

ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

---

Seconda Facoltà di Ingegneria  
Corso di Laurea Magistrale in Ingegneria Informatica

ENGINEERING CONCURRENT AND  
EVENT-DRIVEN WEB APPS: AN  
AGENT-ORIENTED APPROACH BASED ON  
THE SIMPAL LANGUAGE

*Tesi in:*

Programmazione Concorrente e Distribuita LM

*Presentata da:*

FRANCESCO FABBRI

*Relatore:*

Prof. ALESSANDRO RICCI

*Co-relatore:*

Dott. ANDREA SANTI

---

ANNO ACCADEMICO 2011–2012  
SESSIONE II



## Abstract

Web is constantly evolving, thanks to the 2.0 transition, *HTML5* new features and the coming of *cloud-computing*, the gap between Web and traditional desktop applications is tailing off. *Web-apps* are more and more widespread and bring several benefits compared to traditional ones. On the other hand reference technologies, JavaScript primarily, are not keeping pace, so a paradigm shift is taking place in Web programming, and so many new languages and technologies are coming out. First objective of this thesis is to survey the reference and state-of-art technologies for client-side Web programming focusing in particular on what concerns concurrency and asynchronous programming. Taking into account the problems that affect existing technologies, we finally design *simpAL-web*, an innovative approach to tackle *Web-apps* development, based on the Agent-oriented programming abstraction and the *simpAL* language.

**Keywords:** *Web 2.0, JavaScript, Dart, Asynchronous programming, Event-driven programming, Agent-oriented programming, simpAL*



## Sommario

Il Web è in continua evoluzione, grazie alla transizione verso il 2.0, alle nuove funzionalità introdotte con *HTML5* ed all'avvento del *cloud-computing*, il divario tra le applicazioni Web e quelle desktop tradizionali va assottigliandosi. Le *Web-apps* sono sempre più diffuse e presentano diversi vantaggi rispetto a quelle tradizionali. D'altra parte le tecnologie di riferimento, JavaScript in primis, non stanno tenendo il passo, motivo per cui la programmazione Web sta andando incontro ad un cambio di paradigma e nuovi linguaggi e tecnologie stanno spuntando sempre più numerosi.

Primo obiettivo di questa tesi è di passare al vaglio le tecnologie di riferimento ed allo stato dell'arte per quel che riguarda la programmazione Web client-side, porgendo particolare attenzione agli aspetti inerenti la concorrenza e la programmazione asincrona. Considerando i principali problemi di cui soffrono le attuali tecnologie passeremo infine alla progettazione di *simpAL-web*, un approccio innovativo con cui affrontare lo sviluppo di *Web-apps* basato sulla programmazione orientata agli Agenti e sul linguaggio *simpAL*.

**Parole chiave:** *Web 2.0, JavaScript, Dart, Asynchronous programming, Event-driven programming, Agent-oriented programming, simpAL*



*To my parents,  
who taught me what in life  
is really important.*





# Contents

<b>Abstract</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>4</b>
2.1 JavaScript . . . . .	4
2.1.1 JavaScript drawbacks . . . . .	5
2.1.2 The JavaScript frameworks ecosystem . . . . .	7
2.1.3 JavaScript as intermediate language . . . . .	10
2.2 Toward the next generation of structured Web languages . . . . .	13
2.2.1 Dart . . . . .	14
2.2.2 TypeScript . . . . .	16
2.2.3 What's the future <i>lingua franca</i> for the Web? . . . . .	18
2.3 Web asynchronous programming . . . . .	20
2.3.1 Asynchronous programming troubles . . . . .	23
2.3.2 Futures and Promises . . . . .	24
2.3.3 JavaScript jQuery Promises . . . . .	26
2.3.3.1 Deferred and Promise . . . . .	26
2.3.3.2 Promises and AJAX . . . . .	27
2.3.3.3 Promises progress . . . . .	28
2.3.3.4 Promise pipelining . . . . .	29
2.3.3.5 Combining Promises . . . . .	31
2.3.4 Dart Futures . . . . .	33
2.4 Concurrency in Web applications . . . . .	34
2.4.1 HTML5 Web Workers . . . . .	35

2.4.1.1	Dedicated Workers . . . . .	37
2.4.1.2	Shared Workers . . . . .	40
2.4.1.3	Web Worker’s event loop . . . . .	42
2.4.1.4	Mixing Web Workers and asynchronous programming . . . . .	44
2.4.2	Dart Isolates . . . . .	46
2.4.3	Other technologies . . . . .	49
2.5	A case study Web-app . . . . .	49
2.6	Open issues in reference technologies . . . . .	52
2.6.1	Asynchronous spaghetti . . . . .	52
2.6.2	Asynchronous programming and Inversion of Control . . . . .	55
2.6.3	Issues inherent the Actor model . . . . .	55
<b>3</b>	<b>The simpAL language</b>	<b>57</b>
3.1	Main concepts . . . . .	58
3.2	Programming agents . . . . .	60
3.2.1	Roles and tasks . . . . .	61
3.2.2	Agent scripts and plans . . . . .	62
3.2.2.1	Action rules: events, conditions, actions . . . . .	63
3.2.2.2	Tasks as first-class entities . . . . .	65
3.2.3	The agent control loop . . . . .	66
3.3	Programming artifact-based environments . . . . .	68
3.3.1	Usage interfaces . . . . .	69
3.3.2	Artifact templates . . . . .	69
3.4	Defining the organization . . . . .	71
3.5	simpAL benefits . . . . .	72
3.5.1	Asynchronous programming without Inversion of Control . . . . .	72
3.5.2	Integration of autonomous and reactive behaviours . . . . .	73
3.5.3	Concurrency . . . . .	73
3.5.4	Error checking at compile time . . . . .	74
<b>4</b>	<b>simpAL-web</b>	<b>75</b>
4.1	Requirements and assumptions . . . . .	76

4.2	Modeling simpAL Web applications . . . . .	77
4.3	Modeling the Web environment . . . . .	78
4.3.1	The Web Page artifact . . . . .	79
4.3.2	The Web Element artifact . . . . .	81
4.3.3	The Clock artifact . . . . .	83
4.3.4	Interaction with Web services . . . . .	84
4.4	Web programming in simpAL-web . . . . .	87
4.4.1	Dealing with asynchronous programming . . . . .	88
4.4.2	Dealing with concurrency . . . . .	91
4.4.3	Dealing with complex applicazions . . . . .	94
4.5	About simpAL-web implementation . . . . .	97
<b>5</b>	<b>Conclusions</b>	<b>100</b>
5.1	Future works . . . . .	102
	<b>List of Tables</b>	<b>105</b>
	<b>List of Figures</b>	<b>106</b>
	<b>List of Listings</b>	<b>107</b>
	<b>Bibliography</b>	<b>109</b>
<b>A</b>	<b>Chapter 2 sources</b>	<b>114</b>
A.1	HTML documents . . . . .	114
A.2	JavaScript sources of “What friends most like?” app . . . . .	116
<b>B</b>	<b>Chapter 4 sources</b>	<b>124</b>
B.1	WebPageArtifact implementation . . . . .	124
B.2	WebElemArtifact implementation . . . . .	125
B.3	simpAL sources of “Battle of the bands” app . . . . .	127
B.4	simpAL sources of “Stoppable counter” app . . . . .	130
B.5	simpAL sources of “What friends most like?” app . . . . .	131



# Chapter 1

## Introduction

Since it's introduction in early 1990s, the World Wide Web has swiftly grown, driven by ever greater amounts of online information, commerce, entertainment and social networking, and coming pervasively in our every day life. With the Web 2.0 transition, once static, HTML pages “come to life” thanks to JavaScript language and AJAX technology enriching the user interaction and becoming out-and-out Web applications. Well-known examples of popular Web applications are for instance Google Docs<sup>1</sup>, an office suite utterly Web-based which also allows the cooperation among many users, or Google Mail<sup>2</sup>, representative of all those Web-based mail clients. The deployment of these so-called Web apps is an ever more widespread trend and the Web is becoming the *de facto* deployment environment for new software systems mainly thanks to the following factors. First is that all Web languages and technologies, for instance HTML, CSS, ECMAScript, AJAX, JSON are open standards, mainly headed by the World Wide Web Consortium (W3C<sup>3</sup>). Because of this and given the ubiquity of Web browsers which, except for notable exceptions comply the standards, leads to the huge benefit for which Web apps are portable for-free virtually on every platform. Ubiquity is a key aspect especially in relation to the nowadays increasing spread of mobile devices such as smartphones and tablets, each one with its own hardware capabilities and

---

<sup>1</sup><http://docs.google.com>

<sup>2</sup><http://mail.google.com>

<sup>3</sup><http://www.w3.org>

operating systems. Considering in addition that besides the mobile platforms, thanks to the evolution of Web standards, Web applications are able to compete with traditional desktop-based ones, the Web development is definitely proposed as “Write once, run anywhere” target since it allows to develop applications regardless from any specific type of operating system, computer or device. A further strong point concerns maintainability since all the new features are implemented on the server and forthwith available to all the users. Thanks to these benefits, the trend in the software industries is definitely to move applications, including those desktop-based, to the World Wide Web, causing a fundamental change in the way people develop, deploy and use software, consider for instance the coming of cloud computing and Software as a Service (SaaS).

As stated before Web technologies are constantly evolving with the aim of bridging the gap between Web applications and native ones, whether they are mobile or desktop-based. For example the newer HTML version 5 standard brings in Web applications features like media playback, Web storage, 3d graphic and concurrency. On the other side, with the Web 2.0 transition and the coming of AJAX, it has been introduced the ability to update parts of the user interface without reloading the entire page each time when something changes. This made Web applications to behave much like desktop ones, and often these kinds of applications are denoted as Rich Internet Applications (RIAs)<sup>4</sup>. Anyhow to date the gap between Web applications and native ones persists and it’s measurable mainly in terms of usability, performance and ease of development. Especially for what concerns the latter aspect the technologies have not evolved enough yet, so Web applications development mainly suffers of problems such as the lack of structure, the poorness of software engineer methodologies and tooling. Furthermore, JavaScript, the standard language *de facto* for client-side Web programming has several drawbacks and, as we will see in the remainder, is particularly inadequate for in-the-large programming.

Objective of the first part of this thesis is therefore to analyze the reference

---

<sup>4</sup>Not to be confused with those platforms such as Adobe Flash, Microsoft Silverlight and Oracle JavaFX which are targeted to Rich Internet Applications development too, but run in proprietary virtual machines and are typically provided as Web browsers plugins.

technologies for client-side Web programming, primarily JavaScript, investigating for the major issues that afflict the development of Web applications and ultimately understanding how state of art technologies such as Dart, TypeScript and JavaScript frameworks tackle some of these issues. However as we will see some of these issues remain unresolved, in particular those related to asynchronous programming and concurrency. Taking in account these problems, in the second part of the thesis we propose a paradigm shift in Web programming toward the Agent-oriented programming (AOP) paradigm and the simpAL language in particular. Differently to other Agent-oriented languages introduced in the (Distributed) Artificial Intelligence context, simpAL is intended to investigate the Agent-oriented as general-purpose programming approach, extending the Object-oriented programming with a further abstraction layer to deal with concurrent and distributed systems design and development, based on the Human-inspired computing metaphor. For the purpose to integrate the simpAL language in Web applications we finally design the simpAL-web platform with which we test the effectiveness of simpAL in dealing with the issues of Web programming.

The reminder of this thesis is organized as follows: in Chapter 2 we firstly take a survey of the reference and state-of-art languages and technologies in the current Web scene. Then we focus on two key aspects that is asynchronous programming and concurrency considering how they are tackled by current languages and providing several examples. We conclude the chapter taking some considerations about open issues concerning the aspects described before. In Chapter 3 we introduce the Agent-oriented programming paradigm and we make an overview of the simpAL language which we have designated as the reference language for our investigation for Agent-oriented Web programming. In Chapter 4 we design the simpAL-web platform and we tackle the issues identified in the previous chapters with the simpAL language. In Chapter 5 we finally take some considerations about our investigation, on the results achieved and the future works.

# Chapter 2

## Background

Aim of this first chapter is to provide an overview of state-of-art and most popular languages and technologies for client-side Web applications programming. We will consider the paradigm shift that is taking place between unstructured and untyped languages such ECMAScript, for many reasons unsuitable for what concerns in-the-large programming, toward structured and optionally typed ones. We focus in particular to what concerns concurrency and asynchronous programming and how these features are provided in reference languages namely ECMAScript and Dart. At last, we take some considerations about several issues that affect the existing technologies and programming models, which will support the design of our innovative approach, based on Agent-oriented programming and the simpAL language, that will be treated in the following chapters.

### 2.1 JavaScript

According to the Wikipedia definition [27], JavaScript is a prototype-based scripting language that is dynamic, weakly typed and has first-class functions. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. JavaScript is dynamically typed in the sense that types are associated with values, not with variables. Objects are effectively associative arrays, so the dotted notation it's only syntactic sugar.



Prototyping is used as inheritance mechanism instead of classes, a prototype is an object from which other objects inherit properties. Functions are first-class entities and they are themselves objects, they can be assigned to variables, passed as arguments, returned by other functions, and manipulated like any other object. Functions in addition to their lexical closures makes JavaScript a functional language. Nowadays JavaScript is the world's most ubiquitous computing runtime and it can be considered the lingua franca of the Web.

It was originally developed in Netscape, by Brendan Eich under the name Mocha and then introduced as LiveScript in Netscape Navigator 2.0 in September 1995. Due to the widespread success of JavaScript as client-side scripting language for Web pages and the birth of compatible dialects by competing vendors, the year later Netscape delivered JavaScript to Ecma International for standardization so it was formalized in the ECMA-262 specification as ECMAScript language standard. It quickly reached the 3rd Edition in 1999, however, because of political differences concerning language complexity during the development, the 4th Edition was abandoned and it took ten years to come to the current 5th Edition. Features proposed for ECMAScript 4 was concerning the introduction of classes, and static types for the purpose to better support programming in the large. The controversies rose when it is realized that these new features would made the language backwards incompatible “breaking the Web”. Anyway many of semantic and syntactic innovations proposed in the 4th Edition, introduction of classes first of all, will be part of the 6th “Harmony” Edition [9].

### **2.1.1 JavaScript drawbacks**

JavaScript is a very powerful language and it boasts several advantages compared to other competitor languages in the Web scene and beyond. JavaScript is relatively simple to learn and implement, it now runs fastly on client thanks to browser engines performance improvements. Despite this, one of the most argued criticism moved to JavaScript is about the language suitability for in-the-large programming, where for in-the-large programming is intended Web applications with a big codebase, long development cycle and large teams.

With regard to this topic the Web developers community is splitted along two factions. The firsts claim that it's possible to tame JavaScript applying a strict discipline of programming eventually relying on one or many popular frameworks. Nevertheless an ever growing share of developers and experts in the field claim instead that JavaScript as-is is not suitable for large-scale Web programming. During an interview at the 2012 Lang.NEXT conference<sup>1</sup> Microsoft engineer Anders Hejlsberg states that you can write large-scale Web applications, but you can not maintain it. It's widely acknowledged that JavaScript applications suffer in maintainability primarily due to the lack of structure in JavaScript code and secondly, for the inadequacy of development tools. The absence of a static type system leads to be able to identify, only at runtime, errors identifiable at development time besides the inability to provide modern development features such as code refactoring, content assist and code navigation. JavaScript is also blamed to violate established software engineering principles, such as modularity, reusability, scalability and so on. For example consider that it is common practice in current Web applications to mix up user interface and business logic code, violating the separation of concerns and the Model-View-Controller (MVC) pattern in software engineering that is known to be helpful for long-term maintainability. In addition, due to it's event-driven programming model, it reintroduces problems concerning the control flow fragmentation and incomprehensibility just like *spaghetti code* as aftermath of the *GOTO* statement in the 1970s [17].

One of the key issues in JavaScript applications development concerns compatibility across multiple browsers. In particular the issue is due to few differences between the DOM (Document Object Model) interfaces, which are not part of the ECMAScript standard but are formalized in a multi-level specification by the W3C<sup>2</sup>. Despite the first standardization was dated 1997 [25], API differences still hold over between specific vendor implementations, especially with elderly browsers. Anyway these inconsistencies must be taken into account by Web developers in order to guarantee full accessibility and cross-

---

<sup>1</sup><http://channel9.msdn.com/Events/Lang-NEXT/Lang-NEXT-2012/Panel-Web-and-Cloud-Programming>

<sup>2</sup><http://www.w3.org/DOM/DOMTR>

browser compatibility to their applications, to this end many approaches could be adopted. Generally developers could programmatically check for browser features and give different behaviours to their applications according to the availability or not of these features. Things getting harder when for instance two browsers provide the same feature that behaves in a different way. In this cases the only solution could be to use browser-detection techniques and give the application different behaviours according to the browser vendor and version. Nevertheless the cleanest and definitely most used approach is to rely on frameworks and libraries that wrap access to the DOM entities providing an abstraction layer from browsers proprietary APIs. This is the case of jQuery and other frameworks that we will see in the following.

Another JavaScript weak point concerns finally the security, since it's a potential infection medium for malicious software gathered from the Web. User agents (browsers) typically adopt two different approaches for the purpose to contain risks tied to security. The former consists in running scripts inside a sandboxed environment in which they are bounded to perform Web-related operations only, preventing for example that scripts may access file or spawn processes. The latter is known as *same-origin policy* restriction whereby scripts from one domain can not have access to information such as cookies, usernames or passwords referring to another domain. This restriction is intended to avoid the so-called *cross-site* vulnerabilities which comprise in turn *cross-site scripting* and *cross-site request forgery*. All of these security vulnerabilities are usually caused by browser leaks or errors in the sandbox implementation that allows for instance buffer overflows.

### **2.1.2 The JavaScript frameworks ecosystem**

For the purpose to speed up and simplify the development of Web apps, and to tame some of the drawbacks seen before, so many frameworks and libraries built on top of JavaScript are available over the Web. Each of these frameworks have it's own goals and leads to different benefits, so the common approach is to combine many of them together and build the Web applications upon them. The JavaScript frameworks ecosystem is constantly evolving and in

Figure 2.1 are shown some of the most popular frameworks on the current Web landscape.



Figure 2.1: JavaScript frameworks ecosystem

Despite their relative abundance it's possible to classify frameworks in two big families according to goals they intended to pursue.

- *jQuery and DOM manipulation frameworks*

As you can understand from jQuery<sup>3</sup> motto, “The Write Less, Do More, Library”, this set of frameworks have the main purpose to speed up and simplify the development of JavaScript applications. jQuery is a free, open source library created in 2006 by John Resig and now it's used by over 55% of the 10,000 most visited Web sites [28]. jQuery provides many functionality in order to navigate and alter the document, freeing the developer from vendor-specific DOM API implementation, selecting and manipulating DOM elements through CSS selectors, simplifying events and AJAX requests, handling and creating effects and animations. Furthermore jQuery is cross-compatible and introduces a level of

---

<sup>3</sup><http://jQuery.com>

abstraction for browser low-level interaction that Web-apps built upon it may exploit in order to be much more portable. While it's still based on pure JavaScript, jQuery concise and enrich its syntax primarily thanks to the dollar sign \$ alias. Notice the elegance of the short code in Listing 2.1, in which an “Hello World!” message is shown inside the HTML element identified by “welcomeDiv” once the document is loaded.

---

```
1 $(document).ready(function(){
2     $('welcomeDiv').text('Hello World!');
3 });
```

---

Listing 2.1: jQuery “Hello World!”

- *Backbone.js and MVC frameworks*

The latter set of frameworks focus instead on developer friendliness and try in some way to give a structure to Web applications. The common intention behind most of these frameworks is to take the well known Model-View-Controller (MVC) pattern from software engineering and bring it into the Web-app deployment. The separation of concerns introduced by the MVC pattern is very useful as soon as applications grow in complexity. Backbone.js<sup>4</sup> is representative for this class of frameworks and is intended to give structure to Web applications by providing models with key-value binding and custom events, collections with a rich API of enumerable functions, views with declarative event handling, and connects it all to your existing API over a RESTful JSON interface. Backbone.js allows to represent data as Models, which can be created, validated, destroyed, and saved to the server. The architecture works in a such way that each time the model is updated by the GUI one or many events are triggered so all the views that display the model's state can be notified of the change and may cause the DOM to update.

---

<sup>4</sup><http://backbonejs.org>

- *Node.js*

An other widely used JavaScript framework which deserves to be mentioned although it's a far cry from those seen before and the client-side Web programming is Node.js<sup>5</sup>. Created in 2009 by Ryan Dahl for the purpose to easily building fast, scalable Web servers. Node.js introduces an event-driven, non-blocking I/O model that makes them lightweight and efficient. The following example (Listing 2.2) taken from the Node.js Web site shows in which simple way it's possible to write a Web server.

---

```
1 var http = require('http');
2 http.createServer(function (req, res) {
3     res.writeHead(200, {'Content-Type':
4                       'text/plain'});
5     res.end('Hello World\n');
6 }).listen(1337, '127.0.0.1');
```

---

Listing 2.2: Node.js tasting

### 2.1.3 JavaScript as intermediate language

A totally different, and quite new, approach exploits JavaScript ubiquitousness in order to engineer Web languages from scratch that compile to it. Under these conditions JavaScript thus assumes a role of intermediate language, the Web bytecode. This most comprehensive solution to build a new language can tackle the drawbacks that affects JavaScript in a more effectively way, adding on new features or making code easier to write. These new languages can be built from scratch, by porting of other languages or simply super-set extensions of JavaScript with which for instance can share the basic syntax. The Figure 2.2 below shows some of the most popular languages and tools that exploit the JavaScript compilation.

---

<sup>5</sup><http://nodejs.org>

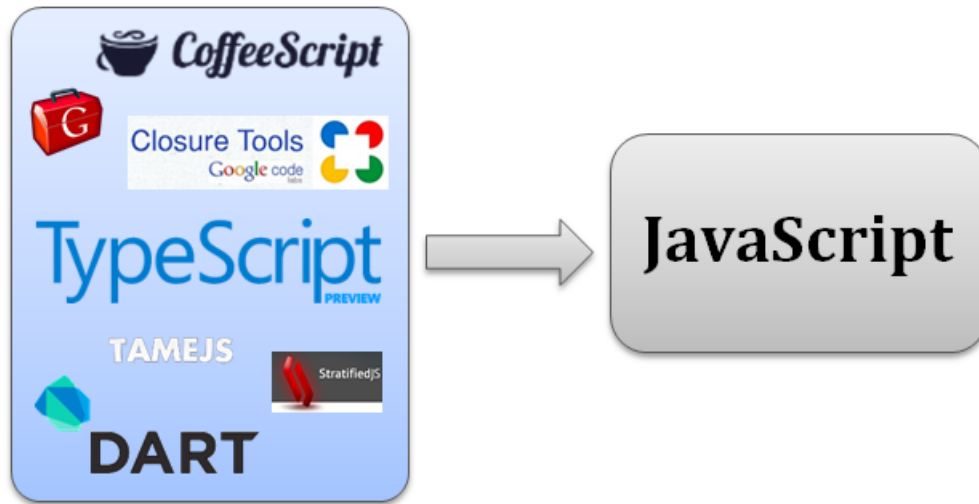


Figure 2.2: Languages and tools that compile to JavaScript

- *CoffeeScript*  
Designed by Jeremy Ashkenas in 2009 the main CoffeeScript<sup>6</sup> goal is to shrink the JavaScript programming making code more readable and concise, typically 1/3 fewer lines. It introduces syntactic sugar influenced by Ruby and Python. According to the CoffeeScript Web site it tends to run as fast or faster than the equivalent handwritten JavaScript. What should not be forgotten about CoffeeScript is that "It's just JavaScript", the code compiles one-to-one into the equivalent JavaScript and it's possible to use any existing JavaScript library seamlessly from CoffeeScript.
- *Google Web Toolkit*  
Introduced in 2006, main aim of Google Web Toolkit<sup>7</sup> (GWT) is to tackle JavaScript maintainability and reusability issues by building a language powerful enough and suitable for in-the-large programming. The GWT compiler compiles a sub-set of Java to JavaScript. Using GWT, developers can develop and debug Ajax applications in the Java language using the Java development tools of their choice.

<sup>6</sup><http://coffeescript.org>

<sup>7</sup><https://developers.google.com/web-toolkit>

- *StratifiedJS and TameJS*

StratifiedJS<sup>8</sup> and TameJS<sup>9</sup> are representative for all those languages and toolkits whose purpose is to discipline the JavaScript asynchronous programming model in order to avoid *spaghetti codes*. StratifiedJS extends JavaScript with a small number of constructs for asynchronous programming allowing to express asynchronous control flow in a straightforward sequential style. Taking up somehow the multi-threaded model, StratifiedJS organizes code in logical units they call *strata*, which differently to JavaScript are allowed to block at a particular point to be picked up again later at the same point where it left off. For instance, similarly to the *Thread.sleep()* Java statement, StratifiedJS *hold()* blocks for a given period of time. While one stratum is blocked, other strata can execute, but only one stratum is allowed to run at any one time and it's executed atomically up until the point where it either finishes or suspends. The language provides moreover a set of asynchronous constructs for combining strata, for instance parallel composition *waitfor/and* and alternatives composition *waitfor/or*. Notice the cleanliness of the simple, self explainable snippet in Listing 2.3 taken from the StratifiedJS Web site.

---

```
1 var news;
2
3 waitfor {
4     news = http.get("http://news.bbc.co.uk");
5 }
6 or {
7     hold(1000);
8     news = http.get("http://news.cnn.com");
9 }
10 or {
11     hold(1000*60);
12     throw "sorry, no news. timeout";
13 }
```

---

<sup>8</sup><http://onilabs.com/stratifiedjs>

<sup>9</sup><http://tamejs.org>



```
14  
15 show(news);
```

---

Listing 2.3: Tasting StratifiedJS strata

TameJS instead adds two features to JavaScript, namely *await* and *defer*. An *await* block defines code that won't return until each asynchronous task defined with *defer* has been completed.

- *Dart and TypeScript*

Two recently further alternatives are the languages Dart<sup>10</sup> and TypeScript<sup>11</sup> introduced respectively by Google in 2011 and by Microsoft in 2012. Both can compile to JavaScript and attempt to enforce structure in Web applications for the purpose to make them scalable in complexity. We will focus on these technologies in the following paragraphs.

## 2.2 Toward the next generation of structured Web languages

Recently an ever more growing share of software industries and Web developers complain about and accuse JavaScript not to be suitable for what concerns in-the-large programming, especially for ever more complex and bulky Web applications. This is largely attributable to JavaScript lack in structuring code. What developers and people from the industries would like is the support for constructs and features taken from the Object-Oriented Programming (OOP) such modules, classes interfaces and inheritance, that is everything that today, outside the Web, allows to build robust, reliable, reusable, maintainable and scalable systems. This should enfasts the deployment of large-scale Web applications porting the well-known design patterns typical of the OOP. At last this would promote also the separation of concerns between software components allowing teams of developers working on the same Web application without

---

<sup>10</sup><http://www.dartlang.org>

<sup>11</sup><http://www.typescriptlang.org>

interferences. The total lack of static typing is a further big handicap for in-the-large programming primarily for two reasons. Since there aren't any kind of checks on types, most of errors which are trappable at development time, are raised only at runtime. Moreover having a structured language and even only optionally typed brings the benefit of being able to build modern development tools which support all the AST-based features like code refactoring, content assist and code navigation that greatly improve the development experience. Many new Web languages were designed along this direction, nowadays most important are surely Google Dart and Microsoft Typescript. Summarizing, a paradigm shift is taking place in Web development scene, from the plain, untyped JavaScript language we are moving to structured, optionally typed, powerful languages.

### **2.2.1 Dart**

Google proposes the new Dart language with the main purpose to improve the state of the art of Web programming. According to the language specification [15] Dart is a class-based, single-inheritance, pure object-oriented programming language. Dart is optionally typed and supports reified generics. Dart was firstly introduced at the GOTO conference in October 2011 by Google's engineers Lars Bak and Gilad Bracha. According to a renowned leaked memo that summarizes a meeting of Google's most influent Web teams about the future of JavaScript [10] it's possible to understand the Web leading company's designs and reasons behind the Dart language, in the memo named Dash. As they state the ambitious goal of Dart "is ultimately to replace JavaScript as the lingua franca of Web development on the open Web platform". But why a new language? Why not trying to improve JavaScript? They justify this schism supporting that in large, complex applications, the kind that Google specializes in, are struggling against the platform and working with a language that cannot be toolled and has inherent performance problems. Moreover also in smaller-scale applications developers have to face a confusing labyrinth of frameworks and incompatible design patterns. At the same time, however, Dart should maintain the dynamic nature of Javascript and remains easy to

learn and to code with. So the purposes for which Dart is designed to are mainly: performances, developer usability, ability to be tooled, and security. For what concerns performances it must be possible to create VMs that do not suffer of the problems that all EcmaScript VMs have. Regarding developer usability, a key aspect is covered by the optional typing. Dart programs can be statically checked. The static checker will report some violations of the type rules, but such violations do not abort compilation or preclude execution. Dart still remains therefore a dynamic language. The optional typing is also fundamental for the ability of the language to be tooled, for large-scale projects that require code-comprehension features such as refactoring, code completion, code navigation, and more. Again with a view to large-scale programming, besides the optional static typing, key features are also classes and libraries. Dart supports classes as a fundamental structural building block for libraries and apps. Classes define the structure of an object, and you can extend them to create more specialized definitions. Libraries instead give modularity to Dart applications organizing the code in “units of encapsulation” which may be mutually recursive. The snippet in Listing 2.4 shows a simple Dart class taken from the Web site, you can notice the named constructor feature at line 7.

---

```
1 import 'dart:math';
2 class Point {
3   final num x, y;
4
5   Point(this.x, this.y);
6   Point.zero() : x = 0, y = 0;
7
8   num distanceTo(Point other) {
9     var dx = x - other.x;
10    var dy = y - other.y;
11    return sqrt(dx * dx + dy * dy);
12  }
13 }
```

---

Listing 2.4: Google Dart sample class

Dart is proposed as multiple target language. Must foregone targets are surely modern Web browser upon which, Google wish, it will be natively supported by dedicated VMs, such as JavaScript ones. However, ways in a such direction are harder since to this day no browser vendors have planned to natively support Dart, except for Google Chrome (Dartium browser) and first it will be adopted by other browsers Dart should become an open standard. Google is aware that it will be a huge challenge to convince other browser vendors to take in account a new language. For this purpose, as alternative way, it was introduced a cross compiler to JavaScript (dart2js, also coded in Dart) in order to make Dart applications to run over any legacy ECMAScript platform. Google designs Dart in a such way that a large subset of it can be compiled to JavaScript and they claim that dart2js is intended to implement the full Dart language specification and semantics [11]. Maybe Dart is still work-in-progress, however to this day there still are several semantic differences between applications executed through the native VM and compiled to JS, concerning particularly isolates<sup>12</sup> <sup>13</sup>.

The language is moreover targeted to front-end server too, in a similar way that Node.js works with JavaScript. This will allow large scale applications to unify on a single language for client and front end code. For this purpose the Dart project ships libraries for network IO, files and directories and the VM is designed also to run Dart programs on the server or command line.

## 2.2.2 TypeScript

More recent, and based on a totally different approach, is instead the Microsoft TypeScript language. The language development was led by Microsoft's engineer Anders Hejlsberg and was announced in October 2012. Hejlsberg is a prominent figure in the field of mainstream programming languages, he worked in Borland to Turbo Pascal and Delphi and in Microsoft where he has been the lead architect of the team developing the language C# [23]. Differently from Dart, TypeScript is an optionally static typed superset of ECMAScript that

---

<sup>12</sup><http://code.google.com/p/dart/issues/detail?id=4689>

<sup>13</sup>[https://groups.google.com/a/dartlang.org/forum/#!msg/misc/PyWBh13\\_10o/NQgemdPLS4kJ](https://groups.google.com/a/dartlang.org/forum/#!msg/misc/PyWBh13_10o/NQgemdPLS4kJ)

compiles to clean, plain JavaScript code which runs on any browser, or in any other ECMAScript3-compatible environment, including Node.js. TypeScript, in accordance with Microsoft “Embrace & Extend” philosophy, is fully compatible with the legacy ECMAScript syntax, every JavaScript program is also a TypeScript program, so it’s possible to use existing JavaScript code, incorporate popular JavaScript libraries, and be called from other JavaScript code. Microsoft describe TypeScript in the language specification [16] as syntactic sugar for JavaScript. Besides the compatibility with JavaScript, TypeScript introduces some features that makes the language suitable for application-scale development. The main reason behind TypeScript is in fact to meet the needs of the JavaScript programming teams that build and maintain large Web applications. TypeScript efforts the large-scale programming introducing on the one hand OOP constructs and functionalities, and on the other tools that en-fast ad enrich the developmente experience. TypeScript allows interfaces and classes declaration, and supports to organize code in dynamical modules, to help build robust components. The snippet in Listing 2.5 shows the Point class seen before in Dart implemented in TypeScript.

---

```
1  class Point {
2      x: number;
3      y: number;
4
5      constructor(x: number, y: number) {
6          this.x = x;
7          this.y = y;
8      }
9
10     getDist() {
11         return Math.sqrt(this.x * this.x +
12                             this.y * this.y);
13     }
14 }
```

---

Listing 2.5: Microsoft TypeScript sample class

Hejlsberg says TypeScript is actually based on proposed features of ECMAScript 6 “Harmony” including classes and modules [5]. TypeScript’s optional type system enables JavaScript programmers to use highly-productive development tools and practices: static checking, symbol-based navigation, statement completion, and code re-factoring. Somehow TypeScript pursues the same intents of the Dart language, but without a radical departure from JavaScript.

### 2.2.3 What’s the future *lingua franca* for the Web?

According to the paradigm shift whose taking place in the Web programming the possible scenario may mainly consists in pension off JavaScript, or improve it. These approaches are embodied by Dart on the one hand and by TypeScript and ECMAScript 6 on the other. So the question naturally arises, what will be the new *lingua franca* for the Web?

Let’s start taking some considerations about the languages syntax and semantic [8]. TypeScript is fully compatible with JavaScript extending its syntax and semantic only for what concerns the new features, that is optional typing and modules, classes and interfaces support. This is definitely winning since the effort required to milion of developers who already use JavaScript is minimal. Dart, on the other hand turns completely away from JavaScript introducing from scratch a new syntax (as close to the JavaScript one) and a new clear semantic for the main purpose to finally solve all JavaScript inherent issues, that can not be solved simply evolving the language concerning performances and the semantic. For instance when a Dart application tries to accede to a property which is not defined inside an object the VM will raise an exception instead of returning “undefined” such in JavaScript and so in TypeScript.

For what concerns compatibility with the huge amount of JavaScript frameworks and libraries, TypeScript being a super-set of JavaScript is fully compatible and interoperable for free with all JavaScript stuff. Dart, on the contrary, breaking away from JavaScript also loses everything is good. According to Dart FAQ [11] “moving to a new language will be a very large undertaking.

The specifics of how inter-operation with current Javascript would work is still an open question”.

Considering the support to asynchronous programming and concurrency, Dart introduce a semantic for Futures and Isolates respectively. However this leads to issues since arise semantic inconsistencies, for instance between Isolates and HTML5 Web Workers, both mechanisms for actor-based concurrency. Everything works fine until the Dart code is executed inside the VM, but what happens when it’s compiled to JavaScript? How Isolates will be mapped onto Web Workers? Probably things are still work in progress, but to now the Dart documentation says nothing about and, by the way, when Isolates are compiled to JavaScript they wont use Web Workers at all, running asynchronously but sequentially by the main UI worker <sup>14</sup>. TypeScript, more shrewdly, does not alter the semantics delegating matters to HTML5 Web Workers and libraries such jQuery Promises respectively.

On the other hand TypeScript carries with it all the problems that afflicts JavaScript including performances. Performances and predictability in performances are instead Dart primarily goals, notice, only for browser equipped with its native VM. Also for what concerns VMs Dart and TypeScript designers have different visions. Google push for implementig the Dart VM inside every browser, key aspect in order to take full advantage of Dart power and innovations. However, if other browser vendors should decide not to support Dart implementing its VM, the language runtime should be installed apart. In this scenario Dart may be confined to a plugin role, how technologies doomed to extinction such as Adobe Flash and Microsoft Silverlight, or exploited only as compile to JavaScript language. Regarding TypeScript instead, Microsoft has no plans for VM or any type of better support over the proprietary Internet Explorer platform.

Summing up TypeScript option is relatively low risk and should bring several benefits in the short-term compared to Dart. On the other hand Dart is very innovative language and promises to finally solve all JavaScript issue so it’s benefits may be many more despite it’s high risk, for the reasons seen

---

<sup>14</sup><https://groups.google.com/a/dartlang.org/forum/?fromgroups=#!topic/misc/koe2uTknJk>

above. Another factor to finally keep an eye on is the ECMAScript 6 “Harmony” standardization process which promises among other to bring classes, a module system and optional typing as well as constructs to tackle *Asynchronous spaghetti*, namely generators, directly in the JavaScript standard. Must be said that this process may be very slow, consider that these features, and more, was already proposed in ECMAScript 4 that takes about ten years to be then abandoned. Nevertheless, if the new standardization process would succeed it would bring back JavaScript to its former glory. Sure is that, at least for many years to come, JavaScript in native or compiled way will unopposed reign.

## 2.3 Web asynchronous programming

For asynchronous, or event-driven, programming is intended a paradigm in which the flow of the program is determined by events. This paradigm is widely used in graphical user interfaces where for example actions on keyboard keys or mouse buttons trigger different behaviours in the application. This is also the programming paradigm on which Web applications are based on. In Web applications events mainly concern changes or something happens in the DOM, notifications from pending AJAX requests or timing occurrences. Careful not to confuse asynchronous programming with concurrency, events may be asynchronous but are typically managed by the same control flow through the so-called event-loop. Each time an event occurs it's enqueued in the event-queue, the main control flow is organized as an event loop which each time picks an event from the event-queue and fires it, or blocking when the queue is empty, something like the pseudocode snippet in Figure 2.3. The application code is finally structured in terms of event handlers, also known as callbacks, attached to the application specific events, and invoked by the main thread whenever the related event is fired.



---

```
1: while true do
2:   ev ← PICKEVENT()
3:   PROCESS(ev)
4: end while
```

---

Figure 2.3: Event loop pseudocode

Events can be therefore queued while code is running, but don't forget they can not fire until the main thread is free, this is fundamental. In the following two examples we will try to better understand how things effectively work. In the snippet below (Listing 2.6) two callbacks are attached respectively to the start and stop buttons through the nice jQuery syntax. Through the start button is activated a loop that increments the value of a globale variable and prints it on the console, until a flag is setted; the button stop resets this flag. Once clicked on the start button, what we expect is to see prints on the console until the stop button will be clicked, however the browser get stuck since the start callback never returns and the event loop can not proceed in processing the stop event.

---

```
1 var counter = 0;
2 var enabled = false;
3
4 $('#buttonStart').click(function() {
5     enabled = true;
6     while (enabled)
7         console.log(counter++);
8 });
9
10 $('#buttonStop').click(function() {
11     enabled = false;
12 });
```

---

Listing 2.6: Understanding the event loop, first example

In general, considering again the event-loop snippet in Figure 2.3, the

loss of responsiveness in the GUI occurs whenever is fired an event to which are attached callbacks that takes long time or even never terminate. This is completely analogous to what happens in windowing systems and GUI of standalone platforms such as Windows Forms and Java AWT with the Event Dispatcher Thread (EDT). Not to be confused are instead the asynchronous events as introduced above with the notion of events in mainstream languages such C# based on the observer pattern.

The latter example (Listing 2.7) taken from [6] focus instead on the behaviour of a Web application when multiple events are triggered at the same time. In the snippet the JavaScript function `setTimeout` is invoked three times scheduling the print of the variable `i` with a delay of 0 milliseconds. What we expect as output in the console is something like “1 2 3”, however the effective result is “4 4 4”. Despite the timeout event occurs immediatly, because of the 0 milliseconds delay, all three the events are enqueued in the event-queue and fired as soon as the main thread is free, that is after the for loop is completed and the value of `i` is 4.

---

```
1 for (var i = 1; i <= 3; i++) {
2     setTimeout(function() {
3         console.log(i);
4     }, 0);
5 };
```

---

Listing 2.7: Understanding the event loop, second example

Despite its simplicity, the asynchronous paradigm leads to several issues. The primarily and most important one is surely the so-called *Asynchronous spaghetti* that we will explain in the following section. In general are as well problematic matters concerning the control flow of asynchronous operations, for instance lets consider the exception handling of an asynchronous operation like the one in the snippet below (Listing 2.8) inspired again by [6]. When the exception is notified at runtime the stack trace reports only an error in function C, but what happened to functions A and B? As seen before, since the JavaScript `setTimeout()` function schedules an asynchronous callback, this is

enqueued in the event-queue and then directly executed by the main thread in the event-loop, after the completion of function A and B. Moreover a common mistake is to put a try-catch block around asynchronous code such in the snippet in Listing 2.8, where the exception thrown in function C is not trapped by the try-catch statement.

---

```
1  try {
2      setTimeout(function A() {
3          setTimeout(function B() {
4              setTimeout(function C() {
5                  throw new Error('Async error!');
6              }, 0);
7          }, 0);
8      }, 0);
9  }
10 catch (ex) {
11     console.log('ASYNC EX:' + ex);
12 }
```

---

Listing 2.8: Exceptions in asynchronous code

Another serious problem, that we will not cover in this thesis, concerns testing of asynchronous Web applications. A bit like in concurrent programming, asynchronous events can be interleaved in arbitrary order leading to many different scenarios in the application behaviour.

### 2.3.1 Asynchronous programming troubles

Despite its simplicity, the asynchronous paradigm leads to several issues. Since the business logic is splitted between many event handlers, in which the software is organized to, the control flow too is non-linear, fragmented and so harder to be understood. Given the similarity with problems arising by the use of the *GOTO* statement, this effect takes the name of *Asynchronous spaghetti*. Even more problematic is the situation in which code in event handlers goes to act on global, shared variables. *Asynchronous spaghetti* is observable in

JavaScript primarily in terms of callback nesting. Callbacks nested in callbacks is a quite common pattern in JavaScript programs, mainly due to the ease with which it can be defined anonymous functions, resulting in the so-called *Matryoshka doll* programming style [6]. The snippet in Listing 2.9 is taken by a popular post on the Node.js Google Group<sup>15</sup> where a user complains for the level of nesting in his code.

---

```
1 mainWindow.menu("File", function(err, file) {
2   if(err) throw err;
3   file.openMenu(function(err, menu) {
4     if(err) throw err;
5     menu.item("Open", function(err, item) {
6       if(err) throw err;
7       item.click(function(err) {
8         if(err) throw err;
9         mainWindow.getChild(type('Window'),
10        function(err) {
11          if(err) throw err;
12          ...
13        });
14      });
15    });
16  });
17 });
```

---

Listing 2.9: “I love async, but I can’t code like this”

### 2.3.2 Futures and Promises

The term Promise can be dated back to 1976 when it was introduced by Daniel P. Friedman and David Wise<sup>16</sup> and made their first appearances in programming languages such as MultiLisp, E and Act. According to [14]

<sup>15</sup>[http://groups.google.com/group/nodejs/browse\\_thread/thread/c334947643c80968](http://groups.google.com/group/nodejs/browse_thread/thread/c334947643c80968)

<sup>16</sup>Friedman, Daniel; David Wise (1976). “The Impact of Applicative Programming on Multiprocessing”. International Conference on Parallel Processing, pp. 263-272.

“A Promise is a place holder for a value that will exist in the future. It is created at the time a call is made. The call computes the value of the Promise, running in parallel with the program that made the call. When it completes, its results are stored in the Promise and can then be claimed by the caller”. Therefore they describe an object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is yet incomplete [26]. Promises are two states entities: *blocked*, until the value is unavailable and, then, *ready*. Once a Promise is ready it remains ready from then on and its value never changes again. In common usage, the terms Promise, Future and Delay are roughly synonymous, however in [14] it is stated that Promises are strongly typed extensions of Futures. Great benefit of Promises is that these can be chained to each others enabling the creation of asynchronous workflows. Named as Promises pipelining, the idea was still introduced in [14] while they were working to an efficient asynchronous mechanism for procedure call in alternative to RPC, integrated in the language Argus. In the Web asynchronous programming we will exploit precisely these concepts of Promise and Promises pipeline for the purpose to unnest, in most cases, the *Matryoshka doll* code results of callbacks handling. According to Promises and Futures classification, these ones used in Web programming are explicit and with non-blocking semantic. Explicit Promise are usually supported by library and expectes that should be the user to call a function in order to obtain the value. Non-blocking semantic means instead that the value of the Promise is accessed asynchronously. Another important aspect related to Promises concerns the read-only views. In several programming languages and in the libraries that we’ll see in the following, it is possible to obtain a read-only view of a Future, which allows reading its value when resolved, but does not permit resolving it. Support for read-only views is based on the *Least Authority* principle, according which the ability to set the value of a Future has to be restricted only to subjects that need to set it. As we will see in jQuery, Promises are read-only views of Deferreds as in Dard Futures are read-only views of Completers.

### 2.3.3 JavaScript jQuery Promises

Promises were introduced in the popular jQuery JavaScript library, version 1.5 taking cue from `dojo.Deferred`<sup>17</sup> and the CommonJS Promises/A specification<sup>18</sup> inspired in turn by the Python Twisted framework<sup>19</sup>. Promises represent asynchronous tasks with two possible outcomes, success or failure, and hold callbacks that fire when one outcome or the other has occurred. So at any time a Promise can be in pending, resolved or rejected state [6]. As seen above, nested callbacks make the code hard to be understood. Promises were proposed as ultimate solution to *Asynchronous spaghetti* in the form of nested callbacks, in most cases in fact, it's possible to linearize the flow primarily thanks to the Promise's pipe mechanism, that we'll better describe in the following. Another important benefit introduced by Promises concerns encapsulation. Let's think to an application in which for instance the result of an AJAX request affects several parts of the application behaviour. Common practice would expect that all the changes that affect the application in all its parts are made in the AJAX completion callback. However thanks to this new approach it's possible to pass around the Promise object so that each part of the application can register its own callback independently by the others.

#### 2.3.3.1 Deferred and Promise

Deferred and Promise terms are often used interchangeably to refer the same concept. Formally the difference is that Promises are read-only views of Deferreds, so a Deferred is a Promise too with, in addition, operations such *reject()* and *resolve()* that triggers respectively the *fail()* and *done()* callbacks. Notice that Deferred has associated with one and only Promise and shares with it all the callbacks internally. Therefore differences between Deferred and Promise are finally for safety purpose, Promises are read-only views of Deferreds, so the one who create the Deferred object keep it hidden and then resolve or reject it, sharing with others only Promises. The pseudo UML

---

<sup>17</sup><http://dojotoolkit.org/reference-guide/1.8/dojo/Deferred.html>

<sup>18</sup><http://wiki.commonjs.org/wiki/Promises/A>

<sup>19</sup><http://twistedmatrix.com/trac>

class diagram below (Figure 2.4) summarize the API to jQuery Deferred and Promises.

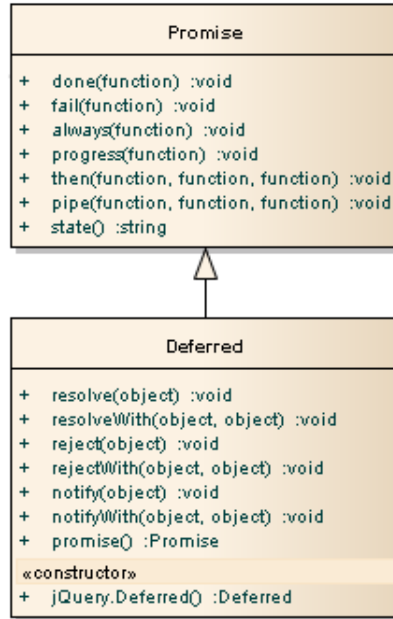


Figure 2.4: jQuery Promises API

### 2.3.3.2 Promises and AJAX

In general is therefore always possible to wrap up an asynchronous function in the Deferred/Promise construct. Particularly convenient is to have AJAX functionalities working with Promises. In the snippet in Listing 2.10 the function *ajax()* wraps an AJAX get request, returning the Promise object which will be used to retrieve the response in the future. Notice how functionalities such this new one can greatly simplify the code of Web 2.0 applications. For this reasons libraries such jQuery already provide these AJAX functions in Deferred way such as *jQuery.ajax()*, *jQuery.get()*, *jQuery.post()* (or simply *\$.ajax()*, *\$.get()*, *\$.post()*).

---

```

1 function ajax(url) {
2     var deferred = new $.Deferred();
  
```

```

3
4     var req = new XMLHttpRequest();
5     req.addEventListener('load', function () {
6         if (req.status == 200)
7             deferred.resolve(req.responseText);
8         else
9             deferred.reject('AJAX failed: '
10                + req.status);
11     }, false);
12     req.open('GET', url, true);
13     req.send();
14
15     return deferred.promise();
16 }
17
18 ajax('http://www.google.com')
19     .done(function(res) {
20         // Do something
21     });

```

---

Listing 2.10: Promising AJAX

### 2.3.3.3 Promises progress

Going back to the diagram in Figure 2.4 it can be noticed that Promises provides also the *progress* callback. This feature, introduced later in jQuery 1.7, allows the creator of the Deferred to *notify* the Promise subscribers with data related to progress of the asynchronous task. Just like resolve and reject, notify can take arbitrary arguments. Take a look at the snippet in Listing 2.11 were through *setTimeout()* and Promises progress feature it's implemented a simple countdown.

---

```

1 function countdown(s) {
2     var deferred = new $.Deferred();
3
4     function step(ts) {
5         if (ts > 0) {

```



```

6         deferred.notify(ts);
7         setTimeout(function() {
8             step(ts-1);
9         }, 1000);
10    }
11    else
12        deferred.resolve();
13    }
14    step(s);
15
16    return deferred.promise();
17 }
18
19 countdown(10)
20     .progress(function(t) {
21         log('-' + t);
22     })
23     .done(function() {
24         log('GO');
25     });

```

---

Listing 2.11: JavaScript countdown

### 2.3.3.4 Promise pipelining

The most important benefit introduced by the Promises is surely the ability to chaining Promises together in a pipeline fashion. Through the *pipe()* function it's possible to schedule a further asynchronous task, described in term of Promise, to be started at the completion of a Promise. The *pipe()* function retrieves in the same way a Promise that represents a single asynchronous task, union of all the tasks in the pipeline. The resultant Promise succeed if all the Promises in the pipeline are resolved, or fails as soon as one Promise is rejected. This is useful both for attach handler at the end or at the progress of the whole pipeline but, above all, for a centralized errors handling. In order to appreciate the benefits given by the pipe mechanism lets consider a dummy application which must at first query a RESTful Web service, then, perform a request to an other Web service based on the first response, and finally, display

this last response on the Web page. This dummy application is coded in the Listing 2.12 below.

---

```
1 $.ajax({
2     url: 'http://example.com/serviceA/' + reqA,
3     async: true
4 })
5 .pipe(function (e) {
6     var def = new $.Deferred();
7     var reqB = computeNewRequest(e.data);
8     def.resolve(reqB);
9     return def.promise();
10 })
11 .pipe(function (reqB) {
12     return $.ajax({
13         url: 'http://example.com/serviceB/' + reqB,
14         async: true
15     });
16 })
17 .done(function (e) {
18     display(e.data);
19 });
20 .fail(function (ex) {
21     console.log(ex);
22 });
```

---

Listing 2.12: Asynchronous workflow with pipes

Notice that without Promise pipelining the code of the dummy application would look like *spaghetti* in Listing 2.13.

---

```
1 $.ajax({
2     url: 'http://example.com/serviceA/' + reqA,
3     async: true
4 })
5 .done(function (e) {
6     var reqB = computeNewRequest(e.data);
7     $.ajax({
```

```

8         url: 'http://example.com/serviceB/' + reqB,
9         async: true
10    })
11    .done(function (e) {
12        display(e.data);
13    })
14    .fail(function (ex) {
15        console.log(ex);
16    });
17 })
18 .fail(function (ex) {
19     console.log(ex);
20 });

```

---

Listing 2.13: Asynchronous workflow without pipes

So, one of the primary benefits introduced by the Promise pipelining is to organize linear workflows, untangling *spaghetti code* caused by callbacks nesting. However, as we will see in the remainder of the discussion, this is only partially true since there are many scenario in which the workflow dynamically changes on the basis of task's result in the workflow itself. An other issue is related to usability, when we build asynchronous workflows through Promise pipelining, all the tasks part of the workflow have to be handled asynchronously, including those which for themselves would be not. Consider for instance lines 5 to 10 of Listing 2.12, purpose of this snippet is only to compute the argument of the second request (line 7), anyhow it has to be mapped as Deferred in order to allow that following tasks can be piped. By the way, as you can see from the two snippets below, an other great benefit given by Promises pipelining is to centralize the error handling implementing only a *fail()* callback for the whole workflow avoiding the need to handle each task individually (like in Listing 2.13).

### 2.3.3.5 Combining Promises

Another powerfull and fundamental Deferred/Promise mechanism, orthogonal to pipelining, is given by *jQuery.when()* (or simply *\$.when()*) function that

allows to process many asynchronous Promises in parallel and react when all the Promises are fulfilled. The *jQuery.when()* function take as input many Promises and returns a Promise object that represents the parallel execution of all the Promises. The Promise it generates is resolved as soon as all of the given Promises are resolved, or rejected as soon as any one of the given Promises is rejected. In the Listing below 2.14 is shown the JavaScript code for the demo application called “Battle of the bands” in which, through two asynchronous AJAX requests the YouTube service is queried in order to retrieve the view count related to two rock music masterpieces, in order to elect as winner the most listened one. Thanks to the *jQuery.when()* function it’s possible to attach the callback to the completion of both the requests, handled as Futures, and then determine the winner based on the view counts contained in the responses, available as arguments of the *jQuery.when()* callback.

---

```
1 function youtubeSearch(search) {
2     return $.ajax({
3         url: 'http://gdata.youtube.com/feeds/api/videos?'
4             + 'q=' + encodeURIComponent(search)
5             + '&orderby=viewCount&max-results=1'
6             + '&v=2&alt=jsonc',
7         dataType: 'jsonp',
8         async: true
9     });
10 }
11
12 var contenders =
13     new Array('Led Zeppelin Stairway to heaven',
14             'Pink Floyd The wall');
15
16 var promise1 = youtubeSearch(contenders[0]);
17 var promise2 = youtubeSearch(contenders[1]);
18
19 var winner;
20 $.when(promise1, promise2).done(
21     function (r1, r2) {
22         winner = (r1[0].data.items[0].viewCount >
23                 r2[0].data.items[0].viewCount ?
```

```

24         0 : 1);
25     $('#res').text('The winner is: '
26         + contenders[winner]);
27     }
28 );

```

---

Listing 2.14: JavaScript “Battle of the bands”

Notice that a very common error, effect of asynchronous programming, is to access a variable setted by a callback, before that the callback is effectively performed. To draw on the “Battle of the bands” demo application seen before, in Listing 2.15 is highlighted the error when accessing the variable *winner* outside the *\$.when()* callback.

---

```

1  ...
2  var winner;
3  $.when(promise1, promise2).done(
4      function (r1, r2) {
5          winner = (r1[0].data.items[0].viewCount >
6                  r2[0].data.items[0].viewCount ?
7                  0 : 1);
8      }
9  );
10 $('#res').text('The winner is: ' + contenders[winner]);
11 // winner = undefined

```

---

Listing 2.15: JavaScript “Battle of the bands” error

Thanks to the JavaScript *prototype.apply()* function it’s also possible to pass to *jQuery.when()* an array of Promises instead of a comma separated pre-defined set of Promises as arguments.

### 2.3.4 Dart Futures

Dart in a very similar way provides the Completer/Future mechanism. While the semantic is practically the same of jQuery Deferred/Promise there are

slight differences in syntax. The differences compared with jQuery, as summarized in Table 2.1, are minimal. Compared to Promises, Futures doesn't support the progress notification of an asynchronous task, or at least for the moment. Both Completer and Future bring instead the advantage to be (optionally) generic typed, so the result value produced by the asynchronous task will be of the type specified in the Future.

<b>jQuery Promise</b>	<b>Dart Future&lt;T&gt;</b>	<b>jQuery Deferred</b>	<b>Dart Completer&lt;T&gt;</b>
done()	then()	resolve()	complete()
fail()	handleException()	reject()	completeException()
progress()	-	notify()	-
always()	onComplete()	promise()	future
then()	-		
pipe()	chain()		
stete()	isComplete		
jQuery.when()	Futures.wait()		

Table 2.1: Comparison between jQuery Promises and Dart Futures APIs

## 2.4 Concurrency in Web applications

One of the elements that underlie the gap between Web and desktop-based native applications surely concerns performances. JavaScript engines are becoming increasingly faster and powerful, however this is still not enough. Due to the gigahertz limit, modern CPUs are even more multi-cored, including those ones onboard to mobile devices. Even so JavaScript historically suffers of an important limitation: all its execution process remains inside a unique thread, therefore failing to fully exploit the underlying hardware. So usually happens that long-running computations or script that never ends will freeze the Web page. HTML5 bridge the gap enabling modern Web applications with concurrency in Actor-like fashion, through the so-called Web Workers. Actors are computational entities with their own control flow that interact each others by asynchronous message passing. For this purpose actors are equipped with a queue in which incoming messages are stored, and its control flow is driven by the message-loop, whereby at each step a message is picked from the

queue (if any is available) and a proper message handler is selected. Anyway details about the actor model is outside the scope of this thesis so for further informations refer to [2] and [1].

Before Web Workers, JavaScript developers try to “mimic” concurrency exploiting asynchronous techniques such *setTimeout()* and *setInterval()* methods<sup>20</sup>. Anyway these timing techniques simply schedule functions that will be enqueued (as seen in the previous section 2.3) and then are executed by the same main thread in a sequential way. In these terms concurrency was seen as an hack.

### 2.4.1 HTML5 Web Workers

Web Workers were introduced in HTML5 and defines an API for spawning background scripts which communicates each others through message passing in an Actor-like fashion. Typical scenarios in which Web Workers could be used for are, for instance, computationally long-running task keeping the user interface responsive or tasks for background I/O through *XmlHttpRequests*. Because of their multi-threaded nature, Web Workers are bounded to use only a subset of JavaScript features. For example the biggest limitation is that Web Workers can not access the Document Object Model (DOM) because this would not be thread-safe and may leads to race conditions. So only the so-called main UI worker has the permission to access the DOM and other workers that want to interact with it must pass by this UI worker. Web Workers are instead allowed to use *XmlHttpRequests*, as well as timing functions such *setTimeout()/clearTimeout()* and *setInterval()/clearInterval()*, import external scripts through the *importScripts()* method, access the application cache and spawning other Web Workers [3]. As seen before, Web Workers communicate through message passing, which takes place through ports of which each worker is equipped with. For safety purpose messages are serialized, typically through JSON, however this leads to performance issues especially for large messages and heavy data structures. For the purpose to

---

<sup>20</sup><http://www.codeproject.com/Articles/271571/Introduction-to-HTML5-Web-Workers-The-JavaScript-M>

enhance Web Workers message passing performances, techniques like transferable objects were proposed, where messages are not serialized but directly passed (as reference) to the recipient worker transferring the ownership of the content too. For what concerns errors handling, when an error is thrown inside a worker an event containing the file name and line number of JavaScript file where the error occurred besides a message that provides a meaningful description of the error itself is fired. Therefore the parent worker by registering the proper event handler can perceive and react accordingly. Web Workers can be terminated both from the parent worker through the *terminate()* method and by itself calling *self.close()*. According to the specification [13] when a worker is terminating the user agent atomically discard any tasks that have been added to the event-loop's task queues and set the worker's closing flag to prevents any further tasks from being queued. A great benefit from Actor model in Web Workers concerns the ability of a worker to recursively spawn child workers in order, for instance, to tackle a complex task exploiting concurrency. However, according to the W3C specification Web Workers [13] are relatively heavy-weight, and are intended to be used in small numbers. Generally, workers are expected to be long-lived, have a high start-up performance cost, and a high per-instance memory cost. In relation to performances, a key aspect concerns how workers are mapped onto physical threads. Most browsers adopt a one-to-one mapping, spawning separate threads (or processes) for each worker, so it's easy when the number of workers grow, to see system performances degrade. Beside the mapping between workers and physical threads, several differences subsist in Web Workers implementations and support between different browsers and the W3C specification, in Table 2.2 main ones are summarized. The W3C Web Workers specification provides two type of workers which differ mainly in the use: the Dedicated and Shared Web Workers, that we will be explored in the following paragraphs.



	Chrome 23.0.1271	Firefox 16.0.2	Opera 12.10	Safari 5.1.7	Internet Explorer 10.0.1008
Dedicated workers	✓	✓	✓	✓	✓
Dedicated sub-workers <sup>21</sup>	✗	✓	✓	✗	✓
Shared workers	✓	✗	✓	✓	✗
Transferable objects	✓	✗	✗	✗	✗

Table 2.2: Browsers compatibilities to HTML5 Web Workers

#### 2.4.1.1 Dedicated Workers

Dedicated workers are created by and linked to their parent workers which are binded to by an implicit message port, used as communication channel between the parent worker and the Dedicated worker itself. A Dedicated worker is created through the *Worker()* constructor that accepts as only argument the JavaScript file which contains the code of the worker itself and that usually must satisfy the *same-origin* policy. The worker is effectively spawned when the parent send to it the first message. As seen before dedicated workers act as if they had an implicit message port associated with them so through the *self.onmessage* they can attach a callback that runs when a message is received from the parent and through the *self.postMessage()* they can send messages back to the parent worker.

For the purpose to test concurrency in Web applications we introduce the demo application called “Stoppable Counter” (see the HTML page in Listing A.1) in which through appropriate buttons it’s possible to activate and stop a concurrent counter that cyclically increments a value displayed in the page. Since Web Workers behave as actors, it’s not possible to implement a worker which, when receiving the start message, starts an endless loop that

---

<sup>21</sup>Dedicated workers spawned by other dedicated workers, such in Listings 2.17 2.18

increments the value, because this would stall the actor's event loop making it unresponsive to new messages (see example in Listings 2.21 2.22). So a well-known possible workaround consists in breaking the loop control flow through messages sent by the worker to itself for the purpose to keep responsiveness. However a limitation due to the implicitness of the message port in Dedicated workers is that the worker can not send messages to itself. So the trick adopted to implement the above-mentioned application through Dedicated Web Workers is to split the business logic between a couple of workers. The first (Listing 2.17) is responsible for the interaction with the UI main worker (Listing 2.16) inside the page and the communication with the second worker (Listing 2.18), which keeps the state of the counter. When the second worker receives a message, increments the value and sends it back to the first worker which, in turn, sends the value back to the UI main worker that displays it and then sends again a message to the second worker, until the stop message is not received.

---

```
1 var worker = new Worker('workerMaster.js');
2 worker.onmessage = function (e) {
3     if (e.data.msg == 'ret')
4         $('#res').text('' + e.data.val);
5     else if (e.data.msg == 'log')
6         log(e.data.val);
7 };
8 worker.onerror = function (e) {
9     log('ERROR: ' + e.message + ' @ ' +
10        e.filename + ':' + e.lineno);
11 };
12
13 $('#buttonStart').click(function() {
14     worker.postMessage({'cmd': 'start'});
15 });
16 $('#buttonStop').click(function() {
17     worker.postMessage({'cmd': 'stop'});
18 });
```

---

Listing 2.16: Dedicated worker counter – main.js

---

```
1 var subWorker;
2 var enabled = false;
3
4 onmessage = function (e) {
5     if (e.data.cmd == 'start' && !enabled) {
6         enabled = true;
7
8         subWorker = new Worker('worker.js');
9         subWorker.onmessage = function (e) {
10            postMessage(e.data);
11            if (enabled)
12                subWorker.postMessage('');
13        };
14        subWorker.postMessage('');
15
16        postMessage({'msg':'log',
17                    'val':'Worker started!'});
18    }
19    else if (e.data.cmd == 'stop' && enabled) {
20        enabled = false;
21        postMessage({'msg':'log',
22                    'val':'Worker stopped!'});
23    }
24 };
```

---

Listing 2.17: Dedicated worker counter – workerMaster.js

---

```
1 var counter = 0;
2
3 onmessage = function (e) {
4     counter = counter + 1;
5     postMessage({ 'msg': 'ret', 'val': counter });
6 };
```

---

Listing 2.18: Dedicated worker counter – worker.js

The limitation introduced by the Dedicated workers implicit port could be circumvented again by exploiting the asynchronous *setTimeout()* function

instead of the second worker. As will be explained in the next section 2.4.1.3 “Web Worker’s event loop”, thanks to the `setTimeout(func, 0)` function it would be possible in fact to schedule the callback `func` to be executed at the next event-loop step, keeping the worker responsive to new messages too.

Notice that the possibility to dynamically change the `self.onmessage` handler, also in the callback upon receipt of the message itself, precisely corresponds to the actor *became*, that is the action whereby an actor designates the behavior to be used for the next message it receives.

### 2.4.1.2 Shared Workers

Shared workers, on the other hand, once created are reachable from all the workers that want to communicate with them, in respect to the *same-origin* policy and so inside the same application. For this purpose Shared workers are named, in a such way that can be referenced by other workers. Shared workers are therefore created through the `SharedWorker()` constructor, which takes as arguments the URL to the JavaScript file that codes it and, eventually, the name of the worker. The name is useful when, for instance, many Shared workers are spawn starting from the same JavaScript file. The typical use case for Shared workers is when you want to provide a common service inside a Web application in order, for instance, to keep a shared state. Unlike Dedicated workers, they explicitly have multiple message ports and as many related message handlers, one for each connection with other workers, so there are slightly differences in APIs, especially on the worker side. Through the `self.onconnect` event the Shared worker is announced when a client worker spawn or reference it. So then, the Shared worker can retrieve the port object used for the communication with the caller worker from the `onconnect` event argument and attaches an handler to the `port.onmessage` event or use the `port.postMessage()` to initialize the communication.

In Listings 2.19 2.20 below the above-mentioned “Stoppable counter” application is implemented through a single Shared worker exploiting, this time, the self sending ability of Shared workers.

---

```

1 var worker = new SharedWorker('worker.js');
2 worker.onmessage = function (e) {
3     if (e.data.msg == 'ret')
4         $('#res').text('' + e.data.val);
5     else if (e.data.msg == 'log')
6         log(e.data.val);
7 };
8 worker.onerror = function (e) {
9     log('ERROR: ' + e.message + ' @ ' +
10        e.filename + ':' + e.lineno);
11 };
12
13 $('#buttonStart').click(function() {
14     worker.postMessage({'cmd': 'start'});
15 });
16 $('#buttonStop').click(function() {
17     worker.postMessage({'cmd': 'stop'});
18 });

```

---

Listing 2.19: Shared worker counter – main.js

---

```

1 var counter = 0;
2 var enabled = false;
3 var portUI;
4
5 onconnect = function (ce) {
6     ce.ports[0].onmessage = function (mgs) {
7         messageHandler(mgs, ce.ports[0]);
8     };
9 };
10
11 var messageHandler = function (e, portSender) {
12     if (e.data.cmd == 'start' && !enabled) {
13         if (portUI == null)
14             portUI = portSender;
15
16         enabled = true;
17         postInc();
18         portUI.postMessage({'msg': 'log',

```

```

19         'val': 'Worker started!'}));
20     }
21     else if (e.data.cmd == 'stop' && enabled) {
22         enabled = false;
23         portUI.postMessage({'msg': 'log',
24             'val': 'Worker stopped!'});
25     }
26     else if (e.data.cmd == 'inc') {
27         counter = counter + 1;
28         portUI.postMessage({'msg': 'ret',
29             'val': counter});
30         if (enabled)
31             postInc();
32     }
33 };
34
35 function postInc() {
36     var sw = new SharedWorker('worker.js');
37     sw.port.onmessage = messageHandler;
38     sw.port.postMessage({'cmd': 'inc'});
39 }

```

---

Listing 2.20: Shared worker counter – worker.js

### 2.4.1.3 Web Worker’s event loop

Introducing concurrency in Web applications through Web Workers leads to the conceptual problem to integrate two different programming models that is the actor model and the JavaScript asynchronous model. Actually, the integration of the two models is quite simple since both are based on a cyclic behaviour. In fact the event-loop and event-queue that underlie the JavaScript asynchronous model are very similar to the receive loop and the message queue in the actor model. The solution is therefore to unify the two loops in the Worker’s event loop, enqueueing in the same event-queue events and received messages. In a similar way to what described in Figure 2.3 the event-loop cyclically, until the closing flag is not set, waits until the queue is empty, picks each time an event or a message from the event queue and calls respectively the

event handler or the *port.onmessage* callback. The example in the following snippets it is significant to understand how the Web Worker's event loop really works. The UI main Worker (Listing 2.21) spawn a Dedicated Worker and sends to it start and stop messages through the related buttons. When the Dedicated Worker (Listing 2.22) receives the start message it starts a loop where the variable *counter* is incremented, until the flag *enabled* is set. When it receives the stop message the flag is resetted.

---

```
1 var worker = new Worker('worker.js');
2 worker.onmessage = function (e) {
3     if (e.data.msg == 'ret')
4         $('#res').text('' + e.data.val);
5     else if (e.data.msg == 'log')
6         log(e.data.val);
7 };
8 worker.onerror = function (e) {
9     log('ERROR: ' + e.message + ' @ ' +
10        e.filename + ':' + e.lineno);
11 };
12
13 $('#buttonStart').click(function() {
14     worker.postMessage({'cmd':'start'});
15 });
16 $('#buttonStop').click(function() {
17     worker.postMessage({'cmd':'stop'});
18 });
```

---

Listing 2.21: Unresponsive counter – main.js

---

```
1 var counter = 0;
2 var enabled = false;
3
4 onmessage = function (e) {
5     if (e.data.cmd == 'start' && !enabled) {
6         enabled = true;
7         postMessage({'msg':'log',
```

```

8             'val': 'Worker started!'}));
9     while (enabled) {
10         counter++;
11         postMessage({'msg': 'ret', 'val': counter});
12     }
13 }
14 else if (e.data.cmd == 'stop' && enabled) {
15     enabled = false;
16     postMessage({'msg': 'log',
17                 'val': 'Worker stopped!'});
18 }
19 };

```

---

Listing 2.22: Unresponsive counter – worker.js

What we would expect is that, once the start button is pressed, the Dedicated Worker will start increment the counter, until the user will click on the stop button. However once started the Dedicated Worker will ignore any later commands, that happens because the increment cycle never yield, so the Worker's event loop stalls and the stop command, that effectively is on the message-queue, will never be processed.

#### 2.4.1.4 Mixing Web Workers and asynchronous programming

Concurrency through Web Workers and asynchronous programming based on jQuery Deferred/Promises can be mixed in several ways, for instance in order to retrieve, as Future, the result produced by a computation executed concurrently through a separate Web Worker. Many approaches are proposed on the Web, the one that we present in the following is inspired by [7]. In the example in the snippet below (Listing 2.23) through the *\$.doAsync()* function is requested to a worker coded in the script *adder.js* (Listing 2.24) to concurrently sum the values stored in the array passed as argument. The result of the concurrent computation is returned as Future, so it's possible to attach callbacks for the completion or the progress of the task, or again to play with the *\$.when()* function, such in the example, where two concurrent sum tasks are launched through respective workers and, at the completion of both, it's



printed on the page the overall sum. The `$.doAsync()` function contains the business logic needed to spawn the Dedicated worker and manage the interaction between the messages exchanged and the Deferred's `resolve()`, `reject()` and `notify()` operations, through a predetermined protocol.

---

```
1 $.doAsync = function (workerId, workerUrl, args) {
2     var def = new $.Deferred();
3
4     var worker = new Worker(workerUrl);
5     worker.onmessage = function (event) {
6         if (event.data.progress != null)
7             def.notify({'workerId' : workerId,
8                         'port' : worker,
9                         'data' : event.data.progress});
10        else if (event.data.ret != null)
11            def.resolve({'workerId' : workerId,
12                       'data' : event.data.ret });
13    };
14    worker.onerror = function (ex) {
15        def.reject({'workerId' : workerId,
16                  'error' : ex });
17    };
18    worker.postMessage(args);
19    return def.promise();
20 };
21
22 $(document).ready(function () {
23     var future1 = $.doAsync('worker1',
24                            'adder.js',
25                            [0,1,2,3,4]);
26     var future2 = $.doAsync('worker2',
27                            'adder.js',
28                            [5,6,7,8,9]);
29
30     $.when(future1, future2).done(
31         function (r1, r2) {
32             $('#res').text('The result is: '
33                            + (r1.data+r2.data));
34         });
35 });
```

---

Listing 2.23: Deferred worker – main.js

---

```
1 onmessage = function (e) {
2     var a = e.data
3     var sum = 0;
4
5     for (var i=0; i<a.length; i++)
6         sum = sum + a[i];
7     postMessage({'ret' : sum});
8 };
```

---

Listing 2.24: Deferred worker – adder.js

## 2.4.2 Dart Isolates

Also in Dart concurrency is provided in actor-like fashion through the so-called Isolates, not to be confused with HTML5 Web Workers since the semantic is slightly different. According to Dart documentation [24], an isolate is a unit of concurrency and it has its own memory and its own thread of control. Isolates communicate by message passing and no mutable state is ever shared between them. For this purpose messages are serialized before received in a such way to ensure that one isolate cannot directly change the state in another isolate. In addition to performance improvements given by concurrency, Isolate were proposed by Google for security matters too, for example to run third-party code more securely thanks to the heap separation. Dart Isolates and `SendPort` are packaged in the isolates library `dart:isolate` that has to be imported when using isolates. An Isolate can be spawned through the `spawnFunction()` function that expects as only argument a top-level function or a static method as entry point and returns a `SendPort` that can be used in order to communicate with the newly created Isolate. This entry point function should not expect

arguments and should return void. An Isolate can be spawned too starting from an URI which represents the file that contains the Isolate code, through the `spawnUri()` function. Messages are sent and received between Isolates through ports and they can contain primitive values, data structures such as lists or maps or instances of `SendPort` used to refer another Isolate. Before long it will be possible to pass any object in messages, now this feature is available only when the application runs in the Dart VM. Ports, that is the way in which Isolates exchange messages, can be of two types: `SendPorts` and `ReceivePorts`. Messages are sent through the `SendPort send()` method and received attaching callbacks to `ReceivePort receive`. An isolate, when spawned, has a default `ReceivePort` retrievable by the top-level property `port`. Anyway it's possible to instantiate more `ReceivePorts` for the same Isolate, for the purpose to route messages to different ports and callbacks. An Isolate lives until it has an open `ReceivePort`. In addition to the `send()` method the `SendPort` interface provides also the `call()` mechanism that embeds a request-response pattern, providing the response as a `Future`, returned by the `call()` invocation. The snippet in Listing 2.25 shows the demo application introduced for Web Worker coded in Dart with Isolates.

Notice, from the language documentation [24], that Isolates might run in a separate process or thread depending on the implementation, or eventually wrapped by Web Workers when compiled to JavaScript. To date Isolates specification seems still ongoing, APIs has been refactored many times and several semantic inconsistencies persists between Isolates compiled to JavaScript and runned in the native VM<sup>22</sup>. Between the features removed in the last API refactoring there are the so-called LightIsolates that is isolates which can access the DOM, and are therefore executed by the main UI thread.

---

```
1 #import('dart:html');
2 #import('dart:isolate');
3
4 void main() {
```

---

<sup>22</sup>See my discussion on Google Groups “About Dart isolates” at <https://groups.google.com/a/dartlang.org/forum/?fromgroups=!topic/misc/koe2uTknJk>

```

5   SendPort counterPort = spawnFunction(isolateCounter);
6
7   ReceivePort receiver = new ReceivePort();
8   receiver.receive((msg, _) {
9       if (msg["msg"] == "ret") {
10          var n = msg["val"];
11          query("#res").innerHTML = "$n";
12      }
13      else if (msg["msg"] == "log")
14          log(msg["val"]);
15  });
16
17  query("#btnStart").on.click.add((event) {
18      counterPort.send({"cmd":"start"},
19                      receiver.toSendPort());
20  });
21
22  query("#btnStop").on.click.add((event) {
23      counterPort.send({"cmd":"stop"},
24                      receiver.toSendPort());
25  });
26  }
27
28  void isolateCounter() {
29      var counter = 0;
30      var enabled = false;
31
32      port.receive((msg, reply) {
33          if (msg["cmd"] == "start" && !enabled) {
34              enabled = true;
35              port.toSendPort().send({"cmd":"inc"}, reply);
36              reply.send({"msg":"log", "val":"Isolate started!"});
37          }
38          else if (msg["cmd"] == "stop" && enabled) {
39              enabled = false;
40              reply.send({"msg":"log", "val":"Isolate stopped!"});
41          }
42          else if (msg["cmd"] == "inc") {
43              if (enabled) {
44                  counter = counter + 1;
45                  reply.send({"msg":"ret", "val":counter});
46                  port.toSendPort().send({"cmd":"inc"}, reply);

```

```
47     }  
48   }  
49   });  
50 }
```

---

Listing 2.25: Dart counter

### 2.4.3 Other technologies

There are several other proposals that intend to extend JavaScript with concurrency support. One of these is River Trail<sup>23</sup> developed since 2011 by Intel that aims to enhance JavaScript with data parallelism support. It's strongly based on the map-reduce programming model, providing *ParallelArrays* as immutable data structures processed by five main functions that are *map()*, *reduce()*, *scan()*, *filter()*, *scatter()*. To date only available as prototype through Mozilla Firefox extension, it's proposed for inclusion in the next ECMAScript standard.

## 2.5 A case study Web-app

For the purpose to apply technologies seen so far to a practical case, and even to compare these technologies with the programming model that we will introduce in Chapter 4 we designed a non-trivial demo Web application as case study, more complex than the examples seen so far. This application should be representative of all those 2.0 Web applications which, thanks to AJAX, interact dynamically and in asynchronous way with Web services and need to exploit concurrency for both performances and responsiveness matters. Moreover the term Web applications are meant here in the sense of single-paged Rich Internet Applications (RIAs) as described in the introduction.

Aim of the case study application is to draw up a ranking of what your Facebook<sup>24</sup> friends most likes. Given that the amount of friends may be high,

---

<sup>23</sup><https://github.com/RiverTrail/RiverTrail>

<sup>24</sup><http://www.facebook.com>

and therefore the crawling operation may take long time, the partial top-ten ranking must be updated in real time while the crawling operation goes by and, in addition, the user must be able to pause the process and resume it at a later time. For this purpose the application will have to connect to Facebook’s RESTfull services through the Graph API <sup>25</sup> in order to query the friends list and informations about likes for each friend. The total number of AJAX requests will be then given by  $\#REQUESTS = \#FRIENDS + 1$ . Since in almost all browsers there are limitations about the maximum number of asynchronous AJAX requests alive is inconceivable to structure the whole application in terms of asynchronous programming. A better approach could involve the use of several Web actors (HTML5 Web Workers or Isolates in Dart) each one with its responsibilities and tasks. In this direction, the architecture in Figure 2.5 is proposed, where the application business logic is mainly splitted between the view actor, responsible for the page displaying and the interaction with the user and the controller actor responsible for the out-and-out computation.

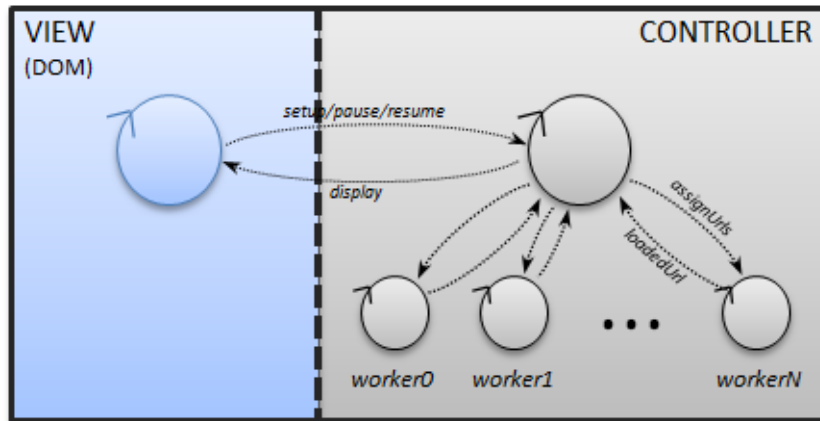


Figure 2.5: Case study Web-app architecture

Then, on the controller side, the friends crawling task can be parallelized splitting the whole work between a pool of sub-workers, according to a master-









<sup>25</sup><http://developers.facebook.com/docs/reference/api/>

worker architecture, where each sub-worker is responsible for the crawling of a sub-set of friends. The screenshot in Figure 2.6 below shows instead the final output produced by the application. The Web page, which represents the application's GUI, is built starting from the HTML template shown in Listing A.2, and it's used by all the implementations that we will see in the following.

## Facebook top likes

Find what your friends most like...

Pause 626/626 friends crawled!

1		<a href="#">Sabato Travolta</a> 167 x 
2		<a href="#">Giuseppe Giacobazzi</a> 161 x 
3		<a href="#">La Lessa Standard</a> 138 x 
4		<a href="#">Paolo Bitta</a> 120 x 
5		<a href="#">Romagna</a> 118 x 
6		<a href="#">valentino rossi (the doctor)</a> 103 x 
7		<a href="#">5 minuti e arrivo!</a> 101 x 
8		<a href="#">CESENA</a> 95 x 
9		<a href="#">Will Smith</a> 92 x 
10		<a href="#">Dialecto romagnolo</a> 89 x 

Log:  
Controller paused!  
Controller resumed!

Figure 2.6: Case study Web-app output

We implemented the case study application described above in JavaScript, since the Dart implementation should be very similar. In the JavaScript implementation (see Appendix A.2) Dedicated workers are used to implement the architecture in Figure 2.5. In particular the main UI worker (Listing A.3) is responsible for the interaction with the DOM and for spawning the controller worker (Listing A.5), responsible instead for the out-and-out application business logic, by using a pool of sub-workers (Listing A.6).

## 2.6 Open issues in reference technologies

We would conclude this chapter resuming some considerations about problems underlying the reference technologies. Mainly these problems concerns asynchronous programming, which is the reference programming model for (client-side) Web technologies, and the actor model that is the way in which these technologies provide concurrency.

### 2.6.1 Asynchronous spaghetti

As seen in section 2.3 one of the major drawbacks in asynchronous programming is the so-called *Asynchronous spaghetti*, whereby the control flow logic is fragmented among many event handlers in a way like *GOTO* statements did before the coming of structured programming. In JavaScript, and other reference Web technologies, this is mainly perceived in terms of callbacks nesting in association with anonymous functions, that are used in nesting, with asynchronous functions such timing and AJAX ones. This is proved to be an error prone approach (see the example in Listing 2.15) and in general nest properly callbacks to achieve the workflow desired is a programming discipline and it is not supported by the language itself. Promises/Futures are proposed in order to address the *Asynchronous spaghetti* problem. Through them, especially thanks to *pipe()/chain()* functions it's possible to configure workflows, made of chained asynchronous tasks, that allows to bypass the callback nesting. As already seen in most cases it's possible to unnest callbacks through a Promises/Futures pipeline (see Listings 2.13 2.12). However, this is not al-



ways possible, especially with dynamic workflows, that is when is not possible to pipelining tasks a priori since running or not a task is determined by the outcome of a previous task, in these cases nesting still the only way. Consider for example a workflow, more complex than those seen before, represented by the UML Activity Diagram in Figure 2.7 where the execution of sequences  $T2 \rightarrow ReqC \rightarrow T3$  or  $ReqD \rightarrow T4$  are conditioned by the outcome of task  $T1$ .

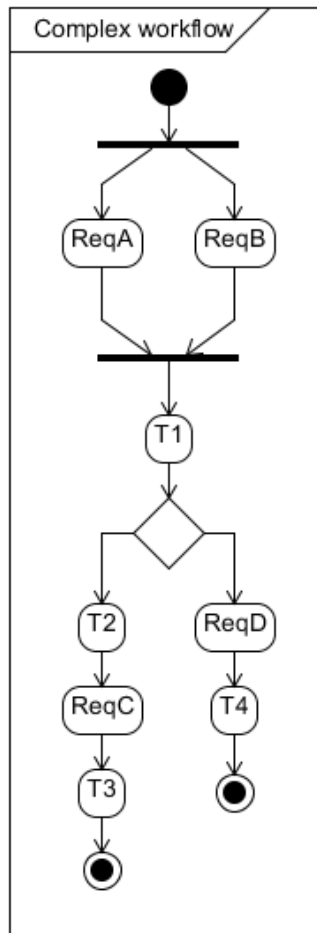


Figure 2.7: UML Activity diagram of a complex workflow

In this case, also the cleanest implementation (see the JavaScript snippet in Listing 2.26) has at least one level of nesting, at the completion of task  $T1$ .

---

```

1  var futureA = $.ajax({
2      url: 'http://example.com/serviceA/' + reqA,
3      async: true
4  })
5  var futureB = $.ajax({
6      url: 'http://example.com/serviceB/' + reqB,
7      async: true
8  });
9
10 $.when(futureA, futureB)
11     .pipe(function (respA, respB) {
12         var def = new $.Deferred();
13         var outT1 = computeT1(respA, respB);
14         def.resolve(outT1);
15         return def.promise();
16     })
17     .done(function (outT1) {
18         if (check(outT1)) {
19             computeT2();
20             $.ajax({
21                 url: 'http://example.com/serviceC/' + reqC,
22                 async: true
23             })
24             .done(function (respC) {
25                 computeT3(respC);
26             })
27         }
28         else {
29             $.ajax({
30                 url: 'http://example.com/serviceD/' + reqD,
31                 async: true
32             })
33             .done(function (respD) {
34                 computeT4(respD);
35             })
36         }
37     });

```

---

Listing 2.26: Complex workflow JavaScript implementation

Moreover, a further issue related to Promises/Futures and asynchronous

workflows concerns the execution model. In fact when coding a Promises/-Futures pipeline, such as the one in the example above, configuration and execution are mixed together, so it becomes really difficult to understand how the workflow is effectively executed, that means which operations are carried out and at what time.

### **2.6.2 Asynchronous programming and Inversion of Control**

Besides *Asynchronous spaghetti* another more general issue that underlies the event-driven programming paradigm, the so-called *Inversion of Control*. As stated in [12], instead of calling blocking operations (typical examples from our Web context are waiting for user inputs, or for asynchronous requests completion), a program merely registers its interest to be resumed on the occurrence of certain events, through event handlers (callbacks) that are installed in the execution environment and called when the events occur. Anyway, the program never calls these event handlers itself, instead the execution environment dispatches events to the installed handlers. Thus, control over the execution of program logic results inverted.

### **2.6.3 Issues inherent the Actor model**

As seen in section 2.4 both HTML5 Web Workers and Dart Isolates, that is the common mechanisms whereby reference technologies provides concurrency, are based on the Actor model. Actors, as well as objects, are purely reactive entities, they do something only in reaction to the reception of a message, furthermore they can not block or do long-term computation otherwise the actor lose responsiveness to new messages. Being purely reactive entities, actors do not provide native means to effectively integrate also proactive behaviours. In contrast to actors there are purely proactive, or autonomous, entities represented for instance by threads or processes. Problems typical arise when trying to integrate autonomous and reactive behaviours [21]. In this regard as typical example can be considered the “Stoppable Counter” application de-

scribed in section 2.4 where a Web actor has to repeatedly increment a value and, at the same time, be responsive to the stop message. As seen in all the implementations of this demo application, a typical workaround to integrate autonomous behaviour in actors, without sacrificing reactivity, is to fragment the long-term tasks, interleaving them with self-sended messages. However this solution suffer of a weak abstraction and modularity [21].

# Chapter 3

## The simpAL language

Due to the coming of multi-core and many-core platforms, and the fundamental turn of software toward concurrency and distribution, many frameworks and libraries are developed on top of the mainstream languages and technologies in order to fully exploit the hardware resources. However, the underlying programming paradigm is still the Object-oriented one, or a few little extensions, see for instance the Actor-model mentioned in the previous chapter. Conversely, the simpAL language, introduces a further abstraction layer to deal with concurrent and distributed systems design and development, based on the Human-inspired computing metaphor and the Agent-oriented programming paradigm.

Agent-oriented programming has been introduced in the (Distributed) Artificial Intelligence context, and since then many agent and multi-agent programming languages have been proposed in literature. The focus of these works have been mainly on architectures, theories, languages to program agents and multi-agent systems in the (D)AI context, with the purpose of find appropriate computation models and architectures to design intelligent software entities exhibiting some level of autonomy in achieving complex goals. The simpAL language is instead deeply different, and is intended to investigate Agent-oriented programming as a general-purpose programming approach, as evolution of the Object-oriented and Actor based ones, focusing on the design and implementation of concurrent and possibly distributed systems. simpAL

has been designed from scratch with software development in mind, for the purpose to bring robustness, usability and flexibility from mainstream programming languages to Agent-oriented ones.

simpAL is a strongly-typed programming language, and extends a pure Object-oriented layer in the sense that agents and artifacts are meant to be used as coarse grain abstractions to define the shape of the organization, in particular of the control part of it, while everything that concerns data structures representation and purely transformational computation is demanded to the Object-oriented programming layer, in particular to a subset of Java.

In the remainder of this chapter we will take a concise survey about simpAL, focusing in particular to those aspects related to the objectives of this thesis. We first outline simpAL main concepts, and then take a deeper look on agents and artifacts programming, we will conclude taking some considerations about how simpAL can improve the Web programming, especially for what concerns concurrency and asynchronous programming. Most of the material and images are taken from [19] [20] [21] [22], so for more informations please refer to these papers and to the simpAL SourceForge project<sup>1</sup>.

### 3.1 Main concepts

The simpAL language and infrastructure draw inspiration from Human-inspired computing and the Activity theory, and integrates elements from the the Agents and Artifacts (A&A) conceptual model [18]. In particular human organizations are taken as natural high-level metaphor to define the structure and behavior of (complex) programs, where articulated concurrent and coordinated activities take place, distributed in time and space [20]. The members of simpAL organizations are called *agents* and, similarly to humans, they are in charge of performing autonomously some tasks eventually interacting with other agents and with the environment in which they are situated. Autonomously means in this case that, given a task to do, they pro-actively decide what are the best actions to perform and when to do them, promptly reacting

---

<sup>1</sup><http://sourceforge.net/projects/simpal/>

to relevant events from their environment and fully encapsulating the control of their behavior. The other main abstraction, beside agents, is the environment in which agents are situated. The environment in human organizations plays a key role, as the context mediating and then supporting members individual and cooperative tasks, through the use of shared tools and resources. So if agents are the abstraction meant to model active, task-oriented, autonomous behaviors, *artifacts* on the other side are meant to be effective for modeling non-autonomous components encapsulating and modularizing functionalities that can be suitably exploited by agents [20]. Agents can use environment artifacts in the sense that they can request operations provided by an artifact or perceive its observable properties. Like artifacts in the human case, artifacts in simpAL can be dynamically created and disposed by agents, and eventually can be designed to be composed, for the purpose to create complex artifacts by connecting simpler ones. As mentioned before, like in the human case, agents can communicate directly with other agents through asynchronous message passing, in an Actor-like fashion. Alternatively artifacts can be useful to support indirect forms of communication and coordination, for instance through blackboards or tuple spaces. At last a further abstraction is needed to represent the overall structure and topology of a simpAL program, which can be distributed, the so-called *workspaces*. A workspace is a logical container for agents and artifacts, possibly running on a different node of the network. The set of all the workspaces is called *organization* and represents the structure of a simpAL program as a whole. Summarizing, in Figure 3.1 is represented the overall structure of a simpAL program. The organization contains a set of possibly distributed workspaces which in turn contain agents and artifacts. Agents can *use* or *observe* artifacts in its own workspace or in a different one, or directly communicate with other agents.

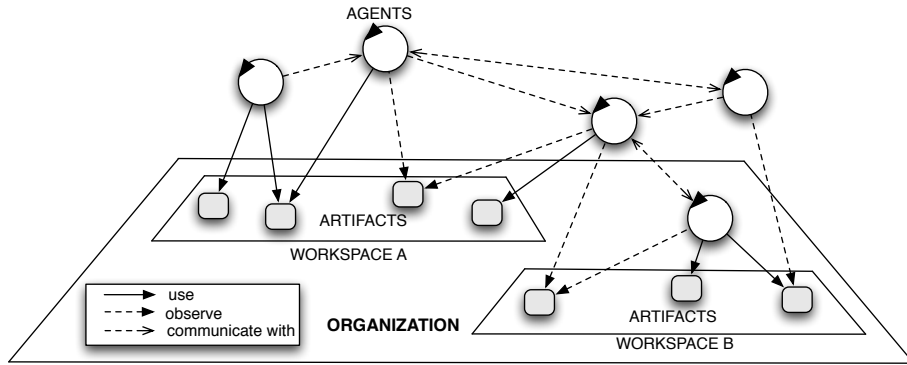


Figure 3.1: simpAL overview

For the purpose to bridge the abstraction gap between design and programming and to tackle the development of complex software systems, in a model-driven perspective, the basic idea behind simpAL is to keep these high-level abstractions alive from design time to runtime.

## 3.2 Programming agents

Agents are the simpAL main abstraction, used to model those parts of the program that are in charge of performing autonomously some tasks eventually interacting with other agents and with the environment where they are situated. The agent abstraction mainly includes the following concepts: *tasks*, to represent the description of the jobs that agents have to do; *plans*, encapsulating the procedural knowledge needed to accomplish the tasks; *actions*, which are the moves that agents can do, depending on the environment in which they are situated, in order to carry on their tasks; *percepts*, that are the events that agents asynchronously observe from the environment to which they may need to react, in order to do such jobs [21]. According to the engineering principle of separation between interface and implementation, the agent programming in simpAL involves on the one hand the introduction of *roles*, which type agents and describe their interface in terms of *tasks* that agents are able to do. On the other hand *agent scripts* contain the actual implementation for the tasks in the roles they declare to play, through *plans*. In the following we analyze



more in detail these concepts.

### 3.2.1 Roles and tasks

The notion of role explicitly defines the type of a simpAL agent as the set of possible types of tasks that any agent playing that role is able to do. As interfaces in Object-oriented programming, roles define the agent's contract with respect to the environment where it is immersed. A role is identified by a name and groups the definition of the set of task types [22]. Tasks, as we will see in the following, are first-class concepts in simpAL, they are typed too and intended to describe a well-defined unit of work to be done. Each task is identified by a name and contains a set of predefined optional attributes, useful to fully describe the task itself. These attributes allow for example to specify the task input/output arguments (*input-params* and *output-params* blocks), or to constrain the agent interactions with other agents while the task is running (*understand* and *talks-about* blocks). Agents typing, through roles, is particularly useful for several purposes, for instance it allows error checking at compile time, or in the future it will allow to define the well-known notions of inheritance and polymorphism for agents too. The example in Listing 3.1 shows the role specification for an *InteractiveCounter* agent, an entity who repeatedly increment a counter, which has capable of performing the tasks *Boot* and *Increment*. Specifying the input parameters for the latter task it's possible to set the increment value besides a threshold value at which the agent must stop increment. While performing the *Increment* task other agents can send messages to the agent which implements this role for the purpose to *start*, *stop* or *pause* it.

---

```
1  role InteractiveCounterRole {
2      task Boot {
3      }
4
5      task Increment {
6          input-params {
7              incrementValue: int;
```

```

8         maxValue: int;
9     }
10
11     understands {
12         start: boolean;
13         stop: boolean;
14         reset: boolean;
15     }
16 }
17 }

```

---

Listing 3.1: Definition of the *InteractiveCounter* role in simpAL

### 3.2.2 Agent scripts and plans

While roles provide agent interfaces in terms of tasks, expressing what an agent have to do, agent scripts on the counterpart specify how to fulfill tasks, through the so called plans. Again if roles and tasks are intended to specify the structure of agents, scripts and plans are responsible to define the agents behaviour. Scripts therefore represent modules of agent behavior which contain the set of plans corresponding to tasks in the roles that they declare to play, and a set of beliefs that can be accessed by all the plans declared into that script. Beliefs represent the knowledge owned by agents, similarly to variables in mainstream programming languages they are declared specifying a name, the type of the information they represent and optionally an initial value. For the same task an agent script can provide several plans which can be used in different contexts, on the basis of the agent belief base. By loading a script, an agent adds to its belief base the beliefs declared in the script and the plans of the script to its plan library [21]. Analogously to classes in Object-oriented programming, scripts provide a form of encapsulation and information hiding in agent implementation: they bundle together beliefs as informational state and plans as procedural knowledge, making beliefs only accessible internally to plans. Plans provide instead an explicit mechanism to modularize agent overall behavior [19].

### 3.2.2.1 Action rules: events, conditions, actions

The procedural knowledge provided by plans is expressed in form of *action rules* blocks. Action rules are similar to Event-Condition-Action (ECA) rules in which the specified action  $A$  is performed when the related event  $E$  occurs and the condition  $C$  holds. Action rule blocks, denoted by  $\{ \dots \}$ , contain a set of action rules and eventually other nested action rule blocks. Attributes can be assigned to action rule blocks for the purpose to specify behavioural properties. Among these the *completed-when* attribute specifies a condition that determines the block completion, *atomically* specifies that the action rule block must be executed without being interrupted or interleaved with blocks of other plans in execution, and then *using* attribute, used to specify a set of artifacts used or observed inside the block. At runtime, when entering a block where an artifact is used, automatically the observable properties of the artifact are continuously perceived and their values is stored in corresponding beliefs in the agent belief base, updated in the sense stage of the agent control loop [20]. An action rule block is considered completed as soon as there are no more action rules that can be triggered or the condition specified in the *completed-when* attribute is met.

Events are modeled as changes of some beliefs which belong to the agent belief base. These events mainly concern percepts received from the environment, messages received by other agents, notification of completion or failure of labelled actions and the passing of time. For example it's possible to react on the changing of an observable property on an artifact in use through the syntax *changed*  $\langle Belief \rangle$ , or react to a message received from another agent through the syntax *told*  $\langle Belief \rangle$ . In both cases these beliefs are dynamical in the sense that the belief on the agent side is automatically added to the belief base when is needed, for instance when an artifact is used, or a task that declare the understand attribute is implemented. Through the keywords *when* and *every-time* it's possible to specify in addition to react respectively exactly once, when the event first occurs, or every time it occurs. Conditions are instead simply specified in terms of boolean expressions over agent beliefs.

Actions in action rules can be either *internal*, if they affect the internal state

of the agent, or *external* if they affect the environment in which the agent is situated or concern the communication with other agents. Internal actions typically regard the definition and assignment of beliefs, the manipulation of Java objects and the invocation of methods over them. Also control structure such as *if*, *for* and *while* constructs are provided as internal actions. External actions can be instead operations requested to artifacts, artifacts creation or disposal, agents spawning or termination, communicative actions toward other agents, through the *tell* keyword, or task assignment to an existing agent, through the *do-task* keyword. As stated before action rule blocks are the basic construct for specifying the procedural knowledge in plans. By their nature, if more than one action rule blocks are provided for a plan, these are all activated simultaneously. For this reason simpAL provides syntactic sugar to ease the coding of sequences of actions, if two action rule blocks are separated by a semicolon the predefined meaning is that the second block can be selected only after that the first is completed. Furthermore if event and condition are not specified the action by default can be selected and executed immediately, and only once.

The snippet in Listing 3.2 shows an agent script which implements the *InteractiveCounterRole* role specified in Listing 3.1. The agent script provides two plans, one for each task defined in the role. The first plan contains only one action, with no event and condition, which requests the *print()* operation to the artifact *console* in the workspace *main*. We will tackle the artifact programming in more detail in the next sections. The *Increment* plan specify instead the main behaviour of the agent: first the agent waits until any other agent send him the *start* message (line 10). Once received this message the action rules block at lines 14-22 is activated, so the agent starts the increase activity, until it will receive a *stop* message, or the counter exceeds the threshold value, and the *completed-when* condition is met (lines 12-13). In the increase activity the agent repeatedly request the *inc()* operation to the counter artifact and then prints the updated value on the *console* artifact. Notice that thanks to the “;” syntactic sugar the *print()* operation is requested only when the action before, that is the *inc()* operation, is completed. At

the same time that the *repeatedly* block is performed, if a *reset* message is received the *counter* is reset thanks to the *every-time* block at line 20.

---

```
1 agent-script InteractiveCounterAgent
2   implements InteractiveCounterRole
3   in ExampleOrgModel
4   {
5     plan-for Boot using: console@main {
6       print(msg: "InteractiveCounter is booting...")
7     }
8
9     plan-for Increment {
10      every-time told this-task.start =>
11        using: counter@main
12        completed-when: this-task.stop
13          || value >= this-task.maxValue
14      {
15        repeatedly using: console@main {
16          inc(qty: this-task.incrementValue);
17          print(msg: "" + value)
18        }
19
20        every-time told this-task.reset =>
21          reset()
22      }
23    }
24  }
```

---

Listing 3.2: Definition of the *InteractiveCounter* agent script in simpAL

### 3.2.2.2 Tasks as first-class entities

As stated before, differently from mainstream languages, simpAL provides tasks as first-class entities. A rich set of built-in internal actions are in fact provided for the purpose to directly manipulate tasks and intentions. We have just seen the *do-task* action whereby it's possible to assign a task to an agent, similarly there are actions to drop, suspend or resume tasks. Through

*is-ongoing* or *is-done* is instead possible to check if the task corresponding to a belief is completed or still executing. Again the *drop-all-tasks* action drops all the ongoing tasks and the related intentions, but the current one, while the *forget-all-plans* removes from the intentions stack all the action rule blocks except the one at top level. The support of tasks as first-class entities is particular usefull for the purpose to structure complex plans, splitting the whole business logic of a complex task among many sub-plans, according to the modularity principle.

### 3.2.3 The agent control loop

The agent control architecture, or agent control loop, is a key aspect in sim-pAL and it's inspired by the reasoning cycle of Belief–Desire–Intention (BDI) architectures, in particular it can be regarded as a simplification of AgentSpeak and Jason [4] ones. It is the foundation for integration of autonomous and reactive behaviours and, as you can notice in Figure 3.3, it's characterized by three conceptual stages which are repeatedly performed: *sense*, *plan*, *act*.

- *Sense stage*

In the sense stage the belief base of an agent is updated by processing events from the *external event queue*, that is those ones coming from the environment and from other agents, if any are available. These events are related to changes of the observable properties in artifacts currently used, incoming messages from other agents, or again notifications of success or failure for previously executed actions. As seen in the previous sections these events are all modeled as changes in the so-called agent's dynamical beliefs. Changes in agent's beliefs caused by external events are in turn modeled as events, enqueued to the *internal event queue*, which will be accessed in the next stage.

- *Plan stage*

Many tasks can be carried on simultaneously by the agent, and for each ongoing task there is a stack of plans in execution (intentions), since the execution of a plan may involve the execution of many sub-plans.

---

```

1: while true do
2:                                     ▷ SENSE stage
3:   if external events queue not empty then
4:     ev ← PICKEXTEVENT()
5:     UPDATEAGENTSTATE(ev)
6:   end if
7:                                     ▷ PLAN stage
8:   if new tasks todo then
9:     for each new task to-do task do
10:      plan ← SELECTPLAN(task, belBase, planLib)
11:      CREATENEWINTENTION(plan, task)
12:    end for
13:   end if
14:   actList ← []
15:   for each ongoing intention i do
16:     if TASKFULFILLED(i) then
17:       DROPINTENTION(i)
18:     end if
19:     actList ← actList + SELECTACTIONS(i, belBase)
20:   end for
21:                                     ▷ ACT stage
22:   for each action act in actList do
23:     EXECUTE(act)
24:   end for
25: end while

```

---

Figure 3.2: simpAL control loop pseudocode

So, first of all for each new task to do a plan is selected from the agent plan-library and a new intention is instantiated. Main purpose of this plan stage is to focus on a task, through a certain scheduling policy, and to select all the actions amenable to be performed between the action rules currently available on the top of the intentions stack. Intentions that have achieved their goal are dropped.

- *Act stage*

Objective of this act stage is finally to perform all the actions selected in the plan stage. Internal actions are executed atomically in this stage while external ones, which typically correspond to operations over artifacts are simply started. The outcome of these actions, success or failure,

will be possibly perceived as an asynchronous event in one next future cycle.

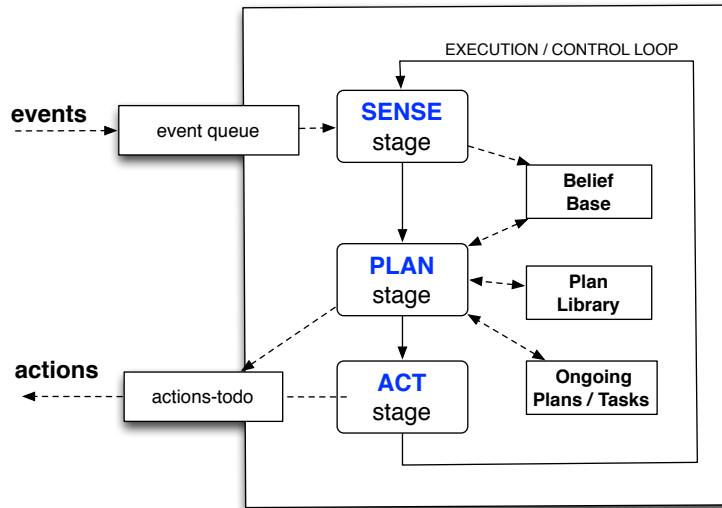


Figure 3.3: simpAL agent architecture

Notice that continuously executing these three stages, conceptually, the agent control flow is never blocked [19]. Moreover the agent control loop can be framed as a fine-grained extension of the actor event loop (Figure 2.3), the main differences are that agents can cycle even if there are no external events to process and intentions are not meant to be fully executed and completed in one cycle. [21].

### 3.3 Programming artifact-based environments

As stated before the environment is constituted by a dynamic set of artifacts, grouped in possibly distributed workspaces. Artifacts can be conceived as passive modules, just like classes or monitors in Object-oriented programming. Each artifact provides a set of observable properties, that the agents using it may perceive, and a set of operations, requested by agents through actions. Analogously to the agents case, also for artifacts the definition of



artifact functionalities and their concrete implementations are kept separate. Therefore on the one hand we have artifact models, which define artifact *usage interfaces*, while on the other *artifact templates*, which provide the concrete implementations for these models.

### 3.3.1 Usage interfaces

Usage interfaces are identified by a name and characterize an artifact defining both a set of observable properties and a set of operation signatures. Observable properties, similarly to variables, are declared through a name, a type and a value. Operations may accept keyword-based parameters, so when an agent requests an operation with parameters it must specify them in form of *keyword : value*, in any order. Many of the operation parameters may be declared as action feedback (adding the *#out* keyword to the parameter definition<sup>2</sup>), that is computed by the operation and returned back to the agent when the operation is completed, just like output parameters. The snippet in Listing 3.3 shows the interface definition for the *Counter* artifact, used in the previous examples. Notice the declaration of the observable property *value* and the artifact operations, with their parameters.

---

```
1 usage-interface CounterInterface {
2     obs-prop value: int;
3
4     operation inc(qty: int);
5     operation reset();
6 }
```

---

Listing 3.3: Definition of the Counter artifact interface in simpAL

### 3.3.2 Artifact templates

Artifact templates provides instead the implementation for artifact operations, and they can be used to create instances of artifacts, just like classes in Object-

---

<sup>2</sup>e.g. *operation get(item : int #out)*

oriented programming. Operations are implemented simply through sequences of statements, in pure imperative style, using classic control structures and eventually Java objects in addition to primitive values. Artifact templates also provide a form of encapsulation, since they can define some hidden (not observable) state variable and internal operations useful for implementing artifact functionalities. Specific primitives are provided for the purpose to shape the operation execution, for instance the *await* statement allows to suspend the operation until the specified condition is met without blocking the artifact, that is allowing other operations to be executed. The operation execution semantic in simpAL artifacts is similar to monitor procedures ones since only one operation can be in execution at a certain time and, if many suspended operation can be resumed, one has to be selected. The snippet in Listing 3.4 shows the concrete implementation of the *Counter* artifact template, which implements the *CounterInterface* specified in Listing 3.3. Notice the implementation of the operations as simple sequence of statements in pure imperative style besides the *init* operation which must be provided by each artifact template.

---

```
1 artifact CounterArtifact
2     implements CounterInterface
3 {
4     init (initValue: int) {
5         value = initValue;
6     }
7
8     operation inc(qty: int) {
9         value = value + qty;
10    }
11
12    operation reset() {
13        value = 0;
14    }
15 }
```

---

Listing 3.4: Definition of the Counter artifact template in simpAL

## 3.4 Defining the organization

The organization abstraction in simpAL represents the program overall structure, also for this aspect the model and implementation are kept separate. The *organization model* specifies the set of workspaces that constitute the program. Each workspace is identified by a name and contains the sets of all roles and usage interfaces related to the agents and artifacts respectively which will be part of the workspace itself. The snippet in Listing 3.5 shows the definition of the organization model for the example program described in the previous sections. The *ExampleOrgModel* is declared to contain the *main* workspace which in turn contains the *incrementerAgent* agent besides the *counter* and *console* artifacts. Notice that the organization model describes the entities contained by workspaces in term of interfaces, that is roles and usage interfaces.

---

```
1  org-model ExampleOrgModel {
2      workspace main {
3          counter: CounterInterface
4          console: ConsoleInterface
5
6          incrementerAgent: InteractiveCounterRole
7      }
8  }
```

---

Listing 3.5: Definition of the Example organization model in simpAL

The *organization* defines instead the concrete implementation of the organization model. It specifies for each workspace the initial set of agents and artifacts. The initial set means that agents can be dynamical spawned and terminated by other agents as artifacts can be dynamical created and disposed at run time. For agents it is specified the agent script to load and possibly the initial task to perform besides a set of configuration parameters for the agent script. For artifacts the template is instead specified possibly including some initialization parameters. At last through a configuration file it's possi-

ble to specify further deployment informations, for instance Internet addresses in order to distribute the workspaces among an IP network. The snippet in Listing 3.6 shows at last the concrete definition of the example organization, implementing the organization model specified in Listing 3.5. Differently from the organization model the organization defines the effective entities which compose the simpAL program in terms of agents, for which are provided the loading script and the initial task, and artifacts for which are instead provided the template and proper initialization parameters.

---

```
1 org ExampleOrg implements ExampleOrgModel {
2     workspace main {
3         counter = CounterArtifact(initValue: 0)
4         console = ConsoleArtifact()
5
6         incrementerAgent = InteractiveCounterAgent()
7             init-task: Boot()
8     }
9 }
```

---

Listing 3.6: Definition of the Example concrete organization in simpAL

## 3.5 simpAL benefits

In this conclusive section we take some considerations about benefits that simpAL introduce, in particular regarding to concurrent and event-driven Web programming, objective of this thesis.

### 3.5.1 Asynchronous programming without Inversion of Control

As seen in the previous chapter a typical problem in asynchronous programming concerns the so-called *Inversion of Control* that occurs because event-handlers (callbacks), registered by the program main control flow, are never

directly called by the main control flow itself but instead invoked by the execution environment when the related events occurs. In client-side Web programming, JavaScript in particular, due to the combination of callbacks and anonymous functions, *Inversion of Control* is the ordinary. In simpAL instead there is no *Inversion of Control* since event handlers are modeled directly by action rules, which are evaluated and possibly selected and related actions executed by the same logical control flow, that is the agent control loop.

### 3.5.2 Integration of autonomous and reactive behaviours

One of the biggest advantages given by Agent-oriented programming and simpAL in particular is definitely the ability to integrate autonomous and reactive behaviours [21]. This is also one of the most interesting features with regard to the aims of this thesis. HTML5 Web Workers and Dart Isolates, that is the mechanisms in which the current generation of Web technologies supports concurrency are purely reactive and based on the Actor model. From actors they inherit the limitations too, in particular for what concerns the implementation of autonomous behaviours, as we have seen in the previous chapter. Thanks to their architecture, which is in turn inspired by the Belief-Desire-Intention one, agents, differently from actors as well as objects, are not based on the reactivity principle and they do something not because they receive a message, but because they have some tasks to do.

### 3.5.3 Concurrency

One of the major motivations behind simpAL is to provide a support for developing concurrent and distributed programs, abstracting from low-level mechanisms such as those in multi-threaded programming. So in a simpAL program, agents are executed concurrently, as well as operations on distinct artifacts; also, for the same agent, the execution of an external action (operation) is asynchronous with respect to the execution of the agent cycle [21]. Furthermore simpAL, like certain Actor-based languages and frameworks provide concurrency at a logical level, in the sense that the runtime platform is

responsible for effectively map these concurrent entities and activities on physical OS threads according to the resources available in the system. This makes it possible to have a large number of agents running concurrently, keeping a good degree of scalability. This approach may be advantageous in Web programming compared to the existing state-of-art technologies which commonly provide concurrency through Actor-based heavy-weight entities intended to be used in a small number, consider for instance HTML5 Web Workers analyzed in the previous chapter. Another great benefit, but which will not deepen because out of the scope of this thesis, is the fully transparency of simpAL programs with respect to distribution.

### **3.5.4 Error checking at compile time**

As seen in the previous chapter also typing is a very useful feature, almost essential in languages targeted to in-the-large programming. simpAL strong type system allows among other to detect errors at compile time such as incompatible assignments, referencing of non existing symbols, redefinition of symbols in the same context, and so on. Generally error checking in simpAL may be useful to check the compliance between model definitions and counterpart implementations, for instance to check if an agent script which implements a certain role provides at least a plan for each task defined in the role. Moreover the concept of role helps to inspect the behavior of agents aiming at interacting with other agents implementing a certain role, checking that they would request the accomplishment of only those tasks that are specified by the role, and that they would send only those messages enlisted in the role [20]. Usage interfaces are instead helpful to check the correspondence between agents actions and artifacts operations as well as errors concerning observable properties and beliefs whereby agents perceive them.

# Chapter 4

## **simpAL-web**

Once introduced the simpAL language and outlined the features that may be useful in Web programming, in this chapter we start focusing on how to conceive a Web application under the new Agent-oriented programming paradigm exploiting the abstractions provided by the simpAL language. We outline a basic structure for simpAL Web applications in terms of agents and artifacts and then we discuss about the design and implementation of the simpAL-web platform, that is the foundational layer that allows simpAL to become a Web client-side programming language. It should provide features to handle Web stuff such as HTML Web documents, AJAX requests and so on. The simpAL-web platform represents in fact the basic environment of a simpAL Web program. Key component of this platform will be a custom Web browser, in charge of displaying the Web documents and managing the interaction with the user. Once designed the simpAL-web platform, we will be able to try simpAL in Web programming context, in particular compared to some examples taken from Chapter 2. We will conclude taking some consideration about those benefits and innovations that simpAL, and the simpAL-web platform, bring in design and development of Web applications.

## 4.1 Requirements and assumptions

Since the ultimate objective of the simpAL-web platform is now to enable simpAL to be tried in Web programming context, what we would design and implement will be a prototype platform, provided with a subset of all the features available to current Web applications through the DOM APIs and other Web standards. For this purpose we make some assumptions and we narrow the programming of simpAL Web applications to the following aspects:

- *Single-page Web applications* — As stated in [29] “is a Web application that fits on a single Web page with the goal of providing a more fluid user experience akin to a desktop application. Either all necessary code is retrieved with a single page load, or partial changes are performed loading new code on demand from the Web server, usually driven by user actions. The page does not automatically reload during user interaction with the application, nor does control transfer to another page. Updates to the displayed page may or may not involve interaction with a server”. This means that the simpAL Web program acts on a single Web page, represented by an HTML document loaded when the application starts.
- *Elements accessible by id* — All the HTML elements inside the page on which the simpAL Web program may operate must be marked with a unique identifier, specified through the *id* HTML attribute.
- *Perceive certain events from certain elements* — It means that Web agents should be able to perceive only those events they are interested in, and for certain elements only.
- *Access to element’s content and attributes* — Finally, Web agents should be able to read and update the content and attributes of a certain HTML element.



## 4.2 Modeling simpAL Web applications

Once defined the features of interest in Web applications for the objectives of our experimentations we now focus on how to model such applications in simpAL. Differently for standard Web languages, such as JavaScript, where the business logic of the application is flat and eventually spreaded among several script files, in simpAL we can organize such business logic by distinguishing between those active parts, that will be modeled as agents, and those passive parts which are encapsulated by artifacts and used by agents. So our simpAL Web applications will be composed at least by one agent – that we will henceforth call *Web agent(s)* – which will encapsulate the essential application business logic purged from all those aspect that concerns low level interactions which will be demanded to the Web environment. The introduction of many Web agents will be useful for instance when applying the Master-Worker pattern or in general other techniques of division of labor. A set of built-in artifacts will be provided by the simpAL platform in order to allow Web agents to easily interact with the Web page and elements, abstracting from the specific logic used to access them. Excluding these artifacts which are standard and defined by the simpAL platform, first aspect to consider during the design of a simpAL Web application is to separate functionalities between those ones that can be provided by artifacts and those others that can be provided by agents. The main aim is to simplify and clean as much as possible the programming on the Web agent side. What we propose definitively is a new computational model in which Web agents are the main entities that implement the essential business logic of the application by interacting with other Web agents and the Web environment. As mentioned before thanks to their architecture such Web agents can handle in a better way the control flow so as to make easier aspects such as asynchronous programming and concurrency. In Figure 4.1 is shown the structure of a typical simpAL Web application, anticipating the next section we can see how many Web agents cooperate together by exchanging messages and by using the artifacts which represents the Web environment. Some of these artifacts will be provided by the simpAL-web platform and will be responsible to manage the page display and the interaction with the user,

some other artifacts can be instead programmed by the application developers in order to wrap the interaction with Web services and to model passive parts of the application.

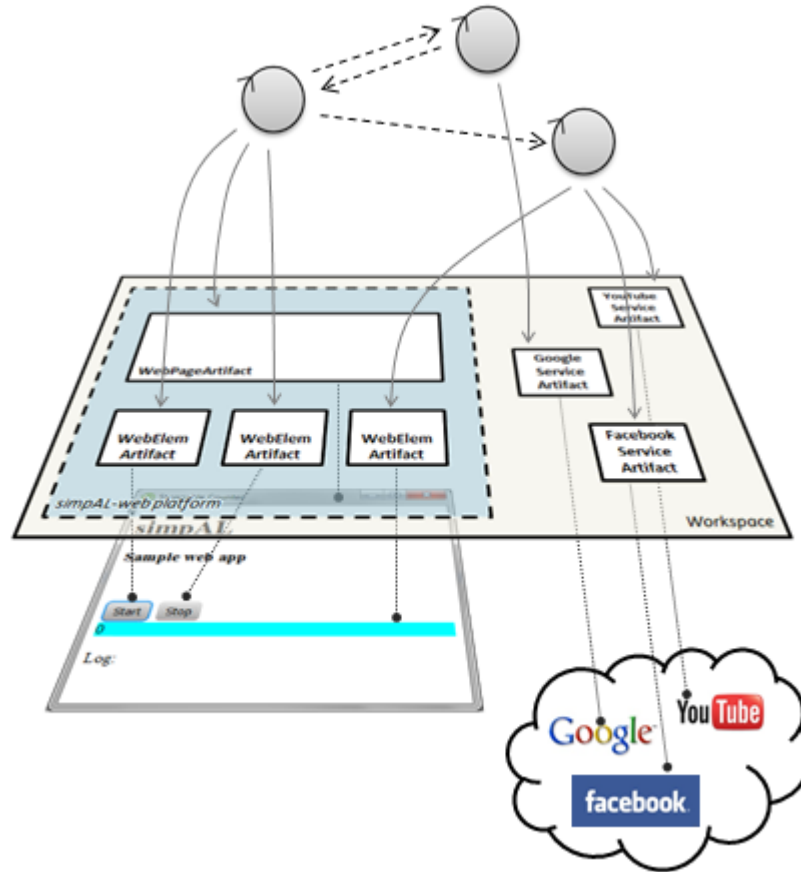


Figure 4.1: Structure of a simpAL Web application

### 4.3 Modeling the Web environment

Therefore first of all we focus on the design of the simpAL-web platform, that is the layer which interface the simpAL runtime with the GUI (simpAL-web browser) responsible to display Web documents and managing the user interactions. As stated before this platform can also be conceived as the basic environment of all simpAL Web programs. Through this layer in fact simpAL agents must be able to manipulate the Web documents and its ele-

ments, to perceive user actions on the browser and to exchange data possibly asynchronously with Web services, just like current Web 2.0 applications do. Again as the Document Object Model (DOM) APIs provides an interface for JavaScript programs, the simpAL-web platform should provide an interface for simpAL agents. However, since the DOM APIs was conceived with the Object-oriented paradigm in mind, and designed with the purpose to ease and provide in the best way dynamical access and update to the content structure and style of documents, it is not true that this is the best way in Agent-oriented programming too. In fact as mentioned in Chapter 3 with the Agent-oriented programming paradigm and simpAL we are at an higher level of abstraction and what we want is an interface that makes Web programming, on the agents side, the most simple and natural as possible. Notice that, thanks to the ability of simpAL to deal with Java objects, it would be theoretically possible to use the DOM APIs as are, anyway this is definitely not the most natural approach. What we will do instead is to rethink an interface from scratch, keeping the Agent-oriented paradigm in mind, and with the aim to make the programming of Web agents as natural as possible. It is therefore natural to model this interface in terms of artifacts, that will constitute the environment in which Web agents will live. In the design of these artifacts we focus essentially on what a Web agent must be able to do on the artifact and what it must be able to perceive from the artifact itself. Accordingly we will design the artifact in terms of operations and observable properties respectively.

### **4.3.1 The Web Page artifact**

As stated before we start modeling the Web page artifact, which represents through an HTML template the GUI for the Web application. Main purpose of this artifact is to act as a container of those HTML elements which compose the GUI and to easily provide access to these elements, roughly like the *document* object in the DOM APIs. To this end the Web page artifact can be designed by providing a proper mapping between HTML elements and corresponding artifacts in the simpAL-web environment. This can be achieved in several ways, the extreme approaches are the one in which to each HTML

element corresponds an element artifact, and the other one in which there is an only big artifact that provides operations and collects events from all the HTML elements, that is the Web page. Therefore if on the one side we have an overhead due to the handling as artifacts of also those HTML elements that are not of interest for the Web application, on the other having only one artifact involves a low level of abstraction. The solution we have chosen is in the middle since we decided to represents each HTML element with an element artifact but only for those elements which effectively affect the Web application behaviour, in the sense that they raise events which must be perceived from Web agents or are able to be manipulated by them. So the number of element artifacts in the page artifacts will always be less than or equal to the number of actual HTML elements in the document. To this end the Web page artifact should provide at least an operation that, given the identifier of the HTML elements returns as output parameter the Web element artifact that wraps it. In the *WebPageInterface* usage interface of the page artifact this corresponds to the *getElement()* operation, this may be requested at any time and by any Web agent, so the concrete implementation of the Web page artifact will be responsible to dynamically create the new element artifact and binding it to the HTML element. Furthermore it would be useful that Web agents were able to perceive events related to the page loading and closing. The issue is that simpAL currently doesn't natively provides a way to model events so, as we shall see in the following, we will have to represent events as changes of the observable properties states, and so on the agent side changes in the belief base. For this purpose the *WebPageInterface* provides the *loaded* and *closed* observable properties as booleans, which are switched to true when the page, will be loaded and closed respectively. Notice at last that the *WebPageInterface* (Listing 4.1) doesn't provide operations to specify the URL where to load the Web page since it will be indicated in *WebPageArtifact* template init parameters.

---

```
1 usage-interface WebPageInterface {
2     obs-prop loaded: boolean;
```

```

3   obs-prop closed: boolean;
4
5   operation getElement(selector: String,
6       elemArtifact: WebElemInterface #out);
7 }

```

---

Listing 4.1: Specification of the Web page artifact interface

### 4.3.2 The Web Element artifact

The Web Element artifacts as stated before represent instead the pieces of which the Web page is made of, on which the Web agents will act. It wraps an HTML element and provides among other functionalities to retrieve and update the content of the element, operations *getContent()* and *setContent()* respectively, and element attributes, through operations *getAttribute()* and *setAttribute()*. Another purpose of this element artifact is to map events from the HTML element in changes of the observable properties which may be perceived by Web agents. Taking a look at the artifact usage interface *WebElemInterface* in Listing 4.2, observable properties at lines 2-6 represent mouse related events: *clicks* and *doubleClicks* model the mouse clicks as the number of times that the mouse was clicked, or double-clicked<sup>1</sup>; *pressed* becomes true when a mouse button is pressed to the element, false when it is released; similarly *focused* is true when the cursor enters the element and false when it leaves it; *pointerCoord* instead represent the position in coordinates  $(x, y)$  of the cursor while it moves within the element. Similarly the observable properties at lines 8-9 represent events from the keyboard: *keyPressed* is true when a key is pressed on the element and becomes false as soon the key is released; *key* represents instead the Unicode corresponding to the last pressed key.

---

<sup>1</sup>Notice that *clicks* and *doubleClicks* are declared as integers since the click event is instantaneous and it's poorly modeled with changes of boolean states. On the Web agent side reacting to a click it means specify an action rule like *when changed clicks => {}* for *clicks* as integer, instead of *when changed clicked : clicked => {}* for *clicked* as boolean

---

```

1  usage-interface WebElemInterface {
2      obs-prop clicks: int;
3      obs-prop doubleClicks: int;
4      obs-prop pressed: boolean;
5      obs-prop focused: boolean;
6      obs-prop pointerCoord: simpal.web.PointerCoordinates;
7
8      obs-prop keyPressed: boolean;
9      obs-prop key: int;
10
11     operation getContent(html: String #out);
12     operation setContent(html: String);
13
14     operation getAttribute(attribute: String,
15                             value: String #out);
16     operation setAttribute(attribute: String,
17                             value: String);
18 }

```

---

Listing 4.2: Specification of the Web element artifact interface

Notice that accordingly to this approach all HTML elements are wrapped uniformly by the element artifact. This is in agreement with the DOM specification which provides a set of common methods and standard events<sup>2</sup> defined for almost every elements. However, always with the aim to simplify programming on the agent side it might be useful to have different kinds of element artifacts on the basis of the specific HTML elements. For instance we would like to have the observable property *text* for artifacts related to those elements such as text inputs and textareas, or the observable property *progress* for artifacts which wraps media elements. Therefore would be natural to model these new element artifacts as a hierarchy where for instance *WebTextElemInterface* and *WebMediaElemInterface* extend the root *WebElemInterface* inheriting its observable properties and operations and by adding others specific to the kind of HTML element they wrap. Anyway since simpAL does not support yet inheritance for agents and artifacts and these new artifacts are not

---

<sup>2</sup>[http://www.w3schools.com/jsref/dom\\_obj\\_event.asp](http://www.w3schools.com/jsref/dom_obj_event.asp)

significant for the following tests about simpAL and Web programming they will not be provided in this prototype version of the simpAL-web platform.

### 4.3.3 The Clock artifact

Timing functionalities similarly to JavaScript *setInterval()* are provided in simpAL through the built-in clock artifact. From the usage interface in Listing 4.3 you can see that are provide operations to enable and disable it besides to set its clock rate. On the agent side the passing of time can be perceived through the *time* observable property which is updated by the clock artifact at the pre-defined rate. The bottom of Listing 4.3 (lines 11-20) shows the snippet of a simple application which periodically displays on the Web page the current time. Notice that while in JavaScript *setInterval()* and *setTimeout()* functions are often used to emulate concurrency besides to tricks on the event-loop, in simpAL Web applications the clock artifact is intended to be used for timing purposes only.

---

```
1  usage-interface ClockInterface {
2      obs-prop time: long;
3
4      operation switchOn();
5      operation switchOff();
6      operation setRate(rate: int);
7  }
8
9  ...
10
11 using: clock@main {
12     dateFormat : java.text.DateFormat
13         = new java.text.SimpleDateFormat("HH:mm:ss")
14
15     switchOn()
16
17     every-time changed time => using: divOut {
18         date: java.util.Date = new java.util.Date();
19         setContent(html: dateFormat.format(date))
20     }
```

---

Listing 4.3: Specification of the Clock artifact interface and sample usage

#### 4.3.4 Interaction with Web services

Like AJAX requests in traditional Web 2.0 applications, also simpAL-web requires mechanisms to enable the applications to interact with (RESTful) Web services, possibly asynchronously. A trivial approach would have been to introduce an artifact which provides operations that wrap HTTP actions such as *get* and *post*, just like the *XMLHttpRequest* object does in JavaScript. Anyway once again this is definitely not the most natural approach on the agent side programming. Since simpAL provides an higher level of abstraction what we would like is ad-hoc artifacts that wrap in a proper way Web services functionalities, possibly even regardless to HTTP actions, for the purpose to ease the use on the agent side. So it will be responsibility of the artifact to actually interface to the Web service, performing the proper HTTP requests, and providing the outcome through operations or observable properties to Web agents. Therefore besides the programming of agents, the programming of service wrapper artifacts is demanded to the Web application developer too. In order to simplify the development of these artifacts many Java libraries are provided in order to handling HTTP requests (package *simpal.web.http*) and JSON serialization/deserialization (package *simpal.web.json*). It's possible to conceive these artifacts mainly in two ways.

- *Artifacts as requests to Web services*

In some cases it may be useful to simply conceive the artifact itself as a request to a Web service. Similarly to how *jQuery* AJAX operations work, according to this approach the artifact acts as the “future” of a request to the Web service, whose completion will be notified to Web agents typically in form of status change of an observable property. Under these conditions this kind of artifacts are one-time usable because related to a specific request.



- *Artifacts as Web service proxies*

However in general it is preferable an approach at higher level of abstraction so the artifact no longer models the request to the Web service but the service itself, always for the purpose of making simple and clean as much as possible the programming on the Web agent side. For instance we would like to handle the Facebook Web service as it was the Facebook homepage, in which newer feeds are updated real time and through proper operations we may publish a new status or upload a picture. So this new kind of artifacts will provide a set of operations to act on the service, and a set of observable properties, updated upon status changes in the distributed service. In the simplest case the artifact can wrap the functionalities of the Web service through proper operations which eventually return some informations. Consider to this end the artifact interface in Listing 4.4 that through the *queryViewCount()* operation requests to the YouTube REST APIs the number of visualizations for the first item that match the search criteria and returns the result through the *count* output parameter.

---

```
1   operation queryViewCount(search: String,  
2                               count: long #out);  
3 }
```

---

Listing 4.4: simpAL-Web YouTube service artifact

In general anyhow is possible to structure the proxy artifact without being constrained by the structure of the Web service itself, in some cases even regardless to the nature of the HTTP protocol. There may be several services for which we would like to be able to exploit the fact that certain distributed informations or events may be perceived by Web agents as state changes of observable properties. Consider for instance a Web application whose objective is to display and keep updated the newest feed posted on your wall. Since, by its nature, the HTTP protocol does not support a server-side *push* action, the typical approach in

JavaScript would be to program a script which poll on the Web service scheduling periodical AJAX requests through the timer object. In a such way however in some sense the logic of the script is dirty because the logical flow responsible for the periodical requests interleave the application main control flow. Conversely in simpAL it is possible to separate the concerns by making the proxy artifact (Listing 4.5) responsible to perform the periodical requests to the service, by periodically executing an internal operation which update the status of an observable properties. So the Web agent (Listing 4.6), whose task is to display the latest feed from the Facebook service, delegates the entire logic needed to interact with the service to the artifact, and it provide only the behaviour to react on the updating of the observable property which represent the latest feed. In a such way, on the Web agent side, is as if the Web service would provide a *push* action and all the complexity due to the interaction with the service is concealed behind the artifact. Again in a such way we can conceive the artifact as a “sensor” or a connection to the outside world.

---

```
1 usage-interface FacebookInterface {
2     obs-prop lastFeed: String;
3 }
```

---

Listing 4.5: simpAL-web Facebook artifact interface

---

```
1 agent-script FeedRefreshAgent
2     implements FeedRefreshRole
3     in FBLastFeedOrgModel
4 {
5     plan-for Refresh using: page@main {
6         when loaded => {
7             divOut: WebElemInterface
8             getElement(selector: "divResult",
9                 elemArtifact: divOut);
10
11         using: fb@main {
```

```

12         every-time changed lastFeed =>
13             using: divOut
14         {
15             setContent(html: lastFeed)
16         }
17     }
18 }
19 }
20 }

```

---

Listing 4.6: simpAL-web Facebook last feed updater agent

## 4.4 Web programming in simpAL-web

Once introduced the simpAL-web platform and all the elements needed to enable simpAL to work in a Web environment, in this section we explain how to program simpAL Web applications. For this purpose we will provide several examples, some of which are taken from the Chapter 2. According these examples we also take some considerations about the benefits and drawbacks brought by the simpAL Agen-oriented programming paradigm and the simpAL-Web platform.

First of all we consider a typical “Hello world” application where the *GreetingAgent*, whose behaviour is specified by the agent script in Listing 4.7, once the appropriate Web page is loaded (line 6), first retrieves the element artifact which corresponds to the *div* element in the page identified by *divResult* (lines 7-9), and then sets its content displaying the “Hello World” message (lines 11-13). Notice that the *setContent()* operation may not be required to the element artifact *divOut* until the reference to the artifact itself was not obtained through the *getElement()* operation on the page artifact. This justifies the semicolon at the end of line 9 which forces the sequence between the action rule blocks at line 8-9 and at lines 11-13. From this first example we can see the basic behaviour of the main Web agent which in general should wait that the page was loaded and should request, through the *getElement()* operation, the references to those artifacts which represent the

HTML elements that are used or observed by the main agent itself. So far the simpAL Web application is very similar to the corresponding JavaScript or Dart implementation.

---

```
1  agent-script GreetingAgent
2      implements GreetingRole
3      in HelloWorldOrgModel
4  {
5      plan-for Greet using: page@main {
6          when loaded => {
7              divOut: WebElemInterface
8              getElement(selector: "divResult",
9                          elemArtifact: divOut);
10
11             using: divOut {
12                 setContent(html: "Hello World!")
13             }
14         }
15     }
16 }
```

---

Listing 4.7: simpAL-web Hello World

#### 4.4.1 Dealing with asynchronous programming

The second example we introduce is a slight variation of the “Hello world” above in which the message is displayed upon the div element only once that the user clicks on a button. Through this second example we can taste for the first time how to tackle asynchronous programming in simpAL-web, that is what in JavaScript we would have handled through callbacks. Differently from the previous example now the *GreetingAgent2* requests to the page artifact the reference to the *buttonGreet* too, which will be observed in the following action rule block (lines 16-20) in order to react when the button was clicked. Notice that while in JavaScript we should provide a function, even if anonymous, to be attached as callback to the button *onclick* event, compared to the previous

example in simpAL the changes to the code are minimal. Besides in JavaScript the logic flow is broken because of the callback while in simpAL it is like the *when pressed* action rule (line 17) would block the control flow, and none of the actions inside the block would be activated until the button was pressed. Actually the *when* condition does not have the effect of blocking the control flow as for instance would do a *wait* statement on a Java thread, but simply prevents that the action body of the action rule would be performed until the press event is not occurred, according to the simpAL Agent architecture in 3.2.3. So a great benefit is that simpAL supports asynchronous programming keeping the control flow intact and linear. Besides, as mentioned in the previous chapter, simpAL also solves the Inversion of Control issue thanks to the Agent architecture which fully encapsulate its control logic.

---

```
1  agent-script GreetingAgent2
2      implements GreetingRole
3      in HelloWorldOrgModel
4  {
5      plan-for Greet using: page@main {
6          when loaded => {
7              buttonGreet: WebElemInterface
8              divOut: WebElemInterface
9
10             {
11                 getElement(selector: "buttonGreet",
12                             elemArtifact: buttonGreet)
13                 getElement(selector: "divOutput",
14                             elemArtifact: divOut)
15             };
16
17             using: buttonGreet {
18                 when pressed using: divOut {
19                     setContent(html: "Hello World!")
20                     on divOut
21                 }
22             }
23         }
24     }
```

---

 Listing 4.8: simpAL-web Hello World 2

Another example related to asynchronous programming is given by the one in Listing 4.9 which resume the "Battle of the bands" application by providing the simpAL implementation. Once the page was loaded and the reference to the output *div* element was obtained the YouTube service is queried through the wrapper artifact *YouTubeServiceArtifact* (Listing B.5) about the number of visualizations of two song entries (provided as task input parameters in the organization definition B.7). Since no semicolons are specified the two operations *queryViewCount()* are requested to the *YouTubeServiceArtifact* simultaneously and can be carried out possibly concurrently by the artifact<sup>3</sup>. Notice again that labeling the *queryViewCount()* operations (lines 15 and 17) we can react when both the requests are completed, thanks to the simpAL's built-in function *is-done* that checks if an action rule block or a task is successfully completed, in order to display the winner. Once again simpAL compared for instance to the JavaScript solution based on Promises and *jQuery.when()* in Listing 2.14 brings the benefit to natively provide functions to manage tasks and action rule blocks as first class entities, besides the integrity of the control flow seen before.

---

```

1  agent-script RefereeAgent
2      implements RefereeRole
3      in BattleOfBandsOrgModel
4  {
5      plan-for Play using: page@main {
6          when loaded => {
7              divOut: WebElemInterface
8              getElement(selector: "divResult",
9                          elemArtifact: divOut);
10
11              using: youTube@main {
```

---

<sup>3</sup>Notice that in the actual implementation of the *YouTubeServiceArtifact*, as by default in the artifact semantic, the requests are serialized

```

12     viewCount1: long
13     viewCount2: long
14
15     queryViewCount(search: this-task.contender1,
16                     count: viewCount1) #req1
17     queryViewCount(search: this-task.contender2,
18                     count: viewCount2) #req2
19
20     when is-done req1 && is-done req2 =>
21         using: divOut {
22             if (viewCount1 > viewCount2) {
23                 setContent(html: "The winner is: "
24                                 + this-task.contender1
25                                 + " (" + viewCount1 + ")")
26             }
27             else {
28                 setContent(html: "The winner is: "
29                                 + this-task.contender2
30                                 + " (" + viewCount2 + ")")
31             }
32         }
33     }
34 }
35 }
36 }

```

---

Listing 4.9: simpAL-web "Battle of the bands"

#### 4.4.2 Dealing with concurrency

In the next example we resume instead the "Stoppable counter" application we have used to introduce concurrency in Web programming through HTML5 Web Workers and Dart Isolates. In order to understand if simpAL deals better for this aspect too in Listing 4.10 we provide the script of the main Web agent. Resuming what the application should do, the Web page provides two buttons used by the user to start and stop a counter which repeatedly increments a value and displays it on an output *div*. Clearly should be handled the fact that clicks on start and stop buttons have an effect only if the counter was

in idle or running status respectively. Furthermore the main requirement is responsiveness, this means that as soon the user will click on the start button the counter will start increment, and vice versa as soon as the stop button will be clicked the counter must stop too. The essential behaviour of the application is specified among lines 20-30, through the action rule at line 21 the agent reacts first when the *buttonStart* is pressed by activating the *repeatedly* block which request the *inc()* operation to the counter artifact and then through the *setContent()* operation it displays the value, corresponding to the namesake observable property, on the output *div*. The stop condition is given by the *completed-when* clause at line 23 which means that while performing the *repeatedly* block as soon as the *buttonStop* is pressed it terminates the whole block in execution. Notice that this is consistent with the semantic given for the application in the sense that when the counter is in the idle status the agent can react only to the pressure of *buttonStart* (line 21) so state changes on the *buttonStop* artifact will be ignored as vice versa when the counter is running (*repeatedly* block) only the pressure of *buttonStop* is perceived (line 23) since the action rule block corresponding to *every-time pressed* for the *buttonStart* artifact is at a lower level on the intentions stack. Notice that the disambiguations for the *pressed* observable property (line 23) and for the *setContent()* operation (line 27) are necessary since three artifacts of the same type *WebElemInterface* are used inside the same block. For what concerns concurrency as mentioned in the previous chapter simpAL brings several benefits, in particular considering this example compared to the JavaScript solutions (Listings 2.17, 2.18, 2.20) and Dart ones (Listing 2.25). The main aspect surely concerns the ability of simpal to integrate autonomous and reactive behaviours as you can see in Listing 4.10 in fact the Web agent is able to iteratively increase the counter but being reactive to the pressure of *buttonStop* too, aspect impossible to achieve with Web actors seen in Chapter 2 except for tricks. Besides while in Actor-based solutions we had to design the protocol through which the main actor and the counter actor exchange messages, in simpAL the agent script *StoppableCounterAgent* fully encapsulates the whole application logic.



---

```

1  agent-script StoppableCounterAgent
2      implements StoppableCounterRole
3      in StoppableCounterOrgModel
4  {
5      plan-for UpdateCounter using: page@main {
6          when loaded => {
7              buttonStart: WebElemInterface
8              buttonStop: WebElemInterface
9              divOut: WebElemInterface
10
11             {
12                 getElement(selector: "buttonStart",
13                             elemArtifact: buttonStart)
14                 getElement(selector: "buttonStop",
15                             elemArtifact: buttonStop)
16                 getElement(selector: "divResult",
17                             elemArtifact: divOut)
18             };
19
20             using: buttonStart {
21                 every-time pressed =>
22                     using: buttonStop, divOut, counter
23                     completed-when: pressed in buttonStop
24                 {
25                     repeatedly {
26                         inc();
27                         setContent(html: "" + value)
28                         on divOut
29                     }
30                 }
31             }
32         }
33     }
34 }

```

---

Listing 4.10: simpAL-web "Stoppable counter"

### 4.4.3 Dealing with complex applications

In the last example we tackle with simpAL-web the case study Web application introduced in 2.5 which should be representative for all those applications in the real world that is more complex than the examples seen so far and mix aspects related to concurrency and asynchronous programming besides involving aspects related to in-the-large programming. For demonstration purposes the application is structured according to a Master-Worker architecture where the agent scripts in Listings 4.11 and 4.12 implement respectively the behaviour for the master agent and a worker agent. Aim of the master agent is, once the page was loaded, to query the Facebook service in order to get the list of friends and to assign a slice of it to each workers (two for instance in our organization B.20) in such a way that they can examine then what friends like. Since the master agent's behaviour is fairly complex we exploit the modularity mechanism provided by simpAL that allows to organize the whole behaviour in several tasks which will be handled by as many plans. So once the page was loaded through the *do-task* action the master agent schedule two new tasks (lines 17-18) which will be performed concurrently. The former is responsible to load the friends from the Facebook account and to assign them to the workers, by using the *assign-task* action which schedule a new task to be executed by a certain agent (lines 22-43). The latter instead is responsible to display on the Web page the report containing what friend most like at the same time it is built by workers by observing the *htmlOutput* property of the *ReportBuilderArtifact* (lines 45-51).

---

```
1  agent-script MasterAgent
2      implements MasterRole
3      in FBTopLikesOrgModel
4  {
5      buttonStop: WebElemInterface
6      divOut: WebElemInterface
7
8      plan-for Boot using: page@main {
9          when loaded => {
```

```

10     {
11         getElement(selector: "buttonStop",
12                     elemArtifact: buttonStop)
13         getElement(selector: "divResult",
14                     elemArtifact: divOut)
15     };
16
17     do-task new-task LoadFriends()
18     do-task new-task DisplayOutput()
19 }
20 }
21
22 plan-for LoadFriends {
23     using: fb@main {
24         friendIds: java.util.List
25         length: int
26         halfLength: int
27
28         queryFriends(friendIds: friendIds);
29         length = friendIds.size();
30         halfLength = length / 2;
31
32         {
33             assign-task
34                 new-task LoadLikes(friendIds:
35                                     friendIds.subList(0, halfLength))
36                 to: worker1@main
37             assign-task
38                 new-task LoadLikes(friendIds:
39                                     friendIds.subList(halfLength, length))
40                 to: worker2@main
41         }
42     }
43 }
44
45 plan-for DisplayOutput {
46     using: reportBuilder@main {
47         every-time changed htmlOutput => using: divOut {
48             setContent(html: htmlOutput)
49         }
50     }
51 }

```

```

52
53     task LoadFriends {}
54     task DisplayOutput {}
55 }

```

---

Listing 4.11: simpAL-web "What friends most like?" MasterAgent script

On the other side, the worker agent will be activated when the master request to it a *LoadLikes* task so it implements the logic needed to query the like data for each friend and to populate the report through the *ReportBuilder-Artifact*. As you can notice the simpAL implementation of this last example does not provide a mechanism to suspend and restart the like crawling, like instead the JavaScript implementation did. However, thanks to simpAL ability to manage tasks as first class entities, this would be easy to implement for instance by using the built-in *suspend-task* and *resume-task* actions. This last example shows how in general it is possible in simpAL to modularize behaviours through tasks and plans to deal with complex and large-scale Web applications. This is applicable to both programming of a single agent or agent architectures.

---

```

1  agent-script WorkerAgent
2      implements WorkerRole
3      in FBTopLikesOrgModel
4  {
5      plan-for LoadLikes {
6          i: int = 0
7          while (i < this-task.friendIds.size())
8              using: fb@main, reportBuilder@main
9              {
10                 tid: String = (String) this-task.friendIds.get(i);
11                 tlikes: java.util.List;
12
13                 queryLikes(friendId: tid, likes: tlikes);
14                 addLikes(newLikes: tlikes);
15                 i++
16             }
17     }

```

---

Listing 4.12: simpAL-web "What friends most like?" WorkerAgent script

## 4.5 About simpAL-web implementation

In this last section we will provide further details about the actual implementation of the simpAL-web platform, we have built to experience our trials. Core component and essential for the platform itself is the Web browser, responsible for retrieving, rendering and showing HTML pages besides managing the interaction between the user and the Web page. Such browser should then be somehow interfaced to simpAL-web artifacts which will act on it. To this end the Web browser should also be programmatically accessible, possibly in Java, by the artifacts in order to inspect and manipulate the page DOM and catch the user inputs. Since it's unthinkable to design and develop a Web browser from scratch we were looking for existing technologies and frameworks that met our requirements. The decision was between extending an open-source Web browser (such Mozilla Firefox or Google Chromium), using Java applets that are supported by most browsers, or relying on the JavaFX platform which provides an embedded browser. Finally, we settled for the JavaFX option since it offers several advantages compared to other alternatives, that is it natively provides API to full access and manipulate DOM elements and it is completely based and programmable in Java. Furthermore being based on the WebKit<sup>4</sup> engine it supports all the new HTML5 features. For the purpose to interface the simpAL runtime and the JavaFX application which function as Web browser the architecture described by the UML diagram in Figure 4.2 has been realized. Notice on the left side the *JFXBrowser* class which wraps the JavaFX application composed essentially of a *WebEngine* and a *WebView* components which together form the browser. On the right side there are instead the simpAL *WebPageArtifact* and *WebElemArtifact* artifacts used by the Web agents as part of the simpAL Web application. The middleware which interconnects

---

<sup>4</sup><http://www.webkit.org/>

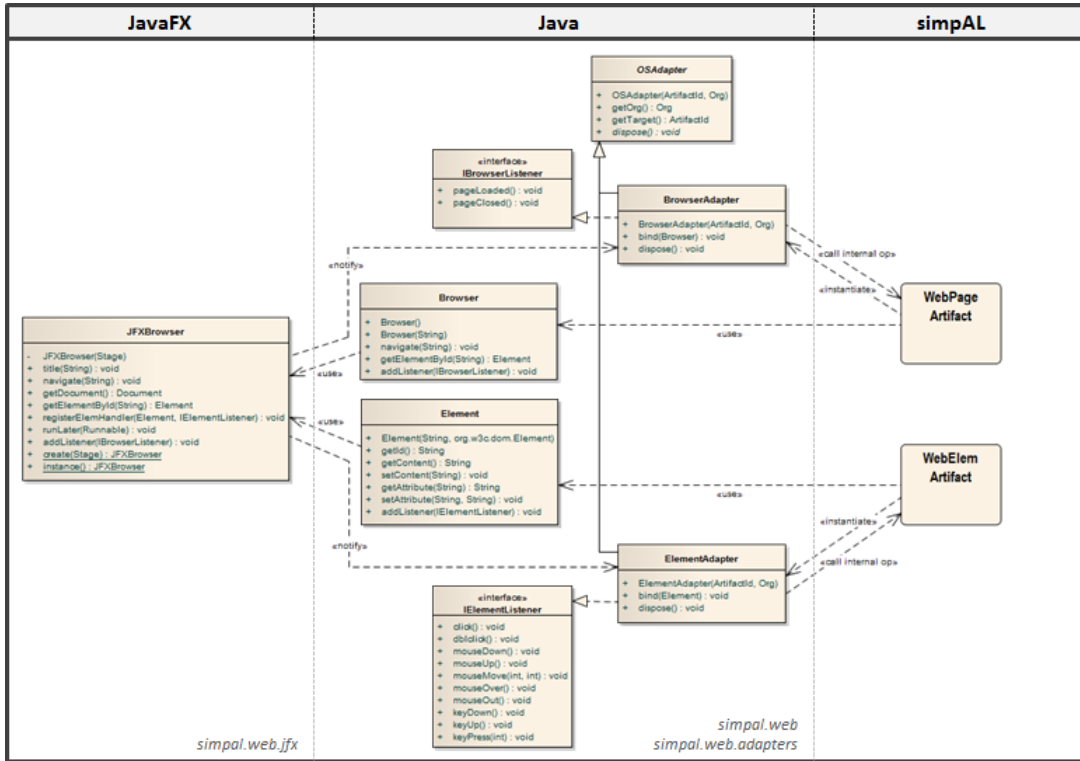


Figure 4.2: UML diagram of the simpAL-web architecture

these two sides should therefore provides the following functionalities: enable the artifacts to access and manipulate the DOM elements of the Web page displayed in the browser, for instance to set the html content of an element, or to retrieve the element that correspond to a given id; and notify the artifacts about the events that occur on the browser side. For this purpose the *Browser* and *Element* classes wrap the browser functionalities which are used by artifacts respectively to act on the Web page and elements. Conversely the *BrowserAdapter* and *ElementAdapter* are instantiated by the artifacts and are responsible to notify them events respectively from the Web page and elements updating their observable properties by calling “internal” pre-defined operations on the artifacts. *BrowserAdapter* and *ElementAdapter* classes implements respectively the *IBrowserListener* and *IElementListener* interfaces in order to be notified by the *JFXBrowser* component and extend the *OSAdapter* class of the simpAL runtime which provide mechanisms to call artifacts “inter-

nal” operations. For further details about the platform implementation you can take a look at the actual simpAL-web codes for *WebPageArtifact* and *WebElemArtifact* in Listings B.1 and B.2 respectively.

# Chapter 5

## Conclusions

Main objective of this thesis was to tackle client-side Web application programming especially for what concerns concurrency and asynchronous programming, starting from existing reference and state-of-art technologies and finally by promoting a new approach based on the Agent-oriented programming paradigm and the simpAL language. In conclusion we would take some considerations about our work by summarizing strengths and weaknesses of Web programming in simpAL compared to reference technologies. In particular the main problems we have identified in reference technologies typically result from lack in the abstraction level of which instead simpAL does not suffer. From our analysis and experiments made through the simpAL-web platform we found simpAL and Agent-oriented Web programming convenient, compared to traditional technologies, under several aspects which can be grouped into the three following main categories:

- *Asynchronous programming*

Since the JavaScript language as well as Dart and all other nowadays languages are not natively designed to support asynchronous programming, in the meaning that they not provide primitives to handle events, this feature is typically supported through callbacks. The callback mechanism is the actual main issue of current technologies since it implicitly involves fragmentation of the control logic resulting in *Asynchronous spaghetti*. As in most cases Promises/Futures make it possible to mit-



igate the problem, as demonstrated in section 2.6.1 in other cases the problem still remains. Conversely the simpAL conceptual model natively supports asynchronous programming in a such way that agents can react to the occurrence of events, which correspond to state changes in artifacts observable properties or messages between agents, through the *action rule*'s Event-Condition-Action mechanism which is the fundamental building block for the agent's behaviour. So we can say that in a sense action rules in simpAL enforce structure in asynchronous programming. As mentioned before another issue due to the callback mechanism is the so-called *Inversion of Control*. In this case simpAL solves the problem since, according to the agent architecture, the whole control logic of an agent is carried on by the same logical control flow that is the agent's control loop.

- *Concurrency*

Also for what concerns concurrency simpAL, thanks to its higher layer of abstractions, allows to address problems that plague current technologies. In simpAL agents are executed concurrently, as well as operations on distinct artifacts so primarily it allows to abstract from low-level mechanisms such as those in multi-threaded programming. Besides simpAL provides concurrency at a logical level, in the sense that several agents may be running on a single physical thread. This leads to a good degree of scalability compared for instance to HTML5 Web Workers which are mapped one-by-one to OS threads. The greatest benefit of simpAL agents in this context is surely the ability to mix autonomous and reactive behaviours since both HTML5 Web Workers and Dart Isolates, being based on the Actor-model, are pure reactive entities so they do something only in reaction to the reception of a message and they can not block or do long-term computations. Moreover differently to current technologies where only the "main" Web actor (Web Worker or Isolate) can interact with the DOM, thanks to the atomic execution of operations on artifacts, all the agents of a simpAL Web application can act on the artifacts which wraps the Web page and its elements.

- *Software engineering and in-the-large programming*

Separation between interfaces and implementations (roles and agent scripts, usage-interfaces and artifact templates, org-model and org) besides separation between application active and passive parts, agents and artifacts respectively, and modularity through tasks and plans are all elements of simpAL inspired by software engineering principles and preparatory for in-the-large programming. Another important aspect for this purpose is the ability to do error checking at compile time, thanks to simpAL strong type system. Again simpAL Web applications would be more maintainable in the sense that adding a new behaviour to an application, possibly concurrent to other existing behaviours, thanks to tasks, action rule blocks and in general simpAL higher level abstractions would be easier than in the corresponding JavaScript application where for instance the implementation of the new feature may require modifying several parts of the application code.

However, also our approach has some weaknesses, for example the simpAL strong type system may be too invasive while developing small-scale applications or prototypes, where for instance the JavaScript dynamic typing might be preferable. To this end the best choice would be to adopt an optional static typing just like in Dart and TypeScript languages.

## 5.1 Future works

Ultimate goal of this work was to provide a foundation to explore how issues related to Web programming can be tackled and partially addressed through the Agent-oriented programming paradigm and the simpAL language in particular. Notice first that all considerations done so far including the benefits identified are bound by the assumptions made at the beginning of Chapter 4, so future explorations may be directed to extend the analysis of this approach to Web programming in general. simpAL too is an early technology and several issues are still work in progress such as for instance mechanisms like sub-typing and inheritance for agents and artifacts in order to better sup-

porting extensibility and reuse besides the linkability between artifacts. Both these aspects might be useful to better engineering the simpAL-web platform for example to model hierarchies of elements artifacts. For example we would like text input and textarea element artifacts would provide the *text* observable property in addition to all those inherited from the default element artifact.

Another important aspect totally neglected because beyond the scope of this thesis concerns performance. Future works could focus on measuring simpAL-web performances compared to mainstream reference technologies.

In addition, to the current state simpAL-web still remains a prototype platform, so future works may be directed to improve robustness and integration. To these ends it might be interesting and useful to deploy simpAL virtual machines as extensions of most used Web browser, such as Google Chrome and Mozilla Firefox.



# List of Tables

2.1	Comparison between jQuery Promises and Dart Futures APIs	34
2.2	Browsers compatibilities to HTML5 Web Workers . . . . .	37

# List of Figures

2.1	JavaScript frameworks ecosystem . . . . .	8
2.2	Languages and tools that compile to JavaScript . . . . .	11
2.3	Event loop pseudocode . . . . .	21
2.4	jQuery Promises API . . . . .	27
2.5	Case study Web-app architecture . . . . .	50
2.6	Case study Web-app output . . . . .	51
2.7	UML Activity diagram of a complex workflow . . . . .	53
3.1	simpAL overview . . . . .	60
3.2	simpAL control loop pseudocode . . . . .	67
3.3	simpAL agent architecture . . . . .	68
4.1	Structure of a simpAL Web application . . . . .	78
4.2	UML diagram of the simpAL-web architecture . . . . .	98

# List of Listings

2.1	jQuery “Hello World!” . . . . .	9
2.2	Node.js tasting . . . . .	10
2.3	Tasting StratifiedJS strata . . . . .	12
2.4	Google Dart sample class . . . . .	15
2.5	Microsoft TypeScript sample class . . . . .	17
2.6	Understanding the event loop, first example . . . . .	21
2.7	Understanding the event loop, second example . . . . .	22
2.8	Exceptions in asynchronous code . . . . .	23
2.9	“I love async, but I can’t code like this” . . . . .	24
2.10	Promising AJAX . . . . .	27
2.11	JavaScript countdown . . . . .	28
2.12	Asynchronous workflow with pipes . . . . .	30
2.13	Asynchronous workflow without pipes . . . . .	30
2.14	JavaScript “Battle of the bands” . . . . .	32
2.15	JavaScript “Battle of the bands” error . . . . .	33
2.16	Dedicated worker counter – main.js . . . . .	38
2.17	Dedicated worker counter – workerMaster.js . . . . .	39
2.18	Dedicated worker counter – worker.js . . . . .	39
2.19	Shared worker counter – main.js . . . . .	40
2.20	Shared worker counter – worker.js . . . . .	41
2.21	Unresponsive counter – main.js . . . . .	43
2.22	Unresponsive counter – worker.js . . . . .	43
2.23	Deferred worker – main.js . . . . .	45
2.24	Deferred worker – adder.js . . . . .	46
2.25	Dart counter . . . . .	47

2.26	Complex workflow JavaScript implementation . . . . .	54
3.1	Definition of the <i>InteractiveCounter</i> role in simpAL . . . . .	61
3.2	Definition of the <i>InteractiveCounter</i> agent script in simpAL . . . . .	65
3.3	Definition of the Counter artifact interface in simpAL . . . . .	69
3.4	Definition of the Counter artifact template in simpAL . . . . .	70
3.5	Definition of the Example organization model in simpAL . . . . .	71
3.6	Definition of the Example concrete organization in simpAL . . . . .	72
4.1	Specification of the Web page artifact interface . . . . .	80
4.2	Specification of the Web element artifact interface . . . . .	82
4.3	Specification of the Clock artifact interface and sample usage . . . . .	83
4.4	simpAL-Web YouTube service artifact . . . . .	85
4.5	simpAL-web Facebook artifact interface . . . . .	86
4.6	simpAL-web Facebook last feed updater agent . . . . .	86
4.7	simpAL-web Hello World . . . . .	88
4.8	simpAL-web Hello World 2 . . . . .	89
4.9	simpAL-web "Battle of the bands" . . . . .	90
4.10	simpAL-web "Stoppable counter" . . . . .	92
4.11	simpAL-web "What friends most like?" MasterAgent script . . . . .	94
4.12	simpAL-web "What friends most like?" WorkerAgent script . . . . .	96



# Bibliography

- [1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.
- [2] G. Agha and C. Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. Technical report, Cambridge, MA, USA, 1985.
- [3] E. Bidelman. The basics of web workers — HTML5 Rocks. <http://www.html5rocks.com/en/tutorials/workers/basics/>, Jul. 2010.
- [4] R. H. Bordini, M. Wooldridge, and J. F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.
- [5] P. Bright. Microsoft typescript: the javascript we need, or a solution looking for a problem? <http://arstechnica.com/information-technology/2012/10/microsoft-typescript-the-javascript-we-need-or-a-solution-looking-for-a-problem/>, Oct. 2012.
- [6] T. Burnham. *Async JavaScript*. Leanpub, 2012.
- [7] J. Cardy. Combining jquery deferred with the html5 web workers api — codeproject. <http://www.codeproject.com/Articles/168604/Combining-jQuery-Deferred-with-the-HTML5-Web-Worke>, Mar. 2011.
- [8] Channel 9. Anders Hejlsberg and Lars Bak: TypeScript, JavaScript, and Dart. <http://channel9.msdn.com/Shows/Going+Deep/Anders-Hejlsberg-and-Lars-Bak-TypeScript-JavaScript-and-Dart>, Oct. 2012.

- [9] ECMA TC39 committee. Draft specification for es.next (ecma-262 edition 6). [http://wiki.ecmascript.org/doku.php?id=harmony:specification\\_drafts](http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts), Sept. 2012.
- [10] Google. "Future of Javascript" doc from our internal "JavaScript Summit". <https://gist.github.com/1208618>, Nov. 2010.
- [11] Google. Dart — frequently asked questions (faq). [http://www.dartlang.org/support/faq.html?utm\\_source=site&utm\\_medium=footer&utm\\_campaign=homepage#why-dart](http://www.dartlang.org/support/faq.html?utm_source=site&utm_medium=footer&utm_campaign=homepage#why-dart), Feb. 2012.
- [12] P. Haller and M. Odersky. Event-based programming without inversion of control. In *Proceedings of the 7th joint conference on Modular Programming Languages, JMLC'06*, pages 4–22, Berlin, Heidelberg, 2006. Springer-Verlag.
- [13] I. Hickson. Web workers. Technical report, W3C, May 2012. <http://www.w3.org/TR/workers/>.
- [14] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23(7):260–267, June 1988.
- [15] The Dart Team — Google. The dart programming language specification. <http://www.dartlang.org/docs/spec/latest/dart-language-specification.pdf>, Oct. 2012.
- [16] Microsoft. Typescript language specification. <http://www.typescriptlang.org/Content/TypeScript%20Language%20Specification.pdf>, Oct. 2012.
- [17] T. Mikkonen and A. Taivalsaari. Web applications - spaghetti code for the 21st century. In *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications, SERA '08*, pages 319–328, Washington, DC, USA, 2008. IEEE Computer Society.

- [18] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, Dec. 2008.
- [19] A. Ricci and A. Santi. Designing a general-purpose programming language based on agent-oriented abstractions: the simpal project. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPEs'11, NEAT'11, &#38; VMIL'11, SPLASH '11 Workshops*, pages 159–170, New York, NY, USA, 2011. ACM.
- [20] A. Ricci and A. Santi. Agent-oriented programming in simpal. Technical report, University of Bologna, 2012.
- [21] A. Ricci and A. Santi. Programming abstractions for integrating autonomous and reactive behaviors: An agent-oriented approach. Technical report, University of Bologna, Oct. 2012.
- [22] A. Ricci and A. Santi. Typing multi-agent programs in simpal. Technical report, University of Bologna, Giu. 2012.
- [23] S. Somasegar. Typescript: Javascript development at application scale. <http://blogs.msdn.com/b/somasegar/archive/2012/10/01/typescript-javascript-development-at-application-scale.aspx>, Oct. 2012.
- [24] K. Walrath. *Dart Up and Running*. Oreilly & Associates Inc, 2012.
- [25] Wikipedia. Document object model — wikipedia, the free encyclopedia. [http://en.wikipedia.org/w/index.php?title=Document\\_Object\\_Model&oldid=518651562](http://en.wikipedia.org/w/index.php?title=Document_Object_Model&oldid=518651562), 2012.
- [26] Wikipedia. Futures and promises — wikipedia, the free encyclopedia. [http://en.wikipedia.org/w/index.php?title=Futures\\_and\\_promises&oldid=519425800](http://en.wikipedia.org/w/index.php?title=Futures_and_promises&oldid=519425800), 2012.
- [27] Wikipedia. Javascript — wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=JavaScript&oldid=519072745>, 2012.

- [28] Wikipedia. JQuery — wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=jQuery&oldid=517526755>, 2012.
- [29] Wikipedia. Single-page application — wikipedia, the free encyclopedia, 2012.



# Appendix A

## Chapter 2 sources

### A.1 HTML documents

---

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Stoppable counter</title>
5   <meta charset="utf-8" />
6   <script type="text/javascript">
7     function log(msg) {
8       var fragment = document.createDocumentFragment();
9       fragment.appendChild(document.createTextNode(msg));
10      fragment.appendChild(document.createElement('br'));
11      document.querySelector("#log").appendChild(fragment);
12    }
13  </script>
14  <script type="text/javascript" src="main.js"></script>
15 </head>
16 <body>
17   <h1>Stoppable counter</h1>
18   <h3>A demo app for concurrency</h3>
19   <br /><br />
20
21   <button type="button" id="buttonStart">Start</button>
22   <button type="button" id="buttonStop">Stop</button>
23   <br />
```

```

24
25 <div id="res" style="background-color: Aqua">?
26 </div>
27 <br />
28 <div id="log">Log:<br />
29 </div>
30 </body>
31 </html>

```

---

Listing A.1: HTML of "Stoppable counter" app

---

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Facebook top likes</title>
5 <meta charset="utf-8" />
6
7 <script type="text/javascript" src="jquery.js"></script>
8 <script type="text/javascript" src="common.js"></script>
9 <script type="text/javascript" src="main.js"></script>
10 </head>
11 <body>
12 <h1>Facebook top likes</h1>
13 <h3>Find what your friends most like...</h3>
14
15 <div id="cmd" style="background-color:#f2f2f2;
16 width:70%; padding:2px;">
17 <button id="buttonPause" type="button">
18 Pause
19 </button>
20 <p id="status" style="display:inline;">
21 0/? friends crawled!
22 </p>
23 </div>
24 <br /><br />
25
26 <div id="res" style="background-color:#d8dfea;
27 width:70%;">
28 </div>

```

```
29     <br />
30     <div id="log">Log:<br /></div>
31 </body>
32 </html>
```

---

Listing A.2: HTML of "What friends most like?" app

## A.2 JavaScript sources of "What friends most like?" app

---

```
1 var controllerWorker;
2
3 function spawnController() {
4     controllerWorker = new Worker('controllerWorker.js');
5     controllerWorker.onmessage = function (msg) {
6         if (msg.data.ret)
7             $("#res").html(msg.data.ret);
8         else if (msg.data.status)
9             $("#status").html(msg.data.status);
10        else if (msg.data.log)
11            log(msg.data.log);
12    };
13    controllerWorker.onerror = function (ex) {
14        log('ERROR: ' + ex.message + ' @ '
15            + ex.filename + ':' + ex.lineno);
16    };
17    controllerWorker.postMessage({'op': 'start'});
18 }
19
20 function pause() {
21     $('#buttonPause').text('Resume');
22     controllerWorker.postMessage({'op': 'pause'});
23 }
24
25 function resume() {
26     $('#buttonPause').text('Pause');
```



```

27     controllerWorker.postMessage({'op': 'resume'});
28 }
29
30 $(document).ready(function () {
31     $("#buttonPause").click(function () {
32         if ($("#buttonPause").text().indexOf('Pause') >= 0)
33             pause();
34         else
35             resume();
36     });
37
38     spawnController();
39 });

```

---

Listing A.3: main.js

```

1 function log(msg) {
2     var fragment = document.createDocumentFragment();
3     fragment.appendChild(document.createTextNode(msg));
4     fragment.appendChild(document.createElement('br'));
5     document.querySelector("#log").appendChild(fragment);
6 }

```

---

Listing A.4: common.js

```

1 var WORKERS_POOL_SIZE = 4;
2 var subworkers;
3
4 var nFriendsDone = 0, nFriendsTot;
5
6 onmessage = function (msg) {
7     if (msg.data.op == 'start')
8         onSetup();
9     else if (msg.data.op == 'resume')
10        onResume();
11    else if (msg.data.op == 'pause')

```

```

12         onPause();
13     };
14
15     function onSetup() {
16         importScripts('worker_common.js');
17         var securityToken = requestFBAccessToken();
18
19         ajaxGet('https://graph.facebook.com/me/friends'
20             + '?access_token=' securityToken,
21             true,
22             function (resp) {
23                 var friendsData = [];
24                 for (var i = 0; i < resp.data.length; i++) {
25                     var fid = resp.data[i].id;
26                     friendsData.push('https://graph.facebook.com/'
27                         + fid
28                         + '/likes?access_token='
29                         + securityToken);
30                 }
31                 spawnSubworkers(friendsData);
32
33                 nFriendsTot = friendsData.length;
34                 postMessage({'status':produceHtmlStatus()});
35             }
36     );
37 }
38
39 function onResume() {
40     for (var i=0; i<WORKERS_POOL_SIZE; i++)
41         subworkers[i].postMessage({'enabled':'on'});
42     postMessage({'log':'Controller resumed!'});
43 }
44
45 function onPause() {
46     for (var k in subworkers)
47         subworkers[k].postMessage({'enabled':'off'});
48     postMessage({'log':'Controller paused!'});
49 }
50
51 function spawnSubworkers(urls) {
52     var part = computeTaskPartitioning(urls.length);
53     subworkers = [];

```

```

54     for (var i=0; i<WORKERS_POOL_SIZE; i++) {
55         var subworker = new Worker('likesLoaderWorker.js');
56         subworkers[i] = subworker;
57
58         subworker.onmessage = function (msg) {
59             if (msg.data.log)
60                 postMessage(msg.data);
61             else {
62                 mergeResult(msg.data.data);
63                 subworkers[msg.data.id].postMessage('');
64             }
65         };
66         subworker.onerror = function (ex) {
67             throw 'ERROR: ' + ex.message + ' @ '
68                 + ex.filename + ':' + ex.lineno;
69         };
70         var data = urls.slice(part[i].from, part[i].to);
71         subworker.postMessage({'id':i, 'data':data});
72         subworker.postMessage({'enabled':'on'});
73     }
74 }
75
76 function computeTaskPartitioning(len) {
77     var partition = [];
78     var ntxw = Math.floor(len / WORKERS_POOL_SIZE);
79     var rest = len % WORKERS_POOL_SIZE;
80
81     var k = 0;
82     for (var i=0; i<WORKERS_POOL_SIZE; i++) {
83         var qty = ntxw + (rest == 0 ? 0 : 1);
84         if (rest > 0)
85             rest--;
86
87         partition.push({'from':k, 'to':k+qty});
88         k = k+qty;
89     }
90     return partition;
91 }
92
93 function requestFBAccessToken() {
94     // Facebook Graph API Explorer temporary token
95     return 'AAACEdEose0cBAEGzvR09W15JL7K04I7vfYZCZCZAFZCwy' +

```

```

96         'ZAXQ7dDTPtk5tJAN6GpIkw0jcM402vUPTGOT0Iy8uhs4ln' +
97         'haGPusJAWZAXNiH4XXZCj34YJn82';
98     }
99
100     var likesMap = new Array();
101     var likesIdToName = new Array();
102
103     function mergeResult(data) {
104         for (var lid in data) {
105             if (likesMap[lid] != null)
106                 likesMap[lid]++;
107             else {
108                 likesMap[lid] = 1;
109                 likesIdToName[lid] = data[lid];
110             }
111         }
112
113         nFriendsDone++;
114         postMessage({'ret':produceHtmlSummary()});
115         postMessage({'status':produceHtmlStatus()});
116     }
117
118     function produceHtmlSummary() {
119         var summaryEntriesLimit = 10;
120         var ids = new Array();
121
122         var max = 0;
123         for (var lid in likesMap)
124             if (likesMap[lid] > max)
125                 max = likesMap[lid];
126
127         outer: for (var i = max; i > 0; i--) {
128             for (var lid in likesMap) {
129                 if (likesMap[lid] == i) {
130                     ids.push(lid);
131                     if (ids.length >= summaryEntriesLimit)
132                         break outer;
133                 }
134             }
135         }
136
137         var summary = '<table>';

```

```

138     for (var i = 0; i < ids.length; i++)
139         summary = summary + '<tr><td>' + (i + 1) + '</td>' +
140             '<td>' + '</td>' +
142             '<td><a href="https://www.facebook.com/' + ids[i] +
143             '">' + likesIdToName[ids[i]] + '</a><br/>' +
144             likesMap[ids[i]] + ' x ' +
147             '<br /></td></tr>';
148     summary = summary + '</table>'
149     return summary;
150 }
151
152 function produceHtmlStatus() {
153     var status = '' + nFriendsDone + '/'
154     + nFriendsTot + ' friends crawled!'
155 }

```

---

Listing A.5: controllerWorker.js

---

```

1  var urlQueue;
2  var enabled = false;
3
4  onmessage = function (msg) {
5      if (msg.data.data) {
6          importScripts('worker_common.js');
7          id = msg.data.id;
8          urlQueue = msg.data.data;
9      }
10     else {
11         if (msg.data.enabled)
12             enabled = (msg.data.enabled == 'on');
13         if (enabled)
14             doNextTask();
15     }
16 }
17
18 function doNextTask() {

```

```

19     var url = urlQueue.pop();
20     ajaxGet(url,
21         false,
22         function (resp) {
23             var likesData = [];
24             for (var i = 0; i < resp.data.length; i++)
25                 likesData[resp.data[i].id]
26                     = resp.data[i].name;
27             postMessage({'id':id, 'data':likesData});
28         }
29     );
30 }

```

---

Listing A.6: likesLoaderWorker.js

```

1 function ajaxGet(url, async, okFunct, failFunct) {
2     var req = new XMLHttpRequest();
3     req.addEventListener('load', function () {
4         if (req.status == 200)
5             okFunct(JSON.parse(req.responseText));
6         else
7             throw 'XMLHttpRequest for url ' + url +
8                 ' failed: ' + req.status;
9     }, false);
10    req.open('GET', url, async);
11    req.send();
12 }

```

---

Listing A.7: worker\_common.js



# Appendix B

## Chapter 4 sources

### B.1 WebPageArtifact implementation

---

```
1 artifact WebPageArtifact
2     implements WebPageInterface
3 {
4
5     browser: simpal.web.Browser;
6
7     init (url: String, title: String) {
8         loaded = false;
9         closed = false;
10
11         browser = new simpal.web.Browser(title);
12         browserAdapter: simpal.web.adapters.BrowserAdapter =
13             new simpal.web.adapters.BrowserAdapter(me, korg);
14         browserAdapter.bind(browser);
15         korg.registerAdapter(browserAdapter);
16
17         browser.navigate(url);
18     }
19
20
21     operation getElement(selector: String,
22                         elemArtifact: WebElemInterface #out) {
23         elem: simpal.web.Element =
```



```

24     browser.getElementById(selector);
25     elemArtifactId : simpal.runtime.ArtifactId =
26         simpal.web.adapters.ElementAdapter
27             .createArtifactForElement(elem, korg);
28     elemArtifact = cast-obj-to-simpal-type
29         WebElemInterface elemArtifactId;
30 }
31
32
33 // Adapters operations
34
35 operation adapterLoad() {
36     loaded = true;
37 }
38
39 operation adapterExit() {
40     closed = true;
41 }
42 }

```

---

Listing B.1: simpAL-web WebPageArtifact implementation

## B.2 WebElemArtifact implementation

---

```

1 artifact WebElemArtifact implements WebElemInterface {
2
3     elem_id: String;
4     elem: simpal.web.Element;
5
6     init (id: String, element: simpal.web.Element) {
7         elem_id = id;
8         elem = element;
9
10        clicks = 0;
11        doubleClicks = 0;
12        pressed = false;
13        focused = false;

```

```

14     pointerCoord = new
15         simpal.web.PointerCoordinates(0, 0);
16
17     keyPressed = false;
18     key = 0;
19
20     elemAdapter: simpal.web.adapters.ElementAdapter =
21         new simpal.web.adapters.ElementAdapter(me, korg);
22     elemAdapter.bind(elem);
23     korg.registerAdapter(elemAdapter);
24 }
25
26
27 operation getContent(html: String #out) {
28     html = elem.getContent();
29 }
30
31 operation setContent(html: String) {
32     elem.setContent(html);
33 }
34
35 operation getAttribute(attr: String, value: String #out) {
36     value = elem.getAttribute(attr);
37 }
38
39 operation setAttribute(attr: String, value: String) {
40     elem.setAttribute(attr, value);
41 }
42
43
44 // Adapters operations
45
46 operation adapterMouseClicked() {
47     clicks++;
48 }
49
50 operation adapterMouseDown() {
51     pressed = true;
52 }
53
54 operation adapterMouseDown() {
55     pressed = true;

```

```

56     }
57
58     operation adapterMouseUp() {
59         pressed = false;
60     }
61
62     operation adapterMouseEnter() {
63         focused = true;
64     }
65
66     operation adapterMouseLeave() {
67         focused = false;
68     }
69
70     operation adapterMouseMove(x: int, y: int) {
71         pointerCoord = new simpal.web.PointerCoordinates(x, y);
72     }
73
74
75     operation adapterKeyDown() {
76         keyPressed = false;
77     }
78
79     operation adapterKeyUp() {
80         keyPressed = true;
81     }
82
83     operation adapterKeyPress(k: int) {
84         key = k;
85     }
86 }

```

---

Listing B.2: simpAL-web WebElemArtifact implementation

## B.3 simpAL sources of “Battle of the bands” app

---

```

1 role RefereeRole {

```

```

2     task Play{
3         input-params {
4             contender1: String;
5             contender2: String;
6         }
7     }
8 }

```

---

Listing B.3: RefereeRole.simpal

---

```

1 usage-interface YouTubeServiceInterface {
2     operation queryViewCount(search: String,
3                             count: long #out);
4 }

```

---

Listing B.4: YouTubeServiceInterface.simpal

---

```

1 artifact YouTubeServiceArtifact
2     implements YouTubeServiceInterface
3 {
4     init () {
5     }
6
7     operation queryViewCount(search: String,
8                             count: long #out) {
9         encodedSearch : String =
10            java.net.URLEncoder.encode(search, "UTF-8");
11
12         result : simpal.web.json.JSONObject =
13            simpal.web.http.HTTPClient.getJSON(
14                "http://gdata.youtube.com/feeds/api/videos?q="
15                + encodedSearch
16                + "&orderby=viewCount&max-results=1 "
17                + "&v=2&alt=jsonc");
18
19         count = result.getJSONObject("data")

```

```
20         .getJSONArray("items")
21         .getJSONObject(0)
22         .getLong("viewCount");
23     }
24 }
```

---

Listing B.5: YouTubeServiceArtifact.simpal

---

```
1  org-model BattleOfBandsOrgModel {
2      workspace main {
3          page: WebPageInterface
4          youtube: YouTubeServiceInterface
5
6          refereeAgent: RefereeRole
7      }
8 }
```

---

Listing B.6: BattleOfBandsOrgModel.simpal

---

```
1  org BattleOfBandsOrg
2      implements BattleOfBandsOrgModel
3  {
4      workspace main {
5          page = WebPageArtifact(url: "testpage.htm",
6                                  title: "Battle Of The Bands")
7          youtube = YouTubeServiceArtifact()
8
9          refereeAgent = RefereeAgent()
10             init-task: Play(contender1:
11                             "Led Zeppelin Stairway to heaven",
12                             contender2:
13                             "Pink Floyd The wall")
14     }
15 }
```

---

Listing B.7: BattleOfBandsOrg.simpal

## B.4 simpAL sources of “Stoppable counter” app

---

```
1 role StoppableCounterRole {
2   task UpdateCounter{}
3 }
```

---

Listing B.8: StoppableCounterRole.simpal

---

```
1 usage-interface CounterInterface {
2   obs-prop value: long;
3
4   operation inc();
5 }
```

---

Listing B.9: CounterInterface.simpal

---

```
1 artifact CounterArtifact
2   implements CounterInterface
3 {
4   init() {
5     value = 0;
6   }
7
8   operation inc() {
9     value = value + 1;
10  }
11 }
```

---

Listing B.10: CounterArtifact.simpal

---

```
1 org-model StoppableCounterOrgModel {
```

---

```

2 workspace main {
3     counter: CounterInterface
4     page: WebPageInterface
5
6     counterAgent: StoppableCounterRole
7 }
8 }

```

---

Listing B.11: StoppableCounterOrgModel.simpal

---

```

1 org StoppableCounterOrg
2     implements StoppableCounterOrgModel
3 {
4     workspace main {
5         counter = CounterArtifact()
6         page = WebPageArtifact(url: "testpage.htm",
7                                 title: "Stoppable Counter")
8
9         counterAgent = StoppableCounterAgent()
10        init-task: UpdateCounter()
11    }
12 }
13 }

```

---

Listing B.12: StoppableCounterOrg.simpal

## B.5 simpAL sources of “What friends most like?” app

---

```

1 role MasterRole {
2     task Boot {
3     }
4 }

```

---

Listing B.13: MasterRole.simpal

---

```
1 role WorkerRole {
2     task LoadLikes {
3         input-params {
4             friendIds: java.util.List;
5         }
6     }
7 }
```

---

Listing B.14: WorkerRole.simpal

---

```
1 usage-interface FacebookInterface {
2     operation queryFriends(friendIds: java.util.List #out);
3     operation queryLikes(friendId: String,
4                           likes: java.util.List #out);
5 }
```

---

Listing B.15: FacebookInterface.simpal

---

```
1 artifact FacebookArtifact
2     implements FacebookInterface
3 {
4     accessToken: String = "AAACEdEose0cBAEdvFeZBQ4GCs5YTvG"
5                           + "vwyqL9gvE9Kwer0wRwkcQqjpfqmvhb"
6                           + "CFZBMd4rKaBKMrtpC8y91dz0HkisMmC"
7                           + "z5hrUVZCbQyW06YNMfTwn0k";
8
9     init {
10    }
11
12    operation queryFriends(friendIds: java.util.List #out) {
```



```

13     result : simpal.web.json.JSONObject =
14         simpal.web.http.HTTPClient.getJSON(
15             "https://graph.facebook.com/me/friends"
16             + "?access_token=" + accessToken);
17
18     friendIds = new java.util.ArrayList();
19     i: int = 0;
20
21     while (i < result.getJSONArray("data").length()) {
22         friendIds.add("" + result.getJSONArray("data")
23             .getJSONObject(i).getLong("id"));
24         i++;
25     }
26 }
27
28 operation queryLikes(friendId: String,
29                     likes: java.util.List #out) {
30     result : simpal.web.json.JSONObject =
31         simpal.web.http.HTTPClient.getJSON(
32             "https://graph.facebook.com/" + friendId +
33             "/likes?access_token=" + accessToken);
34
35     likes = new java.util.ArrayList();
36     i: int = 0;
37
38     while (i < result.getJSONArray("data").length()) {
39         current : simpal.web.json.JSONObject =
40             result.getJSONArray("data").getJSONObject(i);
41         likes.add(new fbtoplikes.LikeEntry(
42             current.getString("id"),
43             current.getString("name"),
44             "https://graph.facebook.com/" +
45             current.getString("id") + "/picture"));
46         i++;
47     }
48 }
49 }

```

---

Listing B.16: FacebookInterface.simpal

---

```
1 usage-interface ReportBuilderInterface {
2     obs-prop htmlOutput: String;
3
4     operation addLikes(newLikes: java.util.List);
5 }
```

---

Listing B.17: ReportBuilderInterface.simpal

---

```
1 artifact ReportBuilderArtifact
2     implements ReportBuilderInterface
3 {
4     likeCache: fbtoplikes.LikeCache;
5
6     init {
7         likeCache = new fbtoplikes.LikeCache();
8     }
9
10    operation addLikes(newLikes: java.util.List) {
11        likeCache.put(newLikes);
12        htmlOutput = likeCache.getHtmlOutput();
13    }
14 }
```

---

Listing B.18: ReportBuilderArtifact.simpal

---

```
1 org-model FBTopLikesOrgModel {
2     workspace main {
3         fb: FacebookInterface
4         reportBuilder: ReportBuilderInterface
5         page: WebPageInterface
6
7         master: MasterRole
8         worker1: WorkerRole
9         worker2: WorkerRole
10    }
11 }
```

---

Listing B.19: FBTopLikesOrgModel.simpal

---

```
1  org FBTopLikesOrg
2      implements FBTopLikesOrgModel
3  {
4      workspace main {
5          fb = FacebookArtifact()
6          reportBuilder = ReportBuilderArtifact()
7          page = WebPageArtifact(url: "testpage.htm",
8                                  title: "Facebook top likes")
9
10         master = MasterAgent() init-task: Boot()
11         worker1 = WorkerAgent()
12         worker2 = WorkerAgent()
13     }
14 }
```

---

Listing B.20: FBTopLikesOrgModel.simpal