Modularization of the Sisyphus Proteomics Tool

Arthur van Dorp

Master’s Thesis Project Report

Information and Communication Systems Group
Department of Computer Science
ETH Zurich

http://www.iks.ethz.ch/

November 2008 – April 2009

Supervised by:
Dipl. Ing. Inf. ETH Thomas Heinis
Prof. Dr. Gustavo Alonso

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Abstract

Proteomics is the analysis of large quantities of proteins. The Sisyphus Proteomics Tool is a web application helping scientists to interpret the results of tandem mass spectrometry experiments.

This thesis presents the modularization of the Sisyphus application in the context of the OSGi dynamic component system for Java. The already “horizontally” layered architecture of the Sisyphus system has been modularized in the “vertical” direction. The extension/extension-point-like architecture envisioned at first has been adapted to accommodate for the use of Google Web Toolkit on the client side.
Contents

1 Introduction 5
   1.1 The Sisyphus Proteomics Tool 5
   1.2 Call for Modularization 6
   1.3 OSGi 7
   1.4 Google Web Toolkit 11

2 Architecture 12
   2.1 The Unmodularized Sisyphus System 12
   2.2 Vertical Modularization 13
   2.3 Modules 14
   2.4 Modularizing GWT 15
      2.4.1 Discarded Solutions 15
      2.4.2 A Semi-Dynamic Solution 15
   2.5 Implementing Modules 16
   2.6 The Module Contributions 20
      2.6.1 Protein and Peptide List Columns 20
      2.6.2 Experiment Panel or Experiment Statistics Panel 20
      2.6.3 Protein Details 20
      2.6.4 SQL Queries 21
      2.6.5 Database Schema Enhancement 21
      2.6.6 Http Servlet 22

3 Implementation 24
   3.1 OSGi 24
      3.1.1 Declarative Services 24
      3.1.2 OSGi, Third Party Libraries and Classloading 25
   3.2 Module and SisyphusCore Development 27
      3.2.1 Sisyphus Module Project Wizard 27
      3.2.2 CodeGenerator and Database Access Layer 28
   3.3 Logging 31
   3.4 Configuration 32
   3.5 GWT and OSGi 32
   3.6 Extraction, Modification and Compilation of the Client Side 33
   3.7 Module Contributions 35
      3.7.1 Usability Considerations 36
4 Evaluation 37
  4.1 Performance Impact of Modularization 37
    4.1.1 Performance Deliberations 37
    4.1.2 Performance Measurements 38
  4.2 Evaluation of the Approach Taken 40

5 Conclusion 42
  5.1 Future Work 44

List of Figures and Code Listings 46

Bibliography 49
Chapter 1

Introduction

In a cell, the DNA in the nucleus is transcribed to messenger RNA and in turn the mRNA is translated to an amino acid sequence. System biology followed the same steps and went from analyzing DNA sequences to mRNA sequences, and from there to looking directly at the sequences of proteins. The field of proteomics [Pandey and Mann, 2000] tries to sequence all proteins in a biological system (e.g. a cell, an organelle, a body fluid, all under various conditions) and to identify their specific roles and interactions. Proteomics is rapidly pushing towards higher protein sensitivity and higher spatial and temporal resolution.

It is hoped that the results of the research of proteomics will find their way into clinical applications. The main focus is on the area of biomarker research, meaning the identification of disease specific proteins from a patient sample, for instance a blood or tissue sample. E.g. a triplet of blood serum biomarkers have been found which can predict ovarian cancer when measured together [Colantonio and Chan, 2005].

Mass spectrometry, especially the so called tandem mass spectrometry (called like this because it involves more than one spectroscopy step separated in time or space) [Beavis et al., 1983], is one of the tools used in proteomics. To process the data from the spectroscopy experiments, the Trans Proteomic Pipeline (TPP, [Nesvizhskii et al., 2003]), a set of software tools helping the scientists analyzing and understanding the results, has been developed. It specifically identifies and quantifies the proteins found in the experiments.

The resulting data from the TPP consists mainly of a list of found peptides\textsuperscript{1} and proteins, a score denoting the probability of correct protein identification and the quantity of the identified proteins in the sample (optionally, depending on the type of experiment). To be useful to the biologists, this data has to be supplemented with information from various databases maintained by different proteomics organizations and university institutes.

1.1 The Sisyphus Proteomics Tool

The Sisyphus Proteomics Tool, which is being developed at the Information and Communication Systems Group\textsuperscript{2} of the ETH Zurich, is meant to be used to manage and present these results and the supplemental data from external sources. It is

\textsuperscript{1}A peptide is a short sequence of amino acids.

geared to replace a pre-existing application also called Sisyphus. The old system used so far is a Filemaker application which has some limitations regarding its extensibility and its scalability with respect to the number of users and experiments. The new Sisyphus system is a web application specifically built with overcoming the shortcomings of the old system in mind. An overview over its architecture can be found in section 2.1.

For each experiment Sisyphus imports the protXML\(^3\) file. protXML is the file format returned by the TPP tool chain after analyzing the raw data from the experiment. After parsing the experiment data, Sisyphus associates each identified protein with data from the UniProt\(^4\) resource, a central protein data repository. In a third step, the identified proteins are annotated with additional information either from external sources or from calculating the annotation data within Sisyphus. The additional data can e.g. be solubility predictions or a categorization of the proteins into protein families (denoting a group of evolutionarily related proteins).

For a more in depth overview of the data sources used in the Sisyphus tool see the master’s thesis “Data Integration in the Sisyphus Proteomics Tool” [Remund, 2008]. For other proteomics databases and software tools consult the website\(^5\) of the ExPASy project [Gasteiger et al., 2003].

The researcher using Sisyphus has to be able to interpret the presented data. Depending on the current needs of the user different views on the data have to be presented. If necessary these views have to be user-adaptable on the fly, e.g. by offering filtering rules to be applied.

The Sisyphus application has to be able to manage a large amount of data. Experiments are run frequently and each can return many hundred identified proteins. This means that the architecture of Sisyphus has to be fast and scalable.

### 1.2 Call for Modularization

**Challenge** On one hand, the Sisyphus system is already modular. It expressly separates the code, the database schema and the database queries from each other (see figure 2.2 for a schematic view of the layering). On the other hand, adding new functionality, e.g. a new GUI component or a new data source, often affects several of those layers. This can lead to tighter coupling within those layers than wished for, impeding future code modification and code maintenance.

**Goal** The main goal of this thesis is the modularization of the ‘vertical’ concerns of the application (as opposed to the current ‘horizontal’ layering, which should be conserved). This means identifying what parts can be decoupled and how their interfaces should be designed.

There are different stakeholders to keep in mind when devising architectural changes to Sisyphus: The end-users, usually biologists trying to use Sisyphus to better understand the results of their experiments; the future developers and maintainers of the Sisyphus core system, most likely computer scientists or trained programmers and the developers of new modules. Those could be programmers or biologists with an interest in programming. So as auxiliary goals we could say that ease of development

---


4See [http://www.ebi.ac.uk/uniprot/](http://www.ebi.ac.uk/uniprot/) (Retrieved on 2009-04-30).

and enhancement should be at least maintained, or better, improved, and the end-user experience should get better (thanks to more analytical tools available due to easier development) and certainly not worse (e.g. through performance losses).

1.3 OSGi

To make the kind of aforementioned modularization possible, OSGi, a “dynamic component system for Java”\(^6\), was predetermined as the new runtime environment for Sisyphus.

The OSGi Alliance \([\text{OSG}, 2009]\) publishes a set of specifications \([\text{OSG}, 2007a, \text{OSG}, 2007b]\) describing a framework based on the Java Virtual Machine (JVM, \([\text{Lindholm and Yelling, 1999}]\)) and allowing the construction of dynamic modularized Java programs, closing a gap not covered by Java itself. The systems resulting from those specifications and the specifications themselves are furthermore simply called OSGi.

OSGi describes how independent parts of the system (called bundles) interact with each other (via services and a service registry). It defines the life-cycle of those bundles and how they can be added, removed or updated at runtime. It specifies code signing and permission techniques to ensure the security of the system. All these parts are described in the so called Core Specification \([\text{OSG}, 2007a]\).

A second document called the Service Compendium \([\text{OSG}, 2007b]\) defines various services which use the framework from the Core Specification as foundation. Among them are logging and administration facilities as well as device access and configuration resources.

OSGi promises\(^7\) reduced complexity because the modules (bundles) in OSGi hide their internals from each other, decoupling the components of the resulting system from each other, which in turn also should enable more reuse. OSGi furthermore strives to make deployment of bundles easy and meanwhile enabling dynamic updates and versioning of bundles. Besides all this it also aims to be small (some implementations are a few hundred kilobytes) and fast\(^8\).

OSGi is used in a wide range of real world applications. Well known projects like Eclipse\(^9\) or the Spring framework\(^10\) use it, but also more hidden applications like Siemens’ TLA\(^11\), a software platform for a car infotainment system, or the hardware platform OMBC\(^12\) offered by the Fraunhofer Institute for Integrated Circuits employ OSGi.

In the following paragraphs a few OSGi specific terms are explained in more detail.

\(^7\)See [http://www.osgi.org/About/WhyOSGi](http://www.osgi.org/About/WhyOSGi) (Retrieved on 2009-04-30).
\(^8\)E.g. classloading should be faster than in a normal Java environment where every JAR file and directory on the classpath has to be searched whereas OSGi can leverage its knowledge of each bundle’s imports and exports and confine the search to the classpath of the specific bundle.
Bundles  A bundle as described in the Core Specification [OSG, 2007a, p. 25 ff.] is the main building block of OSGi. A bundle consists of a description called “manifest”, Java classes and optionally other resources like JAR-files, images or anything that can be stored in a file. Usually everything is stored within a JAR file. Some implementations, e.g. Equinox, allow also unpacked or so called exploded bundles which simply live as a collection of files and directories inside a file system directory.

The manifest describes the bundle. It specifies among other things a name and a version number, delivers information about dependencies on other bundles (either via Java package imports or by demanding specific bundles, both optionally specific to a given version range), describes which Java packages it exports for the benefit of other bundles and indicates, if necessary, a class implementing the Activator interface called by OSGi upon bundle start and stop.

A bundle in this thesis is depicted with a rectangle, its name and a stylized OSGi logo in the upper right corner as shown on the right hand side.
**Services** Bundles not only can import packages from other bundles, they can also subscribe to so called services which are offered from other bundles in the system. Each bundle can offer any number of services and also use any number of services. A service is in essence a plain old Java object (POJO) which can be published, looked up in a registry and used by other bundles. Bundles can also subscribe to registration and deregistration events making the system highly dynamic.

However, being dynamic also comes with a cost. For the developer, the highest cost is the increased complexity as one has to cope with services suddenly becoming unavailable or getting published. There are several approaches to alleviate that problem. Spring\(^a\), iPOJO\(^b\), Eclipse’s extension points and the Declarative Services of the OSGi platform itself are among them.

The diagram on the right shows Bundle A offering a service and Bundle B using it. Additionally Bundle B imports Java packages from Bundle A.


Declarative Services and Components
Since release 4, the OSGi framework offers the possibility to use and offer services declaratively [Wütherich et al., 2008, p. 201 ff.]. Instead of actively registering its services with the service registry and looking up other services or tracking them with a ServiceTracker, the bundle declares which services it offers, which services it needs and which services it can use optionally. Declarative Services are specified and described in the chapter Declarative Services Specification of the Service Compendium [OSG, 2007b, p. 281 ff.].
The Declarative Services specification introduces so called components. A bundle can have any number of components. Each component can offer at most one service and use any number of services. These dependencies are described in the so called component description which is an XML file located inside the /OSGI-INF directory of the bundle. It also describes which class implements the component and of what type it is.
There are three different types of components: Immediate, delayed and factory components. They differ in the way how and when they are created. For the purposes of this thesis only immediate and delayed components are of interest.
Components have their own life-cycle inside their host bundle. Immediate components on one hand are instantiated (and therefore started) as soon as all their dependencies are resolved. A delayed component on the other hand is only instantiated at the time its service is requested (which means that a delayed component without a declared service does not make sense). The life cycles of all declared services are controlled by the Service Component Runtime, SCR for short.
The picture on the right shows two declarative services components of an OSGi bundle, one using three services and offering one itself and the other using two services and offering none itself. Because so far all bundles used within Sisyphus compromise only one component, components are not depicted separately in any of the illustrations later in the text.

Using declarative services promises less \textit{startup time} and \textit{memory consumption} because only bundles which are needed are activated, and reduced \textit{complexity} because bundles do not have to actively manage their mutual dependencies, which can be tedious and error prone.
1.4 Google Web Toolkit

Sisyphus uses the open source Google Web Toolkit (GWT)\(^\text{13}\) for its presentation layer, for the application layer on the client side (see figure 2.2) and for the communication between the server and the client.

GWT compiles Java code to JavaScript and some HTML and CSS code. There are some limitations regarding what classes from the Java Standard Edition can be used\(^\text{14}\). Especially everything related to concurrency is not supported\(^\text{15}\).

GWT promises the easy development of well performing AJAX applications which work among a range of supported browsers. To make the created JavaScript perform well, Google Web Toolkit compiles a separate application for each targeted browser platform and for each user interface language. A user therefore does not have to download any code not relevant to her. On the other hand this increases compile time.

Additionally, GWT offers several ways to communicate between the server and the client. Sisyphus uses GWT’s RPC mechanism to make requests from the client on the server. The requests are answered asynchronously using the XMLHttpRequest interface of the DOM API of the browser.

Unfortunately GWT does not yet support the linking of independently compiled parts to a whole application\(^\text{16}\). This makes modularizing the client code significantly more difficult.

\(^{13}\)See http://code.google.com/webtoolkit/ (Retrieved on 2009-04-29).

\(^{14}\)All the limitations can be found at http://code.google.com/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&DevGuideJavaCompatibility (Retrieved on 2009-04-02).

\(^{15}\)Browser-side JavaScript is still single-threaded, this is due to change with HTML 5, where some high-level concurrency tools are planned. See http://www.w3.org/TR/html5/ (Retrieved on 2009-04-29).

\(^{16}\)This should change with a future release of the Google Web Toolkit. The planned architecture is explained under http://code.google.com/p/google-web-toolkit/wiki/CodeSplitting (Retrieved on 2009-04-27).
Chapter 2

Architecture

2.1 The Unmodularized Sisyphus System

The design of the unmodularized (or more correctly less-modularized) Sisyphus system is sketched in figure 2.1 which details the building blocks used. Figure 2.2 shows the architecture in a much more abstracted way reducing its complexity to a few layers and making it easier to discuss.

![Diagram of Sisyphus system architecture](image)

Figure 2.1: The architecture of the Sisyphus system before modularization showing the most important libraries and programs used.
A distinctive feature of the Sisyphus application is the three-part persistence layer: It consists of code to access the database, a database schema definition and SQL queries. At development time an ANT\(^1\)-script populates the database with the tables defined in the schema definition file and generates DynaBeans\(^2\) and database-querying code from the SQL queries.

The application logic layer on the client (browser) side is more important than in more traditional web application architectures where the browser side often only has the role of the presentation layer. With Sisyphus, the client side’s application layer is as important as the one on the server side. Additionally, when looking at the top of figure 2.1, one does not only see more application logic, but also a whole database, caching data received from the server, and a logging framework reporting logging information back to the server side. This is a consequence of the way the Google Web Toolkit allows for easy development of browser side code in Java which is compiled into JavaScript\(^3\) with some HTML and CSS added in.

![Diagram of Sisyphus system's architecture before modularization](image)

**Figure 2.2:** An abstracted view of the Sisyphus system’s architecture before modularization. Compared to more customary web application architectures more of the application logic resides on the client side.

### 2.2 Vertical Modularization

In order to reach the goals stated in section 1.2, i.e. the “vertical” modularization of the Sisyphus system, the “horizontal” modularization has to be complemented by a modularization in the “vertical” direction as illustrated in figure 2.3. A core with the same horizontal layers as the preexisting Sisyphus system is accompanied by modules which are also layered. While the schematic diagram implies that every of the preexisting layers exists for each module, this is not a prerequisite. A module can make contributions only to the GUI layer or it can live solely on the server side without being viewable to the end-user at all. Any other conceivable combination is possible. I.e. a module usually will not contribute its own database, but it could.

The goal is to add a “vertical” modularization without compromising the “horizontal” layering.

### 2.3 Modules

Each module is designed as an independently existing OSGi bundle (see section 1.3) and can make predefined contributions to the core system which is itself an OSGi bundle. (The core system is called SisyphusCore further on in this thesis. It is called like this in the source code as well.) The SisyphusCore has an immediate declarative service component while each module has (at least) one delayed declarative service component (see section 1.3), meaning that the core is started by the Service Component Runtime as soon as its dependencies are met, while the modules only are started when they are requested.

The contributions a module can make to the Sisyphus system as a whole can be of different types. It might for example calculate new data based on the results of an experiment (e.g. predict the water solubility of a newly identified protein based on the hydrophilicity of its constituting amino acids), store the results of this computation in the database and show the result as part of a protein list shown to the user when looking at the data of an experiment. Such a module would have to offer to the SisyphusCore a database schema enhancing contribution, an interface for the client side to retrieve that data from the server side and an extra column for the table detailing the proteins found in an experiment. The module contribution types devised during this thesis project are detailed in section 2.6.

The SisyphusCore has been designed under the assumption that modules are independent from each other. There are e.g. no provisions in the SisyphusCore which would allow two modules to communicate with each other via the core. If module developers wish to make modules with dependencies on each other, they have to devise this without the SisyphusCore.

Initially it was planned that a module offers one OSGi service per contribution type it supplies. This would be almost equivalent to the way the Eclipse development environment uses plug-ins, extension points and extensions [Gamma and Beck, 2003]. For a short comparison between Eclipse’s extensions and OSGi’s declarative services

---

*Figure 2.3: The conceptual architecture of the Sisyphus system after modularization. The goal is to add a “vertical” modularization without compromising the “horizontal” layering.*

---

(and also OSGi services and Spring-OSGi) see [Bartlett, 2007]. The two main reasons for going with OSGi’s declarative services instead of Eclipse’s plug-ins are that plug-ins would need an additional bundle not specified in the OSGi specification and that dynamic plug-ins must be tracked with ExtensionTrackers instead of the declarative and dynamic way which declarative services offer.

2.4 Modularizing GWT

During the course of this thesis it became apparent that the client side, which uses Google Web Toolkit, can not be easily modularized in a way which would allow the use of one declarative service (or plug-in) per contribution and meanwhile preserving the simplicity of the declarative services framework. This is due to the fact that GWT applications (as of now) can not be precompiled in parts and be linked together later (see section 1.4). They always have to be compiled to one monolithic JavaScript application.

2.4.1 Discarded Solutions

The most straightforward solution would be to implement linking of precompiled GWT components. This was looked at and deemed to be infeasible in the timeframe of this thesis project.

The problem could be circumvented by independently compiling the GUI contributions of the SisyphusCore and of each of the modules and then combining them outside of GWT, e.g. by embedding the GWT applications inside an HTML div or iframe, or by any other means. Such an approach would have several drawbacks:

A single GWT application already taxes the computer of the client side heavily. Now each module would have to load a complete GWT application, duplicating parts of the JavaScript code, each adding to memory consumption and each putting more pressure on the CPU.

The different GUI contributions must be able to communicate with the core’s GUI (e.g. to obtain experiment data). This would require creating public JavaScript APIs to enable GWT application interaction. This might be feasible although there are many unknowns, e.g. whether there would be concurrency problems in the single threaded browser environment.

Using an HTML div to host each GWT application would allow access to the web page’s Document Object Model (DOM) from every GWT application, but it would almost certainly also lead to problems with element ID clashes (instances of the same GWT classes from different applications might assign the same IDs to different DOM elements). An HTML iframe would isolate the DOMs, avoiding ID clashes, but also prevent direct access from one application to the elements of another application resulting in the need for bigger JavaScript APIs and negatively impacting performance.

Because of those anticipated problems, this type of solution was discarded.

2.4.2 A Semi-Dynamic Solution

If multiple GWT applications (for one client) are not feasible (for now), it stands to be reasoned that using one monolithic GWT application would be inevitable if one
does not want to discard GWT completely. The goal of modularizing the Sisyphus system, including its GUI parts, is not questioned. The proposed solution is to let the modules provide their GUI contributions as Java source code. A subsystem of the SisyphusCore then extracts that source code from the modules and uses the GWT compiler to build a monolithic GWT application. The newly compiled application, including http servlets provided by modules, is served through an http service installed into the OSGi platform. How the extraction of the source code and its compilation is handled is described in section 3.5.

To accommodate to the needs of this “semi-dynamic” solution, only one service is offered per module. The interface which the modules implement is called c.e.s.osgi.SisyphusModuleService\(^5\). The declarative services Service Component Runtime automatically binds the module to the SisyphusCore as soon as they are resolved (see figure 2.5).

The reason for using only one service instead of one for each offered contribution is mainly simplicity. The SisyphusCore on the client side should know about the GUI contributions the modules make in order to integrate them into its own GUI. But it has no direct access to them because the Service Component Runtime only informs the SisyphusCore on the server side. One could devise a protocol through which the server side informs the client side about the offered contributions. But because the server side anyway has to compile the client side at a point in time when it already knows everything about the modules which are currently active, it can simply compile this information into the monolithic client code.

The interface c.e.s.osgi.SisyphusModuleService implemented by each module gives access to the contributions of the respective module. In an intermediary step after the extraction of the client side source code, one file is modified and one file is created. The first file contains a singleton called ProxyImpl with a static block initializing a list of all the ModuleGuis implemented by the modules. The static block is run on instantiation of the singleton. Listing 2.4 shows the class ProxyImpl after three modules have been activated.

The second file is a GWT module description\(^6\) (a GWT module as such has nothing to do with a Sisyphus module) which links the GWT code of the Sisyphus modules into the core GWT source. How this works in detail is explained in section 2.5.

Using only one service is not a drawback for the SisyphusCore. It has to react exactly once on the presence of a new module and can therefore simplify its “reacting on the presence of contributions” logic. The core can work with a single “door-opener” API as long as it gives access to all the contributions it is interested in.

### 2.5 Implementing Modules

A module implements at least the Module abstract class and the interface c.e.s.osgi.SisyphusModuleService. If it offers any client side contributions it also implements the ModuleGui interface. See figure 2.6 for a graphical depiction of the three entities. This might seem at first sight overly complex, but on closer inspection it separates concerns and does not make implementing a module more difficult.

\(^5\)c.e.s stands for ch.ethz.sisyphus and is used in this text to shorten package names.

// Acts as proxy between the SisyphusCore GUI and the GUI parts provided by the modules.

// Implemented as an enum to provide singleton properties.

// This class might be enhanced at OSGi runtime with static{} blocks for populating the various Collections with content used at GWT compile time.

public enum ProxyImpl implements Proxy {

    INSTANCE;

    private static ArrayList<ModuleGui> moduleGuis = new ArrayList<ModuleGui>();
    // Do not remove the following identifier. It is used for pre-compilation modification.
    static {
        ProxyImpl.INSTANCE
            .putModuleGui(ch.ethz.sisyphus.modules.crossExperiment.client.CrossExperimentGui.INSTANCE);
        ProxyImpl.INSTANCE
            .putModuleGui(ch.ethz.sisyphus.modules.sequence.client.SequenceGui.INSTANCE);
        ProxyImpl.INSTANCE
            .putModuleGui(ch.ethz.sisyphus.modules.iTRAQ.client.ITRAQGui.INSTANCE);
    }

    public Collection<ModuleGui> getModuleGuis() {
        // Return a shallow copy.
        return new ArrayList<ModuleGui>(moduleGuis);
    }

    void putModuleGui(ModuleGui moduleGui) {
        moduleGuis.add(moduleGui);
    }
}

Listing 2.4: The class ProxyImpl after the activation of three modules. This class is modified by the SisyphusCore at server side runtime before client side compilation. Like this the client side has access to the implementations of the interface ModuleGui and via them also to the implementations of the abstract class Module.

The abstract class Module is confined to those properties of a module which are shared among client and server side. It offers a unique identifier for the module as well as a human readable name and a human readable short description which is also suitable for being displayed in the GUI. It also offers a set of enum elements which enumerate the types of contributions the module makes. See listing 2.7 for the
Figure 2.5: The core of the Sisyphus application declares itself as a consumer of services which implement the interface `c.e.s.osgi.SisyphusModuleService`. A Sisyphus module is by definition a bundle which offers a service implementing that interface.

The interface `SisyphusModuleService` is, as explained before, the one offered by modules as a service and the one the SisyphusCore expects. Apart from that this interface is extremely small. The `getBundle()` method returns the bundle instance of the module which is used to get to its classloader. The classloader is needed to access various non-OSGi libraries as described in section 3.5. The other method is `getModule()` which returns the `Module` instance whose information is shared between the client and the server side (the instance itself is not shared, each side has an identical instance of the class).

The interface `ModuleGui` is a client side entity which allows the client side to access the `Module` instance via the `getModule()` method the same way as with the `SisyphusModuleService` interface on the server side (see figure 2.6). Additionally it offers methods to retrieve the client side contribution types which is shown in figure 2.8.

Figure 2.6: A module which makes client and server side contributions is defined by the implementations of the two interfaces `SisyphusModuleService` and `ModuleGui` and of the abstract class `Module`. A `SisyphusModuleService` instance can not exist on the client side because it has to maintain a reference to the bundle classloader. A `ModuleGui` instance only exists on the client side and manages all the contributions of the module to the GUI. The `Module` instance offers all the properties of a module which are shared among server and client side.
The different types of contributions a module can make to SisyphusCore.

```java
public enum ModuleContributions {
    /** One or more columns for the proteins list. */
    PROTEIN_COLUMNS,
    /** Information for the protein details view. */
    PROTEIN_DETAIL,
    /** One or more columns for the peptide list. */
    PEPTIDE_COLUMNS,
    /** Panel in the experiment main tree. */
    EXPERIMENT_PANEL,
    /** Panel for the experiment statistics subtree. */
    EXPERIMENT_STATISTICS_PANEL,
    /** SQL-Query/Queries to be performed on DB. This also entails that the
     * CodeGenerator creates a GWT-'moduleId'Service for direct access to the
     * query results. Register the service by implementing
     * c.e.s.osgi.HttpServletProvider or register it yourself. */
    SQL_QUERIES,
    /** DB schema contributions in the form of DDL files. */
    DB_SCHEMA,
    /** Implements HttpServletProvider to contribute an Http Servlet to be handled by
     * the SisyphusCore. */
    HTTP_SERVLET
}
```

Listing 2.7: The enum ModuleContributions lists all the different contributions a module can make to the Sisyphus application.

Figure 2.8: The moduleGui implementations of the modules give access to their GUI contributions. The client side of the SisyphusCore in turn gains access to the moduleGuis via the singleton ProxyImpl.
2.6 The Module Contributions

All contributions from modules supported so far are enumerated in the enum `ModuleContributions` (see listing 2.7). New types of contributions have to be added to this enum with a short explanatory text. The `Module`-instance of each module returns a `java.util.EnumSet<ModuleContributions>` instance enumerating the contributions it makes. The different types of contributions are explained in more depth in the following paragraphs.

2.6.1 Protein and Peptide List Columns

Protein columns and peptide columns contributions are combined server and client side contributions which add arbitrarily many columns to the list views of the experiments’ protein and peptide panels respectively. In order to do so they have to supply a list with the names of the columns and a string containing an SQL subquery. This is done by letting the `Module` implementation additionally implement the interface `ProteinColumnContribution` and/or the interface `PeptideColumnContribution`.

It might be perceived as uncommon that this kind of functionality is implemented by directly assembling SQL statements, but Sisyphus is a highly performance sensitive application. Alternative solutions which would add the functionality at a higher horizontal layer would abandon the speed improvements the query optimizer of the database can achieve.

All subqueries for columns in the preexisting Sisyphus system were fairly trivial (this is also due to the fact that the SQL parser used does not understand overly complex queries) and are only dependent on a few primary keys from the queried tables. This enables a simple architecture where the final SQL query is put together on the fly. Like this the number of round trips to the database does not grow with the number of contributed columns and the query optimizer of the database has all the information it needs.

2.6.2 Experiment Panel or Experiment Statistics Panel

These contributions add new GUI panels to the experiments subtree or the experiments statistics subtree respectively and as such are pure client side contributions. The `getExperimentPanelContribution()` and `getExperimentStatisticsPanelContribution()` methods of the `moduleGui` have to be implemented respectively to return an `ExperimentPanelContribution` instance mainly consisting of a `PanelBuilder` to generate the appropriate GUI for a given experiment ID.

2.6.3 Protein Details

A protein detail contribution is shown together with other protein details contributions on a two columns panel when a protein entry in the protein list of an experiment is clicked on. The `getProteinDetailsContribution()` method of the `moduleGui` returns a `ProteinDetailsContribution` delivering a GWT `FieldSet` to be displayed for the specific protein. Figure 2.9 shows how such a protein detail contribution can look like. Protein detail contributions are pure client side contributions.
2.6.4 SQL Queries

A module can define its own SQL SELECT statements for which Java DynaBeans and GWT RPC services can be generated (see section 3.2.2 about the code generator). The queries have to be put into the /sqlQueries directory of the module's bundle. The queries can be stored in two different kinds of files. Those ending with .sql may only contain one query and the filename determines the name of the generated beans and method names. Those ending with .sql2 may contain several queries. Each on its own line, preceded by the query name and a colon.

While a developer implementing a module making an SQL query contribution only needs to add the queries to the server side, SQL query contributions are not pure server side contributions. The CodeGenerator will create not only server side code to access the database with the new queries, but it will also create beans wrapping the returned querying data which are available on the client and the server side and it will also create the appropriate services to access the data from the client side, adding service stubs to the client side code. Therefore are SQL query contributions combined server and client side contributions.

2.6.5 Database Schema Enhancement

Modules might want to add their own tables to the database, either to store data from external sources or to cache expensive computations. To keep the tables of different modules separated from each other and from those of the SisyphusCore, one has somehow to keep track of them. This could be done by storing metadata about the tables somewhere or by appending uniquely identifiable prefixes to the table names. Another solution would be using the schema support of the Postgresql\(^7\) database. Schema in this context is not to be confused with the often used term schema meaning all the table-, column-, key- and index definitions which comprise the data structure definition of a database. To differentiate between the two, a schema in terms of Postgresql is always called a Postgresql-schema\(^8\) in this text. Postgresql-schemas are an additional namespace token between the database and

---


the tables. The rights of database users can be restricted on Postgresql-schema level.

Using Postgresql-schemas means taking advantage of an existing and proven framework. In the end employing a metadata system could mean duplicating the functionality of the Postgresql-schema implementation. The table-prefix idea was discarded because preexisting tables already have names as long as the maximum allowed table identifier length and the relationship between the modules and the prefixes would have to be maintained, ending up with a metadata system anyway. The drawback of using Postgresql-schemas is that the so far used libraries do not support them. Section 3.2.2 explains how Postgresql-schema support was added to them.

A module wishing to add its own tables to the database must do so in a separate Postgresql-schema. The Postgresql-schema will be named like the unique identifier set in the implementation of the abstract class `Module`. The module supplies the data structure definition as one or more DDL-file in the Turbine XML format\(^9\) in the `/ddlFiles` directory. Together with the tables of the Postgresql-schema of the SisyphusCore (the default “public” schema) all foreign-key dependencies must be resolvable, otherwise the tables will not be created.

Each module offering a database schema enhancement contribution is responsible for the evolution of its database schema as well as the management of provenance data for the stored data. Sisyphus does not yet offer predefined mechanisms for doing that.

A database schema enhancement contribution is a pure server side contribution. The CodeGenerator will create classes for modifying (inserting, updating and deleting) data in the new tables, but it will not generate any code for accessing the data in the tables. This can be done with a SQL queries contribution.

### 2.6.6 Http Servlet

A module wishing to contribute an Http Servlet which will be managed by the SisyphusCore implements the `HttpServletProvider` interface. The interface implementation must indicate the `HttpServlet` implementing class and its registration path for the servlet container. Modules can contribute any kind of Http Servlet, it does not have to be one implementing a GWT RPC service. To facilitate the handling of Http Servlets the abstract class `Module` already implements a method returning a default servlet registration path depending on the identifier of the module.

Such an Http Servlet contribution will be managed by the SisyphusCore (see figure 2.10) which will register the servlet for the module as soon as the compilation of the client side was successful. Section 3.5 describes in detail how that works.

From a source code point of view, an Http Servlet contribution is a pure server side contribution. In practice it usually will be called from the client side in conjunction with an SQL queries contribution, which is why the abstract class `Module`, which is accessible both on the client and the server side, offers the default registration path and not the pure server side interface `SisyphusModuleService`.

Figure 2.10: The Sisyphus modules which offer their own Http Servlets (generated by the CodeGenerator or hand written) can let the SisyphusCore bundle handle their registration with the servlets container.
Chapter 3

Implementation

3.1 OSGi

There are various implementations of the OSGi framework. Sisyphus uses the Equinox implementation [Gruber et al., 2005] which is one of five platforms certified as OSGi Service Platform Release 4 compliant. Eclipse is built upon the Equinox framework since version 3. There are no specific technical reasons for choosing Equinox over other implementations, merely familiarity with the platform and the liberal Eclipse Public License it is published under. Sisyphus does not use any Equinox specific enhancements to OSGi at runtime, but at development time it uses some Equinox and Eclipse specifics.

If possible the Require-Bundle manifest header is not used in bundles provided by Sisyphus. This is primarily to avoid hard coded dependencies on other bundles and to keep things as loosely coupled as possible. Secondarily, there are other problems with requiring specific bundles, like split packages, mutable exports and shadowing which are described in detail in the Core Specification [OSG, 2007a, p. 66 ff.]

3.1.1 Declarative Services

The modularized Sisyphus system uses the Declarative Services [OSG, 2007b, p. 281 ff.] of OSGi to manage the life cycle of components and the service dependencies between Sisyphus bundles. The idea behind using Declarative Services for Sisyphus instead of managing bundles the conventional way with ServiceTrackers or ServiceListeners is a reduction in complexity.

As explained in section 1.3 there exist immediate and delayed components. All Sisyphus modules and the CodeGenerator implement a delayed component which is only activated when their service is requested. For the modules the requesting bundle would be the SisyphusCore, requesting all services which implement the interface c.e.s.osgi.SisyphusModuleService, and for the CodeGenerator it is the Equinox command line interpreter which requests the CodeGenerator’s CommandProvider service. The SisyphusCore, the ConsoleLogger, the osgiJavaUtilLoggingHandler and the osgiLog4jAppender all implement an immediate component. They are started as soon as their dependencies are resolved.

---

Listing 3.1: The Declarative Services component declaration of a Sisyphus module. Sisyphus modules are delayed components and only get activated when the SisyphusCore is started. Each Sisyphus module provides a service implementing the `c.e.s.osgi.SisyphusModuleService` interface.

Listing 3.1 shows a typical component declaration of a Sisyphus module. A developer only writing Sisyphus modules will rarely have to write or edit such a declaration. It is generated for each new module by the wizard described in section 3.2.1.

Bundles using Declarative Services components do not have an Activator class like conventional bundles. Instead each component’s declaration file indicates an implementation class which will get instantiated upon component activation. This class also implements the interface of the service the component (optionally) offers. For each service used by the component a binding and an unbinding method, which are called upon service availability or disappearance respectively, is declared. The example in Listing 3.1 binds to a `LogService` if one is available.

One point of possible confusion for developers new to Declarative Services is that the life cycle of a component is not the same as the life cycle of its bundle, they are separate. While a bundle has the states `Installed`, `Resolved`, `Starting`, `Active`, `Stopping` and `Uninstalled`, an immediate component has only the two states `Unsatisfied` and `Active`. A delayed component has additionally a third state called `Registered` (meaning its dependencies are resolved, but its service has not yet been requested). A component can only be activated if the bundle containing it is in state `Active`.

### 3.1.2 OSGi, Third Party Libraries and Classloading

Classloading in OSGi is different than with plain Java. The special classloading architecture of OSGi plays an important part in OSGi’s promise of delivering a well designed componentization platform, because ultimately a classloader is the only
actor in a Java virtual machine that can allow or prevent other parts of the system to use a certain class. But it also leads to problems when using libraries not specifically designed to be used within OSGi, surprising the unsuspecting developer new to the world of OSGi.

Each bundle in OSGi has its own classloader with its parent set to the so called boot classloader. All classloading requests for the java.* package hierarchy have to be delegated to that parent classloader. This is the first place where third party libraries might run afoul if they use their own classloader which might be doing things differently. Interesting exceptions like java.lang.Object Class Not Found can be the result.

Packages can be specifically declared as “boot delegated” by the runtime parameter org.osgi.framework.bootdelegation. Such a package is always first searched via the parent classloader. For Sisyphus it was necessary to declare the package javax.xml.namespace as “boot delegated” because it is supplied by Java itself and not somewhere in a bundle.

If a class to be loaded is imported from another bundle (as indicated in the manifest of the bundle), the classloader delegates classloading to the classloader of the other bundle. This also ensures that classes which are in turn needed for an imported class are reachable even if they have not been exported.

In a next step the class is searched for in required bundles (Sisyphus bundles do not use Require-Bundle). If still not successful, the classloader searches the classpath of its own bundle and then of its fragments (no fragment bundles have been used in Sisyphus).

As a last resort, a so-called dynamic import is tried: If the package of the class is exported by any bundle that bundle’s classloader is used.

If the class still has not been found a Class Not Found exception is thrown.

Not only libraries working with their own classloaders can be problematic. Sometimes a library does not even have to bring its own classloaders with it. The Java compiler itself does sometimes insert a reference to a superclass if a method of that class is used in a subclass. This is seen as a bug in the Java compiler by OSGi proponents.

If a library causes problems with classloading, then there are different possibilities to handle them. Within the Sisyphus application the most used workaround is to pass the bundle with the guilty library the classloader of the requesting bundle. Usually one will obtain the classloader of a certain class in the requesting bundle and later add it to the currently executing thread in the bundle with the classloader modifying library, as exemplified by the following two lines of code:

```java
ClassLoader classloader = SomeClass.class.getClassLoader();
...
Thread.currentThread().setContextClassLoader(classloader);
```

Sometimes it will also be necessary to import additional classes from the badly behaving library, because the chain for loading indirectly used classes might be broken.

The third workaround used within Sisyphus is the above mentioned parameter org.osgi.framework.bootdelegation, which can be set at the startup of the OSGi framework.
3.2 Module and SisyphusCore Development

Figure 3.2: At development time the “New Sisyphus Module Project Wizard” can be used to create a new project in the Eclipse development environment. The CodeGenerator can create schemas and tables in the database according to DDL files and generate beans and GWTRemoteServiceServlets according to given SQL SELECT queries.

As shown in figure 3.2 the modularized Sisyphus offers two tools to facilitate development. One is the new module project wizard described in section 3.2.1 and the other the so called CodeGenerator detailed in section 3.2.2. While the wizard concerns only the module development, the CodeGenerator is used for both module and SisyphusCore programming.

3.2.1 Sisyphus Module Project Wizard

The goal of the new Eclipse project creation wizard was to make Sisyphus module development easier for new developers and less tedious for those already knowing the modularized Sisyphus.

The wizard uses a set of templates to create a new Eclipse bundle project complete with manifest file, component description, *Module* and *ModuleGui* skeletons, */sqlQueries* and */ddlFiles* directories if applicable.

The implementation of the wizard uses code from the class *org.eclipse.ui.wizards.newresource.BasicNewProjectResourceWizard* from the Eclipse project. The class is unfortunately not suitable for extension and therefore large parts of code were copied. The original code is under the Eclipse Public License 1.0 which does not put many restrictions on the distribution of the code or resulting programs. It essentially demands that other developers are indemnified against legal claims resulting through the use of the code, similar to what other libraries included in the Sisyphus application request.

Using the Module Project Wizard

The Sisyphus Module Project Wizard can be directly installed into a local Eclipse installation from the update site called *wizardRepo* in the SVN repository of Sisyphus.

---

When defining a new install site with “Add site...” within the Eclipse update manager, the directory containing the site.xml file (i.e. the base directory of the wizardRepo project) has to be chosen as local update site.

After its installation the window reachable via the menu File/New Project has a new entry called Sisyphus with a subentry Sisyphus Module Project. On the first wizard page, the user can choose the name of the new module, what contributions it will make and, if applicable, the Eclipse working set she wants to add it to (see the right-hand side of figure 3.3). Inter project dependencies can be indicated on the second page.

After running the wizard a new project complete with bundle manifest and component declaration, source folders and skeletons of the different module implementing classes exists in the Eclipse workspace. Positions in the code where the user has to add implementing code are denoted with TODO task tags. Some of the generated files are not nicely formatted, pressing Ctrl+Shift+F when editing the respective file in Eclipse auto-formats it nicely.

3.2.2 CodeGenerator and Database Access Layer

The “CodeGenerator” is an OSGi bundle aiding in managing the database access layer. It is the successor of the ANT-script build-eclipse.xml used in prior versions of Sisyphus. It is an Equinox console tool and therefore Equinox specific. The command generate followed by a bundle number initiates the code generation for that bundle. The indicated bundle must either be a Sisyphus module or the SisyphusCore.

In a first step the CodeGenerator reads the DDL files (see section 2.6.5) from the given bundle and additionally from the SisyphusCore (needless to say only once if the given bundle is the SisyphusCore) to construct an in-memory model of the
database. It uses an adapted version of Apache DDLUtils\textsuperscript{4} to construct the database model. The model is checked for consistency, especially all foreign-key relations have to resolve. If the model is consistent in itself the not yet existing tables in the database are created using the SchemaWriter class.

Afterwards classes for code insertion, updating and deletion are written. If the generate command was called on the SisyphusCore all its formerly generated DB manipulating classes are replaced. If the command was called on a module, only classes writing to the Postgresql-schema of the module are generated and placed into the source code of the module. The CodeGenerator depends on the (expanded) bundles being developed within Eclipse. Otherwise it will fail with an appropriate error message and it will not write anything into the bundles. The created classes are called \texttt{DBDeleteUtilities}, \texttt{DBInsertUtilities} and \texttt{DBUpdateUtilities} respectively (for modules and SisyphusCore).

If there are SQL files present in the bundle, the SQL parser will parse them and build a model of the query. With different templates and using the created model of the database, POJOs representing the queried data, DB querying classes (using DDLUtils) and server and client GWT RPC code to easily access the database from the client side are created. The following list enumerates the created files. If not indicated otherwise, the filenames are prepended with the module’s unique identifier (with SisyphusCore in case of the SisyphusCore). The filenames are prefixed with an ellipsis (…) to indicate this.

\texttt{..DBUtils}  This class provides the server side with methods to retrieve data from the DB according to the provided SQL queries. For each query a \texttt{get...()} and \texttt{getFirst...()} method exists, named according to the query’s name, returning all results or the first result of the query.

\texttt{..QuerySQLBuilder}  The \texttt{..DBUtils} class uses the \texttt{..QuerySQLBuilder} class to retrieve the needed query according to the \texttt{get...()} and \texttt{getFirst...()} method called. This class is usually not directly used by developers. It resides on the server side.

\texttt{DynaBeans}  For each query a class named like the query (without prepended module identifier) is created. It is a DynaBean used with the DDLUtils library\textsuperscript{5}. These container classes are what the \texttt{..DBUtils} class returns as results. The created DynaBean classes are usable both on the server and the client side.

\texttt{..DDLUtils}  The generated \texttt{..DDLUtils} class wraps the results from the queries into the appropriate DynaBeans. This class is usually not directly used by developers and resides on the server side.

\texttt{..Service}  This interface defines how the client can access the data from the server. It mirrors the methods available in the \texttt{..DBUtils} class and resides on both the client and the server side. It extends the \texttt{RemoteService} interface of GWT.

...ServiceImpl This class implements the interface ...Service by forwarding requests to ...DBUtils. It extends the class OsgiGwtRemoteServiceServlet which in turn wraps the GWT class GwtRemoteServiceServlet to provide access to the GWT RPC mechanism inside OSGi. This is a purely server side class.

...ServiceAsync This is an interface giving the client asynchronous access to the service defined in the ...Service interface. Figure 3.4 shows an excerpt from an example of such an interface. The developer always has to give an accordingly typed AsyncCallback object to which the service will return its results. This interface resides on the client side.

```java
public void getExample(
    Integer exampleId,
    AsyncCallback<List<ch.ethz.sisyphus.modules.some.generated.containers.Example>> callback);
```

```java
public void getFirstExample(
    Integer exampleId,
    AsyncCallback<ch.ethz.sisyphus.modules.some.generated.containers.Example> callback);
```

Listing 3.4: An excerpt from a generated interface SomeServiceAsync for a fictional query named “Example” and a fictional module named “some”. For every query which a module provides, a DynaBean class will be generated named after the query. The fictional query in this example would have only one parameter of type INTEGER.

During this thesis project the DDLUtils library has been adapted to support Postgresql-schemas as far as the code paths are used by the CodeGenerator and SisyphusCore (it still does not support Postgresql-schemas when writing a DDL file, this would be a relatively small change though). The Postgresql-schema support was implemented by using the schema field of DDLUtils’ org.apache.ddlutils.model.Table class, which was already present but not read or written to anywhere in the code, and adapting all table handling code areas of DDLUtils, including its DDL parser.

The classes DBWrapper, SchemaWriter and TableWrapper, as well as all the Velocity templates and classes and the SQL parser used to drive the templating engine are now Postgresql-schema aware as well.

The DDLUtils library has also been adapted to build a database model not only from one DDL file, but from arbitrarily many. With this change, modules can provide their own tables to the database. The developer can now choose between building an in itself consistent model of a database defined in DDL files or to skip the consistency check. To this end, the class org.apache.ddlutils.io.DatabaseIO now embodies in addition to the preexisting method read(File) also the methods read(Collection) and readWithoutInitDB(Collection) which build a database model from several files, the latter without checking it for consistency. (DDLUtils does not support Java generics yet, the Collections have to be Collections of objects of type java.io.File.)

---

The CodeGenerator uses this to first check the combined DDL files from a module and the SisyphusCore for missing foreign-keys and then build a (not necessarily in itself consistent) model only for the module, later writing the tables of this smaller model to the database.

All the templates and all code generating classes have been revised and were adapted. Java generics are now used in all generated classes. Generated code has now a header stating that it was generated and when it was generated. Generated Java code within modules does not share packages with hand written code to reduce potential developer confusion.

LONGVARCHAR and VARCHAR ‘NULL’ values were stored as the character sequence ‘NULL’ instead of SQL ‘NULL’ values. This has been rectified.

3.3 Logging

OSGi offers its own logging framework with the two services called LogService and LogReader which are specified in [OSG, 2007b, p. 5 ff.]. The LogService gathers all logging messages from its subscribers while the LogReader disseminates the messages which it receives from the LogService to its subscribers.

![Diagram of logging services](image)

Figure 3.5: The Log4jAppender gathers all logging messages reaching the Log4j logging system. The JDKLogAppender does the same for the native Java logging framework and the one from Apache commons. All messages end up at the ConsoleLogger where they are filtered and written to the console.

Unfortunately, different libraries used by Sisyphus use different logging facilities. Jetty uses the Apache commons logging framework\(^7\), even the Jetty version which is available as an OSGi bundle does not use the OSGi LogService. Log4j\(^8\) is used by preexisting Sisyphus code and several libraries. The native logging framework of Java (in J2SE since version 1.4) is also used. To merge these log message streams and divert them to the OSGi LogService, three bundles were written, one gathering

---


all Log4j messages, the second collecting data from Apache commons and the native Java logging systems, and a third filtering and then writing the messages received from the LogReader to the console. Figure 3.5 gives an overview of this system. The ConsoleLogger can easily be replaced or accompanied by another bundle e.g. for logging to a file or a database.

Regarding the now implemented logging appenders a word of caution is expedient: Some applications, e.g. gwt-log\(^9\) and Velocity\(^10\), try to auto detect what logging frameworks are present on startup and then decide based on their own preset preference, which framework to use. A change in the activation order of bundles and components can therefore have unforeseen consequences on logging.

### 3.4 Configuration

The Sisyphus code already used the Apache commons configuration tool\(^11\) (employing an XML configuration file) before beginning with the horizontal modularization. Switching to the Config Admin Service (see [Wütherich et al., 2008, p. 227 ff.] and [OSG, 2007b, p. 65 ff.]) of OSGi would be possible. Its main advantage is the possibility to change the configuration at runtime which is something Sisyphus does not need at the moment. The main cost of using the Config Admin Service would be increased complexity and the development time spent to change from one system to the other. Therefore the old configuration system has been kept.

Several configuration properties previously defined in separate places of the source code have been moved to the preexisting configuration file (living under the package ch.ethz.sisyphus.config in the SisyphusCore bundle), especially all absolute filesystem paths. The concrete configuration file is chosen dynamically at startup to make it more convenient for different developers to use Sisyphus from the same SVN repository. The file ch.ethz.sisyphus.config.Config.java contains the necessary selection logic. If the Java system property ch.ethz.sisyphus.ConfigFile is set (relative to the Config.java file or absolute, settable e.g. with the -D parameter of Java or at runtime before the SisyphusCore bundle starts) the given configuration is chosen. Otherwise it will look for configuration files based on the system’s hostname. If neither strategy is successful, Sisyphus stops its execution.

The configuration file defines the database connection and credentials to be used, filesystem paths (especially the directory where the client side code will be extracted to and compiled, for performance reasons this should not be on a network drive) and whether the compiled GWT client code should be obfuscated and optimized (advisable in a production setting) or not (advisable during development).

### 3.5 GWT and OSGi

Running GWT within OSGi, especially running modularized GWT code inside OSGi, requires a few issues to be solved which will be explained in the following paragraphs.

On one hand the RemoteServiceServlet class of GWT (enabling RPC calls from the client to the server) uses its own classloader, which specialcases the serialization

---


and deserialization of the classes in the java packages `java.lang`, `java.sql` and `java.util`. OSGi on the other hand requires that classloading requests for the whole `java.*` package hierarchy are delegated to the parent classloader\(^{12}\). This is in more detail described in section 3.1.2.

At first during this thesis project, only objects outside the `java.*` hierarchy were sent. Suddenly after enabling more services, spurious `java.lang.Object Class Not Found` exceptions showed up. After identifying the problem the adapter class `OsgiGwtRemoteServiceServlet` was implemented. It has a constructor which takes a classloader instance. On every service call this classloader is attached to the current thread.

Every module willing to use GWT RPC calls has to use the adapter class `OsgiGwtRemoteServiceServlet` and provide its own classloader upon instantiation. Additionally its bundle has to import the packages `com.google.gwt.user.client.rpc.core.java.lang`, `sql` and `util`. GWT maps the `java.*` hierarchy to those classes and serialization and deserialization would fail without them.

The `GwtRemoteLoggerServiceImpl` class used for logging messages from the client on the server has been adapted very similarly with a class called `OsgiGwtRemoteLoggerServiceImpl`.

The OSGi Http Servlet implementation, the Http Service [OSG, 2007b, p. 17 ff.], supports only the Java Servlet API in version 2.1. GWT requires a higher version for RPC calls. An initially used adapter class extending the `RemoteServiceServlet` class of GWT worked well. Later during the thesis project the OSGi Http Service implementation of OSGi was replaced by Jetty and this adapter class became redundant. If in the future Jetty were to be replaced with a different OSGi Http Service provider, its Java Servlet API version would have to be checked.

### 3.6 Extraction, Modification and Compilation of the Client Side

As laid out in section 2.4.2, the GWT client code has to be compiled “on the fly”, i.e. every time a module contributes new client source code that code has to be extracted, the `ProxyImpl` class and a GWT module description have to be modified and all the client code compiled.

GWT compiles can take their time. Sisyphus is quite a large system and it takes several minutes on an Intel Pentium 4 processor to compile the Sisyphus client code for all supported browser platforms. Setting the property `user.agent` in the configuration of the GWT base module to the browser used during development time can reduce the time needed for compilation by one third.

Another step taken to minimize extraction and compilation time was to divide the source code into client and server side code. Before the modularization all code was stored together in the `/src` directory. Now in the SisyphusCore as well as in the modules the code is split up into client code in the `/gwtCode` directories and the server code in the `/src` directories. There exists code which is compiled for both the client and the server side. This code resides in the `/gwtCode` directories too.

Probably even more important is the logic for starting and stopping the extraction,

\(^{12}\)A good overview over how classloading regularly should look like within OSGi is given by the diagram in [OSG, 2007a, p. 54].
modification and compilation machinery. The ExtractionAndCompilation class implementing this mechanism has two executors waiting for tasks to do. One for extracting all the client source code from incoming bundles and the other, blocked as long as the extraction executor is not idle for more than a few seconds, for modifying the ProxyImpl class and starting the compilation. If at any time new work to extract and compile comes in, the compilation is immediately stopped and the whole procedure restarted. The whole process is depicted in figure 3.6.

Figure 3.6: Extraction and compilation of the client side Google Web Toolkit source code. The extraction executor is run after the SisyphusCore has been started. If at any time new modules with more GWT source code to be extracted are registered with the core the state changes to EXTRACTING again. Upon successful compilation the SisyphusCore registers all GWTRemoteServiceServlets with the Jetty servlet container.

The whole procedure is started as soon as SisyphusCore itself starts up. As soon as the ExtractionAndCompilation class reaches the “DONE” state, all Http Servlets of the SisyphusCore and those given to the core for registration by modules are registered with the servlet container (Sisyphus uses Jetty).

Not all compilations which should fail due to bugs in the client source code (at

13Interestingly, the GWT compiler process sometimes needs two emulated Ctrl-C keystrokes sent to its input stream and the destroy() method called to stop it, otherwise it would just go on.
development time) are detected by the SisyphusCore because the GWT compiler tries to work around bugs by disabling the faulty parts of the code instead of stopping the compilation. The emitted logging statements explain which parts of the source code have consequently been omitted by the compiler.

As a side note, the `ExtractionAndCompilation` class is one of several classes which implement a singleton. Instead of implementing them with one of the common singleton patterns, e.g. with a public final field or with a static factory (and a private constructor in both cases), singletons are simply enums with one single element (for simplicity’s sake called INSTANCE). This has the advantage that the JVM guarantees the singleton property and there are no serialization headaches [Bloch, 2008, p. 18].

### 3.7 Module Contributions

The module contributions are implemented according to the architecture set out in section 2.6.

A newly devised module is shown in figure 3.7. It is implemented as a combination of the three contributions Http Servlet, experiment panel and SQL query. It shows how a part of a new cross-experiment panel could look like. The user can drag an experiment from the left-hand side experiment navigation tree to another experiment. An RPC request (built with the CodeGenerator for the contributed SQL query) will be sent to the server side. The result is graphically interpreted on the client side. This example shows how much the proteins found in two perturbation runs overlap. Perturbation runs are experiments where one parameter, e.g. the temperature, of the examined system is changed systematically from experiment to experiment). The module is still a prototype and would have to be revised for production use.

Figure 3.7: An example cross-experiment display. The user can drag and drop an experiment tree node into the panel and the calculated protein occurrence overlap is shown.
One of the examples where preexisting functionality was removed from the not modularized Sisyphus system and placed into its own module is the iTRAQ [Gafken and Lampe, 2006] experiment panel. Figure 3.8 shows how it looks like since its inception as module. Except for the tooltip for its tree node it looks like it did before modularization. Below the surface not only the experiment panel builder for iTRAQ has been removed from the SisyphusCore but also its SQL query and the accompanying GWT RPC service. The core is now devoid of any iTRAQ-reminders where before parts from different core layers had to know about it. This exemplifies nicely the new “vertical” modularization.

Figure 3.8: The iTRAQ-module’s experiment panel contribution. The opened experiment’s sub-tree is now alphabetically ordered to avoid arbitrary node reordering upon module installation which could confuse users. The tooltips for better discoverability are also new.

### 3.7.1 Usability Considerations

Figure 3.8 also shows two precautionary steps which have been taken to avoid degradation in usability. A dynamic system like the modularized Sisyphus application can confuse users, because parts of the user interface change upon installation or removal of a module. To alleviate this potential user confusion a bit, tree nodes in subtrees where elements can come and go dynamically are now sorted alphabetically. A user will therefore find nodes (barring node renaming) always in the same order, abiding to the principle of least astonishment [Marcus and van Dam, 1991]. Alternatively one could devise a system where the module developers could indicate where exactly in the interface their contribution should be placed, e.g. by using a priority number, but this would need coordination among the module developers. Additionally, nodes contributed by a module now show a tooltip with a short text giving more information upon hovering over the nodes with the mouse cursor.
Chapter 4

Evaluation

4.1 Performance Impact of Modularization

One of the auxiliary goals stated in section 1.2 was that the performance of the modularized Sisyphus system should not be worse than of the unmodularized application. The following two sections will look at whether this has been achieved from an argumentative and a measurement based view.

4.1.1 Performance Deliberations

Except for the issue of having to recompile the client side of the application upon module installation as detailed in section 2.4.2, the modularization of Sisyphus should not have had perceivable influence on the performance of the application.

There are four performance sensitive hot-spots in Sisyphus where a change in the architecture could have had an impact:

- The import of data from external sources. This code has remained untouched by the modularization.
- The querying of the database. The queries remained the same as before therefore bringing no change in performance with them. The exception is the possibility to dynamically add or remove subqueries from some queries in order to add or remove columns to protein and peptide lists. Given that the same columns as before are requested, the queries differ only in the order of the subqueries in the main SELECT clause, causing no performance impact.
- The serialization and deserialization of data sent to the client remained identical.
- As discussed in section 2.4.2 a fully dynamical modularization of the client side (without support from GWT for doing so) would have had a huge impact on the performance. The chosen solution with its compilation of the client side to a monolithic GWT application avoids this. There have been some changes to the client side code in order to accommodate for the modules and some refactoring has been done, especially adding Java generics where indicated. A manual side by side comparison of the GUI did not identify any perceptible performance differences.

Given those considerations, no performance impacts on the end-user are to be expected.
4.1.2 Performance Measurements

To check the theoretical and subjective findings from above, especially regarding the performance of the application as perceived by the user, several measurements comparing the performance of Sisyphus before and after the horizontal modularization were taken.

All of them were executed on the same personal computer with a 3.40 GHz Intel Pentium 4 CPU and 3.4 GB memory, running both the client and the server side (including the database server). This has been deemed acceptable because the measurements are not meant to find performance bottlenecks, only performance regressions. Firefox 3.0.10 was used as the browser for executing the client side. All measurements were taken on the same data set for both the old and the new system.

For all measurements the average runtime, the standard deviation and the minimal measured time (where useful, serving as an indicator for the best achievable time, i.e. a maximum minimum) are reported. The error bars in the diagrams measure twice the value of the standard deviation.

<table>
<thead>
<tr>
<th></th>
<th>$\bar{o}$</th>
<th>$\sigma$</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSGi</td>
<td>1059</td>
<td>18</td>
<td>1007</td>
</tr>
<tr>
<td>Jetty</td>
<td>2508</td>
<td>832</td>
<td>1530</td>
</tr>
</tbody>
</table>

Figure 4.1 compares the startup time of OSGi, used in the newly modularized Sisyphus system, to the startup time of the servlet container Jetty, used for the unmodularized Sisyphus application. The startup times are only partly comparable. The startup time reported for OSGi is the time after which the bundles consoleLogger, osgiJavaUtilLoggingAppender, osgiLog4jAppender, javax.servlet, org.apache.commons.logging, org.apache.log4j, org.eclipse.equinox.ds, org.apache.log4j, org.eclipse.equinox.http.jetty, org.eclipse.equinox.http.servlet, org.eclipse.equinox.log, org.eclipse.equinox.util, org.eclipse.osgi, org.eclipse.osgi.services and org.mortbay.jetty have been started. This equates to the base system the modularized Sisyphus runs on. The startup time for Jetty includes the time for registering the servlets of Sisyphus with the servlet container and is therefore expected to be longer. Why the startup time for Jetty fluctuates rather heavily is unknown.
Figure 4.2: The time a call from the client side to the SimpleSisyphusService needs until the client registers its return. Each value has been measured 10 times.

Figure 4.2 shows how long a call to the SimpleSisyphusService for populating the whole experiment tree takes from the client side for both the modularized and the unmodularized Sisyphus application. No significant difference between the two systems has been found.

Figure 4.3: The time until the whole Sisyphus page has loaded (i.e. its DOM has been created by the browser). This includes both calls to the server side and client side rendering. Each value has been measured 10 times.

Figure 4.3 contrasts the time needed to load the whole Sisyphus application in the browser, including calls to the server and rendering of the JavaScript generated web page, between the two systems. No significant difference in loading time has been found. What is striking is the much higher variance in the page loading time for the unmodularized system versus the one for the modularized application, caused by two outliers in the data set of the unmodularized system. Removing the outliers yields a much lower standard deviation of 0.8 s for the unmodularized Sisyphus system and an average within the standard deviation of the modularized Sisyphus
application. The cause of the outliers is unknown. They could be the result of an interfering JVM garbage collector running on the server side or something similar.

<table>
<thead>
<tr>
<th>s</th>
<th>φ</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modularized</td>
<td>5.17</td>
<td>3.27</td>
</tr>
<tr>
<td>Unmodularized</td>
<td>5.85</td>
<td>3.4</td>
</tr>
</tbody>
</table>

Figure 4.4: The time until a protein list has completely loaded was measured. 5 different protein lists were used, for both systems the same set of lists.

The measurement shown in figure 4.4 is conceptionally different to the preceding measurements as it compares the loading time for a set of 5 protein lists instead of loading the same list 5 times. The load times for both systems are in the same range.

It can be concluded that the modularization of Sisyphus did not have a perceivable impact on its performance except for the need to recompile the client code upon module installation.

4.2 Evaluation of the Approach Taken

When starting this thesis project it was expected that most of the available time would be invested in defining module contribution types and refactoring the existing system into modules offering such contributions, even leaving some time to design new views on the protein data. Instead two sets of problems used up most of the time:

One was the issue of the not-so-well modularizable client side code created by the Google Web Toolkit, which has been detailed in section 2.4. The design and implementation of the workaround to this problem has taken a considerable amount of time.

The other group of problems was related to OSGi in combination with third party libraries. These issues were encountered when moving the server side part of the Sisyphus application onto the OSGi platform. From mundane things like problems with the different logging frameworks used (see section 3.3) to the much more intricate classloading problems in conjunction with other libraries (see section 3.1.2), all those problems took a considerable time to solve.
In hindsight, three notable mistakes were made during this thesis project: (1) The envisioned architecture was not questioned enough, probably because it seemed sound and is very intuitive. (2) The move to OSGi and the modularization of the GWT client side were carried out in parallel with one code basis, mixing different concerns and complicating development, especially making the identification of the source of a newly upcoming problem difficult. (3) The impedance mismatch between the dynamic OSGi framework and the forced monolithic client side, requiring much otherwise unneeded code, was underestimated.

Looking back, a more sound approach would probably have been to stage the development: (1) The problem of modularizing GWT should have been tackled first. (2) If a similar solution for modularizing GWT would have been found as during this thesis project, it would probably have been advisable to postpone the switch to OSGi and use an intermediary solution in the meantime (see paragraph below). (3) In a last step the pre-modularized system could have been switched over to OSGi.

The intermediary, non-OSGi solution mentioned above could have looked like the following: A vertical modularization could have been done by creating a separate Java Eclipse project per module, giving each access to the classpath of the core part, and compiling everything, client and server side, to two monolithic parts. On the positive side this would have allowed to start the modularization without complicating it through new constraints from a new platform. On the negative side it might take more effort from the developers to really decouple a system if the platform does not enforce it.

Depending on whether new problems will arise from using a monolithic client side on the intrinsically modular OSGi platform, this intermediary solution could still be useful now. As it is difficult to predict whether such problems will surface, no advice on how to proceed will be given here.
Chapter 5

Conclusion

The goal of further modularizing the Sisyphus proteomics tool was laid out in section 1.2. This chapter discusses in how far that goal has been met and how the modularization affected, positively or negatively, other parts of the system.

**Architecture** The architecture with its special treatment of the client side is more intricate than initially envisioned and wished for. The probably biggest problem with an architecture which does not follow well known structures found in other systems is that developers have to learn it explicitly to work on the application. Nonetheless can be concluded that the desired “vertical” modularization has been achieved. It is now possible to create modules which touch all “horizontal” levels of the Sisyphus system without modifying the core system at all. A module can be from very lightweight to very encompassing and complex. A module can use the resources offered by the SisyphusCore or it can implement its own solutions (it could e.g. devise its own caching system). The module developer can therefore enjoy much flexibility.

When modularizing into a new direction, the modularization in another direction can be at risk. When looking at the architecture of Sisyphus this could have meant loosing the “horizontal” layering. If one considers the currently implemented system, the DB access layer is still well decoupled. Someone wishing to replace it would have to replace the libraries and the DDLUtils specific code in the CodeGenerator and then apply the CodeGenerator to the SisyphusCore and to all modules, automatically replacing the DB access code. The dependency on GWT is very high on the client side. That has been the case before and is now probably even more pronounced.

**Dynamism** The system is dynamic. Installation, updating and removal of modules is possible at all times and is trivial for the system administrator, but takes some time because the client side has to be recompiled. With current personal computers this takes several minutes. On one hand this does not pose a problem for end-users of the system because module updates will not happen on e.g. an hourly basis. For developers on the other hand this is annoying because many wish to continuously test their code. Two things can alleviate the problem somewhat: GWT can be instructed to compile only towards one platform, cutting down compilation time by approximately one third for the Sisyphus platform. Modern IDEs catch most errors which would otherwise be found at compile time during code entry, requiring less testing and therefore less compilation runs.
**Performance** Maintaining performance was stated as an auxiliary goal at the beginning of this text. Section 4.1 shows that this goal has been achieved with the exception of having to pay the price of a client side compile on each module installation or deinstallation. No perceivable performance impact on the end-user experience has been found.

**Developer Friendliness** Two negative points concerning developer friendliness were already mentioned: The unfamiliar architecture and the compile time penalty upon module installation. On the positive side the modularization as such already helps module developers to concentrate on their task at hand.

The new Eclipse module project creation wizard decreases the time needed to write boiler-plate code both for new and for experienced developers and it guides new developers to the code parts where the module implementation has to be done.

The CodeGenerator replaced the existing ANT-script and as such does not help the developer more than the old system. Only the end-to-end support of Java generics in the generated code helps the developer to avoid type errors and the separation of generated and hand written code helps avoiding developer confusion.

On the negative side, it has to be said that the complexity of the whole system has grown. The OSGi framework was added to the list of technologies a Sisyphus developer has to be familiar with. Additionally, OSGi can complicate the use of other third party libraries, especially if they bring their own classloaders with them like GWT and Velocity.

**Maintainability** Principally, modularization and therefore looser coupling of a system positively affects its maintainability. All in all this should also be the case for the modularized Sisyphus system.

Maintainability-wise one can mention on the negative side, that the two third party libraries DDLUtils and Velocity were adapted to the needs of Sisyphus. Because these adaptations were not feasible without touching the source code of the libraries (instead of e.g. using adapter classes), the libraries will also have to be maintained as long as they are needed.

The added complexity through the unusual architecture and through the introduction of OSGi not only impacts developer friendliness but also maintainability.

**End-User Friendliness** The end-user will rarely realize a difference to the unmodularized Sisyphus application. The most obvious change for the user will be the appearance or disappearance of added or removed GUI contributions on system updates. A change of order among the user interface parts not installed or removed by an update to the Sisyphus system would confuse the user and has been prevented by ordering interface items like tree nodes alphabetically. In some cases this might mean the loss of a preexisting work-flow-driven or semantic ordering, potentially negatively influencing usability.

The real benefits to the end-user should come from more and better analysis tools thanks to the benefits of the modularized system for the developers.

**Choice of Platform** As set forth in section 1.3 OSGi promises to reduce complexity, enable reuse, allow for easy deployment, enable dynamic updates and be small and fast and there is little doubt that the OSGi framework is a good solution
to build a componentized application from scratch. There is also good evidence that it is possible to move big and complex projects successfully onto the OSGi platform, as the transition of Eclipse to OSGi has shown [Gruber et al., 2005].

Nonetheless did it take an unforeseen long time to move the existing code base over to OSGi. Part of the reason was that the approach taken for doing so was non-optimal as explained in section 4.2 and another that the choice of third party libraries already used by Sisyphus was not optimal for being used in conjunction with OSGi.

One can conclude from this project that OSGi in effect does allow for well performing componentization with easy deployment and updates. Whether a reduction in complexity has resulted in this case is difficult to tell yet an will be seen during future development of the Sisyphus application. One of the questions remaining is, whether the relatively high initial cost of moving to OSGi now has been paid or whether integration problems will persist and haunt future developers.

**Declarative Services**  So far the choice of Declarative Services as the tool for module life cycle management has been proven good. The promise of reduced complexity in handling component interdependencies has been kept.

### 5.1 Future Work

**More Modularization Work**

While the code needed to modularize the Sisyphus system is in place, there remain many parts which have not yet been modularized. E.g. only some of the entities shown on the protein pages are already modularized as a protein detail contribution.

Not only is not everything yet modularized in terms of the contribution types defined during this thesis project, but there certainly will be opportunities to create more contribution types and define more abstraction interfaces, be it on the client or the server side or combined contributions.

The modularization work during this thesis was purely in the vertical direction, only with an eye on not making the horizontal layer tighter coupled. It would probably be a worthwhile endeavor to make the SisyphusCore smaller than it is now. A lot of the libraries could be put into their own bundles, letting the SisyphusCore only import what it really needs. It could also be an interesting undertaking to try putting as many of the horizontal layers as possible into bundles as small as possible.

Once Google Web Toolkit supports the dynamic addition of precompiled parts a lot of the complex runtime code driving the extraction, modification and compilation of the client side code could be abandoned, lowering maintenance cost. The initially envisioned architecture with one declarative service per module contribution could be reevaluated.

**Third Party Libraries**

All used third party libraries could be examined to see whether they are (1) still needed, (2) available in an OSGi enabled version and/or (3) whether they could be replaced by another library with OSGi support or even a service from the OSGi service compendium.
The GWT-EXT widget toolkit library used on the front-end is being discontinued\(^1\) and should probably be replaced. The GWT-EXT developers moved on to SmartGWT\(^2\) which might be a possible replacement.

**Persistence Layer**

On one hand the custom-made three-part persistence layer of Sisyphus offers great performance. On the other hand new developers are unfamiliar with it, it supports only a limited choice of SELECT statements, it is quite complex and it requires the maintenance of an adapted DDLUtils library and a special SQL parser. Commonly used Java persistence layers could be evaluated to see whether some of them would offer the same performance with a lower maintenance cost.

**Overall Architecture**

As detailed in section 4.2 the Sisyphus application could refrain from using OSGi as long as Google Web Toolkit on the client side does not allow for precompiled components to be linked together. This would make a lot of the now implemented logic superfluous, reducing complexity and improving maintainability.

**Database Schema Evolution**

The Sisyphus application does not have automated support for database schema evolution. Only new tables can be added to the database by adding them to a DDL file in the SisyphusCore or a module and then running the CodeGenerator for the respective bundle. There are no provisions for changing tables, e.g. removing or adding tables columns. The SisyphusCore and the modules can adapt the database manually by directly accessing the JDBC connection to the database and issuing the appropriate SQL statements. A more abstracted method could be implemented.

**Database Access Control**

While on the one hand Postgresql-schema support has been implemented during this thesis project, nicely separating the data of the modules from the SisyphusCore, there are on the other hand no enforced restrictions for the modules in place yet, just the appeal to the module developers not to modify content out of their scope of responsibility. Each module would need its own database user and for each module the Postgresql-schema would have to be set read-only for that user while full access would be granted to the module’s own Postgresql-schema.

**Data Provenance**

There is no predefined mechanism in the Sisyphus application to manage provenance data. Each module has to devise its own process to store provenance data. A unified approach could be offered.


Security Considerations

Currently Sisyphus does not put any restrictions on what a user can do. If this will be a requirement in a future version, a validation layer on the server side will be necessary. Google Web Toolkit facilitates the development of powerful client side applications, making it possible to do a lot of data manipulation already in the browser. Data sent back to the server nonetheless has to be checked for validity, just as with conventional web application architectures.

User Authentication, Authorization and Roles

Sisyphus currently is not user and user role aware. In the future there will certainly be the question of restricting access to the application depending on different user roles. E.g. some users might have read-only access while others might be allowed to import new experiment data or administrator all the data in the system.

The Sisyphus Module Project Wizard

The Sisyphus Module Project Wizard developed during this thesis project could be further enhanced. Properties like a unique identifying name or a short description of the module, which have to be added to the class Module anyway, could already be configured via the wizard. Furthermore some of the generated Java source files are not very nicely formatted. The wizard could call Eclipse’ auto formatter after file creation to rectify this. And it should be adapted to newly defined contribution types.
# List of Figures and Code Listings

2.1 The architecture of the Sisyphus system before modularization showing the most important libraries and programs used. .......................... 12

2.2 An abstracted view of the Sisyphus system’s architecture before modularization. Compared to more customary web application architectures more of the application logic resides on the client side. ............... 13

2.3 The conceptual architecture of the Sisyphus system after modularization. The goal is to add a “vertical” modularization without compromising the “horizontal” layering. ................................. 14

2.4 The class `ProxyImpl` after the activation of three modules. This class is modified by the SisyphusCore at server side runtime before client side compilation. Like this the client side has access to the implementations of the interface `ModuleGui` and via them also to the implementations of the abstract class `Module`. ................................. 17

2.5 The core of the Sisyphus application declares itself as a consumer of services which implement the interface `c.e.s.osgi.SisyphusModuleService`. A Sisyphus module is by definition a bundle which offers a service implementing that interface ................................................................. 18

2.6 A module which makes client and server side contributions is defined by the implementations of the two interfaces `SisyphusModuleService` and `ModuleGui` and of the abstract class `Module`. A `SisyphusModuleService` instance can not exist on the client side because it has to maintain a reference to the bundle classloader. A `ModuleGui` instance only exists on the client side and manages all the contributions of the module to the GUI. The `Module` instance offers all the properties of a module which are shared among server and client side. ................................. 18

2.7 The enum `ModuleContributions` lists all the different contributions a module can make to the Sisyphus application................................. 19

2.8 The `moduleGui` implementations of the modules give access to their GUI contributions. The client side of the SisyphusCore in turn gains access to the `moduleGuis` via the singleton `ProxyImpl`. ................................. 19

2.9 An example of a protein details contribution featuring an amino acid sequence viewer. ................................................................. 21

2.10 The Sisyphus modules which offer their own Http Servlets (generated by the CodeGenerator or hand written) can let the SisyphusCore bundle handle their registration with the servlets container. ............... 23
3.1 The Declarative Services component declaration of a Sisyphus module.
Sisyphus modules are delayed components and only get activated when
the SisyphusCore is started. Each Sisyphus module provides a service
implementing the c.e.s.osgi.SisyphusModuleService interface. 25

3.2 At development time the “New Sisyphus Module Project Wizard”
can be used to create a new project in the Eclipse development
environment. The CodeGenerator can create schemas and tables
in the database according to DDL files and generate beans and
GWTRemoteServiceServlets according to given SQL SELECT queries. 27

3.3 A screenshot showing the “New Project Module Wizard” for the
Eclipse IDE. Depending on the user’s choices a new project with a
code skeleton is created. 28

3.4 An excerpt from a generated interface SomeServiceAsync for a fic-
tional query named “Example” and a fictional module named “some”.
For every query which a module provides, a DynaBean class will be
generated named after the query. The fictional query in this example
would have only one parameter of type INTEGER. 30

3.5 The Log4jAppender gathers all logging messages reaching the Log4j
logging system. The JDKLogAppender does the same for the native
Java logging framework and the one from Apache commons. All
messages end up at the ConsoleLogger where they are filtered and
written to the console. 31

3.6 Extraction and compilation of the client side Google Web Toolkit
source code. The extraction executor is run after the SisyphusCore
has been started. If at any time new modules with more GWT
source code to be extracted are registered with the core the state
changes to EXTRACTING again. Upon successful compilation the
SisyphusCore registers all GWTRemoteServiceServlets with the Jetty
servlet container. 34

3.7 An example cross-experiment display. The user can drag and drop
an experiment tree node into the panel and the calculated protein
occurrence overlap is shown. 35

3.8 The iTRAQ-module’s experiment panel contribution. The opened
experiment’s sub-tree is now alphabetically ordered to avoid arbitrary
node reordering upon module installation which could confuse users.
The tooltips for better discoverability are also new. 36

4.1 Startup time of the OSGi base system vs. startup time of Jetty in the
standalone situation before vertical modularization. Each value has
been measured 10 times. 38

4.2 The time a call from the client side to the SimpleSisyphusService
needs until the client registers its return. Each value has been measured
10 times. 39

4.3 The time until the whole Sisyphus page has loaded (i.e. its DOM has
been created by the browser). This includes both calls to the server
side and client side rendering. Each value has been measured 10 times. 39

4.4 The time until a protein list has completely loaded was measured.
5 different protein lists were used, for both systems the same set of lists. 40
Bibliography


