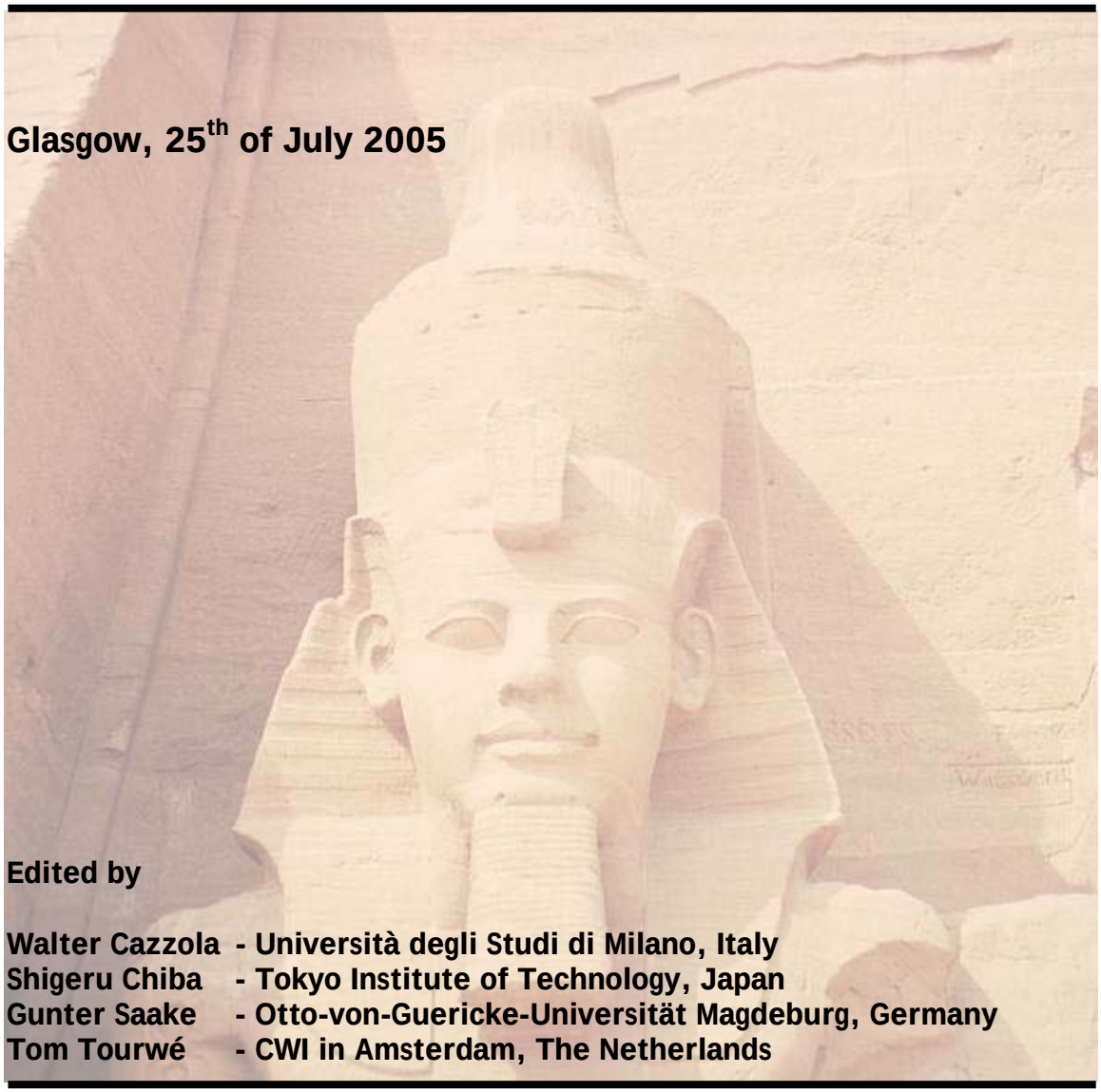
RAM-SE'05 – ECOOP'05 Workshop on Reflection, AOP, and Meta-Data for Software Evolution

(Proceedings)



Glasgow, 25th of July 2005

Edited by

Walter Cazzola - Università degli Studi di Milano, Italy
Shigeru Chiba - Tokyo Institute of Technology, Japan
Gunter Saake - Otto-von-Guericke-Universität Magdeburg, Germany
Tom Tourwé - CWI in Amsterdam, The Netherlands

Foreword

Software evolution and adaptation is a research area, as also the name states, in continuous evolution, that offers stimulating challenges for both academic and industrial researchers. The evolution of software systems, to face unexpected situations or just for improving their features, relies on software engineering techniques and methodologies. Nowadays a similar approach is not applicable in all situations e.g., for evolving nonstopping systems or systems whose code is not available.

Reflection and aspect-oriented programming are young disciplines that are steadily attracting attention within the community of object-oriented researchers and practitioners. The properties of transparency, separation of concerns, and extensibility supported by reflection and aspect-oriented programming have largely been accepted as useful for software development and design. Reflective features have been included in successful software development technologies such as the Java language and the .NET framework. Reflection has proved to be useful in some of the most challenging areas of software engineering, including Component-Based Software Development (CBSD), as demonstrated by extensive use of the reflective concept of introspection in the Enterprise JavaBeans component technology.

Features of reflection such as transparency, separation of concerns, and extensibility seem to be perfect tools to aid the dynamic evolution of running systems. They provide the basic mechanisms for adapting (i.e., evolving) a system without directly altering the existing system. Aspect-oriented programming can simplify code instrumentation providing a few mechanisms, such as the join point model, that permit of evincing some points (*join points*) in the code or in the computation that can be modified by weaving new functionality (aspects) on them in a second time. Meta-data represent the glue between the system to be adapted and how this has to be adapted; the techniques that rely on meta-data can be used to inspect the system and to dig out the necessary data for designing the heuristic that the reflective and aspect-oriented mechanisms use for managing the evolution.

It is our belief that current trends in ongoing research in reflection, aspect-oriented programming and software evolution clearly indicate that an inter-disciplinary approach would be of utmost relevance for both. Therefore, we felt the necessity of investigating the benefits that the use of these techniques on the evolution of object-oriented software systems could bring. In particular we were and we continue to be interested in determining how these techniques can be integrated together with more traditional approaches to evolve a system and in discovering the benefits we get from their use.

Software engineering may benefit from a cross-fertilization with reflection and aspect-oriented programming in several ways. Reflective features such as transparency, separation of concerns, and extensibility are likely to be of increasing relevance in the modern software engineering scenario, where the trend is towards systems that exhibit sophisticated functional and non-functional requirements; that are built from independently developed and evolved COTS (commercial off-the-shelf) components; that support plug-and-play, end-user directed reconfigurability; that make extensive use of networking and internetworking; that can be automatically upgraded through the Internet; that are open; and so on. Several of these issues bring forth the need for a system to manage itself to some extent, to inspect components' interfaces dynamically, to augment its application-specific functionality with additional properties, and so on. From a pragmatic point of view, several reflective and aspect-oriented techniques and technologies lend themselves to be employed in addressing these issues. On a more conceptual level, several key reflective and aspect-oriented principles could play an interesting role as general software design and evolution principles. Even more fundamentally, reflection and aspect-oriented programming may provide a cleaner conceptual framework than that underlying the rather 'ad-hoc' solutions embedded in most commercial platforms and technologies, including CBSD technologies, system management technologies, and so on. The transparent nature of reflection makes it well suited to address problems such as evolution of legacy systems, customizable software, product families, and more. The scope of application of reflective and aspect-oriented concepts in software evolution conceptually spans activities related to all the phases of software life-cycle, from analysis and architectural design to development, reuse, maintenance, and, therefore also evolution.

The overall goal of this workshop – as well as of its previous edition – was that of supporting circulation of ideas between these disciplines. Several interactions were expected to take place between reflection, aspect-oriented programming and meta-data for the software evolution, some of which we cannot even foresee. Both the application of reflective or aspect-oriented techniques and concepts to software evolution are likely to support improvement and deeper understanding of these areas. This workshop has represented a good meeting-point for people working in the software evolution area, and an occasion to present reflective, aspect-oriented, and meta-data based solutions to evolutionary problems, and new ideas straddling these areas, to provide a discussion forum, and to allow new collaboration projects to be established. The workshop is a full day meeting. One part of the workshop will be devoted to presentation of papers, and another to panels and to the exchange of ideas among participants.

In this second edition of the workshop, we had an interesting keynote by Oscar Nierstrasz on *the revival of the dynamic languages*. This keynote was an interesting experiment that has raised several issues and lively discussion among the workshop attendees. To the interested reader, more on this keynote can be read in the paper:

– Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gälli and Roel Wuyts. On the Revival of Dynamic Languages. In the Proceedings of Software Composition 2005. Lecture Notes in Computer Science 3628, pages 1-13. 2005.

This volume gathers together all the position papers accepted for presentation at the second edition of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05), held in Glasgow on the 25th of July, during the ECOOP'05 conference. We have received many interesting submission and due to time restrictions and to quality insurance we had to choice few of them, the papers that, in our opinion, are more or less evidently interrelated to feed up a more lively discussion during the workshop. Now, few months after the workshop, we can state that we achieved our goal, presentations were interesting and the subsequent panels grew up lively and rich of ideas and proposals. We are sure that in the next months we will see many papers by the workshop attendees and fruit of such a lively discussions.

The success of the workshop is mainly due to the people that have attended it and to their effort to participate to the discussions. The following is the list of the attendees in alphabetical order.

Apel, Sven	Jędrzejek, Czesław	Rank, Stephen
Bencomo, Nelly	Le Botlan, Didier	Rashid, Awais
Chitchyan, Ruzanna	Leich, Thomas	Reinsch, Michael
Coady, Yvonne	Mohd Ali, Noorazean	Staijen, Tom
Cointe, Pierre	Mosconi, Marco	Südholt, Mario
Ebraert, Peter	Nierstrasz, Oscar	Vandewoude, Yves
Gibbs, Celina	Ostermann, Klaus	Watanabe, Takuo
Havinga, Wilke	Pini, Sonia	Weston, Nathan
Hutchins, DeLesley		

A special thank is for the three chairmen (Yvonne Coady, Oscar Nierstrasz, and Takuo Watanabe) that governed the panels at the end of each session.

We have also to thank the Department of Informatics and Communication of the University of Milan, the Department of Mathematical and Computing Sciences of the Tokyo institute of Technology and the Institute für Technische und Betriebliche Informationssysteme, Otto-von-Guericke-Universität Magdeburg for their various supports.

October 2005

W. Cazzola, S. Chiba, G. Saake and T. Tourwé
RAM-SE'05 Organizers

Contents

Mechanisms for Supporting Software Evolution

Technological Limits for Software Evolution

Tools and Middleware for Software Evolution

Mechanisms for Supporting Software Evolution

Chairman: Oscar Nierstrasz, Universität Bern, Switzerland

Combining Feature-Oriented and Aspect-Oriented Programming to Support Software Evolution

Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake

Department of Computer Science
University of Magdeburg, Germany
email: {apel,leich,rosenmue,saake}@iti.cs.uni-magdeburg.de

Abstract. Starting from the advantages of using Feature-Oriented Programming (FOP) and program families to support software evolution, this paper discusses the drawbacks of current FOP techniques. In particular we address the insufficient crosscutting modularity that complicates software evolution. To overcome this tension we propose the integration of concepts of Aspect-Oriented Programming (AOP) into existing FOP solutions. As study object we utilize FeatureC++, a proprietary extension to C++ that supports FOP. After a short introduction to basic language features of FeatureC++, we summarize the problems regarding the crosscutting modularity. In doing so, we point to the strengths of AOP that can help. Thereupon, we introduce three approaches that combine FOP and AOP concepts: *Multi Mixins*, *Aspectual Mixins*, and *Aspectual Mixin Layers*. Furthermore, we discuss their benefits for software evolution.

1 Introduction

Nowadays software is subject to frequent changes in order to react to altering and evolving requirements. The process of continuous adaptation, extension, and customization is known as *software evolution*. This article focuses on the evolution of the design and the implementation base. The idealized goal of software engineers is to reuse as much as possible code from previous development stages to build a new version of the software. To achieve this, software must be designed reusable, extensible, and customizable. A heavily discussed approach to implement software with such virtues to support software evolution are program families [18]. Program families group programs with similar functionalities in families. The key idea is to arrange the design and implementation as a layered stack of functionalities. Different programs consist of different layers. Thus, implemented layers can be reused in multiple programs. A fine-grained layered architecture leads to reusable, extensible, and customizable software [18]. Representative studies in the domains of databases [4], middleware [1], avionics [3], and network protocols [4] show that *Feature-Oriented Programming (FOP)* [5]

and *Mixin Layers* [21] are appropriate to implement such layered, step-wise refined architectures. However, FOP[1] yields some problems in expressing features and evolving software:

1. FOP lacks adequate crosscutting modularity. During the evolution, software have to be adapted to fit unanticipated requirements and circumstances. This results in modifications and extensions that crosscut many existing implementation units in numerous ways [13].
2. Currently FOP is still an academic concept that is not widely accepted in the industry. We argue that is because of the focus on Java that is not acceptable in many domains, e.g. operating systems, databases, middleware, realtime embedded systems, etc. Even these domains demand for appropriate support of software evolution. Currently, C++-based solutions are too complex and hard to use [21, 19]. Moreover, an adequate tool support is missing.

Consequently, our contribution is to solve both problems, supporting crosscutting modularity and using C++ as base language. We have developed FeatureC++[2], an extension to C++ that supports FOP [2]. This article focuses primarily on the first problem and presents our investigations in solving the problem of insufficient crosscutting modularity. FeatureC++ serves as study object and representative FOP language. A detailed introduction to FeatureC++ is given in [2]. Our approach to improve the crosscutting modularity is to combine traditional FOP concepts with concepts of *Aspect-Oriented Programming (AOP)* [13]. AOP focuses on the separation and modularization of crosscutting concerns and is therefore best qualified to improve FOP. We have elaborated three ways to integrate AOP concepts into FOP: *Multi Mixins, Aspectual Mixins, Aspectual Mixin Layers*. This article introduces and compares them, as well as discusses their pros and cons with regard to software evolution.

The remaining article is structured as follows: Section 2 gives some background information about FOP and AOP. Section 3 introduces the basic language concepts of FeatureC++. Thereupon, Section 4 reviews the problems of FOP in modularizing crosscutting concerns. In this regard, we point to the advantages of AOP to solve these problems. Section 5 introduces our three approaches to combine AOP and FOP, and discusses theirs pros and cons. Afterwards, Section 6 reviews a selection of related work. Finally, Section 7 gives a conclusion.

2 Background

Pioneer work on software modularity was made by Dijkstra [11] and Parnas [18]. They have proposed the principle of *separation of concerns*. The idea is to separate each concern of a software system in a separate modular unit. They argue

[1] In the remaining article we presume that Mixin Layers are used to implement feature-oriented programs.

[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

that this lead to maintainable, comprehensible software, which can be easily reused, customized, and evolved.

AOP was introduced by Kiczales et al. [13]. The aim of AOP is to separate crosscutting concerns. Common object-oriented methods fail in this context [13, 10]. The idea behind AOP is to implement so called orthogonal features as *Aspects*. This prevents the known phenomena of code tangling and scattering. The core features are implemented as components, as with common design and implementation methods. Using join point specifications (*pointcuts*), an aspect weaver brings aspects and components together. Due to the ability to implement unanticipated features in a modular way AOP is an important technique to ease software evolution [12]. *AspectJ*[3] and *AspectC++*[4] are prominent AOP extensions to Java and C++.

FOP studies feature modularity in program families [5]. The idea of FOP is to build software by composing *features*. Features are basic building blocks that satisfy intuitive user-formulated requirements on the software system. Features refine other features incrementally. This *step-wise refinement* leads to a layered stack of features. *Mixin Layers* are one appropriate technique to implement features and layered designs [21]. The basic idea is that features are often implemented by a collaboration of class fragments (a.k.a. *roles*). A Mixin Layer is a static component encapsulating fragments of several different classes (*Mixins*) so that all fragments are composed consistently. Advantages are the high degree of modularity and the easy composition [21]. *AHEAD* is an architectural model for FOP and a basis for large-scale compositional programming [5]. It extends the concept of FOP to all software artifacts, e.g. UML diagrams, documentation, etc. It makes a broad consistent software evolution possible. The *AHEAD Tool Suite (ATS)*[5], including the *Jak* language, implements AHEAD for Java.

3 Overview of FeatureC++

This section gives a short overview of FeatureC++. For a more detailed introduction we refer to [2].

3.1 Introduction to Basic Concepts

In order to implement FeatureC++, we have adopted the basic concepts of the ATS: Features are implemented by Mixin Layers. A Mixin Layer consists of a set of collaborating Mixins (which implement class fragments). Figure 1 depicts a stack of three Mixin Layers $(1-3)$ in top down order. The Mixin Layers crosscut multiple classes $(A-C)$. The rounded boxes represent the Mixins. These Mixins that belong to and constitute together a complete class are called *refinement chain*. Solid lines represent refinement relationships and connect refinement chains (Fig. 1). Roots of a refinement chain are called *constants*; All

[3] http://eclipse.org/aspectj/
[4] http://www.aspectc.org/
[5] http://www.cs.utexas.edu/users/schwartz/Hello.html

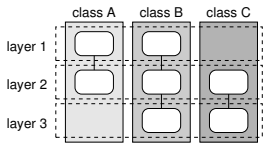


Fig. 1. Stack of Mixin Layers.

other Mixins are called *refinements*. A Mixin A that is refined by Mixin B is called *parent* Mixin or parent class of Mixin B. Consequently, Mixin B is the *child* class or child Mixin of A. Similarly, we speak of parent and child Mixin Layers. In FeatureC++ Mixin Layers are represented by file system directories. Therefore, FeatureC++ represents them not explicitly (this follows the principle of AHEAD). Those Mixins, found inside the directories are assigned to be the members of the enclosing Mixin Layer.

3.2 Syntax of Basic Language Features

FeatureC++ adopts the syntax of the Jak language [5]. The following paragraphs introduce the most important language features by example, a buffer that serializes and stores objects.

Constants and Refinements. Each constant and refinement is implemented as a Mixin inside exactly one source file. Each constant is the root of a chain of refinements (see Fig. 2).

```
1 class Buffer {
2   char *buf;
3   void put(char *s) { /* ... */ }
4 };
```

Fig. 2. Defining a basic buffer.

```
1 refines class Buffer {
2   int length;
3   int getLength() { /* ... */ }
4 };
```

Fig. 3. Adding a length attribute and an access method.

```
1 refines class Buffer {
2   void put(char *s) {
3     if(strlen(s) + getLength() < MAX_LEN)
4       super::put(s);
5   }
6 };
```

Fig. 4. Limiting the buffer length.

Refinements refine constants as well as other refinements. They are declared by the keyword *refines* (see Fig. 3). Usually, they introduce new members attributes and methods (Lines 2-3).

Extending Methods. Refinements can extend[6] methods of their parent classes (see Fig. 4). To access the extended method the *super* keyword is used (Line 4). *Super* refers to the type of the parent Mixin. It has a similar semantic to the Java *super* keyword and is related to the *proceed* keyword of AspectJ and AspectC++.

Further Language Features. Due to the space limitations, we omit a discussion of the below listed language feature of FeatureC++. A detailed introduction can be found in [2].

- FeatureC++ supports multiple inheritance, templates for generic programming, accessing overloaded methods from extern, as well as refinements of static methods, structs, and destructors.
- FeatureC++ solves several problems regarding class hierarchy extensions that are caused by the divergence of variations and extensions.
- FeatureC++ solves the constructor problem that occurs in incremental designs and results in unnecessary constructor redefinitions (cf. [20]).

4 Problems of FOP and how AOP could help

This section reviews problems of FOP regarding crosscutting modularity and software evolution. The purpose of FOP is to implement program families. Commonly, their design and implementation is well planned. FOP yields promising results in this respect (see [4, 3, 6, 7]). However, problems occur in implementing unanticipated features: We argue that the frequently needed, unanticipated modifications and extensions of evolving software cause code tangling and code scattering. Mostly these new features are crosscutting concerns, and FOP is not able to modularize them all appropriately (as we will see soon). From this point of view we perceive the solution to the problem of insufficient crosscutting modularity as an improvement for software evolvability. The following paragraphs introduce the key problems and point to strengths of AOP in these respects. The discussion of the problems extends [17, 2].

[6] With 'extend' we refer to overriding and to call the overridden method.

Homogeneous vs. Heterogeneous Crosscuts. Homogeneous crosscutting concerns are distributed over several join points but apply every time the same code, e.g. logging; Heterogeneous crosscuts apply varying code, e.g. authentication [8]. Common AOP languages focus on homogeneous concerns whereas FOP languages deal with heterogeneous concerns. Indeed, both language paradigms can deal with both types of concerns but often this results in complicated code, code redundancy, and inelegant workarounds. However, both are important for software evolution. Consequently, our objective is to enhance FOP with the opportunity to handle homogeneous concerns in an adequate way.

Static vs. Dynamic Crosscutting. Both FOP and AOP deal with dynamic crosscutting[7]. Dynamic crosscutting affects the runtime behavior and depends on the control flow. Static crosscutting affects the static structure of a base feature. We argue, however, that the way AOP deals with dynamic crosscutting, namely by using pointcut expressions and advices, is more expressive. Feature binding specifications as "bind feature A to all calls to method m that are in the control flow of method c and only if expression e is true" are difficult to express in FOP languages. With regard to software evolution, we argue the more complex a software becomes (as this is the case of evolving software) the more the programmer needs to specify such complex feature bindings.

Hierarchy-Conforming Refinements. Using FOP, feature refinements depend on the structure of parent features. Usually, a feature refines a set of classes and extends methods. For each implementation unit we want to refine, we have to introduce a new unit. In fact, the programmer is forced to express new features in terms of structural elements of the existing features. This becomes problematic if new features are implemented at a different abstraction level. AOP is able to implement non-hierarchy-conforming refinements by using wildcards in pointcut expressions [17]. The problem of a raising abstraction level is serious to evolving software because at the beginning of building software the abstraction of subsequent development phases cannot be foreseen. If the programmer is forced to express new features using abstractions of former features the code becomes unnecessary complicated, bloated, and difficult to understand.

We clarify this by an example (adopted from [17]). As basic feature we consider a stock information broker. This feature should be refined by a pricing feature. Whereas the broker is expressed in terms of stock information, requests, brokers, clients and database connections, the pricing feature is expressed using the intuitive product-consumer-pattern. FOP is not able to change the abstraction level accordingly. Instead, AOP is able to implement non-hierarchy-conforming refinements by using wildcards in pointcut expressions [17].

Excessive Method Extensions. The problem of excessive method extensions occurs (1) if a feature crosscuts a large fraction of existing implementation units and (2) if it is a homogeneous concern. For instance, if a feature wants to add

[7] Note that dynamic crosscutting is not dynamic weaving.

multi-threading support, it has to extend lots of methods, and adds synchronization code. This code is in almost all methods the same and therefore redundant, e.g. setting lock variables. AOP deals with this problem by using wildcards in pointcut expressions to specify a set of target methods (join points). This prevents code redundancies and eases software evolution.

Method Interface Extensions. The problem of method interface extensions frequently occurs in incremental designs. As an extended interface we understand an extended argument list. This problem occurs if refinements require additional parameters, e.g. an additional session id or a reference to a locking variable. Indeed, using some workaround this problem could be avoided. But AOP with its pointcut mechanism is much more elegant [17].

Unpredictable Aspect Composition. This problem regards AOP languages only. Nevertheless it is of importance because we want to integrate AOP mechanisms into FOP. The problem of current AOP languages is that the binding of aspects is independent of the current development stage. That means an aspect may affect subsequent integrated features. This can lead to unpredicted effects, e.g. an aspect is unintentionally bound to new features. In [15] an alternative composition mechanism is proposed. They argue that with regard to software (program family) evolution, features should only affect features of prior development stages. Current AOP languages, e.g. AspectJ and AspectC++, do not follow this principle. This decreases aspect reuse and complicates incremental design. Consequently, our approaches satisfy this principle.

5 Enhancing FOP with AOP concepts

This section presents our first results in integrating AOP concepts into FOP in order to support software evolution. The presented approaches show that there are numerous ways to implement that symbiosis.

5.1 Multi Mixins

Our first idea to prevent a programmer from excessive method extensions, hierarchy-conforming refinements, and to support homogeneous crosscuts were *Multi Mixins*. The key idea, instead of refining one Mixin by another one Mixin only, is to refine a whole set of parent Mixins. Such sets are specified by wildcards ('%') adopted from AspectC++. Both Multi Mixins, depicted in Figure 5, use wildcards to specify the Mixins and methods they refine. The first refines all classes that start with "*Buffer*" (Line 1). The second refines all methods of Buffer that start with "*put*" (Line 3-5). The meaning of the first type of refinement is straight forward: The wildcard *Buffer%* has the same effect as one creates a set of new refinements for each found Mixin that matches the pattern (*Buffer%*). This type of Multi Mixin eases the implementation of static homogeneous features in FOP.

```
1 refines class Buffer% { /* ... */ };
2
3 refines class Buffer {
4   void put%(...) { /* ... */ }
5 };
```

Fig. 5. Two Multi Mixins that refine sets of Mixins and methods.

The second type of Multi Mixins, which refines methods, eases the expression of dynamic homogeneous features. Similar to pointcuts and advices in AOP languages, one code fragment can be assigned to multiple methods. However, with Multi Mixins it is not possible to implement *execution* or *cflow* pointcuts.

5.2 Aspectual Mixin Layers

The idea behind *Aspectual Mixin Layers* is to embed aspects into Mixin Layers. Each Mixin Layer contains a set of Mixins and a set of aspects. Hence, Mixins implement heterogeneous and hierarchy-conforming crosscutting, whereas aspects express homogeneous and non-hierarchy-conforming crosscutting. In other words, Mixins refine other Mixins and depend, therefore, on the structure of the parent layer. These refinements follow the static structure of the parent features and encapsulate heterogeneous crosscuts. Aspects refine a set of parent Mixins by intercepting method calls and executions as well as attribute accesses. Therefore, aspects encapsulate homogeneous and non-hierarchy-conforming refinements. Furthermore, they support advanced dynamic crosscutting.

Figure 6 shows a stack of Mixin Layers that implement some buffer functionality, in particular, a basic buffer with iterator, a separated allocator, synchronization, and logging support.

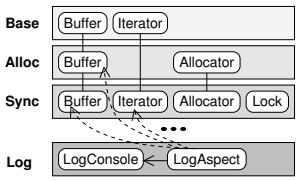


Fig. 6. Implementing a logging feature using Aspectual Mixin Layers.

Whereas the first three features are implemented as common Mixin Layers, the *Logging* feature is implemented as an Aspectual Mixin Layer. It consists of a logging aspect and a logging console. The logging console prints out the logging stream and is implemented using a common Mixin. The logging aspect captures a set of methods that will be refined with logging code (dashed arrows).

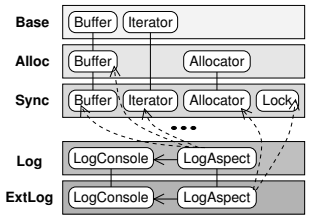


Fig. 7. Refining an Aspectual Mixin Layer.

```
1 refines aspect LogAspect {
2   void print() {
3     changeFormat();
4     super::print();
5   }
6   pointcut log() = call("%␣%::get()") || super::log();
7 };
```

Fig. 8. An aspect embedded into a Mixin Layer.

This refinement is homogeneous, non-hierarchy-conforming, and depends on the runtime control flow (dynamic crosscutting). Moreover, the use of wildcards prevents the programmer of excessive method extensions. Without Aspectual Mixin Layers the programmer has to extend all target methods manually.

A further highlight of Aspectual Mixin Layers is that aspects can refine other aspects. Figure 7 shows an Aspectual Mixin Layer that refines the logging aspect by additional join points to extend the set of intercepted methods. Additionally, the logging console is refined by additional functionality, e.g. a modified output format.

Aspects can refine the methods of parents aspect as well as the parent pointcuts. This allows to easily reuse and extend of existing join point specifications (as in the logging example). Note that refining/extending aspects is conceptually different than applying aspects themselves. Whereas the former case results in a transformation of the aspect code before applying them to the target program, the latter case applies the aspects in two steps which leads to two independent aspect instances.

To express aspects in Aspectual Mixin Layers we adopt the syntax of AspectC++. Figure 8 depicts an aspect refinement that extends a logging feature including a logging aspect. It overrides a parent method in order to adjust the output format (Line 2-5) and refines a parent pointcut to extend the set of target join points (Line 6). Both is done using the *super* keyword.

```
1 refines class Buffer {
2   int length() { /* ... */ }
3   pointcut log() = call("% Buffer::%(...)");
4 };
```

Fig. 9. Combining Mixins and AOP elements.

5.3 Aspectual Mixins

The idea of *Aspectual Mixins* is to apply AOP language concepts directly to Mixins. In this approach, Mixins refine other Mixins as with common FeatureC++ but also define pointcuts and advices (see Fig. 9). In other words, Aspectual Mixins are similar to Aspectual Mixin Layers but integrate pointcuts and advices directly into its Mixin definition. In the following, we discuss only the important differences:

The set of pointcuts, advices, and aspect-specific attributes and methods is called *aspectual subset* of the *overall* Mixin. This mixture of AOP concepts and Mixins reveals some interesting issues: Using Aspectual Mixins the instantiation of aspects is triggered by the overall Mixin instances. Regarding the above presented example, the buffer Mixin (Fig. 9, Lines 1-4) and its aspectual subset (Line 3) are instantiated as many times as the buffer. This corresponds to the *perObject* qualifier of AspectJ. However, in many cases only one aspect instance is needed. To overcome this problem, we think of introducing a *perObject* and *perClass* qualifier to distinguish these cases. This introduces a second problem: If an aspect, part of an Aspectual Mixin, uses non-static members of the overall Mixin it depends on the Mixin instance. In this case, it is forbidden to use the *perClass* qualifier. FeatureC++ must guarantee that *perClass* Aspectual Mixins, especially their aspectual subset, only access static members of the overall Mixin instance. In case of *perObject* Aspectual Mixins this is not necessary.

5.4 Discussion

All three approaches provide solutions for problems of FOP with crosscutting modularity discussed in Section 4:

- support homogeneous and heterogeneous crosscuts (1)
- extended dynamic crosscutting (pointcuts, etc.) (2)
- non-hierarchy-conforming refinements (3)
- prevent excessive method extensions (4)
- handling method interface extensions (5)

Table 1 summarizes the improvements to FOP with respect to the above presented problems.

approach	(1)	(2)	(3)	(4)	(5)
Multi Mixins	√	–	√	√	(√)
Aspectual Mixin Layers	√	√	√	√	√
Aspectual Mixins	√	√	√	√	√

Table 1. Evaluation of approaches.

5.5 Bounding Quantification.

A further highlight of all three AOP extensions is a specific bounding mechanism that supports a better incremental design and that prevents unpredictable aspect composition (cf. Sec. 4). This mechanism bounds aspects and their effects on the target program. To implement this bounding mechanism the user-declared join point specifications must be restructured: Type names in wildcards are translated in order to match only these types that are declared by the current and the parent layers. Each wildcard expression that contains a type name is translated into a set of new expressions that refer to all type names of the parent classes. Figure 10 shows a synchronization aspect that is part of an Aspectual Mixin Layer. It has two parent layers (*Base*, *Log*) and several child layers. Using this novel bounding mechanism, FeatureC++ transforms the aspect and the pointcut as depicted in Figure 11. This transformation works similar for Aspectual Mixins. In case of Multi Mixins we have to add a mechanism for combining wildcard expression logically.

```
1 aspect SyncAspect {
2   pointcut sync() : call("% Buffer::put(...)");
3 };
```

Fig. 10. A synchronization aspect with a simple pointcut expression.

```
1 aspect SyncAspect_Sync {
2   pointcut sync() : call("% Buffer_Sync::put(...)")
3             || call("% Buffer_Log::put(...)")
4             || call("% Buffer_Base::put(...)");
5 };
```

Fig. 11. Bounding quantification by transforming pointcuts.

Finally, we want to emphasize that all three approaches are not specific to FeatureC++. All concepts can be applied to other AOP/FOP languages.

6 Related Work

Several approaches aim to combine AOP and FOP. Mezini et al. argue that using AOP as well as FOP standalone lacks feature modularity [17]. They propose *Caesar* as combined approach. Similar to FeatureC++, Caesar supports dynamic crosscutting using pointcuts. Instead of FeatureC++, the focus of Caesar is on aspect reuse and on-demand remodularization. *Aspectual Collaborations* proposed by Lieberherr et al. [14] encapsulate aspects into modules, with expected and provided interfaces. The rationales behind this approach are similar to Caesar. Colyer et al. propose the *principle of dependency alignment*: a set of guidelines for structuring features in modules and aspects with regard to program families [9]. They distinguish between orthogonal and weak-orthogonal features/concerns. Loughran et al. support the evolution of program families with *Framed Aspects* [16]. They combine the advantages of frames and AOP, to serve unanticipated requirements.

7 Conclusion

In this paper we argued that common FOP techniques are important for software evolution and appropriate for implementing program families. However, we discussed the drawbacks regarding crosscutting modularity and the missing support of C++. We stated that the shortcomings in the crosscutting modularity cause problems in implementing unanticipated features. Often, these features are wide-spread crosscutting concerns. The discussed problems of FOP in these regards complicate the evolution of software. Consequently, we have presented our approach: FeatureC++ supports FOP in C++ and solves several problems regarding the lacking crosscutting modularity by adopting AOP concepts. In this paper, we have focused on solutions to these problems to ease evolvability of software. We have summarized the problems of FOP, advantages of AOP in these respects, and presented three approaches to solve these problems: Multi Mixins, Aspectual Mixins and Aspectual Mixin Layers. Whereas, the first two approaches are only of conceptual nature, we have implemented the third approach and enhanced FeatureC++ with the ability to express Aspectual Mixin Layers. A first prototype can be found at the FeatureC++ web site[8]. In ongoing work we will apply all three approaches to real-world case studies.

References

1. S. Apel and K. Böhm. Towards the Development of Ubiquitous Middleware Product Lines. In *Proceedings of the ASE Workshop on Software Engineering and Middleware (SEM)*, volume 3437 of *Lecture Notes on Computer Science*. Springer, 2005.

[8] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

2. S. Apel et al. FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++. Technical report, Deptartment of Computer Science, Otto-von-Guericke University, Magdeburg, Germany, 2005.
3. D. Batory et al. Creating Reference Architectures: An Example from Avionics. In *Proceedings of the Symposium on Software Reusability (SSR)*, 1995.
4. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4), 1992.
5. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6), 2004.
6. D. Batory and J. Thomas. P2: A Lightweight DBMS Generator. *Journal of Intelligent Information Systems*, 9(2), 1997.
7. R. Cardone et al. Using Mixins to Build Flexible Widgets. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, 2002.
8. A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, 2004.
9. A. Colyer, A. Rashid, and G. Blair. On the Separation of Concerns in Program Families. Technical Report COMP-001-2004, Computing Department, Lancaster University, 2004.
10. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
11. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
12. R. Filman et al. *Aspect-Oriented Software Development*. Addison Wesley, 2004.
13. G. Kiczales et al. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.
14. K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5), 2003.
15. R. E. Lopez-Herrejon and D. Batory. Improving Incremental Development in AspectJ by Bounding Quantification. In *Software Engineering Properties and Languages for Aspect Technologies*, 2005.
16. N. Loughran et al. Supporting Product Line Evolution with Framed Aspects. In *Proceedings of the AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2004.
17. M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT Foundations of Software Engineering (FSE)*, 2004.
18. D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering (TSE)*, SE-5(2), 1979.
19. V. Singhal and D. Batory. P++: A Language for Large-Scale Reusable Software Components. In *Proceedings of the Workshop on Software Reuse*, 1993.
20. Y. Smaragdakis and D. Batory. Mixin-Based Programming in C++. In *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)*, 2000.
21. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2), 2002.

Modular Aspect Verification for Safer Aspect-Based Evolution

Nathan Weston, Francois Taiani, Awais Rashid

Computing Department, InfoLab21, Lancaster University, UK.
{westonn,f.taiani,marash}@comp.lancs.ac.uk

Abstract. A long-term research goal for Aspect-Oriented Programming is the modular verification of aspects such that safe evolution and reuse is facilitated. However, one of the fundamental problems with verifying aspect-oriented programs is the inability to determine the effect of the weaving process on the control flow of the program, and thus on the state of the system and subsequently the properties that hold or are introduced. We propose a novel approach to modular verification of aspect-oriented systems using aspect tagging and Data Flow analysis of Control Flow Graphs.

1 Introduction

The increasing adoption of Aspect Oriented Programming (AOP) has considerably improved the evolvability of cross-cutting concerns (monitoring, security, replication, distribution) in complex software platforms.

The power of AOP essentially lies in its ability to impact a very large code base at run-time with only one aspect. Because of this power, however, it can be extremely difficult to predict the effect of an aspect on a base program, a particularly critical issue when AOP is to be used for software evolution. How can we be sure that an aspect achieves what it is meant to? How can we prove that it does not violate properties of the base program that must be preserved? How can we verify that it does not interfere with properties other aspects are trying to introduce?

The evolution of cross-cutting concerns would benefit enormously from well-developed formal techniques to answer these questions. Ideally such techniques should provide a framework with which to check AO programs at an early stage, in order to reuse and adapt aspects in a way which is formally verifiable.

Verification techniques are being developed for AO systems, but they still lag far behind what as been achieved for the static analysis of procedural and object-oriented programs. Our intuition is that, with the proper abstractions, existing aspect-free approaches (intra- and inter-procedural analysis, points-to analysis, abstract interpretation) can be specifically adapted to AO programs to meet their particular requirements. In this paper we discuss the properties such an "aspect aware" verification approach should have to be suitable for program evolution (Section 2). Current work is presented in Section 3. We then present

how the ideal could be realised in the particular case of data flow analysis using a technique we have termed "aspect tagging" (Section 4). Section 5 concludes the paper.

2 Problem Statement

Any AO program consisting of a base P and a woven aspect a can be represented by an equivalent standalone "aspect-free" program Q, on which traditional static analysis can be performed. This approach, however, suffers from a number of deficiencies that make it unattractive for aspect based software evolution. Firstly, it is very difficult to trace results obtained on Q back to the original aspect-oriented program $P + a$. Secondly, no general statement on the properties of a can be made, except in conjunction with a specific base program. This requires the whole analysis to be repeated for each base program on which a is applied. This limitation puts particular constraints on any evolution process based on program families. Thirdly, in decoupling the analysis from the AO structure of the original code, such an approach effectively bars any optimisation based on the AO nature of the program.

To circumvent those deficiencies we think that an *aspect-aware* verification approach should have the following desirable characteristics:

Modularity We feel that the naïve approach outlined above (performing analysis on the woven bytecode, thus determining whether $P + a = Q$ in terms of the properties that need to be maintained) neglects one of the key features of AOP - that is a *modular* framework, and thus requires *modular* analysis techniques. The criteria of modularity can be further broken into two sub-criteria:

- **Comprehensibility** The results we obtain using the analysis should be able to be *back-tracked* to the original program structure - that is, understandable using the terms of the encapsulation which the original program structure afforded. In real terms, this means that we will be able to see how the aspect itself has affected the properties of the system as a whole, not just how the system behaves.
- **Reuse** The results should be encapsulated in the same dimension as the aspect - that is, if the aspect is used with a different base program, the results should be able to be (at least partially) reused. This is a stronger property than comprehensibility.

Efficiency/Scalability The usefulness of formal methods for checking of safety properties is proportional to the efficiency with which they can be applied. Therefore, any analysis we can perform must have the ability to be applied within a reasonable time-frame, defined partially by the cost of failure of the system - that is, how safety-critical it is. The analysis must also be scalable - an analysis which is only applicable to trivial programs is fairly pointless. We would expect such an ideal analysis to scale to large industry-grade programs with multiple interacting aspects, as well as dynamic approaches.

Portability A desirable property of the analysis is the ability to be adapted to different languages, approaches and architectures, to maximise its usefulness.

One of the major challenges facing formal methods with AOP is determining a proper *abstraction* of the code such that program verification which fulfils the criteria above can be performed. This abstraction needs to be both *correct* - that is, encompass all the executions of the system that make sense or that we want to check - and *feasible* - that is, not containing so many possible states that state space explosion occurs and checking becomes unreasonable or useless.

This task becomes particularly hard in the presence of AOP's dynamic features, such that it can become infeasibly expensive to determine the execution of a AO system before run-time. For example, dynamic aspects could be woven at run-time; the behaviour of compile-time woven aspects could be affected by dynamic parameters; or dynamic joinpoints such as AspectJ's `cflow` could be used. An extreme total abstraction could then be that every potential aspect advice is applied at every potential joinpoint. This would clearly produce an absurd and useless abstraction, which would most likely be unable to positively determine any properties of the system.

Clearly, what is required is an abstraction of the system which is accurate enough to be *sound* - that is, proving something is true (or not) of the abstraction means that it is true (or not) of the actual system - yet useful enough to be as *complete* as we need - that is, able to give a definite answer to a proposition. If the abstraction is sound but not complete, we allow answers of "maybe" for every question we ask of the system, which is technically correct but not very helpful[1]. On the other hand, we would not want a system which gave us a definite answer for the abstraction which was not correct for the actual system. Thus, our abstraction must be correct, at least for the properties we want to check, and sound for the analysis we wish to perform.

We also require that our abstraction be modular - that is, that it retains the program structure of the actual system. We say this because we want the results of our analysis to be reused along with the aspect - recalling our example, we want our programmer to be able to get the aspect from the library and use verification techniques to determine that it will, indeed, work with his system. For this to work, it would be immensely helpful to have some result already present in the system in order to reduce computation time and effort.

Hence, we require a partial abstraction of the system, which provides us with an estimated set of potential executions which is as close as possible to the true set. Determining this abstraction is a matter of applying effective program analysis techniques - correct data-flow analysis combined with constraint-based analysis - to enable abstract interpretation [14, 7]. As we will show in Section 4, these techniques do not scale well to AO systems and require adaptation.

[1] Similarly, abstractions are often only sound for a particular class of answers - for example, if the abstraction answers "yes" for an analysis, we know the answer is "yes" for the original system; but if it answers "no" we cannot be sure. This strongly affects our choice of abstraction.

3 Current Work

There have been several notable efforts in the field of applying program analysis techniques to AOP. While all these efforts take slightly different approaches, the end goal is broadly similar - *modular* verification of aspects. The ideal goal is a complete proof that states that for every possible base system on which an aspect can be woven, and for every possible weaving within that system, the aspect will always:

1. Maintain desired properties of the base system such that the augmented (woven) system has the same properties as the original
2. Introduce its own properties to the augmented system correctly
3. Maintain desired properties that other aspects introduce

This goal is still a long way off for program analysis and, as such, most approaches seek to restrict the problem in some way.

We will divide discussion in this area into two sections - *static code analysis* techniques and *other approaches*.

3.1 Static Analysis

Recent static analysis techniques have related closely to the *categorisation* of aspects. An early attempt at this was suggested by Katz and Gil [11], in which three broad categories were proposed:

Spectative. These are simply monitoring aspects whose function is to record the actions of the base system without affecting them whatsoever.

Regulative. These aspects do not change the actions or basic functionality of the underlying system, but are often used to determine control flow in the system - an example being a contract enforcement aspect which decides whether a method is called based on pre-conditions.

Invasive. These aspects actively change the functionality or state of the underlying system in various ways. In powerful AOP systems, aspects can modify the values of both class and instance attributes or introduce their own, call methods before and after the advised joinpoint or even skip the joinpoint code completely (as is the case in an AspectJ advice with no `proceed()` statement).

Two other works also propose a classification system. Clifton and Leavens [5] suggest *observers/spectators* and *assistants* - similar to spectative and invasive aspects - and propose an extension to aspect languages by which the base object includes explicit references to the aspects which observe or assist it, enabling a more modular reasoning. Similarly, Rinard et al [16] propose a finer-grained categorisation coupled with a more powerful analysis to automatically classify interaction between advices and methods. Their work adapts an existing object oriented analysis to aspect oriented programs.

Work from Sereni and de Moor [17] proposes a reduced pointcut model based on regular expressions and use a meet-over-all-paths analysis which produces an optimised way of joinpoint matching. Although the primary goal of this work is optimisation, they acknowledge that the work could be used to determine aspect interaction - that is, when two or more different pieces of advice may be executed at the same joinpoint.

3.2 Other Approaches

Two other approaches [19, 21] encapsulate model checking assertions (using Bandera [6] and Jpf [15] respectively) within aspects, thus achieving some level of modularity. However, the actual checking then occurs on the augmented (woven) system, which prevents (partial) verification results to be attached to aspects for reuse on other base programs.

Krishnamurthi et al. [13] attempt a more modular model checking [10] technique whereby the final finite-state machine (FSM) of the woven system is constructed from the code before the aspects are woven - that is, an estimation of the final behaviour of the system is created. This uses a sophisticated backward flow analysis to determine the location of joinpoints and inserts calls to the FSM of the advice which would apply at the point. However, this approach only works for aspects which are guaranteed to return the system to the state in which the advice was called - in the terminology of [19], *spectative* aspects - which turn out to be a remarkably small subset of possible aspects. This technique can also only determine whether properties of the original base system are not violated, not whether the aspect introduces its own properties properly or affects the behaviour of other aspects.

Finally, there has been some work on formally identifying and resolving conflict or interaction between aspects. Sihman and Katz [18] develop a calculus for their superimposition system [20] which defines a general methodology for calculating how superimpositions are composed together before they are applied to the underlying system based on their specifications. However, this has yet to be implemented in a concrete AOP language.

Similarly, Douence et al [8, 9] develop an abstract formal semantics for aspects which includes rules for composition based on precedence. They demonstrate how defining composition with an order results in different behaviour which can be formally specified and hence provides a possible basis for analysis. They also propose rules for the detection of interaction. Again, this provides a very strong semantic base, but as yet is unimplemented.

The analysis system proposed by Rinard et al [16] has potential for detecting interference between aspects. In general, static analysis techniques such as program slicing [2, 3] have application in this field, although so far this has received little exploration.

In summary, the range of formal program analysis techniques currently under development for AOP systems is widening, reflecting the increasing confidence

in both AOSD and formal methods. However, in the early years of the AO paradigm, program analysis techniques are generally at an early level, tend to be application-dependent at least in their implementation, or reduce the problem somewhat by considering a subset of AOP features.

In particular, flow analysis techniques that are so far developed rely on representing the aspect-oriented program in such a way that existing (object-oriented) data and control flow analysis can be applied. This necessarily means that, at the flow analysis level, we end up treating the base program and aspects as a combined, woven, object-oriented system. Even when the program is represented in a graph with the distinction between base and aspect emphasised, as in [23, 24], the analysis then occurs on the complete program. This means that *modular* analysis of the aspect's behaviour independently of a base is restricted. It is this restriction that we aim to address in our work, in the development of a modular aspect-oriented data and control flow analysis.

4 Data Flow Analysis of Aspect-Oriented Programs

4.1 Summary of Proposed Approach

Our approach can be summarised as follows:

1. Obtain the bytecode of base and aspect;
2. Classify the aspect with respect to the base;
3. Create abstract control-flow graphs of both base and aspect;
4. Tag the CFG of the aspect;
5. Create a graph transformation of the base using the CFG of the aspect;
6. Use the resulting model to create an abstraction of the augmented system using data-flow analysis.

To implement this approach two main technical goals must be achieved:

Tagging The first goal is the ability to reason about an aspect and a base such that they remain distinct in our analysis. We achieve this by the process in which we construct the augmented CFG - that is, the CFG which represents possible executions of a woven base program and aspect - by tagging the nodes of the aspect advice and using these tags in the control flow analysis we perform.

Data Flow Analysis (DFA) The second goal is the data-flow analysis of the augmented CFG. The realisation of the first goal ensures that this analysis is modular, as the effects of the aspect can be clearly seen and backtracked to the original structure via the tags we have introduced. The transformation of the CFG enables us to map existing techniques to aspect-oriented programs.

An initial difficulty is finding the location of joinpoints at which the aspect advice might be applied in our system at pre-weave time. Different AOP models use a variety of *pointcut descriptors* (PCDs) at which advice can applied, some of

which are more difficult to statically determine than others. At this stage we use a simple PCD model based on pattern-matching of method signatures, with the aim of extending the model as the approach is developed, perhaps using abstract interpretation[7] for control-flow based PCDs.

From this, we extract control flow graphs from the bytecode of the base program and the aspect (extracted from the AspectJ compiler[1]). We then *tag* each node of the aspect's CFG to show us that it is part of the aspect and not the base. This is represented in Fig. 1 by means of a dashed box. When the CFGs are composed to form a model of the augmented system, the tags are maintained and give us the basis for a modular reasoning framework.

We then construct an augmented CFG by adding transitions from the joinpoints to the aspect's CFG, using an extension of the currently available Soot methods for doing so. This is comparable to existing techniques used for interprocedural analysis, and so we transform the CFG in such a way that these traditional approaches can be used. One difficulty in the CFG transformation is the problem of aspect pointcuts which have formal parameters that need to be bound. We envisage this being equivalent to inserting a decision node based on the predicates of the joinpoint with a "method call" to the advice node if the predicates evaluate to true.[2]

For example, for simple advices, we can simply add a transition from each node corresponding to a pointcut at which the aspect applies to the beginning of the CFG of the aspect's advice, and a similar return transition, depending on what kind of advice is being applied (see Fig. 1 for an example).

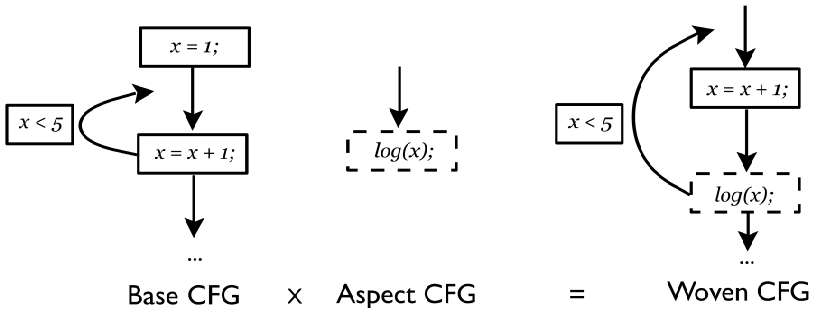


Fig. 1. The result of weaving a logging aspect on a base program consisting of an advised while loop

After this, we are left with an abstract augmented CFG on which we can perform data flow analysis.

[2] At this stage we only consider homogeneous aspects, i.e. aspects consisting of one advice relating to one concern. Heterogeneous aspects will be considered later in the development of our approach.

Here we use the classification of an aspect[16, 5, 20, 12] to determine what analysis to perform. For example, if the aspect is spectative[20] (that is, does not affect the state of the base system - e.g. a logging aspect), we do not need to check for violation of properties in the base system at all, reducing the intensiveness of the analysis. The ability to cut out stages of the analysis also enables us to reduce the level of abstraction we need to perform, meaning that we have a higher probability of obtaining meaningful results.

4.2 Adapting a Simple Analysis

To illustrate this, we show how we would attempt to adapt a simple data-flow analysis to a program in the presence of aspects. *Live variables analysis* is a classical data-flow analysis which aims to determine whether there exists, at a program point p, a path from the exit of p to a use of a variable such that there are no points on the path which redefines the variable [14]. That is to say, it aims to compute, for a given program point p, which of the variables currently defined at p can still have an impact on remaining execution of the program (i.e. are still "alive").

It is a backward flow analysis, and uses two flow sets gen_{LV}, the set of variables which appear in a block; and $kill_{LV}$, the set of variables which are killed (that is, redefined) in a block. The two flow functions $LV_{exit}(l)$ and $LV_{entry}(l)$ then calculate which variables are live at, respectively, the exit and entry of a program block labelled l. They are defined thus:

$$LV_{exit}(l) = \begin{cases} \emptyset & if\ l \in final(S_\star) \\ \bigcup\{LV_{entry}(l') \mid (l', l) \in flow^R(S_\star)\} & otherwise \end{cases}$$

$$LV_{entry}(l) = (LV_{exit}(l) \backslash kill_{LV}(B^l)) \cup gen_{LV}(B^l)$$
$$where B^l \in blocks(S_\star)$$

where $S_\star$ is the program we are analysing; $(l', l) \in flow^R(S_\star)$ means that the program flows forwards from l to l' and thus backwards from l' to l; and B^l is the program block in $S_\star$ which has the label l. Intuitively, then, the set of equations says that, at the exit to a block, the set of live variables is exactly the set of live variables at the entry of the block following it; and at the entry to a block, the set of live variables is the set of live variables at the exit, minus those variables that have been killed (i.e. redefined) in the block, plus those variables that have been used in the block.

This analysis works well for simple programming languages without functions or procedures, that is, *intraprocedural analysis* which only operates within a single control flow. *Interprocedural analysis* [14] - that is, analysis which takes into account control being passed to other procedures, functions and advices - introduces concepts such as call and return labelling and parameter passing for procedural languages, and there has been significant work on adapting this for object-oriented languages already, e.g. [4]. Further adaptation to more complex

languages requires significantly more sophisticated techniques for determining control flow, parameter passing and dynamic features of the language.

To adapt this analysis to be a) applicable to aspect-oriented languages and b) modular, we introduce the notion of *tagged flow sets*. The idea is that we encapsulate the data-flow information which is provided by the aspect in separate flow sets such that we can perform intraprocedural analysis on the aspect code, while retaining the ability to perform interprocedural analysis on the whole program. In other words, we can see how the whole program's properties are affected by the introduction of an aspect by considering the whole program. However, we can also see how an aspect would affect a certain base program given certain values for the binding of its abstract entities. We can then use this information to begin to extrapolate how the aspect would behave given certain *classes* of values - for example, whether a field is bound to a positive or a negative number - and thus create abstractions of how the aspect will affect a system, and thus create partial results which can be reused.

We introduce a set *advices*, which is the advices α which apply at a certain join point[3] ($JP_\star$) in the program $S_\star$. The advices applicable at a certain block (program point) are given by the function:

$$advices : Blocks_\star \rightarrow \mathcal{P}(Adv \times JP_\star)$$

$$where\ advices(b \in Blocks_\star) = \bigcap\{(\alpha, j) \in Adv \times JP_\star |\ b\ matches\ j\}$$

With this framework, we can reformulate the classical live variables analysis with tagged flow sets gen^A_{LV} and $kill^A_{LV}$ for an aspect A which introduces a `before()` advice α (this would be slightly different for different kinds of advice). LV_{exit} remains the same (as $flow^R$ will include the aspect statements as well as the base code), but LV_{entry} now has two forms:

$$LV_{entry}(l \in advices(S_\star)) = (LV_{exit}(l) \backslash kill^A_{LV}(B^l)) \cup gen^A_{LV}(B^l) \quad (1)$$

$$LV_{entry}(l \notin advices) = \begin{array}{l}((LV_{exit}(l) \backslash kill_{LV}(B^l)) \cup gen_{LV}(B^l)) \\ \cap(\bigcup\{LV_{entry}(l') \mid l' \in advices(l)\})\end{array} \quad (2)$$

So we now have two equations for computing live variables - one for when were dealing with a block of code thats in some aspect advice (equation (1)), and one when it isnt (equation (2)). When we are dealing with advice code, we have the same equation as previously, except using the tagged flow sets. When the code is in the base system, we compute the same as before, but we have to add in the information from the advice - so we also work out which variables have been killed from the advices which apply at that program point.

Intuitively, then, we formulate the live variables analysis based, not only on the proceeding statements in the base program, but also in the statements

[3] Here we assume static joinpoints. For dynamic joinpoints, we would have to consider the various joinpoint *shadows* - that is, the static code points at which dynamic aspects *could* apply.

contained within the code of the aspects which apply at the program point in question. Thus, we retain the encapsulation required, while still having the ability to evaluate the whole program as a single entity.

We plan to extend the Soot framework[22] to implement our approach. One of the benefits of this sophisticated optimisation framework is the ability to transform Java bytecode into an intermediate representation called Jimple, on which inspection and analysis can be performed.

4.3 Future Work

The modular verification, as described above, of a concrete aspect statically woven in a concrete base system is an appreciably difficult task which we hope our approach goes some way to resolve. However, the verification of *generic* aspects and bases is more difficult still - given an aspect with a abstract advice and an undefined joinpoint, can properties be verified? Conversely, can concrete aspect be subject to formal analysis even without a concrete base on which to weave?

We envisage that our approach can be used to facilitate more modular reasoning about the effect of generic aspects on an arbitrary base program, a future goal for our approach. Given the Soot framework's ability to generate classfiles from scratch, we may be able to produce a skeleton base program (or *dummy* program[19]) on which the weaving of a concrete aspect can be checked. Again, we hope to able to use the categorisation of the aspect to restrict the set of possible programs and/or program executions on which the weaving of the aspect makes sense, to reduce the resource intensiveness of this approach.

Especially, we envisage an application in the extremely difficult discipline of verifying dynamic AOP systems - that is, systems on which aspects can be woven, changed or removed while the program is running. Being able to produce partial results about the weaving of an aspect before it is due to be weaved would be a significant step forward in the goal of effective and verifiable reuse and evolution of dynamic Aspect-Oriented Programs.

5 Conclusion

We have presented a novel approach to the verification of aspects based on control flow analysis, using tagging to keep the base and the aspect distinct in our analysis such that the results can be backtracked to the original program structure. We envisage that bringing structural knowledge to the complex action of flow analysis will enable much more efficient static reasoning of aspect-oriented programs, and we hope to be able to map existing flow analysis techniques to analysis of such programs. We have shown possible extensions in the fields of verifying abstract aspects on abitrary base systems and verifying dynamic AOP systems.

References

1. AspectJ. Home page of the aspectj project. http://eclipse.org/aspectj.
2. Davide Balzarotti and Mattia Monga. Using program slicing to analyze aspect-oriented composition. In *In Proceedings of Foundations of Aspect-Oriented Languages Workshop 2004*, 2004.
3. Davide Balzarotti and Mattia Monga. Slicing aspectj woven code. In *In Proceedings on Foundations of Aspect-Oriented Languages Workshop 2004*, 2005.
4. Ramkrishna Chatterjee. *Modular Data-Flow Analysis of Statically Typed Object-Oriented Programming Languages*. PhD thesis, Graduate School, New Brunswick Rutgers, The State University of New Jersey, 2000.
5. Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. Technical Report 02-04a, Iowa State University, Department of Computer Science, April 2002.
6. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawm Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 2000 International Conference on Software Engineering*, 2000.
7. Patrick Cousot. Abstract interpretation. Technical report, LIENS, 1996.
8. Remi Douence, Pascal Fradet, and Mario Sudholt. Detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, 2002.
9. Remi Douence, Pascal Fradet, and Mario Sudholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, 2004.
10. Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.
11. Shmuel Katz and Joseph Gil. Aspects and superimpositions. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 308–309, London, UK, 1999. Springer-Verlag.
12. Jorg Kienzle, Yang Yu, and Jie Xiong. On composition and reuse of aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Engineering*, 2004.
13. Shiram Krishnamurthi, Kathi Fisher, and Michael Greenberg. Verifying aspect advice modularly. In *Proceedings of the ACM SIGSOFT*, 2004.
14. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2nd edition, 2005.
15. Java PathFinder. Home page of the jpf project. http://javapathfinder.sourceforge.net.
16. Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th International Symposium on Foundations of Software Engineering*, 2004.
17. Damien Sereni and Oege de Moor. Static analysis of aspects. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, 2003.
18. Marcelo Sihman and Shmuel Katz. A calculus of superimpositions for distributed systems. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Engineering*, 2002.
19. Marcelo Sihman and Shmuel Katz. Model checking applications of aspects and superimpositions. In *Proceedings of the 2003 Workshop on Foundations of Aspect-Oriented Languages*, 2003.

20. Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *The British Computer Society Computer Journal*, 46(5), 2003.
21. Naoyasu Ubayashi and Tetsuo Tamai. Aspect-oriented programming with model checking. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, 2002.
22. Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.
23. Jianjun Zhao. Slicing aspect-oriented software. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 251, Washington, DC, USA, 2002. IEEE Computer Society.
24. Jianjun Zhao and Martin Rinard. System dependence graph construction for aspect-oriented programs. Technical Report MIT-LCS-TR-891, Massachusetts Institute of Technology, 2003.

Towards Reusable Heterogeneous Data-Centric Disentangled Parts

Michael Reinsch and Takuo Watanabe

Department of Computer Science,
Graduate School of Information Science and Technology,
Tokyo Institute of Technology,
2-12-1 Oookayama, Meguro-ku, Tokyo 152-8552, Japan,
mr@uue.org and takuo@acm.org

Abstract. This paper presents our ongoing research towards a safe system evolution. Our approach is based on data-centric, object-oriented systems. It is our hope that such architectures will be helpful in integrating large, loosely coupled systems, as suggested by ubiquitous computing. Within those systems we utilise (i) multi-dimensional separation of concerns, (ii) explicit, language-independent type declarations in the form of an ontology and (iii) component technology. This results in *parts*, a construct which allows a developer to group methods according to their different concerns. Those methods are then "attached" to the ontology on which they operate. This combined approach makes it possible to cope with a growing code base, safely reuse structure and code supporting a safe system evolution.
Keywords: object-orientation, data-centric architecture, reuse, ontology, separation of concerns, software evolution, components

1 Introduction

System integration still provides a great challenge. When looking at today's applications and electronic devices, the amount of integration is still very low. Seen from the perspective of ubiquitous computing [1], system integration in heterogeneous environments is probably one of the key aspects.

In our ongoing effort to research ways to build software that easily integrates with other software, we identified that data-centric architectures could be helpful. However there are some problems regarding code reuse, safe combination and safe evolution which arise easily within those architectures. Safe combination and safe evolution of systems need even more attention when taking ubiquitous computing into account, as this implies a large, integrated, loosely coupled system, whose components are changed independently from each other.

1.1 Data-Centric Architecture

A data-centric architecture should help ease system integration by shifting the system architect's perspective: Data becomes the most important part of the

system, with the functionality grouped around the data, providing the user with different tools to manipulate this data. Thus it becomes important to share this data between all tools.

We see this in contrast to the traditional, application-centric architecture, in which full-blown applications come with a large, but more or less fixed set of functionality and lock the data in their own and sometimes proprietary file formats.

The tools in a data-centric architecture on the other hand can be small and can concentrate on one specific function. Through loosely coupled cooperation between those tools, a large, flexible and integrated software system should be achievable.

As an example for such a system consider a personal information manager. Following the above description, such a system would consist of several tools, e.g. an address book, a task manager, an email client, a calendar and so on. All those tools would share their data, e.g. the calendar could show all related tasks, emails, contact information. The whole system would be loosely coupled, tools could be added, removed and replaced by different versions or variants. The data could also be replicated on different devices which provide different tools. Projected further in an ubiquitous computing environment, one ends up with several loosely coupled devices, all sharing and contributing to the same distributed data base.

To implement such a data-centric software system, an object-orientated approach seems to be adequate. This allows the creation of a model which is closely related to the real world and it also allows to utilise abstraction, information hiding and reuse. In addition, component technology can provide the techniques to realise safe deployment and safe reuse of components through contracts, and it can help to achieve a more loosely coupled system [2].

However, a naive implementation will most likely lead into problems, caused by system evolution. The rest of this section is devoted to discuss such problems.

1.2 Object-Orientation, Information Hiding and Reusability

For this discussion it is first necessary to examine the ways an object-oriented system can be designed. Our main objective is to create a high quality, simple to understand and maintainable software system whose object-oriented code makes use of information hiding and can be reused.

In order to follow the principle of information hiding, it is necessary to not expose the internal structure of an object to external entities. Unfortunately this is often done, e.g. by utilising reputedly safe getter and setter methods [3], and is likely to cause problems when changes to the internal structure are required. For example, picture a change which introduces a new field to a class. This new field could influence the computation of other values throughout the whole system.

But on the other hand, without the possibility to access the internal structure of a class from the outside, this class likely ends up as an example of the anti-

pattern "The Blob" [4], because it has to provide all functionality which requires access to its internal structure.

Thus a solution which can allow for code growth without an impact on information hiding is desirable.

A viable compromise for this dilemma seems to be provided by structural design patterns. They make use of several objects and the internal structure of an object is opened only to a small number of objects. This however increases the complexity of the solution and most likely also the complexity of the whole system, since other design patterns such as abstract factories then probably need to be applied as well. Thus, in using structural design patterns we trade information hiding with an increase in complexity.

While this seems to solve the problem for one static software system (assuming the number of classes with access to the internal structure of an object is low and well documented), changes to the internal structure of an object can still cause problems when considering reuse in several independent applications. Those changes need to be tracked and each application needs to be explicitly checked by a developer because there is no formal description stating the dependency to the internal structure of an object. Within this context getter and setter methods might be regarded as even worse as they might obfuscate the fact that the internal structure of an object is accessed.

1.3 Dependencies, Information Hiding and Reusability

Regarding dependencies, a similar discussion to the one above is required. The main question is once again, where code should be placed which introduces a new dependency to another class. Again there are two choices: Either it is placed within the class itself or in a helper class by e.g. applying a matching design pattern like dependency injection. Moving this code into a helper class normally comes with advantages but also increases complexity.

Additionally, another conceptional problem cannot be solved that way: If two different versions of a class are required within the same application and the execution environment does not support versioning, the classes are likely to clash and consequently cause the application to malfunction.

1.4 Summary

From the discussion above it is obvious that object-orientation by itself does not provide a good solution for the described problems. It is necessary to explicitly add extra structure to cope with the growth of code. And this still leaves the questions concerning system integrity unanswered in the cases when code is reused or parts of a system are changed.

An approach which addresses the problems concerning complexity, reuse and information hiding as discussed above, and in addition allows to detect changes within the internal structure could thus simplify development and allow safer reuse, safer system composition and safer system evolution.

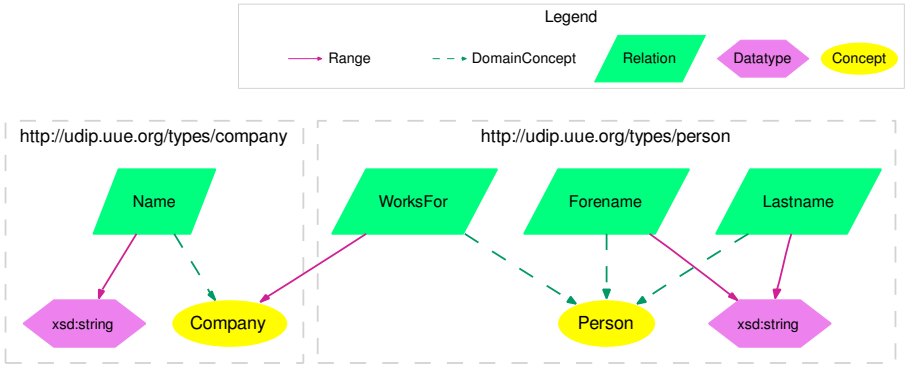


Fig. 1. Very simple ontology describing the concepts `Person` and `Company` and their relations

2 Concepts

In this section we present the concepts on which our approach is based. Generally speaking, we combine multi-dimensional separation of concerns with techniques known from component systems and add an explicit and language-independent type declaration in form of an ontology. In the following subsections we discuss this in more detail, and in section 3 we then present a possible implementation which also serves as an example for the points discussed here.

2.1 Ontology

In order to encourage a data-centric architecture, we explicitly model the data structure of a software system using a language-independent ontology. A very simple ontology describing the concepts `Person` and `Company` and their relations is provided in Figure 1 as an example. The system's data itself can then be stored within that ontology. This achieves language-independence for data as well, and allows reflection and reasoning over the stored data. From the meta-data contained in the ontology, code for different programming languages can be generated. All this then allows us to integrate code which is based on the same ontology, albeit possibly written in different programming languages.

In addition, it is also possible to store the declarations of the deployed parts (see next subsection for the term "parts") in the ontology. Therefore we added an additional complete meta-type system within the ontology which uses the terminology common in object-orientation and which fits our needs better than the one used by the ontology language itself.

The goal was to create a complete self-describing system. Hence it is possible to reflect, for example, about the current deployment status and change system

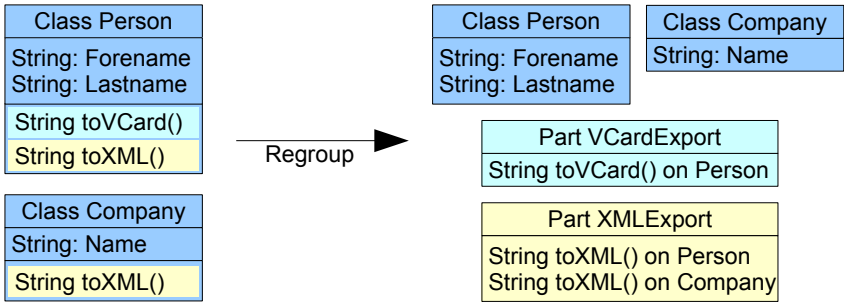


Fig. 2. Regrouping of methods in parts according to their concern

properties at run time. Using this mechanism, dynamic system adaptation could be implemented.

2.2 Parts

To cope with dependencies, information hiding, reuse and a growing code base, we introduce *parts*. Parts mainly serve to allow multi-dimensional separation of concerns [5], but in another way they also serve as small components as they explicitly define their dependencies, contractually specify their interface, and form the unit of composition and deployment [2].

To realise this, a part defines which other parts and concepts from the ontology it requires. It also provides a list of methods which are to be "attached" to the ontology. This list of methods also forms the exported interface of that part.

For example, a part which provides the functionality to export the content of a `Person`-object in the VCard-format might be defined like this:

```
Part: VCardExport
  Expects: The ontology in Figure 1
  Exports: String toVCard() on Person
  Required Parts: None
```

The implementation (see section 3 for an example) of this method, which is also provided by the part, has access to the internal structure of the object it is attached to. To be more precise: It has access to the internal structure of the object as it is defined in the `Expects`-statement. It cannot access any additional internal fields which might exist because a different part requires them. So the `Expects`-statement declares both, the requirements and the restrictions for a part. In addition, the implementation of a method can only execute the methods defined in its part or in one of the required parts, on those objects it can access.

Hence in comparison to typical object-orientated design, parts allow to regroup methods according to the concern they belong to, leading to a multi-dimensional separation of concerns. In the example illustrated in Figure 2, the methods of the classes `Person` and `Company` are regrouped in two parts, one for VCard-export and one for XML-export. This effectively separates the method

declarations and implementations from the class definition. Through this separation, only the field declarations remain in the class definition which can thus be replaced with the corresponding concept from the ontology. To ensure the safety of this separation, the requirements are explicitly specified by the part's `Expects`-statement and thus can be contractually assured.

2.3 Summary and Implications

With this approach all methods are now associated with a class and a part. The parts are used to group those methods together which belong to the same concern. Parts thus help to remove all unwanted methods from the view of other parts. They also define a namespace within the class to avoid clashes of method names. In addition, the implementation of a method is no longer done within the class but within the part.

This separation of class and method using parts has several advantages. First, it prevents a class from becoming large and hard to maintain. The code for a part also only needs to be deployed when it is required. Second, the implementation for parts can be written in different programming languages, as the authority defining the structure and types (i.e. the ontology) is language neutral. Third, the code for a part can also be executed on a remote host, allowing e.g. the small or GUI related parts to reside on a thin client whereas the parts doing heavy data processing could reside on a server. Forth, by explicitly defining the data structure a part expects, changes to the internal structure can be detected at deployment time and incompatible parts can be rejected.

When taking changes into consideration which might be required to a software system, we can divide them into the following categories and discuss the possible impact on the system for each category:

1. Changes to the code (i.e. within a part):
   We assume here that the same cases as for other object-oriented systems apply: A change can be either internal and thus not require any additional changes, or change the signature of the method and thus all code calling the method needs to be adapted. This can be detected at deployment time.
2. Changes to the data structure (i.e. to the ontology):
   This kind of change requires all parts which attach methods to the changed concept to be checked against the change. This is required to ensure system integrity and might trigger changes to the code. In other object-oriented systems this check is also required but not enforced. We are planning in researching ways to relax or automate this check.
3. Changes to the middleware:
   Changes within this category can trigger everything, from no further changes, to changes to every part, and changes to the ontology. So special care needs to be taken for those changes.
4. Changes to the programming language, operating system or hardware:
   In the worst case the middleware and all parts required to run in the changed environment need to be ported. The ontology can be reused.

3 Early Prototype

To further test this approach we implemented a prototype in Java 5. We chose Java to show that a simple translation to an object-oriented language is possible. The prototype also states that (i) each part should be compilable on its own without a special compiler, and (ii) the strong typing of Java should be kept and no casts should be required to access methods or attributes.

The ontology used by this prototype is stored in OWL [6]. This makes it easier to create and read the ontology because tools and libraries to process OWL are available.

The prototype mainly consists of a code generator. First, for every part interfaces like the following one are generated:

```
public interface VCardExport {
    String toVCard();
}
```

Then for each concept in the ontology a class is generated:

```
public abstract class Person extends Subject {
    public final VCardExport cVCardExport
        = new VCardExport() {...}
    protected final String getForename() {...}
...
```

The ontology's relations are translated into attributes (e.g. `getForename`) of those classes. Every class to which methods are attached, gets additional final attributes whose types are one of the parts' interfaces (e.g. `cVCardExport`). The generated code within the class dispatches all calls to a core object which holds all attributes as well as all registered part implementations.

An implementation for the part `VCardExport` could then look like this:

```
public class PersonVCardImpl
    extends Person implements VCardExport {

    public String toVCard() {
        StringBuffer res = new StringBuffer();
        res.append("BEGIN:VCARD\nVERSION:3.0\n");
        res.append("N:").append(getForename());
...
```

To be able to call this method, some part must define that it uses the concept `Person` and the part `VCardExport`. Then an object within the part can obtain a reference to a `Person`-object and call the method like this:

```
String vcard = myPerson.cVCardExport.toVCard();
```

With this we have shown that an easy translation to an object-oriented language is possible.

4 Related Work

In this section a discussion about related work is presented. We first introduce the related work briefly and then discuss the similarities and differences.

4.1 Subject-Oriented and Hyperspace Programming

The main concept behind hyperspace programming (being by itself a more generalised form of subject-oriented programming [7]) is the multi-dimensional separation of concerns [8]. Therefore it is quite similar to our approach, and in fact we borrowed several ideas from their approach. But there are some differences:

We bring in techniques from component systems which allow us to define clear dependencies. Through the dependencies between our parts, it is possible to build a layered system. Thus we do not require the parts to be complete applications on their own. We instead see our parts as being quite small, capturing what perhaps could also be called mini-concerns, and to be reused by other parts.

Composition rules are considered to be an important part of hyperspace programming. In our approach on the other hand, the types used by the parts are standardised through the ontology, eliminating the need to create mappings between classes and attributes. Though in a later stage, we might reintroduce some kind of composition rules in the form of mappings between different ontologies.

Through the meta-data present in our approach, it is also safe to compose the system at run-time, eliminating the need of a special compiler. Additionally, this meta-data can also be exploited in other ways as discussed earlier.

4.2 Aspect-Oriented Programming

In aspect-oriented programming (AOP) the main focus lies on cross-cutting aspects which it tries to consolidate [9]. Thus there are similarities to our approach which also allows us to group cross-cutting concerns together. But both approaches are different in the same way that hyperspace programming is also different from AOP: In AOP the aspects are weaved into a core implementation. With our approach, a core implementation does not exist, but only different concerns. Additionally, the granularity is different: AOP allows us to weave code into methods, whereas our approach allows adding methods to classes.

4.3 Code Generation from Ontologies

Several code generators exist which generate code based on an ontology, where the most similar approaches try to translate as much of the semantics of an ontology language to a programming language, e.g. [10,11,12,13]. All approaches so far expose the internal structure of the ontology, e.g. by utilising getter and setter methods in object oriented languages. Adding new code to the generated code is possible, e.g. by extending the generated class if the code generator itself does not respect developer supplied code. But to our knowledge, no other approach similar to the one presented in this paper exists yet.

5 Concluding Remarks

We presented a combination of multi-dimensional separation of concerns with techniques known from component systems and an explicit and language-independent type declaration in form of an ontology. Through the separation of methods from their classes (which are defined by the ontology's concepts) and by regrouping those methods in parts we gain extra flexibility, can cope with the growth of the code base and reuse those parts. In addition, existing parts can be easily extended by adding new parts, or replaced by newer versions. Through explicitly stated dependencies between parts and the definition of the required type structure for a part, we achieve a safe method to combine several parts to a larger system: Incompatibilities can be detected and handled at the time of deployment. Finally, the usage of a language-independent ontology to define the types of the system helps in reuse of not only code but also the structure of a system.

6 Future Work

In this paper we presented a snapshot of our ongoing research in the area of software evolution. Hence this work is not complete, much remains to be done:

First, our existing prototype needs to be further improved and extended, and automatic object persistency as well as object versioning has to be added. Then we plan on implementing a larger set of integrated, yet loosely coupled tools, demonstrating software evolution. With this code base we then hope to be able to further examine safe composition and re-composition at run-time. Based on those results, automatic system adaptation could be studied. We also plan on exploring new ways on how to further profit from the rich meta-data present in our approach, for instance to implement an automated deployment and undeployment (garbage collection) mechanism for our parts. Finally, the step into an ubiquitous environment could be done, by utilising automated object replication and distributed parts, while taking care of data and system security.

References

1. Weiser, M.: The Computer for the 21st Century. Scientific American (1991) Online available at `http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html`.
2. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison Wesley (1998)
3. Holub, A.: Why getter and setter methods are evil. JavaWorld (2003)
4. Brown, W.J., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley (1998)
5. Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proceedings of the 1999 International Conference on Software Engineering. (1999) 107–119

6. Web-Ontology Working Group: OWL Web Ontology Language, Overview. Online available at http://www.w3.org/TR/owl-features/ (2004)
7. Harrison, W., Ossher, H.: Subject-Oriented Programming: A Critique of Pure Objects. In: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA '93). (1993) 411–428
8. Ossher, H., Tarr, P.: 10. In: Software Architectures and Component Technology: Multi-dimensional Separation of Concerns and the Hyperspace Approach. Kluwer Academic Publishers (2002)
9. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). (1997)
10. Eberhart, A.: Automatic Generation of Java/SQL Based Inference Engines from RDF Schema and RuleML. In: Proceedings of the First International Semantic Web Conference on the Semantic Web (ISWC), London, UK, Springer-Verlag (2002) 102–116
11. JSave Extension for Protégé. Online available at http://protege.stanford.edu/plugins/jsave/ (2005)
12. Beangenerator for Protégé. Online available at http://acklin.nl/page.php?id=34 (2005)
13. Kalyanpur, A., Pastor, D., Battle, S., Padget, J.: Automatic Mapping of OWL Ontologies into Java. In: Proceedings of Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE). (2004)

Technological Limits for Software Evolution

Chairman: Yvonne Coady, University of Victoria, Canada

Pitfalls in unanticipated dynamic software evolution

Peter Ebraert[1⋆], Yves Vandewoude[2⋆], Theo D'Hondt[1] and Yolande Berbers[2]

[1] Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium
[2] KULeuven Department of Computer Science, Celestijnenlaan 200A, B-3001 Leuven, Belgium

Abstract. The authors of this paper have all developed a framework that allows runtime adaptation of software systems. Based on our experiences, we wish to summarize common pitfalls concerning dynamic software evolution. Systems for dynamic adaptation typically follow a certain process which is used as a starting point in this paper. The problems that occur in the different steps of this evolution process are given and a suggestion is made on how these problems can be tackled. The reader will notice that the solution to most of the pitfalls lies in the use of reflection, meta-data and meta-object protocols. We conclude that reflection or meta-object protocol manipulations are indispensable in the process of dynamic software evolution and that better language support is needed.

1 Problem Statement

Lehman [1] defines software evolution as the collection of all programming activities intended to generate a new version from an older and operational version. The problem of software evolution occurs after the initial delivery of the software and typically deals with bugfixes and the addition, change or removal of functionality to the system. Different sources estimate that evolution is responsible for 50% [2] to 90% [3] of the total cost of software. The following quote by Keith Bennet [4] perfectly describes the real difficulty of software evolution: its unanticipated nature. *"The fundamental problem, supported by 40 years of hard experience, is that many changes actually required are those that the original designers cannot even conceive of."*

In most of the cases, software evolution is performed on systems that are shut down. However, there are some systems that can not be shut down because of some specific reasons (such as safety or financial aspects). Well known examples are web services, telecommunication switches, banking systems, airport traffic control systems or military systems. Adapting such systems without halting them is a challenging operation that encompasses many different problems. Those problems are tackled in the field of dynamic software evolution.

⋆ Authors funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen)"

The following section starts with an overview of all difficulties that come with the dynamicity of dynamic software evolution. Afterwards, it presents a commonly accepted process that can be followed in order to cope with those difficulties. Section 3 lists all the pitfalls that rise if one wants to follow the process stated in section 2. In sections 4 and 5 we conclude that, in order to resolve the pitfalls, both reflection and meta-data are indispensable and that they can only be offered by languages with a fully reflective meta-object protocol.

2 Unanticipated dynamic software evolution

In his PhD report, Oriol states three main reasons that make unanticipated dynamic software evolution such a hard undertaking: coping with *active threads*, *state transfer* and *uncertainty*. In this section we first explain those three issues and then then see how the most commonly accepted process for dynamic software evolution handles those issues. Step by step, we discuss all the phases of this process.

In an object oriented program, the execution state is typically represented by the state of all the living objects (housed by instance variables), and the state of the runtime stack. Changing an object oriented program while it is active, means that parts of the objects may be modified during their execution (a typical dynamic evolution mechanism). This could lead to inconsistent states (states from which the program cannot finnish correctly). This is why *active threads* must be taken care of, before a change is actually performed on a running system.

Inconsistencies might also occur when a certain change requires us to replace existing entities by new entities. When an old entity is replaced by a new one, we must incorporate the state of the old entity into the new entity, for not ending in an inconsistent state. The process of porting the state from an old entity to a new one, is called *state transfer*.

The last reason in the *uncertainty* of applying a change to a running system. As we are applying a change to a running system, we do not have a test-phase, in which we verify that changing the program results in unwanted behavior. Below, we enumerate 6 steps that overcome the three above mentioned issues. This is why those steps form the typical process of dynamic software evolution.

1. **Offline activities.** Before a dynamic update is deployed, it must first be implemented. Offline activities start by locating all structures or entities that are affected by the changes. This problem is often referred to as *dependency finding*. The new code is then implemented according to the renewed specifications of the system. In most cases, the design and source code of the old version are present, since it is likely that new versions are implemented by the owner of the old version. If this is not the case, an attempt must be made to recover these from the software artefacts that are available (such as the running system) using *reengineering techniques*. Finally, the correct behaviour of the new version must be verified using either formal proofs or

extensive testing. In some cases, the deployment of the new version itself is also tested by deploying it on a duplicate copy of the running system.

2. **Addition of new code to the running system.** The complexity of *introducing new code* into the running system strongly depends on the programming language and environment used. It is easy for languages as SmallTalk or CLOS, harder for statically typed languages as Java or C# (since code can not easily be reloaded), and very hard for languages that are compiled to native code such as C or Assembly.
3. **Deactivation of affected entities.** Dynamic adaptation of an active system entity can result in inconsistencies that may lead the application to an erroneous state. *Program consistency* can be preserved by deactivating all entities or structures within the application that are affected by the change.
4. **Transformation of affected entities.** This phase consists of the actual transformation from the old version to the new version and is composed of transforming behavior and porting state. In class based languages, behavior is captured in method definitions and in the inheritance hierarchy. As such, behavior transformations boil down to class based modifications. The most difficult part of this phase however, is the transformation of runtime state that is contained in variables throughout the system to preserve *state consistency*.
5. **Online verification of new code.** Once the transformation has completed, we wish to *verify* a number of conditions to guarantuee its correctness. This is achieved by evaluating a number of conditions and invariants on the new code version. If these checks fail, a *rollback* mechanism must make sure that the previous state is restored.
6. **(Re)activation of the halted entities.** The last step consists of reactivating all the entities that were deactivated earlier in the process.

Relating to the issues identified by oriol, steps 3 and 6 are present to cope with active threads, step 4 deals with state transfer and the tests in steps 1 and 5 are present to lower incertainty. In the next section, we discuss common pitfalls that occur in this process and suggest possible solutions.

3 Pitfalls

3.1 Dependency finding and reengineering

In order to implement a new version of a software system, it is crucial to obtain its architecture and design if not already present. This information is required for the identification of all the entities that have some relation with the evolving part of the software. In addition, we need it for providing new source code that will fit in the existing system.

Recovering the design of an application has to be done by using both static and dynamic information. Static information describes the structure of the software as it is written in the source code, while dynamic information describes what is really happening at runtime. It can be perfectly possible that structural

information shows that two classes are just a bit related, as there is only one method call from one class to the other. However, it is possible that dynamic information shows that the same call occurs continuously when running the application, making the two classes very related. This explains why both dynamic and static analysis result in more realistic design recovery.

Static information can be obtained by looking at the application's implementation. Practically this can be done in two different ways; by looking at the source code or by using introspection (= reflective capabilities of observing the application).

Dynamic information can be gathered by monitoring the behavior of the running application. This is typically done by using a layered approach. Implementations of this approach include the adaptation of the metaobject protocol in such a way that all baselevel executions are intercepted and monitored. Another implementation consists of the instrumentation of base-level entities with calls to a monitor by means of intercession (= reflective capabilities of modifying the application).

3.2 Introduction of new code.

Whereas safely introducing new code to a running system is a non-trivial accomplishment by itself for languages such as C or assembly [5, 6], modern languages such as Java, Smalltalk or C# allow a programmer to add new code in a safe and extremely convenient manner.

However, the ability to add new code to a running system is by itself not sufficient for dynamic adaptation. Unanticipated software evolution almost always includes modifications to existing code. Although this is not a problem for purely dynamically typed languages such as Smalltalk, it is a much harder problem for statically typed languages such as Java or C# (for a extensive discussion on the influence of a programming language and its type system on runtime evolution we refer to [7]). For statically typed languages, unloading or modifying code which is already loaded is prohibited due to safety restrictions that are enforced by the language model. For instance, Java ensures that all methods or fields that are used also exist. Such a guarantee can not be given if class definitions can change at runtime unless extensive and frequent runtime type checks are added which would degrade runtime performance significantly. This is a sacrifice designers of statically typed languages are not prepared to make.

In Java, the problem is partially circumvented using the classloader mechanism. Since classes loaded by different classloaders are considered to be distinct types, the classloader architecture also allows different versions of a class to be used simultaneously. Such techniques are used by component runtime environments to (un)load different components independently. While this approach is sufficient for loading modified code into the system, it causes some important problems related to dynamic adaptation: an object of version $n + 1$ can not be assigned to a variable of version n. This problem is called the version barrier [8], and is especially relevant for state transfer between different versions of an

application. As we will see in the section 3.4, reflection and meta-data will play a vital role in solving these problems.

3.3 Program consistency and deactivation.

It is clear that for a dynamic update to succeed, arbitrary changes to the software can not be allowed. For example, online software replacement may not be feasible if the new version of the program is an entirely different program. It is vital that a runtime change preserves program consistency. An informal definition of a consistent application state is a state from which the program continues execution in a correct manner rather than progressing towards an error state. An application can be seen as moving from one consistent state to the next. Since state is distributed throughout the system, different state structures can be temporarily inconsistent with oneanother. It is vital that the application is in a consistent state before runtime modifications are performed. Kramer and Magee introduced the concept of quiescence in [9]. While their work was originally in the field of distributed systems consisting of a set of distinct nodes, it can be applied to dynamic adaptations of object-oriented or component-oriented applications as well.

In order to ensure that no communication or method calls are active during the modification of a certain entity, the entity must be deactivated. Any deactivated entity will queue all incoming transaction request and postpone their execution until the entity is reactivated. Different implementations are possible to achieve this goal. In [10], a wrapper based approach is used. A wrapper is added to each system entity that adds additional functionality for activation or deactivation. *Futures* are returned to the caller as a return value. These futures will be resolved when the entity is reactivated. Messages to futures result in futures themselves. This chain of futures continues until a side-effect occurs, after which the application is halted until the entities are reactivated. In [11], communication between components is asynchronous and realised by sending messages through *connectors* that are capable of queuing messages until the component is reactivated. Since there are no return values, the concept of futures is not required.

In the end, the implementation of a deactivation scheme strongly depends on reflection: communication is reified into messages that can be queued until further notice. The advantage of reflection is that it allows the addition deactivation logic without modifying the components themselves.

3.4 State transfer and consistency.

Although deactivation is essential for dynamic software adaptation, it is not sufficient by itself. True dynamic evolution requires that the state from the previous version is imported in the new version of the software. The assumption of quiescence ensures that all state is contained in the instance variables of the

different objects that make up the application or component (assuming an object oriented paradigm). Two possible approaches exist to achieve state transfer between different versions:

Indirect: The old version exports its state in some *abstract form* which is later interpreted by the new version. In some cases, the exported state can be written to disk.

Direct: The new version *directly* interprets the state from the old version.

Although the first version seems more convenient at first, it has some major disadvantages. First of all, in order for the exported state to be in an *abstract* form, a generally accepted ontology must exist so that all parties can agree on the semantics of this abstract state. Such an ontology only exists for certain domains, severely limiting the practical approach of this technique. In addition, this requires that each entity implements a state export function, even if it may never be used. This lays a huge additional burdon on the programmer. The second technique does not suffer from these restrictions. State is extracted using either getters/setters, or, more likely, using reflection.

For statically typed languages, the presence of different application domains or classloaders (see section 3.2) further complicates the adaptation, since communication between different versions is strictly limited to known common types (eliminating the ability to extract state using getter-methods and increasing the dependency on reflection). The presence of such an architecture also results in a *cascading effect of changes*, which eliminates the possibility of preserving large (unchanged) portions of the application or component. Indeed, a type A which has not changed by itself, but that contains a reference to a changed type B will not be able to use the new version of B due to the version barrier. Transforming objects of A to refer to the new version of B causes a cascading effect, since all types referring to A would require changes as well. A solution to this problem was proposed in [8], in which Sato and Chiba introduce Negligent Classloaders, which relax the version barrier under certain circomstances. An alternative technique would be to change the classloader of unmodified types from the old version to the classloader of the new version, allowing them to be integrated in the object tree of the new version. Both solutions require virtual machine adaptations.

It is unlikely that a generic solution can be developped without strongly depending on both reflection and meta-data. Regardless of how the actual state transition logic is generated, transferring state between different versions requires information which is not always present in the sources of the different versions, and therefore would need to be added using meta-data. Both the extraction of the state from the old version, and its insertion in the new version require reflective operations (the adaptation is unanticipated and its likely that the running version does not have the required extraction functionality).

3.5 Verification and rollback.

Deepak Gupta has proven [12] that full automatic verification of the correctness of an update is computationally undecideable. Therefore, the designer of the update must include a number of checks with the new version. These checks can either be executed before the transition, verifying that certain unwanted states are not present, or after, to insure that the update was indeed succesfull. An example of the a pre-condition that is commonly used for dynamic adaptations is ensuring that a component is not involved in a transaction [13, 9]. As long as the precondition is not satisfied, the update is delayed. After the update, additional sanity checks can be executed on new version before it is reactivated. Postconditions are also commonly used to verify the result of dynamic aspect weaving (for example, in [14] the authors verify their aspect compositions using postconditions). Reflective mechanisms are necessary, not only to extract these conditions from the new version, but also to evaluate them. Indeed, it is likely that these conditions will use state from the new version that can not be accessed without reflection.

If one of the tests fails, a rollback is required to restore the original system. Following the process that was described in section 2, the rollback will only have to be applied on deactivated entities. This is achieved by maintaining of a copy of the original entities that are to be restored during rollback. This copy can be retrieved using introspection and restored using intercession.

4 Need for language support

As we have seen above, reflection and meta-object protocol manipulations are indispensable for allowing dynamic software evolution. Our findings and other peoples findings (at RAM-SE 2005) clearly indicate that more language support is needed for easing dynamic software evolution. Current mainstream languages (like C, C++, Java and C#) are not sufficient as their abilities towards reflection and meta-protocol manipulations are too limited: mainly because of two reasons.

The first problem is the class-loading principle which loads classes in the memory of the virtual machine at the time of their first useage. The problem with classloading is that after a class is loaded into memory, no more changes are allowed to these classes since object instances may already exist [3]. Therefore, changes made to the class after it has been loaded are not propagated at runtime. This can be overcome by modifying the virtual machine [15–18] or changing the classloader mechanism [8]. Although some of these techniques are indeed capable of alleviating current limitations of languages such as Java and C#, these modifications also changes the semantics of the original language and therefore break compatibility with the mainstream version of the language.

The second problem lies in the expressiveness of the languages. In order to apply meta-object manipulations of certain concepts of a program, we need those

[3] Java Hotswap does allow some changes, but these changes are extremely limited in scope.

concepts to be fully reified. Languages such as C, C++, C# and Java have a lot of concepts which are not yet reified (for instance the method lookup). Because of that, those concepts cannot be inspected or changed. As an alternative, we propose languages with a full reflective meta-object protocol (with all concepts fully reified). While full reflection might impose a runtime overhead, those languages offer more capabilities for meta-object manipulations and reflection, and thus for dynamic software evolution. Examples of such languages are Smalltalk and CLOS.

5 Conclusion

We start this paper by giving an overview of common difficulties that come with dynamic software evolution. In particular these are related to state transfer, active threads and uncertainty. A typical process of dynamic software evolution is then presented that copes with these difficulties. However, in this process a number of pitfalls show up. An explanation is given for each of these pitfalls and a conceptual solution is suggested. Our conclusions confirm that both reflection and meta-object protocol manipulations are indispensable in the field of dynamic software maintenance. Taking into account the mainstream languages (Java, C, C++ and C#), we claim that those languages are not well suited for dynamic evolution as they have intrinsic problems that hinder it. We therefor suggest to use languages like CLOS and Smalltalk.

References

1. Lehman, M., Ramil, J.: Towards a theory of software evolution - and its practical impact. Invited Lecture, Proc. Intl. Symp. on Principles of Software Evolution (2000) 2–11
2. Lientz, B., Swanson, E.: Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations. Addison-Wesley (1980)
3. Erlikh, L.: Leveraging legacy system dollars for e-business. IEEE IT Pro (2000) 17–23
4. Bennet, K.H., Rajlich, V.: Software maintenance and evolution: A roadmap. Future of Software Engineering. (2000)
5. Segal, M.E., Frieder, O.: On-the-fly program modification: Systems for a dynamic updating. IEEE Software **10** (1993) 53–65
6. Hicks, M.: Dynamic Software Updating. PhD thesis, Department of Computer and Information Science, University of Pennsylvania (2001)
7. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Influence of type systems on dynamic software evolution. Technical Report CW410, KULeuven, Belgium (2005)
8. Sato, Y., Chiba, S.: Negligent class loaders for software evolution. In Cazzola, W., Chiba, S., Saake, G., eds.: ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04), Oslo, Norway, Fakultät für Informatik, Universität Magdeburg (2004)

9. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering **16** (1990) 1293–1306
10. Ebraert, P., Mens, T., D'Hondt, T.: Enabling dynamic software evolution through automatic refactorings. In: Proceedings of the Workshop on Software Evolution Transformations (SET2004), Delft, Netherlands (2004)
11. Vandewoude, Y., Rigole, P., Urting, D., Berbers, Y.: Draco : An adaptive runtime environment for components. Technical Report CW372, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (2003)
12. Gupta, D.: On-line Software Version Change. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur (1994)
13. Janssens, N., Michiels, S., Mahieu, T., Verbaeten, P.: Towards Hot-Swappable System Software: The DIPS/CuPS Component Framework. In: Proceedings of the Seventh International Workshop on Component-Oriented Programming, Malaga, Spain (2002)
14. Klaeren, H., Pulvermüller, E., Rashid, A., Speck, A.: Aspect composition applying the design by contract principle. In: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering, Erfurt, Germany (2000) 57–69
15. Malabarba, S., Pandey, R., Gragg, J., Barr, E., Barnes, J.F.: Runtime support for type-safe dynamic java classes. In: Proceedings of the 14th European Conference on Object-Oriented Programming. (2000)
16. Andersson, J., Ritzau, T.: Dynamic code update in JDRUM. In: Workshop on Software Engineering for Wearable and Pervasive Computing, Limerick, Ireland (2000)
17. Andersson, J., Comstedt, M., Ritzau, T.: Run-Time support for dynamic Java Architectures. In: ECOOP'98 Workshop on Object-Oriented Software Architectures, Brussels, Belgium (1998)
18. Ritzau, T., Andersson, J.: Dynamic deployment of java applications. In: Java for Embedded Systems, London, United Kingdom (2000)

Architectural Reflection for Software Evolution

Stephen Rank

Department of Computing and Informatics,
University of Lincoln.
srank@lincoln.ac.uk

Abstract. Software evolution is expensive. Lehman identifies several problems associated with it: Continuous adaptation, increasing complexity, continuing growth, and declining quality. This paper proposes that a reflective software engineering environment will address these problems by employing languages and techniques from the software architecture community.

Creating a software system will involve manipulating a collection of views, including low-level code views and high-level architectural views which will be tied together using reflection. This coupling will allow the development environment to automatically identify inconsistencies between the views, and support software engineers in managing architectures during evolution.

This paper proposes a research programme which will result in a software engineering environment which addresses problems of software evolution and the maintenance of consistency between architectural views of a software system.

1 Introduction

Software evolution is expensive, with costs variously estimated as constituting 50–70% of the total lifecycle costs of software [1]. This paper proposes a software engineering environment which will enable software engineers to produce software that is easier and cheaper to evolve as its requirements change. The environment will use reflection to maintain a consistent set of views of a software system at several levels of abstraction, up to and including the architectural level.

The proposed environment will ensure that software engineers always have up-to-date knowledge of the architecture and design of a software system, enabling them to make informed decisions during its evolution, and to avoid some of the problems associated with degradation of structure during evolution. Architectural constraints, both within and across views, will be automatically monitored, using design critics [15] to inform software engineers of inconsistencies and potential problems with a software system.

2 Problems

There are several problems associated with the evolution of software. This paper proposes a software engineering environment which is designed to address five of these problems, selected from Lehman's laws of software evolution [2]:

Continuous Adaptation "E-type systems[1] must be continually adapted else they become progressively less satisfactory"

Increasing Complexity "As an E-type system evolves its complexity increases unless work is done to maintain or reduce it"

Continuing Growth "The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime"

Declining Quality "The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes."

This paper takes the position that loss of knowledge about architectural structure is a cause of several of these problems: knowledge is dispersed throughout the documentation, code, configuration management tools, build tools, and often only maintained tacitly by developers. Section 3 identifies the architectural principles used in this paper, and section 4 introduces reflection as used in this work.

Because knowledge of and constraints upon software architecture are not always made explicit by developers, these structures tend to degrade as software evolves. As the structure is not known, there are extra comprehension costs in maintenance. Worse, if the documentation is wrong/out of date, work can proceed on incorrect assumptions about the current and desired structure and content of the software system.

This paper proposes that a software engineering environment which makes use of reflection can be used to:

- Automatically maintain the consistency of our documentation with respect to the various different "views" of a system;
- Make the architecture of a system visible and manipulable at run-time;
- Allow intervention at a higher level than the source;
- Manage and maintain safety properties and other desirable features;
- Allow automatic analysis of architectural properties of software systems.

A programme of research, leading to a software system which can support the above activities is proposed in section 5, and further work is identified in section 6.

[1] Software systems that are *E*mbedded in a real-world environment, as contrasted to P-type software which solves approximations of real-world problems with well-defined input, such as weather forecasting, and S-type software, which is formally and completely specified as a function from its input to its output.

3 Software Architecture

Software architecture is the study of the structure of software systems, including inter-component relationships [3, 4]. One of the first definitions of architecture, still widely-used, is "Software Architecture = {Elements, Form, Rationale}" [5]. Interactions are considered first-class entities [6]. Architectural description languages model both components and connectors.

There are many important structures in a software system. Kruchten suggested a "4+1" view model, in which four structural views are bound together with a fifth "scenarios" view [7]:

Logical The object-oriented decomposition, commonly modelled using class diagrams.

Process Models "non-functional" requirements (*eg*, performance, availability). Often modelled using collaboration or interaction diagrams.

Development Modular decomposition, modelled with component diagrams.

Physical The mapping of the software to the hardware; usually modelled with deployment diagrams.

Scenarios The "+1" view; a set of scenarios used to motivate the development and assist in the verification of the system.

Documentation of architecture is a key issue: "The essence of the activity is writing down—and keeping current—the results of architectural decisions" [8]. Maintaining a correct and up-to-date architectural model assists with the problems identified in Lehman's laws of Continuing Growth and Declining Quality (described in section 2). Additionally, knowledge and control of the architecture of a software system is required to support the multi-level feedback nature of evolutionary processes.

4 Reflection, Architecture, and Evolution

Software reflection has been used in many ways to support software evolution [9–12]. This work has usually focused on enabling evolution to take place on a particular system at a design level, rather than on using reflection to ensure consistency between multiple levels of view. It is possible to use reflective operations to map simple architectural changes down to the implementation [9, 13], allowing modifications to the system to be carried out a relatively high level.

Information obtained by reflection tends to be limited to structural and low-level behavioural information. There is a lot of architectural information generated by software engineers during the production and modification of a system. Kruchten's views [7] can be considered to be bound together by reflection:

Logical Available as a result of reflection or introspection on source code (or intermediate representations such as Java bytecode), as it most closely corresponds to the source of the system.

Process Partially available from reflection and static analysis of code, but often only made explicit in requirements or specification documents.

Development Partially available from reflection or even SCM data.
Physical Partially related to the process view, and available in some cases from deployment descriptors.
Scenarios Not available from the code (except in very specialised cases).

Reflection can be used to bind these views together, ensuring the correspondences between them are maintained, and allowing changes made in one view to be automatically reflected in the others.

Reflective modelling of the *architecture* of software systems can support run-time software evolution. We can support multiple views in ways that make them consistent.

In order to accomplish the goal of knowing the architecture of a system at run-time, the following are required [14]:

Monitoring The ability to to see the architecture at run-time
Interpretation Making sense of the data from monitoring, looking for 'problems' (in any sense of the word)
Resolution Fixing problems: manually, semi-automatically, or automatically. Determining where problems are and what to do about them.
Adaptation The ability to make changes to the architecture of the system.

5 Proposal

This section proposes a software development environment which, using reflection, automatically maintains consistency between a collection of views of a software system. A programme of research and development leading to such an environment is proposed, and potential benefits are discussed.

5.1 A New Kind of Development Environment

It is proposed that a software development and analysis environment will be created. This environment will use reflection to support architectural (and other) approaches to software evolution. Reflection will be used to ensure that different views of the system are synchronised (and to enable highlighting of incompatibilities between these views).

Figure 1 shows the "Lingua Franca" approach: one single representation, with several views (a code view, a use-case view, an architectural view, *etc*) available by applying different 'lenses' to the central representation of the system. By creating appropriate 'lenses', other kinds of views can be created, such as a configuration management or deployment views.

Figure 2 shows the converse case: a specialist language for each view. Boxes in the diagram represent views, while arrows indicate that two views have a correspondence relationship. In this case, there is a specialist language for each kind of view, and correspondences between views are maintained pairwise.

The most obvious problem with the specialist language approach is the proliferation of languages, and the problem of maintaining consistency between the

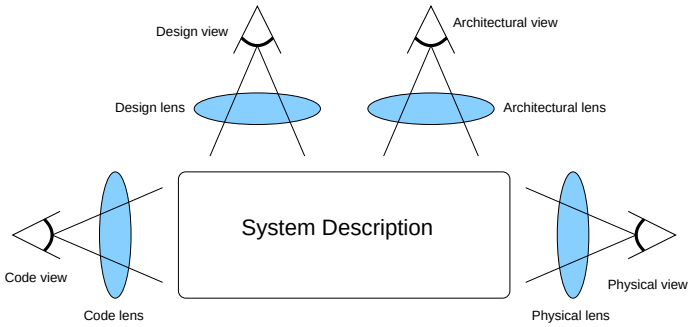


Fig. 1. Lingua Franca Approach

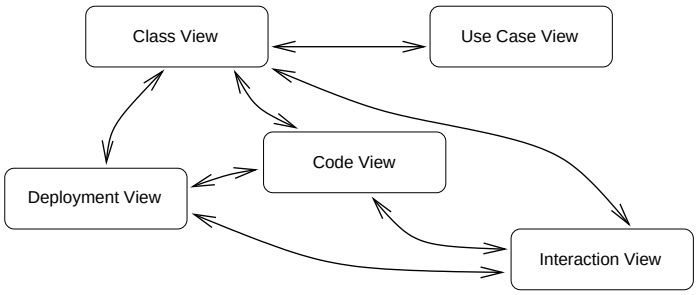


Fig. 2. Specialist Language Approach

views. With the 'lingua franca' approach, these problems are avoided. The most serious problem here is to create a suitable representation that is rich enough to record all information necessary for each potential view.

There are several potential approaches to building such an environment, including the following:

- Adapt current IDE technology (*eg*, Emacs, Eclipse). These are mainly code-level tools, with higher-level modelling considered as an extra facility.
- Adapt current design tools (such as ArgoUML). There is some level of traceability between the design (usually in a language such as UML) and the code, and support for multiple views (such as UML's different diagrams). ArgoUML also includes design critics [15], which go some way towards the analyses that are proposed here.
- Adapt current architectural tools (such as the Software Architect's Assistant). These have an architectural focus, with (currently) few code-level features.
- Start from scratch. This approach leads to an exact match of the system to our requirements, but is slow and inefficient in terms of development effort.

Modelling the architecture explicitly, with proper traceability from higher level constructs to the code will support evolution. Software will be constructed

using the appropriate techniques for each kind of entity, and the development environment will manage traceability and identify inconsistencies between different artefacts. The environment will enable software engineers to:

Manage change better because we can interconnections and dependencies are visible;

Discover inconsistencies within and between views automatically in some cases;

Know that our views are correct as they will be automatically extracted from the actual system.

This will reduce costs because: 'comprehension' tasks will produce correct information by definition; some kinds of 'unsafe' changes will be warned against or disallowed; high-level reuse will be supported by high-level knowledge; structure will be made explicit and thus degrading changes will become more obvious.

In order to develop a suitable development environment, the following steps are proposed:

- Develop reflective modelling of each view of a system;
- Allow a software engineer to change each view and have the actual system automatically updated;
- Check and enforce consistency within views;
- Check and enforce consistency between views;
- Support automatic architectural analysis within views;
- Support automatic architectural analysis across views;
- Allow the modelling of patterns and architectural styles in each view;
- Allow the modelling of patterns and styles across views.

To support dynamic replacement of components (*eg*, upgrading a component), it is necessary to support the transfer of state between the old and the new version. This is possible in a semi-automatic fashion [16]. Enforcing consistency requires the use of a suitable logic for describing constraints and evaluating models against them. Support for automatic modelling of patterns and the enforcement of consistency across views will require extensions to this logic to provide suitable mechanism for describing patterns in terms of the architectural features they demand.

6 Further Work

There are several potential problems with the kind of development environment proposed in this paper. In this section, some of them are identified and discussed, and potential means of addressing them are identified.

A system which dynamically supports software evolution must itself be capable of evolving. If it is to remain useful, it must be capable of supported features not considered at the time of its first development [17]. In this example,

it may become necessary to develop additional views, or to allow new kinds of constraints between views.

Some, especially in the extreme programming and agile methods communities, take the view that documentation (such as the architectural views proposed here) are superfluous and should be disposed of (for example: "There's this big assumption that diagrams, use cases and the like must be kept in synch with the code, and if they aren't they become completely useless. XP says to write them if you need them and then throw them away."[2]). This is (at least partly) due to the perceived effort involved with maintaining multiple views of the same system, the costs associated with inconsistencies, and the perception that the documentation is of little use anyway. Removing some sources of inconsistency will lessen the desire to discard documentation. Extreme programmers often take the view that the code and test cases together form the documentation, and resist any attempts to create other types of documentation (seen as "the tradeoff to get less functionality and more paper" [18]). On the other hand, there is much research and industrial effort expended in program comprehension and other reverse engineering tasks. This effort would be mitigated by the automatic generation and maintenance of the views proposed in this paper.

In order to carry this work forward, it is essential that a rigorous evaluation technique is devised, and objectively applied to the software and methods developed.

7 Conclusions

Software evolution is a hard problem, which is expensive to tackle. In this paper, a programme of research leading to a software engineering environment has been proposed. This environment will tackle some of the problems of software evolution identified by Lehman [2]. Using software reflection, multiple consistent views of the same system will be maintained, and problems will be identified (in some cases automatically). Reflection provides a means to maintain architectural models which are timely, correct, useful, and consistent.

The main problems to be tackled immediately are the creation of a suitable representation for software systems, definition of the properties which will be analysed, and creation of the mechanisms for ensuring consistency between multiple views.

References

1. Nosek, J.T., Palvia, P.: Software maintenance management: Changes in the last decade. Journal of Software Maintenance: Research and Practice **2**(3) (1990) 157–174
2. Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., Turski, W.M.: Metrics and laws of software evolution—The nineties view. In El Eman, K., Madhavji, N.H.,

[2] `http://c2.com/cgi/wiki?CritiqueOfUseCases`

eds.: Elements of Software Process Assessment and Improvement, Albuquerque, New Mexico, IEEE CS Press (1997) 20–32
3. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall (1996)
4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. S.E.I. Series in Software Engineering. Addison-Wesley (1998)
5. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes **17**(4) (1992) 40–52
6. Shaw, M.: Procedure calls are the assembly language of software interconnection: Connectors deserve first class status. Technical Report CMU/SEI-94-TR-2, Software Engineering Institute, Carnegie Mellon University (1993) Presented at the Workshop of Software Design, 1994. Published in the proceedings: LNCS 1994.
7. Kruchten, P.: Architectural blueprints—The "4+1" view model of software architecture. IEEE Software **12**(6) (1995) 42–50
8. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R.: Documenting Software Architectures: Views and Beyond. Addison Wesley (2002)
9. Cazzola, W., Savigni, A., Sosio, A., Tisato, F.: Architectural reflection: Bridging the gap between a running system and its specification. In: Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering, Florence, Italy (1998)
10. Masuhara, H., Yonezawa, A.: A reflective approach to support software evolution. In: Proceedings of International Workshop on the Principles of Software Evolution. (1998) 135–139
11. Dowling, J., Cahill, V.: Dynamic software evolution and the k-component model. In: Proceedings of the OOPSLA 2001 Workshop on Software Evolution. (2001)
12. Cazzola, W., Pini, S., Ancona, M.: Evolving pointcut definition to get software evolution. In: Proceedings of RAM-SE'04, the ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution, Oslo, Norway (2004) 83–88
13. Rank, S.: A Reflective Architecture to Support Dynamic Software Evolution. PhD thesis, University of Durham (2002)
14. Garlan, D., Schmerl, B.: Using architectural models at runtime: Research challenges. In: Proceedings of the European Workshop on Software Architectures, St Andrews (2004)
15. Robbins, J.E., Redmiles, D.F.: Software architecture critics in the Argo design environment. Knowledge-Based Systems **5**(1) (1998) 47–60
16. Vandewoude, Y., Berbers, Y.: Component state mapping for runtime evolution. In: Proceedings of the 2005 International Conference on Programming Languages and Compilers, Las Vegas, Nevada, USA (2005) 230–236
17. Bennett, K., Rajlich, V.: Software maintenance and evolution. In: The Future of Software Engineering, ACM Press (2000) 75–87
18. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)

The Role of Design Information in Software Evolution

Walter Cazzola[1], Sonia Pini[2], and Massimo Ancona[2]

[1] Department of Informatics and Communication,
Università degli Studi di Milano, Italy
cazzola@dico.unimi.it
[2] Department of Informatics and Computer Science
Università degli Studi di Genova, Italy
{pini|ancona}@disi.unige.it

Abstract. Software modeling has received a lot a of attention in the last decade and now is an important support for the design process.
Actually, the design process is very important to the usability and understandability of the system, for example functional requirements present a *complete* description of how the system will function from the user's perspective, while non-functional requirements dictate properties and impose constraints on the project or system.
The design models and implementation code must be strictly connected, i.e. we must have correlation and consistency between the two previous views, and this correlation must exist during all the software cycle. Often, the early stages of development, the specifications and the design of the system, are ignored once the code has been developed. This practice cause a lot of problems, in particular when the system must evolve. Nowadays, to maintain a software is a difficult task, since there is a high coupling degree between the software itself and its environment. Often, changes in the environment cause changes in the software, in other words, the system must evolve itself to follow the evolution of its environment.
Typically, a design is created initially, but as the code gets written and modified, the design is not updated to reflect such changes.
This paper describes and discusses how the design information can be used to drive the software evolution and consequently to maintain consistency among design and code.

1 Introduction

In the last few years, methodologies to automate part of the or the whole software life cycle have been widely studied in software system development. These methodologies can be used to create and/or maintain software, i.e. they are applicable to all the phases of the software life cycle.

Evolution and maintenance are phenomena more present in the software development area, since an intrinsic property of software in *real world* environment is its need to evolve. The laws of software evolution [11] said 'a program that is used in a real-world environment must change, or became progressively less useful in that environment'.

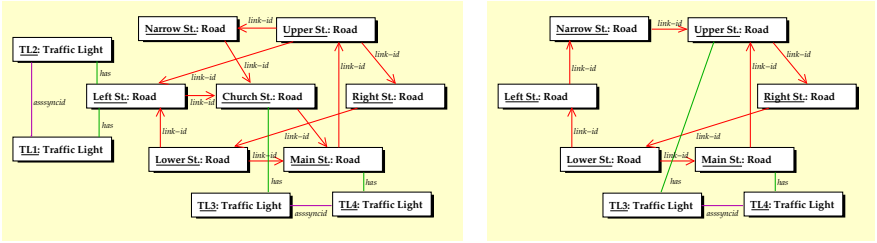
When a program evolves, it becomes more complex and automatic techniques to support these phenomena are fundamental to improve the management of unanticipated software evolution and the software efficiency.

The design process is very important to the usability and understandability of the system, for example functional requirements present a *complete* description of how the system will function from the user's perspective, while non-functional requirements dictate properties and impose constraints on the project or system. At the beginning design view and implementation view are consistent since one is derived and developed form the other, and we must preserve the correlation between the two view, and this correlation must exist for all the software life-cycle. But often, during the evolution and maintenance phase a discrepancy between the two view can occur, because the initial stages of development (the system specifications and the design) are ignored once the code has been developed. The main problem is the fact that models are shown like only an intermediate step in the software development life-cycle. This practice causes several problems when the system must evolve, because the evolution of only one view of the system causes a *gap* between them, that could create confusion, misunderstanding and mistakes.

The term *gap* has been stated by Rumbaugh [17], 'too often, there is still a gap between concept and execution' as a problem area in his retrospective review of O-O methodology. In the past developers have tried different ways to link design to implementation, and to tackle this gap problem. One of the noted effort in this endeavor is by looking at the gap from program viewpoint, describing the mapping from O-O programming languages properties to their corresponding UML concepts.

To most people software is the code that is the and result of the software development process. When a company starts developing a new product, it typically uses a clean forward engineering scheme and goes (iteratively or not) through requirements analysis, high-level design, design and implementation phases. This development process changes when a first implementation is finished. From then on, the implementation receives more and more attention. This restricted view of software is one of the main causes of the many problems associated with software development and its evolution. For example, a kind of evolution could require to add new functionality not available in the earlier version of the system. When the change is not applied also to the design view, it is hard for the manager, programmer and customer to have the opportunity to plan future directions, goals, schedule and the necessary budget, since the design view could not provide an immediate and understandable global view of the system consistent with the code. Moreover, it also hinders the integration of new functionality.

This problem, where implementation and design evolve in different directions because they are no explicitly related, is also know as *drift/erosion*. It is well-known that uncontrolled change to software can lead to increasing evolution cost caused by deteriorating structure and compromised system quality [10]. For complex systems, the need to carefully manage system evolution is a critical task. Wide integrated software system must continuously change to satisfy the evolving requirements. Adapting such software can be very difficult, because of the software size and complexity and variety of users with conflicting requirements. When there is a gap between the views it become difficult or impossible to know all the necessary changes to apply, therefore the evolution



(a) Standard layout (b) Modified layout

Fig. 1. Object diagrams: a) object diagram for UTCS before the evolution b) modified object diagram after the evolution.

cannot be planned *in-the-large*. To simplify the evolution process, it is necessary to have a *global view* of the system to apply all the evolutionary steps.

In our point of view, the global view may be well represented by the design information, because it is usually graphic, and more intuitive and understandable than the code.

The paper describes and discusses how the design information can be used to drive the software evolution and consequently to preserve the consistency among design and code views.

2 System Evolution through Design Information

With the help of an example, we describe how the design information, in our case UML [2] specifications, can be used to evolve a software system.

The *Urban Traffic Control System* (UCTS) is a typical example of system subject to unpredictable evolution, since the requirements can dynamically change and the system should adapt itself, as soon as possible, to such changes. Unanticipated software evolution is not something for which one can prepare during the design of a software system. Examples of unanticipated and hard to plan problems may be: road maintenance, traffic lights disruptions, car crashes, traffic jam and so on.

In the design phase, software engineers produce a lot of diagrams to identify domain concepts, to describe the business processes, and the physical structure of the environment, and so on.

The design information we consider can be divided in two categories: system structure and behavior.

- Structural Design Information is an explicit description of the structure of the system.
- Behavioral Design Information describe the computations and the communications carried out by the application objects, e.g. we consider object behavior, collaboration between objects, state of the object, and so on.

Structure and behavior of the system are modeled by class/object diagrams, sequence or collaboration diagrams and state diagrams.

The UTCS for our simple city, can be described by the object diagram showed in Fig. 1a, that defines the interconnections among roads and crossroads and a statechart that express the dependencies among traffic lights.

These diagrams well describe the system structure and behavior and its evolution should pass through these data to be well planned and integrated with the existing code.

We suppose that the system evolves because of a car accident that temporarily blocks the traffic flow in *Church Street*. To face a similar event forces several small changes in the whole city structure and, consequently, to the traffic flow. Several streets must be followed in a different direction to allow cars of reaching every place in the city. Traffic lights governing the traffic in and out of the blocked street must be turned off.

All the changes required must be applied both in design and implementation view to maintain the consistency between the views of the system.

The evolved system, as well as the original system, can be or better should be modeled by using, for example, UML diagrams. If this is the context of the evolution, then the original diagrams can determine the code to be adapted, the evolved diagrams specify how the code has to be adapted, and moreover the difference between such diagrams before and after the evolution represent the evolution itself.

Applying evolution at high level of abstraction, like at UML level has the advantage that software designers don't have to worry about the syntax of all possible programming language, and have a global view of the system. And as Blaha et al. [1] had said "Models allow a developer to focus on the essential aspects of an application and defer details".

The Fig. 1b show how the object model changes after this event. By comparing the diagrams in Fig. 1 it is possible to understand how the UCTS is configured after the evolution.

When the models are evolved it is necessary to map this evolution into the implementation code. To realize this issues, in our opinion, we must have got the design information available not only at design time, but also at compile and run time, to obtain this result our idea is to maintain this design information inside the system and to link every component of the UML diagrams to the respective portion of code using meta-data facility.

In this way, using the two object diagrams, before and after the evolution, and the meta-data inside the code it is possible to propagate the evolution to the code.

With the increased interest in evolution and maintenance, UML vendors seek ways to support software developers in applying maintenance. The problem with such tools is that the UML meta-model is inadequate to maintain the consistency between the model and the code while one of them gets evolved [15].

In the past developers have tried different ways to link design to implementation, and to tackle this problem, to date, no existing, general purpose methodology for this issue. We propose an infrastructure to dynamic adapt software system, called RAMSES [4]. Our project involves a reflective middleware whose aim consists of consistently evolving software systems, both design information and implementation code, against

run-time changes. This middleware provides the ability to change the system at run-time without stopping it, by using its design information. Many important applications must run continuously and without interruption. This is especially true of mission critical applications, such as telephone traffic control system, and air traffic control system. Our goal is to show that dynamic software evolution can be achieved in a practical manner that is flexible, efficient, robust, and easy to use.

RAMSES (Reflective and Adaptive Middleware for Software Evolution of Systems) performs two phases to carry out our dynamic self-adaptation. In the first phase, the RAMSES's meta-level extracts the design information as XMI (XML Meta-data Interchange [13]) schema from the base application and it reifies them in the meta-level to constitute the meta-data. Whereas, in the second phase, RAMSES's meta-level plans the dynamic adaptation of the base-level system, get the run-time events, evolves the meta-data against the detected event, checks the consistency, and finally reflects the modified data to the base-level.

In this paper, we treat only the role of the design information in software evolution.

3 Defining Meta-Data

In our opinion, a simple way for maintaining consistency is that design information are linked with the source code. To synchronize design and implementation, our proposal consists of using a mechanism that permits to express design as a set of meta-data over the implementation. To back our idea we have this definition: *Software design is an abstraction of implementation*. This definition is consistent with descriptions of design that can be found in software engineering literature [7], and moreover this definition said that design is explicitly related to the implementation. To express design information into the code of the system as meta-data, it is necessary to analyze what is a meta-data and how it can be used.

Meta-data literally *data about data*, or also *information about information* is a term used in several communities in different ways.

We can say that meta-data are structured information that describes, explains, locates or otherwise makes it easier to retrieve, use, or manage an information resource.

There are three main type of meta-data:

- *Descriptive meta-data* describes a resource for purposes such as discovery and identification, e.g., documentation.
- *Structural meta-data* indicates how compound objects are put together, e.g.,. they are used to describe the structure, layout and contents of an artifact.
- *Administrative meta-data* provides information to help manage an artifact, e.g., version control, location information, acquisition information.

An important reason to create descriptive meta-data is to facilitate the discovery of relevant information by describing an artifact with meta-data simplify its understandability by a program, promoting the interoperability. For our scope, we need to identify an artifact and to link up it with its design information. These meta-data could be automatically derived (extracted) from the design models of the system, and then automatically inserted into the code in the right places. To interleave the design information with the

related code, is the better way of rendering the code well documented and of granting the consistency and a prompt update of the design and implementation views. There are two ways to obtain a high coupling between design information and system code, the first consists of deriving the design information from the system code, e.g. by using tools for reverse engineering it is possible to obtain the UML diagrams from code, the second consists of deriving the skeleton of the program from the design information, e.g. tools as Rational Rose, and Poseidon permits of generating the code directly from the UML diagrams.

An implementing mechanism to link up design information and system code could be the meta-data facility present in a lot of programming languages. In general, meta-data describe the implemented code, by storing information regarding classes, methods, and types.

Several modern programming languages provide the programmers with a facility for annotating the code with meta-data. In the case of the Java programming language, for example, this facility allows developers to define custom *annotation types* and to *annotate* fields, methods, classes, and other program elements with *annotations* corresponding to these types. Development and deployment tools can read these annotations and process them producing additional Java programming language source files, XML document, or other artifacts to be used in conjunction with the program containing the annotations.

Our idea consists of using the Java meta-data facility to express design information over the implementation. In particular, we think to use as design information the UML diagrams, or more just to use the textual representation of the UML diagrams.

4 UML as Meta-Data

The UML is *de facto* the standard (graphical) language used during the design process, therefore our project considers its diagrams as a good representation of the system design information.

Our scope is to simplify the evolution/maintenance mechanism. That is, to render the changes required by the evolution immediately available both to the design models and to the implementation, all that we will have as direct consequence the maintenance of the consistency among the design and the code.

In our view, the UML diagrams and the code are seen as different views (design view and implementation view) on a software system, so that consistency between the views is preserved by modeling a coherent refactoring of these views. To realize our project, in particular we need to identify which diagrams are affected by the evolution and also which pieces of software these diagrams describe, in other words, we need a precise mapping between the two views mentioned above. The UML diagrams are, typically, available at design time, to maintain the mapping between the design and implementation view and then the consistency among design models and implementation model during the evolution phase, all this information must be available also at loading and run-time.

Our proposal consists of decorating the system code with the design information. In this way, we obtain a twofold advantage: to render the design information available

at run-time; and, to create a mapping between the design and the implementation view. The decoration will be realized by using Java annotations. Since UML is a graphical language it is difficult to deal automatically with its diagrams, therefore, we have to convert them into a textual representation to use them as meta-data.

We adopt, as most of the UML tools, the XML Meta-data Interchange (XMI [13]) as handling form for the design information. XMI provides a translation of UML diagrams in a text-based form more suitable for run-time manipulation. The XMI standard gives a guideline for translating each UML diagram in XML. Each diagram is assimilated to a graph whose nodes are the diagram's components (e.g., classes, states, actions and so on), and arcs represents the relation among the components. The graph is decorated with XML tag describing the properties of the corresponding UML component.

An example of the translation between UML diagram and XML is showed in the following listing.

```
<UML:Object xmi.id = 'Im169f2c98m10436f02a32mm7cfb'
            name = 'TL2' visibility = 'public' isSpecification = 'false'>
   <UML:Instance.classifier>
      <UML:Class xmi.idref = 'Im13db344bm1041dfafc5emm7ec5'/>
   </UML:Instance.classifier>
   <UML:Instance.linkEnd>
      <UML:LinkEnd xmi.idref = 'Im169f2c98m10436f02a32mm7cdb'/>
   </UML:Instance.linkEnd>
</UML:Object>

<UML:Object xmi.id = 'Im169f2c98m10436f02a32mm7cec'
            name = 'Left St' visibility = 'public' isSpecification = 'false'>
  <UML:Instance.classifier>
     <UML:Class xmi.idref = 'Im13db344bm1041dfafc5emm7c0a'/>
  </UML:Instance.classifier>
  <UML:Instance.linkEnd>
     <UML:LinkEnd xmi.idref = 'Im169f2c98m10436f02a32mm7cdc'/>
  </UML:Instance.linkEnd>
  <UML:Instance.ownedLink>
     <UML:Link xmi.id='Im169f2c98m10436f02a32mm7cdd' name='has' isSpecification='false'>
        <UML:Link.connection>
           <UML:LinkEnd xmi.id = 'Im169f2c98m10436f02a32mm7cdc' isSpecification = 'false'>
              <UML:LinkEnd.instance>
                 <UML:Object xmi.idref = 'Im169f2c98m10436f02a32mm7cec'/>
              </UML:LinkEnd.instance>
           </UML:LinkEnd>
           <UML:LinkEnd xmi.id = 'Im169f2c98m10436f02a32mm7cdb' isSpecification = 'false'>
              <UML:LinkEnd.instance>
                 <UML:Object xmi.idref = 'Im169f2c98m10436f02a32mm7cfb'/>
              </UML:LinkEnd.instance>
           </UML:LinkEnd>
        </UML:Link.connection>
     </UML:Link>
  </UML:Instance.ownedLink>
</UML:Object>
```

The above portion of XMI code translates part of the object diagram showed in Fig. 1a. In particular, it describes the object named `TL2` and `Left St` and their interconnection. The instances description of a class is grouped into the XMI tag `UML.Object`. The two occurrences showed in the above snippet describe respectively the object `TL2` and `Left St` in Fig. 1a. The name of the instance is contained in the attribute `name`, whereas the type of the instance is contained in the sub-tag `Class`. The `xmi.idref`

refers to description of the corresponding class into the class diagram. The `has` association is described through the tags `UML:Instance.linkEnd` that specify which instances are involved into the association and the tag `UML:Instance.ownedLink` that describes the nature of the association.

5 How maintain consistency between design and implementation

Our goal, consists of transform system models and code to implement required modifications, and propagate the transformation effect across the views, this can be faced using model-driven approach [9], i.e. to tackle the problem of evolving complex software systems by raising the level of abstraction from source code to models, and then maintain the consistency between model and code.

Since software designers think about evolution at the design level, and since design information provide an immediate and understandable global view of the system, it is quite natural to exploit the UML and its unified meta-model for expressing evolution.

Model-driven engineering is a software engineering approach that promotes the usage of *models* and *transformations* as primary artifacts. In our context, we can said design information is a model and implementation code is a model; and following this reasoning evolution and consistency between design and code is simply a model transformation.

We could use, in first phase of the evolution, *horizontal transformation* [6] on design models to describe the evolution of the system, and *vertical transformation* [6] in last phase of evolution to propagate the evolution at the implementation in order to maintain consistency among the models. A crucial aspect of model transformation is *model consistency*, since UML model is typically composed of many different diagrams, the consistency between all these diagrams need to maintained when any of them evolves.

Graph transformation seems to be a suitable technology and associated formalism to specify and apply model transformations for the following reasons: first graphs are a natural representation of models that are intrinsically graph-based; last graphs transformation theory provides a formal foundation for the automatic application of model transformations. Table 1 show a direct correspondence between software evolution and graph transformation.

Evolution	**Graph transformation**
Software Artifact	column Graph
Evolution	Graph Production
Composite Evolution	Composition of Graph Production
Evolution Application	Graph Transformation

Table 1. Correspondence between evolution and graph transformation.

6 How to Use Meta-Data for Evolution

To annotate the code with design information we have to extract from each UML diagram its XMI description that represents the perfect reification of the design information at run-time.

The meta-data provided by a single UML diagram are many and, above all, refer to different part of code, e.g., a class diagram describes every class in the system and their relations, and this information encode both the class definitions and the definition of some of their attributes, that have to be annotated. Since the main elements of a sequence diagram are objects and messages, from them it is possible to extract information regarding the instances of a class, and the interactions among them, e.g., creation, invocation of methods, destruction and so on. All this information is inserted into the body of methods as annotations.

The right positioning of the annotations (i.e., from UML design information to Java meta-data) is possible by mapping the UML model components to the OMG Interface Description Language (IDL) and achieved by applying the meta-object facility (MOF)-IDL mapping. The existence of this IDL representation of UML means that each UML element, such as associations, classes, actions, operations and so on, has an IDL description. The last step to complete the mapping consists of applying an IDL to Java mapping (e.g., Java _IDL).

To realize the insertion of the XMI code into the right code place, we use Java annotation facility. An annotation is a tag that we insert into the source code. It does not alter the semantics of the code, but instead allows an external application of recognizing and interpreting the tag for its purpose.

We go to explain how to use the annotations for our scope. Java allows the programmer to define and use user-defined annotation types. The facility consists of a syntax for declaring annotation types, a syntax for annotating declarations, APIs for reading annotations, and a class file representation for annotations. To create an annotation we need to define an annotation type first. Annotation types are defined like interfaces with an '@' (at) sign before the interface definition, and annotations are specified in the program source by using the '@' sign, followed by the annotation name.

The following listing shows the Java annotation type `CLASS` declaration, this kind of annotation will decorate the classes of the system, and the values of the attributes of each annotation derives by the corresponding class diagram.

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.CLASS)
public @interface CLASS{
       String XMI_ID();
       String XMI_name();
       ATTRIBUTE[] attributes();      // array of annotation type ATTRIBUTE
       ASSOCIATION[] associations(); //array of annotation type ASSOCIATION
       METHOD[] methods();           // array of annotation type METHOD
}
```

Note that the annotation type declaration is itself annotated. Such annotations are called meta-annotations. The first (`@Retention(RetentionPolicy.RUNTIME)`) indi-

cates that annotations with this type are to be retained by the virtual machine so they can be reflectively read at run-time. The second (`@Target(ElementType.CLASS)`) indicates that this annotation type can be used to annotate only class (type) declarations.

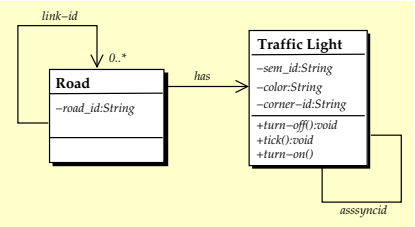


Fig. 2. An UTCS class diagram fragment

```
@Retention(RetentionPolicy.RUNTIME)

public @interface MESSAGE{
        String XMI_ID();
        String XMI_name();
        OBJECT Link-start();
        OBJECT Link-end();
}
```

Fig. 3. declaration of annotation type MESSAGE

The following annotation is derived from the class diagram showed in Fig. 2.

```
@CLASS(XMI_ID="Im13db344bm1041dfafc5emm7c0a",
        XMI_name="Road",
        attributes={@ATTRIBUTE(XMI_ID="Im13db344bm1041dfafc5emm7bf6",
                               XMI_name="road_id"),
                    @ATTRIBUTE(XMI_ID="Im13db344bm1041dfafc5emm7be4",
                               XMI_name="road_link")},
        associations={@ASSOCIATION(XMI_ID="Im13db344bm1041dfafc5emm7bba",
                                   XMI_name="has",
                                   multiplicity="Im13db344bm1041dfafc5emm7bbe",
                                   associationEnd="Im13db344bm1041dfafc5emm7ec5")},
        ...
)

public class Road{
        private String road_id;
        private String road_link;
        private Traffic_Light[] hastrafficlights;
...
}
```

The declaration of the annotation type `MESSAGE` showed Fig. 3 will be used to decorate the pieces of code, statements and so on, which map the message exchanged among objects, the values of the attributes of each annotation derives by the corresponding sequence and collaboration diagrams.

The Java annotation mechanism is not completely adequate for our purposes, because it permits annotating only the declarations whereas the UML diagrams have a finer granularity. The sequence diagrams have information about blocks of statement, and then the linked annotations would to be inserted inside the bodies, the the present mechanism of Java does not allow this.

To overcome this problem, we are extending the Java annotation mechanism and therefore the Java language to support custom annotations on arbitrary code blocks

or statements. This new Java dialect, called @Java extends the syntax of the Java language to allow a more general form of annotation. To carry out this job we are benefiting of our experience on [a]C# [5].

Obviously, the mechanism to insert the annotations into the application code is completely transparent to the developer because it is realized as a preprocessor that analyzes the design information and annotates on-the-fly the code by byte-code instrumentation.

In this way any kind of evolution could be developed at the design level (i.e. at the design view of the system), simply modifying all the necessary diagrams and dynamically realized by retrieving the related annotations and instrumenting the code according to the planned evolution.

7 Conclusions

This paper presented an approach to use the design information for the dynamic software evolution. This approach is based on some key concepts. The first concept is to maintain a strict correlation between the design information and the application code, in an automatic way. The second is to map all the evolutionary steps both in the design view and in the application code, so that the previous requirement is always satisfied. Into our work, we have used as design information UML diagram, and as programming language Java. The correlation between the two views of the system is realized thanks to the XMI description extracted by the UML diagrams, and thanks to the annotation facility of Java programming language.

References

1. M. Blaha and W. Premerlani. A catalog of object model transformations. In *WCRE '96: Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, page 87, Washington, DC, USA, 1996. IEEE Computer Society.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, third edition, February 1999.
3. Walter Cazzola, Antonio Cisternino, and Diego Colombo. [a]C#: C# with a Customizable Code Annotation Mechanism. In *Proceedings of the 10th Annual ACM Symposium on Applied Computing (SAC'05)*, Santa Fe, New Mexico, USA, on 13th-17th of March 2005. ACM Press.
4. Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. RAMSES: a Reflective Middleware for Software Evolution. In *Proceedings of the 1st ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04)*, in 18th European Conference on Object-Oriented Programming (ECOOP'04), pages 21–26, Oslo, Norway, on 15th June 2004.
5. Walter Cazzola, Sonia Pini, and Massimo Ancona. AOP for Software Evolution: A Design Oriented Approach. In *Proceedings of the 10th Annual ACM Symposium on Applied Computing (SAC'05)*, Santa Fe, New Mexico, USA, on 13th-17th of March 2005. ACM Press.
6. Robert B. France and James M. Bieman. Multi-view software evolution: A UML-based framework for evolving object-oriented software. In *ICSM*, pages 386–, 2001.
7. Adele Goldberg and Kenneth S. Rubin. *Succeeding with objects: decision frameworks for project management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

8. Timothy J. Grose, Gary C. Doney, and Brodsky Stephan A. *Mastering XMl: Java Programming with XMl, XML, and UML*. John Willy & Sons, Inc., April 2002.
9. Stuart Kent. Model driven engineering. In *IFM*, pages 286–298, 2002.
10. K. Kowalczykiewicz and D. Weiss. Traceability: Taming uncontrolled change in software development, 2002.
11. M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.
12. Bennet P. Lientz, E. Burton Swanson, and Gail E. Tompkins. Characteristics of Application Software Maintenance. *Communications of the ACM*, 21(6):466–471, June 1978.
13. OMG. OMG-XML Metadata Interchange (XMI) Specification, v1.2. OMG Modeling and Metadata Specifications available at `http://www.omg.org`, January 2002.
14. J. Pierce, M. D. Smith, and T. Mudge. Instrumentation Tools. In Anthony Finkelstein, editor, *Fast Simulation of Computer Architectures*, chapter 4. Kluwer Academic Publishers, Boston, MA, USA, 1995.
15. Tom Mens Pieter Van Gorp, Hans Stenten and Serge Demeyer. Enabling and using the uml for model-driven refactoring. In Stéphane Ducasse Serge Demeyer and Kim Mens, editors, *Proceedings WOOR'03 (ECOOP'03 Workshop on Object-Oriented Re-engineering)*, pages 37–40. Universiteit Antwerpen, July 2003.
16. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
17. James E. Rumbaugh. Modeling through the years. *JOOP*, 10(4):16–19, 1997.

Tools and Middleware for Softare Evolution

Chairman: Takuo Watanabe, Tokyo Institute of Technology, Japan

Towards a Meta-Modelling Approach to Configurable Middleware

Nelly Bencomo[1], Gordon Blair[1], Geoff Coulson[1], and Thais Batista[2]

[1] Comp. Dept., InfoLab21, Lancaster University, Lancaster, LA1 4WA, UK
{nelly, gordon, geoff}@comp.lancs.ac.uk
[2] Comp. Scs. Dept., UFRN, 59072-970, Natal - RN, Brazil
thais@ufrnet.br

Abstract. In our research we are studying how to combine modelling, meta-modelling, and reflection to systematically generate middleware configurations that can be targeted at different application domains and deployment environments. Despite this generality our approach adopts a uniform set of concepts: components, components frameworks, and reflection. Components and component frameworks provide structure, and reflection provides dynamic (re)configuration and extensibility for run-time evolution and adaptation. In this paper we present meta-models that capture the generality inherent to our approach and form a basis for automatic generation of extensible "middleware families" that can be instantiated differently depending on the application domain, QoS, deployment environment and degree of dynamic re-configurability required.

1. Introduction

Reflection has now emerged as an important technique in the support of more configurable and re-configurable middleware [1]. A number of experimental reflective middleware platforms have been developed and used in industry. In our research [14] we complement the use of reflection with the notions of *components* (language-independent units of dynamic deployment), *component frameworks* (collections of components that address a specific area of concern and accept additional plug-in components) [12], and *middleware families* (abstract collections of component frameworks that are tailored to specific application domains and deployment environments). In addition, middleware families employ reflection [9,11] to discover the current structure and behaviour of the family instantiation, and to allow selected changes at run-time for dynamically consistent evolution and adaptation. The end result is a flexible middleware architecture that can be straightforwardly specialised to a wide range of domains including multimedia, embedded systems [4], and mobile computing [7].

Challenging new requirements emerge when working with such architecture. Middleware developers are faced with a large number of variability decisions when planning configurations at various stages of the development cycle. These include decisions in design, component development, integration, deployment and even at run-time. These factors make it error-prone to manually guarantee that all these decisions

are consistent. In addition, such *ad hoc* approaches do not offer a formal foundation for verification that the ultimately configured middleware will offer the required functionality.

To address these issues, we are currently investigating the use of Model-Driven Software Development (MDSD) techniques. MDSD is a new paradigm that encompasses domain analysis, meta-modelling and model-driven code generation. We believe that MDSD has great potential in systematically generating configurations of middleware families. In our MDSD-based approach, we propose the capture of the fundamental component-based programming concepts in a core set of meta-model elements called a *kernel*. All middleware family members regardless of their domain share this minimum set of concepts. On top of this, we propose a set of extension meta-models which capture the extensibility characteristics of our underlying (concrete) component model and which can be plugged in as appropriate.

In the remainder of this paper we first, in section 2, introduce the main concepts of our concrete component model. Then, in section 3, we discuss the MDSD-based modelling of these concepts and show our current model-based realisation. Then in section 4 we discuss the application of the meta-models and models in generating middleware families. Finally, we present conclusions and discuss future work in section 5.

2. A happy family: Lancaster's reflective middleware

As mentioned, our notion of middleware families is based on three key concepts: *components*, *components frameworks*, and *reflection*. Both the middleware platform and the application are built from interconnected sets of components. The underlying component model is based on OpenCOM[5], a general-purpose and language independent component-based systems building technology. OpenCOM supports the construction of dynamic systems that may require run-time reconfiguration. It is straightforwardly deployable in a wide range of deployment environments ranging from standard PCs, resource-poor PDAs, embedded systems with no OS support, and high speed network processors. Components are complemented by the coarser-grained notion of *component frameworks* (CFs) [12]. A CF is a set of components that cooperate to address a required functionality or structure (e.g. service discovery and advertising, security etc). CFs also accept additional 'plug-in' components that change and extend behaviour. Many interpretations of the CF notion foresee only design-time or build-time plugability. In our interpretation run-time plugability is also included, and CFs actively police attempts to plug in new components according to well-defined policies and constraints.

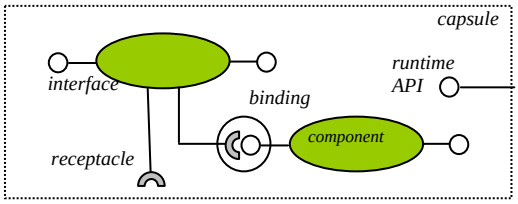


Figure 1. The OpenCOM main concepts

The basic concepts of OpenCOM are depicted in figure 1. Specifically, capsules are containing entities that offer a component run-time (CRT) API for the loading, binding etc. of components. Components are language-independent units of deployment that support interfaces and receptacles (receptacles are "required interfaces" that indicate a unit of service requirement). Bindings are associations between a single interface and a single receptacle. The CRT API is roughly as follows (many details have been omitted for reasons of space). The role of the *notify()* call is discussed below.

struct load(comp_type name);
status unload(struct t);
comp_inst bind(ipnt_inst interface, ipnt_inst receptacle);
status notify(callback c);

The architecture into which this fits is shown in figure 2. The layer immediately above the CRT consists of the so-called caplet extensions and a set of reflective extensions. The role of the caplet CF is to provide structured support for extensibility at the deployment environment level in terms of pluggable caplets which are subscopes within a capsule that are used for a variety of purposes including sandboxing and supporting heterogeneous programming languages. The reflective services then provide generic support for target system reconfiguration—i.e. inspecting, adapting and extending the structure and behaviour of systems at runtime (see below). Both the caplet and reflective extensions are independently and optionally deployable (using the CRT), and their precise configuration can be tailored to the needs of the target system and deployment environment.

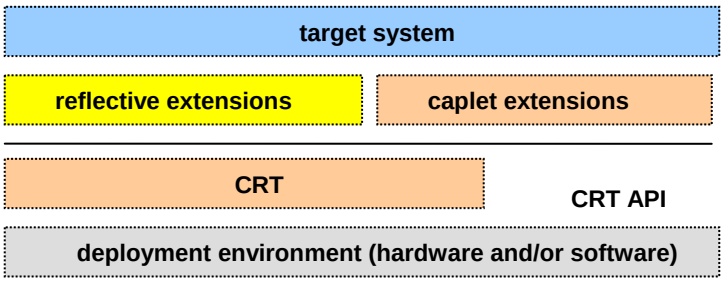


Figure 2.OpenCOM Architecture

The Reflection Services

As mentioned, reflection is used to support introspection and adaptation of the underlying component/ CF structures [1]. A pillar of our approach to reflection is to provide an extensible suite of orthogonal meta-models each of which is optional and can be dynamically loaded when required, and unloaded when no longer required. The meta-models manage both evolution and consistency of the base-level system. The motivation of this approach is to provide a separation of concerns at the meta-

level and hence reduce complexity. Three reflective meta-models are currently supported:

The architecture meta-model represents the current topology of a composition of components within a capsule; it is used to inspect (discover), adapt and extend a set of components. For example, we might want to change or insert a compression component to operate efficiently over a wireless link. This meta-model provides access to the implementation of the meta-component that has a component graph where components are nodes and bindings are arcs. Inspection is achieved by traversing the graph, and adaptation/extension is realized by inserting or removing nodes or arcs.

The interface meta-model supports the dynamic discovery of the set of interfaces defined on a component; support is also provided for the dynamic invocation of methods defined on these interfaces [1]. Both capabilities together enable the invocation of interfaces whose types were unknown at design time.

The interception meta-model supports the dynamic interception of incoming method calls on interfaces and also the association of pre- and post-method-call code [1]. The code elements that are interposed are called interceptors. For example, in the above wireless link scenario we might want to use an interceptor to monitor the conditions under which the compressor should be switched.

Causal connection between the base-level system and the meta-models is achieved via the above-mentioned *notify()* operation from the CRT API. This operation allows meta-models to register a callback that is invoked every time a subsequent call (bind, load, etc) is made on the CRT. The callback invocation contains all the parameter values of the call and so gives the callback holder a complete picture of all activity in the capsule. As an example, the architecture meta-model uses a notify callback to keep itself updated with information associated with the internal topology of the capsule contents. In case a meta-model needs to change the base-level configuration in some way, it simply invokes the respective operation (bind, load etc.) in the API. In this way, the causal-connection relation between the base and the meta level is maintained.

3. Modelling

In this section we present a set of meta-models specified using UML [15]. This set of meta-models support our abstract specification of families of middleware. Figure 3 shows the three packages that comprise the OpenCOM meta-model. The *Kernel* package includes the fundamental model elements of OpenCOM: viz. component, capsule, interface, receptacle, binding, composite component and component framework. On top of this, the *Caplet Extensions* package includes the fundamental model elements of the caplet extensions in OpenCOM: viz. caplets, loaders, and binders. This package provides structured support for extensibility at the deployment environment level in terms of pluggable extensions. The *Reflective Extensions* package

includes the fundamental model elements of the OpenCOM reflective meta-models (see the three reflective packages in Figure 3).

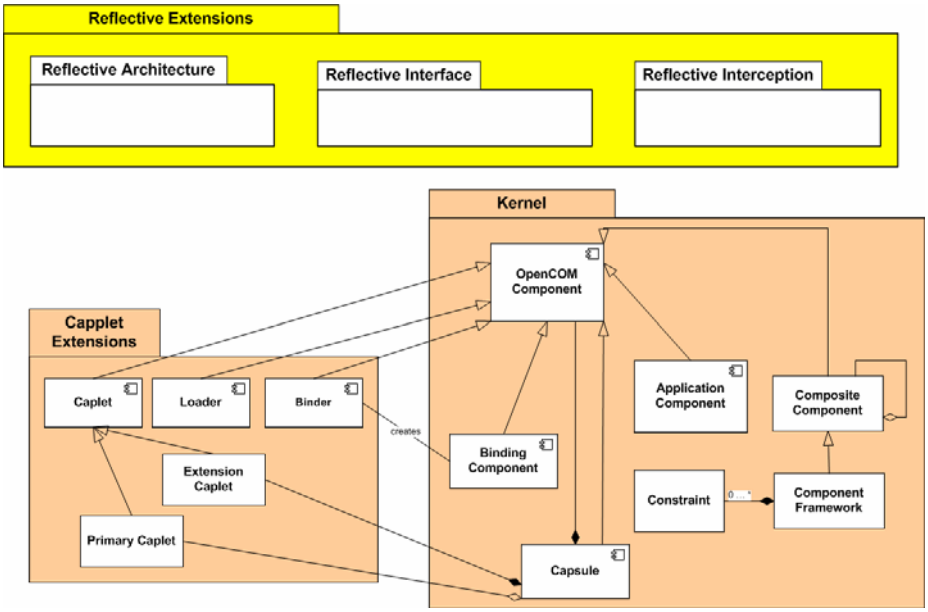


Figure 3. The OpenCOM Meta-model

Crucially, these meta-models are organised in terms of a structure with two orthogonal dimensions. One axis of this is the UML hierarchy (with its layer M0, M1, M2, and M3) and the other is a division into base level and (reflective) meta-level[1]. This is illustrated in Figure 4. All the meta-models (packages) we have described above populate the UML M2 level; however, while the kernel and caplet extensions packages live in the base-level, the reflective extensions live in the meta-level. The intention is that middleware family specifications will populate the UML M1 level and, of course, instantiations of these specifications will populate the UML M0 level. As a consequence, in implementation the meta-objects are optional and can be dynamically loaded/unloaded when required.

Figure 4 also shows an example instantiation of the model in terms of a very simple application example that illustrated the intended use of the model. This is a "calculator" which contains three sub-components; an adder, a multiplier and a calculator. The calculator component offers the services of adding and multiplying based on the services of the adder and multiplier components.

[1] Note that there is a potentially-confusing terminological clash here between the UML "meta-level" and "reflective meta-levels". These two concepts are entirely distinct; nevertheless we are forced to employ both of these terms because they are so well established in their respective communities.

Figure 4 only shows details about the reflective architecture package. Work related to the reflective interception and interface packages have been done but it is not shown in this paper for reasons of space.

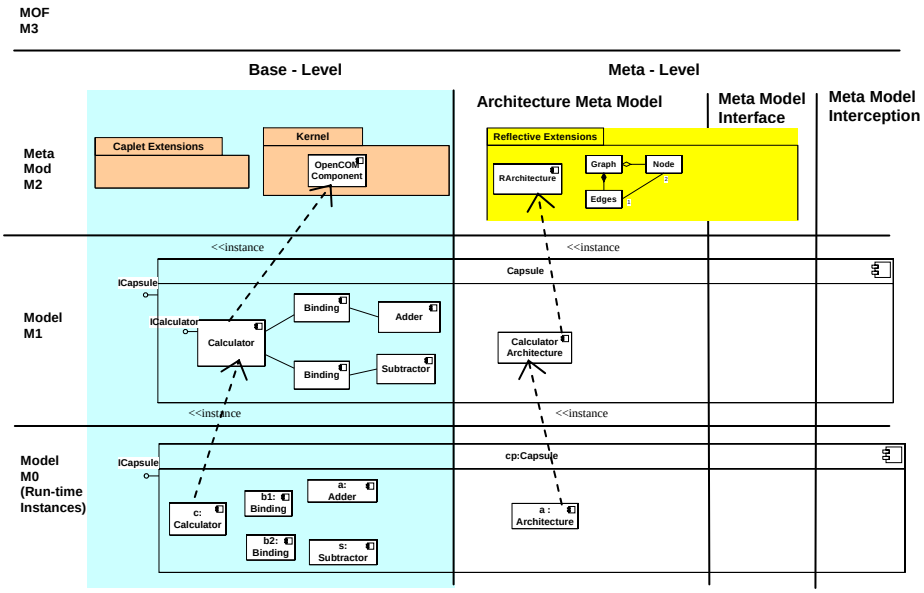


Figure 4. UML Models and Reflective Architecture of a Calculator Configuration

Finally, Figure 5 shows how the causal connection between the base and meta-levels is generically modelled in terms of a UML sequence diagram. This is based on the semantics of the *notify*() operation discussed above. The sequence diagram represents (part of) the dynamic behaviour specification of OpenCOM as opposed to the static structure of the models that was shown in Figure 4. Different models at level M1 will reuse or instantiate this generic causal-connection diagram.

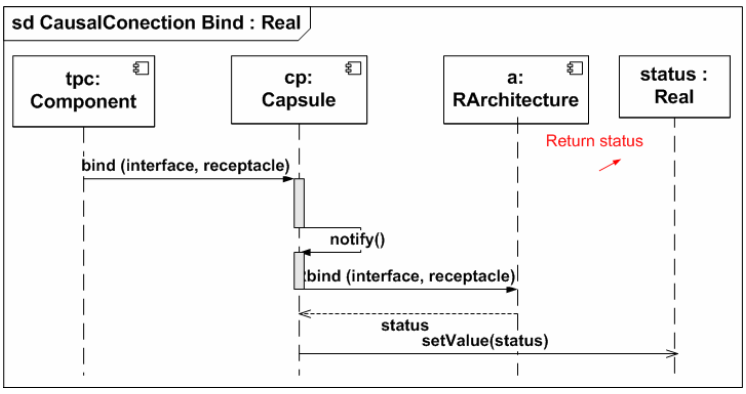


Figure 5. Sequence Diagram for Causal Connection when calling the *bind* operation

4. Discussion

We now turn to the application of the meta-modelling concepts to give support to the specification and efficient generation of middleware families. We apply the approach in terms of both *configuration* (i.e. establishing an initial set of components in a target deployment environment, and *reconfiguration* (i.e. making changes to the initial set of components at runtime).

In outline, different middleware configurations are generated from models that are written in terms of the above meta-models. The models are sufficiently abstract that a number of different concrete OpenCOM-level configurations of components can be generated from them (i.e. the mapping of UML to Open COM components is not simply 1:1). The concrete configurations that are generated are determined by the following dimensions of variability:

- quality of service (QoS)
- deployment environment
- (re)configurability

The *QoS dimension* allows the abstract-to-concrete mapping to be influenced by consideration such as mobility (e.g. whether the components should be able to migrate), dependability (e.g. whether certain components should be replicated), or security (e.g. whether certain components are allowed to dynamically load other components). For example, consider an application with a QoS requirement for mobile code. In the generated Open COM-level configuration, this will indicate the inclusion of the caplet extension. The caplet extension is needed because of its support for sandboxing untrusted components, and for its provision of specialised loaders that are able to load remote objects. It will also indicate the inclusion of a security CF to validate the remote components. All of this machinery will be transparently instantiated without having to be explicitly present in the UML model.

The *deployment environment dimension* refers to the resource capabilities of the hardware/software environment in which the system will be deployed. Consider, for example, a distributed application that is deployed in a heterogeneous environment consisting of PCs, PDAs and resource-poor sensor motes. While it would be unproblematic to deploy the whole of the reflective extensions package on the PCs and maybe the PDAs, this may not be possible on the sensor motes where perhaps only components related to the kernel package might be deployed. This would preclude the use, e.g., of the caplet extensions on the motes and thus restrict the functionality available in that environment. We are currently working on the design of specific middleware configurations addressing embedded systems domains where extremely resource-constrained environments are found [4].

Finally, the *configurability dimension* refers to the degree of reflective support that will be required at runtime. This essentially determines which of the reflective extensions will be instantiated. For example, if performance monitoring for QoS purposes is required, the interception meta-model would be included but not the others. Alternatively, if the application might need components to be added or replaced at runtime, the architecture meta-model would additionally be needed [8].

The above example raised the possibility of multiple dimensions potentially cross-cutting each other (i.e. QoS and configurability). Such cross-cutting is expected to be a common occurrence. Aspect Oriented Software Development (AOSD) offers techniques that may help us address this problem.

5. Conclusions and Future Work

We have developed a set of meta-models that assist in the specification of middleware families and in the generation of specific family members which are determined by *quality of service*, *deployment environment* and *configurability* dimensions of variability. The meta-models capture the main concepts of the design philosophy of our middleware family: components, components frameworks, reflection for dynamic (re)configuration and extensibility. First, a package called *Kernel* containing the meta-model of the fundamental concepts is proposed. The UML specifications of reflective meta-models and caplets as extensions of the kernel are then presented in the packages *Caplet Extensions* and *Reflective Extensions*. As a result, at runtime the components/CFs related to caplets extensions and the meta-objects are optionally dynamically (un)loaded when pluggable extensions and reflective capabilities are required. In the particular case of the modelling of reflection, this paper describes how meta-models and models specify the causal connection between the base and meta-level.

We are now investigating how to generate different middleware configurations while keeping decisions that are generic to a set of configurations at the meta-model level design. More work has to be done to completely identify the variability among the related configurations (members) of middleware families to support an efficient generation of configurations. Another key area of future work will be to maintain the UML models at runtime and to keep this causally connected with the underlying running system. We think that it will give principled support for run-time evolution and adaptation. We also plan to investigate how solutions for the crosscutting problems we described can be found in the area AOSD[13]. In this sense, AOSD techniques should also be applicable and defined at the design level using UML as computational reflection has been applied and identified in our approach.

Acknowledgement: This research is part-financed by the RUNES project. RUNES is supported by research funding from European Commission's 6th framework Programme under contract number IST-004536.

References

1. Bencomo N., Blair G.: Raising a Reflective Family, Models and Aspects - Handling Crosscutting Concerns in MDSD, ECOOP, Scotland, 2005
2. Blair, G., Coulson, G., Grace, P.: Research Directions in Reflective Middleware: the Lancaster Experience, Proc. 3rd Workshop on Reflective and Adaptive Middleware (RM2004), (2004), 262-267
3. Clark, M., Blair, G.S., Coulson, G.: Parlavantzas, N., An Efficient Component Model for the Construction of Adaptive Middleware, Proc. IFIP Middleware 2001, Germany, (2001)

4. Costa, P., Coulson, G., Mascolo, C., Picco, G.P., Zachariadis, S.: The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems, PIMRC05,(2005)
5. Coulson, G., Blair, G.S., Grace, P., Joolia, A., Lee, K., Ueyama, J.: A Component Model for Building Systems Software, Proc. IASTED Software Engineering and Applications (SEA'04), USA, (2004)
6. Gabriel R., Bobroe D., White J., CLOS in Context – The Shape of the Design Space, in Object-Oriented Programming – the CLOS perspective, Chapter 2, MIT Press, 1993, 29-61
7. Grace P., Blair G. Samuel S.: ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability. Proc in International Symposium on Distributed Objects and Applications (2003)
8. Grace, P., Coulson, G., Blair, G., Mathy, L., Yeung, W.K., Cai, W, Duce, D., Cooper, C.: GRIDKIT: Pluggable Overlay Networks for Grid Computing, Proc. Distributed Objects and Applications, (2004)
9. Maes, P., "Concepts and Experiments in Computational Reflection", Proc. OOPSLA'87, Vol. 22 of ACM SIGPLAN Notices, pp147-155, ACM Press, 1987.
10. Okamura H., Ishikawa Y., Tokoro M.: Metalevel Decomposition in AL-1/D, Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software (1993), 110-127
11. Smith B.: Reflection and Semantics in a Procedural Language. PhD thesis, MIT Laboratory of Computer Science, (1982)
12. Szyperski C.: Component Software: Beyond Object-Oriented Programming, Addison-Wesley, (2002)
13. Aspect-Oriented Software Development Community: http://aosd.net/
14. Middleware at Lancaster: http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/index.php
15. OMG Unified Modelling Language - UML : http://www.uml.org/

MADAPT: Managed Aspects for Dynamic Adaptation based on Profiling Techniques

Robin Liu[1], Celina Gibbs[1] and Yvonne Coady[1]

[1] University of Victoria, Department of Computer Science,
PO Box CSC Victoria, British Columbia, Canada
{cliu, celinag, ycoady}@cs.uvic.ca

Abstract. An increasingly significant cost associated with dynamically adaptive middleware is the complexity of managing the code responsible for adaptive behaviour. It is not surprising that, due to the fine-grained nature of trace-data collection and the subtle adaptation that can result, more flexible systems are typically more complex to manage. This paper makes the case for using aspect-oriented programming (AOP) [6] as a means to achieve adaptive middleware based on fine-grained, customizable, profiling techniques. A feasibility-study combining Java Management Extensions (JMX) [3] and AOP shows the effectiveness of the synergy between the management support for application services offered by JMX, and the structured support for crosscutting concerns offered by AOP.

1 Introduction

Management of adaptive middleware must provide a comprehensive means of structuring code that is responsible for adaptation. Complex collections of probes for trace-data are often scattered throughout the system in an unclear way, and accompanied by subtle reconfiguration strategies that become tangled with other concerns in the system.

Java Management Extensions (JMX) [3] is a standard architecture specifically designed for management of commonly used Java-based services. By defining and implementing a management interface, any java object can be a Managed Bean (MBean) and use JMX Agent services (dynamic loading, monitoring, timer and relationship services), connectors and protocol adaptors. Though this support enhances the manageability of certain concerns associated with adaptive behaviour, the inherent scattering and tangling of fine-grained profiling instrumentation across an application still remains.

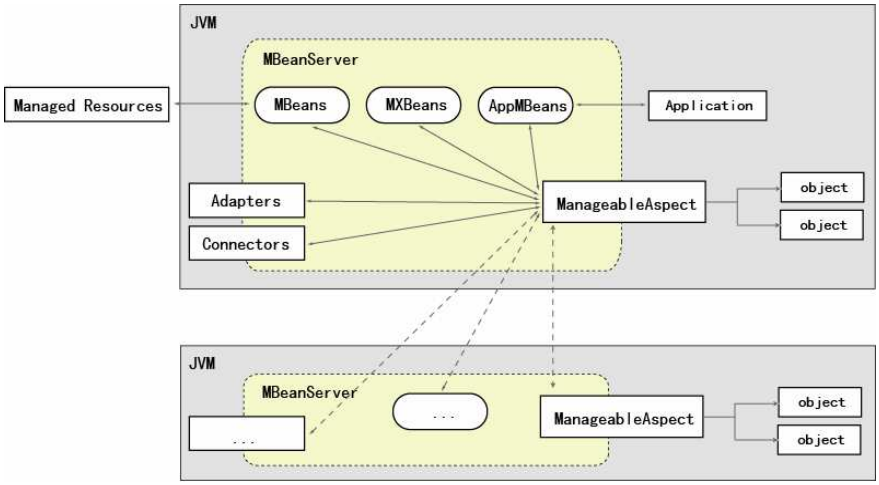


Fig. 1. High-level architecture: aspects (denoted as ManageableAspect) are managed as MBeans.

The goal of aspect-oriented programming (AOP) [6] is to provide structure for cross-cutting concerns. In our experiment, we leveraged the management facilities offered by JMX and the structured approach to crosscutting concerns offered by AOP to construct application-specific adaptive thread and memory management. A high-level overview of this architecture is highlighted in Figure 1. MBeans structure the management of standard JMX services, and aspects as MBeans further structure the crosscutting code responsible for adaptation management.

The rest of the paper proceeds as follows: Section 2 provides background for JMX, AOP, and profiling. The details of our MADAPT (Managed Aspects for Dynamic Adaptation based on Profiling Techniques) proof-of-concept prototype are described in Section 3. Section 4 presents conclusions and future work.

2 Background and Related Work

The motivation to combine JMX technology with AOP methodology stems from a common underlying theme: concerted efforts to improve the modularity of complex systems. The impact modularity has on productivity is becoming increasingly evident as development moves to open source projects.

Profiling is a classic example of a concern that does not typically adhere to traditional structural boundaries, but is a necessary ingredient for dynamically adaptable systems. Before launching into details of our experimental study with MADAPT, we briefly provide background showing how these three pieces of the puzzle, JMX, AOP and profiling, all fit together.

2.1 JMX Background

Originally known as Sun's JMAPI, JMX is gaining momentum as an underlying architecture for J2EE servers. MBeans act as wrappers, providing localized management for applications, components, or resources in a distributed setting. MBean servers are a registry for MBeans, exposing interfaces for local/remote management. An MBean server is lightweight, and parts of a server infrastructure are implemented as MBeans. JMX thus supports highly modular and customizable server architectures.

2.2 AOP Background

AOP is gaining momentum as a methodology facilitating modularization of crosscutting concerns – concerns that are present in more than one module, and cannot be better modularized through traditional means.

An aspect is a module that structures crosscutting implementation. Looking at an aspect, a developer can see both the internal structure of a crosscutting concern, and its interaction with the rest of the program during execution. As brief example of the mechanisms used in the particular incarnation of aspect-oriented programming used in MADAPT, AspectWerkz [2], the aspect below captures all calls to the constructor of myObject and simply prints tracing information:

```
package myapp;
import org.codehaus.aspectwerkz.joinpoint.JoinPoint;
public class MyAspect {
   /**
    * @Around call(myapp.myObject.new(..))
    */
    public void addToCreate(JoinPoint joinPoint) {
        System.out.println("before creating...");
        joinPoint.proceed();
        System.out.println("after creating...");
    }
}
```

The *@Around* annotation associates the execution of the aspect's *addToCreate*() method with the execution of the constructor .

2.3 Profiling Background

Log-based performance profiling has been used in distributed systems [5], operating systems [11], and adaptive applications [8], and continues to be used for performance analysis and fault detection. A common characteristic of implementations based on profiling technology is the fact that instrumentation requires invasive changes to multiple modules of the target system. In essence, profiling is a classic crosscutting concern.

Though we are ultimately advocating a customizable, application-specific approach to profiling, it is important to note that tools for profiling based on well-

known published interfaces are gaining momentum. Magpie [7] profiles websites to measure resource consumption (CPU, disk, network usage) of HTTP requests, and builds probabilistic models for performance prediction, tuning and diagnosis. Pinpoint [4] uses a similar approach, relying on profiling, analysis and anomaly detection for fault detection. Commercial request tracing systems include PerformaSure[9] and AppAssure [1].

3 Feasibility Study

This study demonstrates the feasibility of combining JMX and AspectWerkz to provide synergistic JMX/AOP management for dynamic adaptation. The two examples in our prototype, the ThreadManagerAspect and the MemoryManagerAspect, show how aspects can be used to: (1) structure application-specific crosscutting concerns, (2) implement JMX MBeans, and (3) enable crosscutting concerns to be managed through standard JMX services.

3.1 ThreadManagerAspect

The ThreadManagerAspect structures a crosscutting concern associated with the creation of new threads in a given application. This aspect impacts all points during execution when a new thread is created. For example, if an application creates a thread for each HTTP request it receives, and (within another class/component) creates a thread for each disk request, all of these points would be structured within this aspect.

As shown in Figure 2 (left-hand side), the ThreadManagerAspect uses the ThreadMXBean to monitor the thread system of the local Java virtual machine. Platform MXBeans are part of the Java 2 Platform, Standard Edition (J2SE) 1.5 release. MXBeans provide a standard monitor management interface for the JVM as well as the operating system. This allows the aspect to be managed using standard JMX services. Figure 2 (right- hand side) also depicts how this aspect crosscuts all objects of an application that create threads. A closer look at the implementation details of this aspect shows how the MBean services and the crosscutting structure are combined within the aspect.

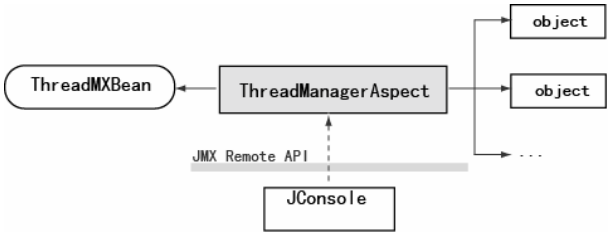


Fig 2. ThreadManagerAspect

3.1.1 Implementation Details

In order to make the ThreadManagerAspect an MBean, and hence manageable through JMX services, we first define an interface shown at the top of Figure 3 (lines 1-4), the ThreadManagerAspectMBean interface, and use it in the aspect. Methods in this interface will be exposed to JMX services.

The ThreadManagerAspect has a local variable maxThreadCount (line 11), which is the application-specific upper limit of the number of live threads. Ultimately, it is maxThreadCount that we intend to expose for local/remote management by implementing the two operations: getMaxThreadCount() and setMaxThreadCount(), defined in the interface.

In the constructor of the ThreadManagerAspect (lines 13-17), a reference to the platform MBeanServer from the ManagementFactory establishes the current server (there can be more than one MBeanServer per-JVM), and then registers the aspect as an MBean with that server. The constructor also obtains the ThreadMXBean from the ManagementFactory.

As indicated by the @Around AspectWerkz annotation (line 20), the logMethod is defined to execute whenever there is a method call to construct a Thread (or Thread subtype) from within any class in the prototypeApplication package. In this way, the crosscutting concern spans all objects in the prototypeApplication that create threads. At the point a new thread is being constructed, the logMethod() is invoked. This method retrieves the number of live threads from the ThreadMXBean and compares this value to the pre-defined maxThreadCount. If the number is greater that the limit, our prototype simply throws a runtime exception. Otherwise, the execution proceeds to the constructor (line 27).

As an MBean, the ThreadManagerAspect is eligible for JMX services, such as JConsole, the monitoring and management GUI shown in Figure 4.

```
1  public interface ThreadManagerAspectMBean {
2    public void setMaxThreadCount(int maxThreadCount);
3    public int getMaxThreadCount();
4  }
5
6  /**
7   * @Aspect perJVM
8   */
9  public class ThreadManagerAspect implements ThreadManagerAspectMBean {
10   private int maxThreadCount = 20;
11   private final ThreadMXBean threadMB;
12
13   public ThreadManagerAspect(final CrossCuttingInfo info) {
14     server = ManagementFactory.getPlatformMBeanServer();
15     server.registerMBean(this, ...);
16     threadMB = ManagementFactory.getThreadMXBean();
17   }
18
19   /**
20    * @Around call(java.lang.Thread+.new(..)) && within(prototypeApplication.*)
21    */
22   public Object logMethod(final JoinPoint joinPoint) throws Throwable {
23     int threadCount = threadMB.getThreadCount();
24     if (threadCount > maxThreadCount) {
25       throw new RuntimeException("...");
26     }
27     final Object result = joinPoint.proceed();
28     return result;
29   }
30
31   public int getMaxThreadCount() {
32     return this.maxThreadCount;
33   }
34
35   public void setMaxThreadCount(int maxThreadCount) {
36     if (maxThreadCount <= 0) {
37       throw new IllegalArgumentException("Maximum thread count can not be negative");
38     }
39     this.maxThreadCount = maxThreadCount;
40   }
41 }
```

Fig 3. ThreadManagerAspect Code

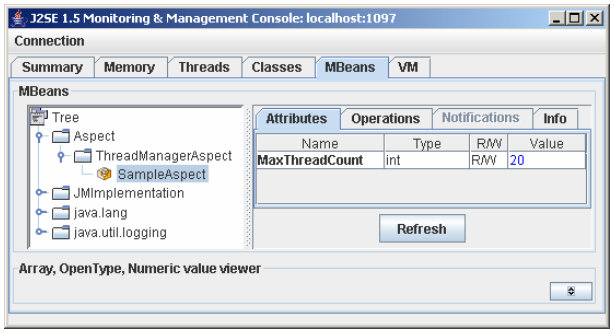


Fig 4. JConsole screenshot for the ThreadManagerAspect.

The SampleAspect shown in the JConsole is the ThreadManagerAspect instance registered in the MBeanServer as an MBean. JMX services can connect to any local/remote JVM, and all the running MBeans can be displayed and managed through this interface. JConsole also allows developers to view MBean information (registered name and class), view/set MBean attributes, call MBean operations, and view MBean notifications

Figure 4 shows the results of having the registered MBean ThreadManagerAspect discovered by a remote JConsole (using JMX Remote API), and maxThreadCount is shown as a read/write attribute. In the console, the maxThreadCount can be set to a new value which is passed to the ThreadManagerAspect's setMaxThreadCount() method.

3.1.2 Beyond Proof-of-Concept

The ThreadManagerAspect presented here is a simple proof-of- concept, demonstrating the synergy between JMX and AOP, but we plan implement more sophisticated thread management strategies. For instance, instead of simply throwing a RuntimeException, the aspect could:

a) block the method call, or
b) expose further strategies as manageable operations through the ThreadManagerAspectMBean interface

Furthermore, to reduce the thread creation overhead and garbage collection costs, the ThreadManagerAspect could also implement an adaptive pool of reusable threads and could dynamically adjust the size of the pool based on the usage pattern of an application.

We are also exploring the option of using the ThreadManagerAspect to monitor thread system(s) not only in the local virtual machine, but also in remote ones. Using these JMX services, it would be possible to effectively implement a distributed load balancing system as part of the ThreadManagerAspect.

3.2 MemoryManager Aspect

The MemoryManagerAspect structures a crosscutting concern associated with memory intensive operations, such as loading a large XML file into the memory. It essentially monitors the memory system and other related systems, such as the garbage collection system, of the Java virtual machine.

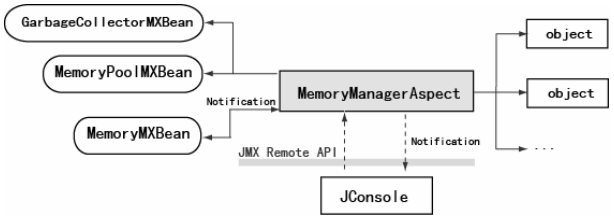


Fig 5. MemoryManagerAspect

As shown in Figure 5 (left-hand side), the MemoryManagerAspect collects information from three local platform MXBeans: the GarbageCollectorMXBean, the MemoryPoolMXBean, and the MemoryMXBean. As with the previous aspect, Figure 5 (right-hand side) also depicts that this aspect structures a concern that crosscuts objects in a given application, and can be manipulated through JConsole.

Garbage collection information, such as the total number of collections that have occurred, and the elapsed time of a collection, are retrieved from the GarbageCollectorMXBean. Since a Java virtual machine typically has more than one memory pool, the MemoryPoolMXBean is used to obtain detailed information of an individual pool, such as current and peak memory usage, usage threshold, and the collection usage threshold.

The MemoryMXBean provides basic memory usage information, such as initial size of memory and used/committed size of both the heap and non-heap memories. In addition to providing memory usage information for polling, the MemoryMXBean emits usage threshold exceeded notifications to its registered listener(s). These notifications are sent when the Java virtual machine detects that the memory usage of a memory pool exceeds its predefined threshold [3]. In order to receive these notifications, the MemoryManagerAspect implements the NotificationListener (Figure 6, line 3) interface and registers itself to the MemoryMXBean upon instantiation (Figure 6, line 6).

The MemoryManagerAspect differs from the ThreadManagerAspect in that it simply collects all the information mentioned above and sends this information out as notifications. The strategy of sending notification is exposed as a manageable attribute. The aspect can then be configured to send out notifications upon each invocation of any of its targets and/or when it receives usage threshold exceeded notification from the MemoryMXBean.

3.2.1 Implementation Details

```
1  public class MemoryManagerAspect
2      extends NotificationBroadcasterSupport
3      implements MemoryManagerAspectMBean, NotificationListener {
4
5    public MemoryManagerAspect(final CrossCuttingInfo info) {
6      ((NotificationEmitter) memoryMB).addNotificationListener(this, null, null);
7    }
8
9    public void handleNotification(Notification notification, Object handback) {
10     if (notification.getType().equals(MemoryNotificationInfo.MEMORY_THRESHOLD_EXCEEDED) &&
11         getNotifyUponTargetInvocation()) {
12       sendNotification(...);
13     }
14   }
15 }
```

Fig 6: Code fragment of the MemoryManagerAspect.

Figure 6 shows a code fragment associated with our prototype implementation of the MemoryManagerAsepct. This aspect extends NotificationBroadcasterSupport (line 2) in order to send JMX notifications, and implements NotificationListener (line 3) in order to listen for the memory threshold exceeded notification from the MemoryMX-Bean. In the constructor, the MemoryManagerAspect adds itself to MemoryMX-Bean as a notification listener (line 6). The handleNotification method receives notification, checks its type and whether the notifyUponTargetInvocation is enabled (lines 10,11), and sends notifications to its own listeners (line 12). The following screen shots below show how this aspect can be managed through the JConsole interface.

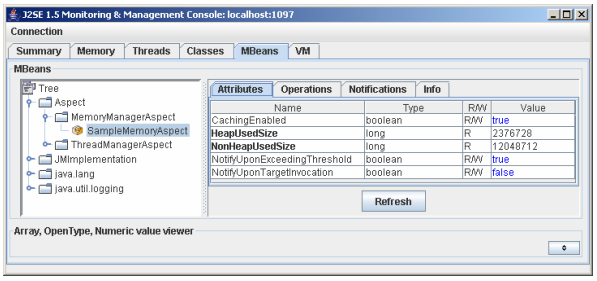


Fig 7. JConsole screenshot for the attributes of the MemoryManagerAspect.

Figure 7 shows all the management attributes exposed by the MemoryManagerAspectMBean interface. The HeapUsedSize and NonHeapUsedSize are read only, while the rest of the attributes are read/write. CachingEnabled is used to control the caching mechanism provided by the MemoryManagerAspect. The other two attributes indicate when notification should be sent out.

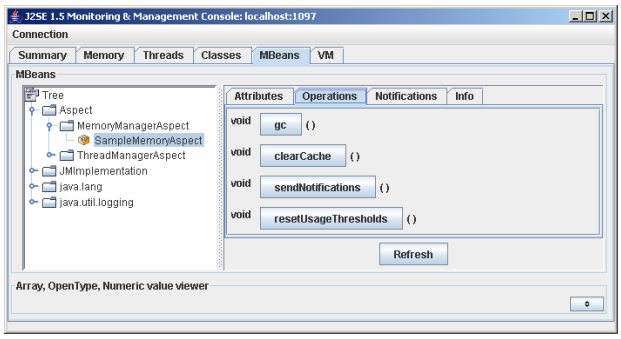


Fig 8. JConsole screenshot for the operations of the MemoryManagerAspect.

Figure 8 shows the management operations exposed by the MemoryManagerAspectMBean interface. The gc() operation is for garbage collection (it calls the MemoryMXBean's gc()). The clearCache() operation clears the cached objects. The sendNotifications() operation sends notifications containing the latest memory information. The resetUsageThreshold() resets the threshold values to default values initially set by JVM.

3.2.2 Beyond Proof-of-Concept

To collect additional information in the MemoryManagerAspect, we can add vendor-specific or platform-specific methods which are not part of standard MXBean interface. For example, to collect information about the last garbage collection operation, we can use a method called getLastGcInfo(), which is available in the sun.management.GarbageCollectorImpl class. This class is Sun's implementation of the GarbageCollectorMXBean interface. Though the disadvantage of incorporating this type of information is the poor portability and reusability that results, the advantage is the ways in which these customized approaches can further optimize and tune application-specific needs.

4. Future Work and Conclusions

The current prototype has shown several encouraging proof-of-concept results with respect to dynamic adaptation based on application profiling. Ongoing work is focused on exploring additional management capabilities, and assessing this approach within new environments and coupled with further AOP structural support.

4.1 Containers

One of the key things we plan to investigate in terms of additional management capabilities is the idea of a manageable AspectContainer, overviewed in Figure 9. This use of customized containers can aid management of instantiation concerns associated with AspectWerkz. Due to the fact that the AspectWerkz implementation uses reflection to instantiate aspects at runtime, every aspect in AspectWerkz has to provide either no constructor at all or one of the two pre-defined constructors.

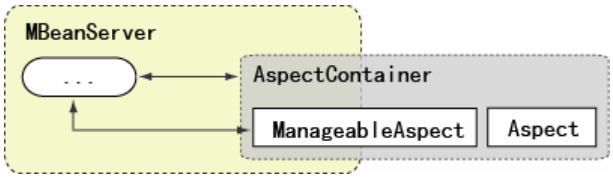


Fig 9. AspectContainer

The AspectContainer provides centralized management and control of all contained aspects, such as: aspect lifecycle management, configuration and deployment. It registers itself to the MBeanServer as an MBean in order to expose some generic manageability, while the manageability exposed by individual manageable aspects remains aspect-specific. This two-level approach can achieve much higher degree of flexibility and manageability. For example, the ThreadManagerAspect and MemoryManagerAspect explicitly implement JMX interfaces and register themselves to the MBeanServer in their constructor. If an AspectContainer was used, such JMX related code could be moved out of the aspects and put into the container's implementation. Moreover, since the container can keep track of all the contained aspects, it can provide some information for statistics or reflection about itself and its contained aspects.

In addition to being manageable through JMX, the AspectContainer can also be implemented to fit into other container-centric frameworks, such as the Spring Framework [12] and PicoContainer [10], in order to make the contained aspects as framework specific components. Then they can have access to or be accessed by container services and other components (probably in other containers).

4.2 Environments and Additional Structure

In terms of future environments, we plan to investigate crosscutting concerns within an open-source J2EE server built on a JMX microkernel, and experiment with fine-grained load balancing aspects within cluster environments. In terms of language features, we plan to investigate available metadata (JSR-175) for additional management capabilities, utilize control flow information structured within aspects for path-specific resource provisioning, and experiment with dynamic aspects (aspects loaded/unloaded at runtime) for adaptation.

4.3 Conclusions

The feasibility study shows how MADAPT can combine MBeans/aspects to improve modularity of code responsible for dynamic adaptation. As opposed to having this code scattered and tangled across classes and components, the examples show how profiling techniques used by adaptation code can be localized, structured, and managed as aspects.

Acknowledgments
We would like to thank Andrew Warfield for insightful discussions and valuable feedback.

References

1. AppAssure, www.alignmentsoftware.com.
2. AspectWerkz, http://aspectwerkz.codehaus.org/index.html.
3. JMX, http://java.sun.com/products/JavaManagement.
4. M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem determination in large, dynamic, Internet services. Proc. International Conference on Dependable Systems and Networks (IPDS Track), pages 595-604, June 2002.
5. G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. Proc. 3rd Symposium on Operating Systems Design and Implementation (OSDI'99), pages 187-200, Feb. 1999.
6. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, Aspect-Oriented Programming, European Conference on Object-Oriented Programming (ECOOP), 1997.
7. Magpie, http://research.microsoft.com/projects/magpie/.
8. D. Narayanan, J. Flinn, and M. Satyanarayanan. Using history to improve mobile application adaptation. Proc. 3rd IEEE Workshop on Mobile Computing Systems and Applications, pages 31-40, Dec. 2000.
9. PerformaSure, www.sitraka.com/software/performasure.
10. PicoContainer, http://www.picocontainer.org/
11. M. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. Proc. 6th Workshop on Hot Topics in Operating Systems (HotOS-VI), pages 124-129, May 1997.
12. Spring Framework, http://www.springframework.org/Baldonado, M., Chang, C.-C.K., Gravano, L., Paepcke, A.: The Stanford Digital Library Metadata Architecture. Int. J. Digit. Libr. 1 (1997) 108–121

A Biologist's View of Software Evolution

DeLesley Hutchins

CISA, University of Edinburgh
d.s.hutchins@sms.ed.ac.uk

Abstract. The term "software evolution" is generally used as an analogy for biological evolution. This paper explores that analogy in more depth, analyzing software evolution from the biologist's point of view. I give a basic introduction to fitness landscapes and genetic algorithms, and describe two major issues that effect evolvability: local optima in the search space, and genotype to phenotype maps. Aspects and meta-programs address both of these issues. Encapsulation can be seen as a technique for reducing epistasis and pleiotropy, while meta-programming can be seen as an analog to morphogenesis.

1 Introduction

The term "software evolution" is usually used as an analogy for biological evolution. Biological species gradually change over the course of tens of millions of years, adapting in response to changing environmental conditions. A piece of software likewise changes over time, in response to changing requirements and needs.

What distinguishes "software evolution" from other forms of software development is that each version of an evolving program must be valid. It must at least compile and run, even if it is not entirely bug-free. Some software development teams enforce this discipline with weekly or even daily builds. In an truly evolutionary development process, such as that advocated by Extreme Programming [3], it is not possible to do major code rewrites. Updates must be kept small, and each update must change a working program into another working program.

This, too, is analogous to biological evolution. Each organism in the real world must produce viable offspring. These offspring are slightly different from their parents, but they must be capable of surviving on their own. Major change occurs only after thousands of generations.

Is this just an analogy, or is there something deeper? The purpose of this paper is to look at the problem from a biologist's point of view.

Section 2 introduces genetic algorithms and fitness landscapes, which are mathematical models that can be used to understand evolutionary processes, and applies these models to software. Section 3 discusses how *epistasis* and *pleiotropy* affect evolvability, and discusses how these ideas can be interpreted in the context of software engineering. Section 4 discusses how *morphogensis*, which is the process of biological growth and development, is related to meta-programming. As it turns out, there is strong evidence which supports the idea

that aspects [12] [11] and meta-programs are, indeed, important tools for software evolution. Section 5 concludes with a few brief thoughts about how these ideas can be used to improve programming languages.

2 Genetic and Evolutionary Algorithms

One problem with applying biological models to software is that our current understanding of the genome is extremely limited. Although the DNA sequences for several species are now available, there is no "road map" of gene function. The protein folding problem — the problem of predicting the 3-dimensional shape of a protein molecule, given the DNA sequence which encodes that protein — is computationally intractable. Understanding the effect of a given protein on organism development and function is even harder. In fact, we do not even know which portions of the genome encode useful information.

Moreover, we can only extract DNA from living creatures. The genomes of past organisms are unavailable, which makes it difficult to study how they have changed over time.

Evolutionary algorithms are computational models which abstract away from the biochemical details of DNA. [5] [6] [14] EAs treat evolution as a form of *search* through a space of all possible solutions.

2.1 Hillclimbing

The simplest form of search, known as "hillclimbing", is essentially a form of gradient ascent. The algorithm is defined as follows:

Let $\mathbb{S}$ be the set of all possible **solutions** to a problem. These solutions can be numbers, vectors, graphs or even programs; the definition of what constitutes a "solution" depends on the problem being solved.

Let $\mathbb{C}$ be the set of **chromosomes**. Each solution must be represented as a data structure of type $\mathbb{C}$, which encodes the solution in a form that can be manipulated. Early work on evolutionary algorithms ignored the difference between a solution and the encoding of that solution, but this distinction turns out to be critical for understanding the evolutionary process. In biology, the elements of $\mathbb{C}$ are strands of DNA, while the elements of $\mathbb{S}$ are the actual organisms — two very different things.

Let $f : \mathbb{S} \to \mathbb{R}$ be a **fitness function**, which computes a numeric value $f(s)$ for every solution $s \in \mathbb{S}$. This value, called the *fitness* of the solution, is a measure of "how good" the solution is. If $f(s_1) > f(s_2)$, then s_1 is a better solution than s_2.

Let $g : \mathbb{C} \to \mathbb{S}$ be a **representation function**, which interprets a chromosome c as a solution. This function is usually onto, which guarantees that every solution has an encoding. If it is also one-to-one, then every solution has a unique encoding, but this is not required.

Let $m : \mathbb{C} \to \mathcal{P}(\mathbb{C})$, where $\mathcal{P}$ denotes the power set, be a **mutation operator**, which is used to generate new chromosomes from existing ones. This

operator defines the set of all possible single-point mutations for a given chromosome. Mutations must be reversible: $\forall c.\ c \in m(c')$ implies $c' \in m(c)$. In other words, if one mutation changes a chromosome, there must be another mutation which restores the chromosome to its original state.

The algorithm itself starts with an initial chromosome c_0, which is chosen randomly from $\mathbb{C}$. The next chromosome, c_{n+1}, is chosen randomly from $m(c_n)$, such that $f(r(c_{n+1})) \geq f(r(c_n))$.

This definition generates a sequence of chromosomes, where each new chromosome in the sequence represents a better solution than the previous one. The actual sequence is random, but the limit will be some local optimum in the search space. The term "hillclimbing" is a metaphor; it conjures the image of a blind hiker, who climbs to the top of a mountain, step by step, by always walking uphill.

2.2 Modeling the Human Programmer

Hillclimbing offers a reasonably good model of a single human programmer. The run-time behavior of a program, measured over all possible inputs, is a solution. The program requirements constitute a fitness function. If the run-time behavior of a program matches the requirements, then the program has good fitness.

The chromosome in this case is the source code of the program, which must be evaluated at run-time to yield a behavior. The interpreter and/or compiler thus constitute a representation function. There are many different programs which all yield the same behavior, so the representation function in this case is clearly not one-to-one. Moreover, the choice of programming language is important; solutions which are easy to express in one language may be hard to express in another.

The mutation operator can be defined as follows: for a given program p, $m(p)$ is the set of all programs that can be created by the insertion, deletion, or modification of a single node in the abstract syntax tree (AST) of p. Although programmers actually work with programs in the form of ASCII text, we wish to restrict our model to the set of syntactically valid programs.

Even this definition is problematic, since a single change to the AST will generally result in a program that is either not well-typed, or has a behavior that is much worse than the original. Most improvements to real-world programs involve many small changes to the AST, even if the programmer is developing the code incrementally.

Human beings must be regarded as "intelligent mutators", who have some knowledge of the local shape of the search space, and can plan out a sequence of mutations that lead to a better program. Nevertheless, a programmer's knowledge is limited. He or she may be able to plan out a short sequence of mutations, but only up to a certain point; the larger the change to the source code, the more difficult it becomes to predict the effect of that change.

The process of programming is still evolutionary so long as the number of mutations made during a single edit-run-debug cycle is small when compared to

the total number of mutations needed to get from the initial prototype to the finished product.

2.3 Genetic Algorithms

Most research on evolution in computer science has focused on genetic algorithms (GAs). GAs extend the basic hill-climbing model by introducing a population of chromosomes, along with a crossover operator.

Let $\otimes : \mathbb{C} \times \mathbb{C} \rightarrow \mathcal{P}(\mathbb{C})$ be a crossover operator. This operator generates the set of all possible chromosomes which can be constructed by drawing some genes from one parent, and the rest from the other parent. Intuitively, crossover mixes and matches different combinations of genes from two sources.

The algorithm starts with a population of chromosomes $c_1..c_n$, which are randomly selected from $\mathbb{C}$. It proceeds as follows:

- Calculate the fitness $f(g(c_i))$ of each chromosome c_i in the population.
- Remove the worst chromosome from the population.
- Select two of the better chromosomes – call them c_x and c_y. These are chosen at random, but the choice is weighted by fitness, so better solutions are more likely to be chosen.
- Choose a new chromosome c_z at random from $(c_x \otimes c_y)$.
- Do one of the following:
  (*a*) Add c_z to the population.
  (*b*) Choose a new chromosome at random from $m(c_z)$, and add it to population. The probability of choosing this option is small, and is controlled by the mutation rate.
- Repeat.

There is a difference of opinion between the biological literature and the computer science literature on the role that crossover plays in evolution. When genetic algorithms are used in computer science, crossover is the dominant search operator. The initial population of chromosomes is distributed throughout the search space, and crossover forces the population to converge to a single solution. Once the population has converged, the algorithm terminates.

Biological populations are quite different, because every species has already converged. Although there are differences between individuals, these differences are relatively small. Biologists thus treat mutation as the dominant operator. The assumption is that most mutations are bad, and bad mutations die out quickly. Good mutations are rare enough that each one has time to propagate through the gene pool before the next good mutation appears. Under this assumption, the population remains converged, and evolution with crossover can be treated as a form of hillclimbing.

I adopt the biological interpretation here. A GA with a population of size n corresponds to a development team of n people, who are all working on the same application. The application may have multiple branches of development, which need to be merged together. The crossover operator serves the same role

as CVS, or other version-control systems; it is responsible for merging code from one branch with code from another.

There is one major difference between crossover and CVS. In a software development environment, each change to the source code has a time stamp. CVS assumes that recent modifications are better, so in most cases it knows which changes to merge. In biological evolution, this assumption is not valid. Instead, crossover generates many different combinations of genes at random, and then tests each one via natural selection. After many such trials, only the combinations which contain the "good genes" from both parents will survive.

Although the mechanisms are different, the end result is the same. Whenever a beneficial change is made to one chromosome, that change is incorporated into the gene pool, where it can be improved by further mutations.

2.4 Fitness Landscapes

The mutation operator defines a distance metric on the search space. Since mutations are reversible, the set of chromosomes can be treated as an undirected graph, where two chromosomes c_1 and c_2 are connected by an edge if $c_1 \in m(c_2)$. The distance between two chromosomes is the shortest path between them — the minimum number of mutations needed to get from one to the other.

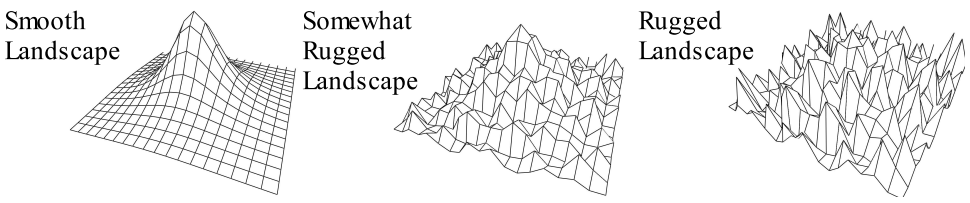
The combination of distance metric + fitness function means that genetic search spaces can be regarded as a *fitness landscapes*. [10] [9] The shape of the landscape determines how easy it is to search. "Smooth" landscapes are easy to search, whereas "rugged" landscapes are more difficult. There are two main factors which influence shape.

The first factor is called *fitness distance correlation*. [8] In a smooth landscape, the difference in fitness between two chromosomes is correlated with the distance between them; small changes to the chromosome result in small changes in fitness. In a rugged landscape, this is not true.

The second factor is the number and size of local optima. A local optimum is a chromosome c, which does not represent the best solution available, but for which all mutations result in a worse solution. The fundamental limitation of hillclimbing is that the search can get stuck in local optima. The metaphor here is that of a hiker who reaches the top of a small foothill. She can no longer proceed by going uphill; instead she must go down, off the hill, before she can climb the larger mountain.

Rugged landscapes have more local optima than smooth ones. Rugged landscapes also have larger local optima. The size of a local optimum is measured by its basin of attraction: the number of chromosomes c for which a hill-climbing search, starting from c, is guaranteed to end in the local optimum.

The following figure illustrates difference between smooth and rugged landscapes:



3 Dealing with Local Optima

A software engineer encounters a local optimum whenever a small change to one part of the source code is not sufficient to create a working program. If the source code must be modified in several places simultaneously to keep it from breaking, then each such change corresponds to a "downhill step" (i.e. a change which does not improve fitness), which is necessary to get out of the local optimum. Human programmers are intelligent enough to make such simultaneous changes, even rewriting large amounts of code if need be, but true evolutionary systems do not have that luxury.

The size of the local optimim is indicated by the amount of code that must be rewritten. Most software engineers are familiar with the idea that it is sometimes necessary to make major changes to the underlying architecture of an application in order to implement a new feature. In severe cases, the architecture changes so much that the application must be rewritten almost from scratch; such situations are evidence of large local optima.

3.1 Pleiotropy, Epistasis, and Traits

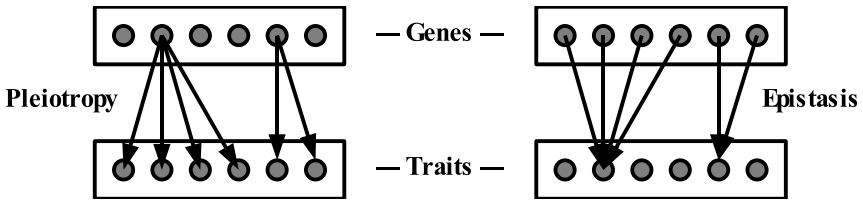
One of the principle aims of evolutionary theory is to understand not only how the shape of the fitness landscape makes evolution easy or difficult, but what causes landscapes to have a particular shape. In biological jargon, the principles of *pleiotropy* and *epistasis* are two of the main factors which influence how rugged the landscape is going to be. [2] [17] These terms describe the mapping between genes and *traits*.

It is somewhat difficult to come up with a rigorous definition of traits. A trait is some property of a solution which has a direct correlation with fitness. In other words, solutions which have the trait perform consistently better than solutions which do not have the trait. For example, thick fur and blubber are survival traits for animals which live in the arctic, while broad leaves are a survival trait for trees in tropical rainforests.

Traits are partial solutions to software requirements. Requirements are generally specified as a list of features or capabilities that a program should have. A trait is any aspect of program behavior which implements all or part of one of those features. (Traits are difficult to define formally because the definition assumes that total solutions can be broken down into partial solutions. Nevertheless, the intuition behind traits should be clear.)

Genes are easier to define. A chromosome is divided into a set of genes, where each gene is the smallest section of the chromosome which can be interpreted as a meaningful unit. In biology, a gene is a sequence of DNA base pairs which codes for a single protein. In software, a gene would be a single line of code.

A gene is *pleiotropic* if it influences multiple unrelated traits in an organism. A group of genes are *epistatic* if they all interact in some way to control a single trait in an organism.



In the case of pleiotropy, the gene is hard to modify because a change which improves one trait will likely impair another. In the case of epistasis, a change to one gene will influence the effects of all the others, with unpredictable results. The whole set of genes typically has to be changed at the same time, which cannot be done in gradual steps.

Both epistasis and pleiotropy make the landscape more rugged because a single change to one gene is much more likely to disrupt the effects of others, thus causing a reduction in overall fitness. If all changes to the gene are disruptive then the result is a local optimum. If only some changes are disruptive then the result is a discontinuity that reduces the fitness distance correlation.

3.2 Modularity and Encapsulation

Pleiotropy crops up in software as the "backwards compatibility problem". Library routines are difficult to modify because they are called from many different parts of the code. Any change to the observable behavior of the routine will thus have far-reaching effects on many traits throughout the program.

Epistasis crops up as the "parallel maintenance problem". When two sections of the code are tightly coupled, a change to one section almost always requires a change to the other. For example, the visitor design pattern is often used to traverse a data structure which is described by several different classes. [4] Every visitor must implement a method for each class. When a new class is added to the data type, a new method must be added to every visitor — a classic case of close coupling.

In a nutshell, this is a biologist's argument for modularity and encapsulation. If the implementation of a feature is properly encapsulated behind a high-level interface, then it is much easier to change that implementation without affecting code that depends on it. This reduces pleiotropy. Similarly, if two pieces of code can be decoupled by placing them into separate modules which communicate through a well-defined interface, then the modules can be updated independently, thus reducing epistasis.

4 Genotype to Phenotype Maps

The definition of an evolutionary algorithm given above involves two functions – a fitness function, and a representation function. Early work with GAs ignored the representation function; fitness was defined directly on chromosomes. It was assumed that the difficulty of searching for a solution was determined by the overall difficulty the problem being solved.

One of the major surprises of this early work was that representation mattered a great deal. It is possible to transform an easy problem into a hard one, or vice versa, by altering the way in which solutions are encoded by chromosomes. [16] For example, an integer can be encoded using either standard binary, or gray codes. A gray code is an encoding in which adjacent integers are a single bit flip apart, which seems to improve hill-climbing performance on some problems.

A more sophisticated example arises in genetic programming. Genetic programming is the application of genetic algorithms to actual computer programs, thus providing a true example of "software evolution". It is possible to encode a program as a linear string of instructions, which is evaluated by a virtual machine. However, such encodings tends to be very "brittle" and difficult to evolve, because the effect of each instruction depends heavily on the instructions that precede it. A better mechanism is to represent programs directly as abstract syntax trees, and modify the mutation and crossover operators so that they operate on trees rather than strings. [13] This dramatically improves performance.

Using an explicit representation function highlights the difference between the *genotype* (the chromosome) and the *phenotype* (the actual solution). [1] [17] The mutation and crossover operators operate on the genotype — e.g. source code or DNA. The fitness function, however, evaluates the phenotype — e.g. the run-time behavior of a program, or the physical body of an organism.

In contrast to most popular descriptions of evolution, natural selection does not select directly for genes. Instead, it selects for traits, which are properties of the phenotype. If there is a *natural* mapping between genotype and phenotype, then each gene (or small group of genes) in the genotype will correspond to a single trait in the phenotype. In this case, natural selection for traits translates directly to selection on the appropriate genes. However, many representation functions do not create a natural mapping.

A natural mapping is not the same as a *direct* mapping, where each gene maps to one specific part of the phenotype. For example, thick fur is an adaptive trait for animals in the arctic. However, there is not a different gene for every hair on the body. Instead, a few genes control the length of all hairs. It is the overall thickness of the fur, not the length of an individual hair, that is the adaptive trait. The same is true of plants. A fern has many branches, but the angle of each branch is the same. It is the distribution of leaves, not the angle of a particular frond, that constitutes a trait.

4.1 Morphogenesis and Meta-Programming

Multicellular organisms create a natural mapping between genotype and phenotype by means of *morphogenesis*. Morphogenesis is the process of growth and development, whereby a single cell repeatedly divides and differentiates to build an entire organism. While the exact mechanisms of morphogenesis are not well understood, it is clear that the process is algorithmic. The evidence lies in the recursive fractal patterns found in almost all living things, from ferns to blood vessels to spiral shells. [15]

We do know that a significant percentage of genes are active only during development. Developmental genes create chemical gradients, and switch on and off in response to signals produced by other genes, thus producing the complex structures found in living organisms. Because each cell in the body has a complete copy of the DNA, a single gene can express itself in many different physical locations.

Aspects and meta-programs serve the same role in software evolution that morphogenesis plays in biological evolution — they help to establish a natural map between genotype and phenotype. The clear lesson from evolutionary theory is that controlling the genotype to phenotype map is the key to evolvability.

A language like C has a fairly direct mapping between source code and machine code; every function or statement can be translated almost directly to (unoptimized) assembly. Since the interpretation of machine code is fixed by the CPU architecture, this means that the genotype to phenotype map is also fixed.

Aspects and meta-programs introduce a more sophisticated genotype to phenotype map. A meta-program algorithmically generates an implementation that is quite different from the source code. This is ideal for situations such as parser generators and DSLs, where a great deal of repetitive code needs to be produced, but where the repetition cannot be encapsulated into simple loops or functions. Aspects are similar. An aspect can weave advice (such as logging code) throughout an application, thus algorithmically generating an implementation.

Work on evolving neural-networks suggests that generating solutions algorithmically does, in fact, lead to more modular and evolvable designs. [7]

5 Conclusion

The word "evolution" in software evolution is more than just an analogy. The set of program requirements constitute a fitness function, and the space of all possible programs constitute a fitness landscape. The act of developing and maintaining a piece of software plots a path across this landscape. Software evolution can be seen as a form of human-guided search for a program that meets the specified requirements.

Encapsulation and modularity are basic techniques which improve software evolvability by "smoothing" the fitness landscape, and reducing the number of local optima. Backwards compatibility problems, parallel maintenance problems, and major code rewrites are symptoms of local optima. All forms of modularity, whether they be functions, classes, or aspects, are useful in this regard.

Aspects and meta-programs go beyond simple modularity, however, because they alter the genotype to phenotype map. By introducing a different mapping, it is possible to completely transform a rugged landscape into a smooth one.

The most sophisticated examples of such mappings are *domain-specific languages*. DSLs abstract away from the implementation language entirely; there is little similarity between the source code (the genotype), and the executable machine code (the phenotype). In a well-designed DSL, terms in the DSL map directly to concepts in the problem domain. Program requirements (the fitness function) are also domain-specific, which means that there is a natural mapping between terms in the DSL, and fitness-correlated traits.

The main weakness of current DSLs is that the translation from the DSL to the implementation language is fixed. Program requirements change over time, and this means that the DSL itself may have to change too. The clear lesson for software engineers is that we need meta-programming tools which allow DSLs to be easily constructed and modified. Good integration between the DSL and the implementation language remains a challenge.

References

1. Lee Altenberg. Genome growth and the evolution of the genotype-phenotype map. Evolution and Biocomputation: Computational Models of Evolution, 1995.
2. Lee Altenberg. Nk fitness landscapes. T. Back, D. Fogel, Z Michalewicz. editors. Handbook of Evolutionary Computation, Section B2.7.2, 1997.
3. Kent Beck and Cynthia Andres. *Extreme Programming Explained*. Addison-Wesley, 2004.
4. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
5. David E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Reading: Addison-Wesley, 1989.
6. David E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publishers, Boston, MA, USA, 2002.
7. Frederic Gruau. Genetic synthesis of modular neural networks. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 318–325, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.
8. Terry Jones and Stephanie Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. Proceedings of the 6th International Conference on Genetic Algorithms, pages 184–192, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.
9. Stuart A. Kauffman. Adaptation on rugged fitness landscapes. Lectures in the Sciences of Complexity. Addison-Wesley, 1989.
10. Stuart A. Kauffman and S. Levin. Towards a general theory of adaptive walks on rugged landscapes. Journal of Theoretical Biology 128: 11-45., 1987.
11. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. Proceedings of ECOOP, 2001.
12. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. volume 1241 of *Proceedings European Conference on Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

13. John R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.
14. Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, USA, 1996.
15. P. Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.
16. Andrew Tuson. *No Optimization Without Representation: A Knowledge Based Systems View of Evolutionary/Neighborhood Search Optimization*. Ph.D. Thesis, University of Edinburgh, 1999.
17. Gunter P. Wagner and Lee Altenberg. Complex adaptations and the evolution of evolvability. Evolution 50 (3): 967-976, 1996.