.

With this assessment and the two implementation strategies at hand, we can now postulate recommendations. Spreading events should be used when the transferred information really is an event or data where only changes are of interest. For example, a window resize or orientation change (for mobile devices) event that has to be delivered to multiple components within the Component Tree.

However, the model member approach has to be preferred when the distributed information is data and read as well as write operations are conceivable. The model component, of course, can take care of validation and data logic by observing the different data operations (2.2).

The pattern supplies us with a universal mechanism of delivering data to child components but does not claim to define where the events or data originate any closer. We will sketch a possible application in the code sample, Listing 5.4.

Rationale

The Component Tree allows us to have communication bubbling up from each component. On its way, an event can notify observers and trigger actions as well as deliver data. But sometimes, we are faced with the necessity of sending data from one component downwards the hierarchy to child components or whole sub-trees. At this point, this pattern steps in by providing a clean and sophisticated way to relay information.

Constraints

The corresponding constraints are outlined in Listing B.6. In both scenarios we determine the receiver to wield the controller trait (4.3), as the controller is the interface of model and view towards the remaining Component Tree. We can see that the events have to be published and observed in spreading mode to enable the correct communication direction. The remaining communication is similar to the one of the Rich Widget pattern, see Chapter 5.4.

Pros and Cons

Advantages

- Abstracts from the event, which triggers the information exchange.
- Is a specialized solution for two separate information categories.
- Maximizes Factual Locality and Exclusive Sovereignty as far as practicable.

Disadvantage

- Debugging the components involved in this pattern can be quite challenging, since the dependencies can stretch over the whole Component Tree and hence might be hard to detect.
- Destroys Strong Cohesion by pulling apart what belongs together. But in some cases this can be the only practical solution.

Both disadvantages can be mitigated with a good component interface documentation.

Alternatives

An alternative might be the Common Ancestor Communication pattern, see 5.8. It establishes the connection between two components located in different sub-trees. Besides that, another option could be the Shared Model Component pattern, see 5.2. But it can only be used in a limited extent, since a whole component and not only an individual model member is shared among components.

Code Sample

```
app.ui.receiver = cs.clazz({
  protos: {
    show: function () {
      cs(this).subscribe({
```

```
 5           name: 'event:window-resized', spool: 'visible',
 6           capturing: false, bubbling: false,
 7           spreading: true,
 8           func: function () {
 9               this.refreshView()
10           }
11       })
12     }
13   }
14 })
15 
16 app.ui.relay = cs.clazz({
17   dynamics: {
18     timer: null
19   },
20   protos: {
21     show: function () {
22       window.onresize = function () {
23         if (this.timer !== null)
24             clearTimeout(this.timer)
25         this.timer = setTimeout(function () {
26           cs(this).publish({
27             name: 'event:window-resized',
28             capturing: false, bubbling: false,
29             spreading: true
30           })
31         }, 1000)
32       }
33     }
34   }
35 })
```

Listing 5.4: A window resize event propagated to sub-trees

Listing 5.4 shows the example of the spreading event approach that is used to propagate window resize events to interested components after a threshold of one second. The important part is in line 7 and 29. Here the event mechanism is switched to spreading mode and thus enables a communication downwards the hierarchy. The observer pattern is implemented as usual through publish and subscribe in line 26 and 4, respectively.

Known Uses

Midas

Whenever a store is selected in the designated list view, it is saved to a shared model property as many other parts of the application display information related to it.

cloudTeX
The currently logged in user is stored in a shared model member to make it available for several equal sub-trees.

MagicSearchTool
The registration form is subdivided into several components that host input fields. To clear all of these input fields, a spreading event is published on the root component of the form.

5.4. Pattern 4: Rich Widget

In many cases it is desirable to provide a dedicated model for a single widget in order to encapsulate the complexity of the presentation logic (see 2.3). Furthermore, a central element that holds the Exclusive Sovereignty over its data can be useful, too. This is, where the Rich Widget pattern kicks in.

Condition

It is appropriate to choose the Rich Widget pattern, if the presentation of a particular widget comprises a significant amount of states and data. Furthermore, extensive presentation logic suggests the use of this pattern for a better logical separation.

Structure

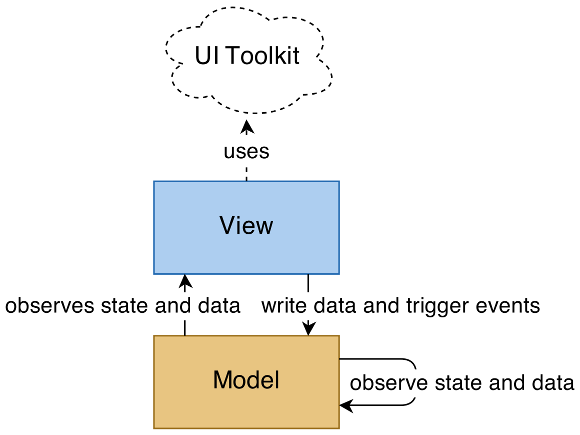


Figure 5.4.

The structural layout of this pattern is an enhanced version of the Lean Widget (5.1). The UI Toolkit is also used to create the graphical representation, here. But with the inherent complexity of the widget, leading to a significant amount of stored data and states, a dedicated model component is the solution of choice. Figure 5.4 shows the composition, where the model has to be implemented as a part of the whole pattern.

Communication

Due to the similarity with the Lean Widget (5.1) pattern, the transmission channels for the view component to ensure the bidirectional data (2.1) and the event binding (2.1) stay the same. Moreover, the resolution of the model field names using scoped properties is no longer necessary, because view and model component are aware of each other. The required fields of the view can be interpreted as an interface that is presumed by the view component. So the view component can be reused if a model component implements this interface, i.e. it provides the necessary fields.

The model component, however, requires new communication paths to perform its presentation logic (2.3). The observer pattern (2.3) is deployed to trigger the presentation logic and necessary values of the model members can be retrieved using the `value` (2.2) method. Since the presentation logic is only allowed to access the data stored in its hosting model, these methods are only allowed onto the model component itself. Any operation involving data of components outside this pattern has to be carried out by the controller like described in the pattern in Chapter 5.8.

Rationale

Keeping all data and its logic together in one place reduces complexity and improves maintainability and understandability of the source code. Thus, the implementation of a dedicated model component for a rather extensive widget is advised. Of course Headless Testing is well supported since the view can be omitted and testing is performed on the sole model component.

Constraints

The pattern allows the same API calls for the view component like the Lean View pattern (5.1) does. Additionally, constraints for the model component have to be deployed. The model is permitted to observe (2.2) itself to perform presentation logic, as well as modifying its content via the `value` (2.2) method. See appendix B.7 for the corresponding DSL implementation.

Pros and Cons

Advantages

- Gives a clean separation of model and view component.
- Provides an abstraction for the view component: interaction is done via the presentation model.
- Clusters methods and data logically, which improves maintainability and also ensures a good Logical Separation and Exclusive Sovereignty.

Disadvantages

- Needs additional code, as there are two classes to implement.
- The model always has to be bundled with the view, hence is less reusable.

Alternatives

It is worth considering the Lean Widget pattern as an alternative, when the model seems to be an over-engineered solution. An indicator is a model consisting only of presentation fields.

The pattern does not limit the number of widgets that are actually implemented in the view. Thus, a slight variant leads to group several optically separated widgets into one single view component with a grouped model.

Code Sample

```
 1 app.ui.basic_text_box = cs.clazz({
 2   mixin: [ cs.marker.view ],
 3   protos: {
 4     render: function () {
 5       var txt = $.markup('text-box')
 6       $('input[type=text]', txt).keyup(function (event) {
 7         cs(this).value('event:keyup', event.keyCode)
 8       })
 9       $('input[type=text]', txt).change(function (event) {
10         cs(this).value('data:filter', event.target.value)
11       })
12       cs(this).observe({
13         name: 'data:filter', spool: 'created',
14         func: function (ev, nVal) {
15           $('input[type=text]', txt).val(nVal)
16         }
17       })
18       cs(this).plug({
19         object: txt,
20         spool: 'created'
21       })
22     }
23   }
24 })
```

Listing 5.5: Complex widget View component

The view component in Listing 5.5 creates and embeds the graphical representation the same way it is performed in the Lean Widget pattern (5.1). The actual difference lies in the fashion event and data binding is implemented. The calls in lines 7, 10 and 12 no longer use the dynamic resolution technique as they are directly wired to the model members.

```
app.ui.basic_text_model = cs.clazz({
  mixin: [ cs.marker.model ],
  protos: {
    create: function () {
      cs(this).model({
        'event:keyup': { value: -1, autoreset: true },
 7         'data:filter': { value: ''                   }
 8       })
 9       cs(this).observe({
10         name: 'event:keyup', spool: 'created',
11         func: function (ev, nVal) {
12           if (nVal === 27 /* ESCAPE */)
13             cs(this).value('data:filter', '')
14         }
15       })
16     }
17   }
18 })
```

Listing 5.6: Complex widget Model component

Listing 5.6 shows the implementation of the model component. It defines the model itself in line 5 with the necessary members “event:keyup” and “data:filter”. Additionally, it starts to observe the “event:keyup” member in line 9 and performs the presentation logic when triggered by the observer (line 12). In this case, when the Escape key was hit the value of “data:filter” is set to an empty string - the filter is cleared.

This is only a minimal running example and should not be considered as a sterling example. Usually, the view is more complex and thus the model may consist of five or more members.

Known Uses

ComponentJS Tracing
The grid component comprises many states and needs extensive data to display its content.

Midas
The chart visualization is also one big widget, since it is too complex and too specific to break it into smaller components.

5.5. Pattern 5: Managing Widget

Widgets that are responsible for the layout of an application often have inherent logic to decide how the different components are arranged. Furthermore, the requirements for the layout dictate that some components can only be displayed mutual exclusively.

Condition

This pattern perfectly fits the needs of an intelligent layout widget. Is the arranging of containing components computed using several state values or are constraints tightened to the states of the managed components, this pattern is recommended. A contraindication for this pattern is the presence of only one managed component, then additional nesting inside a layout component just increases complexity.

Structure

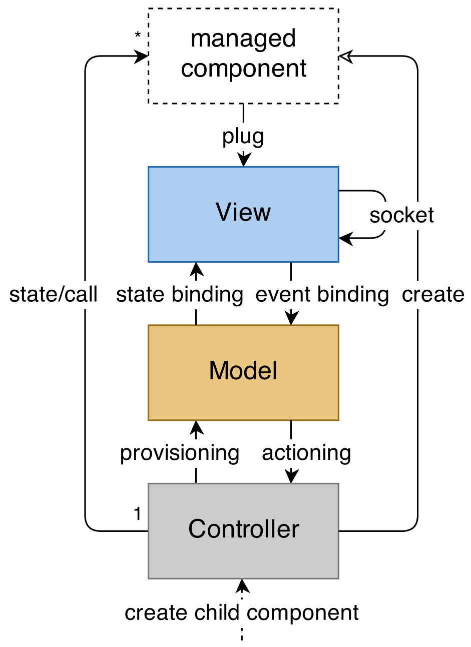


Figure 5.5.

Figure 5.5 shows the structure of this pattern. It consists of the classic MVC Triad (2.3) and the dashed component at the top end suggesting the potentially managed components. As the arrow at the bottom implies, this widget has to be called from an underlying component that is aware of the single components included in the layout.

Communication

We use delegation via the *create child component* service to shift the responsibility to manage the components from a composing component to the layout widget. After the creation request was issued, the controller instantiates the supplied component class and initializes it with the information given in the request. It also has to disable the auto-increase mechanism in order to control the component's state independently. After this setup procedure has finished, the state of all *managed components* is controlled by the widget's controller according to actioning triggered by user events or external state changes. Each newly created component has to plug itself into the underlying view, with one constraint: It must only perform plug operations, when it is in the `materialized` state, since the view prepares the sockets in that state, see line 48 in Listing 5.7.

Rationale

Imagine a layout component **L** and an underlying component **U**. **U** plans to embed several components inside the layout provided by **L**. One way would be to issue `create` calls directly from **U** upon **L** to instantiate the necessary components. If presentation logic (2.3) is necessary for the layout, e.g. a tab layout, where only one tab can be visible at a time, then component **U** has to take care of the states of each created component, although it does not know how the layout is build up. Hence the best practice is to shift that responsibility to component **L** which creates child components and is able to manage them, as a result.

Constraints

The constraints resulting of this pattern are written down in appendix B.8. As the behavior of the model and view component does not deviate from the one implied by their respective trait, no additional constraints have to be set up for them. The constraints for the controller can be condensed to one single condition. It ensures that the required calls `call`, `state` and `create` are only performed on components that are three steps away from it.

Pros and Cons

Advantages

- High Constructional Reusability for pattern instances.
- Precise Logical Separation along with Structural Modularity.
- Support of synchronization between the Component Tree and the DOM tree, due to direct state management of embedded components.

Disadvantage

- Additional components may be introduced although the encapsulated logic is not that complex.
- An underlying component is always needed, even if the widget is only instantiated once.

Alternatives

Should the layout widget not need any logic at all, because it provides a static layout, the Non-Managing Widget (5.6) is a better choice. Another alternative is the Rich Widget pattern (5.4). It should be used, when no generic layout is present.

Code Sample

```
app.ui.managing.widget.controller = cs.clazz({
 protos: {
  create: function () {
   cs(this).register({
    name :'createManaged', spool: 'created',
    func: function (clazz, cfg) {
     var newCmp = cs(this, 'model/view').create(cfg.name, clazz)
     newCmp.call('init', cfg)
     newCmp.property('ComponentJS:state-auto-increase', false)
     cs(this, 'model').value('state:managed-cmps').push(cfg.name)
    }
   })
  },
  show: function () {
   cs(this).observe({
    name: 'event:next', spool: 'created',
    func: function () {
     var all = cs(this).value('state:managed-cmps')
     var cur = cs(this).value('state:current-cmp')
     var newCur = cur + 1 % all
     cs(this, 'model/view' + all[cur]).state('materialized')
     cs(this, 'model/view' + all[newCur]).state('visible')
     cs(this).value('state:current-cmp', newCur)
    }
   })
  }
 }
})

app.ui.managing.widget.model = cs.clazz({
 protos: {
  create: function () {
   cs(this).model({
    'state:managed-cmps': { value: [],      valid:'[object*]' },
    'state:current-cmp' : { value: 0,       valid:'number'    },
    'event:next'        : { value: 'false', valid:'boolean',
                            autoreset: true }
   })
  }
 }
})

app.ui.managing.widget.view = cs.clazz({
 protos: {
  render: function () {
   this.ui = $.markup('pager')

   cs(this).socket({
    ctx: $('#content', this.ui),
    spool: 'created'
   })
  },
  show: function () {
   cs(this).plug({ object: this.ui, spool: 'visible' })
  }
  }
})
```

Listing 5.7: Managing Widget

On creation of the widget, the delegation service is registered in line 5, the model defines members for the managed components and the event binding and the view creates a socket in 48 for the managed components. After the setup, the actioning starting in line 15 deals with the layout logic and handles the managed components like displayed in line 21. The code responsible for triggering the "event:next" is omitted for sake of simplicity. A nice addition would be to set `store` to `true` for "state:current-cmp" for a smoother user experience.

Known Uses

ComponentJS Tracing
: Tab container widget of the whole application, as well as the tab container widget inside the peephole constraints tab.

P1
: Page container widget in the mobile perspective supporting a header, footer and multiple pages.

5.6. Pattern 6: Non-Managing Widget

A non-managing widget is the opposite of pattern (5.5). It contains no logic and does not interact with its child components at all. In its simplest form, it groups different components to one layout, by providing the necessary sockets.

Condition

Embedding multiple widgets into one static layout can be performed in several ways. The approach of choice depends on the amount of presentation logic (2.3) inherent to the layout. When dealing with no logic at all, i.e. only the visual structure, a dedicated view component that serves as an additional layer between the actual model-controller dyad and the particular widgets serves best.

Structure

As shown in Figure 5.6 the central element is of the stand-in type already mentioned in the introduction of this chapter. It can thus either be a single view component or a collection of components, collapsed to one node, representing a shadow tree (5). All components are instantiated by the underlying controller component and hence known by name. The view component, responsible for the static layout, does not know anything about the structure surrounding it. This stands in contrast to the Managing Widget pattern (5.5), where the view knows its direct child components.

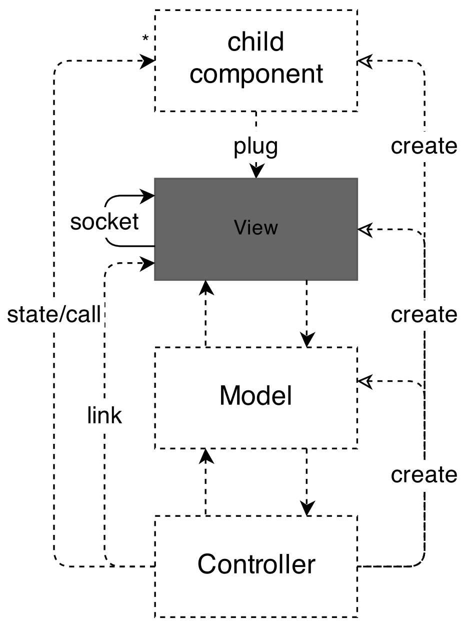


Figure 5.6.

Communication

The view component or shadow component has to provide sockets for all participating widgets and hence has to call the `socket` method (2.2) several times upon itself. Afterwards, the sockets are ready to be linked to the particular child components. This is optional and only necessary, if the child components should be plugged into dedicated sockets. Furthermore, the child components have to be synchronized with the view, concerning the state when they plug themselves into the view. This is usually after the view is in the `materialized` state. To make a child component, the controller creates it first, initializes it using a call and manages its state from then on.

Due to the fact that the creation and state control is performed by another component outside of the scope of this pattern, it is labeled the Non-managing Widget pattern.

Rationale

It is not always necessary to have a full blown MVC triad (2.3) taking care of a simple layout job. To reduce complexity, the model and controller components are omitted, boiling the MVC triad down to a single view component.

Constraints

Like we can see in Listing B.9, building the constraint for this pattern is rather simple. We allow `socket` API calls and make sure that they are targeted onto the view itself. Optionally, we allow controller components to create socket links onto view components to do the correct wiring of child components into the view's layout.

Pros and Cons

Advantages

- Standard layouts have to be implemented only once and are then available for different uses throughout the whole application.
- No complication by introducing unnecessary components, keeping the application as slim as possible.
- Widgets that are using the generic `socket` mechanism (2.2) can be positioned correctly using the `link` feature (2.2).

Disadvantage

- This pattern might often look like the best choice, but introducing additional model and controller components with growing complexity in the aftermath is very difficult.

Alternatives

With increasing complexity of the presentation logic (2.3), a separate model-controller dyad is imposed. The pattern for this case is the Managing Widget pattern (5.5). It unites a generic layout with advanced presentation logic. The Rich Widget pattern (5.4) is an alternative, if the generality of the layout it not present.

Code Sample

```
app.ui.child-component = cs.clazz({
  mixin: [ cs.marker.view ],
  protos: {
    render: function () {
      cs(this).plug({
        object: $.markup('child-markup'),
        spool: 'materialized'
      })
    }
  }
})

app.ui.non-managing-widget = cs.clazz({
  mixin: [ cs.marker.view ],
  dynamics: {
    content: $.markup('three-column-layout')
  },
  protos: {
    render: function () {
      var self = this
      cs(self).socket({
```

```
22         name: 'column1', type: 'jquery',
23         ctx: $('#column1', self.content)
24       })
25       cs(self).socket({
26         name: 'column2', type: 'jquery',
27         ctx: $('#column2', self.content)
28       })
29       cs(self).socket({
30         name: 'column3', type: 'jquery',
31         ctx: $('#column3', self.content)
32       })
33     }
34   }
35 })
36 
37 app.ui.ctrl = cs.clazz({
38   mixin: [ cs.marker.controller ],
39   protos: {
40     create: function () {
41       cs(this).create('view/child1',
42         app.ui.non-managing-widget,
43         app.ui.child-component
44       )
45     },
46     render: function () {
47       var self = this
48       cs(self, 'view').link({
49         target: cs(self, 'view'),
50         socket: 'column1',
51         scope: 'child1'
52       })
53     },
54     show: function () {
55       cs(this, 'view/child1').state('visible')
56     }
57   }
58 })
```

Listing 5.8: Non-Managing Widget

Listing 5.8 shows a small example, how the non-managing widget can be integrated into a Component Tree. First, the participating components have to be created by the controller. This is done in line 41. Now, in this particular instance, we choose a three column layout which is loaded in line 16 and looks like depicted in Listing 5.10 with its style in Listing 5.9. After that, upon entering the `materialized` state through the render method, the non-managing widget defines three sockets, see lines 21, 25 and 29. For one of these sockets, a link is created (2.2) in line 48. It is important that the name of this link is "default" since the view component plugs itself into the next "default" socket it can find, see line 5. This abstraction makes the child component independent from the underlying view component. Line 55 shows, that the capability of managing the child component lies in the controller component.

```css
.column { width:50px; height:90px; float:left; }
```

Listing 5.9: Non-Managing Widget Markup (CSS)

```html
<markup id="three-column-layout">
  <div>
    <div id="column1" class="column"></div>
    <div id="column2" class="column"></div>
    <div id="column3" class="column"></div>
  </div>
</markup>
```

Listing 5.10: Non-Managing Widget Markup (HTML)

Known Uses

ComponentJS Tracing
: A view, providing a box that is capable of having several nested components arranged horizontally.

Midas
: We have two components that are usually the only ones in their parent layout. Now they should be arranged side by side in a common perspective. Therefore, correct links have to be set up to coordinate their positioning.

5.7. Pattern 7: Externalized Presentation Logic

Input validation is an important part of a user interface. It makes the work-flow more convenient for the user and depicts problems clearly, right where they occur. However, from a developers point of view, input validation is a good chance to save unnecessary server communication, if performed on the client-side. Of course, we cannot ignore that for some consistency checks, the comprehensive knowledge of the back-end is indispensable. This pattern outlines how client-side validation of any kind of user input, for example text, selections etc. can be carried out.

Condition

Like mentioned in the introduction, this pattern focuses on client-side validation only. We can divide user input into two validation categories. For one category, validation can be performed by the presentation logic (2.3) in the model, holding the information to be validated. Whereas input that belongs to the other category needs additional data to check its validity, the latter is a clear indicator for this pattern. The result is the necessity to delegate the validation to another component that has all information at hand to decide whether the input is valid or not, which leads us to an externalized presentation logic (2.3).

Structure

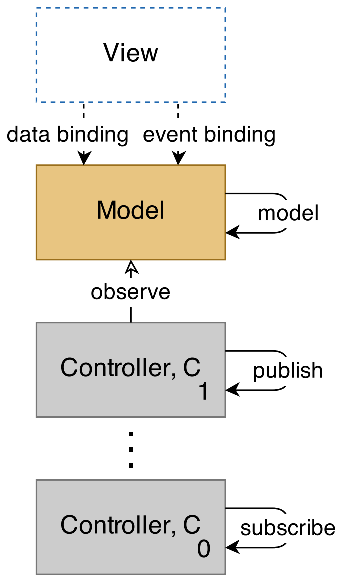


Figure 5.7.

Usually, we would expect that this pattern requires at least one component capturing user input, i.e. a component wielding the view trait (4.1). But we want to abstract from that due to the fact that it is sufficient to have a model component that reflects any user interaction via its model members, given the event (2.1) and data binding (2.1) has been implemented meticulously. This leads us to a model component with its corresponding controller, since the controller is the one allowed to communicate with the rest of the Component Tree. Another controller component located somewhere closer to the root of the Component Tree is providing the concrete validation functionality. Figure 5.7 shows this component constellation.

Communication

The first step of the communication is to let the model component define the presentation model (2.3) upon itself and to wire the data and event binding up. So far, the communication does not deviate from the one for the Rich Widget pattern. As the dashed line of the view component indicates, this component is not part of the pattern but for sake of simplicity we use it as minimal example. Further, it is also conceivable to replace it with any other component that is allowed to access a model, e.g. can our model component be part of the Shared Model Component (5.2) or the model based derivative of the Sub-tree Information Delivery pattern (5.3).

At this point, we have two options at hand. Which one to prefer depends on the validation process of the controller C_0. The first possibility is the interception of set operations on the particular model member by observing them. Afterwards, the changes can be propagated to the remotely located controller, which in turn can instantly return the validation result for them. It thus has a chance to prevent values from being set or can modify them beforehand. This is only possible, if the validation can be performed synchronously.

The second option is to have two independent model members, one for potentially invalid data and one for valid data. The former, transient data must only be used by the validating component, whereas the latter can be observed by each component that relies on valid data. We need this, when the validation is done asynchronously, which tends to be the common case in the user interface domain.

Now the different observers have to be set up. Controller C_1 observes (2.2) the model member containing the value candidate. Furthermore, a second observer on C_0 is waiting for validation requests to pass by. After that, when any component is trying to change the value of a model member, C_1 publishes an event containing the value candidate and optionally the current value. This event is caught by C_0 and its content is validated.

So far both approaches are the same, but they differ in the way the result is translated back into the model. In the synchronous case, the controller C_0 simply returns the validation result. On the other side, in the asynchronous case, the controller sets the value using a callback it has received with the validation request. After that, the model contains valid data in both cases.

Rationale

In many cases, user input can be validated within the model that is tied to the view using event and data binding. As an example, whether a given piece of information is a valid number, string or matches any kind of regular expression can be decided directly on the data. But whenever a parent component of a model is required to validate the integrity of that input, we talk about externalization of the presentation logic. The model component intercepts write operations and exposes the possibility to prevent the value from being set or at least to modify the value before it is permanent.

Constraints

The constraints for the participating components have already been set up in their respective trait chapters. See appendix, Listing B.3 for the constraints relevant for the controllers and Listing B.2 for the model.

Pros and Cons

Advantages

- Delegates the decision making process to the component where the necessary information is available and thus improves the Logical Separation.
- Flexible handling of synchronous as well as asynchronous validation logic.

Disadvantage

- Grows complexity by introducing mid to long range dependencies.

Alternatives

The Common Ancestor Communication pattern (5.8) could serve as an alternative, although it has a slightly different semantic. In the pattern presented in this chapter, the mechanism to control values of model members is offered by an underlying component, but whether it is used for validation or not depends on a component that is located upwards the hierarchy, i.e. it depends on the structure of the application. In the Common Ancestor Communication pattern, the component itself would directly request the validation and therefore strictly enforce it. In contrast, we can say that here an Inversion of Control takes place.

Code Sample

```
app.ui.model = cs.clazz({
  mixin: [ cs.marker.model ],
  protos: {
    create: function () {
      cs(this).model({
        'state:active-tab': { value: 0, valid: 'number' },
        //Addition for asynchronous scenario
        'state:proposed-active-tab': { value: 0, valid: 'number' }
      })
    }
  }
})

app.ui.controller1 = cs.clazz({
  mixin: [ cs.marker.controller ],
  protos: {
    create: function () {
      cs(this).create('model', app.ui.model)
    },
    show: function () {
      var self = this
      //Synchronous scenario
      cs(self, 'model').observe({
        name: 'state:active-tab', operation: 'set',
        func: function (ev, nVal, oVal) {
          var validated = cs(self).publish({
            name: 'validate-active-tab', args: [ nVal, oVal ],
            directresult: true
          })
          ev.result(validated)
        }
      })
      //Asynchronous scenario
      cs(self, 'model').observe({
        name: 'state:proposed-active-tab',
36          func: function (ev, nVal, oVal) {
37            var cb = function (valid) {
38              cs(self, 'model').value('state:active-tab', valid)
39            }
40            cs(self).publish('validate-active-tab', nVal, oVal, cb)
41          }
42        })
43      }
44    }
45  })
46
47  app.ui.controller0 = cs.clazz({
48    mixin: [ cs.marker.controller ],
49    protos: {
50      create: function () {
51        cs(this).create('model', app.ui.model)
52      },
53      show: function () {
54        var self = this
55        //Synchronous scenario
56        cs(self).subscribe({
57          name: 'validate-active-tab',
58          func: function (ev, nVal, oVal) {
59            var conf = confirm('Do you want to switch from tab ' +
60              oVal + ' to tab ' + nVal)
61            ev.result(conf ? nVal : oVal)
62          }
63        })
64        //Asynchronous scenario
65        cs(self).subscribe({
66          name: 'validate-active-tab',
67          func: function (ev, nVal, oVal, cb) {
68            $.post('server', [ nVal, oVal ], function (result) {
69              cb(result ? nVal : oVal)
70            })
71          }
72        })
73      }
74    }
75  })
```

Listing 5.11: Input validation performed by a remotely located controller

As we already know, this pattern can be implemented in two ways. In order to make it better understandable, we provide an example outlining both scenarios. At first, we define a model in line 5 which needs an additional member for the asynchronous variant to store transient data. In both cases controller C_1 observes the model members. In the synchronous case however, the set operation is observed so that the result of the validation procedure can be passed directly into the model as seen in line 23. The asynchronous scenario is shown in line 34, where a callback function is created and provided with the validation event. The reaction on validation requests is implemented in lines 56 respectively 65. Here a modal prompt is used as an example for a synchronous and a network request for an asynchronous routine.

Known Uses

P1
: If an input into the mask is done, the data is sent to a parent component, which has all information at hand to validate the input. Afterwards the correct data is sent back to the initial component.

ComponentJS Tracing
: To guard the selection of an item in a list, the list itself fires a click event for containing items. It also provides a method to select a particular item. Whether the user is allowed to select an item or not, is decided from a logic outside the list component.

5.8. Pattern 8: Common Ancestor Communication

When having a look at the hierarchical structure of the Component Tree and the communication paths described by events traveling along the tree (2.3), we soon notice that it does by default only support communication from the root downwards and from all components upwards to that. But while implementing an application, we may run into the issue of having to communicate between components that maintain a sibling relation. One not far to seek solution would be to propagate an event upwards to the root and an event down to the target component, afterwards. This approach obviously leads to an omnipotent root component that is aware of the coherence of the remaining application and violates Logical Separation as well as the whole component oriented approach.

In contrast, the Common Ancestor Communication pattern allows us to establish a link among sibling components. At this point, instead of choosing the root component as a broker, we use the closest common ancestor connecting the two components that want to communicate.

Condition

Usually the components in an application are highly dependent on each other and even if they are loosely coupled they require each other to conduct a use case. In many cases, the required components for one component are its parent components. Here the standard communication along the Component Tree suffices. But due to hierarchical decomposition (2.1) in some cases they are sibling components, i.e. neither is a parent of the other. If a communication among these components is necessary, we need to use a common ancestor of them to coordinate their communication.

After the connection is established, a peer-to-peer communication via callback functions is possible. It is also conceivable to use this pattern as a replacement for the common model access via the `value` method (2.2). This is recommended, when the requesting component must not be aware of the underlying model or when the model is no ancestor of it.

Structure

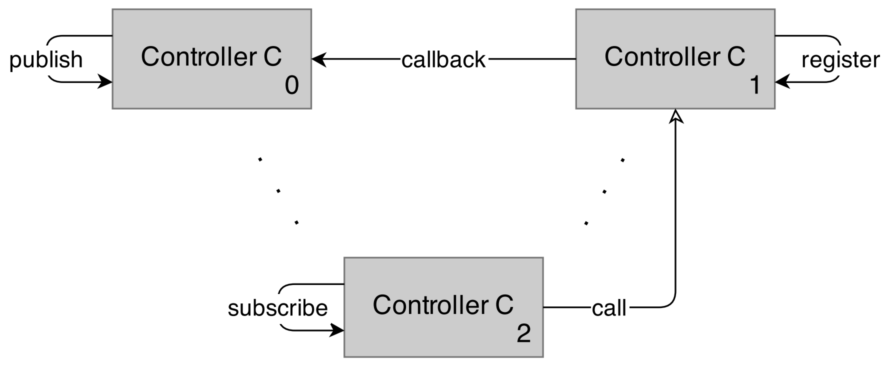


Figure 5.8.

In Figure 5.8 the three participants of this pattern are displayed. First, we see controller C_0, which needs to communicate with controller C_1. Therefore, both are sibling components and require to share controller C_2 as a parent. The structure between C_2 and the siblings is irrelevant for the pattern and thus dropped.

Communication

The Common Ancestor Communication is a highly flexible pattern, which can be applied in many scenarios. The initial situation is that controller C_0 needs to retrieve some piece of information or a task to be performed. This necessity is expressed by publishing an event that bubbles up the Component Tree, giving each component on its path the opportunity to react and deal with it. It is not required that C_0 knows where its request can be fulfilled. This is where controller C_2 comes into play by subscribing to requests it can process. C_2 can be any of the components on the path between the root component and C_0. It knows how to satisfy the request of C_0. Obviously, it can stop the event propagation and hence block any other component from processing it, too.

Outgoing from C_2, several options are conceivable. The first, simple one, is a direct response of C_2, possible when it has the necessary information at hand. A second scenario is the one giving this pattern its name. Controller C_2 has the information, to which sibling of C_0 the request has to be delegated to. This is done using the `call` method (2.2). Of course, the delegate can transitively propagate the request further. In both cases, C_0 provides a callback function with the event, which is later used to send the result of the request back to it. The signature of this callback function is part of the interface definition for components that may answer requests of C_0. Making use of a callback function enables us to handle both, synchronous and asynchronous processing of those requests.

Rationale

Every time an action cannot be fulfilled locally at an individual component, it has to request help from its parent components. In some of these cases, a parent component can directly answer the request and the action can be finalized. In other cases, the component only has the information which other component to ask for help and hence can delegate the request. Both situations are the same from the requesting components point of view. It supplies the help request with a callback which later delivers the result.

Constraints

Listing B.3 of the appendix contains all constraints for this pattern. It allows a controller component to call the `publish`, `register` and `subscribe` API methods onto itself as well as to perform `call` operations on child controllers.

Pros and Cons

Advantages

- Transparent location of C_0 and C_1, leading to good Loose Coupling and Logical Separation.
- Component C_0 does not have to know whether the requested information is retrieved from a third component or C_2 itself.
- Controller C_2 can dynamically decide on which component to dispatch the request of C_0. It even can directly resolve the request, if possible.

Disadvantage

- The delegation of C_2 can lead to a quite long sequence of `call` statements in order to finally reach C_1.

Alternatives

Considering the communication directions from the mediating controller downwards the Component Tree, the Sub-tree Information Delivery pattern (5.3) could be an alternative, when combined with the standard event publishing mechanism (2.2). Due to the uniqueness of this pattern, no other alternative seems feasible.

Code Sample

```
app.ui.controller0 = cs.clazz({
  mixin: [ cs.marker.controller ],
  protos: {
    show: function () {
      var self = this
      var cb = function (result) {
        if (result.success)
          self.user = entityManager.load('User', result.id)
      }
10       cs(self).publish('authenticate', cb)
11     }
12   }
13 })
14 
15 app.ui.controller2 = cs.clazz({
16   mixin: [ cs.marker.controller ],
17   protos: {
18     show: function () {
19       var self = this
20       cs(self).subscribe('authenticate', function (cb) {
21         cs(self, 'login').call('authenticate', cb)
22       })
23     }
24   }
25 })
26 
27 app.ui.controller1 = cs.clazz({
28   mixin: [ cs.marker.controller ],
29   dynamics: {
30     callback: null
31   },
32   protos: {
33     create: function () {
34       var self = this
35       cs(self).register('authenticate', function (cb) {
36         self.callback = cb
37         cs(self).state('visible')
38       })
39     },
40     show: function () {
41       $('#login').click(function () {
42         $.post('server', credentials, function (result) {
43             self.callback(result)
44         })
45       })
46     }
47   }
48 })
```

Listing 5.12: Authentication request delegation

The example in Listing 5.12 involves three controllers. `Controller0`, which needs to load the currently logged in user to display details, `controller2` acting as the broker in this scenario and `controller1`, which can perform user authentication.

First, in line 35, the log-in component registers a service named “authenticate”. It takes a callback as parameter and switches the modal log-in dialog to `visible`, when triggered. We can imagine that `controller0` is the controller of a toolbar component. If the toolbar is shown, then it should display the currently logged in user. To achieve this, it first sets up a callback, in order to receive the users ID, line 6, and then publishes the user authentication request in line 10. This event reaches `controller2` and activates its relaying in line 20. Here the decision is made, to call the log-in dialog, which is an instance of `app.ui.controller1` (see line 27) with the callback function as a parameter. Since asynchronicity is inherent in the log-in dialog, the result can only be delivered when the user hits the “login” button and a request to the server was issued. In line 43, the result of the server communication is passed back to `controller0` using the supplied callback.

Known Uses

ComponentJS Tracing
A common panel component is used as information broker between the different tabs. Each tab can communicate with another one by this common ancestor.

Midas
Works similar to the code sample above. When an entity is requested at the server and it returns “unauthorized”, an event is issued, requesting user authentication. Afterwards the entity is re-requested at the server.

6. Implementation Quality Constraints

This chapter contains the constraints, that are not derived from any particular pattern and thus do not belong to Chapter 5. When having a look at them, it becomes quickly clear, why it is useful to set them up and why they improve the overall quality of an application.

6.1. Unused communication endpoints

Unused code is not really a problem in most deployment scenarios, but may indicate messy code and increase maintenance effort. It can be a result of a lack of developer experience or remnants of the development process. Many Integrated Development Environments (IDE) provide mechanisms to detect this code. We can achieve the same by setting up a few temporal constraints to find unused communication endpoints. The endpoints are introduced by particular operations within our component system. For example, the `observe` operation introduces an endpoint for value changes, triggered by the `value` operation. From this tuple, we can derive that, if the model member, observed by an `observe` call is never set throughout the run of an application, it is likely that the observer is dead code. Listing 6.1 shows a constraint that checks this issue. It is essential, that the **terminate** participant is always the last member in the sequence to be checked.

```
temporal-constraint observe-value {
  rationale {
    Value expects an observe beforehand, otherwise it would be useless.
  }
  sequence {
    a << b << terminate
  }
  filter a {
    operation == 'observe'
  }
  filter b {
    operation == 'value'
    &&  parameters.value != undefined
  }
  link {
      a.parameters.name == b.parameters.name
    &&  isParent(a.origin, b.origin)
  }
}
```

Listing 6.1: Unused observation

It ensures that the constraint is first checked, when the application has terminated and thus communications that would heal a monitor of such an unused-rule cannot be produced any more. The following list contains all the operation pairs we can check for, using analogous constraints.

- observe << value << **terminate**
- register << call << **terminate**
- subscribe << publish << **terminate**
- socket << plug << **terminate**
- property (set) << property (lookup) << **terminate**

6.2. Coding convention enforcement

Another point, where we can improve code quality, is the enforcement of coding conventions. These conventions reduce maintenance effort and make the code more understandable for developers that are new to the project. The particular coding convention this section is aiming at is the correct labeling of model members. As we can see from the user interface component architecture introduced in Chapter 2.3, there are five different types of members designated for the presentation model. It turned out that it improves intelligibility significantly, if members of these categories are uniformly prefixed. For parameter values, the prefix “param:” was chosen, for command values “cmd:” and for the remaining types “state:”, “data:” and “event:”, respectively. It is obvious that we cannot check whether each model member was assigned to the correct group. But we can assure that each of them starts with at least one of the mentioned prefixes. It seems natural to do this check whenever a new model is defined, but since we operate on communication tuples and have only marginal insight on the passed parameters, we cannot perform it at this point.

```
peephole-constraint prefix-check {
  rationale {
    The prefixes for model members have to be chosen according
    to their role within the presentation model.
  }
  condition {
        operation == "value"
    &&  !(startsWith(parameters.name, "param:")
          || startsWith(parameters.name, "data:")
          || startsWith(parameters.name, "state:")
          || startsWith(parameters.name, "cmd:")
          || startsWith(parameters.name, "event:"))
  }
  result FAIL_FINAL
}
```

Listing 6.2: Coding convention enforcement

Together with the assumption that during an examination process the whole program is touched, we can just check for the existence of prefixes whenever a single model member is addressed within a communication. Listing 6.2 shows an example of how we can perform this check. Another way would be to build a constraint for `observe` operations, but since we already ensure in Section 6.1 that no unused observations are made, we can simply stick to monitoring the `value` calls.

6.3. Required communication sequence

In this section we introduce constraints that are more of theoretical nature, since their violation leads to hard faults. Thus, usually they are checked by the component framework itself. In most cases, some communications are required before others can take place, for example each component has to be created before receiving any communication. Listing 6.3 shows this circumstance. Other examples are sockets that has to be defined before any component can plug into them or model members which have to exist when an observation is registered. The latter is modeled in Listing 6.4, which uses the auxiliary functions `isParent` and `contains` in order to first ensure that the model can be accessed by the `observe` respectively `value` call and afterwards to check for the existence of the particular member within that model. A last use case for such a constraint is to check whether a service has been registered before it is called.

```
temporal-constraint existence {
  rationale {
    The component needs to exist before we can interact with it.
  }
  sequence {
    a << b
  }
  filter a {
    operation == 'create'
  }
  filter b {
    operation != 'create'
  }
  link {
    b.origin == a.origin
  }
}
```

Listing 6.3: Required communication sequence - Component existence

```
temporal-constraint model-member-def {
  rationale {
    Model members have to be defined before they can be observed
    or accessed.
  }
  sequence {
    a << b
  }
  filter a {
    operation == 'model'
  }
  filter b {
    operation == 'observe'
    || operation == 'value'
    || operation == 'touch'
  }
  link {
    isParent(a.origin, b.origin)
    && contains(a.parameters, b.parameters.name)
  }
}
```

Listing 6.4: Required communication sequence - Model member existence

6.4. State communication coupling

It is often necessary that a developer is aware of a huge amount of relevant meta information, to conduct the implementation. Most of this information concerns coding guidelines as well as conditions vital for the application to work properly. One of the latter is the definition of the moment when particular actions are allowed to be performed. An example would be: In which state is a view component permitted to plug itself into an underlying socket? There is no unambiguous answer for this question, but it is indispensable to standardize this throughout an application. It increases the possibilities of parallel development and reduces the effort of integrating the components.

```
peephole-constraint plug-state-coupling {
  rationale {
    A view has to conduct its plug operations
    when it is in the materialized state.
  }
  condition {
        operation == "plug"
    &&  originType == "V"
    &&  state(origin) != "materialized"
  }
  result FAIL_FINAL
}

peephole-constraint socket-state-coupling {
  rationale {
    A view has to conduct its socket operations
    when it is in the materialized state.
  }
  condition {
        operation == "socket"
    &&  originType == "V"
    &&  state(origin) != "materialized"
  }
  result FAIL_FINAL
}
```

Listing 6.5: State communication coupling

Listing 6.5 shows a state synchronization for the `socket`, `plug` operation pair. We can bind both operations to the `materialized` state, since the component providing the socket is always a parent of the plugging component and thus will be the first of them to perform the life-cycle method (2.4) to enter this state.

By enforcing this guideline for all components within an application, we can guarantee seamless integration of all visual components. Additionally, we can extend this procedure to standardize other operations as well. An example is the moment, when a view component starts to observe its underlying model. It is legit to bind the `observe` operation to the `visible` state, since earlier data synchronization would just waste performance to actions that are not visible anyway.

7. Applying the Pattern Catalog to Use Cases

We describe three common use cases and how they can be solved by using one single or a composition of patterns from the catalog introduced in Chapter 5. The presented use cases do not claim to provide an exhaustive overview but are only some that we came across during our pattern discovery process. Furthermore, when selecting the examples for this chapter the focus lay on cases where the cooperation between different patterns is vivid.

7.1. Login Dialog

The triad, displayed in Figure 7.1 is a combination of the Rich Widget pattern (5.4) and an instance of the controller component trait (4.3). The model of the Rich Widget contains all necessary information for a login attempt, which are username and password. It also holds an event triggered when the “login” button is pressed and several state variables to indicate server responses as well as client side validation results, which are for example empty credentials.

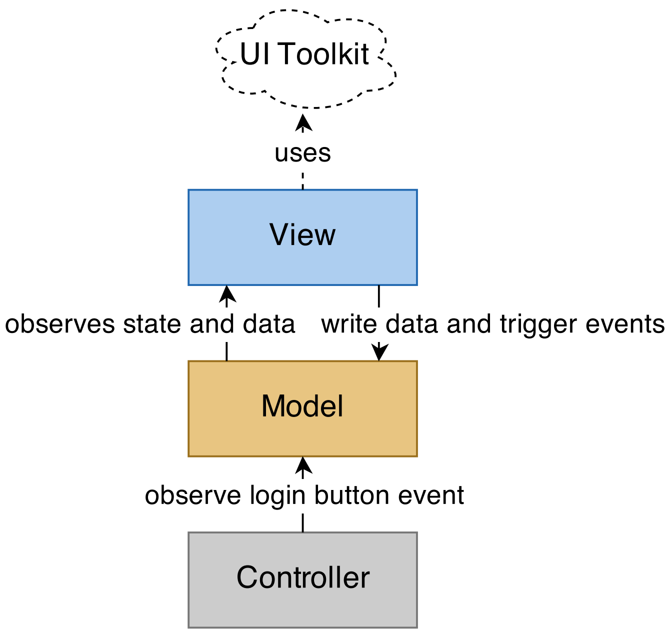


Figure 7.1.

The view of the Rich Widget provides the user interface with its two input fields and the “login” button, embedded into a modal dialog window. The server communication is performed by the controller, which observes the button’s click event.

7.2. Wizard

A wizard is an often used strategy to collect a major set of information from the user in a step-by-step manner. The used patterns are the Managing Widget pattern (5.5) and the Lean Widget pattern (5.1), where the model of the Managing Widget is also acting like described in the Shared Model Component pattern (5.2). We can see this composition in Figure 7.2.

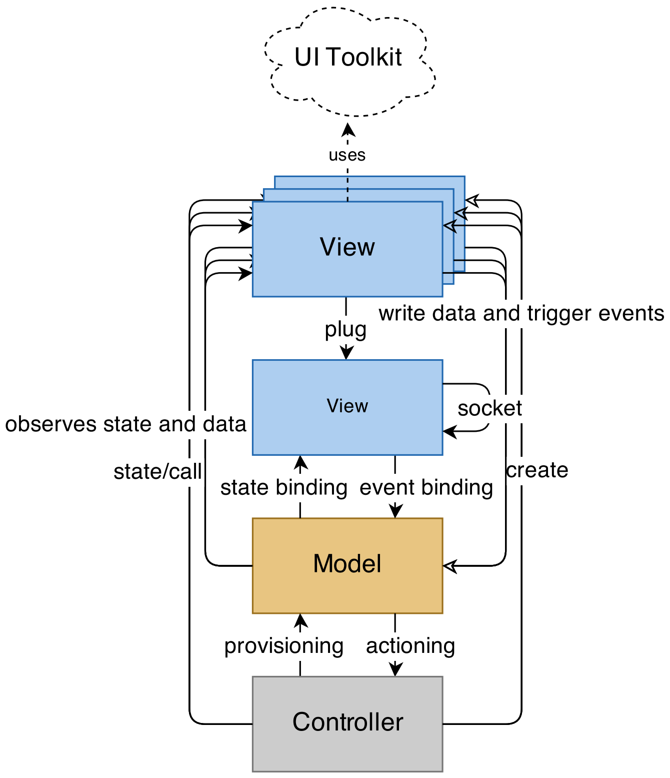


Figure 7.2.

The views on top of the Managing Widget triad provide the layouts for each step the wizard passes through. The navigation and progress indicator is displayed by the comprehensive view of the Managing Widget and managed by the underlying controller. This controller also manages the state of each child view.

7.3. Master-Detail

The third use case we want to have a look at, is the master-detail construct. It consists of two Rich Widgets (5.4). One widget is capable of displaying a list of entities, whereas the other one can present the details of a individual entity instance.

The user can select an entity in the list in order to inspect the details of that entity inside the detail widget. Here, the Common Ancestor Communication pattern (5.8) is used to transmit the selected entity from the master to the detail widget. This use case also allows the usage of several views implementing different representations of a single entity instance. The only requirement for the mediating component is to be a common ancestor of all participating views.

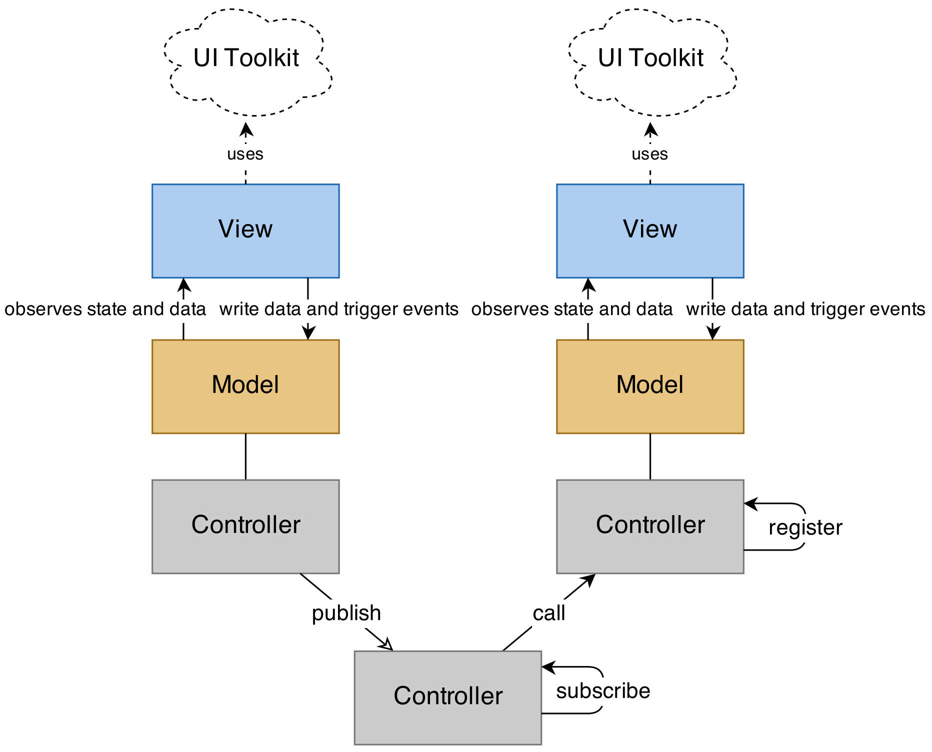


Figure 7.3.

A simple version of this use case is displayed in Figure 7.3. Here, a single master widget on the left provides information for the detail widget on the right.

8. Tooling

The tool, which conducts the constraint checking and the run-time monitoring of the application, is the subject of this chapter. Figure 8.1 shows its architecture. It is split up into three tiers. The one on the top is an arbitrary application on some web server on the web or intranet. The only requirement is that it uses ComponentJS as a component framework. On the other side, at the bottom of our architecture, we have the browser of the developer. The browser has two duties: Displaying the application under scrutiny and doing the actual constraint checking. How this is done is described in detail in Section 8.1.

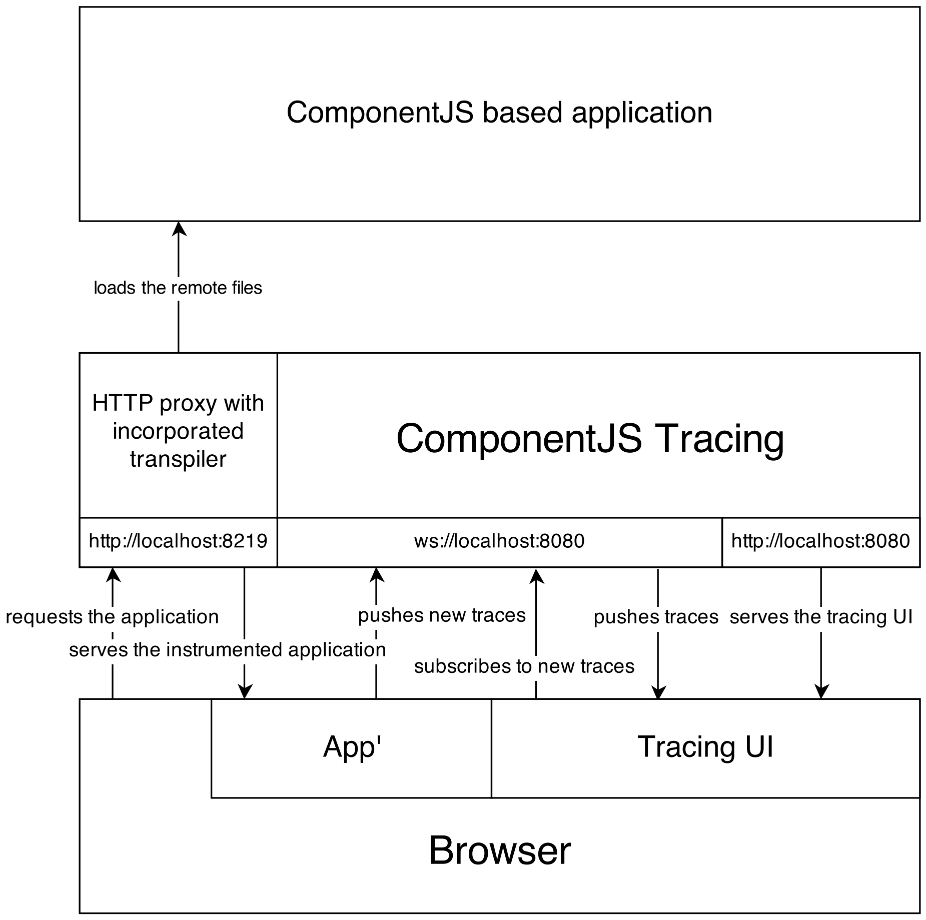


Figure 8.1.: Architecture

A server talking several protocols on three different ports forms the core of our tooling. On the left side of Figure 8.1 we see the HTTP proxy functionality. It is a transparent proxy, hence the application does not notice anything, when loaded through the proxy server. Of course, the browser has to be configured to use it, so that it in turn can read and modify all data passing through. This is necessary to inject the Tracing plug-in 8.2 as well as to instrument the application's code, so that we can later determine which part of the code did what ComponentJS API calls. The instrumentation translates the application, here App, to an instrumented derivative App'. The proxy based approach has one huge advantage: The developer of the system under scrutiny does not have to know or plan that his application is analyzed. We can just start considering any arbitrary application.

Besides the proxy module, the core also includes a web server itself, serving the user interface of our tracing module. We can see this on the right hand side of Figure 8.1. The last server, settled in the core tier, implements the WebSocket protocol. It is used to pass traces, recorded by the injected tracing plug-in to the user interface of the ComponentJS Tracing module, for further examination.

At this point, we want to remark that the production of traces is not limited to ComponentJS and the plug-in, developed within this thesis. With aspect-oriented programming and dependency injection, a Java application can be instrumented and traces can be recorded, as well. These traces can then be directed to the WebSocket server, which in turn can handle them like traces of a ComponentJS based application.

8.1. ComponentJS Tracing

The ComponentJS Tracing module is the core tier of our tool. It is implemented using JavaScript and executed with NodeJS. It consists of a server side part and a client side part. The server application sets up the proxy server as well as the WebSocket relay server. Within the proxy server, a transpiler transforms the code of the application to enable correct detection of the source of a trace.

Proxy Server

In order to be able to intercept HTTP requests, the proxy server uses the NodeJS library "http-proxy-simple" and registers several listeners to capture HTTP traffic. The first listener is placed on "http-intercept-request" events. It removes the HTTP headers, responsible for caching and thus forces that every request delivers a fresh copy and no deprecated code is used. The second listener is attached to "http-intercept-response" events, which are triggered when the remote host has responded but before the response was forwarded to the browser. This listener enables us to look for component code on the one hand and for the ComponentJS library file on the other hand. As the tracing plug-in will later lookup the source component of an API call by walking the caller hierarchy upwards, we need to give anonymous callback functions the information in which component they reside. This is done by wrapping them in a special ComponentJS function call. Listing 8.1 shows a code snippet before and after the transpilation procedure. The "fn" wrapper is annotated with the necessary information for the tracing plug-in.

```
//Before transpilation
cs(this).subscribe({
  name: 'validate-active-tab',
  func: function (ev, nVal, oVal) {
  }
})
//After transpilation
cs(this).subscribe({
  name: 'validate-active-tab',
  func: cs.fn(function (ev, nVal, oVal) {
  })
})
```

Listing 8.1: Transpiler example

Like already mentioned, we also have to modify the ComponentJS library file by appending the tracing plug-in and the tracing remote plug-in, which just receives the traces from the tracing plug-in and passes them to the server using the WebSocket protocol and a third-party library for the pure WebSocket handling. To be able to alter the correct files, the ComponentJS library file as well as the component files have to be specified in a configuration file, loaded on start-up. The remaining HTTP traffic is not affected by the proxy.

Websocket Relay

For efficient relaying, the WebSocket server maintains a room that clients, which are interested in incoming traces, can join. Upon receiving a trace, a broadcast for all clients within that room is performed. This enables us to have several trace consumers listening to the current run at once. Besides the relaying, it can also be specified to log the traces to a separate file as they occur in order to analyze them later. This is useful when we have only one computer to do the execution of the system under scrutiny and the monitoring of traces, whereby the execution should not be influenced by the examination process. Apart from the monitoring feature, the WebSocket server can also forward ComponentJS commands from the ComponentJS Tracing user interface to the application, App', to allow real-time manipulation of its component system.

Client

The client application makes up the biggest part of the tooling. As we will see in the following sections, it records traces, checks constraints and maintains the current state of the system. Furthermore it provides additional tools to analyze the communication within a ComponentJS based application. Three stages were necessary to build it.

Phase 1: Evaluation as a Google Chrome plug-in

Phase 2: Implementation of the application

Phase 3: Further evolution through experience in practice

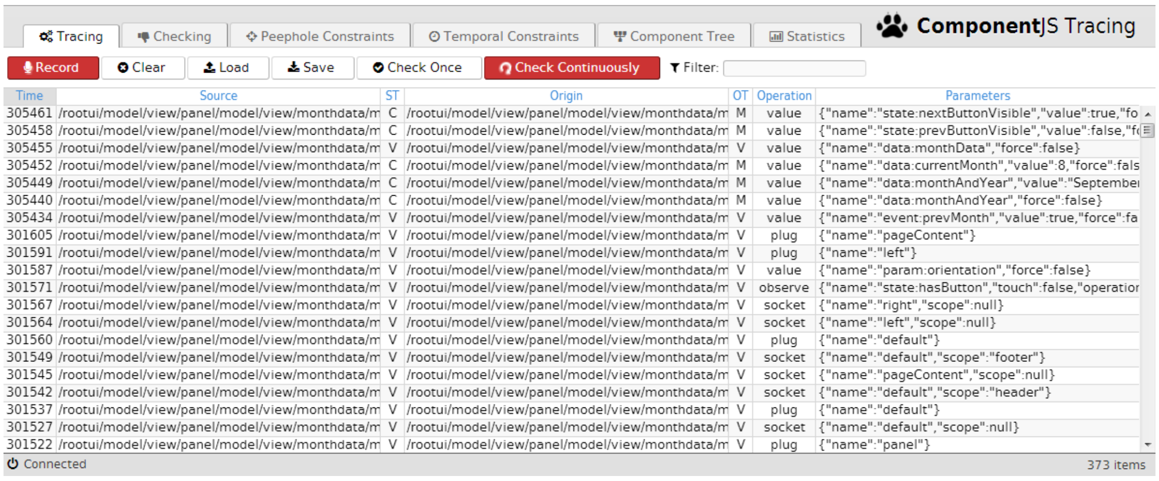
In phase one we planned to build our tooling inside the Google Chrome browser by providing an additional tab for its developers tools. After sophisticated evaluation of the capabilities granted to the developers tools, we came to the conclusion that it was not possible to use this approach to achieve our goal. Further explanation can be found in Section 8.5 in addition to other obstacles we had to overcome.

Phase two, the implementation phase, partially already began in phase one leaving us with a HTML and JavaScript based rudimentary application which could be easily converted into an SPA and hosted by our own web server, implemented in NodeJS. In the planning phase we had already decided to use ComponentJS in order to proof our own concepts and to follow the principle of eat your own dog food, which makes it on the one hand more challenging for us — as we sure would have to deliver a "perfect" architecture — but on the other hand provides an excellent component framework and examples for this thesis, as well.

During the implementation, we followed the MVC/CT architecture pattern described in Chapter 2.3. Inside the architecture, several patterns found throughout this thesis were applied. One example is the Shared Model pattern (5.2) in combination with the Lean Widget pattern (5.1) that were used to implement the toolbar. The Rich Widget pattern (5.4) came to use in order to implement the grid component, which was reused in nearly every tab.

Within phase three, the practice test, we received several change requests and additional requirements from developers and expanded our tool to support the development process as good as possible, which leads us to the tool as it is now. The following sections will describe each part of it in detail.

Tracing



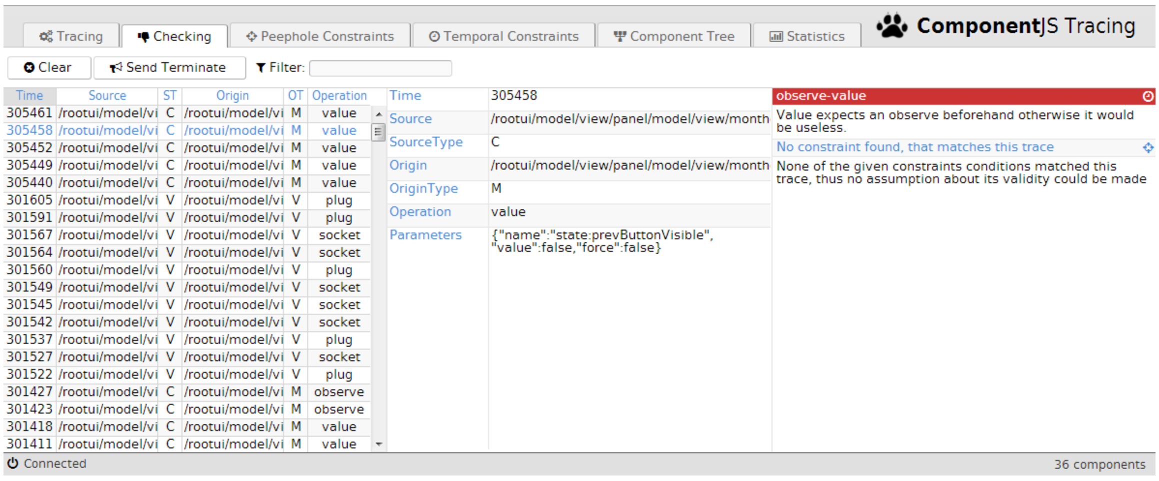
Time	Source	ST	Origin	OT	Operation	Parameters
305461	/rootui/model/view/panel/model/view/monthdata/m	C	/rootui/model/view/panel/model/view/monthdata/m	M	value	{"name":"state:nextButtonVisible","value":true,"fo
305458	/rootui/model/view/panel/model/view/monthdata/m	C	/rootui/model/view/panel/model/view/monthdata/m	M	value	{"name":"state:prevButtonVisible","value":false,"f
305455	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	value	{"name":"data:monthData","force":false}
305452	/rootui/model/view/panel/model/view/monthdata/m	C	/rootui/model/view/panel/model/view/monthdata/m	M	value	{"name":"data:currentMonth","value":8,"force":fals
305449	/rootui/model/view/panel/model/view/monthdata/m	C	/rootui/model/view/panel/model/view/monthdata/m	M	value	{"name":"data:monthAndYear","value":"Septembe
305440	/rootui/model/view/panel/model/view/monthdata/m	C	/rootui/model/view/panel/model/view/monthdata/m	M	value	{"name":"data:monthAndYear","force":false}
305434	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	value	{"name":"event:prevMonth","value":true,"force":fa
301605	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	plug	{"name":"pageContent"}
301591	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	plug	{"name":"left"}
301587	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	value	{"name":"param:orientation","force":false}
301571	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	observe	{"name":"state:hasButton","touch":false,"operatior
301567	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	socket	{"name":"right","scope":null}
301564	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	socket	{"name":"left","scope":null}
301560	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	plug	{"name":"default"}
301549	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	socket	{"name":"default","scope":"footer"}
301545	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	socket	{"name":"pageContent","scope":null}
301542	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	socket	{"name":"default","scope":"header"}
301537	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	plug	{"name":"default"}
301527	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	socket	{"name":"default","scope":null}
301522	/rootui/model/view/panel/model/view/monthdata/m	V	/rootui/model/view/panel/model/view/monthdata/m	V	plug	{"name":"panel"}

Connected 373 items

Figure 8.2.: Tracing

In the tracing tab, Figure 8.2, all traces are collected and displayed to the developer. Using the toolbar, he can save the current run to a file for later examination. As already mentioned in the WebSocket section (8.1), the user can also load old runs by uploading previously exported files or ones recorded by the websocket relay. If the analysis is performed online, then the "check continuously" option can be enabled in order to send each incoming trace instantly to the constraint checker. Otherwise, the "check once" button can be used to replay a present run, like it was recorded and forward it to the constraint checker.

Checking



Time	Source	ST	Origin	OT	Operation
305461	/rootui/model/vi	C	/rootui/model/vi	M	value
305458	/rootui/model/vi	C	/rootui/model/vi	M	value
305452	/rootui/model/vi	C	/rootui/model/vi	M	value
305449	/rootui/model/vi	C	/rootui/model/vi	M	value
305440	/rootui/model/vi	C	/rootui/model/vi	M	value
301605	/rootui/model/vi	V	/rootui/model/vi	V	plug
301591	/rootui/model/vi	V	/rootui/model/vi	V	plug
301567	/rootui/model/vi	V	/rootui/model/vi	V	socket
301564	/rootui/model/vi	V	/rootui/model/vi	V	socket
301560	/rootui/model/vi	V	/rootui/model/vi	V	plug
301549	/rootui/model/vi	V	/rootui/model/vi	V	socket
301545	/rootui/model/vi	V	/rootui/model/vi	V	socket
301542	/rootui/model/vi	V	/rootui/model/vi	V	socket
301537	/rootui/model/vi	V	/rootui/model/vi	V	plug
301527	/rootui/model/vi	V	/rootui/model/vi	V	socket
301522	/rootui/model/vi	V	/rootui/model/vi	V	plug
301427	/rootui/model/vi	C	/rootui/model/vi	M	observe
301423	/rootui/model/vi	C	/rootui/model/vi	M	observe
301418	/rootui/model/vi	C	/rootui/model/vi	M	value
301411	/rootui/model/vi	C	/rootui/model/vi	M	value

Time	305458
Source	/rootui/model/view/panel/model/view/month
SourceType	C
Origin	/rootui/model/view/panel/model/view/month
OriginType	M
Operation	value
Parameters	{"name":"state:prevButtonVisible", "value":false,"force":false}

observe-value

Value expects an observe beforehand otherwise it would be useless.

No constraint found, that matches this trace

None of the given constraints conditions matched this trace, thus no assumption about its validity could be made

Connected

36 components

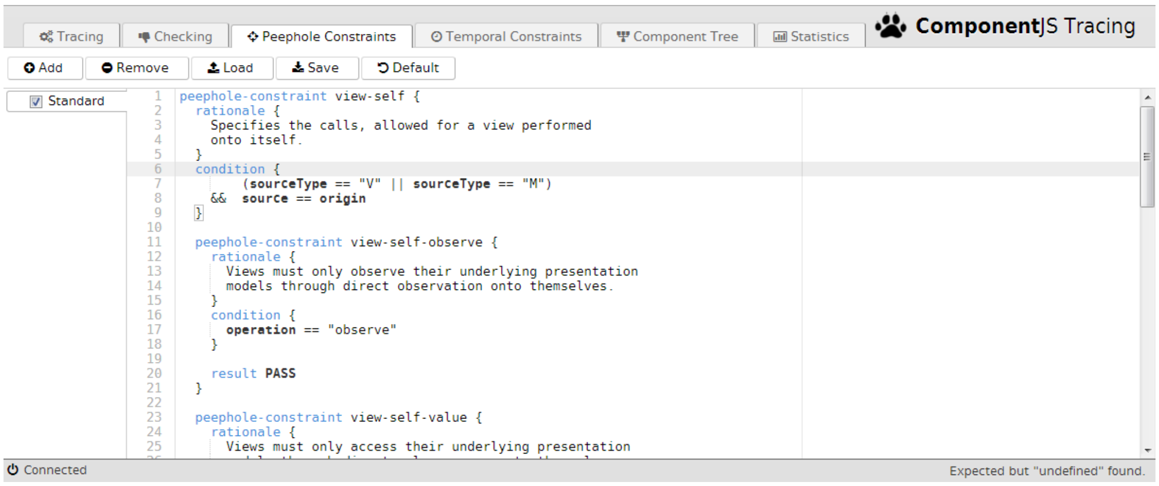
Figure 8.3.: Checking

As we can see in Figure 8.3, the checking section is arranged using a three column layout. Hereby, the grid component, we have already seen in the tracing perspective, is also instantiated, but with a slightly altered configuration allowing it to fit in less space by omitting the parameters column. It is located in the left most column and displays all violating traces, thus it is rather an overview than the actual informative part. In the column in the middle, the user can see the details of a selected trace, including its parameters. The column on the right contains a list of name, rationale tuples that are violated by the particular trace.

The "send terminate" button triggers the checking of constraints, containing the **terminate** participant, as described in the run-time verification chapter (3.4).

Peephole Constraints

The peephole constraints tab, Figure 8.4, is used to manage one or many sets of those constraints. By default, a standard set is provided, containing the constraints delivered with this thesis. The editor supports syntax highlighting for the DSL, describing each constraint. Also included is functionality to export, import and add new sets of constraints. Furthermore, the different sets can be enabled and disabled independently in order to choose the correct compilation of constraints for each particular project. Whenever a constraint is changed or introduced, it is syntactically and semantically checked. For example the unique character of the constraint identifiers can be ensured. If an error occurs, it is displayed in the status bar right at the bottom, see Figure 8.4.

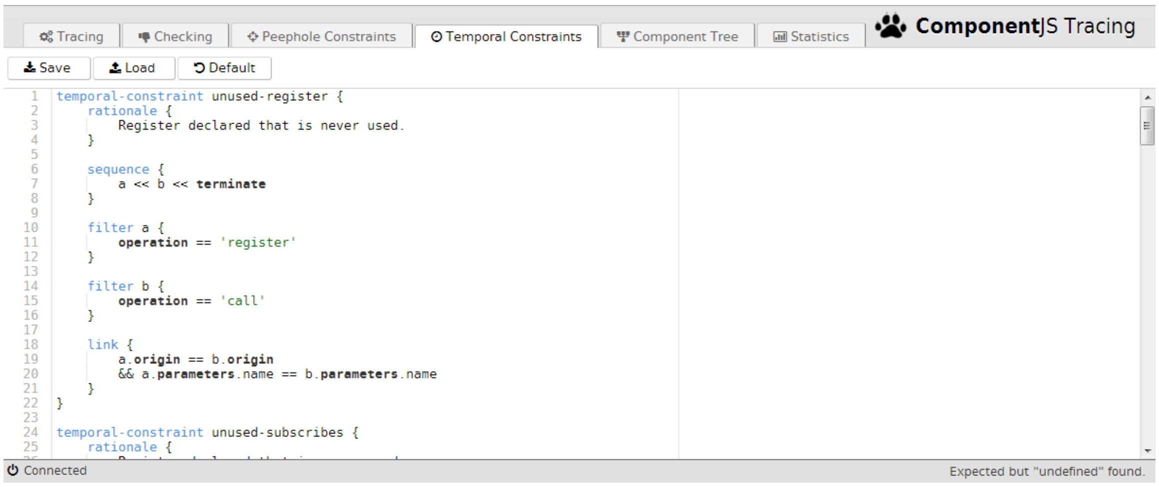


```
peephole-constraint view-self {
  rationale {
    Specifies the calls, allowed for a view performed
    onto itself.
  }
  condition {
        (sourceType == "V" || sourceType == "M")
    &&  source == origin
  }

  peephole-constraint view-self-observe {
    rationale {
      Views must only observe their underlying presentation
      models through direct observation onto themselves.
    }
    condition {
      operation == "observe"
    }

    result PASS
  }

  peephole-constraint view-self-value {
    rationale {
      Views must only access their underlying presentation
```

Figure 8.4.: Peephole Constraints

Temporal Constraints

In order to improve intelligibility and for a clear distinction between the different constraint types, a separated tab (see Figure 8.5) is dedicated to temporal constraints. Of course, it has the same export and import capabilities and supports syntactic as well as semantic validation. In contrast to the peephole constraints, a deeper semantic validation is possible and thus appropriate. First of all, duplicate constraint identifiers are eliminated. Afterwards the sequence section is checked. It has to contain two or more unique participants and the terminate participant has to be the last one, if present. Furthermore, it is checked that each filter expression is associated with a member of the sequence expression. Last, we validate that each participant used in the link segment has a corresponding element in the sequence, as well.



```
temporal-constraint unused-register {
    rationale {
        Register declared that is never used.
    }

    sequence {
        a << b << terminate
    }

    filter a {
        operation == 'register'
    }

    filter b {
        operation == 'call'
    }

    link {
        a.origin == b.origin
        && a.parameters.name == b.parameters.name
    }
}

temporal-constraint unused-subscribes {
    rationale {
```

Figure 8.5.: Temporal Constraints

After the constraint collection has passed both validation steps, a monitor is constructed for each constraint. How these monitors work is described in detail in Section 8.4.

Component Tree

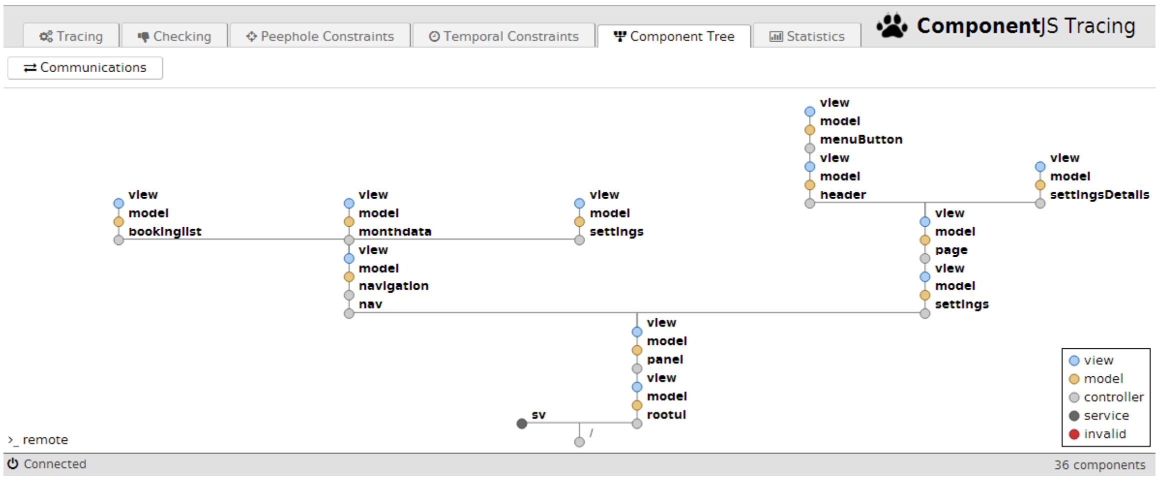


Figure 8.6.: Component Tree

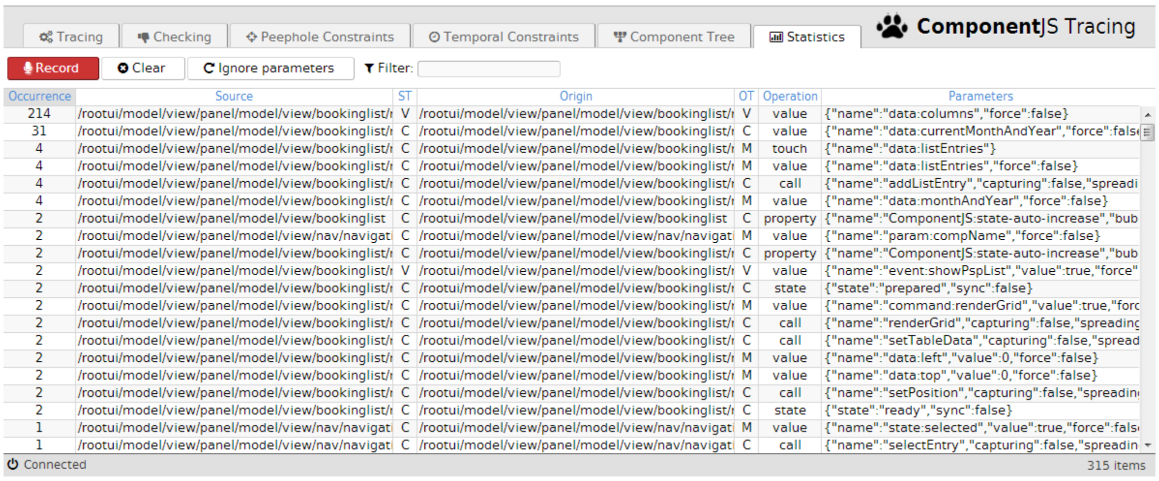
As we have already seen in Chapter 3.4, the availability of information about the systems state plays an essential role for run-time verification. Since we record each communication between components, we do also trace the creation and state changes of each component. These traces are filtered out and visualized by the component tree tab, see Figure 8.6. It displays the components currently present in the application as well as additional information, e.g. their states, subscribed events etc. It also provides an interface for the constraint checker to retrieve this information in order to use it for constraint condition evaluation.

Furthermore, the tab shows the user each communication among the different components upon pressing the “communications” button or hovering over a component. Here, lines between components indicate that a trace was recorded between them. The thickness of the line indicates the amount of communication done, divided into four categories: 10, 20, 30 and more than 40 traces, connecting the components. In addition to the status quo visualization, the component tree also displays constraint violations by coloring these communications red. If a component is doing an invalid API call onto itself, it is completely colored in red.

Apart from the current state, the user can also manipulate the component system of the system under scrutiny by using the remote console, triggered by the “remote” label at the bottom left corner of the tab.

Statistics

Sometimes, aggregation of occurring traces gives a good hint on what is wrong with an application. For example, bad performance leads to poor user experience. Too



Occurrence	Source	ST	Origin	OT	Operation	Parameters
214	/rootui/model/view/panel/model/view/bookinglist/r	V	/rootui/model/view/panel/model/view/bookinglist/r	V	value	{"name":"data:columns","force":false}
31	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	C	value	{"name":"data:currentMonthAndYear","force":false
4	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	M	touch	{"name":"data:listEntries"}
4	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	M	value	{"name":"data:listEntries","force":false}
4	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	C	call	{"name":"addListEntry","capturing":false,"spreadi
4	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	M	value	{"name":"data:monthAndYear","force":false}
2	/rootui/model/view/panel/model/view/bookinglist	C	/rootui/model/view/panel/model/view/bookinglist	C	property	{"name":"ComponentJS:state-auto-increase","bub
2	/rootui/model/view/panel/model/view/nav/navigati	C	/rootui/model/view/panel/model/view/nav/navigati	M	value	{"name":"param:compName","force":false}
2	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	C	property	{"name":"ComponentJS:state-auto-increase","bub
2	/rootui/model/view/panel/model/view/bookinglist/r	V	/rootui/model/view/panel/model/view/bookinglist/r	V	value	{"name":"event:showPspList","value":true,"force"
2	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	C	state	{"state":"prepared","sync":false}
2	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	M	value	{"name":"command:renderGrid","value":true,"forc
2	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	C	call	{"name":"renderGrid","capturing":false,"spreading
2	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	C	call	{"name":"setTableData","capturing":false,"spread
2	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	M	value	{"name":"data:left","value":0,"force":false}
2	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	M	value	{"name":"data:top","value":0,"force":false}
2	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	C	call	{"name":"setPosition","capturing":false,"spreadin
2	/rootui/model/view/panel/model/view/bookinglist/r	C	/rootui/model/view/panel/model/view/bookinglist/r	C	state	{"state":"ready","sync":false}
1	/rootui/model/view/panel/model/view/nav/navigati	C	/rootui/model/view/panel/model/view/nav/navigati	M	value	{"name":"state:selected","value":true,"force":fals
1	/rootui/model/view/panel/model/view/nav/navigati	C	/rootui/model/view/panel/model/view/nav/navigati	C	call	{"name":"selectEntry","capturing":false,"spreadin

Connected

315 items

Figure 8.7.: Statistics

much server communication can be a reason for this. By aggregating the different communications, a developer can see on a first glance, which methods might be called too often and thus represent a good starting point for optimization. In addition to the raw counting ability, the statistics tab (see Figure 8.7) also provides the option to ignore parameters, collapsing the method calls that only differ in their arguments. Further, by using the “clear” and “record” button, the developer can also easily record only a part of the behavior of the application and thus set the focus right where he wants to identify problems. Of course, the grid component is also reused for this perspective.

8.2. Tracing plug-in

The tracing plug-in for ComponentJS is structured into one recording plug-in, which is generating traces from API calls, and two smaller plug-ins that consume these traces and direct them to different outputs. Each trace is an instance of the “Tracing” class. It carries the following information.

Timestamp	Elapsed time since the tracing has begun.
Source	Component issuing the API call.
SourceType	Type of the source component.
Origin	Component receiving the API call.
OriginType	Type of the origin component.
Operation	Name of the issued API call.
Parameters	Parameters of interest of that API call.

Once all information is present, the trace can be flushed triggering the anchor point `ComponentJS:tracing` in which the two other plug-ins latched to process the traces. We can now have a look at the tracing plug-in itself. It uses hooks provided by the ComponentJS library and hereby triggers the tracing algorithm as well as retrieves the relevant information for the traces. The following hooks are deployed to produce traces directly when they are called.

ComponentJS:state-change-call Emits traces of the "state" operation in order to enable ComponentJS Tracing to reflect the current state of all components.

ComponentJS:comp-created Leads to "create" traces to indicate component creation to the component tree tab, mentioned in the last section.

ComponentJS:comp-destroyed Is analogous to the previous hook, while taking care of component destruction.

The most crucial part of the tracing plug-in lies in the determination of the source information of each API call, i.e. the calling component. Since nearly all of these calls are issued within the life-cycle methods (2.4) of the respective component, we have to first annotate these methods with the component by latching into the "ComponentJS:state-method-call". In addition to that, we have to annotate the callbacks used in the API calls with the same information, since they may also contain API calls themselves. Here the "fn" function comes into play, which is wrapped around each callback by the transpiler inside the proxy server. It bridges the information from outside the callback to be available for trace generation inside.

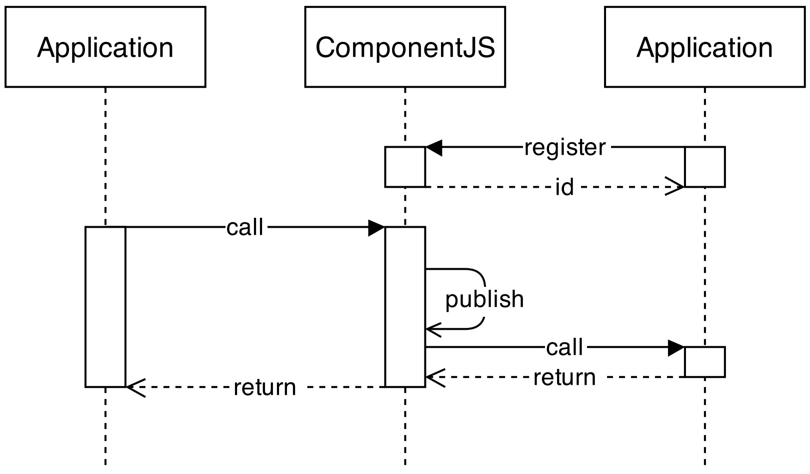


Figure 8.8.: Tracing Plug-in

After this preparation, we are only one step away from the real tracing mechanism. The problem that is still present is outlined in Figure 8.8. The ComponentJS library uses the event mechanism (2.4) for each kind of other API method and thus creating traces on **all** API calls would lead to way to many traces which are not produced by the developer himself but by the component framework. To circumvent this issue, we check whether the source of a trace is the `_cs.internal` component, which is an artificial component used as source, whenever an API call is done by the framework itself. See Chapter 9.2 for the future of this problem.

Finally, we can have a look at the core of this plug-in. In order to create traces, each API method is overwritten, adding the instantiation of a new trace object and afterwards calling the overwritten base function. Additionally, the parameters processing hook is used in order to collect the parameters of interest inside the newly created trace object and to set the source for subsequent API calls to `_cs.internal`.

Now as we have managed to create traces and to distinguish between the ones caused by the developer and the ones introduced by ComponentJS, we can transmit them for further investigation. The two plug-ins already mentioned in the introduction of this section are redirecting them to the developers console or to the WebSocket Relay, respectively.

8.3. DSL Parser

Our first shot for parsing the DSL introduced in Chapter 3.3 was to write a parser on our own in order to maximize efficiency and suitability for our very own needs. During further progress of the thesis and with growing size of our language, we realize that a more flexible approach would be to use a library which allows us to specify our language as a grammar and which would construct a parser according to our production rules. We decide to use a Parsing Expression Grammar (PEG) since it creates very efficient parsers, while their downsides do not interfere with our language requirements. In particular, we choose the PEG.js implementation, which also has a decent error reporting and supports JavaScript like notation in its production rules. Listing 8.2 shows that notation for the rationale section within our DSL. The code inside the curly brackets at the end of the statement enclose the JavaScript code for that production rule, whereas the part in the front specifies the pattern matching.

```
rationale
  = "rationale" _"{" _ text:([^}]*) "}"_ { return text.join('').trim() }
```

Listing 8.2: PEG.js example

Peephole and temporal constraints share many production rules, as we have already seen in their respective sections in Chapter 3.3. For that reason, we used our build tool Grunt to combine several grammar fragment files to one composite file for each constraint type. This approach ensures consistency across the different constraint grammars and automates the common task of copying files together and afterwards compiling them with the PEG.js compiler to generate a parser for them both.

8.4. Constraint checker

The constraint checker is responsible to examine the stream of incoming traces and to check their consistency regarding the constraints specified by the developer. Hereby, the prototype of each incoming trace is enriched by methods to compare, hash, match and evaluate conditions on the data of each particular trace. Auxiliary functions, used inside the conditions are also registered within the enrichment class and new ones can be added easily.

Peephole Constraints

In general, the set of available peephole constraints is ordered in linked list like manner each time, a constraint is changed. The list of constraints first follows the natural order in which the constraints are written down inside the constraint declaration tab and then the positioning directives `after` and `before`. As we have already seen in Chapter 3.3, peephole constraints can be nested arbitrarily. Inside each nesting level, the ordering is also computed as outlined above.

The peephole constraint checker is now ready to analyze incoming constraints. When a trace arrives, we iterate over the list of constraints and the condition of each of them is passed to the enriched trace object in order to evaluate it. If the condition for a constraint, containing sub-constraints, is `true` then the iteration proceeds with the nested constraints first and resumes traversing the current level of constraints afterwards. Each time a leaf constraint, i.e. a constraint with no nested constraints, is reached and the evaluation of its condition returns `true`, the rationale of this constraint is appended to the list of matched constraints and the result of the trace is overwritten with the result of that particular constraint. Like this behavior already indicates, the constraint checker follows last match semantics, which was chosen to provide a more natural sequence when writing down constraints. Through the annotation of all matching constraints, we can identify multiple constraint violations at once, which are later displayed in the checking panel.

Since we piggyback the result and rationales on each trace, the output of the constraint checker is also a stream of traces. We will see in the next section, that this stream is picked up by the next layer of constraint checkers.

Temporal Constraints

The stream of traces, produced by the peephole constraint checker is directed to the temporal constraints checker. However, the temporal constraint checker is not one single instance, but a collection of monitors generated from the temporal constraint specifications.

In a first step, each monitor checks whether an incoming trace matches one of its filters or not. If so, it is sorted into a storage labeled with the matched filter's name. Now the checking mechanism is triggered with a few short circuit evaluations at its beginning. If the matched trace is the first participant in the sequence, then nothing more is to do and the checking is finished. In the other cases, the predecessor of the participant is computed and it is checked if any traces reside in the storage for it. If none are present, then the rule is violated and the evaluation can be aborted by yielding an error. The last possibility is that the matched participant is the last one in the sequence, then the storage of all participants are searched for traces satisfying the link condition of the constraint. Are these traces found, then the evaluation can be aborted and the constraint is not violated, otherwise it is.

One special case is the presence of the *terminate* participant inside the sequence statement. A terminate trace is dropped, when the developer triggers it by pressing the "send terminate" button inside the user interface or if the connection to the system under scrutiny is lost. Then a copy of the sequence is created, inverted and the terminate participant is removed. This process converts the sequence $a << b << terminate$ to $b << a$. After that, all traces in the storage of the new last participant, here a are artificially processed like they have just arrived from the peephole constraint checker.

Again all violating traces are labeled with the rationale of the violated constraint. Thus the rationale as well as the actual data of the trace can be listed in the checking tab (see Section 8.1).

8.5. A hard way to go

There were three hurdles to take until we had a sophisticated solution to trace down API calls:

1. Live tracing of API calls, since these calls are also used inside the library itself. Hereby, the obvious but nevertheless not trivial difficulty was to distinguish between calls inside and outside of ComponentJS.
2. Chrome plug-ins are not able to modify the contents of a web request. The first mock-up of our plug-in is shown in Figure 8.9. This was also completely implemented until the web request problem arose. Luckily, we could reuse nearly all of the code, since the new approach using NodeJS was also based on JavaScript.
3. Writing a http-proxy module for NodeJS in order to modify the content of JavaScript source files. We also published this to the NodeJS module packaging system NPM.

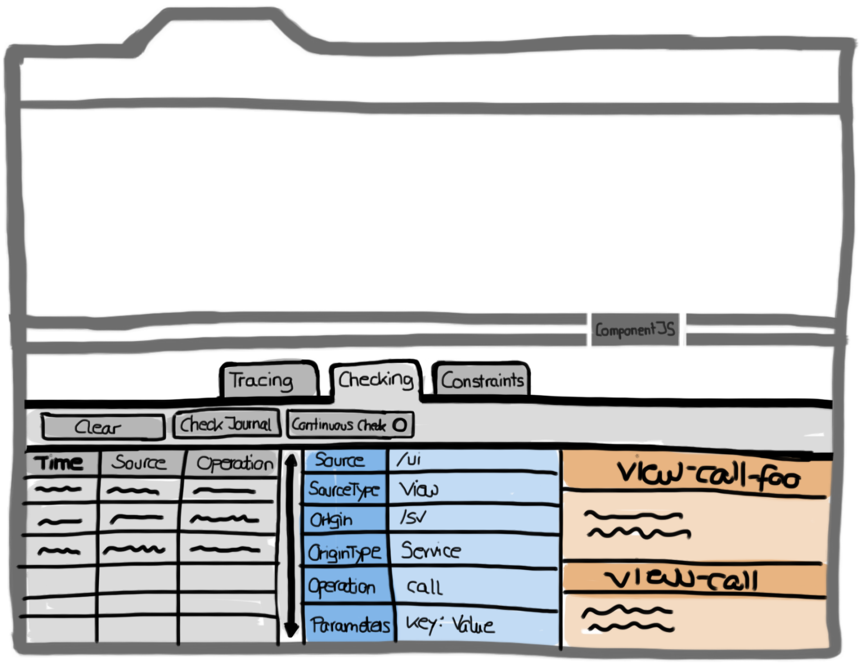


Figure 8.9.: Chrome Plug-in

9. Conclusions

9.1. Summary

We showed a heuristic approach to solve the problem of developing a component architecture for user interfaces within but not limited to the web domain. Therefore, we provide a pattern catalog, which gives an architect a sophisticated well-assorted reference to overcome nearly all requirements addressed to user interfaces nowadays. This was only possible through meticulous examination of existing applications and both their weaknesses as well as their strengths.

As the main focus lay on the communication between the components of an architecture, also the task of monitoring the communication within a running application arose. To deal with this, we provide a tool, which is completely transparent for the software developer. Although the pure monitoring functionality is already a huge improvement when it comes to maintaining or extending user interfaces, it is obvious that it would be an even bigger enhancement, if architectural decisions could be directly checked in the communication stream. That is why we designed a DSL to formulate such checks. The result is two types of constraints: Peephole constraints used to check a single communication between two components and temporal constraints, which are statements about sequences of several communications. Since the developed tool is capable of checking these conditions at run-time, it allows the developer to observe violations as they occur.

Besides the constraints arising from the architecture we identified other areas where our constraint checking proved beneficial. These areas range from coding convention enforcement over unused code detection to best practice imposition.

In the end, the presented approach is designed to improve software quality. This starts in the early phases when the architecture is developed and continues throughout the whole development process, until the product is finalized and can be shipped.

9.2. Future work

Even though we developed a quite wide range of concepts, which were also implemented for practical use, there are still ideas for future development in the field of this thesis.

One thing is that an additional category of constraints, that involves counting the occurrence of traces, for example to model a communication protocol, where as many acknowledgments as messages sent, have to be registered. The theory for this type of conditions is not yet developed but would open up a completely new area.

Another enhancement would be the automatic suggestion of corrections for constraint violations. Here, the structure of the constraint set might play an important role to find the best matching solution. Closely related to this is the idea of detecting used patterns based on the observed communication and simultaneously identifying their correct usage. At the same time, a project specific blacklist for patterns seems legit. Hereby, the developer is notified as a blacklisted pattern is detected.

Additionally, our tool implementation could be enriched by the ability to filter traces by components, so that only the traces of a specified set of components are processed. Along with this, a check might come in handy, verifying whether the selected components follow a particular pattern or not. Another enhancement would be the preparation of further helper functions e.g. `findClosestComponent(start, condition)`, which finds the component, which is closest to the component provided in parameter *start* and matches *condition*.

Last, the integration of our tool into the development process could be subject to improvements, too. It is conceivable to ship pre-built ComponentJS components that come bundled with a set of constraints, ensuring their correct usage. Beyond this enhancement, an consolidation into a continuous integration platform seems to give a good ROI. The best practice would be to take a test runner that makes use of Headless Testing and tries to achieve maximum communication coverage. The application will thus generate as much traces as possible, which can afterwards be analyzed by our tool.

A. Glossary

Adobe Flash™ Multimedia platform to present vector graphics, animations and rich Internet applications. 5

API An Application Programming Interface (API) is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API. 6, 8, 9, 20, 24–27, 29, 31, 42, 49, 59, 74, 79–82, 84, 88, 90

Architecture Fundamentals HTML5 application to browse the architecture fundamentals by Engelschall, Ralf S. (http://architecturefundamentals.com/app.html). 33

aspect-oriented programming Programming paradigm aiming to encapsulate cross-cutting concerns and provide them where necessary. Example: Logging. 74

Backus-Naur Form Notation format for context free grammars. It is often used to describe the structure of languages within computer science. 15

broadcast The simultaneous delivery of a message to all members of a group. 36

cloudTeX Collaborative online LaTeX editor (http://cloud-tex.com/). 36, 41

Component Tree The hierarchical structure of the different components in an application's architecture. 2, 5, 7, 18, 23–27, 31, 36–39, 46, 51, 53, 57–59, 88, 92, 93

ComponentJS ComponentJS is a JavaScript library, providing a run-time component system to structure a user interface hierarchically. 3, 6, 8, 9, 73–76, 80–82, 84, 86

ComponentJS Tracing The tool developed within this thesis. 32, 36, 44, 48, 52, 57, 61

Constructional Reusability High reuse of proven structural components and partial solutions. 34, 46

CSS Cascading Style Sheets. 32, 52

dependency injection Removes hard-coded dependencies among software components. These dependencies might be changed at compile-time as well as at run-time, in some special cases. 74

DOM Document Object Model. 5, 7, 10, 46

Domain specific Information or knowledge that is related to a particular domain. 24

DSL A Domain Specific Language (DSL) is a language, usually simplifying the notation of terms commonly used in a particular domain. 2, 15, 17, 20, 31, 42, 77, 82, 85

eat your own dog food Make use of your own technology to justify its presents and partially proof its convenience. 76

Exclusive Sovereignty Exclusive resource possession by its enclosing component. 33, 34, 36–39, 41, 42

ExtJS Application framework to build interactive web applications written in JavaScript. 5

Factual Locality The spatial and temporal scope of resources is held as narrow as possible. 24, 31, 34, 38, 39

FAP Fundamental Architectural Principles. 15

Grunt An ant like task runner written in JavaScript. 82

Headless Testing Common method to test the substructure of a user interface. The visual components are removed from the application and tests are run on model and controller components only. 6, 30, 31, 42, 86, 90, 93

HMVC Component architecture, where multiple MVC-triads are stacked on top of each other to represent the hierarchical character of a user interface. 3

HTML Hypertext Markup Language. 5, 29, 52

IDE Application providing the necessary tools for computer programmers to write software. 63

Inversion of Control Programming technique that changes the control flow. Usually the control flow is determined by the objects that are statically linked together at compile-time. But if inversion of control is used then this decision is postponed to run-time. Example: Objects register callback functions at run-time, which are called when specific events occur. 34, 55

Java A cross-platform programming language that is concurrent and object-oriented. 74

JavaScript JavaScript is an interpreted programming language, formerly only used on the client side, in browsers. Today with NodeJS also server side applications can be written. 74, 76, 82, 84

jQuery A client-side JavaScript library simplifying the manipulation of and interaction with HTML. 29

jQuery Markup jQuery Markup is a jQuery plug-in for assembling, pre-compiling, expanding and then DOM-injecting HTML markup code fragments based on plain HTML or an arbitrary template language. 29

jQuery UI A JavaScript library providing low-level abstraction of interactions, animations and widgets using HTML. 5

Logical Separation Separation of concerns between the components of a software solution. 7, 23, 42, 46, 54, 57, 59

Loose Coupling Loose coupling in communication and referencing between software components. 30, 31, 36, 38, 59, 90

LTL Modal temporal logic which provides the ability to set up modalities that refer to time. 18–20

MagicSearchTool Tool to administrate card decks (http://magicsearchtool.com/). 41

maintainability The ability to change the system to deal with new technology or to fix defects. [BD04] p.160. 42, 93

Midas An SPA for the visualization of energy consumption for retailer chains. 40, 44, 52, 61

MVC A pattern in software development used to structure a complex concern into a data, presentation and controlling component. 2–6

MVC/CT Model View Controller on a Component Tree. 76

MVP Model View Presenter. 5

NodeJS JavaScript run-time environment built on Google Chromes V8 engine. 74, 76, 84

NPM Module manager for NodeJS, providing capability to install and maintain packages used in NodeJS applications. 84

Overall Simplicity All design aspects of a solution are as less astonishing and as much esoteric as necessary. 31, 34

P1 The msg systems ag internal solution for logging working hours. 48, 57

PAC Similar to the MVC pattern, with the addition that it is inherent hierarchically structured by piling PAC agents up. It serves on a higher level of abstraction than MVC. 3–5

PAC agent The Presentation-Abstraction-Control instances within a PAC architecture. 3

PEG Grammar to describe a formal language by providing a set of rules to recognize strings of that language. It is closely related to top-down parsing languages. 82

PEG.js Implementation of an efficient JavaScript parser generator for PEG grammars (http://pegjs.majda.cz/). 82

ROI Return on investment. 86

Run-time reflection Architectural pattern which is used for run-time analysis of a software system. It consists of four layers: Logging, monitoring, diagnosis and the mitigation layer. The logging layer is the information extraction point, to gather data from an application. This data is then processed by the monitoring and diagnosis layer and finally used by the mitigation layer to influence the run of the software. 20

Smalltalk-80 Object-oriented programming language designed by the Learning Research Group of Xerox PARC. 6

SPA Single-Page applications are characterized by the fact, that all resources necessary for their execution are retrieved at the initial page load. The are used to convey the user experience of a desktop application. 2, 23, 76

Strong Cohesion Strong relationship between functionalities within a single solution component. 38, 39

Structural Modularity Splitting of a solution into manageable structural components. 46

transpiler In contrast to a compiler, which usually translates higher level programming languages to lower level ones, a transpiler sticks to the same level. It can transform from one programming language to another or perform source code rewriting. 74

UI A User Interface (UI) is a mask presented to the user in order to enable interaction with a computer. 4–7, 9, 24, 25

UI Toolkit A toolbox providing an abstraction of the technology used to draw controls and basic User Interface elements. 23, 29, 30, 41, 87, 90

UICA User Interface Component Architecture. 37, 38

user interface element Visual unit in the composition making up the UI. 24, 32

W3C World Wide Web Consortium (W3C) is a consortium to standardize the world wide web. 29

WebSocket WebSocket is a full-duplex communication protocol over a TCP connection. It is intended to close the gap of server to client communication. 74–76, 82

B. Code Snippets/Appendix

```
peephole-constraint view_trait {
  rationale {
    The only points, where the component system touches the UI Toolkit
    are the view components. These components are used to
    render everything, from the topmost dialogs of the application
    down to layout components that combine several nested views
    into one comprehensive view.
  }
  condition {
    sourceType == "V"
    && source == origin
  }

  peephole-constraint observations {
    rationale {
      Views must only access their underlying presentation
      models through calling observe or value onto themselves.
    }
    condition {
      operation == "observe" ||
      operation == "value"
    }
    result PASS
  }
  peephole-constraint sockets {
    rationale {
      Views may provide sockets that other views can plug
      themselves into.
    }
    condition {
      operation == "socket"
    }
    result PASS
  }
  peephole-constraint plugin {
    rationale {
      Views can use sockets of underlying views to embed
      themselves into them.
    }
    condition {
      operation == "plug"
    }
    result PASS
  }
}
```

Listing B.1: Constraint of the View Component Trait (4.1)

```
peephole-constraint model_trait {
  rationale {
    We have several possibilities to store data within an
    application. We can use globally or locally scoped variables,
    member variables of component instances or outsource it
    to another storage layer. But when we want to make the
    data available within the Component Tree, which means that we
    can observe (2.3) and manipulate (2.2)
    it through component systems API, we have to
    declare a model (2.2) and pull the required data into it.
    In general, the model trait should be used whenever a
    component holds any kind of data, listed in the introduction
    of Chapter (4.2).
  }
  condition {
    sourceType == "M"
    && source == origin
  }

  peephole-constraint observations {
    rationale {
      The Model can use observe and value to perform its
      presentation logic and of course it has to define
      the model via the corresponding API method first.
    }
    condition {
      operation == "model" ||
      operation == "observe" ||
      operation == "value"
    }
    result PASS
  }
}
```

Listing B.2: Constraint of the Model Component Trait (4.2)

```
peephole-constraint controller_trait {
  rationale {
    A component with the controller trait is in
    most cases both, interface to the rest of
    the Component Tree and coordinator for its
    sub-tree (2.3).
  }
  condition {
    sourceType == "C"
  }

  peephole-constraint managing {
    rationale {
      The controller publishes events to and registers
      services for the rest of the application. It
      subscribes to events within its sub-tree for
      presentation actioning.
    }
    condition {
      source == origin &&
      (operation == "publish" ||
       operation == "subscribe" ||
       operation == "register")
    }

    result PASS
  }

  peephole-constraint model-com {
    rationale {
      A controller has to read and write data to its
      child model. Additionally it observes model
      members to react on changes.
    }
    condition {
      originType == "M" &&
      distance(source, origin) > 0 &&
      operation == "value" ||
      operation == "observe"
    }

    result PASS
  }

  peephole-constraint service-com {
    rationale {
      Each controller can perform calls to retrieve
      necessary information from a service component.
      It is also valid to perform calls on any child
      controllers.
    }
    condition {
      operation == "call" &&
      (originType == "S" ||
       (distance(source, origin) > 0 &&
        originType == "C"))
    }

    result PASS
  }
}

  peephole-constraint observations {
    rationale {
      The Model can use observe and value to perform its
      presentation logic and of course it has to define
      the model via the corresponding API method first.
    }
    condition {
      operation == "model" ||
      operation == "observe" ||
      operation == "value"
    }
    result PASS
  }
}
```

Listing B.3: Constraint of the Controller Component Trait (4.3)

```
peephole-constraint lean_widget {
  rationale {
    The pattern provides a connection between the UI Toolkit
    domain and the component system. It focuses on Loose Coupling
    between the view and the underlying structure to enable
    Headless Testing while simultaneously providing an
    abstraction of the deployed UI Toolkit for easy
    exchangeability.
  }
  condition {
    sourceType == "V"
  }

  peephole-constraint self {
    condition {
      source == origin
    }

    peephole-constraint model-com {
      rationale {
        Views must only access their underlying presentation
        models through calling observe, value or property
        onto themselves.
      }
      condition {
        operation == "observe" || operation == "property" ||
        operation == "value"
      }
      result PASS
    }

    peephole-constraint struct-com {
      rationale {
        Views are allowed to plug themselves into their
        environment.
      }
      condition {
        operation == "plug"
      }
      result PASS
    }
  }
}
```

Listing B.4: Constraint of Pattern 1: Lean Widget (5.1)

```
peephole-constraint shared_model_component {
  rationale {
    Via providing a way to map data slots of a model seamlessly
    to the storage required by overlying components, this pattern
    offers the possibility to merge a bunch of tiny models into
    one model component. This reduces the implementation effort,
    the run-time overhead and the complexity of the application.
  }
  condition {
    sourceType == "M"
  }

  peephole-constraint self {
    rationale {
      The model needs to observe itself for configuration
      changes of the adjacent view components and provide
      scoped properties for the dynamic bindings lookup. Of
      course it also has to define the model itself.
    }

    condition {
      source == origin
      && (operation == "observe" || operation == "property"
      ||  operation == "model")
    }

    result PASS
  }
}
```

Listing B.5: Constraint of Pattern 2: Shared Model (5.2)

```
peephole-constraint subtree_information_delivery {
  rationale {
    The Component Tree allows us to have communication bubbling
    up from each component. On its way, an event can notify
    observers and trigger actions as well as deliver data.
    But sometimes, we are faced with the necessity of sending
    data from one component downwards the hierarchy to child
    components or whole sub-trees. At this point, this
    pattern steps in by providing a clean and sophisticated
    way to relay information.
  }

  peephole-constraint spreading_event {
    rationale {
      The controller is allowed to publish the spreading
      event upon itself. The receiving component should
      also be a controller since it is the interface to
      the rest of the Component Tree.
    }

    condition {
      sourceType == "C"
      && source == origin
      && (operation == "publish" ||
          operation == "subscribe")
      && parameters.spreading == true
    }

    result PASS
  }

  peephole-constraint shared_member_model {
    rationale {
      The model has to define itself through the model method.
      Presentation logic is triggered by observing and
      manipulating its members. Furthermore, the scoping is
      done using the property method.
    }

    condition {
      sourceType == "M"
      && source == origin
      && (operation == "property" ||
          operation == "value" ||
          operation == "observe" ||
          operation == "model")
    }

    result PASS
  }

  peephole-constraint shared_member_receiver {
    rationale {
      The receiving component should be a controller as it is
      the interface to the rest of the Component Tree. The property
      method is used to lookup the model members that can be
      observed and manipulated.
    }

    condition {
      sourceType == "C"
      && source == origin
      && (operation == "property" ||
          operation == "value" ||
          operation == "observe")
    }

    result PASS
  }
}
```

Listing B.6: Constraint of Pattern 3: Sub-tree Information Delivery (5.3)

```
peephole-constraint rich_widget {
  rationale {
    Keeping all data and its logic together in one place
    reduces complexity and improves maintainability
    and understandability of the source code. Thus,
    the implementation of a dedicated model component for
    a rather extensive widget is advised. Of course
    Headless Testing is well supported since the
    view can be omitted and testing is performed on the
    sole model component.
  }
  peephole-constraint view {
    condition {
      sourceType == "V"
    }

    peephole-constraint self {
      condition {
        source == origin
      }

      peephole-constraint model-com {
        rationale {
          Views must only access their underlying presentation
          models through calling observe, value or property
          onto themselves.
        }
        condition {
          operation == "observe" || operation == "property" ||
          operation == "value"
        }
        result PASS
      }

      peephole-constraint struct-com {
        rationale {
          Views are allowed to plug themselves into their
          environment.
        }
        condition {
          operation == "plug"
        }
        result PASS
      }
    }
  }

  peephole-constraint model {
    condition {
      sourceType == "M"
    }

    peephole-constraint self {
      condition {
        source == origin
      }

      peephole-constraint pres-logic {
        rationale {
          Models are allowed to trigger their presentation logic
          and modify their members by using the observe and value
          method onto themselves.
        }
        condition {
          operation == "observe" || operation == "value"
        }
        result PASS
      }
    }
  }
```

```
}
```

Listing B.7: Constraint of Pattern 4: Rich Widget (5.4)

```
peephole-constraint managing_widget {
  rationale {
    Imagine a layout component L and an underlying component U.
    U plans to embed several components inside the layout provided
    by L. One way would be to issue create calls directly from U
    upon L to instantiate the necessary components. If presentation
    logic (2.3) is necessary for the layout, e.g. a tab layout,
    where only one tab can be visible at a time, then component U
    has to take care of the states of each created component, although
    it does not know how the layout is build up. Hence the best
    practice is to shift that responsibility to component L which
    creates child components and is able to manage them, as a result.
  }
  condition {
    sourceType == "C"
    && (operation == "state" || operation == "call"
    || operation == "property")
    && distance(source, origin) == 3
  }

  result PASS
}
```

Listing B.8: Constraint of Pattern 5: Managing Widget (5.5)

```
peephole-constraint non_managing_widget {
  rationale {
    It is not always necessary to have a full blown MVC triad
    (2.3) taking care of a simple layout job. To reduce
    complexity, the model and controller components are omitted,
    boiling the MVC triad down to a single view component.
  }
  peephole-constraint socket-creation {
    rationale {
      The view used for the static layout has to provide the
      necessary sockets for possible child components.
    }
    condition {
      sourceType == "V"
      && source == origin
      && operation == "socket"
    }
    result PASS
  }

  peephole-constraint coordinate-children {
    rationale {
      Optionally, controller components can create links to direct
      overlying views into the correct positions.
    }
    condition {
      sourceType == "C"
      && originType == "V"
      && operation == "link"
    }

    result PASS
  }
}
```

Listing B.9: Constraint of Pattern 6: Non-Managing Widget (5.6)

C. Bibliography

[A P77] A. Pnueli. „The temporal logic of programs“. In: *Proceedings of the 18th Symposium on the Foundations of Computer Science* FOCS'77 (1977), pp. 46–57.

[BD04] Bernd Brügge and Allen H. Dutoit. *Object oriented software engineering using UML, patterns, and Java*. 2nd ed. Upper Saddle River and NJ: Pearson Education, 2004.

[BM00] Andy Bower and Blair McGlashan. „Twisting the Triad“. In: *ESUG 2000* (2000).

[Bur92] Steve Burbeck. *Applications Programming in Smalltalk-80: How to use Model-View-Controller*. 1992. URL: `http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html` (visited on 04/15/2013).

[CKP00] Jason Cai, Kapila Ranjit, and Gaurav Pal. *HMVC: The layered pattern for developing strong client tiers: The hierarchical model eases the development of a Java-based client tier*. Ed. by JavaWorld. 2000. URL: `http://www.javaworld.com/jw-07-2000/jw-0721-hmvc.html` (visited on 04/20/2013).

[Enga] Ralf S. Engelschall. *ComponentJS API*. URL: `http://componentjs.com/api/api.screen.html` (visited on 11/13/2013).

[Engb] Ralf S. Engelschall. *ComponentJS Features*. URL: `http://componentjs.com/features.html` (visited on 04/15/2013).

[Engc] Ralf S. Engelschall. *User Interface Component Tree Architecture Pattern*. URL: `http://engelschall.com/go/EnTR-04:2013.12`.

[Engd] Ralf S. Engelschall. *User Interface Composition*. URL: `http://engelschall.com/go/EnTR-03:2013.12`.

[Enge] Ralf S. Engelschall. *User Interface Model/View Separation Architecture Pattern Comparison*. URL: `http://engelschall.com/go/EnTR-05:2014.01`.

[Fowa] Martin Fowler. *Flow Synchronization*. URL: `http://martinfowler.com/eaaDev/FlowSynchronization.html` (visited on 04/15/2013).

[Fowb] Martin Fowler. *Notification*. URL: `http://martinfowler.com/eaaDev/Notification.html` (visited on 04/15/2013).

[Fowc] Martin Fowler. *Observer Synchronization*. URL: `http://martinfowler.com/eaaDev/MediatedSynchronization.html` (visited on 04/15/2013).

[Fowd] Martin Fowler. *Passive View*. URL: `http://martinfowler.com/eaaDev/PassiveScreen.html` (visited on 04/15/2013).

[Fowe] Martin Fowler. *Presentation Model*. URL: `http://martinfowler.com/eaaDev/PresentationModel.html` (visited on 04/15/2013).

[Fowf] Martin Fowler. *Separated Presentation*. URL: `http://martinfowler.com/eaaDev/SeparatedPresentation.html` (visited on 04/15/2013).

[Fowg] Martin Fowler. *Supervising Controller*. URL: http://martinfowler.com/eaaDev/SupervisingPresenter.html (visited on 04/15/2013).

[Fow06] Martin Fowler. *Organizing Presentation Logic*. 11.07.2006. URL: http://martinfowler.com/eaaDev/OrganizingPresentations.html (visited on 04/15/2013).

[Gam95] Erich Gamma. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Reading and Mass: Addison-Wesley, 1995.

[Gre07] Derek Greer. *Interactive Application Architecture Patterns*. 2007. (Visited on 04/02/2013).

[HC95] Andrew Hussey and David Carrington. „Comparing two user-interface architectures: MVC and PAC“. In: (1995).

[HO07] Martin Haft and Bernd Olleck. „Komponentenbasierte Client-Architektur“. In: *Informatik-Spektrum* (2007), pp. 143–158.

[Inc] Google Inc. *AngularJS - Superheroic JavaScript MVW Framework*. URL: http://angularjs.org/.

[LC09] Martin Leucker and Schallhart Christian. „A brief account of run-time verification“. In: *Journal of Logic and Algebraic Programming* 78.5 (2009), pp. 293–303.

[LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. „Composing User Interfaces with InterViews“. In: *IEEE* (1989), pp. 8–22.

[OSV10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. 2nd ed. Walnut Creek and Calif: Artima, 2010. ISBN: 0981531644.

[Pan97] Panos Markopoulos. „A compositional model for the formal specification of user interface software“. PhD thesis. London: Queen Mary and Westfield College, 3.1997.

[Pix] Tom Pixley. *Document Object Model (DOM) Level 2 Events Specification*. URL: http://www.w3.org/TR/DOM-Level-2-Events/ (visited on 11/22/2013).

[Pot] Mike Potel. „MVP: Model-View-Presenter: The Taligent Programming Model for C++ and Java“. In: ().

[RB06] Andreas Rausch and Manfred Broy. *Das V-Modell XT: Grundlagen, Erfahrungen und Werkzeuge*. 1. Aufl. Heidelberg and Neckar: Dpunkt, 2006.

[W3C13] Dimitri Glazkov W3C. *Shadow DOM*. Ed. by W3C. 2013. URL: http://www.w3.org/TR/shadow-dom/ (visited on 09/09/2013).