PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

IMPLICIT INTERFACES AS A DYNAMIC ADAPTATION STRATEGY IN FRAMEWORKS

JOHN OWEN ATALA

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
JAIME NAVÓN C.

Santiago de Chile, January 2010

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

IMPLICIT INTERFACES AS A DYNAMIC ADAPTATION STRATEGY IN FRAMEWORKS

JOHN OWEN ATALA

Members of the Committee:
JAIME NAVÓN C.
ANDRÉS NEYEM
LUIS GUERRERO B.
IGNACIO LIRA C.

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, January 2010

A mi familia

ACKNOWLEDGEMENTS

I thank all the people that, with their help, advice, guidance and encouragement enabled me to complete this work on time. There were many who supported me during this long process, and I am grateful for that.

First, I would like to thank my advisor, Jaime Navón. Without his advice, I would not have even entered the Masters program. Without his guidance, I do not know if I would have been able to get somewhere with my ideas. Without his support, I definitely could not have finished this.

Second, I would like to thank my dear friend Raúl Montes for being a constant source of inspiration and motivation to move forward in my work.

Then, I thank Raúl Acuña, who taught me most of what I know about LaTeX and made available to me the source code of his own thesis document, this being a constant guide for me, moreover, being a kind of framework for my thesis.

I would also like to thank my work team at k12, not only for giving me all the time I needed, but also for supporting me in every detail they could.

Also, I thank my family and friends, for pressing me constantly to move forward in my work and specially for their patience and understanding.

Finally, I thank all others who helped me in one way or another.

Contents

List of Figures

ABSTRACT

Web frameworks are becoming a fundamental piece in the development of any web application. They influence not only the development time and effort, but they can also make the difference between a maintainable application or a disposable one.

The experience with frameworks of the last few years shows that some of them are easier to learn and to use by the software developers. This is important not only because it will have an impact in the learning curve, but also because the framework will accompany the application throughout its whole life cycle.

Recently, a new strategy, based on dynamic adaptation, has been proposed so the resulting framework be indeed easy and convenient to use by the developers. The idea is that the framework expose their interfaces in a flexible manner so it can adapt itself to the application.

In spite of some form of dynamic adaptation has begun to be included in modern frameworks and these frameworks are indeed better in terms of flexibility and convenience for the developers, it has be only partially done using specific approaches for each framework.

In this paper we present the implicit interface as a general solution to the dynamic adaptation. In addition, we analyze the key implementation issues that must be considered to build a framework that incorporates this ideas so it can be not only prepared for future evolution but also to do it efficiently.

Finally, we show that these ideas and techniques can be used in other context different to the context of web applications where there is also a layered architecture with one way dependencies.

Keywords: web application, framework, dynamic adaptation, interface

RESUMEN

Los frameworks se han constituido en una pieza fundamental para el desarrollo de aplicaciones web. Ello tiene incidencia en los tiempos y en el esfuerzo de desarrollo y puede hacer la diferencia entre una aplicación mantenible y una desechable.

La experiencia con frameworks de los últimos años muestra que algunos resultan más fáciles de usar por parte de los desarrolladores. Esto es importante no solo porque incide en el tiempo necesario para dominarlo sino que porque además acompañará a la aplicación durante todo su ciclo de vida.

Recientemente se ha propuesto una estrategia para mejorar la experiencia de uso basada en la idea de adaptación dinámica. La idea es que el framework exponga sus interfaces de una manera flexible de modo que pueda adaptarse a la aplicación. A pesar de que esta estrategia ha sido incorporada y ha mejorado la experiencia de uso de los frameworks, sólo se ha hecho con acercamientos parciales y específicos para cada framework.

En este trabajo se propone la interfaz implícita como una solución general a la adaptación dinámica. Junto con ello, también se analizan los aspectos claves de implementación que se deben considerar al desarrollar un framework que incorpore esta idea.

La definición y caracterización de las interfaces implícitas demostró unificar de manera general las estrategias de adaptación dinámicas de los frameworks modernos más usados. Se demostró también que con este enfoque se conservan las ventajas de las soluciones particulares permitiendo además, un desarrollo más eficiente y ágil de nuevos frameworks con un resultado más consistente y mejor para los desarrolladores de aplicaciones.

Finalmente, se muestra que estas técnicas son extendibles mas allá de la Web a otros contextos de software con arquitectura de capas.

Palabras Claves: aplicación web, framework, adaptación dinámica, interfaz

Chapter 1. INTRODUCTION

The Web is a modern phenomenon with deep global social, political and economical implications. The recent developments that make it universally available everywhere at all times will have an even deeper impact in our style of life.

This creation of the humanity that we call the World Wide Web, or simply "the Web", started as a global network of contents presented as sets of web pages. Nevertheless today's Web, specially after the so called Web 2.0 wave (O'Reilly, 2005), needs to be seen with a wider perspective. Web pages are not published buy a few to be read by the many anymore but every user can participate adding new contents or comments to what others publish. Yet more important, the simple hypertext model of a web site was replaced by a much more general purpose application, the web application. Today's web applications, like Gmail for instance, are almost indistinguishable from the regular desktop versions.

The fast rise of the web application can be explained in part by the accelerated evolution of new tools and techniques that facilitate the development of these new kind of applications. Among the new tools the web application frameworks are perhaps the ones with more profound impact in web development. These pieces of software provide a strong basis for the web application to grow upon.

One of the new available techniques related to the rise of frameworks is dynamic adaptation and it has to do with the way the framework and the web application communicate with each other. Traditionally, a framework-based application must strictly follow the obligations that the framework imposes upon it. Dynamic adaptation breaks the strict requirements modality giving more freedom to the application and making the framework to adapt as well.

In this document we propose a new strategy for dynamic adaptation of frameworks and it is organized as follows. In this first chapter introduces the problem and the motivation and provides the context including the state of the art. Chapter 2 includes the material submitted for publication as a journal paper and describes technical details with

detail including the formal presentation of the idea of implicit interfaces as strategy of dynamic adaptation and an implementation prototype as proof of concept. Finally, chapter 3 is dedicated to the conclusions and the future work that could be carried out.

1.1. Web Application Development

What is indeed a web application? To put it in simple terms, a web application is an application that uses the Web and a simple browser to interact with the user. The user sees web pages that provides the needed access point to the functionalities of the application.

A Web application, is different to a desktop application in that most of the associated code executes not on the computer that the user is using to run it but in a remote server machine located somewhere over the internet. The communication between the machines is carried on trough the standard web protocol (HTTP).

Because many of the most important web applications may have thousands of simultaneous users distributed trough the globe, usually there is no one server running the application but by dozens or even hundreds of servers that work together as one.

Web applications have traveled a long way since their beginnings. Today, applications like facebook, twitter or flickr not only exhibit a sophisticated user interface but also enable collaborative work and social interaction.

The fact that every day we see more and more web applications that allow us to carry on many things is not that surprising. What is new is web applications that somehow are being used in scenarios that where previously reserved for desktop applications.

Web applications are taking to the virtual space many common task and services that we used to do in the real world. Not only that but they are also taking a very important role in interpersonal relations and social interactions. In summary the web is every day more the place where to learn, work, share or have fun.

The relevance of the web and the increasing number of web applications present us a new challenge not only because they need to be developed and built at a very fast pace

but also they should be built in a way that can be adapted and extended in the future. This new need takes us directly into web application architecture issues.

The first web applications were built to solve a specific existent need and not many considerations of what was going to happen in the future. They were in some sense disposable applications. Today's applications need to be designed to evolve and adapt according to new needs.

Software engineers have known for years that software should be designed and built thinking not only in present needs but also in the future. It is also a well known fact that lots of money and effort is spent in software maintenance. But the web scenario and the web applications take this to a new dimension. Many times the web application needs to be changed immediately after their release or even before they are released. Very often there is a no precise estimation of the number of users or how the users are going to use the software. To make things even harder, web applications usually are developed in a very short time frame and with hard time deadlines.

The new challenges of web applications are being addressed by a new discipline: Web Engineering (Murugesan, Deshpande, Hansen, & Ginige, 2001). It involves diverse areas including human-computer interaction, user interface, system analysis and design, requirements engineering, hypermedia, information architecture as well as social sciences and graphic design. The wide variety of areas involved talks about the complexity of the task and the need for specialization.

Because the engineering of a web application is a difficult task it seems that a good idea would be not to do it all the time from scratch but start with a basic structure already built according to the best web engineering principles and techniques. Enter the web application framework that fulfills precisely this role and this is why these pieces of software have been gaining in popularity in the last few years.

A web application framework is a piece of software that can not serve as an application by itself but can be customized and extended easily to build a web application. Because the framework is built once and is going to be used for many applications a lot of effort

is invested in doing a great job in terms of web engineering, particularly in the software architecture that will at the end determine how easy is going to be for the web application programmer to use it to build the real web applications.

1.2. Web Frameworks

We go deeper here into web application frameworks in terms of what is exactly a framework, why they are so important in modern web applications and finally what are the things that can make one more or less difficult to learn and to use by the web developer.

First, we need to distinguish that there are several types of software frameworks, from conceptual frameworks to platform frameworks, among others (Shan & Hua, 2006). Here, we do not reffer to such a wide open concept of framework, but to a more specific kind: web framework. According to Shan and Hua (2006) a web framework is "a reusable, skeletal, semi- complete modular platform that can be specialized to produce custom web applications (...). It includes building blocks of services and components that are essential for constructing sophisticated feature-rich business service and collaboration systems". Figure 1.1 shows this idea graphically: the web application is built as a custom developed part resting on top of the web framework.

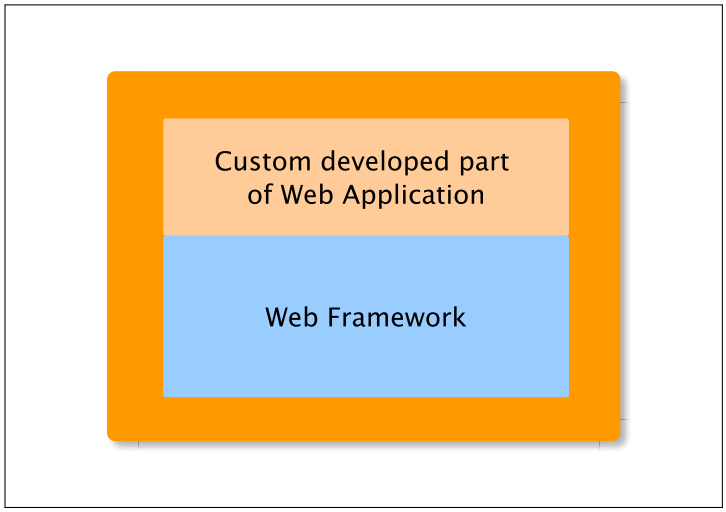

Figure 1.1: Architecture of a web application using a web framework.

It is perhaps always a good idea to use a framework to build a software application because one can complete the task in less time. Nevertheless, in the case of web application it takes supreme relevance because, as we have said before, a web application typically will have to be modified and extended many times even before the first release. Furthermore, web applications are hard to build and to debug later because the server stateless model and because the view component must be clearly isolated from the rest of the application. As Shan and Hua point out "a framework significantly reduce the amount of time, effort, and resources required to develop and maintain web applications" (Shan & Hua, 2006, Why use Web Application Frameworks).

In a web framework most architectural decisions have been already taken so the task to engineer the application in a proper manner is significantly reduced. Because the framework must be of a general purpose it usually include many features that will not be used in some applications. This introduces a new challenge: how to build a powerful framework that covers a wide variety of scenarios and at the same time be easy to learn and easy to use by the web application developer.

Experienced web application developers know that learning a new framework may take more time than the time needed to develop the application itself. If this is true one may well question the idea of using a framework if the application that is going to be developed is just one. The answer is categorically yes since the extra cost involved in learning the framework will be paid back later when the need for extensions or modifications of the application appear. Furthermore, because a web application that is built over a framework has a lot less code that one that is not the simple task of keeping it up to date is easier. All the code that belongs to the framework, which is usually the largest part, is taken care by the framework creators. Finally, another good reason to use a framework is that it force best practices so even inexperienced programmers can produce reasonable good quality code.

Once we have learned the framework, the time to develop applications with it reduces in comparison to do it without a framework. The amount of the reduction however depends on the framework characteristics. Each aspect provided by the framework can be considered a saving. It is code that is not going to be designed, written, tested, documented and maintained. Depending on the framework, aspects as architecture of the solution, data structures, utility libraries, session management, data persistence, web services, etc. may be already solved.

1.3. Dynamic Adaptation

At this point it should be clear that using a web framework can be very useful and has many advantages. This is specially true if we consider a medium to large time frame. At the same time we have said that frameworks can be hard to learn and master because they must be prepared for a wide range of situations and scenarios. We propose in this work a general strategy that may help to build more programmer - friendly frameworks.

A new approach called dynamic adaptation may be considered an important contribution to the solution of this problem. Traditionally, a web developer must known at least the framework architecture and fundamentals before even thinking in start building applications with it. Dynamic adaptation allows the programmer to start using the framework as soon as possible, in the ideal case immediately. The idea is to get the framework to analyze de application code and adapt itself to the developer instead of forcing the developer to adapt to the framework restrictions.

Under this new approach in an ideal scenario the developer gradually discovers the framework characteristics as the need arises instead of having to learn everything beforehand. This ideal scenario is the goal of dynamic adaptation and there are several strategies for getting close to it.

One of the possible dynamic adaptation strategies is the one proposed by Chiba (2005) consisting in tagging key elements of the application that are recognized later in a translation phase to the elements that the framework needs. Under this strategy, it is the translated application the one that is coupled with the framework to get the final executable web application. The original proposal uses the simple Java annotation facilities to tag the application.

Another strategy for dynamic adaptation is the one that is being used in modern frameworks like Struts 2 (Struts Development Team, n.d.) o Tapestry 5 (Ship, 2009). In this case the framework continue imposing its general guidelines to the application but giving some freedom that facilitates the framework appropriation by the developer. This freedom produces certain variations that are interpreted by the framework either during application loading or at runtime.

In spite of the efficacy of dynamic adaptation strategies in facilitating the use of these new frameworks by web application developers it has yet to be conceptualized and formalized. New frameworks simply incorporates what seems to be working for other frameworks without even realizing that some kind of dynamic adaptation is the responsible of the magic. It would be useful and important to have a unifying general strategy and we believe that we have made a modest contribution to this goal.

Our work has indeed two main objectives:

(i) to create awareness about the importance of dynamic adaptation as a transversal general mechanism
(ii) to present a new concept that we call implicit interfaces as a general strategy for dynamic adaptation in frameworks

Some secondary objectives include:

(i) to conceptualize the existing communication mechanisms between the framework and the web application

(ii) to identify common techniques and mechanisms currently used for dynamic adaptation

(iii) to identify the key aspects related to framework implementation related with implicit interfaces or any other form of dynamic adaptation

1.4. Implicit Interfaces

As we said above, dynamic adaptation is a known way to get closer to the goal of a framework easy to use. The current strategies for dynamic adaptation are however either limited, incomplete or inexistent. Our hypothesis is that a more clear and well understood strategy would impact in making dynamic adaptation more popular among frameworks and also in making the frameworks easy to use. Our goal is a strategy for dynamic adaptation that is easy to understand, complete and uniform.

1.5. Framework development

To properly apply the framework modifications that we propose, we need first to review how framework development is done. Roberts and Johnson (1996) describe it as a set of patterns within a temporal context. They are "Three Examples" pattern, "White-box Framework", "Component Library", "Hot Spots", "Pluggable Objects", "Fine-grained Objects", "Black-box Framework", "Visual Builder" and "Language Tools" patterns.

Because the nature of Web Applications and Web Frameworks, some of these patterns, like Visual Builder or Language Tools, usually do not apply since they are unpractical and appears too late – web applications and frameworks usually evolve too fast so such patterns are achieved after framework has changed or worst, never.

At the other hand, almost any Web Framework is released as a Black-Box Framework pattern, with Component Library, Hot Spots and some Fine-grained Objects. Also Pluggable Objects pattern is very common on Web Frameworks.

The main framework modification to implement the proposed dynamic adaptation strategy relies on including implicit interfaces. They should appear in an early stage of

development, changing the framework starting with the White-Box Framework pattern, then Component Library pattern and so on.

These patterns will not change from a conceptual point of view, but they would require modifications on its details. As an example, Black-Box Framework pattern solution says that "use inheritance to organize your component library and composition to combine the components into applications" (Roberts & Johnson, 1996). Now, not only composition will be used to combine them, but also dependency injection (Fowler, 2004) should be considered as an alternative.

1.6. Java and the Java Virtual Machine (JVM)

Because most of the concepts as well as the implementation proof of concept is under the Java platform it may be important to review some key aspects of this important platform. The reader might want to skip this section if he is familiar with the Java 2 platform standard edition (J2SE) 5.0 or superior and the Java Virtual Machine (JVM) or at least with Java interfaces, Java annotations, class loaders and dynamic class changing via bytecode modification.

We start with Java Interfaces . As it occurs in many programming languages, in Java an interface defines a group of methods that must be provided by any class that says that implements this interface. An interface has similar characteristics to a Java class but it also has some limitations:

(i) interface methods can only have modifiers public and abstract. It can not declare methods that are private, protected, final or static. Interface methods are always abstract and public no matter the modifiers.

(ii) An interface can have properties but they must be always public, static and final.

(iii) An interface can have internal classes and the only restriction is that they must be public and static. In other words, an internal class can be abstract or final.

Figure 1.2 shows a few examples that better explain the limitations just described.

```java
package cl.example;

public interface ExampleInterface {

    static int propertyA = 0;
    public static int propertyB = 1;
    final static int propertyC = 2;
    public final static int propertyD = 2;

    void methodA();
    public void methodB();
    abstract void methodC();
    public abstract void methodD();

    class InnerClassA {
        // ...
    }
    public static class InnerClassB {
        // ...
    }
    public static final class InnerClassC {
        // ...
    }
    public static abstract class InnerClassD {
        // ...
    }

}
```

Figure 1.2: Valid properties, methods and classes in a Java interface

Another important element of the Java platform is the annotation mechanism. Java annotations are available since version 5.0 of the J2SE specification. They are special tags that can appear before all modifiers of a class, interface, property, method or parameter. They are recognized by the special character @ at the beginning which is followed by the name of the annotation. Figure 1.3 shows various examples of the use of annotations.

Annotations may include parameters but these are limited to primitive types (Strings, enums and arrays of these with values known at compile time). Arguments appear as a comma separated list of elements in which each one corresponds to a pair “key = value”

```
package cl.example;

import cl.john.tea.annotations.*;

@SuppressWarnings("unused")
public class AnnotationsExample {

    @InjectService
    private String propertyA;

    @AfterRender
    public static void main(@InitParam String[]
args) {
        //...
    }

}
```

Figure 1.3: Examples of possible annotations

or just a value if the key is "value". See figure 1.4 for a few examples of annotations with arguments.

A final aspect specific of the Java platform that is important to understand our proposal is related to the Java Virtual Machine (JVM) and the runtime modification of bytecodes (Sosnoski, 2003).

In very simple terms, a Java program consist of a group of binary files (.class). Each of these files has information that can be loaded as an object of the class `java.lang.Class`. A java program works loading these class objects and executing the sequence of bytecode (this is indeed what we called binary in a java file) in each of them.

In contrast to other languages, in Java classes are initially disconnected and they are loaded only when there is a explicit reference to them. There are three main scenarios where a class is loaded (if the class is not already loaded). The first one is when there is a reference through the `new` operator as for example:

```
package cl.example.entities;

import java.util.Date;
import javax.persistence.*;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Document {

    @Id
    @GeneratedValue
    private int id;

    @Column(unique=true)
    private String name;

    @Column(length=16000,nullable=true)
    private String value;

    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    @ManyToOne(optional=false,cascade=
{CascadeType.MERGE,CascadeType.PERSIST})
    private DocumentExtensionRegistry extension;

    public Document() {
        creationDate = new Date();
    }
    //...
}
```

Figure 1.4: Use of arguments in annotations

```
Figure f = new Square()  -- loads class Square
```

A second case is when the class is statically referenced as for instance:

```
DriverManager.getDriver()  -- loads class DriverManager
```

Finally a class may get loaded when is explicitly referenmced as in:

```
Class.forName("com.example.Circle")  -- loads class Circle
```

Loading of the classes is carried out by a special class called the class loader. The JVM has a few class loaders but an application can add its own class loaders. In general a class loader is simply a class derived form `java.Lang.ClassLoader` that keeps a list of the classes that have been loaded and a reference to a possible class loader father. The fact that the class loaded by a specific class loader belongs to him has important implications.

Because each class loader has a reference to a father class loader a complete tree of class loaders in which the root corresponds to a JVM class loader may be involved. Figure 1.5 shows part of a class loader tree for Apache Tomcat (Sosnoski, 2003).

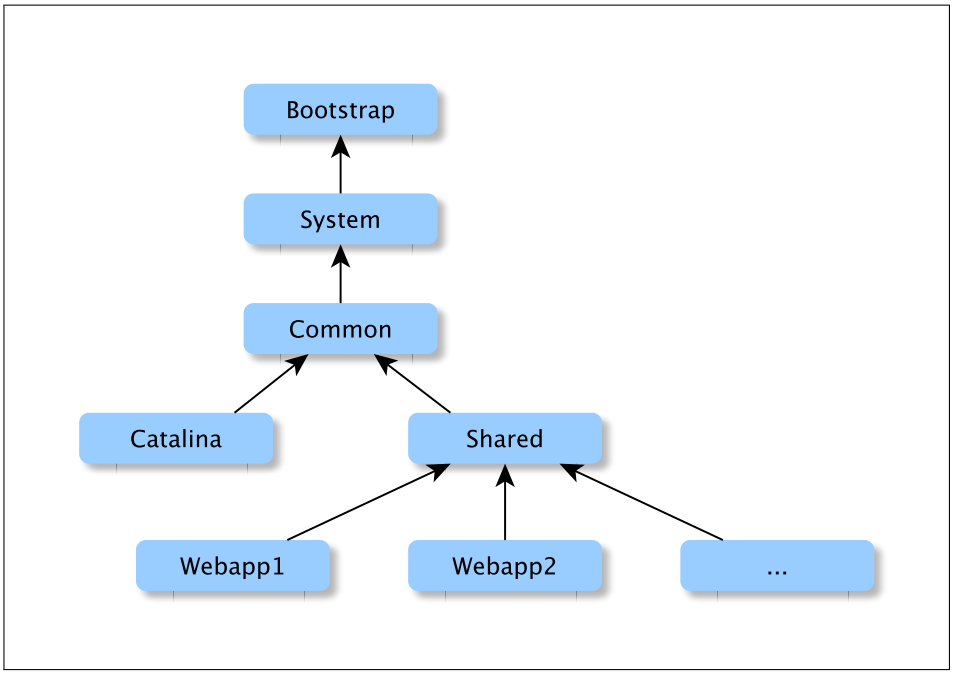

Figure 1.5: Tomcat class loaders

The class loader tree helps us understand what happens when a class is required. The request is received by a specific class loader or this class loader delegates the task to his father. The result is that a class loader can resolve any class request that has loaded himself or any of his ancestors but he does not know anything about his descendents.

Because of this, a class loader typically works requesting the load to the father unless he had loaded it previously. As the class is loaded really by the father but is he who delivers it resolving it as one of its own.

All these aspects must be considered when a new class loader is going to be included. Let's see now how a new class loader can be used to modify the classes in runtime (Sosnoski, 2004). It works as follows:

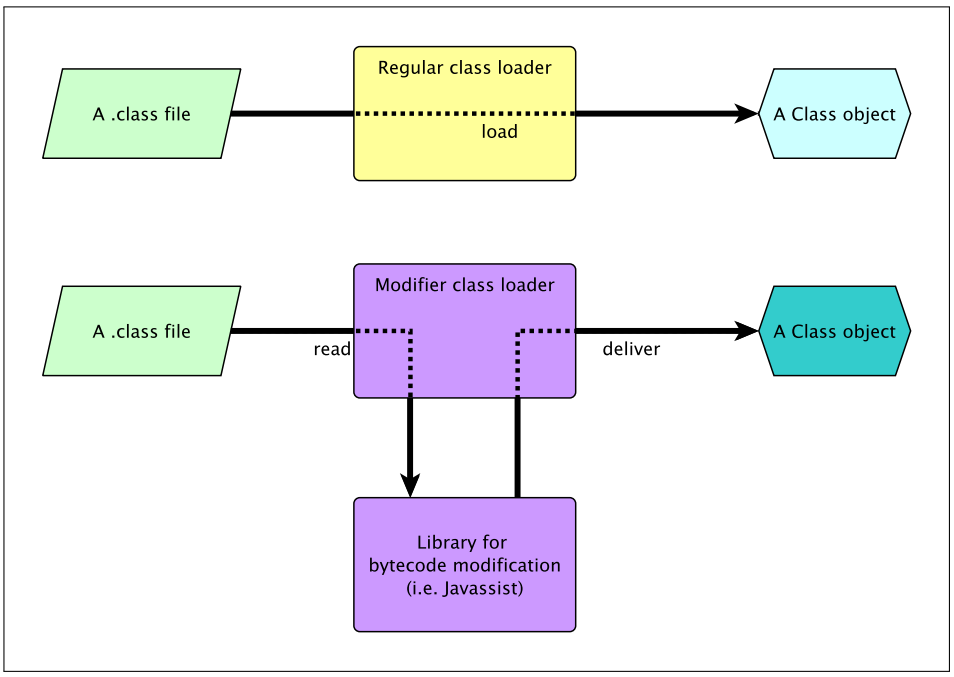

Figure 1.6: The work of a regular class loader compared to the class loader that makes runtime changes to the bytecode

(i) The new class loader must be completely self sufficient and must load classes without delegation. This is necessary because we don't load just the binary (.class) as a regular loader. Figure 1.6 compares this class loader with a regular one.

(ii) The new class loader must not load the class directly but just read the binary .class. The help of external libraries that can read and interpret a .class file might be useful. For example we used the Javassist library (Chiba & Nishizawa, 2003)

that also allows high level bytecode modification, but other libraries that allow working directly with the bytecode like BCEL (Dahm, 2001) can also be used.

(iii) With the binary read and the help of the library we proceed to modify the representation of loaded class.

(iv) Finally, with the modifications already done we get a class object of the modified class, which is delivered as if it was the class originally loaded. Note that for the rest of the Java environment it looks like if the delivered class is the original but we know that the class in memory is different to the class binary that we read. It was changed by the class loader.

Chapter 2. IMPLICIT INTERFACES AS A DYNAMIC ADAPTATION STRATEGY IN FRAMEWORKS

The following chapter is a paper, submitted for publication in the Journal of Web Engineering.

The use of frameworks (Shan & Hua, 2006) to develop web applications has become an established requirement. In some cases, like Ruby on Rails (Bächle & Kirchberg, 2007), the whole platform becomes popular thanks to the availability of a powerful framework. The success of Rails has pushed the development of similar frameworks for other platforms.

There are good reasons to base a web application development on a framework (Shan & Hua, 2006, Why use Web Application Frameworks). It allows a faster development because the framework provides the basis of the application: software libraries, architecture and solutions to many common recurrent problems. Later on, these elements will be crucial in how modular decoupled and extensible the web application will be.

If there are good reasons to use a framework and some framework aspects are crucial in some desirable properties of the application, it makes sense to study some of the key issues involved in how well or bad a given framework will fulfill its duty. Most application developers, the true users of frameworks, will agree in that the mechanisms and facilities that allow communication between the application and the framework are very important. In fact we believe that indeed this is the key issue.

Some of the new frameworks have begun to leverage a group of techniques that include name conventions, and dynamic class modification to produce a framework that can adapt itself to the needs of the programmers. It makes a lot of sense that a framework be able to adapt instead of just the programmers to a rigid framework design.

Some of the most popular frameworks communicate with the application through explicitly defined interfaces. We revise here how this works, and we explain why this approach is not a satisfactory solution.

2.1. The role of the interface

At first sight, it seems reasonable that the communication between the framework and the application occurs through interfaces that must be defined in the framework. At this point, it is convenient to remember what is really an interface. Sullivan et al. (2005) defines interface as "an agreement about properties that an element B should have and that other elements, such as A, may depend upon". So for instance, a Java interface is a series of abstract method declarations and a class B is said that implements such interface when it has methods that match those of the interface. This is an example of what is known as explicit interface.

Explicit interfaces are not the only way an interface is defined. Sometimes the interface is just a group of public class properties and methods that are provided by the framework so the application can use them.

In the case of explicit interfaces some of them may be implemented by the framework itself. In this case, from the application point of view, this is similar to a class interface. Some explicit interfaces, however, are expected to be implemented by the application. We will distinguish these two cases as function interfaces and option interfaces.

In function interfaces, the framework provides functionalities to the application using interfaces, so the application can assume that the framework has some properties that it can use. In this case, the framework provides either an internal class implementation of an explicit interface or simply a class. For the application the way the functionality is provided makes no difference, it is simply a function interface.

In option interfaces, the framework gives the application certain responsibilities and options through interfaces that it needs the application to implement. Again, it doesn't matter if the application must implement an explicit interface or if it has to simply extend a class, in both cases we talk about an option interface. There are two situations that it may be useful to distinguish: in the first, a complete functionality depends on the application implementing the interface, in the second one, the functionality may exists already but the framework provides a way to extend or alter a default behavior. An important

example of an option interface where functionality depends on implementing an interface is a component-based framework (Shan & Hua, 2006). Here the framework expects that classes in the application implement the provided component interface to be considered a component.

2.2. Option Interfaces

We explain here why explicit interfaces are not always a reasonable approach. In particular, explicit interfaces work quite well as function interfaces but they are terrible as option interfaces. To better understand this, a deeper analysis of option interfaces is needed.

As we said before, sometimes, the framework uses an interface to present options to the application to obtain a functionality, to alter the default behavior, or, to extend a functionality. In other words, to do something that the framework does not do, although it acknowledges its existence. From the application point of view, the meaning of this is that the application flow will reach a point where the framework will do nothing although it realizes that something could be done. At this point the application tells the framework what to do.

Another way to see it is through the sequence of method calls. At some point the framework calls a method that does nothing unless the application provides an equivalent corresponding code and this equivalence is found precisely via the explicit interface. The framework knows that method B on interface I is equivalent to default method A so if the application provides the code, the framework can call it instead of A. We can see that, in this context, the real purpose of the explicit interface is to tag a set of methods for the framework. It is the framework the real user of the explicit interface and not the application as we could have thought.

So option interfaces belong to the framework who needs them to understand something unknown of the application. They are not created for the application but for the framework itself.

2.3. Why explicit interfaces as option interfaces are a bad idea

There are two categories of problems: constructive problems and flexibility problems. Constructive problems refer to the way the explicit option interface should be built. For the framework the interface is a set of related methods and all of them will be implemented in some class. But from a specific application point of view. although only a few of them may be interesting, it is forced to implement all. For this reason is very common to see several null methods which is not only useless extra work but also it often leaves inconsistent return values (null, 0, -1). Of course the framework documentation should specify this kind of thing but it is not always available or at hand for the application programmer.

An additional constructive problem is that there is no way for the explicit interface to express dependencies between methods. This information, again, is only available through the framework documentation.

So explicit interfaces, that are supposed to expose or to make explicit the relationship framework-application, fail in this respect. They may look programmatically complete, but they do not include all the information needed to fulfill their role as interface.

Let's examine now the second category of problems, the lack of flexibility. In fact, this represents a severe limitation of explicit interfaces. It has to do with the fact that the methods that implement a given interface must correspond not only in terms of the name of the method but also parameters, return values and throwable exceptions. This eliminates the possibility of using name conventions, or passing just the required parameters.

2.4. Analysis of some popular frameworks

We examine here some of the most popular web application frameworks to see how they solve the above mentioned problems.

2.4.1. Apache Struts

Struts (Li, Ma, Feng, & Ma, 2006) is a request-based framework (Shan & Hua, 2006). It was the Java platform best option for several years and still one of the most popular

Java frameworks. This framework uses action classes as controllers. An action class must extend class `org.apache.struts.action.Action` and override its execute method returning the name of the next element to have the control (usually a JSP page).

In this case, the framework requires not only the implementation of an interface, but also to extend a particular class just to make the call to execute to work. To further complicate things, the actions must be declared in a special configuration file.

Struts has improved things in version 2. Now the action class just implements the interface Action that has a single method called execute. Furthermore, it is even not necessary for the class to declare that it implements the Action interface, it just need to provide the action code. It is also possible to have multiple actions in the same class, in fact, any method without parameters that returns a String can be considered an action.

Some name conventions appear also in Struts 2. A save action not only can be executed by the corresponding save, but also for a doSave method. Name convention is a powerful additional communication mechanism between the framework and the application.

2.4.2. Apache Tapestry

Tapestry (Ship, 2009) is a component-based framework (Li et al., 2006) for the Java platform. In version 4 the page was the primary element which is composed of a HTML template and a Java class (in what follows we will use page and the corresponding Java class as synonyms).

To be considered a page, a Java class should implement the interface `org.apache.tapestry.IPage`. However, due to the complexity of the IPage interface, a typical page will just extend `org.apache.tapestry.html.BasePage`. This needs to be done even when we do not override any of the extended methods. In many cases the provided implementation of BasePage is enough. The mechanism is present just in case we need to override default behavior.

In Tapestry 5 the pages are simple POJOs (Plain Old Java Object) which means they do not need to implement interfaces or extend classes. Nevertheless, in spite of not implementing any interface, there are many methods that by virtue of name convention are going to be interpreted by the framework. For example, a method "onActivate" will be called each time the page is activated simply because the name of the method is that. Any event will be captured by a method whose name begins with the prefix "on" followed by the name of the event. Moreover, it is possible to use a Java Annotation on any method to capture an event no matter its name.

Besides using name convention and annotations, Tapestry introduces some freedom in parameters and return values of the methods. For example a method onActivate may return a void, the name of a page (String), the class of a page (Class), etc. Also it can receive parameters and the method is called when the list of parameters can be filled with the ones passed in the URL.

In fact, Tapestry has found solutions to several of the problems associated with explicit interfaces and somehow it was a source of inspiration for our proposal.

The frameworks we examined reinforced the idea that explicit interfaces used as option interfaces introduce many problems because it is hard to built them in a complete and appropriate way and because the resulting product lacks the needed flexibility. The range of solutions we have seen in the previous frameworks include the following:

(i) simplification of the explicit interfaces (making them just an optional guide)
(ii) use of name conventions
(iii) use of the annotation facilities

2.5. Welcome to implicit interfaces

We propose a new approach to solve the communication problems between the framework and the application: the implicit interface. This new type of interface defines the communications with well known methods in an implicit way, that is with no explicit presence in the implementation code.

The main difference with a explicit interface is that instead of defining methods in code to be interpreted by software, it defines methods in the documentation to be interpreted by software developers. This allows the implicit interface to define more expressive methods because they are defined by their characteristics rather than by the exact declarations. It also allows to define properties. Consequently, the framework can find them searching for those characteristics and software developers have more freedom.

2.6. Specification of implicit interfaces

An implicit interface is an abstract specification of methods and properties that will be interpreted and used by the framework . The specification covers things that are normally described by an explicit interface such as return values, names of the methods, parameters and exceptions but in a different, more dynamic manner:

- return values as a list of possible types and their meaning posibly grouped under some criteria
- name of the methods as expressions and meanings (for example a prefix followed by a keyword)
- parameters not as the ones the method will receive but as the parameters that are available to be delivered to the method (specification includes type, whether it is required or optional and a way to capture it)
- throwable exceptions as a list of exceptions that will be captured

With this specification the implicit interface is at least as expressive as an explicit interface. Limiting the interface to just one return value, literal expressions for the name of the methods, only required parameters and no other exceptions besides those declared, the implicit interface behaves exactly like an explicit one.

But implicit interfaces go much further. For instance, methods may specify:

- required visibility (implementation may be private, protected or package)
- execution context (may specify static methods)

- annotations (these may override other specifications)

Implicit interfaces can do even more. They can specify searchable properties including:

- type (may include a list of possible types and grouping)
- name as expression and meaning
- visibility
- context (static or instance)
- annotations

The remaining question is perhaps where all these specifications can be located and how they could be presented in a unified and global way.

There is no unique answer to that question. For instance we could use a simple text file that describes everything. We suggest using automatically generated documentation like javadoc (*Javadoc Tool Home Page*, 2004) and a basic, commented, explicit interface specification. This interface is not part of the code; it is used only in the generation of the documentation of the complete specification.

Note that due to limitations imposed by the explicit interface, the example methods will be always public and the properties will be public and static even when in the real implicit interface that will not be the case.

2.7. A framework with Implicit Interfaces

As a proof of concept we developed the Tea Micro Framework which uses implicit interfaces. We present first a brief description of the framework and then we distillate the key aspects so they can be ported to any other framework.

2.7.1. The Tea Micro Framework

First of all we should say that Tea is not a complete framework, hence the qualifier micro. Although Tea does not have all that a modern web application framework usually has, it provides the basis of the application by giving support for multiple pages and services, enough for the web application to operate.

We believe that a micro framework is an ideal vehicle to test the potential of implicit interfaces because it is simple enough to allow focus in the relevant issues.

The nucleus of Tea is simple: there is a filter (Alur, Crupi, & Malks, 2001, Intercepting Filter pattern) which has an instance of TeaApplication that captures the requests and pass them to the application. TeaApplication checks if it has a page that matches the URL of the request. In this case the request is processed by that page.

Tea is service-oriented. There is a service registry (TeaRegistry) where the services are stored as a pair implicit interface and corresponding implementation class. When a service is needed, the registry is responsible of instantiating and initializing the service.

Every page is instantiated and initialized by a service (the PageManager) and they have access to other services. These services are provided to the pages and to other services in a transparent way via dependency injection (Fowler, 2004) so the framework load, instantiates and initializes almost every object. Moreover, Tea keeps tracking of the services and pages during their whole lifecycle.

In the case of services, the lifecycle begins when the service is declared, joining an interface with its implementation; at this stage the service implementation is analyzed to be ready to instantiate it when required. Later on, the service is delivered as a proxy of its interface and when one of the methods is called the implementation is instantiated as the proxy back end.

For pages, the lifecycle begins when the application starts (filter initialization) — Pages are searched in a special package (any class in a application subpackage *pages*) where any class is considered a page class. Each page class is analyzed and transformed

by the use of runtime bytecode modification with Javassist (Chiba & Nishizawa, 2003). At this stage the pages are indexed by their URL of request. Later, when the page is needed the framework instantiates one of these modified classes and injects in it all the resources (as services). Then the control is passed to the application calling the methods of the page. When the response to the requests ends the page object is discarded.

2.7.2. Implicit Interfaces in Tea

The main implicit interface in Tea is that associated to the page class. A page class is any class located in the subpackage pages. It does not need any special property or method to be considered page class.

As we explained before, the implicit interface is used to generate documentation (javadoc) We show now how a method and a property are declared and the resulting javadoc that is generated.

As an example, we take the method before render, which gets called before rendering the template of the page. Figure 2.1 shows the implicit interface declaration of the method and figure 2.2 shows the generated javadoc . The method is recognized by its name “beforeRender” but it can be named freely if we use the BeforeRender annotation.

```
/**
 * Before render method will be called before rendering template. If there
 * is no template for this page, no render will occurs, but anyway this
 * method will be called. A PrintWriter is available for this method to
 * write before anything of template rendering is writed in.
 * @ii.return Any return value will be ignored.
 * @ii.name Exactly "beforeRender"
 * @param writer A PrintWriter to write content as response before template
 * rendering
 * @ii.parameters Only a PrintWriter is available as method parameter.
 * @ii.exceptions Throwing exceptions is not allowed.
 * @ii.visibilities Any access level.
 * @ii.context Must be an instance method.
 * @ii.annotations Optinally a
 * {@link cl.john.tea.annotations.BeforeRender @BeforeRender} annotation
 * can be used to name the method freely.
 */
@BeforeRender
void beforeRender(PrintWriter writer);
```

Figure 2.1: Example method of Page implicit interface declaration.

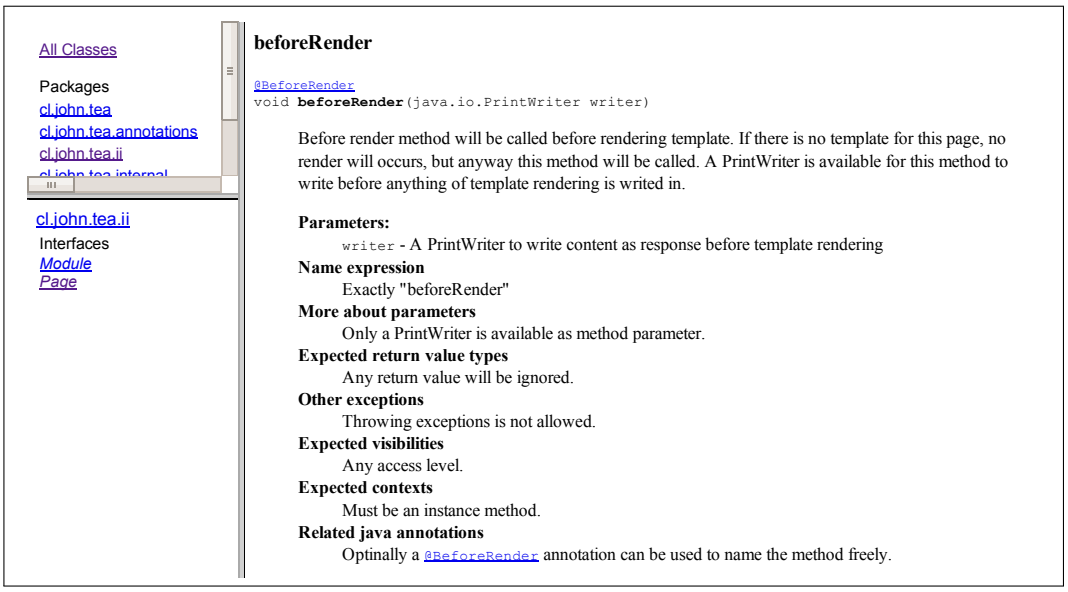

Figure 2.2: javadoc of example method of Page implicit interface.

As an example of the treatment of a property in the implicit interface, we take a service property. This is a property used to get access to an injected service and its type is always the interface of a service. In runtime a service of the same type will be injected before any execution on that page. The implicit declaration is shown in figure 2.3 and the corresponding generated javadoc in figure 2.4. Due to explicit interface limitations, all properties are static and with a fixed value but that has no effect in the real implicit interface.

2.8. Key Issues

The key in a successful implementation of implicit interfaces is understanding how it works. The main “trick” is that the framework takes the class implementing the implicit interface and modifies it in runtime. There are however related issues that are not self evident: class loaders, dependency injection and an optional explicit interface in the back.

About class loaders, Java has a rich class loader tree and web applications use multiple class loaders. Runtime class modification adds even more class loaders. It is important to remember that a class is only recognized by the class loader that loaded it and its children.

```
/**
 * A service object. Object class must be some built-in or provide by
 * application service interface. At runtime, before activation of page
 * every instance object marked with
 * {@link cl.john.tea.annotations.InjectService @InjectService} marker
 * which class type is a properly declared service will be injected in.
 * If the class of this object it's not an interface or it's not a service
 * interface, then an error will occur when application starts.
 * @ii.type Any interface of built-in or provided service.
 * @ii.name Any name.
 * @ii.visibilities Must be private.
 * @ii.context Must be an instance object.
 * @ii.annotations Requires a
 * {@link cl.john.tea.annotations.InjectService @InjectService} marker.
 */
@InjectService
public static Object someService = null;
```

Figure 2.3: Example property of Page implicit interface declaration.

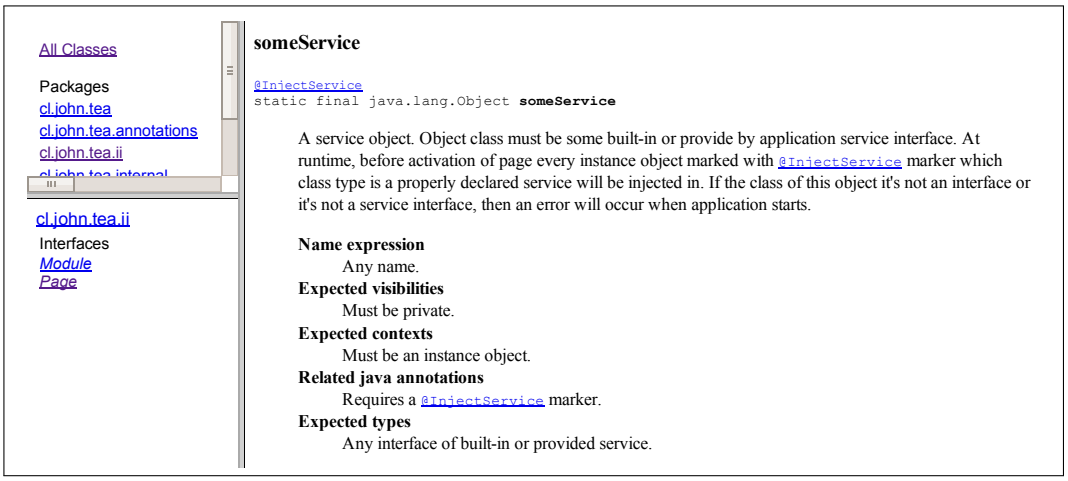

Figure 2.4: javadoc of example property of Page implicit interface.

Classes loaded in sibling loaders will be considered different no matter if they have even exactly the same bytecodes (Sosnoski, 2003) . Also, classes loaded in parent loaders will not be able to see them. Figure 2.5 shows an example tree of a web application class loaders, where only white classes (those that asociated with A class loader or its descendents) can access class A.

One problem we had to face is how to ensure that classes be loaded by the framework class loader and not by other class loader as the associated to the server (probably the same

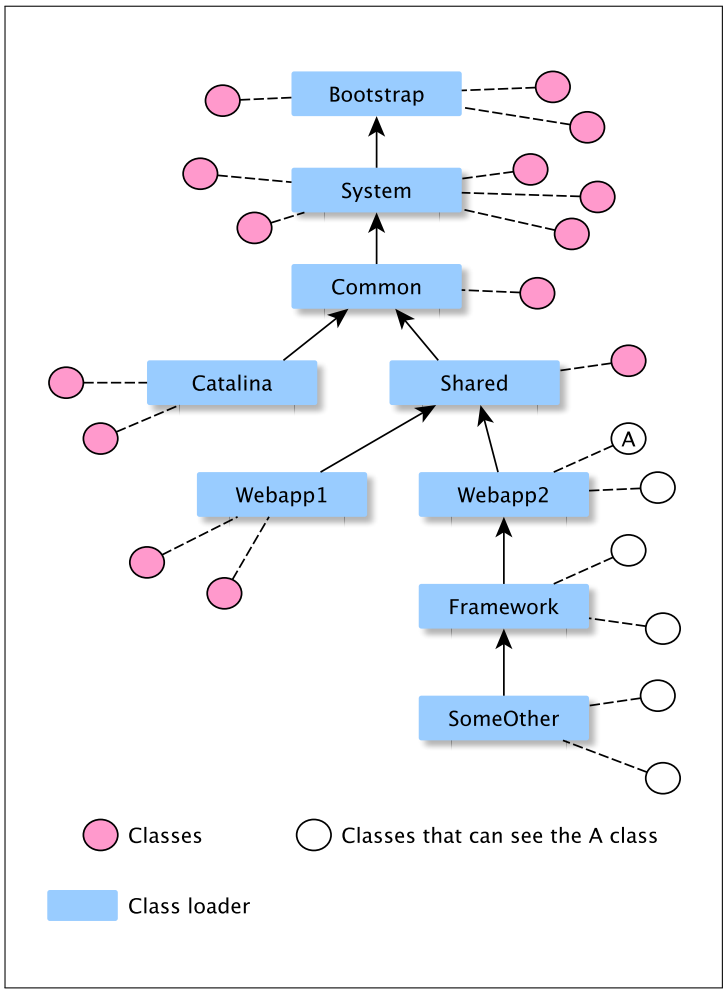

Figure 2.5: Example of access to classes on a tree of class loaders.

one that loaded the framework). We solved this problem using an specific class loader to load classes and through dependency injection techniques (Fowler, 2004) to ensure that classes will not be loaded by some other default class loader. We assume that a framework can do this, becouse we believe that dependency injection should be supported in any modern framework.

With dependency injection we can make the framework responsible of instantiating the application objects. So we do not deliver exactly what is requested but something slightly different: objects of the classes modified to satisfy the implicit interface and another requirements the framework knows.

The use of class loaders with dependency injection works, but modified classes are hard to use as is. But using an explicit hidden interface as a back-end can simplify the framework development. This is because although by the use of reflexion it is possible to find references to method and properties in modified classes that match those of the implicit interface it could be quite hard to do it this way. To understand this consider the page implicit interface and before render method.

Figure 2.6 shows what is needed to execute a beforeRender method just using reflection. Figure 2.7 shows that a Page internal explicit simplifies the same operation.

```
private Method beforeRenderMethod;
// ...

private PrintWriter outputPrintWriter;

// ...

public void executePage(Object page) {
    try {
        // ...
        Class<?>[] expectedArguments = beforeRenderMethod
                .getParameterTypes();
        Object[] arguments = null;

        if (expectedArguments.length == 0) {
            // no parameters
            arguments = new Object[0];
        } else if (expectedArguments.length == 1
                && expectedArguments[0]
                        .isAssignableFrom(PrintWriter.class)) {
            // just the printwriter
            arguments = new Object[] { outputPrintWriter };
        }
        beforeRenderMethod.invoke(page, arguments);
        // ...
    } catch (IllegalArgumentException e) {
        // ...
    } catch (IllegalAccessException e) {
        // ...
    } catch (InvocationTargetException e) {
        // ...
    }
}
```

Figure 2.6: Runtime execution of beforeRender method using reflection.

```
private PrintWriter outputPrintWriter;

// ...

public void executePage(cl.john.tea.internal.Page page) {
    //...
    page.beforeRender(outputPrintWriter);
    //...
}
```

Figure 2.7: Runtime execution of beforeRender method using a hidden explicit interface.

So a little trick here is to define the explicit interface that the framework expects but as a hidden internal interface. In runtime, when the framework analyzes the class to implement the required implicit interface, the class is modified to implement the hidden internal interface. In fact we can dinamically build a new class that works as an adapter that maps the internal explicit interface methods to the ones provided by the application. Figure 2.8 shows how this is achieve for beforeRender method of a page, using Javassist to make that class implements internal explicit interface Page.

One last issue to consider is efficiency. The discovery of implicit interfaces that must be done in runtime may represent a significant additional time, as well as use of reflection in runtime. This can be remedied by using application initialization for discovering and class transformation so performance is not affected during execution: in runtime there are just regular classes calling methods.

2.9. Conclusions and Future work

Web application frameworks are here to stay. To make them more flexible and easier to use by the developers is not a new idea. Chiba (2005) proposed the intensive use of annotations precisely to get this goal.

We examined many of the new ideas that are being implemented in the present frameworks to make them more attractive to programmers and we synthesized all in a concept

```
//... processing pageClass to implement Page interface
try {
    CtMethod[] classMethods = pageClass.getDeclaredMethods();
    // search beforeRender(PrintWriter)
    for (int i = 0; i < classMethods.length; i++) {
        if ("beforeRender".equals(classMethods[i].getName())
                && classMethods[i].getParameterTypes().length == 1
                && classMethods[i].getParameterTypes()[0].equals(printWriterClass)) {
            //already have method. Done.
            return;
        }
    }
    // search beforeRender()
    for (int i = 0; i < classMethods.length; i++) {
        if ("beforeRender".equals(classMethods[i].getName())
                && classMethods[i].getParameterTypes().length == 0) {
            //create method and call this one
            createBeforeRenderMethodThatCallThisOne(pageClass, classMethods[i]);
            return;
        }
    }
    //... search also with annotations
    //... if there is no such method, create empty one.

} catch (NotFoundException e) {
    //...
}
```

Figure 2.8: Part of processing a page class to implement Page interface.

we call implicit interface in contrast with the usual explicit interfaces that are used to communicate the framework with the application. The use of implicit interfaces to this purpose improves the communication with the framework making it flexible and easy to use.

We showed that implicit interfaces are not just a nice theoretical idea by building Tea, a micro framework that implemented these ideas. Using this framework we were able to give the application all the freedom we wanted.

Finally, we highlighted some important issues that should be considered by framework designers that want to implement implicit interfaces: taking care of the different class loaders, using dependency injection to modify classes in runtime before instantiation and using explicit hidden interfaces to end with a final advice on how not to affect the perceived execution time by moving all the additional processing to application initialization time.

We plan to extend our research in two directions: automation and application of these ideas in contexts different to web applications and frameworks.

At present, processing the implicit interface specification is an ad-hoc process. Discovering the classes for each method an property, transformations and all the needed code is all individually tailored for each case. Moreover, any future change in the implicit interface has no effect on the framework. We would like to automate things so the implicit interfaces be connected with the way the framework manages them. We think this could be done by properly annotating methods and properties of the interface instead of just using javadoc.

Finally, we believe that implicit interfaces could find interesting uses beyond the communication between frameworks and web applications. In fact it can be applied to any layered architecture in which the top layer depends on the bottom layer but the bottom layer is completely independent of the top. Because of this, the lower layer can accept different upper layers without any change but the opposite is not true. As a consequence, the lower layer can be built and packaged to be used as a black box by an independent team of programmers. Any communication interface between layers is provides and belongs to the lower layer. It is easy to see that the case of a web application framework is just a special case of this layered architecture with the framework on the bottom and the web application on top.

Chapter 3. CONCLUSION AND FUTURE RESEARCH

3.1. Conclusions

The use of frameworks for web application development is more popular every day. This is because, as we have explained before, there are huge benefits in terms of development effort and even more important yet, in how easy will be to extend and change the application in the future.

There is however a cost. The developer who will use the framework needs to invest a significant amount of time in learning the framework intricacies so he can master the framework before he starts coding the application. Moreover, if the framework imposes strict guidelines, the developer will need to adapt itself to the framework making the less productive phase even longer.

For quite a long time developers have known that some frameworks seem to be better than others in terms of learnability and easy of use. For example it seems that more recent frameworks do better than first generation ones. One of the important aspects that influence how fast a programmer will be effective using a given framework has to do with the way the framework communicate with the web application code.

Previous research had found that the use of something called implicit interfaces produced frameworks that were much friendly to the developer but the idea was still not completely conceptualized and there was no clear and well understood strategy to implement implicit interfaces.

We studied implicit interfaces to deeply understand the important issues and available strategies that could be used to materialize the idea. We proposed a specific strategy and we implemented a micro framework to see if it worked in practice. The result was a framework that exposed its functionalities in a way that is closer to the application programmer who has more freedom to use his own techniques and conventions. The framework gets the ability to help the developer requesting from him only what is strictly necessary.

The integral generated documentation not only unifies the documentation of the interface with the meta information but it is also presented in a clear and familiar way for the developer, as a javadoc. The same approach could be used with any other modern programming language instead of Java that has automatic documentation facilities.

Our strategy for implicit interfaces allows the rise of better frameworks without compromises in performance because the modified Java classes during loading execute as regular classes after they have been loaded and there is no need for complex sophisticated reflexion techniques in runtime.

The final result is a situation where there are only gains. Framework developers not only have been validated and conceptualized ideas that had been introduced in the last years, but they also have now a standard way to incorporate these ideas in the practice. Software developers will have at their disposal better frameworks that are easy to use and less time to learn. Project leaders can complete their projects in less time and the resulting product can be adapted if necessary.

3.2. Future Research Topics

Further research could be focused in two main lines: automating the processing of implicit interfaces and extending these ideas beyond the web and framework-application communication.

At present, the analysis and processing of the implicit interfaces including all the needed class alteration is made as a case by case depending on context and the complexity of the dynamic interface that we are trying to implement. Furthermore, the implicit interface, the component that interprets it and the application implementation do not have any connection at the code level so errors cannot be detected at compile time and they can even pass the class modification phase to appear only at application runtime.

Possible improvements of the present situation could involve connecting the implicit interface with the interpreter trough Java annotations in the implicit interface definition (instead of using it just for documentation). This might alow automated processing of

the classes that implement the interface. Another desirable connection is between the interpreter and the implementation trough an early analysis of it to check for possible bugs.

About exploring the application of these ideas to context different that the one that inspired it, we believe that there is a great potential for any context where the software is architected as two layers that need to communicate via interfaces. It will work always that there is a unidirectional dependency relationship between the layers similar to the one we find between web framework and web application.

Another area that needs to be explored is that related to necessary tools associated to dynamic class alteration. Instrumentation and profiling tools that include techniques for class intervention represent only the surface. We need to perform deeper studies about dynamic class modification because any library that operates with the classes in a direct way either to modify them or by the use of the reflexion API could have potential conflicts with other components trying to do the same.

References

Alur, D., Crupi, J., & Malks, D. (2001). *Core J2EE patterns: Best practices and design strategies* (1st ed.). Pearson Education.

Bächle, M., & Kirchberg, P. (2007). Ruby on rails. *Software, IEEE*, *24*(6), 105–108. Available from http://dx.doi.org/10.1109/MS.2007.176

Chiba, S. (2005). Generative programming from a post object-oriented programming viewpoint. In *Unconventional programming paradigms* (Vol. 3566/2005, pp. 355–366). Springer Berlin / Heidelberg. Available from http://www.springerlink.com/content/3xuyqxlmnvlv8qu8/

Chiba, S., & Nishizawa, M. (2003). An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the 2nd international conference on generative programming and component engineering* (pp. 364–376). Erfurt, Germany: Springer-Verlag New York, Inc. Available from http://portal.acm.org/citation.cfm?id=954186.954208

Dahm, M. (2001, 4). *Byte code engineering with the BCEL API* (Tech. Rep.). Freie Universität Berlin, Institut für Informatik. Available from http://bcel.sourceforge.net/downloads/report.pdf

Fowler, M. (2004, January). *Inversion of control containers and the dependency injection pattern.* Available from http://martinfowler.com/articles/injection.html

Javadoc tool home page. (2004). Sun microsystems. Available from http://java.sun.com/j2se/javadoc/

Li, J., Ma, G., Feng, G., & Ma, Y. (2006). Research on web application of struts framework based on MVC pattern. In *Advanced web and network technologies, and applications* (Vol. 3842/2006, pp. 1029–1032). Springer Berlin / Heidelberg. Available from http://www.springerlink.com/content/f623n5g542x05717/

Murugesan, S., Deshpande, Y., Hansen, S., & Ginige, A. (2001). Web engineering: a new discipline for development of Web-Based systems. In *Web engineering* (Vol. 2016/2001, pp. 3–13). Springer Berlin / Heidelberg. Available from http://www.springerlink.com/content/8ry1c125q4ujt3b9/

O'Reilly, T. (2005, September). *What is web 2.0 - O'Reilly media.* Available from http://oreilly.com/web2/archive/what-is-web-20.html

Roberts, D., & Johnson, R. (1996). Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Proceedings of the third conference on pattern languages and programming* (Vol. 3). Available from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.8767

Shan, T. C., & Hua, W. W. (2006, October). Taxonomy of java web application frameworks. In *e-Business engineering, 2006. ICEBE '06. IEEE international conference on* (pp. 378–385). IEEE Computer Society. Available from http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=4031677

Ship, H. L. (2009). *Apache tapestry.* Available from http://tapestry.apache.org/

Sosnoski, D. (2003, April). *Java programming dynamics, part 1: Java classes and class loading* (CT316). IBM. Available from http://www.ibm.com/developerworks/java/library/j-dyn0429/

Sosnoski, D. (2004, February). *Java programming dynamics, part 5: Transforming classes on-the-fly* (CT316). IBM. Available from http://www.ibm.com/developerworks/java/library/j-dyn0203.html

Struts Development Team. (n.d.). *Apache struts 2.* Available from http://struts.apache.org/2.x/

Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., et al. (2005). Information hiding interfaces for aspect-oriented design. In *Proceedings of the 10th european software engineering conference held jointly with 13th ACM SIGSOFT international symposium on foundations of software engineering*

(pp. 166–175). Lisbon, Portugal: ACM. Available from http://portal.acm.org/citation.cfm?doid=1081706.1081734