Reengineering a Content Manager for Humanoid Robots with Web Technology

by

Eduard Gamonal Capdevila

Submitted to the Facultat d'Informàtica de Barcelona
in partial fulfilment of the requirements for the degree of

Master in Information Technology

at the

UNIVERSITAT POLITÈCNICA DE CATALUNYA

December 2013

Núria Castell
Tutor
Associate Professor

Daniel Pinyol
Supervisor
CTO at PAL Robotics

Reengineering a Content Manager for Humanoid Robots with Web Technology

by

Eduard Gamonal Capdevila

Submitted to the Facultat d'Informàtica de Barcelona
on Dec 18, 2013, in partial fulfilment of the
requirements for the degree of
Master in Information Technology

Abstract

Software products do not degrade with time or with external factors, but they are continually corrected and extended. Using third-party packages is a common practice and creates a dependency with a vendor, who might stop supporting a product. After a certain amount of time, changes become harder to implement and reengineering might be necessary.

This project aims to reengineer a content manager for humanoid robots with web technology at PAL Robotics in order to abandon the current Adobe Flash implementation. This software runs in the robot, displays content applications and handles user interaction. Users create contents with an external RIA. These might contain buttons, images or text, and have an XML representation. The content manager in the robot loads the content, interprets that XML in order to build a GUI with HTML, and interoperates with the robot's system from the browser.

This thesis formally addresses the reengineering of the software based on Pressman's work, in 6 successive steps: software inventory, documentation restructuring, reverse engineering, code restructuring, data restructuring, and forward engineering.

Firstly, it identifies the functional and non-functional requirements of the project. Secondly, it presents the specification emphasising the first three steps, including a conceptual model, system use cases and sequence diagrams. Thirdly, it describes the internal design of the system in the context of the last three steps. It starts by highlighting the system's architecture, its context and the software patterns. Then, it provides a static view with class and packages diagrams, a dynamic view with sequence diagrams and a physical view with a deployment diagram. Fourthly, it illustrates relevant parts of the implementation with code examples. Lastly, it outlines the testing strategy and implementation.

In conclusion, this master's thesis addresses the reengineering of the content manager and develops the new system with accuracy, creativity and consistency. It applies a systematic approach and uses proven techniques like software patterns and a widely-known architecture.

Any sufficiently advanced technology is indistinguishable from magic

Acknowledgments

It has been a long journey since the first year at this university. I would like to thank my parents and my sister for their enduring support, understanding and guiding.

This thesis is the work done at PAL Robotics. I owe many thanks to the team that helped me understand the system and a field, robotics, that was new to me. My colleague Eric Marcos and my director Daniel Pinyol guided and supported me specially in the first weeks of this project, and Dr Ricardo Tellez reviewed part of this document and provided valuable insights.

The generous support of my friends during the last weeks is greatly appreciated. They challenged my ideas and suggested changes that have made this document much clearer: Daniel Andersson, Oriol Arcas, Margaux Bigotte, Oriol Collell, Clara Cowley, Jordi Francès, Manuela Hürlimann, Roger Llorens, José Manuel López, Hèctor Manrique, Álvaro Martínez, Kamilla Nyborg, Marc Rodríguez, and Anna Rosich.

I would like to express my deepest gratitude to the professors in high-school and university that inspired me, showed me that hard work pays off, encouraged me to always pursue bigger challenges, and gave me the opportunity to take semesters at the University of Aberdeen (Scotland) and Northeastern University (Boston, MA).

Finally, I would like to thank my professor Núria Castell for her guidance and support while I completed this work.

Contents

List of Figures

List of Tables

Listings

API Application Programming Interface

CPS Continuation-passing Style

CRUD Create, Read, Update, Delete

CSS Cascading Style Sheet

CSS3 Cascading Style Sheet 3

DB Database

DbC Design by Contract

DOM Document Object Model

DTO Data Transfer Object

E2E End to End

GRASP General Responsibility Assignment Software Patterns

GUI Graphical User Interface

HTML Hyper Text Mark-up Language

HTML5 Hyper Text Mark-up Language 5

HTTP Hyper-text transfer protocol

IDE Integrated Development Environment

JS JavaScript

JSON JavaScript Object Notation

MVC Model View Controller

npm Node Package Manager

OCL Object Constraint Language

OOD Object Oriented Design

OS Operating System

QA Quality Assurance

RIA Rich Internet Application

ROS Robot Operating System

RPC Remote Procedure Call

SASS Syntactically Awesome Syle Sheets

SCSS Sassy CSS

SOA Service Oriented Architecture

SOLID Single responsibility, open/closed, Liskov substitution, interface segregation, dependency inversion

ssh Secure Shell

TDD Test-Driven Development

UI User Interface

UML Unified Modelling Language

URI Uniform Resource Identifier

URL Universal Resource Locator

UX User Experience

W3C World Wide Web Consortium

WWW World Wide Web

XML Extensible Mark-up Language

XP eXtreme Programming

Chapter 1

Introduction

Humanoid service robots of PAL Robotics have a touchscreen on their torso. Using a web application, users create content applications (e.g. buttons that trigger actions in the robot or a picture gallery) that can be displayed on the screen of the robots. The software in the robots depends on Adobe Flash and needs to be reengineered with web technology.

This document is a stand alone presentation of the work done at PAL Robotics, which belongs to a larger project: the Content Management System (codenamed *Flango*). More

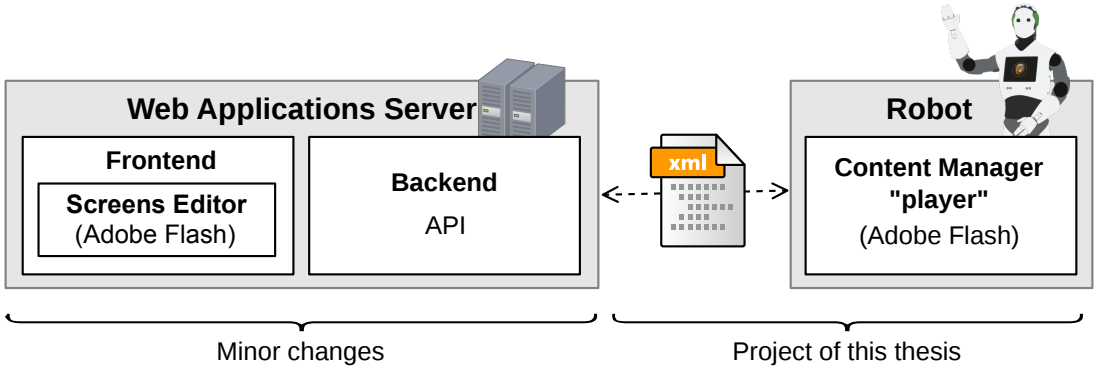

Figure 1-1: Overview of the Content Management System

specifically, **this thesis deals with the reengineering of Flango Content Manager, the part of the system deployed on the robot** that displays content applications on its the touchscreen (figure 1-1). **It does not reengineer the rest of the system**, like the web application (that uses Adobe Flash as well) to generate content applications, although

there might be minor changes to adapt their interfaces to the new technology.

It is desired not only to develop a practical solution but also to guarantee that the final output of the Flango Content Manager is the same that the old system generates.

This thesis is eminently a practical work, although the theoretical background has been extensively explored. Using web technology in a component of the robot is a novelty in the company. It is done to develop a sustainable product based on open standards (as opposed to the current implementation with Adobe Flash) that can be extended and reused for other robotic products that might be designed in the future.

This thesis documents the project of reengineering the Content Manager, in the robot, with web technology instead of Adobe Flash, namely HTML5, CSS3 and JavaScript (JS).

1.1 PAL Robotics

PAL Robotics is a company based in Barcelona dedicated to R&D of humanoid robots and robotic components. An international team of mostly mechanical, electronics and software engineers pushes forward the research on different fields, like speech recognition and generation, computer vision, walking, grasping, machine learning and navigation amongst others. The company has developed 6 humanoid robots until 2013: Reem-A, Reem-B, Reem-H1, H2 and H3, and Reem-C. This project targets Reem-H2 and H3, service robots with wheels and touchscreen.

Reem H2 and H3

Reem-H2 (figure 1-2a on the next page) and H3 (figure 1-2b on the facing page) are humanoid service robots featuring a screen on their torso. They have an autonomous navigation system, speech recognition and voice synthesiser, they can find their way in different settings and help or entertain people in a friendly way. They are intended for use in public places such

(a) Reem-H2 (b) Reem-H3

Figure 1-2: Reem-H Series

as hotels, malls, airports or museums.

Weight	90 Kg
Height	1.70 m
Battery life	8 h
Degrees of freedom	22
Payload	30 Kg (mobile base), 3 Kg/arm
Speed	4 Km/h
Computer	Core 2 Duo + Atom
Sensors	Microphone, stereo-camera, laser, ultrasounds, accelerometers and gyroscopes

Table 1.1: Reem H2 and H3

1.2 The Current System

The Reem-H series have a multi-modal interface: besides speech or joystick manual control, humans can use a touch screen to command the robot or obtain information. Reem-H can display multimedia contents like videos, facts about a company, robot features, language settings, etc.

The Contents Management System comprises *Flango Front-End* and *Flango Back-End* (both in Basestation), and *Flango Content Manager* (in the robot) (see figure 1-1 on page 1). The Basestation is an application server that hosts Flango, a Rich Internet Application (RIA) to manage contents and robots made with Django[1], a widely used framework for Python. Clients can create content applications with the Flango Front-End (Screens Editor), a tool built with Adobe Flash. When content applications are ready, clients can associate 1 application with n robots to display it in the touch screen (figure 1-5 on page 7 and figure 1-6 on page 8).

```
1  <fl:screen xmlns:fl='http://www.pal-robotics.com/flango-namespace'>
2
3     <fl:ui fl:height="190" fl:type="base-button" fl:width="200" fl:x="120.0"
          fl:y="28.0">
4        <fl:onclick>
5               <fl:action fl:type="external_call">
6                      <fl:param fl:name="palEvent" fl:value="doTask"/>
7                      <fl:param fl:name="subtype" fl:value="dipo"/>
8                  </fl:action>
9           </fl:onclick>
10
11       <fl:caption fl:lang="en">Map and Guide</fl:caption>
12       <fl:caption fl:lang="ca">Mapa i Guia</fl:caption>
13       <fl:caption fl:lang="ar">&lt;br/&gt;</fl:caption>
14    </fl:ui>
15
16 </fl:screen>
```

Listing 1.1: Example Screen XML

The elements of a content application (screens and entities) have an XML representation (listing 1.1), an approach similar to popular projects (e.g. Android, NetBeans) and even, in a sense, all RIAs made with Hyper Text Mark-up Language (HTML).

[1] `https://www.djangoproject.com`

The Flango Content Manager interprets the XML and displays the result on the screen (figure 1-4 on the following page).

A content application is essentially a set of screens, navigation, multimedia contents, and entities. The latter are domain objects that can be instantiated and represented in a view. This way a user can create an application that shows information about the company, include buttons to provide an easy way to give commands to the robot (e.g. "follow me", "shake hands"), display videos and picture galleries, etc. All of these components are localisable: They can be resized, repositioned, repainted... depending on the active language. Using other tools, clients can also associate sentences to screens and the text-to-speech system reads them aloud.

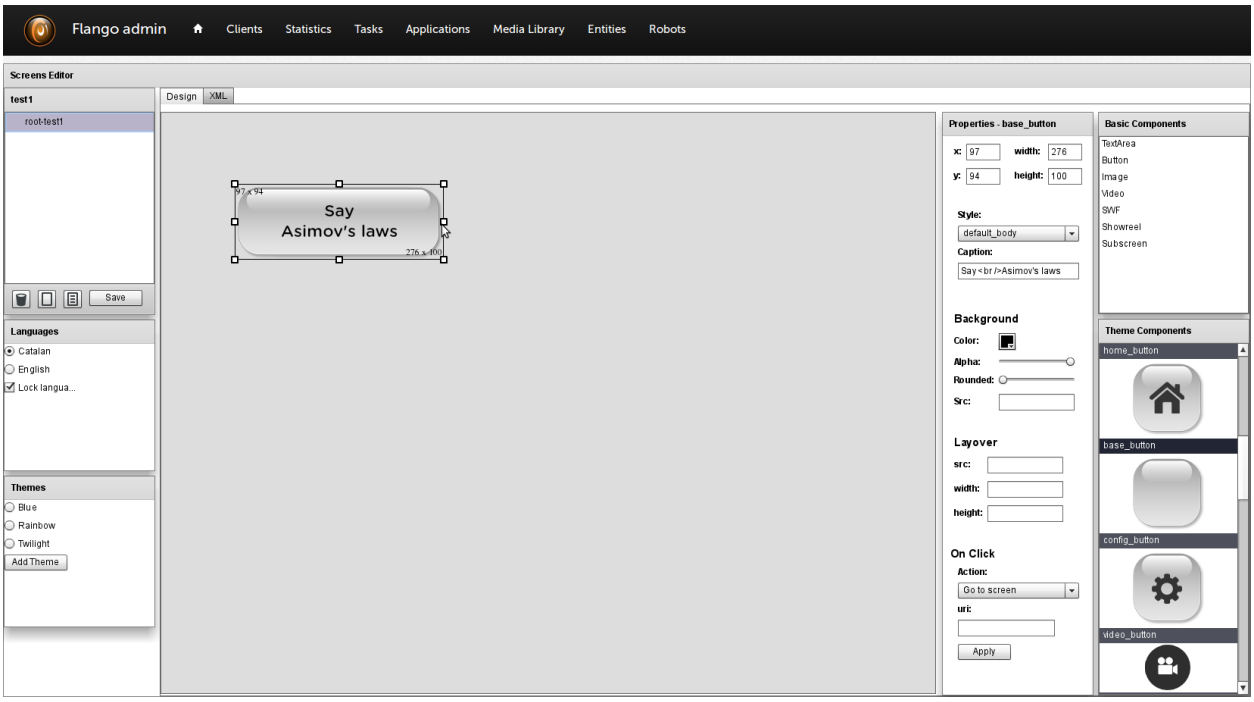

Figure 1-3: Screens Editor, an Adobe Flash application in Flango

Reem-H robots have a built-in software to display the content applications, the Flango Content Manager, which is also made with Adobe Flash. Initially, this software has no Graphical User Interface (GUI) or direct interaction with a person. It transforms XML files into a User Interface (UI) (figure 1-4 on the next page) and, finally, displays the screens and manages the user interaction (e.g. clicks on a button).

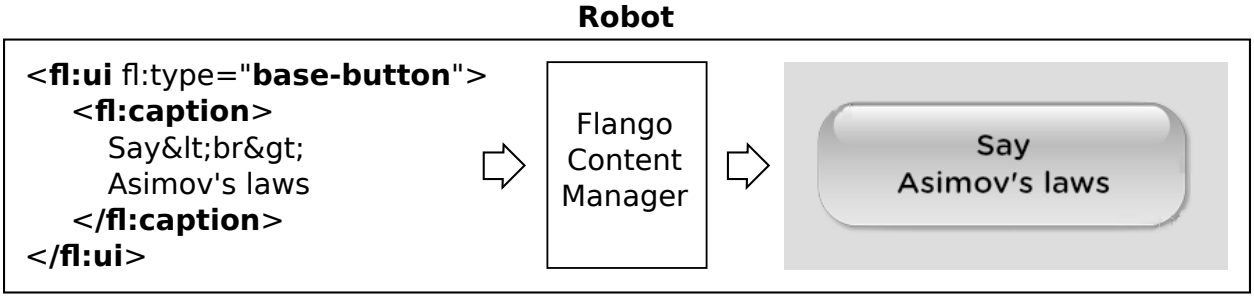

Figure 1-4: Transformation of XML into a GUI

1.3 The New System

The World Wide Web (WWW) was born as a document viewing platform and has evolved gradually into an application platform. First the web consisted of simple pages that contained structured text and some images. Navigation was done using simple links between pages. In a second phase, web sites became interactive: it was the time of animated graphics, browser plug-ins and JavaScript. More recently web sites have adopted features from traditional desktop software in RIAs [1].

For a decade, the Adobe Flash player plug-in has provided a platform to enable rich and interactive content in browsers. However, Adobe has made the updates exclusive to the Google Chrome browser [2]. The browser in the robot application, QtWebBrowser, uses this plug-in but will not receive updates from Adobe. PAL Robotics' systems need to adapt to this new setting. There are two solutions:

- Use Google Chrome instead of QtWebKit
- Use modern web technologies, either in QtWebKit or in another browser

The solution is rethinking the application to be a part of the robot's system and implement it with modern web technology. There is a number of technologies that could be used, but the Web allows the product to be easily portable, very extensible, open to many developers and sustainable as it does not depend on a single company. Hyper Text Mark-up Language 5 (HTML5) is perhaps the best known forthcoming standard in the Web and is typically combined with JavaScript and Cascading Style Sheet 3 (CSS3). It has become such a popular combination that major browser vendors provide support for the latest drafts of the World

Figure 1-5: Preview of a content application

Wide Web Consortium (W3C), like the audio API for HTML5 or the video support. A big number of frameworks with JavaScript have appeared not only client-side (e.g. BackBone, jQuery, Ext JS, Google Web Toolkit, Yahoo User Interface Library...), but even server-side (e.g. NodeJS). Moreover, new HTML5 capabilities and APIs are an opportunity to add features to this project.

The project of this thesis uses Google Angular[2], a MVC framework that lets developers *teach the browser new syntax*. Thus, XML files created with the Screens Editor can be interpreted natively in a browser. This has some advantages over Adobe Flash [3]:

- No use of third-party plug-ins
- Openness
- Reliability, security and performance
- Keeps power consumption lower (e.g. using the GPU to decode video or apply transformations and smooth transitions to elements on the screen)

[2] `http://www.angularjs.org`

Figure 1-6: The content application in the robot

- Touch-enabled: the new screen of Reem H3 is multi-touch. To enable gestures, heavy rewriting of the current Flash application is required.

1.4 Goals

This thesis is part of the specialisation of *Software Engineering and Information Systems*. The goal is to reengineer the Flango Content Manager with web technology. More specifically, the author has to gather the requirements from an existing system, plan in terms of scope, time and cost, make the system specifications, design and implement it using modern web technologies, use quality assurance tools and deploy it successfully in a Reem-H3 robot.

Regarding the project, the goals are:

- Reengineer the current system with web technology
- Develop a testing strategy to make it robust

- Make it compatible with non-reengineered parts of the system

1.5 Organisation of This Document

This document describes the system developed at PAL Robotics to display the content applications created with the Screens Editor on the Reem-H series. It does not cover other software on the robots or in Basestation.

This document describes the project as follows:

Related Work Provides a general overview of topics surrounding those of this thesis, enabling for a deeper understanding of the material at hand. It starts with a description of the reengineering process, followed by a section about web technology and, finally, a description of the methodology of the project – Test-Driven Development (TDD).

Project Management Contains information about the project from the point of view of management. Detailing on schedule, budget and risk analysis. Contains an assessment of the execution.

Requirements States the functional and non-functional minimum requirements, the constraints, and weights the use of off-the-shelf solutions.

Specification Defines what the application does in the context of a reengineering process. It deals with the first three steps: a software inventory, documentation restructuring and a description of the current system (with special attention to context, interoperability and interaction). Contains the specification of the new system: a conceptual model, use cases and the behaviour model, with sequence diagrams and contracts of system operations.

Design Describes the internal design of the software in the context of a reengineering process. It contains the last three steps: code restructuring, data restructuring and forward engineering. Detailing about the system architecture and the context and the patterns used. Contains a static view (class and packages diagram), a dynamic view (sequence diagrams) with examples of the most relevant operations and a physical view (deployment diagrams).

Implementation Holds technical details about the environment, the set-up, the integration in the robot and a larger system and meaningful examples of the technology used. Emphasises the implementation of the patterns described in previous chapters.

Testing Highlights the strategy to implement tests and the integration in the robot system.

Conclusion Presents the conclusions from technical, academic and personal point of view. Includes challenges and future work.

Appendix A Lists UI Components of the old system.

Appendix B Lists unit tests that define the behaviour of most components of this system.

Chapter 2

Related Work

This chapter is a short introduction to the main knowledge areas involved in the development of this project. It provides an overview of what has been done before in the field and the theory the project is based upon. It includes the motivation and the steps to reengineer an application, the evolution of web technology from simple documents to RIAs like the one of this thesis and, finally, the TDD method and the relationship with Design by Contract (DbC)

2.1 Reengineering

This thesis takes over the work of the past two years at PAL Robotics in content management. The initial situation is a system made with Adobe Flash and Django called Flango. The Screens Editor and the robot's Content Manager are Adobe Flash applications. Whereas the first one runs in a modern browser, on desktop computers, the Content Manager runs in the robot. The robot is a Linux system and the browser is a QtWebKit widget embedded in the Qt application that governs the behaviour. The Adobe Flash plug-in will not be updated and this part of the system has to be rewritten in a different technology.

Pressman explains the motivation to reengineer a software in [4]:

> Consider any technology product that has served you well. You use it regularly, but it's getting old. It breaks too often, takes longer to repair than you'd like, and no longer represents the newest technology. What to do? If the product is hardware, you'll likely throw it away and buy a newer model. But if it's custom-built software, that option may not be available. You'll need to rebuild it. You'll create a product with added functionality, better performance and reliability, and improved maintainability. That's what we call reengineering.

Unlike other products, software does not degrade with time or with external factors such as the inclemencies of the weather, power outages or intense use. Software packages are continually adapted, corrected, extended and improved. That can lead to unstable applications or unexpected side effects, even if the original product had been designed initially with the best practices. After a certain amount of time, changes become harder to implement. Sometimes the vendor of frameworks, libraries and third-party software used in a project stops supporting it and the only solution is substituting the components that depend on it. The product becomes unmaintainable in many aspects and reenginering is required.

Software reengineering comprises **6 activitie**s: **inventory analysis**, **document reestructure**, **reverse engineering**, **code restructuring**, **data restructuring** and **forward engineering**. Simple put, it is all about understanding the current logic of a program and modifying the three typical components: data, logic and view. **The project of this thesis is not modifying the current program but developing a new one with new technology, focusing on reverse engineering the original software**.

Doing an inventory of the software that belongs to the system is part of the definition of the project scope. Making a list of programs with the current status, dependencies and physical location helps on getting an overview of the project. Usually software has documentation that needs to be added to the inventory and, sometimes, restructured. After this, the reverse engineering process can start.

Reverse Engineering **Three dimensions** drive tasks in the reverse engineering activity: **abstraction level**, **completeness** and **directionality**. To understand what the software

does, abstraction should be as high as possible. This information can have many details or not (completeness) and can be created in one way (extracting it from the source code) or two-way (feed information to the reengineering tool). This project does not use any computer-assisted reengineering tool.

Understand current program A program normally has **input data**, **logic** that processes it and a **view**: The first activity is understanding the processing. Code can be examined at several **levels of abstraction**: **system**, **program**, **component**, **pattern**, **statement**. Some **key aspects** include the **interaction**, **interoperability** or the **context**. Data structures, classes, schemas in databases, etc have to be identified and possibly redesigned in the next steps. Finally, the UI has to be analysed as well. There are issues related to presentation of data, interaction, usability, etc that have to be identified. This project focuses on system and program levels of abstraction and on interoperability. The software does not have any views or databases to be reconsidered. However, it does have some formats server-side (in the backend) that are candidates to be part of this process (e.g. definition of settings with XML might have to become JavaScript Object Notation (JSON) or have a different structure).

Restructure code and data Sometimes the solution is restructuring the current program. The project of this thesis skips this part because a new application is being developed using the output of the first task: understand the current program Restructuring the code might not be enough in most cases. It can fix immediate issues but it is not a long-term solution. However, when doing so, TDD is useful because tests break immediately and often, specially if they are running in the background or periodically. To improve the software design both data and architecture should be restructured. TDD can provide robustness to this process: if there are tests available, the new design can be proven to work, at least, for the example cases. If not, the solution can be built incrementally with tests that define the requirements of the system (extracted in the first activity, thus using them as an specification.

Forward engineering There is a number of options when it comes to apply changes in the current program. For example, apply patches, redesign parts of the software or all of it. Depending on the circumstances, completely redesigning a software might be costly but, in the long-term, it might be a cost-effective solution. The project of this thesis redesigns the software to incorporate new practices, new technology and, in the future (out of the scope of the thesis), additional requirements. The outcome of this activity is a new software that replaces the current implementation.

2.2 Web Technology

The WWW plays a central role in many people's lives. The first drafts date back to the 1980s, the first web browser prototype by Tim Berners-Lee was built in December 1990 and the Mosaic browser was completed in 1993. After that, Netscape, Mozilla Firefox, Microsoft Internet Explorer, Google Chrome, Safari, Opera, Konqueror and many other browsers appeared, along with some nowadays popular websites (e.g. Amazon, 1995).

2.2.1 From Plain Documents to Applications

The Evolution of the Web Berners-Lee said [5]:

> HyperText is a way to link and access information of various kinds as a web of nodes in which the user can browse at will. Potentially, HyperText provides a single user-interface to many large classes of stored information such as reports, notes, data-bases, computer documentation and on-line systems help.

When the WWW was started in 1990 it was intended to be a document viewing platform. It has evolved in at least **three phases** [1] [6]:

1. **Plain text** files, forms, possibly static images and links.
2. **Programming capabilities** in the browser, plug-ins and animated graphics. Web pages became interactive and client-server communication increased in complexity.

Adobe Flash (at that time Macromedia Flash), ShockWave, Java, QuickTime and a few others allowed the inclusion of multimedia contents or even programs (e.g. Java Applets). Server-side and client-side technology was combined to design new contents, interaction and features.

3. **Applications in the browser**. Web pages are not just documents: they have complex user interaction, they do not require full page refresh and typical use cases have evolved. A big part of applications is executed client-side, taking advantage of the computing capabilities of modern hardware and decreasing the load in servers. In some cases, the application also needs huge server-side resources and uses strategies like cloud computing (e.g. Amazon EC2) to scale up.

Today the web is not only about viewing documents but about world-wide sharing, collaboration and interaction, possibly in real time. Web browsers have become a widely-used platform for software applications. Video editors, spreadsheets, calendars or 3D games used to run exclusively on desktop computers. Today, they run in a variety of devices and some of them even do it in a browser. There are 3D engines ported to Web, real time collaboration, full HD video support without plug-ins and many more features built-in a system, the browser, that is not an ideal execution environment for desktop applications.

RIA RIAs are neither web services or web pages. They are software systems based on technologies and standards of the W3C that provide web specific resources such as content and services through a web browser [7]. They are typically designed as single page websites.

They all have some common characteristics. Amongst others:

- The product
  - **Content** is the core.
  - **Hyper-text**: non-linearity is a main distinction to traditional software systems. There are many ways of landing on a certain page. This can lead to disorientation or cognitive overload for users.
  - **Presentation** aesthetics, usability and interaction are closer to a desktop application than to a web page

- Use
  - **Globality**: Spontaneity and multiculturality of users. Web applications are publicly available.
  - Quality suffers from **unknown network** characteristics
  - **Multiplatform delivery** involves having different devices, browsers and degrees of functionality and performance
  - **Intense network usage**. Remote calls are minimised. The use of software patterns like remote façade, remote proxy, Data Transfer Object (DTO) or Remote Procedure Call (RPC) is frequent
- Evolution
  - Continuous **change**
  - **Competitive pressure**
  - **Fast pace development**

The project of this thesis is, in a sense, both a RIA and a RIAs generator. It works in the browser with web technology and has an intense use of the network to fetch the model or other components to build the screens. The rendered application works in a browser, has an intense use of the network as well and is content-centred. Despite the fact that it works in the specific context of a robot, globality is still an issue. A typical scenario would be a fleet of robots in a conference: users from many different cultural backgrounds can use the application at any time.

2.2.2 Isolation of Applications in the Browser

Many of the websites in the aforementioned third phase contain substantial amounts of client-side code. At the end of the day, a RIA is a distributed application. In the past, the client-side part was thin and simple, whereas today's application have complex logic delegated to client nodes with local storage, hardware-accelerated components, web sockets and other advanced capabilities. This project uses some of them to, for instance, communicate a browser with the robot or to decrease the system load using hardware-accelerated Cascading Style Sheet (CSS) properties.

In spite of the fact that RIAs have grown in complexity and now demand more resources, browsers architectures still do not provide sufficient isolation between concurrently executing programs. A similar problem occurred in early operating systems (e.g. MS-DOS), before processes appeared [8].

Most of browsers have a monolithic architecture with poor isolation between web application instances. Chromium, however, implemented an architecture based on Operating System (OS) processes. Another way of isolating web applications is running them in a plug-in container.

In the first phase, the project of this thesis uses QtWebKit, the port of WebKit for Qt, to display the application in a Qt window and communicate with the software in the robot. The second part uses Google Chrome to meet the technology requirements. QtWebKit components comprise the WebCore and the SquirrelFish Extreme JavaScript engine , which compiles JavaScript into native machine code. It is compatible with Adobe Flash Player but because it will not receive more updates in the future, it will not be used any more. The Adobe Flash plug-in is a black box isolated from other elements in the Document Object Model (DOM). Isolation is not an issue in this project because there is only one application running at a time and, even in a normal use of a browser, with many websites running at once, applications do not have conflicts.

2.3 Test-Driven Development

The software written for this thesis is executed in a complex environment. Testing it properly and ensuring the quality is crucial to avoid unexpected behaviour in one of the most visible parts of this system, the screen. Crashes and UI flaws can lead to triggering wrong external calls to other components of the system (e.g. displaying a Qt window different than expected, starting an action with bad parameters...), blocking access to features or seriously affecting the User Experience (UX).

Reengineering this system has, at least, two sources of potential errors: undocumented

features in the original application and integration of the new software in the current system.

TDD is based on having numerous and very short iterations with these steps:

1. Add an initial (failing) test
2. Run all tests and see if the new one fails
3. Write the minimum amount of code to pass the test
4. Run the automated tests and see them succeed
5. Refactor the code. Conform to standards and best practices

As opposed to classic development methodologies (e.g. waterfall, where tests are done in the last place), with TDD tests go first. Specification and documentation of the system are not artifacts created before the implementation but the tests themselves.

Testing earlier and heavily has benefits over the classical approach:

Efficiency Defects (and their causes) are identified earlier.

Robustness The system is more reliable and stable. The Quality Assurance (QA) process is stricter and easier to maintain.

Maintainability Small fixes can introduce new errors in the system. Testing continuously and producing the minimum amount of code to implement a feature reduces this.

Extensibility eXtreme Programming (XP) embraces change and makes it is easier to adapt to changes in the requirements. During the development, the original software keeps adding features and bug fixes.

This methodology is specially useful in the implementation of this project for two reasons: the Google Angular framework is extremely friendly with unit testing and End to End (E2E) tests thanks to Karma Runner, and the company has the infrastructure to incorporate the project in a continuous integration system with Jenkins.

Chapter 3

Project Management

This chapter describes the planning of this project in terms of scope, schedule, cost and risk. The execution and deviations are shown at the end of this chapter.

3.1 Scope

The system has three parts: the Flango Back-End, the Flango Front-End (with the Screens Editor) both at Basestation, and the content application in the robots (the robot's Content Manager figure 3-1).

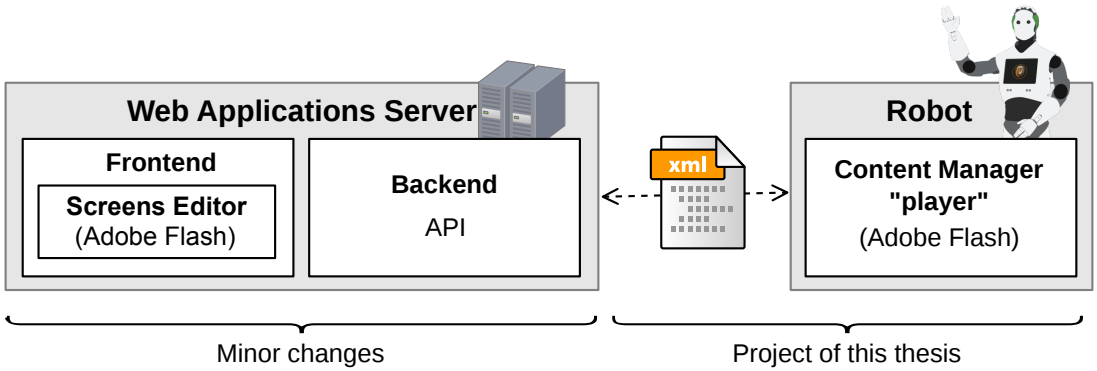

Figure 3-1: Boundaries and context of this project

This project reengineers the robot's Content Manager with web technologies but not the back-end and front-end, that remain in Django and Adobe Flash. The input of the new

Content Manager are content applications, generated with the Screens Editor. The output is the screen rendered with web elements. Content applications have configuration files that define the theme, the location of binary resources (images, videos, etc), the default language and other settings. This project can read and use the configuration files. This project accepts valid syntax generated with the Screens Editor for a comprehensive set of components (`back-button`, `base-button`, `image`, `video`...) and their properties (`width`, `x`, `y`, `background`...), and generates HTML5 that displays them as they are defined in templates.

Developing an intense testing strategy is part of this project. The final artifact is this master's thesis that describes the system and the rationale that guides all decisions.

3.2 Schedule

This project has 4 phases that match the 4 typical groups of processes: initiating, planning, executing (and monitoring + control) and closing. The execution has 3 deliverables: viability and proof of concept, iteration 1 (basic program) and iteration 2 (extended program).

Phase 1: Initiating

April, 1 – April 4 The initiating processes determine the goals of the project and the scope. This stage was partially done before the beginning of this thesis to ensure its viability in the company. It was agreed that the project would reengineer the current system using web technologies (see section 3.1 on the previous page)

Phase 2: Planning

April, 5 – April 6 The first draft of the plan is a rough estimation of the tasks length and a definition of the big milestones. Because there is only one engineer there are no concurrent tasks and free floats are always zero. Other activities of this phase include an estimation of

the resource requirements for upcoming tasks, developing the budget (section 3.3 on page 23) and risk planning (section 3.4 on page 23). See the Gantt chart in figure 3-2 on the following page.

Phase 3: Execution

April, 7 – August 28 The execution phase focuses on building the product. A web engineering process should be incremental, with frequent changes and short iterations [7].

Milestone	Deliverables
Proof of concept	Research on frameworks and tools to develop the project. Choose one. Build a minimal working example to illustrate key aspects of the project.
Iteration 1	Build a prototype comprehensive in number of features but simple in the implementation to validate the technology.
Iteration 2	Extend the prototype of Iteration 1 to complex cases.

Table 3.1: Milestones and deliverables of the execution phase

This phase has 3 big milestones, one per iteration (table 3.1). The outcome of each iteration is a product with a set of stable features and the corresponding unit tests which, in turn, serve for the purpose of documentation.

Phase 4: Closing

August, 29 – September 14 Prepare package with a stable set of features, presentation and documentation.

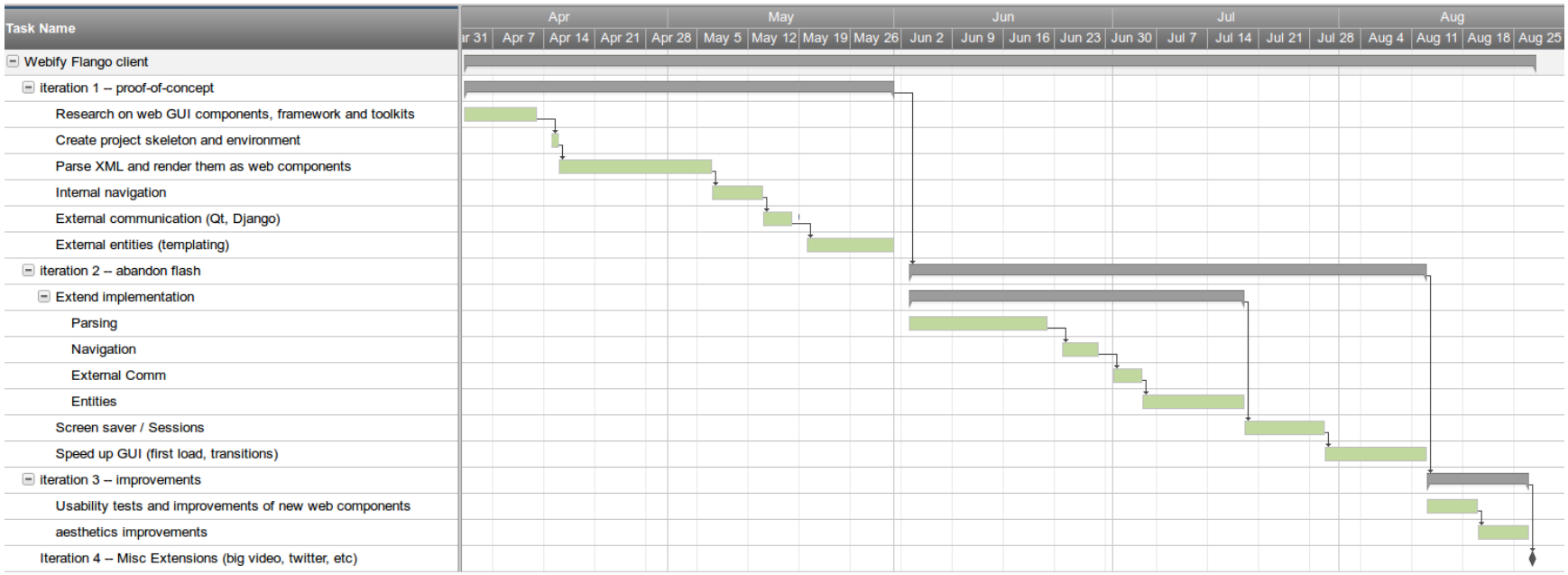

Figure 3-2: Original Gantt Chart

3.3 Cost

The estimated length of the project is $95 days \times 8€ \times 8 hours = 6080€$

Labour	6720€
Hardware	Desktop Computer (amortized) 250€ 22 Monitor (amortized) 50€ Reem H3 testing time 150€
Software	JetBrains WebStorm license 89.10€ GNU/Linux 0€
Overall	6619.10€

Table 3.2: Budget

There are some infrastructure related costs excluded from the budget because they are part of the office maintenance monthly expenses. This is the case of subversion servers, network, power, backups, etc.

3.4 Risk Analysis

1 Schedule: deviation

There might be urgent needs in the company, like demos to potential customers or hot bugs related to content applications

Mitigation Plan Some of the needs are unpredictable. However, preventive maintenance and good documentation of bugs in tickets can reduce the chances that this risk will occur.

Action Plan Do the urgent task and reschedule the project

2 Resources: deviation

The engineer might have to dedicate resources to other tasks in the company unrelated to this project.

Mitigation Plan Coordinate action monthly with the managers and track deviations and progress weekly. Avoid dedicating less than 50% of time to this project.

Action Plan Limit scope and extend deadlines to have the expected number of hours dedicated to this project.

3 Resources: infrastructure crash

The development system might die

Mitigation Plan Use version control systems and do frequent and small commits. Delegate backups to the IT team.

Action Plan Restore backups

4 Resources: testing environment not available

Most of the testing can be done with regular computers. However, final tests have to be conducted with real hardware: ReemH3-1 or 2 and Reem-H2, which are frequently booked to go on tour, perform shows, demos or development.

Mitigation Plan Book 3 days in advance if testing is critical.

Action Plan Share the robot with a team that does not need the media computer of the robot, where this project is hosted.

5 Personnel: Application expert leaves the project

There is only one application expert, who might leave the company. Nobody else has advanced knowledge on the internals of the current implementation.

Mitigation Plan Build a common agenda with the expert and document critical and complex parts of the current application. Find experts in similar technology in the team.

Action Plan Reach him on-line, hire him as consultant, take conflicting features out of the scope of the project.

6 Quality: critical use cases not implemented

One or more critical use cases can not be implemented

Mitigation Plan Verify that the technology allows implementing them. Build a prototype.

Action Plan Find a workaround. Assess the importance of the affected use cases and adapt to change: decide if changing the specification of the use case will fix this.

7 Quality: look and feel of the current theme can not be implemented

The current look and feel and interaction is designed for Adobe Flash. It might not suit the way web technology works.

Mitigation Plan Verify that the technology allows it with a prototype.

Action Plan Change the external design to balance the needs of the technology and the customer

8 Scope: Non-intuitive implementation with web technology

Current implementation is designed for Adobe Flash. Some of the use cases to implement might not fit well with the new technology

Mitigation Plan Analyse and discuss use cases with the author of the old implementation

Action Plan Explore ways to change the implementation of the use case in the Screens Editor (which generates the input for the new program) to fit in the new design

9 Inadequate technology

The chosen technology is inadequate: it does not allow a complete reengineering of the application (100% of chosen use cases), performance is more than 10% lower, it is not reliable and fails in more than 3% of action executions

Mitigation Plan Assess at least 2 solutions before implementing a prototype. Implement a prototype with basic cases of at least each critical feature.

Action Plan Stop the development and evaluate a different technology.

10 Tool: Flango or Screens Editor do not perform as advertised

The current implementation of the backend or the screens editor might have bugs. Some undocumented features or XML elements might be misleading (e.g. The normal behaviour for the QR component is generating a QR code. `<ui type="qr" src="path">` does not generate a QR code, but embeds an image that already exists instead.)

Mitigation Plan Contact the application expert before implementing the feature.

Action Plan Fix the backend (Python) or ask/hire the application expert to fix the screens editor (Adobe Flash)

11 Angular or dependencies do not perform as advertised [1]

The chosen framework and libraries have young maturity and might contain bugs, be subject to rapid changing cycles and new versions might break older versions

Mitigation Plan Use stable versions

Action Plan Investigate the bug, isolate it, create a minimal full working example and, if possible, open a pull request with a fix in Github or notify the error to the project managers.

3.5 Execution

The project was planned initially to start on April 4 and end on September 1 with full-time dedication: 760h. However, some situations caused a first delay and the ending day was pushed to October 10. Another delay appeared and the final date was moved to December 18. the overall number of hours remained approximately the same, 870h (+14%). Delays were caused by changes in the dedication time (with some weeks working only 20% of the

[1] Added after specifications were defined

time on this project) or the appearance of risks. This section summarises the execution of the project, the risks that appeared and the actions taken, the delays and the final result.

Scope

The project had initially 2 iterations (table 3.1 on page 21). Whereas the first iteration was completed in time, in the second iteration entities were removed from the scope in order to deliver the project in time. Parsing of deprecated components was also removed from the scope.

Risks

- **Schedule and resources: deviation**. The company had urgent tasks and it was impossible to work full-time on this project. Among other delays, development was stopped for 2 weeks in June, 4 days in late July, and one week per month until December. The development finished on the last week of November and this document was finished on December 8. The solution was rescheduling all iterations and cutting the scope.
- **Personnel: Application expert leaves the project**. The engineer that developed the old system left the company in September. Since then, only minor bugs have appeared and they have been fixed without contacting him. In September he had documented a big part of the project.

Cost

The final length of the project is $109 days \times 8€ \times 8 hours = 6976€$

Labour	6976€
Hardware	Desktop Computer (amortized) 250€ 22 Monitor (amortized) 50€ Reem H3 testing time 150€
Software	JetBrains WebStorm license 89.10€ GNU/Linux 0€
Overall	7515.10€

Table 3.3: Final cost

Chapter 4

Requirements

Requirements can be gathered using interviews with customers, users and stakeholders, with observation, questionnaires, prototyping or even from an existing application. In this project requirements are obtained mainly from the current implementation of the software (section 2.1 on page 11) and from interviews with the developer of the old version. Some features, either deprecated or with low priority, are left out of the requirements list due to time constraints.

4.1 Functional Requirements

Ready-made Solutions

Since this is a very specific software, there are no off-the-shelf solutions available that meet the requirements of the project. However, there are libraries and frameworks that can be reused:

1. JavaScript frameworks
    - MVC: AngularJS, Spine, Ember.js, Knockout.js, Sprout, Google Closure
    - Oriented to Web Components: AngularJS, Polymer
    - BackBone.js

- Testing: Jasmine, qUnit, phantomJS

2. JavaScript libraries
   - jQuery and plug-ins: Bigvideo.js, jQuery-ui, jquery-mobile , jquery-kinetic
   - Hammer.js, jGestures , iScroll, swipe.js
3. CSS Tools: LessCSS, SASS

Constraints

Certain constraints were defined before the beginning of the project:

1. Technology: the project has to be implemented using modern web technologies (HTML5, JavaScript, CSS).
2. Legal: It has to be ready to be open source

Functional Requirements

1. **Render UI Components with HTML**. UI components, currently defined with XML, have to be eventually transformed to HTML so that a browser can render them.
   (a) Support for **multiple languages** via the parameter `lang`.
   (b) **Components**
      i. **Basic components** `screen`, `subscreen`, `textarea`, `img`, `button`, `video`, `showreel`, `swf`, `layout`, `qr`. Exclude: `group`, `group_button`.
      ii. **Theme components**.
      iii. **Properties inline or in tags**. `x`, `y`, `width`, `height`, `scrolly`, `scrollx`, `loop`, `loopy`, `loopx`, `src`, `style`, `caption`.
      iv. **Action** properties `onclick`, `onload`, `action`, `param`.
2. **Themes**. The current version has a default theme, a set of UI components that extend the basic ones and provide a certain look and feel. All theme components are eventually implemented with a basic component, e.g. a `back-button`, a `home-button`, a `base-button`, etc.

3. Internal **navigation** (subscreens). Subscreens are containers whose content is determined by the Uniform Resource Identifier (URI).
4. **Settings** management. There are three configurations to load: generic settings (default language, paths...), application specific settings (current language, current theme...) and structure of the application (a graph that defines an *id* and a URI for each node (screen)).
5. Basic support for **Entities**.

4.2 Non-Functional Requirements

1. **Target browser**: QtWebKit (Qt 5.0.x) for the first phase, Google Chrome 28 for the second.
2. **Extensibility**: The software has to be extensible. The design has to allow adding new features in the future.
3. **Robustness**: the software has to be reliable and has to be delivered with a comprehensive test suite. It should be compatible with Jenkins and continuous integration.
4. **Hardware**: the software has to perform smoothly in Reem H3 and a multitouch screen.
5. **UX**: the time to change to another screen in the rendered content application should be less than 0.5s. Media contents (images, videos) have to be ready in less than 1s after a screen change. Rendered components should be aesthetically pleasant.
6. **Interoperability**: it has to interoperate at least with:
    - **Backend**: there is an API in a Django backend that provides settings and serves the contents
    - **RobotBehaviour**: the Qt program that runs in the robot and governs the behaviour. Sometimes the touchscreen displays Qt windows, sometimes it displays the content applications.

Chapter 5

Specification

The specification of the program defines **what it does** rather than how. This chapter presents the specification of the system taking into consideration general software engineering principles, the models proposed by Larman in [9] (at least conceptual, use case and behaviour models) and Meyer's DbC. For the particular case of this thesis, the specification takes into account reengineering principles and procedures and TDD as a tool to define what the system does as opposed to a simple testing technique.

Reengineering software comprises 6 activities (section 2.1 on page 11):

1. Inventory Analysis
2. Documentation Restructuring
3. Reverse Engineering
4. Code Restructuring
5. Data Restructuring
6. Forward Engineering

The outcome of these activities derives part of this specification. For example, use cases are extracted from the current system. The first three focus on what the system does and are used in this chapter, whereas the last three are tightly coupled to technology and are expanded in chapter 6 on page 57.

This chapter starts by describing the **current system**: the software parts (**inventory analysis**) and what it does (**reverse engineering** to understand the system with special attention to **context**, **interaction** and **interoperability**). It is followed by a description of what the **new system** has to do: the **specification** comprises a **conceptual model**, a **use case model** and a **behaviour model**.

5.1 Inventory Analysis

A list of software in the organisation makes candidates for reengineering appear. The scope of this thesis is reengineering Flango Content Manager in the robot and it is a decision taken before starting this work.

1 Flango Content Manager (robot)

Year	May 2011
Substantive changes	5 big milestones
Last substantive change	August 2013
System(s) where it resides	Reem H2 and H3
Applications to which it interfaces	Flango Backend; robotBehaviour
DB it accesses	SQLite in the robot (exclusive use)
Projected longevity (in years)	Until September 2013
Projected number of changes (36 months)	0
Annual cost of maintenance	1 software engineer full-time

Table 5.1: Software Inventory: Flango Content Manager (robot)

However, Flango Content Manager uses other software (tables 5.2 and 5.3) that might have to be modified or even reengineered in the future to suit the needs of the new technology in the project.

When it was decided to reengineer Flango Front-End it was not very old but there were major changes already contemplated for the application (e.g. abandon Adobe Flash), that would require big commitment of effort and that would not converge to a sustainable situation. Reengineering costs are lower than the cost of making changes plus the cost of maintenance.

2 Flango Front-End– Screens Editor

Year	December 2012
Substantive changes	4 (first version; some new features; bug fixing)
Last substantive change	August 2013
System(s) where it resides	Basestation
Applications to which it interfaces	Flango Frontend (robot); Flango Backend
DB it accesses	flango (postgresql through the Backend)
Projected longevity (in years)	At least until January 2015
Projected number of changes (36 months)	Bug fixing
Annual cost of maintenance	1 software engineer full-time

Table 5.2: Software Inventory: Flango Front-End– Screens Editor

3 Flango Back-End

Year	October 2012
Substantive changes	3 (first version; statistics; new features)
Last substantive change	August 2013
System(s) where it resides	Basestation
Applications to which it interfaces	Flango Frontend (robot); Flango Screens Editor
DB it accesses	flango (postgresql)
Projected longevity (in years)	At least until January 2015
Projected number of changes (36 months)	Unspecified number of new features and bug fixing
Annual cost of maintenance	1 software engineer full-time

Table 5.3: Software Inventory: Flango Back-End

5.2 Reverse Engineering

Reengineering a software involves reverse engineering, **understanding what the system does**.

There are three key issues in understanding the current system: **abstraction level**, **completeness** and **direction**. This specification uses a system-level abstraction, only takes into account interaction with external elements, and is one-way (from current system to the specification of the new system). Other levels of abstraction do not apply here to keep this specification technology-agnostic. This section focuses on what the current version of the software does and what the new version has to do. It offers some technology-dependant

details to understand the steps to reach the goal and the operations that Basestation exposes.

Details of how this software does the same with a new technology are in chapter 6 on page 57 and chapter 7 on page 111

5.2.1 Context

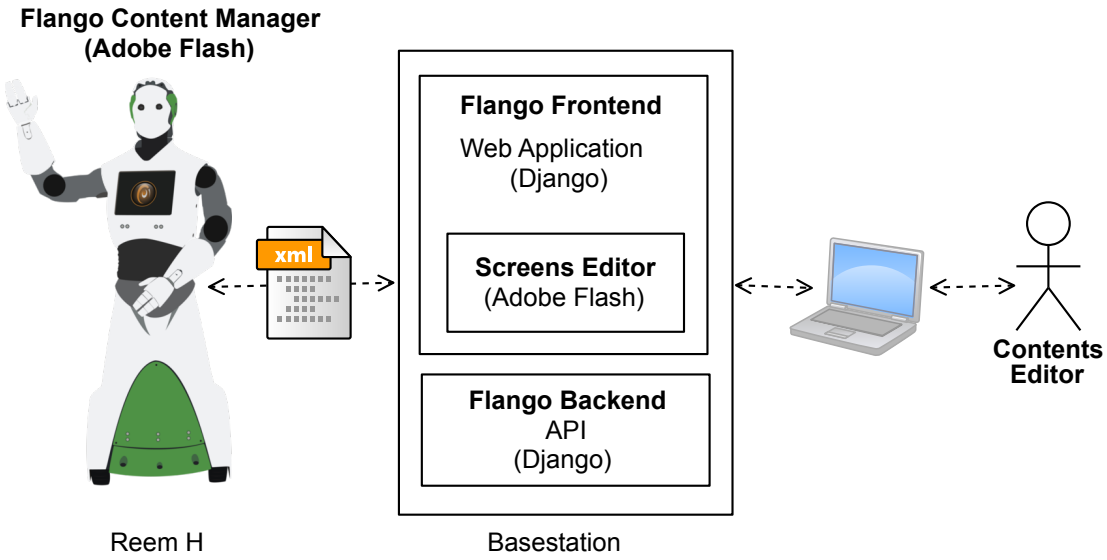

Figure 5-1: Reverse engineering: Current context

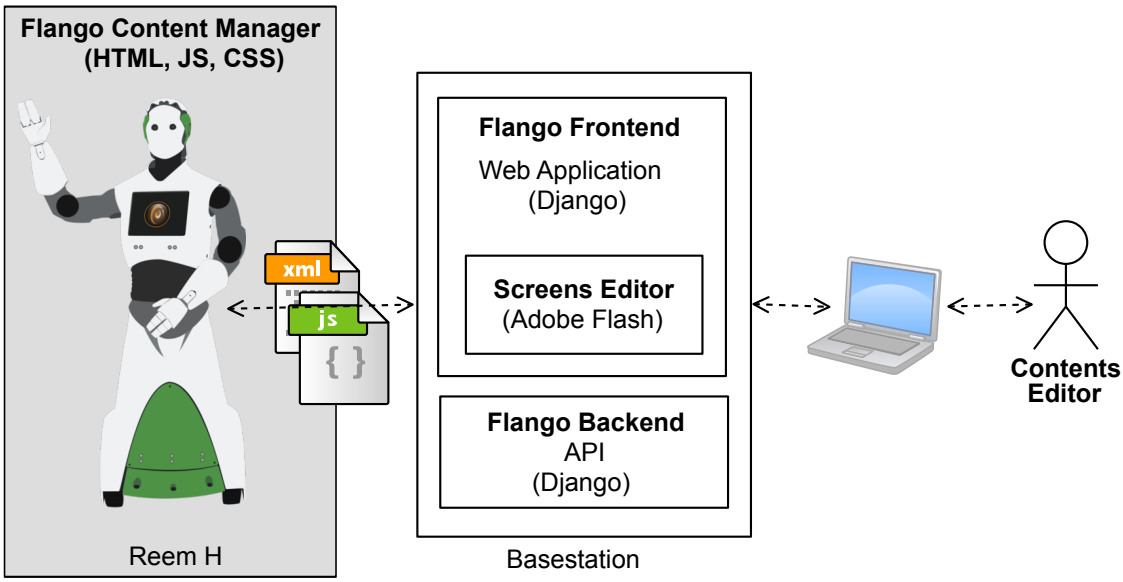

Figure 5-2: Reverse engineering: New context

Reem-H3 has about 600MB of specialised code. The program that glues all software parts and serves as an interface to the hardware is **robotBehaviour**, developed with Qt. A window in this program contains a QtWebKit widget, the embedded web browser that loads the contents for the touchscreen. Contents are fetched from Basestation, a server that hosts Flango, a web application that lets users edit screens (with the Screens Editor), add media contents and synchronise them with robots (figure 5-1 on the facing page). The Flango Back-End has an API to serve the content applications to the robots. The project of this thesis is reengineering the Flango Content Manager, in the robot (figure 5-2 on the preceding page).

5.2.2 Interaction

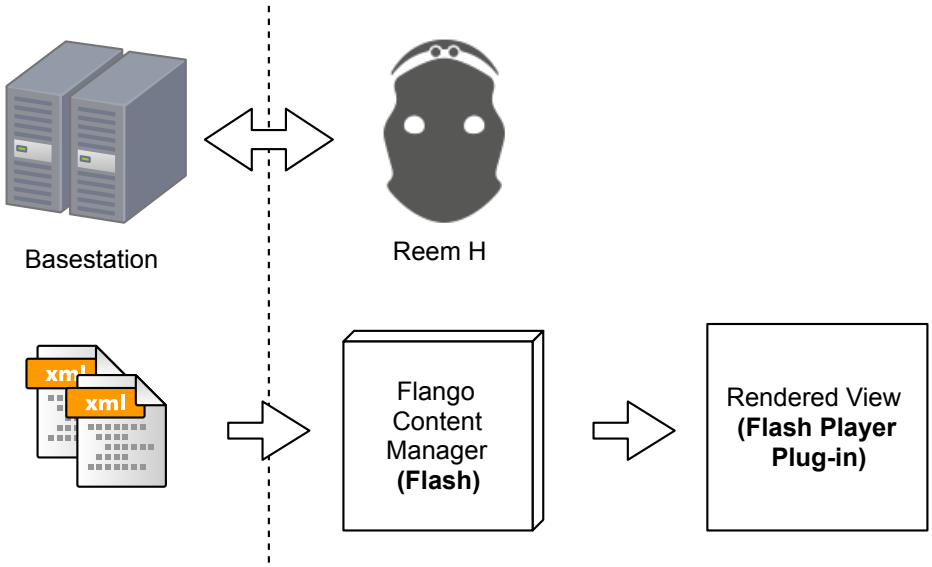

Figure 5-3: Reverse engineering: Current interaction

The Content Manager's input is XML code and the output is an HTML single page application (figure 5-3 and figure 5-4 on the next page). The general boot sequence is:

1. The robot boots
2. robotBehaviour fetches generic settings (XML) from Basestation
3. The Content Manager uses the settings to decide which application to load, the language, the theme, etc. It requests application-specific settings and the application

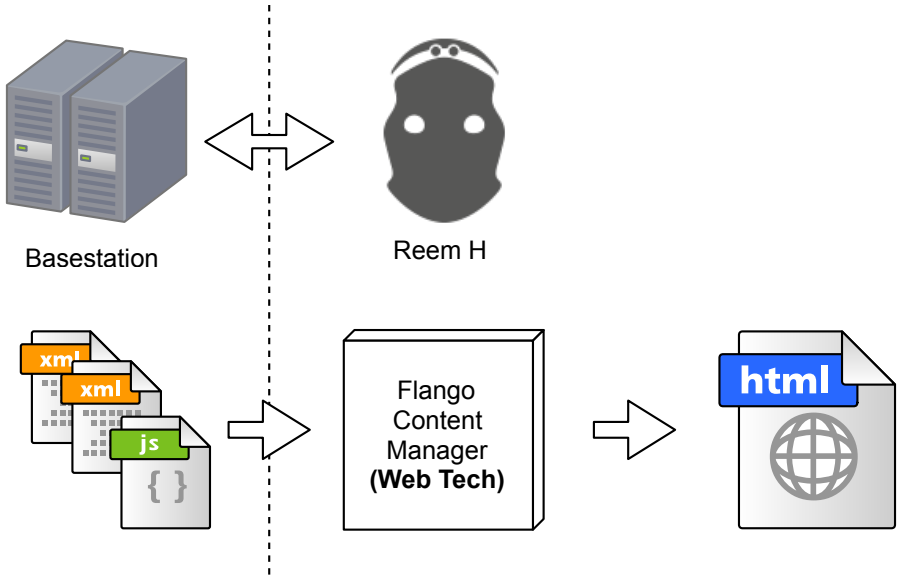

Figure 5-4: Reverse engineering: New interaction

structure. After this, it requests the screens (the XML files) to the Flango Back-End.

4. Basestation serves all the screens
5. The Content Manager renders the screens. If during this process it finds an "entity", it fetches the data from Basestation.

5.2.3 Interoperability

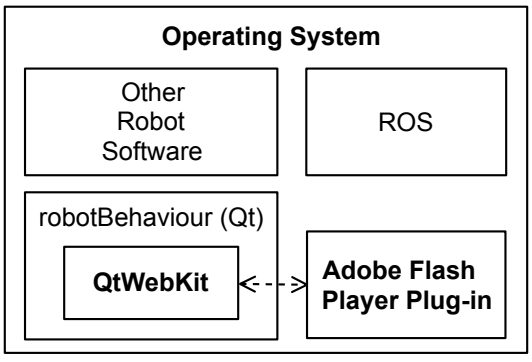

Figure 5-5: Reverse engineering: Current interoperability (in robot)

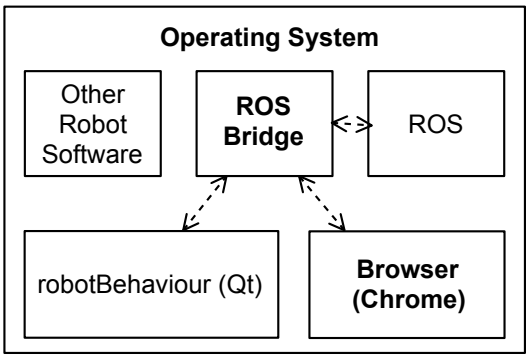

Figure 5-6: Reverse engineering: New interoperability (in robot)

Flango Content Manager interoperates with two systems: robotBehaviour and Flango Back-End (See figures 5-5 and 5-6 on 38).

robotBehaviour Flango Content Manager interoperates with the software of the robot through robotBehaviour, e.g. using text-to-speech capabilities or triggering a batch action (autopresentation, guide user to a certain place...). Likewise, robotBehaviour uses an API in Flango Content Manager, e.g. to restart a session or to display a screen. There is an interface defined for this to allow independent development in both ends.

The current system to exchange messages between the Adobe Flash container and robotBehaviour is based on JavaScript global functions.

To send a message to robotBehaviour:

1. Content Manager (Adobe Flash) opens a new tab with a specific Universal Resource Locator (URL): `flashCallback.html?<paramList>`
2. robotBehaviour intercepts the call, prevents QtWebKit from opening a new tab and executes an action specified with the parameters

To send a message from robotBehaviour:

1. Calls a JS function in the Flash website (`/static/frontend/index.html`)
2. The JS function delegates to the Flash container
3. The application facade of Flango Content Manager starts the required action

The requirements state that the new software has to be implemented using web technology. Despite the fact that this specification should be technology agnostic, to integrate the new software better in the system, the use of Robot Operating System (ROS) Web Tools is suggested to exchange messages. In any case, it is clear that there has to be an interface between robotBehaviour and the browser, as they run in separate processes (as opposed to sharing at least one parent process in the current implementation).

BaseStation (Flango Back-End API) The Content Manager has to fetch data from Basestation in order to load the correct contents. The Flango Back-End has an API for this (table 5.4 on the following page):

Both current and new implementation use this. However, the new implementation uses

Root URL	Matching URLs
`flango-api-0.1/robots`	get_robots add_robot
`flango-api-0.1/bl`	entity/(<entity_name>).xml add_robot
`flango-api-0.1/pal`	home add execute status submit history/(<page>)
`flango-api-0.1/gui` `flango-api-0.1/app`	<app_id>/config.<format>xml|json <app_id>/structure.<format>xml|json <app_id>/screen/(<screen_id>.xml <app_id>/node/<node_id> application node screen allScreens getApps upload_app
`flango-api-0.1/stats`	robot/use/<robotID¿/<duration>/¡data> robot/base app/use/<robotID>/<duration>/¡data> app/top/<robotID>/<duration>/¡data> app/toptable/<page> application/<screen> <name>

Table 5.4: Flango Backend API

JSON because it is more JS-friendly: it has less overhead and it is easier to deserialise.

5.3 Conceptual Model

The new Flango Content Manager has to be compatible with the current implementation of the Screens Editor (Flango Front-End). Otherwise, users would create content applications with the Screens Editor that would look different when they are displayed in the robot. To ensure compatibility, it is necessary a good understanding not only at system level, but at

component level. It is required, then, that the new implementation displays consistently all of the UI components that the Screens Editor can create exactly like it does with the old version. Same applies to settings and use of entities.

There are 7 classes: **ContentsApplication**, **ApplicationNode**, **Configuration**, **Theme**, **Screen**, **UI Components** and **Entity**.

ContentsApplication A contents application with an internal identifier and a descriptive, human-friendly name.

ApplicationNode An application has nodes that match the URI with a screen.

Configuration A contents application has a configuration object that can be mutated during the execution. It represents the state of the application, the permissions, paths, etc.

Theme Themes are the look and feel and default behaviour of UI components. The current implementation only has one (default) theme. However, a contents application can have many themes available. The active theme has to belong to the set of available themes. It encapsulates data that define it (e.g. the name, etc).

Screen The container for UI Components.

UI Components A UI Component is an element of the UI, for example a `Button` or an `Image`, that has an XML code associated. Users drag and drop them to a canvas in the Screens Editor to create the contents. Because the domain of the problem and the classes are shared between the original and the new implementation, the model in figure 5-7 on page 44 is created from the current implementation. To support themes, there are two types of UI Components: basic (UI Component) and theme (UI Theme Component). All UI Theme Components are eventually implemented with a basic UI Component.

Listings A.1 and A.2 in appendix A on page 147 contain a comprehensive list of UI Components extracted from listing the necessary files in a package of the old implementation. All files in A.1 have the corresponding class in the conceptual model if they have to be implemented. The elements of A.2 are in fact folders but they can be considered classes in this specification as they are part of the domain of the problem. However, some elements

are not to be implemented because they are deprecated and some other components are to be implemented in future releases, but not in the software of this thesis.

Excluded elements UI Components: GroupButtonComponent, GroupComponent, ScrollComponent. UI Theme Components: call_to_action_screen, main_menu, smartphone_menu, synchronizing_screen.

Configuration The state of the application is defined with a configuration object sent from the server. A collection of the necessary attributes of the class that encapsulates the configuration is in A.2 and has been extracted from the current implementation as well.

Excluded elements Some Configuration have names tightly coupled to technology (e.g. history, history_current, etc). The concept is included in the specification but not with this specific names to keep it technology agnostic.

Entity Entities are classes that represent classes in the Screens Editor. An example scenario: Alice is creating screens with the Screens Editor. She rented the robot for a congress and wants to add information about job openings in her company. She can define, with Flango Front-End, an entity named "Job" and use it in the screens editor to show the list of all instances of Job. The implementation of Entities is incomplete and does not belong to the scope of this project.

Themes A Contents Application can have themes. There is a default theme that defines the look and feel and default actions on click an on load of all UI Theme Components. All themes have the same UI Theme Components to guarantee that themes can be changed safely. Basic UI Components look always the same way.

Listing 5.1 shows an example of the XML definition of a Button UI Component.

```
1 <ui id="3" type="button">
2     <x lang="en">268</x>
3     <y lang="en">494</y>
```

```
4     <width lang="en">200</width>
5     <height lang="en">100</height>
6     <onload lang="en"/>
7     <onclick lang="en"/>
8     <caption lang="en">Button</caption>
9 </ui>
```

Listing 5.1: Basic UI Component Button XML

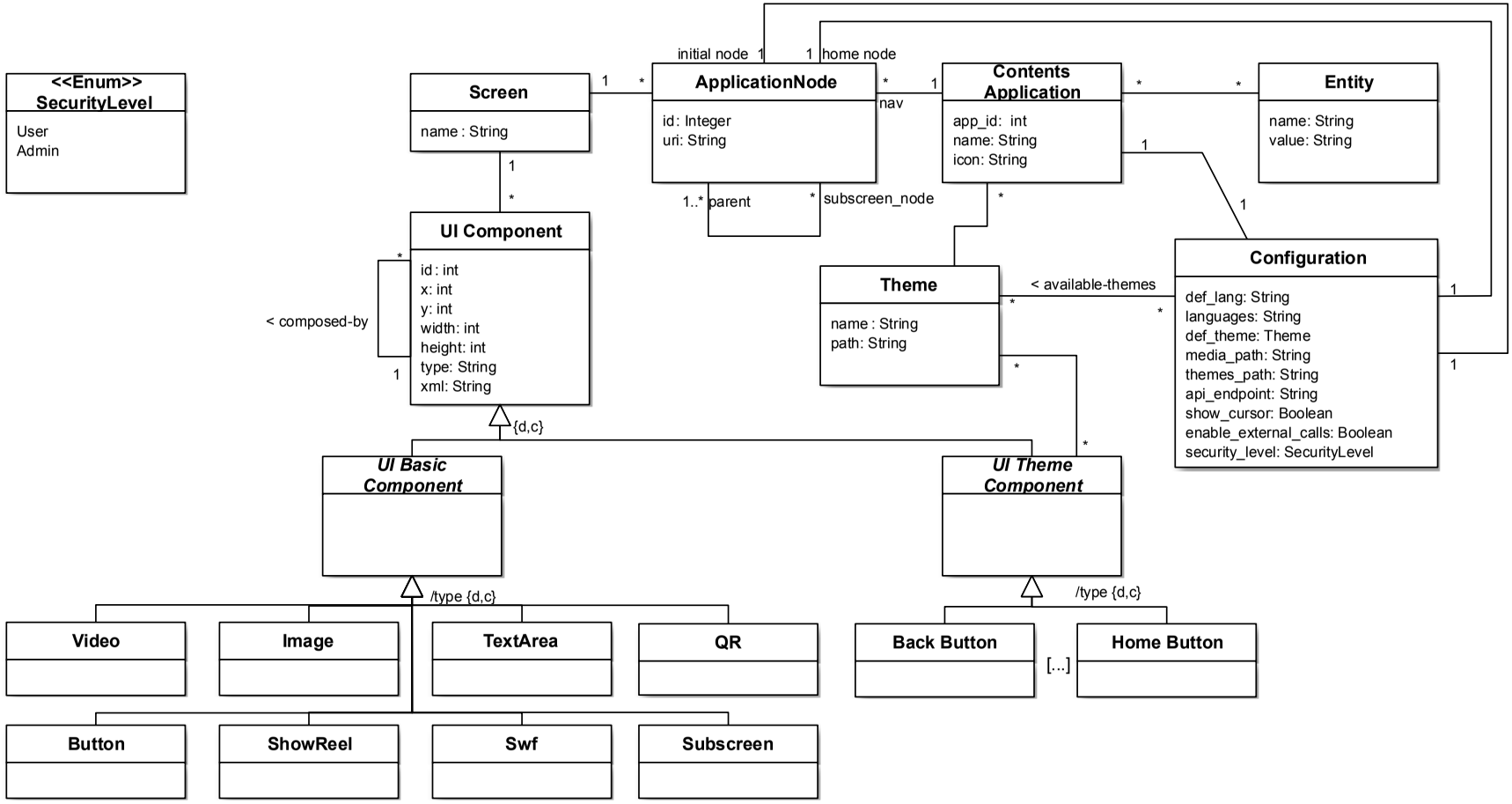

Figure 5-7: Conceptual model

5.4 Use Case Model

This section describes the services of the system as sequences of events triggered by external actors.

Actors

The actors who use the system are:

robotBehaviour The software that governs the GUI of the touchscreen. It can load content applications, bring them to the front, sent them to the back, load other GUIs etc.

Contents Application User A person (or any other agent that can use a touchscreen) who interacts with the loaded contents application. Humans only interact directly with the output of this software (see section 5.2.1 on page 36), which becomes the GUI. For example: the program displays a screen with one button whose action is to make Reem say "hi, my name is Reem". The input is an XML file, the output is a GUI ready to be used. Humans can now touch the button that sends events to robotBehaviour so, essentially, the GUI of the program and not only the output is built as a function of the input. When the GUI is ready, a new actor (Contents Application User) and new use cases appear.

System Use Cases

There is a number of use cases of the software determined by examining the current implementation at a program level. The use cases that are implemented in this project are shown in figure 5-8 on the next page.

A contents application user can see the rendered screens and update some parameters (e.g. change the language) or start applications in the robot (e.g. a Qt window with a map). The software system that controls the robot, robotBehaviour, can also trigger actions in the contents application (for example, it can start it). Amongst others, it can manage some

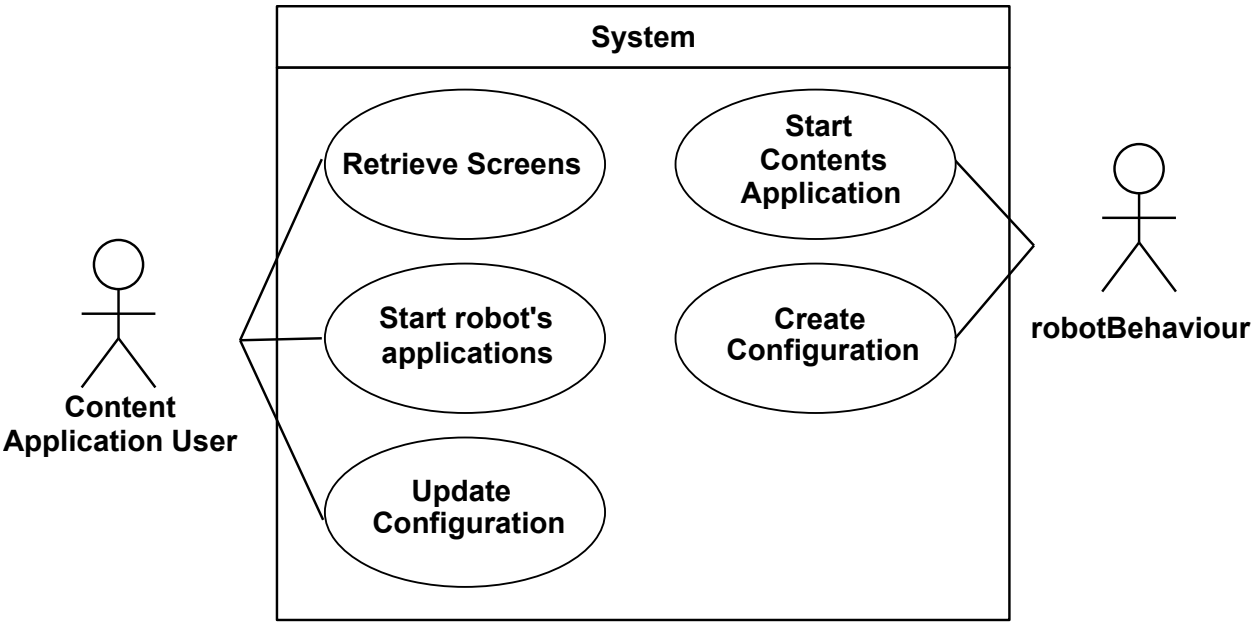

Figure 5-8: System Use Cases

special settings (including: show the cursor, enable debug mode, etc).

UC1 Start Robot's Applications

A user can start robot's applications like face recognition, ball grasping, meet people, etc.

Actors Content Application User

Preconditions robotBehaviour is running

Postconditions The chosen robot application starts

Main Success Scenario

1. The User chooses a robot's application and asks the system to start it
2. The system starts the application

Extensions None

UC2 Start contents application

robotBehaviour wants to display the application on the touchscreen of the robot

Actors robotBehaviour

Preconditions

1. robotBehaviour is running
2. The application has been synchronised with the robot (i.e. the robot has all the necessary parts to load it)

Postconditions The initial screen of the contents application is shown

Main Success Scenario

1. robotBehaviour determines the *app_id* and asks the system to load the application with the given *app_id* and initialisation parameters.
2. include Manage Configuration
3. the system loads the initial screen

Extensions

- The application can not be found
    1. The system shows a default screen
- The application contains errors
    1. The system degrades progressively the contents application

UC3 Manage Configuration – Create

The system fetches the configuration (on load) from the Backend. If it can not be found, it falls back to a default configuration.

Actors robotBehaviour[1]

Preconditions

1. robotBehaviour is running
2. The application has been synchronised with the robot

Postconditions The configuration object is created[2]

[1] as part of the bootstrap sequence of the system
[2] Delete and read configuration are internal operations

Main Success Scenario

1. robotBehaviour starts the Flango Content Manager
2. The system requests the generic configuration to the Backend
3. The Backend sends the generic configuration *c*
4. The system requests the application-specific configuration for application *c.app_id* or *auto*
5. The Backend returns the application-specific configuration *sc*
6. The system requests the application structure
7. The Backend returns the application structure *as*
8. The system reads the URL query string that overwrites parameters
9. The system creates the configuration object

Extensions

- The generic configuration can not be found
    1. The system displays an error message
- The application specific configuration can not be found
    1. The system loads the default application and displays an error message
- The application structure can not be found
    1. The system displays an error message

UC4 Manage Configuration – Update

The contents application user changes a parameter in the configuration or an event related to update the configuration is received

Actors Contents Application User, events

Preconditions

1. robotBehaviour is running
2. The application has been synchronised with the robot
3. The configuration object is initalized

Postconditions The configuration object is mutated

Main Success Scenario

1. The user asks the system to retrieve the configuration object
2. The system returns the configuration object
3. The user changes a value and asks the system to store it (e.g. current language)
4. The system updates the configuration object

Extensions

UC5 Retrieve Screen

The content application user navigates to a screen

Actors Contents Application User

Preconditions

1. robotBehaviour is running
2. The application has been synchronised with the robot
3. The application is loaded

Postconditions The requested screen is shown

Main Success Scenario

1. The user asks the system a list of screens (e.g. a screen with buttons)
2. The system offers a list of screens to go
3. The user chooses a destination
4. The system shows the requested screen

Extensions

- The screen is unavailable
    1. The system shows an error

5.5 Behaviour Model

This thesis combines DbC and TDD, two techniques that are not mutually exclusive. Basically, **tests define the behaviour for specific cases and DbC defines the general behaviour**. Thus, there are no tests for edge cases or negative behaviours because contracts, and more specifically preconditions, protect the component. In spite of the name, TDD tests are executable specifications. They become tests only after having implemented all the functionality being test-driven. Before this, there is nothing to actually test and these artifacts are rather executable specifications, but not tests. **Tests (specifications) are written before the code and define the behaviour of component**s.

Additionally, they are used as documentation of the software but they are not sufficient to completely define the behaviour because they only assert properties by example instead of stating general properties (e.g. "it should return 5" vs "return value $1 \leq x \leq 10$"). The latter can be described with formal specifications, e.g. using Meyer's DbC [10].

In this thesis, tests are implemented with Jasmine and AngularJS but this chapter focuses on what the tests describe rather than how they are executed and controlled.

Combining the two techniques one can use the best parts of each to define the behaviour of the system. With TDD:

- Build only what is required (Keep it simple and control the scope of the project)
- Drive design decisions from tests (i.e. from the specification)

With formal specifications (DbC):

- Higher code quality
- Less checks (preconditions)
- Drive design decisions from preconditions

Contracts of operations define system behaviour and describe the outcome of executing system operations in terms of state changes to domain objects. This section contains the sequence diagrams and contracts of the most relevant operations in the system. However, the most complex part of this project is parsing and rendering the input XML files. Users do

not interact directly with them: they do not execute operations that result in state changes to the domain object. Chapter 6 on page 57 explains this problem and the solution in detail.

5.5.1 Sequence Diagrams

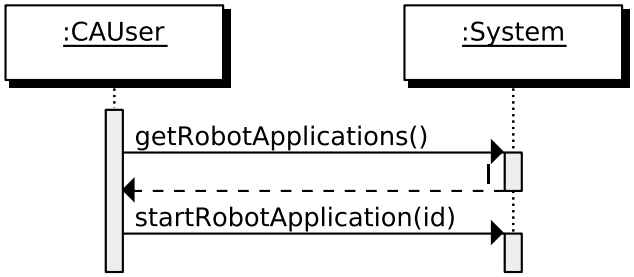

Figure 5-9: System Sequence Diagram: Start Robot Applications

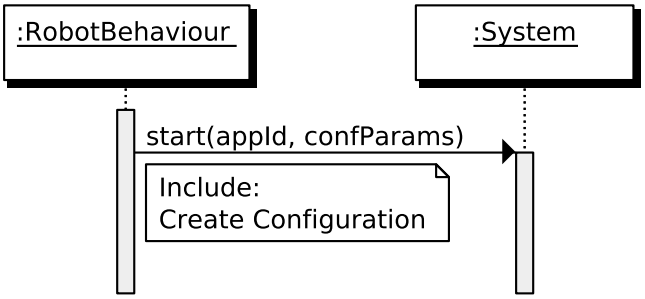

Figure 5-10: System Sequence Diagram: Start Content Applications

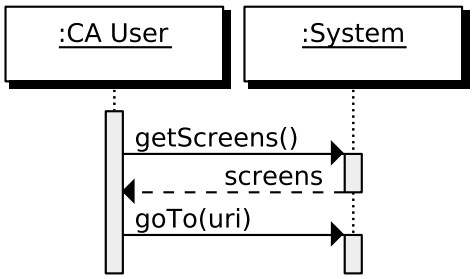

Figure 5-11: System Sequence Diagram: Manage Screens (Read)

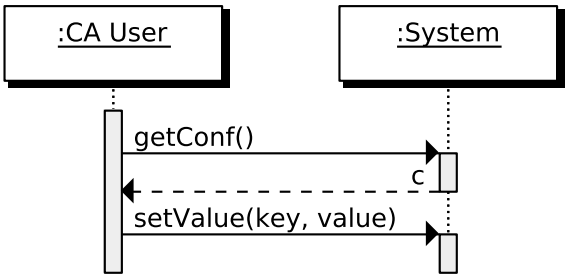

Figure 5-12: System Sequence Diagram: Manage Configuration (Update)

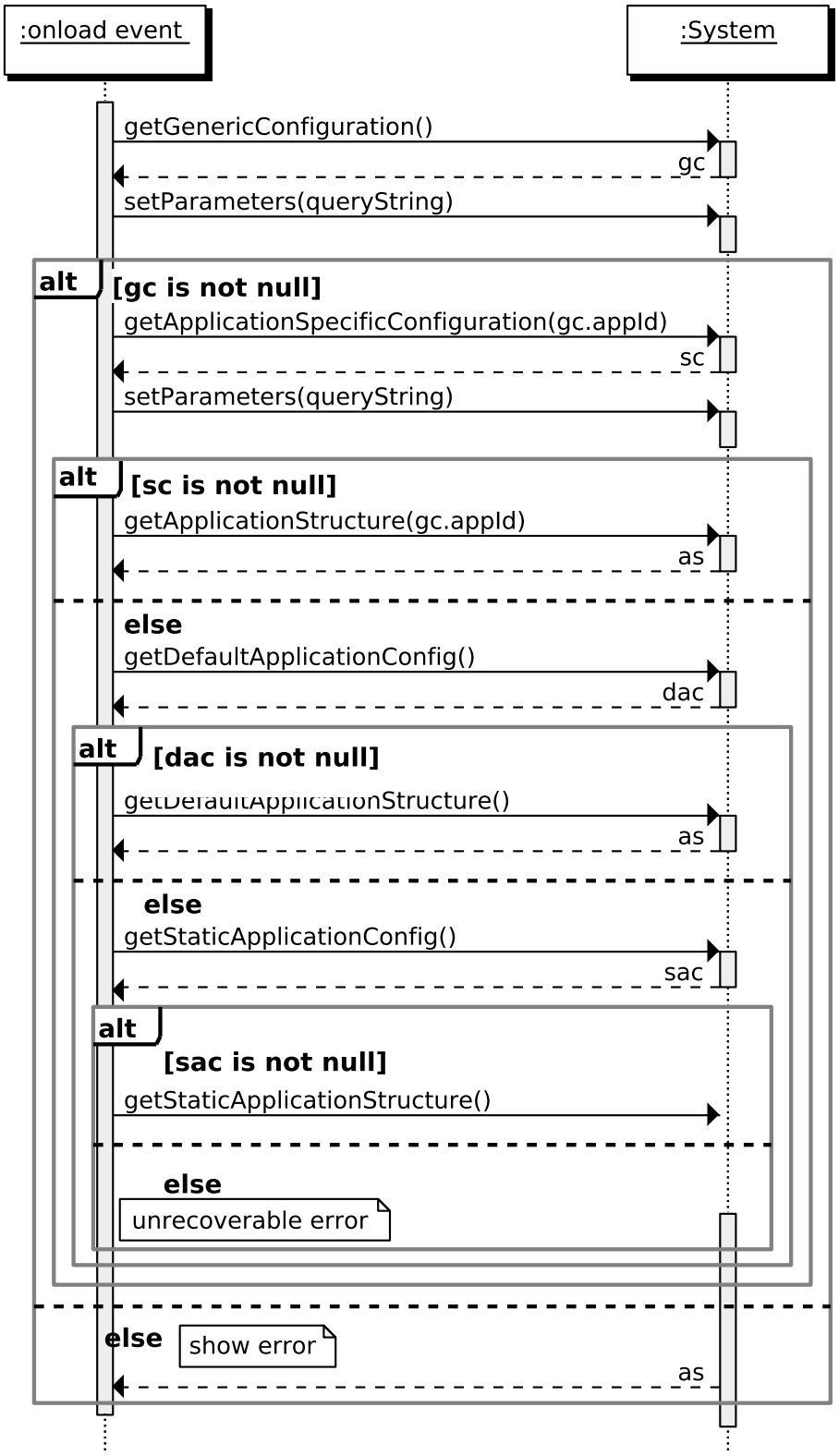

Figure 5-13: System Sequence Diagram: Manage Configuration (Create)

5.5.2 Contracts

C1 getRobotApplications()

Retrieves a list of applications that can be launched from the Flango Content Manager, e.g. Qt applications like navigation or face recognition[3].

Preconditions RobotBehaviour is running

Postconditions A list `l` of robot applications was returned

C2 startRobotApplication(id:String)

Starts the robot application with id `id`, e.g. 'face recognition'

Preconditions `id` is a valid robot application identifier

Postconditions A request to start the robot application with id `id` was sent

C3 start(appId: Integer, confParams: HashMap)

Starts the content application with id `appId` and configuration parameters `confParam` that can overwrite the fetched configuration from the backend

Preconditions `id` is a well-formatted contents application identifier

Postconditions Loads the content application with id = confParams.id or, if not defined, with id = `appId` and displays its initial screen

C4 getScreens(appId: Integer)

Obtains the screens of the contents application with id = `appId`

Preconditions `id` is a well-formatted contents application identifier

Postconditions A list of all screens of content applications with id = `appId` was returned

[3]In the design phase operations only launch valid commands to start applications using an API instead of fetching a list and choosing one.

C5 goTo(appId: Integer, uri: String)

Displays the desired screen

Preconditions

1. `id` is a well-formatted contents application identifier
2. `uri` is a well-formatted screen URI in the contents application

Postconditions `Conf.currentScreen` was set to `uri`

C6 getConf(appId: Integer)

Obtains the local configuration object for the application `appId`. This object is stable during the execution of the program. It does not need to be fetched again from the Backend.

Preconditions `id` is a well-formatted contents application identifier

Postconditions The configuration `c` was returned

C7 setValue(c: Conf, key, value)

Sets a field of the configuration object to the desired value.
e.g. `setValue(c, 'currentLang', 'de')`

Preconditions

1. `key` is a valid field of the configuration object `c`
2. `value` is a valid value for `c[key]`

Postconditions `c[key]` was set to `value`

5.5.3 Unit Tests

The conceptual model in figure 5-7 on page 44 describes the real world and shows the relationship between elements that define the problem. Actors do not interact with most of the classes shown and contracts and sequence diagrams are intended to describe the system as a unit that provides some service to actors. One of the goals of this project is making it

compatible with the current XML that defines the content applications. To meet this goal, it should be clearly specified how XML are parsed. This project uses TDD and has executable specifications for all parts of the system, including those (e.g. internal operations or parsing) that are not strictly eligible for other sections in this chapter. Formally, an executable, technology-agnostic, specification might be written in Object Constraint Language (OCL) and run in a Unified Modelling Language (UML) model. These tests focus on the implementation and are written in JavaScript. Even though they depend on technology, they define the system's behaviour and are considered part of the specification in this document.

Tests are organised in test-suites for components and each suite has several test cases. This section focuses on the XML description.

Chapter 8 on page 133 has a full list and examples of other test suites that refer directly to components of the software (described in chapter 6 on the next page) Tests challenge behaviour like reading text nodes in XML tags, setting properties (i.e. mutating correctly an object in the system), reading order or inheritance of properties, fallback to default values and transformation to HTML code.

An example of specifications by example of the UI directive:

1. inline type property: should have type defined
2. width, height, x, y,
   (a) should set the property to 42 for the default (language-independent)
   (b) should set the property to 40 for Catalan and 42 for French
   (c) should set the property to 40 for Catalan, 42 for French and 20 for default
   (d) should not set any property
   (e) (inline) should set the property to 42 for the default (language-independent)

A comprehensive list is available in chapter B on page 151.

Chapter 6

Design

This chapter describes the internal wiring of the application and the reasons that lead to this design. First, it explains the technology used: the programming language, frameworks and most relevant libraries. Next, it presents the architecture of the system and the application, the software patterns and sequence diagrams of the most relevant operations. Finally, it includes a physical view with deployment diagrams.

Reverse engineering the current project involves 3 activities related to technology and implementation details: **code restructuring**, **data restructuring** and **forward engineering**. No modifications in the code of the old version of Flango Content Manager are scheduled: there is no code or data restructuring. However, there are some adaptations to be done in the Flango Back-End (API), XML and robotBehaviour:

Changes in Flango Back-End The API is designed to return XML in all cases. Most of the time this is desirable (e.g. requests to get screens) but in cases where normal exchange of information is done between the two systems, JSON is a better option for the new technology. It is less verbose (minimise bandwidth) and has a straight-forward translation to JavaScript objects.

Changes in XML The XML syntax is designed to work isolated. The new version, however, can mix XML with HTML thanks to Angular. Some tags of the XML vocabulary conflict with HTML. For instance, `caption`. Solution: add a namespace `fl` to all tags. Therefore, in the new version a `<ui type="base-button"></ui>` becomes
`<fl:ui fl:type="base-button"></fl:ui>` .

Changes in robotBehaviour Current interoperation between Flango Content Manager and the robot is limited to robotBehaviour. This communication is done with JavaScript callbacks and URLs. To decouple the Content Manager as much as possible from the rest of the robot, the new system uses ROS Topics and ROS Bridge

Forward Engineering A program that depends on a dead browser plug-in with performance issues is not sustainable. A solution is completely redesigning, recoding and retesting the program, even with a different technology that makes it more flexible, open to new developers and without performance problems.

6.1 Technology

The project of this thesis, Flango Content Manager, is designed to work client side in a web browser. That is, in the robot, as a client, despite the fact that the server is in the robot as well. The natural technology in this environment is JavaScript, HTML and CSS. It uses the Google Angular framework with jQuery. Angular *teaches the browser new syntax*. That is, instead of parsing the DOM tree and building the GUI, it defines the behaviour of new tags in HTML documents that can be treated as regular HTML nodes in the DOM. It interoperates with other systems built with a different technology: components in the robot use ROS and the Flango Back-End has an API (see sections 5.2.1, 5.2.2 and 5.2.3 on page 36).

6.1.1 JavaScript

JavaScript is probably the most misunderstood and at the same time popular language of all times. It is the language of the web browser and works closely with the DOM, the API of the browser.

Some good and key ideas in the language include **functions**, **loose typing**, **dynamic objects** and an expressive **object literal notations**. On the other side, the programming model is based on global variables (although the language has tools to mitigate this) [11]. **Functions are first class objects** and can be passed as arguments or returned from functions, just like any other object. This allows the language to have function scope and lexical scoping:
Function scope means that variables defined in a function are visible in the entire function, an approach different from block scope in languages like C or Java.
Lexical scoping defines how names are resolved in nested functions: the scope of an inner function contains the scope of the parent function, even if the parent has returned. This scoping is key in the strategy to read and inherit properties with the XML that defines UI Components.

JavaScript is an **object-oriented prototype-based language**, not a class-based language. It has a class-free object system in which objects inherit properties directly from other objects. Behaviour reuse (inheritance) is achieved by cloning existing objects (the prototypes). Crockford explains the prototypal inheritance mechanism [12]:

> So instead of creating classes, you make prototype objects, and then use the object function to make new instances. Objects are mutable in JavaScript, so we can augment the new instances, giving them new fields and methods. These can then act as prototypes for even newer objects. We do not need classes to make lots of similar objects.

6.1.2 Angular

Overview Angular is a MVC framework for dynamic web applications that enforces a strict separation of concerns. HTML was created to declare static documents and needs libraries and frameworks to enable application development. Angular lets developers use HTML in the templates and extend the language syntax to express reusable components. It *teaches the browser new syntax* through a construct called Directive. Features:

- Tools to build Create, Read, Update, Delete (CRUD) Applications: data-binding, basic templating directives, form validation, routing, deep-linking, reusable components, dependency injection.
- Tools to test: unit-testing, end-to-end testing, mocks.

Flango Content Manager reads XML and renders it as HTML. There are two possible approaches: parsing the DOM tree in the browser or teaching the browser new syntax. This project uses the latter: instead of developing an algorithm to traverse the tree and render elements, it defines reusable components (Angular Directives) and includes the XML files in the appropriate places in the DOM. The browser already traverses the tree in order to display the page and there is no need to do it manually. Normally a browser ignores an element that does not belong to the HTML specification (e.g. `<fl:ui fl:type="base-button"></fl:ui>`) With Angular, the browser runs the behaviour defined in the reusable component.

Building Blocks There are several key components in Angular to build the application.

Controller A function that binds the view with the model. It typically assigns objects to the $scope variable to expose them to the template.

Directive A construct to extend HTML with custom attributes and elements. It defines the template of the reusable component, the behaviour, the attributes, etc. For instance, the directive `flBaseButton` states that the node `<fl:ui fl:type="base-button">` in the DOM tree should be transformed into a simple `button` using the theme template `baseButton.xml`.

Service Singleton that encapsulates reusable business logic independent of views. For instance, a service to handle the configuration of the application. They can be injected in controllers, directives or other services.

Filter A function that formats the value of an expression for display to the user. For instance, a filter to display a Number with currency format.

Module These building blocks are grouped in Modules. They can have dependencies between them.

The framework provides all the necessary tools to use unit and E2E testing from the very beginning of the project. Chapter 8 on page 133 explains them in depth.

Compile-Link To make directives possible Angular has a compiler. In this context, compiling means attaching event listeners to the HTML to make it interactive. The compiler traverses the DOM tree in two phases: Firstly it looks for tags that have directives associated (e.g. `<fl:ui fl:type="base-button">` or even native HTML elements `<a>`). Secondly it binds the model to the view. The compiler allows developers to attach behaviour to any HTML element or attribute and even create new HTML elements or attributes with custom behaviour. Angular calls these behaviour extensions directives.

Listings 6.1 and 6.2 illustrate the order of execution.

```
1 <div directive1>
2   <div directive2>
3     <!-- ... -->
4   </div>
5 </div>
```

Listing 6.1: Order of execution of directives

```
1 directive1: compile
2   directive2: compile
3 directive1: controller
4 directive1: pre-link
5   directive2: controller
6   directive2: pre-link
7   directive2: post-link
8 directive1: post-link
```

Listing 6.2: Result of execution of directives

The role of these functions is:

- **Directive Controller**: similar to regular controllers but accessible from other directives.
- **Compile** finds new directives and runs their compile function. Modifications to the DOM before applying the template can be done here.
- **Link**: binds data to the view. Modifications to the final DOM are safe to do here.

Directive At a high level, directives are markers on a DOM element (such as an attribute, element name, CSS class or a comment) that tell Angular's HTML compiler to attach a specified behaviour to it or even replace it with another element. The HTML compiler finds elements that match directives. For example, for `<div ng-controller="myCtrl">`, the element `div` matches the directive `ngController`. An example in Flango Content Manager: when the browser finds `<fl:width>150</fl:width>` in the DOM, the element `width` matches the directive `flWidth`. The `flWidth` directive is restricted to elements (it matches `<fl:width>150</fl:width>` but not `<div fl:width>150</div>` because it is an attribute), and it has a custom `link` function that performs some computation.

6.1.3 HTML5

HTML has been extended towards enabling application development in the browser. This project extensively uses regular HTML and the `<video>` tag to implement the UI Video component. Therefore, new videos are encoded to be compatible with the browser instead of `flv` and `f4v`. For example, `H.264` or `ogv`.

6.1.4 SASS

Writing plain CSS can be tedious and error-prone. SASS stands for Syntactically Awesome CSS and is a pre-processor with syntax advancements that add features to CSS. SASS is an extension of CSS3, adding nested rules, variables, mixins, selector inheritance, and more.

SASS code is translated into well-formated, standard CSS using the command line tool. This makes reusing and extending CSS easier. SASS has two syntaxes. This project uses the newest, known as Sassy CSS (SCSS) (Sassy CSS).

A key feature in this project is the `@extend` directive: it tells Syntactically Awesome Syle Sheets (SASS) that one selector should inherit styles of another selector. Example: the style `base-button` has a grey background. The style `button` sets the size of the element to $100x50px$. Because all `base-button`s are `button`s, the first should inherit the style from the second instead of replicating it, like one would do with Object Oriented Design (OOD). With HTML this would be written `<div class="base-button button" />`. With SASS one can write `.base-button { @extend .button }` and keep HTML cleaner:
`<div class="base-button" />` instead of `<div class="base-button button" />`

SASS variables and mixins make it easier to create themes for Flango Content Manager and provide a way to ensure visual consistency (e.g. all themes can have palettes of the same size).

6.1.5 Robot Web Tools

Robot Web Tools is a collection of open-source modules and tools for building web-based robot applications. It provides 3 core libraries to communicate with ROS on the robot over rosbridge's WebSocket server: roslibjs, ros2js, and ros3djs. This project uses roslibjs[1] to subscribe and publish to ROS Topics.

6.2 Architecture

This section contains details about the architecture of the system (ROS) and the project of this thesis (Flango Content Manager). Flango Content Manager is a component in a system that works with ROS, which is similar to a Service Oriented Architecture (SOA) system,

[1] `https://github.com/RobotWebTools/roslibjs`

and interoperates with it using its message system. The following pages have a high-level description of the system's architecture and the patterns used.

6.2.1 Robot Operating System

The operating system of Reem-H is Ubuntu Linux. Robot Operating System (ROS) is a framework for robot software development that provides operating system-like functionality on a heterogeneous computer cluster. It was originally developed in 2007 as part of the Stanford AI Robot (STAIR) project to create an architectural framework for modular, tool-based development of robotic software. It has a loosely coupled architecture with nodes, messages, topics and services, similar to SOA [13].

Nodes are processes that perform computation, a "software module". Nodes communicate with each other by publishing/subscribing with two protocols:

Topics asynchronous. message passing (publish/subscribe). Typed by a ROS Message
Services synchronous. RPC (pairs of request/response messages). Typed by a ROS Service

Messages are strictly typed data structures. There are some available by default, like `std_msgs/String` and other primitive types. To find a service in this peer-to-peer topology, there is a lookup mechanism: *rosmaster* (see figure 6-1 on the facing page and figure 6-2 on the next page).

The robot has two parts: robotBehaviour and Stacks. Part of the code of the robot (robotBehaviour) does not follow the ROS workflow but still uses Topics to communicate with the Stacks, the part of the project that does use ROS. In Stacks there are Servers and ActionServers that provide Services.

Flango Content Manager is a component in this system that communicates with robotBehaviour using ROS Topics and two types of messages. It is not designed strictly as a service to handle synchronous requests but it is able to use Topics to have asynchronous requests and provide feedback, like ActionServers. A typical use of a ROS Topic is setting the language of the Content Manager or sending an action to robotBehaviour (e.g. shake hands, say a

sentence). robotBehaviour is not a ROS service and therefore it can not handle request/responses like other components in the system. Topics are the only way to communicate with it.

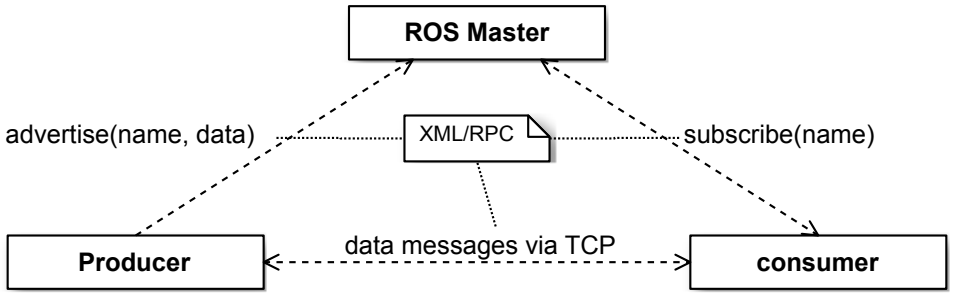

Figure 6-1: High-level view of the ROS Architecture (Topics)

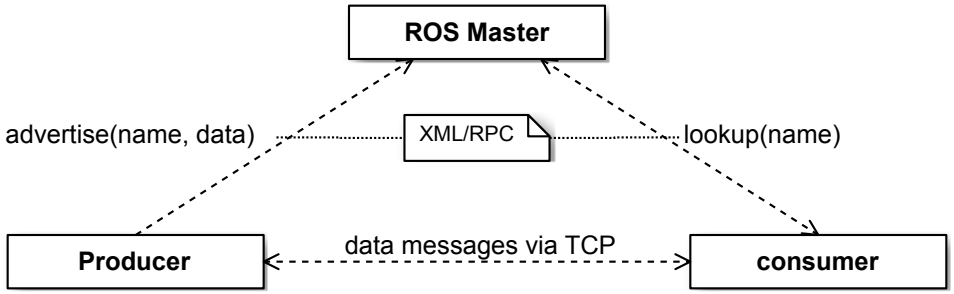

Figure 6-2: High-level view of the ROS Architecture (Services)

6.2.2 Model View Controller (MVC)

Model View Controller (MVC) is a decoupled architecture with a strong separation of responsibilities (figure 6-3 on page 67).

Model (application state). Maintains application state and notifies dependent views and controllers when changes occur (with the observer pattern)

View (output) queries the model to print [parts of] it, listens for changes in the model

Controller (input) Listens for input and tells the model or the view to change accordingly

It is possible to have multiple views and controllers for the same model and they can be reused for other models.

Angular applications take the most of the framework when they are designed with a MVC architecture. A typical use case in a MVC application might be:

- Model: `[orange, apple, pineapple, coconut]`
- View: displays a list of 2 random pieces of fruit
- Controller: logic to fetch two pieces of fruit

Flango Content Manager does not have a GUI or a set of predefined and stable use cases: this changes for every content application. The system only knows the use cases after loading the content application. There can not be controllers for application-specific use cases and the view is created dynamically from the model.

- **Model**: screens definitions, entities, configuration... stored in the backend
- **View**: built dynamically from the model. The application loads screens as the user navigates to URLs and transforms them to HTML so that they can be displayed in the browser.
- **Controller**: Typically, controllers in Angular manage the `$scope` and expose behaviour to the View. This application has some generic (as opposed to application-specific) ones. For example: an instance of `ActionCtrl` for each button, a `ROSBridgeCtrl` to respond to requests from ROS Bridge, etc. Directives also have a special type of controllers: Directive Controllers are defined within the context of one directive but they can be injected into other directives to facilitate inter-directive communication. For example, a controller in the `flUi` directive manages its properties (`x`, `y`, `width`, `height`, `caption`...). `flWidth`, `flX`, and other directives use the controller in `flUi`.

Angular uses an MVC pattern with two-way data binding (figure 6-4 on the next page): changes in the view are reflected to the model immediately and viceversa in order to use the model as a single source of truth. Controllers use the `$scope` to expose values and behaviour to the view: they update the object that is used in the view. The `scope` is an object that refers to the application model. Informally, it is the glue between the controller and the view. Controllers expose values and behaviour via the $scope service but do not manipulate the DOM or read it in any way.

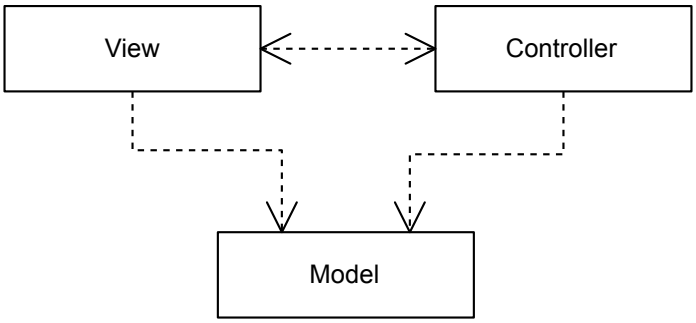

Figure 6-3: MVC overview

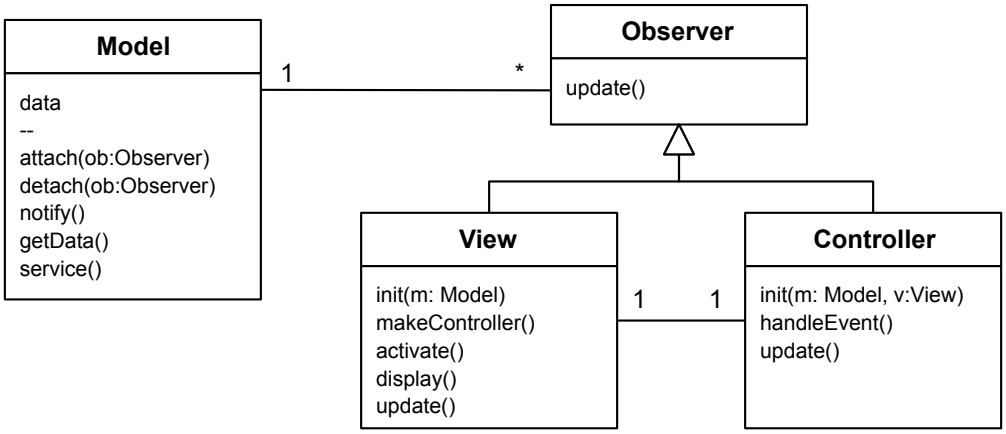

Figure 6-4: MVC with observer pattern

6.2.3 Software Patterns

This subsection is a high-level description of the most relevant software patterns in the project: dependency injection and service locator, factory method, module and revealing module, command, and remote facade.

Dependency Injection and Service Locator After applying the Creator Pattern (GRASP, [14]) class `A` creates objects of dependant class `B`. One of the SOLID principles, *dependency inversion*, states that one should "Depend upon Abstractions. Do not depend upon concretions." **Dependency Injection** is one of the implementations of this principle [15].

In general there are 3 ways an object can use another one in JavaScript:

1. With a `new` operator (`A` creates `B`)
2. Using global variables (`A` uses a global `B`)

3. Pass in the dependency to where it is needed

Dependency Inversion means that objects do not create other objects on which they rely to do their work. They get the dependencies from an outside source (e.g. an XML configuration file). **Dependency Injection** means that this is done in a transparent way for the object. The dependency is supplied to the software component.

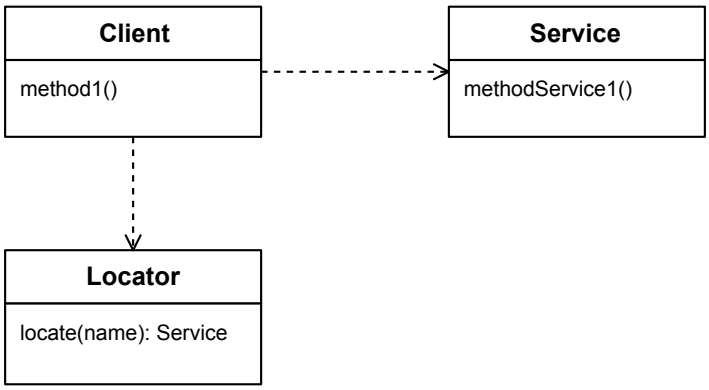

Figure 6-5: Pattern: Service Locator

The framework encourages the use of this pattern in all components. More specifically, the framework uses a **Service Locator** (figure 6-5) called `$injector` that manages the dependency creation. This allows services to be used in controllers and directives by their name. The client class does not need to know the concrete implementation of the service. With this principle it is easy to mock classes for unit testing and change components in general.

Factory Directives, controllers and services are registered in modules. The `module.directive` API registers them. It takes a normalised name and a factory function (figure 6-6 on the next page) that returns an object with the fields that can be exposed to classes that use it. For example, a Directive Definition Object or a Service.

Revealing Module Modules typically help in having separated the units of code for a project. JavaScript does not support the concept of classes but it does support special constructor functions that work with objects. The `new` keyword used in a call to a constructor function makes JavaScript instantiate the new object with members defined by that function.

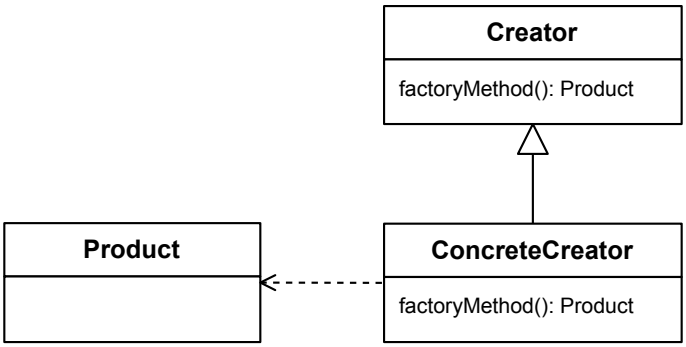

Figure 6-6: Pattern: Factory Method

Modules are a way to emulate the concept of classes. There are several options to implement them. The **module pattern** encapsulates privacy, state and organisation using closures. Only a public API is returned, keeping everything else hidden within the closure private. With the **Revealing Module Pattern** [16] the code of the module is simplified: all variables and functions are defined in the private scope and returns an anonymous object with pointers to the private properties that should be revealed. An implementation is shown in listing 7.12 on page 117.

Remote Facade The software of this thesis is a component in ROS. A way of integrating it with the system would be desinging the business layer as a service[2]. A Remote Facade subscribes and listens to ROS topics and exposes the logic to other components. Normally, remote facades have coarse-grained methods and, in this design, operations attend requests made via ROS topics to behave like a simplified *service layer*.

Command Sometimes requests are issued to objects without knowing anything about the operation being requested or the receiver. The command pattern is a behavioural design pattern in which an object is used to represent all the information needed to call a method at a later time. It lets objects make requests of unspecified application objects by turning the request itself into an object. This object can be stored and passed around like other objects [14]. Other names are **Transaction** or **Action Pattern**

[2]This approach was only considered in the last weeks of development due to changes in robotBehaviour and only a simplified design is completed.

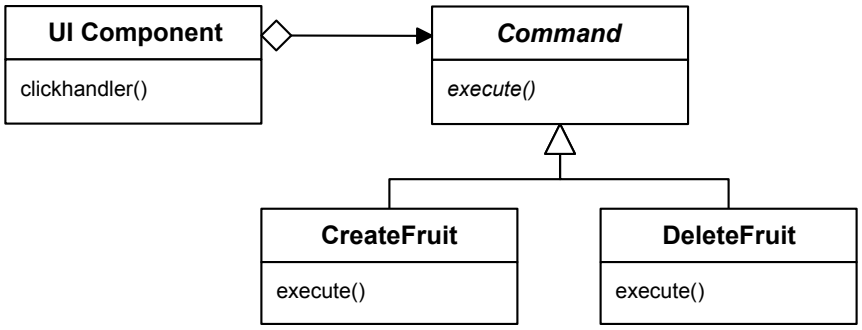

Figure 6-7: Pattern: Command (general)

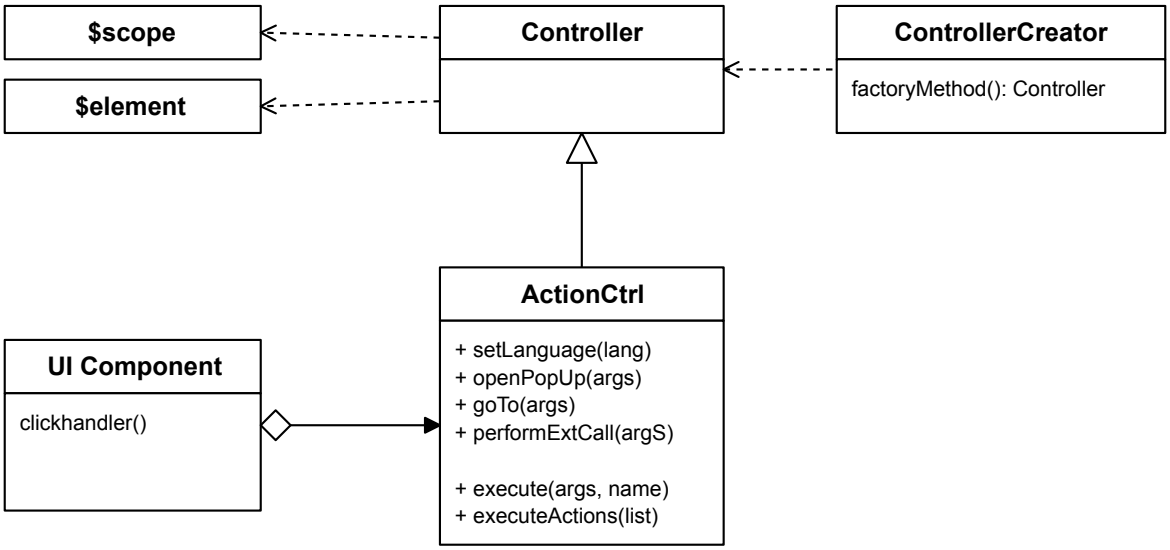

Figure 6-8: Pattern: Command (in Flango Content Manager)

Figure 6-7 shows the class diagram of this pattern in the context a traditional object-oriented programming language. With JavaScript there are no abstract classes. Osmani [16] proposes having only concrete classes with a common `execute()` method. Section 7.4 on page 118 contains a description of the implementation in JavaScript of this pattern.

With a simple controller (figure 6-7) one would call the methods in the object. While this is correct, at the time the GUI is built the `UI Component` does not know about the operations being requested. For example, there is no way to define the behaviour for a `UI Component::Button` click handler. Only the content application knows it. By adding an `execute` method to `ActionCtrl`, used in `UI Components` that accept the tag `onclick` and `action`, methods can be called without knowing them (figure 6-8).

6.2.4 Orientation to Web Components

Web Components is a set of specifications that let web developers use HTML, CSS and JavaScript to build widgets that can be reused easily and reliably. A web component consists of five pieces [17]:

Templates define chunks of mark-up that are inert but can be activated for use later.

Decorators apply templates based on CSS selectors to affect rich visual and behavioural changes to documents.

Custom Elements let authors define their own elements, with new tag names and new script interfaces.

Shadow DOM encapsulates a DOM sub-tree for more reliable composition of user interface elements.

Imports defines how templates, decorators and custom elements are packaged and loaded as a resource.

At the time of developing this project, the specification of web components is still a work in progress. The technology, however, seems to satisfy the needs of the project. There are 2 well-known initiatives that implement an approach to web components: **Google Polymer** and **Google Angular**.

Polymer Polymer is a framework that aims to use Web Components. It is based on Custom Elements, i.e. everything is a component. With Polymer developers can compose and encapsulate bits of HTML that can be used in any other templating system or framework. It uses HTML and the DOM APIs to separate the view (DOM) from the model. Updates to the model are reflected in the DOM and user input in the view is immediately propagated to the model:: fast two-way data binding. However, this framework is in pre-alpha stage and can not be used in a stable project.

Angular The framework of choice for this project has features close to web components: it has declarative templates (in directives) that can be applied for elements, attributes,

comments or classes (like Web Components decorators). Directives are custom elements: behaviour can be placed in a directive controller, in a compile or in a link function. Templates are usually HTML, although it is not shadow DOM. Units of code can be grouped in modules, services, etc and be reused.

6.3 Static View

This section contains a comprehensive description of the objects in the system and their relationships taking into account the project's technology: JavaScript and Angular client-side, Django in the backend, ROS Topics to interoperate with the rest of the system. There are two subsections: a class diagram and a packages diagram.

6.3.1 Class Diagram

JavaScript is an object-oriented language but it is not class-based. It is prototype-based instead. It can be used as a class-based language using constructors but this project does not need to do this: it does not create objects of any domain-specific class. Instead, Angular constructs are used to instantiate services, controllers and directives, the building blocks of the application. Most of the boxes in these diagrams are not strictly classes but objects that the framework instantiates using the provided definitions.

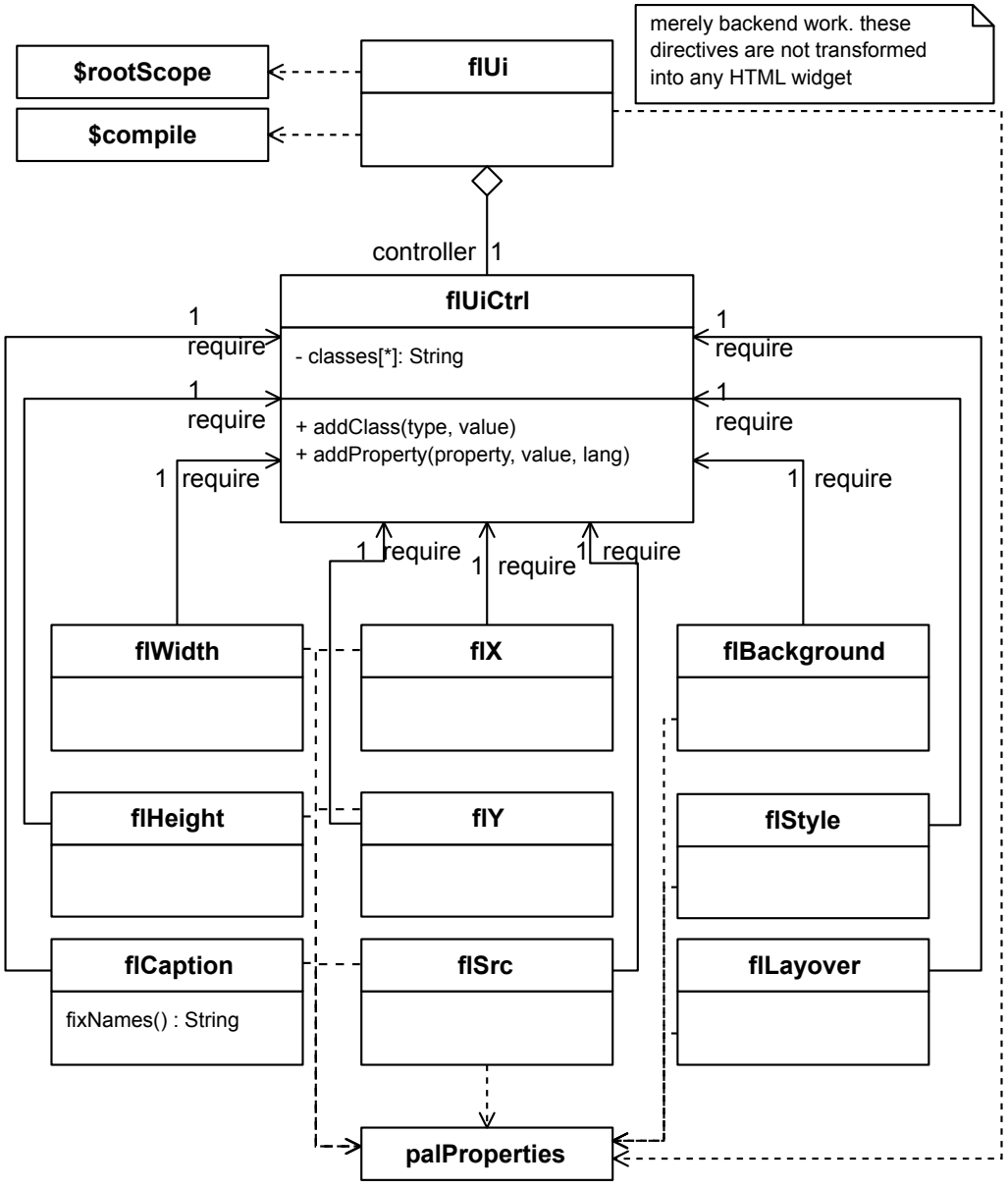

Figure 6-9: Class Diagram: UI

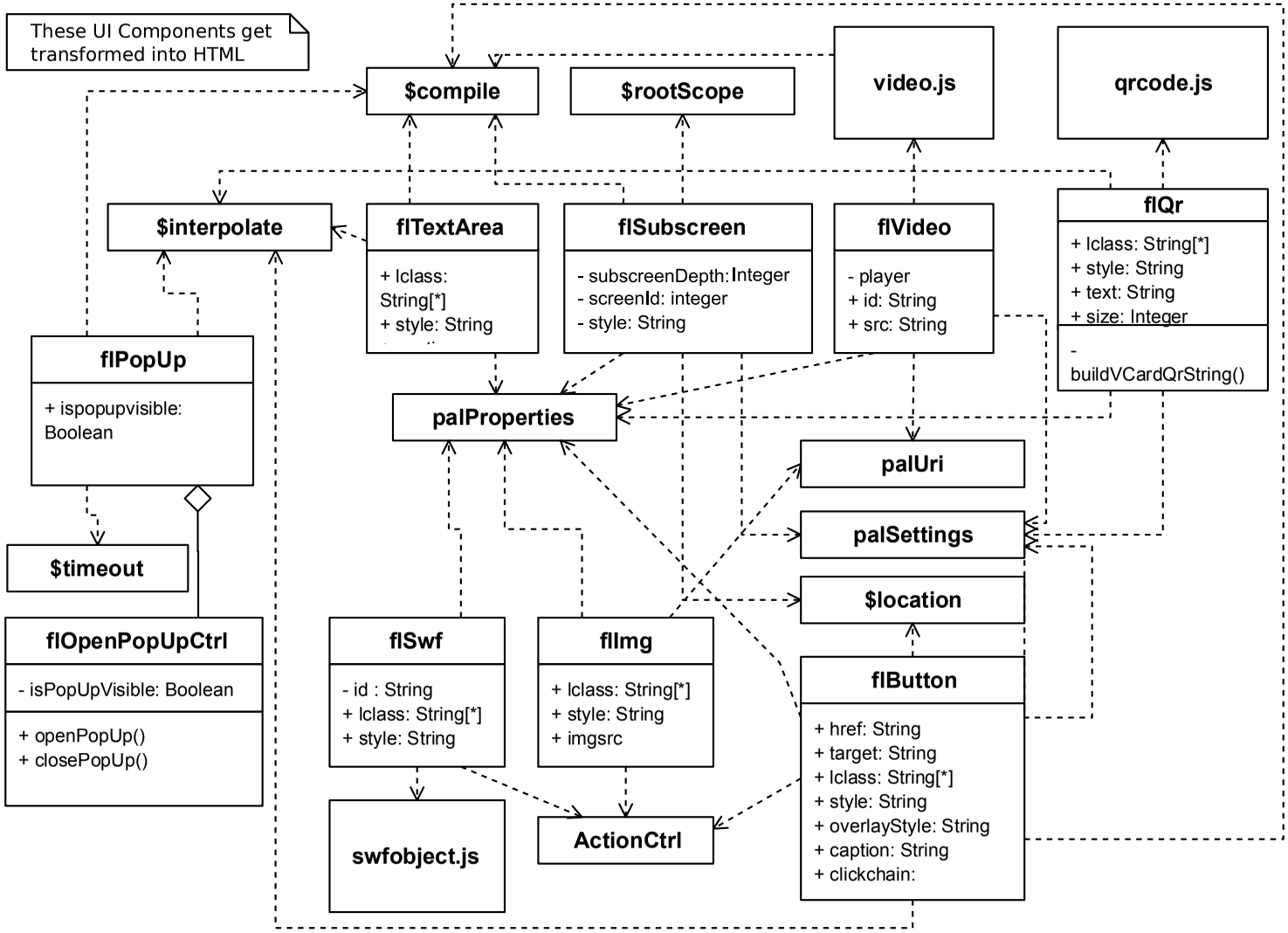

Figure 6-10: Class Diagram: UI Component

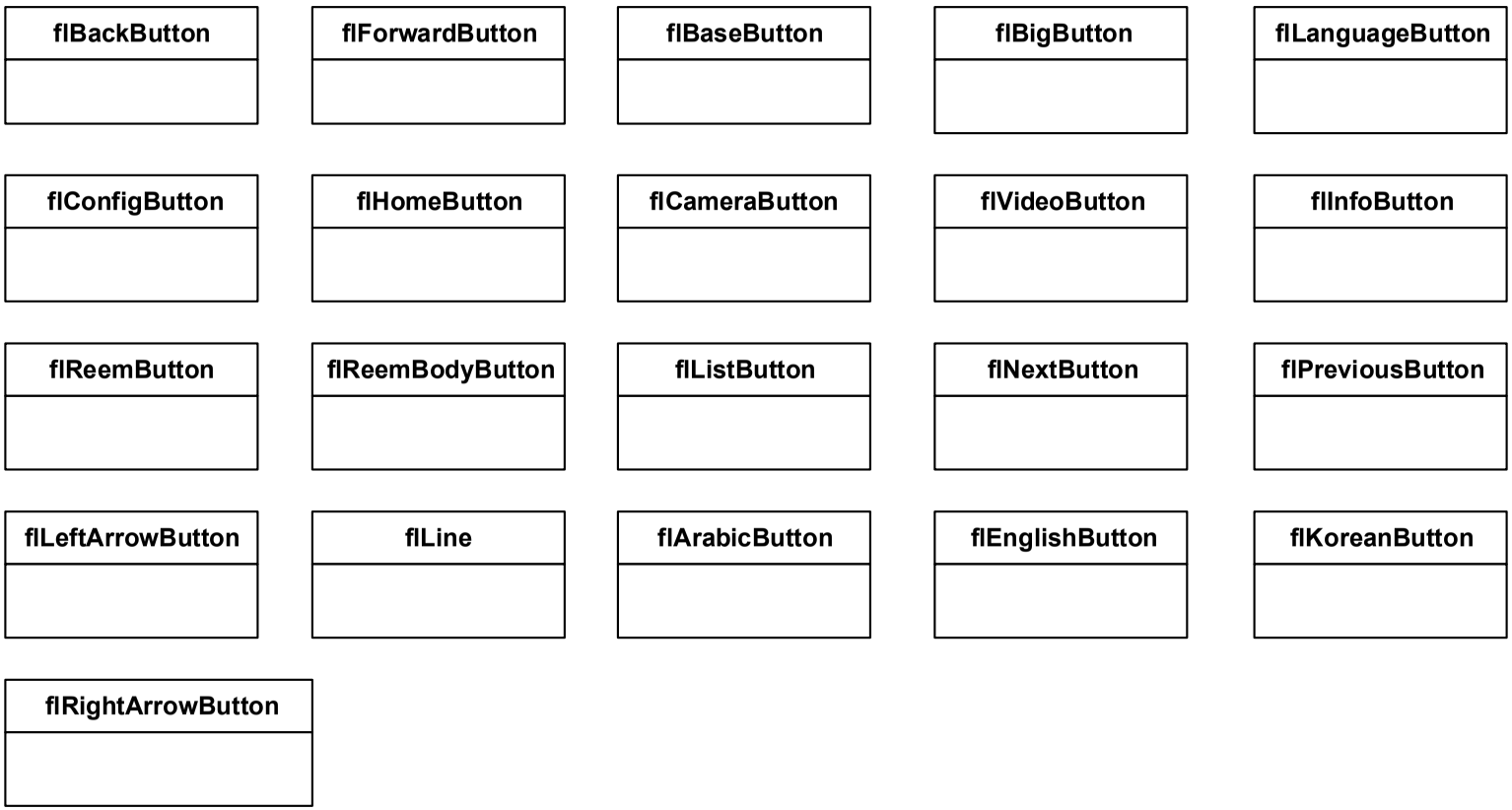

Figure 6-11: Class Diagram: UI Theme Component

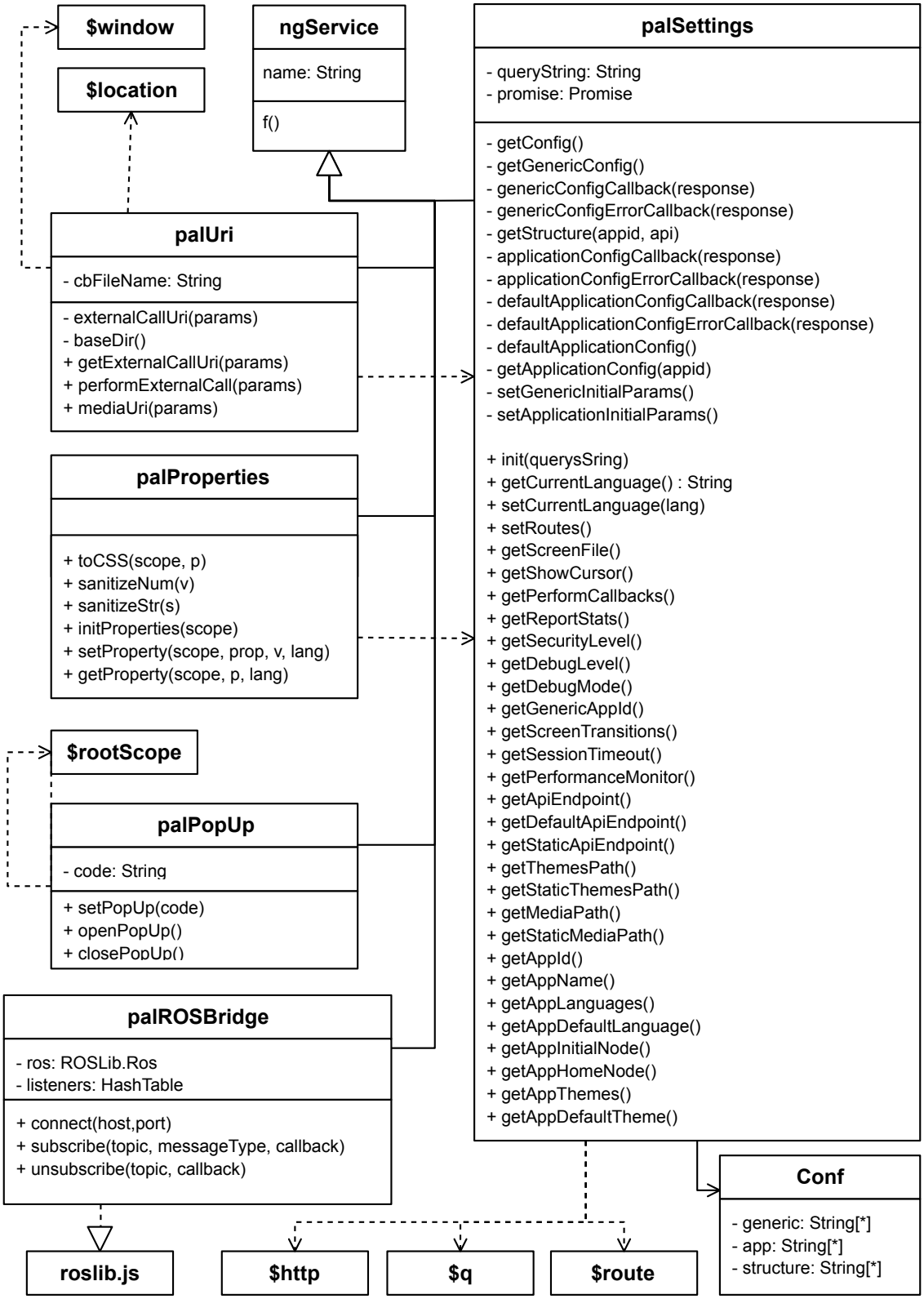

Figure 6-12: Class Diagram: UI services

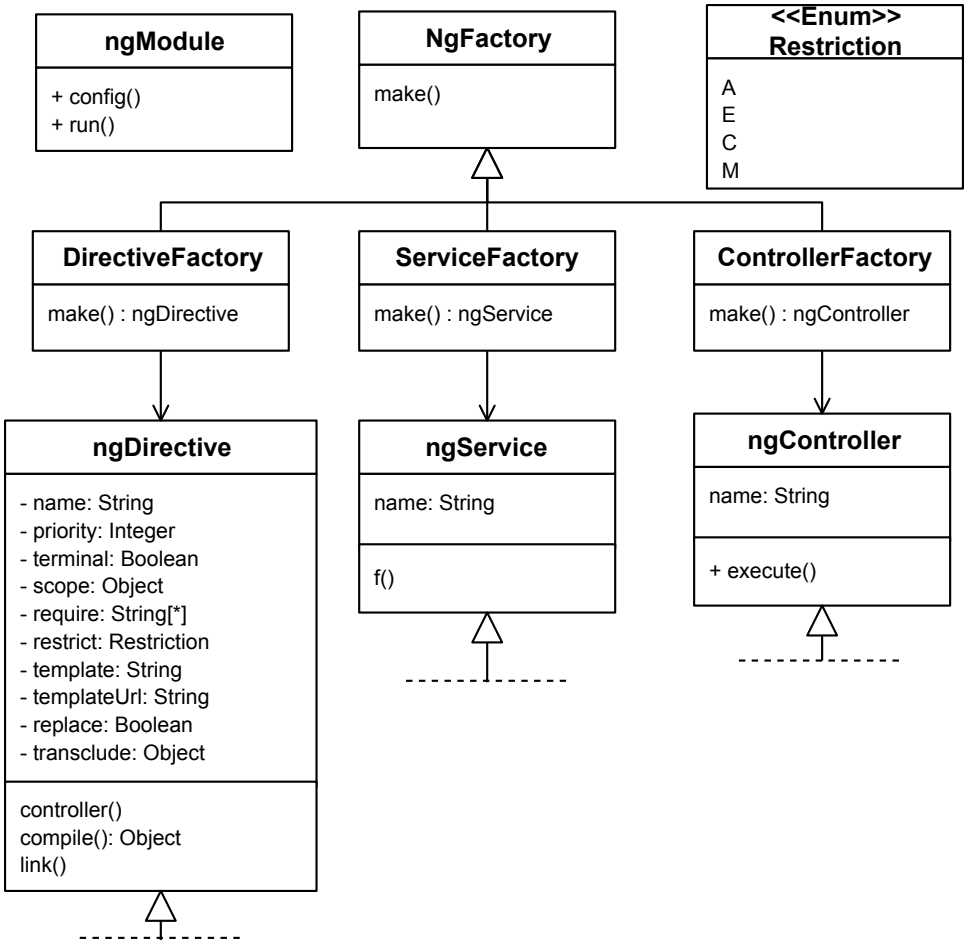

Figure 6-13: Class Diagram: Angular factory

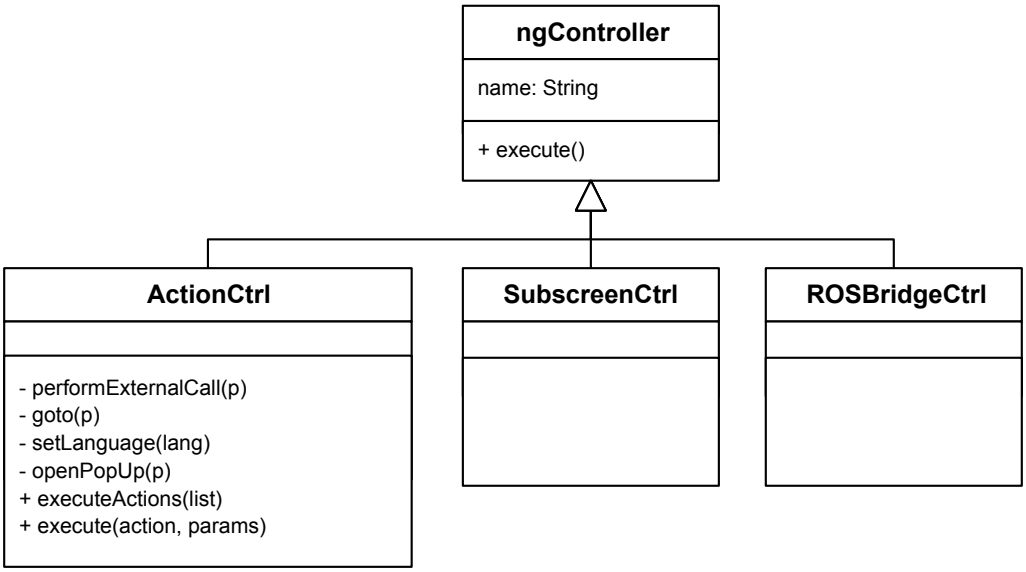

Figure 6-14: Class Diagram: Angular controllers

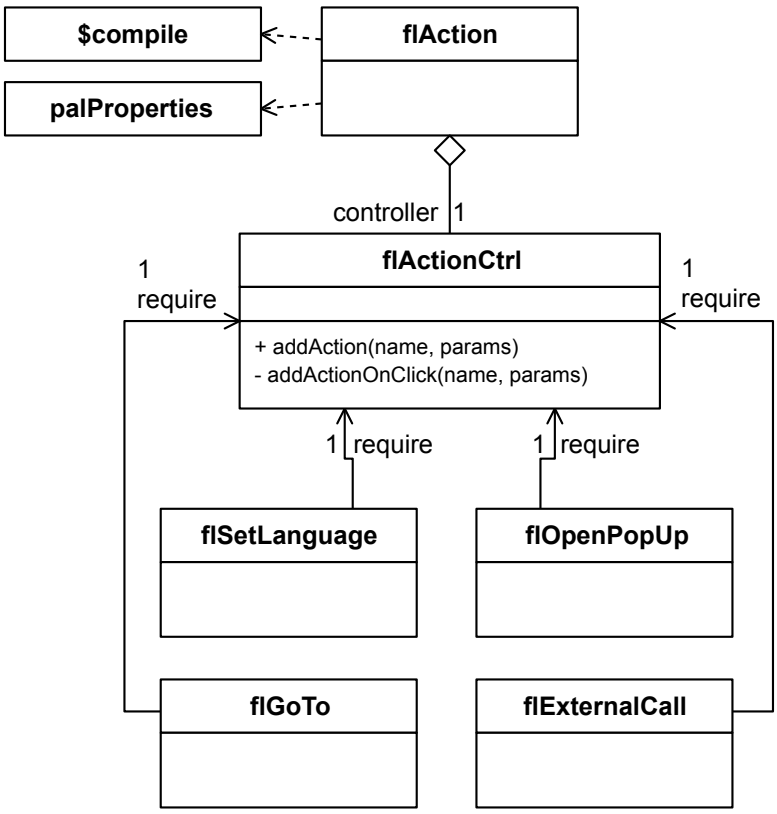

Figure 6-15: Class Diagram: UI action

6.3.2 Packages Diagrams

The concept of package is implemented with Angular modules. Modules can have an array of dependencies.

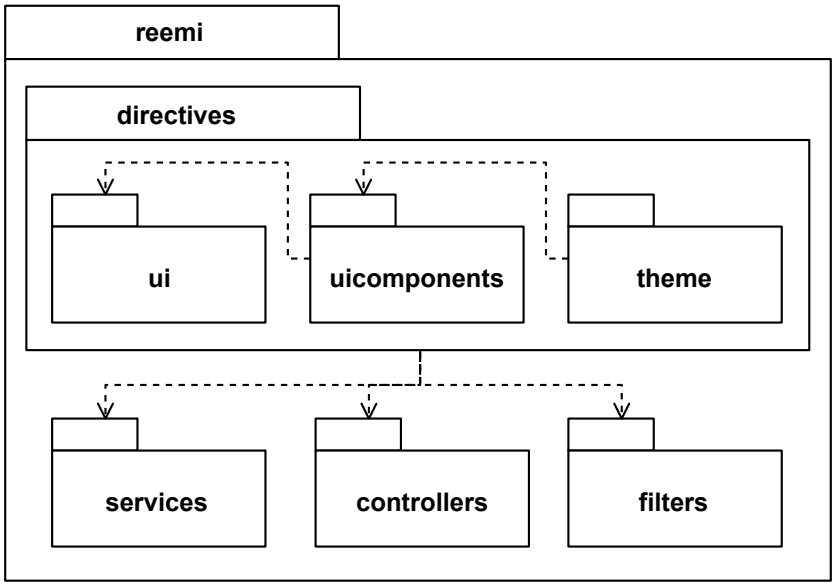

Figure 6-16: Packages Diagram: Application

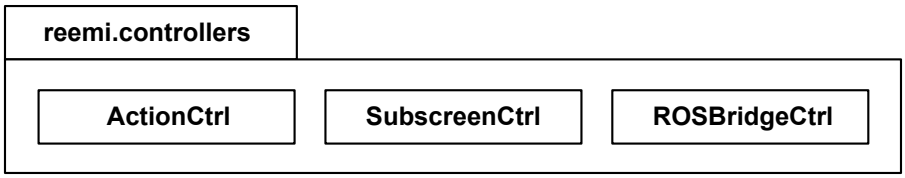

Figure 6-17: Packages Diagram: Controllers

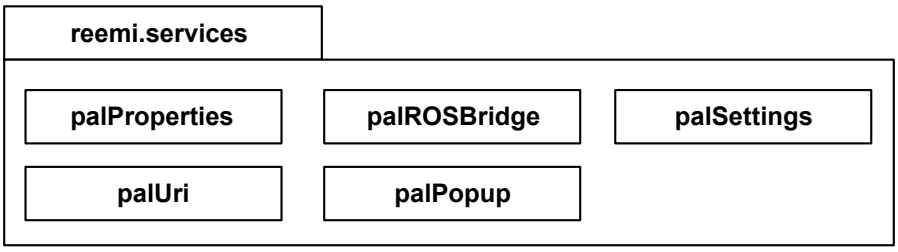

Figure 6-18: Packages Diagram: Services

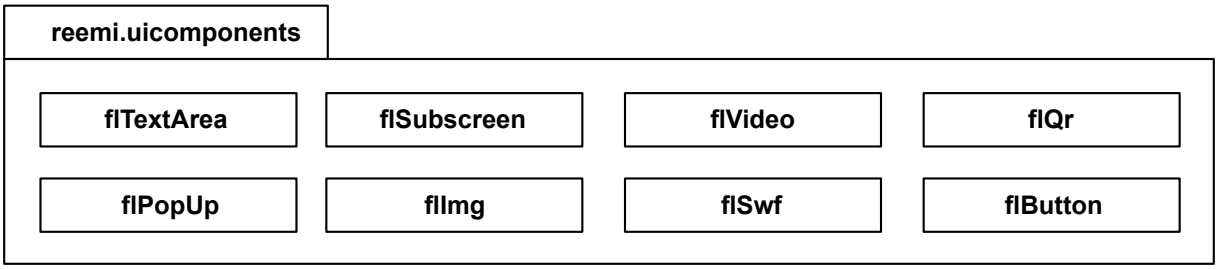

Figure 6-19: Packages Diagram: UI components

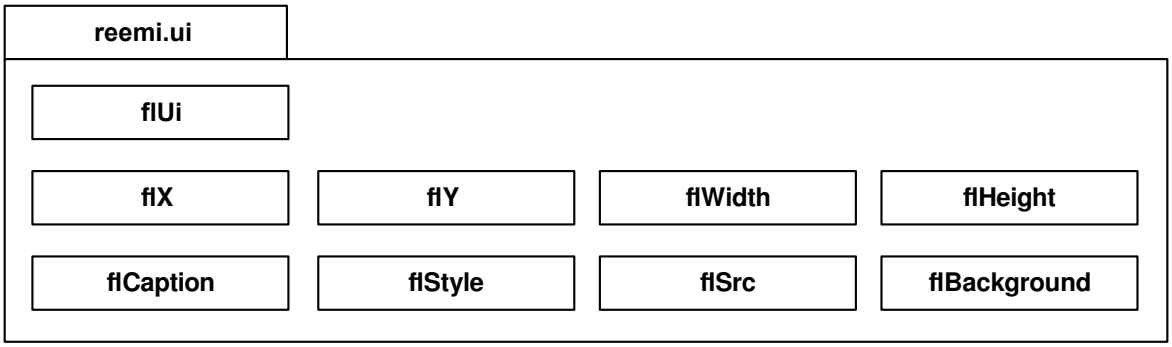

Figure 6-20: Packages Diagram: UI

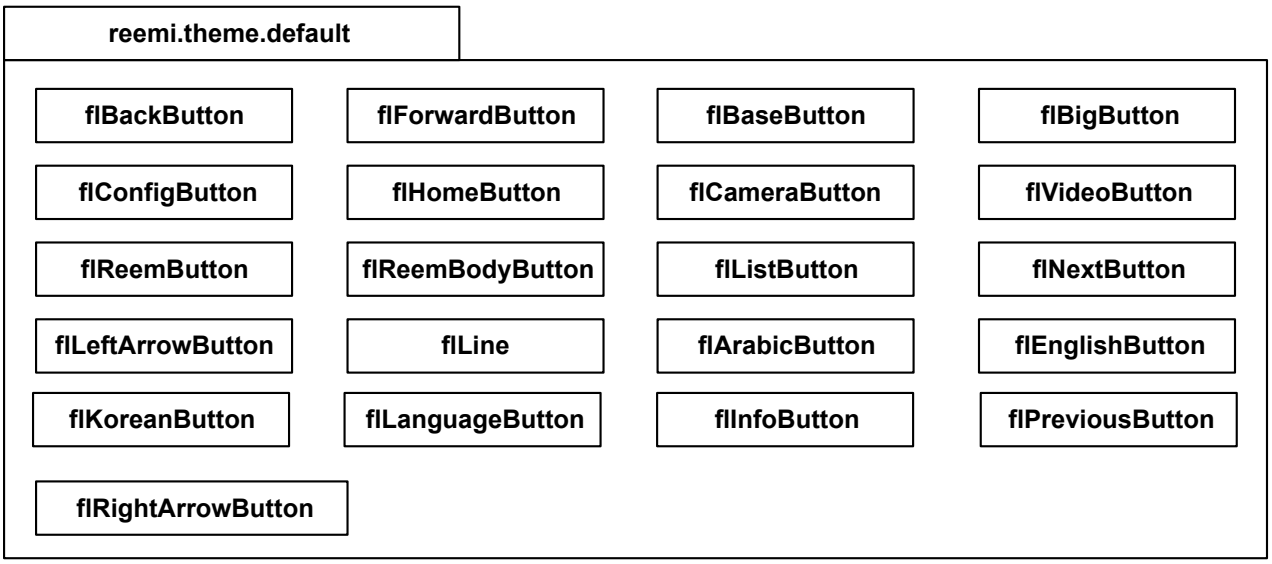

Figure 6-21: Packages Diagram: UI theme components

6.4 Dynamic View

This section contains a detailed description of the most relevant operations. They are grouped in 6 categories: boot-strap and configuration, handling properties and transformation to HTML, internal navigation, and communication with ROS Bridge. It always works the same way: writing the appropriate behaviour in directives (`controller`, `compile` and `link` functions) and injecting the required services.

6.4.1 Boot-strap and Configuration

During the boot-strap process of Angular, the system performs some key operations, like registering modules, setting dependencies and the default routes
(e.g. `http://.../app/index.html#/products-view/14/detail-view`). The Flango Content Manager also fetches de configuration from the backend and builds the local object that represents it. Figures 6-22, 6-23, and 6-24 show the bootstrap of the program.

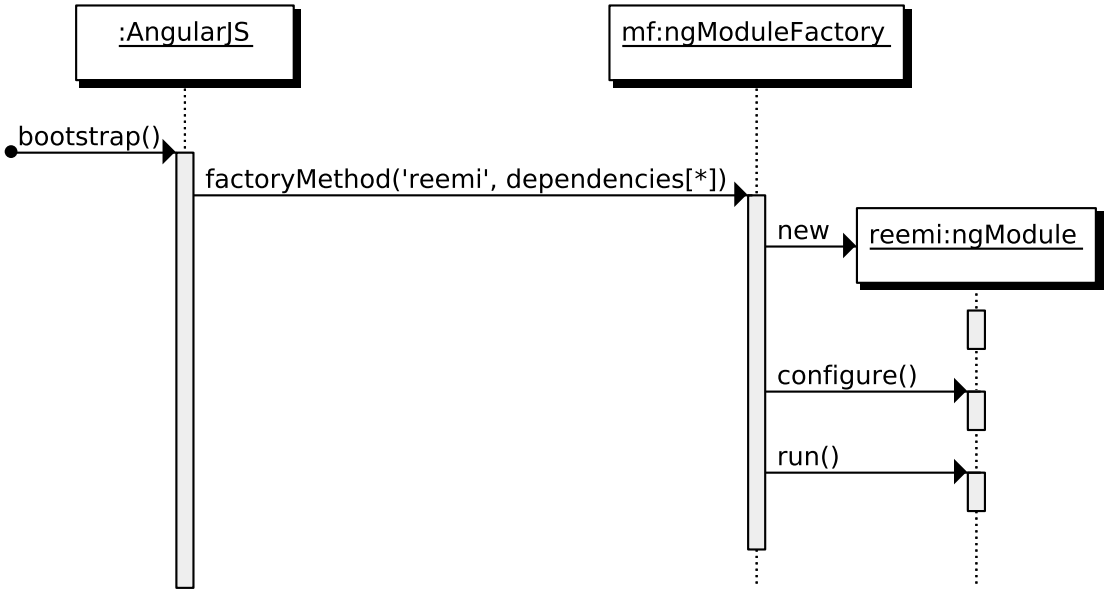

Figure 6-22: Sequence Diagram: Bootstrap 1

The configuration is encapsulated in the `palSettings` angular service which, like all services in Angular, is a singleton. It is created with a factory: when a new instance of `palSettings` is to be created, it runs some code that prepares the instance that is returned. During

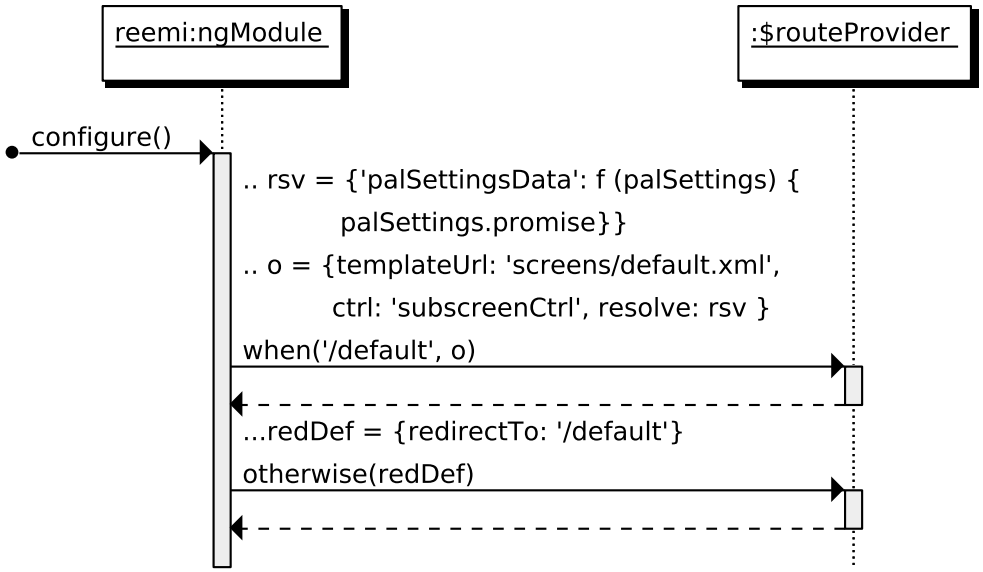

Figure 6-23: Sequence Diagram: Bootstrap 2

the bootstrap process, there is a call to `palSettings.init(queryString)` (figure 6-25 on page 84). This method fetches the configuration from the backend using an asynchronous call (figure 6-26 on page 84) which, in turn, delegates to other internal operations. To avoid errors, the system uses *deferred promises*: the `init` method returns an object immediately and it receives a notification when all asynchronous calls have been completed, that is, when the configuration object is ready to be used in the application.

The configuration object has three parts:

1. **Generic configuration** (figure 6-27 on page 85). Paths, API endpoint, etc.
2. **Application specific configuration** (figure 6-28 on page 86). Available languages, available themes, default language, etc.
3. **Structure** (figure 6-29 on page 87). A graph that matches screens with URI.

It fails gradually: it first tries to fetch the real application. If that fails, it attempts to fetch the default application (figure 6-30 on page 88). If this is not possible, it obtains the static application shipped with the program (See `errorCallback` in figure 6-30 on page 88).

Promises and deferred objects Futures, promises and delays are constructs for synchronising in a programming language. *Promises* were introduced by Daniel P. Friedman

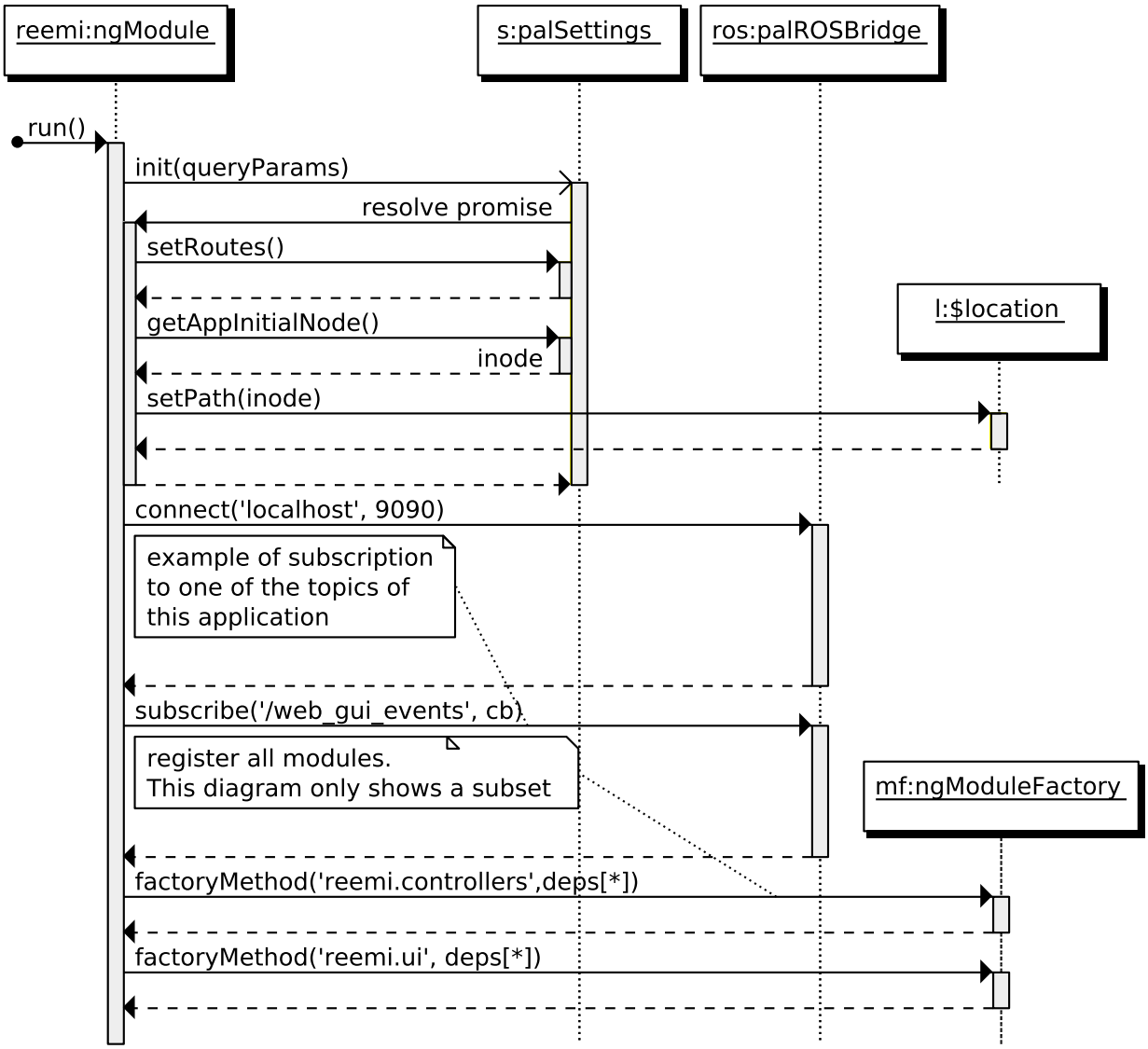

Figure 6-24: Sequence Diagram: Bootstrap 3

and David Wise in 1977, *futures* in 1977 by Henry Baker and Carl Hewitt. The term promise was coined by Liskov and Shrira [18] in 1988.

The three words are often used interchangeably although they have some differences:

- Future: read-only placeholder view of a variable
- Promise: writeable, single assignment container that sets the value of the future.

Essentially, promises represent the result of a task that might have not been completed, something very common in JavaScript. Asynchronous calls are normally handled with callbacks

(see chapter 7 on page 111). However, when there are nested or concurrent asynchronous calls, like the case of fetching the configuration from the Flango Back-End, callbacks become hard to maintain. A solution is defining a promise at the beginning and *resolving* it when all calls are completed (or *rejecting* it when a sufficient number of calls have failed). This way functions can be decoupled and testing is easier.

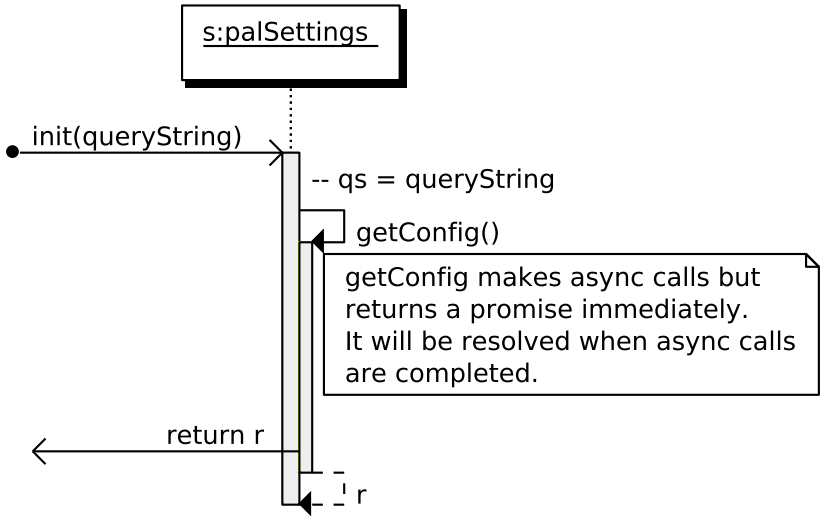

Figure 6-25: Sequence Diagram: init()

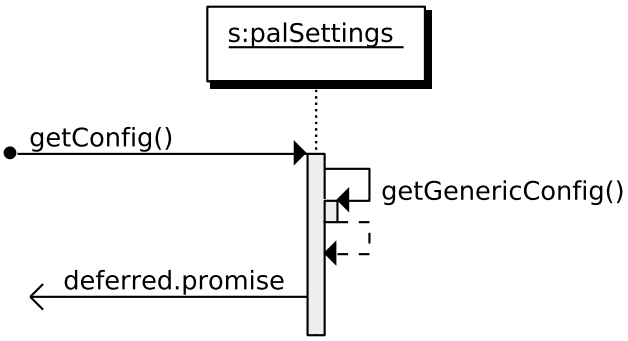

Figure 6-26: Sequence Diagram: getConfig()

Figure 6-27: Sequence Diagram: getGenericConfig()

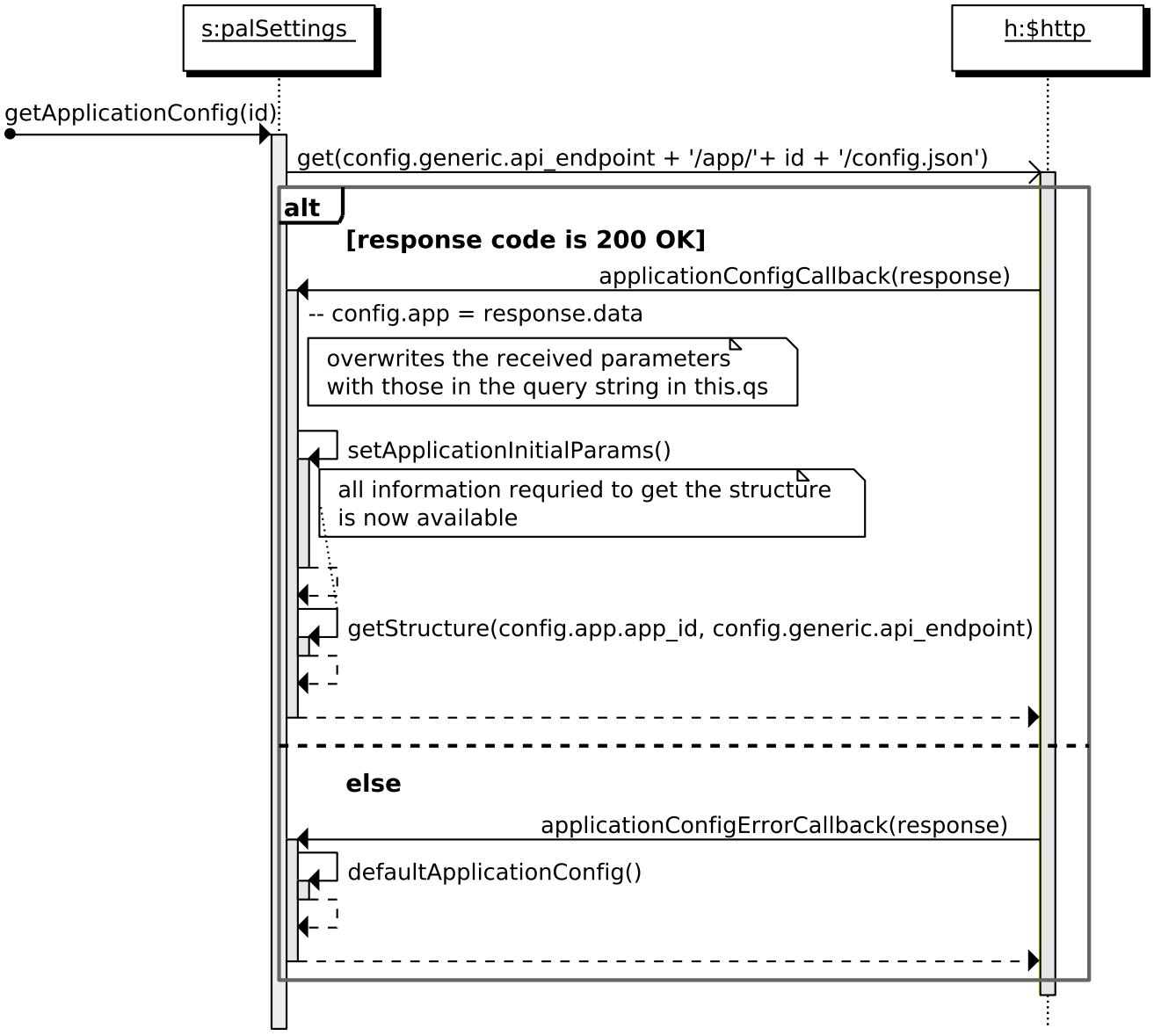

Figure 6-28: Sequence Diagram: getApplicationConfig()

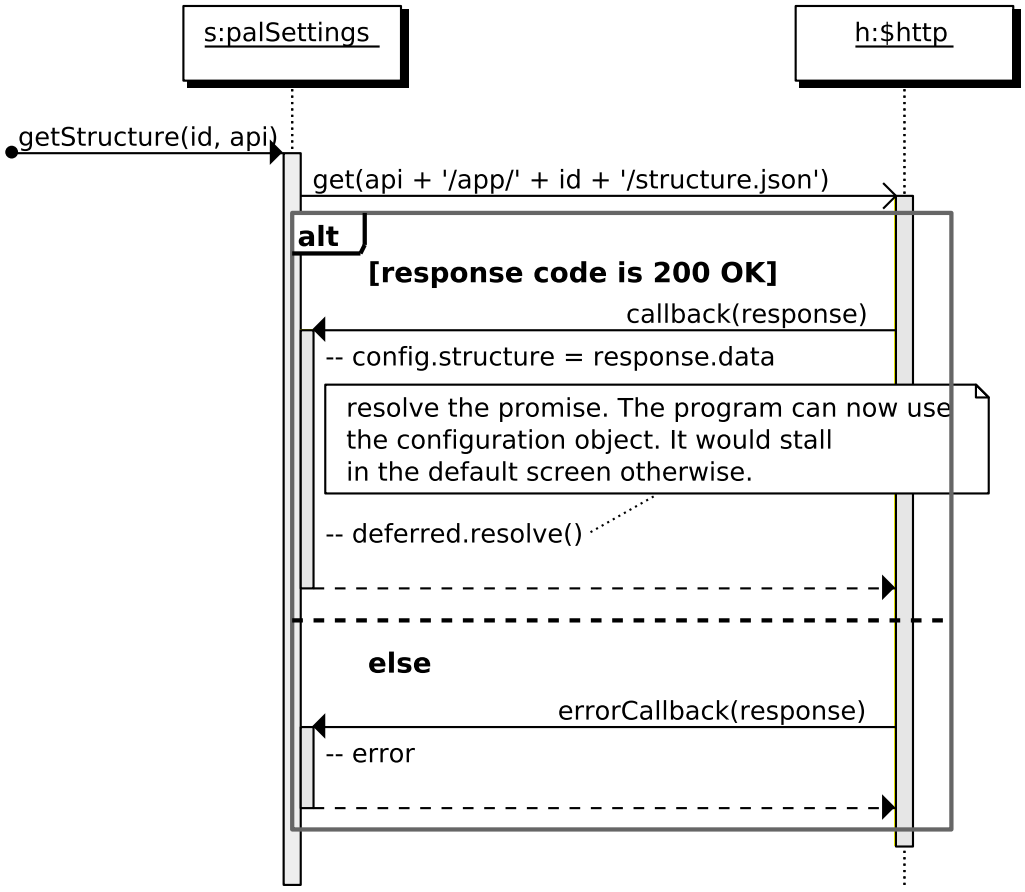

Figure 6-29: Sequence Diagram: getStructure()

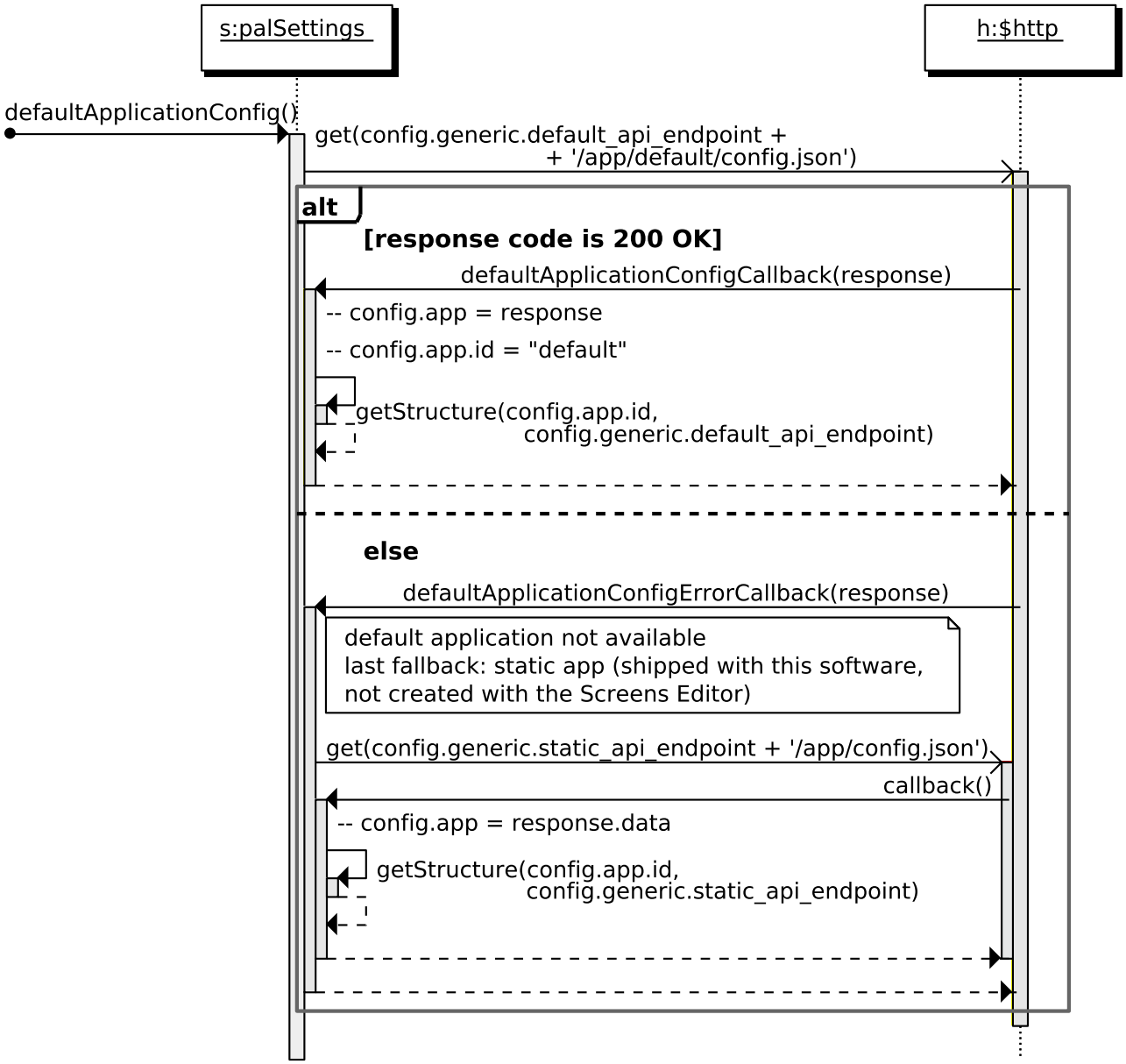

Figure 6-30: Sequence Diagram: defaultApplicationConfig()

6.4.2 Properties management and transformation to HTML

Browsers can only parse HTML nodes. When the parser finds an XML node, it prints the inner text node because HTML also has text nodes. With Angular XML nodes can have behaviour: it teaches the browser new syntax. For example,
`<fl:ui fl:base-button fl:x="100" fl:width="90"></ui>` triggers the directive `flBaseButton`, which encapsulates the behaviour to transform it to HTML:
`<a href="..."><div>text</div></a>`.

To expose values to the view, Angular attaches an object `$scope` to all directives (figure 6-31). A scope can be **isolated**, **prototypically inherited** from the parent, or **shared** with other directives of the same type. This project uses the scope to encapsulate the properties extracted or inferred from the XML node. Thus, any element `<fl:ui>` or concrete types `base-button`, `back-button`, etc has a `scope` object attached that contains all the necessary data to render an HTML node. Scopes are also watched to provide two-way data binding with controllers (see figure 6-4 on page 67)

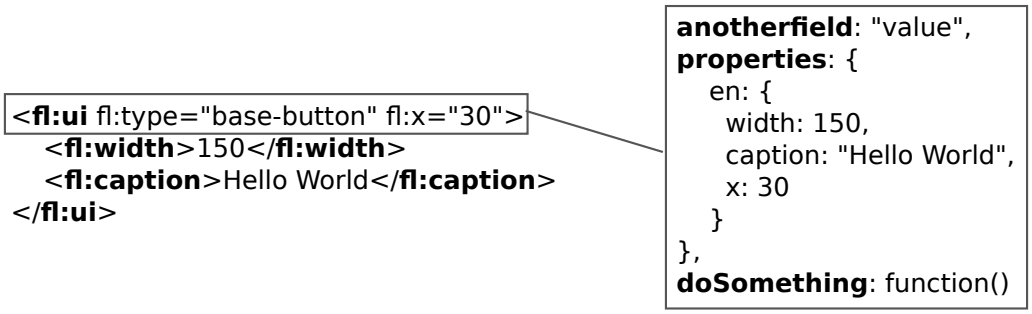

Figure 6-31: Scope object attached to a node

```
<fl:ui fl:id="3" fl:type="base-button">
  <fl:x>124</fl:x>
  <fl:y>123</fl:y>
  <fl:width>100</fl:width>
  <fl:height>100</fl:height>
  <fl:onclick>
    <fl:action type="goto">
      <fl:param fl:name="uri" fl:value="details-view"/>
    </fl:action>
  <fl:caption fl:lang="ca">ignici&oacute;!</fl:caption>
  <fl:caption fl:lang="en">ignite!</fl:caption>
  <fl:caption fl:lang="fr">allez!</fl:caption>
```

```
13   </fl:ui>
```

Listing 6.3: Original XML BaseButton

```
properties: {
    ca: {
        // x,y,w,h and clickchain same as default language
        caption: 'ignici&oacute;!'
    }
    en: {
        x: 124
        y: 123
        width: 100
        height: 100
        clickchain: {
            goto: {
                uri: 'details-view'
            }
        }
        caption: 'ignite!'
    }
    fr: {
        // x,y,w,h and clickchain same as default language
        caption: 'allez!'
    }
}
classes: {
    ui:
        [ 'fl-base-button', 'fl-button' ]
    style:
        [ ]
}
```

Listing 6.4: Properties object

Directives can be **restricted** to **element**, **attribute**, **class** or **comment** (figure 6-13 on page 77). Flango Content Manager is designed to adapt to this syntax and to decouple and reuse elements as much as possible. All UI Components have the tag `<fl:ui>` and an attribute `fl:type` that defines the concrete type. Instead of having all behaviour in one directive `fl:ui` (restricted to Element), it uses `flUi` to rewrite the DOM:

`<fl:ui fl:type="base-button"></ui>` becomes

`<fl:ui base-button></ui>`, where `base-button` is an attribute of the element `fl:ui`. There is a directive `flBaseButton` (restricted to Attribute) that encapsulates the behaviour of this tag. Likewise, it initialises the properties object for this `ui` element and stores the

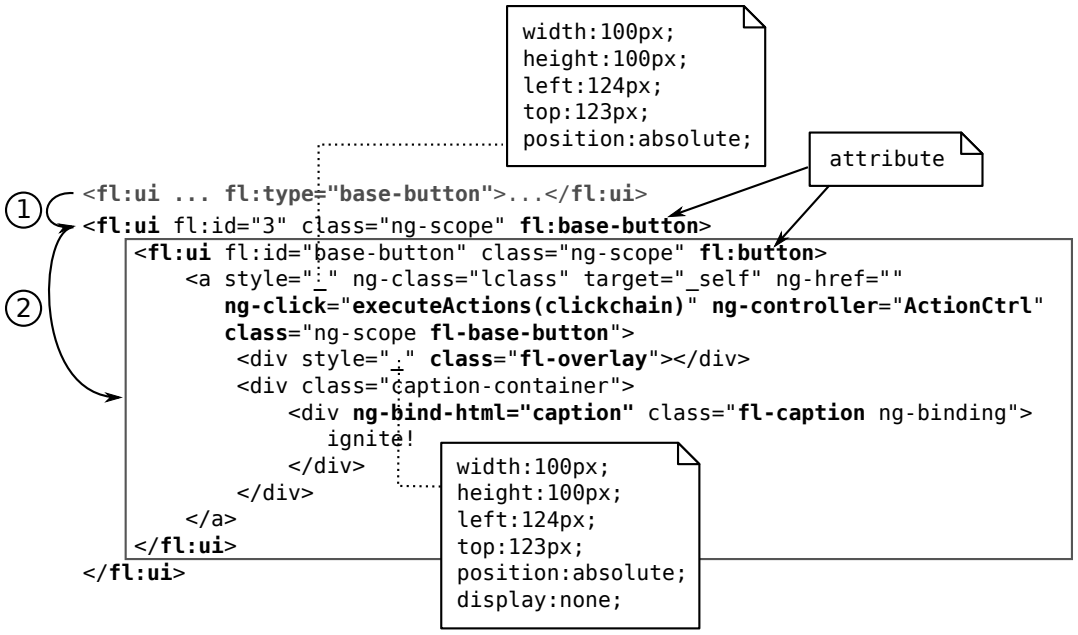

Figure 6-32: Transformation from XML to HTML and styles management

type that will be later used as a value for the `class` attribute in the HTML element. The stylesheet of the theme defines the inheritance of styles: a base-button is implemented with a button and, therefore, inherits all properties of the class `button`.

Figure 6-32 shows the transformation of a base button in the XML screen (listing 6.3) and the use of the properties object in the scope (listing 6.4).

The algorithm to decide the properties for an object and draw it with HTML goes with the flow of the framework:

1. With directive `flUi`: Create an attribute with the value of the `type` attribute (figure 6-33 on the following page). Initialise the properties object in the controller (figure 6-34 on page 93). The framework compiles and links the rest of the code (e.g. the `link()` function in inner `width` tags to read values of inner tags (figure 6-36 on page 95))
2. With directive `flUi` link function (figure 6-35 on page 94), read in-line attributes. Because it reads everything recursively, this is the last function to run. Properties are only set if they have not been set yet: inner tags (more specific) have higher priority than in-line attributes. Recompile the directive to make angular find it.
3. With the new directive (e.g. `flBaseButton`), run component-specific behaviour. `flUi`

does not have a `type` attribute anymore and is ignored. Specific behaviour can be transforming it using a template (figure 6-37 on page 96). Recompile (e.g. the template might, and normally has, tags that can trigger angular directives).

4. Eventually reach a base component and run component-specific behaviour, e.g. create HTML nodes that can actually draw the UI Component.

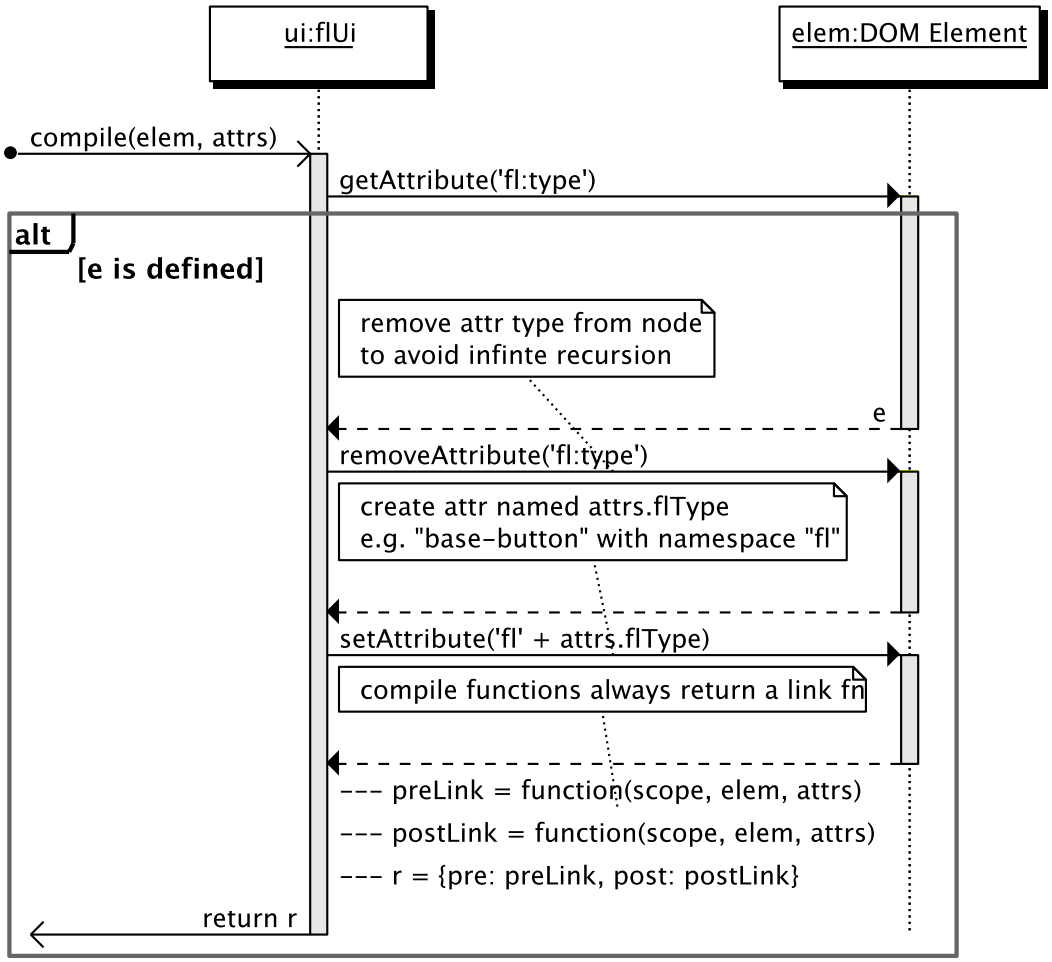

Figure 6-33: Sequence Diagram: flUi::compile

`<fl:ui base-button>...</ui>` eventually becomes
`<div class="base-button" style="...">...</div>`. With this rewrite strategy the directive eventually outputs real HTML that defines the structure of the UI Component and uses the style sheet.

All UI Components have an **HTML template** that defines their structure, a **CSS class** that defines their style, **in-line CSS** that defines their exclusive properties (e.g. position,

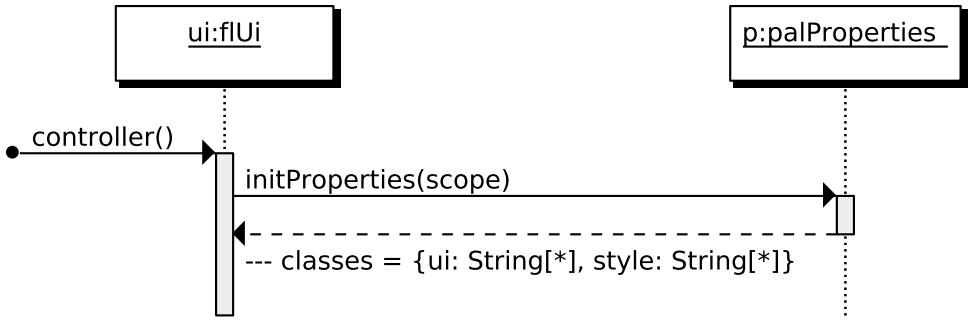

Figure 6-34: Sequence Diagram: flUi::controller

size...) and they are binded to `ActionCtrl` to expose behaviour (e.g. on click).

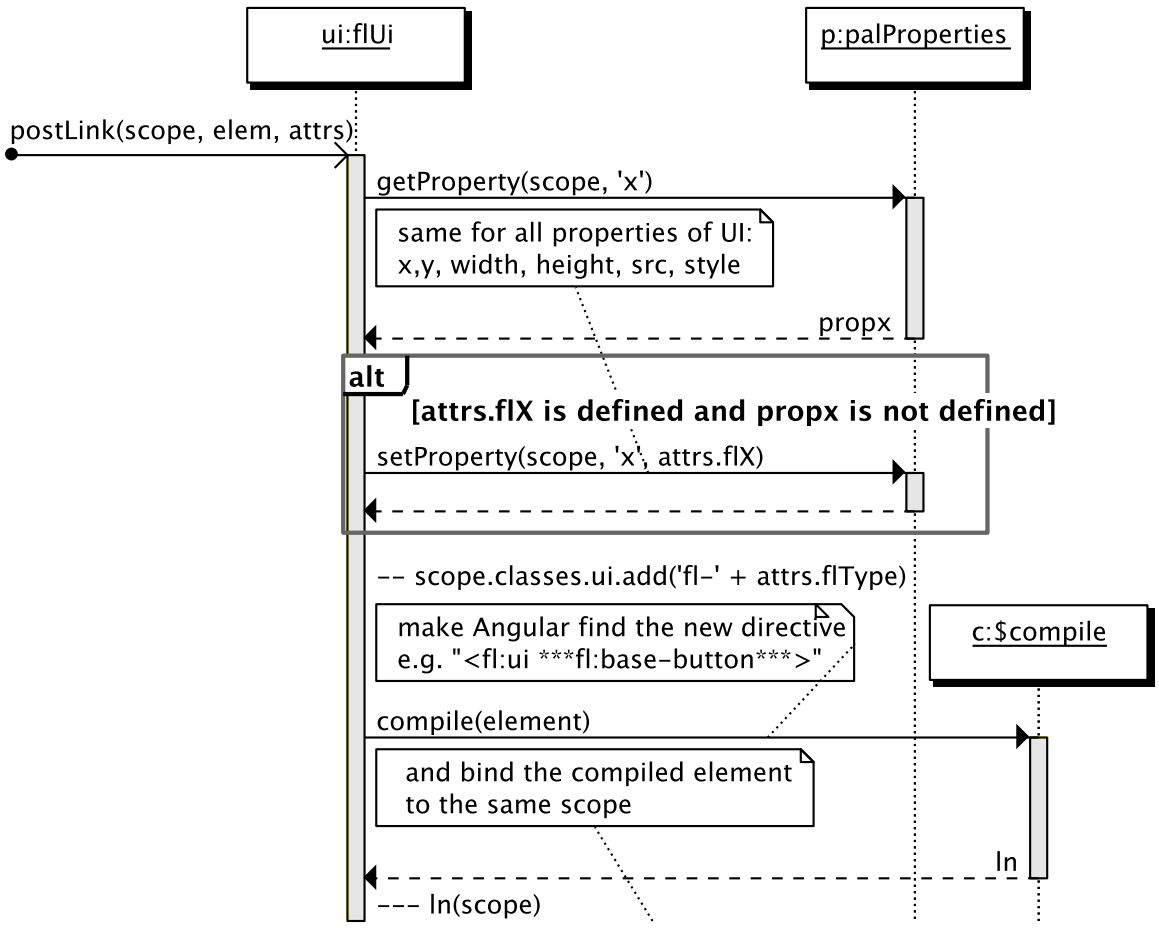

Figure 6-35: Sequence Diagram: flUi::link

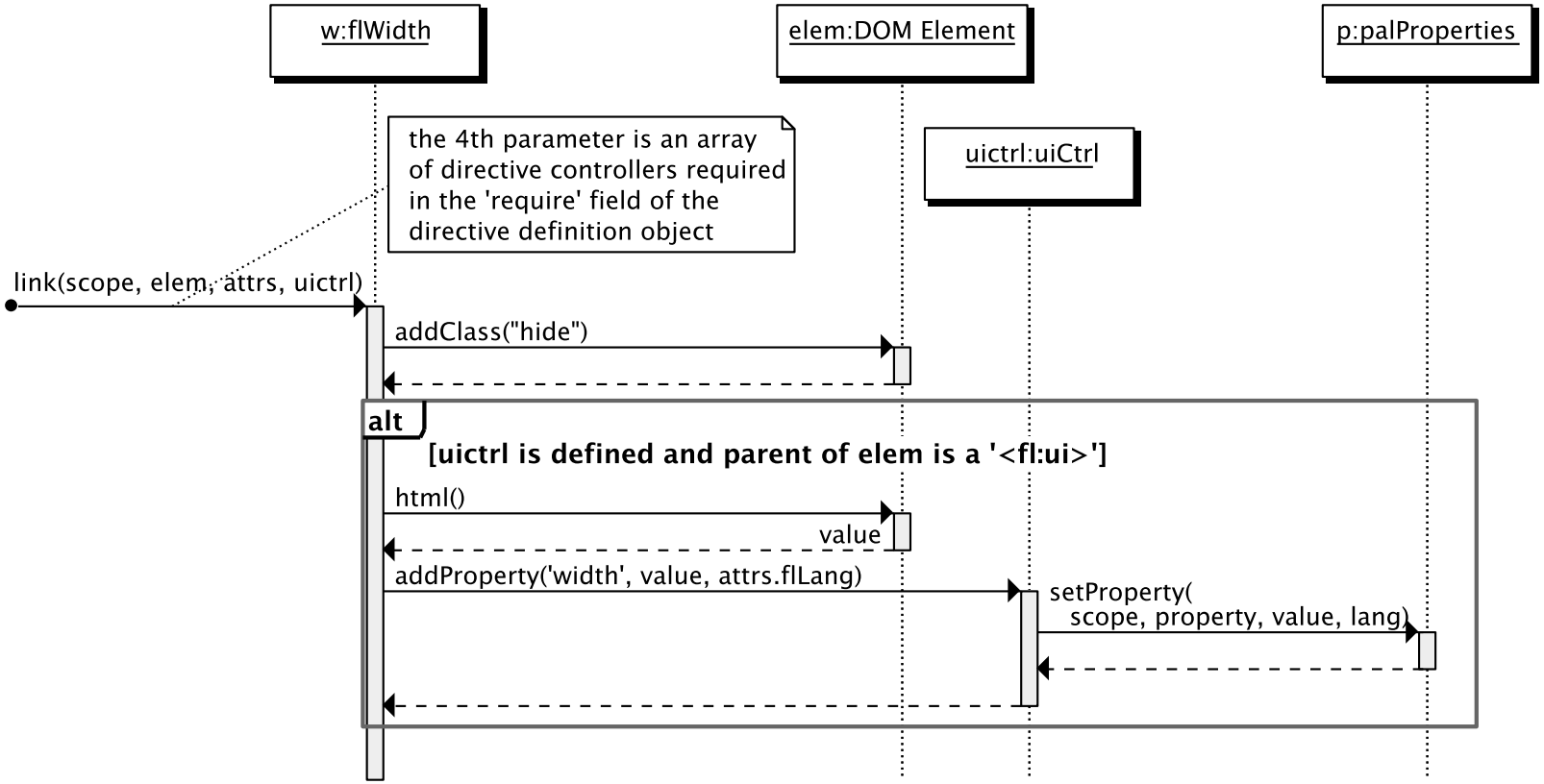

Figure 6-36: Sequence Diagram: flWidth::link

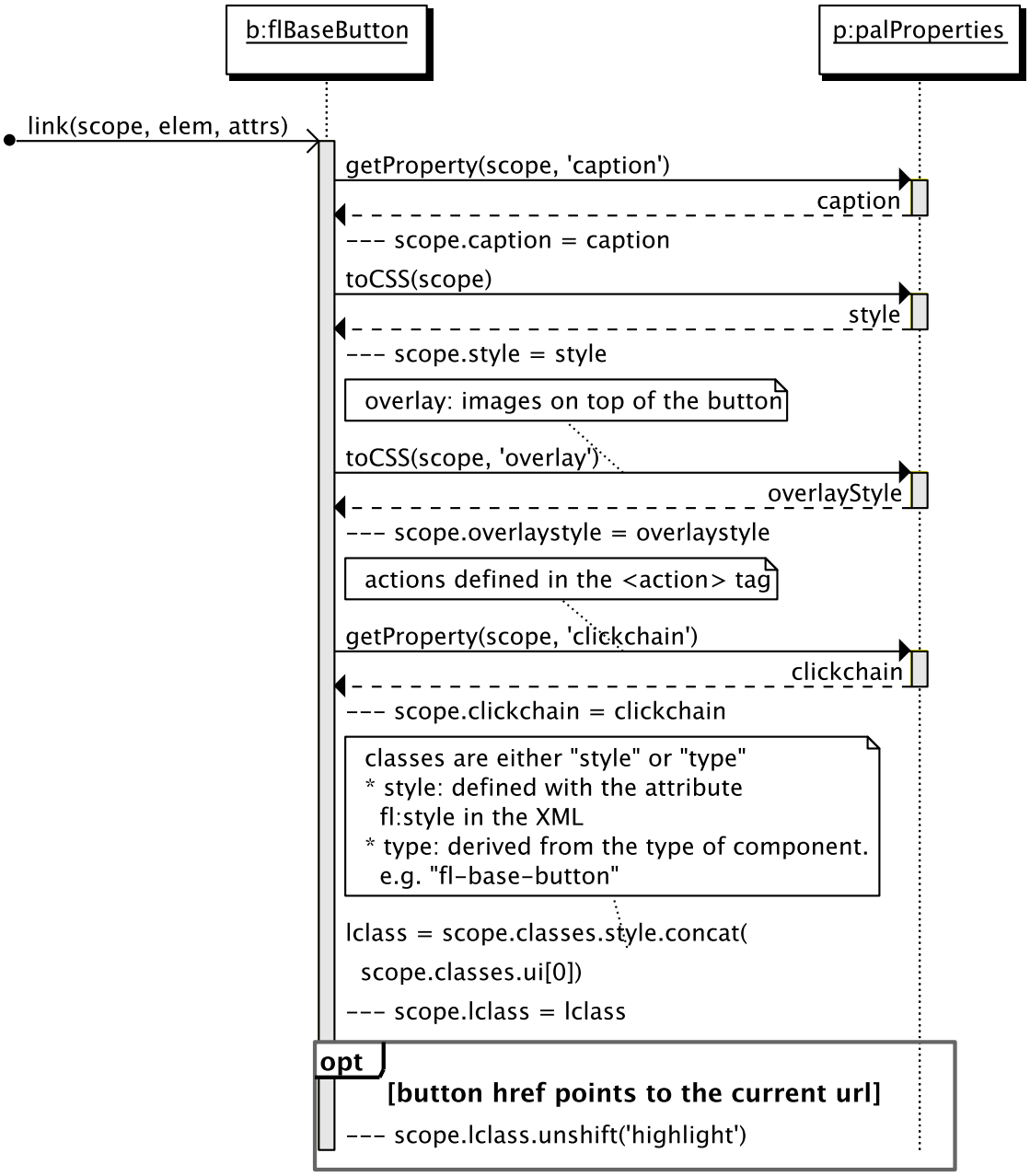

Figure 6-37: Sequence Diagram: flBaseButton::link

Inheritance and composition UI Components can be composited: a `home-button` is implemented with a `base-button`, which in turn is implemented with a `button` that eventually becomes HTML. Each component can have default values for properties. For instance, if `base-button` has a `width` of 100px in the template, and a component that is implemented with `base-button` (e.g. `home-button`) does not define a different width, it sets `width` to 100 in the properties object. This behaviour is achieved with the `compile` and `link` functions of directives. Language-specific properties are decided the same way: it attempts to use current language properties, if they are not defined, it falls back to the default language.

6.4.3 Actions execution

Buttons and other UI Components, like images or QR Codes, respond to clicks. XML can define a list of actions to execute (e.g. set the language, open a popup, go to URI...) The controller ActionCtrl encapsulates the logic to process these commands. Actions are declared in the UI Component (listing 6.5). Directives set the corresponding field in the properties object that represents the chain of actions to execute on click.

```
1  <fl:ui fl:id="english-button" fl:type="base-button">
2      <fl:layover fl:src="[...]/en.png" />
3      <fl:onclick>
4          <fl:action type="set-language">en</fl:action>
5          <fl:action type="external-call">
6              <param name="palEvent" value="setLanguage"/>
7              <param name="language" value="en_GB"/>
8          </fl:action>
9      </fl:onclick>
10 </fl:ui>
```

Listing 6.5: A list of actions in XML

```
1  <a ... ng-click="executeActions(clickchain)"
2         ng-controller="ActionCtrl"
3         class="ng-scope fl-english-button" target="_self" >
4  ...
5  </a>
```

Listing 6.6: Result button (snippet)

When the component is clicked, it calls `executeActions()`, which has a list of actions to execute as a parameter using the *command pattern*.

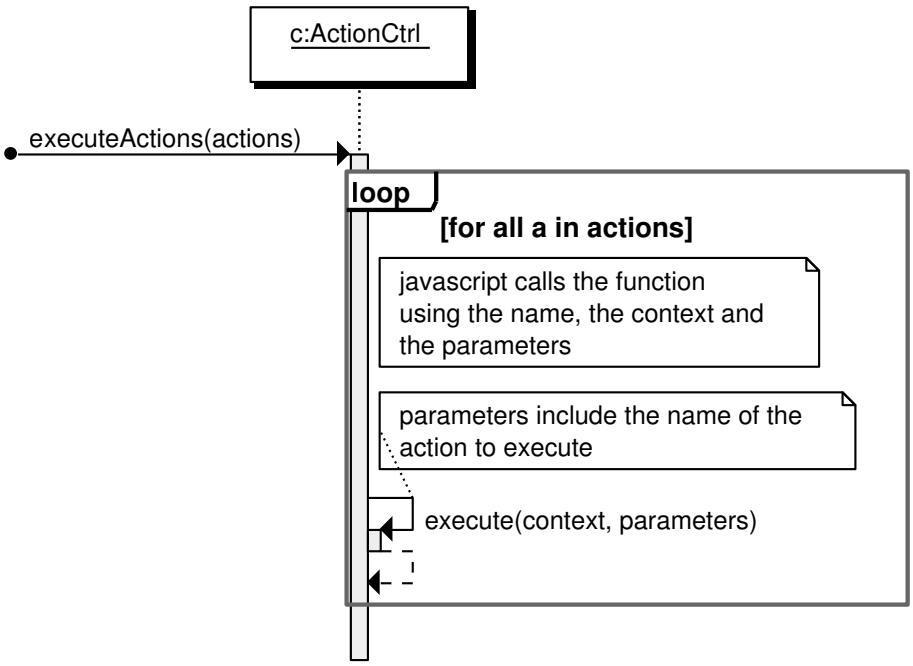

Figure 6-38: Sequence Diagram: ActionCtrl::executeActions

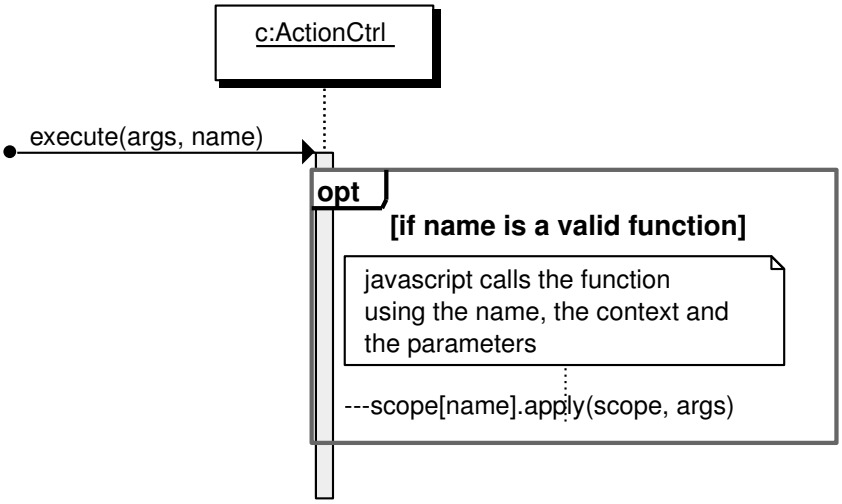

Figure 6-39: Sequence Diagram: ActionCtrl::execute

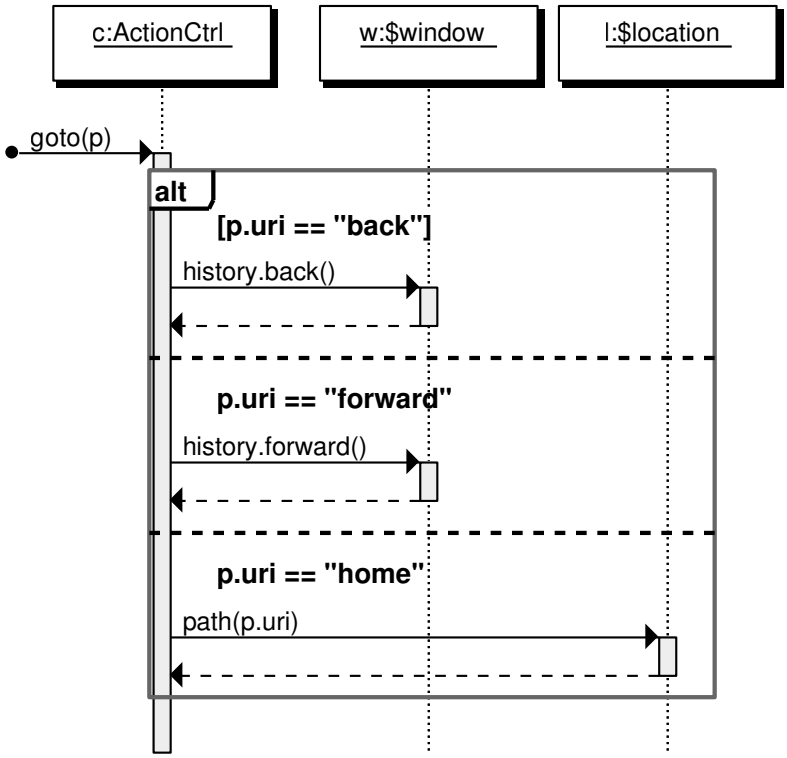

Figure 6-40: Sequence Diagram: ActionCtrl::execute

6.4.4 Internal Navigation

Angular has a navigation system based on URLs. The $route service manages a map that relates URLs with screens, HTML files with directives. On each location change, Angular looks up the destination URL in the map to fetch the correct file.

Flango Content Manager composites screens with subscreens and decides the contents to show using the URL (figure 6-41). e.g. the URL `/main-view/animals-list` does not populate the subscreen `red-panda-details`

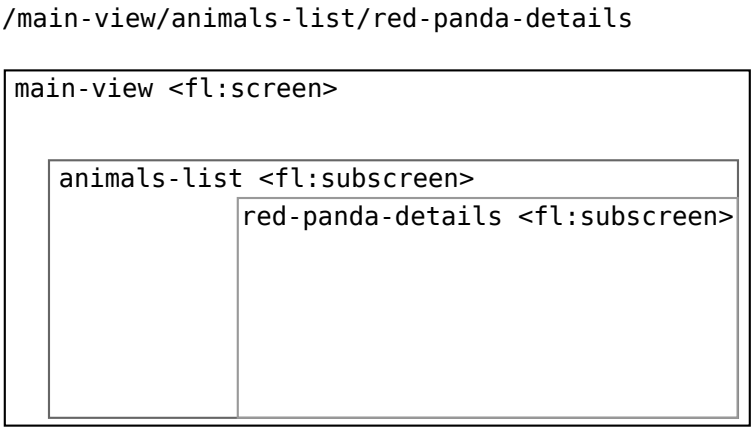

Figure 6-41: Internal navigation with screens and subscreens

The behaviour of subscreens is defined in the `subscreen` directive (figure 6-42 on the facing page).

Only after properties of the UI Component Subscreen have been set (and the corresponding HTML element created), Angular includes the file of the subscreen.

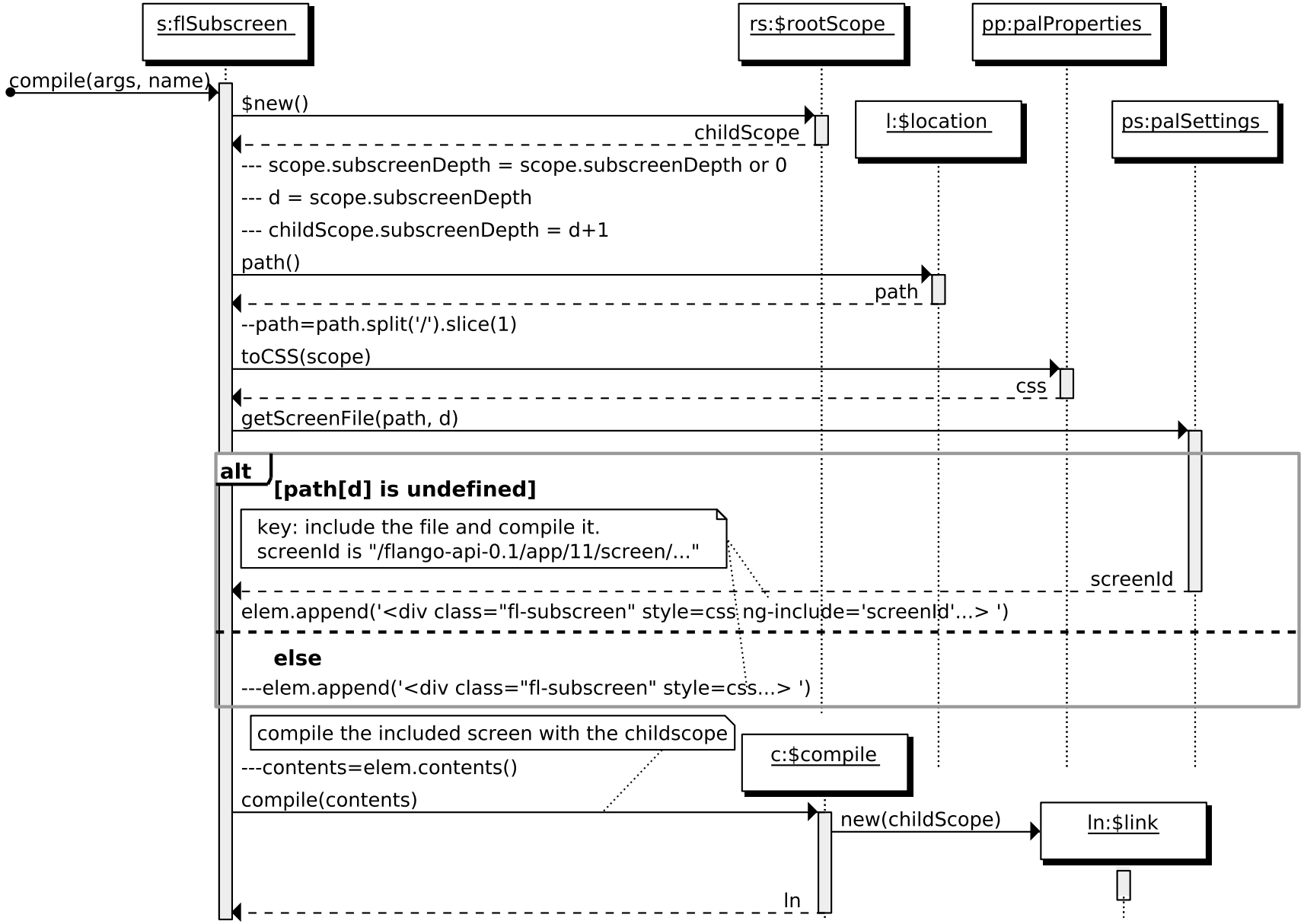

Figure 6-42: Sequence Diagram: Subscreen::compile

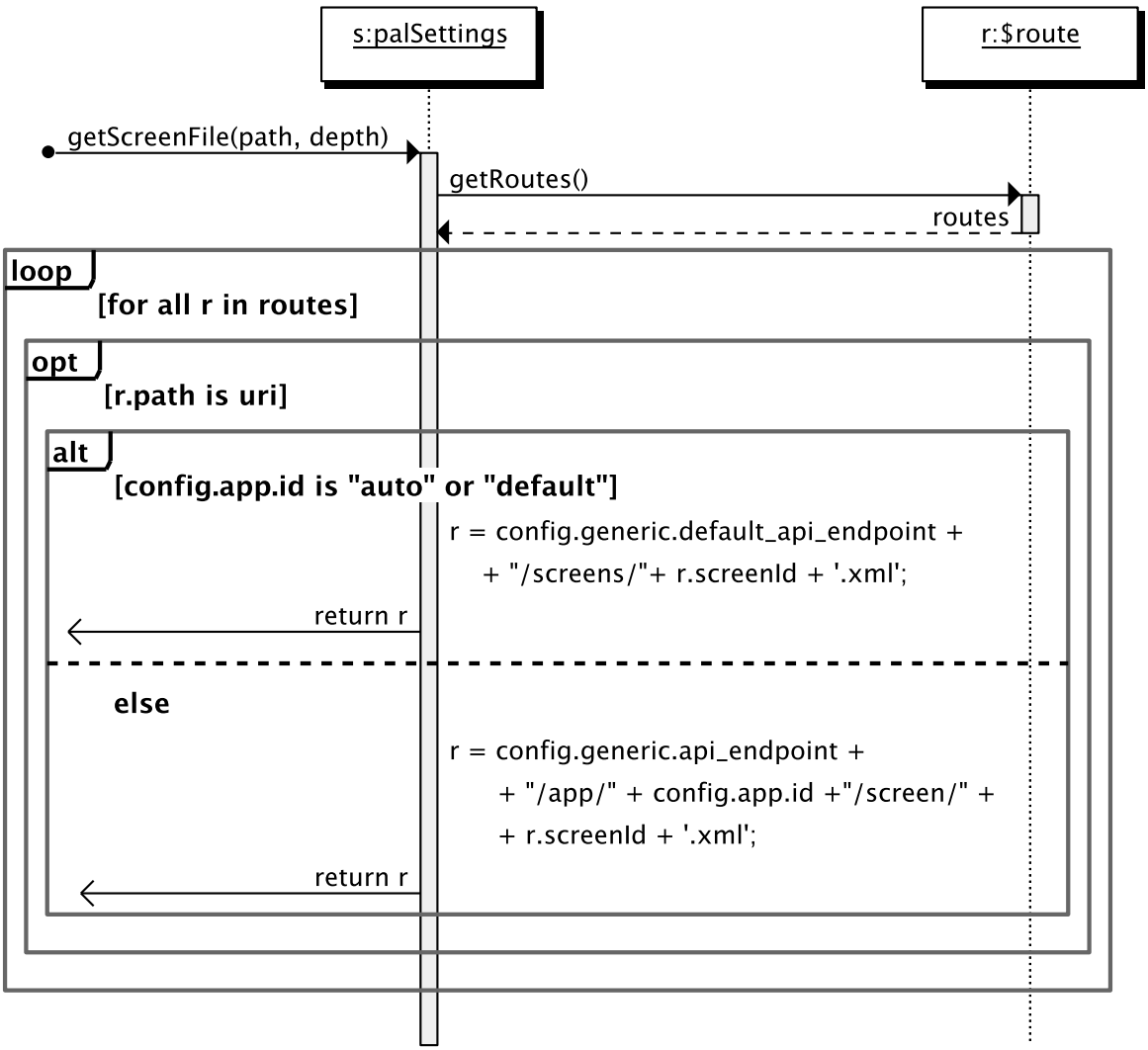

Figure 6-43: Sequence Diagram: palSettings::getScreenFile

6.4.5 External Calls

The `palROSBridge` service can be injected to any component that needs to publish to a topic. palROSBridge.publish(topic, message) uses the roslibjs library to publish the topic. Figure 6-44 illustrates the interaction between a component and `palROSBridge` service.

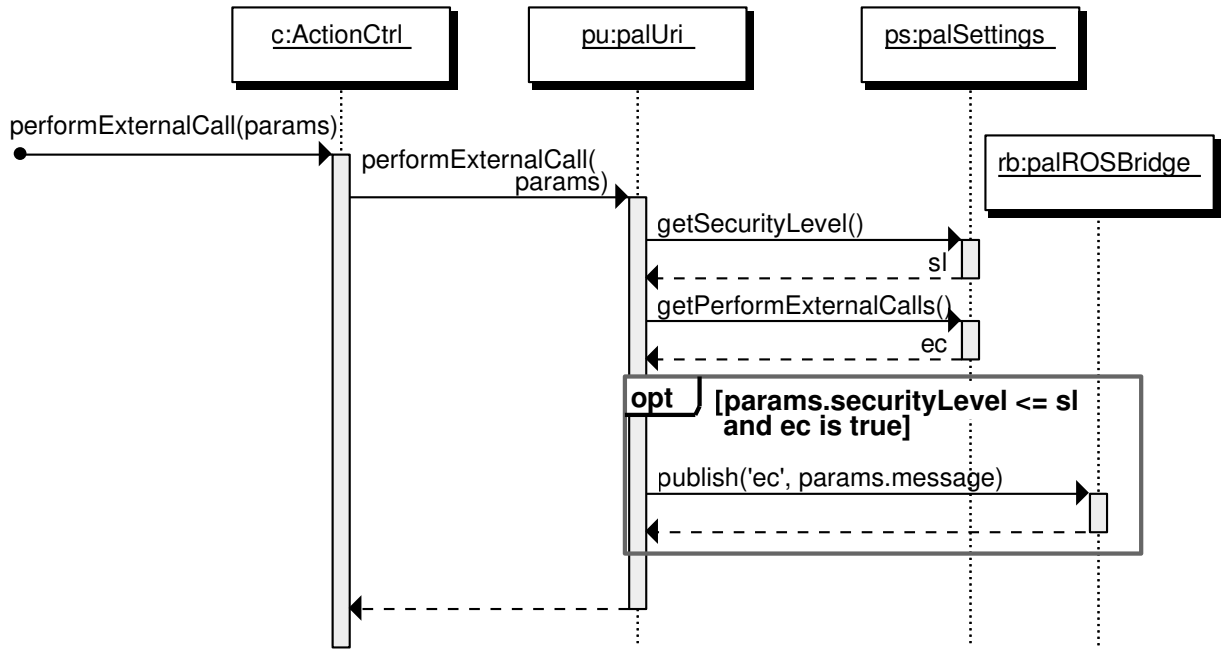

Figure 6-44: Sequence Diagram: Publishing to a topic

6.4.6 Responding to Requests From The Robot

The previous section demonstrated how to publish a message to a ROS Topic. This section shows how the facade ROSBridgeCtrl behaves like a service layer and exposes logic to ROS (figure 6-45 on the following page): it subscribes to the topic that all components of the robot can use to send data to the Flango Content Manager. Whenever a request is received, ROSBridgeCtrl delegates the responsibility to the correct component (figure 6-46 on page 105).

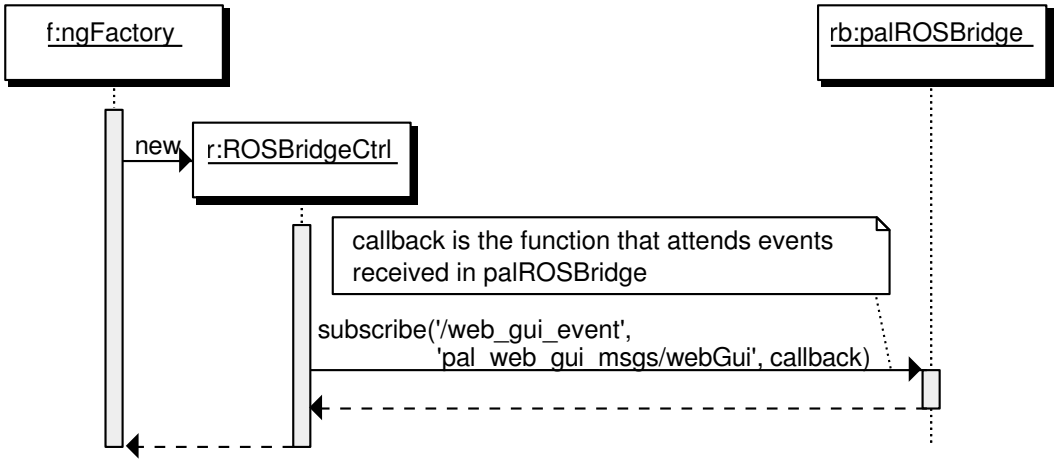

Figure 6-45: Sequence Diagram: ROSBridgeCtrl::instantiation and subscription to topic

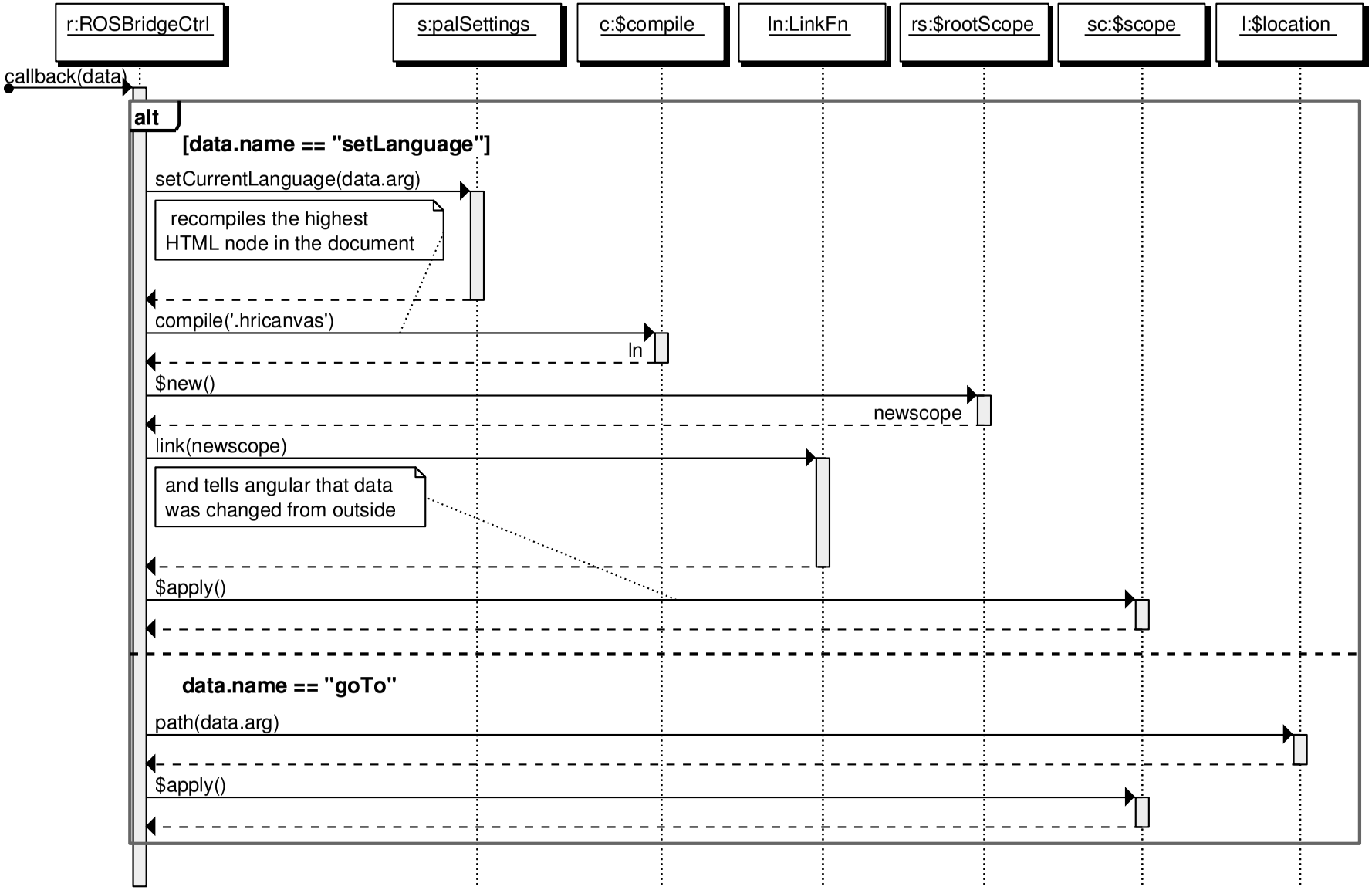

Figure 6-46: Sequence Diagram: ROSBridgeCtrl::Example callback function

6.5 Physical View

This section presents a series of diagrams to illustrate the physical layout of the project: the nodes involved, the components in each node, and a clear separation of the Flango Content Manager and the environment. The application is assembled in a Debian package and deployed to Basestation and the robot. The control scripts of this package perform tasks like copying the files to the correct place (e.g. the public html folder of the web server), initialise data in the backend, etc. The diagrams in this section show the system after the installation.

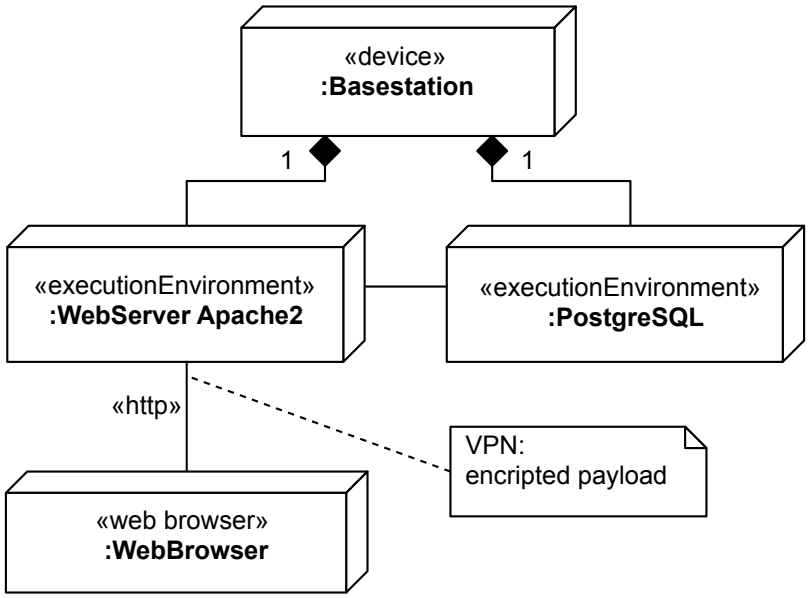

Figure 6-47: Deployment Diagram: Basestation

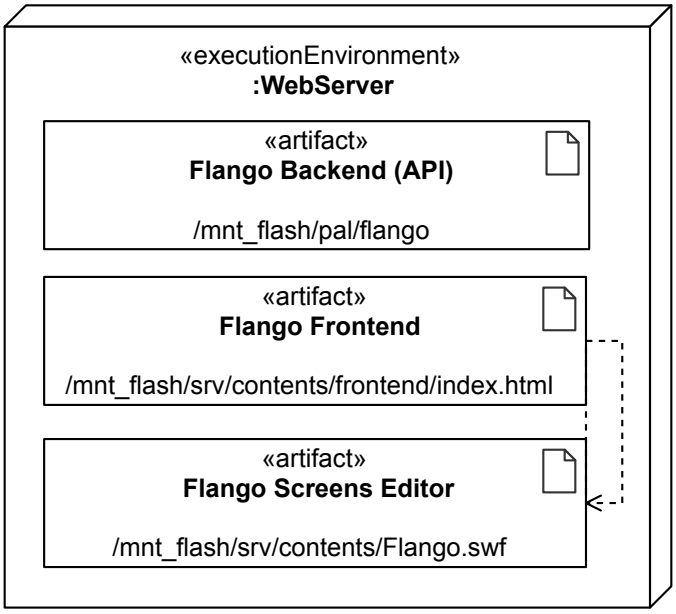

Figure 6-48: Deployment Diagram: Webserver

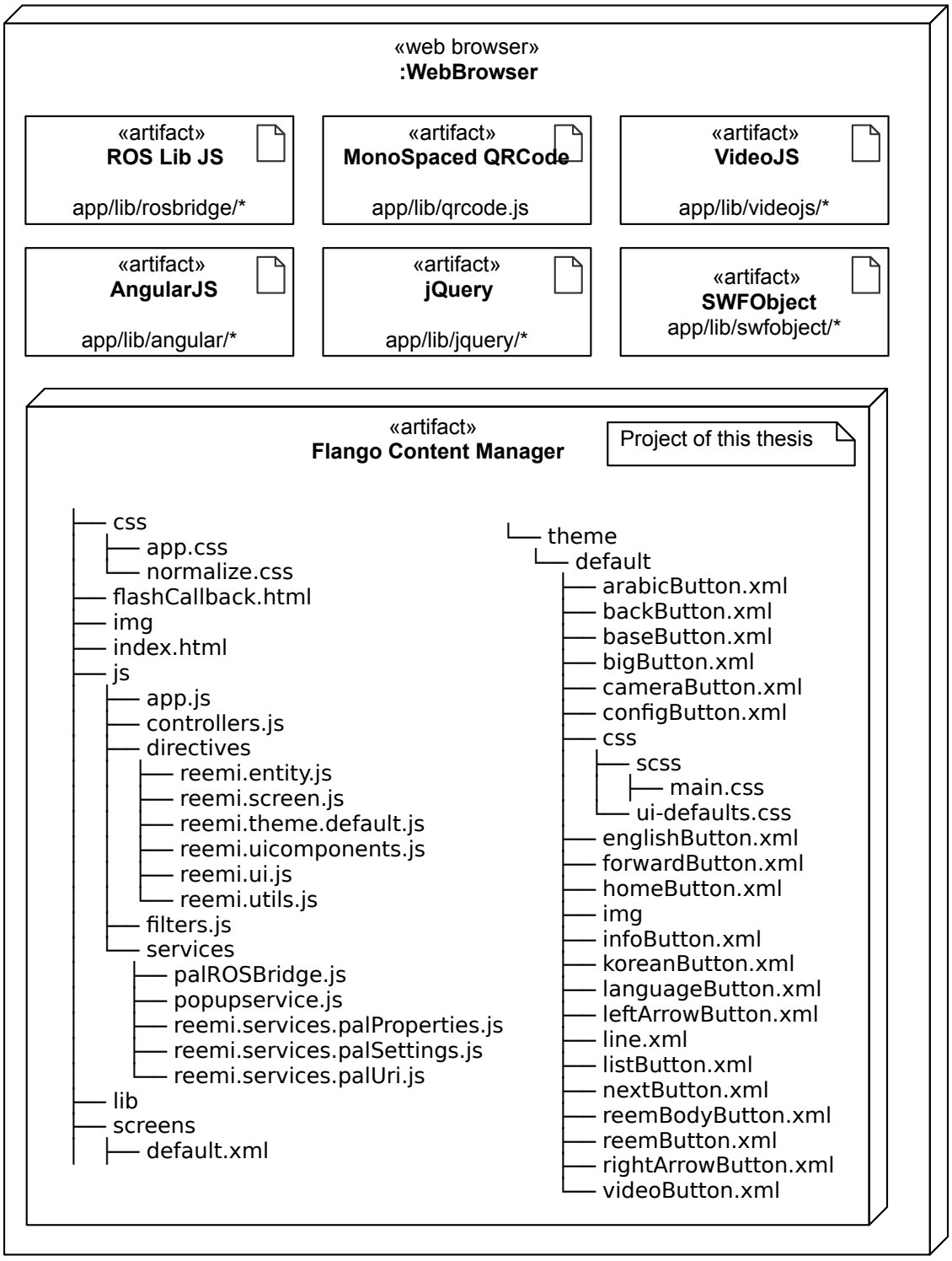

Figure 6-49: Deployment Diagram: Browser

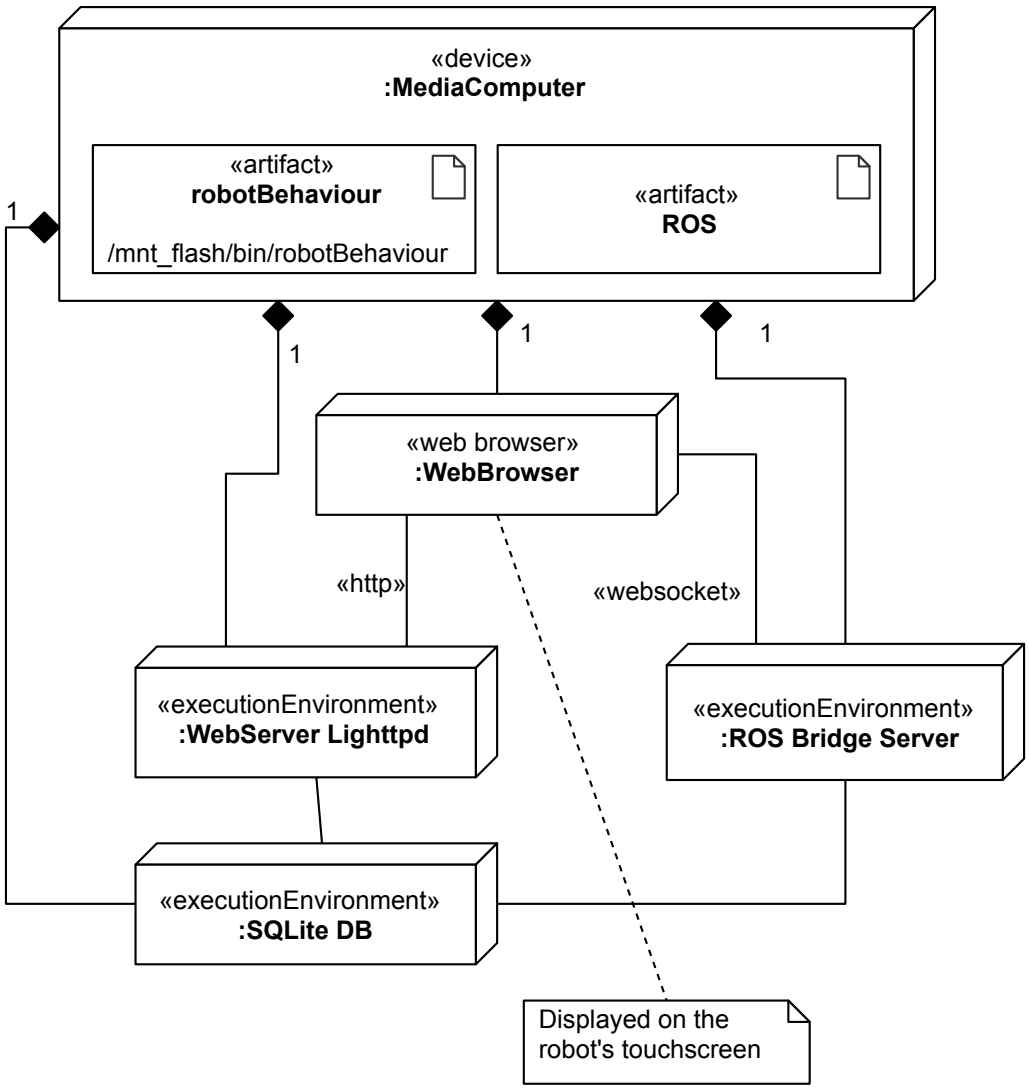

Figure 6-50: Deployment Diagram: Media computer

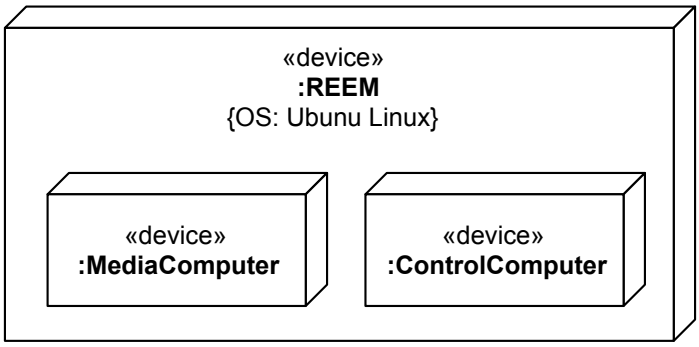

Figure 6-51: Deployment Diagram: Reem

Chapter 7

Implementation

This chapter describes the environment to develop and run Flango Content Manager. It illustrates the implementation of key features with code examples.

7.1 Development Environment Set-Up

Flango Content Manager belongs to a bigger project, Flango and the robot's system. The development environment is adapted to this situation to ensure a correct integration. Moreover, the project uses tools to run unit tests, generate code, etc.

Integration

The files of Flango Content Manager live in the server as part of the Flango Back-End. The backend is developed with Django and requires a set of Python libraries that might not be available in the system. To overcome this situation, Flango is developed in a virtual environment and files are made available to clients with the built-in web server in Django[1], although Basestation uses Apache2[2], and the robot Lighttpd[3]. All the code is in a subversion[4]

[1] `http://www.djangoproject.com`
[2] `http://httpd.apache.org`
[3] `http://www.lighttpd.net`
[4] `http://subversion.tigris.org`

repository.

```
$ mkvirtualenv local-flango
$ workon local-flango
$ pip install --upgrade -r requirements.txt
```

Listing 7.1: Creation of virtual environments

Virtual environments have pre and postactivate scripts to alter the default behaviour and can have versions of Python and other packages different than the system's. Flango needs to modify environment variables have `local-flango` in the local copy of the repository, use a database, set aliases, etc.

After the virtual environment is ready, it is easier to switch development branches (e.g. working on two features at the same time, but using different versions or different databases to avoid conflicts and guarantee that they both have a safe starting point). Django is a web framework for Python and has a built-in development web server that runs on port 8000.

```
~/contentMgmtBranches/trunk/backend$ django runserver
Validating models...

modeltranslation: Registered 0 models for translation () [pid: 5286].
0 errors found
Django version 1.4.3, using settings 'flango.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

"GET /static/flangoh/flangoh/app/index.html HTTP/1.1" 200 2317
```

Listing 7.2: Django web server

The file structure of the whole project is shown in listing 7.3. Folders `build`, `packaging` and `conf` contain scripts that are used when building Debian packages that will deploy the software on the robot and basestation. `src` contains the Flango Back-End, which includes the Flango Front-End (and the Screens Editor). The project of this thesis, Flango Content Manager is in the `static` folder (listing 7.4), that contains files served without any server-side processing.

```
~/contentMgmtBranches/trunk/backend$ ls
build  conf  packaging  requirements.txt  scripts
src    static
```

Listing 7.3: Flango files layout

The browser can open any file in the `static` folder. Alternating between the new and the old version only requires to load `frontend/index.html` (Adobe Flash) or `flangoh/flangoh/app/index.html` (Angular).

```
~/contentMgmtBranches/trunk/backend/static$ ls
admin        flangoh    imports    vpnUsersStatus.xml
css          fonts      js         vpnUsers.xml
exports      frontend   lib
flango-gui   img        media
```

Listing 7.4: Flango static files layout

The development computer also has a working copy of the code for the robot and Basestation in order to run simulations and to test the new Content Manager without using robot time, a scarce resource.

There are 3 run levels (each with a matching executable file):

1. **htmlDialog**. A basic Qt dialog with a QtWebKit widget.
2. **guiServer**. A version with more features enabled, like the sound server, navigation, etc.
3. **stateMachine**. A full program with a testing state machine, Qt, sound server, etc.

An XML file (`robot/sources/etc/reemh3/robot.xml`) defines the URL to load. Listing 7.5 shows a part of the configuration file. In this case, the robot (or the simulation) loads the old version, an HTML file in `http://localhost/static/frontend/index.html` that only contains a full-screen Adobe Flash object, the old Flango Content Manager.

```
<controller type="Gui" implementation="" debug="1">
    <guiController>
    <theme>barcelona</theme>
    <timeout>45</timeout>
    <fullHtml>1</fullHtml><!-- full screen -->
    <views>
        ...
        <view
            task="welcome"
            args="http://localhost/static/frontend/index.html"
            title=""
            runMeetPeople="1"
            activateASR="root/pal/reem-functions" />
...
```

Listing 7.5: Robot configuration file

```
$ nohup roscore &
cd $PAL_ROBOT_DIR/local
$ output/bin/htmlDialog "http://localhost/..."
$ output/bin/guiServer --silent --noNavigation [--basestationFlash]
$ ../sources/bin/testStateMachine.sh --noNavigation
```

Listing 7.6: Execution of a robot simulation

7.6 shows how these binaries are run.

The first version of the new Flango Content Manager could run in any of these three levels. However, the final version runs in Google Chrome in the robot and there is no need to use this simulation. In this case, interoperability is tested with ROS topics in localhost and Google Chrome.

Tools

The development of this project uses a number of tools to automate tasks and control de quality of the code to build a robust product.

Integrated Development Environment (IDE) WebStorm[5] is an IDE based on IntelliJ IDEA designed to develop RIAs. It provides integration with major frameworks, supports files with multiple languages (e.g. HTML with embedded CSS or JavaScript), provides tools for refactoring and on-the-fly code analysis (e.g. with JSHint/JSLint)

Karma Runner and Jasmine Angular mixes very well with TDD. It provides a framework (Jasmine) to write unit tests and a tool to automate them: Karma. Tests can be executed in the background every time the code changes to make them fail early. They run in a new instance of a browser and can be integrated in the IDE to debug easily. Additionally, it can be integrated with continuous integration systems, like Jenkins (used for the rest of the software in the robot).

[5] `http://www.jetbrains.com/webstorm`

Browsers and debugging The first part of the project used a browser with **QtWebkit** both in karma and to launch the application. There was one installation of Qt 4.8 and one of Qt 5.0.x, as Qt 5 was still experimental in the robot and had better support for CSS3 and HTML5. After this first stage, it was required to use a full browser: **Chrome** with Batarang and **Firefox** with Firebug. **Batarang** is an extension for Chrome developed by the Angular Team to display the scope of elements, dependencies, etc. **Firebug** is a widely-used extension for Firefox to debug web applications, modify HTML, CSS or JS on the fly, set breakpoints, or even profiling.

SASS SASS is a scripting language that is interpreted into CSS. It has variables, mixins, inheritance, logical nesting blocks, arithmetic operations and functions among other features, and the code can be separated in several files that are compiled into a single CSS. It has two syntaxes: the original, and SCSS, the newer one used in this project. Listing 7.7 (L19) shows logical nesting and inheritance.

```
1  .fl-button {
2      @include size(100px,100px);
3      ...
4      &.active, &:active {
5         color: #555;
6      }
7
8      // Disabled state
9      &.disabled, &[disabled] {
10         ...
11     }
12 ...
13 }
14
15 .fl-base-button {
16     ...
17     $width: 90px;
18     $height: 90px;
19     @extend .fl-button;
20     @include size($height,$width);
21     ...
22 }
```

Listing 7.7: SASS code example

Connection with robots Robots are essentially two GNU/Linux computers with Secure Shell (ssh) enabled. Reem-H3 has one in the torso (`media`) where Flango is installed, and one in the lower half (`control`). Listing 7.8 illustrates the connection with the media computer.

```
$ ssh pal@reemh3-1m
Welcome to Ubuntu 12.04.1 LTS
 (GNU/Linux 3.2.0-29-generic-pae i686)
...

pal@reemh3-1m:~$ ls /mnt_flash/srv/contents
admin        fonts      media
css          frontend   vpnUsersStatus.xml
exports      img        vpnUsers.xml
flango.db    imports
flango-gui   js
flangoh      lib
```

Listing 7.8: Connection with robots

7.2 Directives and Modules

Modules encapsulate directives. Namespacing in JavaScript can be misleading (e.g. services should have unique names application-wide, even if they are in separate modules and a client only uses one of them). To enforce isolation of components, this project puts all modules in separate closures 7.9.

```
(function () {
   "use strict";
   /* Directives */
   var uicomponentsModule = angular.module('reemi.uicomponents');

   uicomponentsModule.directive('name1', function(...) {...});
   uicomponentsModule.directive('name2', function(...) {...});s
}());
```

Listing 7.9: Closures to isolate modules

Directives are the core of Flango Content Manager. A simple directive that only does background logic (reading the `height` tag) is shown in listing 7.10. It is restricted to elements

(i.e. `<fl:height>100</fl:height>` triggers it but `<fl:ui fl:height="100">` does not)

`palProperties` is a service **injected** in directive.

```
uiModule.directive('flHeight', function (palProperties) {
return {
       restrict: 'E',
       require: '^?flUi',
       link: function (scope, element, attrs, uiCtrl) {
          /* part of the behaviour */
       }
   };
});
```

Listing 7.10: flHeight Directive

7.3 Services

Angular Services can be declared with the `service()` constructor, with a `factory()` or, for complex scenarios, with a `provider()`.

```
   palPropertiesModule.service('palProperties', function (palSettings) {});
   palSettingsModule.factory('palSettings', function ($http, $q...) {});
```

Listing 7.11: Examples of service definition

```
var m = angular.module('palSettingsModule', []);

m.factory('palSettings', function () {
    var getCurrentLanguage = function() { return config.currentLang; };

    return {
        'getCurrentLanguage' : getCurrentLanguage
    }
});
```

Listing 7.12: Example of Factory and Reveal Module Pattern

`service()` (listing 7.13) returns an object, whereas `factory()` (listings 7.12 and 7.14) first creates it and then returns the instance.

```
palPropertiesModule.service('palProperties', function (...) {
    this.toCSS = function (scope, p) { ... };
    this.getProperty = function (scope, property, lang) { ... };
});
```

Listing 7.13: Examples of service definition (service())

```
palSettingsModule.factory('palSettings', function (...) {
    var config, qs;
    var deferred = $q.defer();

    config = {generic: {}, app: {}, structure: {}, routes: {}};

    var getConfig = function () {...};
    var getGenericConfig = function () {...};
    ...

    return {
       'init': init,
       ...
    };
});
```

Listing 7.14: Examples of service definition (factory())

7.4 Controllers

Controllers are key parts of any CRUD application. However, because Flango Content Manager does not know about the specific domain of the content application before running it, the only controllers that exist are those that receive events from the rendered GUI, e.g. `ActionCtrl` (listing 7.15). **Command Pattern** This controller implements the **command pattern** and uses JavaScript's methods `call` and `apply` to call a method by its name and passing in arguments [16].

```
controllersModule.controller('ActionCtrl', function ($window,$scope...) {
    var performExternalCall = function(p) { ... };
    var goto = function (p) {...};
    var setLanguage = function(l) {...};
    var openPopUp = function (args) {...}

    ...
    /**
     * @param l list of functions and their arguments to run
     */
    $scope.executeActions = function (l) {
        angular.forEach(l, function (args, fun) {
            // command pattern @see Osmani:2012
            $scope.execute.apply( $scope, [].slice.call(arguments, 0));
        });
```

```
    };

    $scope.execute = function (args, name) {
        // command pattern @see Osmani:2012
        if ($scope[name]) {
            $scope[name].apply( $scope, [].concat(arguments[0]));
        }
    }

    // and expose internal functions too:
    $scope.performExternalCall = performExternalCall;
    $scope.setLanguage = setLanguage;
    $scope.goto = goto;
    $scope.openPopUp = openPopUp;
});
```

Listing 7.15: ActionCtrl implementation

7.5 Dependency Injection in JavaScript

An instance of type `A` can create instances of type `B` (or use global variables). Both options couple classes `A` and `B` (e.g. hard-coded dependencies in 7.16).

```
/* Example: use of the new operator and hard-coded dependencies */
function Person(name, age, sex, company) {
    this.company = new Company(company);
    this.name = name;
    this.age = age;
    this.sex = sex;
}

function Company(c) {
    this.cname = c;
}

var p1 = new Person("Alice Goodfellow", 25, "m", "NSA");
var p2 = new Person("Bob Chapman", 2, "m", "ESA");
```

Listing 7.16: Hard-coded dependencies

```
function MasterGreater(greeter) {
    this.greeter = greeter;
}

MasterGreater.prototype.hello = function (name) {
    return this.greeter.greet(name);
```

```
}

var Kitt = {
    greet: function (name) {
        return "Hello " + name + ", you may call me KITT.";
    }
};

var Hal9000 = {
    greet: function (name) {
        return "Hello " + name + ", you're looking well today.";
    }
};

var sc1 = new MasterGreater(Kitt);
var sc2 = new MasterGreater(Hal9000);
```

Listing 7.17: Passing on the reference (without service locator)

However, in listing 7.17 `MasterGreater` is not concerned with locating the `greeter` dependency, it is simply handed the `greeter` at runtime. This is desirable, but it puts the responsibility of getting hold of the dependency on the code that constructs `MasterGreater`.

Angular applies the *Dependency inversion* principle (SOLID) intensively and has a service locator to find services and other dependencies. Injectable components are created with a factory and are made available using a global injector. They can be injected just by using their name.

A particular implementation of this principle is in the wrapper for roslibjs: the service `palROSBridge` in the application can be injected and its operations accept a callback function as a parameter enabling a decoupled design.

```
var c = angular.module('controllers', ['palSettingsModule']);

c.controller('ExampleCtrl', function (palSettings, $scope) {
    $scope.getCurrentLanguage = function () {
        return palSettings.getCurrentLanguage();
    }
});
```

Listing 7.18: Injecting services in a controller

They can be obtained manually with the `$injector`

```
inject(function ($injector) {
```

```
2     $httpBackend = $injector.get('$httpBackend');
3 });
```

Listing 7.19: Manual injection (e.g. in a test)

Another example of inversion of control is implemented in section 7.8 on page 129.

A note about optimisation To improve JavaScript performance and reduce network usage, the code can be minified before deployment. This process reduces names of symbols, removes spaces, etc. Dependency injection depends on the name of components. Angular provides a protection for minification: annotations.

```
1 moduleA.run(['palSettings', '$location', '$rootScope', 'palROSBridge',
2     function (palSettings, $location, $rootScope, palROSBridge) {...}
3 ]);
```

Listing 7.20: Annotations to protect injection

Components can be injected as shown in listing 7.20. The names are values of an array which, obviously, are not altered.

7.6 Promises and Deferred Objects

Remote calls in JavaScript are executed asynchronously. The program in listing 7.21 generates the output shown in listing 7.22.

```
1  var result;
2  var asyncF = function () {
3      $http.get('/api/fruit/all').success( response ) {
4          result = response.data;
5          console.log('alert 1', result);
6      };
7
8      console.log('alert 2', result);
9  }
10
11 asyncF();
```

Listing 7.21: JavaScript Asynchronous calls

```
1 alert 2, undefined
```

```
alert 1, { object }
```

Listing 7.22: JavaScript Asynchronous calls (output)

Asynchronous calls are normally handled with callbacks following the Continuation-passing Style (CPS): control is passed explicitly in the form of continuation.

```
var printer = function (p1) {
    // print p1
};

a(printer);
```

Listing 7.23: JavaScript: *printer* will be called when *a* returns

```
/* nested and concurrent calls */

var a = function (callbackFn) {
    var rb, rc;

    $http.get('/api/fruit/all').success(response) {
        b(response.data.fruit[3]);
        c(response.data.fruit[3]);

        var b = function (fruitId) {
            $http.get('/api/fruit/details/' + fruitId)
            .success (response) {.
                rb = response.data;
                next();
            }
            .error (response) {
                console.log("Error!");
            }
        };
        /* same for c() */

        /* return control */
        var next() {
            if (rb && rc) callbackFn([rb, rc]);
        }
    }
    .error (response) {
        console.log('error here');
    }
};
```

Listing 7.24: JavaScript Callbacks

Listing 7.24 (called as in listing 7.23) shows a first approach[6] to callbacks. `$http.get()` takes two parameters, the `success` and the `error` functions. They are executed after `$http.get` has computed the return value, which is passed to the callbacks. However, when there are nested or concurrent asynchronous calls, like the case of fetching the configuration from the Flango Back-End, callbacks become hard to maintain and readability of the code is dramatically affected.

```
1  /* An implementation separating callbacks */
2
3  var a = function (callbackFn) {
4      var rb, rc;
5
6      $http.get('/api/fruit/all').then(aCallback, aErrCallback);
7
8      var aCallback = function (response) {
9          b(response.data.fruit[3]);
10         c(response.data.fruit[3]);
11     };
12
13     var aErrCallback = function (response) { console.log("Error!"); };
14
15     var b = function (fruitId) {
16         $http.get('/api/fruit/details/' + fruitId)
17             .then(bCallback, bErrCallback);
18     };
19
20     var bCallback = function (response) {
21         rb = response.data;
22         next();
23     };
24
25     var bErrCallback = function (response) { console.log("Error!"); };
26
27     /* same for c() */
28
29     /* return control */
30     var next() {
31         if (rb && rc) callbackFn([rb, rc]);
32     }
33 };
```

Listing 7.25: JavaScript Callbacks (separate declaration)

[6]This is a simplification that might not cover all cases

A better approach consists on extracting the body of the callback functions and using only pointers to these functions (listing 7.25). This still does not resolve the problem of concurrency: the result of `a()` is not complete until `b()` and `c()` have completed (e.g. if the returned value of each function had to be aggregated into a `var result = {a: [], b: []}`.

```
/*
  initialisation the palSettings service with promises
  success scenario only
*/

// 1
var getConfig = function () {
    getGenericConfig();
    return deferred.promise;
};

// 2
var getGenericConfig = function () {
    $http.get('/static/frontend/config.json').then(
        genericConfigCallback,
        genericConfigErrorCallback
    );
};

var genericConfigCallback = function(response) {
    config.generic = response.data;
    // overwrite parameters with query string
    setGenericInitialParams();
    getApplicationConfig(config.generic.app_id);
};

// 3
var getApplicationConfig = function (id, api_endpoint) {
    $http.get(api_endpoint + '/app/'+ id + '/config.json')
    .then(
        applicationConfigCallback,
        applicationConfigErrorCallback
    );
};

var applicationConfigCallback = function (response) {
    config.app = response.data;
    // overwrite parameters with query string
    setApplicationInitialParams();
    getStructure(config.app.id, config.generic.api_endpoint);
};

```

```
43 // 5
44 var getStructure = function(id, api){
45     $http.get(api + '/app/' + id + '/structure.json')
46     .then(function (response) {
47         config.structure = response.data;
48         deferred.resolve(); // the promise is resolved! \o/
49     });
50 };
51
52
53 /* public function init() */
54 init: function(p) {
55     qs = p;
56     return getConfig();
57 }
```

Listing 7.26: Angular Promises

Listing 7.26 shows a real example from this project: a successful retrieval of the configuration parts for the application. Angular's `$q` service is an implementation of promises/deferred based on Kris Kowal's Q[7]. A promise is *resolved* only after all data has been fetched correctly from the Flango Back-End. Promises can also be *rejected* if necessary. The code shows how control is passed:

$$init \rightarrow getConfig \rightarrow getGenericConfig \rightarrow getApplicationConfig \rightarrow getStructure$$

The latest two operations could run concurrently but if `getApplicationConfig` failed, `getStructure` would have to be called again with `appId = "default"`

This promise is used to block the loading of the content applications until the palSettings service is correctly initialised. Settings include security parameters like `perform_callbacks` or `security_level`, routes, URI of the home screen, default language... that must not be undefined to guarantee a consistent boot strap of the application.

A note about security Web or distributed systems normally have two levels of security: one client side, one server side. Validation client-side simply helps filter or reduce the amount of remote calls (e.g. if the data of a form is not well-formatted, it should not be sent to

[7] `https://github.com/kriskowal/q`

the server because the client node already knows that it will fail). Because JavaScript code client-side can be easily altered, the server side must not trust any input. However, in this project safe input is assumed even server side because the code runs in a controlled embedded system and the UI blocks access to the code. Thus, if users modify the code client-side, changes do not affect any other user in the system either directly (e.g. a user gaining admin privileges) or indirectly (a user retrieving sensitive data). Only engineers in the company have the ability to alter client-side code. For example: an application runs with `security_level = 1` (user) because it should not be allowed to move the arms or make Reem say sentences aloud. If client-side code is altered and `security_level` is set to `2` (admin), it will send the command to Qt (or publish it to a ROS Topic, depending on the version of this project) and arms will move. robotBehaviour assumes safe input from Flango Content Manager because the code can not be altered except by engineers in the company.

7.7 Transformation to HTML

The `flUi` directive makes decoupling and reusing directives possible (listing 7.27). It removes the type attribute (`fl:type="base-button"` and creates an attribute with that value (`base-button`), which can trigger directives (`flBaseButton`, listing 7.28). Dependencies are **injected** by using their name in the parameters of the function. The `controller` can be called from inner directives (e.g. flHeight in listing 7.32, the `compile` function manipulates the DOM before it is replaced with anything and the `link` function reads in-line attributes (`flX`, `flY`...) in the last place, as the link function of inner elements is executed first. Finally, it adds a class to a vector that is used to set CSS style to the HTML element.

```
1 m.directive('flUi', function ($compile, palProperties, palSettings...) {
2 return {
3     restrict: 'E',
4     replace: false,
5     priority: 10,
6     scope: true,
7
8     controller: function ($scope, $element) {
9         var classes;
```

```
10          palProperties.initProperties($scope);
11
12          classes = ... // init to {style: [], ui: []}
13
14          this.addClass = function (type, value) {
15              ...
16          };
17
18          /* for inner properties */
19          this.addProperty = function (property, value, lang) {
20              palProperties.setProperty($scope, property, value, lang);
21          };
22      },
23
24      compile: function ($tElement, $tAttrs) {
25          var el = $tElement[0];
26          if (el.getAttribute('fl:type')) {
27              el.removeAttribute('fl:type');
28              //key to trigger new directives. adds the new syntax!
29              el.setAttribute('fl:' + $tAttrs.flType, '');
30
31              var postLink = function (scope, element, attrs) {
32                  /*
33                      inner properties are read in the corresponding directive
34                      and have higher priority than inline properties.
35
36                      inline properties are read here
37                  */
38
39                  if (attrs.flX && !palProperties.getProperty(scope, 'x')) {
40                      palProperties.setProperty(scope, 'x', attrs.flX);
41                  }
42
43                  /* accumulates the CSS classes */
44                  scope.classes.ui.push('fl-' + attrs.flType);
45                  $compile(element)(scope);
46              };
47
48              return postLink;
49          }
50      }
51  };
52  });
```

Listing 7.27: Summary of the flUi directive

After compiling, Angular finds a new directive (e.g. `<fl:ui fl:base-button ...>` (`fl:ui` only runs functions if there is a `type` attribute, which was removed, and `flBaseButton` does

have code to run). flBaseButton (listing 7.28) is a directive in the themes module. It only links to a file that contains the template (listing 7.29). It is merely a `button` (a basic UI Component) with certain style set with the CSS class `fl-base-button`.

```
1 themeDefaultModule.directive('flBaseButton', function () {
2     return {
3         templateUrl: 'theme/default/baseButton.xml'
4     };
5 });
```

Listing 7.28: flBaseButton directive

```
1 <fl:ui fl:id="base-button" fl:type="button"></fl:ui>
```

Listing 7.29: flBasebutton directive template

After this template is compiled, Angular finds another directive, `flButton` (listing 7.30), which has the logic to create an HTML node for the actual button displayed on the screen.

```
1  cm.directive('flButton', function ($compile, palProperties...) {
2  return {
3      template: '<a ng-controller="ActionCtrl"
4          ng-click="executeActions(clickchain)"
5          ng-href="{{ href }}" ng-class="lclass"
6          target="{{ target }}" style="{{ style }}">
7             <div class="fl-overlay" style="{{ overlaystyle }}"></div>
8             <div class="caption-container">
9                 <div class="fl-caption" ng-bind-html="caption"></div>
10            </div>
11         </a>',
12
13     link: function (scope, element, attrs) {
14         scope.href = '';
15         scope.target = '_self';
16
17         var e = angular.element('<div />');
18         var cc = palProperties.getProperty(scope, 'caption');
19
20         scope.caption = e.html(cc).text();
21         scope.style = palProperties.toCSS(scope);
22         scope.overlaystyle = palProperties.toCSS(scope, 'overlay');
23         scope.clickchain = palProperties.getProperty(scope, 'clickchain');
24         scope.lclass = scope.classes.style.concat(scope.classes.ui[0]);
25
26         // highlight button for current URL
27         if (scope.clickchain ...) {
28             scope.lclass.unshift('highlight');
29         }
```

```
30      }
31  };
32  });
```

Listing 7.30: flButton directive

```
1  <a style="width:100px;height:100px;left:711px;top:394px;position:absolute;"
2    ng-class="lclass" target="_self" ng-href=""
3    ng-click="executeActions(clickchain)"
4    ng-controller="ActionCtrl" class="ng-scope fl-base-button">
5     <div style="width:100px;height:100px;left:711px;top:394px;
6                 position:absolute;display:none;"
7          class="fl-overlay"></div>
8     <div class="caption-container">
9         <div ng-bind-html="caption" class="fl-caption ng-binding"></div>
10    </div>
11 </a>
```

Listing 7.31: flBasebutton result

At the end of the day, a button is displayed in the browser (listing 7.31) using a link and `div`s with the CSS class `fl-base-button`. This style "includes" the style of a basic `button`. Exclusive properties (like the position or size) are transformed to CSS and added as in-line style in the element.

The result is a button like the one in the old version, one of the goals of the project.

7.8 Integration with ROS

Flango Content Manager is built as a component in ROS that can interoperate with the rest of the system using ROS topics. Listing 7.32 and 7.33 show the creation of a message for a ROS Topic in the software of the robot.

```
1  $ roscd pal_msgs
2  ~/svn/stacks/pal_msgs$ roscreate-pkg pal_rb_flango_msgs
3  Created package directory pal_rb_flango_msgs
4  Created package file pal_rb_flango_msgs/Makefile
5  Created package file pal_rb_flango_msgs/manifest.xml
6  Created package file pal_rb_flango_msgs/CMakeLists.txt
7  Created package file pal_rb_flango_msgs/mainpage.dox
8
9  Please edit pal_rb_flango_msgs/manifest.xml and mainpage.dox
10 to finish creating your package
```

```

~/svn/stacks/pal_msgs$ mkdir pal_rb_flango_msgs/msg

~/svn/stacks/pal_msgs$ vim pal_rb_flango_msgs/msg/rbFlango.msg
# Edit CMake

~/svn/stacks/pal_msgs/pal_rb_flango_msgs$ diff \
  CMakeLists.txt.orig CMakeLists.txt
20c20
< #rosbuild_genmsg()
---
> rosbuild_genmsg()

~/svn/stacks/pal_msgs/pal_rb_flango_msgs$ ls
CMakeLists.txt CMakeLists.txt.orig mainpage.dox Makefile
manifest.xml msg

~/svn/stacks/pal_msgs/pal_rb_flango_msgs$ rosmake
[ rosmake ] rosmake starting...
[ rosmake ] No package specified. Building ['pal_rb_flango_msgs']
[ rosmake ] Packages requested are: ['pal_rb_flango_msgs']
[ rosmake ] Logging to directory ~/.ros/rosmake/rosmake_output-2013...
[ rosmake ] Expanded args ['pal_rb_flango_msgs'] to:
['pal_rb_flango_msgs']
[rosmake-0] Starting >>> pal_rb_flango_msgs [ make ]
[rosmake-0] Finished <<< pal_rb_flango_msgs [PASS] [ 6.49 seconds ]
[ rosmake ] Results:
[ rosmake ] Built 1 packages with 0 failures.
[ rosmake ] Summary output to directory
[ rosmake ] ~/.ros/rosmake/rosmake_output-20131125-202327
~/svn/stacks/pal_msgs/pal_rb_flango_msgs$
```

Listing 7.32: Creation of a ROS Message

```
# message used by rb_flango
string name
# Expected contents:
# goTo
# setLanguage
string arg
```

Listing 7.33: Message type from robotBehaviour to Flango CM (rbFlango.msg)

After compiling the new message becomes available to the system. ROS Bridge provides a JSON interface to ROS. Listing 7.34 shows it running on port 9090

```
~$ nohup roscore &

~$ rosrun rosbridge_server rosbridge.py
```

```
4 [INFO] [WallTime: 1385407762] Rosbridge server started on port 9090
5 [INFO] [WallTime: 1385408718] Client connected. 1 clients total.
6 [INFO] [WallTime: 1385408718] [Client 0] Subscribed to /web_gui_events
```

Listing 7.34: ROS Bridge running

Flango Content Manager uses two topics: one for listening and one for publishing with different types of messages for convenience in the integration with robotBehaviour. It is feasible with only one topic. Topics are created the first time one publishes to them. The example of listing 7.35 shows how a message is published.

```
1 $ rostopic pub /web_gui_event pal_rb_flango_msgs/rbFlango
2 "name: 'setLanguage' arg: 'de'"
3 publishing and latching message. Press ctrl-C to terminate
```

Listing 7.35: Publishing to a topic

```
1 ~$ rostopic echo /web_gui_event
2 name: setLanguage
3 arg: de
4 ---
```

Listing 7.36: Listening to a topic

When Flango Content Manager loads in the browser, a connection is established with the ROS Bridge server using web sockets.

```
1 subscribing /web_gui_event pal_rb_flango_msgs/rbFlango
2 controller got web_gui_event Object { name="setLanguage", arg="en"}
```

Listing 7.37: Listening to a topic (Browser Console)

Flango Content Manager uses `roslibjs` to connect with ROS Bridge. This library uses its own events system and needs to be wrapped in a service to work in the framework.

Essentially, roslibjs has 3 methods: `connect(host, port)`, `subscribe(name, messageType, callback)` (and `unsubscribe(name, callback)`) and `publish(name, message)`.

```
1 ros = new ROSLIB.Ros({
2    url : 'ws://hostname:port'
3 });
```

Listing 7.38: ROSLibJS connect method

Clients can subscribe to topics providing the name of the topic and the name of the type. `ROSLIB.Topic` returns a listener that can wait for events.

```
var listener = new ROSLIB.Topic({
    ros : ros,
    name : '/topic_name',
    messageType : 'std_msgs/String'
});
listener.subscribe(function(message) { /* callback */ } );
```

Listing 7.39: ROSLibJS subscribe method

Inversion of control `palROSBridge` service interfaces to `ROSLib`. Services can be injected and add logic specific for Flango Content Manager. To make the subscribe callback available to the client instance, one of the parameters of `palROSLib::subscribe()` is a function:

```
var listeners = {};
var subscribe = function (name, messageType, callback) {
    listeners[name] = new ROSLIB.Topic({...});
    listeners[name].subscribe(function(message) {
        callback(message);
    });
};
```

Listing 7.40: palROSLib subscribe method

Reveal Pattern This pattern allows for a clearer way of writing JavaScript modules. Private methods are in a private closure and only a few are exposed in the returned object as a public API with pointers to the internal operation.

```
palROSBridgeModule.factory('palROSBridge', function () {
    /* init logic*/
    var ros, listeners = {};
    var connect = function(host, port) {...};
    ...
    // ROSLIB.Topic.subscribe creates a list of subscribers.
    // this is just delegating the task
    var subscribe = function (name, messageType, callback) {...};
    var unsubscribe = function (name, callback) {...};
    // Reveal methods as a public API
    var service = {
        'connect': connect,
        'subscribe': subscribe,
        'unsubscribe': unsubscribe,
        ...
   };
   return service;
});
```

Listing 7.41: Reveal Pattern in palROSBridge

Chapter 8

Testing

This chapter contains a description of how TDD is applied to this project.

Considering testing as equal in importance to application writing makes the code to be more robust, lowers the cost of maintenance and helps to have a better internal design.

8.1 Tools and Test Types

Angular comes with tools to automate testing:

Jasmine [1] A behaviour-driven development framework for testing JavaScript code. It provides a simple and self-descriptive syntax to write test suites.

Angular Mocks there are stub objects ready to load in tests (e.g. `$httpBackend` mocks Hyper-text transfer protocol (HTTP) calls)

Karma [2] is a test runner to automate the execution of the test suites.

TDD is explained in section 2.3 on page 17.

E2E simulate user interaction by writing scenario tests. They describe the expected behaviour of the application given a state and an interaction. Each test has commands (e.g.

[1] `http://pivotal.github.io/jasmine/`
[2] `http://karma-runner.github.io/`

"click on button with $id = 3$") and expectations (e.g. "the background colour is now shiny red").

Unit tests are created before the code itself is written (i.e. they first specify the behaviour of the software and then test the implementation). Each test focuses on an individual and small part of the code: given an initial situation, the unit testing framework runs the operation being tested and finally asserts the result.

This project only uses unit testing to define the specification of the software and to test directives, services and controllers. It is not possible to conduct E2E testing because the UI is built in run-time. this testing strategy fits well with normal CRUD applications, when developers know about the use cases of the software. The software of this thesis only deals with rendering content applications, which can be of any nature. If this thesis had focuses on reengineering one of the content applications, E2E would have been a useful tool.

8.2 Tests Definition

Tests are written with Jasmine and are grouped in test suites. These are `describe` blocks that contain tests in `it` blocks.

```
1  describe('Sample service method', function () {
2      // beforEach clauses
3      // afterEach clauses
4
5      it('should return 7', function () {
6          // 1. set the initial situation
7          // 2. run operations
8          // 3. asserts
9      });
10 });
```

Listing 8.1: Structure of a test suite

Listing 8.1 shows the structure of a test suite: it can perform logic before and after each test (e.g. instantiate an object or flush $http requests to guarantee that all tests start with the same system state and that do no test depends on another one). All tests have the same pattern:

1. Set the initial situation (e.g. create an HTML node)
2. Run an operation (e.g. compile the node)
3. Assert the results (e.g. expect a property in scope to have been set, or a counter to have been increased)

```
describe('Sample service method', function () {
    // beforEach clauses
    // afterEach clauses

    it('should return 7', function () {
        // 1. set the initial situation
        // 2. run operations
        // 3. asserts
    });
});
```

Listing 8.2: Structure of a test suite

Listing 8.2 demonstrates a real test. The `width` directive test suite (called simply "width") belongs to a test suite called "UI directive". It contains several`it` blocks that check one and only one result (e.g. the test for getting the value is called "should set the property to 40 for Catalan and 42 for French"). When karma runner executes all test suites, it creates a report with sentences like "UI directive: width directive should set the property to 42 for the default language". The initial situation is an XML node, the operation run is $compile() and the asserts are the `expect()` operations.

Dependency injection Even though the internal design of the application minimises the coupling of components and tries to maximise cohesion, there are dependencies between some classes. This is critical when it involves remote requests or using operations that are defined in another user class (as opposed to using well-tested code in the Angular core). It is then desirable to use (and re-use) mocks in tests. Dependency injection allows tests to mock dependencies: it loads a module with a mocked dependency of the same name that simply returns hard-coded values but offers the same interface. No modifications in the real component are needed.

```
var $compile, palProperties, palSettings;

beforeEach(module('reemi.ui'));
beforeEach(module('reemi.mocks.palSettings'));
```

```
...
beforeEach(inject(function(_$compile_, _palProperties_, _palSettings_) {
    $compile = _$compile_;
    ...
    palProperties = _palProperties_;
    palSettings = _palSettings_;
    ...
});
```

Listing 8.3: Dependency injection in tests

Listing 8.3 shows the implementation: modules are loaded to make them available to the test suite. In the `beforeEach` clause it stores a reference to the injected element to a local variable. Whenever a test calls an operation of the injected service, it will use the mock.

```
var m = angular.module('reemi.mocks.palProperties', []);

m.service('palProperties', function () {
       this.toCSS = function (params) {
      return "color:red;";
       }
});
```

Listing 8.4: Mocks

Mocks are components declared with the same name but in a different module (8.4). They are excluded from deployment to the robot.

Chapter B on page 151 contains a list of most of the relevant tests.

8.3 Test Automation

Fail early, fail often: karma runner is a process watching the project. It runs all tests on every change without disturbing the development and in less than two seconds. Karma is a *spectacular test runner for JavaScript* that can run tests in real browsers like Chrome or a program with QtWebKit and that can be integrated with WebStorm or launched from the console (listing 8.5)

```
$ ./scripts/test.sh
INFO [karma]: Karma server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
```

```
4 INFO [Chrome 28.0 (Linux)]: Connected on socket
5     id DxbVJNX0jIoe1CbaWf9V
6 Chrome 28.0 (Linux): Executed 143 of 143 SUCCESS
7 (0.843 secs / 0.432 secs)
```

Listing 8.5: Launching Karma Runner

It can be installed with Node Package Manager (npm) and configured with a file. The configuration for this thesis includes jQuery, mocks and configures the browser to suit the needs of the environment. In the first phase of the project, it used a browser based on QtWebKit instead of Chrome (8.6).

```
1 browsers: ['/opt/Qt5.0.2/5.0.2/gcc/
2   examples/webkitwidgets/browser/browser']
```

Listing 8.6: Configuration to use QtWebkit browser

The goal is to automate the testing in an environment as close to the real one as possible. With this automation, refactoring is easier and more robust, and regressions are avoided.

Chapter 9

Conclusions

9.1 Conclusions

Software products do not degrade with time or with external factors but they are continually adapted, corrected and extended. Changes in the project or dependency on third-party software can lead to critical situations (e.g. the vendor stops supporting the product) and make the software unmaintainable. Reengineering the system, altering the internal structure of the code or even substituting it without changing the external behaviour, becomes a requirement.

This thesis was eminently a practical work with a clear goal: reengineer Flango Content Manager with web technology.

The introduction provided an overview of the project including the context and the boundaries of the problem that this thesis was going to address.

The second chapter contained a general overview of topics that structure and support this thesis. It described the reengineering process proposed by Pressman [4], the evolution and present of web technology and the methodology of this project.

Chapters 3 to 8 described the development of this project from the perspective of software engineering. Despite the fact that the software was developed in small iterations, each

chapter presents a view of the system: plan, requirements, specification, internal design and deployment, details on the implementation, and testing.

Initially, the project was **planned** in terms of scope, schedule and budget. A risk analysis was conducted in order to prevent and mitigate situations that could make the project fail. During the development, deviations were tracked and understood and the plan was adapted accordingly. The overall number of hours was approximately the same (+14%) and the scope was slightly smaller.

Requirements were then gathered from exploring the old version and interviewing the previous developer. This approach saved time and provided a deep understanding of the system. Functional and non-functional requirements were listed along with constraints and off-the-shelf solutions.

After this, the **specification** of the system was made: system use cases, conceptual model and behaviour model, which combined test-driven development, where tests are executable specifications, and design by contract. Even though this was not a traditional approach to creating the specification of a system, it fitted very well in the development methodology. This part was kept as technology-agnostic as possible and emphasised the first three steps in a reengineering process: inventory analysis, documentation restructuring and reverse engineering. This *reverse engineering* step was crucial to understand the old system and became the most complex and extensive step.

The system was then **designed** internally with class and packages diagrams, sequence diagrams of relevant operations and deployment diagrams. The software was built with a MVC architecture in a SOA-like ecosystem (ROS) and used the following patterns: dependency injection, command, factory, revealing module and remote facade. Designing the software as a service would have been a better choice to make integration with ROS easier. In any case, the solution proposed is robust and interoperates correctly with the robot's system by using ROS Bridge and with Basestation, by using JSON. This part was done after choosing the technology and emphasised the last three steps in a reengineering process: code restructuring, data restructuring and forward engineering. Since the new application substituted the old one, only the *forward engineering* step was relevant, but minor changes in the Flango

Back-End and robotBehaviour were done in the *code/data restructuring* steps.

Finally, the software was **implemented** with Google Angular (JavaScript), HTML, SASS, ROSLibJS and other JavaScript libraries. There were minor changes in the back-end (Python) and in the screens editor. Amongst other improvements, using web technology made the new system more flexible (changeability of themes, addition of new components), easier to debug, and interoperable (access to ROS Topics and, therefore, to hardware).

Google Angular was used in an exotic fashion to fit in the constraints of the XML, which resulted in only using only some of the features of the framework (or even overriding a few of them) but also in reducing the workload and the amount of code to write. SASS enabled thinking in object oriented design even for style sheets, code reuse and efficiency in the browser.

Unit **tests** were used as **executable specifications** and helped design less coupled and more cohesive components.

It is clear that there are many ways to reengineer a system. Pressman's book provided a framework to divide the process in phases, set boundaries, discover new tasks (e.g. extending the Flango Back-End to obtain data in a suitable format) and structure this document.

All the goals of this project have been fulfilled:

1. The new software works, covers 80% of the most common features and can substitute the old Adobe Flash system safely: it is ready for production.
2. It has a strong testing strategy with TDD.
3. It is compatible with non-reengineered parts of the system (Flango Back-End and Flango Front-End).

Even though there are some limitations, I believe that this master's thesis has addressed the reengineering of the content manager and developed the new system with accuracy, creativity and consistency. It applied a systematic approach and used proven techniques like software patterns and widely-known architectures.

9.2 Personal Insights

This work is the artifact that puts an end to a two-year masters' course that has helped me strengthen my computer science and software engineering education. In these 4 semesters I have gained a diverse and strong background in my field, including topics like compilers or artificial intelligence, a project-driven approach to software design and databases, or even business and project management. Part of this Master was followed at Northeastern University (Boston, MA), where I learned about research and explored topics that did not belong to my field of specialisation, such as machine learning. I also learned about human-computer interaction and web technology, which helped me understand the context of this project.

Specifically, this thesis gave me the opportunity to research topics that did not belong to the syllabus of the courses I took. For example, I researched service-oriented architectures, integration of systems, reengineering of software or applying and implementing software patterns in a real environment. I worked with innovative technology that does not fit in classical software design models, like JavaScript, and I reviewed, extended and fixed code in Python or ActionScript developed by previous members of the team.

This project was developed for a company that specialises in humanoid robots. Most of the engineers work with either Python or C++, but not with web technology. Being a one-man team helped me find ways to overcome all kind of problems in order to make the project successful. Furthermore, this set up forced me to think about the trade-offs between time and the scope of projects, and to build the product incrementally.

In conclusion, this master's thesis consolidates not only my technical skills, but also an essential skill for a software engineer: problem-solving.

9.3 Future Work

Although I believe that this master's thesis provides a comprehensive and solid description of the system previously described and developed, it is important to note that there are limitations that could be the base of future work.

Project Scope This thesis presented only some parts of the Flango Content Manager. A number of features were left out of the scope due to time constraints and should therefore be added in the upcoming weeks. These include the implementation of Entities (incomplete at the time of writing this document) and the management of user sessions.

Content and container Reading inner and inherited properties (`<elem><width>30</width></elem>`) instead of reading them in-line (`<elem width="30" />`) becomes cumbersome when UI Components are in a container defined in the XML (`<fl:ui fl:type="container">...</elem>`). With a first strategy, containers were modified by inner attributes of UI Components. The current strategy works but it makes logic reuse and maintenance harder than necessary. In the future, a better design should be implemented. The most frequent use of containers is displaying entities. This code is not generated by the screens editor. Therefore, it could be changed to regular Angular loop directives.

SOA and integration There are at least two clients that use the software: the robot and a content application user. It was only in the last weeks of development that we considered developing the software as a service, like other components in ROS that use the messaging system. Due to time constraints, the software was not originally designed as a service but included a remote facade to attend requests from the robot. This was a simple approach to a simple need (less than 10 types of requests). Redesigning this remote facade should be considered to allow the reuse of components and make the logic of Flango Content Manager a service in the robot.

Theming The previous version only had one theme. The new version supports themes but the implementation is incomplete. SASS elements, such as variables, *mixins* and inheritance will allow quick development of the look and feel of content applications.

Appendices

Appendix A

UI Components

```
ButtonComponent.as
GroupButtonComponent.as
GroupComponent.as
ImageComponent.as
MenuComponent.as
QRComponent.as
ScreenComponent.as
ScrollComponent.as
ShowReelComponent.as
SubscreenComponent.as
SWFComponent.as
TextareaComponent.as
UIComponent.as
VideoComponent.as
```

Listing A.1: Basic UI Components

```
arabic_button
back_button
base_button
big_button
call_to_action_screen
camera_button
config_button
english_button
forward_button
home_button
info_button
korean_button
language_button
left_arrow_button
```

```
line
list_button
main_menu
next_button
reem_body_button
reem_button
right_arrow_button
smartphone_menu
synchronizing_screen
video_button
wide_button
```

Listing A.2: Theme UI Components

```
api_endpoint:String
app_id:String
app_name:String
current_node:String
debug_level:String
debug_mode:Boolean
default_api_endpoint:String
def_lang:String
def_theme:String
enable_stats_bkp:Boolean
enable_stats:Boolean
history:Array
history_current:int
history_limit:int
home_node:String
initial_node:String
lang_list:Object
lang:String
log:String
media_path:String
performance_monitor:Boolean
perform_callbacks_bkp:Boolean
perform_callbacks:Boolean
redirects:Array
report_stats:Boolean
robot_id:String
scale_factor:Number
screensaver_node:String
screens:Object
screen_transitions:Boolean
security_level:int
session_fadeout:int
session_timeout:int
show_cursor:Boolean
static_api_endpoint:String
```

```
static_media_path:String
static_themes_path:String
structure:XML
theme_list:Object
themes_path:String
theme:String
themes:XMLList
zone:String
```

Listing A.3: Application State (Settings class)

Appendix B

Unit Tests

Unit testing UI directive

1. inline type property: should have type defined
2. width
   (a) should have the class hide
   (b) should set the property to 42 for the default (language-independent)
   (c) should set the property to 40 for Catalan and 42 for French
   (d) should set the property to 40 for Catalan, 42 for French and 20 for default
   (e) should not set any property
   (f) (inline) should set the property to 42 for the default (language-independent)
3. height
   (a) should have the class hide
   (b) should set the property to 42 for the default (language-independent)
   (c) should set the property to 40 for Catalan and 42 for French
   (d) should set the property to 40 for Catalan, 42 for French and 20 for default
   (e) should not set the property
   (f) (inline) should set the property to 42 for the default language
4. x
   (a) should have the class hide
   (b) should set the property to 42 for the default language
   (c) should set the property to 40 for Catalan and 42 for French
   (d) should set the property to 40 for Catalan, 42 for French and 20 for default
   (e) should not set the property
   (f) (inline) should set the property to 42 for the default language
5. y
   (a) should have the class hide
   (b) should set the property to 42 for the default language

(c) should set the property to 40 for Catalan and 42 for French
(d) should set the property to 40 for Catalan, 42 for French and 20 for default
(e) should not set the property
(f) (inline) should set the property to 42 for the default language

6. src
   (a) should have the class hide
   (b) should set the property to "dummy string" for the default language
   (c) should set the property to 40 for Catalan and 42 for French
   (d) should set the property to 40 for Catalan, 42 for French and 20 for default
   (e) should not set the property
   (f) (inline) should set the property to 42 for the default language
7. style
   (a) should have the class hide
   (b) should set scope.classes.ui to "whatever" and scope.classes.style to "nice"
   (c) (inline) should set scope.classes.ui to "whatever" and scope.classes.style to "nice"
8. flCaption
   (a) should have the class hide
   (b) should write "dummy string" for default language (and nothing else)
   (c) should set English caption to "dummy string"
9. layover
   (a) should have the class hide
   (b) should add the overlay property empty
   (c) should add the overlay property with width, height and src
   (d) should add the overlay property with inline width, height and src
10. goto: should read the parameters
11. external call: should read the parameters
12. set language: should read the language
13. open popup: should read the code of the popup
14. background
    (a) should read the parameters
    (b) should read the inline parameters

Unit testing UI directive – ambiguity

1. (width) should use inner value (20) instead of inline value (10)
2. (height) should use inner value (20) instead of inline (10)
3. (x) should use inner value (20) instead of inline (10)
4. (y) should use inner value (20) instead of inline (10)
5. (src) should use inner value ("dummy string") instead of inline ("smart string")
6. (palStyle) should give inner value ("nice") higher priority than inline ("ugly")
7. (palStyle) should give inner values ("big" and "red") higher priority than inline ("ugly")

Unit testing UI Components directives

1. subscreen
   (a) should always be transformed to <fl:ui fl:subscreen>
   (b) should include a screen file (always uses filename screens/99.xml here) in the inner div
   (c) should add a ¡div>with at least the subscreen css class
2. img
   (a) should be transformed to <fl:ui fl:img>
   (b) should be replaced with <img>
   (c) should set the ng-src attribute in <img>
3. button
   (a) should be transformed to <fl:ui fl:button>
   (b) should be replaced with <a>
4. textarea
   (a) should be transformed to <fl:ui fl:textarea>
   (b) should be replaced with
5. video: should be transformed to <fl:ui video>

Unit testing UI subscreen with real palSettings

1. should include the file for route /root/pal/reem-video (98.xml) in the inner div

Unit testing: Testing modules and dependencies

1. Reemi Module:
   (a) should be registered
   (b) Dependencies:
      i. should have reemi.ui as a dependency
      ii. should have reemi.uicomponents as a dependency
      iii. should have reemi.utils as a dependency
      iv. should have reemi.theme.default as a dependency
      v. should have reemi.services.palSettings as a dependency
      vi. should have reemi.services.palProperties as a dependency
      vii. should have reemi.services.palUri as a dependency

palProperties Service

1. should return a sanitized number
2. should return a sanitized string
3. should have transformed a set of flango properties to CSS

4. should create a data structure to store properties
5. should set a property in the specified language
6. should set a language-independent property
7. should get a property in the current language, or fallback to the language-independent property

palSettings Service

1. generic configuration method
   (a) getShowCursor() should return true
   (b) getPerformCallbacks() should return false
   (c) getReportStats() should return true
   (d) getSecurityLevel() should return 2
   (e) getDebugLevel() should return "ALL"
   (f) getDebugMode() should return true
   (g) getScreenTransitions() should return false
   (h) getSessionTimeout() should return 999999999
   (i) getPerformanceMonitor() should return false
   (j) getApiEndpoint() should return "/flango-api-0.1"
   (k) getDefaultApiEndpoint() should return "/static/frontend/content"
   (l) getStaticApiEndpoint() should return "content"
   (m) getThemesPath() should return "/static/frontend/themes"
   (n) getStaticThemesPath() should return "themes"
   (o) getMediaPath() should return "/static/media"
   (p) getStaticMediaPath() should return "content/assets"
2. app configuration method
   (a) getAppId() should return 13
   (b) getAppName() should return Barcelona
   (c) getAppLanguages() should return array of languages
   (d) getAppDefaultLanguage() should return "en"
   (e) getAppInitialNode() should return "/root/main-menu"
   (f) getAppHomeNode() should return "/root/main-menu"
   (g) getAppThemes() should return array of themes
   (h) getAppDefaultTheme() should return the name of the default theme
3. routes
   (a) should set 24 routes
   (b) should define at least the routes declared in the response json file
   (c) should define a default route
4. other methods
   (a) getCurrentLanguage() should return the default language
   (b) setCurrentLanguage(ca) should set current language to ca
   (c) getScreenFile should find the correct file (using the api endpoint)
   (d) getScreenFile should find the correct file (with url params)
5. initialization

(a) should load the configuration and structure for appid 13 (exists)
(b) should load the default contents when app_id is wrong

palUri Service

1. should get the root when trying to make an external call without params
2. should form the url for an external call, with params
3. should open the external call url in a new tab
4. should not open the external call url in a new tab (permission to perform callbacks not granted)
5. should not open the external call url in a new tab (security level is lower than the one set in the configuration)
6. should open the external call url in a new tab (security level is greater or equal than the one set in the configuration)

Bibliography

[1] M. Anttonen, A. Salminen, T. Mikkonen, and A. Taivalsaari, "Transforming the web into a real application platform: new technologies, emerging trends and missing pieces," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011, pp. 800–807. [Online]. Available: http://doi.acm.org/10.1145/1982185.1982357

[2] Adobe, "Adobe roadmap for the flash runtimes," 2013, [Accessed 8-June-2013]. [Online]. Available: http://adobe.com/go/flashplayer_roadmap

[3] S. Jobs, "Thoughts on flash," Apple Inc., 2010, [Accessed 8-June-2013]. [Online]. Available: http://www.apple.com/hotnews/thoughts-on-flash

[4] R. Pressman, *Software Engineering: A Practitioner's Approach*, 6th ed. New York, NY, USA: McGraw-Hill, Inc., 2005.

[5] T. Berners-Lee and R. Cailliau, "Worldwideweb: Proposal for a hypertexts project," 1990, [Accessed 9-June-2013]. [Online]. Available: http://www.w3.org/Proposal.html

[6] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz, "Web browser as an application platform: the lively kernel experience," Mountain View, CA, USA, Tech. Rep., 2008.

[7] G. Kappel, B. Pröll, S. Reich, and W. Retschitzegger, *Web Engineering: The Discipline of Systematic Development of Web Applications*. Wiley, 2006.

[8] C. Reis and S. D. Gribble, "Isolating web programs in modern browser architectures," in *Proceedings of the 4th ACM European conference on Computer systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 219–232. [Online]. Available: http://doi.acm.org/10.1145/1519065.1519090

[9] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[10] H. Baumeister, "Combining formal specifications with test driven development," *Extreme Programming and Agile Methods-XP/Agile ...*, no. 2, 2004. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-27777-4_1

[11] D. Crockford, *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[12] ——, "Prototypal inheritance in javascripts," 2008, [Accessed 1-November-2013]. [Online]. Available: http://javascript.crockford.com/prototypal.html

[13] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[15] M. Fowler, *Patterns of enterprise application architecture*, 17th ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., July 2011.

[16] A. Osmani, *Learning JavaScript Design Patterns*. O'Reilly Media, Incorporated, 2012. [Online]. Available: http://addyosmani.com/resources/essentialjsdesignpatterns/book

[17] W. A. TaskForce, "Introduction to web components," W3C, 2013, [Accessed 10-August-2013]. [Online]. Available: http://www.w3.org/TR/2013/WD-components-intro-20130606/

[18] B. Liskov and L. Shrira, "Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems," *SIGPLAN Not.*, vol. 23, no. 7, pp. 260–267, Jun. 1988. [Online]. Available: http://doi.acm.org/10.1145/960116.54016

[19] S. Stefanov, *JavaScript Patterns*. O'Reilly Media, 2010.

[20] P. B. Darwin and P. Kozlowski, *AngularJS Web Application Development*, 1st ed. Packt Publishing, 2013.

[21] T. A. Team, "The angularjs guide," 2013, [Accessed 22-April-2013]. [Online]. Available: http://docs.angularjs.org/guide