Trinity: The OSGi module development server

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems Group

Master's Thesis

Trinity: The OSGi module development server

by
Jérémie Granat

Due date 20.7.2008

Advisor:
Jan S. Rellermeyer

Systems Group

ETH Zurich
Department of Computer Science
8092 Zurich
Switzerland

Acknowledgement

This thesis is submitted for partial fulfillment of the requirements of the Master of Science degree in Computer Science at the Swiss Federal Institute of Technology (ETH) Zürich. The thesis was done during a six month project from January 21st 2008 to July 20th 2008 in the Information and Communication Systems Research Group led by Prof. Gustavo Alonso at the Department of Computer Science of the Swiss Federal Institute of Technology Zürich.

First of all, I would like to thank Professor Gustavo Alonso for giving me the opportunity to work on this topic. I am grateful for the valuable discussions I had with the members of the Information and Communication Systems Research Group, especially Jan S. Rellermeyer, who spent some of his valuable time to counsel me and answer all the silly questions I could think of with profound patience and knowledge. I would also like to thank my current employer, the Swiss Railway (SBB AG), which allowed me to finish my studies at the ETH Zürich and gave me time, resources and support during the whole period of my Master.

This work is dedicated to my family and friends who have given me strong support during my time at the ETHZ.

Zürich, July 20th, 2008.

Jérémie T. Granat

For every problem, there is a solution that is simple, neat, and wrong.
(H. L. Mencken)

The underlying complexity of a given problem is constant. It can be hidden, but it does not go away.
(Matt's first law of software complexity)

Abstract

This thesis presents the Trinity Server, a Server supporting the life cycle of dynamic application from creation to deployment, allowing developers to predict the behavior of OSGi applications in different configurations. Furthermore, it enables them to run tests from the smallest unit at the method level to high level tests like load testing in live OSGi environments which can be fully customized by the developers.

The OSGi specification is a dynamic module system for Java. It provides services for service oriented computing through the uses of bundles. Bundles are a packaging and delivery format for components or service implementations. A specific bundle may have dependencies on arbitrary packages and/or on arbitrary services which are provided by other bundles. The OSGi specification mandates the declaration of package dependencies, but there is no such mechanism for declaring service dependencies. The developer needs a method to know in advance if his application depending on specific bundles will work with other version or other implementation. Users can install the application in environments the developer has no knowledge about. The risk to run into complex integration and configuration problems when different versions are available is high. In order to reduce this risk, every combinatorial possibility has to be tested. The developer has to verify that the application is always working within its expected behavior. However, there are currently no tools available to test the variants.

This master thesis remedies this situation by describing the Trinity Server, a bundle management server which maintains all possible configurations of its bundles, continuously performs unit and integration tests on them, and allows the deployment and publishing of the successfully tested configurations.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Component-Based Software Engineering (CBSE) and Service-Oriented Architectures (SOA) are prominent software architecture approaches nowadays[41]. Two popular specifications reflecting those architectures are Java EE[64], formally known as J2EE, and OSGi[52, 55]. Those specifications cover, at different levels, a set of important features that characterize components and services. Conceptual parallels can be made between them to better understand the problems programmers have when they implement applications using those architectures.

1.1.1 Application Lifecycle

The traditional life cycle of a project is composed out of 4 phases: Initiation, Planning, Execution and Closure.[42]

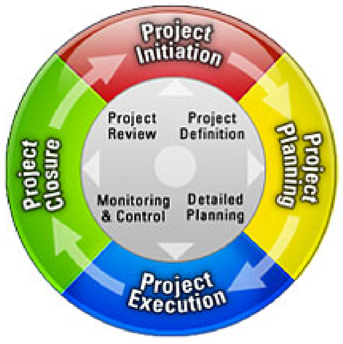


Figure 1.1: Typical Project Lifecycle[42]

- *The Initiation Phase* is typically needed for the development of the different business cases: Creating the project team, getting the necessary financial support, defining the project lead etc.

- *The Planning Phase* consist of creating a project plan, finding and assigning the resources, creating a quality, risk and acceptance plan, and defining the different milestones.
- *The Execution Phase* involves the building and testing of the application as well as the monitoring of the different plans.
- *The Closure* is the delivery of the product and the review of the project.

An application is planned, build and run on the target systems. The number of cycles changed depending on the management model used (Waterfall[59], Spiral[6] or Iterative[23]). However, the traditional lifetime of an application is always the same from requirements, to design and implementation, to deployment, to operation.

Open World System

The dynamic nature of OSGi[76] based systems as well as service-oriented compositions requires a re-thinking of the whole lifetime of an application. The OSGi framework[47] is a sophisticated solution for developing and managing modular software. The Framework itself can be seen as an open world system[37] with the modules being open to context changes from other dynamically loaded modules while the application is executing. This invalidates the traditional solution software designers have adopted to create new application versions. The changes they design, implement and test can have results which were never anticipated. An OSGi Application designer must take into account that the program under development is going to systems that could possibly interfere with his application. However, the overall functionality and quality of the application will still be under the responsibility of the developer. This high degree of dynamism puts a indisputable emphasis on the need to validate the correctness of the application — in as many environments as possible.

Integrated Tool Environment

There are existing tools to develop OSGi bundles[71] and upcoming approaches like p2[73] to deploy these bundles to client machines. There are also tools assisting in finding the correlations between tests and lines of code covered by the tests.[78]. However, the current tools do not yet support the testing of OSGi applications at runtime. There are also no tools supporting the whole software life cycle of OSGi applications, which typically consists of iterations of development, test, and deployment to the end users. Moreover, the different tools are stand-alone applications and do not necessary communicate with each other, which can lead to integration problems or inconsistencies in the deployed applications.

1.1.2 Code Evolution

Parallels between SOA and OSGi Code Evolution

The use of regression tests[83, 19, 3] can reduce the integration problems with version changes[44] but most of the time, components grow or change due to new requirements and the interfaces grow with them. We can distinguish three types of changes in such a component[56]:

- *Implementation change only (patches):* As the implementation is isolated from the user, this change will not have any effect on the composition provided the regression tests are successfully passed.
- *Compatible interface change:* The compatibility for Java interfaces is more complex than for Web Services / XML and can be found in [65]. If the interfaces are compatible, it is the responsibility of the importing bundle to choose the correct version if they want to use the new functionality.
- *Incompatible interface changes:* In this case, the importing client must make sure to choose the right version. Old client could have runtime problems if the Manifest does not exclude the incorrect version.

A tool for systematic testing of the compatibility in component evolution is currently non-existent.

Code Isolation

Software developers are not just producing new software. Studies have shown that between 50%[67] and 67%[61] of the cost of software is in changes made to the software after the project is deemed complete. Of those 67%, about half is in the fixing of bugs. The other half is either improving the working code, or adding new features to the already existing software.[39]

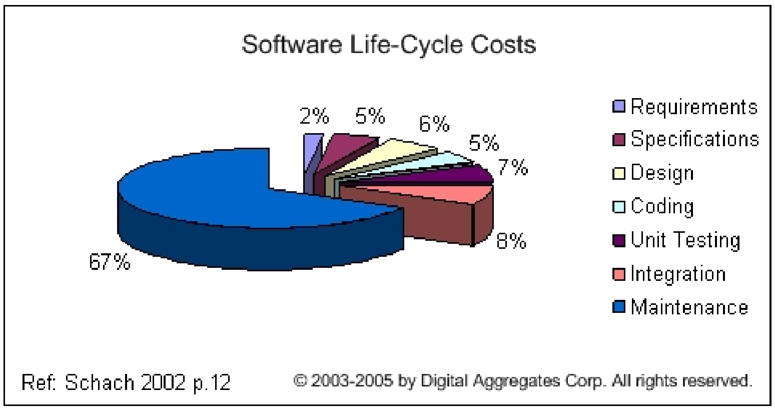


Figure 1.2: Software life cycle costs[28]

Loosely coupled systems like OSGi or Web Services allow breaking the system down into independent modules (the services) which interact with each other. In theory, services can be developed in isolation and integrated at a later point in time, provided that the interfaces (the “contracts” between the components) are defined and “used as they were intended to be”. Interfaces are defined with a concrete idea of their interaction in mind but will be used by others for completely different purposes. In OSGi, the framework facilitates the complete isolation of the service implementation in bundles, sharing only what is needed to communicate and to use methods of other components. The problem of this approach is that the usual code evolution mechanism fails. This is a general problem encountered in service oriented systems[40].

Versioning and Impact Problems

When developers modify service-driven applications by altering the service interfaces, two problems arise: First, the new versions of the services are no longer synchronized with the older versions. Second, the lack of a robust, scientifically substantiated mechanism for identifying the places in the code where objects are defined and used forces developers into a manual maintenance exercises. This is when versioning conflict between different bundle releases appears. As more versions appear over time, there is a very serious risk of conflicting modifications and inconsistency.

1.1.3 Framework Independence

The OSGi platform and the JEE platform are specifications which defines the container for the applications to run in. Both of them describe the format of the components that can be installed in the environment as well as the semantics for the definition of the interfaces. They also describe the basic services the platform should implement for the life-cycle behavior of the components, their security, and how they interact with each other.
The different implementations are based on those specifications, but at the same time, offers a multitude of services, which, when used by the developers renders the application framework dependent. This can lead to differences in the expected behavior of the application or to errors and crashes if it is installed on a different framework or version thereof than the one it has been tested on.

1.1.4 Testing Dynamic Systems

The current testing tools do not work well with dynamic[14] or service oriented[84] systems. Applications implementing those architectures are often seen as a grouping of more or less loosely coupled components interacting with each other, each implementing a small part of the whole. Those components exhibit a very small part of their behavior in static ways and rely most of the time on the runtime environment and the underlying framework for meaningful interaction. This is the case for technologies such as JEE and OSGi.

Mock Objects

The biggest problem with testing SOA/OSGi code, beside the correctness at the method level, is at the bundle boundary. A lot of problems occur when a method in a bundle calls another method on an object defined in a different bundle. This is the case for any nontrivial SOA/OSGi system. The use of Mock Objects[58] for simulating the behavior of the other components enables the programmer to concentrate the testing effort on the component under development. However, a mock object cannot fully replace the real environment. Furthermore, mocks can be time consuming to write and to maintain. Often, to effectively or non-trivially test a service, dynamic behavior from the mock implementation is required. E.g. the mock object cannot just send back a simple value but has to implement some application logic and react accordingly. A developer can get distracted from the original goals of the project by building and maintaining complex mock/simulator systems.

Containerless Testing

Containerless tests are tests executed in an environment different from the one the component under test were designed to run in. The most obvious example is the use of JUnit Test[21] on the console or in an Eclipse environment to test an OSGi module or a dynamic web component. Containerless tests are sometimes not trivial to set up because all project effort is typically driven toward getting the container-based code to function correctly. Basically, the programmer has to simulate the framework and the other components (using mocks) in a predefined configuration with very specific start parameters[1]. Even if the tests are useful to check the correct implementation of some functionality, the results are not really meaningful concerning their correctness as an active system or its stability when other components are installed in the same environment.

Testing with different versions

Code evolves over time. In a conventional application, the use of regression test[19, 3, 44, 29], unit test, and other controlling features[25, 57] helps projects in holding the changes under control. The problem of open world systems is that a programmer or a development team cannot be sure that the version of the used services will be the same on every platform on which the program is eventually deployed. This is even more complex since services may or may not have been created by the same individual. Different teams may have created alternative implementations of a given service. Testing every variant [11, 7, 82] is an exponential endeavor and can currently only be done manually. Using formal method to infer the correctness of a program does not help since every variant still has to be inferred[9, 22].

1.2 Contributions

The main purpose of Trinity is to address these points. The Trinity server introduces the notion of *managed bundles.* Managed bundles are OSGi modules under the control of the Trinity server.

1.2.1 Tracking the Application Lifecycle

The Trinity System uses a truly continuous integration model that first analyzes the bundles to determine its dependencies and capabilities. They are then assembled in such a way as to fulfill the mandatory requirements and tested with the existing unit tests in as many different valid configurations as possible to increase the coverage of the unit tests and achieve generality. Through the testing of the different configurations, a dependency matrix can be assembled which shows the bundles interoperating correctly and pointing the programmer to the problems found in the incompatible settings.

Products in use which experience trouble with the bundles of other vendors can use Trinity by adding those bundles as *binary managed bundles* which will automatically create the appropriate configurations and have the tests run.

1.2.2 Testing Code Evolution

The Trinity System is version-aware and creates a set of configurations for every version of a bundle added to the system taking the version constraints of the other bundles into account. It will also test the resulting configuration to detect constraints defined in the manifest of bundles which have not been narrowed enough. This ensures that versioning conflicts between bundles are caught quickly and can be remedied at an early stage. It also allows the programmers to narrow the version constraints, thereby improving the resolving process by excluding all incompatible versions.

The evolution of the code can also be tested explicitly by testing the application using the update mechanism of the child OSGi instance. Thus ensuring that the new bundle version will work with the application without restarting it.

1.2.3 Testing for Framework Independence

The Trinity System provides the user with a test bed implementation. The user can customize and extend it for his own needs and thus explicitly test for framework independence by implementing a test bed for every framework vendors and version he wants the application to be compatible with. The test bed implementation is not limited to the framework, and could, in theory, encompass the creation of a complete environment using virtual machines[81] and test the application on different OS, Java versions, framework versions and so on.

1.2.4 Alternative Configuration Testing

The Trinity System provides mechanism to automatically generate the different possible configurations and tests them against user defined and generated test cases. Thereby, the administrator is able to catch integration problem or version incompatibilities at a much earlier stage. The tests and the bundles are installed on a new process, the test bed, where they are automatically executed. This setup allows expected and unexpected exceptions to be thrown without affecting the server. The tests can then still be evaluated, even if an error crashed the whole system.

The user-defined test cases can be implemented as conventional JUnit[13] test cases. Additionally, they can implement Trinity specific methods to get access to the BundleContext and the Trinity configuration, which will be injected at runtime. The test cases themselves are packed into test bundles by the Trinity server with all the necessary dependency headers and activation mechanism generated to allow the correct and automatic running of the tests.

1.3 Outline of the thesis

The *first Chapter* described the different problems encountered when writing component based applications and dynamic systems. It also presented how the Trinity System helped in solving these problems. *Chapter 2* introduces the relevant background information this thesis is built upon and the other published solutions found handling the same problems. It shows the different technologies used in the implementation of the system. It also gives a brief overview of the overall methodologies the system can be used in. *Chapter 3* describes a bird's eye perspective of the system and the role of the different Trinity Servers. The *fourth chapter* explains the Build Server and the different steps a bundle has to go through to be added into the system. The Test Server is described in *chapter 5* in more detail and explains how the bundles are tested and the result are propagated and saved. The tests and benchmarks done using the Trinity System prototype implementation can be found in *chapter 6*. *Chapter 7* provides an outlook on the work that will be done in the future and the different concept which needs more attentions, in particular the implementation of the Deploy Server as a possible extension of the *p2* System. Finally, *Chapter 8* presents the conclusions of this thesis. This is followed by *the Appendix* with the different properties used by the Trinity System and the public interfaces to extend its functionality. *A glossary* and *the bibliography* conclude the thesis.

Chapter 2

Background

This chapter describes the methodologies and technologies Trinity is based on. It also describes the different product alternatives already available. Some research has been done on the potential cause of the problems of code evolution and versioning in SOA[26, 4, 38, 56], but very few conjectures have been made about this subject on OSGi.

2.1 Related Work

2.1.1 SAVVY

A method to facilitate open world programming is SAVVY (Service Analysis, Verification and Validation methodologY), a methodology for lifelong validation of service compositions[10]. SAVVY is composed of five steps consisting of

- The definition of the desired service.
- The addition of the different requirements with the functional and non-functional attributes.
- The analysis and verification of the composition
- The deployment of the service with monitoring tools attached to it to check the validation.
- The monitoring of the running service, with a feedback loop for contract violations back to the correct authority.

SAVVY is still in a very preliminary phase of its development and is only a high-level, long-term vision. Furthermore SAVVY focus on Web Services as an example of open world system like other works on the subject of monitoring [16, 36]. The specific aspects of OSGi, a representative of an open world system, are not taken into account. Figure 2.1 on page 10 shows the proposed five step development process.

2.1.2 OSGi Test Tool

The OSGi test harness[35] was developed to test OSGi framework component in a runtime environment. It was designed before JUnit was around. The main

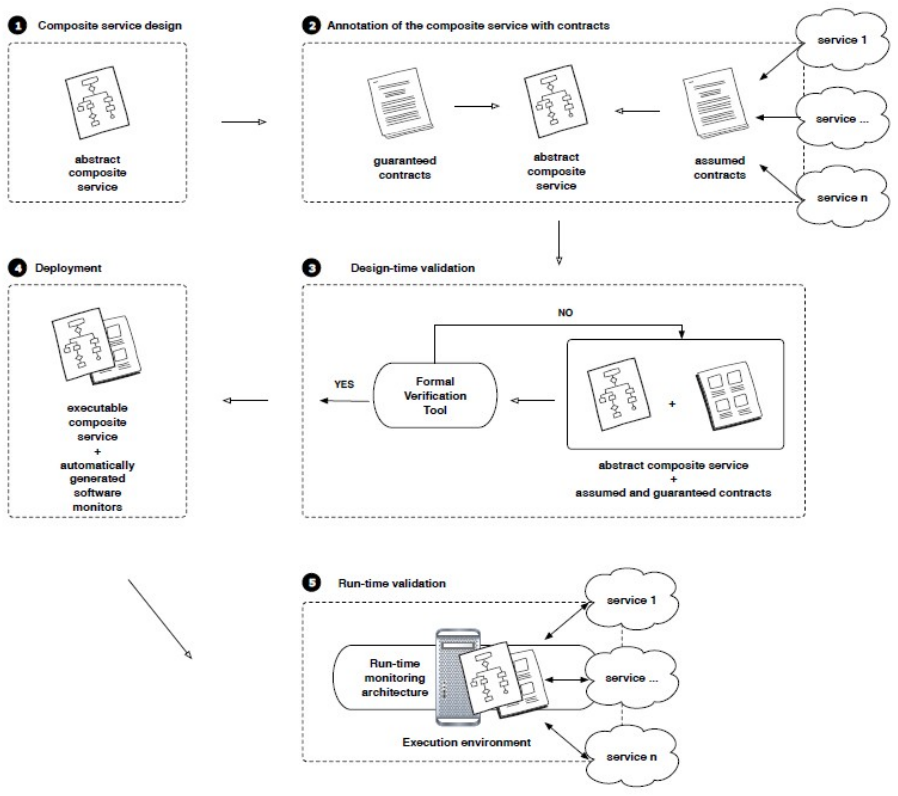
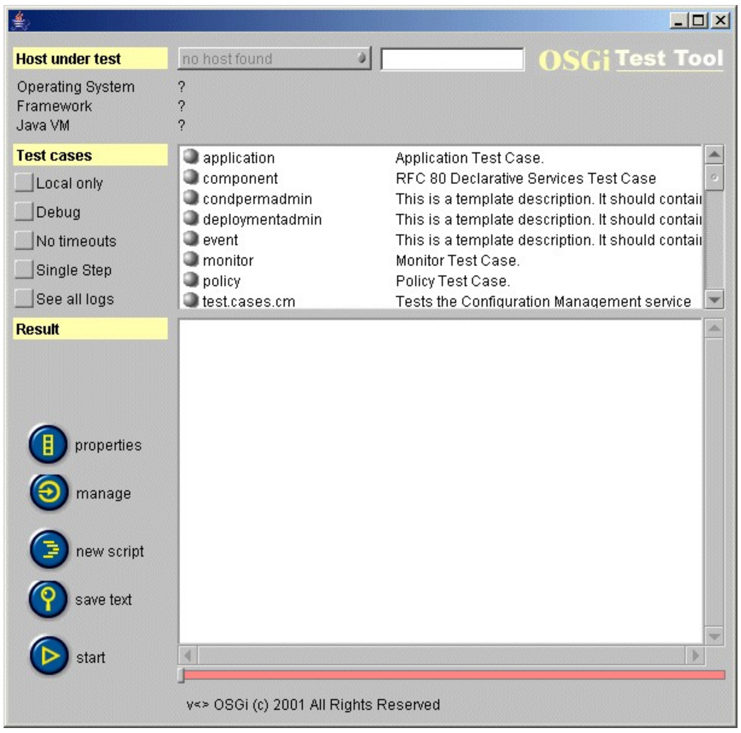

1 Composite service design
abstract composite service
2 Annotation of the composite service with contracts
guaranteed contracts
abstract composite service
assumed contracts
service 1
service ...
service n
3 Design-time validation
NO
YES
Formal Verification Tool
abstract composite service + assumed and guaranteed contracts
4 Deployment
executable composite service + automatically generated software monitors
5 Run-time validation
Run-time monitoring architecture
Execution environment
service 1
service ...
service n

Figure 2.1: SAVVY-aware development process.[10]

idea of the OSGi test harness is to have the test cases send their output to the console, which is then compared to a reference output. The differences are flagged as errors for the tester. As JUnit became increasingly popular, it was decided to adopt the structure and ideas of JUnit in the test harness.

Nowadays test cases created with the OSGi test harness are virtually indistinguishable from the JUnit API. Unfortunately, JUnit itself could not be adopted because of the severe constraints on the target environment. OSGi had to work in embedded systems and JUnit, as is, is not well suited for the task. The original test harness, which was developed in 1999, already had an architecture where the UI was running in another process so that the requirements on the target could be kept to an absolute minimum.
The test harness has the possibility to run the test case in two different processes. Each test case can run partly in the Director (the GUI host) and partly in the target. This architecture significantly reduces the need for setup instructions before the test can be run, a major headache with larger test suites.

At first sight, it might appear that bundle functionality is the only focus of the unit test. However, in OSGi applications, bundle management turned out to be as important to the testing process as the function tests were in more con-



Host under test: no host found

Operating System	?
Framework	?
Java VM	?

Local only
Debug
No timeouts
Single Step
See all logs

application	Application Test Case.
component	RFC 80 Declarative Services Test Case
condpermadmin	This is a template description. It should contai
deploymentadmin	This is a template description. It should contai
event	This is a template description. It should contai
monitor	Monitor Test Case.
policy	Policy Test Case.
test.cases.cm	Tests the Configuration Management service

v<> OSGi (c) 2001 All Rights Reserved

Figure 2.2: Screenshot of the OSGi Test Tool[35]

ventional applications and special support for the management of the bundles had to be added. The start of a bundle or the update of a bundle with a newer version can also be tested with the test harness.

The test suites can be run from a rather nice looking GUI (see figure 2.2) but can also be run from a script or command line.

2.1.3 Patent: Method of Testing Open Services Gateway

The "Method of Testing Open Services Gateway" patent[60] defines a method to effectively test whether a service bundle operates by loading it in a configuration of predefined bundles and testing it with predefined test methods. The OSGi service platform test tool of this patent includes *a framework* in which test cases are installed, *an adapter* enabling the framework to function in a test execution environment, *a director bundle* providing a UI of the test tool, *services* that manage the test and OSGi services, and *an agent bundle* relying message movement with the test case bundle to a framework of a test host.
The OSGi service platform test tool, as shown in figure 2.3 on page 12, is placed in a test manager host. The manager is composed of an OSGI framework and an adapter bundle. A system bundle, a director bundle and other services are then loaded by the OSGi Framework to perform the tests. The Java virtual machine is an external component and provides an environment for the whole Framework.

The method of testing an OSGi service in this system is basically defined into 3 steps:

- Setting up the whole test environment (director, agent and service bundle).
- Installing the different test cases.
- Running the Tests and getting the output.

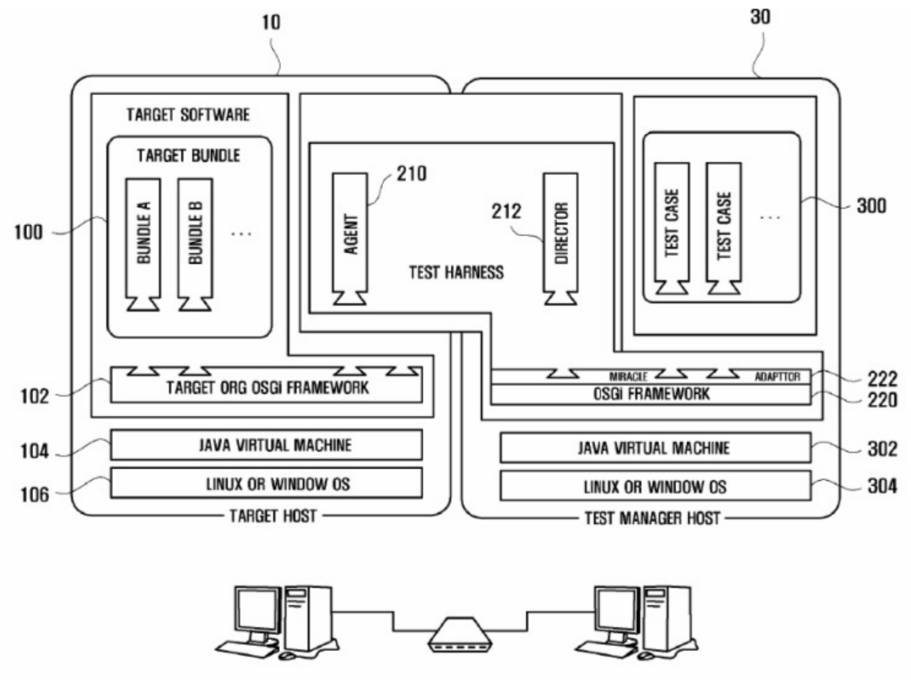


Figure 2.3: Main architecture of the OSGi test server[60]

The patent is dated 2005 and there are currently no public implementation available. The test server as described in the paper does not make any uses of external test frameworks and describes a proprietary system with no possibility of extending it for user specific purposes.

2.1.4 Spring Dynamic Module

The Spring Framework[63] is the leading full-stack Java/JEE application framework. The main purpose of Spring is to provide an effective way of managing the business objects of an application. Its intend is to facilitate the use of J2EE and provides a consistent configuration handling throughout applications. The Spring Framework makes unit testing of application easier by allowing the user to build application using Plain Old Java Objects (POJOs) instead of EJBs or OSGI Bundles. It has recently published its first version (1.0.2) of the Spring Dynamic Modules[62]. The Spring Dynamic Modules for OSGi enables developers to write Spring applications that run in an OSGi framework. OSGi enabled

Spring applications automatically acquire the benefit of the OSGi framework and have a better separation of the different modules, the ability to dynamically add, remove and update modules at runtime, deploy multiple versions of a module in parallel as well as a dynamic service model.

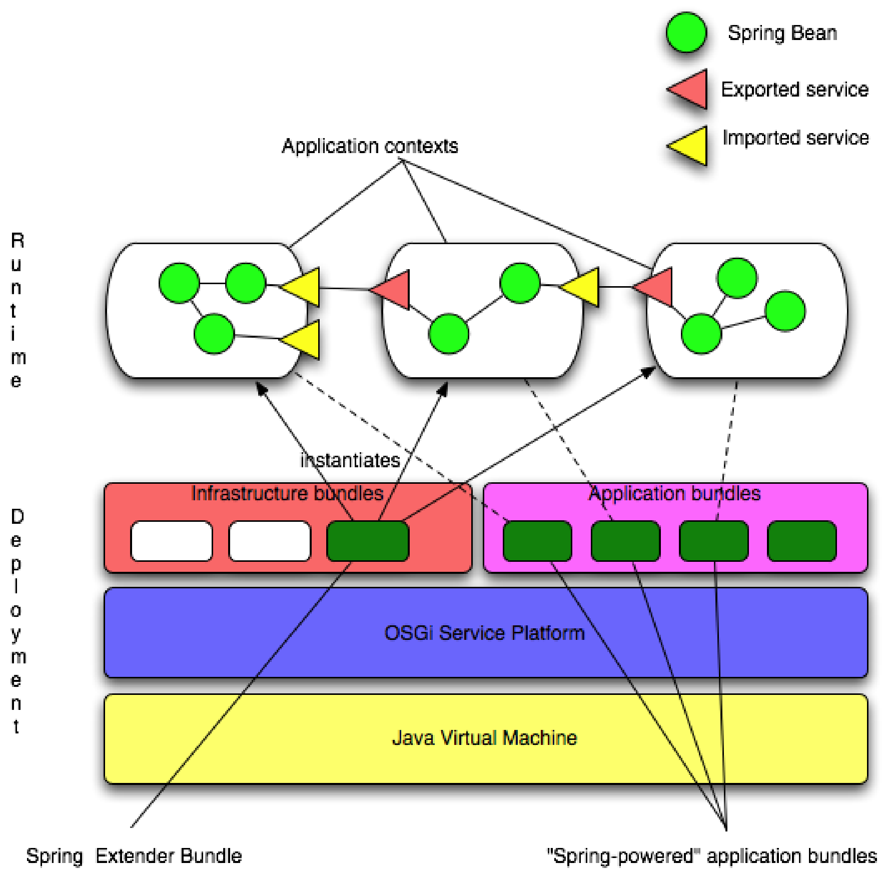


Figure 2.4: Example of an OSGi Spring Application[63]

Spring DM provides the following advantages[51]:

- Spring facilitates the splitting of application logic into modules.
- Several versions of a module can be deployed at the same time.
- Modules are able to automatically find and use services which other modules installed in the system provide.
- Modules can be dynamically deployed, updated and removed from the running system.
- The Spring Framework is responsible for the instantiation, configuration, composition and decoration of components within and across modules.

In Spring DM, the spring-based application logic is packed as bundles. A bundle contains one Spring application context and a Spring-based application may consist of multiple bundles. The hope of Spring DM is to allow the programmer to build on OSGi, but deploy in Tomcat, WebSphere, WebLogic or any other application server. Basically, it gives an "OSGi powered Web Application" view[2].
Integration testing with Spring-OSGi can be done by extending a predefined class to automatically create an OSGi bundle containing the tests on the fly. This class handles starting an OSGi container, installs all bundles and runs tests inside OSGi. Currently, it supports Equinox, Felix and Knopflerfish. There is also a maven property to simply switch containers with profiles (i.e. -Pfelix). To interact with OSGi, Spring DM provides a number of conveniences like the ability to automatically start other bundles after a specific one has been started, add default property values or create bundles on the fly and add them to the OSGi installation. In addition, it offers the possibility to create OSGi mock objects derived from a specific Spring DM package to test the bundles created using the Framework.

Here is an example on how to use mock objects in Spring DM:

```
private ServiceReference reference;
private BundleContext bundleContext;
private Object service;

protected void setUp() throws Exception {
  reference = new MockServiceReference();
  bundleContext = new MockBundleContext() {
    public ServiceReference getServiceReference(String clazz) {
      return reference;
    }

    public ServiceReference[] getServiceReferences(String clazz,
                   String filter) throws InvalidSyntaxException {
      return new ServiceReference[] { reference };
    }
    public Object getService(ServiceReference ref) {
      if (reference == ref)
        return service;
      super.getService(ref);
    }
  };
  ...
}
public void testComponent() throws Exception {
  OsgiComponent comp = new OsgiComponent(bundleContext);
  assertSame(reference, comp.getReference());
  assertSame(object, comp.getTarget());
}
```

2.2 Used Technologies and Methodologies

2.2.1 Iterative Development

Iterative development is a cyclical software development meta-process. A meta-process supports the effort of creating flexible process models. The purpose of process models is to document and communicate processes and to enhance the reuse of processes. Iterative development was created in response to the weaknesses of the waterfall model[59]. It is an essential part of different development paradigms like the Rational Unified Process[43], eXtreme Programming[85] and other models related to Agile Software Development[5].

Program Lifecycle

The fundamental idea behind iterative enhancement is to develop a software system incrementally. I. e. allowing the developer to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system. In each iteration, design modifications are made and new functional capabilities are added.
The procedure itself consists of the *Initialization* step, the *Iteration* step, and the *Deployment* step as seen in figure 2.5.

- The initialization step creates a base version of the system. The goal for this initial implementation is to create a product to which the user can react.
- The iteration involves the redesign and implementation of new features and the analysis of the current version of the system. The goal for the design and implementation of any iteration is to be simple, straightforward, and modular, supporting redesign and enhancements. The analysis of an iteration is based upon the tests and user feedback, and the program analysis facilities available. It involves analysis of the structure, modularity, usability, reliability and efficiency.
- The deployment step is the last step in the process and involve the publishing of the finished product and its delivery to the users.[23]

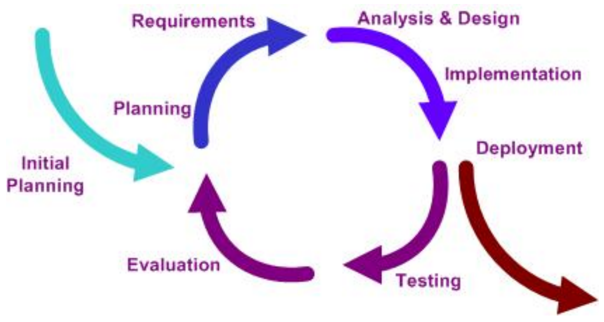


Figure 2.5: The Iterative Development Process[20]

Continuous Integration

Continuous Integration is a software development practice centered on the frequent commitment of the individual work of the programmers into the central source repository, thereby achieving an integration of the individual pieces of code multiple times a day. Normally, each commit is followed by an automatic building process and a test run to detect the integration errors as quickly as possible. The feedback from projects using this method was that it leads to significantly less integration problems and allowed for a more rapid development with less undiscovered errors.[17]

The greatest and most wide ranging benefit of Continuous Integration is reduced risk. When component are fitted together as the last step before deploying into a productive environment, the whole application falls apart. This is known as the big-bang integration problem[8]. In big bang integration testing, individual modules of the programs are not integrated until a milestone has been reach or the individual program parts are finished. In this approach, the program is integrated without any formal integration testing, and then run to ensures that all the components are working properly. There are many disadvantages of the big bang approach:

- Interface contracts have been changed without notifying those using them.
- False assumptions have been made on what or how to return the results.
- It is very difficult to isolate the found defects, as it is very difficult to tell whether the defect is in a component or result in the wrong usage of an interface.

Add to the sudden explosion of problems found in the code the pressure of a looming deadline (or the greater pressure of being behind schedule) and the situation becomes a very explosive one that could have been avoided by a few simple guidelines.
Continuous Integration gives the project members the possibility to know at all times what the project status is, what works, what does not and what are the outstanding errors in the system. The errors are easier to find and to remove as everything has been written with self-testing code in mind. Since everyone is only changing a little bit at a time each time the whole application is built, the search for a bug is at an earlier stage and easier to find. As a result projects with Continuous Integration tend to have dramatically less errors in production.

Of course, the benefits of Continuous Integration are directly related to how good the test suite covers the functionality of the application. It's quite easy to create a test suite that does not really make any difference (by having every method return true, as an example).

2.2.2 The OSGi Service Platform

The Open Services Gateway Initiative (OSGi) was founded by 15 companies and incorporated as a non-profit corporation in May 1999. The initiative has been formed in order to define and promote open specifications for the conception of managed components for applications ranging from small embedded to large-scale client/server systems.[12]

OSGi Bundle

The basic component unit in OSGi is called the bundle. A bundle is a packaging format for the applications that extends the normal Java ARchive (JAR) file. The JAR file, in turn, is fully compatible with the ZIP files format. The main contents of the bundle are the class files and a Manifest file. Class files are the executable part of a bundle. In Java, classes are grouped in packages, which have unique names within the jar file. The bundles provide services and packages to others and make uses of services and packages from other bundles. The services that a bundle provides to other components are registered with the framework which provides a service registry for that purpose. Services themselves are Java interface definitions with an intentional semantic.
A JAR file additionally adds a *Manifest file* containing information about the content of the bundle, the meta data. A Manifest stores information about the contents of the JAR file in headers. Some headers are predefined by the JAR Manifest specification[66] but the total set of headers is freely extendable and the values can be localized. The OSGi Alliance has defined a number of additional Manifest headers to allow the JAR file to be used in an OSGi Service Platform.

OSGi Architecture

Basically, the OSGi Architecture is a dynamic module system for Java and provides a set of specifications and tools allowing the construction of small, reusable components: the bundles. A set of bundles can be packed and deployed as an application.[76]

The OSGi Architecture is composed of different layers: The execution environment depends on the device the framework is running on. The other components completing the framework are the Service Registry, the Bundle Life Cycle Service and the Bundle Class Loader (Modules) as seen in the picture below:

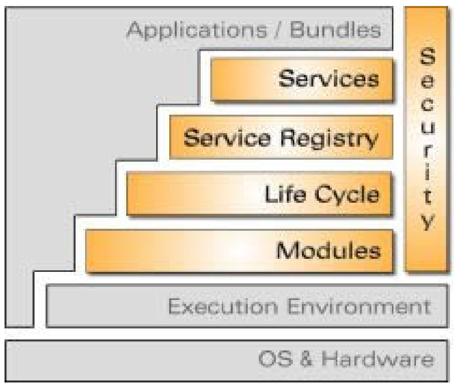


Figure 2.6: The OSGi Service Gateway Architecture[76]

The Bundle Life Cycle Service allows the management of the bundle at runtime. A bundle can be in one of the following states[50]:

- *active* - the bundle is now running
- *installed* - the bundle is installed but not resolved
- *resolved* - the bundle is resolved and is able to be started
- *starting* - the bundle is in the process of starting
- *stopping* - the bundle is in the process of stopping
- *uninstalled* - the bundle is uninstalled and may not be used

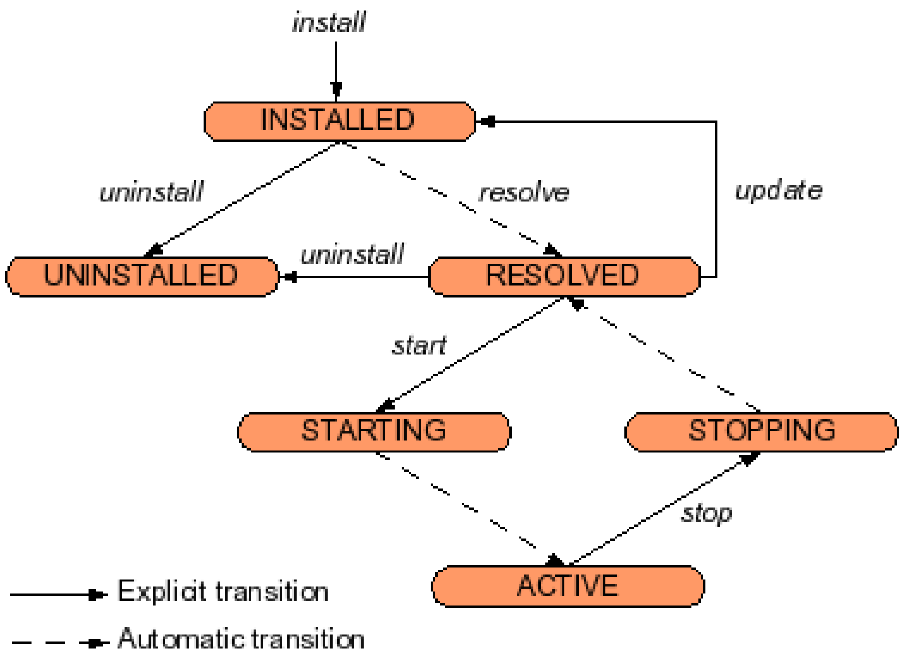


Figure 2.7: The physical bundle life-cycle[50]

The Manifest File

The manifest is a file called MANIFEST.MF in the bundle's JAR file. It contains meta-data describing, among other things:

- The name of the bundle
- The name of the class used as starting point.
- A description of the bundle.
- The required Java version or environment.

- The name of the bundle author.
- The bundle's internal classpath.
- The native libraries required by the bundle.
- The Java packages that the bundle requires or provides.

Additionally the required and/or provided services of a bundle may be specified. It is a structured file with attribute-value pairs (i.e., like Java properties) where each attribute has a defined syntax that must be used to specify its values. For a more detailed overview about the syntax and semantics of the different headers and their attributes see[75].

```
1  Manifest-Version: 1.0
2  Bundle-ManifestVersion: 2
3  Bundle-Name: Trinity Bundle Manager
4  Bundle-SymbolicName: ch.ethz.jgranat.trinity.bm
5  Bundle-Version: 1.0.0
6  Bundle-Activator: ch.ethz.jgranat.trinity.bm.BundleActivator
7  Bundle-ClassPath: .,
      lib/dependency1.jar
8  Import-Package: org.osgi.framework;version="1.3.0"
9  Export-Package: ch.ethz.jgranat.trinity.bm;version="1.1.0"
```

Example of a Manifest File

The Bundle represented by this Manifest File is named "Trinity Bundle Manager" (3). Its namespace is defined by the SymbolicName header (4) and is

``ch.ethz.jgranat.trinity.bm''.

It exports one package (9) and imports the OSGi Framework package (8). It also uses an internal jar file (7). The entry point for the bundle is the class

``ch.ethz.jgranat.trinity.bm.BundleActivator''

where specific methods will be called when the bundle is started and stopped (6).

Sharing packages

The OSGi Framework treats packages as an integral part of the bundle and, if not specified otherwise, keeps them private from the other handlers. However, it is possible to share packages between bundles by specifying them in the import and export headers of the Manifest file. Exporting means that packages (a set of classes) are made available to other bundles. Importing means that packages need to be available (exported) from other bundles in order for the bundle to work. If multiple bundles are exporting the same package (e.g., with a different version), the framework selects an appropriate version for each bundle importing this package.
Even if the framework does not allow more than one package with the same version to be shared at runtime, multiple bundles can still provide the same class with different versions. This is a potential cause for version incompatibility between different bundles if the restrictions on the imports have not been narrowed enough. A related problem is the direct dependency on other bundles.

Sharing Services

The OSGi Platform defines the Service Registry, which dynamically links different bundles together while tracking their state and dependencies. Bundles can then register objects, search for objects given a specific filter or receive notifications when services matching the filter are registered or unregistered. Object registered in this manner are called services and can be registered under different names and with a set of properties.

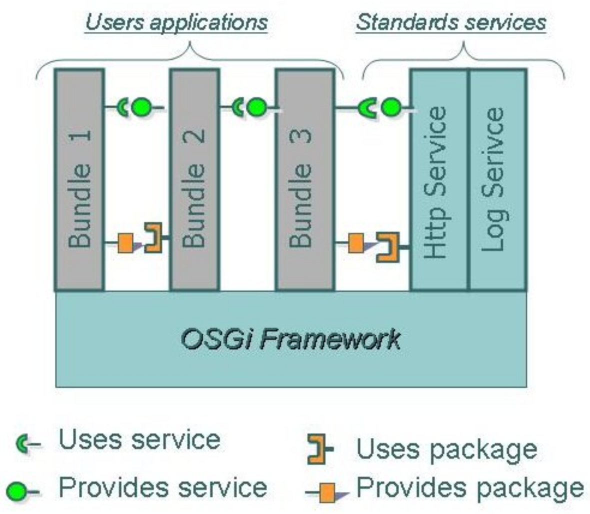


Figure 2.8: OSGi Bundle Dependencies[45]

As with the packages, a bundle can register services to offer some functions to the other bundles in the framework and can specify the services it needs as well. There are, however, some differences:

- Package imports are tight coupling and have to be fulfilled in order for the bundle to be resolved and started. They can be seen as mandatory requirements. Service dependencies are loose couplings and not required for the resolution of the application. They can be seen as optional requirements. However, bundles import services for a reason and not finding them *may* result in the application not working according to its specification.
- The resolving of a package import is done by the framework, which will automatically select the best choice from a list of possible matches. This means that only one version of a package will ever be imported in a specific bundle. The Service Registry allows for more than one service registered under the same name to be retrieved and used, which offers a more complex composition on the service level ($1 \mapsto m$ for services instead of $1 \mapsto 1$ for packages).

```
public class Activator implements BundleActivator {
  public void start(BundleContext context) throws Exception {
    context.registerService(IBundleManager.class.getName(),
                            new DefaultBundleManager(context),
                            new Properties());
  }
  ...
}
```

Example of registering a service in the OSGi Framework

Events

Beside the direct communication between bundles that tight (packages) or loose (services) coupling allows, the OSGi framework has also specified a complete indirect communication layer that is based on the *publish/subscribe* messaging pattern[77]. This pattern decouples sources from their handlers by interposing an event channel between them. The publisher posts events to the channel, which identifies the handlers needed to be notified and then takes care of the notification process. The EventAdmin service provides a place for bundles to publish events, regardless of their destination. It is also used by event handlers to subscribe to specific types of events. Events are published under a topic, together with a number of event properties. Event handlers can specify a filter to control the events they receive on a very fine grained basis.

The topic of an event defines the type of the event. It is fairly granular in order to give handlers the opportunity to register for just the events they are interested in. When a topic is designed, its name should not include any other information, such as the publisher of the event or the data associated with the event, those parts are intended to be stored in the event properties. The topic is intended to serve as a first-level filter for determining which handlers should receive the event. EventAdmin service implementations use the structure of the topic to optimize the dispatching of the events to the handlers.

Event Handlers must be registered as services with the OSGi framework under the object class `org.osgi.service.event.EventHandler`. Event handlers should be registered with a property (constant from the EventConstants class) EVENT_TOPIC. The value being a String[] object that describes which topics the handler is interested in. A wildcard ('*') may be used as the last token of a topic name, for example `ch/ethz/jgranat/*`. This matches any topic that shares the same first tokens. For example, `ch/ethz/jgranat/*` matches `ch/ethz/jgranat/beginMethod`. Event handlers which have not specified the EVENT_TOPIC service property must not receive events.

```
public registerEvent implements BundleActivator, EventHandler {

  public void start(BundleContext context) {
    this.eventAdmin = new ServiceTracker(context,
                          "org.osgi.service.event.EventAdmin",
                          null);
    this.eventAdmin.open();
    Hashtable ht = new Hashtable();
    ht.put(EventConstants.EVENT_TOPIC,
           new String[] { ITestRunner.TEST_TOPIC_CHILD + "/*" });

    context.registerService(EventHandler.class.getName(), this, ht);
  }
  public void handleEvent(Event event ) {
    // Handle the received events...
  }
}
```

Example of an Event Handler

Event Publisher To fire an event, the event source must retrieve the EventAdmin service from the OSGi service registry. Then it creates the event object and calls one of the EventAdmin service's methods to fire the event either synchronously or asynchronously.

```
public class myPub {
  ServiceTracker tracker;

  public myPub(BundleContext context) {
    tracker = new ServiceTracker(context,
                                 EventAdmin.class.getName(),
                                 null);
    tracker.open();
  }

  public void doSomething() {
    EventAdmin ea = (EventAdmin) tracker.getService();
    if ( ea != null )
      ea.sendEvent(new Event("ch/ethz/jgranat", new Hashtable() ));
  }
}
```

Example of an Event Publisher

2.2.3 Distributed OSGi

The OSGi specifications define a small layer where multiple Java based components cooperate together as an application. It is a very powerful system that allow the partial isolation of the different component in a single JVM.[74]. But sometimes, using a single instance is not enough and several frameworks have been created to allow a transparent communication between different OSGi frameworks.

R-OSGi

R-OSGi[30] is a middleware layer on top of OSGi that extends the ordinary OSGi framework to incorporate support for distributed module communication and management.

R-OSGi makes the following contributions:[27]

- *Seamless embedding in OSGi:* From the OSGi framework's point of view, local and remote services are indistinguishable. Existing OSGi applications can be distributed using R-OSGi without modification.
- *Reliability:* The distribution of services does not add new failure patterns to an OSGi application. Developers deal with network-related errors in the same way they deal with errors caused by module interaction.
- *Generality:* The middleware is not tailored to a subset of potential services. Every valid OSGi service is potentially accessible by remote peers.
- *Portability:* The middleware runs Java VM implementations for typical resource-constrained mobile devices, such as PDAs or smartphones. The resource requirements of R-OSGi are by design modest.
- *Adaptivity:* R-OSGi does not impose role assignments (e.g., client or server). The relation between modules is generally symmetric and so is the distributed application generated by R-OSGi.
- *Efficiency:* R-OSGi is fast, its performance is comparable to the (highly optimized) Java 5 RMI implementation, and is two orders of magnitude faster than UPnP.

2.2.4 OSGi Bundle Repository

OSGi specifications are being adopted at an increasing rate. The number of bundles available worldwide is likely in the thousands, if not low ten thousands. Although many of these bundles are proprietary and not suitable for distribution, there are a large number of distributable bundles available. The current situation is that vendors have proprietary bundle repositories. However, in the open source community, the OSGi Bundle Repository allows end users to discover bundles using a command line tool that runs on any OSGi Framework.[49] The OSGi Bundle Repository[54] is an incubator and repository for OSGi bundles. OBR has two main goals:

- Provide a repository of useful and/or didactic bundles that can be easily deployed into existing OSGi frameworks.
- Promote a community effort around bundle creation by increasing the visibility of individual bundles.

2.2.5 Testing

JUnit[32] is a unit testing framework for the Java programming language. Created by Kent Beck and Erich Gamma, JUnit is one of the xUnit families of frameworks that originated with Kent Beck's SUnit. JUnit has spawned its own ecosystem of JUnit extensions.[13]

Design

JUnit is designed around two key design patterns: the *Command pattern* and the *Composite pattern*.
A *TestCase* is a command object. Any class that contains test methods should

subclass the TestCase class. A TestCase can define any number of public testXXX() methods. To check pre and post conditions, the Assert superclass defines the necessary assertXXX() methods in all variations. A very useful method is assertEquals which is present for the primitive types as well as one for objects. The arguments to the assert methods almost always take the same form: "message, expected-value, actual-value" or the simpler version "expected-value, actual-value". As a rule a message should always be supplied because it helps provide additional information when reading failed tests.
TestCase subclasses can use the setUp() and tearDown() methods to initialize and release some initial environment. Each test runs in its own context, calling setUp() before and tearDown() after each test method to ensure there can be no side effects among test runs.
TestCase instances can be composed into *TestSuite* hierarchies that automatically invoke all the testXXX() methods defined in each TestCase instances. A TestSuite is a composite of other tests, either TestCase instances or other TestSuite instances. The composite behavior exhibited by the TestSuite allows you to assemble test suites of test suites of tests, to an arbitrary depth, and run all the tests automatically and uniformly to yield a single pass or fail status.

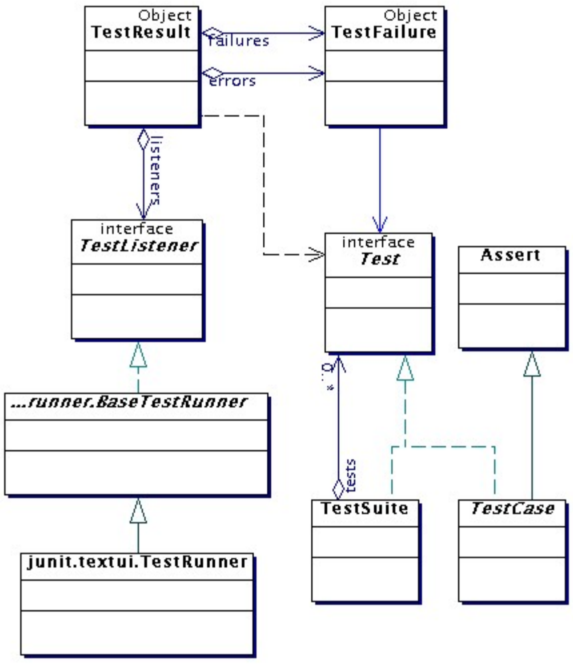


Figure 2.9: High Level Class Diagram of junit.framework

Example

```
public class MoneyTest extends TestCase {
    public void testSimpleAdd() {
        Money m12CHF= new Money(12, "CHF");  // (1)
        Money m14CHF= new Money(14, "CHF");
        Money expected= new Money(26, "CHF");
        Money result= m12CHF.add(m14CHF);    // (2)
        Assert.assertTrue(expected.equals(result));     // (3)
    }
}
```

Example of a JUnit Test Case

A good example on how to use JUnit is written in the JUnit Cookbook[31]. The newest version (Version 4.x) also allows the use of annotation to specify TestCases.

2.2.6 ASM

ASM is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or to dynamically generate classes, directly in binary form[46].
The main problem when manipulating a Java class is that a compiled class is just an array of bytes, which is almost impossible to modify directly. In fact, it is even difficult to just read a compiled class, in order to analyze its tree structure. ASM uses the visitor design pattern without representing the visited tree with objects and can effectively achieve a performant byte code manipulation.
The ASM library provides two APIs for generating and transforming compiled classes: the core API provides an event based representation of classes, while the tree API provides an object based representation.

Chapter 3

The Trinity Application

3.1 Overview of the System

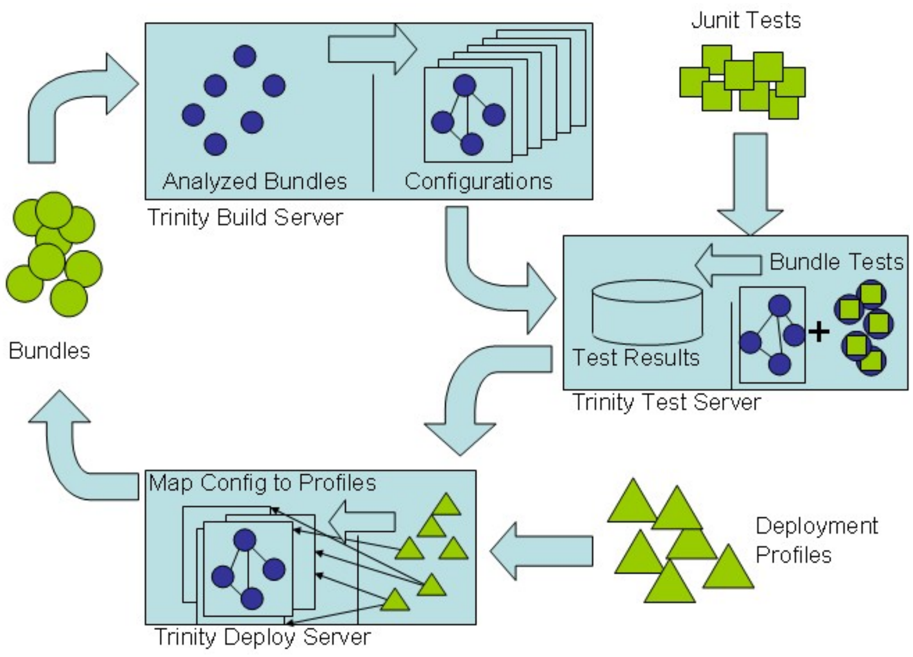


Figure 3.1: The main idea of the Trinity Application

The Trinity Server is the vision and the end system this master thesis started to create. The main idea is to support the entire programmatic view of an iterative program life cycle in an active, open world system. It is composed of the three main phase 'Building', 'Testing' and 'Deploying', coupled with client components in form of Eclipse Plug-ins to help the programmer see, manage and configure the servers, their components and the final configurations to export. The idea behind the Trinity System is to create a loop of the three main phase of building the application, testing it, deploying it and building again using the feedbacks of the test and deployment to create better software.

3.1.1 Build Server

The Build Server is composed of a server construct and a few client plug-ins which are responsible for the management of the different project under the Trinity rule as well as a mean to get feedback on the different analysis reports the server is creating. The client plug-ins were out of scope of the master thesis, remain a part of the final implementation nonetheless.
The server part of the Build Server is responsible for the creation and the static analysis of the managed bundles. The server is made of different component interacting with each other.

- *Bundle Manager*: is responsible for the management of the bundle, the verification of its structure and the extraction of its meta-data.
- *Configuration Manager*: creates and manages all the possible configurations .

3.1.2 Test Server

The Test Server is responsible for the testing of the different configurations. The main idea of the Test Server is to have a system to systematically install and test the different configurations with a series of user-defined tests. Almost everything is customizable, extendable and/or replaceable.

- *Test bundles*: The Test Server takes predefined project tests (Test Directories) and automatically generates test bundles to test the configurations. There is also the possibility to add static or dynamic test generators[33, 80] to create more complex integration tests. However, complex test generators are not in the scope of this master thesis.
- *Child Creation*: The Test Server tests the configurations on a predefined child instance. This child instance is fully configurable and can be any OSGi Framework in any version.
- *Parallel Testing*: The Test Server facilitates the configuration of the number of child instances running in parallel. It speeds up the testing process considerably but may need more configuration work if the tested bundles need unsharable resources like server sockets or exclusive rights to specific files.
- *Test Reporting*: The Test Server sends the results of the tests through the EventAdmin, allowing any listener to report the results in the manner it prefers.

3.1.3 Deploy Server

The Deploy Server is composed of the server and some client plug-ins responsible for the publishing of the user-defined configurations created and tested by the Trinity Test Server. The Deploy Server was out of scope of the master thesis, remain a part of the final System nonetheless. The Deploy Server is responsible for the creation and deployment of profiles compatible with distribution programs and is also responsible for the security and policy aspects (e.g., who can download the bundles under a specific configuration). A more complete specification of the Deploy Server can be found in section 7.3 on page 62

3.1.4 Interaction between the servers

The lines between the different servers are not as defined as it appears at first glance. Some concepts, like the idea of an object representing the meta-data of a bundle or one representing a configuration are found in all three parts and, even if their "creator" server can be found, may be more central to another server then the one it was created in.

Basically, the Build Server is centered on individual bundles and their meta-data, creating configurations in the process. The Test Server is centered around the configurations as its main purpose is to test and evaluate them. The Deploy Server is also centered around configurations but with a greater emphasis on user involvement for the promoting of the different configurations and the creation of rules and deploys policies.

Chapter 4

Trinity Build Server

4.1 Architecture

The Trinity Build Server is used to build and analyze the different projects (managed bundles) under its control and create all the possible configurations that can resolve it. The meta-data extracted from the bundle are only derived from static content analysis and will be refined in the Test server where the dynamic behavior of the bundles will be analyzed. The Build Server is composed of different components that all have their definite role.

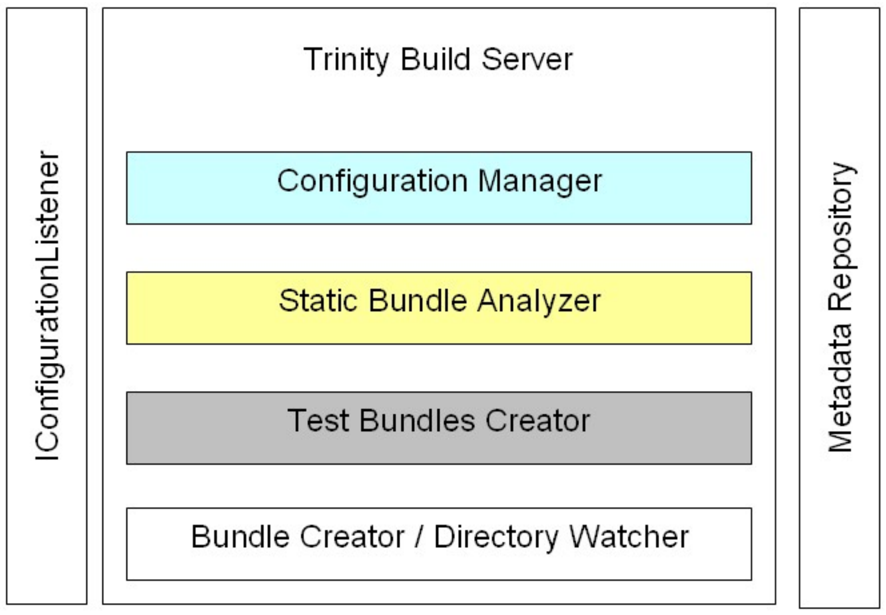


Figure 4.1: Trinity Build Server: Overview

4.2 Staged Test Pipeline (First part)

Bundles goes through a multi-step process before they are deemed “prepared” to be tested and either deployed on the clients or made available to others through public repositories. To ensure that the quality of the bundles stays high, the bundles are routed through a series of tests and analyzed to make sure the standards are correctly implemented. This process is called the Staged Test Pipeline and the first part is shown in the figure below.

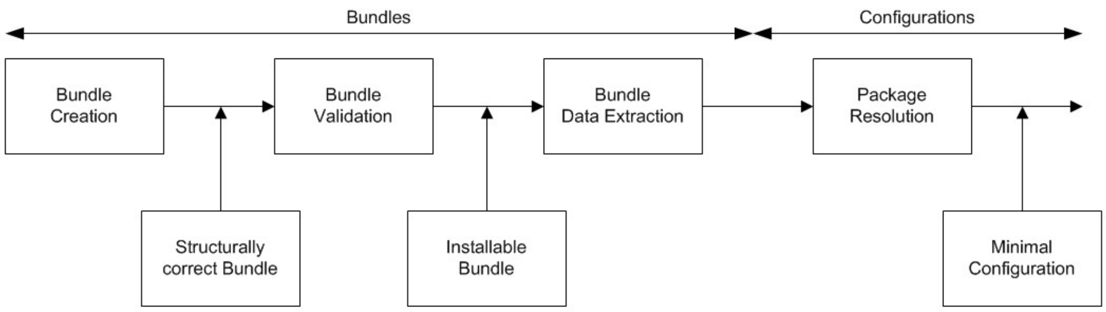


Figure 4.2: The Test pipeline (first part)

4.3 Creation Stage

4.3.1 Bundle Creator

The Bundle Creator, as the name suggests, is used to build all the artifacts in the managed projects and to prepare them for analysis. This can also be done by an external tool like Ant[68] or Maven[70]. After running through the Bundle Creator, a project has been transformed to a bundle in the form of a Jar file and the server first has to validate it for structural integrity. After the first step, the bundle can be considered as *structurally correct.*

4.3.2 Directory Watcher

The Directory Watcher is used as an alternative to the Bundle Creator when external tools are used to create the bundles. The entry point for the different bundles of a project is then one or more directories. The server reacts to bundles being added, removed or updated inside the directories. Depending on what happens, the server starts the analysis of the added or updated bundles or tells the configuration manager to remove all configuration containing the removed bundles.

4.4 Validation Stage

The second step of the validation is the inspection of the content of the bundle and of the used classes through the import packages. If the newly created bundle has import statements inside the code that are neither declared in the manifest files nor resolvable inside the bundle, there is the possibility of a “ClassNotFoundException” when the code with the missing class is run. This is obviously not the expected behavior of a bundle and the Build Manager will mark the bundle as being structurally incorrect.

The third step validates the import packages themselves as specified in the OSGi Core Specification. If both validations succeed, the bundle is actually *installable* on an OSGi framework and can be seen as a valid bundle.

4.4.1 Static Bundle Analyzer

The Static Bundle Analyzer is actually a two-phase-analyzer. It implements both the validation phase and the extraction phase.

Bundle Validation

It is first used on all Bundles created by the Bundle Creator or found by the Directory Watcher to validate it. This is done by analyzing the classes in the bundle and finding out which classes are referenced. The Analyzer then tries to resolve all the references either by looking at the declared import statements found in the Manifest file or by other packages in the bundle itself. After successfully validating the bundle with the first step, the Bundle is deemed installable.

Validation Algorithm

The validation is done using the byte-code analyzer of ASM (For more details on ASM, see section 2.2.6 on page 25) to retrieve all class references found in all the classes of the bundle. The list of all dependencies is build using the results. The analyzer then creates a list of classes defined in the bundle itself and subtracts this list from the dependency list. It also reads the import metadata and subtracts all classes having the same package name assuming that the classes will be found in other bundles. This may not always be the case as there is no way at this point to check if the imported package actually contains all the classes that needs importing. The algorithm is then used on all libraries (jar files) found inside the bundle to further remove internal dependencies. At the end of the analysis, the list of classes should be empty as all dependencies should have been resolved internally or with the import statements.

The Keyword `Ignore-Package` was also implemented by removing all packages declared after this keyword from the package list dependencies. This is not strictly OSGi conform, but is widely used and was implemented for compatibility issues.

```
checkJarFile(jarfile)
  Check if SymbolicName is found (mandatory field)
  Check that no 'java.' import is found
  Check that no ^java.' export is found
  Check that no import is found twice with the same name
  List dependencies.
  for every class file 'i' in jarFile
    List class names <- references to other classes in class 'i'
    add all class names to dependencies

  for every jar file 'i' in jarFile
    checkJarFile(jar file 'i')

  remove all system classes from dependencies
  remove all internal packages from dependencies
  remove all bundle requirements from dependencies

isValid(jarfile)
  return (dependencies is empty)
```

The validation algorithm in pseudo code

Meta Data Extraction

The second part of the analysis is the extraction of the meta-data. All meta-data are extracted statically from the bundle and become an intrinsic part of the bundle.

The meta-data includes:

- Imported Packages
- Exported Packages
- Bundle Dependencies (required bundles)
- Registered Services
- Consumed Services

The meta-data of the bundles are needed in the Bundle Configuration Manager for the next step of the process.

Extraction Algorithm

The extraction of the attributes is done by finding the MANIFEST.MF file and retrieving all the relevant information found there. For more details on how the Manifest file is built, see section 2.2.2 on page 18. As the definition of exported services in the manifest is optional and generally not complete, the extraction of the imported and exported services is done using the byte-code analyzer. The analyzer looks for code parts with the signature of the BundleContext and the method call for registering a service (both for a single service name[1] and for an array of service names[2]). It does the same thing for the imported services by looking at code parts with the signature of the serviceTracker constructor (both

[1] BundleContext.registerService(String name, Object service, Dictionary properties)
[2] BundleContext.registerService(String[] name, Object service, Dictionary properties)

with a filter[3] or with a name[4]) and the signature of the BundleContext and the method call for getting a service reference[5] or references[6].

```
extractInfo(bundle)
  save attributes

  convert imports to requirements
  find imported services
  convert imported services to requirements

  convert exports to capabilities
  find exported services
  convert exported services to capabilities

  remove reexport statement
  remove optional imports
```

The extraction algorithm in pseudo code

4.4.2 The IBundleInfoListener Interface

After the validation stage and the extraction stage, the bundles are able to be used in configurations. At this stage of the pipeline, there is a “Hook” for programmers wanting to extend the functionality of the Trinity System. It is possible to build notification or viewer plug-ins based on the IBundleInfoListener interface. This interface is NOT, as the name might indicate, based on the listener pattern. It is based on the pattern recommended by most OSGi programmers called “White Board Pattern”[48], where the notification is based on the inversion of control principle. Inversion of control can be briefly summarized as the ”don’t call us, we will call you” principle. This means that the IBundleInfoListener does not need to register itself on the object it wants to listen to as might be expected from the “standard” listener pattern but only need to register a service implementing the required interface to get the needed information.

To create new Plug-ins that can be notified by the OSGi System, the programmer only needs to create and register a service implementing the given interface (see section B.1 on page 71 for more details on the interface).

4.5 Minimal Configuration Creation Stage

The next stage tries to resolve all tight couplings defined through package imports and generate a set of minimal configurations that, when installed on an OSGi framework, would make the bundle resolvable. The resolving of all the

[3]new ServiceTracker(BundleContext context, Filter filter, ServiceTrackerCustomizer properties)

[4]new ServiceTracker(BundleContext context, String name, ServiceTrackerCustomizer properties)

[5]BundleContext.getServiceReference(BundleContext bundleContext, String filter)

[6]BundleContext.getServiceReferences(BundleContext bundleContext, String[] classes)

needed bundles is a prerequisite of a functioning installation. The configurations are then *minimal or resolvable configurations*. Those configurations could be used for mock tests with specific mock bundles implementing the optional needed services or for specific bundle test (stand alone tests) if the bundle can be started without any of its optional requirements being fulfilled.

The last step of the configuration creation, done by the Test Server, is to take the minimal configurations and to merge them with one or more configurations until all the optional requirements have been met. The resulting configurations are called *complete configurations* and are later used in the Test Server for integration tests and analysis.

Disregarding the optional requirements, a bundle can be drawn as a quadrangle with the mandatory capabilities on top of it and the mandatory requirements at the bottom. A valid configuration does not leave any mandatory requirements unfulfilled and can thus be drawn as a triangle with the mandatory capabilities of the root bundle at the top.

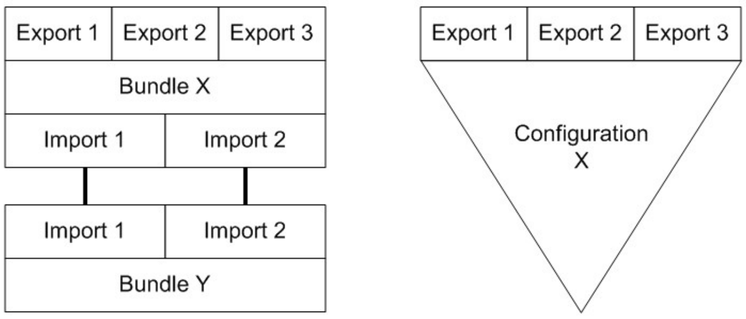


Figure 4.3: Example of a schematic representation of a configuration

4.5.1 Bundle Configuration Manager

The Bundle Configuration Manager is used after the bundles have been validated and analyzed.

Minimal Configuration Creation

The Bundle Analyzer has found out all the dependencies of the bundles (optional and mandatory, used and published) and the manager can now define all the possible configurations a specific bundle can be used in that resolves all mandatory dependencies (import packages). This is done by following the OBR[49] model and interfaces but returning all resolvable configuration instead of returning only the first one.

Building Configurations

Bundle Graph If all loose bindings are removed from the equation, a configuration can be seen as an acyclic graph of bundles connected to each other by the import and export statements in their manifest. A node in a bundle graph is defined by its bundle and an optional number of import and export bindings. One of the main problems of such a structure is the time it takes to test configurations for equality or to calculate the hash number. To prevent wasting time the configurations and nodes can cache their state and only recalculate it if structural changes have been detected.

Configuration Algorithm The configuration described in figure 4.3 on page 36 can be broken down into a root bundle X with 3 capabilities and 2 requirements coupled to a configuration Y containing bundle Y with 2 capabilities and no requirements.

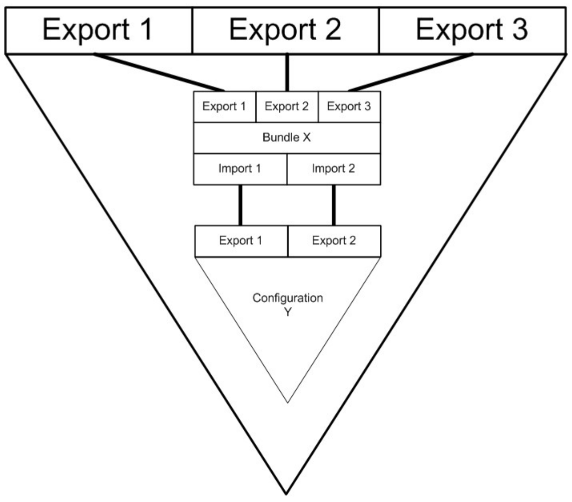


Figure 4.4: Representing a configuration with sub-configurations

It is easy to see that configuration trees are recursive structures. For any given bundle, a configuration becomes the bundle as the root object with 0 to n “sub-configurations” depending on how many imports the root-bundle has. The algorithm to calculate all possible configurations for the bundle is then finding all “sub-configurations” fitting into the bindings of the node. The sub-configurations themselves are in no way different than the “super-configuration” and are composed of a root with a bundle and other “sub-configuration” for each needed binding. The minimal configuration for a core bundle is the bundle itself (as it does not have any dependencies).

```
addBundle(bundle)
  add the bundle to the stack
  check if bundles can be resolved
  resolve all new configurations that have the new bundle as root
```

```
if the resolving created new configurations
  for every new config
    find all bundles that are unresolved and may need the new config
    add the bundles to the stack to be resolved
find all configurations where the new config could be swapped in
remove bundle from stack
resolve next bundle while stack is not empty
remove bundle from stack
```

The Minimal Configuration Creation Algorithm in Pseudo-Code

4.5.2 The IConfigurationListener Interface

The other bundles are notified by the system when new configurations have been created using the "White Board Pattern". It would be very easy to build viewer plug-ins based on the IConfigurationListener interface to show the different configurations generated when a new bundle has been inserted.

To create new Plug-ins that can be notified by the OSGi System, the programmer only needs to create and register a service implementing the given interface (see section B.3 on page 71 for more details on the interface).

4.5.3 The IConfiguration Interface

The collection received by the different ConfigurationListeners contains objects implementing the IConfiguration interface, an abstraction of a configuration. Through this interface, the programmer is able to check the completeness (all requirements are fulfilled) and the validity of the configuration. He can also retrieve information about the requirements and capabilities of the configuration, and get the bundles the configuration is made of (see section B.8 on page 74 for more details on the interface).

4.5.4 Configuration Export

The possibility to print the different configurations into Graphviz-style files has been implemented for visualization and documentation purposes.

Graphviz is an open source graph visualization software. It has both a web and an interactive graphical interfaces, auxiliary tools, libraries, and language bindings. The Graphviz layout programs takes descriptions of graphs in a simple text language, and make diagrams in several useful formats such as images and SVG for web pages, Postscript for inclusion in PDF or other documents.[24]

```
digraph "G" {
  graph [ size="200,200", rotate="90", ratio="compress"];
  node [  label = "\N" ];
  subgraph Configuration_0612 {
    graph [ label = "Configuration 0612"];
// creating the nodes
    "{{ch.ethz.package1|ch.ethz.package4}|Config 0612.bundle22|{}}"
```

```
          [shape=record, color=lightblue];
    "{{ch.ethz.package6}|Config 0612.bundle8|{}}"
          [shape=record, color=lightblue];
...
// creating the links
    "{{ch.ethz.package5}|Config 0612.bundle7|{ch.ethz.package1|ch.ethz.package3}}"
           -> "{{ch.ethz.package1|ch.ethz.package3}|Config 0612.bundle4|{}}"
              [color=blue,label="ch.ethz.package3"]
...
  }
}
```

Example of a configuration in Graphviz text format

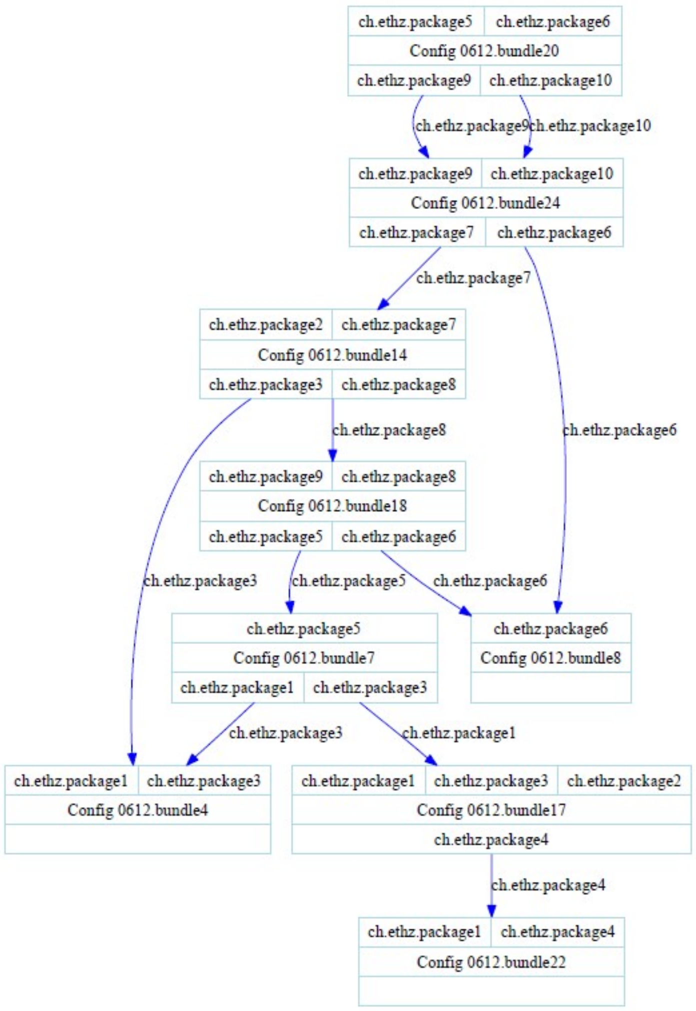


Figure 4.5: Example of a configuration in Graphviz graphic format

Chapter 5

Trinity Test Server

5.1 Staged Test Pipeline (second part)

The second part of the Test Pipeline (figure 5.1) begins where the first part stopped (see figure 4.2 on page 32) and concentrates on the configurations instead of the bundles. The Configurations also go through a multi-step process before they are deemed “prepared” to be deployed (*valid configurations*).

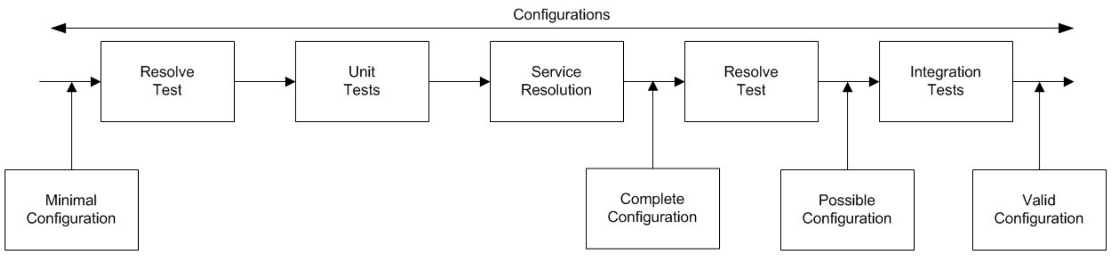


Figure 5.1: The Test Pipeline (second part)

5.1.1 Unit Tests

A minimal configuration is not the same as a complete (or possible) configuration as only a subset of the requirements (the mandatory ones) has been resolved with absolute certainty. Nevertheless, there are some test that can already be performed with the minimal configuration. The most basic one is installing all the bundle in an instance and finding out if they all resolve correctly. Another possibility is to take some JUnit component tests provided by the developer and run them in the child instance. Depending on the result of those tests, the next step (testing a complete configuration based on the minimal one) can be processed or not. As an example, if the minimal configuration does not resolve correctly, even when all it’s mandatory requirements have been fulfilled, then there may be something wrong with the whole configuration and will probably not resolve even when adding the completing bundles.

5.1.2 Integration Tests

The tests which can be run using the complete configurations based on the resolvable configuration provided by the Configuration Manager are more involved than method or component tests. The user can create JUnit Integration test that will be packed and installed with the application and that will run on a configuration that resolves all the requirements.
Integration tests could involve the creation of a Mock Object implementing a specific interface, pack it in a bundle and register is in the child OSGi framework instead of the "normal" application implementation to test if the declared exceptions can be thrown safely. The importing bundle should not crash because the mock threw a known exception. Another possibility could be for the mock object to return different values within the range of the used type.

5.1.3 Higher-Level Tests

Using the Repository Manager (see section 5.5 on page 44), the Trinity System can install "system" bundles on the child framework at runtime and start them. The dynamic test bundles coupled with those from the Repository Manager could be used to test higher level requirements like:

- *Resource conflict.* Most system resources are finite and things such as ports or exclusive rights to directories cannot be shared among specific bundles.
- *Load behavior.* Most systems are poorly stress tested and therefore do not perform well or even within their specification past a given load. Some bundles could be created to specifically test the behavior of the application by producing load.
- *Code coverage.* A code coverage system could very easily be implemented with only very small changes to the Test Orchestrator. The analysis would then allow the programmers to know which part of the code was never called by the application and thus, could hide bugs.

5.2 Test Orchestrator

The Test Orchestrator is the anchor point in testing OSGi components. It takes all the new configurations from the Bundle Configuration Manager as well as all the available TestCases from the Test Manager and tests every configuration against them. For each configuration, the Orchestrator creates a child OSGI instance of a predefined type, upload the bundles of the configuration and those of the TestCases and route the result of the tests to the Test Reporters and the Test Repository.

The child instances are used to allow runtime errors to be caught (or produced) by the TestCases.

```
Get all Tests

For every new Configurations
```

```
Create a child instance
Install all bundles in the child
Test if all bundle resolve

If they resolve:
  get all complete Configuration from the Manager

    For every complete Configuration
      Clean the child (remove all bundles)
      Install the needed bundles
      Run the tests
      Collect the results

Destroy child instance
```

The Algorithm of the Orchestrator in pseudo code

5.3 Test Manager

The Test Manager is responsible for providing the tests for the managed bundles. Using a base directory specified in the properties (which can be overridden through setting the appropriate environment variables), the Test Manager will create a test bundle for each sub-directory found inside the root.

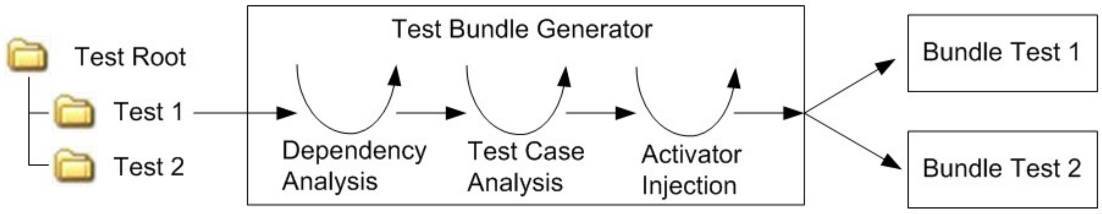


Figure 5.2: Generation of Test Bundles

The Bundles are saved in the Test Repository Manager to improve performance. The bundles will only be generated if the corresponding subdirectory has a newer time stamp than the one used to generate the file.

The interface of the TestManager is very simple (see section B.2 on page 71) and only allows the creation of new tests. The returned value is a collection of *JarWrapper*s. A JarWrapper is, as the name indicates, a wrapper over a bundle and is used by the Orchestrator to access the generated test bundles.

5.3.1 JUnit Bundle Generator

The generated bundles contain a constructed MANIFEST.MF with a predefined “test runner” class as Activator and a dependency import-package list. A property file will also be created containing all the TestCases found coupled with their methods (test***). A mechanism is in place in the child instance to inject the context and the configuration objects into the TestCases to facilitate creating integration test cases. Both object can be used in the TestCase classes by calling the "this.getConfig()" and "this.getContext()" methods defined in the ITest interface.

```
Manifest-Version: 1.0
Bundle-Name: TestBundle-startStop
Bundle-Activator:
      ch.ethz.jgranat.testserver.jarmaker.TestBundleActivator
Bundle-ManifestVersion: 2
Bundle-Description: on-the-fly test bundle
Bundle-SymbolicName: TestBundle-startStop
Import-Package: ch.ethz.jgranat.testserver.core.bcm,
   ch.ethz.jgranat.testserver.core.bm,
   ch.ethz.jgranat.testserver.core.util,
   junit.framework,org.osgi.framework
```

Example of a Manifest file for a generated test bundle.

5.4 Test Repository Manager

The Test Repository Manager is used to store the test bundles the Test Manager generated. The current implementation only saves the bundle in a Map. It means that the Tests are not persisted anywhere and are lost at server restart. However, implementing a new Test Repository Manager with predefined tests or with a connection to a database is very easy. The Test Manager also allows parallel implementations to work and would then save the tests on different mediums.
To implement an alternative Test Repository Manager, the programmer only has to create and register a service implementing the `ITestRepositoryManager` interface (see section B.4 on page 72 for more details on the interface).

5.5 Report Manager

The Report Manager follows a double purpose:

1. *Saving the finished tests.* The current implementation just saves the test suites in-memory in a Map but other repositories could persist the result in a database or on the file system for greater security.

2. *Monitoring the Children.* Additionally to collecting the tests, the Report Manager has two methods that are used to notify it when a new configuration starts and stops being tested. Those method could be used to start (or install) special monitoring bundles on the child that would collect runtime data like memory usage or thread count.

There is currently no way to get the results from the reports directly from the Repository Manager as the bundles responsible for the displaying of the relevant information were not in the scope of the master thesis and have been left for future work.

5.6 Test Child Framework

The Test Child Framework is intended to be used to test the configurations in respect to conformity (using special JUnit integration bundles), stability (using “destructive bundles”) and durability (using a load tester or something similar). It provides a complete sandbox model for the configuration and could be extended or replaced to include not just the already defined OSGi frameworks but a whole complete system with OS and specific “deployment dependencies” like drivers, Java runtime and so on.
The child framework interfaces, the IChildInstance (see section B.6 on page 72 for more details on the interface), is a generic interface to an OSGi framework. It defines most of the methods needed for the management of the bundles as well as method to handle the child (output handling, start and stop, ect). It is declared in the core package.

The Test Orchestrator uses the Factory Pattern to create new child instances through the IChildFactory interface (see section B.5 on page 72 for more details on the interface) to define at runtime the type of child instances that should be created to test the bundles.

5.6.1 Default Child Creator

The current implementation of the child framework creates a new instance of a predefined configuration based on the properties given either in the ConfigProperties.txt file in the core package or through the environment properties. It creates the new instance on the same server (using “localhost” to open the console and R-OSGi ports) in a subdirectory based on the predefined root directory. The whole subdirectory will be deleted as soon as all the tests have been run.

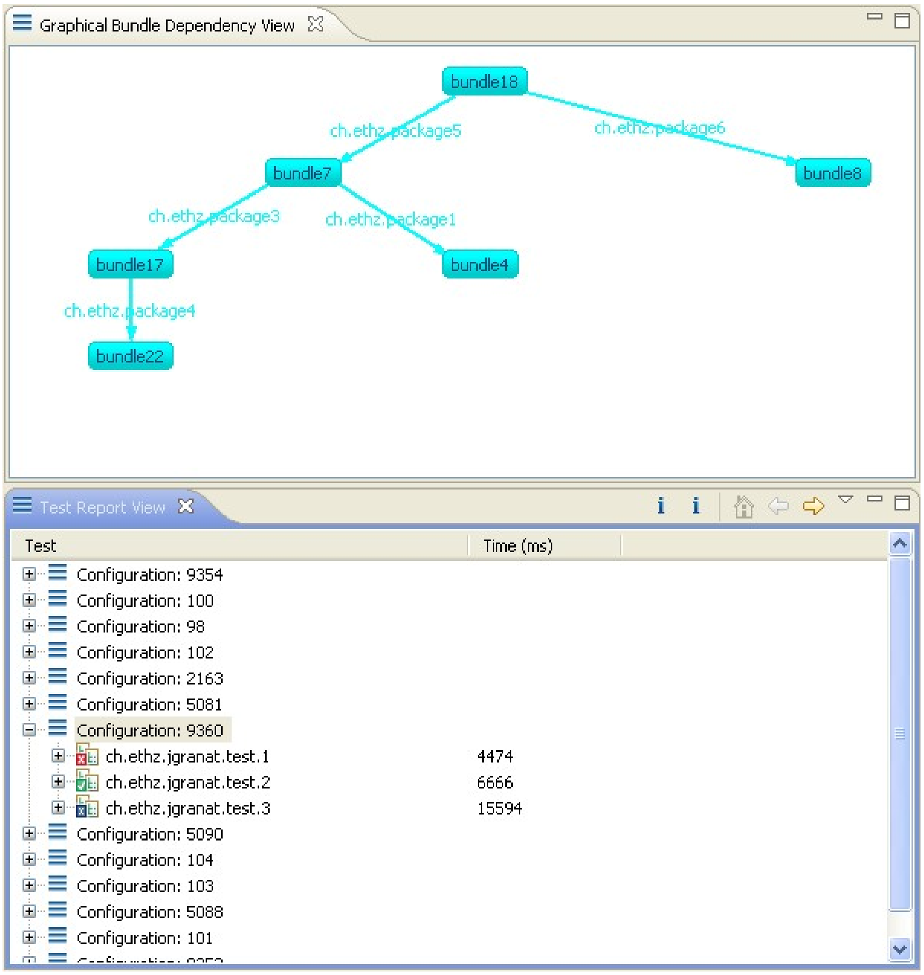
5.7 Test Evaluator

Evaluating the data collected during tests can be very difficult and most of the time, very project dependent. The Trinity Server can only evaluate tests in a most primitive way (test failed/passed) and any higher evaluation algorithm has to be expressed either by visual plug-ins where the programmer can gives feedback on tests which will then be taken into account later on or by programmed evaluation routines in a customized evaluator.

To create a new TestEvaluator, the programmer only has to create and register a service implementing the `ITestEvaluator` interface (see section B.9 on page 74 for more details on the interface).

5.8 Test Reporter

A prototype of a Test Reporter is also included in the Trinity server, showing the different tested configurations and their test times. The whole system is built on events and does not request the focus of the eclipse environment to enable the list to be filled in the background. Of course, another implementation could be created (or one running in parallel) by creating a plug-in listening to event in the form: "`ch/ethz/jgranat/testserver/core/Parent/*`". See section C on page 77 for more information about the generated event and their uses.



Test	Time (ms)
Configuration: 9354	
Configuration: 100	
Configuration: 98	
Configuration: 102	
Configuration: 2163	
Configuration: 5081	
Configuration: 9360	
ch.ethz.jgranat.test.1	4474
ch.ethz.jgranat.test.2	6666
ch.ethz.jgranat.test.3	15594
Configuration: 5090	
Configuration: 104	
Configuration: 103	
Configuration: 5088	
Configuration: 101	

Figure 5.3: Example of a configuration in the Test Reporter

The Test Reporter is composed of two dependent parts.

- *The Test Report View* is a list of configurations containing every test done on them while the Test Reporter has been connected to the Trinity Test Server. Other implementations could query the Test Repository Manager for older tests and configurations. The tests shown in this tree have the same categorization as the JUnit tests.
  - *Success* shows that the test was run successfully and that all assertions were correct (see the JUnit section on page 23 for more information on assertion).

  - *Failure* shows that one or more assertion in the current test was incorrect and thus failed the test.
  - *Error* shows that an unexpected exception was thrown and not caught by the application under test. In contrast to failures, errors should never reflect an incomplete or failed test but should always express a bug in the system.

- *The Graph Bundle Dependency View* presents a graphical view of the configuration selected in the Test View, showing the bundles and how they relate to each others through the import and export of packages and services. There are currently not graphical differences between package and service dependencies and bundle informations are not shown beside the symbolic name of the bundle.

Chapter 6

Test and Benchmarks

6.1 Trinity Build Server

The benchmark tests of the Trinity Build Server were done on an IBM Thinkpad model portable computer. The use of a slightly outdated computer as well as using a portable computer allows for more pronounced differences in the configuration creation time and gives us a “lower boundary” for the time needed to produce any results in a real, productive environment.

In order to evaluate the creation of configuration, it was decided to take “real”, known applications and put them under the management of the Trinity Server.

6.1.1 Apache Felix

Apache Felix[69] is a community effort to implement the OSGi R4 Service Platform, which includes the OSGi framework and standard services, as well as providing and supporting other interesting OSGi-related technologies.

The current test was done using the version 636355 from the source repository. Out of the 80 Jar files that were build using the source export, 73 were actual bundles and were installable in the Trinity Server. The others were Java libraries and were added to the classpath of the configuration manager. The classpath is used to remove dependencies that would have been resolved through other channels and not through the bundle resolution mechanism provided by the OSGi Framework.

Felix Test 1

The first test created 918 configurations out of 73 bundles. However, 3 bundles could not be resolved by the Trinity Server due to missing imports.
The bundles were added into the system by ordering the list of bundles by name and adding one after another with a two second pause in-between. As can be seen in figure 6.1 on page 50, there are two bundles central to Apache Felix that generates a lot of configurations which resolves the majority of the bundles and two smaller ones generating a smaller set. The central bundles were added as Bundle Nr. 33 and Bundle Nr. 67. The fact that 2 to 4 bundles are creating

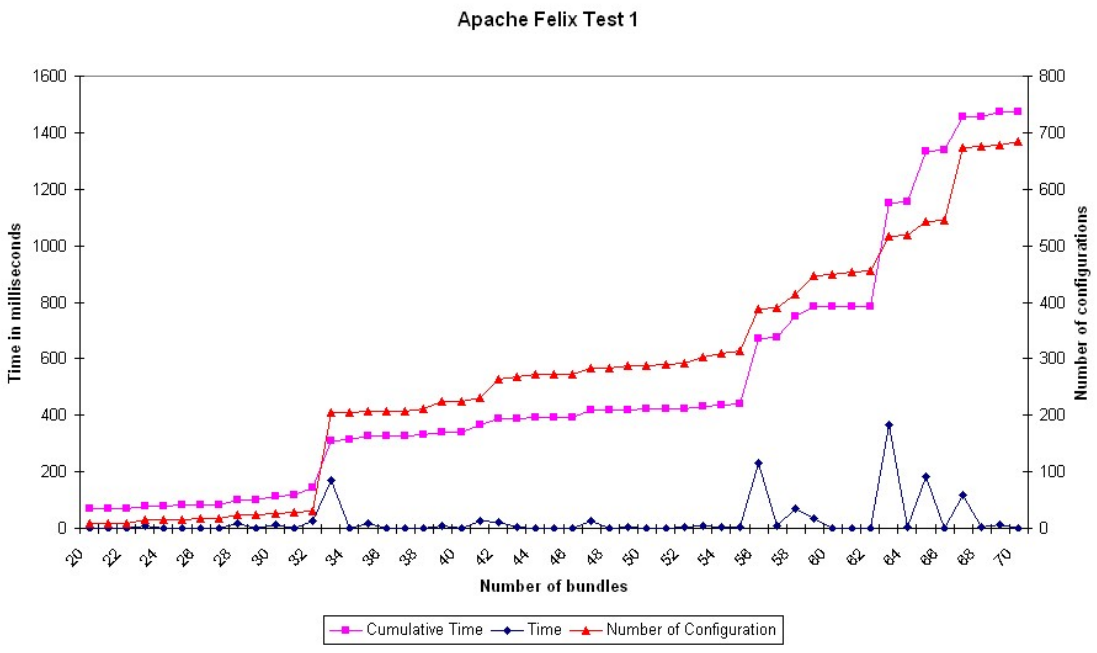


Figure 6.1: Apache Felix Test 1

so much differences points to a star architecture with core bundles as the main body and satellite bundles depending on them. These bundles are central to the correct functioning of the application and a lot of other bundles need some capabilities from them. This means that changes made in these bundles should be tested with even greater care. The risk of incompatibility issues increases with the number of dependents on the changing component

The graph shows the direct dependencies between the number of created configurations and the time needed to add the bundle. The first 20 Bundles have been removed from the graph as the number of generated configurations from the bundles between the 1st and the 25th are not significant and stays at about 25. It also shows that the number of bundles in the system only lengthens the time needed to generate a configuration by very little. If a bundle does not create any configurations, the number of bundles already in the system does not impede the time needed to add it. The time needed to create a configuration, however grows with the number of bundles in the system, as can be seen when comparing the time needed to add bundle Nr. 33 and bundle Nr. 67 and the number of new configuration generated by the addition.

Felix Test 2

The second test also created 918 configurations out of 73 bundles. Again, 3 bundles could not be resolved by the Trinity Server because of missing imports. The second test was done by adding the bundles using a random generator that creates a stream of pseudo-random numbers. The generator uses a 48-bit seed, which is modified using a linear congruential formula[34]. The first 40 Bundles

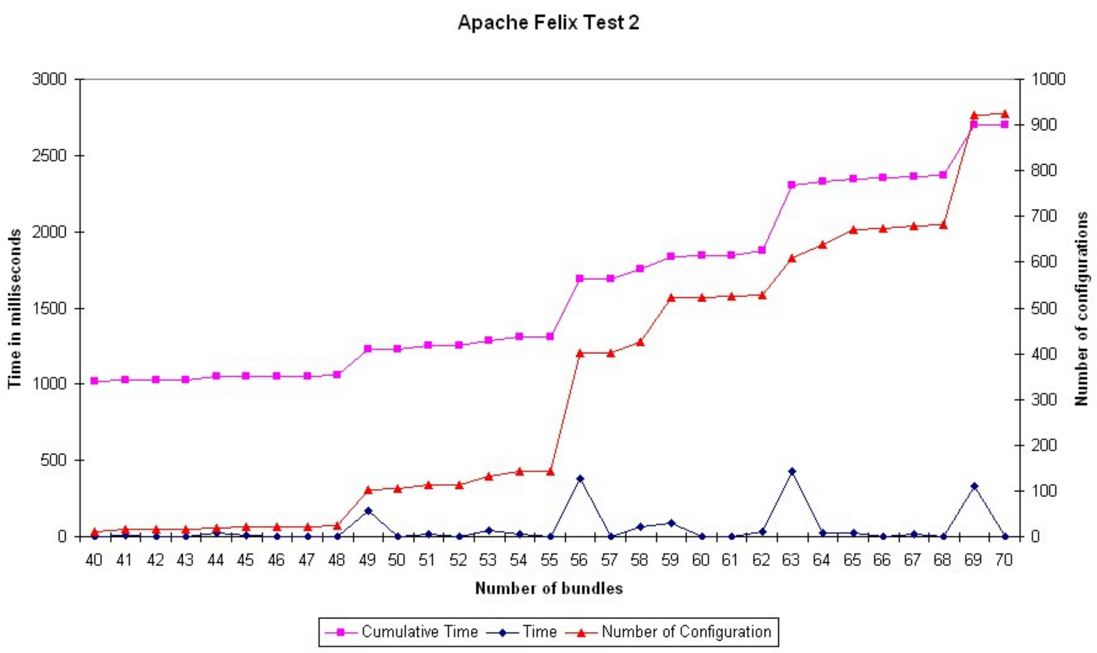


Figure 6.2: Apache Felix Test 2

have been removed from the graph as the number of generated configurations from the bundles between the 1st and the 48th are not significant and stays at about 25.

In this test, only 4 central bundles could be picked out of the graph. Again, two of them seem to be more central than the others and produce significantly more configurations. Those bundles are Nr. 56 and Nr. 69. The graph shows that as soon as one of those bundles is added to the system, configurations are generated almost every time a new bundle is added. The time needed to generate the "smaller steps" is more or less constant.

6.1.2 Newton

Newton[53] is a distributed OSGi framework in which the components can be simple POJOs or wrappers around components based on other models.
Newton creates a framework allowing the installation and removal of code through the network at runtime. Newton also dynamically wires up runtime service dependencies between components and rewires them as service provider components come and go.
Newton is able to install and manage component distributed across a large number of JVMs, continually comparing the deployed composite graph to a specified target state and making adjustments in response to failures and network topology changes.
Newton makes use of OSGi for wiring up composites within a single JVM and Jini technology for tracking and wiring up dependencies between composites in

different JVMs.

For the testing of the Trinity system, the release version 1.2.3 from the web page was downloaded and used. Out of the 176 Jar files that were found in the release, 129 were actual bundles and were deemed installable by the Trinity Server. The others were Java libraries and a subset of 33 were needed and added to the classpath of the configuration manager.

Newton Test 1

The first test created 249 configurations out of 129 bundles. 4 Bundles could not be resolved because of 2 import. Querying the resolver provided an explanation: 2 bundles were exporting a package that the other one needed, thus creating a cycle in the resolving tree (org.cauldron.newton.cds.remote.protocol.jar needs org.cauldron.newton.cds.remote.api.jar and vis versa). This does not necessarily reflect a problem in the Newton application but rather the way the generation of the configuration is implemented. Currently, the configuration generator cannot handle dependency cycles and those two bundles were thus deemed irresolvable.

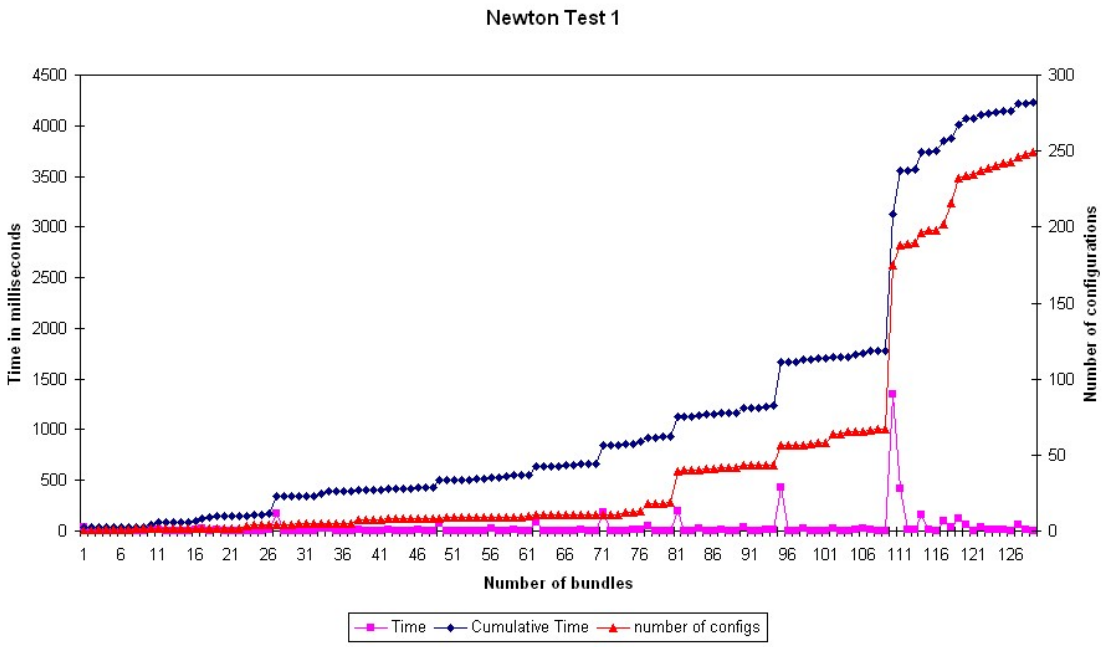


Figure 6.3: Newton Test 1

Following the time needed to add the bundles and the resulting number of new configurations added to the system, 7 bundles can be identified as playing an important role. The fact that the number of configurations is low compared to Felix coupled with the high number of central bundles points to a layered architecture with almost no tight dependencies between layers.

Newton Test 2

The second test also created 249 configurations out of 129 bundles. 4 Bundles could not be resolved because of a dependency cycle.

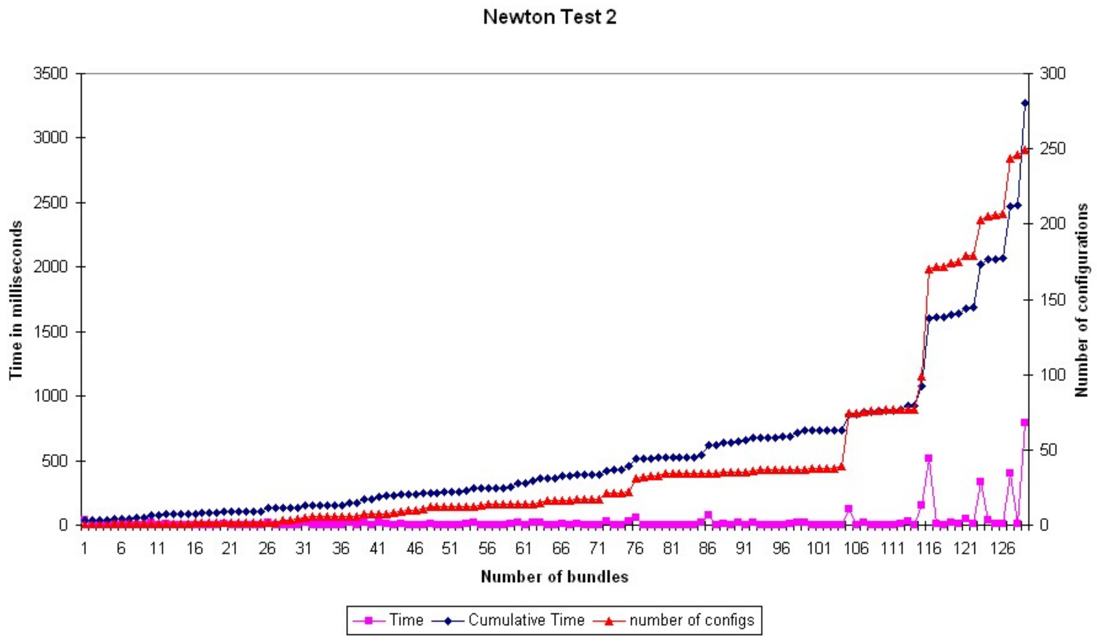


Figure 6.4: Newton Test 2

In this test, bundle Nr. 110 stands out as being a central bundle as the number of configuration doubles due to its addition. The other 6 are still there, but stand out a lot less.

6.1.3 Permutation of Configuration Generation

A permutation test was done to get a clearer picture of the dependency between the number of configurations and the time needed to add a bundle. The test consists of using one of the application (in this case, Apache Felix) and add its bundles to the configuration manager. The order in which the bundles are entered into the system is completely randomized.
The entire process of entering the bundles into the system one at a time and measuring the time needed to generate the configurations was done 1000 times, each with a different bundle ordering. Three mean values were calculated with the measurements:

- *The mean time* is the time in milliseconds needed to add bundle Nr. X into the system where X is a random bundle.
- *The number of configuration* is the number of minimal configuration in the system after bundle Nr. X has been added to the system.
- *The number of unresolved bundles* is the mean number of bundles without

any configuration because some mandatory dependency is missing in the system.

Nr. of Bundles	Mean Time (ms)	Nr. of Configuration	Unresolved Bundles
1	22.650961	0	1
50	46.34488	2	48
55	66.175356	46	16
60	98.28614	133	8
65	164.96752	375	4
68	199.751354	683	3
69	213.96892	789	3
70	226.56507	918	3

Table 6.1: Mean Configuration Generation with Apache Felix

Collecting the results in a graph shows a clearer picture as to the tendencies of both time consumption and number of configurations.

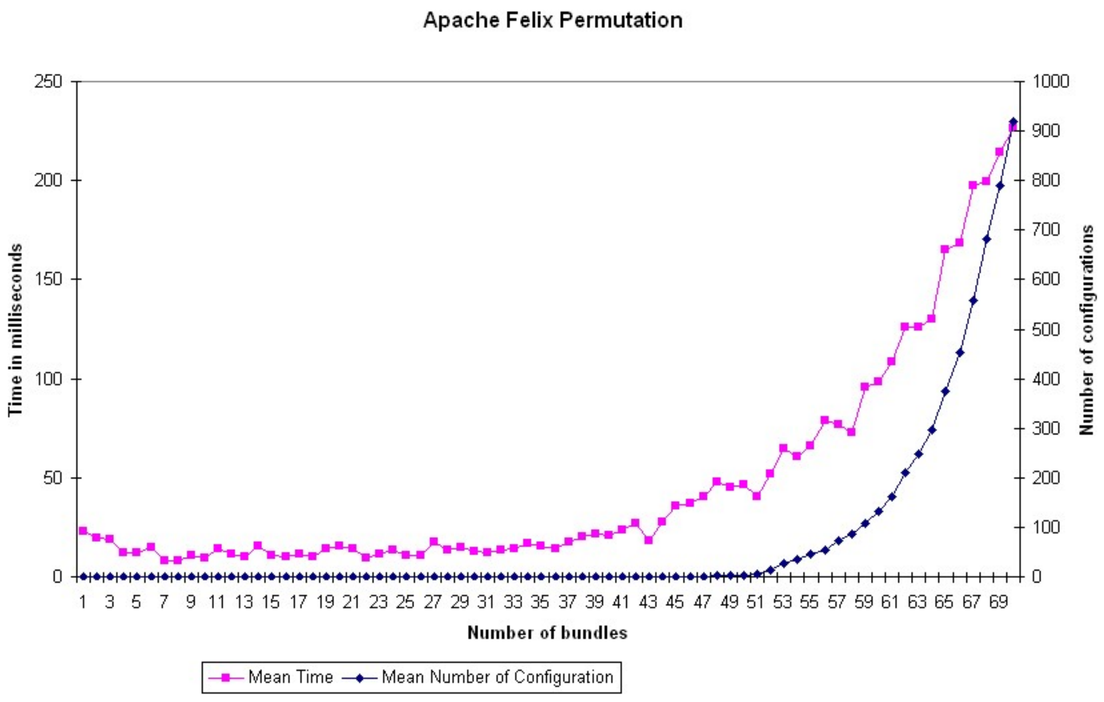


Figure 6.5: Mean Configuration Creation with Apache Felix

The figure 6.5 shows that the generation of configuration in the system is growing exponentially. The time needed to add a bundle is closely related to the number of generated configuration and also tends toward an exponential curve. The fact that the addition of the first few bundles to the system takes more time than some of the latter ones may be due to some JVM specific resource allocation algorithm.

6.1.4 Scalability Tests

After testing the Trinity System with two "real world" applications, a more complex scenario was tried.

One Apache Felix bundle in three was installed in two different versions in the Trinity System. The number of bundles in the system went from 70 valid bundles to 103 where 30 bundles were redundant and taken from older Felix version. Assuming that the dependencies did not change radically between one version to the next, the number of configurations should double as each bundle is interchangeable.

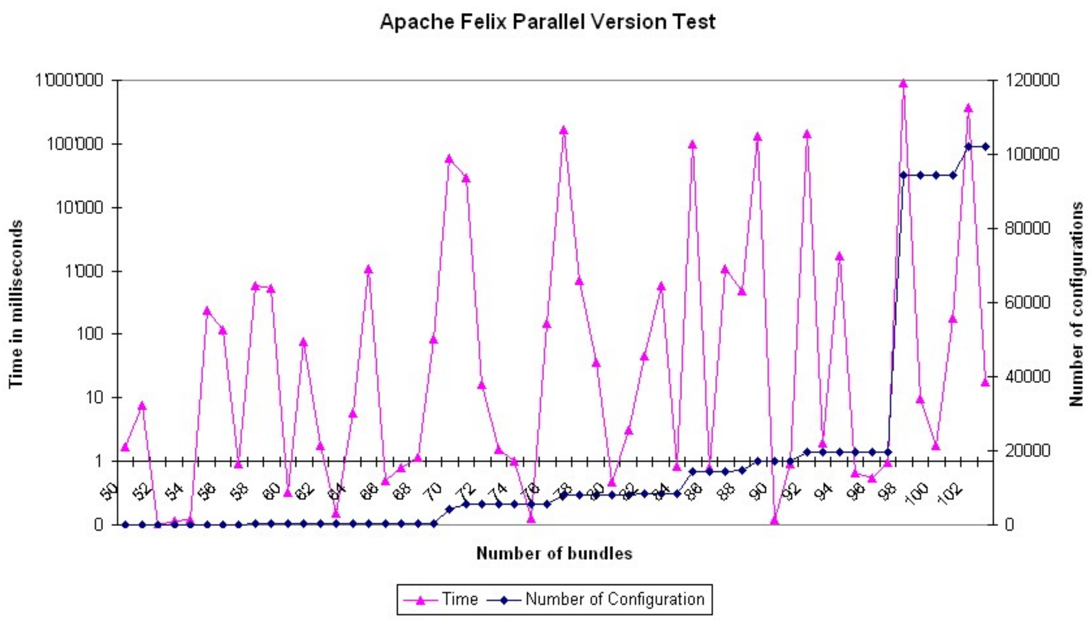


Figure 6.6: Scalability Test with Apache Felix

In the current implementation of the configuration object, the optimization of the scalability is not a priority. A configuration is a heavy-weight object and takes resources both in idle time due to its structure and memory footprint and at calculation time due to the costs of its operations. The Configuration Manager is also not optimized for scalability and has operations that could be optimized to run in linear instead of exponential time. The Configuration Manager holds every configuration in-memory. As the number of configuration increase, which doubles almost constantly when 2 bundles with the same name but different versions are inserted into the system, the resource problem becomes acute and the server can easily throw an "Out of Memory Exception".

This could easily be corrected by using a more appropriate container for the configurations.

After setting the maximum heap size of the Java virtual machine to 1024 Megabytes, the Trinity application finished the test successfully. Graph 6.1.4 on page 55 shows the explosive generation of configurations and their impact on the generation time. The first 50 Bundles are irrelevant to the test and were removed from the graph. As can be seen in the graph, a redesign of the configuration objects and its operations is needed to handle the expected number of configurations the system would have to generate if used over a long period of time.

6.1.5 Test Conclusion

It is quite clear from the permutation test that the number of configuration in the system is growing exponentially. However the application tests show that adding a bundle from a "normal" application into the system does not necessarily use a lot of time.

The primary factor to calculate the time needed to add a bundle is the number of configuration the bundle creates when entering the system. The tests show that the number of configurations calculated by the Trinity Build Server is not dependent on the order in which the bundles are added in the system.

The tests also allowed some conclusions about the applications under test.

- *Central Bundles*: The central bundles of applications could be found through their impact on the configurations. Central bundles are important to an application as their changes have a greater impact on the whole functionality.

- *Missing Requirements*: Bundle which do not pass the installable stage or which do not resolve can be queried as to which import is missing. This helps the developer find build errors when a library is not integrated into the bundle of when a bundle is missing.

6.2 Trinity Test Server

The Test Server was tested using the same version of Apache Felix as before. 73 installable bundles were added to the Trinity System and every one of the 918 generated configurations was systematically tested. The number of parallel child instance was reduced to one (full serial testing) to remove any resource allocation errors.

Some of the configurations could not be tested, throwing an Exception before the actual test could be performed. The reasons for the exceptions, however, were most likely stemming from problems of the composed application and not from the test framework. A possible explanation of the problems has been provided.

6.2.1 Jmood

Exception 1

After all the Mbeans of JMood have been registered, an exception is thrown by the framework while still in the `Activator.start()` method.

```
java.io.IOException:
   Cannot bind to URL [rmi://xxxxxxxx:1199/server]:

java.rmi.ServerException: RemoteException occurred in server thread;
   nested exception is:   java.rmi.UnmarshalException:
       error unmarshalling arguments;
   nested exception is:   java.lang.ClassNotFoundException:
       javax.management.remote.rmi.RMIServerImpl_Stub
       (no security manager: RMI class loader disabled)
```

Reason

The current version of JMood creates its own security Manager if the framework does not provide one. The provided security manager is very simple and allows everything. The current child framework does not start an active security manager per default but the default security manager of JMood did not start.

After starting the security manager, another error was found. The RMI port 1199 is read from a property file inside the bundle and cannot be changed through system properties. This means that JMood will not work if the default port is not available. This bug has been documented in the JMood quickstart text file in the Felix repository (http://svn.apache.org/repos/asf/felix/trunk/jmood/QUICKSTART.txt).

Exception 2

The current Orchestrator tries to first test the resolvability of the configuration before testing it. To do this, the bundles of the configurations are installed and the status is queried. If all the bundles are resolved by the framework, the bundles are deinstalled and reinstalled in order after installing the test bundles. The JMood bundles do not lend themselves to such a method. After reinstalling the bundle, an exception is thrown while starting it again.

```
org.osgi.framework.BundleException:
    Exception in org.apache.felix.jmood.Activator.start()
    of bundle org.apache.felix.org.apache.felix.jmood.
Caused by: javax.management.InstanceAlreadyExistsException:
    osgi.core:type=controller
```

Reason

It seems that the JMood bundle does not unregister its MBean instances from the MBean repository it creates in the background at first start-up. The correct behavior should be to remove the managed instances from the registry when the bundle is deinstalled. If JMood is responsible for the registry, it should also stop it. This bug has been added into the Felix bug tracking system as bug Nr. 623 (https://issues.apache.org/jira/browse/FELIX-623).

6.2.2 Service Binder Example

Exception

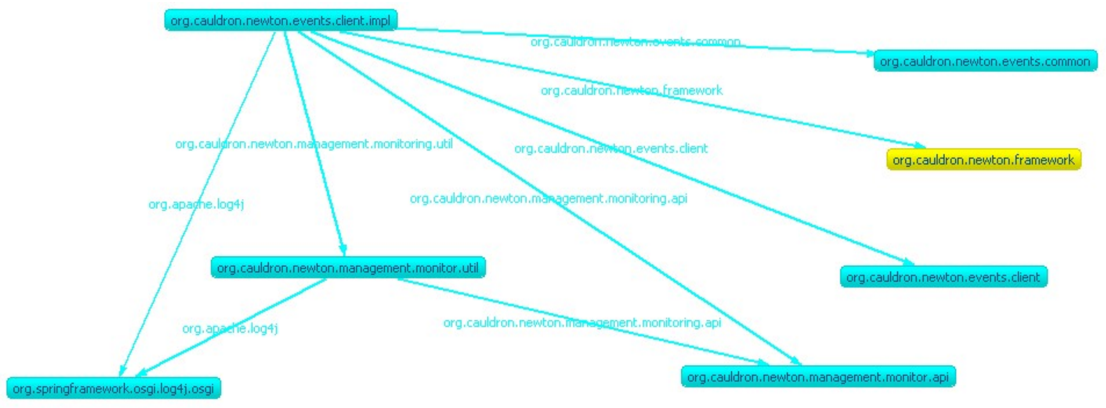
The service binder example bundle threw an exception that at start-up that seemed suspect.

```
org.osgi.framework.BundleException:
   Exception in org.apache.felix.examples.spellcheckbinder.Activator.start()
java.io.FileNotFoundException: MetaData file not found at:
   /metadata.xml
```

Reason

This exception could also be due to a build error and not an actual bug in Felix. There were no predefined build of the example bundles found on the Felix web page to check the used bundle against. To find the error, the source repository was consulted and the project "examples.spellcheckbinder" was compared with the "spellcheckbinder" project. Both the project structure and the pom.xml files were similar. This confirmed the problem as a faulty build.

6.3 Newton


org.cauldron.newton.events.client.impl
org.cauldron.newton.events.common
org.cauldron.newton.events.common
org.cauldron.newton.framework
org.cauldron.newton.framework
org.cauldron.newton.management.monitoring.util
org.cauldron.newton.events.client
org.apache.log4j
org.cauldron.newton.management.monitoring.api
org.cauldron.newton.management.monitor.util
org.cauldron.newton.events.client
org.apache.log4j
org.cauldron.newton.management.monitoring.api
org.springframework.osgi.log4j.osgi
org.cauldron.newton.management.monitor.api

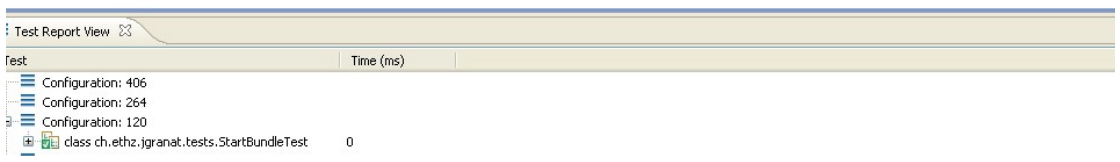

```
Test Report View
Test                                         Time (ms)
  Configuration: 406
  Configuration: 264
  Configuration: 120
    class ch.ethz.jgranat.tests.StartBundleTest   0
```

Figure 6.7: Newton Configuration in Test

6.3.1 Incorrect Bundles

Some bundles in Newton could not be started with a standard OSGi framework because of exceptions thrown while trying to resolve the bundles. The bundles in question could not be started due to errors in the Manifest file. Most of the time, a missing import statement was the cause of the exception.

Here some examples:

- *org.cauldron.newton.frameworkintercept.jar* cannot be resolved because it uses org.osgi.service.packageadmin.PackageAdmin in the Activator without importing the packageadmin package in the manifest file.
- *org.cauldron.newton.instrumentation.jar* cannot be resolved because it uses classes from the org.osgi.framework package without importing it in its manifest file.

6.3.2 Libraries

The Newton application seems to load standard Java libraries into the OSGi Framework through its own mechanism. The standard procedure[15] to incorporate standard Java libraries into OSGi as an extension of the classpath is to declare them as Bundle Fragments to the System Bundle and load them at start-up. This was not done with the Newton libraries and a lot of exceptions were found due to this deviation.

Chapter 7

Future Work

7.1 Trinity Build Server

The Trinity Server in its current version does not make any distinction between *binary bundles* and *source bundles*. *Binary bundles* are bundles developed elsewhere and used by the project, for example a bundle wrapping some open source libraries or a bundle from an affiliated project with a different project life cycle. Binary bundles should be treated as black box and should not be tested specifically in unit tests but only as part of a whole application. *Source bundles* are the bundles created by the project and should be tested in unit tests as well as in the integration tests. Furthermore, a more complete integration into current version control systems like CVS[18] or Subversion[79] would allow automatic snapshot generation and test. The more fine grained the tests are run; the faster the errors are found and corrected. Another possibility would be to integrate the Build Server with other build tools like Ant[68] or Maven[70] to integrate the testing of the different components into the whole building lifecycle.

7.2 Trinity Test Server

Currently, the Test Server has only basic test cases that check the starting functionality of the bundle in the configuration. The higher level tests as described in section 5.1.3 on page 42 have not been implemented yet. Those tests would be a great help to the developers, in particular to those currently writing server-side OSGi applications where scalability is a very important factor as a non-functional requirement.
The Trinity Test Server is not integrated with any form of bug tracking system and one of the nice feature that could be implemented would be the tracking of bugs found in the system by the Test Evaluator and the ability to change the state of the bugs when all the test for a specific bug pass.
The Test Reporter (see section 5.8 on page 45) is a very basic implementation of the tooling the tester could have at his disposition to help the development of better software. The handling of the configurations could be explored further and control given to the tester to save it, redo the tests on it or exclude it from the configuration pool.

7.3 Trinity Deploy Server

Due to a very early evaluation of the Equinox Provisioning Platform (p2 for short)[73], the Deploy Server was removed from the project scope to concentrate the resources on the Build and Test server. However, the Deploy Server is a essential part of the system and should not be left undefined. Here is a possible implementation solution based on the new p2 system.

7.3.1 p2

As of Eclipse project build I20080305 (shortly before Eclipse 3.4/Ganymede M6), the Eclipse SDK contains a new provisioning system called Equinox p2. p2 replaces the Update Manager as a mechanism for managing an Eclipse installation, the search for updates, and the installation of new functionality.[72]

High Level View

An overview of the p2 System and its intended uses can be seen below:

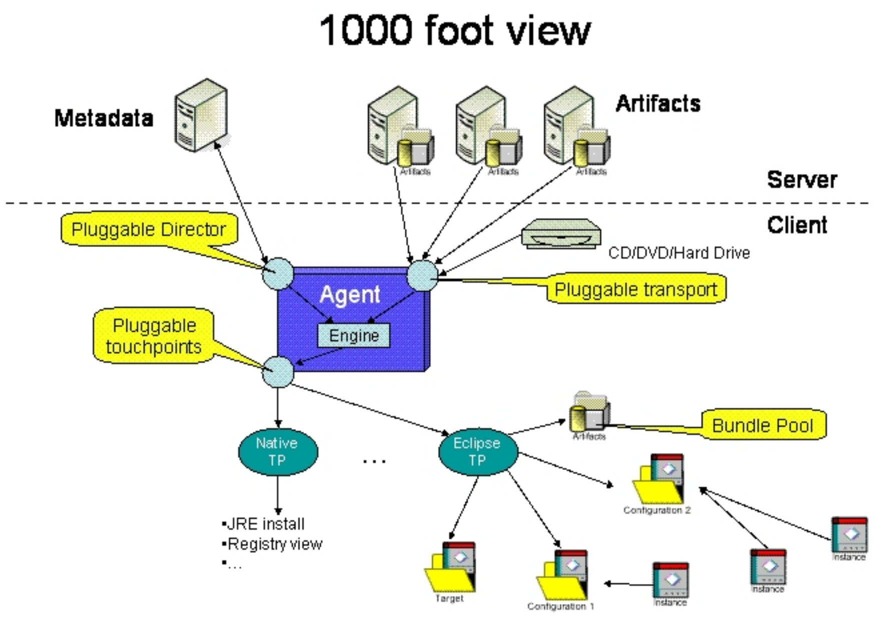


Figure 7.1: High Level View of the p2 System[72]

To understand the system better, a few key components have to be explained in more details:

- *Agent*: The provisioning infrastructure on client machines is generally referred to as the agent. Agents can manage themselves as well as other profiles. An agent may run separate from any other Eclipse system being managed or may be embedded inside another Eclipse system.

- *Artifact*: Artifacts are the elements that are ultimately provisioned to a profile. All managed bundles will be provided as artifacts.

- *Profile*: Profiles are the unit of management in the system. That is, the provisioning infrastructure can manage individual (or collections of) profiles. Profiles are analogous to Trinity Configurations.

- *Installable Units*: The Installable Units are wrappers around artifacts to allow the publishing of the artifact's meta-data and providing a link to the artifact itself. This allows the provisioning system to reason about profiles using the meta data instead of the artifacts themselves, and saving the time needed to download them. The Trinity configurations are based on an acyclic graph of nodes containing the dependency relationships of the artifacts, the artifact itself and a `IBundleInfo` Object which contains the meta data of the artifact, thus providing a convenient 1-to-1 relationship between BundleInfo objects and Installable Units.

- *Touchpoints*: IUs can be stamped with a type. Using this type the engine identifies the touchpoint responsible for marrying the IU with the related system. Each project within the Trinity Server may implement such a touchpoint to put the project bundle in the appropriate spot, adding entries to the necessary configuration files and setting default settings. The set of touchpoints is open-ended.

7.3.2 Promoted Configuration

The number of valid configurations, depending on the number of concurrent version and alternative bundle implementation in the Trinity Server can be quite big and will grow over time. Not all valid configurations should be made available to the end user and thus the concept of *Promoted Configurations* has been defined.
A promoted configuration may be any configuration found on the Trinity Test Server as some test may fail due to missing features but the developer will still want to publish them. A promoted configuration is made available to the p2 clients using the already implemented mechanism of update scheduling and policies, where users are allowed to have detected updates automatically downloaded.

7.3.3 Deploy Policies and Security

The Deploy Server needs to create *profiles* of the different users connecting to it. The users should be able to authenticate themselves and should have a defined role filtering the promoted configuration available.

- *Tester*. The Tester role is given to developer testing the configuration on their system. This role does not filter any of the promoted configurations. The user will be notified of any changes and can select any configuration.

- *Friend.* The Friend role is given to developers of a different project that has a very tight coupling to the configurations made available by the server. The project has to use the latest stable version of a configuration filtering all the intermediate configurations in-between.

- *Distribution.* The Distribution role is given to all users who want to use the application and want to have the latest release of an application. All other configurations in-between are ignored.

The binary bundles are source bundles somewhere and depend on a team creating new version with new features and bug fixes. Ideally, those bundles are managed by other Trinity Servers and new configuration matching the need of the project can be automatically downloaded. This would create a network of project servers propagating tested bundles. Of course, to create a Trinity Network, developer should be able to create profiles on their Build Server to connect to a Deploy Server with a specific role thereby creating the loop Build - Test - Deploy - Build talked about in section 3.1 on page 27.

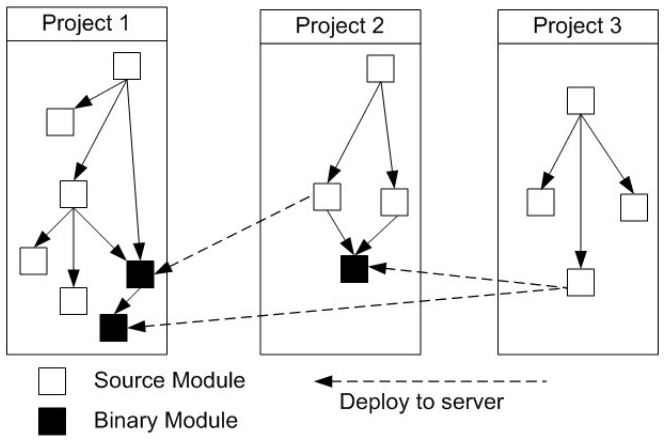


Figure 7.2: Trinity Server Network

Chapter 8

Conclusions

The Trinity System shows that it is possible to test OSGi applications in a real environment, providing analysis tools for the individual components as well as for the whole application.

The generation and test of the different configurations creates a new dimension in the continuous integration process, as not just one but all possibilities are checked. The System allows the concept of a configuration to be generalized as a way to see applications. A valid configuration can be seen as a guideline to a working application as its presence is central to an application from concept to delivery.

The Trinity System provides definite advantages to the developer team of an OSGi software project. The project manager can verify the compatibility of his application through the integration tests and can decide its maturity level from the evaluation of the tests. He also has the proof that it has been written within its specification and can keep track of its compatibility level as an OSGi framework independent software.

It also has a direct impact on the application developer, as the feedback loop between programming and testing in a real OSGi environment is shortened as much as possible. The ability to write tests working inside an OSGi framework allows the detection of the bugs at a very early stage, which makes them easier to correct. The system also helps in support cases by allowing the developer to test specific configurations (given by the user having a problem) and running tests against it to find the problems that may have been overlooked during the deployment. The analysis of the bundles offers the possibility to implements metrics and refactoring helps and to graphically see the dependencies of a bundle in relation to others.

The release manager has a tool to track down the functioning configurations and can select them to be published and downloaded to the clients. As configurations exist in parallel, it is easy for him to see when a new version should be created and in which granularity (nightly build, stable, release, etc), and when to remove the old ones. It is also possible to follow the different version of a project and trace the incompatibilities due to service changes.

Appendix A

Trinity Properties

A.1 General Properties

ch.ethz.jgranat.testserver.core.debug   Switch the Trinity System to debug mode. The debug mode will add more information on the console and do more check than the normal mode. The response time will be slower than normal mode.
ch.ethz.jgranat.testserver.core.silent   Start the application in Silent mode. If set to false, will not start the services of the application when the bundles are started.
ch.ethz.jgranat.testserver.core.logtoconsole   Log the received log messages to the console. If other implementations of the Log Service are available, the value should be set to false.

Table A.1: Properties used by the Core bundle

A.2 Directory Watcher

ch.ethz.jgranat.testserver.bm.dirwatcher.local   If set to true, the Directory Watcher also watch the bundles of the OSGi framework it is running in. This is used if the dirwatcher is placed in a child instance to create IBundleInfo object of the bundles installed in the system.
ch.ethz.jgranat.testserver.bm.dirwatcher.dir   A comma-separated list of URLs defining where to watch out for bundles that should be “installed” in the trinity server.   The URL should be of the form `file://c:/master/watched.jars`.

Table A.2: Properties used by the Directory Watcher

A.3 Test Manager

ch.ethz.jgranat.testserver.trm.basetestdir
A path to the base directory where the tests are saved.

Table A.3: Properties used by the Test Manager

A.4 Test Orchestrator

ch.ethz.jgranat.testserver.torc.parallel
The number of parallel child instance the Test Orchestrator will create to run the tests on the different Configurations

Table A.4: Properties used by the Test Orchestrator

A.5 Bundle Manager

ch.ethz.jgranat.testserver.bm.libs.dir
A list of directory pathes where the libraries used by the application are temporarily stored. The dependencies of the bundles will be adjusted according to the libraries found in these directories
ch.ethz.jgranat.testserver.bm.SystemJars
A list of file pathes pointing to the system libraries. Typically The Java runtime. The dependencies of the bundles will be adjusted according to the libraries found in this path

Table A.5: Properties used by the Bundle Manager

A.6 External Library Properties

ch.ethz.iks.r_osgi.port
The R-OSGi property is used by the Trinity System itself to allow other applications components to connect to it. An example is the default implementation of the Test Reporter.

Table A.6: Properties used by external libraries

A.7 Default Child Creator

ch.ethz.jgranat.testserver.torc.configs The directory where the different framework configurations are placed. Each configuration should be in a subdirectory with this path as root.
ch.ethz.jgranat.testserver.torc.currentconfig The configuration to load and to test the bundles in. The configuration can be completely arbitrary (it can have it's own config.ini).
ch.ethz.jgranat.testserver.torc.basedir The path where the temporary children will be copied to. This directory is used to create and delete the different Children and should not be write protected.
ch.ethz.jgranat.testserver.torc.child.maxwait Maximum number of retry until the Test Orchestrator kills the child with a termination order. This is only used if the child does not respond to a normal close order.
ch.ethz.jgranat.testserver.torc.child.timetowait Maximum time to wait in milliseconds for the close order to work. The Child will retry after the given time frame if the instance is still active.
ch.ethz.jgranat.testserver.torc.child.r_osgi.port The first R-OSGi port used in the child instance. This port will be increated for each new Child instance as the current version of R-OSGi cannot handle port re-use.

Table A.7: Properties used by the Default Child Creator

Appendix B

Trinity Public Interfaces

B.1 The IBundleListener Interface

Implementations of the IBundleListener interfaces will get notified when a new bundle has been installed or deinstalled in the Trinity System.

void	*bundleInstalled(IBundleInfo bundle)* A new bundle has been installed and approved.
void	*bundleDeinstalled(IBundleInfo bundle)* A bundle has been deinstalled from the system.
void	*bundleUpdated(IBundleInfo bundle)* An already installed bundle has been updated.

Table B.1: The IBundleListener Interface

B.2 The ITestManager Interface

ITestManager implementations will be asked bei the Test Orchestrator to deliver new test based on the given configuration.

Collection	*createNewTests(IConfiguration configInfo)* Creates new tests based on the path in the properties and the Test Repository Manager

Table B.2: The ITestManager Interface

B.3 The IConfigurationListener Interface

Implementations of the IConfigurationListener interfaces will get notified when new configurations has been created in the Trinity System.

void	*newConfigurationsCreated(Collection configs)* New configuration have been created by the system and can now be evaluated

Table B.3: The IConfigurationListener Interface

B.4 The ITestRepositoryManager Interface

ITestRepositoryManager implementations are responsible for the saving of the tests artifacts. The getTests method will be called by the Test Orchestrator on every implementation.

void	*saveTest(JarWrapper test, IConfiguration config)* Saves the given test, linked to the given configuration. It is not mandatory to link the test to the configuration.
void	*saveAllTests(Collection tests, IConfiguration config)* Saves all tests in the collection linked to the given configuration. It is not mandatory to link the test to the configuration.
Collection	*getTests(IConfiguration config)* Returns all Tests linked with the configuration The returned tests should be compatible with the requirements and capabilities of the configuration.

Table B.4: The ITestRepositoryManager Interface

B.5 The IChildFactory Interface

The IChildFactory interface is used to create new child instances.

IChildInstance	*createChild()* throws InitializationException Creates a new child instance based on the predefined configuration.
IChildConfiguration	*getConfiguration()* throws InitializationException Returns the predefined child configuration.

Table B.5: The IChildFactory Interface

B.6 The IChildInstance Interface

The IChildInstance represent an abtraction of an OSGi independent instance. The methods of this interface are quite extensive and only the most important ones are shown below.

StreamProxy	*getOut()* Returns the stream to the "System.out" of the child instance
StreamProxy	*getError()* Returns the stream to the "System.err" of the child instance
IBundleInfo	*installBundle(String name, InputStream bundle)* Install the Bundle in the stream with the given name in the child framework. Returns an IBundleInfo if the installation was successful, null otherwise.
void	*removeBundle(Long id)* Remove the bundle with the id 'id' from the child framework.
Collection	*getInstalledBundles()* Returns all installed bundles.
boolean	*isAllResolved()* Check whether the bundles installed on the child instance all resolve correctly or not.

Table B.6: Some of the IChildInstance Methods.

B.7 The IReportManager Interface

IReportManager implementation are responsible for the monitoring of the tests.

void	*beginMonitoring(IConfiguration configInfo, IChildInstance child)* Notifies the Report Manager that a test series on a configuration has begun. The child instance where the tests are being done is passed as argument to install and/or start monitoring bundles.
void	*endMonitoring(IConfiguration configInfo, IChildInstance child)* Notifies the Report Manager that a test series on a configuration has ended. The child instance where the tests are being done is passed as argument to either do some cleaning-up or retrieve data.
void	*testEnded(ITestSuiteResult result, IConfiguration config)* This method is used as an alternative to the event driven notification. It notify the Report Manager that a test has just ended and gives the result of the test as a parameter.

Table B.7: The IReportManager Interface

B.8 The IConfiguration Interface

The IConfiguration Interface represent a configuration. A configuration is a set of dependent bundles.

boolean	*hasBundle(IBundleInfo bundle)* Check whether the given bundle is in the configuration or not.
IBundleInfo[]	*getBundles()* Returns all the bundle contained in the configuration.
Capability[]	*getExports()* Returns all Capabilities (exporing packages and registered services) of the configuration.
Requirement[]	*getImports()* Returns all Requirements (importing packages and used services) of the configuration.
boolean	*isComplete()* Check whether all Requirements (importing packages and used services) are fullfilled by the configuration.
Collection	*getMissingRequirements()* Returns a collection of Requirements that are not fullfilled by the configuration.

Table B.8: The IConfiguration Interface

B.9 The ITestEvaluator Interface

ITestEvaluation	*getTestEvaluation(IConfiguration configInfo)* Get the evaluation of the given configuration. An evaluation is based on the tests done using the configuration and user specific evaluation criteria.
void	*addTestResult(IConfiguration config, ITestSuiteResult result)* Add the test suite result of a finished test done using the given configuration.

Table B.9: The ITestEvaluator Interface

B.10 The IBundleManager Interface

The IBundleManager implementation publish the management of the bundles as an alternative to the BundleContext object. This interface is used to communicate with the child instances (installing/starting bundles). It was created to allow the transparent bundle management between different OSGi instances.

IBundleInfo	*installBundle(String name, InputStream bundle)* Installs the given bundle in the current OSGi instance with the given name. Returns *null* if the bundle could not be installed.
void	*uninstallBundle(long id)* Uninstalls the bundle with the given id. The id is the one given by the OSGi instance the BundleManager is running in.
void	*startBundle(long id)* Starts the bundle with the given id. The id is the one given by the OSGi instance the BundleManager is running in.
void	*stopBundle(long id)* Stops the bundle with the given id. The id is the one given by the OSGi instance the BundleManager is running in.
void	*updateBundle(long id)* Updates the bundle with the given id. The id is the one given by the OSGi instance the BundleManager is running in.
Collection	*getInstalledBundles()* Returns a collection of IBundleInfo Objects representing the installed bundles in the OSGi instance the BundleManager is installed in.
boolean	*isAllResolved()* Returns true if all the bundles installed in the OSGi instance the BundleManager is in have the status "Resolved" or above.

Table B.10: The IBundleManager Interface

B.11 The IBundleInfo Interface

IBundleInfo implementations represent a Bundle artifact and all its meta data. It extends the interface "org.osgi.service.obr.Resource" adding only some getters.

Bundle	*getBundle()* Returns the underlying bundle object represented by this IBundleInfo object
void	*refreshInfo()* Forces a refresh of the bundle information
int	*getStatus()* Returns the status of the bundle.

Table B.11: The IBundleInfo Interface

B.12 The ITestResult Interface

An ITestResult implementation is returned for each Test found in the generated test bundles. This interface is closely related to the JUnit *TestResult* Object. Another interface, the *ITestSuiteResult* is analogous to the ITestResult interface but represent a suite (collection) of test cases.

boolean	*wasSuccessful()* Returns whether the entire test was successful or not.
int	*errorCount()* Returns the number of detected errors.
Enumeration	*errors()* Returns an Enumerations for the list of errors.
int	*failureCount()* Returns the number of detected failures.
Enumeration	*failures()* Returns an Enumerations for the list of failures.
int	*runCount()* Return the number of time this test was run in the child instance.
long	*getTestTime()* Returns the time in milliseconds the test took to run.
String	*getTestName()* Returns the name of the test.
Collection	*getTestMethodNames()* Returns a list of names representing the names of the methods this test is composed of.
String	*getTestClass()* Returns a String representation of the class name of the test class.

Table B.12: The ITestResult Interface

Appendix C

Trinity Generated Event

C.1 Event Sets

There are 2 different set of events that are generated by the Trinity System. The publishing of the events is based on the distributed event implementation of R-OSGi which can distribute the events between different JVMs. The distribution of the events happens transparently and client plug-ins just have to connect to the server through the R-OSGi port to get the relevant events generated by the Test Orchestrator.

The first set is reserved for the Test Orchestrator and contains events generated by the child instances. The second set is created by the Test Orchestrator and is intended for all other interested parties. The mapping of event is completely 1-to-1. The Test Orchestrator will transparently consume the child events and forward it in the corresponding parent event with all its attributes copied from the child event.

The events have the same schema but different root definitions:

ch/ethz/jgranat/testserver/core/Child The root "object" for all child events. The Child events are reserved for the Test Orchestrator and should not be consumed by anyone else.
ch/ethz/jgranat/testserver/core/Parent The root "object" for all Trinity events. The Test Orchestrator will re-send the child events and change the root Object but transparently pass every other attribute of the event.

Table C.1: The 2 event roots

C.2 Default Events

Both sets and all the attributes used in the Trinity System are defined in the core interface ch.ethz.jgranat.testserver.core.child.ITestRunner. The events generated by the child instance are optional, meaning that new implementations may not send the whole set, and are generated to make the whole testing more user friendly. The more events the child instance is sending, the more interactive the user interfaces can be written. The minimum list of attributes is also defined but can be extended to fit the need of future clients or child instances.

The default events and their meaning can be seen in the table below:

.../*BEGINMETHOD* This event is sent just before a test method is started. The parameters for this events are: *testname*: The name of the overall test class. *testmethodname*: The name of the starting method. *testconfig*: The Configuration Object.
.../*ENDMETHOD* This event is sent just after a test method has ended. The parameters for this events are: *testname*: The name of the overall test class. *testmethodname*: The name of the starting method. *testconfig*: The configuration object. *testresult*: The result (an ITestResult implementation) of the method test.
.../*BEGINTEST* This event is sent just before a test class is started. The parameters for this events are: *testname*: The name of the overall test class. *testconfig*: The Configuration Object.
.../*ENDTEST* This event is sent just after a test class has ended. The parameters for this events are: *testname*: The name of the overall test class. *testconfig*: The configuration object. *testresult*: The result (an ITestSuiteResult implementation) of the method test.
.../*FINISHED* This event is sent just after the last test class has ended or if an error happened in the whole test process. The parameters for this events are: *testconfig*: The configuration object.

Table C.2: The events generated by the Trinity System

The default events are generated in a structured way but may not be received in the same order as they were sent, depending on the EventAdmin implementation. The default event structure is shown in the graph below.

```
sent by the Child and forwarded by the Test Orchestrator:
for every test found
  send /BEGINTEST
  for every method in TestCase
    send /BEGINMETHOD
    test method
    send /ENDMETHOD
  send /ENDTEST

sent by the Test Orchestrator:
/FINIHSED
```

Appendix D

Errors in Trinity

D.1 Used Library

D.1.1 R-OSGi

Reuse of Ports

The current R-OSGi implementation does not work well with the reuse of ports for new child frameworks. The first implementation of the child framework tried to reuse the ports of destroyed children. The R-OSGi library, however, tried to use the same proxy bundle for a different instance and threw an exception. This bug has been submitted into the bug tracking system of R-OSGi (http://sourceforge.net/tracker/index.php?func=detail&aid=2010407&group_id=158382&atid=807609)

Restart Problem

The current R-OSGi implementation does not allow a restart of the bundle. R-OSGi creates a ServerSocket to listen to incoming calls from other OSGi instances at start-up and does not destroy it at shut down. Instead, a new port is opened. This can cause strange behavior in the eclipse client plug-ins when trying to access the reports on the server. This bug has been submitted into the bug tracking system of R-OSGi (http://sourceforge.net/tracker/index.php?func=detail&aid=2010399&group_id=158382&atid=807609).

ClassCastException

The R-OSGi library threw a ClassCastException on rare occasions. The error could not be reproduced consistantly to understand what exactly went wrong in the communication.

```
WARNING: ch.ethz.iks.r_osgi.messages.StreamRequestMessage
   cannot be cast to ch.ethz.iks.r_osgi.messages.MethodResultMessage

java.lang.ClassCastException: ch.ethz.iks.r_osgi.messages.StreamRequestMessage
      cannot be cast to ch.ethz.iks.r_osgi.messages.MethodResultMessage
```

```
at ch.ethz.iks.r_osgi.impl.ChannelEndpointImpl.
            invokeMethod(ChannelEndpointImpl.java:301)
at ch.ethz.iks.r_osgi.impl.ChannelEndpointMultiplexer.
            invokeMethod(ChannelEndpointMultiplexer.java:125)
at proxy.bchoaoaob_feggcj.ch.ethz.jgranat.testserver.core.
            bm.IBundleManagerImpl.installBundle(Unknown Source)
  . . .
```

D.2 Trinity Errors

D.2.1 Socket Check

After extensive tests, it was noted that the current implementation of the child instance of the Trinity Test Server did not check for socket availability at creation time. This led to false negatives where the System marked a configuration as not working when in fact the error was due to a port already in use by another system.

```
java.net.BindException: Address already in use: JVM_Bind
at java.net.PlainSocketImpl.socketBind(Native Method)
at java.net.PlainSocketImpl.bind(PlainSocketImpl.java:359)
at java.net.ServerSocket.bind(ServerSocket.java:319)
at java.net.ServerSocket.<init>(ServerSocket.java:185)
at java.net.ServerSocket.<init>(ServerSocket.java:97)
at org.eclipse.osgi.framework.internal.core./
          FrameworkConsole.getSocketStream(FrameworkConsole.java:125)
  at org.eclipse.osgi.framework.internal.core./
          FrameworkConsole.run(FrameworkConsole.java:216)
at java.lang.Thread.run(Thread.java:619)
```

D.2.2 RequireBundle Keyword

The Trinity Analyzer System cannot, as yet, resolve the “RequireBundle” keyword. The current analyzer view every bundle as a “closed entity” with defined package dependencies. The RequireBundle keyword allows packages to be loaded and accessed by the bundle without any explicit declaration. This is currently not implemented in the system and can lead to correct bundles not being added to the system.

D.2.3 Service Analyzer

The service analyzer, as described in section 4.4.1 on page 33, can only find services which use the OSGi framework interface to register its services. In the case of Apache Felix (see section 6.1.1 on page 49 for more details on Felix), the services have been encapsulated in an application specific wrapper. The number of services found in cases like Felix is therefore not complete.

A new version of the analyzer could find those encapsulation and add them to the list of patterns defining the consuming or publishing of services.

D.2.4 Dependency Cycle

As seen with the Newton test application (see section 6.1.2 on page 52 for more details), The Trinity System cannot, as yet, resolve dependency cycles. A dependency cycle occurs when bundle A needs package B while providing package A and bundle B needs package A while providing package B.

Glossary

Bundle Graph A Bundle Graph is the meta data representation of a configuration. It represent a set of bundles connected to each other by their bindings. Each node in the graph is defined as the meta data of the bundle, the bundle itself and its bindings to other bundles. Each edge is part of a node and connect to another node with a requirement as edge qualifier, 38, 68

Bundle Meta-Data The Meta-Data of a bundle is the information about the bundle that can be read or calculated out of it. It contains essential information about the bundle (e.g., names, ids, version numbers, dependencies, etc), 34

Capability An OBR keyword representing anything that can be described with a set of properties (e.g. A package or service export, a bundle, an Execution Environment, a Display type, the size of the available memory, etc). Capabilities are named. The reason they are named is so that they can only be provided to requirements with the same name. Capabilities can originate from resources, but can also be innate in the environment, 39

Child Instance A child instance is a separate process created for the express purpose of testing some configuration against a series of tests. The minimum (and currently implemented) child instance is a separate OSGi process where the bundle are remotely installed and started. There is no restrictions as to the nature of a child instance and, depending on the application, could involve the specification of different Java version or different operating systems or the use of a complete mobile environment, 42, 45

Configuration In the most abstract sense of the term, a configuration is a set of OSGi Bundles. In Trinity, the sense of a configuration has been narrowed to bundles having some kind of connection to each other in the form of requirements/capabilities. A configuration is then essentially a bundle graph, 28

Minimal Configuration A minimal configuration or resolvable configuration is a set of bundle with no missing mandatory requirements (tight coupling). However, the requirements taken into account are "only" the OSGi application specific ones (imports of packages or services), 36

Complete Configuration A complete configuration is a set of bundle with neither mandatory nor optional requirements missing (tight and loose coupling). However, the requirements taken into account are “only” the OSGi application specific ones (imports of packages or services) , 36, 41

Promoted Configuration A promoted configuration is a configuration that has been selected to be published on the deploy server and can be downloaded by the client systems, 68

JarWrapper A JarWrapper is a wrapper over a dynamically created jar file used to make it more convenient to access it, 43

Loose Coupling Loose couplings are optional requirements and are expressed as service imports/exports in OSGi. It means that the requirements does not have to be fulfilled for the bundle to resolve and start, 36

Managed Bundle A Managed Bundle is a bundle which has been added to the Trinity System, 31

Binary Bundle A managed bundles developed outside the control of the Trinity System and added as a “used bundle”. Binary bundles are not meant to be unit tested, 65

Source Bundle A managed bundles developed by the project the Trinity System is responsible of. Source bundles are unit tested and have a higher rate of versioning cycle as binary bundles, 65

Profile A profile is the P2 analogy to a configuration. It represent a set of bundles that can be installed on a client system, 68

Requirement An OBR keyword representing the counterpart to (or need for) capabilities. It is expressed as a filter on a resource. The filter must only be matched to capabilities with the same name. A requirement matches a capability when its filter matches any of the properties defined in that capability. A requirement can optionally contain a reason. A reason is a short description that is applicable when a requirement is the cause for the selection of a resource, 39

Staged Test Pipeline A multi-step process defining the steps a bundle goes through from its creation to its deployment as part of a working profile. The Staged Test Pipeline is composed of two distinct part. The first part defines the steps related to a single bundle, the second part defines the steps related to entire configurations, 32, 41

Test Bundle A test bundle is a generated bundle containing JUnit tests that will be packaged in an OSGi bundle and installed on the child instance to test the configuration. The dependencies of the bundle is analyzed at creation time and the test bundle will only be added to the configurations where all its requirements can be resolved, 28, 43

Tight Coupling Tight couplings are mandatory requirements/capabilities and are expressed as package imports/exports in OSGi. It means that the requirements have to be fulfilled for the bundle to resolve and start, 36, 41

White Board Pattern Programming Pattern where the notification is based on the inversion of control principle, 36, 39

Bibliography

[1] A. Assad, A. Santos, L.-F. Guimaraes. Service-oriented architecture testing design and practices. http://www.cesar.org.br/pdf/ServiceOrientedArchitectureTestingDesignandPractices.pdf, 2006.

[2] A. Colyer. Spring osgi with adrian colyer. http://raibledesigns.com/rd/entry/tse_spring_osgi_with_adrian, 2006.

[3] A. Orso, H. Do, G. Rothermel, M. J. Harrold, and D. Rosenblum. Using component metadata to regression test component-based software. In *Journal of Software Testing, Verification, and Reliability*, volume 17, pages 61–94, June 2007.

[4] A. Ryan, J. Newmarch. An architecture for component evolution. In *Consumer Communications and Networking Conference*, pages 498–503. IEEE Computer Society, 2005.

[5] S. Ambler. Agile software development. http://www.agilemodeling.com/essays/agileSoftwareDevelopment.htm, 2007.

[6] B. Boehm. The spiral model. http://www.maxwideman.com/papers/linearity/spiral.htm, 1988.

[7] C. Oriat. Jartege: A tool for random generation of unit tests for java classes. In *QoSA / SOQUA*, pages 242 – 256, September 2005.

[8] Codice Software. Integration strategies. http://codicesoftware.blogspot.com/2008/04/integration-strategies.html, 2008.

[9] D. Angluin and C. H. Smith. Inductive inference: Theory and methods. In *ACM Computing Surveys*, volume 15, pages 237–269, September 1983.

[10] D. Bianculli, C. Ghezzi. Towards a methodology for lifelong validation of service compositions. In *Proceedings of the 2nd international workshop on Systems development in SOA environments*, pages 7–12. Association for Computing Machinery, 2008.

[11] D. M. Cohen, S. R. Dalal, M. L. Fredman, G.C. Patton. The aetg design: an approach to testing based on combinatorial design. In *trans on Software Engineering*, volume 23, pages 437–444. IEEE, 1997.

[12] D. Nieland, W. Dunn, D. Kamlani. Osgi provides open platform for the internet-enabled car. http://www.osgi.org/wiki/uploads/News/pressrel1016900.pdf, 2008.

[13] D. Saff. JUnit. http://en.wikipedia.org/wiki/JUnit, 2007.

[14] E. W. Weisstein. Definition of a dynamical system. http://mathworld.wolfram.com/DynamicalSystem.html, 2008.

[15] Equinox development mailing list. System bundle package exports. http://osdir.com/ml/ide.eclipse.equinox.devel/2007-01/msg00043.html, 2007.

[16] F. Barbon, P. Traverso, M. Pistore, M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS 06 Proceedings*, pages 63 – 71. IEEE Computer Society, 2006.

[17] M. Fowler. Continuous Integration. http://www.martinfowler.com/articles/continuousIntegration.html, 2006.

[18] Free Software Foundation. CVS - Concurrent Versions System. http://www.nongnu.org/cvs/, 1998.

[19] G. Rothermel and M. J. Harrold. Selecting regression tests for object-oriented software. In *the International Conference on Software Maintenance*, pages 14–25, September 1994.

[20] H. Vo. Iterative process. http://www.vocw.edu.vn/content/m10078/latest/, 2008.

[21] IEEE Standards Board. *IEEE Standard for Software Unit Testing: An American National Standard*, volume 2, chapter IEEE Standards: Software Engineering, pages 1008–1987. The Institute of Electrical and Electronics Engineers, 1999 edition edition, 1987.

[22] R. Iosif. Formal verification applied to java concurrent software. In *International Conference on Software Engineering archive*, pages 707 – 709, 2000.

[23] Iterative Development. Definition: Iterative and incremental development. http://en.wikipedia.org/wiki/Iterative_and_incremental_development, 2008.

[24] J. Ellson, E. Gansner. Graphwiz. http://www.graphviz.org/, 2004.

[25] J. Hartmann and D. Robson. Revalidation during the software maintenance phase. In *Conference on Software Maintenance*, pages 70–79, October 1989.

[26] J. Maron, G. Pavlik. The evolution of soa application development. http://si.vse.cz/archiv/clanky/2006/maron.pdf, 2006.

[27] J. S. Rellermeyer, G. Alonso, T. Roscoe. R-osgi Distributed applications through software modularization. In *Middleware 2007*, volume Volume 4834/2007 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin / Heidelberg, 2007.

[28] y. . . h. J. Sloan a.k.a. Chip Overclock, title = The Total Cost of Code Ownership.

[29] J. Yang and D. Evans. Automatically inferring temporal properties for program evolution. In *International Symposium on Software Reliability Engineering*, pages 340 – 351, November 2004.

[30] Jan S. Rellermeyer. R-OSGi. http://r-osgi.sourceforge.net/index.html, 2008.

[31] JUnit Community. JUnit cookbook. http://junit.sourceforge.net/doc/testinfected/testing.htm, 2003.

[32] JUnit Community. JUnit homepage. http://www.junit.org/, 2008.

[33] Kellerman Software. NUnit Test Generator. http://www.kellermansoftware.com/p-30-NUnit%20Test%20GeneratorSpecifications.aspx?mode=Specifications, 2008.

[34] D. Knuth. *The Art of Computer Programming*, volume 3, chapter 3.2.1. Addison-Wesley, 1998.

[35] P. Kriens. OSGi and testing. http://www.aqute.biz/Blog/2005-06-27, 2005.

[36] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, P. Spoletini. Validation of web service compositions, 2007.

[37] L. Baresi, E. Di Nitto, C. Ghezzi. Toward open-world software Issue and challenges. In *Computer*, volume 39, pages 36–43. IEEE Computer Society Press, October 2006.

[38] L. G. Smith, D. Kontogiannis. Soam 2008: 2nd workshop on soa-based systems maintenance and evolution. In *Software Maintenance and Reengineering*, pages 336–337, 2008.

[39] L. Hatton. Does OO really match the way we think? http://www.leshatton.org/Documents/OO_IS698.pdf, 1997.

[40] M. Kajko-Mattsson, G. A. Lewis, D. B. Smith. Roles for maintenance and evolution of soa-based systems. http://www.cs.vu.nl/csmr2007/workshops/3-2007.

[41] M. S. Mimoso. Survey: SOA prominent on 2005 budgets. http://searchsoa.techtarget.com/news/article/0,289142,sid26_gci1010622,00.html, 2004.

[42] Method123. Project management life cycle. http://www.method123.com/project-lifecycle.php, 2003.

[43] Methods and Tools. Understanding the unified process (up). http://www.methodsandtools.com/archive/archive.php?id=32, 2002.

[44] M.J. Harold, J.D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oritented class structures. In *the 14th International Conference on Software Engineering*, pages 68–80, May 1992.

[45] OASIS Research Team. OSGi Bundle dependency. http://proactive.inria.fr/release-doc/html/ProActiveManual.html#OSGi_html_overview, 2007.

[46] ObjectWeb. Definition asm. http://asm.objectweb.org/, 2007.

[47] Open Service Gateway Initiative. Osgi homepage. http://www.osgi.org/, 2008.

[48] OSGi Alliance. Listener pattern considered harmful: The "whiteboard" pattern. http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf, 2004.

[49] OSGi Alliance. RFC-0112 bundle repository. http://www.osgi.org/download/rfc-0112_BundleRepository.pdf, 2005.

[50] OSGi Alliance. Bundle: OSGi Service Platform release 4 version 4.1. http://www2.osgi.org/javadoc/r4/org/osgi/framework/Bundle.html, 2007.

[51] OSOA Collaboration. Power combination: Sca, osgi and spring. http://www.osoa.org/download/attachments/250/Power_Combination_SCA_Spring_OSGi.pdf, 2006.

[52] P. Collet, T. Coupaye, H. Chang, L. Seinturier, G. Dufrêne. Components and services: A marriage of reason. http://www.i3s.unice.fr/~mh/RR/2007/RR-07.17-P.COLLET.pdf, 2007.

[53] Paremus Limited. Newton framework. http://newton.codecauldron.org/site/index.html, 2008.

[54] R. S. Hall. Oscar bundle repository. http://oscar-osgi.sourceforge.net/, 2005.

[55] R. S. Hall, H. Cervantes. Challenges in building service-oriented applications for osgi. In *Communications Managzine*, volume 42, pages 144–149. IEEE Computer Society, 2004.

[56] R. Z. Weinreich, T. Draheim, D. Draheim. A versioning model for enterprise services. In *Advanced Information Networking and Applications Workshops*, pages 570–575. IEEE Computer Society, 2007.

[57] S. Elbaum, D. Gable, G. Rothermel. The impact of software evolution on code coverage information. In *IEEE International Conference on Software Maintenance (ICSM'01)*, page 170. IEEE Computer Society, 2001.

[58] S. Freeman, N. Pryce, T. Mackinnon, J. Walnes. Mock roles, not objects. http://jmock.org/oopsla2004.pdf, 2004.

[59] S. J. Zeil. The waterfall model. http://www.cs.odu.edu/~zeil/cs451/Lectures/01overview/process2/process2_htsu2.html, 1999.

[60] S. Jeong, G. Yeo, Y. Jang, S. Sung, H. Lee. Method of testing open services gateway initiative serviceplatform and test tool using the method. http://www.wipo.int/pctdb/en/wo.jsp?wo=2005081106, 2005.

[61] S. R. Schach. *Object-Oriented and Classical Software Engineering.* McGraw-Hill, 2004.

[62] Spring Community. Spring Dynamic Modules for OSGi Service Platforms. http://www.springframework.org/osgi/, 2008.

[63] Spring Community. Spring framework. http://www.springframework.org/, 2008.

[64] Sun Microsystem. Java Platform, Enterprise Edition (Java EE). http://java.sun.com/javaee/, 2006.

[65] Sun Microsystems. Sun microsystems: Versioning of serializeable objects. http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/version.html#9419, 2004.

[66] Sun Microsystems Inc. JAR File Specification. Technical report, http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html#JAR%20Manifest, 2003.

[67] T. Standish. An essay on software reuse. Transactions on Software Engineering SE-10, 1984.

[68] The Apache Software Foundation. Ant: Another neat tool. http://ant.apache.org/, 2008.

[69] The Apache Software Foundation. Apache felix. http://felix.apache.org/site/index.html, 2008.

[70] The Apache Software Foundation. Maven build project. http://maven.apache.org/, 2008.

[71] The Eclipse Foundation. Eclipse. http://www.eclipse.org/, 2008.

[72] The Eclipse Foundation. Equinox p2 getting started. http://wiki.eclipse.org/Equinox_Provisioning_Getting_Started, 2008.

[73] The Eclipse Foundation. Equinox p2 wiki. http://wiki.eclipse.org/Equinox_Provisioning/, 2008.

[74] The Eclipse Foundation. Osgi technology. http://www.osgi.org/About/Technology, 2008.

[75] The OSGi Alliance. *OSGi Service Platform Core Specification.*

[76] The OSGi Alliance. About the OSGi Service Platform. Technical report, http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf, 2007.

[77] The OSGi Alliance. *OSGi Service Platform Service Compendium.* I O S Press, 2007.

[78] ThoughtWorks. Cruise control. http://cruisecontrol.sourceforge.net, 2001.

[79] Tigris.org Community. Subversion. http://subversion.tigris.org/, 2006.

[80] Verifysoft Technology. Conformiq qtronic. http://www.verifysoft.com/en_qtronic_testautomation.html, 2008.

[81] VMWare. Vmware homepage. http://www.vmware.com/, 2004.

[82] W. Visser, C. S. Pasareanu, S. Khurshid. Test input generation with java pathfinder. In *International Symposium on Software Testing and Analysis*, pages 97 – 107. ACM: Association for Computing Machinery, 2004.

[83] Wikipedia. Regression testing definition. http://en.wikipedia.org/wiki/Regression_test, 2008.

[84] W.T. Tsai. *Service-oriented system engineering: a new paradigm*, volume Service-Oriented System Engineering, chapter 1, pages 3–6. INSPEC Accession, October 2005.

[85] XP. eXtreme Programming: A gentle introduction., 2006.