Slice-Oriented Programming for Resource-Constrained Target Environments

by

Keith John Nixon, B.Sc.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

August 2011

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Keith John Nixon

August 31, 2011

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this dissertation upon request.

Keith John Nixon

August 31, 2011

To “6”.

Acknowledgments

Firstly, my sincere gratitude to my supervisor, Dr. Siobhán Clarke, for both the guidance and critique necessary for completing this work. I'd also like to thank Siobhán for her tireless endeavours, for us as a class, throughout the year.

I'd also like to thank my family for both supporting and encouraging me throughout the year, while I negotiated the formidable NDS workload.

It has been a very challenging year; probably my greatest challenge to date. However, it has been both an enjoyable and fulfilling adventure.

Keith John Nixon

University of Dublin, Trinity College

August 2011

Slice-Oriented Programming for Resource-Constrained Target Environments

Keith John Nixon, M.Sc.
University of Dublin, Trinity College, 2011

Supervisor: Dr. Siobhán Clarke

An embedded device is usually, but not always, a hardware-constrained resource which can be leveraged to provide information pertaining to the object in which the device resides. By virtue of their ubiquitous nature, embedded devices are at the forefront of the prodigious shift towards the Web of Things. This emerging paradigm aims to integrate inanimate, everyday objects into the World Wide Web. As a result, unprecedented opportunities for service-oriented applications across multiple domains are possible.
However, embedded devices may be characterised by a constrained operational profile. Or, they may be limited by the vagaries typical of mobile environments, such as intermittent network availability. Notwithstanding, it is necessary that web services account for these drawbacks. By doing so, an acceptable QoS can be maintained for the client such that the end users experience is impeded as little as possible. At both build-time and maintenance time, this requires the programmer to compose service code appropriate to the different operational contexts of a client device. However, using conventional techniques can introduce issues such as poor compositional flexibility, additional object dependencies and additional code modifications. A higher level of programming abstrac-

tion of the service composition process, with supporting functional decomposition and composition programming environment would ameliorate these difficulties. Such facilities would facilitate the specification and updating of the services composition without incurring additional object dependencies and code modifications.

This dissertation introduces a domain-specific language, which allows a programmer to stipulate appropriate service compositions by automatically decomposing the aggregate service logic into executable slices. This process may occur at either build-time or maintenance-time. A slice is defined as an executable subset of the services aggregate logic. Slices are associated with profiles defined in an instance of the domain-specific language, each of which defines a range of client contexts. As a result, slices can be delivered to a service client in accordance with the clients operational context. This also ensures that the service client receives an acceptable QoS for each service invocation. The service composition technique, as explicated in this dissertation, yields favourable performance metrics in terms of time taken to fulfil service invocations on the service host. Additionally, evaluations of a service built using the slice-oriented programming technique show reduced levels of complexity when compared to alternative approaches, easing service maintainability.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

The Internet of Things (IoT) is both an exciting and incipient technological shift that is set to change human interaction with the physical world. The IoT is a term that loosely describes inanimate objects, comprising embedded devices, which can interconnect via wireless networks [2]. There are several fundamental factors that can be attributed to the advent of the IoT, which are, the use of the Internet as a communications platform, more affordable wireless technology, and better processing power for embedded devices [3]. The collective culmination of these factors yields greater potential for web service applications within the realms of information, analysis, automation and control.

The Web of Things (WoT) is seen as the successor to the IoT. The primary objective of the WoT is to integrate inanimate objects into the web as resources which can be accessed and manipulated on demand [4]. In order to achieve this objective, there exists the need for a platform independent architecture to facilitate inter-device communication [5]. A service-oriented architecture (SOA) is the most auspicious framework to allow devices to integrate into the WoT, as it provides a communications platform whereby an inanimate object can be accessed in a platform-independent manner.

1.2 Motivation

The platform-neutrality of SOA applications ensures that potentially disparate devices can exchange information irrespective of their underlying hardware specifications. However, embedded devices may be characterised by a resource-constrained hardware profile. Furthermore, environmental characteristics such as location can have an impact on operational factors such as network availability. When a programmer is building an SOA application, the service logic should be developed with these constraints in mind. In doing so, the application can be tailored to facilitate an eclectic range of clients with different operational profiles. In doing so, it is envisaged that a better application Quality of Service (QoS) can be delivered to the client device.

Tailoring the application will require the programmer to build different compositions of the underlying application logic, appropriate to the different client operational contexts. The use of different object-oriented creational patterns can allow the service logic to be composed for different client profiles. However, these creational patterns are not particularly flexible when recomposing the service logic for 2 reasons. First, additional dependencies are unduly introduced and secondly, additional source code modifications are required. These additional dependencies can increase the complexity of the underlying application logic.

1.3 Research Question

The research question that this dissertation aims to answer is as follows:
Can a higher level of programming abstraction of the build-time service logic composition process, provide greater flexibility in delivering an acceptable application QoS for the client device, without detrimentally affecting the maintainability of the service?

1.4 Research aims

A programming model is proposed which allows a service programmer to specify QoS-related compositions of a service's logic at build-time. This is to ensure that the service client can benefit from an acceptable application QoS at run time. However, this programming model must be flexible enough to allow the service logic to be recomposed for changing requirements over time. Consequently, the research aims are as follows:

- Develop a prototype programming methodology which will provide the programmer with the necessary provisions to specify and generate QoS-related compositions of the service logic.
- Build a real, fully-functional service using the proposed methodology.
- Build a test client application for a resource-constrained device which can invoke the service under different simulated operational contexts.
- Evaluate the service with a focus on
  - perceived application QoS from a client perspective
  - overall service performance
  - resulting service complexity

1.5 Outline

This dissertation is organised as follows

Chapter 2 explores a state of the art in mobile context-awareness and program slicing.

Chapter 3 delves into the design requirements for developing a prototype of the programming model. Design decisions are also discussed in this chapter.

Chapter 4 covers the steps required to implement the requirements proposed in the design chapter.

Chapter 5 details the methodology for evaluating the proposed programming model. Application QoS, performance and complexity are then evaluated with regard to a service developed using the proposed programming model.

Chapter 6 concludes this dissertation and discusses future work and improvements.

Chapter 2

State of the Art

2.1 Mobile computing context awareness

The inception of context-aware computing in 1994 was defined as the ability of a mobile application to be aware of its operational environment and to dynamically adapt itself to the environment accordingly [6]. Since the genesis of context-aware computing, this quintessential philosophy has remained fast.

In terms of mobile computing, context can be described as

- the state of the underlying hardware resources within the device
- the environment in which the device is located
- user preferences

Mundane examples of hardware resources within a device that would be of interest to context-aware applications would be

- battery power
- cpu
- dynamic memory

- hard disk capacity

The quintessential idea of context-aware applications is that they can adapt themselves according to a device's changing context, in order to maintain an acceptable QoS for the user. Consider the case of a user who is using a mobile application which requires a 3G network connection, at a minimum, in order to download information from a service. Hypothetically, the user enters a locality where 3G is no longer available. When such an event arises, there may be a performance hit which impedes the end user's experience of the application. Such a scenario exemplifies the need for applications that can in some way adapt themselves with regard to the vagaries of a mobile execution environment. Context-aware middleware is a solution to this problem, and a variety of context-aware middleware will be examined in this section.

2.1.1 Semantic service-oriented context-aware middleware

A service-oriented context-aware middleware is proposed in [7]. In this approach, contexts are represented as Web Ontology Language (OWL) predicates, thus allowing contexts to be semantically reasoned. The proposed architecture comprises 3 layers. The lower layer consists of context providers. These providers allow context garnered from cross-domain embedded devices to be abstracted for upper layers. The middle layer houses a context interpreter which provides the abstraction for lower layer contexts. This context interpreter is also connected to a knowledge base, which means that contexts can be added, queried, updated or deleted on the fly. Mobile services comprise the upper layer, and they can interpret contexts and adapt their behaviour accordingly. Also, the programmer can define rules for the mobile services that are fired when certain contexts change at run time. This is similar to the slice-oriented approach in that different compositions of the service can be defined at build-time; however the compositions are static which means that there is no run time adaptation. Additionally, the semantic-oriented middleware allows contexts to be specified at run time, whereas the slice-oriented contexts are statically defined. In

other words, the service must be stopped in order to update any context-related logic. However, the semantic-oriented approach requires a gateway for the context reasoning process, in order to relieve the embedded device of the context reasoning process. The slice-oriented approach does not require any sort of gateway, as the context-reasoning process is very simple and does not incur considerable overhead.

2.1.2 JCAF

The Java Context Awareness Framework (JCAF) [8] presents an extensible framework for building context-aware applications which have an experimental scope. In a similar manner to that of [7], the scope of JCAF is an event-based, service-oriented infrastructure. The run time infrastructure of JCAF comprises distributed context services which are arranged in a peer-to-peer fashion. Each of these services may interact in order to obtain context-specific information. The context information is stored within *entities* that reside within an *entity container* located in a context service. Context clients may subscribe to an entity's context via the context container, and changes to that particular entity's context can be propagated, as events, to all the registered listeners. Within the scope of this dissertation, a service provider need only be aware of a client's context at invocation time and therefore does not need to retain any context-related state for the client.

2.1.3 VOLARE

Within a service-oriented environment, context-aware middleware may be utilised to renegotiate web service QoS levels, based on a device's context. If a suitable QoS level cannot be renegotiated, an alternative web service binding may ensue. VOLARE [9] is a mobile middleware that does just that. During the run time operation of an application, VOLARE can infer the context of the mobile device in order to determine if it has surpassed or fallen below a certain threshold. Based on existing service bindings, VOLARE can then request an adaptation of the QoS levels for these bindings. Additionally, during

the service discovery phase, VOLARE can intercept a service request, examine the context of the device, and subsequently adapt service discovery based upon the application's QoS requirements. These application QoS requirements are specified using an adaptation policy language. VOLARE's service discovery phase introduces an intermediary dependency between the client device and a QoS broker. This means that the service request must be forwarded to a QoS broker first, which will allow the middleware to select an appropriate web service based on its QoS requirements [10]. The slice-oriented approach does not require a service broker because it is concerned with application QoS as opposed to conventional web service QoS, such as reliability, scalability, capacity [11] etc.

2.1.4 CARISMA

Reflection [12] within programming languages can be described as the ability of a program to observe and reason about its own state (introspection) as well as act upon this information to manipulate its execution state, interpretation or meaning (intercession) [13]. The CARISMA middleware [14] makes use of reflection for dynamic application adaptation based on context awareness. In a similar nature to slice-oriented programming, CARISMA is based on the idea that different applications require services to be delivered in accordance with the device's context. With CARISMA, this is achieved by correlating a profile with each application on the client device. At run time, when an application requests a service operation, the application's profile is sent to the CARISMA middleware. The middleware subsequently examines the underlying contexts that relate to the service in that profile, and then makes an informed decision as to which policy to apply for that service. For example, consider a service that delivers news feeds to an application running on a client device. With CARISMA, the news feed application on the client device will have a profile that contains an entry for the news feed service. Within this profile, there can be 2 policies that can be applied within the middleware in order to determine how the news feed service is delivered. When a service request is made by

an application on the device, the context configurations for that application are used by the middleware to query the underlying hardware resources. The contexts are then evaluated against the application profile in order to determine which service policy is applied. CARISMA exposes a meta-application programming interface (API), that ultimately allows an application profile to be inspected or tailored by an end user. This means that by virtue of reflection, the CARISMA middleware can be tailored on a per-application basis to alter how a service is delivered. This is the same principal as the slice-oriented approach. However, the difference is that the customisation of the application is performed server-side, at build-time.

A drawback to CARISMA is that a service may only be delivered using 1 policy at a time, and the amalgamation of multiple policies is prohibited. Moreover, the number of policies that can be applied to a profile is limited to a maximum of 10, as the overhead incurred by CARISMA increases linearly based upon the number of policies in an application profile. There is no limit to the number of profiles that can applied to a service developed using the slice-oriented approach. Also, this dissertation does not use run time adaptation, but rather, static compositions of the service which are defined at build-time. Finally, there is no reflection at play in this dissertation, meaning that the typical inherent overheads of using such a technique are absent.

2.1.5 mobiPADs

Reflection is also a key factor in the mobiPADs [15] context-aware service execution platform, which allows mobile services to be deployed within a wireless environment. The services are deployed as *mobilets*, which are chained service objects representing a configuration, which can be updated when the device's context changes. The fundamental idea of a mobilet is comparable to a Java applet[1], whereby processing is shared between a client residing on the mobile device, and a server connected to the internet.

Context-awareness is achieved by an extensive event model, where event source objects

[1] http://java.sun.com/applets/

allow event listeners to be registered, removed and notified, in a similar manner to that of JCAF. Event source objects will monitor underlying context, for example bandwidth and cpu load. When a change is detected in these contexts, the relevant event source objects will notify registered event listeners, which will trigger application adaptation in one of two ways. The first is a reconfiguration of the service chain by adding or removing mobilets. The second approach, which is more granular, involves relaying the context updates to each mobilet in the service chain, which will then allow them to readjust.

A disadvantage of sharing processing between the mobilet on the client and the mobilet on the server, is that the client must have the mobilet code on board. This means that if the server does not possess mobilets that are part of a service configuration, the client must push them to the server. Thus for each change in the device's context, there exists the possibility of the client having to upload a mobilet to the server, which then needs to be deployed. This means service interruption, and also introduces a tight coupling between the configuration on the client device, and the configuration on the server.

2.1.6 MADAM

The MADAM middleware [16] uses a different approach to context-aware application adaptation. MADAM introduces the idea of *planning* within context-aware middleware. Planning describes the run time process of both selecting and evaluating different compositions (known as *plans*) of the application's component framework, in order to determine which one will yield the highest *utility* for the end user. The utility is a measure of the expected degree of end-user fulfillment, and is calculated using *property predictor* functions for each component. This is a composite function which yields values indicating how the application QoS will be affected by assuming a particular composition.

2.1.7 MUSIC

The MUSIC middleware [17], which is an extension to MADAM, uses the same approach to context-aware adaptation with the exception that interchangeable support is provided for services as part of the planning process. This means that when there is a context change on a device, the MUSIC middleware can incorporate remote services as part of the updated application configuration. In light of this, the objective of MUSIC is to facilitate support for the operational vagaries associated with remote service providers. When a service is discovered by MUSIC, it will attempt to negotiate a reasonable service level agreement (SLA) with the service. This SLA agreement is then stored for later use in the planning process. Conversely, the service provider QoS can be negotiated during the planning process, as long as the SLA with the service provider is itself dynamic.

No empirical performance metrics are published to elucidate what type of overhead is associated with the MADAM and MUSIC component-based planning frameworks. However, it is safe to postulate that some sort of performance overhead must exist by virtue of the different plans, including their property predictor and utility functions, that must be evaluated during a change in operational context. The dynamic service QoS negotiation capabilities of the MUSIC middleware will also add to the total cost of the overhead associated with the planning process. The solution outlined in this dissertation will not support dynamic negotiation, nor the advertisement of SLAs. However, it will provision the programmer with a domain specific language (DSL), which will allow the specification of ranges of QoS that are implicitly catered for.

2.2 Program slicing

Program slicing [18] is a term that was first coined by Mark Weiser in 1981. Weiser proposed a technique of *slicing* a program, which entails decomposing a program into a minimal independent form of the original source. This decomposed subset of the original source code is known as a *program slice*. A *slicing criterion* represents a point of interest

in the program such that a slice can be calculated. A criterion C in a program P can be defined as

$$C =< i, V >$$

where i is a point in the program P, and V is a subset of variables in P. When a program slice is calculated according to the slicing criterion C, the slice will comprise all of the constituent parts of the program P that *may* affect the set of variables V. Slicing a program according to its slicing criterion, is referred to as static program slicing. However, dynamic program slicing [19] is a way of slicing a program according to the parts of a program that *do* affect the subset variables V, for a particular criterion. It is this decompositional nature of program slicing that resonates with the slice-oriented methodology outlined in this dissertation.

The construction of program dependence graphs are a popular technique for calculating a program slice. The program dependence graph [20], which was originally pitched as a means for compiler optimisation, was also proposed as a useful technique for performing intra-procedural program slicing. The dependence graph models statements and control predicates as vertices, while data and control dependencies are modelled as edges, as shown in figure 2.1.

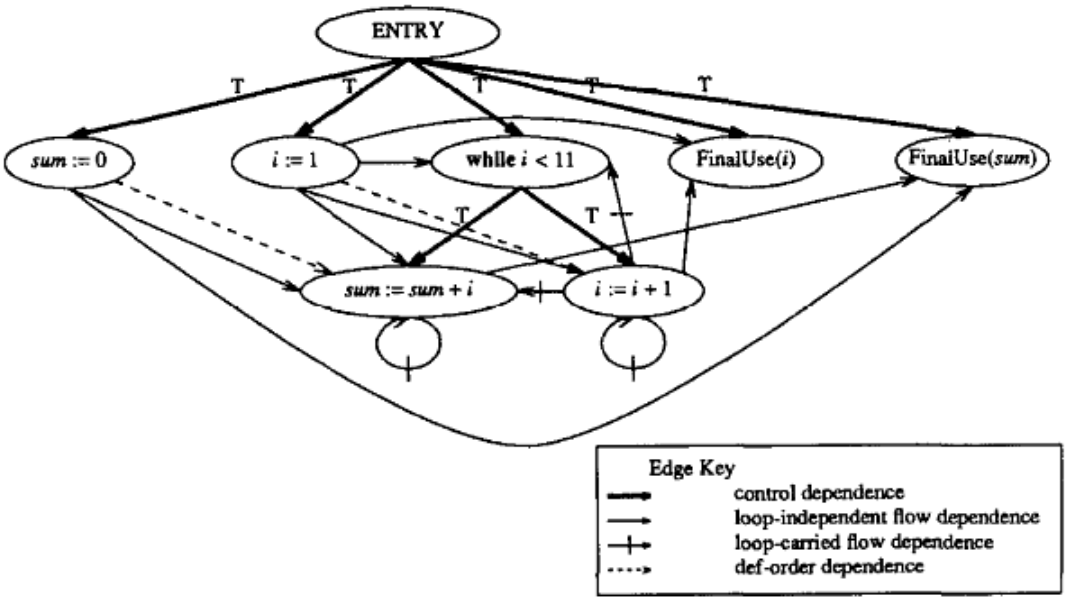

Figure 2.1: Example of a program dependence graph [1]

Slices are calculated by choosing a vertice and traversing the dependence graph backwards, which is also known as calculating vertex reachability. The system dependence graph [1] extends the program dependence graph to facilitate inter-procedural slicing, and has been extended to support object-oriented programs as well [21, 22, 23, 24, 25]. The difference between intraprocedural-slicing and interprocedural-slicing, is that interprocedural-slicing allows program slices to be calculated that cross procedure boundaries. Figure 2.2 shows an example of a system dependence graph, which is a collection of program dependence graphs connected by both call and parameter edges. Call edges connect method call vertices with method entry vertices, while parameter edges represent the data flow between method parameters and method arguments. The problem however, is that slices calculated from a system dependence graph are not executable [26]. This dissertation requires the calculation of executable slices, and as such requires a more bespoke dependency graph.

There are many applications of program slicing such as code maintenance, parallel computing, code comprehension and testing [27]. Another application of program slicing is a scenario-oriented program slicing technique [28]. This approach slices programs under

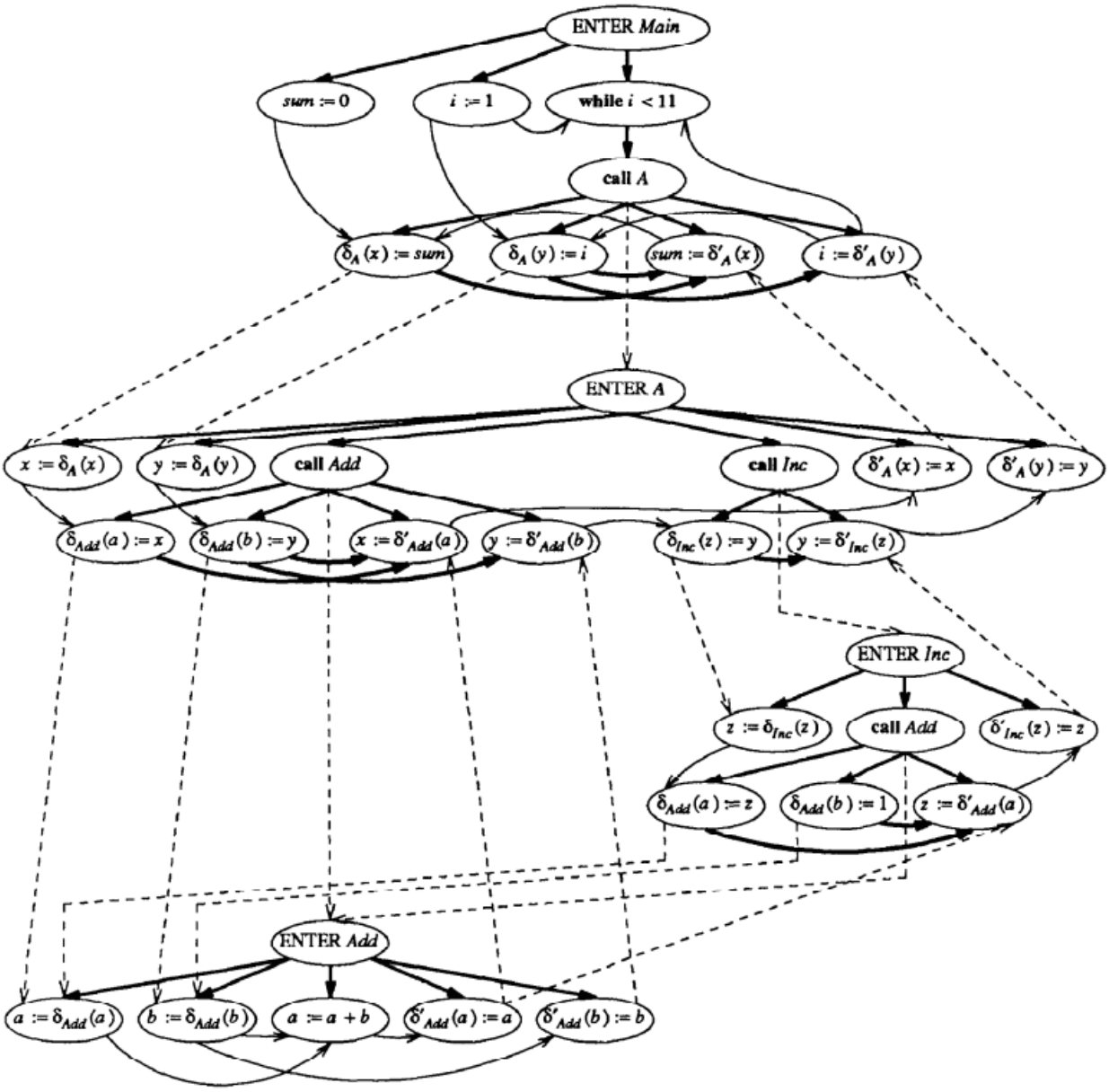

Figure 2.2: Example of a system dependence graph [1]. Control edges are shown in bold, and arcs represent intraprocedural flow dependencies. Parameter-in, parameter-out and call edges are represented by dashed lines

different execution scenarios, where a scenario is represented as a UML sequence diagram. The sequence diagrams are transposed into more abstract entities, from which slices can be derived using the following formula

$$\xi =< S, T, X >$$

where S denotes the scope where a scenario is actually reified. T is the trace, which symbolises all of the method calls and method returns that are executed in the program. The methods included in a trace will be determined by the dependencies within the enclosing code blocks, which contain control predicates. The corollary is that only certain methods will need to be called, whereas other methods within the sequence will be excluded. These excluded methods are denoted by the set X. This approach bears similarity to the slice-oriented methodology in that code relevant to a scenario needs to be identified before slicing occurs. With the slice-oriented methodology, a criteria represents certain client context conditions which must be true in order for certain code to be executed. These criteria must be identified before slicing occurs. The scenario-oriented slicing methodology however, requires more intermediate steps before slicing can occur.

Another similar application of program slicing, which inspires this dissertation, uses XML tags instead of UML diagrams to identify parts of a program that are associated with a particular scenario [29]. Using this approach allows 2 types of slices to be generated. The first type is a scenario-specific slice, which identifies source code that is unique to a specified scenario id. The second type of slice identifies a compilable scenario-specific slice. The authors developed a macro for the Visual Studio 2005 IDE which allows the insertion of XML tags at different points in the source code, and the tags have different scope depending on where they are incorporated. Instead of preceding methods with comment-based XML tags, the slice-oriented methodology will use Java annotations. Also, for a scenario-specific slice, the authors do not indicate exactly how externally dependent code is determined, in order to make the scenario slice compilable; there is no mention of the application of any sort of dependence graph. Furthermore, making code scenario-specific

requires the inclusion of XML tags before *all* code that is relevant for the scenario. This includes methods and class variables, and could result in code that is very cluttered and difficult to comprehend. The slice-oriented approach only requires that methods be annotated with a single annotation, containing only 1 associated element.

2.3 Summary

This chapter has conveyed the seemingly inextricable link between context-awareness and application adaptation, in order to ensure that the end user's experience is impeded in as minimal-a-manner as possible. The different approaches that have been explored in this chapter show how context-aware middleware can perform application adaptation [14], service adaptation [7] and service binding with QoS negotiation [9, 17]. The use of reflection is a popular technique for the introspection of run time configurations [14, 15] in order to adapt an application effectively. However, leveraging reflection for software adaptation requires careful consideration. For example, the use of reflection for intercepting method calls has the propensity to incur a considerable overhead [30]. Although the VOLARE middleware intercepts service requests, it is not specified exactly how this is done. Notwithstanding, the overhead associated with operation interception reflection would not bode well for a resource-constrained device. The use of a QoS broker also featured in [9], which is not an ideal solution for use in ubiquitous computing due to the vagaries of network connectivity. The slice-oriented methodology will relieve the client device of application adaptation; it will only require the client to garner different context properties from its underlying operating system, which will be sent as a serialised object to the service.

Examples of program slicing have shown that slicing is a useful technique for decomposing a program in accordance with abstract scenarios [28, 29]. A scenario is a rather subjective term, which means that it can be extended to fit the slicing technique within this dissertation. As mentioned, the slice-oriented methodology requires executable slices;

system dependence graphs will not suffice as executable slices cannot be generated. The result is that a more bespoke dependence graph will be required in order to generate executable program slices.

Chapter 3

Design

3.1 Requirements

The design of the slice-oriented programming model is discussed in this chapter, and is closely aligned with the research aims of this dissertation. The generation of the web service slices will comprise a 2 stage process, namely functional decomposition and program slicing. In order to reify the slice-oriented programming model, there are 5 core requirements to consider in the design phase. These are:

- The design of a DSL grammar
- The design of a parser for parsing a DSL instance
- The design of a functional decomposition engine in order to decompose service functionality, according to a DSL instance
- The design of a dependence graph required for slicing
- The design of a program slicer for further decomposing source code

The implementation of these requirements will result in a synthesised programming-tool, which will assist the programmer in generating the necessary QoS-related compositions

at build-time, after the core service functionality has been fully implemented. These requirements will be explored in further detail in this chapter.

3.1.1 Domain-specific language

The benefit of web services is that they are innately platform independent. This means that applications can be represented in a uniform manner, within an environment characterised by hardware heterogeneity [31]. However, it is important that the SOA application also account for hardware heterogeneity from a QoS perspective. This means that the application should be inherently *open* in so far that it can be recomposed to facilitate clients of varying hardware heterogeneity over time [32]. By using a composition language, the level of abstraction of the service logic composition can be raised from the use of programming and inheritance, to the explicit composition of the service's software components [33].

A DSL allows this higher level of abstraction to be modeled such that at both build and maintenance-time, it is not incumbent on the programmer to make any code modifications, in order to adjust the service logic composition. Before proceeding further, it is necessary to discern the traditional meaning of service composition with that used in this dissertation. Within an SOA environment, service composition relates to *composite* service composition, involving the interaction of a multitude of web services via a web service orchestration language. In this dissertation, service composition relates to the composition of an application's constituent software components, within a stand-alone web service's application logic. Using a DSL to raise the level of abstraction for the service programmer exhibits additional benefits. For example, programmer productivity is increased by virtue of the ease of understandability of the DSL, as well as the auto-generation of code after the DSL is transposed to a programming language [34].

The DSL required for the specification of the slice-oriented compositions will comprise 3 sections as shown in figure 3.1. The first section of the DSL will allow the programmer

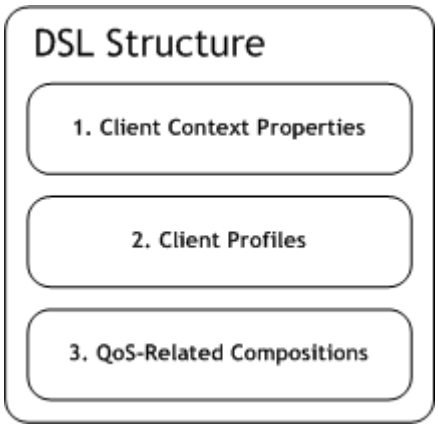

Figure 3.1: Proposed DSL structure

to define context properties. These properties will relate to the information that must be garnered from a client device at service invocation time, in order to execute the appropriate service slice. After the service programmer defines the client context properties, a class will be automatically generated that will encapsulate these properties. This class will be serialisable, and will be sent from a service client to the service as part of the invocation. The benefit is that this serialisable object can be advertised in the service's web services description language (WSDL) file. The second section of the DSL will use these context properties to allow the programmer to semantically categorise ranges of these context properties. These categorisations will be referred to hereafter as *client profiles*. An example of a client profile could be a *WeakClient* profile. This particular profile would encompass a modest range of values for the client context properties defined in the DSL. Listing 3.1 shows an example of how a *WeakClient* would be defined in the DSL.

Listing 3.1: Example definition of a "WeakClient" client profile

```
1 definitions {
2    WeakClient {
3       cpu < 120, //120 Mhz
4       ram < 10,  //10 MB
5    }
6 }
```

The definitions of the client profiles will be wrapped in a struct entitled *definitions*. The programmer will be able to auto-generate a class that can bind, or return, an executable slice at run time, based upon the profile of the client making a service request. This class will be referenced hereafter as a *SliceBinder*. The third and final section of the DSL allows the service programmer to define the functional decompositions, from which the final executable slices will be generated. Figure 3.2 shows the linking of client profiles to the underlying service logic, *before* slicing occurs.

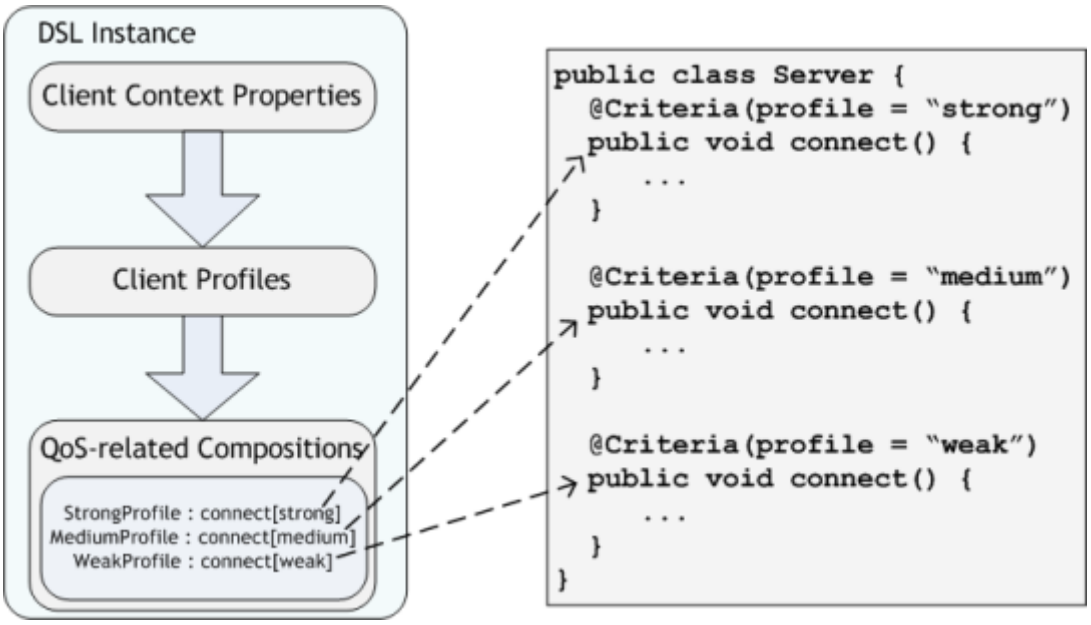

Figure 3.2: QoS-related compositions section of dsl links client profiles with underlying service logic. Note that this diagram shows service code that has not yet been sliced

3.1.2 Domain-specific language parser

After the programmer has finalised the DSL instance for the compositions that are to be generated, the DSL instance will need to be transposed to a format that can be understood by the functional decomposition engine. This will require accessing an in-memory structural representation of the DSL, so that the relevant information can be extracted. An XML format of the DSL instance would be the most judicious choice, as an XML document is easily comprehended, extensible and platform-neutral.

3.1.3 Functional decomposition

The functional decomposition stage will be the first of the 2 step process for generating the final web service slices. Several prerequisites must first be addressed in order for the development of the decomposition engine to succeed. First, the decomposition engine will need to make copies of source code which is to be decomposed. Secondly, the decomposition engine will require that the programmer place markers in the source code to flag functionality that is QoS-centric. These markers will be used to correlate functionality with specific client profiles, through the QoS-related compositions section of the DSL, as shown in figure 3.2. The programmer will avail of *Criteria* annotations which can precede QoS-related methods in the underlying service source code. A criteria annotation will have 1 element, entitled *profile*, which serves as a handle for referencing these methods from the DSL. Methods annotated with a criteria annotation will be known hereafter as *criteria methods*.

In order to perform the actual decompositions, the decomposition engine will need functionality to parse the service source code such that criteria methods can be removed according to the QoS-related compositions in the DSL. Before all of this can happen however, the composition XML file, generated by the DSL parser, will need to be read into memory so that the decomposition engine can infer the decompositions it needs to perform. The resulting code from this stage will then need to be sliced.

3.1.4 Dependency graph generation

A member dependence graph will need to be constructed for each method in the decomposed source code, and collectively, these graphs will represent the class member dependencies for each method. The rationale for this step is explicated with an example.

Listing 3.2: Simple FileUploader class with criteria methods

```
1  public class FileUploader {
2     private static final int compressionLevel = 3;
3
4     @Criteria(profile = "powerful")
5     public boolean uploadFile(File file) {
6        // upload the file using wifi connection...
7     }
8
9     @Criteria(profile = "average")
10    public boolean uploadFile(File file) {
11       File compressedFile = compressFile(file);
12       // upload the file using 3G connection...
13    }
14
15    private File compressFile(File f) {
16       // compress the file to reduce size
17       // according to the "compressionLevel" variable
18    }
19 }
```

Consider the *FileUploader* example class shown in listing 3.2, which contains 2 criteria methods for uploading a file to a repository. Supposing the decomposition engine, as part of the decomposition process, has to remove the *uploadFile* method annotated with the *average* criteria. When this method is removed, the *compressFile* method, which was only used by the now-deleted uploadFile method, still remains in the class. Additionally, the *compressionLevel* class variable, which is only required by the compressFile method, is still present. This situation is undesirable for several reasons

- The FileUploader class now contains redundant code which is not required by any API related functionality
- This redundant code may present an ambiguous situation for the programmer when attempting to comprehend the code
- The number of lines of code is unduly increased by the presence of the redundant class members

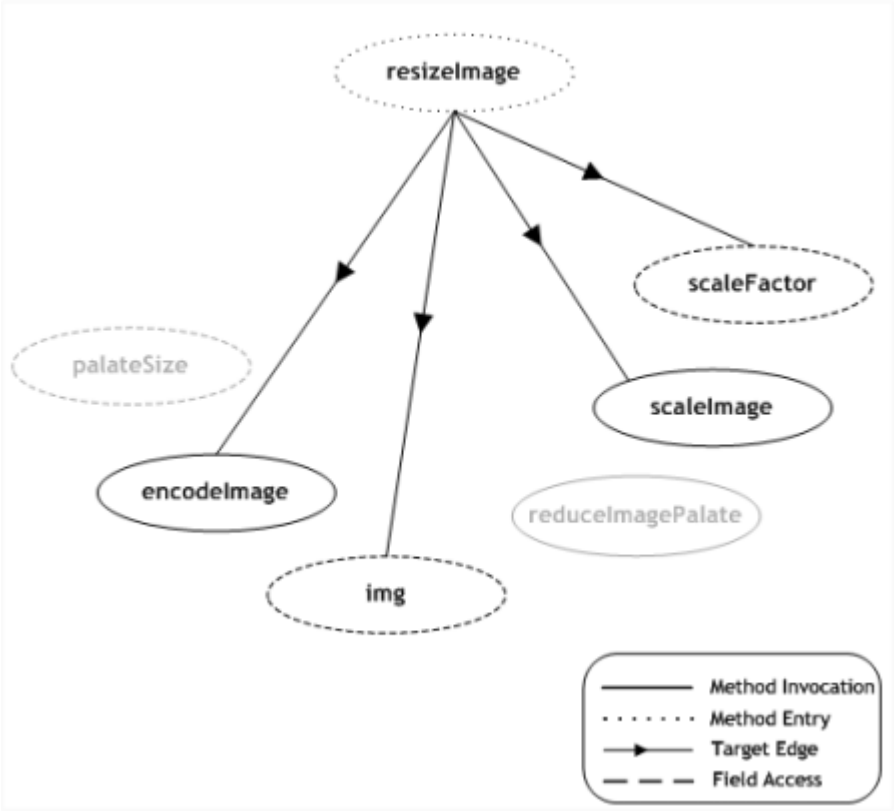

Figure 3.3: An example member dependence graph for a method entitled *resizeImage*. The graph shows the different types of dependencies that can exist between a method, and other class members. The members that are shaded out, are other class members which are not dependees of the resizeImage method. Local members are not included in the member dependence graph.

As a result, the complexity of the FileUploader class will be higher than if the compressFile method were not present. The member dependence graph will be used to infer the dependencies that exist between remaining class members, after a class has been decomposed by the decomposition engine. These dependencies will be represented by target edges. A class member can be either a method or a non-local variable, and will be represented as a vertex within the graph. Figure 3.3 shows a sample member dependence graph.

3.1.5 Program slicer

The program slicer is the 2nd step, of the 2 stage process, for generating executable slices from the web service logic. The program slicer will further decompose source code that has been output from the decomposition engine. It will request the generation of a member dependence graph for each method, and will then use this graph to calculate the executable slices. Recall that vertices in the member dependence graph represent variables and methods, and edges are indicative of dependencies between these class members. The slicing will thus be a graph reachability problem. For example, for a given graph G, the reachability R, of vertex j from vertex i, is defined as

$$R(i, j) = 1$$

if j can be reached from i. Otherwise

$$R(i, j) = 0$$

if j cannot be reached from i. This will allow redundant code elements to be removed from the class and will result in the final executable slice. One crucial point here is that only private class members are sliced out of the source code. Public members are not sliced out, as doing so could break the class API. It is therefore assumed that the service programmer can employ good encapsulation where necessary.

3.2 Design decisions

Each of the build-time generated slices is calculated from a class written by the programmer. In a way, the slices can be *perceived* as children of the class from which they are derived. In reality however, slices are not subclasses of the class from which they are calculated, as they do not extend it. Therefore, the slices cannot be used in a polymorphic manner without the introduction of some sort of interface. For example, consider a class *ServerFileManager* that contains a *sendFile* method as shown in listing 3.3. An

executable ServerFileManager slice, for file transfer using TCP/IP over a wifi connection, is shown in listing 3.4. Similarly, an executable ServerFileManager slice for file transfer using (UDP) over a 3G connection, is shown in listing 3.5.

Listing 3.3: An example class for transferring files to a server

```
1  public class ServerFileManager {
2     private Socket tcpConn;
3     private DatagramSocket udpConn;
4
5     @Criteria(profile = "wifi")
6     public void sendFile(File file) {
7        tcpConn = new Socket(...);
8        ObjectOutputStream oos =
9                new ObjectOutputStream(tcpConn.getOutputStream());
10       oos.write(file);
11    }
12
13    @Criteria(profile = "3G")
14    public void sendFile(File file) {
15       udpConn = new DatagramSocket(...);
16       byte[] fileBytes = getBytes(file);
17       DatagramPacket packet = new DatagramPacket(fileBytes);
18       udpConn.send(fileBytes);
19    }
20
21    private byte[] getBytes(File file) {
22       byte[] bytes = new byte[(int)file.length()];
23       // get the bytes for the file...
24       return bytes;
25    }
26 }
```

Listing 3.4: ServerFileManager slice for using TCP/IP to transfer a file using wifi

```
1  public class ServerFileManager {
2     private Socket tcpConn;
3
```

```
4     public void sendFile(File file) {
5        tcpConn = new Socket(...);
6        ObjectOutputStream oos =
7                new ObjectOutputStream(tcpConn.getOutputStream());
8        oos.write(file);
9     }
10 }
```

Listing 3.5: ServerFileManager slice for file transfer using UDP with a 3G connection

```
1  public class ServerFileManager {
2     private DatagramSocket udpConn;
3
4     public void sendFile(File file) {
5        udpConn = new DatagramSocket(...);
6        byte[] fileBytes = getBytes(file);
7        DatagramPacket packet = new DatagramPacket(fileBytes);
8        udpConn.send(fileBytes);
9     }
10
11    private byte[] getBytes(File file) {
12       byte[] bytes = new byte[(int)file.length()];
13       // get the bytes for the file...
14       return bytes;
15    }
16 }
```

Supposing that a ServerFileManager client class wishes to invoke the sendFile method. Imagine that this client is characterised by a 3G connection, as specified in the DSL. In this particular case, the selected slice should be the ServerFileManager slice for 3G clients, as shown in listing 3.5. Recall that the responsibility of selecting the correct slice to execute for a client lies with the SliceBinder. The SliceBinder will know what slice to select, as it contains the necessary conditional predicates as defined in the client profile section of the DSL. However, the issue here is that the client shouldn't care what slice it receives from the SliceBinder. All that it should worry about is that it has a ServerFileManager instance. In other words, the slices that are returned to clients from

the SliceBinder should be polymorphic. In order to achieve polymorphism, it was decided that the programmer create only 1 interface for the class representing the entry point into the compositions generated according to the DSL. Slices that are generated from this class will also implement this interface, which means that polymorphism is achieved. Figure 3.4 shows a UML class diagram for the ServerFileManager and SliceBinder classes, as well as the generated slices. The SliceBinder would contain 1 or more *NetworkFileManager* instances, each of which references one of the ServerFileManager slices. A client would call the *getImplementation* method of the SliceBinder and pass in its operational profile in the form of a *ClientProfile* instance. This class contains the context properties as defined in the context properties section of the DSL. The SliceBinder will then return the slice which will give the best QoS for the client.

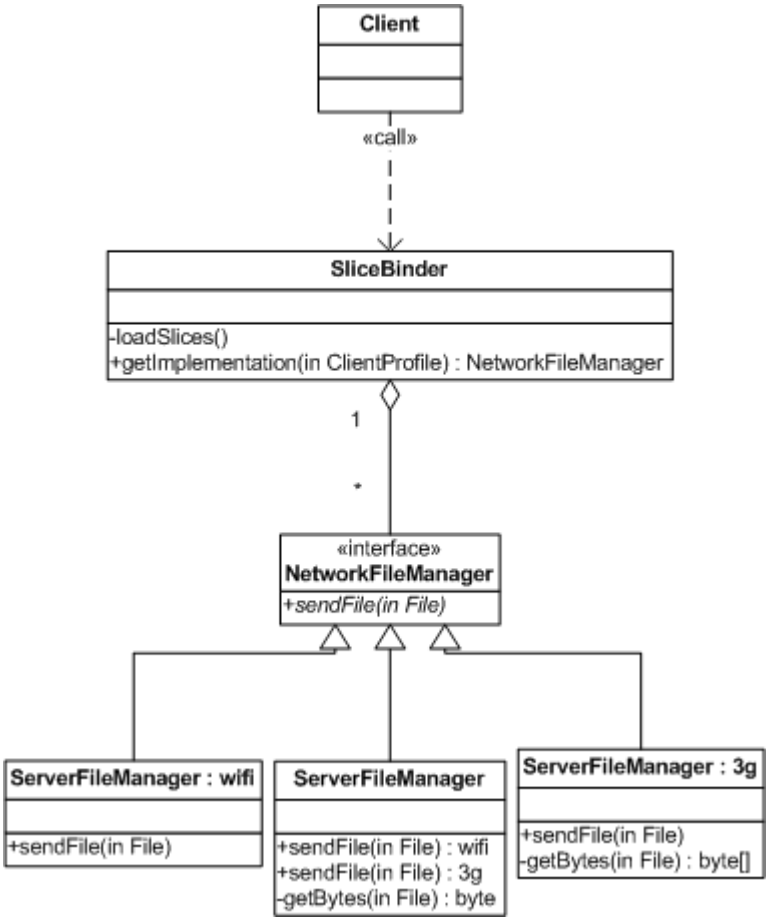

Figure 3.4: UML diagram showing the introduced interface *NetworkFileManager* to ensure that the *ServerFileManager* slices are polymorphic.

Another important decision which impacts the implementation, is where to place the generated slices. It was decided to place each generated slice into a package with a customised qualified name. This qualified name will be the same name as the enclosing package of the class from which the slice is calculated, except that it will also contain the name of the client profile with which this slice is associated. For example, assume the package name of the ServerFileManager class from listing 3.3 is *server.io*. The generated ServerFileManager slice for wifi clients will thus be placed into a package entitled *server.wifi.io*, while the ServerFileManager slice for 3g clients will be placed into a package with a qualified name of *server.3g.io*. The alternative to placing the slices into proprietary packages would be to rename the classes, as packages cannot contain classes with duplicate names. Using packages is a cleaner solution, as their purpose after all is to avoid naming conflicts.

Apart from automating the generation of the slices, it was also decided to automate the generation of the ClientProfile and SliceBinder classes, in order to make the programmer more productive. The information necessary to build these classes can be extracted from a DSL instance. Also, the decomposition and slicing processes will be coalesced into one operation, that the programmer can execute within an integrated development environment (IDE).

3.3 Programmer use case

Figure 3.5 shows a use case diagram providing an overview of the steps the programmer will take to generate the executable slices, from start to finish. The first step is to define the DSL instance. Then, the programmer can auto-generate the ClientProfile class, containing the context properties from the DSL. Next the programmer can generate the executable slices. Following this, the programmer will develop the entry point interface for the slices. The final step involves the programmer auto-generating the SliceBinder class, so that the executable slices can be returned to a caller.

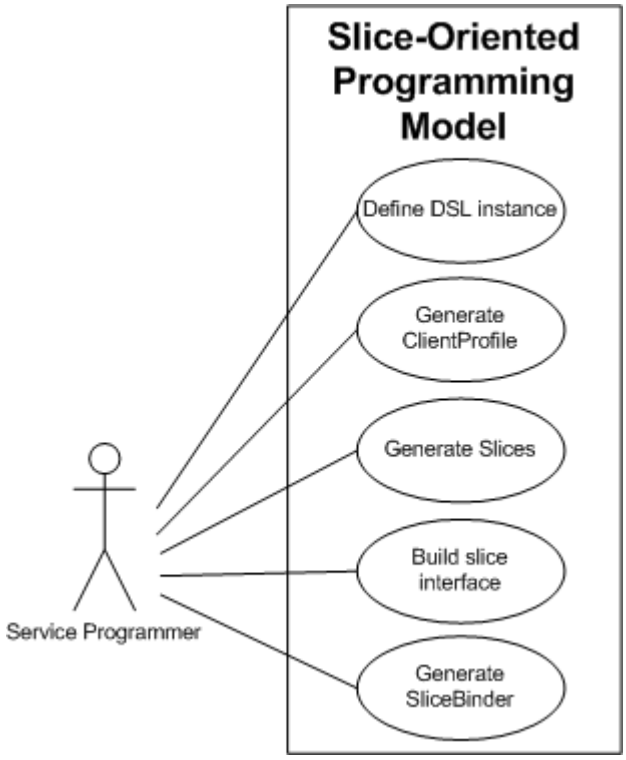

Figure 3.5: Slice-oriented programmer use case

Chapter 4

Implementation

This chapter will explore the implementation steps necessary to develop a prototype slice-oriented programming tool. The implementation details will be aligned with the design requirements presented in the previous design chapter. The chapter begins by exploring the technologies chosen for the implementation.

4.1 Chosen technologies

4.1.1 Programming language

The Java programming language was chosen as the language with which to develop the slice-oriented tool. It was also intended to support only Java projects for the generation of the executable slices. However, the tool could be extended to support other object-oriented programming languages that allow code to be annotated. The tool was developed using Java sdk version1.6.0_20[1]. The Java sdk version 1.6.0_26[2] was initially used to develop the tool, however there were issues using this version, which will be outlined later in this chapter.

[1]http://www.oracle.com/technetwork/java/javase/6u20-142805.html
[2]http://www.oracle.com/technetwork/java/javase/6u26releasenotes-401875.html

4.1.2 Environment

The Eclipse IDE[3] was chosen as the target environment in which the slice-oriented tool can be used, and the version used for development was Helios version 3.6.1. One of the real benefits of Eclipse is that it is an open and extensible environment by virtue of its plugin framework. This meant that a working prototype of the slice-oriented tool could be delivered as an eclipse plugin. Furthermore, an added benefit of using Eclipse was the availability of the multitude of native frameworks, which provide support for many different design requirements.

4.1.3 Domain specific language

The Xtext plugin[4] is one such native framework, which allows DSL grammars to be defined. The end product is a parser, a metamodel and a text editor for creating DSL instances, coalesced into a single project. Creating a DSL instance using the generated text editor provides a standard Eclipse outline view for the DSL, a compiler for flagging syntax errors, code completion functionality as well as a parser for interpreting the DSL instance. For any programmer familiar with Eclipse, creating a DSL instance via the generated text editor is both simple and intuitive.

4.1.4 DSL transformations

Xpand[5] is a statically typed template language that can be used to generate code from the Eclipse Modelling Framework (EMF) representation of a DSL. It supports the definition of templates for any languages, and as such, will allow a DSL instance to be transposed to an XML format. It also provides a comprehensive syntax for defining and expanding template blocks, control flow, file generation and much more.

[3]http://www.eclipse.org/
[4]http://www.eclipse.org/Xtext/
[5]http://www.eclipse.org/modeling/m2t/?project=xpand

4.1.5 Source code manipulation

Java Development Tooling[6] (JDT) is a framework which provides all the necessary plugins to allow a programmer to develop a Java program in Eclipse. Collectively, these plugins make up the Eclipse *Java Perspective*. Included in the JDT is an infrastructure known as *Core*[7], which comprises a set of classes that model Java programs and resources within Eclipse. By leveraging the JDT Core API, Java programs, and their corresponding resources, can be manipulated for specific purposes. This api, which is contained in the org.eclipse.jdt.core package, will provide the functionality required for building the decomposition engine. The Core API also contains functionality for representing Java source code as an abstract syntax tree (AST), which is contained in the org.eclipse.jdt.core.dom package. This will be used for constructing member dependence graphs in order to perform slicing.

4.1.6 Alternative technologies

Several slicing tools were considered for the implementation of the slicer, none of which were a suitable fit. As a result, the slicer for this dissertation was developed entirely from scratch. Some of the slicing tools that were examined were:

- JSlice[35] is a dynamic slicing tool used for code comprehension and debugging. This tool was of no practical use to this dissertation as the generated slices are not executable. Also, the slicer only works on Fedora versions 3 and 4. The slice-oriented prototype tool however, was developed on a machine running Windows 7.

- Kaveri[36] is an Eclipse plugin, which uses the Indus slicing framework[8] to provide a user interface (UI) for slicing Java programs. This seemed like a promising tool to extend for this dissertation, however there were 2 major problems. Kaveri only works within Eclipse version 3.1.2, and it requires Java version 1.4.2 to run. The

[6]http://help.eclipse.org/indigo/nav/3
[7]http://help.eclipse.org/indigo/topic/org.eclipse.jdt.doc.isv/guide/jdt_int_core.htm
[8]http://indus.projects.cis.ksu.edu/

slice-oriented prototype requires Java version 1.5 at a minimum, as it relies on the use of Java annotations.

- StaticSlicer[9] is another tool used for generating static slices from a Java program. Once again however, the generated slices are not executable.

The Soot[10] framework is a tool used to optimise or transform Java bytecode. Bytecode can be transformed into a typed 3-address representation, known as *Jimple*. The jimple code is a convenient way of performing byte code analysis, as there are only 15 different statements that need be interpreted. Also, there is an extensive api for performing different types of analysis on jimple representations. For example, the Soot API allows call graphs to be constructed for specific entry points in a program. In addition, data dependencies can be analysed for each of the methods in the call graph. However, this approach did not succeed for 2 reasons. The first reason was that the entry points needed for generating a call graph can only be specified for a class that contains a *main* method. Entry points cannot be defined for arbitrary methods in a class. Secondly, the overall quality of the API documentation for the Soot framework is very poor, which can hamper progress considerably. In the end, it was also decided to build the required dependence graph from scratch, using the org.eclipse.jdt.core.dom package.

4.2 Deployment

The slice-oriented prototype was developed as an Eclipse plugin that can be installed in any Eclipse Helios IDE. Deploying the prototype as a plugin was more beneficial as it has greater portability and ease of use for the service programmer. The plugin includes 3 actions that the service programmer can invoke, which are:

- *Generate ClientProfile*, generates a serialisable class enclosing the client context properties as defined in the DSL

[9] http://sourceforge.net/projects/someslice/
[10] http://www.sable.mcgill.ca/soot/

- *Generate slices*, performs functional decomposition and slices the resulting source code
- *Generate slice binder*, generates a binder for returning executable slices with regard to the client's operational context

Note that the *Generate slices* action includes both the functional decomposition and slicing steps together. Each of these actions are implemented in the plugin as commands, which are extension points defined within the plugin. A plugin extension point represents a point to which other plugins can contribute functionality. Commands represent functionality in an abstract manner, such that different implementations can be used by a single command. There are 3 requirements for implementing a plugin command, which are, the declaration of the command itself, a handler which defines the command's functionality, and the UI widget to which the command contributes. Declaring the command and its menu contribution are simple tasks that can be achieved using point and click functionality. The implementation of the command handler however, requires the declaration of a class which extends the abstract class *org.eclipse.ui.command.AbstractHandler*. This class contains a concrete method *execute*, which takes an *ExecutionEvent* instance as a run time argument. The run time ExecutionEvent instance encapsulates all the necessary information that the execute method needs. For example, the trigger that fired the execute method, and the application context at that point in time. There is an AbstractHandler concrete implementation for each of the 3 commands defined within the slice-oriented tool. Each command is placed in the package explorer menu by leveraging the *org.eclipse.ui.menus* extension point. This means that the programmer simply makes a right click to view the menu with the 3 commands.

4.3 DSL implementation

The DSL grammar for defining QoS-compositions is show in Appendix B. The grammar syntax that Xtext employs is very similar in nature to extended backus-naur form. At the heart of Xtext is EMF, which comprises a modelling and code generation framework. EMF can model a DSL grammar using the XML metadata interchange (XMI) standard. EMF's *ECore* meta-model can then reify the XMI instance as a series of Java classes. Xtext relies on the ECore meta-model to build an in-memory object graph, or AST, representing a DSL instance. An overview of the ECore meta-model is shown in figure 4.1.

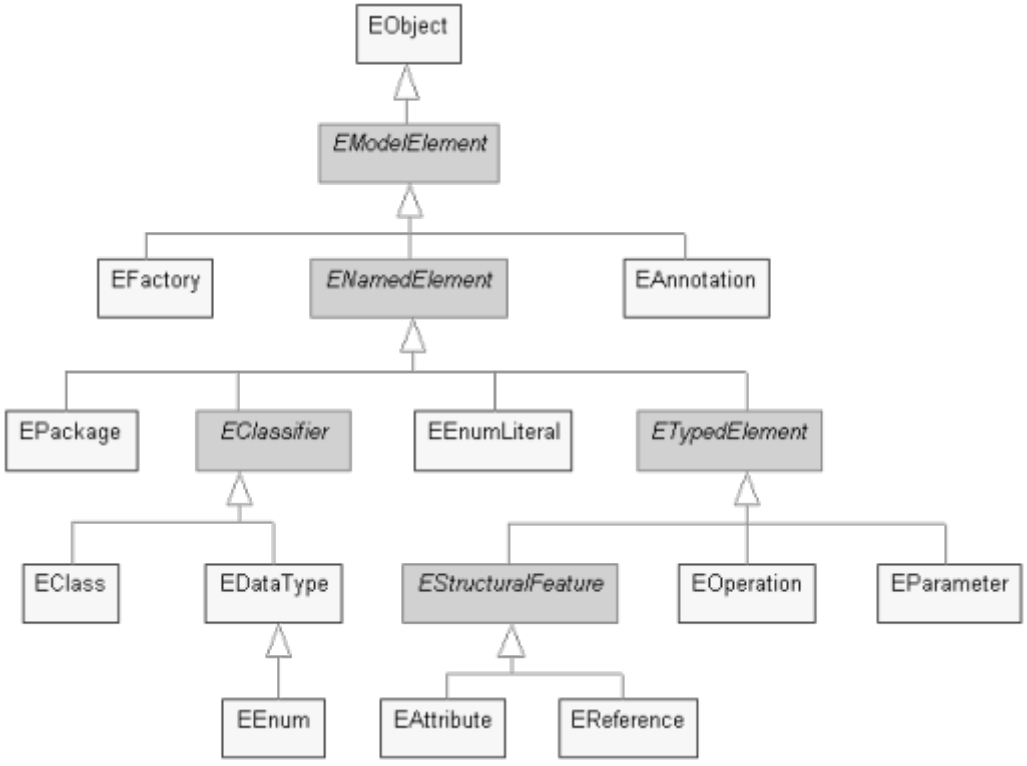

Figure 4.1: An overview of the EMF Ecore meta-model. The shaded classes represent abstract classes.

The most relevant classes to point out from the ECore model are

- EClass, represents a class which may contain attributes
- EObject, the base class from which all EClass classes are derived
- EAttribute, represents a class attribute which consists of a name and a type

- EDataType, used to represent classes that do not derive from the EObject base class
- EReference, used to associate EClasses with each other
- EPackage, represents a collection of EClasses

When defining the DSL grammar, 2 types of rules can be used, namely parser rules and terminal rules. When a DSL instance is parsed, the lexing phase will parse the text input into a series of tokens. These tokens are distinguished according to terminal rules defined in the DSL grammar. An example of a terminal rule from the slice-oriented DSL grammar is shown in listing 4.1.

Listing 4.1: Example of a terminal rule that builds *double* primitive data types from tokens received by the lexer

```
1 terminal DOUBLE :
2     INT '.' (INT)+
3 ;
```

A parser rule defines a rule whereby a parser tree can be constructed which comprises a series of tokens. In Xtext, these rules are used to build the AST which comprises a series of a EClasses. Listing 4.2 shows the second statement within the slice-oriented DSL grammar. This is a declaration of an EPackage, which will encapsulate all of the EClasses generated from the different parser rules within the DSL. It includes the name of the EPackage, which is *criteriaDsl*, and a uniform resource identifier (URI) for the package.

Listing 4.2: The EPackage declaration within the slice-oriented DSL grammar

```
1 generate criteriaDsl "http://www.xtext.org/slicing/CriteriaDsl"
```

The *Model* parser rule, as defined in the slice-oriented DSL, is classified as the entry rule for the DSL, as shown in listing 4.3. This rule contains a *projectName* attribute, which indicates the Java project where the service functionality resides.

Listing 4.3: Example of the *Model* entry rule for the slice-oriented DSL grammar

```
1 Model :
2     "project" projectName=STRING
3     (properties+=Property)+
4     (definitions+=Definition)+
5     (criteria+=Criterion)+
6 ;
```

The reason for allowing the programmer to specify the project name in the DSL is because there can be many projects in an Eclipse workspace, and the slice-oriented tool needs to know from which project to generate the slices. The *properties* attribute contains a list of *Property* objects, which represent the different client context properties that can be defined by the programmer. The *definitions* attribute references a series of *Definition* objects, each of which represents a client profile. Each client profile is characterised by programmer-defined conditions that must evaluate to *true*, in order for the presence of a particular type of client to be detected by the web service. The DSL prototype only allows 2 conditions per client profile, however this can be extended. Finally a list of *Criterion* objects, which represent the QoS-related compositions, is assigned to the *criteria* attribute in the Model object. The Property, Definition and Criterion parser rules are shown in Appendix B, while figure 4.2 shows an overview of the structure of *an instance* of the DSL grammar. Client context properties are defined in the *properties* struct, client profiles are defined in the *definitions* struct, while each QoS-related composition is defined in its own *Criterion* struct.

After the grammar has been defined, a series of compiler components can then be automatically generated for the slice-oriented DSL using a standard Xtext workflow. The components include a code formatter, an Eclipse outline view configuration, a content

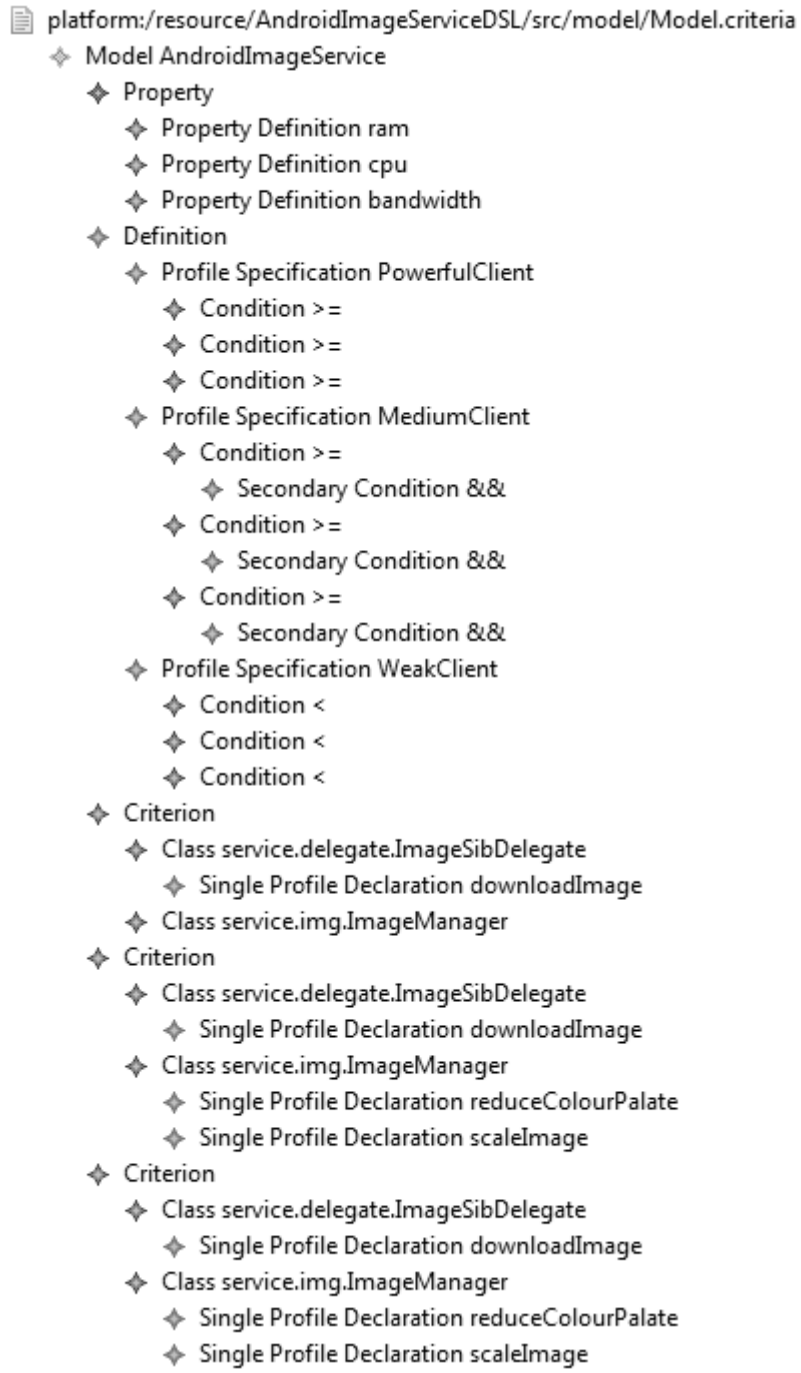

Figure 4.2: Example overview of the structure of a slice-oriented DSL instance

assist, a new-project wizard, a model validator, as well as EMF code for the generated models. This means, when defining a new DSL instance, the programmer gets real-time syntax checking and code assist, as well as an outline view of the DSL instance.

4.4 DSL transformation

Now that the DSL grammar has been defined, an instance of the DSL needs to be translated into a format suitable for the decomposition engine. During the design phase, it was decided that the decomposition engine would consume an XML format of a DSL instance, in order to perform the decompositions on the source code. This was a 2 step process. The first step was to define a template using Xpand, while the second step was to define an XML parser for the decomposition engine.

4.4.1 Xpand template

The benefit of Xpand, is that it can be used in conjunction with Xtext. It acts as a bridge by providing a level of abstraction for accessing run time objects generated from the DSL by EMF Ecore. The development of the Xpand template was straightforward by virtue of its intuitive syntax. XML tags are simply wrapped around the Xpand syntax in the template, in order for the final XML format to the defined. As mentioned in the DSL implementation section, one of the auto-generated artifacts from running the DSL workflow is a *new project wizard*, which allows programmers to create instances of the slice-oriented DSL. The DSL workflow also allows a dependency to be injected into DSL instances created using the new project wizard. This dependency can specify the project that contains the Xpand template necessary for translating a DSL instance into XML. This means that the service programmer will not have to explicitly redefine the necessary Xpand template for every slice-oriented DSL instance that is created. Figure 4.3 shows an overview of the transformation process. After the service programmer has defined the DSL instance, they need to run a workflow which will generate the final *composition.xml*

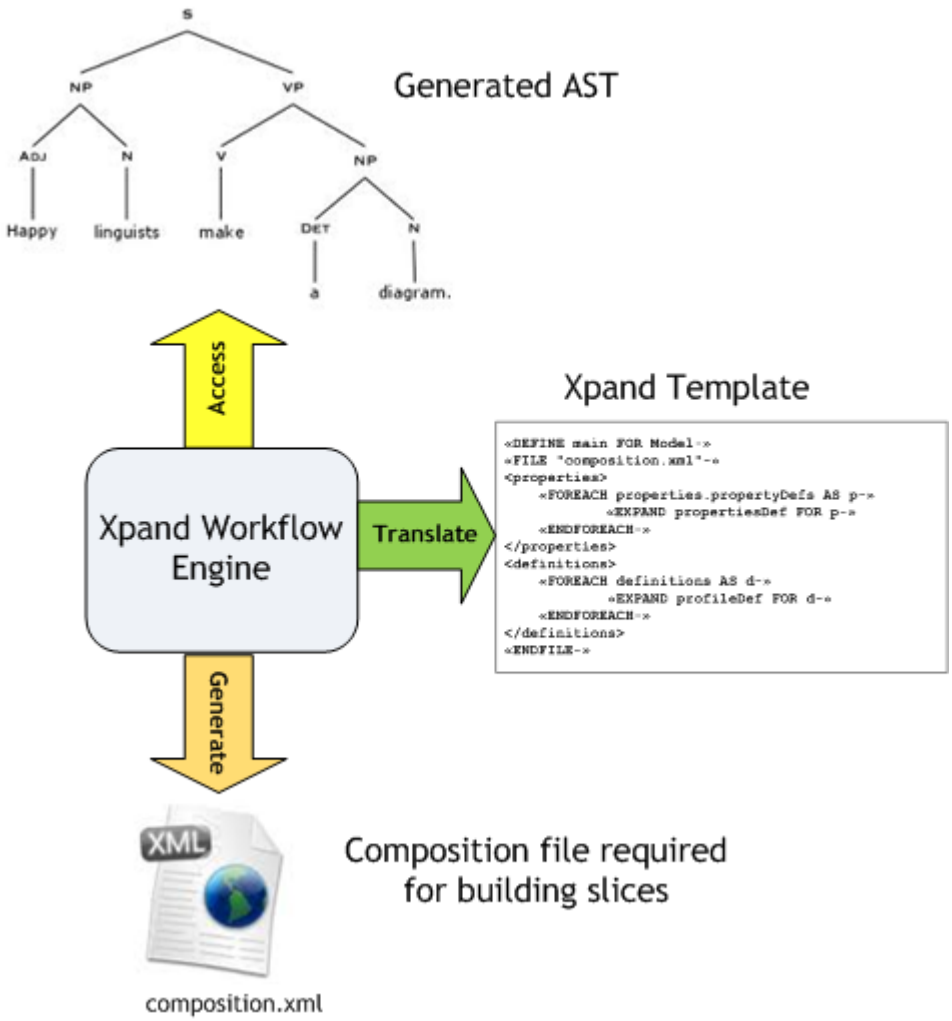

Figure 4.3: Overview of the process for transforming a slice-oriented DSL instance to XML

file from the template. This workflow is intrinsic to the Xpand infrastructure.

4.4.2 XML parser

An XML parser was developed for the decomposition engine in order to parse the auto-generated composition XML file. This parser, which is defined in the *Parsing* module, was developed using the event-based Streaming API for XML (StAX)[11]. The meant that the entire XML file did not need to be read into memory, which results in a much smaller memory footprint than using a document object model (DOM) parser. The parser exposes an API for reading the different sections of a DSL instance, and is shown in table 4.1. A requirement for the parser is that there are classes which can be used to model the different elements within the XML composition file. These data classes were defined in the *DSL Modelling* module.

Table 4.1: DSLParser API

Method name	**Return Type**	**Description**
getProjectName()	String	Returns the project name from the DSL
getDefinitions()	List : Definition	Returns the client profile definitions from the DSL
getProperties()	List : Property	Returns the client context properties from the DSL
getCompositions()	Map : String ⇒ CriterionWrapper	Returns a map of the client profile names mapped to their compositions

4.5 Functional decomposition engine

At the heart of the decomposition engine is the org.eclipse.jdt.core API. This API represents a *Java Model*, which allows Java resources to be accessed and manipulated as required. There are a total of 17 different Java elements that can be represented within

[11] http://stax.codehaus.org/Home

the Java model, as shown in table 4.2. A variety of these elements were used to build a *CriteriaProcessor* decomposition engine whose operation is best describe by example in figure 4.4.

The example scenario shows 2 classes written by the service programmer, namely *ImageManager* and *Scaler*. The ImageManager class contains 2 criteria methods, which are named *sendImage*. Likewise, the Scaler class also contains 2 criteria methods named *scale*. The corresponding DSL instance is also shown. Within the definitions struct of the DSL, 2 client profiles are defined, which are a *powerfulClient* profile and a *weakClient* profile. Below the definitions section of the DSL, there are 2 criterion structs defined, one for each client profile. At build-time when the slice-oriented tool is run, the CriteriaProcessor will use the Parsing module to read the DSL into memory. After this step, every class containing criteria methods will be loaded into memory using the JDT Core API. Then, the CriteriaProcessor will examine each of the QoS-related compositions, which are represented by the criterion structs in the DSL. For each class name in the criterion struct, the CriteriaProcessor will examine the criteria methods that are to *remain* in the corresponding source class. All other criteria methods for that particular method will be removed, using the JDT Core API. For example, consider the *powerfulClient* criterion struct shown in the example DSL. The CriteriaProcessor will interpret this as - "*Remove all 'sendImage' criteria methods from class 'ImageManager' except those annotated with a 'strong' criteria'. Remove all 'scale' criteria methods from class 'Scaler' except those annotated with a 'strong' criteria*". The exact same process applies to the *weakClient* criterion in the DSL. Looking at the ImageManager class in the diagram, it is evident which criteria methods belong to which composition, and which composition belongs to which client profile. The green line shows the criteria methods that belong to the composition associated with a powerfulClient, while the orange line shows the criteria methods that belong to a weakClient. After running the decompositions, the CriteriaProcessor will then output the resulting decomposed source code. It must be noted that the number of copies of a class involved in the decomposition process, will be equal to the number of

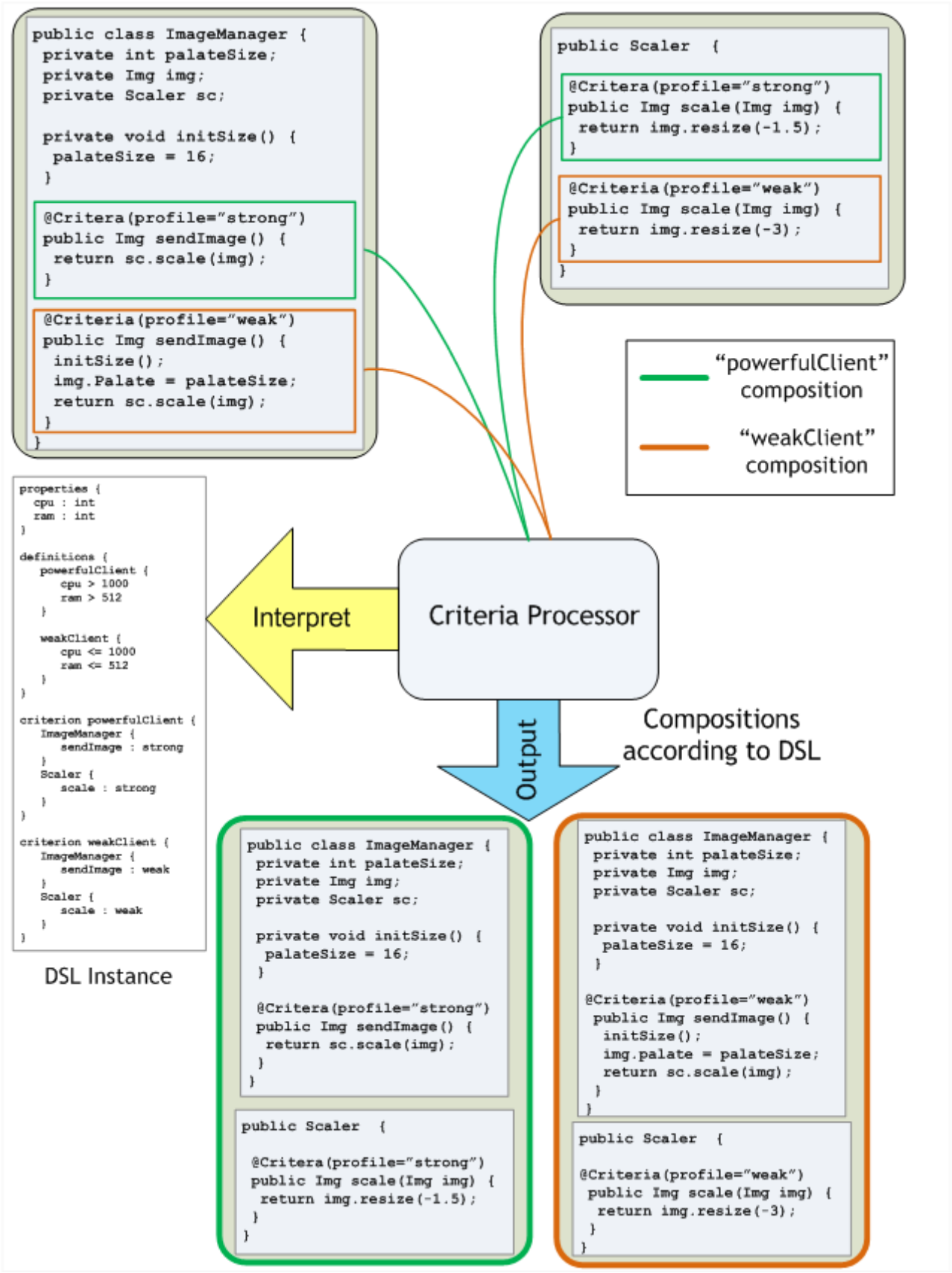

Figure 4.4: Overview of the decomposition process

DSL-specified compositions involving that class, assuming that a composition comprises a combination of all the disjoint criteria methods defined in that class. As the diagram shows, there are now 2 copes of both the ImageManager and Scaler classes which are ready for slicing.

Table 4.2: Eclipse JDT Core Java model elements

Element	**Description**
IJavaModel	Represents the root Java element, corresponding to the workspace.
IJavaProject	Represents a Java project in the workspace. (Child of IJavaModel)
IPackageFragmentRoot	Represents a set of package fragments as a folder, JAR or ZIP file. (Child of IJavaProject)
IPackageFragment	Represents the portion of the workspace that corresponds to an entire package, or a portion of the package. (Child of IPackageFragmentRoot)
ICompilationUnit	Represents a Java source (.java) file. (Child of IPackageFragment)
IPackageDeclaration	Represents a package declaration in a compilation unit. (Child of ICompilationUnit)
IImportContainer	Represents the collection of package import declarations in a compilation unit. (Child of ICompilationUnit)
IImportDeclaration	Represents a single package import declaration. (Child of IImportContainer)
IType	Represents either a source type inside a compilation unit, or a binary type inside a class file.
IField	Represents a field inside a type. (Child of IType)
IMethod	Represents a method or constructor inside a type. (Child of IType)
IInitializer	Represents a static or instance initializer inside a type. (Child of IType)
IClassFile	Represents a compiled (binary) type. (Child of IPackageFragment)
ITypeParameter	Represents a type parameter.
ILocalVariable	Represents a local variable in a method or an initializer.
IAnnotation	Represents a Java 5 annotation.
IAnnotatable	Represents a type, a field, a method, a local variable, or a package declaration that can be annotated with one or several IAnnotations.

4.6 Member dependence graph and program slicer

The member dependence graph functionality was developed entirely from scratch as slicing the decomposed source code required a bespoke dependence graph. The eclipse.org.jdt.core.dom proved a useful API for generating the member dependence graphs, as it allows a compilation unit to be represented as an AST at the statement level. This meant that the

dependencies for each method in the decomposed class could be determined by drilling down to the expressions within these statements, in order to determine what methods and fields are dependees of each expression. An expression within the Java programming language is any construct which evaluates to a single value. The expression construct itself can comprise method invocations, class instantiations, variable assignments and operators. A statement expression, is an expression followed by a semi-colon. Examples of statement expressions include assignment expressions, pre or postfix increment or decrement expressions, method invocations and class instantiations.

Within the Core DOM API, source constructs are represented as instances of type *ASTNode*, and each ASTNode belongs to a unique AST instance. The *ASTParser* class was used to parse the ICompilationUnits generated by the decomposition engine, into *CompilationUnit* ASTNode types. Every ASTNode has an *accept* method for accepting *ASTVisitor* instances, which can "visit" the different node types in that AST. Thus, a visitor was created for visiting the methods within each of the ICompilationUnits generated by the decomposition engine.

The AST parser can accept an ASTVisitor and invoke the visitor's *visit* method for each *MethodDeclaration* ASTNode within the tree. Within the visit method, the visitor can then access the body of the MethodDeclaration instance and recursively iterate through each statement, drilling down to any expressions that are indicative of a method invocation or field access. Only method invocations involving source methods within the same class are considered. Similarly, only field access for fields within the method's enclosing class are considered. A method or field access represents a dependency, and as such will be modeled in the member dependence graph as a target edge. Figure 4.5 shows a sample AST for each MethodDeclaration AST node type within the ImageManager class, decomposed according to a weakClient composition. The diagram also shows the decomposed source code generated from the CriteriaProcessor module. The decomposed classes, which are represented as ICompilationUnits, are then parsed as ASTs by the *Graphs* module. An instance of class *MethodVisitor*, which extends class ASTVisitor, is then accepted

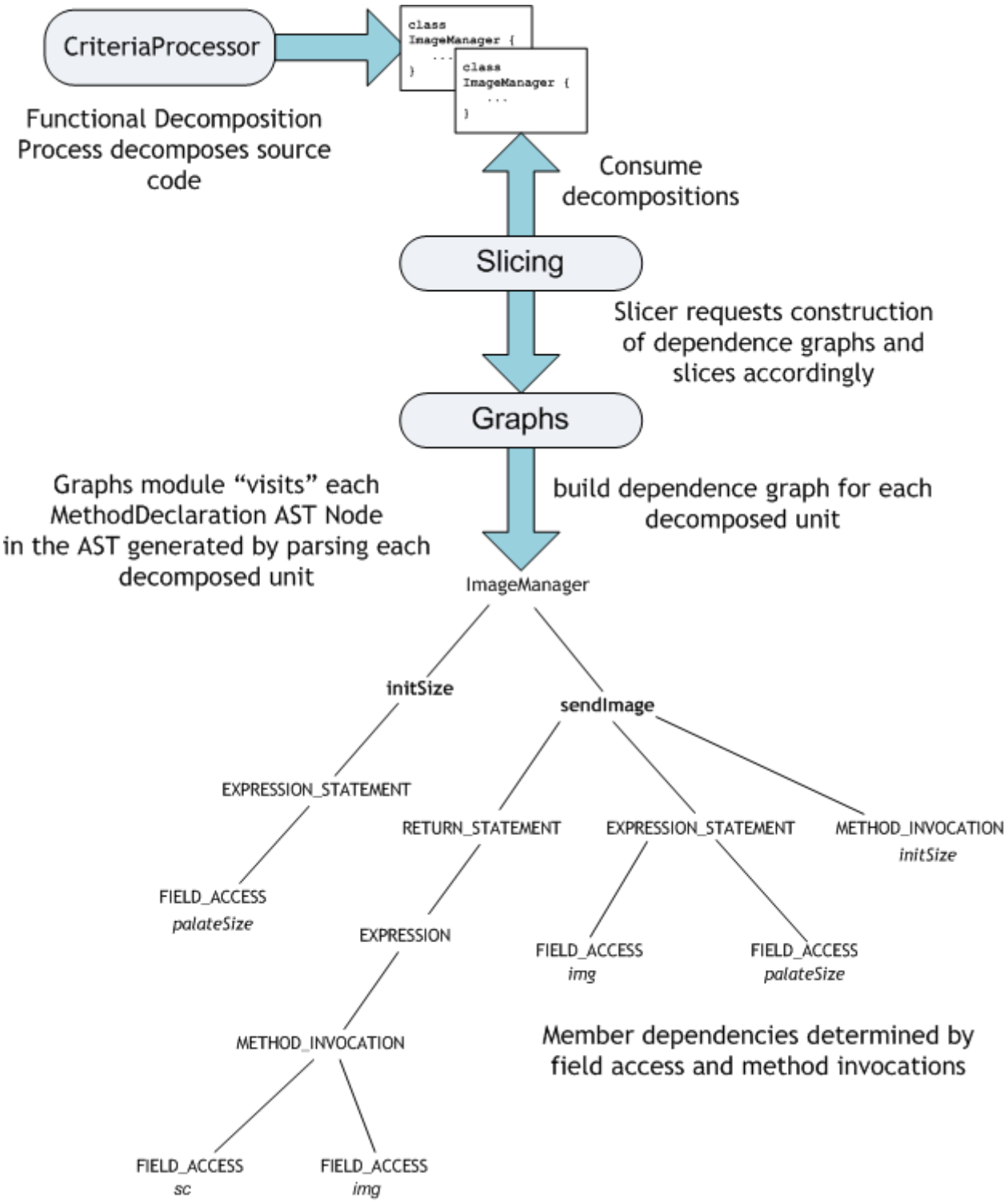

Figure 4.5: Example AST for decomposed class *ImageManager*. *MethodDeclaration* ASTNodes are shown in bold text in the tree. Names of dependent fields and methods are show in italics within the leaf nodes.

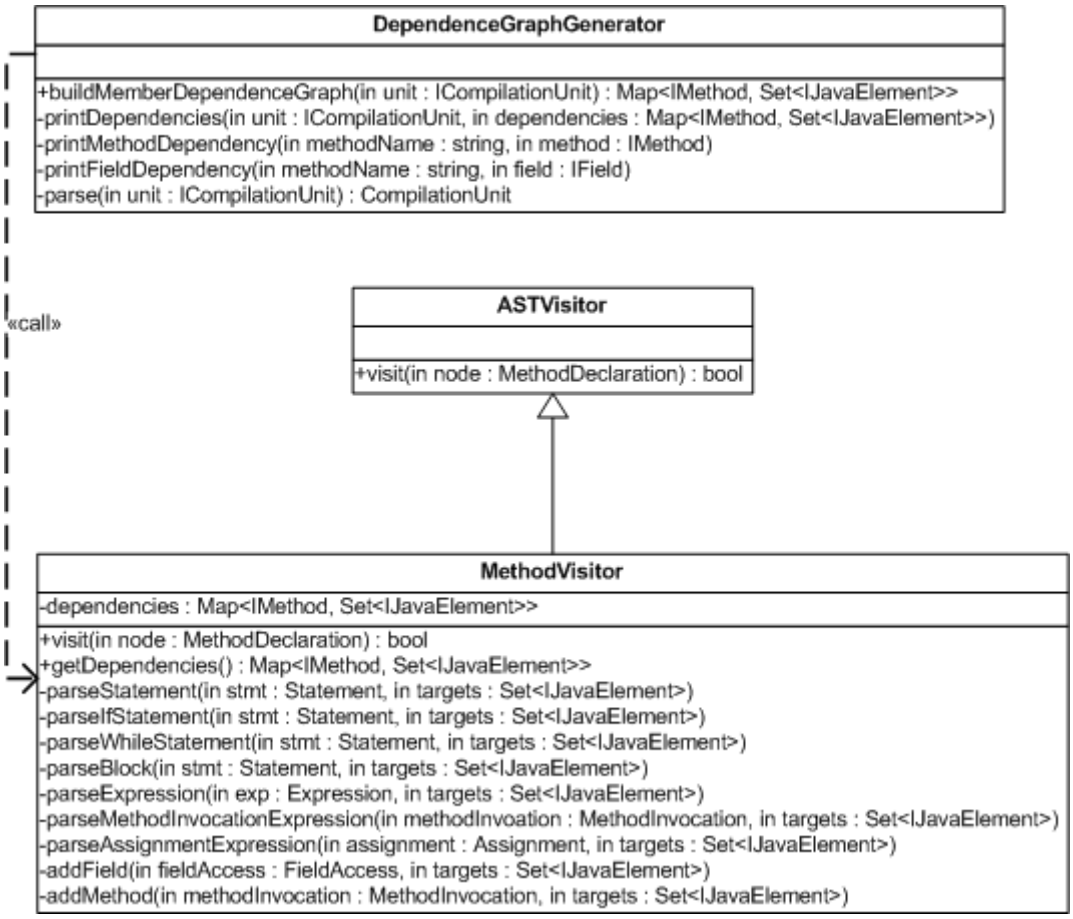

Figure 4.6: Class diagram for graphs module

and invoked by the CompilationUnit, so that its methods can be visited. The diagram shows an AST for methods *initSize* and *sendImage*, from the generated decomposed ImageManager class for a weakClient composition, as shown previously in figure 4.4. Each leaf node in the depicted AST shows the corresponding dependent member within the enclosing CompilationUnit. Finally, an overview of the classes in the graphs module is shown in figure 4.6. The *MethodVisitor* class contains 2 public methods within its API, *visit* and *getDependencies*. The visit method takes an instance of class MethodDeclaration, and is invoked by the CompilationUnit after it has been parsed by the AST parser. Method *getDependencies* will return the member dependencies for a single method as a Map. The map's key is an *IJavaElement* representing the source method, and the value is the set of target edges, or dependencies, for that method. The diagram also shows some private methods within the MethodVisitor for parsing statements and expressions

within a method's body. These are just a small subset of the entire suite of methods for parsing statements and expressions that were developed for the graph generator. The *DependenceGraphGenerator* class contains a public method *buildMemberDependenceGraph*, which takes ICompilationUnits generated by the decomposition engine, and builds a corresponding member dependence graph. This method first calls the private method *parse* in order to parse an ICompilationUnit into an AST, which is represented by a CompilationUnit. This method returns the dependence graph as the same map returned from the getDependencies method, in class MethodVisitor.

The program slicer will call the buildMemberDependenceGraph method for each ICompilationUnit generated by the CriteriaProcessor module. As mentioned in the design chapter, slicing is performed as a graph reachability problem. The resulting algorithm is thus relatively straightforward, and is outlined using pseudocode in listing 4.4.

Listing 4.4: Overview of slicing algorithm.

```
1  public class DecompositionSlicer extends AbstractHandler {
2     public Object execute(ExecutionEvent event) throws ExecutionException {
3        for (ICompilationUnit unit : getDecomposedUnits()) {
4
5           // get member dependence graph for each ICompilationUnit
6           Map<IMethod, Set<IJavaElement>> memberDependenceGraph =
7              MethodDependenceGraph.buildMemberDependenceGraph(unit);
8              sliceCompilationUnit(unit, memberDependenceGraph);
9        }
10    }
11
12    private void sliceCompilationUnit(ICompilationUnit unit,
13          Map<IMethod, Set<IJavaElement>> dependencyGraph) {
14
15       // Get all private methods and fields for the ICompilationUnit
16       Set<IMethod> privateUnitMethods = getPrivateMethods(unit);
17       Set<IField> privateFields = getPrivateFields(unit);
18
19       // Delete any private methods that are not reachable
20       for (IMethod privateMethod : privateMethods) {
```

```
21          if (!targetMethodEdgeExists(privateMethod, dependencyGraph)) {
22             privateMethod.delete();
23             dependencyGraph.delete(privateMethod);
24          }
25       }
26
27       // Delete any private fields that are not reachable
28       for (IField privateField : privateFields) {
29          if (!targetFieldEdgeExists(privateField, dependencyGraph)) {
30             privateField.delete();
31          }
32       }
33
34       // delete any redundant imports that may be associated
35       // with members that have been removed
36       if (codeWasSliced) {
37          deleteRedundantImports(unit);
38       }
39    }
40 }
```

The first step in the slicing algorithm is the examination of the Java model for the ICompilationUnit, in order to return all the private methods and fields in the class. Then, the algorithm will iterate over all the private methods for the class and will check if each method is the destination of a target edge in the dependency graph. If it is not, then the method will be deleted. Next, the algorithm will iterate through all the private fields in the class. Again, each field is checked to see if it is a dependee by checking the dependency graph for any target edges that point to this field. If the field has no dependents, then it is deleted. The algorithm also works for transitive dependencies. Finally, any redundant class imports that are left over after slicing are also removed.

4.7 Architecture overview

Figure 4.7 shows an overview of the functional architecture implemented for the slice-oriented prototype tool. The architecture is broken into 3 areas. The 2 areas that coa-

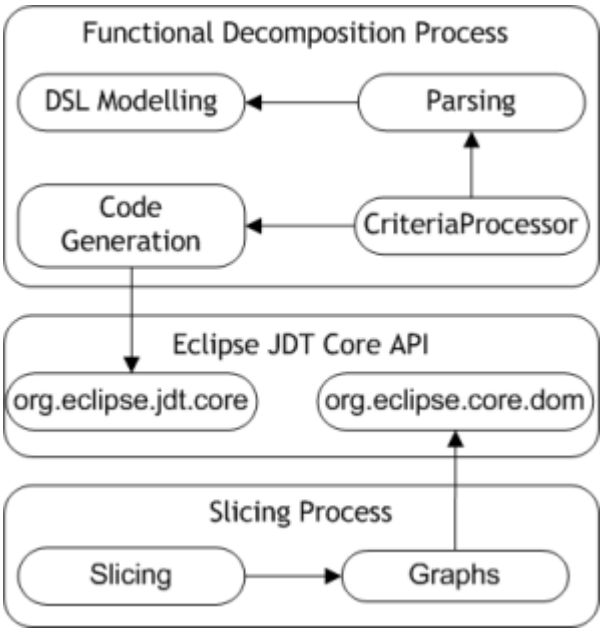

Figure 4.7: Functional architecture overview

lesce into the slice-oriented programming methodology are the Functional Decomposition Process and the Slicing Process, and both of these processes rely on the the functionality provided by the Eclipse JDT Core API. Since the *Parsing* module is responsible for parsing the XML representation of the DSL instance, it will depend on the DSL Modelling component. The *Code Generation* module is used for writing the source code decompositions to file, generating packages as well as any sort of source code manipulation. This explains why it is a dependee of the CriteriaProcessor module. The CriteriaProcessor is also dependent upon the Parsing module in order to determine the QoS-related compositions that are to be generated. Finally, the slicing functionality is modularised into the *Slicing* component. It relies upon the *Graphs* module to generate the member dependence graphs necessary for slicing.

4.8 Implementation issues

There were 3 main issues encountered while implementing the slice-oriented prototype tool. The first issue was a technical issue relating to intermittent memory leaks while

using Xtext in an Eclipse run time environment. The memory leaks usually manifested themselves while attempting to run the Xpand template engine, after creating a DSL instance. The exact error was a *java.lang.OutOfMemoryError* relating to the PermGen area of the heap memory. This particular area of the heap is responsible for storing Class objects that are loaded by Java's ClassLoader, and its use yields better performance for the Garbage Collection process. To rectify this issue, the size of the PermGen area of memory can be increased using the JVM *-XX:MaxPermSize* argument. However, this did not seem to fix the memory leak. So it was decided to downgrade the Java version from jdk 1.6.0_26 to jdk 1.6.0_20. After doing so, the memory leaks subsided.

The second issue related to a bug in the JDT Core API. After source code has been decomposed by the decomposition engine, the underlying import statements may need to be updated to point to other decomposed classes. Attempting to use the Core API to amend imports resulted in a run time exception, which was reported as a bug to the Eclipse community. The workaround was to first delete the import and then recreate it with the correct package name.

The final issue to contend with wasn't a technical issue per se, but rather a functional issue relating to the use of criteria methods within the JDT Core API. The issue is best clarified with an example. Listing 4.5 shows a class which contains 3 criteria methods. Imagine that a particular composition requires that the Foo class be decomposed to remove the wifi and 3g *bar* methods. After the wifi bar method has first been deleted, the Java model will believe that there are no more bar methods within the Foo class. It correctly assumes that a class can only have 1 method with a particular signature. So, when the CriteriaProcessor attempts to delete the 3g bar method, a run time exception will be thrown, stating that the bar method no longer exists. The only workaround for this issue was to reopen the corresponding ICompilationUnit each time a criteria method is deleted. Reopening the ICompilationUnit will refresh the Java model, which means the 3g bar method can then be deleted.

Listing 4.5: Example class showing 3 criteria methods.

```
1  public class Foo {
2     @Criteria(profile = "wifi")
3     public void bar() {
4        // ...
5     }
6
7     @Criteria(profile = "3g")
8     public void bar() {
9        // ...
10    }
11
12    @Criteria(profile = "gprs")
13    public void bar() {
14       // ...
15    }
16 }
```

Chapter 5

Evaluations

5.1 Methodology

In order to evaluate the slice-oriented programming model proposed in this dissertation, a SOAP [37] service was built which contained functionality that allowed client devices to download portable network graphics (PNG) images. A PNG image employs lossless data compression, and as such, is suitable for scaling with no loss in quality. Furthermore, a PNG image supports palette-based indexing for its pixel colours, which does not incur as high a memory footprint as other image formats. Collectively, these properties provided a good scenario for specifying different QoS-related compositions, of the image functionality, within the SOAP service.

Based on this, a DSL instance was defined which included 3 properties representing the client's operational context. The 3 properties were *cpu, ram* and *bandwidth*. 3 client profiles were then defined in the DSL covering different ranges of these context properties. The 3 profiles were entitled, *PowerfulClient, MediumClient* and *WeakClient*. The last section of the DSL included the QoS-related composition information for the web service functionality, as shown in table 5.1. The 2nd row in table 5.1 shows the use of the *void* keyword within the decomposition process. This row can interpreted as *"all criteria methods are to be removed from class service.img.ImageManager"*.

Client Profile	Package Name	Class Name	Criteria Method Name	Criteria Profile
PowerfulClient	service.delegate	ImageSIBDelegate	downloadImage	strong
PowerfulClient	service.img	ImageManager	void	void
MediumClient	service.delegate	ImageSIBDelegate	downloadImage	medium
MediumClient	service.img	ImageManager	reduceColourPalate scaleImage	medium medium
WeakClient	service.delegate	ImageSIBDelegate	downloadImage	weak
WeakClient	service.img	ImageManager	reduceColourPalate scaleImage	weak weak

Table 5.1: QoS compositions as stipulated in DSL instance

The SOAP service was constructed using JAX-WS[1], which is now part of the Java 6 implementation. Building a SOAP service using JAX-WS requires the definition of a service endpoint interface (SEI), and a service implementation bean (SIB). The SEI exposes the WSDL operations that the client device can invoke, and the SIB implements the operations defined in the SEI. A UML overview of the image functionality, before slicing, is depicted in figure 5.1. In the UML diagram, the *ImageSEI* interface represents the SEI, and the *ImageSIB* class represents the SIB.

The ImageSIB class delegates calls to its *downloadImage* method to an *ImageSIBDelegate* instance. There are criteria methods defined in classes ImageSIBDelegate and ImageManager in order for the slices to be generated. The UML diagram also shows that the downloadImage method, defined in the ImageSEI interface, takes a single parameter which is a ClientProfile instance. This auto-generated class instance contains the client context properties as defined in the DSL instance for this use case. Figure 5.2 shows a UML overview of the image service functionality *after* slicing. The left side of the diagram shows a nice holistic view of the generated slices, or, more formally, QoS-related compositions. The slices comprising a WeakClient QoS composition are shown in yellow while the slices for a MediumClient QoS composition are shown in amber. Finally, the slices for a PowerfulClient QoS composition are shown in green. Note that the PowerfulClient composition does not include any functionality to manipulate the PNG image

[1]http://jax-ws.java.net

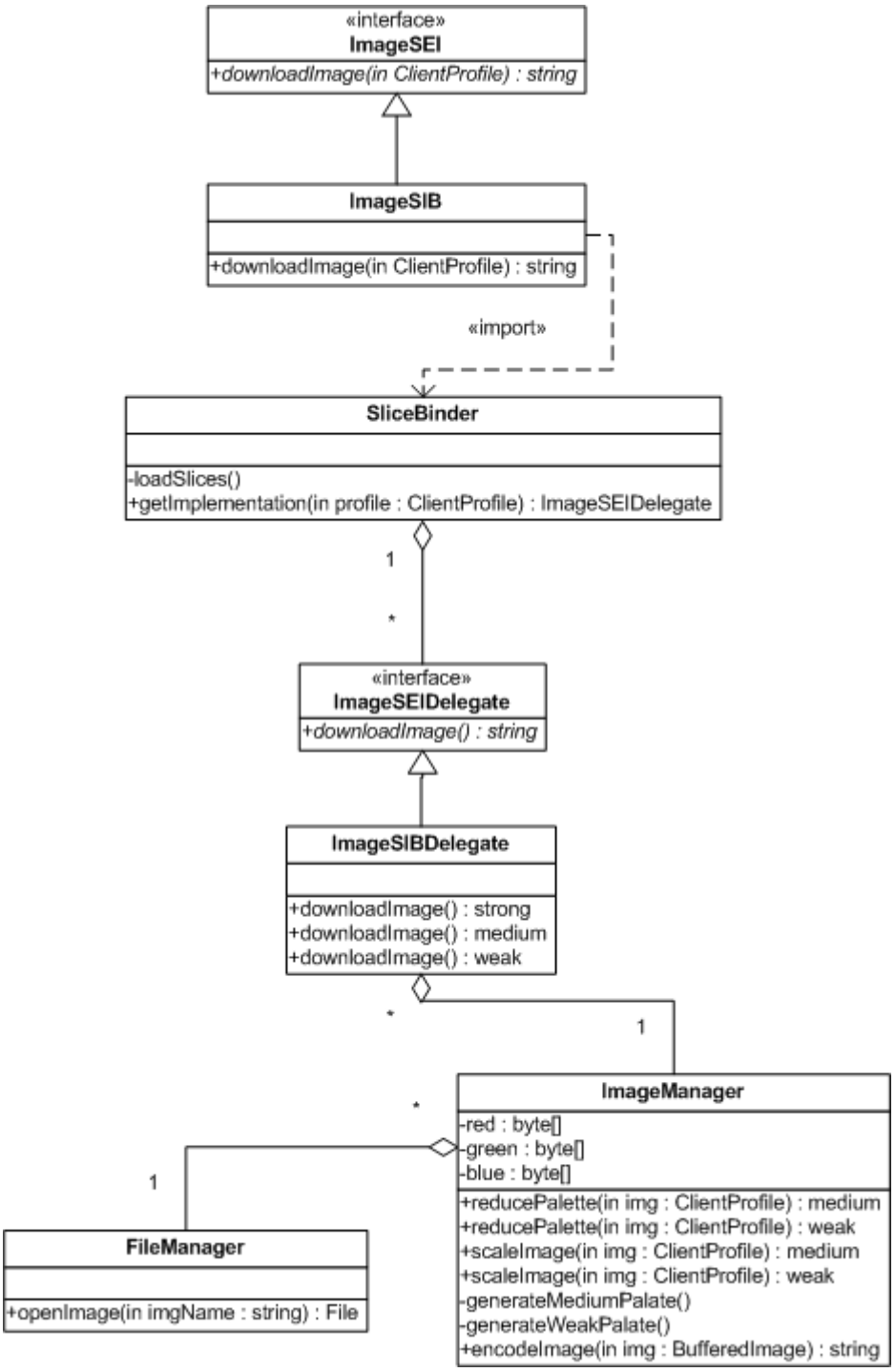

Figure 5.1: Overview of image service functionality before slicing

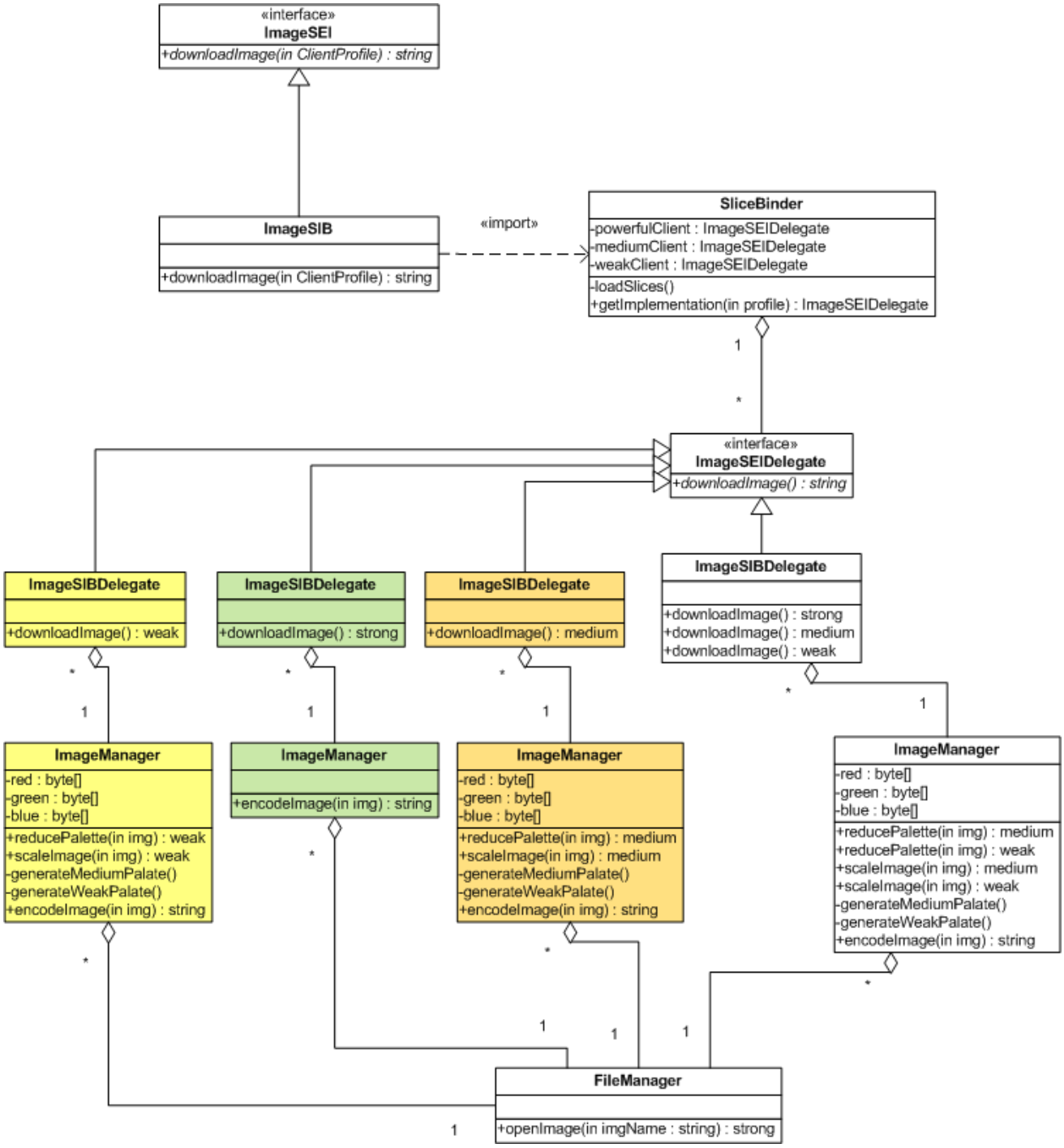

Figure 5.2: Overview of image service functionality after slicing

because of the use of the void keyword for excluding all criteria methods during decomposition. Also note that the SliceBinder contains 3 *ImageSEIDelegate* references, each of which points to an *ImageSIBDelegate* slice implementing this interface. This acts as the interface into the generated QoS compositions. For example, when a MediumClient invokes the service, the service implementation bean (ImageSIB) will ask the SliceBinder for an ImageSIBDelegate instance to execute for the MediumClient. The ImageSIB will then call the downloadImage method on this slice, which will execute all the necessary logic within the composition for a MediumClient.

The same service functionality was also implemented using different software creational patterns, which were relevant for this particular scenario. The different patterns used were:

- Abstract factory pattern
- Builder pattern
- Conventional approach
- Dependency injection pattern

The abstract factory creational pattern provides a factory interface from which different service beans can be created, each one representing a different composition of the underlying service logic. Similarly, the builder pattern allows the construction of different compositions of the service logic via an abstract class, which specifies the different steps for building the composition. Dependency injection allows the different objects involved in the composition of the service bean to be injected into classes at run time. This is instead of a class having to explicitly instantiate the objects it depends on. Google Guice[2] is a lightweight dependency framework which was used to implement dependency injection. Finally, the *conventional* approach to composing a service bean, did not involve the use of a creational pattern. Instead of using inheritance or dependency injection, the

[2]http://code.google.com/p/google-guice/

conventional approach simply defines methods necessary for the different client profiles. This means that at run time, when a ClientProfile instance is received from a client device, it must be propagated throughout the code in order to invoke the correct method. This resulted in peppering the code with conditional logic, in order to invoke the correct method with regard to the received ClientProfile instance. A simple example is shown in listing 5.1. In order to invoke any of the methods in the ImageManager class, the calling class will need to conditionally evaluate the ClientProfile instance received from the client device.

Listing 5.1: Example of source code structure for a conventional approach

```
1  public class ImageManager {
2     public Img scaleImageMedium() {
3        // ...
4     }
5
6     public Img scaleImageStrong() {
7        // ...
8     }
9
10    public Img scaleImageWeak() {
11       // ...
12    }
13 }
```

5.1.1 Performance

In order to gauge the type of service QoS that would be experienced by the end-user, a client application was built for an Android-enabled mobile device in order to test the image service. The application used the ksoap2 [3] framework for communicating with the image service. ksoap2 is a lightweight soap framework tailored to work on constrained Java devices. At run time, the android application allowed different values to be entered

[3] http://code.google.com/p/ksoap2-android/

for the cpu, ram and bandwidth, in order to simulate the different client profiles that were defined in the DSL. The original PNG image used for the use case experiments had an original file size of 220kilobytes. In terms of the image's dimensions, it had a width of 331 pixels and a height of 235 pixels. There were a total of 3 invocations of the image service using the android application, one for each client profile defined in the DSL. The load-time and run time performance of the slices was also evaluated. This was done by measuring the time taken to instantiate 1000 ImageSIB objects, and then taking the average time, measured in nanoseconds. The run time performance is representative of the time taken for a slice to execute, *after* the image service receives a client request. This was measured by instantiating an instance of the ImageSIB class, and invoking the downloadImage 1000 times for each of the different client profiles in the DSL. The average time, measured in milliseconds, was then recorded.

5.1.2 Software metrics

Software metrics are a series of quantitative properties that allow the overall quality of a software program to be evaluated. These evaluations allow a programmer to identify different areas of the program that don't conform to good design. It's important to identify areas of the software that are indicative of poor design, as failing to rectify these discrepancies can result in a high degree of complexity for the software [38]. Good design means good overall software quality. The use of software metrics can confirm the quality of a piece of software, and can be used to estimate its overall complexity [39]. The maintainability of software, is inextricably linked to its underlying complexity [40] and affects how easily a programmer can understand, adapt and extend it. Chidamber and Kemerer proposed a suite of object-oriented software metrics [41]. These same metrics have been validated as a useful set of quality metrics for evaluating fault-proneness within different design phases of a software project [42]. Furthermore, the Chidamber and Kemerer software metrics have been proven to be effective in estimating the maintenance effort

for object-oriented software [43]. Other software metrics, such as lines of code (LOC) [44] and Cyclomatic complexity [45] are code metrics, and as such are more suited for post-implementation evaluations [39].

The evaluation of the slice-oriented methodology, for the aforementioned use case, will calculate the following Chidamber and Kemerer metrics - weighted methods per class (WMC), depth of inheritance tree (DIT), number of children (NOC), coupling between objects (CBO) and lack of cohesion in methods (LCOM). LOC and average cyclomatic complexity will also be evaluated. Collectively, these metrics will be used as a basis to investigate the maintainability of the image service logic, which has been composed using the slice-oriented technique. The evaluated metrics that are presented in this dissertation, are derived from the final stage of the entire slice-oriented process, in other words, after the executable slices have been generated. In addition, the metrics are calculated using Google's CodePro Analytix software testing framework[4].

The cyclomatic complexity of a single method, is a measure of the number of distinct paths of execution within that method [45]. It is measured by adding the one path for the method with each of the paths created by conditional predicates, such as *if* and *for* etc. The cyclomatic complexity of a method should ideally be less than 10. The WMC metric for a class, is calculated as the sum of the cyclomatic complexity of each of its constituent methods. It therefore allows the complexity of an object to be quantified. The cardinality of class methods, as well as their corresponding complexity, allows the time and effort for maintaining the class to be estimated. Quite simply, the DIT for a class is a measure of the distance of a class in the hierarchy right up to the root class. In other words, the DIT metric measures how many ancestor classes could potentially affect a particular class. The NOC for a particular class, is a measure of the number of immediate sub-classes, or children of that class. Quintessentially, it measures how many children of a class will inherit properties of a parent class. CBO is a metric measuring the number of couplings between a chosen class and additional classes in an application. Specifically, a coupling is

[4]http://code.google.com/javadevtools/codepro/doc/index.html

represented by the degree to which methods of one class, use methods or instance variables of another class. The CBO in this dissertation is evaluated by measuring the afferent and efferent couplings for a target class. Considering a software component A, the number of afferent couplings is a measure of the number of classes *outside* of A, that depend on classes *inside* of A. The efferent couplings is a measure of the number of classes *within* A, that depend on classes *outside* of A. For a particular class, LCOM is a measure for gauging the degree of similarity of its methods. It counts the number of method pairs whose similarity is 0. A higher LCOM indicates more disparity between the methods in a class, and is indicative of lower class cohesion. When measuring the LCOM for a class in this dissertation, instance variables that are not used by methods, or methods that do not use instance variables, are not included in the LCOM calculation. LOC measures the source lines of code in a class. Although LOC is a widely-used size metric, its use is often criticised due to fundamental ambiguities associated with source code [46]. For example, consider a line of source code which comprises multiple statements. Although this single line of code does indeed constitute one physical line of code, it contains statements of varying complexity. However, some believe the use of LOC as a suitable metric for estimating the maintainability of a program is justified [47], whereas others argue against it [48]. The slice-oriented programming evaluations will include the LOC metric, as it can be used in combination with other metrics.

5.2 Performance evaluations

5.2.1 Application QoS

Figure 5.3 shows a screenshot of a PNG image delivered to the android client device, whose context was categorised as a PowerfulClient by the image service.

Figure 5.3: Image displayed on android device for a *PowerfulClient* profile

The image is in no way distorted because at build-time, according to the DSL instance, the PNG image was not to be manipulated in a QoS-related way. The context was simulated as a device with a cpu of 1GHz, ram 512MB and a network bandwidth of 1Mbps. The total round-trip-time (RTT) for the request was 3.636 seconds. The simulated client context was then degraded to a cpu of 700Mhz, ram 70MB and network bandwidth of 700Kbps. Again the image service was invoked, and this resulted in the service characterising the client device as a MediumClient profile. Figure 5.4 shows the same PNG image returned from the image service, except altered for a MediumClient.

Figure 5.4: Image displayed on android device for a *MediumClient* profile

As is evident from the diagram, the PNG image dimensions have been reduced to a width of 261 pixels and a height of 185 pixels. Additionally, the PNG colour depth was reduced to 4 bits, meaning 16 colours in total. This results in less colour in the PNG image and can be seen in the screenshot. By virtue of reducing the colour depth to 16, and scaling the image down, the image file size was reduced to 91kb. The RTT for a service request assuming a MediumClient profile was 1.418 seconds, which was less than half the time taken for a PowerfulClient profile. Finally, the simulated client context was reduced to a cpu of 20Mhz, a ram of 8MB and a network bandwidth of 10Kbps. When invoking the downloadImage operation of the image service, the client context was characterised as a WeakClient profile. The resulting PNG image is shown in figure 5.5.

Figure 5.5: Image displayed on android device for a *WeakClient* profile

Again, the manipulations of the PNG image for a WeakClient profile have resulted in the image dimensions being scaled down even further. Also, the colour depth for the PNG image has been reduced from 4 bits to 3 bits, resulting in only 8 possible colours. Additionally, the image's dimensions were scaled down even further to a width of 191 pixels and a height of 136 pixels. Collectively, these manipulations resulted in a final image file size of 37kb. The RTT for invoking the service using a WeakClient profile resulted in the lowest RTT of all client profiles, with a time of 1.072 seconds.

The perception of the QoS for the end user is entirely subjective. In this case, it depends on how satisfied the end user is with the quality of the image returned from the image service, as well as the total RTT for the request. It was observed that the RTT for downloading the image decreased, as the context of the device degraded from a PowerfulClient profile, to a WeakClient profile. In this use case, the sacrifice for a quicker RTT was a degradation in image quality, yet the image was still discernible irrespective of the client profile. If it so happened that the programmer decided that the PNG was too large for the MediumClient profile, for example, the service would need to be recomposed.

In this case, the image scaling functionality for a WeakClient profile, would be more suitable for reducing the PNG image size for a MediumClient profile. With the creational patterns, achieving this modification would require the introduction of an additional class dependency, along with a code modification in order to call the appropriate method from the dependee. However, with the slice-oriented model, all that is required to recompose the service logic is to update the QoS-related compositions section of the DSL instance. In other words, no additional dependencies are introduced, nor are any code modifications required by virtue of the flexibility offered by the DSL.

5.2.2 Load-time

The results for loading an implementation bean for each of the different creational patterns are depicted in figure 5.6, and are measured in nanoseconds.

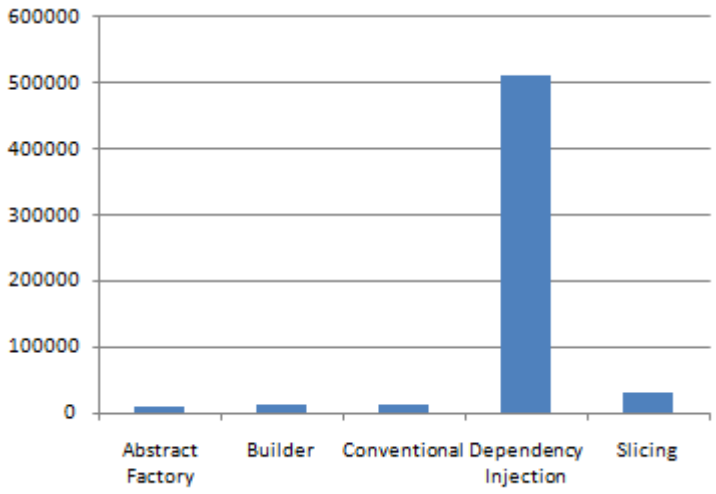

Figure 5.6: Time to load a service implementation bean in nanoseconds

The creation of a service bean, as composed using the abstract factory pattern, gives the quickest load time of 9130 nanoseconds. The reason is that the implementation bean does not contain any instance reference variables. Creating a service bean using the conventional approach took on average 11424 nanoseconds. When instantiating a service bean composed using the conventional technique, there are 2 instance reference variables

defined within the service bean, namely an ImageManager and a FileManager. These 2 variables are instantiated as part of their declaration, and as such, their corresponding classes are loaded by Java's system ClassLoader. Subsequently, the Java Virtual Machine (JVM) will need to allocate heap space for the class' instance variables. A service bean created using the builder approach contains 4 instance variables, only one of which, in turn, contains a method-instantiated instance variable. This approach returned an average of 12688 nanoseconds. Instantiating a service bean using the slicing technique incurs a higher average load time of 31113 nanoseconds. The reason for the higher average load-time, is a result of the auto-generated SliceBinder instance that must be placed into the service bean as an instance variable. When the service bean is then loaded, it will instantiate the binder, which in turn instantiates the necessary slices that make up the composition. Each slice in turn contains its own object graph that needs to be constructed by the JVM. Collectively, this attributes to a longer load-time than the other compositional approaches. Finally, instantiating a service bean that has been composed using dependency injection incurs a load-time of 513148. This is up to 16 times longer than that of the slicing technique, and 56 times longer than the abstract factory approach. The reason for this additional overhead is twofold. First, when the service bean is created, the binding modules for dependency injection must be generated. Secondly, Google Guice will take this binding module and build up an object graph to fulfill the underlying dependencies. Although a higher load time is incurred for service beans composed using the slicing technique, the instantiation of a service bean only occurs at service start-time.

5.2.3 Run time

The run time performance metrics for the slicing technique are much more favourable than the load-time metrics. Examining the run time metrics in figure 5.7, the slice-oriented technique is quicker in fulfilling service requests for a weak and medium profile, than all other approaches. Only the abstract factory approach is marginally quicker for fulfilling

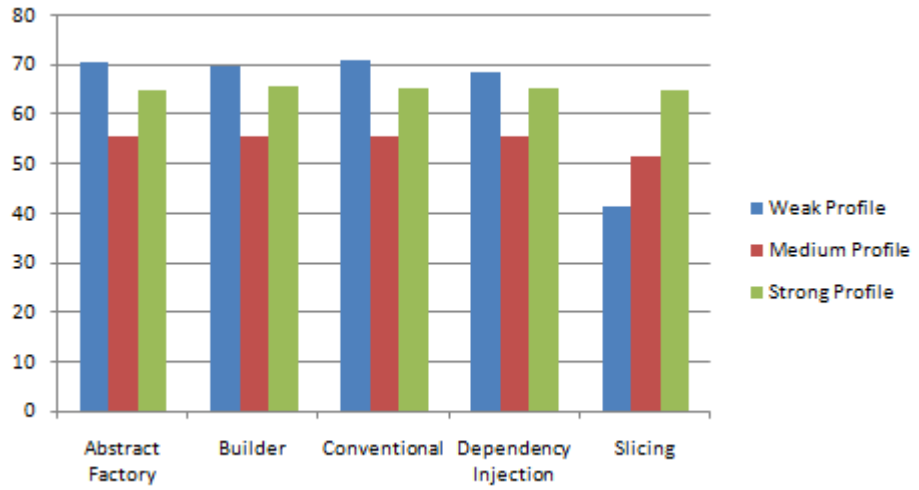

Figure 5.7: Total time taken for a service bean to fulfill a request in milliseconds

requests from client's that are characterised by a strong profile. The reason for the quicker response time using the slice-oriented approach, is a result of how the program is designed. Since the SliceBinder will load all the necessary slices at service initialisation time, when a service request is received from a client, the slices are already loaded in memory. The corollary is that no further initialisation is necessary, which means better performance metrics. This is in contrast to the abstract factory approach, for example, which needs to instantiate a factory implementation that pertains to the client profile at hand. Figure 5.8 shows the average cpu usage for fulfilling a service request from a client characterised by a weak profile. In addition, figure 5.9 shows the average heap size for the same scenario. The averages are taken from 1000 service requests. The 2 graphs show a lower average cpu usage and lower average heap size required for the slice-oriented approach.

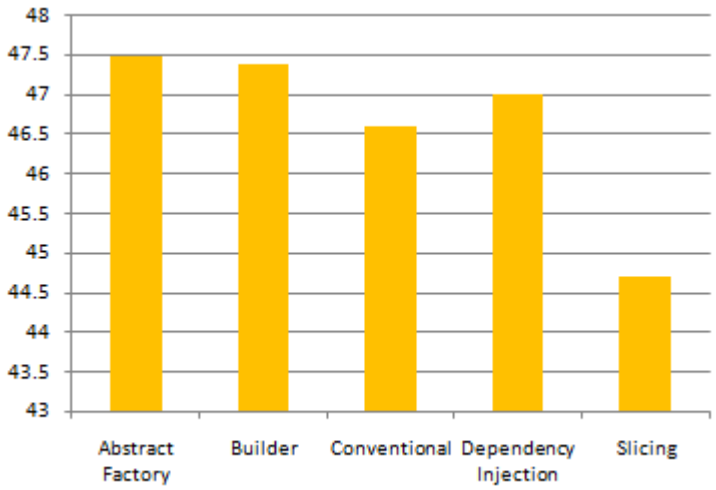

Figure 5.8: Average % cpu usage for servicing a request from a client characterised by a WeakClient profile

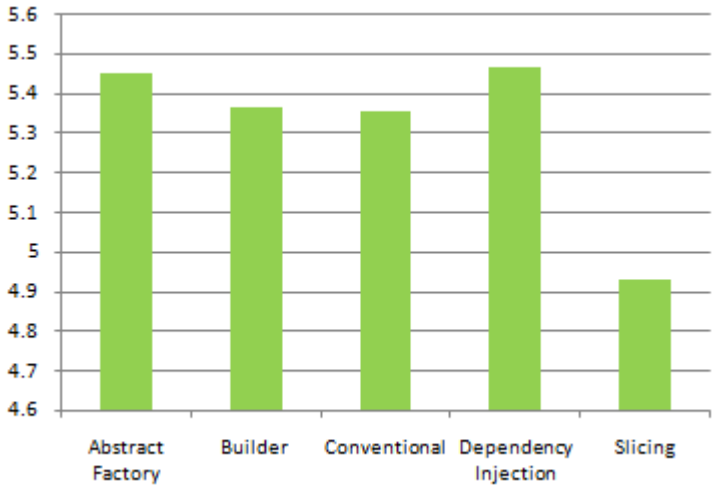

Figure 5.9: Average heap size for servicing a request from a client characterised by a WeakClient profile, in megabytes

5.3 Software metrics evaluations

5.3.1 Weighted methods per class

Figure 5.10 shows that the slice-oriented technique for composing the service logic incurred an overall WMC metric of 61, which was the second highest of all approaches.

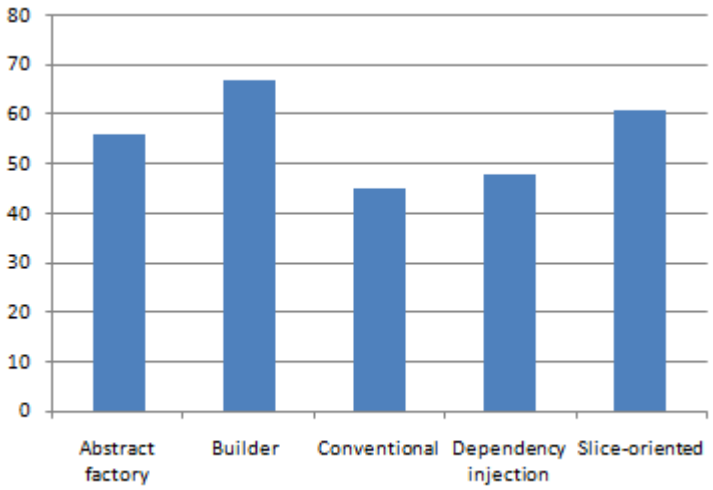

Figure 5.10: Weighted methods per type

Only the builder pattern had a higher WMC metric of 67. The WMC metric is indicative of the estimated time and effort required to develop and maintain the core service logic, moving forward. The reason for the relatively high WMC for the slice-oriented approach is due to the number of additional software modules, that are generated using this technique. After functionally decomposing the service logic, the number of software modules that will be generated will be a factor of the number of compositions defined in the DSL. For example, for the use case outlined in this chapter, the original service logic that contains the criteria methods is modularised in 2 components. Since the DSL instance defines 3 different compositions, the number of resulting modules after functional decomposition will increase from 2 to 6. This will inevitably affect the WMC, as there are now more methods from which to generate it.

The WMC metric is inextricably linked to the cardinality and cyclomatic complexity of class methods [41]. The average number of methods within the slice-oriented approach, as show in figure 5.11, is 1.7. This is lower than both the builder and conventional approaches. The benefit of a lower method count per type is that it indicates greater class reuse [41]. Furthermore, although the conventional approach has the lowest WMC with a value of 45, figure 5.12 shows that its average cyclomatic complexity is higher than every

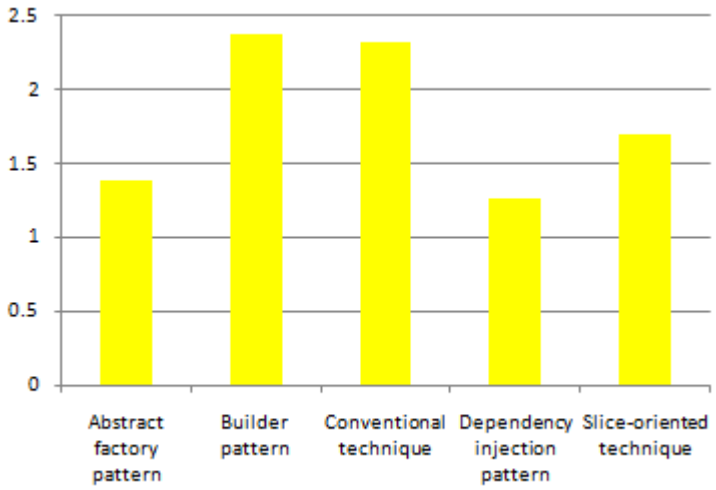

Figure 5.11: Average methods per type

other approach. This is because the conventional approach is peppered with conditional statements. These statements are necessary in order to determine the correct logic to invoke, with regard to the current client profile. The average cyclomatic complexity for the slice-oriented approach is 1.52, which is well below the recommended threshold of 10 [49]. The *overall* average cyclomatic complexity for all approaches is 1.64, which shows us that the slice-oriented complexity is just below this average.

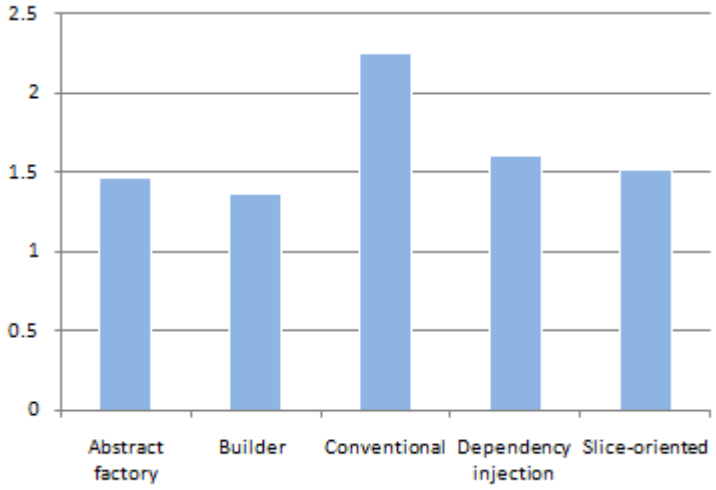

Figure 5.12: Average cyclomatic complexity per method

5.3.2 Depth of inheritance tree

The average DIT metric for each approach is shown in figure 5.13. The slice-oriented approach has the lowest average DIT of all approaches, with a value of 1.64. The builder pattern for composing the service logic yields the highest DIT with 2.22, which is still considerably low, and as such, won't pose considerable issues for maintainability.

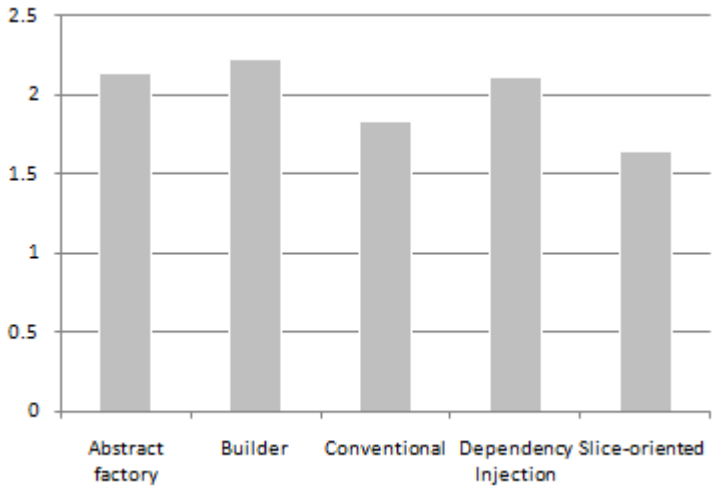

Figure 5.13: Average depth of inheritance tree

The reason for the lower DIT metric with the slice-oriented approach is because the other methodologies, such as the abstract and builder, inherently rely upon abstract classes. These abstract classes must be extended by concrete implementations, which will result in a higher DIT metric. A lower DIT is more favourable, which means that there are less classes in the inheritance hierarchy. This means that there are ultimately less methods that are inherited from the root of the object graph, resulting in less complexity [49][41]. There is a comparison that can be made between the slice-oriented approach, and one which relies on inheritance, such as the abstract factory pattern. The similarity is that the slice-oriented approach allows multiple definitions of a single method *within* the declaring class, through the use of the criteria annotations. This is comparable to overriding a superclass method multiple times using class extensions.

5.3.3 Number of children

The average NOC for the core service logic using the slice-oriented approach, as shown in figure 5.14, is 0.25.

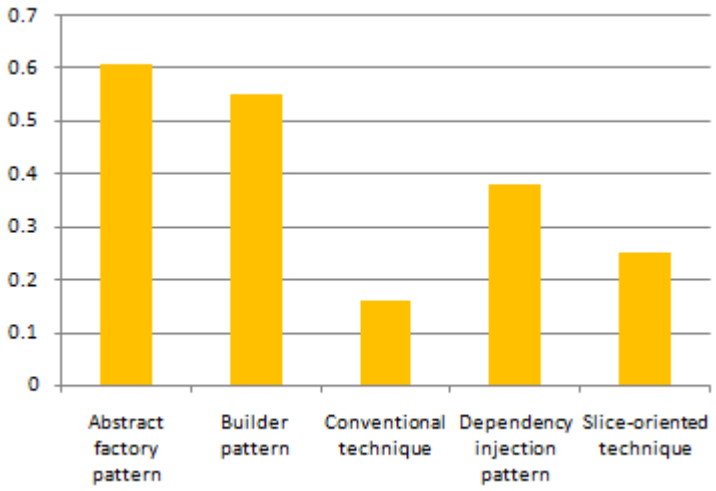

Figure 5.14: Average number of children per type

Again, the slice-oriented technique has a lower NOC value than other approaches, as it doesn't innately rely on inheritance to generate the service logic. A higher NOC can be a sign of the misuse of subclassing [49]. The reason is that classes that are further up in the inheritance hierarchy, should have a higher number of children than those lower down in the hierarchy. From a maintainability perspective, the more children that a class has, the greater the testing effort required for changes to methods in the parent class [41]. However, although the NOC metric for the slice-oriented approach is low, the auto-generated slices are comparable to subclasses, as they are generated from the logic containing the criteria methods. With this in mind, the NOC metric doesn't provide an accurate result here for the slice-oriented approach; it can be argued that the more slices that are generated, the greater the maintainability effort required for testing. Conversely, when making code modifications, the programmer can do so in the class from which the slices are calculated. After, all that is required is for the slice-oriented tool to be run again, in order to regenerate the slices.

5.3.4 Couplings between objects

Figure 5.15 shows the couplings for the different approaches.

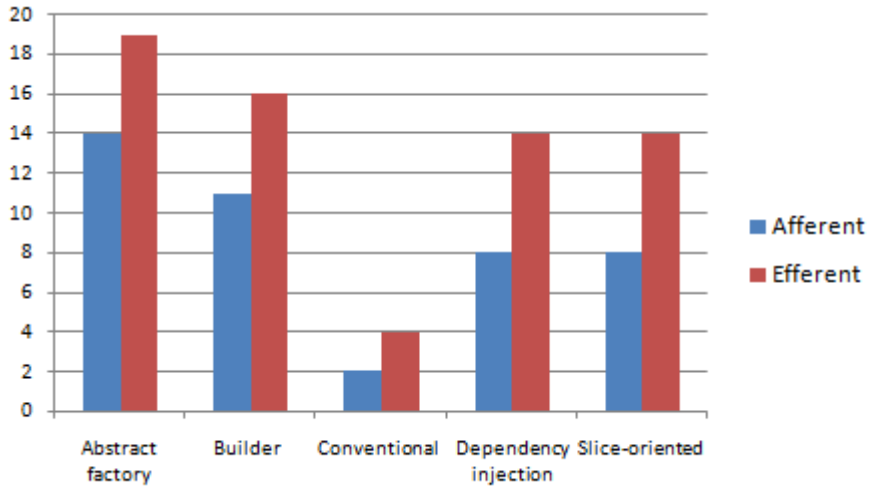

Figure 5.15: Afferent and efferent couplings

The slice-oriented approach has a total of 8 afferent couplings and 14 efferent couplings. The afferent couplings for the slices are representative of the dependencies between the SliceBinder and the generated slices. Recall that the SliceBinder is responsible for returning a slice to the implementation bean, based on the profile of the client that is invoking the service. As a result, it will inevitably have dependencies on each of the slices. The efferent couplings for the slices, are equivalent to the dependencies that exist between the core service logic, from which the slices are generated. This is similar in nature to couplings that are inherited from a superclass. A higher degree of coupling between objects *outside* of the inheritance hierarchy is unfavourable, as it can increase the sensitivity to change within the code [41]. This can result in poor modularity for the software, and as a result, impact the reusability of the code. In summary, the afferent and efferent couplings that have been measured here, are indicative of couplings *within* the inheritance hierarchy. As a result, it does not reflect as a high a sensitivity to change as the metrics would suggest.

5.3.5 Lack of cohesion in methods

A low LCOM metric means greater underlying cohesion within the class, and as a result, greater encapsulation. Conversely, a higher LCOM metric indicates a higher number of methods operating on disjoint sets of instance variables. In other words, there is a lack of similarity between the methods, and ultimately, less cohesion within the class. It follows that lack of cohesion incurs a greater level of complexity, and a higher probability of errors during the development process [41]. Since the maintainability of a system is correlated with the underlying complexity, the higher LCOM will mean a greater maintainability effort for the programmer. In figure 5.16, it is observed that the LCOM for the slice-oriented approach is 0, along with both the abstract factory and conventional techniques.

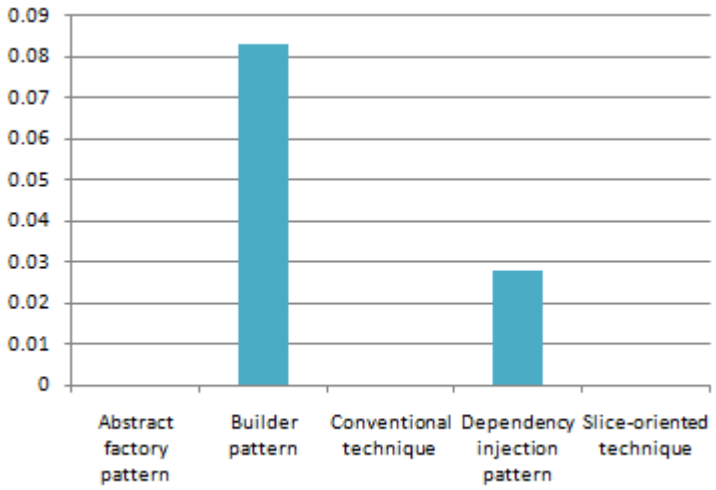

Figure 5.16: Lack of cohesion in methods

When an LCOM value of 0 is observed, there are more method pairs which don't share instance variables than methods that do share instance variables. In other words, the LCOM is undefined. Figure 5.16 shows that there were only 2 defined LCOM values, those of the builder and dependency patterns. These values were 0.083 and 0.028 respectively, which suggest strong cohesion.

5.3.6 Lines of code

Figure 5.17 shows that the LOC metric for the slice-oriented approach was the highest of all approaches, with a value of 507. The closest LOC to this was that of the Builder pattern, with a value of 389.

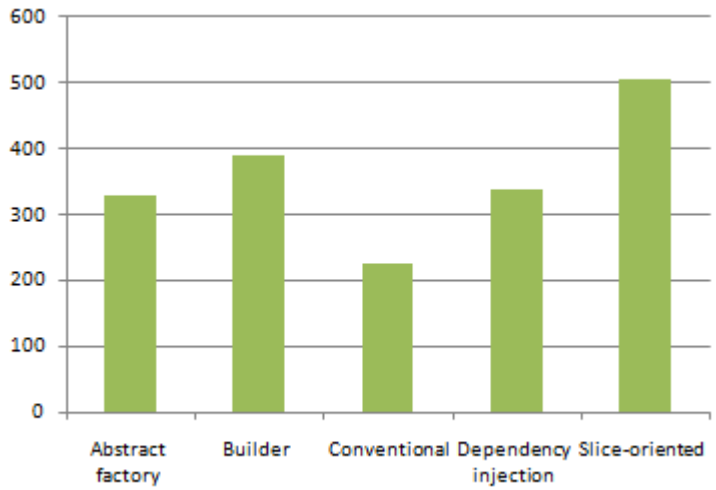

Figure 5.17: Total lines of code

Methods containing a large quantity of code, will pose problems for the programmer and will adversely affect program understandability, reusability and maintainability [49]. So, in theory, the high LOC suggests that the slice-oriented approach will not bode well for ease of maintenance. However, the high LOC value for the slice-oriented approach is a result of the additional code from the auto-generation of the 6 slices. This results in some boilerplate code that otherwise would be hidden if inheritance was used. For example, in the ImageManager class, there is a method entitled "encodeImage", which employs base64 encoding to convert an image to a String. This method is present in 3 of the 6 slices that are generated from the ImageManager class.

Since executable slices are generated from the ImageManager class, it can therefore be comparable to a superclass. If the ImageManager class was indeed a superclass, the "encodeImage" method would be compartmentalised in subclasses, and as such, would not be counted in the total LOC metric. Instead, the LOC metric for the slice-oriented

approach does count these methods, hence the higher overall LOC.

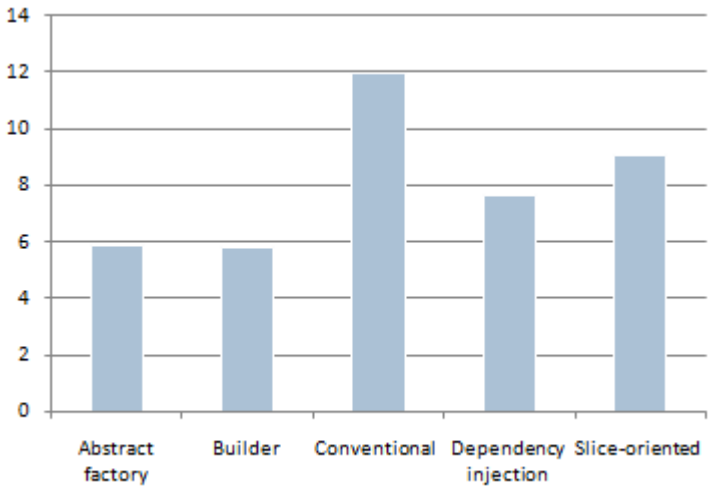

Figure 5.18: Average lines of code per method

In figure 5.18, the metrics relating to the average LOC *per method* are shown. What is evident is that the slice-oriented approach has, on average, 9 lines of code per method, which is low. A lower LOC metric per method means that the programmer incurs less effort in understanding the method. Additionally, a lower LOC metric per method can increase the longevity of a program [50].

5.4 Evaluations summary

The slice-oriented program model incurred a higher load time than most other approaches. However, this overhead meant that more auspicious performance metrics, in terms of average time taken to fulfill client service requests, could be achieved. It was shown that the slice-oriented model resulted in a higher *lines of code* metric than other approaches, since it does not rely on inheritance, unlike the abstract factory and builder approaches. The tradeoff here is that the slice-oriented model has a lower depth of inheritance tree than all other approaches. Furthermore, the average lines of code per method was lower for the slice-oriented methodology, than the conventional approach. Finally, the average cyclomatic complexity for the slice-oriented programming model was below the average of

1.64. In conclusion, this shows that the slice-oriented approach is not the most complex of all evaluated approaches. This means that the estimated maintainability effort will not be among the highest of all evaluated techniques. It is estimated that the builder and conventional approaches incur a higher maintainability effort than the slice-oriented model. It was also shown, through a practical use case, that the slice-oriented model can indeed deliver an acceptable application QoS for the end user. However, the programmer must use discretion and consider potential QoS tradeoffs carefully, when using the slice-oriented approach.

Chapter 6

Conclusions

This dissertation proposed a build-time slice-oriented programming model, to allow a programmer to specify QoS-related compositions of a web service's logic. These compositions result in slices of the overall service logic, which can be correlated with different client operational contexts. By virtue of the implementation of a slice-oriented prototype tool, it has been shown that the level of abstraction of the service composition process can indeed be raised. This in turn provides the programmer with the flexibility required for recomposing the slices at maintenance-time, without introducing additional dependencies or code modifications. Also, the evaluation of an SOA application, developed using the slice-oriented prototype tool, yielded favourable run time performance in terms of time taken to fulfill service requests. It was also shown that the generated slices can deliver an acceptable application QoS, for the service client, with regard to its operational context. In addition, the service logic composed using the slice-oriented prototype did not exhibit the greatest level of complexity, when compared with the same service logic composed using different object-oriented creational patterns. This shows that the estimated maintainability effort, for service logic composed using the slice-oriented methodology, would not be significantly higher when compared to the creational patterns. Although the evaluated service was a SOAP service, the slice-oriented methodology is scalable and could be applied to other architectures. For example, the representational state transfer (REST)

architecture [51] has been studied extensively for integrating embedded devices into the WoT [52, 53, 54, 55]. The slice-oriented model could also be applied to REST based web services. In fact, it could be applied to just about any scenario where QoS-related software compositions are required.

6.1 Future work

The slice-oriented programming model relies on the use of criteria methods for defining QoS-related logic. Criteria methods for a particular function have the exact same name and signature, since the slice-oriented model does not explicitly rely on inheritance. The other syntactic alternative for defining methods with the same name, within a single class, is method overloading. However, method overloading only allows the same method name to be reused, if the method's argument list is changed. This means that it is not a viable option for the slice-oriented methodology, as the API for the generated slices would not be consistent. Thus, the slice-oriented criteria methods must be compiled individually, which can be tedious. The alternative would be to investigate the viability of a framework, which would allow the declaration of methods with identical signatures, within the same enclosing class. This would involve a bytecode manipulation of each method name. More specifically, there would need to be a mapping from the source code method name to the underlying bytecode name. Also, the original method name would need to be preserved in the bytecode, so that polymorphism could be preserved. Updating the method names in the bytecode, would have no underlying implications for the compositions specified in the DSL.

Appendix A

Abbreviations

Short Term	Expanded Term
API	Application programming interface
AST	Abstract syntax tree
CBO	Coupling between objects
DIT	Depth of inheritance tree
DOM	Document object model
EMF	Eclipse modelling framework
IDE	Integrated development environment
IoT	Internet of Things
JDT	Java development tooling
JVM	Java virtual machine
LCOM	Lack of cohesion in methods
LOC	Lines of code
NOC	Number of children
PNG	Portable network graphics
QoS	Quality of service
REST	Representational state transfer
RTT	Round trip time
SEI	Service endpoint interface
SIB	Service implementation bean
SLA	Service level agreement
SOA	Service-oriented architecture

Short Term	Expanded Term
StAX	Streaming API for XML
UDP	User datagram protocol
UI	User interface
URI	Uniform resource identifier
WMC	Weighted methods per class
WoT	Web of Things
WSDL	Web service description language
XMI	XML metadata interchange

Appendix B

DSL Grammar

```
1  grammar org.xtext.slicing.CriteriaDsl with org.eclipse.xtext.common.Terminals
2
3  generate criteriaDsl "http://www.xtext.org/slicing/CriteriaDsl"
4
5  Model :
6      "project" projectName=STRING
7      (properties+=Property)+
8      (definitions+=Definition)+
9      (criteria+=Criterion)+
10 ;
11
12 //============ GRAMMAR FOR DEFINING CLIENT CONTEXT PROPERTIES ==================
13 Property :
14     "properties" "{"
15         (propertyDefs+=PropertyDefinition)+
16     "}"
17 ;
18
19 PropertyDefinition :
20     name=ID ":" type=typeDef
21 ;
22
23 typeDef :
24     "int"
25     | "double"
```

```
26      | "long"
27  ;
28
29  //============ GRAMMAR FOR DEFINING CLIENT PROFILES ============================
30  Definition :
31      "definitions" "{"
32          (profiles+=ProfileSpecification)+
33      "}"
34  ;
35
36  ProfileSpecification :
37      name=ID isDefault=("default")? "{"
38          (conditions+=Condition)+
39      "}"
40  ;
41
42  Condition :
43      property=[PropertyDefinition] operator=ComparisonOperation value=ConditionValue
44      (compoundCondition=SecondaryCondition)?
45  ;
46
47  SecondaryCondition :
48      compoundOperator=('&&' | '||') property=[PropertyDefinition] operator=↩
            ComparisonOperation value=ConditionValue
49  ;
50
51  ComparisonOperation :
52      ('!=' | '==' | '>' | '>=' | '<' | '<=')
53  ;
54
55  ConditionValue :
56      DOUBLE | INT
57  ;
58
59  terminal DOUBLE :
60      INT '.' (INT)+
61  ;
62
63  //============ GRAMMAR FOR DEFINING QOS-RELATED COMPOSITIONS ==================
64  Criterion :
65      "Criterion" name=[ProfileSpecification] "{"
66          (classes+=Class)+
```

```
67      "}"
68  ;
69
70  Class :
71      "class" name=ClassFQN "{"
72          ((singleProfileDeclarations+=SingleProfileDeclaration)+ | globalProfile=↩
                GlobalProfileDeclaration | isVoid=VoidDeclaration)
73      "}"
74  ;
75
76  VoidDeclaration :
77      "void"
78  ;
79
80  ClassFQN :
81      ID ("." ID)*
82  ;
83
84  SingleProfileDeclaration :
85      (methodName=ID ":" methodAnnotationName=ID)
86  ;
87
88  GlobalProfileDeclaration :
89      annotationName=ID "*"
90  ;
```

Bibliography

[1] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*, vol. 12, pp. 26–60, January 1990.

[2] M. Chui, M. Löffler, and R. Roberts, "The Internet of Things," *McKinsey Quarterly*, Mar. 2010. [Online]. Available: http://tc.indymedia.org/files/bigbrother/internet-spy-all.pdf

[3] J. Conti, "The internet of things," *Communications Engineer*, vol. 4, no. 6, pp. 20 –25, 2006.

[4] V. Stirbu, "Towards a restful plug and play experience in the web of things," *International Conference on Semantic Computing*, vol. 0, pp. 512–517, 2008.

[5] D. Guinard, "Towards opportunistic applications in a web of things," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, 292010-april2 2010, pp. 863 –864.

[6] B. Schilit and M. Theimer, "Disseminating active map information to mobile hosts," *Network, IEEE*, vol. 8, no. 5, pp. 22 – 32, 1994.

[7] T. Gu, H. K. Pung, and D. Q. Zhang, "A middleware for building context-aware mobile services," in *Vehicular Technology Conference, 2004. VTC 2004-Spring. 2004 IEEE 59th*, vol. 5, May 2004, pp. 2656 – 2660 Vol.5.

[8] J. E. Bardram, "The java context awareness framework (jcaf) - a service infrastructure and programming framework for context-aware applications," in *Pervasive*

Computing, ser. Lecture Notes in Computer Science, H. W. Gellersen, R. Want, and A. Schmidt, Eds. Springer Berlin / Heidelberg, 2005, vol. 3468, pp. 98–115.

[9] P. Papakos, L. Capra, and D. S. Rosenblum, "Volare: context-aware adaptive cloud service discovery for mobile systems," in *Proceedings of the 9th International Workshop on Adaptive and Reflective Middleware*, ser. ARM '10. New York, NY, USA: ACM, 2010, pp. 32–38.

[10] M. A. Serhani, R. Dssouli, A. Hafid, and H. Sahraoui, "A qos broker based architecture for efficient web services selection," *Web Services, IEEE International Conference on*, vol. 0, pp. 113–120, 2005.

[11] W3C. (2003) Qos for web services: Requirements and possible approaches. [Online]. Available: http://www.w3c.or.kr/kr-office/TR/2003/ws-qos/

[12] B. Smith, "Procedural reflection in programming languages," Ph.D. dissertation, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, Cambridge, MA, USA, February 1982.

[13] F.-N. Demers and J. Malenfant, "Reflection in logic, functional and object-oriented programming: a short comparative study," in *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, 1995, pp. 29–38.

[14] L. Capra, W. Emmerich, and C. Mascolo, "Carisma: Context-aware reflective middleware system for mobile applications," *IEEE Transactions on Software Engineering*, vol. 29, pp. 929–945, 2003.

[15] A. T. Chan and S.-N. Chuang, "Mobipads: A reflective middleware for context-aware mobile computing," *IEEE Transactions on Software Engineering*, vol. 29, pp. 1072–1085, 2003.

[16] M. Alia, F. Eliassen, S. Hallsteinsen, and E. Stav, "Madam: towards a flexible planning-based middleware," in *Proceedings of the 2006 international workshop on*

Self-adaptation and self-managing systems, ser. SEAMS '06. New York, NY, USA: ACM, 2006, pp. 96–96.

[17] R. Rouvoy, F. Eliassen, J. Floch, S. Hallsteinsen, and E. Stav, "Composing components and services using a planning-based adaptation middleware," in *Software Composition*, ser. Lecture Notes in Computer Science, C. Pautasso and r. Tanter, Eds. Springer Berlin / Heidelberg, 2008, vol. 4954, pp. 52–67.

[18] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*, ser. ICSE '81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.

[19] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *SIGPLAN Not.*, vol. 25, pp. 246–256, June 1990.

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, pp. 319–349, July 1987.

[21] L. Larsen and M. Harrold, "Slicing object-oriented software," *Software Engineering, International Conference on*, vol. 0, p. 495, 1996.

[22] D. Liang and M. Harrold, "Slicing objects using system dependence graphs," in *Software Maintenance, 1998. Proceedings. International Conference on*, nov 1998, pp. 358 –367.

[23] S. Sinha, M. J. Harrold, and G. Rothermel, "System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow," *Software Engineering, International Conference on*, vol. 0, p. 432, 1999.

[24] J. Zhao, "Applying program dependence analysis to java software," in *Proc. Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium*, 1998, pp. 162–169.

[25] N. Walkinshaw, M. Roper, and M. Wood, "The java system dependence graph," in *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, sept. 2003, pp. 55 – 64.

[26] J. Krinke, "Advanced slicing of sequential and concurrent programs," Ph.D. dissertation, Universitat Passau, Passau, Germany, April 2003. [Online]. Available: http://www.cs.ucl.ac.uk/staff/j.krinke/publications/krinke-phd.pdf

[27] F. Tip, "A survey of program slicing techniques," *JOURNAL OF PROGRAMMING LANGUAGES*, vol. 3, pp. 121–189, 1995.

[28] J. Qian and B. Xu, "Scenario oriented program slicing," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 748–752.

[29] A. Campbell and A. Cox, "Scenario-based program slicing," in *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, 312008-april4 2008, pp. 428 –436.

[30] G. Coulson, G. Blair, and P. Grace, "On the performance of reflective systems software," in *Performance, Computing, and Communications, 2004 IEEE International Conference on*, 2004, pp. 763 – 769.

[31] F. Curbera, W. A. Nagy, and S. Weerawarana, "Web services: Why and how," in *In OOPSLA 2001 Workshop on Object-Oriented Web Services. ACM*, 2001.

[32] O. Nierstrasz and L. Dami, *Component-oriented software technology.* Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1995, pp. 3–28.

[33] O. Nierstrasz and T. Meijler, "Requirements for a composition language," in *Object-Based Models and Languages for Concurrent Systems*, ser. Lecture Notes in Computer Science, P. Ciancarini, O. Nierstrasz, and A. Yonezawa, Eds. Springer Berlin / Heidelberg, 1995, vol. 924, pp. 147–161.

[34] J. Gray and G. Karsai, "An examination of dsls for concisely representing model traversals and transformations," in *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, jan. 2003, p. 10 pp.

[35] T. Wang and A. Roychoudhury, "Using compressed bytecode traces for slicing Java programs," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2004, pp. 512–521.

[36] G. Jayaraman, V. Ranganath, and J. Hatcliff, "Kaveri: Delivering the indus java program slicer to eclipse," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, M. Cerioli, Ed. Springer Berlin / Heidelberg, 2005, vol. 3442, pp. 269–272.

[37] W3C, "Soap Version 1.2," www.w3.org/TR/soap/, 2007. [Online]. Available: http://www.w3.org/TR/soap/

[38] R. D. Banker, S. M. Datar, and D. Zweig, "Software complexity and maintainability," in *Proceedings of the Tenth International Conference on Information Systems*, 1989, pp. 247–255.

[39] R. Hudli, C. Hoskins, and A. Hudli, "Software metrics for object-oriented designs," in *Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. Proceedings., IEEE International Conference on*, oct 1994, pp. 492 –495.

[40] W. Harrison, K. Magel, R. Kluczny, and A. DeKock, "Applying software complexity metrics to program maintenance," *Computer*, vol. 15, no. 9, pp. 65 – 79, sep 1982.

[41] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," in *Conference proceedings on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '91. New York, NY, USA: ACM, 1991, pp. 197–211.

[42] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics

as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, no. 10, pp. 751 –761, oct 1996.

[43] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111 – 122, 1993. [Online]. Available: http://www.sciencedirect.com/science/article/pii/016412129390077B

[44] A. Albrecht and J. Gaffney, J.E., "Software function, source lines of code, and development effort prediction: A software science validation," *Software Engineering, IEEE Transactions on*, vol. SE-9, no. 6, pp. 639 – 648, nov. 1983.

[45] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308 – 320, dec. 1976.

[46] D. Tegarden, S. Sheetz, and D. Monarchi, "Effectiveness of traditional software metrics for object-oriented systems," in *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, vol. iv, jan 1992, pp. 359 –368 vol.4.

[47] M. Dagpinar and J. Jahnke, "Predicting maintainability with object-oriented metrics -an empirical comparison," in *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, nov. 2003, pp. 155 – 164.

[48] C. Jones, "Software metrics: good, bad and missing," *Computer*, vol. 27, no. 9, pp. 98 –100, sep 1994.

[49] L. H. Rosenberg and E. H. Lawrence, "Software quality metrics for object-oriented environments," *Crosstalk*, vol. 10, 1997.

[50] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*, ser. Addison-Wesley object technology series. Addison-Wesley, 2001.

[51] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, Irvine, California,

2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[52] D. Guinard and V. Trifa, "Towards the web of things: Web mashups for embedded devices," in *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences)*, Madrid, Spain, Apr. 2009.

[53] B. Ostermaier, F. Schlup, and M. Kovatsch, "Leveraging the web of things for rapid prototyping of ubicomp applications," in *Proceedings of the 12th ACM international conference adjunct papers on Ubiquitous computing*, ser. Ubicomp '10. New York, NY, USA: ACM, 2010, pp. 375–376.

[54] E. W. (school Of Information and U. Berkeley, "Putting things to rest," 2007. [Online]. Available: http://escholarship.org/uc/item/1786t1dm.pdf

[55] B. Ostermaier, F. Schlup, and K. Romer, "Webplug: A framework for the web of things," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, 292010-april2 2010, pp. 690 –695.