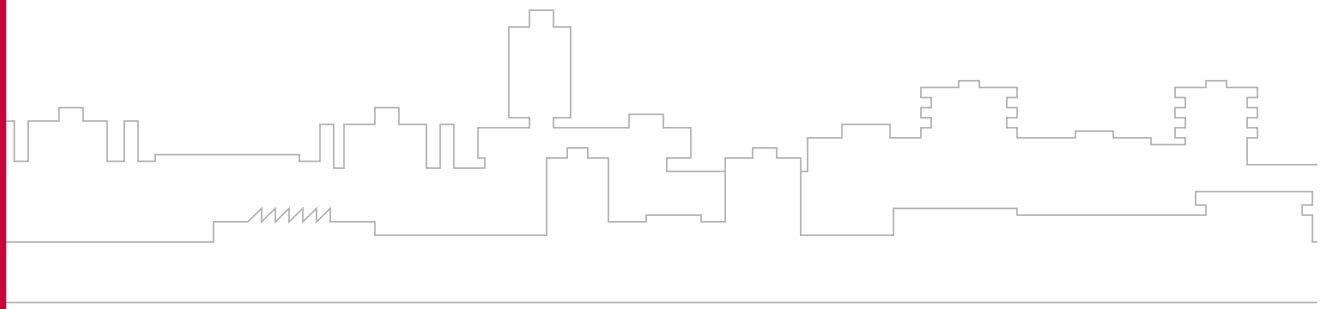
Pedro J. Clemente
Miguel A. Pérez
Sergio Lujan
Hans Reiser

13th Workshop for Phd Students in Object Oriented Programming:

Summary and Accepted Papers



Technical Report TR-I4-03-03
2003-10-06

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany

Informatik 4 (Distributed Systems and Operating Systems)
Prof. Dr. Wolfgang Schröder-Preikschat

13th Workshop for Phd Students in Object Oriented Programming

The workshop summary will be published in:
Frank Buschmann, Alejandro Buchmann (eds): ECOOP 2003 workshop reader. LNCS, Sprinter, 2003

Table of contents

Summary of 13th Workshop for Phd Students in Object Oriented Programming

Pedro J. Clemente[1], Miguel A. Pérez[1], Sergio Lujan[2] and Hans Reiser[3]

Department of Computer Science. University of Extremadura, Spain*
[2] Department of Languages and Computer Science. University of Alicante, Spain.
[3] Department of Distributed Systems and Operating Systems. University of Erlangen-Nürnberg, Germany.

{jclemente, toledano}@unex.es, sergio.lujan@ua.es, reiser@cs.fau.de

Abstract. The objective of the 13th edition of Ph Doctoral Students in Object-Oriented Systems workshop (PHDOOS) was to offer an opportunity for PhD students to meet and share their research experiences, and to discover commonalities in research and student ship. In this way, the participants may receive insightful comment about their research, learn about related work and initiate future research collaborations. So, PHDOOS is a gauge to detect hot spots in current lines of research and new avenues of work in objects-oriented concepts.

1 Introduction

At its 13th edition, the PhDOOS workshop established the annual meeting of PhD students in object-orientation. The main objective of the workshop is to offer an opportunity for PhD students to meet and share their research experiences, to discover commonalities in research and studentship, and to foster a collaborative environment for joint problem solving.

The workshop also aims at strengthening the International Network of PhD Students in Object-Oriented Systems[1] initiated during the 1st edition of this workshop series at the European Conference on Object Oriented Programming (ECOOP), held in Geneva, in 1991. This network has counts approximately 120 members from all over the world. There is a mailing list and a WWW site, used mainly for information and discussion about OO-related topics. Since its foundation, the International Network of PhD Students has proposed and created a workshop for PhD students in association with ECOOP each year. This 13th edition makes PHDOOS a classical workshop in ECOOP.

At this edition, the workshop was divided into plenary sessions, discussion session and a conference. The sessions were determined according to the research interests of the participants. Potential topics of the workshop were those of the main ECOOP conference, i.e. all topics related to object technology including but not restricted to: analysis and design methods, real-time, parallel systems,

* The organization of this workshop has been partially financed by CICYT, project number TIC02-04309-C02-01

patterns, distributed and mobile object systems, aspects oriented programming, frameworks, software architectures, software components, reflection, adaptability, reusability and theoretical foundations. Due to the heterogeneous nature of the topic of the papers received, the workshop conclusions are focused on the interesting research areas and on solving common problems.

The participants had a 20 minute presentation at the workshop (including questions and discussions). The discussion group was based on the research interests of the participants. Finally, the conference featured a speaker who is invited to talk about interesting research, personal experiences or research methodology. This conference was a unique opportunity to hear or ask things not discussed elsewhere, and to have an "unplugged" discussion with a well-known personality from our field. This year, the speakers was the professor Robert E. Filman

This paper is organized into the following points. The next section is focused on the participants of the workshop. In section three, the presented papers are explained and main topics are summarized. The fourth section talks about a conferences, and the fifth section includes the final discussion. Finally the paper concludes with workshop conclusions and bibliographic references.

2 PHDOOS workshop participants.

In previous editions [2,3,4,5] of this workshop there have been more or less twenty participants working for 2 days. However, this year the coincidence with a Doctoral Symposium cut the number of participants with papers to 10, thereby reducing the length of this workshop to 1 day. We can divide the participants into four groups:

- Participants with papers
- Organizing Committee.
- Program Committee
- Invited speaker.

2.1 Participants with papers

The number of papers received was 11, and 9 were accepted. The attendants with accepted papers were the following:

- M. Devi Prasad. Manipal Center for Information Science, Manipal Academy of Higher Education
- Hervé Paulino. Department of Computer Science. Faculty of Sciences and Technology. New University of Lisbon.
- Sari R. ElDadah, Nidal Al-Said. Department of Information Systems. Arab Academy For Banking and Financial Sciences.
- Ademar Aguiar. Faculdade de Engenharia da Universidade do Porto.
- Balázs Ugron. Department of General Computer Science. Eötvös Loránd University, Budapest.

– Amparo Navasa Martínez. Quercus Software Engineering Group. University of Extremadura.
– Susanne Jucknath. Institute of Software Engineering and Theoretical Computer Science Technical University of Berlin.
– Jan Wloka. Fraunhofer FIRST.
– Szabolcs Hajdara. Department of General Computer Science. Eötvös Loránd University, Budapest.
– Andreas I. Schmied. Distributed Systems Laboratory. University of Ulm

2.2 Organizing Committee

The Organizing Committee of Ph Doctoral Object-Oriented Systems is made up of volunteers participants in the previous workshop. Organizers in earlier editions advise these volunteers. This year, the organizers were: Sergio Luján Mora, Sérgio Soares, Hans Reiser, Pedro José Clemente Martín and Miguel Angel Pérez Toledano.

Sergio Luján-Mora is a Lecturer at the Computer Science School at the University of Alicante, Spain. He received a BS in 1997 and a Master in Computer Science in 1998 from the University of Alicante. Currently he is a doctoral student in the Dept. of Language and Information Systems being advised by Dr. Juan Trujillo. His research spans the fields of Multidimensional Databases, Data Warehouses, OLAP techniques, Database Conceptual Modeling and Object Oriented Design and Methodologies, Web Engineering and Web programming.

Sérgio Soares is currently a Ph.D. student at the Computer Science Center of the Federal University of Pernambuco. His current research interests include implementation methods and aspect-oriented programming. He is also a Assistant Professor at the Statistics and Computer Science Department of the Catholic University of Pernambuco

Hans Reiser studied Computer Science at the University of Erlangen-Nürnberg, obtained Dipl. Inf. degree in computer science in spring of 2001. Since 6/2001 he has been employed as researcher at the Department of Distributed Systems and Operating Systems at University of Erlangen-Nürnberg. He is member of the AspectIX research team (a joint group of our department and distributed systems department of University of Ulm), doing research on adaptive middleware for large-scale distributed systems. The primary research focus for PhD is software based fault tolerance in distributed systems.

Pedro José Clemente Martín graduated with a Dipl. Inf. in Computer Sciences from the University of Extremadura (Spain) in 1998.He is a Lecturer at the Computer Science Department at the University of Extremadura, Spain and a member of the Quercus Software Engineering Group. His current research interests are Aspect Oriented Programming and Component based Software Engineering. His PhD is focused on the interconnection of components to build software systems using aspect oriented programming.

Miguel Angel Pérez Toledano graduated with a Dipl. Inf. in Computer Sciences from the Polithecnic University of Cataluña (Spain) in 1993. Now, he is

working at the University of Extremadura as Associated Professor. He participated in the organization of the PhD workshop of ECOOP'02, and is a member of the Quercus Software Engineering Group. His current research interests are Semantic Description of Components and Repositories of Software Components. His PhD is focused on the selection and retrieval of components from repositories using semantic descriptions.

2.3 Program Committee

The program committee was composed of senior researchers with a strong background in some object-oriented topic. The review process is designed to ensure that every participant is able to present some relevant and well prepared material. This edition, the program committee was composed by:

- Marcelo Faro do Amaral Lemos (Federal University of Pernambuco, Brazil)
- Márcio Lopes Cornelio (Federal University of Pernambuco, Brazil)
- Juan C. Trujillo (University of Alicante, Spain)
- Fernando Sánchez Figueroa (University of Extremadura, Spain)
- Juan Manuel Murillo Rodríguez (University of Extremadura, Spain)
- Rüdiger Kapitza (University of Erlangen-Nürnberg, Germany)
- Andreas Schmied (Univiversity of Ulm, Germany)
- José Samos (University of Granada, Spain)

2.4 Invited speaker.

For this edition, the invited speaker to PhDOOS was professor Robert E. Filman. He is working at the Research Institute for Advanced Computer Science (RIACS), NASA Ames Center. His work about Aspect Software Oriented Development is recognized by the international computer science research community.

3 Research Hot Points

One of the main workshop objectives is to detect the principal research lines following Object Oriented Programming and Programming Technologies. In this sense, the following topics have been presented and have been discussed:

- Aspect Oriented Software Development
- Documentation and Categorization of Software
- Agents Technologies

3.1 Aspect Oriented Software Development

Aspect Oriented Software Development has been the most important point discussed in this workshop due to the fact that sixty percent of the accepted papers dealt with this topic. In this sense, we can find several topics and focuses that

allow us to ensure that the area of Aspect Oriented Technologies is currently very important.

The subjects treated included extension and description of Aspect-Oriented Languages, Aspect Oriented Code Visualization, Composition of Aspects, and domain of Aspect-Oriented application.

About *Aspect-Oriented Languages* there are two approaches: to extend an existing language (for example, AspectJ) and to define new languages to describe and compose aspects (for example, using ADL).

Prassat [6] suggests that AspectJ does not treat type-casting as an important operation in programs execution. He demonstrates that Java programs use type casting primarily for retrieving references to new types from distantly related ones. He investigates AspectJs inadequacy in selecting execution points that use type-casting to yield new object or inheritance references. He demonstrates the necessity for a reference creation by type casting joinpoints and argues that its addition makes AspectJs existing model more expressive.

Amparo Navasa [7] claims that to extract the aspect code, crosscutting the functional one makes it easier to implement aspect oriented systems using AOP languages, but it is possible to take away the problem of AO from the implementation phase to the design. In this phase, crosscutting functional code concerns can be treated as independent entities. Her ideas are based on developing aspect oriented systems taking into account the benefits of applying CBSE at the early stages of AO systems development, particularly at architectural design. Concretely, AOSD at the design level as a co-ordination problem, using an ADL to formalize it. This means a new language to define aspects and compose them from the design time viewpoint.

Aspect Oriented Software Visualization presents a growing potential due to the fact that graphic tools do not currently exist to visualize the aspect code execution. In this sense, one intention of Software Visualization is to form a picture in the user's mind of what this software is about, what happens during the execution and what should happen from the programmer's point of view[8]. This is especially helpful with existing software and non-existing documentation. Therefore the amount of existing software is increasing the need to understand this software too.

From *Composition of Aspects* viewpoint, there is special interest in Distributed Systems, because they contain lots of cross-cutt and tangled code fragments to fulfill their services. Merging a formerly separated aspect code with a program code by means of aspect-oriented programming is enabled through a couple of available technologies that manipulate program sources. Unfortunately, these tools or aspect weavers both operate on a distinct coarse-grained level (types, methods) and fulfill only a restricted a-priori known set of manipulations.

However, to weave several aspect code fragments which could have been constructed by independent teams for more than one concern simultaneously a composition that not only concatenates aspects, but also manages join effects between them, reveals several complex, possibility interfering weaving demands[9].

The use of *Aspect Oriented* in specific domains or concrete areas in Software Engineering is other growing area. In this sense, Hajdara[10] presented a solution to apply Aspect Oriented technologies to handle different non-functional properties like synchronization specification of parallel systems.

Refactoring and Aspect Orientation (AO) are both concepts for decoupling, decomposition, and simplification of object-oriented code. Refactoring is meant to guide the improvement of existing designs. For this reason it is the main practice in eXtreme Programming to implement 'embrace change' in a safe and reliable way. Aspect orientation on the other hand offers a new powerful encapsulation concept especially for coping with so called crosscutting concerns. Although refactoring and AO have the same goals, their current forms impede each other. Since the development of modular systems has become more and more difficult a combined application of refactoring and AO is still a desirable goal and would be a great help for developers[11].

3.2 Documentation and Categorization of Software

Two papers were presented on this topic during the workshop. The first paper "A Process-based Framework for Automatic Categorization of Web Documents" from Sari R. ElDadah[12], presented the design of a Framework for the development of Automatic Text Categorization applications of Web Documents. The process, composed of 4 activities, (identifying significant categories, finding the best description for each category, classifying the documents into the identified categories and personalizing the categories and their relevant descriptions according to the user preference) is conducted from the various Automatic Text Categorization methods developed so far, and described based on Petri Nets process description language. The paper concluded with notes about the ATC process.

The second paper presented on this topic was titled "A Minimalist Approach to Framework Documentation" from Ademar Aguiar and Gabriel David[13]. This paper proposes a documenting approach for frameworks that aims to be simple and economical to adopt. It reuses existing documentation styles, techniques and tools and combines them in a way that follows the design principles of minimalist instruction theory. The minimalist approach proposes a documentation model, a documentation process, and a set of tools built to support the approach (Extensible Soft Doc).

3.3 Agents Technologies

In this subject, we can include the paper "Mob: a Scripting Language for Programming Web Agents" showed by Hervé Paulino[14]. Mob: a scripting language for programming mobile agents in distributed environments was presented. The semantics of the language is based on the DiTyCO (Distributed TYped Concurrent Objects) process calculus. Current frameworks that support mobil agents are mostly implemented by defining a set of Java classes that must then be extended to implement a given agent behavior. Mob is a simple scripting language

that allows the definition of mobile agents and their interaction, an approach similar to D'Agents. Mob is compiled into a process-calculus based kernel-language, and its semantics can be formally proved correct relative to the base calculus.

4 Workshop Conference

The workshop conference entitled *Aspect Oriented Software Development* was presented by Robert E, Filman.

Robert presented an interesting conference which has four parts:

- Object Infrastructure Frameworks (OIF)
- Aspect-Oriented Software Development
- AOP through Quantification over Events
- Research Remarks

The conference began with an introduction about Robert's latest research projects. He then introduced the concept of *Aspect Oriented Software Development,* and a way to obtain AOP through Quantification over Events. Finally, some remarks about research directions were presented.

Now, we are going to summarize each part of this presentation, because Robert's affirmations are very interesting, above all for students interested in Aspect-Oriented.

4.1 Object Infrastructure Framework (OIF)

Distributed Computing systems is difficult mainly for the following reasons:

- It is hard to archive systems with systematic properties (called *Ilities*) like *Reliability, Security, Quality of Service, or Scalability.*
- Distribution is complex for the following reasons: concurrence is complicated, distributed algorithmics are difficult to implement, every policy must be realized in every component, frameworks can be difficult to use, etc.

The introduction of a Component based Architecture require separating the component functionality and the non-functional properties. These non-functional properties should be inserted into components and allow for the interaction (communications) among components.

This idea has been implemented using Object Infrastructure Frameworks (OIF) [15]. OIF allows for the injection of behavior on the communications paths between components, using *injectors* because they are discrete, uniform objects, by object/methods and dynamically configurable. This idea permit the implementation of non-functional properties like *injectors*, and then they can be applied to the components; for example, injectors can encrypt and decrypt the communications among components.

OIF is an Aspect Oriented Programming mechanism due to the fact that:

- It allows separating concerns into injectors
- It wrapping technology
- It piggy-backs on distributed-object technology (CORBA)

4.2 Aspect-Oriented Software Development

How can we structure our programming languages do help us archive such ilities(Reliability, Security, Quality of Services, Evolvability, etc.)?

Separation of Concerns is an interesting strategy to structure our programming languages because a fundamental engineering principle is that of *separation of concerns.*

Separation of Concerns promises better maintainability, evolvability, Reusability and Adaptability. Concerns occur at both the User/requirements level and Design/implementation level[16].

Concerns cross-cut can be Applied to different modules in a variety of places, and must be composed to build running systems.

In conventional programming, the code for different concerns often becomes mixed together (tangled-code).

Aspect Oriented Programming modularize concerns that would otherwise be tangled. AOP provides mechanisms to weave together the separate concerns.

Implementation Mechanism The following division allows for the description of the common AOP implementation mechanisms used and the usual platforms used:

- *Wrapping technologies*: Composition filters, JAC
- *Frameworks*: Aspect-Moderator Framework
- *Compilation technologies:* AspectJ, HyperJ
- *Post-processing strategies*: JOIE, JMangler
- *Traversals*: DJ
- *Event-based*: EAOP
- *Meta-level strategies*: Bouraqadi et al., Sullivan, QSOUL/Logic Meta-Programming

4.3 AOP through Quantification over Events

A single concerns can be applied to many places in the code, but the we need to quantify it.

Concerns can be quantified over the static(lexical) form of the program, semantic (reflective) structure of the program structures and the events that happen in the dynamic execution of a system.

To take the expressiveness in quantification to its extreme is to be able to quantify over all the history of events in a program execution. The events are with respect to the abstract interpreter of a language. However, language definitions do not define their abstract interpreters.

As a consequence, we are able to describe interesting points in the program (lexical structure of the program, reflective structure of the classes and dynamic execution of the system), and then to describe the change in behavior desired at these points. The shadow of a description is the places in the code where the description might happen, for example, the event *invoking a subprogram* represents in a syntactic expression *subprogram calls.* It is necessary to define these events, capture these, and to change the behavior at this point. For more detail, please refer to [17].

4.4 Research remarks

This section of the conference presents the main research directions about *Aspect Oriented Software Development*, and this information should be useful for current and prospective Phd Students.

Research regime

- Define a language of events and actions on those events.
- Determine how each event is reflected (or can be made visible) in source code.
- Create a system to transform programs with respect to these events and actions.
- Developing an environment for experimenting with AOP languages (DSL for AOP)

Real AOP Value

- We don't have to define all these policies before building the system
- Developers of tools, services, and repositories can remain (almost) completely ignorant of these issues
- We can change the policies without reprogramming the system
- We can change the policies of a running system

Open Research Directions

Languages for doing AOP

- Hardly seen the end of this topic
- Join points
- Weaving mechanisms
- Handling conflicts among aspects

The software engineering of AOP systems

- Modeling aspects: From models to executable code
- Debugging aspects independently of the underlying system
- Tools for recognizing and factoring concerns

Applying AOP to particular tasks

- Monitoring/debugging
- Version control/provenance
- Web/system services
- User-centered computing
- Reliable systems
- System management

5 Summary of the discussion group

Although the workshop has a wide focus it turned out that most participants are working in research areas closely related to aspect oriented programming. Instead of having small subgroup discussions, the group opted for one plenary sessions discussing topics on AOP-related topics. In the process of selecting appropriate topics, we came up with the following main issues:

5.1 Shall aspects be supported as first class entities?

The situation today is that most AOP languages do not support aspects as first class entities. This is however due to the simple pragmatic way these languages are implemented. One may anticipate that the next generation of AOP languages will provide support for aspects as first class entities. The main benefits which we expect from such future developments are reusability of aspects, inheritance between aspects, and dynamically adaptive aspects. These areas still offer some research potential.

5.2 Does AOP make sense with non-OOP paradigms?

This issue was only briefly discussed. The group agreed that in paradigms like functional or imperative programming, separation of concerns is an equally required design method, and AOP is useful technique to support this. However, related to the generalization of aspects discussed later, non-OOP aspect orientation will have somewhat different requirements on potential join-point definition than in the case of OOP.

5.3 What are adequate techniques to understand AOP programs?

One major problem with AOP is that while it simplifies things on a rather abstract level, it gets more difficult to understand the concrete behavior of your program at a lower level. Current visualization techniques, already offered in some development environments, are not yet adequate for larger projects. The issues to be supported by visualization techniques are documentation, testing and debugging. The demand for such techniques will further rise, if aspect code and functional code shall be developed independently

5.4 What purposes shall aspects be used for?

One widespread use of aspects is to restructure existing software (refactoring to increase modularization), with the goal to improve structure and maintainability. However, we anticipate that in the future, AOP will also be applied for new applications, starting in the design phase. For this purpose, the availability of reusable "aspect components", addressed in the next item, will be essential. A different question is whether AOP techniques may and shall be used for modifying existing applications, that were developed without considering such later modification. However, we were rather reluctant to consider this as a good AOP technique.

5.5 Is is feasible to make aspects reusable?

Closely related to the previous topic is this important issue. In our opinion, the most problematic matter is the definition of join points. In current AOP languages, the definition of aspects is always tailored individually to one specific application. Even in such a specific case, existing AOP tools are usable best if the application and the aspects are written by the same developer. Also, even a small modification to one application easily makes aspects unusable. We all agree that this is a highly disappointing situation.

Having reusable aspects is highly desirable, but it requires further research on how this might be done. An important issue in this context is the question of whether aspects can be defined without them? limiting to an specific AOP language. Ultimately, AOP needs to be done already in the design phase of application development.

5.6 Conclusions

In spite of the fact that AOP has matured for over the years, several issues can be found that are still relevant for future research. The most important issue we found are the definition of join points targeting at reusability of aspects, and tool support for visualizing and understanding aspect oriented applications.

References

1. International Network for PhD Students in Object Oriented Systems (PhDOOS). http://www.ecoop.org/phdoos/ (1991)
2. 9th Workshop for Ph Doctoral Students in Objects Oriented Systems. http://www.comp.lancs.ac.uk/computing/users/marash/PhDOOS99/ (1999)
3. 10th Workshop for Ph Doctoral Students in Objects Oriented Systems. http://people.inf.elte.hu/phdws/ (2000)
4. 11th Workshop for Ph Doctoral Students in Objects Oriented Systems. http://www.st.informatik.tu-darmstadt.de/phdws/ (2001)
5. 12th Workshop for Ph Doctoral Students in Objects Oriented Systems. http://www.softlab.ece.ntua.gr/facilities/public/AD/phdoos02/ (2002)
6. Prasad, M.D.: Typecasting as a new join point in AspectJ. 13th Workshop for Phd Students in Object Oriented Programming at ECOOP. Darmstadt, Germany. (2003)
7. A.Navasa, M.A.Pérez, J.M.: Using an adl to design aspect oriented systems. 13th Workshop for Phd Students in Object Oriented Programming at ECOOP. Darmstadt, Germany. (2003)
8. Jucknath, S.: Software visualization and aspect-oriented software development. 13th Workshop for Phd Students in Object Oriented Programming at ECOOP. Darmstadt, Germany. (2003)
9. Andreas I. Schmied, F.J.H.: Composing non-orthogonal aspects. 13th Workshop for Phd Students in Object Oriented Programming at ECOOP. Darmstadt, Germany. (2003)

10. Hajdara, S.: An example of generating the synchronization code of a system composed by many similar objects. 13th Workshop for Phd Students in Object Oriented Programming at ECOOP. Darmstadt, Germany. (2003)
11. Wloka, J.: Refactoring in the presence of aspects. 13th Workshop for Phd Students in Object Oriented Programming at ECOOP. Darmstadt, Germany. (2003)
12. Sari R. ElDadah, N.A.S.: A process-based framework for automatic categorization of web documents. 13th Workshop for Phd Students in Object Oriented Programming at ECOOP. Darmstadt, Germany. (2003)
13. Ademar Aguiar, G.D.: A minimalist approach to framework documentation. 13th Workshop for Phd Students in Object Oriented Programming at ECOOP. Darmstadt, Germany. (2003)
14. Herve Paulino, L.L., Silva, F.: Mob: a scripting language for programming web agents. 13th Workshop for Phd Students in Object Oriented Programming at ECOOP. Darmstadt, Germany. (2003)
15. Robert E. Filman, Stu Barrett, D.D.L., Lindero, T.: Iserting ilities by controlling communications. Communications of the ACM, January **45, No 1** (2002) 118–122
16. Filman, R.E., Friedman., D.P.: Aspect-oriented programming is quantification and obliviousness. Workshop on Advanced Separation of Concerns. OOPSLA, Minneapolis (2000)
17. Filman, R.E., Havelund, K.: Source-code instrumentation and quantification of events. Workshop on Foundations Of Aspect-Oriented Languages (FOAL) at AOSD Conference. Twente, Netherlands. (2002)

Typecasting As a New Join Point in AspectJ

M. Devi Prasad
Manipal Center for Information Science, Manipal Academy of Higher Education
Manipal -576119, Karnataka, India
devi.prasad@mahe.manipal.edu
Telephone: +91-08252-573491

ABSTRACT

AspectJ does not treat typecasting as an important operation in program's execution. However, down casting and cross casting play an important role in overall behavior of a java program. We demonstrate that Java programs use type casting primarily for retrieving references to new types from distantly related ones. We investigate AspectJ's inadequacy in selecting execution points that use typecasting to yield new object or interface references. We demonstrate the necessity for a "reference creation by type casting" joinpoint and argue that its addition makes AspectJ's existing model more expressive. We bring out characteristics of such a joinpoint and illustrate its usage.

1. INTRODUCTION

AspectJ [1] is a general purpose programming language based on Java for modularizing crosscutting concerns [3]. Often, crosscutting concerns are features of an application, including (but not limited to) logging, tracing, synchronization, and caching. Experience shows that implementing these features in traditional OO languages results in scattered and/or tangled code [3]. AspectJ introduces new modularizing construct named as 'aspect'. An aspect helps in expressing new behavior in an additive manner over and above the existing OO implementation. It does so by providing language level expressions to identify and augment key structural and behavioral elements in the underlying OO system.

Essentially, aspects are useful in providing incremental extensions to components. Since components are accessed using well known interfaces, any conceived functionality enhancement achieved by aspects has to be interface centered and must be transparent to existing client code. Additionally, the new functionality may necessitate sharing state and information among members of the introduced abstraction. Some design patterns, such as Decorator or Proxy [4], are helpful in modularizing certain concerns. AspectJ can be effectively used to create design patterns [5]. When AspectJ is employed for this purpose, some important consequences of its join point model should be borne in mind:

1. When all methods of a class or interface require distinct advice, declaring pointcuts designating individual methods and providing separate advice for each pointcut is both unwieldy and undesirable. They tend to be fragmented.

2. Only around advice can define a return type for advice body. Therefore, around advice can be employed to wrap a newly retrieved interface or object reference. It can then return the reference to this wrapper instead of the wrapped reference.

Statement (2) above suggests that by suitably advising around the join points that create a new instance or obtain an interface reference, we can avoid fragmented advice constructs. Using decorators gives an OO flavor compared to separate around advice modularized in an aspect. The former lends itself to the benefits of behavioral extension by inheritance.

Such a solution relies on the expressiveness of the join point model to pick out object or interface reference creation points. In Java programming language, three kinds of expressions can generate a new reference: (1) the 'new' operator call, (2) (factory) method call, and (3) type casting expression. AspectJ can directly express only the first two join points among the three listed here.

In this paper we show that reference generation by type casting is an important programming technique used in practical programs. We argue that using AspectJ, creating decorator like wrappers is not possible in cases that use down casting or cross casting expressions. We propose a new join point to specifically capture type casting expressions that generate new references to objects or interfaces.

The rest of the paper is organized as follows: In section 2 we give a motivating example to illustrate the problem. In section 3 we show inadequacy of AspectJ in providing a simple and effective solution to this problem. In section 4 we propose a new joinpoint for capturing reference creation with type casting that improves AspectJ's expressiveness. In section 5 we discuss a prototypical implementation of this idea. We conclude section 6 by summarizing the results and discuss intended future work.

2. ILLUSTRATIVE EXAMPLE

In this section we introduce an example for reference in the subsequent sections. In this example we consider a distributed service implemented using the Java RMI infrastructure. The remote server component implements two interfaces, **InStream** and **OutStream**. The declaration of interfaces and a typical client program is also shown here. Since the remote component implementation is irrelevant to this discussion, it is not produced here. Details such as error handling or remote exception processing are not shown for brevity.

```
interface InStream extends Remote {
    int  available();
    void close();
    int  read();
    int  read(byte[] data);
}
```

```
interface OutStream extends Remote {
    void flush();
    void close();
    int write(int data);
    int write(byte[] data);
}
```

```
class Consumer {
    public long streamDataIn(Remote r) {
        long totalStreamed = 0;

        // get required interface reference
        InStream in = (InStream) r;

        // use 'in' to stream data & update
        //'totalStreamed'
        …
        return totalStreamed;
    }

    public void streamDataOut(Remote r) {
        // get required interface reference
        OutStream out = (OutStream) r;

        // use 'out' to stream data
        …
    }
}
```

```
// client of the Java RMI component…
import java.rmi.*;
class main {
    public static void main(String args[]) {
        Remote r = Naming.lookup
            ("rmi://server/ServiceProviderImpl");
        Consumer c = new Consumer();
        c.streamDataIn(r);
        c.streamDataOut(r);
        …
    }
}
```

A client streams data from the remote service using the **InStream** interface and it sends out data using the **OutStream** interface. The **Consumer** class implements the interaction between the client and the service. In Java RMI, each remote interface must inherit from a tagging interface named **Remote**. Its sole purpose is to advertise to the RMI runtime that the interface can be used for remote invocations.

Given such a component with well-known interfaces, we are interested in providing, say, a feature like client side caching for improved performance. Moreover, we expect this feature to be supplemented transparently to the client code shown here. Our aim will be to determine the limitations in AspectJ's expressiveness in handling issues that are unique to implementations represented by this example. We will also be interested in improving AspectJ's vocabulary to deal with these discrepancies.

3. INADEQUACY OF AspectJ JOIN POINT MODEL

Consider the case where the client wishes to provide caching responsibilities to **OutStream** object. For this, the caching strategy must coordinate buffer accesses among **OutStream** methods on the client side. For instance, many write operations on **OutStream** may update the local cache before requesting a **close** on the stream. However, new responsibilities of **close** include flushing the cache contents before actually closing the stream. Similarly when the local cache is full, invoking a write operation must ensure that the buffer is synced to the server before further caching.

3.1 Modeling caching concern using around advice constructs

As a first approximation, we try to encapsulate the caching concern as an aspect with unique around advice matching each method of the **OutStream** interface.

```
aspect CacheOutStream {
    byte []cache = new byte[…];
    int curOffset = 0;

    int around(OutStream out, byte[] data):
        (target(out) && call(int write(..)) && args(data) && !within(CacheOutStream) ) {
        if (cache does not have room for new data) {
            out.write(cache, curOffset);
            curOffset = 0;
        }
        // In any case, append 'data' to cache and update 'curOffset'
        copyAndUpdateOffset(data);
        // return # bytes actually written
        return …
    }
    int around(OutStream out, byte data):
        (target(out) && call(int write(..)) && args(data) && !within (CacheOutStream)) {
        //similar to the version given above
        …
    }
    void around(OutStream out):
            (target(out) && call(void close()) && !within (CacheOutStream)) {
        out.flush(); //if there is data in cache – need to flush it
        out.close();
    }
    void around(OutStream out):
            (target(out) && call(void flush()) && !within (CacheOutStream)) {
        if (curOffset > 0) { //if there is data in cache – need to flush it
            out.write(cache);
            out.flush();
        }
    }
}
```

It is evident from this example that specifying proper pointcuts poses a non trivial challenge. We should also be careful to avoid recursion in advice execution. In addition, we should take special care in declaring the pointcut to specifically avoid matching **OutStream.write()** calls made within the aspect declaration.

We can also infer from this example that when it is necessary to impart additional behavior for an underlying module, all or a large number of elements belonging to that module might require enhancement. When methods of an interface demand coordinated behavior, advising individual methods with unique responsibilities becomes tedious. Even when such advices are provided, program readability or comprehension may get affected because it requires some effort for one to associate an advice with a method.

A decorator design pattern [4] represents an alternate solution in such contexts. A decorator provides additional behavior around an existing implementation by implementing the same interface(s) as the target object. So it conforms to the interface layout expected by the client.

3.2 Modeling caching concern using decorator and around advice constructs

In most of the cases, decorator can be created by advicing around join points that generate objects or interface references. For example, the following around advice intercepts requests to create a new instance of RMI component and decorates the fresh object with a wrapper **RemoteServiceDecorator**. This wrapper object maintains reference to the original remote object. The decorator decides when certain calls need delegation and appropriately routes call to the remote object.

```
aspect CachingOutStreamDecorator {
    …
    Remote around() : call(public static Remote java.rmi.Naming.lookup(..)) {
        Remote r = proceed();
        Remote remoteDecorator = new RemoteServiceDecorator (r):
        return remoteDecorator;
    }
}
```

We can set up similar advice around object creation joinpoints that use the 'new' operator. These two joinpoints (a factory method call and constructor call) represent important junctures in program execution where new object or interface references are either generated or transferred. With this arrangement, the aspect **CachingOutStreamDecorator** trivially captures decorator creation concern. Actual caching concern is implemented by the **RemoteServiceDecorator**.

However, a careful analysis of this solution uncovers a serious problem. In the current example, the client can freely downcast or crosscast one reference to the other among **InStream**, **OutStream** or **Remote** interface types. Since the around advice wraps remote object reference with a decorator object, the client obtains a **RemoteServiceDecorator** reference instead of remote object reference. This implies that this decorator should implement all interfaces that the remote component exposes. Otherwise, subsequent type cast to some expected remote interface in the client code would throw runtime exception. In our example, **RemoteServiceDecorator**, therefore, should implement a trivial version of **InStream** that simply forwards calls to remote object along with an implementation of **OutStream** that actually encapsulates the caching concern.

Expecting clients to provide trivial implementation for interfaces of no interest to the application is not only counter intuitive but also inefficient. Given the simplicity of the problem at hand, we should be able to provide a simple and efficient solution.

3.3 Type cast as a major join point

In our RMI example, the **streamDataIn** and **streamDataOut** methods (of the **Consumer** class) are designed to work with one interface type, **InStream** and **OutStream** respectively. We wish to decorate only the **OutStream** interface, perhaps only inside **streamDataOut** method. However, **streamDataIn** and **streamDataOut** methods obtain reference to a sub type (**InStream** or **OutStream**) by down casting a reference variable of the super type (**Remote**). Because the remote object in our example implements both **InStream** and **OutStream**, a cross cast from **InStream** to **OutStream** is a perfectly valid operation.

Type casts of these two kinds are generally employed in traversing down and across type hierarchies. It is a common technique used in languages such as Java that lack support for templates or generic types. Even languages that support template types provide for down cast or cross cast. C++, for instance, provides three different flavors of type cast expressions. We can view these expressions as events that generate new information from the currently available one. Therefore they convey important purpose in program's execution.

AspectJ however does not treat down casting and cross casting to be interesting join points in a program's behavior. Thus, in AspectJ, there is no point cut expression that can pick out these two constructs for further advice. This also means that there is no way in AspectJ to create efficient and minimal decorator per interface if the program chooses "reference creation by down casting" pattern as illustrated in our example.

4. A PROPOSAL FOR NEW JOINPOINT

In this section, we give some salient features of a "reference generation with down (or cross) cast" join point and show its typical usage within AspectJ. We term this join point as a **reftrans** join point. It is a point in the program execution where

- A super type reference is down cast to a sub type
- Some interface or object reference is cross cast to another type

The above definition considers only reference variables as the source (i.e., right hand side (RHS)) of type casting. The two other kinds of expressions that can yield an interface or object reference are a method call or new operator. Since AspectJ supports them directly, there is no necessity to treat them special.

Here is a pointcut declaration that can pick out **reftrans** joinpoints. We assume that the **OutStreamDecorator** class provides caching support for the **OutStream** clients.

```
aspect RefTransDecorator {
    pointcut OutStreamFromRemote(Remote r) : args(r) && reftrans((OutStream) r);

    OutStream around(Remote remObj) : OutStreamFromRemote(remObj) {
        OutStreamDecorator outDecorator = new OutStreamDecorator((OutStream) remObj);
        return outDecorator;
    }
}
```

The emphasized part of pointcut declaration represents support for picking out a type cast from super type (**Remote**) to sub type (**OutStream**). The declaration contains two parts: an **args** declarator from the AspectJ vocabulary combined with **reftrans**. The **args** declaration indicates source type of the cast expression. The **reftrans** pointcut specifies the argument picked out by **args** and the target type of cast expression. In this manner, the complete context for type casting is inferred. When a join point matches this declaration, AspectJ picks up the actual argument at that join point and makes it available at the advice. When the **RefTransDecorator** aspect (listed above) is applied to our example code, the statement

```
OutStream out = (OutStream) r;
```

in **streamDataOut** method of **Consumer** class matches the **OutStreamFromRemote** point cut. The **remObj** parameter of the around advice is bound to reference **r** from the type casting statement. The advice body passes this reference to the **OutStreamDecorator** constructor.

This simple example illustrates combining the **reftrans** join point with an around advice to create a wrapper for spontaneously created interface references. It provides a simple yet flexible alternate to writing many **after** and **before** advices for individual members of an interface. It gives developers more expressive power at a little added cost.

5. A PROTOTYPE IMPLEMENTATION WITHIN AspectJ

We have modified the AspectJ compiler [2] to support the proposed **reftrans** joinpoint. The support for this feature meshes well with existing syntax and semantics of AspectJ. A primitive pointcut representing **reftrans** is the only addition to the existing repertoire. Normal AspectJ point cut declaration (PCD) can combine this pointcut with other pointcuts in the usual manner.

6. CONCLUSION AND FUTURE DIRECTIONS

There are two major contributions in this paper. First, we have illustrated that in some cases generating a decorator for an object gives more flexibility than developing separate **before** and **after** advice for individual methods of an object. Second, we demonstrated the necessity to capture particular kinds of type cast expressions as important points in program execution. We showed how, under certain circumstances, the absence of a **reftrans** like join point in AspectJ causes inconvenience in creating decorator objects. In addition to these, we have provided a proof of concept implementation of this facility within AspectJ.

The current implementation does not handle certain special cases. For instance, in the following Java statement

InStream in = (InStream) (OutStream) r;

type coercion is repeatedly applied to obtain same result as in the following pair of statements:

OutStream out = (OutStream) r; InStream in = (InStream) out;

The current implementation considers only those statements that down cast or cross cast a reference variable. I plan to study cases where repeated type coercion is meaningful and extend the implementation accordingly.

Java currently does not support operator overloading. In contrast, C++ allows operator overloading. In C++, one can overload various type coercion operators for a user-defined type. The translator for C++ wires up a call to the appropriate overloaded coercion method. I plan to study the implication operator overloading on the design of join points in general and to **reftrans** in specific. In addition, the proposed generics support for Java [6] will undoubtedly reduce the necessity of type coercion. On the other hand, for dynamic type discovery (as shown in the RMI example in this paper), type casting will continue to be an important language element. I plan to study the effect of generics on the join point model of AspectJ.

Elsewhere researchers have discussed the necessity of more expressive and precise join points for current generation of AOP languages [7]. This paper has suggested a "low-level" join point that works only at the code level. AspectJ is also a language with low-level support for specifying and composing crosscutting code into a core system. There is certainly a need for a means to separate crosscutting concerns across the lifecycle [8]. We have been working on a project that attempts to reverse engineer an AspectJ based software system into an extended UML model and a collection of OCL constraints [9].

7. ACKNOWLEDGEMENTS

The anonymous reviewers of the initially submitted version of this paper gave good advice on the content, organization and flow of material. Professor B.D. Chaudhari helped shaping the ideas in the initial stages and provided comments on the earlier drafts of this paper.

8. REFERENCES

1. AspectJ Programming Guide. http://www.eclipse.org/aspectj/
2. AspectJ 1.0.6 source code http://www.eclipse.org/aspectj/
3. Gregor Kiczales, et al., *Aspect-Oriented Programming*. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), June 1997.
4. E. Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Reading, MA: 1995
5. Jan Hannemann and Gregor Kiczales. *Design Pattern Implementation in Java and AspectJ.* OOPSLA - 2002.
6. JSR-000014, *Adding Generics to the Java™ Programming Language*. http://www.jcp.org/aboutJava/communityprocess/review/jsr014/
7. Gregor Kiczales. *The Fun Has Just Begun*. Keynote Address, AOSD - 2003. (http://aosd.net/archive/2003/kiczales-aosd-2003.ppt)
8. Siobhan Clarke and Robert J. Walker. *Towards a Standard Design Language for AOSD,* AOSD - 2002.
9. Kleppe and J. Warmer. *The Object Constraint languag, Precise modeling with UML.* Addison Wesley Professional.

Mob: a Scripting Language for Programming Web Agents

Hervé Paulino[1], Luís Lopes[2], and Fernando Silva[2]

[1] Department of Informatics. Faculty of Sciences and Technology. New University of Lisbon.
email: herve@di.fct.unl.pt
[2] Department of Computer Science. Faculty of Science. University of Oporto.
email: {lblopes,fds}@ncc.up.pt

Abstract. Mobile agents are the latest software technology to program flexible and efficient distributed applications, since they are independent programs that travel over the network, focusing on local communication, rather than the usual communication paradigms. Most current systems implement semantics that are hard if not impossible to prove correct. In this paper we present Mob, a scripting language for web agents encoded on top of a process calculus and with provably sound semantics that allows interaction with programs written in many programming languages.

Keywords: mobile agents, Internet computing, distributed systems, process calculus

1 Introduction and Motivation

The introduction of the π-calculus [7, 13] and other related process calculi, in the early nineties, as a model for concurrent distributed systems, provided the theoretical framework upon which researchers could build solid specifications. The main abstractions in these calculi are processes, representing arbitrary computations and channels, representing places where processes synchronize and exchange data. Recent extensions of these models introduced another fundamental abstraction, sites, which denote places in a network where processes run. These extensions allowed, for the first time, the modeling of complex distributed systems with mobile resources [2, 6, 10, 12, 17].

The mobile resources supported by these recent extensions are a powerful abstraction for the development of mobile agent frameworks. Mobile agents add to regular agents the capability of traveling to multiple locations in the network, by saving their state and restoring it in the new host. As they travel, they work on behalf of the user, such as collecting information or delivering requests. This mobility greatly enhances the productivity of each computing element in the network and creates a powerful computing environment, focusing on local interaction. Thus, our mobile agents are independent programs that travel over the network, focusing in local communication, rather than the usual communication paradigms (e.g., client-server).

In this paper we present a scripting language, Mob, for programming mobile agents in distributed environments. The semantics of the language is based on the DiTyCO (Distributed TYped Concurrent Objects) process calculus [17]. The run-time for the language is provided by the current implementations of the DiTyCO [11]. In particular, we rely on it for interprocess communication and code migration over the network.

The development of mobile agents requires a software infrastructure that provides migration and communication facilities, among others. Current frameworks that support mobile agents are mostly implemented by defining a set of Java classes that must then be extended to implement a given agent behavior, such as Aglets [3], Mole [8], or Klava [9]. MOB, on the other hand, is a simple scripting language that allows the definition of mobile agents and their interaction, an approach similar to D'Agents [15]. However, MOB applications may interact with external services programmed in other languages than MOB. Furthermore, the language is compiled into a process-calculus based kernel-language, and its semantics can be formally proved correct relative to the base calculus. Therefore, in this sense, MOB features some language security. Correctness or type-safety results are difficult to produce for most of the current systems. For example, only a subset of Java programs can be proved to be correct and type-safe [14].

The interaction between MOB applications and external services, provided by MOB enabled hosts, can be programmed in many languages such as Java, C, TCL or Perl. The philosophy is similar to that used in the MIME type recognition. The runtime engine matches a file type against a set of internally known types and either launches the corresponding application or simply executes the code.

The remainder of the paper is structured as follows: section 2 describes the target language, TyCO, in which MOB is encoded; section 3 describes the MOB programming language; section 4 provides some MOB programming examples; section 5 describes the compilation of a MOB program in TyCO; and finally section 6 describes the on-going research and future work.

2 The Target Language - DiTyCO

Our target language is based on a process calculus, in the line of the asynchronous π-calculus, named DiTyCO [17]. The main abstractions of the (centralized) calculus are channels (communication endpoints), objects (collections of methods that wait for incoming messages at channels) and asynchronous messages (method invocations targeted to channels). It is also possible to define process definitions, parameterized on a set of variables, that may be instantiated anywhere in the program (this allows for unbounded behavior). The abstract syntax for the core language is the following:

P	::=	$\mathbf{0}$	terminated process
	|	$P \mid P$	concurrent composition
	|	**new** x P	new local variable
	|	$x!l[\tilde{v}]$	asynchronous message
	|	$x?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$	object
	|	**def** $X_1(\tilde{x}_1) = P_1 \ \ldots \ X_n(\tilde{x}_n) = P_n$ **in** P	definition
	|	$X[\tilde{v}]$	instantiation
	|	**if** v **then** P **else** Q	conditional execution

where x represents a variable, v a value (a variable or a channel), X a process definition and, l a method label.

From an operational point of view, centralized DiTyCO computations evolve for two reasons: object-message reduction (i.e., the execution of a method in an object in response to the reception of a message) and, instantiation of definitions. These actions can be described more precisely as follows:

$$x?\{\ldots, l(\tilde{x}) = P, \ldots\} \mid x!l[\tilde{v}] \rightarrow \{\tilde{v}/\tilde{x}\}P$$

The message $x!l[\tilde{v}]$ targeted to channel x, invokes the method l in an object $x?\{\ldots, l(\tilde{x}) = P, \ldots\}$ at channel x. The result is the body of the method, P, running with the parameters $\tilde{x}$ substituted by the arguments $\tilde{v}$. For instantiations we have something very similar.

$$\textsf{def } \ldots\ X(\tilde{x}) = P\ \ldots\ \textsf{in } X[\tilde{v}] \mid Q \rightarrow \textsf{def } \ldots\ X(\tilde{x}) = P\ \ldots\ \textsf{in } \{\tilde{v}/\tilde{x}\}P \mid Q$$

A new instance $X[\tilde{v}]$ of the definition X is created. The result is a new process with the same body, P, as the definition but with the parameters $\tilde{x}$ substituted for the arguments $\tilde{v}$ given in the instantiation.

This kernel language constitutes a kind of assembly language upon which higher level programming abstractions can be implemented as derived constructs.

The full, distributed, calculus grows from the centralized version by adding a new layer of abstraction representing a network of *locations*, identified by r, s, where processes run.

N	::=	$\mathbf{0}$	terminated network
	|	$N \parallel N$	concurrent composition
	|	**new** $x@s\ N$	new local variable
	|	**def** $D@s$ **in** N	definition
	|	$s[P]$	location with running process

This additional layer does not however introduce new reduction operations in the calculus. In fact, reduction can only be performed locally at locations, either by communications or instantiations as described above.

As can be observed from the above syntax, all resources are lexically bound to the locations they are created on. Thus, a message or object located at some channel $x@s$ must first move to location s to in order to reduce. Similarly, an instantiation of a definition $X@s$ must move to location s in order to reduce.

To preserve the lexical bindings of resources, every time one moves to another location, all its free identifiers (references for resources it uses) are translated on-the-fly. This is represented by a transformation σ_{rs} meaning "translation of identifiers when moving from location r to location s".

The lexical scope on resources together with the requirement of local reduction induce the following rules for resource migration:

Message Migration	$r[x@s!l[\tilde{v}]] \rightarrow s[x!l[\tilde{v}\sigma_{rs}]]$
Object Migration	$r[x@s?M] \rightarrow s[x?M\sigma_{rs}]$
Remote Instantiation	**def** $X@s(\tilde{x}) = P$ **in** $r[X@s[\tilde{v}]] \rightarrow$ **def** $X@s(\tilde{x}) = P$ **in** $s[X[\tilde{v}\sigma_{rs}]]$

One final word is required on: (a) lexical scope, and; (b) local reduction, since they are design goals for the language. Lexical scope is an important property since it provides the compiler and run-time system with important information on the origin of a resource. This is important namely for safety reasons (e.g., does the resource come from a trusted location ?) and for implementation reasons (e.g., where do we allocate the data-structures for it ? Do they move around in the network ?).

Local reduction is also of the utmost importance. Client-server interactions for example occur within a location, with much lower overheads than in the standard Client-Server model where interactions required maintaining remote sessions open and the exchange of many messages drastically reducing the available bandwidth of a network. In the novel paradigms for Web Computing [1], client applications move to server locations where they

interact with a local session. They return to their original location after the local session is complete.

The following programming example illustrates the use of these primitives and derived constructs. We use the (**let**/**in**) derived constructs, defined in [16], for synchronous method calls.

```
def Cell (self, value) =
    self ? {
        read (replyto) = replyto![value] | Cell [self, value]
        write (newValue) : Cell [self, newValue]
    }
in def
    IntegerCell(self, value) = Cell [self, value]
    StringCell(self, value) = Cell [self, value]
in
    new c IntegerCell[c, 4] | let i = c!read[] in io!printi[i]
```

The general Cell template stores a value and features read and write methods to retrieve or change its contents. Type specific templates, such as IntegerCell, based on the general Cell template will be used further on on the paper, to encode values in DiTyCO. In this example, an IntegerCell is created to store value 4 and the read method is used to retrieve this value in order to print it to the standard output. Notice that Cell is polymorphic on value.

3 The Source Language - Mob

The Mob programming language is a simple, easy-to-use, scripting language. Its main abstractions are *mobile agents* that can be grouped in *communicators* allowing hierarchies to be formed, group communication and synchronization. The abstract syntax for the kernel of the language is as follows:

```
            Program ::= AgentDef | AgentDef Program | InstructionList | InstructionList Program
           AgentDef ::= agent id { AttributeDef Init Do Iterators}
      AttributeDef ::= [] | id ; AttributeDef
               Init ::= init CodeBlock
                 Do ::= do CodeBlock
          Iterators ::= [] | next CodeBlock previous CodeBlock
         CodeBlock ::= Instruction | { InstructionList }
   InstructionList ::= [] | Instruction ; InstructionList | Instruction \n InstructionList
       Instruction ::= NewComm | NewAgent | Statement | id = Command | Command
           NewComm ::= id = communicator StringLiteral | communicator StringLiteral
          NewAgent ::= id = agentof id Attributes | agentof id Attributes
```

The language defines a set of reserved words for constructs and built-in attributes, here written in boldface.

3.1 The Agent Abstraction

The development of a Mob mobile agent implies two stages: the first (**init** section) consists on a setup that runs prior to the agent's actual execution. Usually it is used to assign initial values to the agent's attributes. The second (**do** section) defines the agent's behavior throughout its journey.

A Mob agent features several built-in attributes: **email**: email address of the agent's owner; **owner**: identification of the agent's owner; **home**: home hosts of the agent; **itinerary**: the agent's itinerary; **strategy**: definition of a strategy of how the itinerary must be traveled; and **sindex**: the index in the itinerary of the current host. Attributes **owner**, **email** and **home** are read-only, while all the others can be altered during the agent's execution.

Although Mob features several strategies for traversing the itinerary, namely: list, tree, circular and scatterjoin, it allows the programmer to define new ones. An agent's itinerary is seen as an object that can be managed through two iterators **next** and **previous**.

Beside the built-in attributes, an agent may feature as many attributes as the programmer wishes. Their usefulness is to hold values to be retrieved when the agent migrates back home. The following example presents the skeleton of a Mob agent definition, Airline, that includes a new user-defined attribute, price.

```
agent Airline {
    price;
    init { price = 0 }
    do { // Implementation of the agent's actions/behavior }
}
```

Now that an agent behavior is defined an undetermined number of agents can be created. The following example creates an agent named airline owned by johndoe and with **home** hosts host1 and host2. One can also launch several agents at once using the **-n** flag. airlineList will contain the returned list of agent identifiers Notice that the attribute initialization supplied in the agent constructor will not override the ones in the **init** section.

```
airline = agentof Airline -u "johndoe" -h "host1 host2"
airlineList = agentof Airline -n 10 -u "johndoe" -h "host1 host2"
```

Each agent must be associated to an owner, defined in an entry of the Unix-like file named passwd. An entry of such a file must contain the user's login, name, password and group membership. Following the Unix policy for user management, users may belong to groups defined in the groups file, sharing their access permissions. Each Mob enabled host must own both files in order to authenticate each incoming agent. As featured in FTP servers, an agent can present itself as anonymous for limited access to local resources.

3.2 The Communicator Abstraction

Communicators are conceptually equivalent to MPI communicators [4] and allow group communication and synchronization. As presented in the grammar, the **communicator** construct only requires the list of agents (may be empty) that will start the communicator. Other agents may join later.

3.3 Instructions

Mob features a rather small but fully functional set of instructions. Most of the statements included in Mob are common in all scripting languages (**for**, **while**, **if**, **foreach** and **switch**), the only difference lies in the **try** instruction, a little different from the usual error catching instructions found in, for instance, TCL. Its syntax is similar to the try/catch exception handler instruction of Java, allowing specific handling of different types of local run-time system exceptions. Mob provides instructions to define a mobile agent's behavior and its interaction with other agents and external services. These commands can be grouped in the following main sets:

1. agent manipulation: **clone**.
2. mobility: **go**.
3. check-pointing: **savestate** and **getstate**.
4. inter-agent communication: asynchronous (**send**, **recv**, **recvfrom**), synchronous (**bsend**, **brecv**, **brecvfrom**), communicator-wide (**csend**) and multicast (**msend**). There are variants of these functions for use with the HTTP and SMTP protocols (e.g., **httpcsend**; **smtprecv**). These variants are useful to bypass firewalls that only allow connections to ports of regular services.
5. managing communicators: **cjoin** and **cleave**.
6. execution of external commands: **exec**. This functionality allows the execution of commands external to the Mob language. The Mob system features a set of service providers that enable communication through known protocols, such as HTTP, SMTP, SQL and FTP. The interaction with these providers is possible through **exec**'s protocol flag.
7. input/output: Mob's input/output instructions are implemented as syntactic sugar for the **exec** instruction. **open** filename could also be written as **exec** -p fs open filename.

4 Programming with Mob

Now that the language syntax is presented, this section introduces simple Mob programming examples.

We intend to develop two agents: one, airline, capable of querying each host of its itinerary for the price of one airline ticket from Lisbon to Las Vegas; and a second, hotel, capable of querying the hosts of its itinerary for a single's room in a Las Vegas hotel.

In the airline example, the **init** section will set the itinerary as the first ten results of a query to a search engine, and price as zero. The actual program starts with a query to a hypothetical ticketsDB database for the price of the tickets. Note that the syntax of the query will be defined by the implementation of the ticketsDB server and not by the language. The execution of **exec** is protected by a **try** instruction. If no exception is caught the program continues and newprice and price are compared, otherwise nothing is done. Once the end of the program is reached the agent migrates to the next host in its itinerary (default strategy) and restarts the execution of the program. When all of the itinerary has been processed, the agent migrates to one of the hosts defined in the **home** attribute.

In the hotel example, to enhance the efficiency the search is divided among several agents, all members of a ghotel communicator. This provides group communication to spawn new cheaper prices among the agents.

In order to avoid a needless search of an hotel if there are no available airline tickets to Las Vegas, the airline agent can interact with the ghotel communicator through the **csend** command, and inform all the agents from ghotel that they can finish their execution and return home.

```
agent Airline {
    price // declaration of the price attribute
    init {
        itinerary = exec -p http -n 10 www.search_engine.com "airline company" // query engine for itinerary
        price = 0 // initialize price attribute
    }
```

```
    do {
        try { // protect database access with a exception handling mechanism
            newprice = exec -p sql ticketsDB ""price" "Lisbon" "Las Vegas"" // query database for ticket price
            if (newprice < price || price == 0)
                price = newprice
        }
        catch // exception caught
            write log "Could not access database in " + hostname // cannot access database
        if (sindex+1 == [lsize itinerary] && price == 0) // is the search over and no ticket is found?
            csend ghotel "stop" // no ticket found, send "stop" message to ghotel communicator
    }
}
airline = agentof Airline -u "johndoe" -h "host1 host2" // create a new agent
```

In this second example, the **exec** to the search engine is now done outside the agent's definition. The result is scattered among the 10 agents launched. Notice that each agent joins the ghotel communicator in the **init** section and that all the agents terminate their execution and return home if they receive the stop message from the airline agent. Also notice that, every time a cheaper price is found it is spawned to the communicator.

```
agent Hotel {
    price // declaration of the price attribute
    init {
        strategy = "list" // definition of the agent's strategy
        cjoin ghotel // join the ghotel communicator
    }
    do {
        if ([recvfrom airline] == "stop") // did the airline agent terminate without finding any tickets?
            go -h home// search is over, migrate home
        try { // protect database access with a exception handling mechanism
            newprice = exec -p sql hotelDB "price" "single room" // query database for hotel room price
            if (price < newprice) {
                price = newprice
                csend ghotel price // spawn new price to the communicator
            }
            newprice = recv // probe and receive (if any) a new price from the communicator
            if (price < newprice)
                price = newprice
        }
        catch // exception caught
            exec -p smtp email "Could not access database in " + hostname // could not access database
    }
}
ghotel = communicator// create communicator
list = exec -p http -n 50 www.search_engine.com "hotel Las Vegas" // query search engine
for (i = 0; i < 50; i = i+10)
    agentof Hotel -i [lrange list i 10] // create new agents with a sublist of list as the itinerary
```

5 The Compilation Scheme

In this section we briefly sketch how a Mob program can be encoded into the DiTyCO language and run-time. Using the airline example, the agent is encoded into a AirlineAgent extension of the general agent definition Agent. This extension overrides the init and do methods and introduces a new attribute, price.

The following code illustrates the encoding of the airline agent into DiTyCO. The full encoding of the Mob language into DiTyCO may be found in [5].

```
def AirlineAgent(self, name, ..., homes, ..., price) =
    self ? {
        init() =
            {- Encoding of the init section. -}
            AirlineAgent[self, name, ..., homes, ..., price] |
            self!do[]

        do() =
            {- Encoding of the do section. -}
            AirlineAgent[self, name, ..., homes, ..., price]

        {- all the methods inherited from Agent -}
    }
in
```

The instantiation of the Airline definition in Mob corresponds of an instantiation of the AirlineAgent definition in DiTyCO. Continuing with the airline example, the following agent construct

```
airline = agentof Airline -u "johndoe" -h "host1 host2"
```

is encoded in the following DiTyCO code.

```
(1) new var10 StartList[var10] |
    new var11 AddToList[var11, "host2", var10] |
    new var1 AddToList[var1, "host1", var11] |
    ...
(2) new airlineUser StringCell[airlineUser, "johndoe"] |
    ...
    new airlineHomes ListCell[airlineHomes, var1] |
    ...
    new airlineUserDefined0 IntegerCell[airlineUserDefined0, 0] |
(3) new airline AirlineAgent[airline, airlineUser, ..., airlineHomes, ..., airlineUserDefined0] |
    airline!init[]
```

The **agentof** encoding of the airline agent is divided in three sections: the first (1) is dedicated to constructing all the lists required by the AirlineAgent definition (e.g., itinerary, homes, lists for managing incoming and outgoing messages, ...); the second (2) for building Cell objects for each user accessible attributes; and finally (3) creating the airline object and starting its execution, by invoking its init method.

After this static encoding into DiTyCO the Mob program may be compiled and executed using the DiTyCO run-time engine.

6 Conclusions and Future Work

Mob is currently under implementation. All the features, excluding external services and exception mechanisms, are fully encoded in DiTyCO. Future work will focus on the encoding and development of external services, such as, recognition/execution of programs in several high-level languages, building itineraries through external search engines, database communication, and network communication through known protocols, such as SMTP, FTP, or HTTP.

Once the first prototype is ready, case studies will be programmed to provide a base for discussion of the language's strong and weak points. Work will also be done in security (agents and hosts), and in providing an integrated tool for programming, debugging and monitoring of the agents.

Acknowledgments. This work is partially supported by FCT's project MIMO (contract POSI/CHS/39789/2001) and the CITI research center.

References

1. Fuggetta A., Picco G. P., and Vigna G. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.
2. Fournet C. and Gonthier G. et al. A Calculus of Mobile Agents. In *International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.
3. Lange D. Programming Mobile Agents in Java. In *WWCA 1997*, pages 253–266, 1997.
4. MPI Forum. The MPI Message Passing Interface Standard. www-unix.mcs.anl.gov/mpi/, 1994.
5. Paulino H., Lopes L., and Silva F. Encoding and Compiling Mob in DiTyCO. To appear.
6. Vitek J. and Castagna G. Seal: A Framework for Secure Mobile Computations. In *Workshop on Internet Programming Languages*, 1999.
7. Honda K. and Tokoro M. An Object Calculus for Asynchronous Communication. In *European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.
8. Straber K., Baumann J., and Hohl F. Mole - A Java Based Mobile Agent System. In Mühlhäuser M., editor, *Special Issues in Object Oriented Programming*, pages 301–308, 1997.
9. Bettini L., De Nicola R., and Pugliese R. Klava: a Java Framework for Distributed and Mobile Applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.
10. Cardelli L. and Gordon A. Mobile Ambients. In *Foundations of Software Science and Computation Structures (FoSSaCS'98)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.
11. Lopes L., Silva F., Figueira A., and Vasconcelos V. DiTyCO: Implementing Mobile Objects in the Realm of Process Calculi. In 5^{th} *Mobile Object Systems Workshop (MOS'99)*, June 1999.
12. Abadi M. and Gordon A. A Calculus for Cryptographic Protocols: the Spi-Calculus. In *Computer and Communications Security (CCS'97)*, pages 36–47. The ACM Press, April 1997.
13. Milner R., Parrow J., and Walker D. A Calculus of Mobile Processes (parts I and II). *Information and Computation*, 100(1):1–77, 1992.
14. Drossopoulou S., Eisenbach s., and Khurshid S. Is the java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.
15. Gray R. S. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, 1995.
16. Vasconcelos V. TyCO Gently. DI/FCUL TR 01–4, Departamento de Informática da Faculdade de Ciências de Lisboa, July 2001.
17. Vasconcelos V., Lopes L., and Silva F. Distribution and Mobility with Lexical Scoping in Process Calculi. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science, 1998.

A Process-based Framework for Automatic Categorization of Web Documents

Sari R. ElDadah
Department of Information Systems
Arab Academy For Banking and Financial Sciences
Amman, Jordan
Sdadah@students.aabfs.org

Nidal Al Said
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
nidal@softlab.ece.ntua.gr

Abstract
This paper presents the design of a Framework for the development of Automatic Text Categorization (ATC) applications of Web Documents. The process, composed of 4 activities, is conducted from the various ATC methods developed so far, and described based on Petri Nets process description language. The paper concludes with notes about ATC process.
Keywords: Frameworks, Collaboration, Slots, Tabs, Knobs, Dials, Petri Nets, Agents, Automatic Text Categorization, Collaborative Recommendation, Information Retrieval, Neural Networks, Data Mining.

1. Introduction

Automatic Text Categorization (ATC) is a field of search that has received increasing attention in the last few years, due to the enormous amount of information that has become available on the Internet. As the Internet gains more popularity for being a large source of information, huge amounts of information are either added or viewed every minute. However, the lack of having an effective means to organize information for easy and fast search threatens the essence of the Internet, because regardless of how large and rich a set of information is, it will not be optimally useful if its contents is hard to explore.

Librarians have been arduously performing the task of categorization for centuries [1], but assigning the categorization task to manual procedures would not be an effective solution for an information source like the Internet, and therefore, the idea of automating the categorization task has emerged.

Many ATC methods and algorithms have been developed to automate the categorization process in different levels so far. Information Retrieval [3,7,15], Neural Networks [12,13,24], Knowledge-based Approaches [25], Ontology Classification [1,22,23], and Data Mining [16], just to mention a few, are examples of the fields from where ATC methods have been developed. In all these methods, the categorization task was performed with

different levels of automation, ranging from just classifying the web documents into a predefined set of categories, to completely automating the process of exploring the categories, finding descriptions for each category, and relating each web document to one or more categories based on its relevance to it.

Another major aspect of categorization is the degree of quality, which is not only depending on the efficiency of the method used, but also varies from one user to another according to his/her interests and background knowledge. A car stuff web site may be classified in the category of environment pollution from the viewpoint of a green-peace worker, for example, while the same web document might be categorized as "Sport Cars Group" by another user who is a fan of sport cars. This explains how complex the categorization process of web documents is, and exposes a challenge to the research society in order to achieve the perfect view of the Internet, individually categorized and classified for each user, as depicted in Figure1.

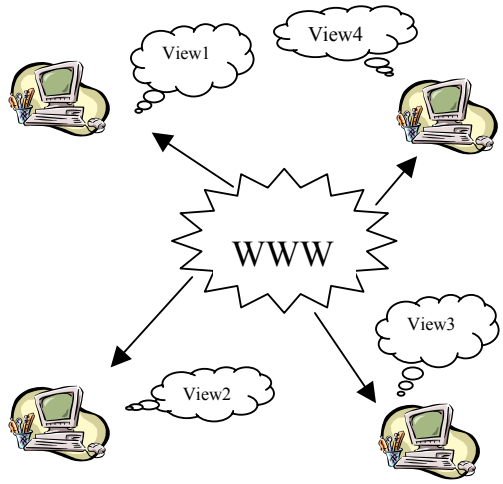


Figure 1. The aimed vision of the Internet

Accomplishing such personalized categorization has many significant impacts and benefits on the world of Internet; i) An effective search through the web reduces the time consumed and provides more relevant search results, ii) A recommendation service can be provided through the well categorized web documents, with the advantage of recommending each user according to his/her view of interests, iii) and This can be further exploited in the E-Business activities, so that when a user searches for a mobile to buy ,for example, an effective search would provide him/her with the best choices available on the web and also recommend according to the user's view of taste.

The next section explains the proposed process for building an efficient ATC, which has the advantages of providing a personalized categorization for each Internet user.

2. The ATC Process

Although there are many methods developed for ATC, the steps followed in each method were algorithm-driven, i.e., each method follows a sequence of steps generated by the algorithm used in the method (e.g., Information Retrieval). ATC methods also vary in the degree of automation of the categorization process. Some ATCs aim to develop a classification scheme, which can be used to categorize documents automatically, using a manual or supervised list of categories [2, 15]. Other methods go beyond this task and automate the exploration of categories [11].

Previous work and research accomplished in the field so far, has four major steps that are conducted and suggested to build an effective ATC. These steps are identified as separate activities in the proposed framework, each one can be seen as a layer that has an interface, predefined input, predefined output, and is independent in the work it performs.

The proposed framework is also designed to possess an object-oriented nature. Each activity can be assigned to a separate object. Two or more activities can also be done at the same object according to the designed components, with largely predefined cooperation patterns between them. This allows components to be replaceable by others that conform to the slots, tabs, knobs, and dials the framework exposes to users who want to adapt the framework to their own context [29].

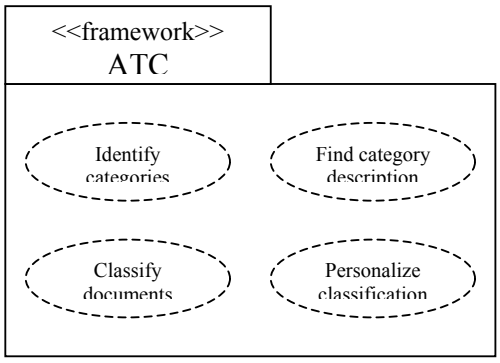


Figure 2. Modeling ATC Framework

In Figure 2, the ATC framework is modeled as a package, which contains four collaborations; each one corresponds to an ATC step. A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all its parts [29]. A collaboration is also the specification of how an operation is realized by a set of classifiers and associations playing specific roles used in a specific way.

2.1 Identifying significant categories

The first step of the ATC is to identify what categories the web documents will be classified into. It is important to determine exactly all the categories, considering almost all the grouping probabilities. This activity is

very complicated to automate, as the task of category identification implies the know-of-semantics for categories [28]. However, preserving this task to be manually performed requires huge and expensive efforts, due to the large amount of web documents, which grows and changes rapidly.

2.2 Finding the best description for each category

In order to identify the categories, each category must have a clear and exact description that differs one category from another. Finding expressive description for every category is a quality measure [3]. Identifying categories and finding their description have been mixed and considered one process in many methods [7, 15]. New trends talk about ontology characteristics of the web documents, where categories are predefined first, and finding their descriptions then follows by experimenting a test sample of web documents [1, 28]. A conclusion from those two trends shows that working in the two activities in parallel is possible.

2.3 Classifying the documents into the identified categories

This task relates a web document to one or more category. The foundation of relating the web document here comes from the basic features and description of the document (i.e., its content). It is important to explore all the possible contributions that a document may have in the different categories, so they can be seen by different levels (according to the a contribution measure) in all possible categories.

2.4 Personalizing the categories, web documents' classification, and their relevant descriptions according to the user preference

Modifying the categorization measures with respect to the user's preferences is a major contribution for the categorization process. Therefore, each user can search the web according to his/her view, which makes surfing web documents easier and more beneficial. Research has been done on personalizing aspects of the Internet, including web site structure and recommendation systems, and can be used to provide a user with a personally categorized view of the web. Collaborative recommendation is a candidate method that can accomplish this task.

In the following section, a model for the four steps is provided using Petri Nets [26], a standard process modeling techniques.

3. ATC Framework Concepts

The four-step process of ATC will be modeled in the proposed framework as activities through the use of standard process modeling techniques. The steps of ATC process are modeled according to [27].

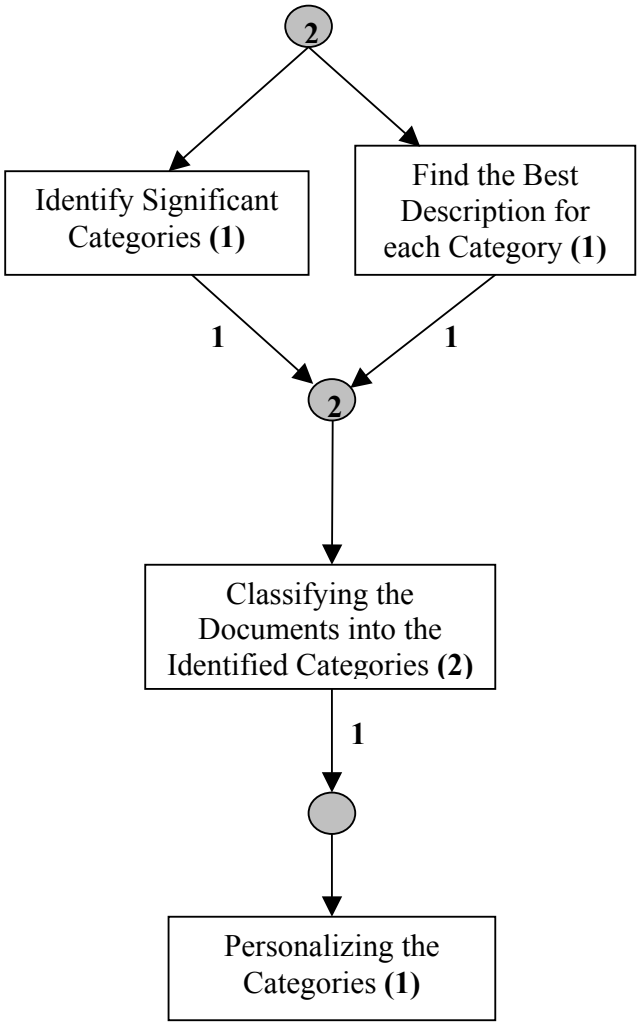


Figure 3. A Petri net Model for the Framework

The four steps are modeled as *Activities*, which generate *Artifacts*, and are performed by *Actors*. The first two activities may be achieved in parallel, as discussed before, but depend exclusively on their outputs. During the execution to an activity, *Fields* of an *Artifact* (output) can be updated by the actor(s) assigned to the activity. Figure 3 illustrates the process description, in which each activity is represented by a box with a name and the number of tokens it requires. Locations are represented by small circles. The number of tokens in each location determines the *State* of the system. The set of activities are capable of being performed either by human (through full handling or some degree of supervision), or automatically by Agents. Pre and post-conditions can be assigned to each state as specifications on artifact fields. In the case that these execution constraints fail, an *Exception* is raised [27].

4. Conclusions

In this paper, a framework for designing Automatic Text Categorization of Web Documents is suggested. An integrated vision for a perfect ATC is proposed, with a discussion of the major four steps that are required to achieve this vision.

We believe that our work is a first step towards defining a complete framework for ATC. There are clearly many problems and research directions concerning the

design of an effective ATC. The proposed step of the ATC process is claimed to be a challenge, in addition to the concerns about privacy of users' data. However, proposing a vision for ATC development clarifies that the research can work for and shares an ambition of acquiring a web that is highly personalized for every user.

5. References

[1] I. Frommholz, "Categorizing Web Documents in Hierarchical Catalogues", Proceedings of the 23rd European Colloquium on Information Retrieval Research, Darmstadt, DE, 2001.

[2] G. Attardi, A. Gull, F. Sebastiani, "Automatic Web Categorization by Link and Context Analysis", Proceedings of THAI-99 1st European Symposium on Telematics Hypermedia and Artificial Intelligence, pp. 105-119, Varese, IT, 1999.

[3] W. Lam, M. E. Ruiz, P. Srinivasan, "Automatic Text Categorization and Its Applications to Text Retrieval", IEEE Transactions on Knowledge and Data Engineering, vol. 11, pp. 865-879, 1999.

[4] W. Lam, K. Y. Lai, "A Meta-learning Approach to Text Categorization", Proceedings of the 24th Annual ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 303-309, Louisiana, USA, 2001.

[5] H. J. Oh, S. H. Myaeng, M. Lee, "A Practical Hypertext Categorization Method Using Links and Incrementally Available Class Information", Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development of Information Retrieval, pp. 264-271, Athens, Greece, 2000.

[6] N. Govert, M. Lalmas, N. Fuhr, "A Probabilistic Description-Oriented Approach for Categorizing Web Documents", Proceedings of the eighth International Conference on Information and Knowledge management, Missouri, USA, 1999

[7] Y. Yang, C. G. Chute, "An Example-Based Mapping Method for Text Categorization and retrieval", ACM Transactions on Information Systems, vol. 12, issue 3, pp. 252-277.

[8] G. Attardi, "Categorization by Context", J.UCS: Journal of Universal Computer Science, vol. 4, issue 9, pp. 719-736, 1998.

[9] O. R. Zaian, M. Antonie, "Classifying Text Documents by Associating terms with Text Categories", Proceedings of the thirteenth Australian Conference on Database Technologies, pp.215-222, Victoria, Australia, 2002.

[10] W. W. Cohen, Y. Singer, "Context-sensitive Learning Methods for Text Categorization", CAN Transactions on Information Systems, vol. 17, issue 2, April 1999.

[11] D. Boley, M. Gini, R. Gross, E. Han, K. Hastings, G. Karypis, V. Kumar, B. Mobasher, J. Moore, "Document Categorization and Query Generation on the World Wide Web Using WebACE", AI Review, vol. 13, issue 5-6, pp. 365-391, 1999.

[12] M. E. Ruiz, P. Srinivasan, "Hierarchical Neural Networks for Text Categorization", Proceedings of SIGIR 22nd ACM International Conference on Research and Development in Information Retrieval, NY, USA, 1999

[13] M. E. Ruiz, P. Srinivasan, "Hierarchical Text Categorization using Neural Networks", Information Retrieval, vol. 5, issue 1, pp. 87-117, 2002

[14] F. Sebastiani, "Machine Learning in Automated Text Categorization", ACM Computing Surveys (CSUR), vol. 34, issue 1, pp. 1-47, NY, USA, 2002

[15] C. C. Aggarwal, S. C. Gates, P. S. Yu, "On the Merits of Building Categorization Systems by Supervised Clustering",

Proceedings of the fifth ACM SIGKODD International Conference on Knowledge Discovery and Data Mining, pp. 352-356, California, USA, 1999
[16] D. Boley, M. Gini, R. Gross, E. Han, K. Hastings, G. Karypis, V. Kumar, B. Mobasher, J. Moore, "Partitioning-based Clustering for Web Document Categorization", Decision Support Systems, vol. 27, issue 3, pp. 329-341, 1999
[17] E. D. Liddy, W. Paik, E. S. Yu, "Text Categorization for Multiple Users Based on Features From a Machine-readable Dictionary", ACM Transaction on Information Systems, vol. 12, issue 3, pp. 278-295, July, 1994
[18] W. j Teahan, "Text Classification and Segmentation Using Minimum Cross-entropy", Proceedings of the 6th International Conference "Recherche d'Information Assistee par Ordinateur", Paris, Fr, 2000
[19] G. Attardi, A. Gull, F. Sebastiani, "Theseus: Categorization b y Context", In poster proceedings on WWW99 – 8th International Conference on The World Wide Web, pp. 136-137, Toronto, CA, 1999
[20] "WebACE: A Web Agent for Document Categorization and Exploration", Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98), pp. 408-415, NY, USA, 1998
[21] S. Dumais, H. Chen, "Hierarchical Classification of Web Documents", Proceedings of the 23rd Annual pp.256-263, Athens, Greece, 2000
[22] A. Pretchner, S. Gauch, "Ontology Based Personalized search", ICTAI, pp.391-398, 1999
[23] L. Gravano, P. G. Ipeirotis, M. Sahami, "QProber: A System for Automatic Classification of Hidden-web Databases", ACM Transactions on Information Systems, vol. 21, issue 1, January, 2003
[24] E. S. Yu, P. C. Koo, E. D. Liddy, "Evolving Intelligent Text-based Agents", Proceedings of the 4th International Conference on Autonomous Agents, pp. 388-395, Barcelona, Spain, 2000
[25] W. Pratt, M. A. Hearst, L. M. Fagan, "A Knowledge-based Approach to Organizing Retrieved Documents", AAAI/IAAI, pp. 80-85, 1999
[26] J. Billington and W. Reisig (eds.), "Application and Theory of Petri Nets", Springer-Verlarg, 1996
[27] M. Fayad, "E-Frame: A Process-based Object-Oriented Framework for E-Commerce", International Conference on Computing (IC 2001), Las Vegas, USA, June 25-28 2001
[28] Y. Labrou, T. Finin, "Yahoo! As an Ontology – Using Yahoo! Categories to Describe Documents", Proceedings of the 8th International Conference on Information and Knowledge Management, Missouri, USA, 1999
[29] Grady Booch, James Rumbaugh, Ivar Jackobson, The Unified Modeling Language User Guide. Addison-Wesley, 1999

A Minimalist Approach to Framework Documentation

Ademar Aguiar and Gabriel David

Faculdade de Engenharia da Universidade do Porto,
Rua Dr. Roberto Frias s/n, 4200-465 Porto, Portugal
{aaguiar,gtd}@fe.up.pt

Abstract. Good quality documentation is an important prerequisite for the effective reuse of object-oriented frameworks. To satisfy the needs of different audiences, framework documentation must integrate several kinds of documents and contents, thus resulting hard, costly and tiresome to produce when not supported by appropriate tools and methods. This research proposes a documenting approach for frameworks that aims to be simple and economic to adopt. It reuses existing documentation styles, techniques and tools and combines them in a way that follows the design principles of minimalist instruction theory. The resulting documents assume the form of *minimalist framework manuals*, a kind of instruction manual that emphasizes the understandability and usability of a framework. In concrete, the minimalist approach proposes a *documentation model*, a *documentation process*, and a *set of tools* built to support the approach—XSDoc[1].

1 Introduction

Object-oriented frameworks are a powerful technique for large-scale reuse. Through design and code reuse, frameworks help developers achieve higher productivity, shorter time to-market, and improved compatibility and consistency [1,2,3].

A framework can be defined as a reusable design of an application together with an implementation [1,4,5,6,7]. As one of the most complex kinds of object-oriented products, frameworks can be particularly hard to understand by first-time users, specially if not accompanied with appropriate documentation [2,8]. Grady Booch clearly stated in [9] that *"the most profoundly elegant framework will never be reused unless the cost of understanding it and then using its abstractions is lower than the programmer's perceived cost of writing them from scratch"*.

This research focus on the problem of producing good quality documentation as a means to improve the understandability and usability of frameworks [10]. The next section briefly overviews this problem and introduces the minimalist instruction theory. Section 3 presents the key concepts of *minimalist framework documentation*, and section 4 outlines the XSDoc infrastructure aimed to support the approach. The final section discusses the results achieved and future work.

2 Motivation and Research Overview

Good quality documentation is a crucial success factor for framework reuse because it helps on the understanding of a new framework, guiding users on the customization process and explaining their design principles and details [2,7]. Among the approaches suggested for documenting frameworks, the cookbook approach [11], the patterns approach [12] and the meta-patterns approach [13] have proven to be effective in reducing the typical long learning curve.

[1] XSDoc pronounces "Extensible Soft Doc"

2.1 The Problem of Documenting Frameworks

To define and write good quality documentation for a framework is not easy, quick or pleasant to do. It is, at least, an order of magnitude more difficult than documenting object-oriented applications or class libraries, because it must cover not only a single concrete product (an application) but, instead, a tool to produce a family of many similar concrete products (a framework). To be complete, the documentation of a framework must describe the application domain covered by the framework, its purpose, how-to-use it, how it works, and details about its internal design, what globally may involve a large diversity of contents and many different ways of presenting them [14]. This inherent complexity results from the following reasons:

- **different audiences** use frameworks in different ways, each with their own documentation requirements: framework selectors, application developers, framework developers and developers of other frameworks;
- **different styles of documents** are used in framework documentation to provide multiple views (static, dynamic, external, internal) at different levels of abstraction (architecture, design, implementation): framework overviews, example applications, cookbooks and recipes, design patterns, use cases, contracts, design notebooks and reference manuals [14];
- **different notations** are needed to represent different kinds of contents: free text, structured text, source code, object models, images, formal specifications, etc.

2.2 Minimalist Instruction Theory

Minimalist documentation is based on the theory of minimalist instruction [15], a theory with foundations in the psychology of learning and problem solving. The minimalist instruction intends to help on the design of instruction material, so that people can learn faster and for longer. The key idea in the minimalist instruction is to minimize the obtrusiveness to the learner of training material, hence the term.

Learners in general don't seem to appreciate to read overviews, reviews and previews of training material. Instead of reading, people seem to be more interested in action, in working on real tasks and in doing their work. To overcome these and other obstacles to learning, minimalist instruction theory is based on *three values*: *more to do*, to allow learners to start immediately training on meaningful realistic tasks; *less to read*, to reduce the amount of reading and other passive activity in training; and *help with errors*, to help making error recognition and recovery less traumatic, more pedagogical and more productive.

As a design theory, the minimalist instruction doesn't prescribe ways of producing minimalist manuals, but instead it defines a set of *design principles*: to motivate people to *train on real tasks* and *get-started fast*; to present topics very briefly in the *order that seems best for the reader*; to support *error recognition and recovery*; and to *try to explore readers' prior knowledge.*

2.3 Research Goals

Much of the work on framework documentation has focused more on finding ways of documenting the design and architecture of frameworks [8] rather than on exploring effective ways of describing the purpose and intended use of frameworks.

Despite the research done, there are still open issues related with framework documentation [14,16], namely: the definition of suitable methods and tools for an effective and economic production of framework documentation; and the exploration of effective ways of describing the purpose and intended use of a framework [8]. This last issue is precisely where minimalist documentation can be helpful, ie, on improving the understandability and usability of frameworks.

The main goal of this work is *to define a flexible approach* capable of reducing the costs, typically high, associated with the production of high-quality framework documentation. A secondary goal is *to evaluate the impact of minimalist documentation* on the understandability and usability of frameworks.

3 Minimalist Framework Documentation

The minimalist approach to framework documentation integrates reuses typical document styles and techniques of framework documentation and combines them in a way that follows the design principles of minimalist instruction theory.

The resulting documentation assumes the form of a *minimalist framework manual*, including information about the application domain, its purpose, how-to-use it, how it works, and internal design details. Typically, minimalist manuals are considered easy to read and understand, thereby contributing for shorter learning curves on how-to-use a system, and leading to a better understanding of the systems being trained [15,17,18].

Among the experiments reported in the literature that used minimalist instruction for documenting frameworks are [18,19].

3.1 Requirements for the Approach

The best mix of document types, writing techniques and presentation styles strongly depends on the specific objectives, context and economics of the project at hands, and even on the psychological and technical characteristics of the team elements. A good approach for documenting frameworks should be able to satisfy a diversity of requirements, with a special emphasis on the following:

- **easy-to-use** by developers, so that the activity of documentation can be a means to improve development productivity and quality, instead of being considered an obstacle, as happens in many development environments;
- **flexible** enough to be easily adaptable to the needs of different projects and development environments;
- **capable of cross-referencing** different kinds of contents using simple linking mechanisms, easy to learn and use;
- **economical**, to reduce the typical high-costs associated with the production of good quality documentation.

To fulfil all these requirements, it is important to have the support of a kind of integrated content management covering the overall process of framework documentation from the initial phases of creation and integration of contents till the last phases of publishing and presentation to target audiences.

The approach informally provides guidance about *what*, *when*, and *how* to document. It consists on a *a documentation model*, a *generic documentation process* and a *set of tools* to make it convenient to use in mainstream development environments.

3.2 Documentation Model

The documentation model is the core of the approach. It enumerates and organizes all the contents and interdependencies (part-of hierarchies, navigational links, and derivations) required to produce a minimalist framework manual. According to its nature, the overall contents can be divided in two main categories:

- **typical of framework documentation**: code examples, recipes and cookbooks, design patterns, framework overviews, reference manuals, design notebooks, use cases, scenarios and contracts;
- **typical of minimalist manuals**: user tasks, usage patterns, task information contexts, error inventories, error recovery guidelines, and classification of contents according to the usage mode (training-wheels, guided-exploration or free exploration).

To be useful, all this web of contents must be presented in an appropriate manner, so that users don't become overwhelmed or lost when using the documentation. Therefore, the overall repository of contents is organized as a virtual *n–dimensional documentation space* defined by *n documentation aspects*. Important examples of documentation aspects are: kind of audience, level of abstraction and level of information granularity. This space can then be divided in *documentation layers* according to the relevance of a specific content for a certain documentation aspect. The distribution of contents along the layers is configurable and is supported by meta-information, either manually annotated or automatically synthesized from the contents.

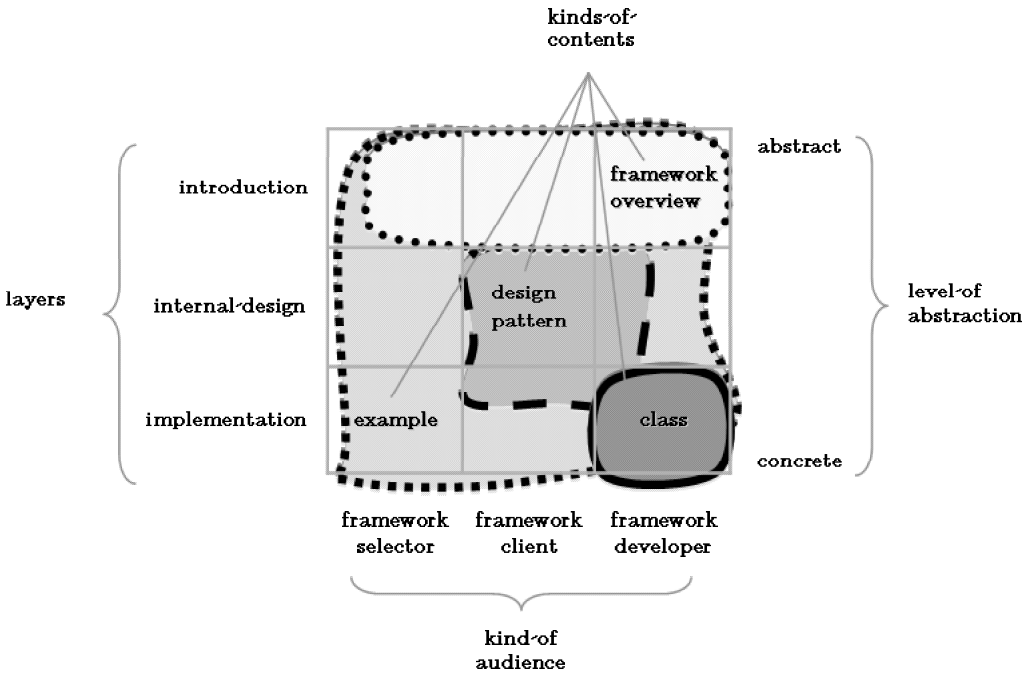


Fig. 1. Example of a configuration of layers.

To illustrate the layered documentation space, it is represented in Fig. 1 a simple configuration of layers for a two-dimensional documentation space defined by the aspects *abstraction level* (abstract, concrete) and *kind of audience* (framework selector,

framework client and framework developer). This configuration defines three layers, named *introduction*, *internal design* and *implementation*, and considers four kinds of contents, *framework overview*, *design pattern*, *example* and *class*. This configuration tells us that: a framework overview contains information relevant to the introduction layer and all kinds of audience; an example contains information relevant to all layers and all kinds of audience; a design pattern contains information relevant for both the internal design and implementation layers, and for framework clients and framework developers; and a class only contains information relevant for the implementation layer and framework developers.

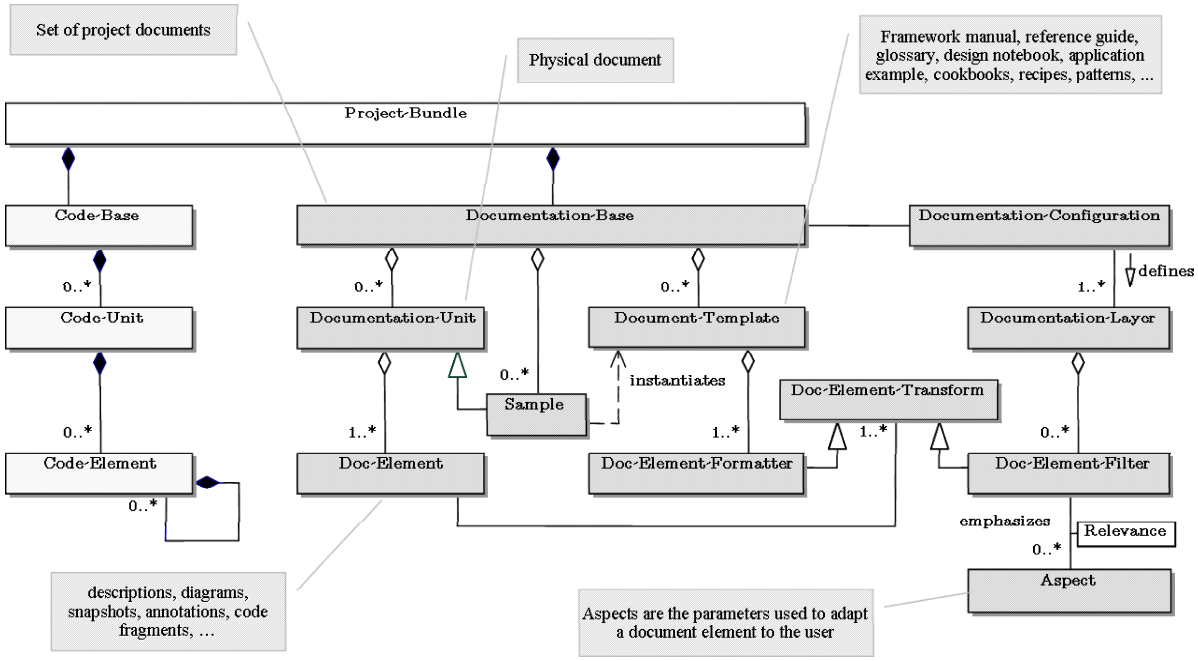


Fig. 2. An overview of the key concepts used in the documentation model.

The key concepts of the model are represented in Fig. 2. Starting from the top concept, we see that the *project bundle* is composed by all the physical *units* of both the *code base* and the *document base*. A *code unit* can be a source code file or an object code file, and contains several *code elements*, such as packages, classes, methods, or fields. On the other hand, a *document unit* can be a file, a database or a repository, and contains several *document elements*, ie, pieces of documentation possibly represented in different notations, such as texts, models, annotations, etc. In addition, there are predefined *document templates* and *sample instances* of these templates, one for each kind of style of document to use, such as use-case template, pattern template, framework overview template, cookbook template, etc. Finally, the model contains also a *documentation configuration* that is used to define the *documentation layers* and the way elements are supposed to be filtered, transformed and formatted according to their relevance to each *aspect* and the layer in focus.

In order to cope with a vast diversity of documentation requirements, the implementation of the model should be extensible so that new custom styles of documents, notations, and layers can be added with a reasonable effort.

3.3 Documentation Process

The documentation process defines the roles, techniques, and activities involved in the production of the minimalist framework manuals. The roles identified are:

- **developers**, such as framework users, framework developers, and framework maintainers, which are responsible for content creation mostly during the development phase;
- **technical writers**, which are responsible to structure, guide, review and conclude the documentation;
- **documentation managers**, which are responsible for configuring and maintaining the documentation base, namely the template documents, template instances, and the filtering, transformation and formatting of documents according to the layers configured.

The production of framework documentation is closely related with the framework design and usage, so, ideally, these activities should be done side by-side, if we want to obtain documentation that is understandable, consistent, and easy-to-maintain.

After a configuration phase, the production of framework documentation starts with the *creation* of the various kinds of contents, and their cross-referencing. Upon creation, the different kinds of contents are normalized, integrated and stored in a repository from where they will be retrieved, transformed, published and presented to target audiences.

This documentation process is generic and was designed considering lightweight processes, which typically allocate very little effort for documentation, thus being very restrictive on adopting a documentation approach. Therefore, the resulting process is simple, flexible and easy to adapt to different development processes and environments, ranging from literate programming environments [20,21] to industrial integrated development environments. In the next section are presented the set of tools currently being prototyped to support this minimalist approach.

4 The XSDoc documentation infrastructure

The XSDoc is an infrastructure based on XML [22] and WikiWikiWeb [23] specially designed to support the production and usage of minimalist framework manuals, and covers all the typical functionalities of a content management system. Currently, XSDoc only supports frameworks written in Java programming language, models described in UML [24], and the integration with the Eclipse IDE.

The XSDoc infrastructure is composed by one Wiki engine (XSDocWiki), a plugin for integration in the Eclipse IDE (XSDocPlugin4Eclipse), and a set of document templates, markup languages, and converters of contents to and from XML. The Fig. 3 illustrates these components as well as their interconnections.

4.1 XSDocWiki

A WikiWikiWeb, or simply a Wiki, is a very innovative and appealing collaboration tool. It can be defined as a web platform for the cooperative edition of documents, where everyone can edit any page, using a simple web browser and invoking the "Edit" option on the top, or bottom, of that page [23]. After saving, the modifications done will be uploaded immediately and made available online.

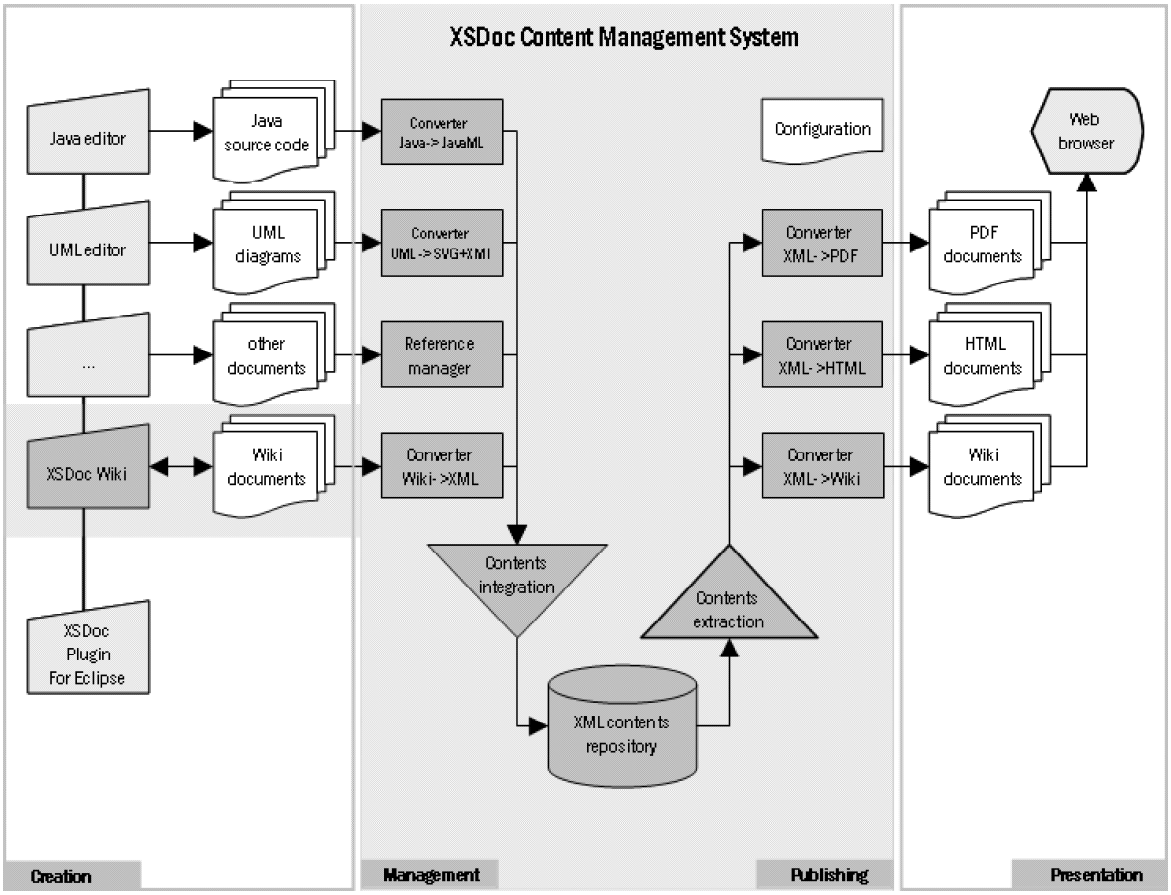


Fig. 3. XSDoc components and their interconnections.

A Wiki uses a very simple markup language to support simple text formatting and a mechanism for automatic linking based on WikiNames[2]. Despite its simplicity the mechanism is very powerful because it works like a late-linking mechanism, thus enabling the dynamic change of the targets. In addition, other kinds of linking mechanisms can be defined using lexical conventions, such as prefixes, suffixes, and name patterns, in general.

The XSDocWiki engine is the main component of the XSDoc infrastructure. It was developed using the VeryQuickWiki engine [25] as a starting base and then extended with several features to support the edition and visualization of minimalist framework manuals, namely the support for processing Java sources, UML diagrams, XML documents, version control systems, a plugin mechanism for adding new types of documents, and a few minimalist controls. The resulting Wiki supports not only the automatic linking between Wiki pages, but also between informal documents, structured documents, source code programs in Java language, and models in UML, using predefined naming conventions that are very easy to learn and use.

With the plugin mechanism, the support for each new style of document (use-case, example, cookbook, pattern, etc) can be added on the fly. A XSDoc plugin includes: a document-template; a set of converters to map that style of documents to and from XML, which can be written in a scripting language (XSL, Javascript, Python, etc.); a

[2] WikiName is a Wiki name because it JoinsCapitalizedWords, which is AnotherWikiName

declaration of which elements may contain Wiki text, so that they can be analyzed and their links connected; and some lexical rules to use during the automatic linking phase. For example, for Java source files, it is declared that javadoc elements may contain Wiki text, thus enabling in javadoc comments the usage of Wiki links to any Wiki page, be it another Java source file, an UML file or a document file, structured or not.

So configured, the XSDocWiki promotes the cooperation of technical and non-technical people on an incremental edition and revision of software documents, ensuring an high availability of contents (always online), and only requires a simple web browser, a tool currently very easy to integrate in a vast majority of development environments.

4.2 XML Converters and Presentation Processors

As most of the contents can be comfortably edited and linked using the Wiki, most of the documentation contents will reside on Wiki pages stored in a file system, a version control system, or a database.

However, Java source code programs and UML diagrams need special processing as they must be converted from their original format to XML using XSL transformers [26], respectively using JavaML [27], SVG [28] and XMI [29] vocabularies.

Before being published and presented, the contents must then be filtered and formatted accordingly. XSDoc is able to output HTML files for online browsing, and PDF files for high-quality printing.

4.3 Integration Mechanisms

The components of XSDoc are closely integrated, both in terms of functionalities and in terms of the information they exchange. The functional integration of the Wiki with the converters and processors is done within the Wiki and its specific extensions.

In terms of the information exchanged between the tools, the integration is achieved through the use of text files and XML files. A markup language (XSDocML[3]) is also used internally to normalize all the contents in a unique schema, when necessary.

One of the goals of the minimalist approach is its seamlessly integration in contemporary development environments. We think that the combined use of XML and Wiki makes this integration successful in almost every industrial development environment with the cost of development of small configurations, considering that XML is widely supported everywhere and the Wiki engine only needs a browser to run.

The integration of the XSDoc infrastructure with IDE's is achieved through the development of specific plugins. The plugin should enable the use of a web browser through which the XSDocWiki can be accessed, and to provide a communication link between the IDE and the XSDocWiki, to enable their interoperation.

Much tighter integration of the XSDoc infrastructure in a development environment can be done with recent IDE's, such as Borland's Together or IBM's Eclipse, which enable in the same environment a synchronized edition of all kinds of contents: source code, UML models, and XSDocWiki documents.

[3] XSDoc Markup Language

4.4 Benefits of XSDoc

During development, it is typical to switch between the tasks of developing code (edit-compile-test), the task of browsing documentation, and the task of writing documentation, if done.

Present IDE's already integrate in a same environment the tasks of developing code, and some of them also enable the browsing of documentation inside the IDE. However, the support for writing documentation inside an IDE is usually very small, consisting only on enabling the writing of documentation in the same files as the source code, using Javadoc comments for example, or specific forms to introduce the Javadoc comments.

But when we need to write documentation at a higher level of abstraction than the source code files enable, such as documenting a pattern instantiation, or describing an architecture, we need to jump out of the IDE and edit the file independently. Worst, if we need to cross-reference the contents of this pattern instantiation document with some source code elements, be it a class, a method or a field, we need to copy-paste the contents from one file to the other (the most usual and easier) or instead define a static cross-reference between the contents. In any case, sooner or later, the document and the source code will become incoherent, because code changes very fast.

The most economic alternative to avoid incoherence between documentation and code is not to write documentation during development, but only at the end; another alternative is to use a literate programming philosophy, but in this case we need to move out from the most popular IDE's and leave their powerful features that help us improve development productivity. XSDoc is another alternative to solve this problem.

With XSDoc integrated in an IDE, the developer have access to a web browser from where he can use the XSDocWiki. When documenting, the developer creates new pages, writes documents, possibly using predefined templates, uses copy-paste and drag-and-drop IDE features, browses the resources, both documents and source code, and defines links to other pages or special contents, like Java source code or UML diagrams, using predefined tags and linking mechanisms.

As an example, to document the instantiation of the `Command` pattern by the class `TestCase` requires the writing of the text shown in Fig.4(a), which produces the result represented in Fig.4(b). Any change on the code will be automatically reflected in the documentation, when the page is refreshed by the browser.

5 Conclusions

Good quality framework documentation helps users to understand the purpose of a framework, to learn how to customize it, and to learn its internal design details. A lot of work already exists on ways of documenting the design and architecture of frameworks, but there are still open issues.

Inspired by the minimalist instruction theory, this research proposes a new approach (model, process and tools) to produce minimalist framework manuals. Simplicity, low cost, and easy-to-use by all the elements of the development team, specially the programmers, are some of the intended qualities of such approach. To make convenient the practical adoption of the approach, a set of tools called XSDoc infrastructure is provided, combining a Wiki engine, document processing with XML technology, and integration in a popular IDE.

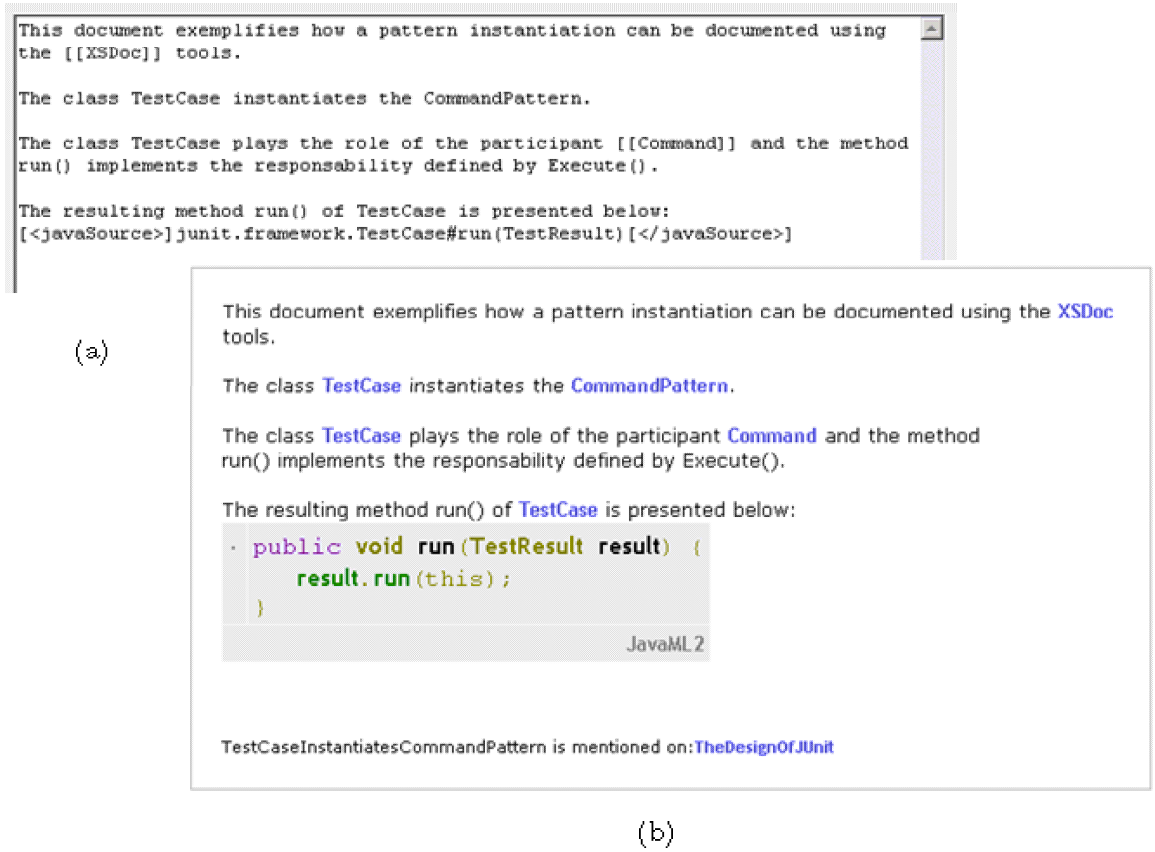


Fig. 4. Documenting the application of the Command pattern to the class TestCase: (a) text written (b) output obtained.

From the resulting work, it can be concluded that the use of the XSDoc infrastructure integrated in an IDE such as Eclipse, can significantly reduce the effort typically needed to document a framework, as it can combine the simplicity, easiness and versatility of the collaborative document edition in the Wiki, with the well-known qualities of XML technology in terms of integration, processing and presentation of information.

Obviously, a Wiki engine adapted to the edition of XML documents can be considered by many as something more difficult to use than a typical Wiki, or a Wiki can be considered a poor XML editor when compared to a typical XML editor. Anyway, the combination of both technologies result in a very attractive infrastructure, whose best qualities can be summarized as: easy to integrate in a framework development environment; easy to use by any element of framework project team (technical or not); promotes the participation of all the team elements in the documentation process; improves the communication between the team elements; provides an easy access, revision and incremental evolution of the documentation; and finally, enables a smooth integration of contents in a controlled and structured way, informally, while preserving the information in an universal format, the XML format.

In future work, the XSDoc tools will be improved with more minimalist features (zoom, exploration mode, error recovery, extensive search) and the minimalist manuals for JUnit and JHotDraw will be concluded and their impact in terms of usability and understandability will be evaluated in comparison with the original framework docu-

mentation. Then, new plugins for integration with other popular IDE's will be developed and other popular Wiki engines will be supported.

References

1. Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Building Application Frameworks — Object-Oriented Foundations of Framework Design.* John Wiley & Sons, 1999.
2. Taligent Press. *Building Object-Oriented Frameworks.* Addison-Wesley, 1994.
3. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of reusable object-oriented software.* Addison-Wesley, 1995.
4. Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.
5. R. Campbell, N. Islam, R. Johnson, P. Kougiouris, and P. Madany. Choices: Framework and refinement, 1991.
6. Ted Lewis, Glenn Andert, Paul Calder, Erich Gamma, Wolfgang Pree, Larry Rosenstein, Kurt Schmucker, André Weigand, and John M. Vlissides. *Object-Oriented Application Frameworks.* Manning Publications Co. / Prentice-Hall, 1995.
7. Mohamed E. Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.
8. Greg Butler and Pierre Denommée. Documenting frameworks. In *Building Application Frameworks — Object-Oriented Foundations of Framework Design* [1], pages 495–504.
9. Grady Booch. Designing an application framework. *Dr. Dobb's Journal*, 19(2), February 1994.
10. Ademar Aguiar. A minimalist approach to framework documentation. In *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 143–144. ACM Press, 2000.
11. Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):27–49, September 1988.
12. Ralph Johnson. Documenting frameworks using patterns. In Andreas Paepcke, editor, *OOPSLA'92 Conference Proceedings*, pages 63–76. ACM Press, October 1992.
13. Wolfgang Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley / ACM Press, 1995.
14. Greg Butler, Rudolf K. Keller, and Hafedh Mili. A framework for framework documentation. http://www.cs.concordia.ca/faculty/gregb, 1998.
15. John M. Carroll. *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill.* MIT Press, 1990.
16. Mohamed E. Fayad. Future trends. In *Building Application Frameworks — Object-Oriented Foundations of Framework Design* [1], pages 617–619.
17. John M. Carroll. *Minimalism Beyond The Nurnberg Funnel.* MIT Press, 1998.
18. Mary Beth Rosson, John M. Carrol, and Rachel K. E. Bellamy. Smalltalk scaffolding: a case study of minimalist instruction. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 423–430. ACM Press, 1990.
19. Ian Chai. *Pedagogical Framework Documentation: How to Document Object-oriented Frameworks - An Empirical Study.* PhD thesis, University of Urbana Champaign, 1999.
20. Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
21. Ross N. Williams. *FunnelWeb User's Manual*, May 1992. v1.0 for FunnnelWeb v3.0, ftp://ftp.adelaide.edu.au/pub/funnelweb.
22. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml) 1.0. w3c recommendation, February 1998. URL:http://www.w3.org/TR/REC-xml.
23. Ward Cunningham. The original wiki front page., 1999. URL:http://c2.com/cgi/wiki.
24. Object Management Group. Unified modeling language specification version 1.4, September 2001. URL:http://www.omg.org/technology/documents/formal/uml.htm.
25. Gareth Cronin and Bill Barnett. Very quick wiki engine homepage. URL:http://veryquickwiki.sourceforge.net/.
26. S. Deach. Extensible stylesheet language (xsl) specification, w3c working draft, January 2000. URL:http://www.w3.org/TR/WD-xsl.
27. Greg J. Badros. JavaML: a markup language for Java source code. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):159–177, 2000.
28. Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. Scalable vector graphics (svg) 1.1 specification, w3c recommendation, January 2003. URL:http://www.w3.org/TR/SVG11.
29. IBM AlphaWorks. Xml metadata interchange (xmi) toolkit, 1999. URL:http://www.alphaworks.ibm.com/tech/xmitoolkit.

Using an ADL to Design Aspect Oriented Systems[1]

A.Navasa, M.A.Pérez, J.M. Murillo
Quercus Software Engineering Group. http://quercusseg.unex.es
University of Extremadura. Spain
{amparonm@unex.es, toledano@unex.es, juanmamu@unex.es}

Abstract

Developing aspect oriented software systems is a complex task. To do it easy, several paradigms are been developed as AOP. Moreover, in the last years AOSD has been generating a great deal of interest to develop AO Systems from early stages. To extract the aspect code crosscutting the functional one makes easier to implement aspect oriented systems using AOP languages, but it is possible to take away the problem of AO from the implementation phase to the design. In this phase, concerns crosscutting functional code can be treated as independent entities. This paper presents our ongoing research developing aspect oriented systems taking into account the benefits of applying CBSE at early stages of AO systems development, particularly at architectural design. We present AOSD at the design level as a co-ordination problem, using an ADL to formalise it

1. Introduction.

Until now, a great deal of work has be carried out proposing aspect separation techniques [Kic+96] to effectively separate the aspect code from the functional code (composition filters [BeAk01], AspectJ [Kic+01], Adaptive programming [LiOrOv01], HiperJ [OsTa01a], [OsTa01b]). During the last few years, concepts from AOP have been extended to the early stages in the software lifecycle, creating Aspect Oriented Software Development (AOSD) as a new discipline of software engineering, which has been generating a great deal of interest for developing systems from the early stages perspective. Its goal is to provide approaches for an early identification, separation, representation and composition of crosscutting concerns [BrMo2003]. From this point of view, aspects are constructions (software artefacts) that can be identified and manipulated throughout the development process. There is a great deal of research at the design phase [ClWa01, SuYa99] and at the requirements phase [Gru99, Ra+02]. In addition, research about aspect separation at platform definition and middleware level can also be included [ClSaPe02, ClHe03]. Work results have shown real benefits by increasing systems productivity, re-usability and adaptability.

It is possible to observe software architecture from an aspect-oriented point of view. For instance, Grundy in [Gru00] considers that at the design level, "aspects specify the provided and required capabilities of components and allow them to specify functional and non-functional characteristics of these capabilities"; therefore, components and aspect can be developed "to produce design-level system components and design-level aspects". However, none of the consulted works consider architectural tools to formalise them through the architectural styles definition, through studying Domain Specific Software Architecture (DSSA), or through defining Architectural Definition Languages.

Given the system specification by the high-level functionality description and a set of aspect policies to be applied, it is necessary to specify the interaction among components to obtain the systems architectural definition. Interconnection and interaction specification can not be trivial, and it is necessary to focus special attention on this point.

However, open systems can be designed to allow us to modify their behaviour due to the application of aspect policies not considered before. In this paper, some ideas about how to support systems evolution at the architectural level using aspect-orientation are presented. From an architecture design perspective, taking into account aspect-orientation techniques, systems accepting modifications to component behaviour could be designed without changing the components. Dynamically adding or eliminating restrictions over component execution

[1] This work has been developed with the support of CICYT under contract TIC2002-04309-C02-01

can cause modifications on system behaviour. These execution restrictions are due to aspect policies and can (perhaps must) be identified at the time of system specification and separately from functional requirements. This will improve the understanding, identification and management of crosscutting concerns at the architectural level. Besides, a better understanding and description of aspects can support the reuse and development of software from the early stages. For this, it is necessary first to identify and describe them in a correct way.

Our aim is to design Aspect Oriented systems, which allow us to modify their component behaviour by dynamically adding restrictions to components performance. We consider that aspect separation at the architectural level can be solved as a co-ordination problem.

In section 2, three pillars on which the current research is based will be presented. In section 3, a dynamic ADL is used to describe aspect-oriented systems at the architectural level. Finally, section 4 shows some conclusions and outlines future works.

2. The Three Cornerstones.

This section refers to the cornerstones on which we base the work here presented; then some reflections about considering aspects at design phase are made. Finally, how to define the interaction between functional and non-functional components is presented.

Because the systems to be designed are complex ones, we consider that our ongoing research is based on three points. The following three disciplines of software engineering give us a base on which to build aspect oriented systems from components at the architectural design. The general idea behind using them together is to give the designer the possibility to build complex systems in a structured way from a set of components. Before, it is necessary that, components come into a formal description, from the specification phase. Bellow are the advantages to doing so.

- CBSE is a discipline that produces systems by the composition of pre-existing, reusable and independent pieces of software, the components, through *plug and play*. It reduces the cost of development, improving the final system's flexibility and reusability. Besides, using components requires a standard previous definition of them to make possible their composition at design level for obtaining a structural system definition. In order to do this, it is necessary for each one to have defined one or more interfaces showing required services (and needed too), *precond* and *postcond* [Sou00], and any other interoperability information needed to obtain and to combine components. After that, interaction can be established. Our work group considers that, using the ideas of CBSE, it is possible to decompose complex systems into a set of components, and use them like *lego* pieces. This lets us add, remove, or modify components easily.
- On the other hand, AOSD is a discipline that provides support for separating concerns crosscutting functional code into non-functional components, and weaving them later. AOSD allows us to treat concerns in a separated way from functional components, it being possible to identify and manipulate them throughout the development process. We distinguish between functional components for defining the system's functional behaviour and aspectual ones for defining its non-functional one.
- The third cornerstone supporting our work is software architecture (SA). This discipline allows the software designer to specify the systems structure in terms of components and connectors, which determine their interaction. But defining interactions is not a trivial question in this new scenario: it is easy to define interaction between functional

components if services required by one of them matches with services offered by the second one (even if they don't match completely). A more difficult question is to achieve interconnection between functional components and aspectual ones. Nevertheless, it is necessary to create explicit relationships between functional and aspectual components.

To formally build the architecture of systems, software architecture mechanisms need to be used (architectural style definitions, ADL, DSSA-domain specific software architecture-, etc.). Some previous work of our research group has revealed that defining an AO system through an adequate ADL is the better way to do it.

At the architectural level, systems elements are components and their interconnections and aspects need to be considered as first class entities (that is, as components). In this lifecycle phase, aspect separation will be done by defining functional and aspectual components and their interaction.

2.1. Defining Aspects at Architecture Design Phase.

Looking at some literature (examples are [Gru00, BrMo03]), several aspect definitions can be found. Generally speaking, we can define an aspect as a set of operations or procedures carrying out a policy or strategy to be applied to functional components and it is orthogonal to them. After applying an aspect to a functional component, its behaviour can be modified. Taking into account this definition, and from the above (CBSE characteristics, AOSD paradigm, SA definition), we can say that, at the architectural level, it is possible to enclose aspects into a special kind of component, named by us as a "non-functional" or "aspectual" component. Thus, an aspect can be treated as a design artefact whose interaction with other components needs to be defined. The following activities are necessary:

First of all, before an aspect can be considered as a component, it is necessary to identify, from the specification or architectural design phases, which aspects can be applied to a system. For example, at the design level, authentication or replication can be considered as aspects, and those can be applied or not throughout the system life, without basic functionality being concerned.

Secondly, to define aspects adequately, it will be necessary to specify them giving their offered and required interfaces, as well as the preconditions and post-conditions for applying them.

Finally, it is possible to define the interaction between aspects and the functional components whose behaviour can be modified. If several aspects can be applied, composition rules must be defined.

Identifying and enclosing aspects at the design phase lets us do software complexity management better. We can manage the crosscutting concerns in functional components by defining over its interfaces the match points where aspect must be applied.

2.2. Interconnecting Functional and Non-functional Components at Software Architecture Level.

At the architectural level, we represent the system structure by components and the interaction between them. The interaction will be done by a communication mechanism: the architectonic connectors. This kind of communication mechanism, in general, isn't trivial because of the high complexity of interactions to be managed [Na+02].

It is known that to make components composition it is necessary that they be specified by their interfaces. This allows for making possible defining anchor points between them. As we know, at thedesign level, one only knows component interfaces. Besides, connectors are the architectonic mechanisms of communication between them.

We consider architectonic connectors as first class entities. Each one specifies how and when an aspect intercepts the normal execution of components connected with it. For this we say that connectors, defining architectonic interaction between functional and aspectual components, must co-ordinate both executions, and they are as co-ordinators of the system execution. This point is important because, until now, components making the system are not specially created to be co-ordinated.

We can say that, at the architectural level, applying aspects to a system can be treated as a co-ordination problem, in which co-ordinated elements do not necessarily know each other, nor have they been created to be co-ordinated. Then, to solve the aspect oriented system design problem we could apply co-ordination exogenous models [Mur01].

To formally define the SA of a system taking into account aspects, we need to use those tools that SA gives us (ADL, Styles, DSSA,…). Our ongoing research is focused on applying a particular ADL to formalise AO systems design.

3. Selecting an Adequate ADL.

Until now several ADL have been developed, each one having different features (Wright [All97], Darwin [MaKr96], Rapide [Luc95], Acme [GaMoWi97]). We only have considered dynamics ADL (Rapide [Luc95], Acme [GaMoWi97]) because our group work wants to promote systems design able to change dynamically, depending on if some aspect policies are applied or not. The problem is that actual dynamics ADL ([Luc95], [GaMoWi97]) are general and do not have primitives to support aspects separation characteristics.

Rapide [Luc95]: This is an event-oriented language that allows one to specify systems in terms of a partially ordered set of events. It allows architectural design to be simulated, and has tools for analysing the result of those simulations. Components computations are triggered by received events, and in turn trigger other computations by sending events to other components. Rapide toolset permits the simulation of such descriptions, animations of those simulations, and analysis of the resulting trace graphs to check for anomalous behaviour. It supports dynamic analysis using simulation technology and dynamic reconfiguration of architectures. It is a good option to represent AO systems, but the latest research on Rapide was published in 1998; hardware platforms over which Rapide is defined are obsolete and working with it implies bringing all the libraries up to date and some of them are not accessible.

Acme [GaMoWi97]: This provides a structural framework for characterising architectures. It provides an interchange format for architectural development tools and environments. It allows us represent events. Communication is established by connectors and co-ordination between components becomes complex. It is possible to consider it as a useful ADL for our research.

LEDA [Can00] is a dynamic ADL, whose main characteristics are:

- It is a dynamic language.
- It doesn't distinguish between components and connectors. In it there are only components and all are components. Components are system elements, each one expressing a part of

the system functionality. Connectors are represented by first class entities, being a special kind of LEDA components.

- It is hardly based on formal concepts because it is defined on π-calculus. For this, system specifications given in LEDA can be analysed and performed to get a prototype.
- *Adapters* are a mechanism to establish the correct communication between components with non-compatible interfaces. This makes it possible to connect components whose behaviour is not compatible between them.
- Components' interconnection is done by *roles* definition, which communicates language components. Observable component behaviour is described by *roles*.
- Components and connectors can be represented by a particular LEDA graphic representation.

From the above, we can say that LEDA is an adequate language to represent AO Systems at design level. Not to distinguish between components and connectors is the main reason for which we decided to use LEDA. Table below (Table 1) expresses this idea matching some LEDA characteristics with the advantages to use it in AOSD:

LEDA characteristics	Advantages to use it in AOSD
It is a dynamic language	This lets us define a system, changing its behaviour by adding/eliminating restrictions at design time.
It doesn't distinguish between components and connectors. Then connectors become represented by first class entities (as components).	Defining the system connectors as LEDA components lets us consider complex interactions between systems components.
System components and connections communication are made in LEDA by the *role* definition	Roles describe the observable component behaviour.
Components and connectors can be represented by a particular LEDA graphic representation.	For this, it is easy to understand the system structure.
System properties can be validated.	This lets us check the system correctness each time a new restriction is added.
A tool lets us generate Java code, and to simulate system behaviour at the design time.	This helps the designer and reduces the implementation time.
LEDA is a general ADL	We suspect that it will be necessary to define new primitives to be able consider aspect.

Table 1. Characteristics of LEDA and advantages to use it in AOSD.

3.1. Applying LEDA to Solve the Problem of Designing AO Systems at Architectural Design Phase.

Solving the problem of designing an AO System at the architectural design phase requires the steps numbered in the first column in Table 2. The second column points out steps to give using LEDA.

To capture formal specifications of system. We distinguish between functional and aspectual specifications.	
To obtain components from specifications. Functional and aspectual components will be searched in repositories (if possible) (1).	Specifications of components found will be translated to LEDA .
Interconnection between functional components has to be defined.	To support interconnection between functional components, LEDA *roles* have to be defined.
To define how to apply aspects to functional components.	To define LEDA *co-ordinator* components making it possible to apply aspects to functional components co-ordinately.
	To define new *roles* to be able to interconnect functional and aspectual components through *co-ordinators* is necessary.
To generate a prototype, in an adequate language, for the new system.	A LEDA tool lets us generate a Java prototype for the new system. We have to: - Generate, to each *co-ordinator* component the associated classes. - Generate associated classes to define *roles*.
	Due to (1) functional and aspectual, components are : - Only a set of specifications. For this it will be necessary to generate the Java classes from LEDA description or - Components found in repositories. Then they can be executed together with the new ones.
To simulate system behaviour at design time.	After obtaining Java code to simulate system behaviour at design time is possible.

Table 2. Steps to solve the problem of designing AO System at architectural design phase using LEDA.

After designing an AO system with LEDA the following considerations can be done:

- Composition between *co-ordinators* and co-ordinated components must be and in fact are dynamic to make possible the application of new aspects to a system.

- Changing the system (by applying new aspects or eliminating them), doesn't mean modifying original components behaviour, nor aspectual components. Only the LEDA system definition has to be modified, adding new roles definition and then generating a new prototype.

3.2. An example.

In this section an example is presented. We describe a case in a normal situation and what happens if a new aspect policy is added. We present how to solve the problem, and how we

could represent it by LEDA language. This representation is partial because defining new aspectual primitives is necessary.

Consider a bank account and a client acceding to it (Figure 1). The Client requests a bank operation from the ServerBank. The ServerBank answers the Client by giving the requested service.

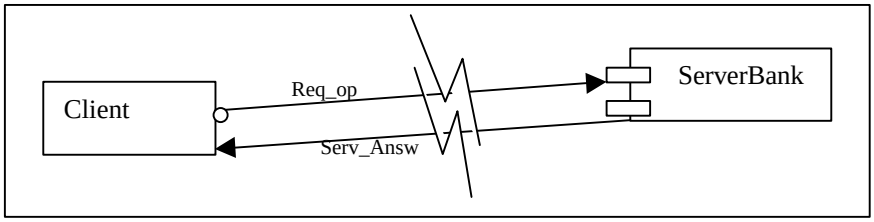


Figure 1. Bank account example.

If one wishes to add a new aspect policy at the ServerBank, the designer has to modify the ServerBank component, including the new operation.

Our ongoing research proposes adding the new aspect without changing the ServerBank component, but adding it to the system as a new component. This will be executed under some restrictions before or after ServerBank performs the requested service. Such a component (aspectual component) and ServerBank component must be executed in a co-ordinated way. We propose including, the aspectual component carrying out the new policy on the one hand, and on the other hand, a co-ordinator component, which can determine if conditions to execute the new task are given (Figure 2).

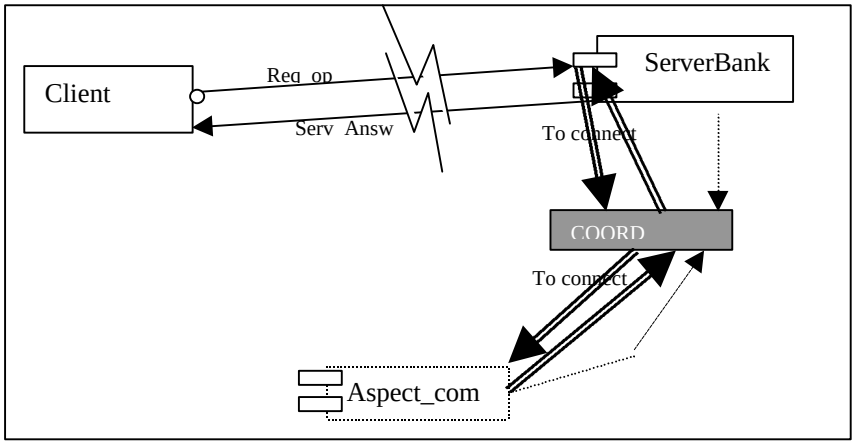


Figure 2. Bank account with an aspect example.

The figure above represents how adding the new policy to the system without changing the components which form the system. We have added the new aspect policy as a component (Aspect_com). Besides we define a new component COORD, which makes possible the interaction between ServerBank and the added policy (Aspect_com). COORD checks if conditions or restrictions to apply the policy are done for the ServerBank component. As it is the design time, we only have access to functional components by their interfaces.

It is possible to represent the above example in LEDA language as in Figure 3. Components are given by rectangles (with shadow representing several ones). *Roles* are represented by lines connecting them, with a label giving the interaction name.

In the problem representation by LEDA, the system component ServerBank and the aspect (Aspect_com) are connected by a connector (a first class entity) defined as a LEDA

component (called *co-ordinator*) represented by the COORD component. *Co-ordinator* COORD performs the *connect role* defined between it and the system components ServerBank and Aspect_com. It also performs the *co-ordinate role* to co-ordinate their execution.

Defining the *co-ordinator* as a first class entity to interconnect aspect components to the other ones in the system is necessary because some preconditions and postconditions need to be analysed for applying an aspect to the system or not.

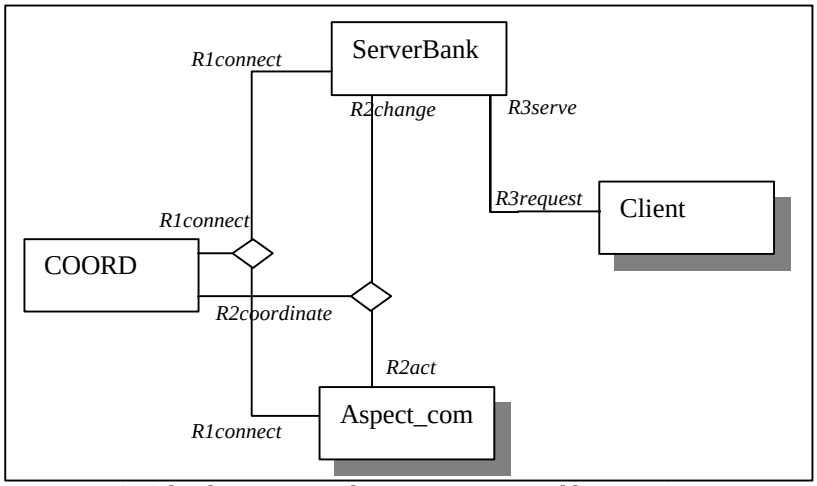


Figure 3. A bank account with aspect represented by LEDA.

In LEDA one *role* for each component interaction with the other ones in the system needs to be defined. The *role* behaviour is done by one or several agent definitions. It is possible to pass data by matching two complementary actions.

3.2.1. Role description:

In this section we are going to describe the *roles* needed to interconnect elements.

R1: Connect: models the interaction between elements to be connected. In the example elements to be connected by R1 are ServerBank - Aspect_com- COORD.

R2 Co-ordinate-Change-Act is defined by the following:
a) Models the co-ordination activities of the *co-ordinator* component (COORD) in its relation with other components to be connected with it. This *role* describes how co-ordination is scheduled by the *co-ordinator* component.
b) Models the possibility of changing the ServeBank behaviour when an aspect component (Aspect_com) is considered.
c) Models how the aspect component (Aspect_Com) can act over the ServerBank component.

R3: Request-Serve: models the relationship between the Client and the ServerBank components as it was defined by the given functional specifications.

Interconnections are established when component instances and *roles* are created (defining attachments). They can be modified in a dynamic way.

3.2.2. Explanation of operation sequence

The following points out the steps given in a bank account operation, by LEDA.

(1) The co-ordinator and the connection between it and other system elements is created (role R1).

(2) When a Client wishes to make a connection to ServerBank, it sends it a request (throughout *role* R3-*Request*).
(3) The request is received by ServerBank and the Client remaind waiting for the answer.
(4) The co-ordinator (COORD) stand by until it detects an event over ServerBank (a request from the Client).
(5) Then the co-ordinator performs R2 *coordinate*. This causes ServerBank to connect (through R2 *change*) with Aspect_com (through R2 *Act*).

3.2.3. LEDA System Definition.

The lines bellow express how we can define a system in LEDA

```
Component Bank_System_+_Aspect {                <--- Component declaration
        Interface none;                         <---Refereed to all the system
        Composition                             <--- System composed by:
                cli: Client;                    <--- * We suppose 1 client only
                coor: Co-ordinator;
                serv: ServerBank;
                asp: Aspect_com;
        Attachments
          asp.connect(connexion)<>coor.connect(connexion)<>serv.connect(connexion)
          asp.act(param)<>coor.coordinnate(param)<>serv.change(param)
          cli.req(datoa)<>serv.serv(datas)
}
```

attachment lets us define links throughout. Communications can be establish with the help of some parameters.

A system **instance** is declared as:

```
Instance SystemAspBank: Bank_system_+_Aspect
```

The whole system will be defined when each role is defined. Some of them can be done by LEDA, but other ones (like R2) need new primitives to be defined. At this moment, this is one of our lines of work.

4. Conclusions and Future Works.

In this paper some guidelines to integrate concepts from Aspect Orientation, CSBE and architectural software have been presented. The work is focused on defining systems changing dynamically their behaviour at design time when new aspect policies are added/eliminated without changing components. We show how aspect separation can be handled by means of SA, in a particular way with a specific ADL. We present the aspect separation problem as one of co-ordination.

Systems to be designed are built from functional components (describing functional behaviour) and components doing aspect policies. It is necessary to establish dynamic interactions among component interfaces as well. Thus, at the architectural level we can modify the system structure without changing components and knowing its behaviour at design time.

Currently we are working along several lines:
- To select the best way to specify functional components (and their interfaces).
- To determine which aspects can be applied at thearchitectural level. Consequently it is necessary for them to be well specified too (their interfaces, preactions, postactions...).

- To know the system throughout the specification of functional and aspect components; we will translate them to LEDA. After that, we will define the dynamic interactions among components. That means we'll have got the system's structural description.
- For getting this it is necessary to define new LEDA primitives to express aspectual concepts and a software tool to manipulate the system at the architectural level, and then generate Java code for getting a prototype.

4. Bibliography:

[All97] R. Allen. *A Formal Approach to Software Architecture*. Doctoral Thesis. School of Computer Science. Carnegie Mellon University. USA CMU-CS-97-144. 1997

[BeAk01] Bergmans L.M., Aksit M. *Compossing Crosscutting Concerns using Composition Filters*. Communications of ACM vol 44, No. 10 pp51-57. 2001.

[BrMo03] I.Brito, A.Moreira. *Towards a Composition Process for Aspect-Oriented Requirements*. Workshop on Early Aspects 2003. AOSD conference. Boston USA, 2003.

[Can00] C. Canal. *Un Lenguaje para la Especificación y Validación de Arquitecturas Software*. Tesis Doctoral. Universidad de Málaga. 2000.

[ClWa01] Clarke, S., Walker, R.J. *Composition Patterns: An Approach to Designing Reusable Aspects*. Proceedings of International Conference of Software Engineering, ICSE 2001. Toronto, Canada 2001.

[ClHe03] Clemente, P., Hernández, J. *Aspect Component Based Software Engineering Systems*. 2nd Workshop on Aspects, Components and Patterns for Infrastructure software. 2003 AOSD Conference. Boston, USA.2003.

[ClSaPe02] Clemente, P., Sánchez, F., Pérez,M.A. *Modelling with UML Component-Based and Aspect Oriented Programming Systems*. 7th Workshop on Component-Oriented Programing (WCOP'02) at. ECOOP'02. Malaga, Spain 2002.

[GaMoWi97] D Garlan, R.T. Monrie, D. Wie. *Acme: An Architecture Description Interchange Language*. In Proceeding of CASCON'97. Ontario Canada 1997.

[Gru99] Grundy, J. *Aspect-Oriented Requirements Engineering for Components-based Software Systems*. 4th IEEE International Symposium on Requirements Engineering. IEEE Computer Society, Limerick, Ireland, 1999, pp.84-91.

[Gru00] Grundy, J. *An Implementaion Architecture for Aspect Oriented Component Engineering*. In Proceedings of the 5th International Conference on Parallel and Distributed Processing Techniques and Applications: Special Session on Aspect-oriented Programming, Las Vegas, June 26-29 2000, CSREA Press

[Ki+96] G. Kiczales et al. *Aspect-Oriented Programming*. In Max Mühlhäuser ed, Special Issues in Object-Oriented Programming, Workshop Reader of the 10th. European Conference on Object-Oriented Programming, ECOOP'96, Dpunkt-Verlag. 1997.

[Kic+01] Kiczales, G, Hilsdale, E. Huguning, J, Kersten, M, Palm, J, Griswold, W. *An Overview of AspectJ*. 2001 Proceedingd of ECOOP, Springer Verang. LNCS 2072.

[LiOrOv01] Lieberherr, K., Orleans, D.,Ovlinger, J. *Aspect Oriented Programing with Adaptive Methods*. Communications of ACM, vol 44, No. 10, pp.39-41. 2001.

[Luc95] D.C. Luckman et al. *Specification and Analisys of Systems Architecture using Rapide*. IEEE Transaction on Software Engineering. San Francisco. USA. 1995.

[MaKr96] J. Magee, J. Kramer. *Dynamic Structure in Software Architectures*, in ACM Foundations of Software Engineering. San Francisco, USA. 1996.

[Mur01] J.M. Murillo. *Coordinated Roles: Un modelo de coordinación de objetos activos*. Doctoral Thesis. Universidad de Extremadura. Spain. 2001

[Na+02] A. Navasa, M.A.Pérez, J.M. Murillo, J. Hernández. *Aspect Oriented Software Architecture: A Structural Perspective*. Workshop on Early Aspect: Aspect-Oriented Requirements Engineering and Architecture Design. 1st International Conference on Aspect-Oriented Software Development (AOSD). April 2002. Enschede. Holanda Web site download.. http://trese.cs.utwente.nl/AOSD-EarlyAspectsWS/workshop_papers.htm

[OsTa01a] Osser H., Tarr P., *Hiper/J. Multidemensional Separation of Concerns for Java*. International Conference on Software Engineering. ACM pp. 734-737. 2001.

[OsTa01b] H. Ossher, P. Tarr. *Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software*. Communications of the ACM, October 2001, vol 44, num 10.

[Ra+02] Rashid A., Sawyer P., Moreira A., Araujo J. *Early Aspects: a Model for Aspect Oriented Requirements Engineering*, IEEE Joint Conference on Requirements Engineering. Essen Germany, September 2002.

[Sou00] D'Souza, D. Objects, *Components and Frameworks with UML*. 2000 Web site: http://www.trireme.u-net.com/catalysis/.

[SuYa99] Suziki, J., Yamamoto, Y. *Extending UML with Aspects: Aspect Support in the Design Phase*. AOP Workshop at ECOOP'99, Lisbon, Portugal, 1999.

Software Visualization and Aspect-Oriented Software Development

Susanne Jucknath, susannej@cs.tu-berlin.de
Institute for Software Engineering and Theoretical Computer Science
Technical University of Berlin

Abstract

One intention of *Software Visualization (SV)* is to form a picture in the users mind of what this software is about. What happens during the execution and what should happen in the programmer‘s point of view. This is especially helpful with existing software and non-existing documentation. So then the amount of existing software is rising the need of understanding this software is rising too. There are several methods of SV for different software paradigms, like object-oriented software development. The intention of this paper is first to examine existing methods for their use in *Aspect-Oriented Software Development (AOSD)* and second to extend three of this concepts for the special needs of AOSD.

1 Introduction

Software Visualization (SV) for imperative software could be easily divided into visualization of the algorithm and visualization of the program code. This separation of concept and implementation is rather difficult for object-oriented software, because the classification of objects itself contains a level of abstraction. James Noble [1] developed in this sense the APMV-Model - Abstraction Program Mapping Visualizing - which respects modularization and information hiding as it is merely based upon objects and not in their implementation.
Not objects inherently but their communication are the base of the so called execution murals [2]. This concept display the all messages between two or more classes. But just displaying this communication data is not enough, since the benefit of a SV should be to gain more information. So it would be nice to see some patterns in this communication or more so, to automatically detect patterns. Therefore we have to decide what the concept of pattern mean in this case. Is is necessary to have strict *Object Identity*, before we decide that two communication streams belong to the same pattern? Or is *Class Identity* satisfactory? There are several definitions possible, who will lead us to Execution Patterns, like they are defined in [3].
Although it is very helpful to analyze program code automatically via execution patterns, we do not use the full a priori knowledge this way, we already have about the abstraction of this program code in Aspect Oriented Software. In Aspect Oriented Software Development (AOSD) another abstraction level is realized than in Object Oriented Software Development (OOSD). Aspects and concerns have a direct impact on the program code but are not a direct part of it. So we can use the methods discussed above - it is object oriented anyway - but we have to extend it to the level of information AOSD includes.
But how could a woven aspect be visualized? We can build a static visualization (e.g. of the class-model) or a dynamic visualization (e.g. of the class communication or the program execution).

In the next two sections, we will shortly discuss static visualization as well as dynamic visualization for AOSD. Based on this, we can estimate in the conclusion sector about the benefit of SV for AOSD.

2 Static Visualization

Visualization of program code often starts with information rendered during the system design, e.g. a static class model. Early approaches of SV for AOSD were made by K.Czarnecki and K.Lieberherr. They displayed the relation between aspect and class in a cross-classified table. This description is sufficient but stands alone, with no direct (visual) connection to the visualization of the class model.
The common visualization of a class model is a graph. It fits also well for AOSD as long as we can model the extra information (which aspect is interacting with which class). This could be done by coloring each class (node) depending on each aspect it effects. Of course with more than three aspects (RGB) we get a varicolored graph.

3 Global View and Local Effects

The static visualization of the class model implies a direct view on the amount of effected code. But there are a few drawbacks. First of all with a large class model (and even worse various aspects) it is hard to separate the wheat from the chaff and to decide which influence is important. Second - as a consequence - it tells few about the behavior at run time. And at last large class models tend to be hard to handle for the kind of information we want to display.
For example, if we want to visualize the difference before and after weaving a new aspect into the code we need a before-and-after encoding. This transcription should be version-dependent but also run time dependent as we want to measure the real amount of difference. This leads us directly to a dynamic visualization.

4 Dynamic Visualization

A dynamic SV could show the influence of an aspect at run time, e.g. to highlight counterproductive aspects. But to show a counterproductive behavior it is essential to show the right information of what happens. This information could contain the communication between objects or the consumption of different resources. In analogy to Audio-Signal-Processing we have to deal with discretisation and synchronization of the message flow. We can avoid this by modeling the message flow to a given time t.

4.1 Information Murals

In [3] execution mural shows the communication between two classes by putting the classes on the y-axis and the boolean information about communication between them at time t on the x-axis. It is not a problem to place independent class communication in a information mural but a multi-communication between different classes turns out to be complex.
For the visualization of a multi-communication the idea of a information mural can be transformed to the visualization of classes and their communication activity. For that we can place the classes on the y-axis and the activity of a class to a given time t on the x-axis. The z-axis is left for changes over time $t_1, ..t_n$.

Let us call such a information mural activity information mural (AIR). We can compute AIRs for different version or stages of software development or before and after weaving an aspect into code. But this alone is not enough for the need of a good SV for AOSD. First of all we have to eliminate the white noise (classes who are constantly communicating) and second we need a metric to measure the effect of the aspect. The white noise can be blanked out by a filter function, which can be directly coupled with the activity function.
To set a metric we can measure the distance between two given AIRs. This distance function is, like the activity function, very dependent on the kind of information we want to collect.

4.2 Graph Animation

An information mural gives a compact picture of what is happening during run time, but not a idea of the reason why it is happening. For this we would need information about coherence between different objects. That was a service delivered by the class model. So the next question is, how to combine the benefit of the dynamic information mural with the static class model. Or in other words, to display the most active classes and their connections over time.
For this we must define *active* via an activity function (AF) for each class. The value one class reaches at time t with the AF computes the visual worth of this class. The visual worth of a class again decides if the class is displayed in the class-model or not. If we do not show all classes in the class model it is naturally not a real class model anymore and should be denoted as momentary class graph (MCG).
With different MCGs to different time steps $t_1, ...t_n$ we can use finally the existing algorithms for graph animation to realize a smooth and user-friendly SV.
Graph Animation as described in [4, 5] defines a mental map and layout quality for each graph and a mental distance between successive graphs.

5 Conclusion

Aspect-Oriented Software Development (AOSD) extends Object-Oriented Software Development (OOSD). As we have seen, we can also extend existing Software Visualization (SV) algorithms for OOSD to Software Visualization for AOSD. We showed one example of static SV for OOSD (the class model) to be supplemented by one color for each aspect to meet a SV for AOSD. Several SV for OOSD could be extended in a similar way. So the question is not only if it is possible to get a easy access to SV for AOSD, but what do we expect from it? Mainly we want to see three facets:

1. What class is affected by which aspect?
2. What behavior shows the program before and after weaving the new aspect in?
3. Resulted this new aspect to the desired behavior?

Point 2 and Point 3 cannot be shown in a static SV, we need a dynamic SV to see the behavior of the program code by run time. Two of the earlier mentioned approaches for dynamic SV for OOSD were *Information Murals* and *Execution Patterns*. Each of them has a great effort on SV for OOSD, but can not handle to show a priori knowledge like aspects.
But a dynamic SV (*Information Murals*) combined with a static SV (the class model) to a program code animation via graph animation could probably serve us with the just right amount of information.

It could be also interesting to extend *Execution Patterns* the way we extended *Information Rurals*. In particular, since there is a very good tool called *Jinsight* [6] which provides for a lot more views on OOSD, for background see also [7, 8].

The need of SV for AOSD is obvious, because it could provide a way better understanding of the assets and drawbacks of AOSD in a given context. Especially if we implement more then one aspect and are not able to overview their interrelation. The question if the ideas discussed in this paper could fulfill this hope is another case. We welcome all kind of feedback, comments and critics.

6 Bibliography

[1] James Noble, *Visualizing Objects:Abstraction, Encapsulation, Aliasing and Ownership,* Springer-Verlag Berlin Heidelberg, 2002
[2] Jerding, Dean and Stasko, John T., *Visualizing Message Patterns in Object-Oriented Program Executions,* Graphics, Visualization and Usability Center, Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-GVU-96-15, May 1996
[3] John Vlissides, Wim De Pauw, David Lorenz and Mark Wegman, *Execution patterns in object-oriented visualization* USENIX Conference on Object-Oriented Technologies and Systems, 1998
[4] Stephan Diehl, Carsten Goerg and Andreas Kerren, *Foresighted Graph layout,* University of Saarbrucken, Technical Report A-02-2000, February 2000
[5] Stephan Diehl and Carsten Goerg, *Graphs, They Are Changing,* Graph Drawing 2002, Springer-Verlag Berlin 2002
[6] Wim DePauw et. al. *Jinsight* , www.alphaworks.ibm.com/tech/jinsight
[7] Wim DePauw, Richard Helm, Doug Kimelman and John Vlissides, *Visualizing the Behaviour of Object-Oriented Systems*, OOPSLA 93, October 1993
[8] Wim DePauw, Doug Kimelman and John Vlissides, *Modeling Object-Oriented Program Execution*, ECOOP 94, July 1994

Refactoring in the Presence of Aspects

Jan Wloka
Fraunhofer FIRST
Jan.Wloka@first.fraunhofer.de

Abstract

Refactoring and Aspect Orientation (AO) are both concepts for decoupling, decomposition, and simplification of object-oriented code. Refactoring is meant to guide the improvement of existing designs. For this reason it is the main practice in eXtreme programming to implement 'embrace change' in a safe and reliable way. Aspect orientation on the other hand offers a new powerful encapsulation concept specifically for coping with so called crosscutting concerns.
Although refactoring and AO have the same goals their current forms impede each other. Since the development of modular systems has become more and more difficult a combined application of refactoring and AO is still a desirable goal and would be a great help for developers.
In this position paper we compare both with the focus on their capabilities as code transformation techniques, and try to reveal influences and direct dependencies between them. Finally, we discuss how refactoring and aspect orientation can help each other to achieve more modular systems.

Keywords:
Refactoring, Aspect Orientation, AOSD, Crosscutting Concerns, Code Transformation, Evolvability

1 Introduction

Software developers work every day on the realization of requirements that are induced by an changing environment. Often these requirements are treated as fixed in order to build a solid specification from it, which in turn should be used as a stable foundation for the further realization stages. However, it is often the case that the environment changes faster than the given requirements are accomplished. In a world where “moving targets” are daily business, developers have to consider present and future requirements in several stages of the development process.
However, predicting what will be needed in the future is a difficult task. Usually new requirements appear or existing change during the development. Those unanticipated changes cause many modifications or even a complete re-implementation of various parts of the system. On the other hand, focussing to much on potential changes of requirements in advance will lead to an over-designed system with a complex, opaque and, in effect, unevolvable structure. Hence, developers are faced every day with changes to existing parts of a software system in order to adapt it to changed environments or to incorporate new features.

A technique which allows to change “... a software system in such a way that is does not alter the external behavior of the code yet improves its internal structure” [Fowler99] is called **Refactoring**. It offers a disciplined and safe way to improve the design of existing code while minimizing the chances to introduce new bugs. Refactoring deals with restructuring a system in such a way that the resulting changes can be implemented without problems. Within eXtreme programming 'embrace change' together with refactoring allows to keep the software system simple without pre-planning its complete design.
Besides keeping software soft, the primary goal of refactoring is simplicity – keeping the system as simple as possible. From this the more concrete goals of elimination of code duplications, good distribution of responsibilities, and code with low complexity can be derived. In general, refactoring decomposes, decouples, and simplifies structure into smaller types and methods. Thus it improves the readability, modularity and therefore the evolvability of a software system.

Aspect Orientation (AO), or more precisely Aspect-Oriented Software Development (AOSD), is a software development paradigm that was found to be able to cope with the issues arisen by crosscutting concerns. Aspects are introduced as new first class entities in order to encapsulate crosscutting concerns. An aspect can be implemented in a separate module, though the functionalities provided by it are spread across the application probably even crosscutted. This distribution of aspect functionalities is performed by a specific integration tool which allows to put a crosscutting implementation of a concern at a single place, i.e. inside of an aspect.
At the programming language level aspects are a new artefact which mainly consists of the aspect code and connector definitions. The **aspect code** is the code

which is to be woven into the base system. In terms of AspectJ [AspectJ] it is called “advice code”. The **connector definitions** specify the join-points where the aspect code should be merged with the base code. With the specification of join-points the connector definition links to the structure and semantics of the base code.

Evolving a system means that its (base) code is refactored or at least modified during the course of software development or maintenance. In current aspect-oriented systems the linkage between aspects and base code is realized using the names of structural elements. Modifications by refactorings, for instance to the names or to methods representing join-points, can render these connections invalid. Therefore, both refactoring and aspect-oriented programming languages in their current forms cannot be used with each other.

In this paper we explore whether refactoring and AO can coexist and what changes are necessary so that both are applicable at the same time. Furthermore, we analyze how refactoring and AO could help each other reaching their common goal of improving the modularity of software systems. We compare the technique refactoring with from a technical point of view aspect weaving based on the assumption that both of them are code transformation techniques.

The rest of the paper is organized as follows. Section 2 introduces the key features of refactoring as well as of AO, and motivates the following analysis. Section 3 compares refactoring and aspect weaving as code transformation techniques. Section 4 reveals possible problems induced by simultaneous use of refactoring and AO. In section 5 we discuss solution possibilities for cooperation and mutual improvement and in section 6 we present related work. Finally, in section 7 we offer some thoughts for future directions.

2 Motivation

Both refactoring and AO strive to increase the modularity of source code. The technique refactoring offers a safe modification of the structure of code elements. Several concrete ways to refactor in specific circumstances have been identified and presented by Martin Fowler in an initial catalogue [Fowler99]. They are described by a name, the problem, the context, and the solution in form of a step-by-step guide. Because the form of a pattern as pioneered by Alexander [Ale77] is fulfilled, these specific guides for performing refactoring can be called **Refactoring Patterns**.

During the modification process the developer follows such pattern, which define modifications in small, atomic, and invertible steps. Lets consider an example refactoring pattern, *Extract Method*:[1]

```
void saveState()
{
   State state = _model.getState();
   XMLWriter writer =
     _db.openConnection();

   // transform to XML
   Buffer buf = new Buffer();
   buf.add(state.getPerson().toXML());
   buf.add(state.getData().toXML());
   buf.add(state.getCharge().toXML());

   writer.write(buf);
}
```

```
void saveState()
{
   State state = _model.getState();
   XMLWriter writer =
     _db.openConnection();

   wirter.write(
     toXML(_model.getState()));
}

Buffer toXML(State state)
{
   Buffer buf = new Buffer();
   buf.add(state.getPerson().toXML());
   buf.add(state.getData().toXML());
   buf.add(state.getCharge().toXML());

   return buf;
}
```

The method `saveState()` does actually more as saving the state of the enclosing object. It also converts the object's state to an XML representation. Therefore, the statements to realize the XML conversion have to be extracted into a separate method.
To achieve this, inside of an existing class a new method is created and given a reasonable name. The desired statements are copied into it, possible temporary variables and parameters are identified and passed into the new method, and finally the extracted code in the source method is replaced by a suitable call to the new method.

The appendant mechanics for this *Extract Method* refactoring pattern would look like the following:

1. Create a new method, and name it after the intention of the method ...
2. Copy the code in question from the source

1 For more details the reader is referred to [Fowler99], page 110.

method into the new target method.
3. Scan the extracted code for references to any variables that are local in scope to the source method. ...
4. See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables.
5. Look to see whether any of these local-scope variables are modified by the extracted code. If only one variable is modified, see whether you can treat the extracted code as a query and assign the result to the variable concerned. ...
6. Pass into the target method as parameters all local-scope variables that are read from the extracted code.
7. Compile when you have dealt with all the locally-scoped variables.
8. Replace the extracted code in the source method with a call to the target method. ...
9. Compile and test.

Performing these steps manually however is costly and error prone, so tools are the preferred way to handle the transformations in a fast and reliable way. The so called **Tool-supported Refactoring** is offered as an integral part of many modern IDE's, such as Eclipse or IDEA. A developer usually selects a desired refactoring pattern for a certain source code element, and the refactoring tool then applies it automatically. Every desired design improvement seems to be just one click away.

Design improvements performed by refactoring are limited to object-oriented modularization concepts. Hence, some design conflicts cannot be solved, often complexity is only reduced locally. In such cases complex structures are not decoupled or removed. They are just moved around or more precisely when ever a certain structure was simplified at one place it has been introduced even more complexity to other structures. The relationships between globally dependent parts of the system are often difficult to refactor.
Such design conflicts are often caused by so called "**Crosscutting Concerns**". A concern in general can be seen as a particular goal, concept, or area of interest. The term "crosscutting" indicates that a concern affects multiple implementation modules. For object oriented system the implementation of such a crosscutting concern is spread across many types and methods. Typical examples for crosscutting concerns are: synchronization, notification, logging, exception-handling, memory-management.

Global design improvements through refactoring are impeded by crosscutting concerns. Therefore they could be seen as an indicator for the weakness of object-oriented modularization concepts. Refactoring of crosscutting concerns causes "oscillating" structure – no fixpoint can be found where every required improvement has been reached. A new concept for encapsulating those crosscutting implementation of concerns is required.

Aspect-oriented Software Development (AOSD) was invent to cope with crosscutting concerns. It provides a new first-class entity – the aspect – to encapsulate crosscutting concerns. Whereas the new artefact aspect seems to solve any problem with crosscutting the underlying aspect-oriented languages need a lot of technical infrastructure to get aspects working.

From a technical point of view – here explained in terms of AspectJ[2] – AO provides two major concepts: **Control flow** and **Type Modification**. The former enables modifications of specific points within the control flow, so called join-points. These exposed points are for example method and constructor invocations, field references and assignments. Aspect code can be inserted directly into the control flow before or after such join-points. Additionally, it is allowed to completely replace the code of a join-point.[3]

The following example illustrates how control flow modifications are defined in AspectJ:

```
class MyClass
{
   public void doSomething()
   {
     /* do something important */
   }
}

aspect MyAspect
{
   before(): execution(
     void MyClass.doSomething())
   {
     /* do something before */
   }
}
```

The class `MyClass` contains a method `doSomething()` providing some important functionality. The aspect `MyAspect` provides functionality that should be performed every time before the method `doSomething()` is called. As indicated by the keywords `before` and `execution` the aspect code is inserted at the very beginning of the body of the method `doSomething()`.

Control flow modifications often require a change to

2 We have chosen AspectJ because it seems to be the most known AO technology.
3 In this paper concentrates on so-called static crosscutting, because dynamic crosscutting and especially dynamic weaving do not modify source code statically, which makes it difficult to compare it to source code transformations, e.g. refactoring.

the structure of types that contain join-points. To this end aspect-oriented programming languages offer the introduction of new fields and methods. Additionally, the inheritance relationships can be changed. Type modifications are used to adapt the internal structure of a type to be suitable for an aspect code. In particular, aspect code added to a certain join-point needs sometimes additional fields are features.

Control flow and type modifications can be defined at one place – inside of an aspect – yet allow a wide spread application of changes to many code elements. The connector definitions specify where an aspect crosscuts the software system using specific rules for the selection of join-points which represent the links between aspects and the remaining system.

Finally, the aspect code must be physically integrated according to the connector definitions. As mentioned above, the system and the aspects are composed by a specific integration tool, called **Aspect Weaver**. It applies all required changes to the existing code elements and inserts the additional aspect functionality. In the example above the aspect weaver would generate the following code:

```
class MyClass
{
   public void doSomething()
   {
     /* do something before */
     /* do something important */
   }
}
```

3 Refactoring and Aspect Weaving compared

In this section we compare the technique refactoring with aspect weaving from a technical point of view focussing on code transformation as an underlying base technology. Aspect weaving is the technique to integrate aspect-oriented with object-oriented code which have to be applied before and AO system offers its complete functionality. That's why we consider in the following only aspect weaving for the comparison.

The technique refactoring offers a safe way to modify the structure of code elements. During the modification process the developer follows refactoring patterns, which define modifications in small, atomic, and invertible steps. Every step defines a distinct modification which can be applied by a tool in an automated way. In the case of tool supported refactoring the developer only selects the desired refactoring pattern, enters parameters (e.g. name for a new method), and then the tool performs every source code transformation automatically. Hence, a refactoring tool can be seen as a source code transformator.

Also aspect weaving may employ transformation of source code to adapt the base system. Of course, not all aspect-oriented approaches employ source code transformation. For example byte code transformations as used by Hyper/J and changes to the runtime environment without modifying the code at all [DucEstMor02] are common approaches, as well. However, to have an uniform base for comparison we only consider aspect weaving as a transformation of source code. In that case, aspect weaving performs transformations which are very similar to those of refactoring, e.g. the insertion of a new method to an existing class[4].

Considering refactoring and aspect weaving from that point of view, both deal with the application of changes to object-oriented code but with different goals[5].

Refactoring on the one hand transforms source code elements in order to improve their structure in such a way that it does not alter the external behavior. Its primary goal is simplicity – keeping the code as simple as possible. To this end, the internal structure is improved to ease comprehension and maintenance.

In terms of the visibility to the developer, refactoring is always invasive, because the source is changed after the application of refactoring. For example, the methods of a class have been simplified after performing *Extract Method* – they are doing less in terms of statements – but the number of methods has increased. The class now offers a new interface, which means the way of using that class from other parts of the system has changed. Therefore, every modification performed by refactoring has a direct impact on the developer as they are directly visible to him.

On the other hand aspect weaving also transforms source code elements but for a quite different reason. The major goal of aspect weaving is the explicit modification of the system's behavior. In particular, an aspect weaver modifies the object-oriented code in order to insert additional functionality at certain join-points, and properly modify existing methods or their enclosing types.

Compared to refactoring, a weaver transforms source code in a non-invasive way, because the source code isn't viewed by the developer anymore after the aspects are woven into it. Weavers are used prior to compilation or execution not during coding. The woven source code is a physically separated copy of the original source code which is left completely unchanged. Aspect weaving has therefore no direct

4 Which transformations are used mainly depends on the implementation of the employed aspect weaver.

5 To prevent confusion, Refactoring and **Aspect Orientation** have the same goal: an improved modular system structure. However, the application of code transformations performed either by a refactoring tool or by an **aspect weaver** follow different goals.

impact on the developer and is also invisible to clients of a given class.

Despite of their different goals, refactoring and aspect weaving have much in common especially regarding tool support. Both aspect weavers and refactoring tools may change object-oriented source code by applying transformations on the source code elements, possibly in an automated fashion. They even use the same transformations while some other transformations are used in the opposite direction. A refactoring tool extracts code into a new method, whereas an aspect weaver may append some code to the very end of a method. With this in mind, one could say refactoring is something like “un-weaving”.

To summarize, both can be used to apply modifications in an automated way. Refactoring on the one hand is invasive, visible to the developer, semantic preserving per definition, but it probably changes the system's structure. Aspect weaving on the other hand modifies code non-invasively, is invisible to the clients of the given class, and purposely changes the system's structure but also its behavior. Source code transformation is a base technology to which both refactoring and aspect weaving can be mapped; cf. [Recoder] (in case weaving is regarded as the static modification of source code).

4 Does Aspect Orientation impede Refactoring?

The application of an aspect-oriented language in real industrial projects seems to offer a powerful modularization concept for the realization of difficult design problems. In this section several problems caused by an integration of refactoring activities and AO are discussed in detail.

Refactoring is about changing the structure and naming of source code elements, for example moving a method to a more suited class since it uses or is used by more features of that class.

With AOSD, developers have to deal with new artifacts. They have to handle and implement aspects, that is, the aspect code which is to be woven into the base program and the connector definition, specifying the join-points where the aspect code should be merged with the base code. By specifying of join-points, the connector definition links to the structure and semantics of the base code.
As an example for a connector definition lets consider an AspectJ pointcut. It provides the developer the means to select a set of join-points by enumeration or by the application of name patterns:

```
pointcut enumarationBasedSelection():
  within(MyClass)
  && (execution(void MyClass.put(int))
    || execution(void MyClass.get()))

pointcut patternBasedSelection():
  within(MyClass)
  && (execution(* put*(*))
    || execution(* get*(*))
```

The first pointcut enumerates the join-point names within the context of their enclosing type, that is, name and structure information are stated explicitly and are therefore directly bound to the aspect.
The second selects join-points by name patterns using wildcards. This allows a more generic description, however it is to some extent bound to names, as well. In general, name-based connector definitions as employed by current aspect-oriented programming languages cause a tight coupling between aspects and the base system.

If a source code element is targeted by an aspect, and it is renamed perhaps by using a refactoring, the connector definition will be rendered invalid. Refactoring – unaware of AO – changes the structure of code fragments and with it the names and relationships of the contained corresponding code elements. If these elements are targeted by such name-based connector definitions they have to be updated as well. Otherwise, the associated aspect code defined to alter the system's behavior at a specific join-point will not be woven, which means that the system will lack the desired behavior.

The tight coupling between aspects and the base system impedes most kinds of modification, and therefore it prevents refactoring almost entirely. After performing a refactoring, verification of the related connector definitions is required in order to ensure its correctness. Broken linkages are hopefully indicated by the aspect compiler, so a manual verification is not required.
Talking about broken linkages between aspects and the base system, renaming is one of the most obvious problems, though many other refactoring patterns also affect aspect orientation. For example *“Inline Method”* will remove a possible join-point and *“Inline Class”* will even remove a complete set of possible join-points. *“Move Method”* will remove the join-point and put it elsewhere. Bigger refactoring patterns like *“Remove Middle Man”* are composed of several smaller refactoring patterns and thus will cause many problems at once.
Thus, refactoring has to be aware of the linkage between aspects and the base; it must always know the exact definition language it is applied to.

Another approach to solve this problem might be a more robust connector definition. Today's aspect-

oriented programming languages lack on expressive, systematic, powerful pointcut languages. The proposal by Kris Gybels and Johan Brichau in [GybBri03] employs a logic meta programming framework with specific predicates to link the aspect to desired join-points. Concrete names can be replaced by queries and thus aspects and the base system to become more loosely coupled.

Currently, the use of AO, especially the use of pointcut languages, make a software system more fragile and sensitive against many kinds of changes.

5 Working hand-in-hand

We have presented several connections of refactoring and aspect orientation. In the following we want to discuss how refactoring and AO could be integrated and how both could help each other to achieve better modularity. At first we state in which way AO may support refactoring and then we show possibilities for refactoring support for AO.

Aspect Orientation allows not only the distribution of responsibilities across classes but also their assignment to aspects; it therefore adds a new dimension of modularity. When the modularization by means of aspects is reached by the use of refactoring then new – even more powerful – refactoring patterns can probably be found.
Such refactoring patterns would guide the developer step-by-step for instance in extracting some behavior from different classes into an aspect. However, the patterns would have to be specific to an underlying programming model. For example, the programming model AspectJ offers introductions and advices, therefore particular refactorings like "*Move to Introduction*" or "*Move to Advice*", will be needed to extract object-oriented code into aspects. In contrast to that, the programming model Object Teams [Herrmann02] provides teams, roles, and connectors, which give rise to refactoring patterns like "*Move to Team*", "*Extract Role Behavior*", and "*Expose to Connector*".

Regardless of the actual programming model these new refactoring patterns will be a great help for the extraction and manipulation of aspects in general.

Apart from that, AO may also be helpful to achieve the major goal of refactoring: the enforcement of "once and only once". Code duplication provides a good example for the limits of object orientation. Duplicated code, or even worse, very similar (but not duplicated) code may be hard to encapsulate by means of object orientation. Especially widespread duplication is made up of often nearly the same code but used in very different contexts. Some aspect weavers create a lot of duplicated or similar code across the system if the aspect code is statically woven at the source code level. It might be interesting to evaluate if a "reverse" action – extracting duplicated or similar code into aspects – is also possible.

Refactoring may allow the restructuring of aspect-oriented programs in a safe way similar to how it does for object-oriented software. Every change like "renaming", "moving", or "extraction" can be performed in a disciplined and safe way. With tool support the restructuring of code would be even more effective and faster. In addition to the links between objects, as caused by inheritance or use relationships, AO introduces a new linkage between different code elements: the aspect – base code relationship. Due to the invisibility of this linkage [Vollmann02] a developer does not know whether a certain code element is connected to an aspect. A solution could be the enhancement of current refactoring patterns in order to take care of existing name-based connector definitions. Furthermore, refactoring tools must be improved to verify and probably adapt these definitions in an automatic fashion.

Besides having aspect-oriented code as the target of refactoring, refactoring can also be a great help in other areas such as to expose a desired point of the control flow. Today's aspect-oriented languages like AspectJ or Hyper/J [Hyper/J] provide powerful mechanisms to crosscut a program's code, but sometimes they are not expressive enough to access every desired join-point within the control flow. The syntax is limited to exposed source code elements like fields and methods. Refactoring could help here to expose a certain point of the control flow and make it accessible. For example, the *Extract Method* refactoring pattern allows the extraction of a block of statements into a new method. An aspect is then able to place its aspect code right before the extracted statements. Here, refactoring improves structure not in the usual sense (simplification), but to insert hooks to ease adaptation. Hence, the safe preparation of join-points for being accessible by aspects is a possible aid for aspect orientation.

6 Related Work

In general, aspect-oriented approaches can be distinguished into language extensions and approaches with a tool-like character. Language extensions provide new syntax elements for an existing programming language. Their primary goal is to provide a single programming language for the implementation of the software system, its aspects, as well as for the definition as to where and how both should be composed. Typical examples are AspectJ [AspectJ], Composition Filters [CF], and Object

Teams [Herrmann02]. Tool-like approaches define a second, more descriptive language to express how the different parts of the software are composed. A typical example is Hyper/J from the IBM hyperspaces people [Hyper/J]. In this paper we concentrated on language extensions mainly because refactoring is about the transformation of source code. Additionally, it simplified the discussion because only one programming language had to be considered. However, for a technical point of view it would also be possible to provide a refactoring tool which is aware of Hyper/J's hyperspace definitions and composition rules.

In addition to the discussed approaches no other research work about the influences of refactoring to other AO approaches seems to be available. Only some brief ideas were proposed, e.g. about eXtreme programming and AOP in general [KirJaiCor02]. Apart from that, another very important issue which arise together with refactoring seems not very well investigated: unit testing of aspects. Currently, just a few thoughts about the employment of data flow testing in order to test aspects have been published [Zhao02].

7 Future Directions

Some connections between aspect orientation and refactoring have been shown, but there are others that might be worth to investigate in future research.

Aspect weaving like refactoring may transform source code. It would be interesting to evaluate what kind of relationships between source code elements are changed by whom. It is not clear what kind of relationships typical implementations of aspect weavers and refactoring tools change. Moreover, it would be interesting to determine what properties of a specific relationship are modified by refactoring and by aspect weaving. Also, there are general purpose transformation tools like [Recoder], which can perform generic transformations upon source code. We want to explore whether such tools can be used as the foundation for refactoring and aspect weaving tools in order to improve the control of code transformation.

More robust linkages between aspects and the object-oriented code are needed. We want to investigate typical changes to source code which affect connector definitions, and we will find out whether there are general differences between control flow and type modifications. Other approaches for more robust connector definitions shall be evaluated. That means in particular how can the required structure information be obtained, is static program analysis sufficient and how can the validity of dynamic connector definitions be checked?

Extensions to existing refactoring patterns are not only needed, they are essential for the application of current refactoring tools in the presence of aspects. We want to investigate how existing refactoring patterns should be extended, particularly in the case of aspect-oriented programming model, Object Teams. In doing so, we want to discover what has to be considered in detail for each known refactoring and if there are recurring changes to all refactoring patterns.

Finally, some new refactoring patterns are needed for guiding the extraction of aspects. What kind of refactoring patterns are needed, in particular for Object Teams? Therefore, we have to determine those transformations that are most often applied during the decomposition of object-oriented code into teams.

Acknowledgements

Thanks to Stephan Herrmann and Thomas Dudziak for valuable feedback and discussions on this research.

8 References

[Ale77] Christopher Alexander et al. *"A Pattern Language: Towns, Buildings, Construction"*. Oxford University Press, 1977.

[AspectJ] The AspectJ project, www.aspectj.org

[AspectJProg] AspectJ Team: The AspectJ Programming Guide. http://aspectj.org/doc/dist/progguide/.

[CF] The Composition Filters homepage, http://trese.cs.utwente.nl/composition_filters/

[DucEstMor02] Frédéric Duclos, Jacky Estublier, and Philippe Morat. *"Describing and Using Non Functional Aspects in Component Based Applications"*, In Proc. of AOSD, 2002.

[Fowler99] Marting Fowler. *"Refactoring: Improving the Design of Existing Code"*. Addison-Wesley Longman, 1999.

[GybBri03] Kris Gybels and Johan Brichau. *"Arranging Language Features for More Robust Pattern-based Crosscuts"*. In Proc. AOSD'03, Boston 2003.

[HanUnl03] Stefan Hanneberg and Rainer Unland. *"Parametric Introductions"*. In Proc. of AOSD, 2003.

[Herrmann02] Stephan Herrman. "Object Teams: Improving Modularity for Crosscutting Collaborations". In Proc. of Net.ObjectDays, Erfurt, 2002.

[Hyper/J] IBM alphaworks, Hyper/J Homepage, http://www.alphaworks.ibm.com/tech/HyperJ

[KirJaiCor02] Michael Kircher, Prashant Jain, and Angelo Corsaro. *"XP + AOP = Better Software?"*. In Proc. XP'02, Alghero, Sardinia, Italy.

[MyAOP1] Ramnivas Laddad. *"I want my AOP!, Part 1"*. JavaWorld Online Magazine, January 2002.

[ObjectTeams] Homepage of the 2nd generation aspect-oriented programming language Object Teams, http://www.objectteams.org.

[Recoder] Homepage of the transformation framework Recoder, http://recoder.sourceforge.net

[Vollmann02] Detlef Vollmann, *"Visibility of Joinpoint in AOP and other Implementation Languages"*, 2002

[Zhao02] Jianjun Zhao. *"Tool Support for Unit Testing of AspectOriented Software"*. OOPSLA'02 Workshop on Tools for Aspect-Oriented Software Development, Seattle, WA, USA, 2002.

An Example of generating the synchronization code of a system composed by many similar objects

Szabolcs Hajdara, Balázs Ugron

(Budapest, Hungary)

Abstract. In this paper we take a synchronization specification of a parallel system in language MPCTL*, then we produce an abstract synchronization skeleton, using an object-oriented extension of P. C. Attie's and E. A. Emerson's method, and finally a concrete Java code will be generated. The described method should ease the handling of synchronization by generating the synchronization code of object-oriented systems, so it should be unnecessary to code the synchronization by hand.

1 Introduction

Synchronization code and real computation code can be separated in the case of most parallel programs. If so, the synchronization part of the program can be specified separately and the synchronization code can be generated from the specification. Let us remark, that this idea might be derived from the base thought of Aspect-Oriented Programming, because the synchronization can be considered as an aspect of the system. More information about AOP can be found in [15]. Other synchronization techniques can be seen for instance in [1] and [6].

The synthesized system of K similar objects is a mechanically constructed correct solution of a precise problem specification given by MPCTL* (Many-Process CTL*) formulas. K is an arbitrary large natural number and an MPCTL* formula consists of a spatial modality followed by a CTL* state formula over uniformly indexed family of atomic propositions.

The method used in this paper applies the technique suggested by P. C. Attie and E. A. Emerson in [10], and it inherits an important advantage of their method, namely how to deal with an arbitrary number of similar objects without incurring the exponential overhead due to the state explosion problem.

To use the method developed by P. C. Attie end E. A. Emerson in [10], we had to solve the handling problem of shared variables by the similar objects. The details can be found in [12].

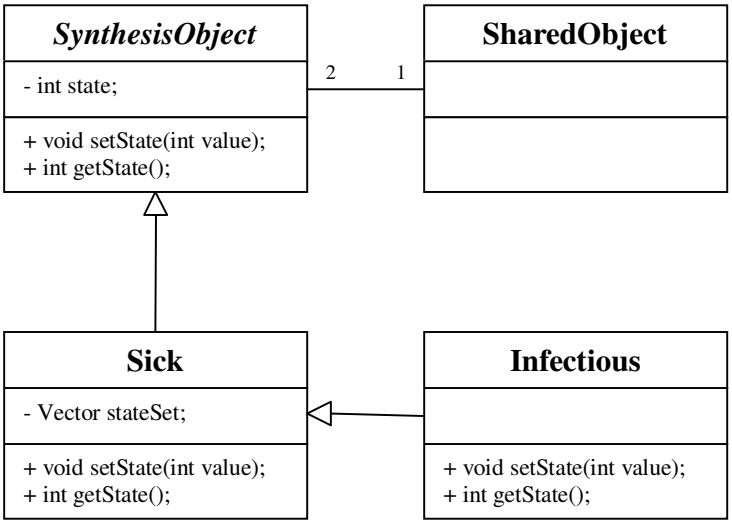


Figure 1: The class diagram of the example

2 The Task

We show the method through an example of a simulation program of a surgery.

Given a surgery, which is accepting patients. Patients can be infectious or non infectious. The doctor suggests that the patients, who think they are infectious, should not stay in the waiting-room if there is some other person in the room, and if there is an infectious patient in the waiting-room then the other patients should stay outside in the bright spring sunhine until the infectious patient in the waiting-room leaves. For the sake of simplicity we do not consider that patients can stay in the surgery, too, we only consider the synchronization of the the patients in the waiting-room.

3 The solution

According to the method described in [12], a new class (*SharedObject*) should be introduced for the synchronization, which class takes part in the synchronization of two objects. Moreover, all classes implement an interface (*SynthesisObject*), which defines the methods needed for the synchronization. According to the above, the class diagram of the system can be designed like in Figure 1.

There will be particular number (defined by the method) of *SyntesisObject* type objects in the class *SharedObject*. The exact description of *SharedObject* can be found in [12]. The overriding of get and set methods is necessary, because the states of the infectious patients should be distinguished from the states of non infectious patients.

3.1 The temporal logic specification

It is clear from the description of the example, that every patient (which is represented by object P_i) can be in one of the following states: N_i (normal), T_i (trying) and S_i (surgery). However, the S state of the infectious patients should be distinguished from the S state of the non infectious patients (so let it be C), because the presence of an infectious patient precludes the possibility of the presence of any other patient.

Using the set of states means that the states of entity P_i (a sick or an infectious) are in set $\{N, T, S, C\}$ (the appropriate atomic propositions are N_i, T_i, S_i and C_i).

An interconnection relation I is introduced to store the process pairs needed to be synchronized. $I(i, j)$ iff processes i and j are interconnected (see [10]).

The temporal logic formulas, which define the restrictions that the system should satisfy, are the following (information about temporal logic can be found in [2], [3], [5], [9], [10] and [11]):

1. initial state (Every object is initially in its normal state. The statement is trivially true for the objects that are not entered into the system yet, because they are not in I):
$$\bigwedge_i N_i$$
2. it is always the case that any move that P_i makes from its normal state leads into its trying state, and such a move is always possible:
$$\bigwedge_i AG(N_i \Rightarrow (AY_iT_i \wedge EX_iT_i))$$
3. it is always the case that any move that P_i makes from its trying state leads into its surgery or infectious in the surgery state:
$$\bigwedge_i AG(T_i \Rightarrow (AY_i(S_i \vee C_i)))$$
4. it is always the case that any move that P_i makes from its surgery state leads into its normal state, and such a move is always possible:
$$\bigwedge_i AG(S_i \Rightarrow (AY_iN_i \wedge EX_iN_i))$$
5. it is always the case that any move P_i makes from its infectious in the surgery state is into its normal state, and such a move is always possible:
$$\bigwedge_i AG(C_i \Rightarrow (AY_iN_i \wedge EX_iN_i))$$
6. P_i is always in exactly one of the states N_i, T_i, S_i or C_i:
$$\bigwedge_i AG(N_i \equiv \neg(T_i \vee S_i \vee C_i))$$

$$\bigwedge_i AG(T_i \equiv \neg(N_i \vee S_i \vee C_i))$$

$$\bigwedge_i AG(S_i \equiv \neg(N_i \vee T_i \vee C_i))$$

$$\bigwedge_i AG(C_i \equiv \neg(N_i \vee T_i \vee S_i))$$

7. P_i does not starve:

$$\bigwedge_i AG(T_i \Rightarrow AF(S_i \vee C_i))$$

8. a transition by one process cannot cause a transition by another:

$$\bigwedge_{ij} AG((N_i \Rightarrow AY_jN_i) \wedge (N_j \Rightarrow AY_iN_j))$$

$$\bigwedge_{ij} AG((T_i \Rightarrow AY_jT_i) \wedge (T_j \Rightarrow AY_iT_j))$$

$$\bigwedge_{ij} AG((S_i \Rightarrow AY_jS_i) \wedge (S_j \Rightarrow AY_iS_j))$$

$$\bigwedge_{ij} AG((C_i \Rightarrow AY_jC_i) \wedge (C_j \Rightarrow AY_iC_j))$$

9. the infectious in the surgery state of a process precludes the surgery and the infectious in the surgery states of the other processes:

$$\bigwedge_{ij} AG((\neg(C_i \wedge C_j)) \wedge (\neg(C_i \wedge S_j)) \wedge (\neg(S_i \wedge C_j)))$$

3.2 Synthesis of the synchronization skeleton

The synthesis of the synchronization code is processed by an object-oriented extension ([12]) of P. C. Attie's and E. A. Emerson's method ([10]), after building the synchronization skeleton of a pair-system by E. A. Emerson and E. M. Clarke's method ([9]), so the abstract synchronization code of the full system is generated. Object-oriented techniques can be found in [7] and [14].

The synchronization skeleton generated by the method related to systems consisting objects is shown in Figure 2. Notation X_i means that object i is in state X, namely $((SynthesisObject)objs.get(i)).getState() == X$.

An infectious patient never can be in state S, and a sick never can be in state C, so these states may be removed from the synchronization skeletons of the appropriate objects. The result is shown in Figure 3.

3.3 Implementation

Let us consider the problem of writing and reading I. The methods used for reading and writing I can be given too; these methods are practically static methods of class *SharedObject*.

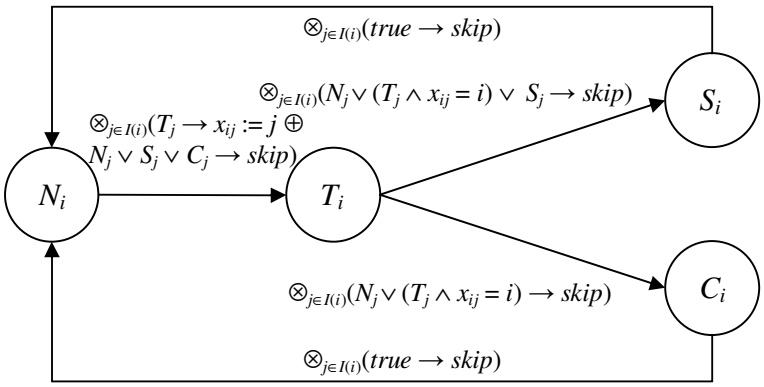


Figure 2: Synchronization skeleton of the example

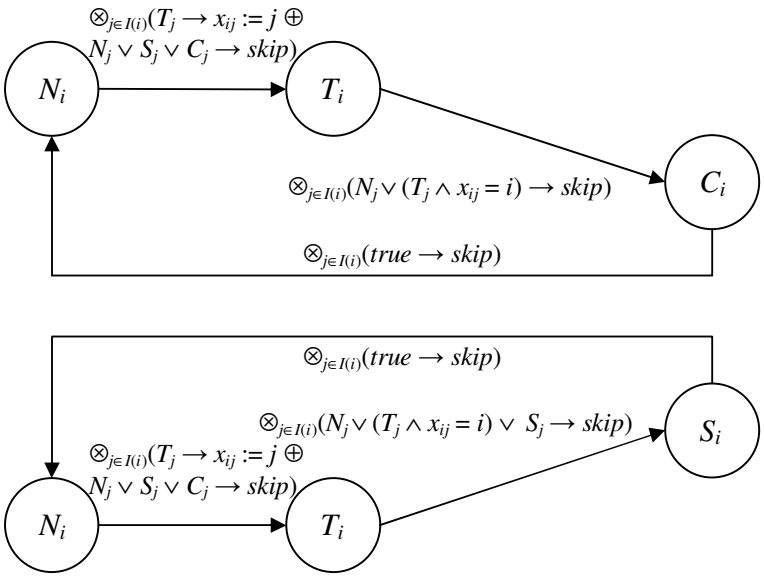


Figure 3: Final synchronization skeleton of Infectious (above) and Sick (below)

Of course, the case is not enabled when I is being changed by an object and I is being read by an other object at the same time. This means that an object can not evaluate transition conditions while an other object is changing I. Furthermore, writing I has to have priority against reading I. To implement these restrictions let us introduce a counter named *readCount* to count the objects reading I, and a counter named *writeCount* to count the objects writing or going to write I as well as counter *readWait* to count the objects which are waiting for I to read. Moreover, let us introduce two semaphores named *readSem* and *writeSem*. Let us consider the possible cases:

- If an object wants to read I and *writeCount* is zero then *readCount* should be incremented by one and the object is allowed to read I.
- If an object has finished reading I then *readCount* should be decremented by one

and if *readCount* is zero but *writeCount* is positive then the first object sleeping on *writeSem* should be awaken.

- If an object is going to read I but *writeCount* is positive then *readWait* should be incremented by one and the object is put to sleep on semaphore *readSem*.
- If an object is going to write I and *readCount* is zero and *writeCount* is zero then *writeCount* should be incremented by one and the object is allowed to write I.
- If an object has finished writing I then *writeCount* should be decremented by one and the following cases are possible:
  - If *writeCount* is positive then the first object that is sleeping on semaphore *writeSem* should be awaken.
  - If *writeCount* is zero but *readWait* is positive then the first object is sleeping on semaphore *readSem* should be awaken.
- If an object is going to write I but *readCount* is positive or *writeCount* is positive then *writeCount* should be incremented by one and the object is put to sleep on semaphore *writeSem*.

The changes of the counters and condition evaluations must work in mutual exclusive mode so these operations must be protected by a semaphore named *mutex*. Before every mentioned operation *mutex* should be let down and *mutex* should be lift up before an object is put to sleep. According to this, we must not lift up *mutex* when an object wakes up another object but we must lift up the semaphore if no another object will be awaken. Furthermore, *readWait* should be decremented by one before a reader object is awaken.

Let us deal with the evaluation of conditions, namely method *setState* in the following. To produce method *setState*, the abstract program of the synchronization, which is a finite deterministic automata, is given by the algorithm. Then we make the condition checker part on the basis of the conditions in the automata and if a given condition is fulfilled then we execute the action part associated with the condition. The automata may be given by a list of the transitions. Only one transition can be generated by the synthesis between two states, so a transition may be built from the following elements: start state, end state, condition (in Polish form expression in order to simplify the evaluation), the list of the operations on the shared variables.

We have to solve the problem of synchronization of the condition evaluation and the execution of the actions belonging to the conditions. Method *setState* uses the values of the shared variables and may change the variables, too, in case the transition is enabled. That is why the shared variables should be changed by at most one object simultaneously. Let us notice that this restrictioin is not enough, because if an object A has evaluated the condition of a transition and finds out that the transition is enabled then object B

changes the values of the shared variables before A would do the transition and so the system may be in inconsistent state. That is why we have to assure that an object can not start evaluating a condition while another object is trying to process a transition (namely, the object has started the evaluation and has not done the action).

Some level of exclusion has to be provided in order to evaluate the conditions, namely, no two objects can be in their condition evaluating phase at the same time.

To solve this issue, let us introduce a token for every connection of every object. Then if an object is going to change its state – so it is going to evaluate a condition – it must ask the tokens of all the objects connected to it. Hence, every element in I has a *token* attribute and a *captureToken* and a *releaseToken* method. The token is a reference to a *SynthesisObject* type object, and its value shows which object owns the token. Value *null* indicates that the token is not owned by any object. The return value of *captureToken* may be *true* or *false*. Value *true* indicates that the token is successfully got, and *false* indicates that the token is reserved. Method *captureToken* works in mutual exclusive mode.

Possibility of deadlock arises in progress of obtaining tokens. Deadlock can be avoided if an object drops all tokens that it owns if it tried to get a token from an object that is already waiting for a token, and the object restarts obtaining token some time later after dropping. It is clear that this implementation may lead to livelock: let us suppose that objects a, b and c are going to obtain tokens from each other. Let a get the token from b, b from c and c from a. Then let a ask the token from c. It is not possible, so a drops all the tokens it owns. Then let c try to get the token from b. It fails, so c drops its tokens too. Then only b has any token. Then let a get token from b, and c from a, then start this process again with a simple modification so that c will be the only object that owns any tokens. And so on.

We mention a method to avoid the possibility of livelock. The method is the introduction of a binary semaphore that is let down by every object for the time while it is trying to obtain tokens. If an object can not get a token then it releases all tokens it got and lifts up the semaphore. The implementation of this semaphore practically should be placed in *SynthesisObject*, because the obtaining of tokens is associated with I. In this case only one object is able to obtain tokens at the same time, so livelock can not take place.

According to the above, taking the abstract code produced by the synthesis into consideration, the algorithm will generate the following concrete code for the class *Sick* (for lack of space only the method *SetState* is considered here; the complete source code can be downloaded from `http://sleet.web.elte.hu/files/surgery.zip`):

```
public class Sick implements SynthesisObject {
...
public void setState(int value) throws Exception {
```

```
boolean succed = false;
SynthesisObjectPair sop;
SynthesisObject so;
if ( !(((state == N) && (value == T)) || ((state == T) && (value == S)) ||
    ((state == S) && (value == N))) )
    throw new Exception("Invalid state transition");
while ( !succed ) {
    succed = true;
    while ( !captureToken() )
        Thread.sleep(1);
    try {
        if ( (state == N) && (value == T) ) {
            for ( int i = 0; i < SharedObject.getICount(); i++ ) {
                sop = SharedObject.getI(i);
                if ( sop.belongToObject(this) ) {
                    so = sop.getOtherObject(this);
                    if ( !(so.getState() == T) && !((so.getState() == N) ||
                         (so.getState() == S) || (so.getState() == C)) )
                        succed = false;
                }
            }
            if ( succed )
                for ( int i = 0; i < SharedObject.getICount(); i++ ) {
                    sop = SharedObject.getI(i);
                    if ( sop.belongToObject(this) ) {
                        so = sop.getOtherObject(this);
                        if ( so.getState() == T )
                            sop.getSharedObject().setV_1(so);
                    }
                }
        }
        if ( (state == T) && (value == S) ) {
            for ( int i = 0; i < SharedObject.getICount(); i++ ) {
                sop = SharedObject.getI(i);
                if ( sop.belongToObject(this) ) {
                    so = sop.getOtherObject(this);
                    if ( !((so.getState() == N) || ((so.getState() == T) &&
                         (sop.getSharedObject().getV_1() == this)) ||
                         (so.getState() == S)) )
                        succed = false;
                }
```

```
                }
            }
            if ( (state == S) && (value == N) ) ; // nop
        }
        finally {
            releaseToken();
        }
        if ( !succed )
            Thread.sleep(10);
    }
    state = value;
}
}
```

4 Conclusion

It is clear from the foregoing, that the described method can be applied with ease for generating the synchronization code of object-oriented systems, so it is unnecessary to code the synchronization by hand.

The state explosion problem is successfully avoided, although, the generated code become more difficult and less effective with the increasing number of classes.

5 Future work

Considering that the synchronization skeleton of individual objects may contain states which can never be taken, the deadlock checker algorithm (the algorithm is detailed in [11]) may result that deadlock is possible, nevertheless deadlock freedom would be set out in the original system. Consequently, deadlock checking possibilities and extra work needed to manage the above issue should be considered.

The implementation of classes *SharedObject*, *Semaphore* and *SynthesisObjectPair* can be applied directly in any system synthesized by the method described above. The implementation of the descendants of *SynthesisObject* should be generated, the implementation of the generator program is in progress.

References

[1] G. R. Andrews, A Method for Solving Synchronization Problems, Science of Computer Programming 13 (1989/90) pp.1-21

[2] Z. Manna, P. Wolper, Synthesis of Communicating Processes from Temporal Logic Specifications, ACM TOPLAS 6 (1984) pp. 68-93

[3] É. Rácz, Specifying a Transaction Manager Using Temporal Logic, In: Proc. of the Third Symposium on Programming Languages and Software Tools, Kaariku, Estonia, (1993) pp. 109-119

[4] L.Kozma, A Transformation of Strongly Correct Concurrent Programs In: Proc. of the Third Hungarian Computer Science Conference (1981) pp. 157-170

[5] F. Kröger, Temporal Logic of Programs, Springer-Verlag, Berlin Heidelberg (1987)

[6] Z. Horváth, The Formal Specification of a Problem Solved by a Parallel Program - A Relational Model, Annales Univ. Sci. Budapest, Sect. Comp. 17 (1998) 173-191

[7] L. Kozma: Shared Data Abstractions, In: Proc. of Fourth Hungarian Computer Science Conference, M. Arató, I. Kátai, L. Varga (eds) Győr, pp. 201-210, (1985)

[8] Kozma László, Varga László, Párhuzamos rendszerek elmezése ELTE, TTK, Informatikai Tanszékcsoport, 2002

[9] E. A. Emerson & E. M. Clarke, Using branching time temporal logic to synthesize synchronization skeletons. Science of Computer Programming, 2, pp. 241 - 266

[10] P. C. Attie & E. A. Emerson, Synthesis of Concurrent Systems with Many Similar Processes, ACM TOPLAS Vol. 20, No. 1, (January 1998) pp. 51-115

[11] P. C. Attie & E. A. Emerson, Synthesis of Concurrent Programs for an Atomic Read/Write Model of Computation ACM TOPLAS Vol. 23, No. 2, (March 2001) pp. 187-242

[12] Sz. Hajdara, L. Kozma and B. Ugron, Synthesis of a system composed by many similar objects, Analess Univ. Sci. Budapest., Sect. Comp. 22 (2003) (Under publishing)

[13] J. Rumbaugh, M. Blacha, W. Premerlani, F. Eddy, W. Lorensen Object-Oriented Modelling and Design, Prentice Hall Inc. (1991)

[14] R. Kurki-Suonio, Fundamentals of object-oriented specification and modeling of collective behaviors, Object-Oriented Behavioral Specifications (Eds. H. Kilov and W. Harvey), Kluwer, (1996) pp. 101-120

[15] Aspect-Oriented Software Development, `http://www.aosd.net`

Szabolcs Hajdara, Balázs Ugron
Department of General Computer Science
Eötvös Loránd University
XI. Pázmány P. sét. 1/c.
H-1117 Budapest, Hungary

Composing Non-Orthogonal Aspects

Andreas I. Schmied, Franz J. Hauck

Distributed Systems Laboratory · University of Ulm, Germany
{schmied, hauck}@informatik.uni-ulm.de

Separation of concerns is a well known and accepted strategy to handle the vast and various aspects of complex systems. Especially, distributed systems contain lots of cross-cutted and tangled code fragments to fulfill their services. Merging formerly separated aspect code with program code by means of aspect-oriented programming is enabled through a couple of available technologies that manipulate program sources. Unfortunately, these tools or aspect weavers both operate on a distinct coarse-grained level (types, methods) and fulfill only a restricted a-priori known set of manipulations.

In contrast, we want to weave several aspect code fragments, which could have been constructed by independent teams, for more than one concern simultaneously. A composition that not only concatenates aspects, but also manages joint effects between them, reveals several complex, possibly interfering weaving demands.

To give feasible development support, our existing code-weaving framework *ADK* will be extended to cope with that widely discussed problem of composing conflicting concerns. We propose the concept of transformation processes. Each code conversion step is paired with some meta-level object that holds detailed information about its semantics (using pre- and postconditions) and the composition demands. This information allows the ADK to decide automatically, whether a complete composition of transformations is executable or not.

1 Introduction

Distributed systems contain a vast amount of cross-cutted and tangled code fragments, e.g., to provide internal synchronization and to support common services like location transparency, transactionality, persistence, live-cycle management, and for us most important, fault tolerance. Separation of concerns [Parn72] is a well known and accepted strategy to handle the arising complexity of the underlying source code.

On the particular field of quality-of-service–aware middleware, one of our [Apx] goals is to weave *non-functional* aspect code of the previously mentioned concerns jointly into actual, thus *functional*, program code. This shall even be possible if the aspect code was in parts developed by independent, mutually unknown teams.

In general, middleware frameworks are giving aid during the process of developing distributed systems. For instance, their development frameworks provide support with specialized generators that build the adapters needed to integrate functional code (e.g., business entities and logic) into the middleware runtime environment. Usually the shape and extent of these adapters is controlled either by an external description of what has to be generated (cp. *CORBA-IDL* [OMG1]) or according to agreements at the language (cp. *Java-RMI* with *Remote Interfaces* [Sun1]), or

maybe at the compiler level (cp. stub compilers for both CORBA-IDL [OMG2] and RMI [Sun2]).

All those similar forms of code generation have one characteristic property: they just *add* certain code fragments to existing code. Usually this is even stricter in terms of adding whole new translation units to a project, i.e., entire new files. In contrast to these additions, complex conversions inside the code are rarely done, at best changing the inheritance hierarchy when introducing a special base-class usage. This conversion is hardly ever done by means of transformation tools, but often manually by the programmer.

Adding files and small code enhancements seems to be sufficient for most daily cases of developing middleware-based software, except for the task of merging totally cross-cutted aspect code with program code. In the latter case, the code manipulator faces the challenge of weaving code fragments at a more fine-grained level into the original program.

One yet simplified example might be to add fault tolerance to all outgoing remote method calls of a class: Special added interceptors may delegate method calls to multiple servers, completely transparent to the programmer's perspective, and return one safely fetched result to the caller. This task is achievable following the paradigm of aspect-oriented programming (AOP) [Kicz97] using aspect weaving technologies like *AspectJ* [Kicz01], *Hyper/J* [TaOs01], or by using our own code weaver *AspectIX ADK* (application development kit) [Walt01][Nenn01]. The major benefit for the developer is, that the replication logic can be written separated from the original code as an aspect; it may then be used by means of aspect weaving for several classes that need replication.

The ADK provides similar functionality of code manipulation with small entities, named *weavelets*. They first of all allegorate fundamental manipulations like *AddInstanceVariable* or *CreateClass*. Each weavelet is designed as a Java class and instantiated per action during a weaver session. Code conversion is potentially ranging from large structures (modules) down to smallest language elements (statements, expressions, operators). Beyond these simple actions, weavelets can be composed to larger units—being weavelets themselves—which accomplish more complex procedures like *CreateDelegateClass* or *AddInvocationCounter*. The latter proceeds delegating to *AddInstanceVariable* and then appends an increment statement to each of the observable methods; the *AppendToMethod* weavelet, also being incorporated here as a sub weavelet, takes care of potential returns or throwable exceptions as a pitfall during the task of "appending to a method's code".

All mentioned technologies provide some mechanism for constructing such chains of weaving commands. Critically, the programmer must exactly know all details about the intent and the effect of each single manipulation step in chain. If we would introduce another code converting process to the above replica example, e.g. providing some caching facility, it would be necessary to declare where or when it should have effect: is the one result picked out of many replicas or each single to be cached. Adding another feature, e.g. a security module, one may encounter the former replication code incompatible with the security code; used together, they may irregularly change the semantics of our program.

A fact of utmost importance for all aspect weavers is their semantics-preserving behavior. However, if the semantics of the origin program needs some adjustments, these should be manageable and deterministic. Hence, composition may be viable with some fully-controlled self-made code manipulations, but grows in complexity with the growth of your aspect code and the amount of separated concerns. It is getting even harder to manage, if you have no or restricted access to the aspect code, that was maybe developed by another team.

We try to solve that problem of composing independently created aspect code with several extensions to our existing tool: weavelets will have reflective data annotated, sufficient to make their joint effects comprehendable and manageable for the ADK runtime.

In the next sections we will present a summary of composition problems, outline our proposed solution, and give some more in-depth view of implementational enhancements.

2 Problem Statement

After the last thirty years of intense discussions about how to identify and how to separate cross-cutting concerns [Parn72][TOHS99], the scientific community strengthened its activities examining how these separated concerns might again be plugged together. This is very challenging especially for non-orthogonally interacting concerns [Silv99], which shall be our problem class of interest.

For this very class of concerns a simple concatenation of weaving operations is not sufficient. One first reason that renders a simple solution impossible are dependencies between these operations. Even independent weaving operations might not be executable in arbitrary order, as the result would not be the expected, because not all operations are commutative regarding the program semantics.

A second and even more interesting reason is that manipulations of one weaving process might conflict with the demands of another or be mutually exclusive [BTGA00]. For instance, the op-

erators of a first manipulation process may delete a certain class that is to be touched by a second one. Thus, attention should be paid to the order of our manipulations: even though the class might have been deleted afterwards, both processes could have taken place consecutively if they were executed in reverse order.

Beyond this, a programmer has to decide which mutual influence chained manipulation processes have on each other. For example, a programmer wants to insert some trace code into *every* method body and some security code that introduces new methods. Both transformations may be applied in arbitrary order, but the developer may prefer a specific one, as he wants to decide, whether the newly added security code is also to be traced or not. In general, some essential factors are not clearly defined:

- the scope of some target quantifier, e.g. the quantifier "*all methods",*
- the point in time, when each single one in a chain of manipulations takes effect,
- the fixed point of recurring weaving processes, e.g. *tracing of traced code*.

For sensible and effective development, a programmer has support for the separation of aspect code from the actual program and for merging aspects back later on, e.g. by aspect languages and weavers. This support is sufficient if the programmer has full control over the entire code, but not if parts were created by unrelated development teams and not entirely revealed. It should be possible to take these parts, *add some information* about how they have to be woven into the code, and yield an acceptable and semantically correct program with enhanced features.

In our current research we address the following questions:

- What does this information cover and how is it declared?
- Is this information sufficient to permit at least semi-automatic weaving?
- Up to what extent is it possible to weave automatically?
- How can the automation process be maximized?
- How does the user intervene to deal with the remaining effort in manual control?

This set of prerequisites should be answered to achieve our main purpose: making domain-independent quality-of-service modules offered by multiple developers integrateable among others within domain-specific application code.

3 Sketch of a Solution

We will start with some initial research, unveiling our project-specific needs in composing concerns and weaving them jointly. Next, we continue identifying the elementary operations to ful-

fill all required manipulation tasks. These operations will be categorized by demands and effects, more precisely by pre- and postconditions.

It will be a recurring process due to changes in the demands of other AspectIX subprojects, which will benefit from the development paradigm outlined in this paper.

These categorizations are necessary preliminaries for the concept of *transformation processes* (TPs), which is meant to give us detailed control over the behavior during complex arrangements of weaving processes. It is comparable to process algebras as used in CSP [Hoar85], but here specifically used in the context of aspect-oriented programming [Andr01].

Every distinct step of manipulation will be equated with a corresponding transformation process at some meta level. A sequence of steps, thus a sequence of TPs, will itself be a TP. Relating to the earlier depicted categorization of operations, each TP will have its pre- and postconditions affixed. As a major consequence, each sequence of TPs—being itself a TP—will also have pre- and postconditions (cp. [Robe99]), though computed ones as drafted in the following formula:

$$(pre_{all} \equiv \phi(pre_1 ... pre_N)) \rightarrow (post_1 \in pre_2) \rightarrow \ldots \rightarrow (post_{all} \equiv \chi(post_N))$$

ϕ is the function that estimates the computed preconditions for all TPs in chain including influences by prior TPs, respectively.

χ is the function that computes an effective postcondition which includes all influences introduced by all previous TPs in chain.

Each step of code conversion is itself implemented by means of refactoring [Fowl97]. Hence, all of its implications on changes of structure and semantics of the program are known before and under control; for instance, the bitter case of inconsistent method renaming will not occur, as it will never be done lacking similar changes to all the clients' call statements. In our opinion, a sequence of manipulation steps can be arranged in a stable fashion, if each self-contained fundamental step is safely conducted based on refactoring.

The crucial point is the impact on these "reliable" manipulations within a composition of TPs. We believe that a stable composition of TPs can be accomplished, if every single TP is stable and all of them fulfill certain composition predicates by pairs. Once more compared to CSP, this is equivalent to a proper *parallel composition* of process automata with certain join points.

Starting with the first in chain of each TP being composed, we try to place them piece by piece in serial order. TPs changing disjoint subsets of the code may run in parallel and join afterwards to a single TP thread at some barrier. If certain TP parts are not composable, then either an error

must be signalled to the user and the entire transformation must be revoked, or some alternative must be chosen to correct the problem—if at all possible.

4 Implementation

The mechanisms, needed to implement the concept of transformation processes, will be outlined in this section. We present the deficiencies of our existing toolkit and some solutions that might be appropriate for full TP support in the future.

We need two types of control mechanisms to achieve this: namely a description of what has to be manipulated and how to put it into practice, and a way of composing multiple code modifications and govern their joint effects.

Regarding the first need, our ADK provides yet a good basis for source code manipulation but has several limitations, highlighted below. A major change will be to encapsulate each single weavelet into an entity (TP object) representing a transformation process. All these TP objects will be visible on a meta layer (cp. computational reflection [Maes87]) and give an insight view on the effects gained by the underlying manipulations. Handling corrigible collisions between TPs, the meta layer will offer several interfaces for the management and the optimization of transformations.

Regarding the second need, assembling weavelets to larger units seems handy and achievable for simple, non-conflicting manipulations, except for interfering ones. Therefore, we require some mechanism and a deployment language to declare how composable manipulations get in touch with each other.

The deployment language needs the ability to express predicates on the composition and competition on TPs. Examples are *depends*, *(not) after/before*, *not–in–parallel–with*, *idempotent*, *moveable–in–chain*, and further constraints. To find an appropriate set of predicates is subject of our research.

Furthermore, each weavelet has to express its effects at some abstract language level, regarding both the structural changes and those on program semantics. Compared towards the semantic analysis of the actual transformable program, it should be possible to understand clearly what effect each transformation step and all TPs together have on a certain program. As semantic analysis in general is very difficult, the ADK will provide some interface, that allows domain-specific analysis modules being plugged in. Then, each self-contained and well-described TP can offer its own mechanism of analysis, enabling the generic ADK runtime proving its compatibility with the program and concurring TPs.

The concept of recursively nested transformation processes gives us great opportunities to optimize the overall process and to fix some current ADK problems, one of which is explained next. The before mentioned AppendToMethod weavelet is a grateful example: one possible implementation would be to wrap the whole former method code with a *try–finally sequence*. For a simple implementation, the next AppendToMethod would wrap the freshly generated sequence redundantly by its own. To inhibit this behavior, a reusage technique was invented that splits the creation of administrative code—e.g. the before mentioned sequence—and the addition of the code being implanted. With this solution an acceptable range of reusage problems are handled; unfortunately the mechanism fails if other weavelets change certain code parts close-by.

It should be possible to run advanced restructuring algorithms on the graph of transformations. This strongly depends on understanding the effects and semantics of our manipulations. As an example, obsolete weavelets could be detected and eliminated (comparable to the removal of loop invariants during compilation) if the targets they touched are to be deleted by a competing weavelet. Certainly, this is only permitted, if the eliminated weavlet has no side effects in evidence. Other weavelets could be reordered or grouped together optimizing the runtime of a long lasting transformation on large projects. The before mentioned unstable mechanism for weavelet reusage could be substituted by grouping several code-appending TPs as sub entries under one singleton providing the administrative structures. Incompatibilities between TPs could be resolved by rearranging them at non-conflicting positions in the chain of composed TPs, provided their given constraint predicates allow us to do so.

Further on, if it is not a–priori recognisable, which additional code will be created, one faces the problem, that an early running on–all–classes TP will not include sources generated afterwards. We need to solve the problem of a fluctuating set of source code.

We propose the encapsulation of all transformation processes within nested spheres of transactions. Thus, each TP runs inside a transaction and also does each sequence of TPs up to the top-level transaction incorporating all others. The trick how to cope with that fluctuating source code is, that a formerly run process can be (partially) rolled back, restructured on demand to grasp the newly generated code, and then be restarted, henceforth considering the completed code.

In addition to using transformation processes inside the ADK machinery, we plan some further enhancements on the toolkit enlarging the usability of its frontend:

We provided an easierly understandable interface to the weavelet architecture. Weavelets can be assembled using an XML-based script, wherein every weavelet action can be represented by an XML element, named according to the weavelet's class name, and may contain nested weavelet elements inside. During interpretation of such a script, the nodes of its DOM representation are replaced in memory by the respective weavelet objects, and chained with sub weavelets if necessary — we call this process, *weavelet tree expansion*.

The ADK next generation aims to treat both the Java and the XML variant equally, by presenting a dual, mutually convertible interface of both, but each with the same features instead of the latter one being just a frontend to Java based weavelets. Analogous to the *inversion* of Java Server Pages to Servlets [Sun3], an XML based transformation script should be transformable into Java code and vice versa; the way back will for sure be the more ambitious one and existing technologies like *XSL Transformations* [Xslt] and the *Tag Library* concept [Sun4] need to be surveyed for sufficience.

Nevertheless, all supported manipulations must be described in a low level manner, but without abstract concepts like point-cuts as found in actual aspect languages [Kicz01]. It still has to be discussed, if we might need a high-level aspect-language for our application area and, if any, whether to invent this language ourselves or to use an existing and approved one as a frontend. In addition to existing languages, we need phrases to formulate *weavelet templates*, as the invention of a TP and the application of its instances on a project's code are supposed to be independent.

Currently all those ideas regarding the ADK were, resp. will be, realized in Java. We intend to abolish this restriction by making the complete process language-independent: Comparable to a *register transfer language* (RTL), used as an intermediate representation of a program during compilation, weaving targets will be parsed and mapped to a *meta grammar*, supporting the style and semantics of ordinary object oriented languages.

The development paradigm of utilizing weavelet-based AOP shall be entirely embedded in a "development process", which first of all means to us, that the developer roles are discriminable and the output of our toolkit is reasonable, understandable, debuggable code without the taste of wizardry.

5 Conclusion

Writing code transformations on behalf of aspect weaving is a complex but practicable way to integrate the code of separated concerns back into ones functional code. Modern software sys-

tems rely on multiple aspects. It is not enough to weave them independently into functional code, instead they have to be composed to achieve some joint effect. Interfering weaving demands, in particular those of independently developed aspect code, render solutions impossible that simply concatenate the aspects with each other.

To cope with dependencies, exclusiveness and causality of composed code transformations, we propose the concept of *transformation processes*. Based on a description of the transformation semantics, pre- and postconditions of each TP step, the next generation of our existing code manipulation tool ADK will be able to control TP behavior and effects during a transformation session. Further on, we plan to use the verbose description of TPs for static and on-demand restructuring to achieve an optimized runtime behavior during transformation of large projects and to solve dynamically occurring insufficiencies or incompatibilities between TPs.

References

Andr01 James H. Andrews, Process-Algebraic Foundations of Aspect-Oriented Programming. *Proc. of the 3rd Int. Conf. on Metalevel Arch. and Sep. of Crosscutting Concerns (Reflection 2001)*, Kyoto, Japan, Sept.2001, Springer LNCS(2192), pp.187-209

Apx Website of the AspectIX project: http://www.aspectix.org

BTGA00 Bergmans, Tekinerdogan, Glandrup, Aksit: On Composing Separated Concerns, Composability and Composition Anomolies, *ACM OOPSLA'2000 workshop on Advanced Separation of Concerns*, Minneapolis, Oct. 2000

Fowl97 Fowler, M. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Reading MA, 1997 — also see: http://www.refactoring.com

Hoar85 C.A.R. Hoare: *Communicating Sequential Processes*, Prentice Hall, 1985

Kicz01 Kiczales, G., Hilsdale, E., Hugonin, J., Kersten, M., Palm, J. & Griswold, W. G., An Overview of AspectJ, in J. L. Knudsen, ed., *ECOOP 2001 — Object-Oriented Programming*, Vol. 2072 of LNCS, Springer-Verlag.

Kicz97 G. Kiczales, et. al.: Aspect-oriented programming. *Proceedings of ECOOP '98, Lecture Notes in Computer Science (LNCS 1241)*. Springer-Verlag, June 1997.

Maes87 Pattie Maes: Concepts and experiments in computational reflection. *Conference proceedings on Object-oriented programming systems, languages and applications*, Orlando, Florida, USA, 1987, ACM Press, pp. 147-155

Nenn01 Andrea Nenni: *Design und Implementierung eines Code-Transformators für den Entwurf verteilter Objekte*. Diploma thesis DA-I4-02-04, Informatik 4, Univ. of Erlangen-Nuremberg, Germany, 2001.

OMG1 Object Management Group (OMG): *Common Object Request Broker Architecture: Core Specification*, Chap.3: OMG IDL Syntax and Semantics, Dec. 2002
http://www.omg.org/technology/documents/formal/corba_2.htm

OMG2 Object Management Group (OMG): *OMG IDL: Details*
http://www.omg.org/gettingstarted/omg_idl.htm

Parn72 D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053-1058, Dec. 1972.

Robe99 Don Roberts: *Practical Analysis for Refactoring.* PhD thesis, University of Illinois at Urbana Champaign, 1999

Silv99 António Rito Silva: Separation and Composition of Overlapping and Interacting Concerns. *OOPSLA'99 First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems*. Denver, Colorado, USA, Nov. 1999

Sun1 Sun Microsystems Inc.: *Java Remote Method Invocation (RMI) Home Page*,
http://java.sun.com/products/jdk/rmi/

Sun2 Sun Microsystems Inc.: *rmic - The Java RMI Compiler*,
http://java.sun.com/j2se/1.4.1/docs/tooldocs/solaris/rmic.html

Sun3 Sun Microsystems Inc.: *Java Server Pages — Dynamically Generated Web Content*
http://java.sun.com/products/jsp/

Sun4 Sun Microsystems Inc.: *Java Server Pages — Tag Libraries*
http://java.sun.com/products/jsp/taglibraries.html

TaOs01 Peri Tarr , Harold Ossher, Hyper/J: multi-dimensional separation of concerns for Java, *Proceedings of the 23rd international conference on Software engineering*, p.729-730, May 12-19, 2001, Toronto, Ontario, Canada

TOHS99 Tarr, Ossher, Harrison, Sutton: N Degrees of Separation: Multi-Dimensional Separation of Concerns. *Proceedings of the International Conference on Software Engineering*, Los Angeles, California, USA, May 1999

Walt01 Christian Walter: *Ein Werkzeug für modulare Code-Transformationen zur Entwicklung verteilter AspectIX Objekte*. Diploma thesis DA-I4-01-08, Informatik 4, Univ. of Erlangen-Nuremberg, Germany, 2001.

Xslt W3C Recommendation 16: *XSL Transformations (XSLT)*, Version 1.0, Nov. 1999
http://www.w3.org/TR/xslt