Service-based Modernization of Java Applications

BY

GIACOMO GHEZZI
Bachelor in Computer Science, Politecnico di Milano, 2004
Master of Science in Computer Science, Politecnico di Milano, 2007

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2007

Chicago, Illinois

ACKNOWLEDGMENTS

First of all I would like to thank my advisor, professor Luciano Baresi for giving me the possibility to do this thesis and for supporting and helping me throughout the entire work, making it possible.

I am indebted to my many student colleagues I met during the years spent at Politecnico di Milano for providing a stimulating and fun environment in which to learn and grow. I am especially grateful to Simone Imeri, with whom I wrote the Bachelor's thesis and studied for several exams, Paolo Beretta and Lorenzo D'Ercole for helping me during some hard times during the semester spent at UIC (University of Illinois at Chicago), especially in Numerical Analysis.

I wish to thank all my past and present friends (you know who you are) for all the fun time spent together, for helping me get through the difficult times and for all the emotional support, comraderie, entertainment and caring they provided.

I wish to thank my entire family, my sisters and my parents, for providing a loving environment and for always being there for me.
A particular "thank you" goes my parents. They bore me, raised me, supported me, taught me

and loved me in every day of my life.

To them I dedicate this thesis.

TABLE OF CONTENTS

CHAPTER			PAGE
1	INTRODUCTION		1
	1.1	Contribution	2
	1.2	Structure	5
2	SOFTWARE MODERNIZATION		6
	2.1	Legacy software	6
	2.2	Transform, replace, rewrite or re-use?	10
	2.2.1	White-Box modernization	15
	2.2.1.1	Legacy application transformation	16
	2.2.1.2	Legacy application rewriting	22
	2.2.2	Legacy application replacement	23
	2.2.3	Black-Box modernization	26
	2.2.4	Final notes	30
3	SOFTWARE ARCHITECTURE ANALYSIS		32
	3.1	Architecture recovery through static analysis	34
	3.2	Architecture recovery through dynamic analysis	41
	3.2.1	Instrumentation	43
	3.2.1.1	Log4J and Java API logging	44
	3.2.1.2	Program instrumentation via Aspect Oriented Programming	46
	3.2.1.3	Other solutions	48
	3.2.2	Mapping from runtime observations to architecture	51
	3.2.2.1	A survey of existing studies and tools	53
	3.3	A comparison between static and dynamic analyses	68
4	MY APPROACH		79
	4.1	Overview of the proposed solution	80
	4.1.1	Architecture extraction	80
	4.1.1.1	DiscoTect	82
	4.1.1.2	Probe aspects	89
	4.1.1.3	DiscoSTEP specifications	91
	4.1.2	Architecture visualization	103
	4.1.2.1	A quick tour of AcmeStudio	103
	4.1.3	Transformation	106
	4.1.3.1	Java Web service implementation	108
	4.1.3.2	Transformation method	116
	4.1.4	Web service creation	117

TABLE OF CONTENTS (Continued)

CHAPTER			PAGE
5	**DISCOTECT**		122
	5.1	Probe aspects	122
	5.1.1	Implementation	123
	5.1.2	Problems encountered	133
	5.2	DiscoSTEP	134
	5.2.1	Implementation of DiscoSTEP specifications	134
	5.2.2	Problems and limitations	160
	5.3	The sample applications	163
	5.3.1	Socket client-server	163
	5.3.2	RMI client-server example	164
6	**MODERNIZER**		167
	6.1	Reasons for using Eclipse	167
	6.2	The plug-in in depth	168
	6.2.1	Convert Action	168
	6.2.2	Compile Action	170
	6.2.3	Run Action	173
	6.2.4	Transform Action	173
	6.2.5	Web Service Deployment	184
	6.2.6	Plug-in preferences	189
7	**CONCLUSIONS**		202
	7.1	Open issues	204
	7.2	Future work	205
	CITED LITERATURE		207
	VITA		216

LIST OF FIGURES

FIGURE		PAGE
1	Overview of the tool, called Modernizer, described in this thesis.	3
2	Legacy system. What next? (Taken from D. Good(1)).	11
3	The combination of factors for choosing between transform, replace, rewrite or re-use.	13
4	The four main key busines objectives.	14
5	A model for legacy application transformations.	17
6	Information System life cycle.	26
7	Overview of the Reflexion Model approach.	37
8	The Dali workbench structure.	39
9	Trace metamodel.	69
10	Scenario diagram metamodel.	70
11	An example cel.	71
12	The cel next to the one shown in Figure Figure 11.	72
13	Summary view for the same system and execution trace of Figures Figure 11 and Figure 12.	73
14	A screenshot of Jive.	74
15	Screenshot of the Histogram view.	75
16	Screenshot of the Reference Pattern view.	75
17	Screenshot of the Execution view.	76
18	Portion of the zoomed Execution view.	77

LIST OF FIGURES (Continued)

FIGURE **PAGE**

19 Screenshot of the Call Tree view. 78

20 The modernization steps of my approach. 81

21 The DiscoTect structure. 84

22 Using AcmeStudio for architectural design. 105

23 The information about ServerEngine service. 119

24 The tester for the ServerEngine service. 121

25 Architecture extracted from the socket client-server. 165

26 Architecture extracted from the RMI client-server. 166

27 Screenshot of the beginning of the conversion process. 169

28 Window dialog with the list of architectures to choose from. 170

29 Overview of a converted project, with the newly created aspects, DiscoSTEP specifications and AcmeStudio file. 171

30 Message dialog at the end of the conversion process. 172

31 Screenshot of the compilation action. 192

32 Message dialog after the successful compilation of a DiscoSTEP specification (note the .epo file, which is the newly compiled specification). . . . 193

33 Message dialog after the showing the errors encountered while compiling a DiscoSTEP specification. 194

34 Message dialog after showing the available DiscoSTEP specifications that can be run. 194

35 An instance of the DiscoTect Runtime Engine running a DiscoSTEP specification. 195

36 The possible transformations for a studied example application. 195

LIST OF FIGURES (Continued)

FIGURE **PAGE**

37 The RMI to Webservice transformation window for the RMI example explained in Section 5.3.2. 195

38 An RMI to Webservice transformation window ready for the transformation. 196

39 Warning message at the end of the transformation process. 196

40 A translated RMI application. 197

41 Ant tasks for a newly created Web service. 198

42 A deployed Web service. 198

43 A deployed Web service. 199

44 The Modernizer preference page, with values set for a specific computer. 200

45 Error message of the compiler due to the missing Cygwin directory. . . . 200

46 Warning message at the end of the RMI to Webservice transformation process due to the missing JaxWs libraries directory. 201

47 Warning message at the end of the RMI to Webservice transformation process due to the missing JavaEE SDK directory. 201

48 Warning message at the end of the RMI to Webservice transformation process due to the missing application server deploy directory or one between the application server username, password or host fields. 201

SUMMARY

The aim of this work is to propose a technique that tries to automatically modernize Java applications into Web services by using the information extracted from the analysis of the runtime behavior of the application under study.

The purpose is to use and adapt existing architecture analysis tools and integrate them with others created from scratch, in a modernization environment prototype that aids and supports the engineer during the whole software modernization process, from the architecture extraction and visualization to the semi-automatic transformation into a new application and the deployment of it. In fact, many techniques to extract meaningful information from an existing system exist, but, as for now, all this useful information is only used for program understanding or, at most, to guide the engineer who will then manually modify the critical part of code. So, the design and implementation phases of this prototype will help to dig deeper into these issues, find what is doable and what is not and see what are the strengths and weaknesses of this approach.

First the main problems related to legacy software are exposed, then the four main modernization techniques (transform, replace, rewrite and re-use) and the reasons for choosing one of them are analyzed.

Then the two main software architecture analysis techniques, static and dynamic analysis, are studied, their advantages and disadvantages analyzed. For both of them a view of the current state of the art is also provided.

SUMMARY (Continued)

Finally the implementation of the prototype addressing the problem of automatic Web service based modernization of Java application is explained. In the beginning, the motivations and goals of the project are given, then the development and implementation of the tool is described in detail.

CHAPTER 1

INTRODUCTION

A large part of currently used software is, or is going to be, obsolete. This is mainly due to changes in the requirements, in the environment and in the available technologies. A radical solution may consist of completely ridesigning and reimplementing the application. This solution is often unfeasible. In fact, the costs can be prohibitive. Furthermore, the application might provide core or mission-critical functionalities that cannot be taken out of service. In some cases, it can be the only repository of the organization's business functionalities not explicitly documented elsewhere (2), and therefore it cannot be thrown away. Finally, today's highly dynamic business environments require flexible applications that can be quickly adapted to the new usage conditions and can easily interact with third-party components. Thus the modernization of these applications becomes more and more important.

Software modernization has to move in two directions: the componentization of the application and the adoption of "open" technologies that make the interaction with third-party heterogeneous components easier.

Often, the modularity of existing applications is extremely limited due to a "centralized" view of the whole problem. On the other hand, the availability of low cost and highly performing network infrastructures is pushing the use of distributed and decentralized applications in which the components are located on different servers and dynamically interact in order to respond to various situations. Consequently, heterogeneous and "closed" applications are being replaced by

"open" solutions, based on well-known and widely used standards that support interoperability of different technologies and components.

Starting from this scenario, the goal of this thesis is to study a model-based technique for the modernization of software applications to be used to transform monolithic and "closed" applications into flexible components open to different contexts.

1.1 Contribution

My work exploits the use of models in the study of an existing application which is going to be modernized. I propose a technique that tries to automatically infer meaningful views and useful information from the application.
Because often the documentation of the existing software is poor, partial or outdated, the existing application is the most valuable thing we can derive information from.
The first valuable model I want to exploit is the software architecture, which shows the logical components and the communication patterns of an application. The application's architecture can be extracted, as we will see later in our work, in two ways: by statically examining the source code or from the analysis of the execution traces of the program (dynamic analysis).
Both types of approaches have their advantages and drawbacks. I opted for using dynamic analysis because it captures the real behavior of an application. An additional motivation is that, its use in the field of automatic program modernization is still largely unexplored.

The main contribution of my work can be described by referring to Figure Figure 1, which illustrates the overall approach to modernization I developed and supported by means of a soft-

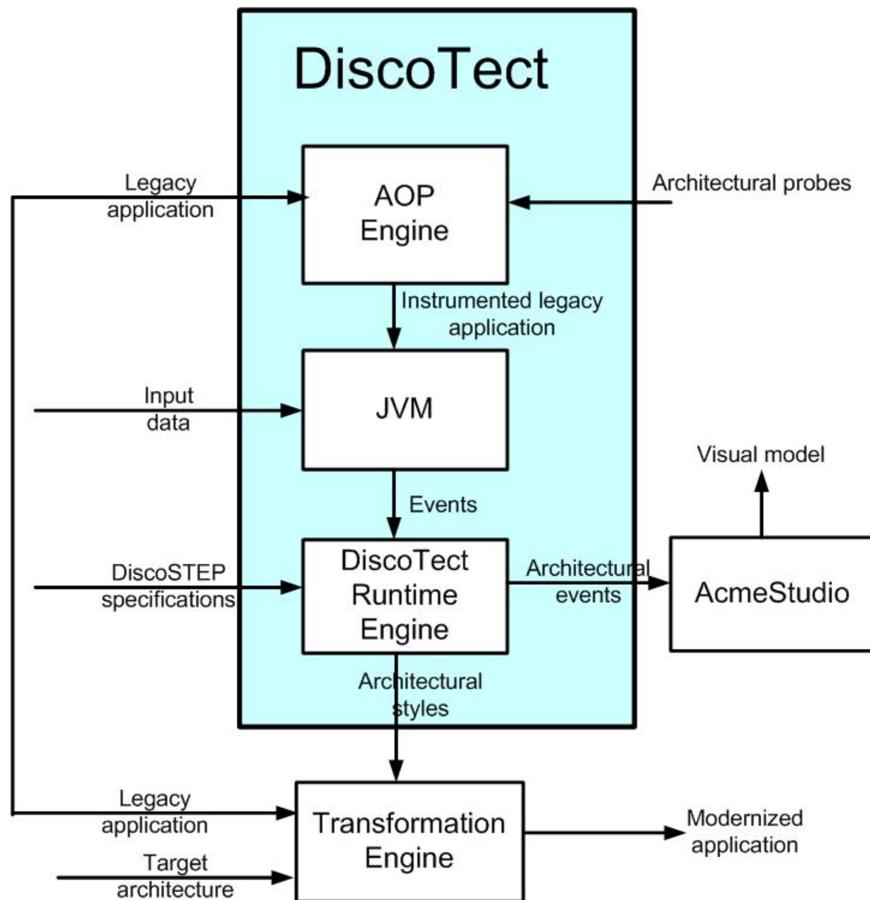


Figure 1. Overview of the tool, called Modernizer, described in this thesis.

ware transformation environment prototype, called Modernizer. The environment incorporates DiscoTect, a new tool developed by Garlan et al. (3), which allows: (1) the monitoring of Java programs executions through the use of aspect oriented programming and (2) the extraction of meaningful information to reconstruct and visualize the high-level architectures.

According to Figure Figure 1, a legacy application is first instrumented with ad-hoc probes, pro-

vided by a software specialist, whose goal is to collect runtime events generated by the running application and analyze them, via a runtime engine, to extract architectural events and styles. Architectural events are input to the AcmeStudio tool, which visualizes the model found. The identified architectural style is passed to a tool that is responsible for transforming the application to a target architecture specified as input.

Besides providing support to the framework illustrated in Figure Figure 1, in my work I instantiated the transformation tool to transform a client-server architecture into a Web service based architecture.

I chose this transformation because RMI applications offer functionalities similar to Web services and can be used for similar purposes. So it makes perfectly sense to write a transformation to upgrade from a language-dependent solution, such as RMI, to a more versatile and universal one offered by Web services.

Other works addressing the issue of service-based modernization of legacy software already exists, such as the work by Sneed et al. (4; 5) and by Canfora et al. (6). Their approach differs from mine because they just wrap the existing system or part of it behind a new Web service shell, while I apply an automatic transformation to the original source code. Another big difference is that, while those techniques apply static analysis to gather the information required for the modernization, I use dynamic analysis.

The use of dynamic analysis differentiates my solution from most of the studies that address the issue of automatic program modernization, as the DMS system (7; 8) or the works by Comella et al. (9), Seacord et al. (10), etc., which all use static analysis techniques.

1.2 Structure

The first part of the thesis gives an introduction to the subject and provides it with a theoretical context.

An overview on software modernization is given in Chapter 2. After giving a brief description to the problems related to legacy software, Chapter 2 explains the four main modernization techniques (transform, replace, rewrite and re-use) and the reasons for choosing one of them.

Chapter 3 focuses on the crucial aspect of software architecture analysis, describing the two main techniques: static and dynamic analysis. For each of them a view of the current state of the art is also provided.

Chapter 5 provides the motivations and goals of the project, along with a description of all its parts and background information about the existing tools used and intergrated in my prototype tool.

The development and implementation of my framework is described in detail in Chapters 5 and 6. Chapter 5 describes all the work done to integrate and use DiscoTect in the prototype, while Chapter 6 describes the prototype and its development, as an Eclipse plug-in, in depth.

The thesis is concluded with Chapter 7, in which I summarize the conclusions and the future work related to my project.

CHAPTER 2

SOFTWARE MODERNIZATION

The goal of this chapter is to understand software modernization and illustrate it from both a general and a process/strategical project management decisions point of view.
The technological aspects of interest for my work will be illustrated in Chapter 3, while in Chapter 4, 5 and 6 I will expose my proposed solution.

2.1 Legacy software

There are vast amounts of legacy systems throughout the world and there are also many definitions of what a legacy system is, but it is of little value to linger on definitions of legacy systems. In this work, with the term "legacy", I will refer to any existing software application that continues to provide core services to an organisation disregarding the age of the technologies and hardware used, such as mainframes, and of its programming languages. Note that "legacy" has little to do with the size or even age of the system; mainframes run 64-bit Linux and Java alongside 1960s code. We could even say that "every application developed with non-current technologies is legacy" to emphasize the fact that we do not have to think about legacy systems as dying "dinosaurs" (11). Any user organization which has been using information technology for more than five years has a legacy software problem. The reason for it is that the innovation cycle of software technology is now less than five years and that all software which has been developed without using the most current technology can be considered outdated (5). Legacy

software thus includes now not only the early languages of the 1970s, such as Fortran, COBOL, PLI , C, but also the Fourth Generation languages of the 1980s, such as ADS-Online, Ideal, Natural, CSP, Oracle Frames, etc. and the object-oriented languages of the 1990s, such as C++, Forté and Smalltalk. Even the early Java programs can now be considered as legacy software.

Usually there are several reasons behind the need to keep a legacy system:

- The costs of redesigning the system can be prohibitive because of its dimensions, complexity or because it is monolithic and scrapping or replacing it often means reengineering the organization's business processes as well.
- The system provides a core or mission critical functionality that cannot be taken out of service and the cost of designing a new system with a similar availability level is high.
- The organization has spent spent a lot of time, money and intellectual capital on it during the years, so it cannot just be thrown away and replaced.
- The system is considered as a repository of organization's business functionalities not explicitly documented anywhere else (2). As also Bisbal (12) says: "replacing legacy systems involves not only the direct financial cost of software development, but also the expense of rediscovering their accumulated knowledge about business rules and processes".
- The way the system works is not well understood anymore because either the designers of the system are not in the organization anymore, the system has not been fully documented or such documentation has been lost.

- The system works well and thus the owner has no reason for changing it. In other words, creating and re-learning a new system would have a prohibitive attendant cost in time and money.

At the same time, it is for those same reasons that legacy systems are considered to be potentially problematic by many software engineers (12; 13). In particular because they often run on obsolete (and usually slow) hardware and many times spare parts for such computers become increasingly difficult to obtain. These systems are also often hard to maintain and improve because, as said before, there is a lack of understanding of the system: the documentation has become inadequate, the manuals got lost over the years or the designers of the system may have left the organization, leaving no one left with the necessary knowledge and the cumulative code changes due to maintainance and improvements over many years often lead to less maintainable and understandable code. The documentation often becomes inadequate because, many times, the artifacts of a software system evolve at different rates. In many cases, when the source code is modified to fix bugs or to add enhancements the specifications and design documents are not updated to reflect these changes. In other cases the design is changed to reflect a new feature before it is actually added in the code, and maybe it will never be. The result is that developers learn not to trust and thus not to use anything other than the source code, making software less reliable and much more difficult to understand and evolve (14). Lehman's second law perfectly describes the effects of change on a system: "As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it." (15).

Typically, over the years, big investments in intellectual capital are spent for the legacy systems, in particular for their maintainance and to keep them working correctly. According to Sommerville (16), 50%-75% of software production costs come from maintenance. It is clear that legacy software applications are very valuable assets for the company that owns them, both for the money spent on them and for the knowledge embodied in those systems, even though the companies' balance sheets rarely show their real value. Due to what has just been said and to their complex and long evolution, legacy applications, like other intangible assets, are harder to imitate by competitors, and thus they can be seen also as a source of competitive advantage (17).

Existing legacy applications, as said before, are the outcome of past capital investments. The value of the application investment tends to decline over time as the business and the technology context changes. This can happen, for example, when the underpinning infrastructure is replaced, web access is required or the weight of changes in the applications and lack of available know-how makes it impossible to continue with enhancements (typical of systems that use vintage code). While the application investment tends to decline over time, the cost of maintaining the system tends to rise, eventually outweighting the cost of replacing both the software and hardware unless some form of emulation or backward compatibility allows the software to run on new hardware. However, many of these systems do still meet the basic needs of the organization and a new system would be extremely expensive: the systems that handle customers' accounts in banks or air traffic control systems are perfect examples. They are often written with vintage code as COBOL, they rely on an ad-hoc infrastructure and the organization can't just stop

them and replace them with a new one due to the demand of extremely high availability and to critical issues on safety and security. Therefore the organization cannot afford to stop them and yet some cannot afford to update them.

Transforming legacy applications is a task with both risks and rewards. It is easy to rely too much on what seems like a stable application and hoping that it will be adequate to keep the business going, at least in the medium term. But these legacy applications are at the heart of today's operations, many times crucial business operations, and if they get too far out of step with business needs the impact will be substantial and possibly catastrophic. Problems with legacy systems might rise due to: inflexibility (takes forever to make changes or can't make changes at all), maintainability (no documentation, no-one understands it or lack of skilled people), accessibility (cannot make it available to customers), cost of operation (runs on costly mainframe infrastructure, high license fees) and interdependency of application and infrastructure (cannot update one without the other). At this point, as it can be seen in Figure Figure 2, we have a choice: do we initiate a process of renovation and transformation or do we write the application off and find a replacement?

2.2 Transform, replace, rewrite or re-use?

System modernization can be distinguished by the level of system understanding required to support the modernization effort (18). Modernization that requires knowledge of the internals of a legacy system is called white-box modernization (composed of application transformation and application rewriting). Modernization that just requires knowledge of the external interfaces of a legacy system is called black-box modernization (also known as application re-use). I also

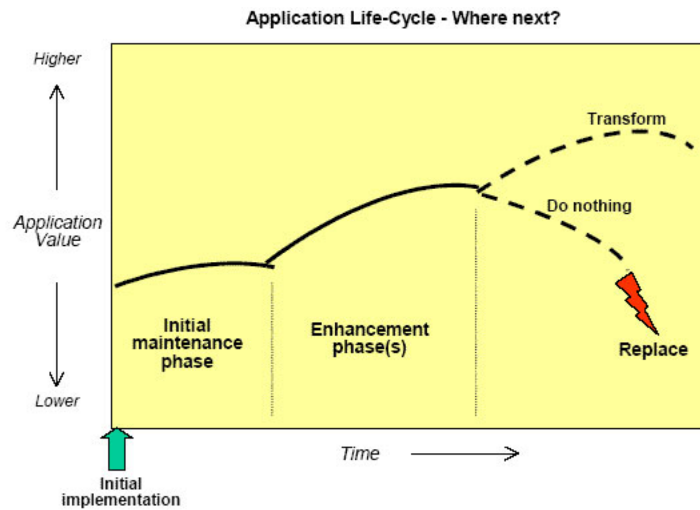


Figure 2. Legacy system. What next? (Taken from D. Good(1)).

include into system modernization, along with the previous two main alternatives, application replacement. Other works (19) consider replacement as part of only system evolution (along with maintenance and modernization), but I included it in modernization as I consider it the extreme solution/approach to the modernization of a system.

Having said this, once the decision to modernize a legacy application has been made, there are four broad alternatives that can be followed: rewrite the application, replace it with a totally new one, reuse the application wrapped with modern technology to make it look and work better or transform it using a combination of automated and/or manual tools. Note that legacy applications that do not support crucial business processes (for example back office systems) may still require modernization, but this typically is a financial decision rather than a strategic technical one and I will not cover them. D. Good (1) describes this decision process as an

evaluation of the quality of the application and of the availability of replacement packages. "Quality" is a pretty subjective term applied to the application itself, regardless of the platform it runs on or the code in use, and should be assessed in terms of such parameters as:

- Current effectiveness (eg. errors generated, number of workarounds, level of support needed).
- Stability of core business rules. In other words: will the application logic stay much the same in the medium-term? If the business model is going to change substantially then this assumption has to be questioned.
- Gaps in the functionality.
- Stage of the legacy life-cycle. In the earlier stages of the life-cycle, a legacy application will likely map closely to functionality requirements, although the platform might be obsolescent.

In summary, the term "quality" in this context is about how well the application does what it does and whether the functionality it lacks is due to its architecture or its infrastructure. In other words, it is about the suitability of the legacy application in business and technical terms. In general the "quality" of an application is considered to be low when the application fails often, rework and workarounds are often required or its architecture is a severe impediment to developing the new required functionality. If the quality of the legacy application is poor and there is a comparable functionality available in a trusted third-party software package, it makes sense to replace it. On the business strategy side, a package is surely not a source of competitive

advantage because, at best, it can deliver a short term advantage, since packages can be easily imitated or reused by other organizations. Figure Figure 3 shows the combinations of the two factors mentioned above that might lead to the four options I already introduced(transform, replace, rewrite or re-use).

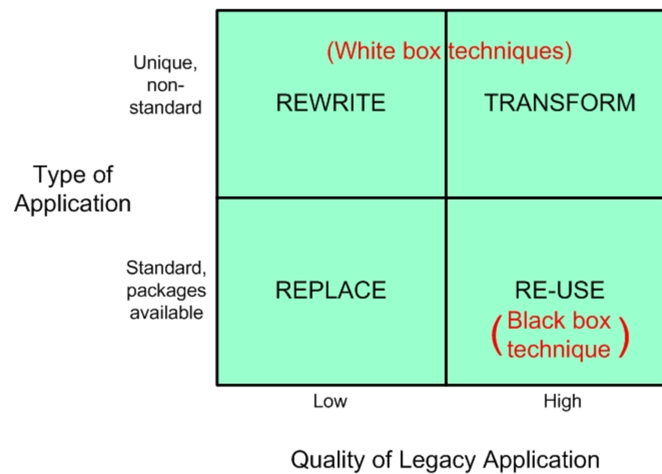
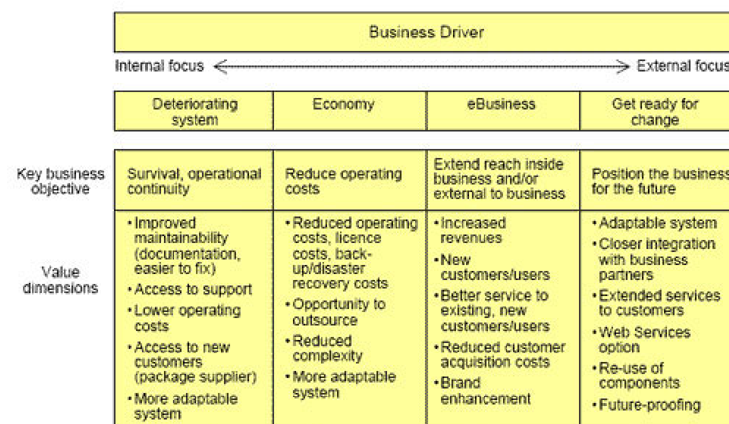


Figure 3. The combination of factors for choosing between transform, replace, rewrite or re-use.

There are four key business objectives that drive a modernization project:

- Reducing operating and maintenance costs and overheads.
- Improving maintainability in situations where no-one knows the application anymore or the documentation is limited/outdated.

- Improving access to the legacy application(s), often to provide Web access to customers and business partners or as a result of a organisational restructuring.
- Positioning for future projects (such as Web Services, or expected business change).



Business Driver

Internal focus ←——→ External focus

	Deteriorating system	Economy	eBusiness	Get ready for change
Key business objective	Survival, operational continuity	Reduce operating costs	Extend reach inside business and/or external to business	Position the business for the future
Value dimensions	• Improved maintainability (documentation, easier to fix) • Access to support • Lower operating costs • Access to new customers (package supplier) • More adaptable system	• Reduced operating costs, licence costs, back-up/disaster recovery costs • Opportunity to outsource • Reduced complexity • More adaptable system	• Increased revenues • New customers/users • Better service to existing, new customers/users • Reduced customer acquisition costs • Brand enhancement	• Adaptable system • Closer integration with business partners • Extended services to customers • Web Services option • Re-use of components • Future-proofing

Figure 4. The four main key busines objectives.

As it can be seen in Figure Figure 4, there can be quite a variation in the values that the business will be looking for from a modernization project, depending on the business objective chosen. The objectives on the left of the model emphasize more on cost: the justification for the transformation project is thus based on survival and cost savings, so the underlying issues will likely be about the platform, languages and data structures, in other words, the architecture. The objectives to the right are about business opportunity and the issues will thus be about functionality and future capability.

I will now take a closer look at the four options I introduced and for each one of them I will also present a checklist containing the main factors that lead to that particular option.

2.2.1 White-Box modernization

As previously said, white-box modernization requires knowledge of the internals of a legacy system. Thus it needs an initial reverse engineering process to gain an understanding of the internal system operation. Components of the system and their relationships need to be identified, to produce a representation of the system at a higher level of abstraction (20). *Program understanding* is the principal form of reverse engineering used in white-box modernization. Program understanding involves modeling the domain, extracting information from the code using appropriate extraction mechanisms, and creating abstractions that help in the understanding of the underlying system structure. Analyzing and understanding old code is a difficult task because with time, every system collapses under its own complexity. After the code is analyzed and understood, white-box modernization often includes, in the case of application transformation, some system or code restructuring. Software restructuring can be defined as "the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics)" (20). This transformation is typically used to improve some quality attribute of the system, like maintainability or performance. I will take a closer look at some techniques to extract several types of information from the system in the next chapter. Now I will pass on describing what I consider the two white-box modernization approaches: transforming and rewriting.

2.2.1.1 Legacy application transformation

Checklist for choosing to transform:

- Good fit of existing application with business needs.
- Moderate functionality changes needed in existing application.
- Significant functionality to be added in a new application and close integration with existing applications required.
- High operational costs of existing application.
- Need to migrate to new platforms, as J2EE or .NET, for strategic reasons.
- Future vision includes Web Services.
- Adding Web access.
- Difficult to find resources to maintain and modify applications on existing platform.

The basic goal of a legacy transformation is to retain (and add to) the value of the legacy asset while modernizing several aspects of it. This transformation can take several forms, for example it might involve translation of the source code, along with some level of re-use of existing code, with the addition of a Web-to-host capability to provide the customer access required by the business. The translation of source code is tipically used for parts of the system that interact with new technologies or that run on new platforms that the old language or programming style don't support or for parts written with some old programming languages/style that only few programmers know well, which would otherwise make the maintenance part extremely costly

and difficult.

On the other hand, the re-use of code is applied to parts that still perform well and don't need big enhancement, following the well known paradigm "if it ain't broke, don't fix it".

The input of the transformation process is a legacy application and the outcome is a transformed application with most of the legacy asset intact, possibly enhanced, and integrated into the overall applications portfolio of the business. The process is subdivided into the several process steps necessary to bring about this transformation. It should be understood that, like most process models, there may be several "paths" through these process steps, and there may be some iterations. For example, it may be decided to re-use some legacy code by 'wrapping' it in Java, while some other code is translated directly into Java; this involves different routes through the sub-processes. Figure Figure 5 illustrate a possible model for legacy applications

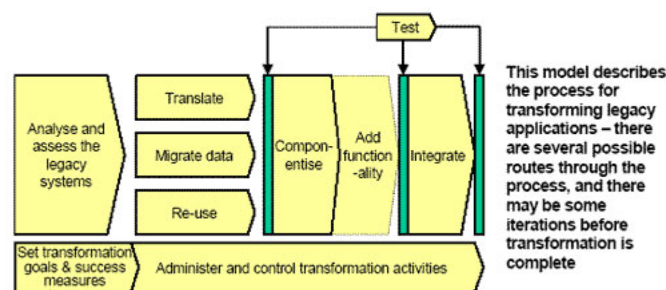


Figure 5. A model for legacy application transformations.

transformations, in which the typical activities for each sub-process can be:

- Analyse and assess the legacy applications:
  - Inventory of all the application "artifacts" (eg. source code, design documents, etc.).
  - Application mining to extract business rules.
  - Map application elements.
  - Evaluate the quality of the existing source code.
  - Analyse database including, tables, views indexes, procedures and triggers.
- Translate:
  - Refactoring (code structure improvement).
  - Automatic code translation.
  - Manual adjustments.
- Migrate data:
  - Restructure data according to a precise data model (e.g. Create relational environment and migrate non-relational data to relational according to a data model).
- Re-use:
  - Extract business rules and objects and move into a new and more modern language, as Java or XML.
  - Keep existing parts of code that still perform well or whose substitution costs and risks are too high (e.g. Wrap COBOL code in Java (and expose only limited parts to Web processes), regenerate COBOL dialect to current standard, etc.)

- Componentise:
  - Extract components and align with business function.
  - Restructure into presentation, persistent data and business logic.
- Add functionality:
  - The stage where additional functionality can be added to meet specific business requirements not already met by the legacy application.
- Integrate:
  - Transfer executables, image files, Java etc. to servers.
  - Web enabling.
  - Install data access link.
  - Performance tuning.
- Set transformation goals and success measures:
  - Agree on the goals of the transformation project with management.
  - Incorporate goals and measures into project plans.
  - Carry out periodic 'sanity checks' to ensure that fundamentals are being adhered to.
- Administer and control the transformation:
  - Version control and tracking.
  - Estimate effort required.

  - Establish metrics (eg. of complexity).
  - Document application (retro-document existing application and/or end result).
  - Choose methodology to govern the transformation.

- Test:

  - Use debugger.
  - Create test scripts.
  - Confidence testing.
  - Customer acceptance testing.

There are many tools for the transformation process, but they are mostly "point solutions", they typically deal with one or two specific programming languages and they are restricted to a small number of target environments. For example, the translators are constructed to handle specific languages, although they can be extended fairly readily to handle other languages as required by the market. DMS (7) is a very big and ambitious project for scalable multiple languages transformation on large software systems (250.000 lines of code or more) but it is still partially in development and limited to few languages as Java and C++ (8). Furthermore, the products are generally limited to one part of the transformation process. This is an inevitable consequence of the structure of the transformation process, because, although the transformation process is presented as a monolithic transformation, in reality the elements are quite different from one another. For example, the Translate activities change code from one language to another, while the Analyse and Assess the Legacy Application activities change a state of

(relative) “ignorance” to one of “knowledge”. Thus it is reasonable to suppose that the tools that support automation of these separate process elements will remain distinct (even when they are accessed through a common interface of a modernization integrated environment). This is not necessarily a bad thing, but it does mean that more than one tool is likely to be needed and some mixing and matching will be necessary and it suggests that some skepticism is required when a supplier talks about a solution that “will take care of everything”.

In order to allow different modernization tools provided by different vendors to interact with each other, exchange data and maybe get integrated in a modernization environment OMG proposed the Knowledge Discovery Metamodel (21; 22; 23) (a MOF 2.0 compliant metamodel, delivered in an XMI format) to represent information related to existing software assets, their operational environments and in general all the information required for modernization needs. More precisely KDM includes, or will include, models covering the following system attributes:

- **Static behaviour of the system:** association, containment, annotated call graphs (method calls, access/modification of data, etc.), inheritance, error handling, exceptions, real time/concurrency.
- **Dynamic/run-time behaviour of the system:** polymorphic calls, profiling to show component usage, unravelled concurrent/distributed/real-time behaviour.
- **Supporting materials:** test materials, requirements and design documentation, etc.
- **Environment:** actual files, their locations, interactions with the user/file system/database/etc. and build mechanisms (makefiles and other dependencies).

The KDM proposal is fairly new, the first final adopted specification (23) has been submitted on June 2006, so it still unknown by many and unused by any application. Note that it is rather complex so it might take some time to understand and master it before being axample to integrate it in an application. But it is clear that it represents a very powerful and important standard (as, for example, UML) that future modernization applications should use to exchange useful data with each other.

2.2.1.2 Legacy application rewriting

Checklist for choosing to rewrite:

- Business rules satisfactory but extensive functionality needs to be added.
- No off-the-shelf solution comes close to meeting the needs.
- Poor quality code, with high maintenance costs.
- Can afford time, cost and disruption involved.

As it can be seen from the above checklist, this solution is tipically used when the quality of the code is low, the maintainance costs get higher and higher as the time goes by, because of the poor quality of the code (10), but the functionalities offered work well except for the need of new features.
The causes of poor quality code are two: the program was badly written from the beginning or the code has deteriorated over time due to changes for maintainance.

Companies often modify software in response to changing business practices, to correct errors, to improve performance, maintainability or other quality attributes. This is in accordance

to Lehman's first law: "A large program that is used undergoes continuing change or becomes progressively less useful"(15). Usually the reasons for software change fall into four categories:

Perfective: Changes made to improve the product or to enhance system attributes. These types of changes are also called enhancements.

Corrective: Changes made to repair defects in the system.

Adaptive: Changes made to keep pace with changing environments, such as new operating systems, language compilers and tools, etc.

Preventive: Changes made to improve the future maintainability and reliability of a system.

Keep in mind that everytime an application needs to be rewritten, off-the-shelf solutions should also be taken into account. In fact, reliable third party applications that meet the business requirements can make an organization save time and capital.

As the name suggests, in this case the application code is to be rewritten, but while the application is the problem, the business rules and data structures are valuable key resources for the new application. Instead of starting to code from scratch, which would be useless and costly, the use of application mining and analysis of code logic and data structures and the use of all the old system's documentation is perfect as a starting point, to create an intial knowledge base containing all the business rules, data structures and logic that worked well in the old system.

2.2.2 Legacy application replacement

Checklist for choosing to replace:

- Application significantly out of line with business needs.
- Systems is undocumented and/or outdated.
- Willing to make changes to business model to fit off-the-shelf solution.
- Can afford time, cost and disruption involved.

Legacy application replacement, also called big bang approach or cold turkey (24), is basically the creation from scratch of a totally new system using modern methodologies and technologies and a new set of requirements (13). Replacement is appropriate for legacy systems that cannot keep pace with business needs and for which other modernization methods are not possible or cost effective. Replacement is normally used with systems that are undocumented, outdated or not extensible. Application replacement is obviously not the right choice if the legacy system contains valuable knowledge for the organization that has to be kept, in this case application rewriting or transformation, for example, should be considered.

Replacement has also high risks that should be carefully evaluated before selecting this technique:

- Replacement is basically building a system from scratch and thus is very resource intensive. So, the organization's IT resources that are typically fully allocated performing maintenance tasks need to be partially reallocated to the new project, thus lowering the maintenance quality. The IT may also not be familiar with new technologies that can be utilized on the new system.

- Replacement requires extensive testing of the new system, while the existing legacy systems are well tested and tuned, and encapsulate considerable business expertise. There is no guarantee that the new system will be as robust or functional as the old one and, as shown in Figure Figure 6, it may cause a period of degraded system functionality with respect to business needs.
- System specification and documentation are often missing, partial or outdated, so it is extremely difficult to gather information to what are the applications, both critical and not, that during the years were created and that relied in some way on the old system. That information must be somehow found, which can be extremely hard, to keep those application working even with the new system.
- If the old legacy system is big, the new project will even be bigger, if not huge. It is known that very big projects have a very high failure rate since many factors as complexity of the distribution of work, IT personnel training, etc. is often underestimated.
- Replacement temporarily doubles the total costs, in fact the organization would have to invest in the development of the new system while keeping the old system and maintaing it (2).

These risks are so high that nearly the 80% of the replacement projects started in 1992/93 in the US, by 1995 were postponed, reorganized or even cancelled (2). So it's not a suprise that many times organizations, because of all these risks and uncertainties keep the *good ole* legacy system.

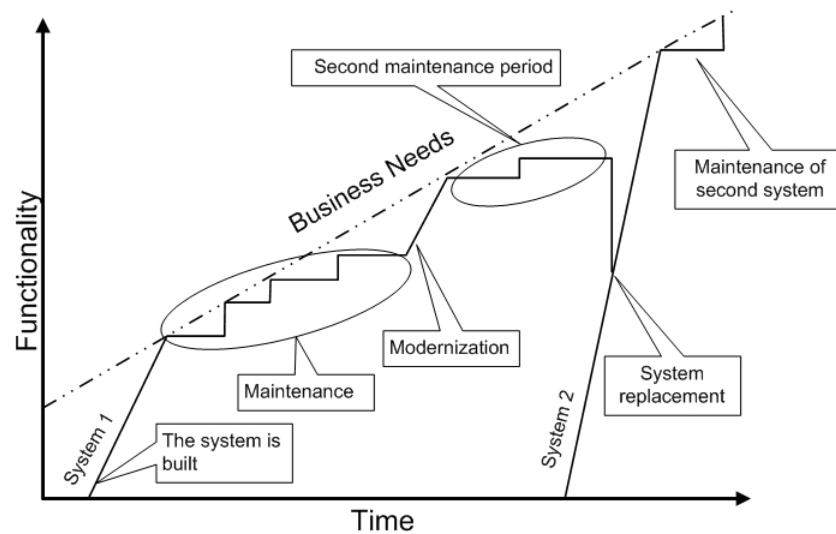


Figure 6. Information System life cycle.

2.2.3 Black-Box modernization

Checklist for choosing to re-use:

- Business rules satisfactory.
- Low operational costs of existing application.
- Difficulties in separating logic from persistent data and presentation layers.
- Simple Web access required, allowing a wrapping solution.
- Have resources to keep core legacy maintained.
- Source code not available.

As it can be seen from the above checklist, this solution is tipically used when the old application is still working correctly, its operational and maintenance costs are low and the

changes required are limited to use of the application in a new context, usually the Web, as in Canfora et al work (25). This solution is also used when the application is still working well, it needs some new features to be added, but the source code is not available (or too difficult to comprehend) and thus only the executables can be used. In this case there is no real reason to throw away the old and functioning application and invest a big amount of time and capital to replace it, transform it or even rewrite it. Why should not we reuse the existing experience if it was successful, rather than creating everything from the beginning (26)? Legacy software re-use is also called a black-box modernization technique, because it just requires knowledge of the external interfaces of the legacy system while all the internals are ignored (9).

There are two possible cases when reusing some software: the application is based on a third-party package or the application has been developed from scratch by the organization. The only difference is that when using a third party application it might be useful at first to upgrade to the latest version then, as for in-house applications, use wrapping techniques to provide the required reach and other functionality improvements.

By using wrappers we can still import all the business logic of the legacy system without having to rely on the old methodologies and technologies. In fact with a wrapper we can hide the effective implementation and functionalities of the system behind a new and well defined interface, usually object oriented, so that the legacy system or different parts of it can be easily integrated in the new environment or can cooperate with new systems, as brand new objects/components (27).

There are two main approaches in the use and creation of wrappers: a simple and quick one and a more complex one.

The first one can be labeled as an “opportunistic” approach. In fact it tries to reuse the existing system as much as possible and in the simplest way: it is the application that guides the design of the wrapper and the goal of the wrapper is simply to export the application interface in the new environment. This new environment is usually the Web, in which the legacy systems are exported as Web Services, as in the works by G. Canfora et al (6) and Sneed and Sneed (4). A legacy transaction particularly significant for the final user is identified and exported on the Web through a simple wrapper. Note that the component only makes the transaction accessible in the new environment but it doesn't offer any kind of decoupling with the old business logic so the client will have to take care of all the eventual processing and translation of the output received. In this approach the focus is not on adding new functionalities or on integrate the system in the new environment but just on exporting it, so it is clear that, with this in mind, the starting point are the old system specifications.

Obviously, the big weakness of this approach is the lack of transparency and decoupling. Every application that wants to access the legacy sytem through the new wrapper will still have to know at least the structure of the output of that particular legacy function.

So now I can export the old system in a new environment, but it still depends a lot on its original implementation and logic. This solution can also lead to what can be called the *black-box syndrome*: the old application, now component or service in the new system, is trusted for what it does, but nobody knows -or wants to know- what actually goes on internally (28).

The second approach, instead of starting bottom-up to try to re-use as much as possible (as in the first approach), is more top-down. At first the whole system is analyzed as it would be implemented from scratch, creating a new object-based model. The legacy system is then studied to see what functions can be useful in that solution and thus re-used; the useful legacy functions found are then scattered and encapsulated in the objects that need them. In reality, a pure top-down approach in unfeasible, but what is most important in this approach is that the outcome is a model based on objects that encapsulate the legacy system, or at least part of it. It can be said that the new model becomes the wrapper.

Unfortunately, this solution is not always practical as it also requires an understanding of the software modules' internals using white-box techniques (19), which cannot be always done.

A methodology for this kind of wrapping that uses a model driven approach is proposed by van den Heuvel (29). This methodology combines both forward engineering and reverse engineering. The forward engineering phase creates a PIM (Platform Independent Model) depicting the business processes and entities as a set of collaborating components. The reverse engineering on the other hand, works the other way around. It starts from the legacy code to abstract a PIM. The two models found are then matched and adapted. During matching, the source specification is compared to the target specification to find and identify reusable fragments of the legacy application. Once a pre-existing legacy component partially matches a new business component, it is wrapped and adapted in order to overcome incompabilities at both the semantic and syntactic level. The problem with this methodology is that it's still only on paper so all

the transformations to create the PIM and the matching and adaptation phases have yet to be developed, so we do not know how much is really doable and what can be automated.

2.2.4 Final notes

Some readers might have noted overlappings between some definitions, now I will try to clarify all of them:

Replacement - Rewriting: What differentiate Replacement from Rewriting is that in the first case the knowledge contained in the legacy application is thrown away as a totally new one is built, while in the second case that knowledge is the starting point. Usually the rewrite option is chosen over replace if the legacy system contains valuable knowledge for the organization and/or is it still fulfills the business needs, except for some new features that are needed.

Re-use - Transformation: What differentiate Re-use from Transformation is that the first one is a black-box technique and a sort of a transformation but superficial and without any modification of the orginal source code. On the other hand, application transformation, according to my definition, is white-box (for a definition of white-box and black-box modernization see paragraph 2.2). Moreover, while transforming I might re-use parts of the system without but along with modifications, code-to-code translations of other parts, etc. while with application re-use (according to my definition) the system is in some way wrapped in a new interface but without any modification of the original source code. So we can say that application re-use might be used as a "stand alone" (see paragraph 2.2.3) technique or as one of the several techniques used by a transformation process (30).

Transformation - Rewriting: What differentiate Transformation from Rewriting is that in the second one the whole system is written from scratch using the old one only as a knowledge base to guide the development. On the otherhand, in the first case, only some parts are rewritten from scratch, others are re-used, others translated with code-to-code translations, etc.

CHAPTER 3

SOFTWARE ARCHITECTURE ANALYSIS

Architecture is one of the most important abstract representations of a software application. In fact it supports many types of high-level inspections and analysis, allowing the designer to determine, for example, if a system's design will satisfy its critical quality requirements before even starting an implementation (3). A crucial issue is then to verify if a system is built consistently with its architectural design. Another crucial aspect, especially in the case of legacy systems with no documentation, is to recover the architectural structure itself.

There are three main techniques that have been studied and developed to determine or enforce relationships between the software architecture and its implementation. The first one consists of enforcing this consistency by construction, embedding architectural constructs in the implementation language used so that tools can automatically check for conformance (For example, a solution for Java, called ArchJava, as been described by Aldrich et al. (31)). A paper by Coelho and Abi-Antoun (32) applies ArchJava to an existing software and studies its limitations.

Architecture consistency could also be enforced through code generation, with the use of tools that create and implementation from a more abstract architectural definition (33).

Ensuring consistency by construction can be highly effective but it has an extremely limited applicability. In fact, it can be applied only in situations where the software system was developed using specific architecture-based development tools, languages and implementation strategies from the beginning; and this is not the case of most of the development projects since the man-

agement tends to ignore this issue. This approach does not work also for systems composed of existing parts or that require a style of architecture or implementation that is not supported by those generation tools, which is the most common case (that includes, obviously, legacy systems). The only way to use this technique with legacy system is to rewrite or transform them to include a programming language that enforces architectural structure at the implementation level (e.g ArchJava). The paper by Coelho and Abi-Antoun (32) describes a case study in which a legacy system was re-engineered using ArchJava, in order to remove or at least limit future architectural erosion and drift.

The other two techniques take two different approaches to reverse engineering, in particular to design recovery. Design recovery is a subset of reverse engineering in which "domain knowledge, external information and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself" (20). Remember that reverse engineering is just a process of examination, not a process of change or replication.

A first approach to design recovery consists of extracting a system architecture from the source code using *static code analysis*. This technique is well known since it has been widely studied, as we will see in the next section. A second approach to design recovery, which is still relatively unexplored, consists of determining the architecture by *examining the runtime behaviour* of the system. The system's execution is, in some way, monitored and those observations can be used to infer its *dynamic architecture*. The potential advantage of this approach is that it applies to any system that can be monitored (which is almost any system). It also works

with systems whose architecture changes dynamically and imposes no a priori restricions on the system implementation or the architectural style. Since there is always no “free meal”, there are several hard technical challenges to make this technique work (3). I will dig a bit deeper into this problem later in this chapter, in Section 3.2.

After briefly describing static and dynamic code analysis it is clear now that, as stated before, they are both reverse engineering tasks since they allow one to identify the system’s components and their interrelationships and to create representations of the system at a higher level of abstraction (20).

Now I will take a closer look at the static and dynamic analysis of a system to recover and analyze its architecture.

3.1 Architecture recovery through static analysis

Static analysis is performed by examining the source code of the target system statically. What can be determined with a static analysis is the overall structure of the system through, for example, a UML class diagram (as defined in (34)) and the behaviour of the system that can be extracted by examining the source code, through, for example, UML sequence and collaboration diagrams (34).

Nowadays, almost all UML CASE tools, both commercial and opensource, can generate UML class diagrams from the source code of an existing system (35; 36; 37; 38; 39; 40) just to name a few, so for a system coded in a supported language, the recovery of the static architecture is trivial. However, there are still some challenges to be addressed, such as how to distinghuish between plain association, aggregation and composition relationships and the reverse engineering

of to-many associations (41); so, manual inspection might still be needed. In addition, static architecture analysis is best suited for object oriented languages, in which modularization and coding patterns can be identified with architectural patterns (3), while might be problematic in the case of other programming language paradigms.

A number of static analysis techniques that extract sequence and collaboration diagrams to better understand how a particular architecture found works have been reported in literature (42; 43; 35; 44; 45).

The tool developed by Tonella et al (42), RevEng, generates both UML collaboration and sequence diagrams from C++ source code, the work by Kollmann et al. (43) generates only collaboration digrams from Java source code while the two works by Rountev (44; 45) generate sequence diagrams from Java source code.

The main difference between those techniques is the analysis of object references in the source code and, thus, the way objects are identified and then displayed in the UML diagrams. In the simplest solutions (35; 43), parameters, local variables and attribute names are used to identify objects.

The solution proposed in (43) first represents the source code of the system that is being studied as an instance of the Java Meta Model, an ad-hoc UML-like metamodel used for the specification of the Java language based on information from the Java Language Specification (46) that focuses only on the parts that are required for the rendering collaboration and interaction between objects, in order to extract and create collaboration diagrams. The final UML

collaboration diagram is then created from the instance of the metamodel found, through a set of mapping rules.

A similar but less coarse-grained approach is the one described in (42), in which all the runtime objects potentially created by a call to a constructor by a single object are represented in the generated sequence diagram. This analysis is less coarse-grained since it basically looks for allocation statements, as **p = new X(..)**, instead of variable, parameter and attribute names. The problem with this technique is that one object in the sequence diagram can represent many runtime objects, leading to inaccurate diagrams. In fact it is impossible to determine statically the number of times a statement is executed, so the multiplicity of each object is unknown.

The technique presented in (44; 45) performs deeper and more sophisticated analysis of the source code to identify the different objects involved in any interaction. In particular this techinque is able to recognize when one call site refers always to the same runtime object so that only one object is displayed in the diagram.

The work proposed by Notkin et al. (47) takes a totally different approach at the static reverse engineering of systems. The authors of this work acknowledge the existence of a huge number of reverse engineering tools that derive high-level models from the source code. Almost all UML CASE tools do that (e.g. (35; 36; 37; 38; 39; 40) just to name a few). All these tools derive very accurate and useful representations of the source, but they may differ from the models sketched by engineers. So they developed an approach that enables an engineer to produce high-level models in a different way. The engineer defines a high-level model of interest, extracts from the source code a source model and defines a declarative mapping between the entities of

the two models. A reflexion model is then computed to show the convergences and divergences between the engineer's high-level model and the source model: The reflexion model summarizes a source model of a sofware system from the viewpoint of a particular high-level model (47). The engineer thus interprets the reflexion model and, if necessary, modifies the input to iteratively compute additional reflexion models and reach a model of the system that has the desired level of precision. Figure Figure 7 shows an overview of this approach. So the goal of this approach is to permit engineers to easily explore structural aspects of large software systems and produce, at low-cost, high-level models that are "good enough" to be used in a particular software engineering activity (e.g. restructuring, reengineering, porting, etc.).

The main problem with this approach is that the engineer has to create its own high-level models

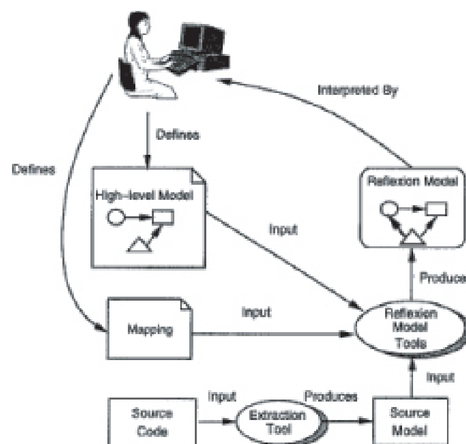


Figure 7. Overview of the Reflexion Model approach.

and declarative mappings between the two models, so she/he needs to have a good knowledge of the system that is being studied and a generic idea of the system's model. Many other tools, on the other hand, assume that the engineer might not know anything about the system under study, which is rather common, and present him a series of solution without the need of interaction as in the solution proposed by (47).

An approach similar to (47) is the prototype Dali, proposed by Kazman and Carrière (48). Dali is an open workbench structured to contain a series of already existing tools that, composed together, aid the user in interpreting architectural information that has been automatically extracted from the system under study. The system is structured as an open, lightweight workbench in order to provide an infrastructure for opportunistic integration of a wide variety of tools and techniques. This is because, since there is a great deal of languages, architectural styles, implementation conventions and so on, no single collection of tools will be sufficient for all architectural extraction and analysis. In this way, new elements can be easily integrated and such integration does not impact other elements of the workbench. Figure Figure 8 shows the overall structure of the Dali workbench.

Dali uses three techniques to reconstruct a system's architecture:

1. Architectural extraction, that extracts the as-implemented architecture from source artifacts such as code and makefiles. The model is then stored in a repository.
2. User defined architectural patterns, that link the as-implemented architecture to the architecture imagined by the user, which can be called as-designed architecture. The as-designed architecture consists of the kinds of abstractions used in architectural repre-

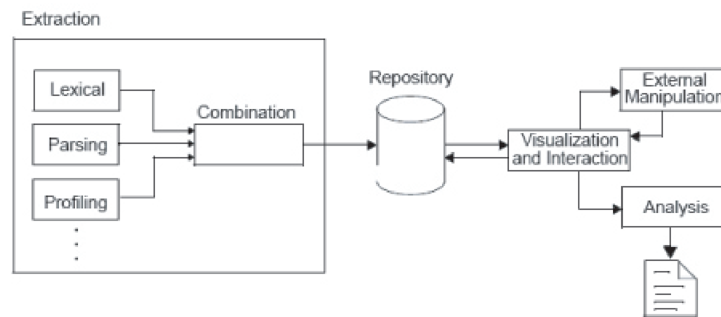


Figure 8. The Dali workbench structure.

sentations (subsystems, high level components, repositories, layers, conceptually-related functionality, etc.). The architecture patterns explicitly link these abstractions to the information extracted in step 1.

3. Visualization of the resulting architecture, that is the extracted information in step 1 organized by the patterns defined in step 2, for validation by the user.

Since, as it was said before, Dali is an open workbench, the tools for architectural extraction, model storage and visualization can vary, depending on the needs. Many tools have already been developed for those tasks, so there was no need to create new ones from scratch. For the sake of knowledge, here is the list of tools with whom the workbench, in order to be tested with real systems, was intially populated:

- Lightweight Source Model Extraction (LSME) (49), Imagix (50), make, and Perl (51) to extract source model information for C and C++,
- PostgreSQL (based on POSTGRES (52)) for model storage.

- IAPR (Interactive Architecture Pattern Recognition) (53), RMTool (47), and Perl for analysis and manipulation.
- AcmeStudio (54) for visualization of the architcture inferred.

The main contribution of Dali is then the finding of an architecture from extracted artifacts through the sinergy of those techniques in a single environment and the user interaction through pattern matching and interpretation.

As the work done by Notkin et al. (47), Dali does not attempt to automatically find and extract the architecture for the user, but it supports the user in defining architectural patterns and in matching those patterns to extracted information. But while the work by Notkin et al. focuses mainly on the reflexion models, without caring about the source model extraction and the visualization parts, Dali provides an open and customizable environment to study and support the whole process of reconstructuring the software architecture (using already existing techniques and tools, including Notkin's work (47)), from the extraction of the source model from existing artifacts up to the visualization of the reflexion model summarizing the source model of a software system from the viewpoint of a particular high-level model specified by the user. So we can say that Dali summarizes the previous works and researches in static software architecture reconstruction and permits to integrate tham in a single workbench.

Understanding the behaviour of a system, in particular an object-oriented one (as the ones considered in the previous techniques), can be a very hard task. Because of inheritance, polymorphism and dynamic binding, the dynamic type of an object, and thus which methods are going to be executed, might be impossible to know (55). In addition to this, there are also

multithreading and distribution that complicates the analysis even more. It is then extremely difficult to create UML sequence diagrams from the analysis of the source code program execution (it might even need complex techniques as symbolic execution); The problem gets even harder in the case of large and complex systems. So, it is clear that if we want to retrieve meaningful informations and create dynamic models of large and complex systems, a dynamic analysis and, thus, executing the system and monitoring the execution, is necessary.

We can say that static analysis can present a complete picture of what could happen at runtime but it does not necessary show what actually happens in a system (56). In fact, since it is limited to static, code-based structures (as classes and packages), the runtime structures, that are the essence of most of the architectural descriptions, are ignored. Actual runtime structures are, in fact, usually unknown until the exection of the program (57; 58). For example, clients and server may come and go dynamically while components may be dynamically loaded.

3.2 Architecture recovery through dynamic analysis

Any dynamic analysis aimed at the discovery of architectures must address three main issues (3).

First, an instrumentation/monitoring strategy has to be devised in order to gather, at runtime, the meaningful events to generate dynamic models while at the same time trying to have a low impact on execution and a reduced overhead, which is usually typical of instrumentation. The kind and amount of information to be gathered depends on the dynamic model and the level of detail needed. As we will see later, there are already many and well known intrumentation techniques.

The second issue is about the mapping of the extracted runtime events to architectural events; in other words, the interpretation of the runtime behaviour in terms of architectural meaningful events. This is, for several issues, a hard problem. First, architectural events usually are not mapped one-to-one with the low-level system observation that can be extracted with any instrumentation technique. An abstract architectural event might involve many runtime events, such as object creation, library calls to runtime infrastructure, initialization of data structures and so on. But at the same time, a single low-level event might represent a series of architectural events. For example, the execution of a procedure call between two objects might signal the creation of a connector and its attachment to the components that represent the two objects. The second problem is that, due mainly to concurrency, architecturally relevant actions are usually interleaved; so, in order to recognize multiple architectural events, we have to cope with concurrent intermediate states (3). A third problem is that usually there are different implementations of the same abstract architectural element; consider, for example, the number of ways to implement pipes. So there is no single standard that indicates what implementation patterns represent specific architectures and, at the same time, there is no single architectural style or pattern that can be used for all systems. We will see later how some approaches try to overcome these two problems.

Finally, the representation and the display of the resulting architecture need to be taken into account. Since this issue is not of big interest I will just briefly look at it later.

3.2.1 Instrumentation

Software instrumentation usually refers to the chunks of code that developers insert in an application to record the values of function parameters, timing statistics and the like. In this case, instrumentation is used to gather at runtime the necessary information to generate the model wanted, while reducing to the maximum extent possible the impact on execution and the human overhead usually associated with instrumentation (56).

A limitation of instrumentation is the effect it might have on the execution of the instrumented system. Delays caused by the execution of the instrumentation code could make the system behave differently from what is expected (e.g. deadlines). This is unfortunately an unavoidable problem, because as said in (59): "observing a system modifies a system". Unless the instrumented system is a hard real-time system with deadlines, the delays introduced by the instrumentation should not change the behavior of the system, if the instrumentation is done smartly and only the necessary parts are instrumented.

I also think that a dynamic architecture analysis tool should adopt the less intrusive intrumentation strategy possible. In fact, most of the times, the system that are analyzed to recover the architecture do not have the source code or, if the code is available, it might have been written by someone else and, thus, it could be extremely difficult to instrument it in the right way.

I will now discuss about the main instrumentation techniques and, in particular, I will focus more on the use of Aspect-Oriented-Programming to support the instrumentation of Java bytecode, which is a very promising technology for this purpose.

3.2.1.1 Log4J and Java API logging

Log4j is an open source logging library developed as a subproject of the Apache Software Foundation's Logging Services Project, based on a logging library developed at IBM in the late 90s. Log4j is widely used in the open source community (e.g in Jboss (60)).

Log4j's architecture is built around three main concepts: *loggers*, *appenders* and *layouts*, that allow developers to log messages according to their type and priority and to control where messages end up. Loggers are objects that an applications first call to start the logging of a message. When given a message to log, a logger wraps the given message in a newly generated LoggingEvent object. The logger then hands off the LoggingEvent to its associated appender. An appender sends the information contained by the LoggingEvent to a specified output destinations. For example, a ConsoleAppender will write the information to System.out, a FileApppender will append it to a log file. Before sending a LoggingEvent information to its final output target, an appender might some standard layout to create a text representation of the information in a standard format. For example, Log4j includes an XMLLayout class that can be used to format LoggingEvents as strings of XML.

LoggingEvents are assigned a level that indicates their priority. The default levels are(ordered from highest to lowest): OFF, FATAL, ERROR, WARN, INFO, DEBUG and ALL. Loggers and appenders are also assigned a level, and, thus, will only execute logging requests that have a level that is equal to or greater than their own. For example, if an appender with a level of ERROR is asked to execute a logging request that has a level of WARN, the appender will not execute the logging request. When a logger is asked to log a message, it first checks that

the level of the request is greater than or equal to its level. If so, it creates a LoggingEvent from the given message and passes it to its appenders, which format it and send it to its output destinations.

The java.util.logging package (a.k.a. JUL), which Sun introduced in 2002 with Java SDK version 1.4, is extremely similar to Log4j. It uses, more or less, the same concepts, but renames some of them.
For example, appenders are "handlers", layouts are "formatters" and LoggingEvents are "LogRecords". JUL uses levels the same way that Log4J does, the only difference is that the default levels are nine, instead of seven. Concepts pretty much map one-to-one from Log4j to JUL; though the two libraries are different in subtle ways, any developer familiar with Log4j needs only to adjust his vocabulary to understand JUL.

While Log4j and JUL are almost conceptually identical, they do differ in terms of functionality. Their difference can be summarized as: "Whatever JUL can do, Log4j can also do and more". They differ most in the areas of useful appender/handler implementations, useful formatter/layout implementations, and configuration flexibility.
For example, JUL contains four concrete handler implementations, while Log4j includes over a dozen appender implementations. JUL's handlers are adequate for basic logging; they allow to write to a buffer, a console, a socket and a file. Log4j's appenders, on the other hand, cover almost every logging output destination possible: they can write to an NT event log, a Unix syslog or even send e-mail. So we can say that Log4j and JUL are very similar APIs that differ conceptually only in small details and in the end do more or less the same things, except Log4j

has more features.

I will stop here and will not cover the other differences, since I only need an overview of the existing instrumentation techniques. It is clear that these to techniques are rather easy to use and inexpensive, in fact they are both designed to make logging/instrumentation as inexpensive as possible (To let code produce fine-grained logging when needed but not slow the application in normal production use, a mechanisms to dynamically change what log messages are produced is adopted) but are extremely intrusive since the required code that calls one of the two API and to produce the required logging has to be added to the original source code.

3.2.1.2 Program instrumentation via Aspect Oriented Programming

Aspect oriented programming (61) is a relatively new paradigm which addresses a central limitation of Object Oriented Programming (and *a fortiori* of older paradigms): the breakdown of modularity that results when the core business logic is combined with other "concerns" such as security, data persistence, transactions, high availability, logging, error handling, etc. as part of the implementation process.

Separation of concerns effectively decomposes the program into loosely coupled modules and classes, which can be developed separately and then combined later to create applications (this strategy is also called "divide and conquer"). All programming methodologies, including procedural programming and object oriented programming, support some separation and encapsulation of concerns (or any area of interest or focus) into single entities. For example, procedures, packages, classes and methods all help programmers encapsulate concerns into single entities.

However, additional logic for other concerns cuts across the boundaries of the modules of a

program. These concerns are then called crosscutting concerns (This terms and most of the terminology used nowadays in AOP comes from (62; 63)). Hence, in the source code of an application, the original model is often obscure by code that implements the crosscutting concerns. This tangling of code reduces modularity, increases complexity and impacts quality. For example, maintenance becomes difficult because modifications in crosscutting concerns must be made across many code modules. In short, crosscutting concerns undermine the many benefits of the separation of concerns and, in particular, of OOP.

The vision of AOP is to restore the overall modularity by extracting the crosscutting concerns, modularizing them as aspects and then weaving them back into the application at well defined locations, called *joinpoints*. Aspects can be discussed for applications written in any kind of programming methodology, in fact, developers have been using forms of interception and dispatch-patching that are similar to some of the implementation techniques for AOP since 1970s.
It is well known and proved though that AOP and OOP complement each other nicely; AOP could be seen as a "superset" of OOP. This is pretty obvious, since the first and most popular AOP language (which also originated the concept of AOP), AspectJ, developed by Gregor Kiczales and his team at Xerox PARC, is for Java programming language (64).

The core business logic and the aspects can then be developed separately using OOP techniques, extended with AOP concepts. AOP then describes how to weave the logic and aspects together to create the application. AOP is also a useful tool for quality-assurance issues. For example, white-box testing can be implemented using test aspects woven into applications to

drive test cases, inject faults, examine program state, etc. Note that AOP is still relatively new and evolving, but implementations for a wide range of languages have been developed or are in the development phase. Obviously, the most mature and used is AspectJ (62; 63) for Java, but there are implementations for C (65), C++ (66), C#, Cobol (67), PHP, etc.
For a list of AOP projects and tools, see (68) and (64).

It is rather clear that Aspect Oriented Programming can be easily used for instrumentation needs, as in (3; 56; 28). By creating joinpoints (and their related advices) for meaningful events and then injecting them into the target system, we can study and extract information from a running system without being intrusive. AspectJ is even less intrusive than other AOP languages since it supports aspect weaving at the bytecode level, so it might not even need the original source code. This is one of the main reason behind the use of AspectJ in (3; 56).

3.2.1.3 Other solutions

The techniques presented before are the easiest to use since they have been widely studied and there is a large community of users that exchange useful information and suggestions through newsgroups and forums. However there are many other techniques that support code instrumentation, going from complete frameworks for instrumentation to very low level solutions. I will briefly introduce some of them.

The low level solutions are based on the use of code profilers, such as JVMPI (69), gprof (70), jProf (71), OProfile (72) and Valgrind (73).
Profiling allows one to learn where a program spends its time and which functions call which other functions while it is executing. This information can be used for a number of purposes.

For example, it can show which pieces of a program are slower than expected, which functions are being called more or less often than expected, etc. For the purpose of instrumentation, it can also tell what the called methods or the execution traces of the program are. Usually, profiling is achieved in three steps:

1. The program to be analyzed is compile and linked, with profiling enabled.
2. The program is executed to generate a profile data file.
3. The profiler then analyzes the profile data and presents the results to the user. Two basic information returned by the profiler are usually the flat profile and the call graph. The flat profile shows how much time the program spent in each function and how many times that function was called. The call graph shows, for each function its callers, its callees and how many times calls occurs. There is also an estimate of how much time was spent in the subroutines of each function.

Unfortunately, these approaches are usually expensive and complex and the resulting tools are hard to develop, maintain and reuse in different contexts (74). The same information can be retrieved in a more convenient way, for example, with the use of AOP.

Apart from these more low-level solutions, there are other tools, like Aprobe (75) and InsECTJ (74), which are complete frameworks for instrumentation to collect dynamic data from running systems. The first one is a commercial patented software, while the second one has been developed by the Georgia Institute of Technology and is freely available.

These two framework have a lot in common and externally they look almost the same, while

they differ on the implementation.

They both are an extensible, configurable and intuitive framework for gathering information from an executing program. These two frameworks have two main characteristics:

1. They provide a large library of probes for collecting different kinds of information for different code entities.
2. They let users define instrumentation tasks, which allow for instrumenting different entities in different parts of the code, collect different information from the different entities and process the information in a customized way.

A probe is typically associated with a program construct (e.g. a method call or the catching of an exception). For each instrumentable entity, these two frameworks can provide various information associated with that entity. For example, in the case of a method call, they both can report the target object and all of the parameters passed to the called method. Instrumentable entities are similar in spirit to joinpoints in AspectJ (62; 63), but they differ with respect to four main points:

1. Instrumentable entities let users fine tune the kind of information collected for each entity, thus improving the efficiency of the framework. (In aspect-oriented programs, a considerable amount of information is always collected at joinpoints, even if not used).
2. The set of instrumentable entities is more extensive than joinpoints. For example, there are no joinpoint counterparts for entities such as basic blocks, predicates and acyclic paths.

Collecting information for these entities using AspectJ is either complex or not possible at all.

3. The set of instrumentable entities is extensible.
4. Instrumentable entities, unlike AspectJ's joinpoints, are specialized for collecting dynamic information, rather than for extending a program's functionality.

The main difference is that Aprobe works both with Java and C, it lets the user define problem specific probes (which is rather easy since they are written in C for C programs or in Java for Java programs (75)) and does not need the source code to work (as AspectJ), while InsECTJ supports only Java, has a finite set of probes (whose implementation is not specified) and it needs to work on the source code.

These sections have covered the main instrumentation techniques. Obviously there are many more techniques in the academic/reserch area, but I only focused on working, well-defined and structured solutions.

3.2.2 Mapping from runtime observations to architecture

As said already before, mapping is about bridging the abstraction gap between system observations and architectural events. There are several issues that make this a hard problem. First, architectural events usually are not mapped one-to-one with the low-level system observation that can be extracted with any instrumentation technique. An abstract architectural event might involve many runtime events, such as object creation, library calls to runtime infrastructure, initialization of data structures and so on. But at the same time, a single low-level event might represent a series of architectural events. For example, the execution of a procedure

call between two objects might signal the creation of a connector and its attachment to the components that represent the two objects. The second problem is that, due mainly to concurrency, architecturally relevant actions are usually interleaved; so, in order to recognize multiple architectural events, we have to cope with concurrent intermediate states (3). A third problem is that there usually are different implementations of the same abstract architectural element; consider, for example, the number of ways to implement pipes. So there is no single standard that indicate what implementation patterns represent specific architectures and, at the same time, there is no single architectural style or pattern that can be used for all systems.

To overcome all those problems, some kind of language or description, that captures the way in which runtime events should be interpreted as operations on elements of the architectural style, is needed. More precisely, a dynamic approach to deterimine the architecture of a running sytem must:

1. Handle M-N mappings between low-level system events and architectural events.
2. Be able to keep track of concurrent and interleaved sets of event traces that lead to architectural events.
3. Be flexible in mapping implementation style to architectural style.

We can say that the mapping phase is the heart of every dynamic approach since it's the part that gives an "architectural meaning" to all the observed runtime events. I will now look at some studies that address this issue.

3.2.2.1 A survey of existing studies and tools

There have been a number of researches about the problem of presenting dynamic information of a running system to an observer. This information can be the system architecture (3), the sequence diagrams of runtime events (56; 55; 76), some other high-level view of the system (77) or, on a lower level side, variables, threads, activations and object interaction, etc. at runtime, as in (78; 79; 77; 28; 80). I will now take a closer look at them.

DiscoTect (3) was developed by CMU SEI mainly to address the issues about mapping for the studies of systems' architectures at runtime introduced before and to have a working tool that allows the dynamic study of systems' architectures.

Events captured from a running system are first filtered, with an an instrumentation tool, in this case AspectJ (62), to select the subset of system observations that must be considered. The resulting stream of meaningful events is then sent to the DiscoTect Runtime Engine (which is the heart of the tool).

The DiscoTect Engine takes as input a specification of the mapping, written in a language called DiscoSTEP (Discovering Structure Through Event Processing) (81; 82), it recognizes interleaved patterns of runtime events and, when appropriate, it produces a set of architectural events as outputs. The DiscoTect Engine, in order to recognize those interleaved patterns, constructs a Colored Petri Net (83). The architectural events produced are then fed to an Architecture Builder that incrementally creates an architectural model to be displayed to a user or processed by architecture analysis tools. In the default case, the architectural model is displayed in AcmeStudio (54), a customizable editing environment and visualization tool for

software architectural designs based on the Acme architectural description language (ADL) (84) that also allows some degrees of architecture analysis. What differentiate DiscoTect from other existing solutions as (31; 85), is that it does not need access to the source code to be annotated in some way. In fact, thanks to AspectJ, the system can be instrumented at the level of bytecode. This is extremely useful in the case of systems whose source code is not available as, for example, many existing legacy systems.

DiscoTect is a very interesting and useful tool that, integrated in a suite of tools, in particular with static analysis ones, should be able to achieve a complete and accurate picture of the system and its behavior. A combination with techniques that extract UML sequence diagrams, as (56), might also be interesting and useful. Since I used DiscoTect in my work, I will take a deeper look at it in the next chapters, when explaining my work.

The work proposed by Briand et al. (56; 55) is in some parts is similar to DiscoTect, but it produces UML sequence diagrams as output. The approach proposed in (56; 55), as DiscoTect, studies the behaviour of a running system through the use of AspectJ-based instrumentation, but instead of the architecture, it outputs of series of sequence diagrams. Instead of a language like DiscoSTEP, in this approach the instrumentation part writes specific text lines to a trace file which is then used to create a class diagram based on a UML-like metamodel specifically created for this purpose, called *trace metamodel*, which can be seen in Figure Figure 9. This information in the form of a class diagram is then transformed to a class diagram of a second custom-made metamodel (called *scenario diagram metamodel*, showed in Figure Figure 10) through a metamodel-to-metamodel transformation. The *scenario diagram metamodel*,

adapted from the UML 2.0 metamodel (34), is the class diagram that describes the structure of sequence diagrams, to simplify the specification of consistency rules and the generation of scenario diagrams from traces. This second class diagram is then used to generate UML sequence diagrams. The main problem is that, since it is a work in progress, the solutions offered now are only partial. In fact as for now, it can only be used in the Java/RMI context and there is no complete tool to be used, but only a case study solution is showed. But it is an interesting work that, for example, combined with a tool that extracts the architecture as DiscoTect, could present to the observer a rather complete and useful view of a system.

Shimba (76; 86), a protoype reverse engineering environment, offers functionalities similar to the ones proposed by Briand et al (56; 55). Both extract behavioral diagrams from Java programs but they differ for a few distinctive features. The first difference is about the instrumentation technique used.

In fact, Shimba uses a customized sdk debugger to collect event trace information. To capture this information, breakpoints are set for the debugger at the first line of all methods, constructors and static initialization blocks of the selected classes. Note that a hit of a breakpoint causes only a short pause of the execution of the system, giving the user no influence on deciding about the continuation of the execution. To limit the size of the generated event traces while still containing the information of interest, some prefiltering of useful and meaningful object interaction is performed.

In order to do this prefiltering, the engineer needs to know the dependencies between the classes of the system; in other words, the static structure of the software. In Shimba, this information

is extracted, viewed and examined using the Rigi reverse engineering environment (87). In this way, the user can easily select only the meaningful classes and/or methods for which the event trace information is collected. By integrating Rigi in the tool Shimba offers to the user also a static view of the reversed engineered system. The use of both static and dynamic views provides extended and useful means to analyze and understand the target system.

The second difference is about the diagrams generated. In fact in this solution the output consists of SCED scenario and state diagrams (88).

SCED scenario diagrams are similar to UML sequence digrams, but they add the concept of a subdiagram box, which is a way to nest sequence diagrams. In addition UML sequence diagrams are not practical for expressing a repetition of a set of sequent disconnected messages.

SCED state diagrams can be considered simplified UML statechart diagrams, in particular with the restriction that they cannot be used to express concurrency.

Shimba uses those two diagrams so that it can, with scenario diagrams, show the sequential interaction among several objects, while with state diagrams it can show the total run-time behavior of a certain object or method, disconnected from the rest of the system.

The use of SCED diagrams instead of the corresponding UML diagrams in Shimba, might somehow be a weakness of the tool. In fact, nowadays, UML has become the most used and known modeling language in computer science and in particular in software engineering; so the use of it could allow Shimba to be more versatile, understandable and to be integrated with other reverse engineering tools (In fact, with the use of XMI (89) a UML model can theoretically be exchanged between tools that are compliant quite easily). Anyhow, since SCED diagrams

are slight variations of UML diagrams (as stated in (86), if required, a translation or porting to the UML standard seems to be easily doable.

Walker et al. (77) developed an off-line, flexible approach for visualizing the operation of an object-oriented system, written in Smalltalk (90), at the architectural level. This technique is based on four main steps:

1. Data is collected from the execution of the system being studied and it is stored to disk.
2. The software engineer designs a high-level model of the system as a set of abstract entities that are selected to investigate the structural aspects of interest.
3. The engineer describes a mapping between the raw dynamic information collected and the abstract entities. For instance, any dynamic information whose identifier begins with "foo" (such as objects whose class name starts with "foo") might be mapped to a "utilities" entity. This mapping is applied to the raw information collected by the tool, producing an abstract visualization of the dynamic data.
4. The software engineer interacts with the visualization, and interprets it to investigate the dynamic behaviour of the system.

In this approach, dynamic system information is collected for every method call, object creation and object deletion performed by the system under study. This information consists of an ordered list of the class of the object calling the method or having the object created, and the class of the object and method being called or returning from, or the class of object being created or deleted. All the trace information is collected by instrumenting the Smalltalk virtual machine

to log the meaningful events as they occur. The collection of the trace information is performed only during portions of the execution; in fact a software engineer needs to understand dynamic problems that depends on particular conditions and parts of the studied system. In this way it is also possible to eliminate extraneous information not of interest and speed up the whole process of trace collection.

The definition of the high-level model and the mapping between dynamical retrieved data and the abstract entities composing the high-level model are almost the same as in the work by Notkin et al. (47) and Kazman et al. (48) and thus will not be explained again. The visualization technique, on the other hand, differs from those works.

In fact, to represent the information collected across a system's execution, the authors use a sequence of *cels*. Each cel displays abstracted dynamic information representing both a particular point in the system's execution and the history of the execution to that point, in other words, a cel shows the events occurred within a particular interval of the system's execution, defined as a set of n events. The user then can move through the sequence of cels sequentially o randomly. A summary view is then provided to show all the events occurring in the trace. Figures Figure 11 and Figure 12 show two subsequent cels, while Figure Figure 13 shows the summary view for the same system and execution trace of those two previous figures.

A cel consists of a canvas upon which are drawn a set of widgets. These widgets consist of:

- Boxes, each representing a set of objects abstracted within the high-level model defined by the user.

- A directed hyperarc between and through some of the boxes.
- A set of directed arcs between pairs of boxes, each of which indicates that a method on an object in the destination box has been invoked from a method on an object in the source box.
- A bar-chart style of histogram associated with each box, indicating the ages of and garbage collection information about the objects associated with that box.
- Annotations and bars within each box.
- Annotations associated with each directed arc.

Each box drawn identically within each cel represents a particular abstract entity specified by the engineer, and thus, does not change through the animation. For example, the grey rectangles labelled Clustering, SimFunc, ModulesAndSuch and Rest in Figures Figure 11, Figure 12 and Figure 13 are boxes corresponding to abstract entities of the same names.

The path of the hyperarc represents the call stack at the end of the current interval being displayed. For example, in Figure Figure 11, the current call stack only travels between the Clustering and Rest boxes (the hyperarc is shown as a dashed black line), while in Figure Figure 12, the call stack has extended to SimFunc as well.

The set of directed arcs represents the total set of calls between boxes up to the current interval (they are shown as solid black). Multiple instances of interaction between two boxes are shown as a number annotating the directed arc. The same two arcs are shown in Figures Figure 11 and Figure 12 from Clustering toRest, with 123 calls (127 in the case of Figure Figure 12), and

fromRest to SimFunc, with 122 calls (127 in the case of Figure Figure 12).

Object creation, age and destruction are a particular focus within the visualization. Each box is annotated with numbers and bars indicating the total number of objects that map to the box that have been allocated and deallocated until the current interval. The length of a bar for a given box is proportional to the maximum number of objects represented by that box over the course of the visualization. For example, the Clustering box of Figure Figure 11 shows that a total of 1127 objects associated with it had been created to this point in the execution and that 1037 of these had been garbage collected.

The histogram associated with each box shows this information as well, but in a more refined form. An object that was created in the interval being displayed has survived for a single interval. Stepping ahead one cel, if it still exists, the object has survived for two intervals, and so on. The k-th bin of the histogram shows the total number of objects mapped to the box that are of age k. To limit the number of bins in the histogram, any object older than some threshold age T is shown in the rightmost bin of the histogram. As an example, the histogram attached to the Clustering box in Figure Figure 11 indicates that all of its 1127 objects were created relatively far in the past, more than 10 intervals before the one being shown here.

Colour is used to differentiate those objects that still exist from those that have been garbage collected; each bar of the histogram is divided into a lower part, marked in a vertical-line pattern, for the living objects and an upper part, marked in a diagonal-line pattern, for the deleted objects. In Figure Figure 11, the upper part of the bar in Clustering's histogram shows that roughly 80% of the old objects have been deallocated. Light grey is used both within the

box annotations and within histograms to show objects that have just been created or deleted. Figure Figure 12, for example, shows the interval immediately after that of Figure Figure 11, in which an additional 324 objects related to Clustering had just been allocated. This allocation is shown both by the light grey portion of the upper bar and the light grey bar in the first bin of the histogram.

This approach, as it might have been already noticed, builds on the software reflexion model technique developed by Notkin et al. (47), but extends it in three fundamental ways: by applying the abstraction approach across discrete intervals of the execution with animation controls, by providing support to map dynamic entities rather than only static entities and by mapping memory (e.g. garbage collection) aspects of an execution in addition to interactions.

Jive (78) and the next researches provide a more low-level type of information about the runtime behavior of a system: variables, threads, activations and object interaction, etc. .
Jive is a real-time dynamic visualization environment developed to be used for real running programs and provide programmers with the information needed to understand what a program is doing as it is doing it.
The key to a successful real-time dynamic visualization system is obtaining trace data with minimal overhead. In order to do this, Jive breaks the execution into intervals and then diplays a summary of what the program did during each interval. In this way, the amount of data that has to conveyed from the application to the visualization tool is cut down substantially. The information gathered and provided for each interval includes:

- What classes were executing.

- The number of calls to or within each class.
- The number of synchronization calls for each class.
- What was being allocated.
- What was being deallocated.
- What threads are in the program.
- The state of each thread.
- The number of blocks caused by each thread.

This information is obtained by patching the user's program and associated libraries and system files using IBM's JikesBT byte code package (91), a Java class library that enables Java programs to create, read, and write binary Java class files. The patching is kept to a minimum by dividing the application's classes into three categories. For the class that are directly in the user's code (called datailed classes), information that considers all methods and details any nested classes for separate visualization, is provided. On the other hand, for library classes (the ones grouped into packages) only events for the initial entry into the library are generated. Finally, classes that are neither detailed nor library are treated at an intermediate level of granularity: nested classes are merged with their parent and only public methods are considered.

Once the data is available, it needs to be visualized. Jive uses what the authors call "box display": each class or thread is represented as a box on the display and within that box one or more colored rectangles. The various statistics can then be reflected, for a particular class, in the vertical and horizontal sizes of the colored display and in the color (hue and saturation) of

the displayed region.

As an example, for the class display, five simultaneous values are tipically displayed:

1. The height of the rectangle is used to indicate the number of calls.
2. The width of the rectangle is used to represent the number of allocations by methods of the class.
3. The hue of the rectangle is used to represent the number of allocations of objects of the given class.
4. The saturation of the rectangle is used as a binary indicator, to show whether the class was used at all during the interval.
5. The brightness of the box is used to represent the number of synchronization events on objects of this class.

Due to the quantity of the information that has to be gathered in real-time, the programs are reported to run with a slowdown of a factor typically between 2 and 3, but given the usual performances of of today's machine, it seems quite acceptable. What seems a bit more of a weakness of Jive is how the information gathered is visualized. In fact the visualization paradigm used does not seem very intuitive and straightforward and a novice user might take some time to get used to it. Figure Figure 14 shows a screenshot of Jive.

Jinsight is a tool developed by Pauw et al. (80) for exploring, as the name suggests, Java programs' runtime behavior visually. In particular, it shows object lifetimes and communication,

performance bottlenecks, thread interactions, deadlocks and garbage collection activity. With Jinsight, a user can explore program execution through several views:

Histogram view It visualizes the resource consumption (CPU and memory) in terms of classes, instances and methods, giving an overview of hot spots in a program's execution. The Histogram view arranges information by class. Each row of the histogram shows the name of a class followed by colored rectangles representing the instances of that class. Colors for instances and classes can depict various criteria: time spent in methods relating to that instance or class, number of calls, amount of memory consumed, number of threads in which the instance or class participates. The lines in the view that connect instances of different classes represent relationships among objects, as it can be seen in Figure Figure 15. It is possible to tell the view to indicate how each object calls, creates or refers to other objects. Seeing connections among objects is useful for detailed investigation of calling and reference relationships but the combinatorial nature of these relationships will make any program larger than "Hello world" hard to examine.

Reference Pattern view This view is used to overcome the complexity when examining the structure of a particular object. In fact, instead of showing individual instances, the Reference Pattern view groups them by type and reference, using twin squares denoting collections of objects. For example, if I need to examine an Hashtable object that points to an array containing 329 HashtableEntry objects and these objects contain 413 references to String objects, which in turn refer to 413 arrays of characters and so on. The complete data structure might contain a huge number of elements, making it difficult to visualize when

fully expanded. Visualizing the pattern of references lets the user view and understand this data structure efficiently, as it can be seen in Figure Figure 16 that shows the previous data structure visualized with the Reference Pattern View.

Execution view As the name suggests, this view helps in getting a better understanding of a program's execution. In this view, that can be seen in Figure Figure 17, time proceeds from top to bottom. Each colored stripe represents the execution of a method on an object. Moving from left to right you get deeper into the execution stack. The length of each stripe (which reflects the time spent in each method), the number of stripes (corresponding to the stack depth) and the colors (denoting the classes) characterize the behavior of each thread. By comparing lanes is possible to see the relative timing of events across threads. By zooming on a section of this view, is it possible also to see the name of the methods invoked during the execution, like a UML sequence diagram, as it can be seen in Figure Figure 18

Call Tree view This view can give more precise numerical measurements of the program execution. In fact, it organizes the sequence of methods invoked as an explorable tree, showing for each method, the percentage of time spent in it (relative to the root of the subtree in which the method resides) and the number of calls to it. A screenshot of this view can is shown in Figure Figure 19.

To collect execution traces, Jinsight can either use a specially instrumented Java virtual machine or a profiling agent along with a standard JVM. The authors focused more on the

visualization part and not much is said on how tracing is actually done and it seems that the user has to do this part by himself, without the support of a specific tool.

The work by Zaidman et al. (28) is a case study on the use of Aspect Orientation to dynamically analyze a system to regain lost knowledge on it. The features that make this work interesting are the fact that it is applied to C coded system and that it uses webmining techniques to extract useful information. In fact, most of the works I encountered deal with object oriented languages as Java or Smalltalk while C is not covered, even though it is widely used in legacy systems and thus issues about modernizing systems based on it exist.
As in DiscoTect (3) and the work by Briand et al. (56), aspects are used to extract meaningful trace information from the execution of the studied system. In this case, the aspect oriented paradigm used is aspicere (65), an AOP implementation for the C language developed by some of the same authors of this case study. Differently from DiscoTect and similarly to the work by Briand et al., aspects write out tracing statements to a file, which is then fed into analysis specific scripts. The three technique used in this case study are: webmining (or dynamic coupling based technique), frequency analysis and test coverage.
The basis for the dynamic coupling based technique is the measurement of runtime coupling between modules of a system. As a matter of fact, modules that have a high level of runtime export coupling, often contain important control structures. As such, these are the ideal candidates to study during early program comprehension. In particular, webmining techniques (usually used on the Internet to identify important web pages by studying the hyperlink structure) are applied to call graphs to uncover important classes.

The frequency based analysis is based on T. Ball (92) concept of "Frequency Spectrum Analysis" that correlates procedures, function and/or methods through their relative calling frequency. The idea is based around the observation that a relatively small number of methods and procedures are responsible for a huge event trace. As such, a lot of repeated calling of procedures happens during the execution of the program. By trying to correlate these frequencies, it is possible to extract information about the sizes of the inputset and outputset and, most interesting for us, calling relationships between methods. More precisely, in this case study, this technique is used to uncover coupling relations between procedures by looking at their calling frequencies. Test coverage analysis is used to measure the code coverage of the application that is being studied. When both static and dynamic information are available, measuring coverage becomes easy. For example, here is how to calculate procedure coverage for a particular module:

$$coverage(A) = dynamic(A)/static(A) * 100$$

Where *dynamic(A)* and *static(A)* are respectively the number of procedures called in module A and the procedures in module A.

In the future, the authors want to try to apply this techniques to Cobol written programs, obviously switching from the use of Aspicere to the use of Cobble (67), an aspect oriented paradigm for Cobol. This work is interesting, in particular because it addresses C written programs (which is the case for many legacy systems) and it uses a webmining technique. I think that the main weakness is the type and volume of the information extracted, which can

be useful to identify a program's hotspots and some relationships between methods, but it does not give a clear view of the overall structure and/or behavior of the model.

3.3 A comparison between static and dynamic analyses

Static and dynamic analyses for the reverse engineering of a software architecture both have a series of advantages and drawbacks. A dynamic approach is a lot more precise than the static one in reporting the objects that actually interacts. On the other hand, the accuracy of a dynamic approach depends heavily on the inputs used to execute the system, while a static approach presents a complete picture of what could happen at runtime, independently of the input, though this might include unfeasible situations. Black-box and white-box test techniques (93) can increase the accuracy of the dynamic approach, but we can never be sure of the completeness of it (56). Moreover, tracing, in dynamic analysis, can modify the runtime behavior of the system under study in the case of hard real-time systems or systems depending some how to time. This clearly calls for more research and experimentation in studying synergies between static and dynamic approaches.

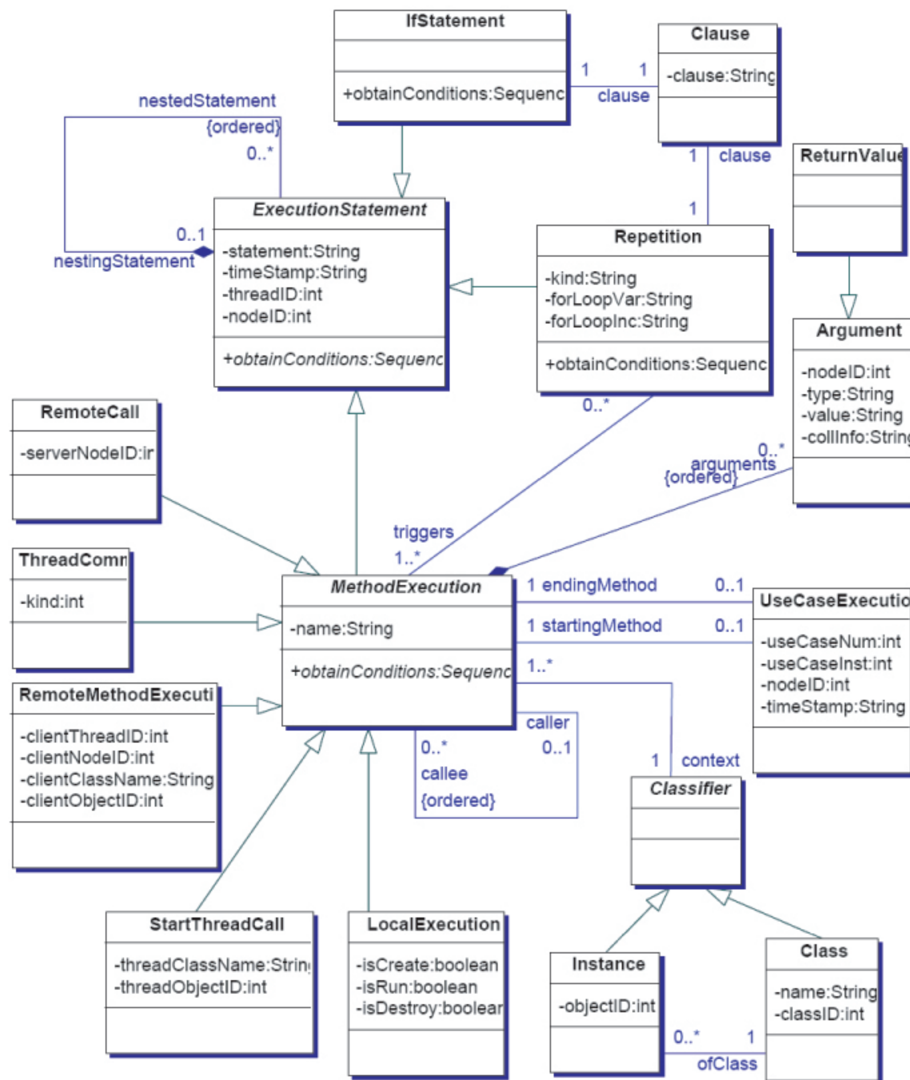


Figure 9. Trace metamodel.

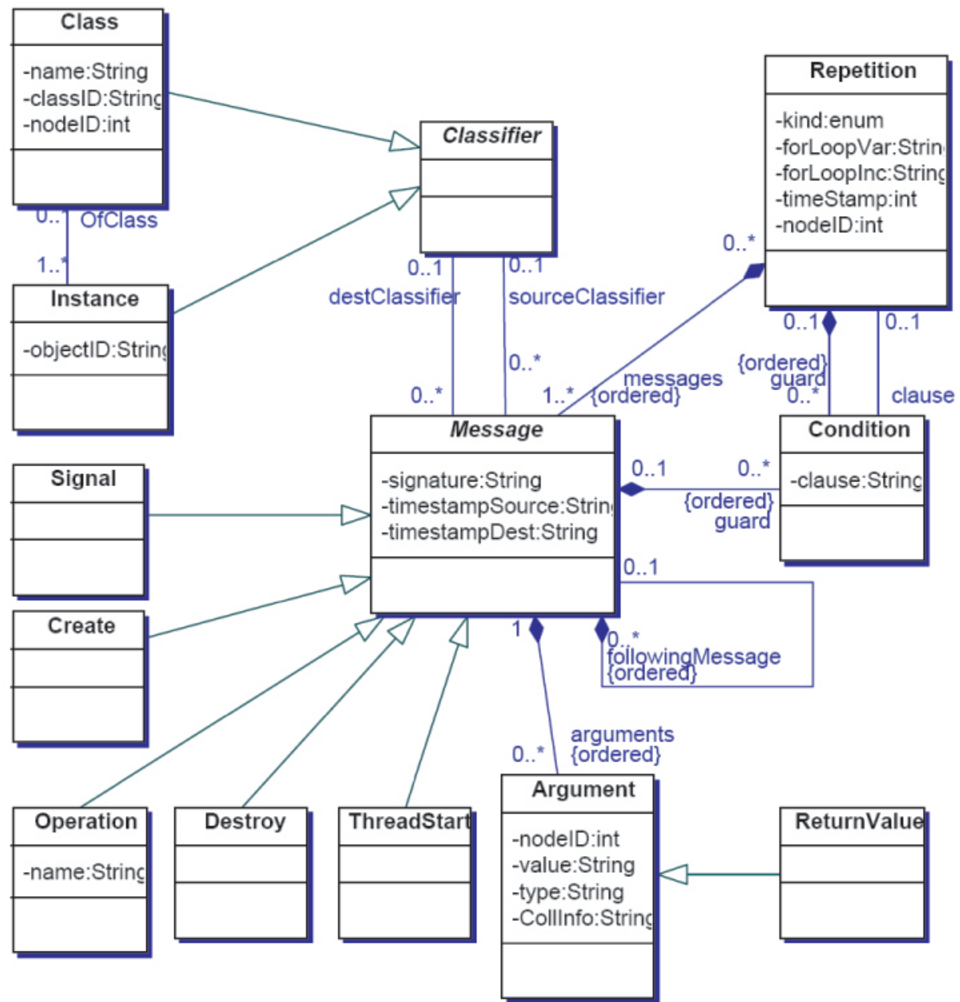


Figure 10. Scenario diagram metamodel.

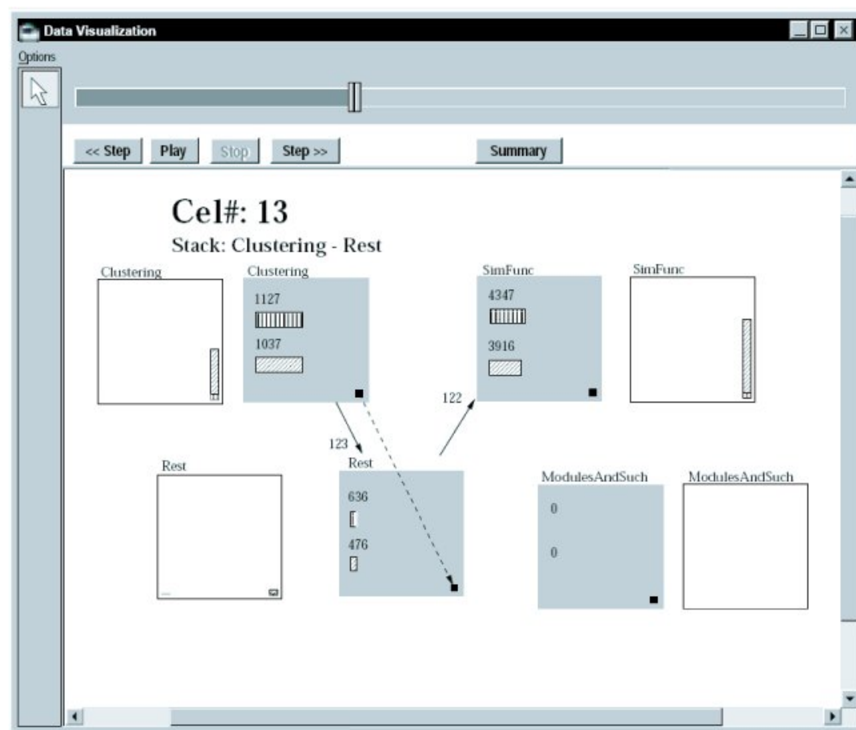


Figure 11. An example cel.

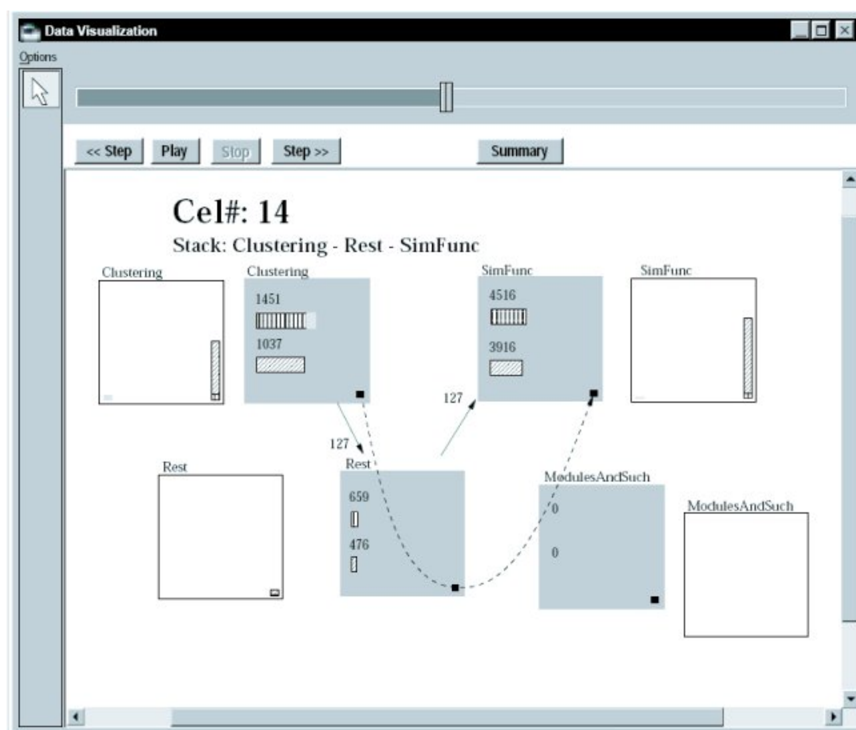


Figure 12. The cel next to the one shown in Figure Figure 11.

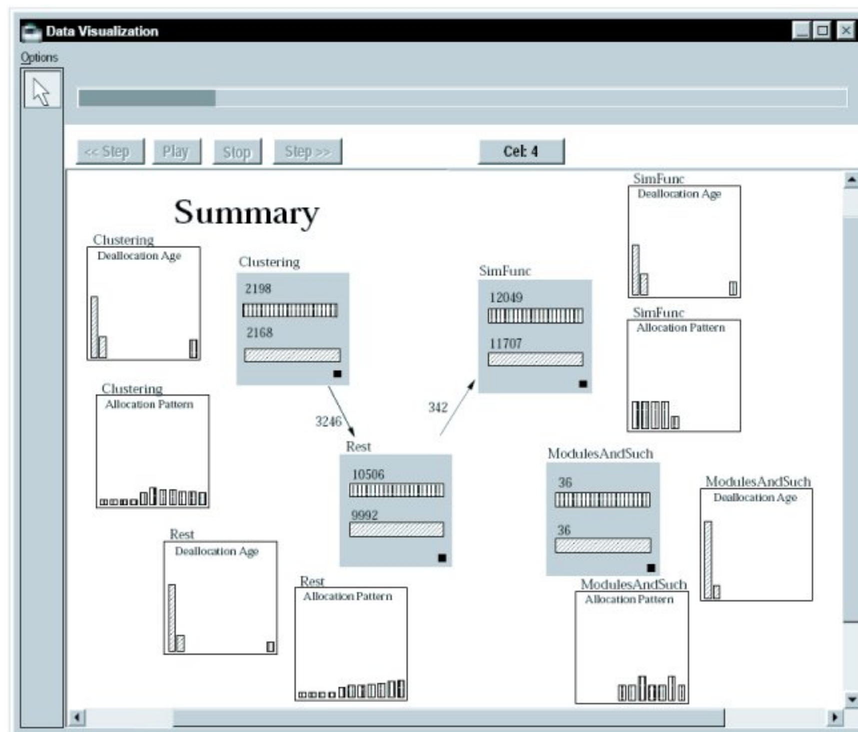


Figure 13. Summary view for the same system and execution trace of Figures Figure 11 and Figure 12.

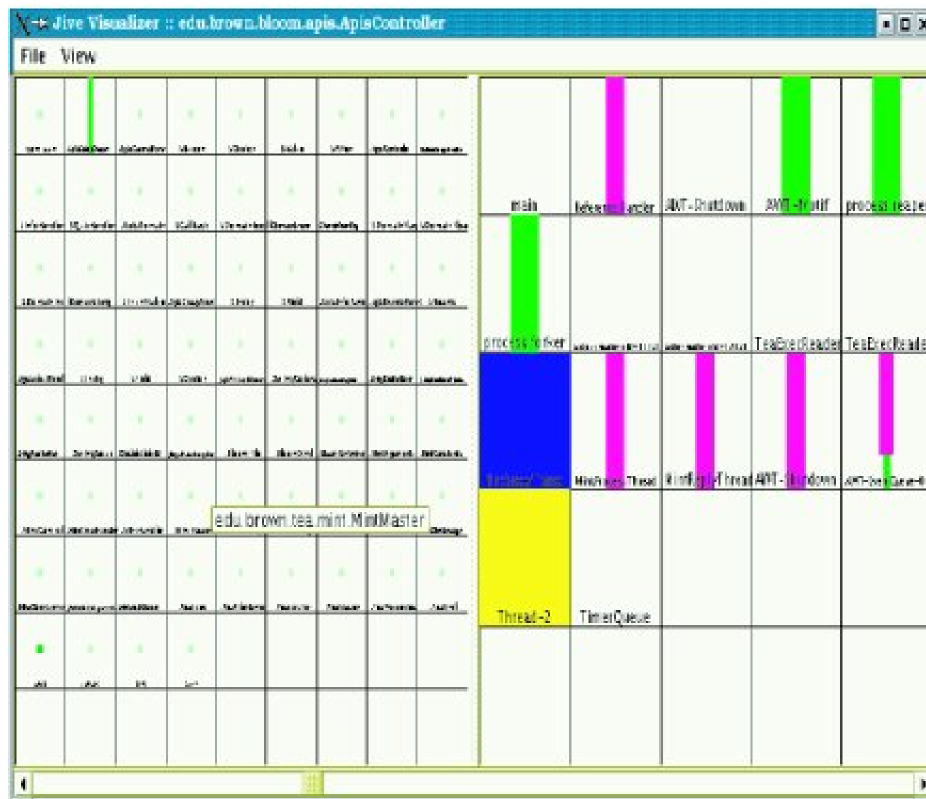


Figure 14. A screenshot of Jive.

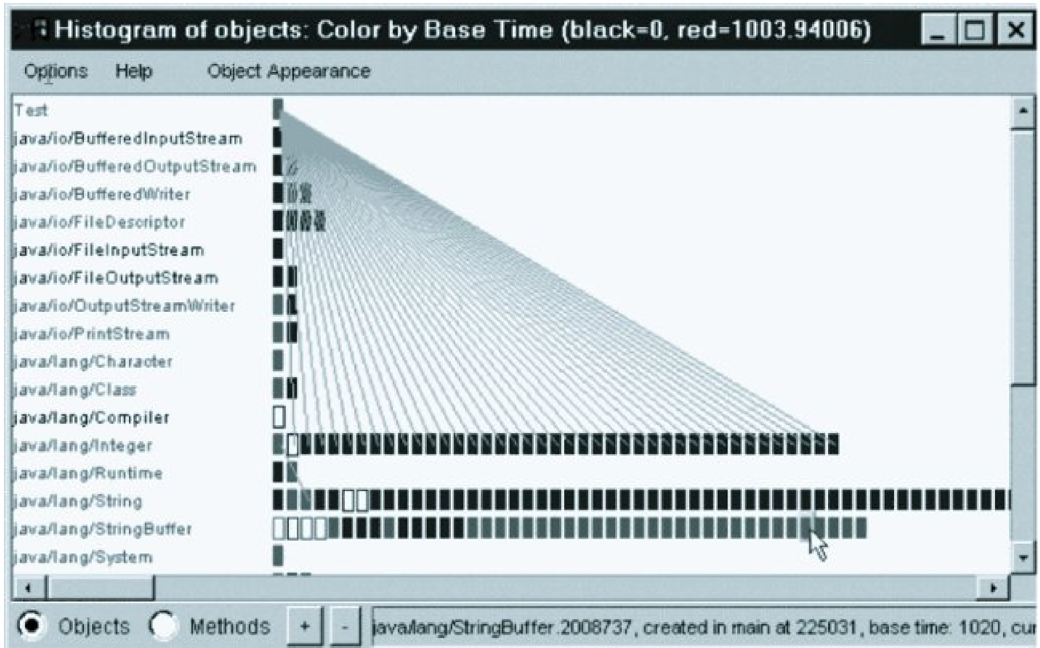


Figure 15. Screenshot of the Histogram view.

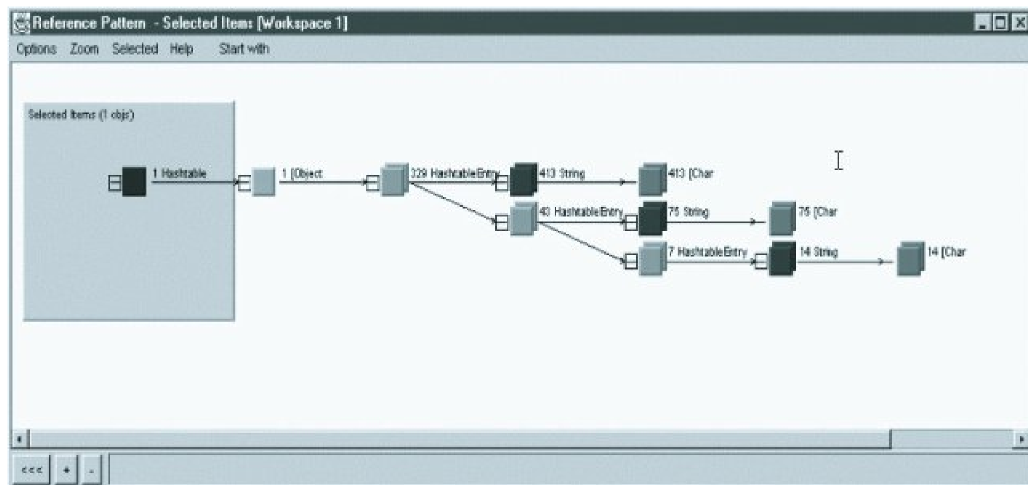


Figure 16. Screenshot of the Reference Pattern view.

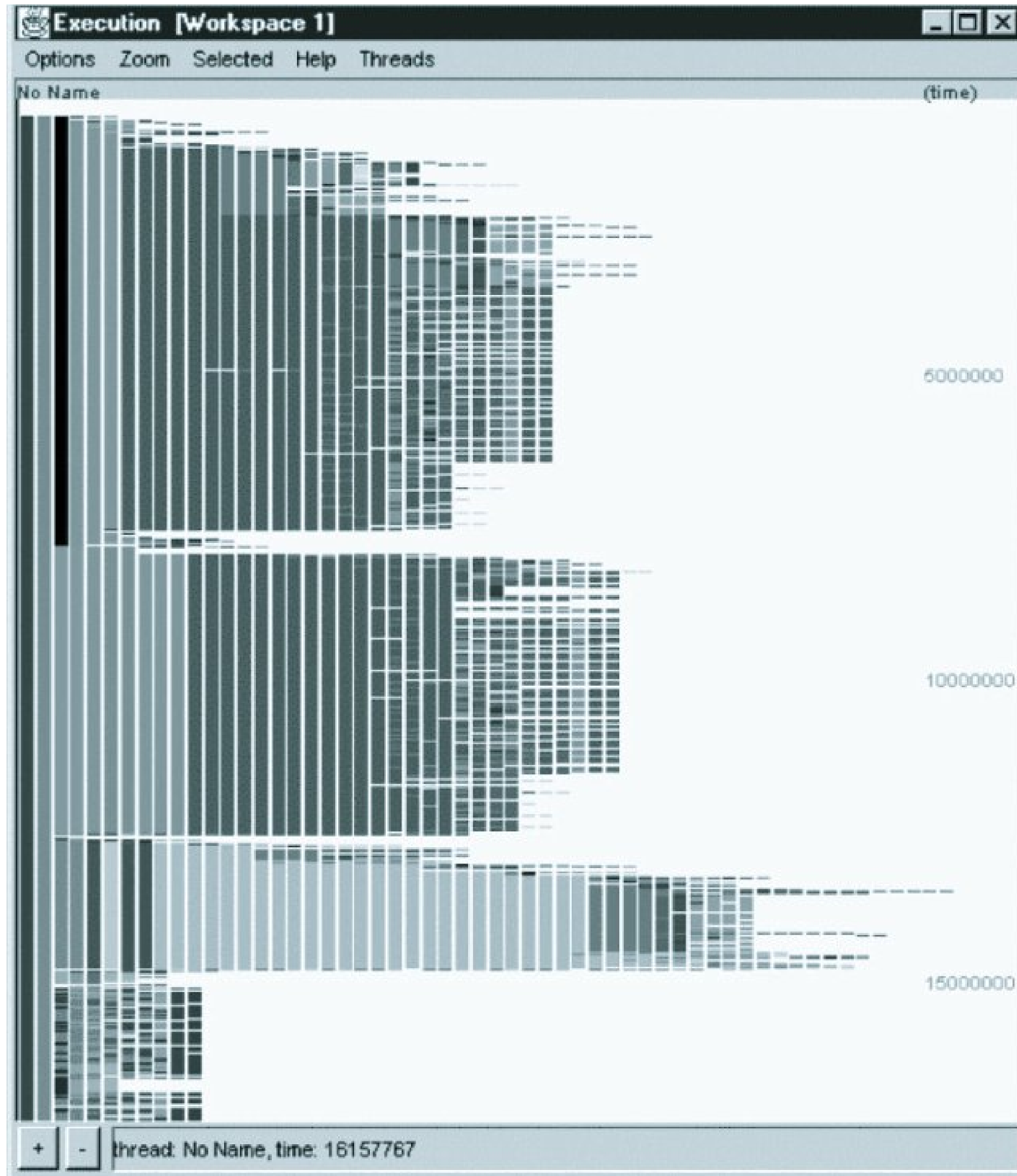


Figure 17. Screenshot of the Execution view.

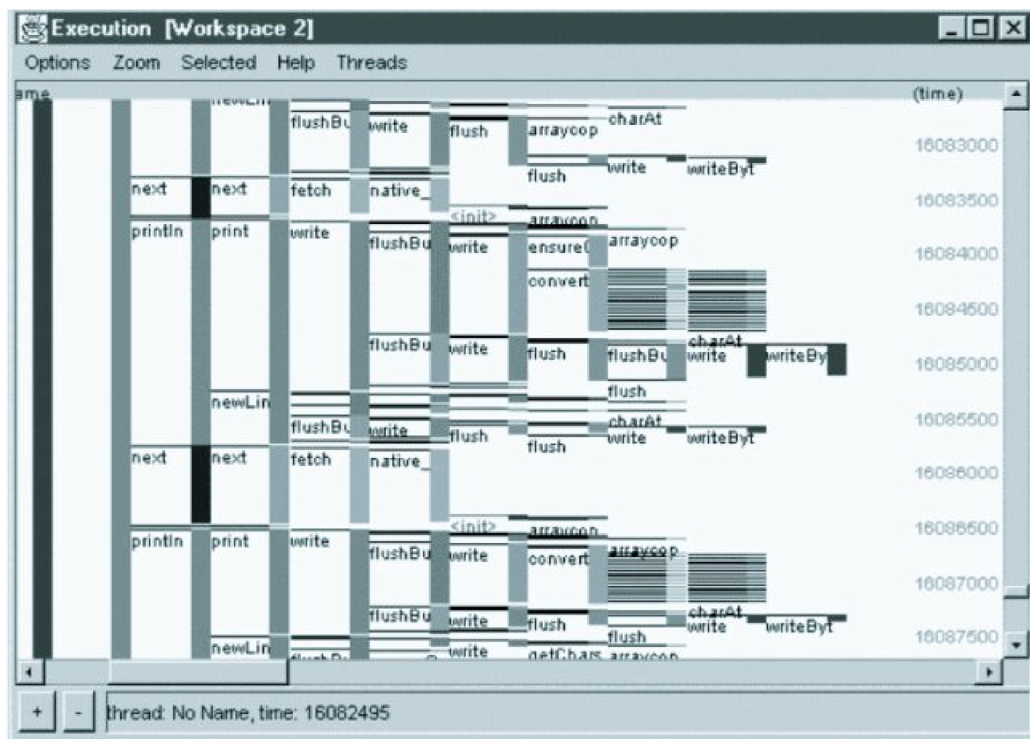


Figure 18. Portion of the zoomed Execution view.

Call Tree: Calls from various methods: 1554 occurrences [Workspace 2]

File Start with Tree Selected Help

calls	contribution %	contribution	number of calls
various methods	100.0%	2174100	1554
print	38.3%	833613	516
write	38.0%	825226	516
next	32.3%	702486	515
fetch	31.8%	691064	515
newLine	26.9%	583812	516
getString	0.5%	10628	515
close	0.3%	6405	1
<init>	0.0%	101	1
append	0.0%	100	2
valueOf	0.0%	48	1
<init>	0.0%	22	1
getChars	0.0%	20	1

Figure 19. Screenshot of the Call Tree view.

CHAPTER 4

MY APPROACH

After having studied modernization of legacy systems in Chapter 2 and software architecture analysis techniques in Chapter 3, I decided to combine them. In particular I wanted to use and adapt existing architecture analysis tools and integrate them with others created from scratch by me, in a modernization environment prototype to be used for case studies and as a starting point for future development. The main goal of this prototype is to aid and support the engineer during the whole software modernization process, from the architecture extraction and visualization to the semi-automatic transformation into a new application and the deployment of it.
In fact, as we saw in Chapter 3, many techniques to extract meaningful information from an existing system exist, but, as for now, all this useful information is only used for program understanding or, at most, to guide the engineer who will then manually modify the critical part of code. From what I know, tools and techniques that, automatically or semi-automatically transform the system with knowledge extracted from the data collected with static and/or dynamic analysis of a system, are still missing. So, the design and implementation phases of my prototype will help me to dig deeper into these issues, find what is doable and what is not and see what are the strengths and weaknesses of this approach.

I decided to use existing tools as a starting point because I think it is no use to create from scratch tools and techniques when well done and functioning artifacts offering some of the needed features already exists and are available. Moreover, my goal is not to create a new architecture

extraction technique but to create a software modernization environment prototype.

As a starting point, I decided to target only Java programs, since, the most complete and working tools that already exists in the academic and research world address the analysis and transformation of Java programs.

4.1 Overview of the proposed solution

In the following sections I will go through all the parts that compose my work. For each of them, I will explain the existing tools used, their integration in my prototype, the parts created by me from scratch and the motivations leading to my choices. The order I will follow in this explanation reflects the order of usage of the single parts in the tool, during a hypothetical modernization project. In the beginning the system is studied to find out if its actual architecture follows a defined and known paradigm; if it does, it is visualized to the user. Then, depending on the architecture found, the original system is semi-automatically transformed using the desired new paradigm; in my case, it is transformed into a Web service. Finally the Web service is built and deployed on the target application server. These steps, which represent also the sequence of my explanation, are schematized in Figure Figure 20.

The approach is implemented as an IBM's Eclipse IDE framework (94) plugin. I will describe the design and implementation of this solution in the next chapter.

4.1.1 Architecture extraction

In this section I will explain the solution used in my tool to extract the architecture from a system.

After having studied both static and dynamic architecture analysis, I decided to use a dynamic

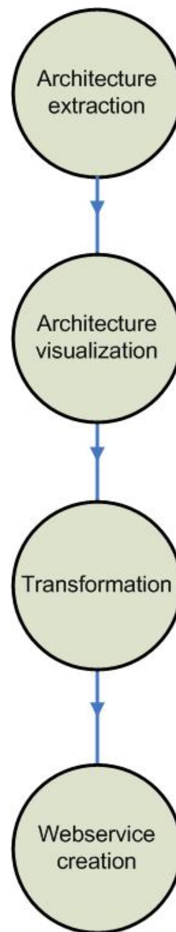


Figure 20. The modernization steps of my approach.

approach because I think that it can give a better and more precise view of the system real structure for my purpose, even though it has its drawbacks, as already exposed in Chapter 3 and in particular in Section 3.3. In fact, a static analysis does not show what actually happens in a system (56): the runtime structures, that are the essence of most of the architectural descriptions, are ignored. So a dynamic approach is a lot more precise than the static one in

reporting the objects that actually interacts.

For example in the case of an RMI client-server application, a static analysis might show the existence of a server and some clients with their methods, but it will not show some very important information as the remote methods of the servers that are actually called by the clients and the ones that are unused. To an extreme, the static view, due to the use of polymorphism, might not even show that the application is based on an RMI client-server, thus making the analysis useless or even erroneous.

Given these premises, I decided to use DiscoTect, because, as we will see in the next section, it allows to extract meaningful events from a system at runtime to check if the system's architecture is consistent with some known design paradigm and then outputs the architecture found. Of course, at first I examined and tested it, to see if it offered the functionalities I needed and if it actually worked. The other dynamic analysis techniques I reviewed, such as (56; 55; 76; 42), were discarded because they extract behavioral diagrams like UML sequence diagrams or statecharts, but do no help discover whether a system's architecture implements a known architectural paradigm. My goal, instead, is to guide a transformation once such style has been discovered

I will now show how DiscoTect is structured and how it works.

4.1.1.1 DiscoTect

DiscoTect (3) was developed by CMU SEI mainly to address the issues about mapping for the studies of systems' architectures at runtime introduced in Section 3.2.2 and to have a working tool that allows dynamic study systems' architectures.

Events captured from a running system are first filtered, with an an instrumentation tool, in

this case AspectJ (62), to select the subset of system observations that must be considered. The resulting stream of meaningful events is then sent to the DiscoTect Runtime Engine (which is the heart of the tool).

The DiscoTect Engine takes in a specification of the mapping, written in a language called DiscoSTEP (Discovering Structure Through Event Processing) (81; 82), it recognizes interleaved patterns of runtime events and, when appropriate, it produces a set of architectural events as outputs. The DiscoTect Engine, in order to recognize those interleaved patterns, constructs a Colored Petri Net. The architectural events produced are then fed to an Architecture Builder that incrementally creates an architectural model to be displayed to a user or processed by architecture analysis tools. In the default case, the architectural model is displayed in AcmeStudio (54), a customizable editing environment and visualization tool for software architectural designs based on the Acme architectural description language (ADL) (84) that also allows some degrees of architecture analysis, I will talk more about it later on.

The overall structure of the system is illustrated in Figure 21. I will now dig a bit deeper into the description of this system.

Events captured from a running system are first filtered to select the subset of system observations that must be considered. This is done with the use of an instrumentation tool, in this case AspectJ (62). The resulting stream of meaningful events is then sent to the DiscoTect Runtime Engine.

The DiscoTect Engine takes in a specification of the mapping, written in a language called DiscoSTEP (Discovering Structure Through Event Processing) (81; 82), it recognize interleaved

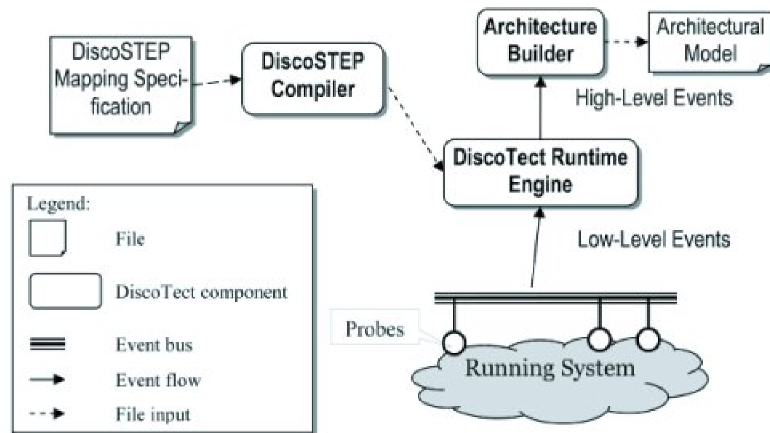


Figure 21. The DiscoTect structure.

patterns of runtime events and, when appropriate, it produces a set of architectural events as outputs. The DiscoTect Engine, in order to recognize those interleaved patterns, constructs a Colored Petri Net. The architectural events produced are then fed to an Architecture Builder that incrementally creates an architectural model to be displayed to a user or processed by architecture analysis tools.

So the three main components are:

DiscoSTEP Mapping Specification. A language called DiscoSTEP was developed for specifying mappings between low-level and architecture events. This language was implemented using JavaCC, a Java-based compiler generator (95).

A DiscoSTEP specification has three main elements: Event types, Rules and Compositions.

Event types are the kind of events that are consumed (runtime events) and produced (architectural events) by a DiscoSTEP program.

Rules determine how to map specific sets of runtime system events to architectural events. They generate, as output, new events for other rules or architectural events. Rules are composed of inputs, outputs, triggers and actions. Inputs and outputs declare the input events that the rule cares about and the output events that the rule generate. Triggers are predicates over the input events, written in XQuery (96), which are parsed and executed by Saxon XQuery processor (97). Actions are assignments to the output events, written in XQuery. Since a single rule is usually only a little fragment of what is needed to create an architecture, rules are collected between each other through the use of compositions. Compositions bind an output a rule to an input of a second rule.

These specifications need to be compiled with the DiscoSTEP compiler, which generates as output the compiled file, which is then used by the DiscoTect Runtime Engine.

The execution semantics of these mappings is defined using Colored Petri Nets. For a complete description of the translation from DiscoSTEP mappings to Colored Petri Nets see (3) and (82).

In Section 4.1.1.3 I will provide a complete explanation on the DiscoSTEP language, in fact, since it is a key point in my work, I decided to have a whole section for its explanation.

DiscoTect Runtime Engine. The runtime engine takes events from the running program and a compiled DiscoSTEP specification as inputs and then runs the DiscoSTEP specification to produce architecture events. System runtime events are first intercepted and if they

are meaningful, they are converted into XML (Extensible Markup Language) (98) streams by probes, the instrumentation part, written in AspectJ. The resulting stream of filtered meaningful events is then emitted to a JMS (99) event bus under the topic *DiscoTectSystem* (DiscoTect uses JBoss (60) as the JMS server). These events can be of two types: init and call events.

An init event represents the initialization or creation of an object through the use of a constructor. So the output XML string in this case contains a timestamp, a unique identifier of the instance, the name of the constructor and the eventual arguments passed to the constructor.

A call event represent the call of a particular method of a class, by a caller. So in this case, the output XML string will contain a timestamp, an identifier of the caller and the called, the name of the called method and the list of the eventual arguments of the method.

The DiscoTect Runtime Engine is listening to the JMS bus and receives each of these events as inputs, then it uses a specific DiscoSTEP specification to recognize interleaved patterns of runtime events and, when appropriate, outputs to the JMS event bus, under the topic *AcmeStudioRemote*, a set of architectural events formatted as XML strings.

I will now show the steps done by the DiscoTect Runtime Engine when it runs DiscoSTEP specifications to discover an architecture to better understand how everything works (partially adapted from (3)):

1. When an event is received, through the JMS event bus under the topic *DiscoTectSystem* by the DiscoTect Runtime Engine, it is associated with any rule of the current DiscoSTEP specification that accepts that type of event as input.
2. For any rule that has a value for each of its inputs, the trigger is evaluated.
3. If a trigger matches for a set of input events, the action related is executed.
4. For output events that are composed with other rules, the event is sent as inputs to those rules. For output events not composed with other rules, the are placed on the JMS event bus (under the topic *AcmeStudioRemote*) as architectural events to be read by an architecture builder.

Architecture Builder. The architecture builder picks up these architectural events produced by the DiscoTect Runtime Engine and incrementally constructs an architectural description, which can then be displayed to a user or processed by other architecture analysis tools. In this solution, the Acme architecture description language (84) and its eclipse-based tool AcmeStudio (54) was used.

Acme was choosen for two reasons: it's developed by the same people that developed DiscoTect and AcmeStudio has analysis capabilities that can be used to check architecture with respect its style or to conduct analyses such as performance or schedulability. Note that the developer say that in principle any architecture description could be used (3), but it has not been proved yet.

DiscoTect has several weaknesses. Some are illustrated in (3), while others where found while learning how to use the whole system and creating and running examples.

The first is that it only works if an implementation obeys regular coding conventions and, thus, I can write or use previuosly written probes. Completely ad-hoc code will not benefit from this technique. Second, it only works if one can identify a target architectural style so that the mapping "knows" the output vocabulary. Third, as any other analysis technique based on runtime observations, it can only analyze what is actually executed, so it is impossible to know for sure if there is any execution that might violate a set of style constraints or produce a different architecture. This problem could be limited with the use of some sort of architectural coverage metrics, similar to coverage metrics for testing, to know with some certainty what the architecture of the system under study is, after having ran the system with various inputs and, thus, having exercised a sufficiently comprehensive part of the system. Fourth, using a different architecture builder and, thus, a different architecture description language and its related tool, would require a series of modifications, so it is not as painless as it might seem by reading the paper. Fifth, the DiscoSTEP mapping needs to be created through an iterate-and-test paradigm and, thus, the results are somewhat dependent on the skill of the creator of the DiscoSTEP specifications. While trying out DiscoTect I encountered this problem.

In fact creating a working DiscoSTEP specification is not an easy task at all for a number of reasons. First, there are small differences between the formal specifications of the DiscoSTEP language presented in (81) and the ones that actually need to be used. For example, as for now, only the bidirectional composition is implemented. Second, the existing documentation about DiscoSTEP is not of a big help in writing new specifications, so I had to "reverse engineer" the existing examples in order to write my own specification and than proceed through a process

of “trial and error” to see if it worked (at times it got really frustrating). Third, to write a specification that can be applied to a whole architectural style and not just to that example, many examples of a system using that same architectural styles and coding conventions are needed. This suggests that this approach is best applied in cases that use well known and widely used styles and conventions.

I will now take a closer look at two vital parts of DiscoTect of particular interest: the probe aspects (written in AspectJ) used for tracing and the DiscoSTEP specifications. In fact they are the parts of DiscoTect that needs to be modified in order to recognize a particular architecture.

4.1.1.2 Probe aspects

The aspects used in DiscoTect are standard AspectJ aspects, except for the addition of a few tricks.

First of all they need to import the library *edu.cmu.cs.acme.discotect.probe.** in order to extend the class *JMSXMLEventEmitter* which contains the methods to emit the meaningful events to the JMS bus as XML strings. These two methods are *emitInit* and *emitCallEvent*, which, as it can be easily infer, are used to emit respectively init events and call events (which have been described before).

The emitInit method requires as input parameters the joinpoint where the method was weaved and the object that resulted from the call the constructor.

So this method should usually be called when the constructor that I intend to monitor returns. Note that the object resulted from the call to a method can be retrieved in AspectJ by adding

"returning (Object result)" to the specific advice, as it can be seen in the following brief example [1].

```
pointcut constructor (): execution (ExampleClass.new(..));


after() returning (Object result): constructor(){
  emitInit (thisJoinPoint, result);
}
```

On the other hand, the emitCallEvent method requires as input parameters the joinpoint of where this call was weaved, the caller that called the method and a tag, in the form of a string, whose usefulness is unknown, since the poor documentation does not say anything about it and it looks like it specifies that the event monitored is a call event, which is already a known fact, since the method is called emitCallEvent. In any case, by always setting this parameter to the string "call", everything will work, with any other value, the system will crash since it will not recognize the system event emmitted.

This method can either be called when the method that I intend to monitor returns or before it executes. Note that the caller of a method can be retrieved in AspectJ by adding "this (caller)" to the specific advice, as it can be seen in the following brief example.

```
pointcut connectMethod (): call (* *.connect (..));


before(Object caller): this (caller) && connectMethod(){
  emitCallEvent (thisJoinPoint, caller, "call");
}
```

---

[1] If unfamiliar with the the syntax and meaning of the AspectJ pointcuts and advices, or with AspectJ in general, the book by Colyer at al. (100) is a very useful resource.

From the information about the joinpoint passed as a parameter, these two methods initialize the connection to the JMS, extract all the necessary information, encapsulate it into an XML string and then send it on the JMS bus.

It is clear that, once familiar with aspect oriented programming and in particular AspectJ, writing “probe aspects” (3) is rather straighforward.

In the next chapter I will show and explain the aspects I actually wrote for my prototype.

4.1.1.3 DiscoSTEP specifications

As already mentioned before, a DiscoSTEP specification has three main elements: Event types, Rules and Compositions (note that, when writing a specification, this order has to be followed).

Event types are the kind of events that are consumed (runtime events) and produced (architectural events) by a DiscoSTEP program. They are represented in XML so that it is possible to costumize events for a particular implementation or architectural style. To do so, XML Schemas are used to specify the valid XML representations that can be used as events by DiscoSTEP. Those XML schemas have to be specified at the beginning of the specification with the following syntax[1]:

```
import < [XML Schema name or complete path] >
```

Note that, if the schema does not resides in the same directory of the specification, the complete path has to be specified.

A schema that has always to be imported is *sys_ events.xsd*, which specify the XML representa-

---

[1]In this and in the following examples about DiscoSTEP specifications, user defined strings will be contained by square brackets to distinguish them from the DiscoSTEP keywords

tion of the events generated by the probe aspects through the JMSXMLEventEmitter, namely, the init and call events. Here is the schema describing these two events:

```
<schema>
  <element name="init">
    <complexType>
      <sequence>
        <element name="arg" minOccurs="0" maxOccurs="unbounded">
          <complexType>
            <attribute name="name" type="string"/>
            <attribute name="value" type="string"/>
          </complexType>
        </element>
      </sequence>
      <attribute name="constructor" type="string"/>
      <attribute name="instance_id" type="string"/>
      <attribute name="timestamp" type="string"/>
    </complexType>
  </element>

  <element name="call">
    <complexType>
      <sequence>
        <element name="arg" minOccurs="0" maxOccurs="unbounded">
          <complexType>
            <attribute name="name" type="string"/>
            <attribute name="value" type="string"/>
```

```
          </complexType>
        </element>
      </sequence>
      <attribute name="method_name" type="string"/>
      <attribute name="caller" type="string"/>
      <attribute name="callee" type="string"/>
      <attribute name="timestamp" type="string"/>
    </complexType>
  </element>
</schema>
```

After having imported the necessary schemas, the event types that will be used in the specification have to be declared. There are two classes of events: output events and input events. Input events are the ones consumed by the rules of the specification, while output events are the ones produced by those rules.

The events types of a specification have to be declared with the following syntax:

```
event {
 input {
  [event type];
  ...
 }
 output {
  [event type];
  ...
 }
}
```

Note that an event type can be declared both as an input and as an output event at the same time.

Rules determine how to map specific sets of runtime system events to architectural events. They generate, as output, new events for other rules (this is why an event type can be declared both as an input and an output event) or architectural events. Rules are composed of inputs, outputs, triggers and actions. Inputs and outputs declare the input events that the rule cares about and the output events that the rule generate. Triggers are predicates over the input events, written in XQuery (96), which are parsed and executed by Saxon XQuery processor (97). Actions are assignments to the output events, written in XQuery.

Here is the outline of a generic rule:

```
rule [rule name] {
 input {
  [event type] $[identifier];
  ...
 }
 output {
  [event type] $[identifier];
  ...
 }
 trigger {? [XQuery predicates] ?}
 action {? [XQuery predicates] ?}
}
```

Since a single rule is usually only a little fragment of what is needed to create an architecture, a DiscoSTEP specification consists of many rules that are connected between each other through the use of a series of input/output bindings collected in a Composition.

A Composition contains a series of bindings between an output event of a rule and an input event of a second rule. This binding can either be directional or bidirectional; in the first case the rule consumes the input event, in the second case the rule fires on an input event without consuming it. Here is the outline of a generic composition containing a bidirectional and a directional bindings:

```
composition [composition name]{
  [source rule name].$[source rule output event identifier]<->
      [target rule name].$[target rule input event identifier]
  [source rule name].$[source rule output event identifier]->
      [target rule name].$[target rule input event identifier]
  ...
}
```

Having covered all the elements composing a DiscoSTEP specification, I will now show the complete overall structure of it:

```
import < [XML Schema name or complete path] >
...
event {
 input {
 [event type];
 ...
 }
 output {
  [event type];
  ...
 }
```

```
}

rule [rule name] {
 input {
  [event type] $[identifier];
  ...
 }
 output {
  [event type] $[identifier];
  ...
 }
 trigger {? [XQuery predicates] ?}
 action {? [XQuery predicates] ?}
}
...
composition [composition name]{
  [source rule name].$[source rule output event identifier]<->
     [target rule name].$[target rule input event identifier]
  [source rule name].$[source rule output event identifier]->
     [target rule name].$[target rule input event identifier]
  ...
}
```

These specifications (written in a file with a .epp extension) need than to be compiled with the DiscoSTEP compiler, which first checks the DiscoSTEP syntax, the XQuery syntax of the predicates of triggers and actions and the conformance of the events with the imported schemas

and then generates as output the compiled file (with a .epo extension), which is then used by the DiscoTect Runtime Engine.

The execution semantics of these mappings is defined using Colored Petri Nets. For a complete description of the translation from DiscoSTEP mappings to Colored Petri Nets see (3; 82).

I will complete this section with a quick example. Let's say I want to see if an application is a socket server and that I already instrumented the code with AspectJ, so that when *ServerSocket server = new ServerSocket(...)* statement is executed an init event is generated and when *Socket socket = server.accept()* statement is executed a call event is generated.

The XML strings representing those two events generated and sent on the JMS bus are:

```
<init constructor="ServerSocket" instance_id="10" timestamp="100">
  <arg name="port" value="1111"/>
</init>


<call method_name="ServerSocket.accept" callee="10" caller="11"
   timestamp="200"/>
```

Note that these string are perfectly constistent with the XML schema I previously showed and that timestamp, instance_id, caller and callee values were arbitrarly choosen by me just to give meaning to the example.

In order to recognize the creation of a socket server I need to write the following rule:

```
rule ServerCreation {
 input {
  init $i;
 }
 output {
```

```
  string $server_id;
  create_component $create_server;
 }
 trigger {?
  contains($i/@constructor, "ServerSocket")
 ?}
 action {?
  let $server_id := $i/@instance_id;
  let $create_server := <create_component name="{$server_id}"
    type="SocketServer"/>;
 ?}
}
```

This rule recognizes the init even generated by the execution of the *new ServerSocket()* since it takes as input an init event and its activation is triggered when the name of the constructor contains the string "ServerSocket". This rule outputs two events: a string containing the instance_id (a unique identifier created by JMSXMLEventEmitter) of the newly created server and a create_component, an XML string representing the architectural event related to the creation of a component[1], in this case of "SocketServer" type with instance_id as its name. Then I need to recognize the call of the *ServerSocket.accept*; which is done with the following rule:

```
rule ClientConnection {
 input {
```

[1]Note that, as already mentioned, this part is customizable, so it can follow any user defined XML schema, in order, for example, to be interpreted by a visualization tool. In this example the syntax was created on the fly by me just to write a complete example, so it actually does not follow any defined schema

```
  call $c;
  string $server_id;
 }
 output {
  create_component $create_client;
  create_component $create_connection;
 }
 trigger {?
  contains($c/@method_name, "ServerSocket.accept")
    and $c/@callee_id=$server_id
 ?}
 action {?
  let $create_component := <create_client name="{$c/@caller_id}"
    type="SocketClient"/>;
  let $create_connection := <create_connection name=concat
    ($c/@caller_id, "-", $server_id) type="Connector"
    from="{$c/@caller_id}" to="{$server_id}"/>;
 ?}
}
```

This rule recognizes the call event generated by the execution of the *ServerSocket.accept* of the server represented by the input string *server_ id*. In fact, it takes as input a call event and a string, and its activation is triggered when the name of the method called contains the string "ServerSocket.accept" and the callee coincide with the the input string *server_ id*. This rule then outputs two architectural events, the first one related to the creation of a component, in this

case of "SocketClient" type with caller_id as its name, the second one related to the connection of this component with another component whose name is the input string *server_ id*.

In order to make the input string *server_ id* in the second rule coincide with the identifier of the server created in the first rule, I need to add to write the following binding in the composition part of the specification:

```
composition Client-Server-composition{

  ServerCreation.$server_id->ClientConnection.$server_id;

}
```

I now only need to do two last things.
First I have to import the necessary schemas, in this case *sys_ events.xsd* that defines the init and call events and *custom_ architectural_ events.xsd* that hypothetically[1] defines the architectural events used in the specification (namely, create_component and create_connection).
Second I have to declare the event types used in the specification and whether they are input and/or output events. In this case the input events are, as for any DiscoSTEP specification, init and call, but also string, since the "ClientConnection" rule needs to take as input a string containing the identifier of the newly created server. The output events, on the other hand, are create_component so the architectural events output to the JMS bus and, obviously, also string. So here is the complete specification:

```
import <sys_events.xsd>

import <custom_architectural_events.xsd>


event{
```

[1]In reality this file does not exist since the syntax for this events was created on the fly by me just to write a complete example

```
 input{
  init;
  call;
  string;
 }
 output{
  string;
  create_component;
 }
}

rule ServerCreation {
 input {
  init $i;
 }
 output {
  string $server_id;
  create_component $create_server;
 }
 trigger {?
  contains($i/@constructor, "ServerSocket")
 ?}
 action {?
  let $server_id := $i/@instance_id;
  let $create_server := <create_component name="{$server_id}"
    type="SocketServer"/>;
```

```
 ?}
}

rule ClientConnection {
 input {
  call $c;
  string $server_id;
 }
 output {
  create_component $create_client;
  create_component $create_connection;
 }
 trigger {?
  contains($c/@method_name, "ServerSocket.accept")
    and $c/@callee_id=$server_id
 ?}
 action {?
  let $create_component := <create_client name="{$c/@caller_id}"
    type="SocketClient"/>;
  let $create_connection := <create_connection name=concat
    ($c/@caller_id, "-", $server_id) type="Connector"
    from="{$c/@caller_id}" to="{$server_id}"/>;
 ?}
}

composition Client-Server-composition{
```

```
  ServerCreation.$server_id->ClientConnection.$server_id;
}
```

4.1.2 Architecture visualization

By using DiscoTect, I already had, hassle free, the visualization of the outcome and its eventual analysis, through AcmeStudio.

In fact, the architectural events produced by the DiscoTect Runtime Engine can be theorically processed and displayed by any architecture analysis tools; but in this first solution offered and in its example cases, the Acme architecture description language (84) and its eclipse-based tool AcmeStudio (54) was used. So I also decided to use AcmeStudio for several reasons. First, DiscoTect is tailored to use AcmeStudio and even though the authors say that, in principle, any architecture description language and tools can be be used (3), it still has to be proven yet; so the use of another architecture description language and a related tool might be, by size and effort an independent research project by itself. Second, AcmeStudio is already written as an Eclipse plugin, which is the desired and planned solution also for my prototype, so this would allow me to combine them (and if necessary make them communicate) quite easily and have all the required functionalities in a single workbench, namely Eclipse.

I will now take a closer look at AcmeStudio.

4.1.2.1 A quick tour of AcmeStudio

AcmeStudio is an adaptable front-end that may be used in a variety of modeling and analysis applications, written, as said before, as an Eclipse plugin, that supports the Acme ADL (84).

Acme is a general purpose ADL that supports component and connector architectures. It

provides a system for defining new component and connector types, rules about their composition and facilities for packaging these up in architectural styles.

AcmeStudio already contains seven built-in architectural styles:

- Client and server family.
- Layered family.
- Pipes and filters family.
- Publish subscribe family.
- Shared data family.
- Three tiered family.
- Tiered family.

Note that in those built-in styles, there is also a client server family, which gives me the possibility to represent the eventual RMI client-server architecture found. This is also another reason that led me to develop, as a first step, a modernizer tool for RMI applications. In fact, by doing so, I already had the AcmeStudio architectural style for the representation, instead of creating a new one.

Figure Figure 22 shows AcmeStudio displaying three architectural models and one architectural style being developed. On the left of each architecture window is a type palette, which displays the available vocabulary for a particular domain (architectural style). For example, in the bottom-left window is an architectural model in a pipe and filter style; the palette allows an architect to drag pipes, filters, data stores and their associated interfaces into the diagram

to create the model. On the other hand, the top-left window depicts a model in a client-server style. The right window is an overview of the definition of the types of a custom made style (PFFam in this case) in this view the user can define new components, connectors, roles, etc. . The bottom-most window shows selected elements from the models and styles in more detail, displaying their properties, rules, substructure, and typing. This window also allows the user to enter values for properties, define rules, etc.

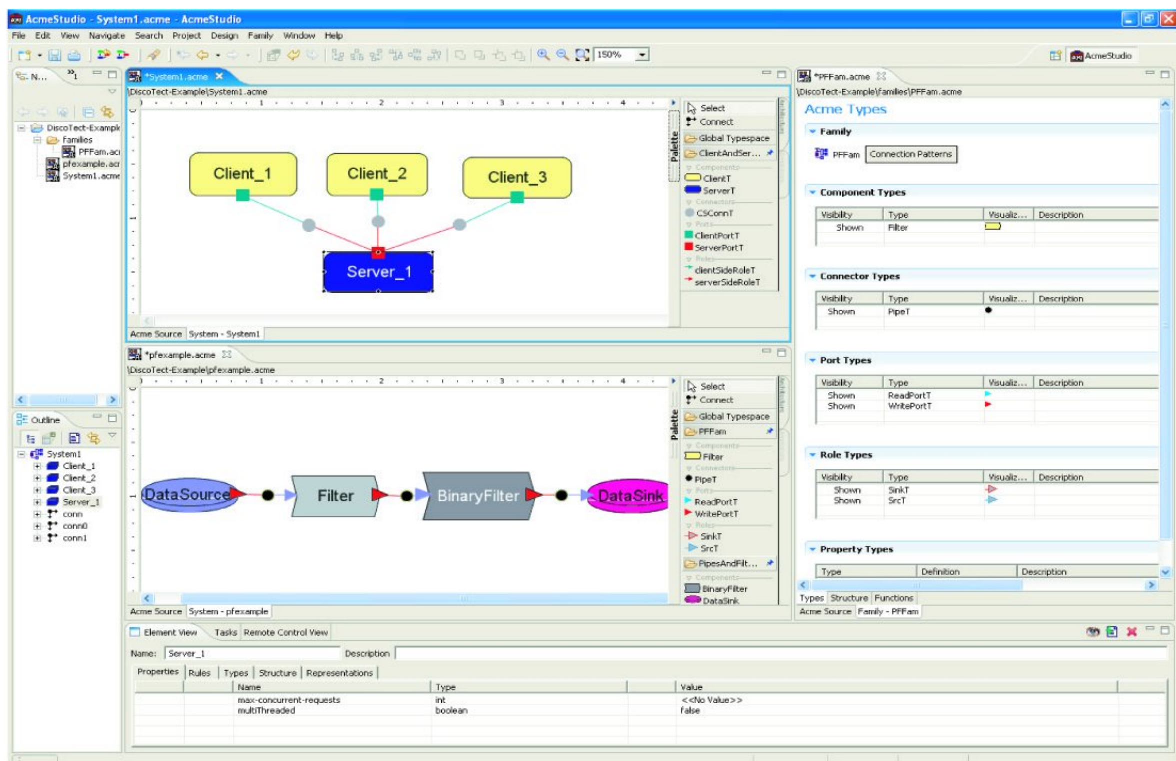


Figure 22. Using AcmeStudio for architectural design.

A very important and useful features of AcmeStudio, already briefly pointed out before, is the fact that, through what it is called Remote Control View, it can start an RMI Remote Control Server or connect to a JMS server to listen to a particular JMS topic. In this way, it can receive architectural events through the network and automatically create a model from them. So models are not just created manually by the user, but their creation can be triggered and done by other application, through a network. This is a key feature in my case, since thanks to this, AcmeStudio can receive the meaningful architectural events filtered by DiscoTect, create a model and show it to the user.

4.1.3 Transformation

The parts that follow represent the heart of my contribution, since they were created from scratch to study the feasibility of automatic or semi-automatic application transformation, given as input the original system and the information extracted from the dynamic analysis of the steps before.
At first I had to decide what kind of transformations to offer in this first version of my prototype, in fact, from this decision depends the transformation method to be applied to the original code.

As a starting point I decided to study only one particular transformation, namely the transformation of Java RMI applications to Web services. This means that the architecture analysis part will monitor the execution of the system, to see if it follows an RMI client-server paradigm and it will collect the necessary information for the eventual trnaformation. I will take a much closer and precise look in the next two chapters.

I opted for the transformation to WebServices because it is one of the hottest “next big thing” computer science, in particular in software modernization, as it can be seen in the number of studies about it (4; 6; 25; 5; 10). There are three main key factors for its success and its usefulness in software modernization.

First is the high decoupling between the service interface and the actual implementation, which allow any type of application to invoke and use a service without caring about the programming language used, the platform on which it runs and so on. The only thing that is really needed is the WSDL definition of the service, that contains all the necessary information to use that service. This is extremely useful in the case of old/legacy systems, that often run on obsolete and ad hoc hardware and using old programming languages that only a few programmers know well.

Second is the network-based nature of web services, which allows heterogenous acces from any kind of application and tool that is connected to the same network of the service and that it dialogs with the service through the use of SOAP messaging protocol, that hides differences in the implementation. In this way, systems that were not accessible by any other applications or that were accessible only through ad-hoc connection and protocols, now are accessible through the network with the use a standardized and known protocol.

Third, is the ease of use and implementation, if using Java. This key factor emerged particularly in the last couple of years, thanks to Java 1.5 and moreover the newest Java 1.6. In fact, a big issue about web services has been the overall development complexity. These difficulties made web service implementation hardly doable without the use of some specific tool, as for example

the Eclipse Web Tools Platform (WTP) (101). But now, with the use of just a few annotation to be added to the code and the use of pre-made Ant (102) scripts to generate the required files and deploy the created service, it is really a matter of minutes to have a first version of a web service up and running. The testing part gets even easier if using the newest Sun application server (103), as we will se in the next section.

To know all this things I had, at first, to study the latter web service implementation for the Java language, JAX-WS (104), which will be the topic of the next section.

4.1.3.1 Java Web service implementation

Sun Microsystems' JAX-WS 2.0 (Java API for XML Web Services) derives from JAX-RPC (Java API for XML-Based RPC) (105), which was at first developed by Sun around 2001. Before than, there was already the SOAP, which described what the messages looked like and WSDL to describe services, but there was nothing that told programmers how to write Web services in Java. So JAX-WS and before that, JAX-RPC, are Sun's answer to the question of how to develop web services easily in Java.

Since JavaEE 5 and Java 6, JAX-RPC has been replaced by JAX-WS, which is basically a better and much simpler version of its successor and it was created with the primary goal to align with industry direction, the was not merely doing RPC Web services, but they were also doing message-oriented Web services. That is why "RPC" was removed from the acronym and replaced with "WS".

JAX-WS 2.0 provides a library of annotations and a toolkit of Ant tasks and command-line utilities that hide the complexity of the underlying XML message protocol. This new release

supports different protocols such as SOAP 1.1, SOAP 1.2 (106) and REST (107), and uses JAXB 2.0, also new to Java EE 5, for XML data binding.

When writing a JAX-WS 2.0 web service, the developer uses annotations to define methods either in an interface or directly in an implementation class (the interface can be automatically generated). This powerful library of annotations eases web service development tasks a lot: with them, plain old Java classes can be turned into web services in matter of minutes. The metadata represented by this annotation can then be processed by specialized tools to generate much of the web service plumbing and dirty details automatically, leaving the uset to concentrate on the business code. There are two possible approaches to create a web service using the JAX-WS toolkit. The user can either start from the WSDL file and generate automatically interface and skeleton implementation classes, or she/he can start from an implementation class and generate automatically both a WSDL file and a Java interface. I will describe the latter approach, since it is the one that I will use. In fact, I have the old classes that need to be modified and then transformed into web services, so it is more reasonable to use the second option since I already have an implementation class.

Now I will quickly show how to write a web service implementation class.

As already said, in JAX-WS 2.0, a web service implementation simply takes the form of an ordinary Java class with some special annotations. The required annotations are two: *@WebService* and *@WebMethod* as showed in the following example.

```
@WebService(serviceName="ExampleService", name="Example")
public class ExampleClassImpl {
```

```
  @WebMethod()
  public String getMessage(String name) {
    return "Hello user " + name;
  }
}
```

The *@WebService* annotation declares the class as a web service. The name property in the @WebService annotation lets you define the web service name (the wdsl:portType attribute in WDSL). The serviceName property lets you define the WDSL service name. These properties are optional; if undefined, sensible default values will be derived from the class name. In my case, as we will see, I left them undefined, since my purpose was to transform an application into a web service and not to fine tune the service.

The @WebMethod annotation is used to indicate which methods are to be exposed by this web service. Also this annotation takes optional parameters that can be used to customize the WDSL values, but the default values are often quite sufficient, so I will not cover them. Note again that, apart from the annotations, this class is a perfectly normal Java class.

The annotated class has then to be compiled with the web service generator or Wsgen (this class, com.sun.tools.ws.ant.WsGen, is provided in the jaxws-tools.jar file).
This task will generate some low-level classes that will handle the SOAP encapsulation for me. In the case of the previous example, the Wsgen task generates two heavily annotated classes, which encapsulate the request and the response to the getMessage() method, respectively:

- GetMessage that encapsulates the request coming from the client invoking the getMessage() method.

- GetMessageResponse that encapsulates the response sent from the web service back to the client.

In general Wsgen generates for each method annotated as WebMethod those two classes.
The wsgen tool will then also generate the WDSL file and XSD schema for the web service, which are needed to publish my web service description to the outside world. Although Wsgen can be used as a command-line tool, in general it should be integrated it into the build process (represented by the XML file *build.xml*). To do this using Ant, you need to declare the Wsgen task, as follows:

```
<taskdef name="wsgen" classname="com.sun.tools.ws.ant.WsGen">
  <classpath refid="jaxws.classpath"/>
</taskdef>
```

Now you need to run Wsgen on all of your server classes. The generic Ant task is shown here:

```
<target name="compile-service" depends="init" description=
      "Compiles the server-side source code">
  <echo message="Compiling the server-side source code...."/>
  <wsgen debug="${debug}" verbose="${verbose}" keep="${keep}" sei="${sei}"
      destdir="${build}" sourcedestdir="${build}">
    <classpath>
      <path refid="jaxws.classpath"/>
      <pathelement location="${build}"/>
    </classpath>
  </wsgen>
</target>
```

Note that many attributes are assigned, instead of values, names preceded by the character $ (also in the following XML fragments), that means that they are assigned values whose value

is specified in an external property file, usually it is called build.properties. This is specified in the build file by a single line:

```
<property file="build.properties"/>
```

We will see the whole ant file for the web service created by my tool in the next chapter, in which I focus more on the actual implementation of my prototype.

The only things that still need to be done to make the service run are: create two XML files, namely *web.xml* and *sun-jaxws.xml*, the .war file and the deploy it on the application server of choice (which, for me was the Sun Application Server PE 9).
The *web.xml* file is used to configure the Sun web service servlet, while the *sun-jaxws.xml* defines a set of "endpoints". Each endpoint represents a particular published web service and includes information about the implementation class, URL pattern, and (optionally) other technical details such as the WSDL service and port. This files need to be places in the WEB-INF directory. For my example the two files would be, for the *web.xml* file:

```
<web-app version="2.5" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <display-name>HelloService</display-name>
  <listener>
    <listener-class>
      com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
  </listener>
  <servlet>
    <display-name>HelloService</display-name>
    <servlet-name>HelloService</servlet-name>
    <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
```

```
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloService</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>
</web-app>
```

and for the *sun-jaxws.xml* file:

```
<endpoints version="2.0">
  <endpoint name="HelloService" implementation=
       "helloservice.endpoint.ExampleClassImpl" url-pattern="/hello"/>
</endpoints>
```

Then you need to create the .war file to be deployed on the application server. Also for this task is preferable to use ant. Usually, a generic task to do this can be:

```
<target name="create-war" description="Packages the WAR file" depends="prepare-assemble">
  <echo message="Creating the WAR...."/>
  <delete file="${assemble.war}/${war.file}"/>
  <war warfile="${assemble.war}/${war.file}" webxml="${conf.dir}/web.xml">
    <webinf dir="${basedir}/${conf.dir}" includes="*.xml" excludes="web.xml"/>
    <zipfileset dir="${basedir}/etc" includes="*.wsdl, *.xsd" prefix="/WEB-INF/wsdl"/>
    <classes dir="${build.classes.home}" excludes="**/*.java"/>
  </war>
</target>
```

As it can be seen, from the *webinf* element in the previous XML fragment, the creation of the .war file requires the .xml files specified before.

At last, but not at least, the service needs to be deployed. This can be easily done in several way. First the .war file can be manually moved to the specific folder of the application server that contains the deployed web services (in the case of JBoss, the path of this folder is ($< ServerInstallationdirectory > /server/default/deploy$, in the case of the Sun Application Server is $< ServerInstallationdirectory > /domains/domain1/autodeploy$). Second, as usual, it can be done through an ant task. This ant task can simply copy the .war file to the server's deploy folder, as in the manual solution, or, in case the Sun Application Server is being used, it can deploy the service through the use of an adminstrator command, as in the following fragment.

```
<target name="deploy">
<antcall target="admin_command_common">
<param name="admin.command" value="deploy ${assemble.war}/${war.file}"/>
</antcall>
</target>


<target name="admin_command_common">
    <echo message="Doing admin task ${admin.command}"/>
    <sun-appserv-admin command="${admin.command}" user="${admin.user}"
        passwordfile= "${admin.password.file}" host="${admin.host}"
        port="${admin.port}" asinstalldir="${javaee.home}"/>
</target>
```

The whole process can look at first quite complex and unclear; that is what I thought at first and many of the tutorials I read, which made the user create all the files from scratch, did

not help at all to clarify the whole process.

What I found extremely helpful was the Java Web Services Tutorial (108) and its examples. In fact, instead of starting from scratch, after having studied a basic example of the tutorial and how the web service compilation and deployment worked, I took all the necessary files (*build.xml, web.xml, build.properties* and *sun-jaxws.xml*) of the example, modified to my need and tested with a simple web service created by me, until it all worked.

In this way, I now have a "universal" template version of those previously cited XML files that can be used for any newly implemented web service, the only thing that has to be modified are the project specific information in *web.xml, build.properties* and *sun-jaxws.xml* files, but this can be easily done, also thanks to the insertion in those files of comments to help the user. In this way, if a user is provided with some quickly and easily modifiable version of those files, the creation and deployment of a web service can really be matter of minutes.

In my prototype, all the web service creation and deployment as been totally automized, in order to aid the user in what I think is the only part of web service creation that can actually create some problems, she/he will just have to invoke the ant build command and, subsequently, a deploy command. We will see this in the next chapter.

On the client side, the web service client simply creates a proxy object and then invokes methods on this proxy. Neither the server nor the client needs to generate or parse SOAP (or REST) messages; the JAX-WS 2.0 API takes care of these tedious and error-prone low-level tasks.

I will not cover the client part, since my purpose is initially to modernize an RMI server into

a web service. In fact, since the client is not developed and used by the same creator of the server, it will be the client creator responsibility to transform it.

4.1.3.2 Transformation method

Having decided a first transformation, namely the source system and the target system paradigms, allowed me to work on how to actually do this transformation in the most automized way.

I had two main possible solution: either create a custom compiler with JavaCC libraries or create some ad-hoc methods.

The first one is certainly an extremely powerful and versatile solution, but I thought it was a bit too much complicated, in particular with respect to my specific needs. This decision has been based on the main consideration that any RMI server class, if coded using standard coding paradigms, does always the same four things.

First it extends *UnicastRemoteObject*. Second it implements an interface that extends *Remote* and defines all the methods that are to be exported as remote methods and that will be implemented by the server class; all these methods have to declare the clause *throws RemoteException*. Third, the server class has to import: *java.rmi.Naming, java.rmi.RMISecurityManager, java.rmi.RemoteException, java.rmi.server.UnicastRemoteObject*. Fourth, the *main* method instantiate a new instance of the server and then calls the *Naming.rebind method* which binds that particular instance of the server to a network address, registers it into the rmiregistry and then just waits for incoming connections from eventual clients.

In addition to this, the code that has to be added to transform it is minimal. In fact, only the

@WebService() annotation before the server class declaration and the *@WebService()* annotations before the remote methods need to be added.

So based on this consideration, I opted for the use of some custom methods to clean the server class and its relative interfaces and to add the required annotation. Obviously, even if a class follows a known paradigm or structure, there can always be some slight modifications, so a totally automized transformation can never happen; the user has always to control the outcome of it and made the required adjustments. What an automatic tool can do, is to annotate somehow the parts of the code that it does not understand or that don't follow the specified paradigm, to guide the user to their manual modification. I will talk more about this in the next chapter when will take a deeper look in the transformation process.

4.1.4 Web service creation

I previously explained how to write Java web services in Section 4.1.3.1. Here I will explain how I addressed the creation part in my project.
Once understood how to make web services work, this part was of quick solution. In fact, once I have the server annotated classes, I just need to compile them, create the .war file and deploy it on the Application Server of choice. I already explained all that process in Section 4.1.3.1. The only thing that was still needed was the integration of that process in my project and the application of it to my modernized application. This is simply done by creating in the directory in which the modernized application resides the XML and the build.properties files that I explained before. In my solution, those files are automatically tuned to the specific application, so that the user just needs to call the ant *build* and the ant *deploy* tasks to have the

service up and running without messing around with the usually obscure XML configuration files. Obviously, if the user needs or want to modify them by hand to fine tune them, it is still possible.

As it can also be noted in Section 4.1.3.1, the ant deploy task was written to deploy the service to the Sun Application Server. During the implementation I also used JBoss, but I then decided to use Sun's as the default one for two main reasons.

First, since I already use other Java technologies (as the language itself, Wsgen, Ant, etc.) I opted for homogeneity, that allow also a better integration. In fact, to deploy the service on the Sun Application Server, it is possible to do it by executing an administrator command through Ant.

Second, the Sun Application Server allows to easily test the deployed web service without the need to implement any client throught the administrator console (while also offering the possibility to see its wsdl, sun-web.xml, web.xml files). Figure Figure 23 shows the page with the information about a deployed service on the Sun Application Server, while Figure Figure 24 show the tester for that same service.

On the other hand, JBoss doesn't offer that, so to test a deployed web service, some other application are needed, as for example soapUI, which still has not the ease of use offered by the Sun Application Server testing facility.

However since JBoss is widely used (also DiscoTect uses it, in particular its JMS service) and to make my solution the most versatile possible, I decided to allow the use of any application server of choice. So I added also a universal deploy task and its related undeploy task, called univ_deploy and univ_undeploy:

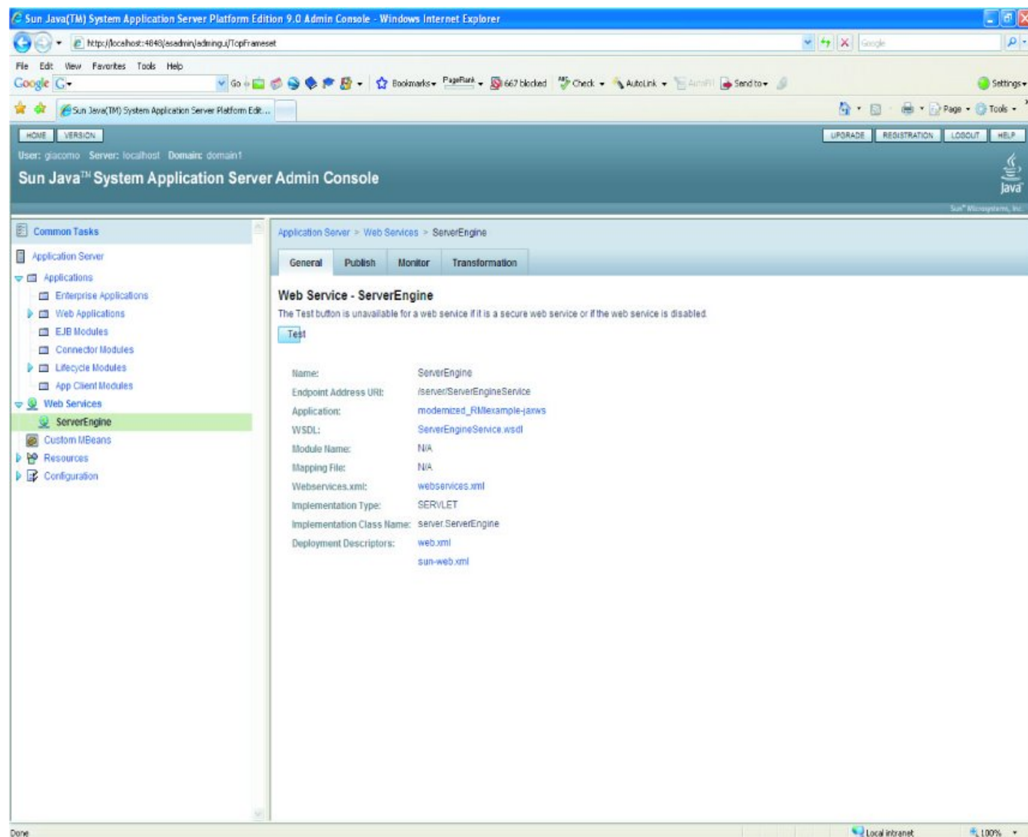


Figure 23. The information about ServerEngine service.

```
<target name="univ_deploy" description="Universal Deploy">
  <copy file="${assemble.war}/${war.file}" todir="${deploy.dir}">
  </copy>
</target>

<target name="univ_undeploy" description="Universal Undeploy">
  <delete file="${deploy.dir}/${war.file}">
  </delete>
```

```
</target>
```

Note that the deploy.dir refers to the deploy directory of the specific application server. So what those two task do is simply copying/deleting the .war file to/from the application server deploy directory. The application server regularly scans that directory to look for new web services in it and so it will notice the new service automatically. The same is done when a .war is deleted from that the directory: the application server notice its deletion and undeploys the service.

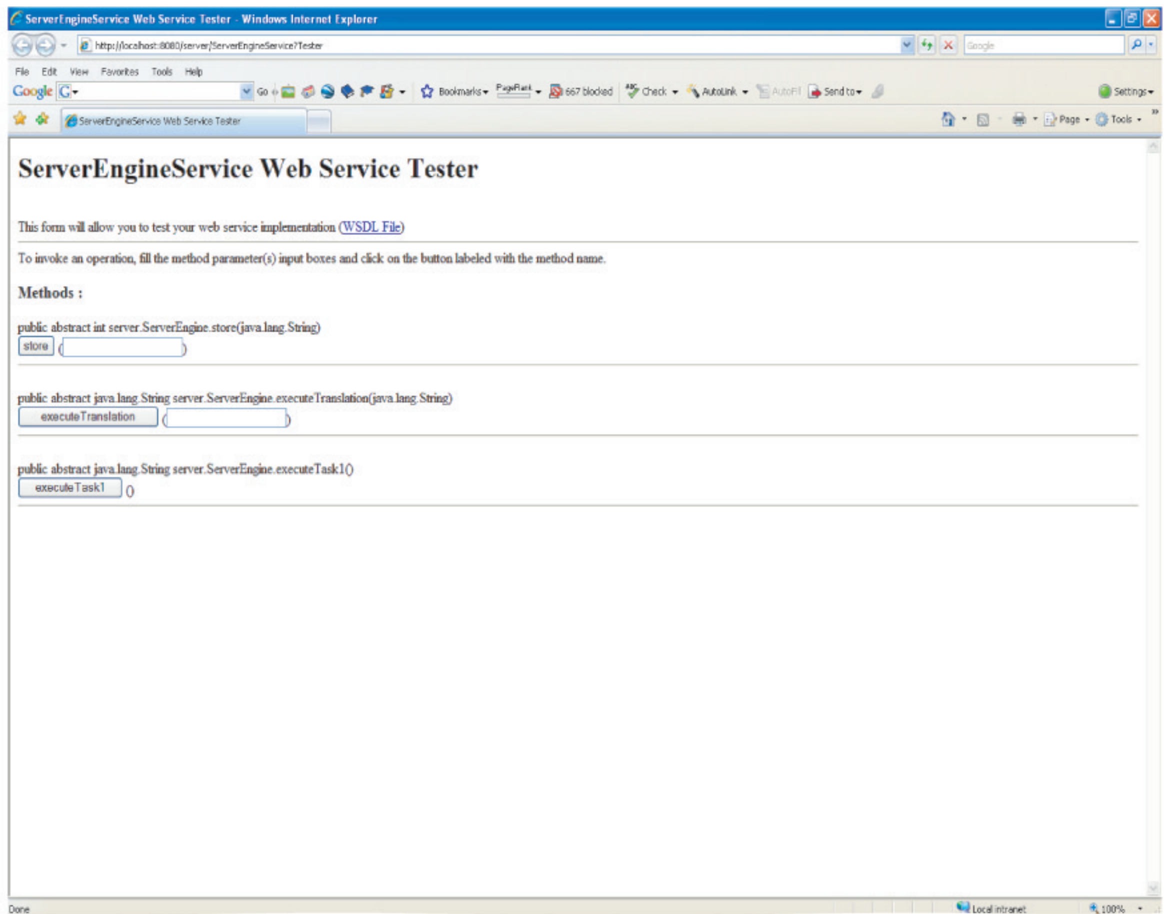


Figure 24. The tester for the ServerEngine service.

CHAPTER 5

DISCOTECT

After having described the several parts composing my work at a high level in the previous chapter, I will now get deeper into more low level details, showing and describing the vital parts of the actual implementation. The order I will follow in this explanation reflects the order of usage of the single parts in the tool during an hypothetical modernization project.
So I will start from the architecture extraction and I will end with the deployment as a Web service of the transformed application.

In this following chapter I will explain all the things I had to do to gain the necessary knowledge on the several part of DiscoTect and the steps done to make it recognize the needed architecture structure, namely an RMI client-server application and, before that, to first test my knowledge, a socket based client-server application.
On the other hand, in the next chapter I will explain the implementation of the prototype tool in depth.

5.1 Probe aspects

As said and explained before in Section 4.1.1.1 and 4.1.1.2, probe aspects are used to extract meaningful information from the execution of the target system.
To start writing probe aspects, I had to first study and learn aspect oriented programming and AspectJ. In my case, I had already done it (mainly with the help of the book by Colyer et al

(100)) while doing my preliminary studies on the different techniques to do system's execution tracing, reported in Chapter 3.

So when I started using DiscoTect, I already had enough knowledge to write aspects. However, to learn how to write functioning aspects for DiscoTect I had to do reverse engineering from the only available example (about a Pipe and Filter system) and proceed with a "trial and error" approach. In fact the documentation for this part is scarce if not unexistent, both in the published paper (3) and in the actual implemented Java classes, so I had to go on to read and interpret the classes of interest, in this case mainly JMSXMLEventEmitter, which is fortunately clean and well written.

5.1.1 Implementation

Oviously, the first aspects I created were basic ones to test my knowledge as I was building it while going through the book I was using (100) and getting used to the new language. Then I moved on the implementation of probe aspects.

The first thing to be done when implementing a probe aspect, before even starting the implementation, is the analysis of the architectural paradigm that we want to try to extract from the system (if it follows that particular paradigm, obviously). This can usually be done by analyzing existing existing systems that are known to follow that particular paradigm, or for some typical paradigms even a programmer's guide might do the work. In this way we can understand what the typical methods for a particular paradigm are and thus write specific aspects that monitors the execution of a system waiting for the call or execution of those methods.

The first probe aspect I created was for the tracing of a basic socket based client-server Java application.

I first scanned the Internet to find some simple example implementations, to see what the key methods for those kind of applications were (the knowledge on sockets was a bit rusty since they were studied for the C language at least 3 years ago). Note that there is no need to have a complex application to extract the key methods of a paradigm, since they are, by definition in every application implementing that particular paradigm, no matter how complex it is.

In the case of the socket based client-server, the key methods I found are:

- *ServerSocket.new()* that indicates that a server socket have been created.
- *Socket.new()* that indicates that a client socket has been created.
- *ServerSocket.accept()* that indicates that a connection between the server and the client that executed the *Socket.new()* constructor has been established. In fact, if a call to *Socket.new()* is followed in the system execution by a call to *Socket.accept()*, it means that the client that made that call is the same one involved in the newly created connection through the *accept* method.

So with this information and with the guidelines exposed in Section 4.1.1.2 on how to write well formed advices for DiscoTect, I wrote the following aspect:

```
import edu.cmu.cs.acme.discotect.probe.*;

import java.net.*;

import java.io.*;
```

```
public aspect MyAspect extends JMSXMLEventEmitter{

  //Restrict classes to those we are interested in
  pointcut projectClasses ():
    (within (example.*) || within (java.io.*) || within (java.net.*));

  //Track the call to main
  pointcut mainMethod ():
    (execution (* example.Start.main(..)) || call (* example.Start.main(..)));

  //Track the call to socket from clients
  pointcut clientAcceptMethod ():
    call (* *Socket.accept(..));

  pointcut newClientSocket ():
    call (Socket.new(..));

  //Track constructors
pointcut serverSocketConstructor ():
call (ServerSocket.new(..));

  //This advises all calls to constructors in the project
  after() returning (Object result):
    projectClasses() && serverSocketConstructor() {
      emitInit (thisJoinPoint, result);
    }
```

```
  after() returning (Object result):

    projectClasses() && newClientSocket(){

      emitInit (thisJoinPoint, result);

  }


  //Advise all calls to accept

  before(Object caller):

    this (caller) && clientAcceptMethod(){

      emitCallEvent (thisJoinPoint, caller, "call");

  }


  //Advise the call tot the main method

  before():

    mainMethod() {

      emitCallEvent(thisJoinPoint,null, "call");

  }

}
```

Note that the advised methods are the ones I talked about before and their syntax is the one I explained in the previous chapter.

After the creation of this probe aspect, I proceeded in implementing the DiscoSTEP specification, which will be showed in Section 5.2.1. Then I tested them with a sample application to see if everything worked and if the actual architecture was really visualized in AcmeStudio; Section 5.3.1 treat this part, showing also a screenshot of the actual graphical outcome. Then I proceeded in the creation of the probe aspect for an RMI application, which is vital for my prototype. As before, I first studied the target application paradigm to see what the key methods

are. This case proved to be more complex, in fact the RMI specific methods, more precisely, *Naming.lookup()* and *Naming.rebind()* are not enough for a precise reconstruction as now we will see.
Naming.rebind() indicates that an RMI server (whose name is in the parameters of that method) has been created and is waiting for incoming connections.
Naming.lookup() indicates that an RMI client has been created (the caller of that method) and can call the remote methods of the remote server whose name was the argument of the lookup method.
The problem is that in this way we cannot know which are the actual remote methods called by the clients. In fact, these methods, by definition, do not have a specific name or signature, since their name and everything else are created by the creator of the application during the development.
To solve this problem, I added to the aspect a pointcut for the call to a generic method:

```
pointcut genericMethod():
  call (* *(..));
```

In this way any call to any method is monitored. I then created an advice that, before that pointcut, extracts the interfaces extended by the target object, which in the case of remote methods, is the interface extending *Remote* which is then implemented by the RMI server (This is how RMI works, as explained in many online examples and also in every Java book(for example, the one by Deitel and Deitel (109))). So once I have extracted the list of those interfaces I can see if it contains *Remote* and if it does, the method I am examining is a remote method and then is meaningful for the architecture reconstruction so it needs to be emitted on the JMS bus.
To make it more clear here is the advice:

```
before(Object tar, Object caller):
```

```
genericMethod() && target(tar) && clientsClasses() && this(caller){
  Class cl[] = tar.getClass().getInterfaces();
  int len = cl.length;
  for(int i=0; i<len; i++){
    if(cl[i].toString().contains("Remote")){
      emitCallEvent(thisJoinPoint,caller, "call");
    }
  }
}
```

To put it in a nutshell, in this way, I can monitor the call to remote methods by clients, which otherwise would be impossible. I did this trick also for the server side, in this way I can find out what the remote methods implemented by the server and accessed by some client are. The methods that are not accessed by any client will obviously be ignored, as I am doing dynamic analysis and thus I want to find out what the methods actually used are.

With this solution, I could not discriminate between the first call to a remote method (server-side) and the successive calls, which is important in order to reconstruct the runtime architecture of an RMI application. In fact, the first call to a remote method has two meanings: it indicates the existence of a particular accessible method on the server and that a client accessed it. On the other hand, the successive calls to that method only indicate that other clients accessed it. To do so, I added a data structure, namely an ArrayList, in the aspect to contain the names of the remote methods called so far. In this way, when a remote method is called, I check if it has already been called once, if not, I add its name to the list of called methods and emit an *Init* event (with emitInit) to indicate that the server has a particular method. Then, indipendently to the fact that the method has already been called or not, a *Call* event (with emitCallEvent) to indicate that a client has accessed that method.

This is particularly useful for the visualization part, as we will see in Section 5.3.2, since in this way I can visualize clients, servers and the logical connection between them, given by the remote methods respectively called and exposed. Here is the complete aspect:

```
import edu.cmu.cs.acme.discotect.probe.*;
import java.rmi.*;
import java.util.ArrayList;

public aspect RMIAspect extends JMSXMLEventEmitter{

  private BufferedWriter print;
  private ArrayList serversMethods;
  private static boolean created = false;
  //Restrict classes to those we are interested in
  pointcut projectClasses ():
    (within (clients.*) || within (server.*) || within (java.rmi.*));

  pointcut serverClasses ():
    within (server.*);

  pointcut clientsClasses ():
    within (clients.*);

  //Tracks the call to main
  pointcut mainMethod ():
    (execution (* *.main(..)) || call (* *.main(..)));
```

```
  //Tracks the call to generic constructors

  pointcut genericConstructor ():

    call (*.new(..));


  //Tracks the call to exportObject

  pointcut rebindMethod ():

    call (* Naming.rebind(..));


//Tracks the call to lookup

pointcut lookupMethod ():

    call (* Naming.lookup(..));


  //Tracks the call to a generic method

  pointcut genericMethod():

    call (* *(..));


  //Tracks the execution of a generic method

  pointcut genericMethod2():

    execution (* *(..));


  /**

   * Advise calls to any method extending Remote made from the clients side.

   */

  before(Object tar, Object caller):

    genericMethod() && target(tar) && clientsClasses() && this(caller){
```

```
    Class cl[] = tar.getClass().getInterfaces();
    int len = cl.length;
    for(int i=0; i<len; i++){
      if(cl[i].toString().contains("Remote")){
        emitCallEvent(thisJoinPoint,caller, "call");
      }
    }
  }

/**
 * Advise calls to any method extending Remote, from the server side.
 */
after(Object tar) returning(Object result):
  genericMethod2() && target(tar) && serverClasses(){
    Class cl[] = tar.getClass().getInterfaces();
    int len = cl.length;
    int i;
    for(i=0; i<len; i++){
      Class superCl[]=cl[i].getInterfaces();
      int len2 = superCl.length;
      int j;
      for(j=0; j<len2; j++){
        if(superCl[j].toString().contains("Remote")){
          String meth = thisJoinPoint.getSignature().getName();
            if(!serversMethods.contains(meth)){
              serversMethods.add(meth);
```

```
              emitInit (thisJoinPoint, result);
            }
          emitCallEvent(thisJoinPoint, null, "call");
        }
      }
    }
   }

  //Advise call to the Naming.lookup method
before(Object caller): lookupMethod() && this(caller){
    if(!created){
      serversMethods = new ArrayList();
      created=true;
    }
    emitCallEvent(thisJoinPoint,caller, "call");
  }

  /**
   * Advise call to the Naming.rebind method.
   */
  before(Object caller): rebindMethod() && this(caller){
    emitCallEvent(thisJoinPoint,caller, "call");
  }

  // Advise the call to the main method from the server classes
  before(): mainMethod() && serverClasses() {
```

```
    if(!created){
      serversMethods = new ArrayList();
      created=true;
    }
    emitCallEvent(thisJoinPoint,null, "call");
  }
}
```

5.1.2 Problems encountered

On the implementative side, the main problem I encountered in this part of the work is the one I already mentioned before, about the lacking of documentation to aid in understanding and developing probe aspects. This problem, together with the initial understanding of AspectJ made the developing at first extremely slow and error prone, but once mastered the few necessary tricks, the implementation became straight forward.

On a more logical side, another problem might arise when I want the probe aspect to be extremely generic in order to be applied to the biggest number of applications implementing a determined paradigm. In fact, in this case, a great deal of attention and a lot of preliminary study has to be done in order to extract the most general sequence of events that characterize a particular architectural paradigm.

Since, the same architectural paradigm can be hypothetically implemented in a number of ways, it is impossible to write a single probe aspect able to recognize all of them. Thus the programmer should focus on writing aspects that recognize applications following standard coding structures

of the target architecture; in this way the aspect is kept the most generic possible, while still being possible to write.

5.2 DiscoSTEP

After the implementation of the probe aspects, I went on to implement the DiscoSTEP specifications, that are used to recognize interleaved patterns of the meaningful runtime events filtered and formatted by the aspects as particular architectural paradigms. In this following section I will explain all the things I had to do to gain the necessary knowledge on the DiscoSTEP language and the steps done to write specifications able to recognize the needed architecture structure, namely an RMI client-server application and, before that, to first test my knowledge, a socket based client-server application.

Note that I actually created the probe aspect for the socket example, then I wrote the corresponding DiscoSTEP specification and only after tested that what I wrote was correct and working, I went back to implementating the probe aspect for the RMI client-server application and so on.

5.2.1 Implementation of DiscoSTEP specifications

Before writing the required specification to recognize an RMI client-server system, I first had to master the DiscoSTEP language, which proved to be quite a challenging task, as we will see in the following section.

Thus I started by writing a client-server system based on sockets in Java and from then I started writing my first DiscoSTEP specification by taking the only available example, a specification for pipe and filter systems, almost as a blueprint.

First of all I had to import the necessary XML schemas that define the several events used in the specification, which are:

sys_events.xsd It specifies the definitions of the two events, call and init, that can be emitted by a probe aspect, through the JMSXMLEventEmitter class, during the runtime monitoring of an application (the definition of this schema has already been shown in the previous chapter).

DASADA-Arch-Mutation-Schema.xsd It specifies the definitions of the architectural events specific to AcmeStudio. In this way the rules in the specification can output meaningful architectural events as XML strings that can interpreted and then visualized by AcmeStudio. Note that in the example of the previous chapter I used an hypothetical custom_architectural_events.xsd, which does not actually exists.

holder.xsd It specifies two useful data structure that can be used to pass information from one rule to another. Here is the actual file:

```
<schema>
  <element name="holder">
    <complexType>
      <attribute name="acmeId" type="string" use="required"/>
      <attribute name="implId" type="string" use="required"/>
      <attribute name="parentId" type="string"/>
    </complexType>
  </element>
  <element name="connection_holder">
```

```
    <complexType>
      <attribute name="reader" type="string" use="required"/>
      <attribute name="writer" type="string" use="required"/>
    </complexType>
  </element>
</schema>
```

The first element is used to contain information about a component that has been created: implId usually contains its identifier generated by the probe aspects at runtime while acmeId usually contains its identifier used in the AcmeStudio output file and parentId is the eventual identifier of its parent (which can be another component or the whole Acme system).

The second name is used to contain information about a connection between two components that can identified by their name and their role in the connection (either reader or writer).

There is also a fourth schema imported in the example specification, *create_ system.xsd*, which is unused, so I decided to leave it out from my specifications.

I then proceeded in the definition of input and output events. As input events I declared holder, since I will need to pass information from one rule to another about the newly created Acme components (as we will see later when discussing about the actual rules of the specification) and then I declared, obviously, init and call since they are the events that come from the system runtime monitoring. As output events I declared dasada-architecture-mutation, which is used to represent the architectural events formatted for AcmeStudio, and holder. In fact, since it is

used to pass information between rules, it needs to be declared both as input and as output. Then I went on doing the actual implementation of the specification by writing the rules.

The first rule is the one that creates a new AcmeStudio system, so I then can add the successive components I find. This rule needs to be enabled when the *main* method of the application is called and must outputs a "new AcmeStudio client-server system" architectural event (we saw in the previous chapter that it already exists a client and server family) and a holder, containing the AcmeStudio id of this system (in this case "NewSystem", arbitrarly chosen by me), so that all the following components can be added to it. Note that this id has to coincide with an already existing .acme file, otherwise with no destination file to save the information to, all the architectural events will be lost.
Here is the actual rule exposing what I just said:

```
rule CreateSystem {
 input {
  call $c;
 }
 output {
  holder $sys_holder;
  dasada-architecture-mutation $sys;
 }
 trigger {? contains ($c/@method_name, ".main") ?}
 action {?
  let $sys_holder := <holder implId="{$c/@callee}" acmeId="NewSystem"/>
  let $sys :=
   <dasada-architecture-mutation>
    <created>
```

```
     <newSystem id="{$sys_holder/@acmeId}">
      <type type="simple" href="ClientAndServerFam"/>
     </newSystem>
    </created>
   </dasada-architecture-mutation>
 ?}
}
```

The second rule is the one that creates a socket server. This rule needs to be enabled when a new *ServerSocket* is instantiated, so when an init event with the attribute *constructor* containing the string “ServerSocket” is received. In order to be able to associate the newly created server to an existing Acme system, this rule has to receive in input also the holder containing information about that system. Then, this rule has to output a “new Acme Server component” architectural event and a holder containing information about the newly created server, more precisely:

- implId contains the identifier generated by the probe aspects at runtime, contained in the attribute *instance_id* of the init event.
- acmeId contains that same identifier preceeded by the string “Server_” and followed by the string “-port_”, which itself is followed by the port number on which the server will be waiting for incoming connections. It will be used as the identifier for the component in the Acme system.

Here is the actual rule exposing what I just said:

```
rule CreateServer {
 input {
```

```
  init $f;
  holder $sys;
 }
 output {
  holder $server_holder;
  dasada-architecture-mutation $server;
 }
 trigger {? contains ($f/@constructor, "ServerSocket") ?}
 action {?
  let $acmeId := concat ("Server_", $f/@instance_id, "-port_", $f/arg[1]/@value)
  let $server_holder := <holder implId="{$f/@instance_id}" acmeId="{$acmeId}"/>
  let $server :=
   <dasada-architecture-mutation>
    <created>
     <newComponent id="{$server_holder/@acmeId}">
      <description/>
       <type type="simple" href="ServerT"/>
       <instanceOf type="simple" href="ServerT"/>
     </newComponent>
     <context type="simple" href="{$sys/@acmeId}"/>
    </created>
  </dasada-architecture-mutation>
 ?}
}
```

The third rule needed is the one that creates the socket client components and connects it to the server. This rule needs to be enabled when a new *Socket* is instantiated on client side and in the mean time the *ServerSocket.accept* method is called on the newly created server whose identifier has been passed as input.

In other words, the rule is activated when an init event with the attribute *constructor* containing the string "java.net.Socket" (not only the "Socket" string otherwise this might also be activated by the creation of a new socket server, whose constructor also contains that substring) is received, along with the call event with the attribute *method_name* containing the string "ServerSocket.accept" and with the attribute *callee* containing the same string contained in the *impl_Id* attribute of the server holder passed as input. In this way, by using a condition on those two events , a client will be created only if the connection is actually accepted by the server, so it will not show connections that actually don't exist since I want the architecture to be the most precise possible.

Then this rule outputs the following "new Acme components" architectural events:

- Client.
- ClientPort to be added to the newly created client.
- ServerPort to be added to the existing server, represented by the holder passed as input.
- ClientServer Connector that will connect the client with the server through the newly created ports.

To actually connect the server with the client through their ports, I need to output two other architectural events that triggere the attachment of the client and server sides of ClientServer

Connector to the newly created ClientPort and ServerPort. In order to be able to associate the newly created component to an existing Acme system, this rule has to receive in input also the holder containing information about that system.
Here is the actual rule exposing what I just said:

```
rule CreateClient {
 input {
  call $d;
  init $c;
  holder $sys;
  holder $server_holder;
 }
 output {
  dasada-architecture-mutation $client;
  dasada-architecture-mutation $clientport;
  dasada-architecture-mutation $serverport;
  dasada-architecture-mutation $socketchannel;
  dasada-architecture-mutation $client_side;
  dasada-architecture-mutation $server_side;
 }
 trigger {? contains ($d/@method_name, "ServerSocket.accept")
   and $d/@callee = $server_holder/@implId
   and contains ($c/@constructor, "java.net.Socket") ?}
 action {?
  let $acmeId := concat ("Client_", $c/@instance_id)
  let $client :=
   <dasada-architecture-mutation>
```

```
  <created>
   <newComponent id= "{$acmeId}">
    <description/>
     <type type="simple" href="ClientT"/>
     <instanceOf type="simple" href="ClientT"/>
   </newComponent>
   <context type="simple" href="{$sys/@acmeId}"/>
  </created>
 </dasada-architecture-mutation>
let $client_holder := <holder implId="{$c/@instance_id}" acmeId="{$acmeId}"/>
let $toServer := concat ("To ", $server_holder/@acmeId)
let $clientport :=
 <dasada-architecture-mutation>
  <created>
   <newPort id="{$toServer}">
    <description/>
     <type type="simple" href="ClientPortT"/>
     <instanceOf type="simple" href="ClientPortT"/>
   </newPort>
   <context type="simple" href="#{$acmeId}"/>
  </created>
 </dasada-architecture-mutation>
let $toClient := concat ("To ", $acmeId)
let $serverport :=
 <dasada-architecture-mutation>
  <created>
```

```
   <newPort id="{$toClient}">
    <description/>
     <type type="simple" href="ServerPortT"/>
     <instanceOf type="simple" href="ServerPortT"/>
   </newPort>
   <context type="simple" href="#{$server_holder/@acmeId}"/>
  </created>
 </dasada-architecture-mutation>
let $cli := concat ($acmeId, ".", $toServer)
let $ser := concat ($server_holder/@acmeId, ".", $toClient)
let $channelName := concat ($server_holder/@acmeId,
   " -- ", $acmeId, " Socket")
let $clientSideRole := concat ($channelName, ".clientSide")
let $serverSideRole := concat ($channelName, ".serverSide")
let $socketchannel :=
 <dasada-architecture-mutation>
  <created>
   <newConnector id="{$channelName}">
    <description/>
     <type type="simple" href="CSConnT"/>
     <instanceOf type="simple" href="CSConnT"/>
   </newConnector>
   <context type="simple" href="{$sys/@acmeId}"/>
  </created>
 </dasada-architecture-mutation>
let $client_side :=
```

```
   <dasada-architecture-mutation>
    <attached>
     <attachedConnector>
      <roleName type="simple" href="{$clientSideRole}"/>
      <portName type="simple" href="{$cli}"/>
     </attachedConnector>
    </attached>
   </dasada-architecture-mutation>
  let $server_side :=
   <dasada-architecture-mutation>
    <attached>
     <attachedConnector>
      <roleName type="simple" href="{$serverSideRole}"/>
      <portName type="simple" href="{$ser}"/>
     </attachedConnector>
    </attached>
   </dasada-architecture-mutation>
 ?}
}
```

At last, I need to add the Composition part to the specification. First, I need to send the information about the Acme system created with the CreateSystem rule to both the CreateServer and CreateClient rules. Second, the information about the server created in the CreateServer rule has to be sent to the CreateClient rule.

Here is the actual rule exposing what I just said:

```
composition ClientServer {
  CreateSystem.$sys_holder<->CreateServer.$sys
```

```
  CreateSystem.$sys_holder<->CreateClient.$sys
  CreateServer.$server_holder<->CreateClient.$server_holder
}
```

Note that this specification works only for an application that has only a server. If I want to recognize also applications that have more than one server I need to make some variations. In particular I need to be able to discriminate between the servers and know what exact server a client access. Otherwise I could have a final architecture in AcmeStudio with erroneous Client-Server connections since a client is connected to a server without checking if it is the one that is actually calls.

A possible solution, that I implemented and tested, is the use of the server port as a discriminant as in the following modified specification (for the sake of synthesis I just show the parts that are added, the rest is the same of the previous solution):

```
import <sys_events.xsd>
...
event {
 input {
  init;
  call;
  holder;
  string;
 }
 output {
  dasada-architecture-mutation;
  holder;
  string;
 }
```

```
}

rule CreateServer {
 input {
  init $f;
  holder $sys;
 }
 output {
  holder $server_holder;
  string $port;
  dasada-architecture-mutation $server;
 }
 trigger {? contains ($f/@constructor, "ServerSocket") ?}
 action {?
  let $acmeId := concat ("Server_", $f/@instance_id, "-port_", $f/arg[1]/@value)
  let $port := $f/arg[1]/@value
  ...
 ?}
}

rule CreateClient {
 input {
  call $d;
  init $c;
  holder $sys;
  string $port;
```

```
  holder $server_holder;
 }
 output {
  dasada-architecture-mutation $client;
  dasada-architecture-mutation $clientport;
  ...
 }
 trigger {? contains ($d/@method_name, "ServerSocket.accept")
  and $d/@callee = $server_holder/@implId
  and contains ($c/@constructor, "java.net.Socket")
  and $c/arg[2]/@value = $port
 ?}
 action {?
 ...
 ?}

composition ClientServer {
 ...
 CreateServer.$port->CreateClient.$port
 ...
}
```

Note that this still will not work in the case of distributed client-server systems, in which some server have the same port number.

After having tested the DiscoSTEP specification (the outcome of this test will be shown in Section 5.3.1 I wrote and having gained confidence in my knowledge of this new language,

I went on writing the probe aspects (see Section 5.1.1) and implementing the specification for RMI client-server applications, which is now explained.

First of all I had to import the necessary XML schemas that define the several events used in the specification, which are the same of the previous specification. I then proceeded in the definition of input and output events. As for the XML schemas imported, also the declared events are the same of the previous specification, with the addition as both input and output event of the type string as it is needed for a couple of rules, as we will see. I then went on doing the actual implementation of the specification by writing the rules.

The first rule is the one that creates a new AcmeStudio system, so then I can add the successive components I create. It is exactly the same of the one already specified for the previous specification so I will not show it again.

The second rule is the one that creates an RMI server. This rule needs to be enabled when the *Naming.rebind* is called; so when a call event with the attribute *method_name* containing the string "Naming.rebind" is received. Then, this rule has to output a "new Acme Server component" architectural event and a holder containing information about the newly created server, more precisely:

- implId contains the server name specified in first argument of the method.
- acmeId contains that same name preceeded by the string "Server". It will be used as the identifier for the component in the Acme system.

As already said before, in order to be able to associate the newly created components to an existing Acme system, a rule has to receive in input also the holder containing information

about that system.

Here is the actual rule exposing what I just said:

```
rule CreateServer {
 input {
  call $c;
  holder $sys;
 }
 output {
  dasada-architecture-mutation $server;
  holder $server_holder;
  }
 trigger {? contains ($c/@method_name, "Naming.rebind") ?}
 action {?
  let $translated := translate( $c/arg[1]/@value, ":", ",")
  let $acmeId := concat ("Server ", $translated)
  let $server :=
   <dasada-architecture-mutation>
    <created>
     <newComponent id="{$acmeId}">
      <description/>
       <type type="simple" href="ServerT"/>
    <instanceOf type="simple" href="ServerT"/>
    </newComponent>
     <context type="simple" href="{$sys/@acmeId}"/>
    </created>
   </dasada-architecture-mutation>
```

```
  let $server_holder := <holder implId =
    "{$c/arg[1]/@value}" acmeId="{$acmeId}"/>
 ?}
}
```

The third rule needed is the one that creates the RMI client component. This rule needs to be enabled when the *Naming.lookup* is called with the identifier of the newly created server (its name used for the rebinding) passed as the only argument. In other word, the rule is enabled when it receives a call event with the attribute *method_name* containing the string "Naming.lookup" and the first attribute *value* of the complex element *arg* containing the server identifier passed as input. The rule then outputs a "new Acme Client component" architectural event and a holder containg information about the newly created RMI client, more precisely:

- implId contains the client identifier generated by the probe aspects at runtime, contained in the attribute *instance_id* of the init event.
- acmeId contains that same identifier preceeded by the string "Client_". It will be used as the identifier for the component in the Acme system.

Here is the actual rule exposing what I just said:

```
rule CreateClient {
 input {
  call $d;
  holder $server_holder;
  holder $sys;
```

```
 }
 output {
  dasada-architecture-mutation $client;
  holder $client_holder;
 }
 trigger {? contains ($d/@method_name, "Naming.lookup")
  and $d/arg[1]/@value = $server_holder/@implId ?}
 action {?
  let $acmeId := concat ("Client_", $d/@caller)
  let $client :=
   <dasada-architecture-mutation>
    <created>
     <newComponent id="{$acmeId}">
      <description/>
       <type type="simple" href="ClientT"/>
       <instanceOf type="simple" href="ClientT"/>
     </newComponent>
     <context type="simple" href="{$sys/@acmeId}"/>
    </created>
   </dasada-architecture-mutation>
  let $client_holder := <holder implId="{$d/@caller}" acmeId="{$acmeId}"/>
 ?}
}
```

The fourth rule is used to create a port element on existing Client components for each remote method called by them.

I know for sure that every single method called by a client is a remote one, because I implemented the probe aspect to do so (as explained in the previous section). So the rule needs to be enabled when a generic method is called by an already existing client, whose identifier is passed as input. I had also to control that the method is not "Naming.lookup" (which is the only other method instrumented on the client side), otherwise this rule would also be triggered by the call of a non remote method, leading to an erroneous architecture. In fact, a first call to "Naming.lookup" would just activate the CreateClient rule which will produce as output a holder that, with a second call to that method would also activate the CreateClientRMIPort.

The rule then outputs a "new Acme Port component" architectural event. Port that will be attached to the already existing client represented by the holder received as input. It then also outputs the name of the remote method that triggered the rule and a holder containing information about the newly created component, more precisely:

- implId contains the name of the remote method that triggered the rule.
- acmeId contains that same name followed by the string "Client port". It will be used as the identifier for the component in the Acme system.
- parentId contains the acmeId of the client holder passed as input (which is "Client_" followed by the client identifier generated by the probe aspects at runtime).

Here is the actual rule exposing what I just said:

```
rule CreateClientRMIPort {
 input {
  call $c;
```

```
 holder $client_holder;
}
output {
 dasada-architecture-mutation $clientport;
 string $method;
 holder $clientport_holder;
}
 trigger {? $c/@caller = $client_holder/@implId and
   not (contains ($c/@method_name, "Naming.lookup")) ?}
 action {?
  let $method := substring-after(substring-after($c/@method_name, "."), ".")
  let $acmeId := concat ($method, " Client port")
  let $clientport_holder := <holder acmeId="{$acmeId}"
    implId="{$c/@method_name}" parentId="{$client_holder/@acmeId}"/>
  let $clientport :=
   <dasada-architecture-mutation>
    <created>
     <newPort id="{$acmeId}">
      <description/>
       <type type="simple" href="ClientPortT"/>
       <instanceOf type="simple" href="ClientPortT"/>
     </newPort>
     <context type="simple" href="#{$client_holder/@acmeId}"/>
    </created>
   </dasada-architecture-mutation>
 ?}
```

```
}
```

The fifth rule is used to create a port element on the existing Server component for each remote method exposed that it is actually accessed by someone.

I showed before (in the implementation of the probe aspect for the RMI application) that, when a remote method on the server side is called for the first time, it emits, apart from the call event that is always emitted, an init event. I came up with this solution to be able to trigger this particular rule. In fact, by doing so, this rule is triggered only for the first server side call to a particular remote method so that for every different remote method only a single port is attached to the server. To be sure that the init event that triggers the rule is actually generated by the execution of a remote method, I control that its name (that in the case of the init event is contained in the *constructor* attribute) coincide with the name of an already called remote method on the client side, which is passed as input (a remote method is first called on the client-side and then the call is forwarded on the server-side).

The rule then outputs a "new Acme Port component" architectural event. Port that will be attached to the already existing server represented by the holder received as input. It then also outputs the name of the remote method that triggered the rule and a holder containing information about the newly created component, more precisely:

- implId contains the string passed as input.
- acmeId contains that same string followed by the string "Server port". It will be used as the identifier for the component in the Acme system.

- parentId contains the acmeId of the server holder passed as input (which is “Server” followed by the server name).

Here is the actual rule exposing what I just said:

```
rule CreateServerRMIPort {
 input {
  init $i;
  holder $server_holder;
  string $method;
 }
 output {
  dasada-architecture-mutation $serverport;
  holder $serverport_holder;
 }
 trigger {? contains ($i/@constructor, $method) ?}
 action {?
  let $acmeId := concat ($method, " Server port")
  let $serverport_holder := <holder acmeId="{$acmeId}"
    implId="{$method}" parentId="{$server_holder/@acmeId}"/>
  let $serverport :=
   <dasada-architecture-mutation>
    <created>
     <newPort id="{$acmeId}">
      <description/>
       <type type="simple" href="ServerPortT"/>
       <instanceOf type="simple" href="ServerPortT"/>
```

```
    </newPort>
    <context type="simple" href="#{$server_holder/@acmeId}"/>
   </created>
  </dasada-architecture-mutation>
 ?}
}
```

The last rule is used to create a new Connector element, representing the call made to a remote method, and attach it to two already existing server and client ports. Thus the rule takes as input a call event, a string containing the name of a remote method that has been called by a client, one holder containing the information on an existing server port and another one on an existing client port.

Given this inputs, this rule triggers only when the call event passed as input is a server side remote method that has already been called by a server (so when the call event attribute *method_name* has the same value of the string passed as input) and the relative server and client port related to it have already been created (so when the clientport_holder and serverport_holder are available as input).

The rule then outputs a "new ClientServer Connector component" architectural event. The client and server sides of this connector are attached to the ClientPort and ServerPort referenced by the input holders (clientport_holder and serverport_holder) with the use of two AcmeStudio "attach connector" architectural events.

Here is the actual rule exposing what I just said:

```
rule Connect {
 input {
  call $c;
  holder $sys;
  holder $clientport_holder;
```

```
 holder $serverport_holder;
 string $method;
}
output {
 dasada-architecture-mutation $socketchannel;
 dasada-architecture-mutation $client_side;
 dasada-architecture-mutation $server_side;
}
trigger {? contains ($c/@method_name, $method)
 and contains ($clientport_holder/@implId, $method)
 and contains ($serverport_holder/@implId, $method)
?}
action {?
 let $channelName := concat ("Server ", $serverport_holder/@parentId,
   " -- Client ", $clientport_holder/@parentId, " ", $method, " Channel")
 let $clientSideRole := concat ($channelName, ".clientSide")
 let $serverSideRole := concat ($channelName, ".serverSide")
 let $cli := concat ($clientport_holder/@parentId, ".",
   $clientport_holder/@acmeId)
 let $ser := concat ($serverport_holder/@parentId, ".",
   $serverport_holder/@acmeId)
 let $socketchannel :=
  <dasada-architecture-mutation>
   <created>
    <newConnector id="{$channelName}">
     <description/>
```

```
      <type type="simple" href="CSConnT"/>
      <instanceOf type="simple" href="CSConnT"/>
     </newConnector>
     <context type="simple" href="{$sys/@acmeId}"/>
    </created>
   </dasada-architecture-mutation>
  let $client_side :=
   <dasada-architecture-mutation>
    <attached>
     <attachedConnector>
      <roleName type="simple" href="{$clientSideRole}"/>
      <portName type="simple" href="{$cli}"/>
     </attachedConnector>
    </attached>
   </dasada-architecture-mutation>
  let $server_side :=
   <dasada-architecture-mutation>
    <attached>
     <attachedConnector>
      <roleName type="simple" href="{$serverSideRole}"/>
      <portName type="simple" href="{$ser}"/>
     </attachedConnector>
    </attached>
   </dasada-architecture-mutation>
 ?}
}
```

Note that with these rules, I create and visualize the connections between the clients and the server established through the call of a remote method only when this call is actually executed by the server and not when its execution is requested by the client. In this way, if a call fails for any reason before being executed by the server, it will not be created and shown in the final architecture, thus leading to a more precise outcome.

At last, I need to add the Composition part to the specification, which once specified the rules is pretty straight-forward. First, for the reasons I already explained, I need to send the information about the Acme system created with the CreateSystem rule to all the rules that create new components: CreateServer, CreateClient and Connect. Second, the information about the RMI server created in the CreateServer rule has to be sent to the CreateClient rule, so that I can check that the client connects to an already existing server, and to the CreateServerRMIPort rule, so that the port created with this rule can be attached to the server. Third the information about the RMI client created in the CreateClient has to be sent to the CreateClientRMIPort rule so that the port created with this rule can be attached to the client. Fourth the name of the remote method found with the CreateClientRMIPort rule has to be sent to the CreateServerRMIPort and Connect rules. Fifth the information about the server port created in the CreateServerRMIPort rule and about the client port created in the CreateSClientRMIPort rule has to be sent to the Connect rule, in this way I can know what the ports that have to be connected are.
Here is the actual rule exposing what I just said:

```
composition RMI {

  CreateSystem.$sys_holder<->CreateServer.$sys

  CreateSystem.$sys_holder<->CreateClient.$sys

  CreateSystem.$sys_holder<->Connect.$sys

  CreateServer.$server_holder<->CreateClient.$server_holder
```

```
  CreateServer.$server_holder<->CreateServerRMIPort.$server_holder
  CreateClient.$client_holder<->CreateClientRMIPort.$client_holder
  CreateClientRMIPort.$method<->CreateServerRMIPort.$method
  CreateClientRMIPort.$method<->Connect.$method
  CreateClientRMIPort.$clientport_holder->Connect.$clientport_holder
  CreateServerRMIPort.$serverport_holder<->Connect.$serverport_holder
}
```

5.2.2 Problems and limitations

Understanding well the DiscoSTEP specifications and especially how to write a functioning one, came out to be one of the most challenging task. In this case, compared to the probe aspects problems cited before, I had more documentation (the published paper and its appendix, (3; 81; 82)), but it was not of much help for three main reasons.

First, the paper (3) offers a brief example of a DiscoSTEP program that is in some part wrong, in other useless for my purpose. In fact the XML representation of the call and init events used in it is not consistent with the one actually used (almost all the attributes have different names). Then, the architectural events of the rules do not follow the Acme syntax that is actually used in practice in the available pipe and filter example.

Second, the Appendix B (82) contains only the formal semantics of the language, which is no use for my purpose.

Third, Appendix A (81) contains useful information, in particular the concrete syntax of DiscoSTEP, but the example is almost useless, since its syntax is in many parts not the one that

is actually used. In fact I compared it with the specification of the supplied pipe and filter example and the two did not coincide.

So, given these problems, I had to reverse engineer from the only existing and functioning example and proceed with a "trial and error" approach until I gained enough knowledge on the language. This required quite some time, especially the part related to the architectural events. In fact, the other parts of a DiscoSTEP specification have a quite simple structure and after an initial phase spent on the understanding of the language, most of the time spent on these parts was about getting the specification recognize only the required target architecture. On the other hand, almost all the time spent on the implementation of the output events was on understanding how to write architectural events since there can be of many types and the available example only showed a couple of them. While for the other parts of the specification some documentation, even if insufficient, existed, this part totally lacked of it with the exception of the XML schema definining all the possible events (*DASADA-Arch-Mutation-Schema.xsd*) that is surely not a clear and easily readable document (this schema also lacks of comments that at least might have helped its understanding).
Another problem that actually came out while talking with one of the main developer of DiscoTect is that, as for now, the directional bindings (used in the Composition part of the specification) is not implemented yet and as for now it behaves exactly as the bidirectional one. This can be a problem in some application because in this way events sent from one rule to another will never be consumed, thus possibly causing the erroneous triggering of rules. For example, let's look at the following specification fragment:

```
rule Rule1 {
 input {
  ...
```

```
  }
 output {
  ...
  string $output_string;
 }
 trigger {? ... ?}
 action {?
  ...
  let $output_string := "output"
 ?}
}

rule Rule2 {
 input {
  ...
  string $input_string;
 }
 output {
  ...
 }
 trigger {?
  input_string = "output"
  ...
 ?}
 action {?
  ...
```

```
 ?}

composition ClientServer {
 ...
 Rule1.$output_string->Rule2.$input_string
}
```

In this case, if “Rule1” fires even only once, “Rule2” will keep firing continuously even if I wanted to fire only once, as it was declared in the composition.

5.3 The sample applications

In this section I will cover the applications I created to test the probe aspects and the DiscoSTEP specifications I implemented and that have been explained in the previous two sections. I will quickly explain the strucuture of the applications, their behaviour and the AcmeStudio output after the dynamic analysis of them. Note that these applications I built before actually implementing the related aspects and specifications, so that I could test the things I wrote as I was writing them. But since it is their runtime structure visualized by AcmeStudio, which comes after the explaination of the aspects and the DiscoSTEP specifications because, that is meaningful to me, I decided to put this part here.

5.3.1 Socket client-server

The client-server socket application is based on a server that for every incoming connection creates a new thread that receives a message from the client that established the connection, prints it on screen, sends back an acknowledgement, closes the connection and terminates. The client, on the other hand, sets up a connection with the server, sends a message, waits a re-

sponse, prints that response and then closes the connection.

As it can be seen, this is a very simple application, but it exposes the main functionalities of a socket based application. Moreover, there is no need to create a more complex application, since the methods that characterize an application as a socket based one (as we saw in Section 5.1.1) are, by definition, present in every application following that paradigm, from the most simple to huge and complex systems.

Figure Figure 25 shows a screenshot of what DiscoTect extracted and showed through AcmeStudio from the execution of this application with the probe aspects I described before weaved in it. Note on the lower part of the screenshot the *Remote Control View* showing a couple of architectural events received through the JMS bus (the actual events formatted as the XML strings specified in the DiscoSTEP specification in purple, the action taken by AcmeStudio given these events in blue).

5.3.2 RMI client-server example

The RMI client-server application is based on a server that exports three remote methods:

- *executeTask_1*, which returns a string containing a message from the server.
- *executeTranslation*, which translate the string passed as single argument of the method and returns the outcome.
- *store*, which save to a file the string passed as single argument of the method and returns 1 if it has been correctly saved, otherwise 0.

Then, there are three clients that call the remote methods of this server. One calls *executeTask_1* and *executeTranslation*, a second one calls only calls *executeTranslation*, while the third calls

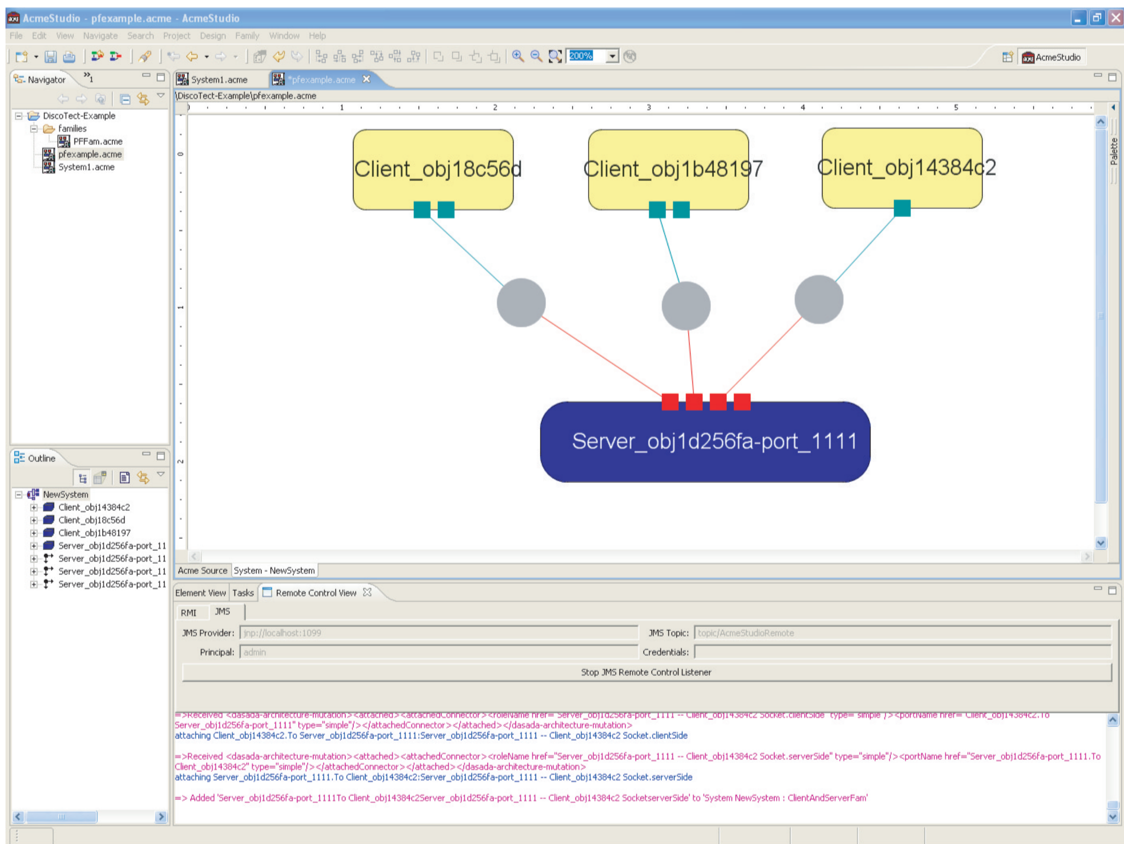


Figure 25. Architecture extracted from the socket client-server.

executeTranslation and then stores the result of that method with the *store* remote method. As in the previous example, also this application is quite simple but it exposes and uses all the key RMI methods. Figure Figure 26 shows a screenshot of what DiscoTect extracted and showed through AcmeStudio from the execution of this application with the probe aspects I described before weaved in it. In the low left side of that screenshot, which shows the outline of the whole architecture, it can be seen that all the ports and connectors carry in their name the remote

method that they represent. This makes the architecture found a lot more meaningful, since it shows the actual methods exposed and called by the single clients.

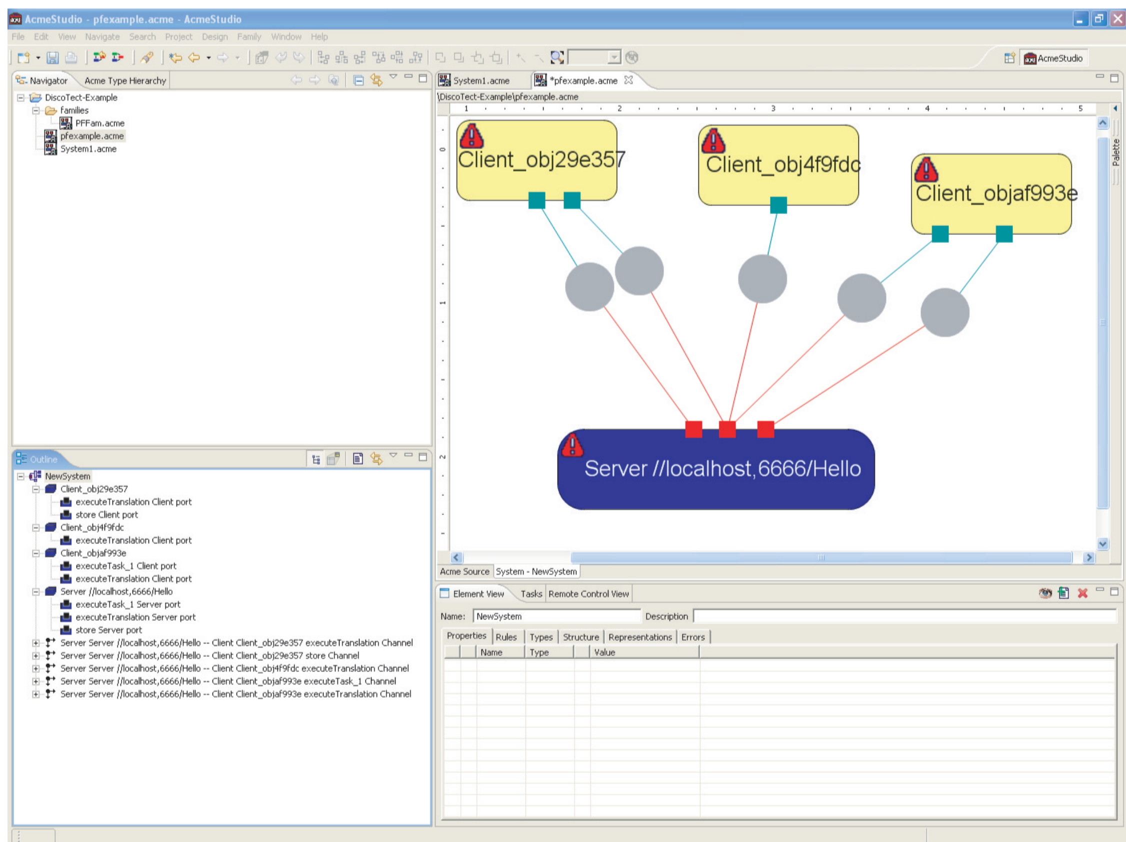


Figure 26. Architecture extracted from the RMI client-server.

CHAPTER 6

MODERNIZER

In this following chapter I will explain the Eclipse plug-in I developed, named Modernizer, to support the semi-automatic transformation Java applications; in this first version I implemented the transformation of RMI applications to Webservices. In particular, this plugin integrates DiscoTect to extract from the system under study the required information to trigger and drive a specific transformation (even though as for now, only one transformation is offered). Then at the very end of the process I offered functionalities to deploy the created Web service on the application server of choice.

The order of the description is the same I followed before: it reflects the order of usage of the single parts during the use of the plugin for the modernization of an existing application. So I start from the definition and compilation of the probe aspects and DiscoSTEP specifications and I end with the deployment of the Web service representing the newly modernized application.

6.1 Reasons for using Eclipse

I decided to develop my prototype on Eclipse for many reasons.

First of all by using it I do not have to write the complete application from scratch, since Eclipse, with its Rich Client Platform already offers many tested features, as a portable widget toolkit (through the Standard Widget Toolkit, SWT), file buffers, text handling, text editors (thorugh JFace) and views, editors, perspectives, wizards (through The Eclipse Workbench). In this way

I can focus only on the parts closely related to the project, while letting Eclipse take care of all the rest.

Second, all the existing tools used are already developed as Eclipse plugin: AcmeStudio, AspectJ and Java development tools (110; 111); so building my prototype on the same platform allows a very fast and easy integration of all the components. The only part that was not already developed as an Eclipse plugin is the core of the DiscoTect Runtime Engine (the core of that system) which, as we will se, was integrated in my prototype and thus in Eclipse by me.

In this way, a user does not have to cope with several application and tools, she/he just needs to open the existing Java application project to study and modernize with Eclipse and all the functionalities she/he needs are already there, in one single workbench.

6.2 The plug-in in depth

6.2.1 Convert Action

This action is used to convert a normal Eclipse Java application project to a Modernizer project so that the application can be studied with the functionalities offered by Modernizer and then modernized.

When the user starts this conversion by choosing the option “Convert to Modernizer Project” as shown in Figure Figure 27, she/he is first asked to choose a series of architectures to be used to test the application under study. Figure Figure 28 shows the window dialog containing the list of architectures the user has to choose from.

This will lead to the creation of:

- A DiscoSTEP specification for each type of architecture chosen.

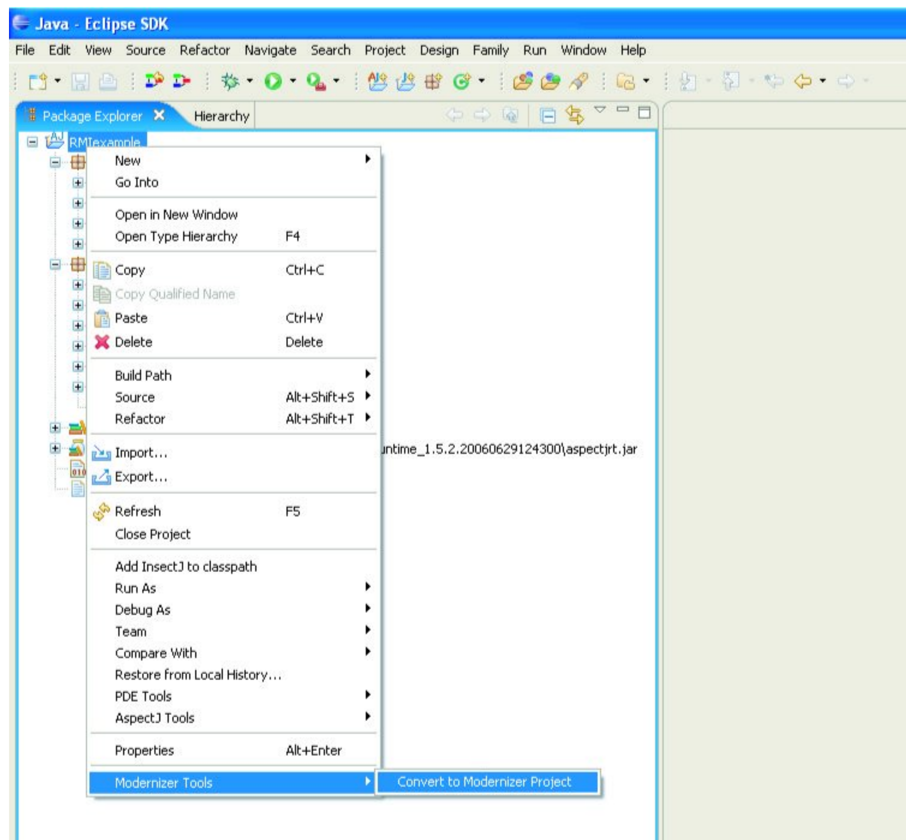


Figure 27. Screenshot of the beginning of the conversion process.

- An empty AcmeStudio file importing, for each type of architecture chosen, the related Acme family.
- A probe aspect for each type of architecture chosen.

All these files are then added to the project, as it can be seen in Figure Figure 29. In addition to this, the jar files (*DiscoTect.jar* and *jbossall-client.jar*) required by the probe aspects to work are added to the project's Java build path. At last, the Modernizer nature is added to the

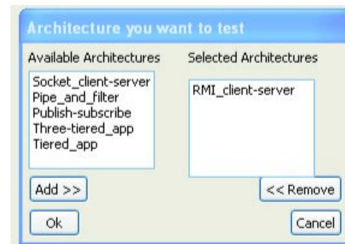


Figure 28. Window dialog with the list of architectures to choose from.

project natures. I decided to create this custom nature so that the specific functionalities of my plugin cannot be activated and applied in any project, but just on those having that nature.

When the conversion has been done, a message dialog, like the one shown in Figure Figure 30, is opened. As it can be noted, if the conversion was successful, the message dialog also reminds the client to add the string "-Dj2eeImpl=jboss" to the VM arguments of the project, which is needed for the probe aspect to emit the filtered events on the JMS bus.

Note that a Modernizer project, in order to work, has to be converted also to an AspectJ project, since aspects are used by DiscoTect for architecture analysis.

6.2.2 Compile Action

This action, as the name suggests allows the user to compile a DiscoSTEP specification as shown in Figure Figure 31.

In fact, in order to be used by the DiscoTect Runtime Engine, a specification has to be compiled with the DiscoStep Compiler. I decided to let the user compile the specifications

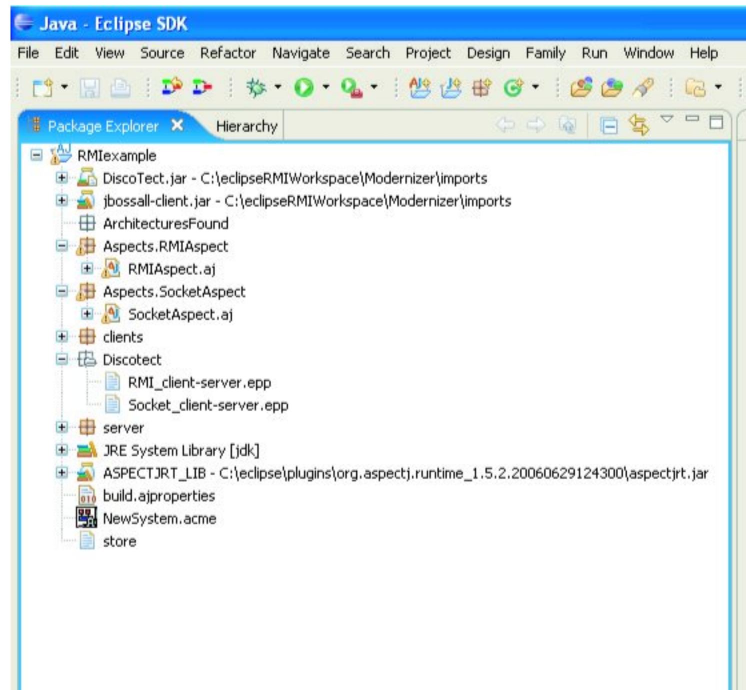


Figure 29. Overview of a converted project, with the newly created aspects, DiscoSTEP specifications and AcmeStudio file.

instead of giving him the ones already compiled because in this way, if necessary, she/he can modify the specification or even create a new one to fit it to his needs.

When implementing this feature, I had to solve two main problems: run the DiscoSTEP from Eclipse and retrieve the output messages of the compilation. Since the DiscoSTEP compiler needs to be run under cygwin, I first studied how to call cygwin from the Windows command prompt, since the command prompt is can then be easily ran from Java with the "Runtime.getRuntime().exec(...)" method.

In particular I studied the cygwin shell, since the compiler is called with the *dsc.sh* shell script

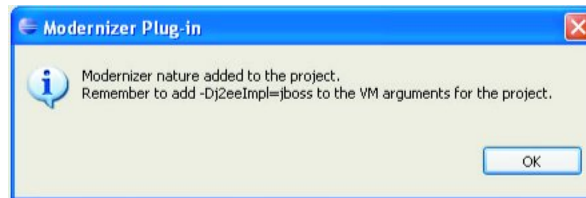


Figure 30. Message dialog at the end of the conversion process.

provided in the DiscoTect distribution. I found out that the cygwin shell and all the others cygwin programs can be called as standard program, without having to open a cygwin terminal. So, for example, to execute a cygwin shell script from a Windows command prompt the string *cygwin_directory_path/bin/sh shell_script_path eventual_script_parameters* is only needed. This solved the problem of running the compiler from Eclipse, which thus can be done by automatically opening a command prompt and making it execute the string *cygwin_directory/bin/sh script_directory/dsc.sh file_to_be_compiled*.

To retrieve the output messages of the compiler, I then simply redirected its output to a file, which in cygwin can be done by adding the string > *output_file* after the script/command to be executed.

So in my case the final string I used to compile a DiscoSTEP specification and capture the compilator output is: "cygwin_directory/bin/sh dsc.sh file_to_be_compiled > output_file". When the compilation is done, the output file is then read by the plugin, which shows on a message dialog the results: if the compilation was successful, as in Figure Figure 32, or if there are errors in the specification and what/where those errors are, as in Figure Figure 33.

6.2.3 Run Action

This action allows the user to run the DiscoTect Runtime Engine with some of the DiscoSTEP specifications associated to the project.
When the user chooses the option "Run DiscoSTEP specifications", she/he is first asked to choose the specifications she/he wants to run through a window dialog containing the list of all the specifications associated to the project that have already been compiled (obviously, the ones not compiled will not be shown since they can't be run on the DiscoTect Runtime Engine). Figure Figure 34 shows this message dialog.

Each selected specification is then run with a new instance of the DiscoTect Runtime Engine. Since, as the compiler, also the Runtime Engine has to be executed through cygwin (with the *dsr.sh* shell script), in order to run the specifications, I applied the exact same solution exposed in the previous section.
As outcome of this action, each selected specification is then run on an instance of the Runtime Engine (through a Windows command prompt window), waiting for events emitted from probe aspects to be interpreted, as shown in Figure Figure 35.

6.2.4 Transform Action

This action is the most important in my prototype since is the one that from the information extracted from the dynamic analysis, it proposes a list of possible transformations to be applied to the application and then depending on the transformation selected, semi-automatically transform it.

To make all this possible, first of all I had to find a way to somehow let my prototype understand what are the type of architecture found during the dynamic analysis. In fact, DiscoTect provides only an Acme model of the architecture found which then the user can look and interpret to extract the information they think is useful. But it is not easy, if not impossible, for another application to automatically extract from the model found all the required information to apply the chosen transformation the application.

So I decided to augment the existing probe aspects to make them log to a file all the meaningful information of the monitored application. In particular, for my first implemented transformation, RMI to Web service, the information needed is the signatures of all the remote methods and the server class name. In fact, in this way I know what the class and the methods that need to be transformed are.
So, I make the advice that intercepts the creation of the new server write to the log file the line “Server class name:” followed by the name of the server, which is done with the *thisJoinPoint.getSignature().getDeclaringTypeName()* method. For the remote methods signatures, I make the advice that intercepts the server side calls of remote methods write to the log file the line “Server remote method signature:” followed by the signature of the method, which in this case is done by calling “thisJoinPoint.getSignature().toString()”.
Obviously, a remote method can be called several times, but I am interested only in its first call so I need to write it only once on the log, since what is meaningful to me is its existence, not its call frequency. The implementation of this additional requirement was quick, since in my aspect I already discriminated the first call to a remote method with respect to all the following calls (see Section 5.1.1), so I only had to insert the logging part in the part of the advice related to the first occurrence of the method.
Here are the two advices of the probe aspect for RMI applications previously explained that were modified:

```
after(Object tar) returning(Object result):
  genericMethod2() && target(tar) && serverClasses(){
   Class cl[] = tar.getClass().getInterfaces();
     int len = cl.length;
     int i;
     for(i=0; i<len; i++){
       Class superCl[]=cl[i].getInterfaces();
       int len2 = superCl.length;
       int j;
       for(j=0; j<len2; j++){
         if(superCl[j].toString().contains("Remote")){
           String meth = thisJoinPoint.getSignature().getName();
           if(!serversMethods.contains(meth)){
           /** Here is where I write to a file the remote methods signatures **/
            try {
              this.print.append("Server remote method signature: " +
                 thisJoinPoint.getSignature().toString() + "\n");
              this.print.flush();
            }
            catch (IOException e) {
             System.out.println("Error while writing a server remote method signature");
             e.printStackTrace();
            }
            serversMethods.add(meth);
            emitInit (thisJoinPoint, result);
           }
```

```
          emitCallEvent(thisJoinPoint, null, "call");
        }
      }
    }
  }
...

...
before(): mainMethod() && serverClasses() {
  if(!created){
    serversMethods = new ArrayList();
    created=true;
    /**
     * Here is where I open a file and write in it the server name, it will then
     * be left open so also the remote method names can be written in it.
     * The correct name of output file will be written by the Modernizer plugin.
     * Just put leave the name written instead of its real name and the plug-in
     * will take care of it.
     **/
    try {
      this.print = new BufferedWriter(new FileWriter
        ("C:/runtime-New_configuration/RMIexample/ArchitecturesFound/RMIAspectOutput"));
      this.print.write("Server class name: "
        + thisJoinPoint.getSignature().getDeclaringTypeName() + "\n");
    } catch (IOException e) {
      System.out.println("Error while writing the server class name");
```

```
      e.printStackTrace();
    }
  }
    emitCallEvent(thisJoinPoint,null, "call");
}
```

Note that in this case, the log file is wrote in the *C:/runtime-New_ configuration/RMIexample/ArchitecturesFound/* directory, which is not a hard-wired directory as it seems, otherwise it would only work with a specific project. The project specific directory is added to the code when the project is converted to a Modernizer one. The *ArchitecturesFound* folder contains the log files created by the all the probe aspects used for that particular project.

Here is the log file after the execution of the sample RMI application I already introduced in Section 5.3.2:

```
Server class name: server.ServerEngine
Server remote method signature: String server.ServerEngine.executeTask_1()
Server remote method signature: String server.ServerEngine.executeTranslation(String)
Server remote method signature: int server.ServerEngine.store(String)
```

In addition to this, my prototype even by only scanning quickly the names of the files in the *ArchitecturesFound* folder of the project can have an idea of what architectures might be implemented by the system under study. In fact the probe aspects are written so that they name of the log file with part of their own name. For example, in the previous example, the RMIAspect probe aspect names the the log file *RMIAspectOutput*; the SocketAspect names it *SocketAspectOutput*. So, if I know what aspect generated the output, I also know the architecture that was found since each aspect is in charge to test if the application under study implements

a single architecture.

Note that I showed only the case of the probe aspect for the RMI client-server application since it is crucial for my prototype, but any aspect that has to be used to extract any other architecture has to be augmented with this type of logging I just explained. In fact I did that also with the other aspect I wrote, the one for the client-server based on sockets but I will not show it here since it is not crucial for this part of the work.

Thus, when the user selects the "Transform the system" option, she/he is first asked to choose the transformation to apply to the existing application, through a window dialog containing the list of all the possible transformations depending on the architectures that were found during the analysis. In my first case, as I already wrote, the only transformation implemented is the RMI client-server to Web service, but as it can be seen in Figure Figure 36, if also other architectures are found, in particular the Socket client-server, the hypothetical transformations are presented to the user even though they are not implemented (they can be considered as "stubs" for possible future additions).

After the user selects the desired transformation, in my case RMI to Web service, a second message dialog window is opened, as showed in Figure Figure 37. This window shows all the remote methods that were discovered during the execution of the application, the location of the server class and of the interface extending "Remote" that the server needs to implement (this is part of the typical java RMI coding paradigm).

All this information is automatically extracted by the probe aspect log file I just explained. The remote methods are directly extracted from that file and showed on that window. The actual

server class is searched on the project directory by using the server class name that was saved on the log file. The server class found is then read to extract the name of the interface extending "Remote", whose actual file is then searched on the project directory.
I need to know the system path of the classes of the server and of that interface since thay are the classes that need to be modified in the transformation. In fact those two classes, in a standard RMI coding paradigm, are the only two containing the RMI logic, thus they are the only two that need to be modified.

If there was any problem in automatically finding them (error in the log file, classes not in the project directory, etc.), these two important classes can even be searched by the user by hand. This problem might arise especially for the interface, since sometimes they do not resides in the project directory since they can be used by different applications and, thus, projects.
This can be seen in Figure Figure 37, even though in this case the classes were found so the buttons and fields related to the search are disabled so the user cannot mess things up.
So, in order to start the transformation, the user selects the remote methods she/he wants to be transformed in WebMethods in the Webservice and the server class and interface if the they were not automatically found and then presses the "OK" button. Figure Figure 38 shows the window ready for the "OK" button to be pressed and the transformation to be started.

For the transformation process I implemented a class called "RMITranslator". This translator first cleans the server and the interface from all the parts related to RMI and then adds the

required annotations to turn the server into a Web service.

The cleaning of the interface is very simple, since only the following strings have to be removed:

- *import java.rmi.Remote;*
- *import java.rmi.RemoteException;*
- *extends Remote;*
- *throws RemoteException*

In fact what the interface does is only declare the remote methods, which all have to throw a RemoteException, and extend the Remote interface.

The cleaning of the server is more complex. First of all the following strings have to be removed:

- *extends UnicastRemoteObject*
- *Naming.rebind(...);*
- *import java.rmi.Naming;*
- *import java.rmi.RMISecurityManager;*
- *import java.rmi.RemoteException;*
- *import java.rmi.server.UnicastRemoteObject;*
- *throws RemoteException*
- *System.out.println(...);*

Note that all the *System.out.println(...)* methods are removed, since in a Web service they are useless beacuse there is no output console server side to write to. In my first solution I decided to

cancel them, but another possible solution might be to substitute every call with a new method writing to a file the string previously wrote on the output console. Another solution might be leaving all the calls to that methods also in the transformed version and inform the user that they have to be modified or canceled, depending on the needs.

Then also the *main(String[] args)* method has to canceled since a Web service cannot have it. In fact it is instantiated when a client calls it through the Network, never as a standard Java application (through a console). All the things inside the *main(String[] args)* method cannot just be deleted, because there might also be some parts necessary for the application to work, and thus also necessary for the new Web service. In my solution all the code in that method is moved to the server constructor and then the user has to check what has to be actually deleted, what has to be taken and what has to be modified in order to make it working in the new solution.

Another problem is that in the original server (and also in any standard Java application) information might be passed to the application, when invoked, with the *main(String[] args)* method arguments. This is obviously impossible in the case of Webservices. But if *args* is actually used, it might be used to pass some important information for the server, so it cannot be ignored. In my solution if the translator finds out that *args* is actually used, it outputs a warning message to the user, as in Figure Figure 39 and it adds a specific comment in the translated server class:

TRANSLATOR IMPORTANT MESSAGE: You should carefully check the main and modify

it. In particular because the original solution used parameters passed throught command line input at invocation (which is impossibile for WebServices).

Note that, in order to make this translator applicable to any RMI server I had also to take care of arbitrarly spacing between keywords.
In fact, different programmers or programming environment might format the code with different spacing bewteen words. So when searching for one of the strings to be canceled cited above, for example *throws RemoteException*, I have to keep in mind that there might be any number greater than one of white spaces between those two words. To overcome this problem I used Java regular expressions specified as strings. So, for example, the translator will look for *\s*throws\s*RemoteException\s**, which correspond to any *throws RemoteException* clause with any number of preceding, trailing and between the two words white spaces. By using regular expressions I can also find all the imports to be removed with this single expression: *\s*import\s*java.rmi.*\s**; which represent any import of some class of the java.rmi package. In fact all the imports I need to remove are related to the java.rmi package.

After the removal of the RMI parts is done, the necessary code for the Web service is added. Which constists of:

- *import javax.jws.WebMethod;*
- *import javax.jws.WebService;*
- *@WebService* before the server class definition.

- *@WebMethod* before the definition of any method to be exported as service (all the ones previously found with the architecture analysis and selected by the user at the beginning of the transformation).

After this, the transformation process is done. A new Eclipse project is then created for the transformed application, containing the newly transformed server and related interface along with all the eventual remaining source files of the original application. In fact, what I transform is the type of access to methods, from remote to Web service, but I do not change all the eventual logic behind those methods.
In addition to this, the required Java library for Web service creation (*jaxws-api.jar, jaxws-rt.jar, jaxws-tools.jar, jsr181-api.jar, jsr250-api.jar*) and all the files required for the deployment of the new Web service (*build.xml, web.xml, sun-web.xml, sun-jaxws.xml, build.properties*, which I already explained in Section 4.1.3.1) tuned for the specific Web service created, are added to the classpath of the project. I will talk more about these files in Section 6.2.5, in which I will talk about the actual deployment of the newly created Web service.
Figure Figure 40 shows the newly created project containing the transformed version of the RMI application I introduced in Section 5.3.2 and all the files I just introduced. Note in Figure Figure 40 that in the server class there is a message from the translator and errors in the constructor due to the fact that some of the code in it comes from the *main* method after its removal, as I previously explained. So the user has to do some modification before actually use the new Web service.

I tried to implement the most generic translator, while making the whole process the most automized possible, so that the biggest number of RMI server can be transformed, as it can be seen with the use of regular expressions explained before, but there still are limitations.
In fact, the applications to be transformed have to follow the standard RMI implementation, which can be found in every Java language book (for example, the one by Deitel and Deitel (109)) and online tutorial. I saw that most of the RMI server follow that paradigm so this requirement is not that limiting. But in any way, in order to make any program transformation automatic or semi-automatic we always have to restrict the target programs to some finite standard paradigms/structures otherwise we would have to take into consideration an infinite type of programs, which is obviously undoable.

6.2.5 Web Service Deployment

As I already said in the previous section, the new project created after the transformation process contains all the necessary files for the compilation and deployment of the new Web service so the user does not have to bother with tedious XML stuff that she/he might not even know. These files are already tailored to the specific transformed application during their creation, so if the user does not have any need to fine tune them, they can be used right away without any modification or adjustment.
In fact, my prototype contains templates of all those files and at the end of the transformation process, it makes a copy of them, adds the application and platform specific informations and then incorporates those modified files in the new project. For example, the following template for the *build.properties* file:

```
# MODIFY WITH THE NAME YOU WANT TO GIVE TO THE .WAR FILE

war.file=
```

```
# MODIFY
context.path=
# MODIFY
url.pattern=
# MODIFY WITH THE RIGHT PACKAGE
sei=
build=
# MODIFY WITH THE DIRECTORY CONTAINING THE SOURCE CODE NEEDED
src=
dist=dist
assemble=assemble
web=web
conf=etc
debug=false
build.classes.home=${build}
keep=false
verbose=true
# MODIFY WITH JAVAEE SDK DIRECTORY
javaee.home=
# MODIFY WITH PROJECT DIRECTORY
project.home=
#modify
admin.password.file=${project.home}/admin-password.txt
# MODIFY WITH APP SERVER HOST
admin.host=
# MODIFY WITH APPSERVER USERNAME
```

```
admin.user=
admin.port=4848
https.port=8181
domain.resources="domain.resources"
domain.resources.port=8080

assemble.war=${assemble}/war
assemble.ear=${assemble}/ear
endpoint.address=http://${admin.host}:
  ${domain.resources.port}/${context.path}/${url.pattern}

#Directory containing configuration files
conf.dir=etc
```

is modified into:

```
# MODIFY WITH THE NAME YOU WANT TO GIVE TO THE .WAR FILE
war.file=modernized_RMIexample-jaxws.war
# MODIFY
context.path=server
# MODIFY
url.pattern=modernized_RMIexample
# MODIFY WITH THE RIGHT PACKAGE
sei=server.ServerEngine

build=
# MODIFY WITH THE DIRECTORY CONTAINING THE SOURCE CODE NEEDED
src=
```

```
dist=dist

assemble=assemble

web=web

conf=etc

debug=false

build.classes.home=${build}

keep=false

verbose=true

# MODIFY WITH JAVAEE SDK DIRECTORY

javaee.home=C:/Sun/SDK

# MODIFY WITH PROJECT DIRECTORY

project.home=C:/runtime-New_configuration/modernized_RMIexample

#modify

admin.password.file=${project.home}/admin-password.txt

# MODIFY WITH APP SERVER HOST

admin.host=localhost

# MODIFY WITH APPSERVER USERNAME

admin.user=administrator

admin.port=4848

https.port=8181

domain.resources="domain.resources"

domain.resources.port=8080


assemble.war=${assemble}/war

assemble.ear=${assemble}/ear

endpoint.address=http://${admin.host}:${domain.resources.port}
```

```
/${context.path}/${url.pattern}

#Directory containing configuration files
conf.dir=etc
#Deploy directory for JBoss or any other app server
deploy.dir=C:/Sun/SDK/domains/domain1/autodeploy
```

Note that, in particular, what is added/modified is the JavaEE directory, the project home directory, the administrator password and username, the path of the newly created .war file and the deploy directory which are all used in the ant task necessary for the building (the JavaEE directory and the project home directory) and the deployment (the administrator password, the username, the path of the .war file and the deploy directory) of the Web service as I already explained in Section 4.1.4.

So the user, after having done the eventual modifications to Web service class to fix problems related to the automatic transformation I explained in the previous section, just needs first to build the service, create the .war file and then deploy it throught the Eclipse Ant console, as shown in Figure Figure 41.

Note that the tasks for the building and the creation of the .war file are grouped into a single task, called *build*. So what the user just needs to do is first run the *build* task and then, after it completed successfully, run the *deploy* or the *univ_ deploy* tasks. The first one is specific to the Sun Application Server, while the second one works for any application server. I also added another task called *build-and-deploy* that does all in one step, for lazy users.

If the user then needs to undeploy the service, he can even do that from the application server,

with specific commands, or through Eclipse, by calling either *undeploy* or the *univ_undeploy* tasks. Figure Figure 42 shows the Web service created from the RMI application explained in Section 5.3.2 deployed on the Sun Application Server.

Figure Figure 43 shows the tester for that same Web service, that is the main reason (as already said in the previous chapter) that made me choose the Sun Application Server as the server of choice. In fact, no other application server has this kind of easy and straight forward testing facility.

As it can be noted in this description, the user does not even have to know how Web services actually works or how they are deployed since the Ant tasks supplied and the files automatically created by the tool take care of everything. Obviously, if he needs to do some fine tuning, he will have to modify by hand those files, but Eclipse offers editors that can help.
In order to allow all this automatic project and files creation, the user has to manually set some basic preferences for my tool prototype, which I will cover in the next section.

6.2.6 Plug-in preferences

In my tool there are some settings that are specific to the computer on which it runs that thus need to be set by hand. That is why I created a preference page, shown in Figure Figure 44, so that the user can manually set these values to make the tool work correctly.

The Cygwin and DiscoTect directories path are needed for the compilation and execution of the DiscoSTEP specifications. In fact, the DiscoTect folder contains the compiler (*dsc.sh*) and the runtime engine (*dsr.sh*) used respectively for compilation and execution of the specifications, while the Cygwin folder contains the shell used for their execution.

If those two directories are not set and the user tries to compile or execute a specificatio, the operation will be aborted and an error message, as the one in Figure Figure 45, will be shown.

The JaxWs libraries directory path is needed for the RMI to Web service transformation. In fact, that directory contains the libraries that have to be added to the Java classpath of the newly created project since they are imported by the newly created Web service. More precisely, what the new service imports is: *javax.jws.WebMethod* and *javax.jws.WebService*.
If this directory is not set the transformation will still be done correctly, but a warning message, as the one in Figure Figure 46, will be shown at the end of the whole process.

The JavaEE SDK directory path is needed for building the Web service, since it contains WsGen, which is the Java Web service compiler that creates all the actual classes implementing the service.
If the directory is not set the transformation will still be done correctly, but a warning message, as the one in Figure Figure 47, will be shown at the end of the whole process. If the user does not do what suggested in the warning message, he will not be able to build and thus deploy the service.

The application server username, password and host are needed only if the user wants to deploy the Web service to the Sun Application Server using the dedicated *deploy* task, while the application server deploy directory path is need if the user wants to deploy the service to a generic application server. So the problem arises when none of those fields are set because, in this case, it will be impossible to deploy the service.
So when the application server deploy directory and at least one between the application server

username, password or host field are not set a warning message at the end of the transformation process, as the one in Figure Figure 48, will be shown.

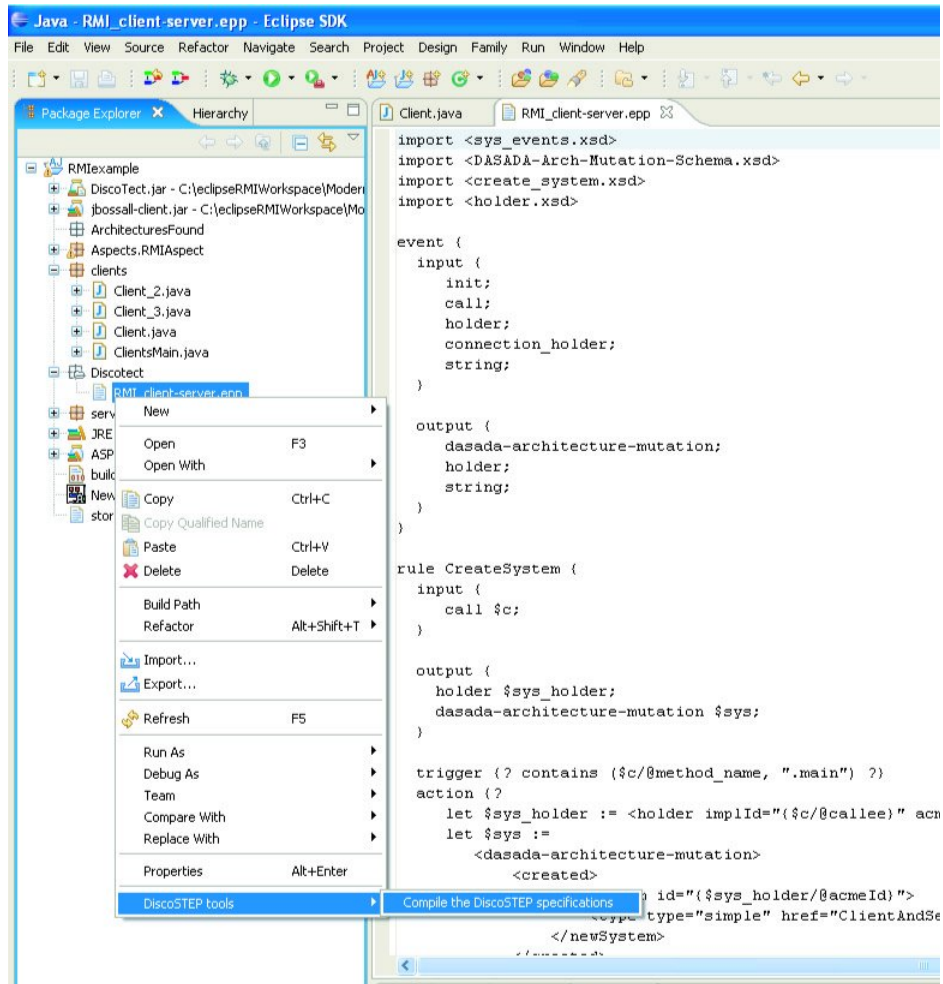


Figure 31. Screenshot of the compilation action.

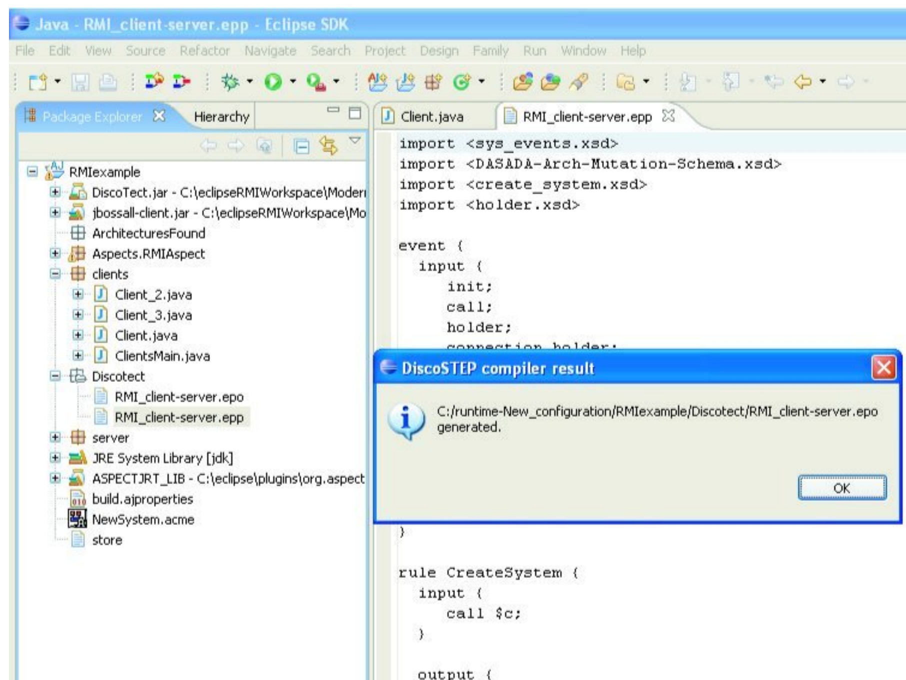


Figure 32. Message dialog after the successful compilation of a DiscoSTEP specification (note the .epo file, which is the newly compiled specification).

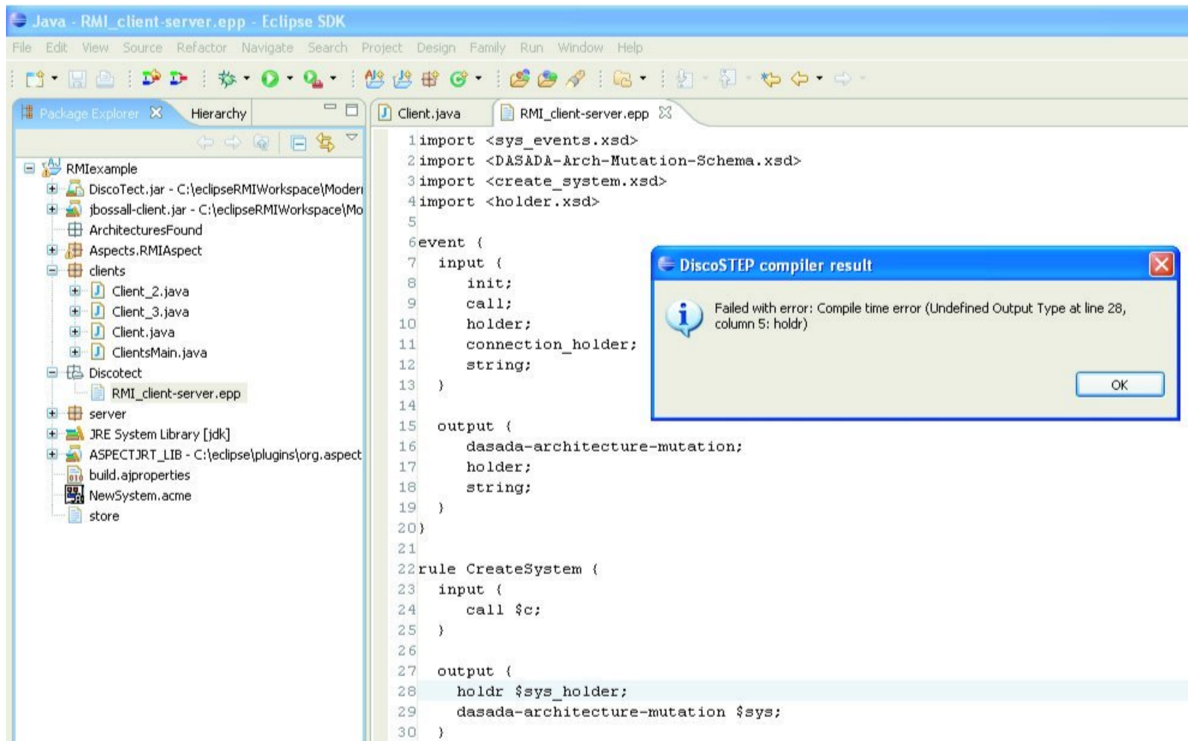


Figure 33. Message dialog after the showing the errors encountered while compiling a DiscoSTEP specification.

Figure 34. Message dialog after showing the available DiscoSTEP specifications that can be run.

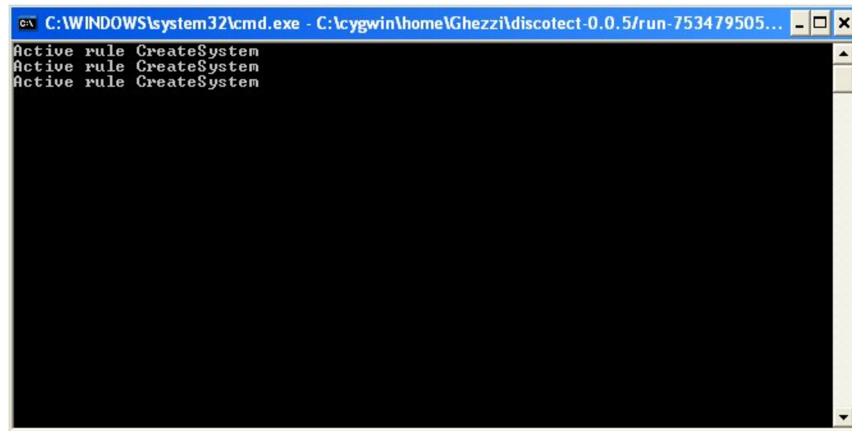


Figure 35. An instance of the DiscoTect Runtime Engine running a DiscoSTEP specification.

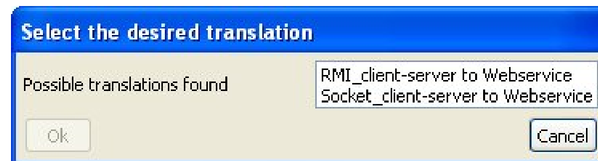


Figure 36. The possible transformations for a studied example application.

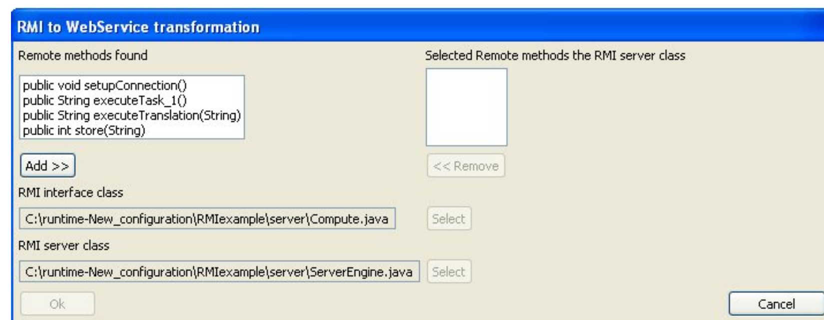


Figure 37. The RMI to Webservice transformation window for the RMI example explained in Section 5.3.2.

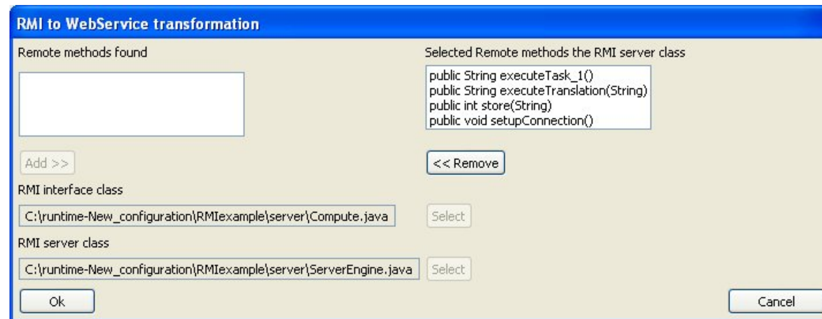


Figure 38. An RMI to Webservice transformation window ready for the transformation.

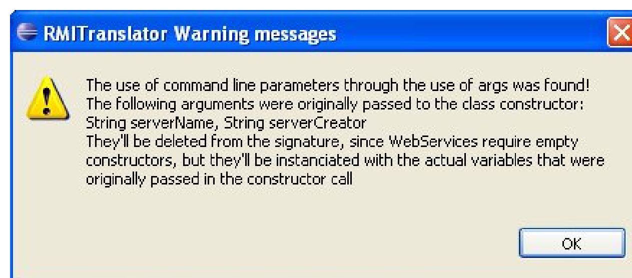


Figure 39. Warning message at the end of the transformation process.

```java
@WebService()
public class ServerEngine  implements Compute{

    private String name;
    private String creator;

/** TRANSLATOR IMPORTANT MESSAGE: You should carefully check the main and modify it.
 * In particular because the original solution used parameters passed throught command line input
 * at invocation (which is impossibile for WebServices) **/
public ServerEngine(){
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            String servName = "MyRmi Server";
            String servCreator = args[0];

            engine.setupConnection();
        }
        catch (Exception e) {
            System.err.println("ServerEngine exception: " + e.getMessage());
            e.printStackTrace();
        }

    serverCreator = servCreator;
    serverName = servName;
        super();
        this.name = serverName;
        this.creator = serverCreator;
    }

    @WebMethod
    public String executeTask_1(){
        return "Hello, from " + this.name + ". Created by: " + this.creator;
    }

    @WebMethod
    public String executeTranslation(String input){
        return StringProcesser.translate(input);
    }

    @WebMethod
    public int store(String input){
        return FileIO.write(input, "store");

    }
```

Figure 40. A translated RMI application.

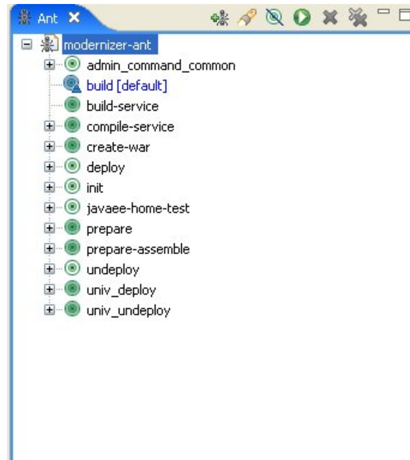


Figure 41. Ant tasks for a newly created Web service.

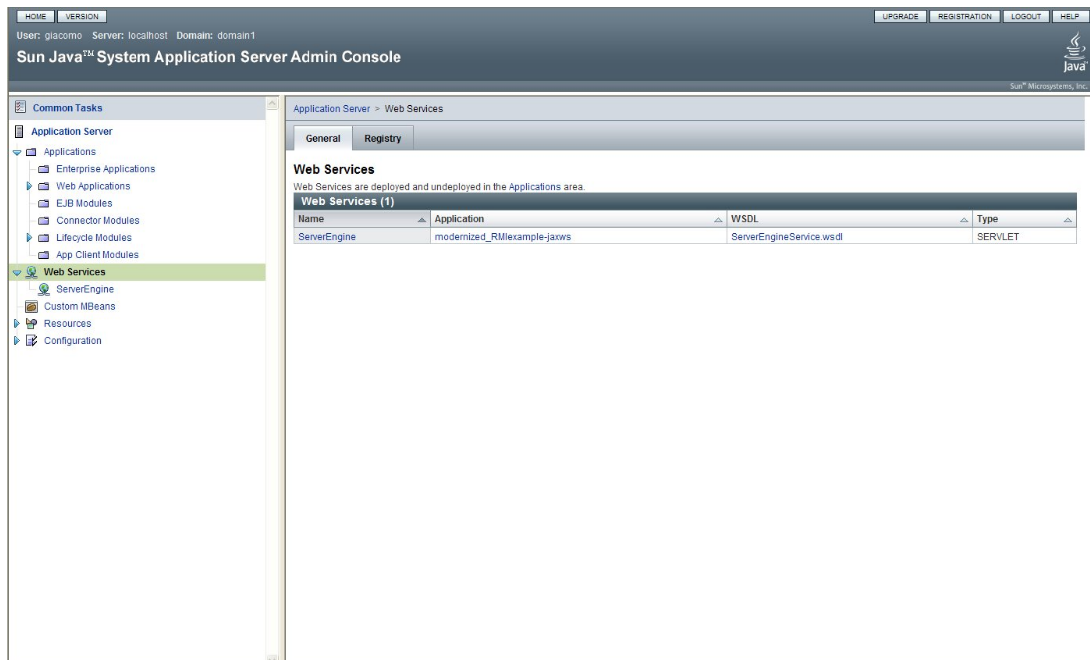


Figure 42. A deployed Web service.

Figure 43. A deployed Web service.

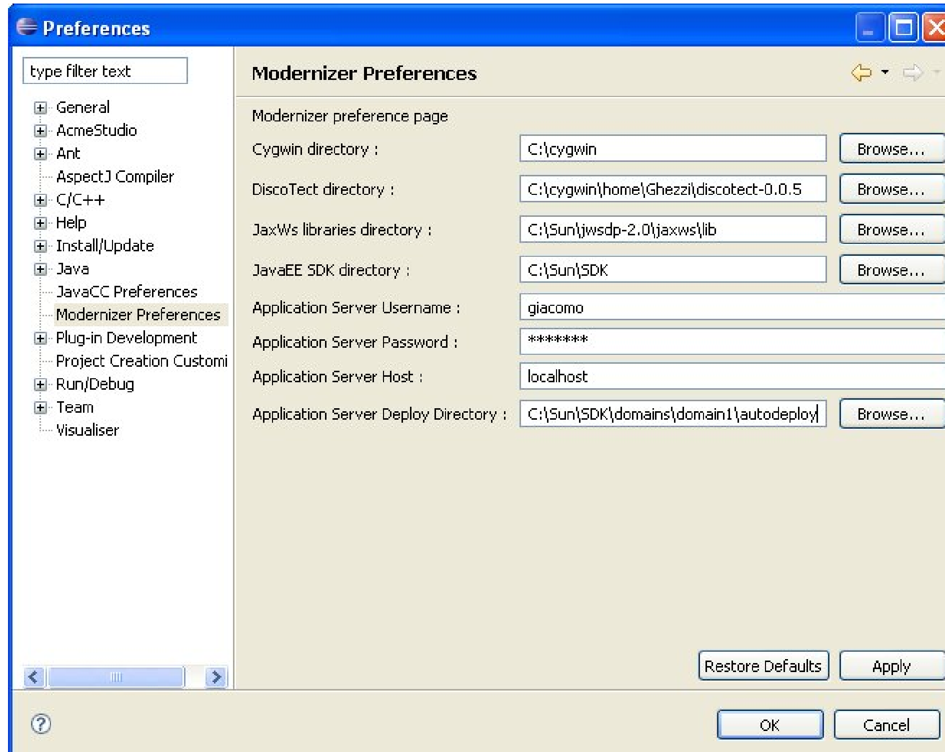


Figure 44. The Modernizer preference page, with values set for a specific computer.

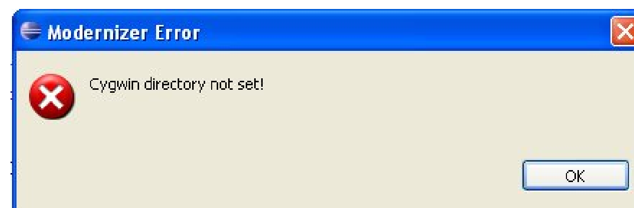


Figure 45. Error message of the compiler due to the missing Cygwin directory.

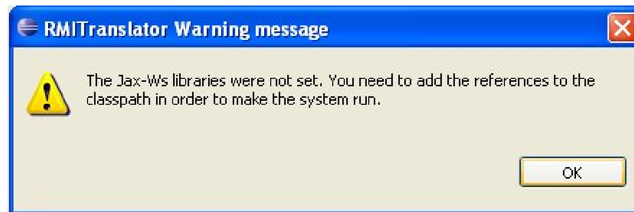


Figure 46. Warning message at the end of the RMI to Webservice transformation process due to the missing JaxWs libraries directory.

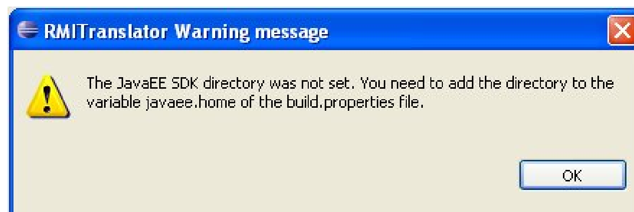


Figure 47. Warning message at the end of the RMI to Webservice transformation process due to the missing JavaEE SDK directory.

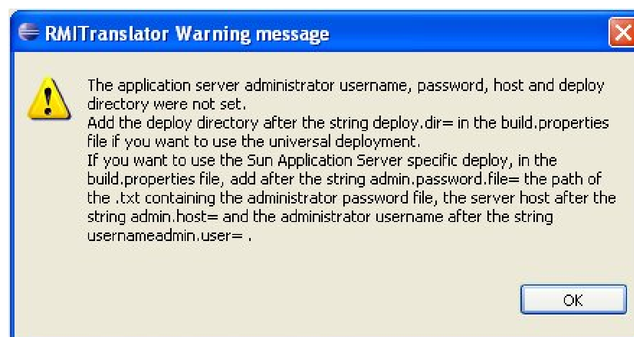


Figure 48. Warning message at the end of the RMI to Webservice transformation process due to the missing application server deploy directory or one between the application server username, password or host fields.

CHAPTER 7

CONCLUSIONS

In this thesis I first illustrated the problem of software modernization and the main techniques adopted for it. Then I described the basis for any modernization action, which is architecture analysis. I described the two main approachs, static and dynamic architecture analysis and for both of them I gave a view of the current state of the art.
After this introduction I illustrated and explained the general framework I developed, which is structured in two steps: identification of the architectural style implemented by the applicaction and transformation into a target architecture.
In particular, I described the specific transformation implemented by the current prototype, which addresses the service-based modernization of Java client-server applications implemented via RMI.

The tool, called Modernizer, and the overall solution I came up with is still a prototype. So it is obviously far from being considered a complete tool but it can be a good starting point for future developments. In fact, it addresses the issue of automatic transformation of Java applications into Web services driven by the analysis of their runtime architecture and behavior.
The approach described in this thesis can be compared with other research efforts. For example, the DMS tool (7; 8) uses static code analysis techniques but does not offer specific architecture-to-architecture transformations. Solutions offering legacy system to Web service transformation,

as the ones proposed by Sneed et al. (4) and by Canfora et al. (6), use black-box modernization techniques (in particular, wrapping). The work by Sneed et al. (4) also uses an old Java Webservice implementation (as explained in Chapter 4 that with Java 6, the implementation of a Web service has been greatly simplified). Other works, as the one by G. Lewis et al. (112) are more theoretical and offer only a series of guidelines to follow when migrating to a Service Oriented architecture.

To the best of my knowledge, however, an approach of the kind described here, based on dynamic analysis, has not been studied before.
In particular, my work showed that it is possible to semi-automatically transform an application (following known coding paradigms/structures) to a Web service offering the same functionalities and deploy it, by using the application's architecture and some other useful information both extracted by monitoring the application's runtime behavior.

To transform the RMI examples into Web services I first had to learn how to use DiscoSTEP and how to develop the necessary probes (see Sections 5.2.1 and 5.1.1).
My vision is that a normal user the Modernizer tool would not have to do that. In fact, I envision a large repository of ready-made DiscoSTEP specifications and probe aspects, offered by the tool, that the user can choose from when modernizing its application. The user should not need to know how to use DiscoSTEP, how to write probe aspects, or AspectJ. The user should just choose from a list the architectures to test, until one is found that is actually implemented by the application. The user then starts the desired transformation that then proceeds semi-automatically using the information extracted from the execution of the application.

7.1 Open issues

The approach proposed has obviously several limitations. First of all it is not universal. In other words it works only with applications following commonly used or known structures, as already explained in Section 6.2.4. The transformation process for applications that do not follow commonly used paradigms might not work or just partially work, so manual transformation by the user might be needed. Another limitation is that as for now, the only transformation allowed is RMI to Web service. Other transformations, however, may be added to the library.
Other limitations are intrinsically due to the use of the dynamic analysis tool, DiscoTect, as I already explained in Sections 5.2.2, 5.1.2 and 4.1.1.1. In particular, with dynamic analysis, we can never be sure that future analysis might show a different architecture or the same one with different details. Unless dynamic analysis is integrated with some static technique, this is a risk that has always to be taken into consideration. An easy and straight forward solution to limit this problem is to run and monitor the system under study many times and then combine all the outcomes to extract a common representation from them.

As for now, my prototype extracts all the required information only after one single execution of the system, but there is nothing that prevents in the future to add to the tool the possibility to extract the required information after a number of executions of the system. In this case, an algorithm that, given all the information extracted from the single runs, extracts only the common parts would have to be implemented.

Finally, a possible criticism that might be raised against this work is that Java applications are rather new and thus do not have the need to be modernized, in particular with respect to old

legacy systems. It can be noticed, however, that some early Java applications can be considered as legacy (as already said in Section 2.1). Moreover, we should not wait for Java applications to become legacy to start studying their modernization and tools for it.

7.2 Future work

There are a number of future works around this the prototype that can be done.
First of all I want to use my tool for other case studies, maybe also a commercial one, then add the automatic transformation for the client side of the RMI application, which is rather easy since basically only the parts in which the remote methods are called have to be modified.
After that I would like my Modernizer tool to support also the service-based modernization of other important and widely used architecture paradigms, as for example socket based client-server applications, which has been partially already implemented (the DiscoSTEP specification and the probe aspect have already been written). Apart from the transformations that convert the application into Web services, I would also like to try to add other types of transformations.
I think that also ad hoc editors (for example, with syntax highlighting a auto-completion) for DiscoSTEP specifications might be useful. In fact I found that using only a basic text editor to write those specification is rather annoying and error prone.
Another interesting future work I already took into consideration, and already started to analyze and study the feasibility, is how to infer the causal sequence in which the newly created web methods and services were called in the original application, in order to automatically create BPEL (113) workflows combining and orchestration the newly created services.

Obviously many others addition can be done to my tool, but I presented the ones that are more inherent to its primary purpose.

CITED LITERATURE

1. Good, D.: Legacy transformation, 2002.

2. Ricca, F. and Tonella, P.: Software systems reengineering: limits and capabilities. Mondo Digitale, September 2006.

3. Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., and Yan, H.: Discovering architectures from running systems. IEEE Transactions on Software Engineering, 7(32), July 2006.

4. Sneed, H. M. and Sneed, S.: Creating web services from legacy host programs. Proceedings of the 5th IEEE International Workshop on Web Site Evolution, pages 59–65, 2003.

5. Sneed, H.: Integrating legacy software into a service oriented architecture. Proceedings of the 10th European Conference on Software Maintenance and Reengineering, March 2006.

6. Canfora, G., Fasolino, A., Frattolillo, G., and Tramontana, P.: Migrating interactive legacy systems to web services. Proceedings of the 10th European Conference on Software Maintenance and Reengineering, 2006.

7. Baxter, I., Pidgeon, C., and Mehlich, M.: Dms: Program transformations for practical scalable software evolution. Proceedings of the International Conference of Software Engineering, 2004.

8. Akers, R., Baxter, I., and Mehlich, M.: Re-engineering c++ component models via automatic program transformation. 12th Working Conference on Reverse Engineering, 2005.

9. Comella-Dorda, S., Seacord, R. C., Wallnau, K., and Robert, J.: A survey of black-box modernization approaches for information systems. Proceedings of International Conference on Software Maintenance, 2000.

10. Seacord, R., Plakosh, D., and Lewis, G.: Modernizing Legacy Systems. Addison-Wesley Professional, 2003.

11. Lucca, G. D.: Legacy systems, 2003.

12. Bisbal, J., Lawless, D., Wu, B., and Grimson, J.: Legacy information system migration: A brief review of problems, solutions and research issues. IEEE Software, (16):103–111, 1999.

13. Bennet, K.: Legacy systems: Coping with success. IEEE Software, (12):19–23, 1995.

14. Reiss, S.: Incremental maintenance of software artifacts. IEEE Transactions on Software Engineering, 32(9), September 2006.

15. Lehman, M. M. and Belady, L.: Program evolution: Processes of software change. 1985.

16. Sommerville, I.: Software engineering. Addison Wesley, 2000.

17. Kaplan and Norton: Measuring the strategic readiness of intangible assets, February 2004.

18. Weiderman, N., Bergey, J., Smith, D., and Tilley, S.: Approaches to legacy system evolution (cmu/sei-97-tr-014). 1997.

19. Plakosh, D., Hissam, S., and Wallnau, K.: Into the black box: A case study in obtaining visibility into commercial software (cmu/sei-99-tn-010). 1999.

20. Chikofsky, E. and II, J. C.: Reverse engineering and design recovery: A taxonomy. pages 13–17, 1990.

21. Gerber, A., Glynn, E., and MacDonald, A.: Modelling for knowledge discovery.

22. Ulrich, W.: Modelling for knowledge discovery. `www.cis.uab.edu/EDOC-MELS/Papers/Ulrich.pdf`.

23. Dalton, M.: Architecture-driven modernization (adm): Knowledge discovery metamodel (kdm) specification. `http://www.omg.org/docs/ptc/06-06-07.pdf`.

24. Bisbal, J., Lawless, D., Wu, B., Grimson, J., Wade, V., Richardson, R., and O'Sullivan, D.: An overview of legacy information system migration. Proceedings of the 4th Asian-Pacific Software Engineering and International Computer Science Conference, March 1997.

25. Bovenzi, D., Canfora, G., and Fasolino, A.: Enabling legacy system accessibility by web heterogeneous clients. Proceedings of the 7th European Conference on Software Maintenance and Reengineering, pages 73–81, 2003.

26. Basili, V., Briand, L., and Melo, W.: How reuse influences productivity in object-oriented systems. Communications of the ACM, 39(10):104–116, October 1996.

27. Shaw, M.: Architecture issues in software reuse: It's not just the functionality, it's the packaging. Proceedings IEEE, Symposium on Software Reusability, April 1995.

28. Zaidman, A., Adams, B., Schutter, K. D., Demeyer, S., Hoffman, G., and Ruyck, B. D.: Regaining lost knowledge through dynamic analysis and aspect orientation, an industrial experience report. Proceedings of the Conference on Software Maintenance and Reengineering, 2006.

29. van den Heuvel, W.: Matching and adaptation: Core techniques for mda-(adm)-driven integration of new business applications with wrapped legacy systems, 2004.

30. Meena, J. and Piyush, M.: Reusing code for modernization of legacy systems. Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice, 2005.

31. Aldrich, J., Chambers, C., and Notkin, D.: Archjava: Connecting software architecture to implementation. Proceedings of the 24th International Conference on Software Engineering, 2002.

32. Abi-Antoun, M. and Coelho, W.: A case study in incremental architecture-based reengineering of a legacy application. Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, 2005.

33. Shaw, M., Deline, R., Klein, D., Ross, T., Young, D., and Zelesnik, G.: Abstractions for software architecture and tools to support them. IEEE Transactions on Software Engineering, 21(4), April 1995.

34. Group, O. M.: Uml 2.0 superstructure specification, final adopted specification, 2003.

35. Borland: Together, 2003. `http://www.borland.com/together`.

36. Poseidon, G.: , "`http://www.gentleware.com`.

37. ArgoUML, T.: , `http://argouml.tigris.org/`.

38. UModel, A.: , `http://www.altova.com/products_umodel.html`.

39. Fujaba: , `http://www.fujaba.de/`.

40. Ideogramic UML, D. E.: , `http://www.ideogramic.com/products/uml/product-info.html`.

41. Kollmann, R., P.Selonen, Stroulia, E., Systa, T., and Zundorf, A.: A study on the current state of the art in tool-supported uml-based static reverse engineering. Proceedings of the IEEE Working Conference on Reverse Engineering, 2002.

42. Tonella, P. and Potrich, A.: Reverse engineering of the interaction diagrams from c++ code. Proceedings of the International Conference on Software Maintenance, 2003.

43. Kollmann, R. and Gogolla, M.: Capturing dynamic program behavior with uml collaboration diagrams. Proceedings of the IEEE European Conference on Software Maintenance and Reengineering, 2001.

44. Rountev, A. and Connel, B.: Object naming analysis for reverse-engineered sequence diagrams. Proceedings of the IEEE International Conference on Software Engineering, 2005.

45. Rountev, A., Volgin, O., and Reddoch, M.: Static control-flow analysis for reverse engineered sequence diagrams. Proceedings of the ACM Workshop on Program Analysis for Software Tools, 2005.

46. Gosling, J., Joy, B., Steele, G., and Bracha, G.: Java Language Specification. Addison-Wesley, third edition.

47. Murphy, G., Notkin, D., and Sullivan, K.: Software reflexion models: Bridging the gap between source and high-level models. Proceedings of SIGSOFT '95 Washington DC, USA, 1995.

48. Kazman, R. and Carrière, S. J.: Playing detective: Reconstructing software architecture from available evidence. Automated Software Engineering, 1999.

49. Murphy, G. and Notkin, D.: Lightweight lexical source model extraction. ACM Transactions on Software Engineering and Methodology, 5(3), 1996.

50. Corporation, I.: , `http://www.imagix.com`.

51. Wall, L. and Schwartz, R.: Programming Perl. O'Reilly and Associates, 1991.

52. Stonebraker, M., Rowe, L., and Hirohama, M.: The implementation of postgres. IEEE Transactions on Knowledge and Data Engineering, 2(1), 1990.

53. Kazman, R. and Burth, M.: Assessing architectural complexity.

54. Schmerl, B. and Garlan, D.: Acmestudio: Supporting style-centered architecture development (demonstration description). Proceedings of the 26th Int'l Conference on Software Engineering, May 2004.

55. Briand, L., Labiche, Y., and Leduc, J.: Tracing distributed systems executions using aspectj. Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005.

56. Briand, L., Labiche, Y., and Leduc, J.: Toward the reverse engineering of uml sequence diagrams for distributed java software. IEEE Transactions on Software Engineering, 32(9), 2006.

57. Garlan, D., Kompanek, A., and Cheng, S.: Reconciling the needs of architectural description with object modeling notations. Science of Computer Programming, 44(1), July 2002.

58. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., R. Little, R. N., and Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison Wesley, 2002.

59. Schutz, W.: The Testability of Distributed Real-Time Systems. Kluwer Academic, 1993.

60. RedHat: Jboss, a division of redhat. `http://www.jboss.com/`.

61. et al., T. E.: Special issue on aspect-oriented programming. Communications of the ACM, 44(10), October 2001.

62. Team, A. D.: Aspectj project. `http://www.eclipse.org/aspectj/`.

63. Kiczales, G., Hilsdale, E., Huginin, J., Kersten, M., Palm, J., and Griswold, W.: Getting started with aspectj. Communications of the ACM, 4(10), October 2001.

64. Wikipedia, t. F. E.: Aspect-oriented programming. http://en.wikipedia.org/wiki/Aspect-oriented_programming.

65. Adams, B. and Tourwe, T.: Aspect orientation for c: Express yourself. 3rd Software-Engineering Properties of Languages and Aspect Technologies Workshop, AOSD, 2005.

66. Spinczyk, O., Gal, A., and Schroder-Preikschat, W.: Aspect c++: An aspect-oriented extension to the c++ programming language. Proceedings of the 40th International Conference Tolls Pacific: Objects for Internet, Mobile and Embedded Applications, 10, 2002.

67. Lammel, R. and Schutter, K.: What does aspect-oriented programming mean to cobol? AOSD '05, 2005.

68. Committee, A.-O. S. D. S.: Aspect-oriented software development. http://aosd.net/.

69. Microsystems, S.: Java virtual machine profiler interface (jvmpi). http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html.

70. Fenlason, J.: Gnu gprof, http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html.

71. Inspector, P.: Jprof (java profiler). http://perfinsp.sourceforge.net/jprof.html.

72. Levon, J., Elie, P., Jones, D., Montgomery, B., Cohen, W., Hoare, G., Hueffner, F., Mwaikambo, Z., Purdie, R., and Baechle, R.: Oprofile - a system profiler for linux. http://oprofile.sourceforge.net/.

73. Developers, V.: Valgrind. http://valgrind.kde.org/.

74. Seesing, A. and Orso, A.: Insectj: A generic instrumentation framework for collecting dynamic information within eclipse. Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange, 2005.

75. Cole, A.: Aprobe: A non-intrusive framework for software instrumentation. www.ocsystems.com.

76. Systa, T., Yu, P., and Muller, H.: Analyzing java software by combining metrics and program visualization. Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR'2000), 2000.

77. Walker, R., Murphy, G., Freeman-Benson, B., Wright, D., Swanson, D., and Isaak, J.: Visualizing dynamic software system information through high-level models. Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1998.

78. Reiss, S.: Jive: Visualizing java in action (demonstration description). Proc. 25th Int'l Conf. Software Eng., 2003.

79. Zeller, A.: Animating data structures in ddd. Proc. SIGCSE/SIGCUE Program Visualization Workshop, 2000.

80. Pauw, W. D., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J., and Yang, J.: Visualizing the execution of java programs. Revised Lectures on Software Visualization, International Seminar, 2001.

81. Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., and Yan, H.: Discovering architectures from running systems appendix a. IEEE Transactions on Software Engineering, 7(32), July 2006.

82. Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., and Yan, H.: Discovering architectures from running systems appendix b. IEEE Transactions on Software Engineering, 7(32), July 2006.

83. Jensen, K.: A brief introduction to coloured petri nets. Proceeding of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS) Workshop, 1997.

84. Garlan, D., Monroe, R., and Wile, D.: Acme: Architectural description of component-based systems. Foundations of Component-Based Systems, 2000.

85. Madhav, N.: Testing ada 95 programs for conformance to rapide architectures. Proc. Reliable Software Technologies-Ada Europe '96, 1996.

86. Systa, T.: Understanding the behavior of java programs. Working Conference on Reverse Engineering, 2000.

87. Muller, H., Wong, K., and Tilley, S.: Understanding software systems using reverse engineering technology. 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS), 1994.

88. Koskimies, K., Mannisto, T., Systa, T., and Tuomi, J.: Automated support for modeling oo software. IEEE Software, 1998.

89. OMG: Mof 2.0 / xmi mapping specification, v2.1. `http://www.omg.org/technology/documents/formal/xmi.htm`.

90. Skublics, S., Klimas, E., Thomas, D., and Pugh, J.: Smalltalk With Style. Prentice Hall, 2002.

91. alphaWorks: Jikes bytecode toolkit. `http://www.alphaworks.ibm.com/tech/jikesbt`.

92. Ball, T.: The concept of dynami analysis. ESEC / SIGSOFT FSE, 1999.

93. Beizer, B.: Software Testing Techniques. Van Nostrand Reinhold, 1990.

94. IBM: Eclipse - an open development platform. `http://www.eclipse.org/`.

95. java.net The Source for Java Technology Collaboration: Java compiler compiler [tm] (javacc [tm]) - the java parser generator. `https://javacc.dev.java.net/`.

96. W3C: , , 2007. `http://www.w3.org/TR/xquery/`.

97. Kay, M.: Saxon, the xslt and xquery processor. `http://saxon.sourceforge.net/`.

98. W3C: , , 2007. `http://www.w3.org/XML/`.

99. Microsystems, S.: Java message service (jms). `http://java.sun.com/products/jms/`.

100. A.Colyer, Clement, A., Harley, G., and Webster, M.: Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. Addison Wesley Professional, 2004.

101. IBM: Web tools platform (wtp) project. `http://www.eclipse.org/webtools/main.php`.

102. Apache ant. `http://ant.apache.org`.

103. Microsystems, S.: Sun java system application server platform edition 9.0. http://www.sun.com/software/products/appsrvr_pe/.

104. Microsystems, S.: Java api for xml web services (jax-ws). http://java.sun.com/webservices/jaxws/index.jsp.

105. Microsystems, S.: Java api for xml-based rpc (jax-rpc). http://java.sun.com/webservices/jaxrpc/overview.html.

106. W3C: Simple object access protocol (soap) 1.2. http://www.w3.org/TR/soap/.

107. Fielding, R. T.: Architectural styles and the design of network-based software architectures, doctoral dissertation, 2000.

108. Microsystems, S.: The java web services tutorial for java web services developer's pack, v2.0, 2006.

109. Deitel, H., Deitel, P., and Associates: Java, How to Program. Prentice Hall, 7 edition, 2006.

110. Foundation, T. E.: Eclipse java development tools (jdt). http://www.eclipse.org/jdt/.

111. Foundation, T. E.: Aspectj development tools (ajdt). http://www.eclipse.org/ajdt/.

112. Lewis, G., Morris, E., O'Brien, L., Smith, D., and Wrage, L.: Smart: The service-oriented migration and reuse technique (cmu/sei-2005-tn-029). 2005.

113. BEA, IBM, Microsoft, AG, S., and Systems, S.: Business process execution language for web services, version 1.1.

VITA

Personal data

- Full name: Giacomo Ghezzi
- Place and date of birth: Italy. January 25th, 1982
- Nationality: Italian
- Address: Piazza Guardi 15, 20133 Milano (MI), Italy
- E-mail address(es): gghezz2@uic.edu - giacomoghezzi@fastwebnet.it

Undergraduate and graduate studies.

- Bachelor's degree: Politecnico di Milano, 2004.
- Master's degree: Politecnico di Milano, 2007 - University of Illinois at Chicago, 2007.