Integration of Programming Environments for Platform Migration

Dissertation

der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Frank Gerhardt
aus Stuttgart

Tübingen
2003

Tag der mündlichen Qualifikation: 29. Januar 2003
Dekan: Prof. Dr. Ulrich Güntzer
1. Berichterstatter: Prof. Dr. Herbert Klaeren
2. Berichterstatter: Prof. Dr. Wolfgang Rosenstiel

Abstract

This dissertation is about tool and process support for migrating applications from one platform to another, specifically from Smalltalk to Java.
Both platforms consist of a programming language, obviously, but also include large class libraries, development tools, and a runtime environment. Despite many high-level similarities, the class libraries are quite different. They provide similar functionality but in different ways. Many differences are so fundamental that migration tools like syntax converters and type analyzers produce imperfect results. The output of these migration tools generally requires a significant amount of manual rework. To perform the rework, a developer will have to use the development environments of both platforms: on the source platform to analyze the code, and on the target platform to fix problems. Furthermore, because he needs to rebuild the application on the target platform from scratch, the application is not executable as a whole. Without being able to run automated tests against the application, it is difficult to stabilize the migrated code.
I propose the *integration* of the two programming environments. This integration supports a more productive *piecemeal* migration process. The environment integration combines both the development and the runtime environments. Instead of working with two code bases, a developer can work on a single integrated hybrid system and perform the migration by replacing original code with migrated code in a single system. The advantage of this approach is that the whole system is executable at all times. Many techniques requiring an executable application can be used to improve the productivity of the migration process, e. g. automated functional and unit testing, inspections of the system at runtime, dynamic type analyses, use of assertions, profiling, etc.
In essence, this thesis aims at making software migrations more agile.

Zusammenfassung

Diese Dissertation beschäftigt sich mit der Werkzeugunterstützung der Migration von Anwendungen von einer Plattform zu einer anderen, speziell von Smalltalk nach Java.
Beide Plattformen bestehen nicht nur aus einer Programmiersprache, sondern enthalten auch eine große Klassenbibliothek, Entwicklungswerkzeuge und eine Laufzeitumgebung. Trotz vieler Gemeinsamkeiten weisen insbesondere die Klassenbibliotheken erhebliche Unterschiede auf. Sie stellen zwar annähernd die gleiche Funktionalität zur Verfügung, aber auf unterschiedliche Weise. Viele Unterschiede sind so grundlegend, dass Migrationswerkzeuge wie Syntax-Konverter und Typ-Analysatoren keine perfekten Ergebnisse erzielen. Der automatisch migrierte Code erfordert meist erhebliche Nacharbeit. Ein Entwickler wird dafür die Entwicklungsumgebungen beider Plattformen verwenden: um den ursprünglichen Code zu verstehen, und um Fehler im migrierten Code zu beheben. Außerdem muss er die Anwendung auf der Zielplattform von Grund auf neu erstellen, so dass sie über lange Zeit nicht als Ganzes ausführbar ist. Ohne die Möglichkeit automatisierte Tests einsetzen zu können, fällt es schwer den migrierten Code zu stablisieren.
Ich schlage die *Integration* der Programmierumgebungen beider Plattformen vor. Diese Integration unterstützt einen schrittweisen Migrationsprozess. Die Integration kombiniert sowohl die Entwicklungs- als auch die Laufzeitumgebungen. Statt mit zwei getrennten Code-Basen arbeiten zu müssen, kann ein Entwickler mit einem integrierten, hybriden System arbeiten und die Migration durchführen, indem er Teile des ursprünglichen Codes durch migriertem Code in diesem hybriden System ersetzt. Der Vorteil dieses Ansatzes ist, dass das Gesamtsystem immer ausführbar ist. Viele Techniken, die ein ausfühbares Programm erfordern, können dadurch angewendet werden, um die Produktivität des Migrationsprozesses zu verbessern: automatisierte Funktions- und Einheitentests, Inspektionen des Systems zur Laufzeit, dynamische Typanalyse, Verwendung von Zusicherungen, Laufzeituntersuchungen usw.
Das Anliegen dieser Arbeit ist Software-Migration agiler zu machen.

Contents

Chapter 1

Introduction

This dissertation is about *tool support for migrating applications* from one platform to another.

This introduction narrows the scope from the larger context of migration (1.1) to the processes, paths, and tool support for migration (1.2). When this stage is set, I expose the main line of thought: the research question and the thesis (1.3). I continue with a short overview of the proposed solution and its application (1.4). Finally, I outline the structure of the dissertation (1.5).

1.1 Larger Context

Platform migration has to be seen in the larger context of platform diversity (1.1.1). This diversity creates many difficulties for organizations that build many applications over a longer period of time (1.1.2). If the burden of the technology mix in an organization increases, it is desirable to migrate older systems to newer platforms (1.1.3).

1.1.1 Diversity of Platforms

The number of programming languages is huge. New languages are invented for specific domains, existing languages evolve over time, new features are added that were found useful.

Even though only a fraction of all programming languages are in widespread use, the number of popular languages is still big. Examples include C++, Smalltalk, VisualBasic, Java, C#, Perl, Python, etc.

Each of these popular languages comes with reusable code—usually in the form of a class library or a framework—and with tools, e. g. an integrated development environment (IDE). Often also a runtime system is included, e. g. a virtual machine or dynamic link libraries. For this combination of language, libraries, tools, and runtime system I use the terms platform *and* programming environment synonymously.[1] I will address the tools of a platform in the next section (1.2).

[1] More definitions can be found in section 3.1 on page 33.

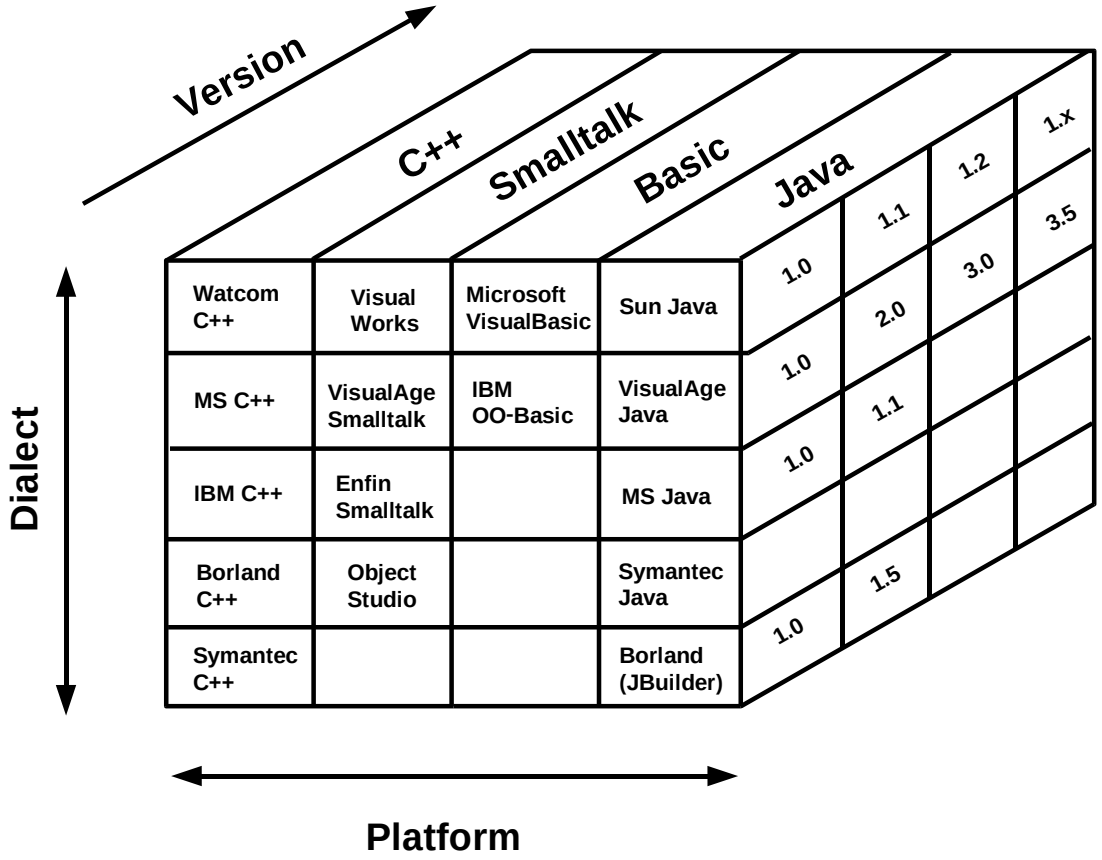

Figure 1.1: The dimensions of platform diversity

Often a popular language is implemented by different vendors. For example, there are a number of C++ and Smalltalk implementations on the market. There are standards, e. g. ANSI and ECMA for these languages, and vendors of implementations try to be compatible with them. But looking at the whole breadth of a platform, only a narrow part is usually covered by a standard. Platforms differ in specific class libraries, e. g. for graphical user interfaces. This is not accidental—it is the domain where the vendors compete and differentiate their offerings.

Figure 1.1 shows three dimensions of platform diversity: the dimension of the different languages, the dimension of the dialects of the languages, and the dimension of the releases of implementations over time.

Obviously, languages and dialects can be quite different. But even different versions of a platform can be different enough to require a migration of existing code. If a vendor releases a new version of a platform which is not backwards compatible to the previous version, then the upgrade can sometimes be as difficult as a migration between different dialects.

A specific platform has to be chosen when a system is built. This corresponds to a point in the three-dimensional space: language, dialect, and version need to be defined. For example a project might chose Microsoft Visual C++ 6.0 (on Windows 2000 SP2).

Such a decision is often a commitment to a whole slice of the cube, i. e. a sequence of

versions, because the comparative advantage of switching to another dialect or language is usually not high enough to justify the effort. If an organization maintains an application over many years it will have to keep it up-to-date with new releases of the platform.

There is a large number of possibilities (points in the three dimensions). When discussing the migration of code from one platform to another below, I will refer frequently to the problem of having so many different source and target platforms.

The advantage of this diversity is that there are many possibilities to choose the right tool for a specific task. The diversity is so large that there are often numerous possibilities even for specific problem domains. This does not create a problem if unrelated projects select different platforms. However, the diversity of platforms causes problem in a corporate environment.

1.1.2 Corporate Environment

Large corporations build many systems over time. They usually have an IT strategy which defines the platforms to use. The motivation for defining an IT strategy is to reduce the number of platforms used within an organization. However, for various reasons the number of platforms will grow over time. There are changes of the IT strategy to include new software technology. There are systems brought into the IT infrastructure by business reorganizations and acquisitions. And there are seemingly unimportant systems using a non-standard platform which grow from a departmental scope to corporate scope.

After many projects have been rolled-out, companies will find that their IT infrastructure consists of several different platforms and resembles the Tower of Babel. They will also find that their organization has built different cultures around each technology, e. g. the mainframe versus the Unix versus the Windows developers. These cultures also resemble the Tower of Babel.

A company needs to maintain a platform if it has systems in operation which use that platform. While the vendor of the platform maintains the technical implementation, a company that uses this platform has to maintain the organizational environment for its successful use:

- For operations, the runtime environment (and possibly the operating system) and tools for operation (e. g. monitoring) need to be available.
- For software maintenance and enhancements, the development tools need to be available. Besides the IDE this will probably include secondary tools like a repository, testing tools, etc.
- The personnel who has the skill to use these tools has to be available.

Having to maintain a platform can be quite a burden. An organization's IT strategy would ideally prescribe only a single platform. Although this is desirable, an IT strategy has to account for different types of projects. Limiting the number to just a few platforms instead of dozens or hundreds is already a success.

1.1.3 Why Migrate?

The choice of a development platform has been done for good reasons: at the time when the decision was made, it was the best choice for a specific problem. What can happen is—over time—that a platform is not the best choice any more and that one wants to migrate one or more applications to a new platform.

The motivation for a migration can lie in issues with the platform itself, or in organizational issues like the organizational “fit” of a platform. These organizational issues apply both at the corporate or departmental level and on the level of an individual developer or a team of developers.

Problems Related to the Platform

The problems with a platform (or product) can be non-technical or technical. The non-technical problems are related to the vendor of a platform and its competitors in the market. The technical problems are limitations of the platform itself.

The major non-technical problem is competition. Science progresses and new platforms emerge. One reason might simply be that there are better alternatives on the market in the meantime. For example, Java got a lot of attention recently and has improved its market acceptance at the cost of other platforms like VisualBasic, Smalltalk, and C++.

Platforms have a life cycle on their own. They compete with each other in the market for acceptance and this competition creates winners and losers. New platforms can cause old platforms to “die”.

Other non-technical problems are related to the vendor of a platform, not to the platform itself. Bad management, bad corporate strategy and other business reasons can get a vendor into trouble. In the end, those difficulties will result in technical problems as products will not be maintained and enhanced as expected. One example for this situation is the disappearance of VisualSmalltalk from the market after the merger of its vendor with another Smalltalk vendor.

The number of technical problems can get quite large. Anyone who has seen a criteria catalog for a product evaluation knows how many aspects can to be looked at. A technical problem can occur in any one of those criteria.

Usually the vendor maintains a platform and fixes these technical problems. The technical problems mentioned here are all problems that *could* get fixed but for some reason are not. Sometimes the customer can help himself to work around a deficiency but in many cases it would require too much effort.

Technical problems are often perceived relative to a—possibly–new competitor. The original platform lacks features compared to newer alternatives. As this is probably true in the beginning, the vendor of the original platform should be expected to keep its product up-to-date. An example is VisualWorks Smalltalk. It lacked COM support and CICS connectivity. Also it seemed more appropriate for client-server architectures than for web application development (applets). If a vendor does not commit to supporting new technologies it can be a major motivation to migrate away from such a platform.

Problems Related to the Organization

To describe the organizational problems I refer to the view that software is knowledge [Armour, 2000]. Software is executable, active knowledge. To only build a simple application one needs to know a vast multitude of details ranging from operating systems, databases, user interfaces, languages, toolkits/frameworks, programming tools, design and architecture concepts. In addition to that one needs knowledge about system operations (installation, monitoring, security, fail-over, load-balancing). Furthermore, knowledge about the problem domain and its processes is necessary. This can be overwhelming.

In maintenance the situation is even more complicated because in order to maintain an application one needs to know the details about the specific versions of each component involved.

The diversity of platforms and its organizational consequences cause of number of problems for large organizations. In the end these problems are all about money, i. e. the *total cost of ownership* (TCO).

These problems fall into three categories: the cost of having a platform, the additional cost for having a possibly outdated platform, and the cost of having an *additional* platform.

First, a platform has to be maintained in operations and in development. This includes license costs, resources for operation, maintenance of the development environment, and the cost for the human resources to make use of the platform.

Second, a platform might not any more provide all the features needed. Then missing features need to be developed in-house. This increases the cost of an old platform compared to another, newer platform that has these features already—possibly for free. The example of the availability of runtime environments on new platforms was given above.

Third, because an old platform is one platform *more* it adds to the overall heterogeneity of the infrastructure. This increases the cost for systems integration and skill distribution. There are solutions for dealing with this heterogeneity such as wrapping old systems or Enterprise Architecture Integration (EAI) but all these solutions cost money and—if overused—might create problems themselves, e. g. a diversity of middle-ware products.

A company is often viewed as a learning organization. From this point of view, the heterogeneity fragments an organization's knowledge. Mainframe, client/server, and web developers form isolated cultures both in operations and development.

But also for a single developer an old platform can be a burden. In software maintenance it is necessary to retain knowledge about old platforms. This includes the language, the libraries, the tools, and many other details. A developer who works also on new systems and uses newer versions of the same platform or completely different platforms will find it hard to maintain the competency on an old platform. First, it is challenging enough to know one platform well. Knowing more than one platform well is even more challenging. Second, the similarities between platforms can be quite confusing and it can be hard to keep the mental models of each platform separate. This is bit like learning French and Italian at the same time.

For these reasons—at some point in time—it becomes desirable to abandon, to "sun-down", a platform. Although, a business case has to be made. All costs have to be quantified and the two options of keeping or abandoning a platform need to be compared.

It is often very hard to make this business case because of the cost and the risk. The *exit cost* to abandon a development platform is extremely high.[2] Also the risks involved are hard to assess.

To achieve the full benefit of abandoning a platform *all* systems using this platform would have to be migrated, i. e. an even higher cost and higher risk. The migration of only a single system is sub-optimal. Of course, it is “easier” to never change the running systems and pay the additional price. Although, in some cases the price is too high, or the missed opportunities of using up-to-date technology are too big. Considering the cost and the risk, applications are rarely migrated—only if it is totally unavoidable.

1.2 Migration Processes and Tool Support

To perform a migration one obviously needs to know what to do. Besides knowing how to migrate the program elements from the source platform to the target platform it is necessary to have an overall process and tools that support the working style of the process (1.2.1). This support is crucial because of the associated cost and risk of a migration as described in the previous section.

However, this tool support is often missing, and a developer has to work with existing tools side-by-side (1.2.2). Especially compared to forward engineering this is unsatisfying (1.2.3).

1.2.1 Migrations Paths and Processes

A software system usually has a layered architecture consisting of hardware, operating system, libraries, programming language.

A *migration path* describes from which layer on the source platform to which layer on the target platform an application will be migrated. The three basic categories of migration paths are migrating to the same layer, to a higher layer, and to a lower layer. I will argue later that a migration at the source code level is the most valuable because only this path results in maintainable code. Another example for migrating at the same layer is to port a virtual machine or a compiler to another operating system. An example for migrating to a lower layer is cross-compilation, and an example for migrating to a higher layer is de-compilation of machine code into portable C code.

There are a number of processes—like in forward engineering—that can be used to perform a migration, e. g. the waterfall, spiral, or an iterative model. A big difference compared to forward engineering is that the migration process has to deal mostly with the development phase since all requirements are fixed. The testing and deployment (roll-out) phases are the same as in other process.

Another difference is that a migration starts with an executable application on the source platform. On the target platform one has to build the same application from scratch. The

[2] For example, at DaimlerChrysler for *one* system the migration was estimated to cost several millions of Dollars.

point is that during the migration the old system can be analyzed and inspected. On the target platform, however, one starts with a minimally functional program that can be hard to compare with the original system and hard to test.

A migration process has not only to describe the phases of the process but also give the process direction. This means to give some recommendation for a developer where in the system to start migrating, how to proceed through the system, and where to finish.

A forward engineering project might create at first user interface prototypes, then add application logic, then add a data access layer and so on. A similar approach can be used to migrate a system: first the user interface, then business objects, then infrastructure. The disadvantage of this approach is that one does not have a system throughout the process that can be tested end-to-end. At first, only the user interface can be tested, later application logic, and only when the migration is finished it is possible to the system end-to-end. I call this the layer-by-layer approach.

Another option is to select some functionality, e. g. a use case, and then migrate the corresponding vertical slice through the architecture which implements the selected functionality. This approach has the advantage that the selected functionality can be compared with the original system. Automated test cases can be used to demonstrate the functional equivalence of both systems.

Tool support is crucial to make this process viable. A developer will have the development environments of the source and the target platform available. These environments are designed for forward engineering but the tasks of a developer during a migration are different.

1.2.2 Working with Two Independent IDEs

Without additional support a developer will have to fall back on the two integrated development environments (IDEs) of the source and target platform. Let's assume that he also uses migration tools, e. g. a syntax converter. These tools are useful but generally produce imperfect results—the generated code will usually not work without manual rework. Let's assume that the developer chooses the approach sketched above to migrate a vertical slice of the source system for some selected functionality.

The developer will first identify the functionality to be migrated. This is a good moment to start writing test cases which capture this functionality. Later these test cases can be used to check if the migrated system has the same behavior as the original system. In the second step the developer identifies the code he needs to work on in the source IDE. He then converts this code to the target platform. He might convert only the part that he needs in that moment, or all code and then only pick the part relevant for the selected functionality. As described above, the converted code will not work immediately. The developer fixes the code in the target IDE to make the test cases work. Whenever something is unclear he switches back to the source IDE to find out what the code should be doing.

This process is not supported by the tools very well. The frequent switching between the IDEs is tedious because they are not integrated in any way. Furthermore, the developer has to work with two different applications, one on the source platform and another one on the target platform. He has to study how the application of the source platform works in order to fix problems on the target platform.

1.2.3 Contrasting with Forward Engineering

Compared to migrating using two IDEs, the tool support in forward engineering is much better.[3] Many elements of modern processes would not be possible without appropriate tool support. Early processes followed the waterfall model. Contemporary compilers were slow, and on big projects a recompilation of a project could take hours or days. It was much later when incremental compilers and interpreted scripting languages supported iterative development processes. Another example is programming in the debugger. This is a technique related to test-first design as advocated e. g. by eXtreme Programming [Beck, 1999].[4][5]

The point to be made here is that a highly productive process requires sophisticated tool support.

1.3 Research Question and Thesis

So far I introduced the topic of migrating applications from one platform to another. I presented some alternative approaches and argued that a productive migration process needs better tool support. The tool support should be better than working with two separate IDEs. The hallmark is set by the process support of modern IDEs for forward engineering.

1.3.1 Problem: Awkward Process Because of Missing Integration

When a developer migrates an application using two IDEs (1.2.2) the main hindrance is that he cannot work on original and migrated code in an integrated way. This problem has two aspects: the IDEs are not integrated with each other, and the code bases are separate. The precise statement of the problem can be found in section 2.5 on page 29.

My idea for a solution is to provide an environment for migration which supports the migration process at the same level as modern IDEs support agile processes. Because implementing a specialized IDE to support migration from platform A to B would be too costly, the migration environment should be a combination of the two available IDEs and allow a working style as if it were one environment. The code bases of both IDEs should be integrated to that a developer should only deal with one set of code—in different languages however.

[3] Forward engineering is the act of developing an application (analysis, design, implementation) and is distinguished from re- and reverse engineering [Chikofsky and Cross, 1990].

[4] A developer first writes a test case and compiles it. Technically, a test case is a subclass of `TestCase` and the individual tests are implemented as methods of this subclass. Since this is the first step the test case will call methods that have not been implemented yet. Nevertheless the test case can be compiled and executed. This obviously results in an error and the debugger presents the halted program. The developer can fill in the implementation for the missing method and resume the execution.

[5] In personal communication, a developer of T-Systems estimated that more than 50% of the code is written in the debugger.

1.3.2 Question: How to Integrate Programming Environments?

This leads to my research question: *How can I easily integrate two programming environments to improve the migration process?*

Any potential solution has to account for the constraints of the domain. Since budgets of migration projects are usually tight, a solution should be easy to implement—in-house, or by a specialized tool vendor. I describe the constraints in detail in section 4.2 on page 63.

1.3.3 Thesis

I propose an integration of programming environments that allows a developer to work with two programming environments in tandem and migrate an application piece by piece. During the migration the developer works with a hybrid system consisting of original and migrated code. The environment integration uses a message passing architecture which is simple to implement and effective. This integration supports a productive migration process.

The exact thesis statement can be found in section 4.1 on page 61.

1.4 Environment Integration and Piecemeal Migration

This section gives a short overview how my proposed integration of programming environments works, and how this supports my *piecemeal* migration process. It also mentions the key contributions made in this dissertation. The detailed description of my solution can be found in chapters 5 and 6.

1.4.1 Environment Integration

The two programming environments of the source and target platform are connected via message passing [Reiss, 1990]. Reiss used statically compiled tools and a fixed message format. Since modern environments for Smalltalk and Java are scriptable I base the integration mechanism on sending *scripts* from one environment to another. This makes the definition of a message format unnecessary and the integration more flexible.

To each environment a *Connector* is added which communicates with the Connector of the other environment. Messages are routed through a central message server as in Reiss' original work.

The *Integrator* is connected to both Connectors and acts both as the message server and as a tool to invoke migration-specific actions, e. g. syntax conversion of some source text. The Integrator uses the message passing infrastructure as well.

A developer works on a piece of an application at a time. Using the Connectors and the Integrator he can convert the source code for this piece and transfer it to the other IDE. The migrated version of the current piece is dynamically connected with the original application using proxies generated by the Integrator. The proxies communicate with their subjects using the same message passing infrastructure as the IDEs.

The new aspect of this solution is that I integrate environments that are integrated within themselves already and that I use scripting inside the environments to achieve high degree of flexibility. The architecture integrates both the development and the runtime environments of both platforms.

1.4.2 Piecemeal Migration

The piecemeal migration process is iterative. I call it *piecemeal* to differentiate it from other iterative processes without support of the integrated environments—the piecemeal migration process is more productive because of the environment integration. The iterations should be as short as possible, i. e. from five minutes to one day. This is in line with other agile methodologies, e. g. eXtreme Programming.

The process emphasizes testing. Often it is considered too much effort to build test cases for a whole system. In my opinion there is no way to demonstrate that the migrated system has the same behavior as the source system other than tests. The tests are *functional* tests, not unit tests. This reduces their number considerably. I also refer to the concept of a test covering [Feathers, 2002], see section 6.2.2.1 on page 96. These are tests with the special purpose to introduce an invariant in the system. They allow for modification of *covered* classes without writing individual tests for each of the covered classes.

The iterations are organized by feature, or use case. For each feature a vertical slice through the layered system architecture is migrated. First, a test is written for this feature. The test has to succeed when applied to the source system. Then the integrated IDEs are used to migrate the relevant code to the target platform. Finally, the developer fixes the migrated code so that the test passes when run against the target system. Then the next iteration starts.

The new aspect of this process is that concepts of other agile methodologies such as short iterations and rigorous testing are applied to platform migration. This is possible because the environment integration provides the necessary tool support for this working style.

1.5 Overview

This dissertation is structured in four levels of abstraction. The two levels in the “middle” are the most important ones. Since this research is about integration of programming environments—to support platform migration—these two important layers are the ones on *integration* (3 and 5) and the *migration process* (2 and 6):

1. Chapters 1 and 7 frame this work at a general level.
2. Chapters 2 and 6 focus on the relationship of migration, migration processes and tool support.
3. Chapters 3 and 5 present the technical details of tool and environment integration.
4. Chapter 4 at the heart of the document states my thesis.

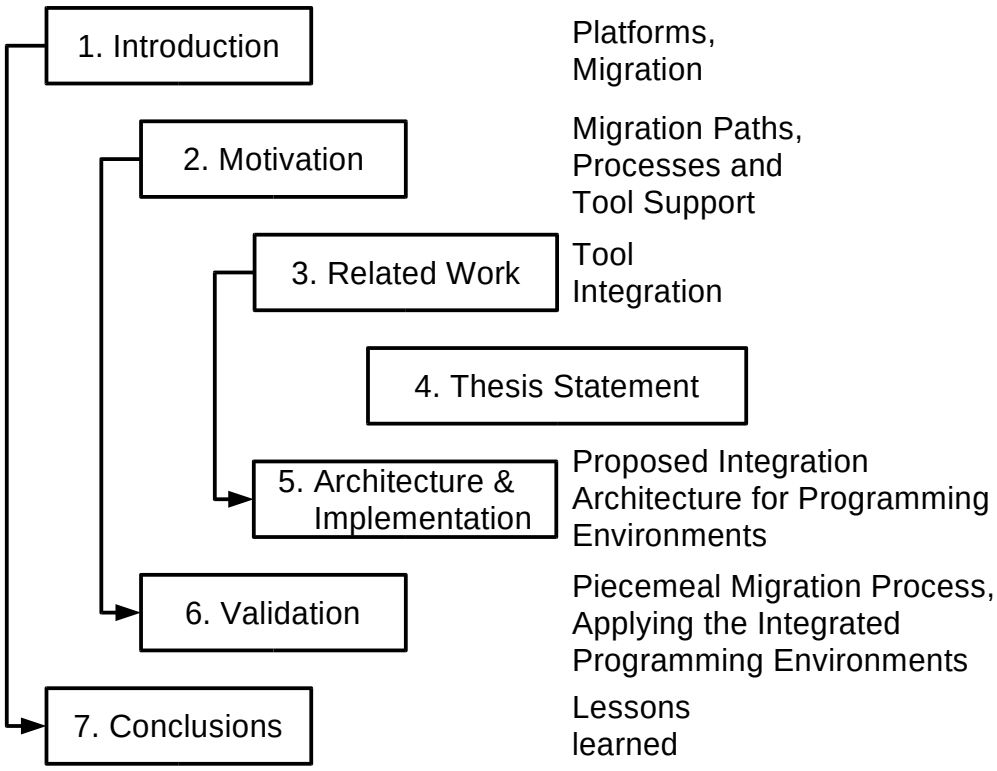

Figure 1.2: Dissertation structure

Chapter 2 defines what a platform migration is. This chapter relates the migration process to its tool support. Specialized tools are missing and one often has to use the programming environments of the source and target platforms in an *unintegrated* way. This leads to my research question: “How can good low-cost tool support for platform migration be provided?” In more detail: “How can the existing programming environments of the source and target platform be integrated to support an efficient migration process?”

Chapter 3 goes into the specifics of tool integration and looks for mechanisms that could be employed for environment integration. Tool integration and environment integration are at two different levels because programming environments are integrated per se while tools are not. I conclude that the existing mechanisms cannot be scaled up to the integration of programming environments.

Chapter 4 formulates my thesis.

Chapter 5 presents the proposed solution: an integration architecture for programming environments. This is on the same level of detail as chapter 3 on tool integration.

Chapter 6 is the validation of my proposed integration architecture for programming environments. Here I demonstrate how to apply integrated programming environments to a migration. It discusses the migration process at the same level of detail as chapter 2.

Chapter 7 wraps up with conclusions.

To get a quick overview of this work I suggest to read the *thesis statement* (4), the short introductions to the main chapters (2, 3, 5, 6), and the conclusion (7). These short introductions will expose the line of thought through the whole dissertation.

Chapter 2

Migration, Processes, and Tool Support

This chapter presents the relationship between the difficulties of a migration (2.1), different options for migration paths (2.2), the processes for migrations (2.3), and the tool support for these migration processes (2.4). I identify the problem of limited process support (2.5) and contrast this limited support with that found in forward engineering. This leads me to formulate a vision of an improved working model for migration (2.5.2). This vision is a first step towards my thesis (4) and my solution (5 and 6).

2.1 Migration

Migration is the act of transferring an application from one platform to another so that the migrated application exhibits exactly the same behavior as before. A platform can be understood as the next lower level of an application. Platforms include hardware, operating systems, databases, window frameworks, programming languages, virtual machines, etc.

The term *porting* is frequently used to describe a migration from one operating system to another (“the Linux port of a program”). When referring to the operating system I will use the term porting. In any other case I will use the term migration, e. g. as in data migration.

Applications which require little or no effort to port are *portable*. If the effort is zero, they are *platform-independent*.

The platforms I consider in this research are two programming languages: Smalltalk and Java. Both are not just mere languages but include a set of class libraries which offer almost every functionality usually found in operating systems. Therefore they can be viewed as full-scale platforms.

It is important to note that porting Smalltalk or Java applications between operating systems is not an issue. Both languages are compiled into a platform-independent bytecode which is executed by a virtual machine. Virtual machines exist for all major operating systems. An application can be “ported” simply by running it on the respective virtual machine for a chosen operating system. Migrating a Smalltalk application to Java implies abandoning the Smalltalk virtual machine and using the Java virtual machine instead. Since the Smalltalk class library depends very much on the virtual machine, a migration to Java also implies using the classes of the Java Development Kit instead of the Smalltalk framework.

2.1.1 Similarities between Smalltalk and Java

Smalltalk and Java have many similarities at the conceptual level. Both share a common mind-set about what an application should look like. Despite many similarities, the differences are far more significant (2.1.2).

The most obvious similarity is that both platforms are fully object-oriented. Each platform consists—among other things—of a language definition and a class library. Classes can be related by single inheritance and the class hierarchy is rooted in `Object`.

Both platforms compile source code into operating-system neutral bytecode. The compiler and the interpreter (virtual machine, VM) are included. Both languages have automatic memory management (garbage collection).

The class libraries of both platforms provide roughly the same functionality. Many features which are part of the language in Java, e. g. threads or inheritance, are implemented in the class library in Smalltalk. To an application, however, it does not matter where a feature is defined as long as it is available.

Both class libraries are huge. The Smalltalk framework consists of thousands of classes and the current Java Development Kit (JDK 1.4) consists of more than 500.000 lines of code (excluding classes where no source is provided). Since both class libraries were developed independently of each other, it is obvious that there is no common naming convention for similar functionality. Classes and methods with the same names are accidental. Based on this observation one could compile a huge list of differences. But naming is not the problem as I will explain in the next section.

2.1.2 Differences between Smalltalk and Java

Differences between Smalltalk and Java exist at all levels. The virtual machines are different and use different bytecodes. The languages and the base classes are different. Also the programming environments are different. I will cover the Smalltalk programming environment in more detail later in section 3.5.3.4 on page 57. This section discusses the differences between the two architectures, languages, and class libraries.

The two platforms have been compared by various authors [Kapp, 1995, Bothner, 1996, Boyd, 1997, Wege, 1998, Boyd, 2000, Eßer et al., 2001]. I limit the presentation here to the most problematic issues for a migration.

It should be noted that there is no single Smalltalk and all comparisons refer to Smalltalk in general as described in [Goldberg and Robson, 1983] or to a specific implementation. My focus is the VisualWorks implementation. The Smalltalk dialects share a common architecture. There are slight differences in the language definitions, e. g. namespaces, and huge differences in the base classes and tools. I will not discuss how to deal with specific differences between base classes because there are too many. Instead, I discuss how to deal with base class incompatibilities in general.

2.1.2.1 Architectural Differences

The main difference between the Smalltalk and the Java platform is that Smalltalk is an *image*-based environment and Java works with files. The Smalltalk image is a memory snapshot of the environment. When the Smalltalk environment is started the virtual machine (VM) loads the image and presents the environment in the state as it was saved. When the Java VM is started, it does not contain any application classes. Classes are loaded into the Java VM by a class-loader starting with the entry point for an application which is passed to the VM as a startup parameter.

The image-based architecture has a number of consequences. First, a Smalltalk application is generally deployed as an image. The image contains all of an application's code and all objects of the Smalltalk environment that have not been stripped to save memory. *Stripping* is the act of removing the development tools from an image in order to obtain a smaller image that can be deployed to end users. The point is that the application depends on objects which are not part of the source code of the application or the base classes. These objects have been provided by the Smalltalk vendor or a developer. A migration has to take these non-code entities into account.[1]

Second, the meaning of a program depends on the image into which it is loaded. In Smalltalk everything is an object—see below—including the Smalltalk compiler. A piece of Smalltalk code is compiled by sending it to the compiler object in the image. The "output" of the compiler is an object that represents the compiled piece of code, e. g. an instance of the class `CompiledMethod`. The point is that the output of the compiler and thus the meaning of a piece of code depends on the contents of the image. Different images can produce different results. The meaning of a Smalltalk program can not be determined statically without referring to an image.

Third, any object in the image can be changed. This can obviously be expected from a programming environment. The possibility to change objects also includes the objects that represent the base classes. In Smalltalk, a developer can change base classes and modifications are immediately reflected in the image. For example, to check whether a string conforms to the format of an ISBN, a developer can *add* a method `isISBN` to the class `String` of the class library. These additions (or modifications) below the level of a class are called *extensions* in Smalltalk. In Java it is impossible to modify base classes, or have compilations units smaller than a class or interface. In the given example the developer would have to add a subclass of `String` where he could add his method. For a migration this means that a Smalltalk application can not be packaged separately from the base classes. It is rather "woven" into the base classes.

Fourth, all Smalltalk development tools are part of the image—I mentioned the compiler already. Every tool stores its data in the image as well. e. g. the VisualWorks GUI Painter stores textual descriptions of user interface (window specs) in class methods. The problem for a migration is that this data has to be considered being a part of an application, and it needs to be migrated as well. However, the specific data structures of these tools are like domain specific languages that are beyond the scope of migrating Smalltalk applications to Java. The fact that tool-specific data ties developers into an IDE is a problem that occurs

[1] The developers of the Squeak Smalltalk implementation have reported that the object structure of the Squeak image cannot be (re-)created from a textual description. The base image has become an "atomic" building block of a distribution.

in many other development environments as well. In Smalltalk it can be considered an advantage that the format of these data structures is available.

2.1.2.2 Language Differences

The platform comparisons mentioned above include extensive lists of differences at the language level. I summarize the differences described in [GEBIT, 2001].

Types Smalltalk is dynamically typed while Java is statically typed.

Primitive values versus objects In Smalltalk everything is an object including characters and integers. Java has primitive types (e. g. `int`) and object types (e. g. `Integer`).

Blocks Blocks group statements in Smalltalk. Furthermore, blocks are objects and the contained statements are only evaluated if the `value` message is sent to a block. In Java $\{\}$ is used only for syntactic grouping. Most uses of blocks can be mapped to Java, but advanced uses require a redesign of the code.

Class methods versus static methods and constructors Smalltalk classes are objects and class methods use the same method dispatch as instance methods. Java does not have a complete meta-model implementation. Instances are created in Smalltalk by sending the message `new` to the class object, while in Java `new` is a reserved word of the language.

Global variables and pool dictionaries Pool dictionaries are shared namespaces for variables. Both are not available on the Java platform but can be emulated.

nil `nil` is an object in Smalltalk, an instance of the class `UndefinedObject`. `nil` can respond to message sends, and `UndefinedObject` can be sub-classed. In Java `null` is just a special literal value. If `nil` is used in other ways than as a literal value, then the relevant part of an application needs to be redesigned, e. g. by emulating the class `UndefinedObject` in Java.

Non-convertible features The Smalltalk methods `become:` for swapping object references and `doesNotUnderstand:` for catching undefined messages have no equivalent in Java and can hardly be emulated.

New features Java has a number of new features which Smalltalk does not have. e. g. Smalltalk uses message passing for exception handling while Java has exception handling as part of the language. In Java a method signature declares the exceptions a method will throw. Callers have to catch or rethrow these exceptions. Since Smalltalk code does not make the use of exceptions explicit in a similar way, Java-style exception handling has to be added to the migrated code, i. e. a sensible use of `throws` declarations and `try-catch` blocks.

For many of these differences it is easy to find a rule describing how to migrate a piece of Smalltalk code. If Smalltalk code uses blocks sensibly then most of them can be converted automatically. However, using blocks as continuations makes a migration more difficult. Several solutions exist for type analysis but they all fail at programs which are larger than

toy examples. Instead, a technique called type prediction can be used to find types by analyzing instances in the image and extracting information from variable names.

Java has many new language features that are not present in Smalltalk, e. g. packages, access modifiers, interfaces, inner classes, etc. Most of them are not a problem for a migration from Smalltalk to Java. With respect to these new features, one could say Java is upwards-compatible with Smalltalk. In some cases—such as exceptions—it is desirable not to emulate the Smalltalk style of exception handling but to use the Java exception handling mechanism in order to comply with established Java conventions.

2.1.2.3 Class Library Differences

The differences between the class libraries of the two platforms are the most significant differences. Both because of the size of the libraries and the different implementation approaches. The size of the libraries leads to a great many of little difficulties that—in most cases—cannot be handled automatically by a conversion tool. The different implementation approaches use different design patterns and make different assumptions about the use of base classes.

First of all, the meaning of a Smalltalk program is not clear (2.1.2.1). The situation is complicated further because both platforms support implementing methods as native code. Native code can be used for application development but it is more frequently found in the base classes. Smalltalk *primitives* are methods which are implemented natively in the virtual machine. The class browser shows only one line of source code for such a primitive: a call statement together with a number identifying the primitive. Such a primitive can have side-effects which cannot be inferred because the source for the native implementation is usually hidden. The implementation of the VM is one of the most hidden secrets of Smalltalk vendors. For a migration this means that one has to deal with base classes for which one has only a fuzzy understanding based on the naming of a class and its methods, comments, and other documentation. This imprecise notion prohibits the use of formal program transformations.

Although an ANSI Standard exists for Smalltalk, it does not help much. First, it covers only a fraction of the class library. This core is not sufficient to write real-world applications. Second, its description of the functionality is informal. E. g. [Cook, 1992] created a formal specification of the collection classes in Smalltalk. To allow for formal transformations a specification like this covering *all* of the class library would be necessary.

On the Java side the situation is similar. Through the Java Native Interface (JNI) it is possible to call native code. The base classes use this extensively—as in Smalltalk.

Another complication for a migration is the use different design patterns in the class library. Chris Wege and I studied the differences at this level in detail. The results are published in his diploma thesis [Wege, 1998].

At a lower level, the base classes use different conventions for indicating problems. Special return values and exceptions report problems and errors. Both platforms use different conventions, e. g. for accessing the elements of arrays.

Since both class libraries provide roughly the same functionality it is possible in many cases to find a class on the target platform that provides the same features. One can define a

mapping for those classes. Where different design patterns are used and the factoring of classes is different, this is problematic. For the reasons described in this section a mapping of class names is only a first cut. When looking at the method level one will find many incompatibilities and since it is impossible to develop formal transformation (which could be proven for correctness) a developer has to fix those problems manually. Given the size of the class libraries, this is an enormous amount of work.

2.2 Migration Paths from Smalltalk to Java

The potential migration paths fall into two major categories. As described in section 2.1.2.1 both platforms consist of a language with an associated class library and an execution environment. The two categories of migration paths focus either on the language or the execution environment.

It is useful to differentiate these kinds of migration paths by what one wants to retain, and what one wants to abandon. Figure 2.1 shows the five path described in this section. The colors show which part of the architecture will be replaced in a migrated system and which will be retained. The numbers refer to the respective subsections.

2.2.1 Universal Virtual Machine

This migration path—and the next—retain a maximum of the Smalltalk language and class library in the migrated system. One could even say this migration path is an attempt to avoid the migration as far as possible by including only very little of the Java platform in the target architecture.

A universal virtual machine (UVM) is a virtual machine which can execute both Smalltalk and Java bytecodes. The Smalltalk and Java VMs use bytecodes which are optimized for each language. The idea of the UVM approach is to let old Smalltalk systems coexist *without* any modifications with Java applications in their natural environment.

Since the Java virtual machine (JVM), as distributed by Sun Microsystems, does not support the execution of Smalltalk bytecodes, such a UVM has to be developed independently. Several implementations have been started: 1. The Smalltalk/X VM has been extended to support Java 1.1 bytecodes [Gittinger, 1999]. 2. IBM's VisualAge/Smalltalk VM can execute Java and has been used as the underlying Java VM in the VisualAge/Java product. 3. ParcPlace started to implement Java support for the VisualWorks VM (code-name Frost). The product has never been finished.

Each of these UVM has its own drawbacks. The Smalltalk/X VM is only useful if one has a Smalltalk/X application—which is rarely the case. Migrating from another Smalltalk dialect to Smalltalk/X would by a major undertaking. Furthermore, the Smalltalk/X VM supports only Java 1.1. The IBM VM was never released for public use outside of the VisualAge/Java product. The ParcPlace VM was never finished and the open-source version does not make any progress.

The benefits of the UVM approach are few compared to the disadvantages. The most significant benefit is that Smalltalk and Java code can coexist. A secondary benefit is that only

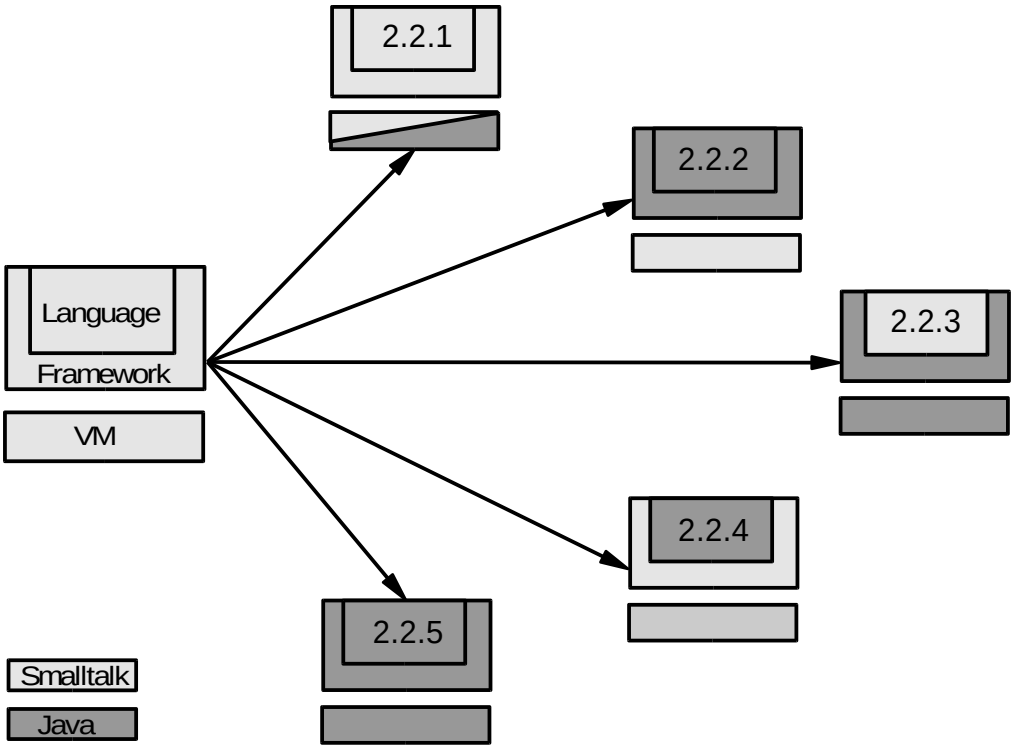

Figure 2.1: Migration Paths

one VM has to run to execute a Smalltalk and a Java application. This can save memory. The major disadvantage of the UVM approach is that such a custom VM is different—of course—from Sun's Java VM. This implies retesting of all applications on the UVM, and (re-)distributing the UVM. In many cases, these efforts have to be considered too high. But even if one accepts the disadvantages of using a UVM, for most projects such a UVM will not be available.

2.2.2 Cross-Compilation

The second migration path is to cross-compile Smalltalk into Java bytecodes. Theoretically, one could start the translation from Smalltalk bytecode to Java bytecode. Since the Smalltalk source code is usually available (or the bytecode can be de-compiled), it makes more sense to start from Smalltalk source code. One option would be to write a direct Smalltalk-to-JVM compiler. A simpler approach is to use Java as an intermediate language. This approach is taken e. g. by Talks2 [Architur, 2001] and Bistro (2.2.4). Cross-compilers have also been developed for many other languages such as Python, Ruby, etc.

The benefit of this approach is that one can keep a Smalltalk mind-set for software development. Generating Java from the Smalltalk source is only an automatic task related to the distribution of the software.

However, this approach requires that the complete Smalltalk class hierarchy is available on the Java VM. Migrating the whole library is a huge effort and—to the best of my knowledge—has not even been attempted by any vendor. Alternatively, one could write an isolation layer

that translates calls to the old Smalltalk interface to calls to the new Java class hierarchy. As with the first option, this would be a huge effort and I'm not aware of any initiatives to do that.

An alternative to emulation and isolation is to map Smalltalk classes to similar Java classes and restructure the application to use the Java classes, see section 2.2.5.

2.2.3 Syntax Translation

Syntax translation is similar to the previous migration path where Java was used as an intermediate language in the compilation process. The difference here is that the converted Java code is used for further development. Therefore, the quality requirements for the maintainability of the Java code are higher here. As a consequence of continuing the development in Java, the Smalltalk programming environment is abandoned after the conversion.

This migration path shares with the previous also the use of the Smalltalk base classes which are emulated on the Java VM. As described before, because of the difficulty to emulate the Smalltalk class hierarchy, this approach requires additional work. In section 2.2.5 I show how the isolated syntax conversion can be applied together with other approaches.

Two examples for syntax converters are ORISABase [Elsner, 1998] for VisualWorks Smalltalk and Gebit's MOVE/S2J for ObjectStudio Smalltalk [GEBIT, 2001]. Both tools have additional migration support beyond syntax conversion.

2.2.4 Hybrid Language

Depending on the goal of a migration it can be a "quick win" to use a hybrid language. A hybrid language is a language designed to be close to both Smalltalk and Java. The closeness to Smalltalk makes it easy to port code to the hybrid language, and the closeness to Java makes it easy to run on a JVM.

Figure 2.1 shows an interesting relationship between this and the previous approach. While the syntax conversion approach adopts the Java language but retains the Smalltalk class library, the hybrid language approach retains the Smalltalk language and adopts the Java class library.

SmallJava [Fussell, 1997b] is a hypothetical language to compare Smalltalk and Java. The author reviews the differences from the perspective of his hybrid language. The Bistro language [Boyd, 1997, Boyd, 1999, Boyd, 2000] is an implementation of such a hybrid language. The author admits that his goal was to embrace the Smalltalk syntax on the Java VM. Bistro works as a preprocessor to the Java compiler which generates Java bytecodes. The Bistro language introduces only very few language features to make the Smalltalk syntax compatible with Java, e. g. Bistro classes and methods are included in braces, and the compilation unit is always a file. Also the keywords `throws`, `try`, and `catch` are added so that the Java exception handling can be used.

The hybrid language approach requires Smalltalk to be migrated to the hybrid language. At that point, an application can be run on the JVM. If this is the primary goal of a migration,

then this approach is useful. If maintainable Java source code is desired, then a second migration would have to replace the Bistro syntax with Java syntax, i. e. one would have to remove the preprocessor from the compilation process. Like the two approaches described earlier, the hybrid language approach also suffers from the focus on the syntax. Difficulties of emulating or mapping the class libraries are not addressed.

2.2.5 Complete Migration

The complete migration path is different from all others in that it abandons *all* elements of the Smalltalk architecture (language, library, and VM) at once. It is the most desirable path if one wants to get rid of all traces of the original Smalltalk platform. This is the case especially if Java developers without a Smalltalk background will have to maintain the application.

Each of the migration paths described above focuses on a specific element of the original architecture shown in figure 2.1. A complete migration will use the other approaches where applicable. e. g. for converting the syntax of Smalltalk programs, the conversion described in section 2.2.3 can be used.

The previous migration paths all mentioned difficulties with migrating the class library. In a complete migration this problem has to be addressed as well. In section 2.2.3 I argued how hard it would be to emulate the whole Smalltalk class library or to develop an isolation layer. In a complete migration this problem can be partly resolved by a technique called *class mapping*.

Class mapping is similar to syntax conversion at a higher level. While syntax conversion works at the language level, the mapping between classes works on the classes of the class library. A framework like the Smalltalk class library can be understood as a higher-level language very much like a pattern language consisting of design patterns. Class mappings define a map from a Smalltalk class to a Java class with the same or similar functionality. The mapping includes also mappings for the method signatures.

The migration tools mentioned above, MOVE/S2J and ORISABase, both include an extensive set of mapping rules. In the case of ORISABase the mapping rules are applied using Prolog clauses and are more powerful than static rules used in MOVE/S2J. Since there are only few similarities in the class libraries (2.1.2.3) between Smalltalk and Java *and* between Smalltalk dialects, the set of mappings required is huge and dialect-specific.

Furthermore, applying these mappings is only an approximation towards a migrated system because Smalltalk class names are mapped to Java classes with *similar* but not with equivalent functionality. Wherever base classes cannot be mapped 100% to Java classes, the generated code will be incorrect. In these cases the application code has to be restructured to use the new classes. The major disadvantage of this migration path is that it is a lot of work.

The advantage of a complete migration is that it results in a Java application that adheres to the Java conventions and is maintainable. Other benefits are:

- The old platform can be abandoned completely since the migrated code bears no heritage from it. This eliminates the cost of ownership for the old platform.

- The number of platforms to operate and maintain is reduced by one. This allows an organization to concentrate its efforts on the (strategic) target platforms.
- Because there is one platform less and more focus on the (strategic) target platforms it is 1. easier to transfer knowledge and people between projects and 2. easier to reuse code between projects.
- The overall IT infrastructure is more flexible and easier to understand, the business can respond easier to change.

The maintainability is the key criteria for justifying the cost of a complete migration. Even though the cost for this migration path will be high, it is the only path I would recommend.

2.3 Migration Processes

A software process for migration is not very different from a process for forward engineering. The main concepts of phases and iterations apply to a migration project as well. However, some differences exist. The main difference is that a migration starts with an executable system. And like any other software process, a new system is delivered at the end.

A migration process has to deal with different activities and work products. Also some mechanisms which are built into processes to manage risks, will be different for a migration.

At first, it might seem that migration is mostly a coding activity. Phases like requirements analysis and design are not relevant. The requirements are already embodied in the existing application. The high-level design—be it 1-, 2-, or n-tier— should be preserved by a migration.

A second look reveals that phases like requirements analysis and design have counterparts in a migration process. Although the requirements for a migration are clear, a developer has to understand what the application is doing and how it is doing it. The corresponding activity is *program understanding*. Also, designing has its place in a migration process. However, not at the level of the overall architecture. Since the development platforms differ in many details, it is necessary to redesign parts of an application to comply with the architectural style imposed by the target platform. Testing is a phase which is relevant for a migration as much as for any other process. In many cases test-suites for existing systems are missing and need to be developed along with the migration.

A migration process also has to address slightly different risks. As mentioned before, the risk of uncertain requirements is rather low. The risk of technical problems with the target platform is the same as in forward engineering. The risk of failure (cancellation, time or budget overrun) is—in my opinion—higher than in forward engineering because of the difficulty of the task.

A process describes the overall path from the beginning of a project to the final delivery of a running system. Such a path can follow either a waterfall or an iterative model. For a migration project this difference boils down to the question: when to arrive at an executable system on the target platform? The options are to assemble the final system from individually migrated components at the end (2.3.1), or early in one of the first iterations of an iterative process (2.3.2).

2.3.1 Waterfall Process

In a waterfall process, all phases follow a logical order. Each phase has to be completed before the next phase can be started. For a migration it could look like this: a developer would convert the whole source code in one step, perform all type analyses next, then map all library classes, etc. If all goes well the developer would finally obtain an executable system.

The main advantage of this model is its simplicity. It is not necessary to worry much about what to do next because during each phase only the same kind of activity is done. The main disadvantage of this model is the inflexibility when problems occur. Fixing mistakes done in earlier phases requires back-tracking to that earlier phase, and following the sequence of phases from there again. A waterfall model works well if the error rate in earlier phases is minimal. As I will explain later, migration tools generally produce imperfect results so that frequent back-tracking has to be expected (2.4.1).

My major concern with the waterfall model is that integration and testing of individually migrated components happens only very late in the process. From the beginning on, a developer has to work with “broken” code. By that I mean working with code which can be compiled (and checked for compilation errors), but not executed as part of the whole application since the whole migrated application is available only at the end of the process. The point is that if the code cannot be run, it cannot be tested.[2]

As mentioned in the introduction, the risk of migration projects is high. Postponing integration and testing increases the risk even more.

2.3.2 Incremental Process

In an incremental process it is not required that a phase must be completed before the next phase can be started. Instead, work can continue in the next phase and remaining work of the previous phase will be addressed in subsequent iterations.

The main advantage of this process is that one can quickly work towards an executable skeleton of the target system. Once the initial target system is executable, it can be tested. Regression testing can demonstrate that modifications have not degraded the behavior of the system. The main disadvantage of iterative processes is that they require more planning and organization than the waterfall model. I believe that the risk reduction (see below) is well worth this additional effort.

An incremental migration process looks like this: A developer works in the first (or first few) iterations towards an executable skeleton of the system. This slice through the system should expose some end-user visible behavior for which a functional test can be written. Functional tests run outside of the system and can be applied to both the original and to the new system in order to compare the behavior of both systems and demonstrate functional equivalence. Once the initial skeleton has been reached, the iterations can be structured by units of end-user visible features (use cases, user stories, etc.).

The point is that executable code provides feedback which can be phased back into the process. For a developer there are a number of benefits:

[2] Components could be tested if unit tests were developed for each component. However, since resources for developing test cases are scarce it is better to focus on the development of functional tests.

- The application can be tested using an automated test-suite.
- Assertions can be used to test invariants.
- Dynamic data structures can be analyzed during execution (using a debugger).

These benefits strengthen the developer's confidence in the code, make searching for bugs considerably easier, and improve the code quality. Compared to the waterfall model, the execution of code will reveal bugs early—these bugs would be found in a waterfall mode only very late. All this together reduces the risk of failure.

2.4 Tool Support for Incremental Migration

In this section I present the tools that are readily available to a developer. For each tool I describe how it supports a source-to-source migration path (2.2) following an incremental process (2.3).

I split the set of tools into two major categories, the specific migration tools (2.4.1) and general purpose tools (2.4.2). I will conclude that a developer will have to use a mix of tools from both categories. This is unproductive because the tools are not integrated.

2.4.1 Specific Migration Tools

Before going into the details of the specific migration tools I describe the context in which such tool are developed (2.4.1.1). Then I discuss the available tools for the areas language conversion (2.4.1.2) and library conversion (2.4.1.3).

2.4.1.1 Migration Tools Market

Compared to the numerous offerings for forward engineering, e. g. Java tools, only a small number of specific migration tools are available on the market. This has two reasons.

First, specific migration tools are hard to build—harder than single-platform tools like IDEs. For a manual migration a developer has to have good knowledge about 1. the source platform, 2. the target platform, and 3. how to express the intent of the original code in terms of the target platform. Given the complexity of platforms like Smalltalk and Java, especially the class libraries, it takes at least several months if not years to master each platform. Developers with these skills are difficult to find. It is even more difficult to automate this task by encoding a developer's knowledge in a software tool.

Another reason for high development costs are the standards that modern IDEs set. Developers expect a similar standard from a migration tool as well. A simple command-line tool does not hold up to this standard. e. g. many of today's IDEs support incremental compilation. The implementation is not impossible, but would require a lot of effort. The consequence is that powerful migration tools are expensive to develop. The cost of developing an in-house migration tool is prohibitive.

Second, the opportunities to earn money with a migration tool are limited. Migration tools from Smalltalk to Java are always dialect-specific. Although there is only one Java, there are various Smalltalk dialects. All tools described in this section are not portable across different Smalltalk dialects. This limits the market for a specific migration tools to only a fraction of the Smalltalk users.

Let's imagine such a tool for a moment. Let's assume it migrates code *perfectly*. The strange thing is that the more the tool gets used, the more the market shrinks. Every investor would expect the sales of a successful tool to go up, but in the case of migration tools the market would vanish quickly. Thus, it is hard to develop a viable business model for the investment.

The consequence is that migration tools for a specific Smalltalk dialect are rare, and many tool vendors apply the 80–20–rule. They support 80% of the migration task with 20% of the effort. Usually, they address the problems at the language level but not the class library level. The remaining 20% are left to the developer. He would have to fix these problems manually.

2.4.1.2 Language Conversion

At the language level, migration tools have to address two tasks. Since Java is strongly typed the types of variables need to be inferred (2.4.1.2). The second task is the conversion of the Smalltalk syntax to Java syntax (2.4.1.2). [Eßer et al., 2001] provide a list of available tools and also include an evaluation.

Type Analysis Type analysis for Smalltalk is based on Agesen's type feedback approach [Agesen, 1995]. This algorithm has been implemented for two Smalltalk dialects. SUGAR is an implementation for Squeak Smalltalk [Garau, 2001]. [Li, 1998] implemented it for VisualWorks Smalltalk. Both implementations are academic prototypes—no commercial implementations exist. The type analysis is computationally intensive. The author of the VisualWorks implementation reported that for small applications the analysis requires at least 24 hours. For real-world applications this is not an option.

There are alternatives to Agesen's algorithm to achieve faster results. One method used by MOVE/S2J [GEBIT, 2001] is called *type prediction*. Types are deduced from variable and parameter names. This approach is a heuristic and can produce wrong results. However, if the Smalltalk code adheres consistently to naming conventions, e. g. as described in [Beck, 1997], the results can be useful. Another source for heuristic type information are class comments. They are a VisualWorks convention used and used throughout the supplied class library. However, many applications do not use class comments.

A second alternative is to use run-time data to analyze types. This can be done manually using the tools of the IDE (debugger, inspector), or automatically. The tools for automatic run-time analysis like profilers obtain information that can be useful. However, these tools are not designed for reengineering. Additional work on the output is necessary to make this data useful. Since the Smalltalk environment is open to modifications, exiting features like conditional breakpoints can easily be modified to log type information.

One should not conclude that knowing the precise types would solve a large part of the migration. It is true that one needs to know which variables reference e. g. integers and which

reference strings. Given the type of a variable, one can generate the Java declaration using the respective Smalltalk class name of the class implementing the type. The generated Java code will be syntactically correct. The problem here is that the class name refers to a Smalltalk class. In some cases there will be an equivalent Java class and the code will also be semantically correct. But in most cases this will not be true, see section 2.4.1.3 below. The type information is useful for domain classes which can be mapped to Java classes one-to-one.

The obtained type information can be stored in a separate file, or phased directly into the syntax conversion. The implementations mentioned above have a different focus than migration to Java. Therefore, the type information is stored in a file and has somehow to be phased into the migration process. If the syntax converter is not aware of this data, it can not use it. Because of this limited data integration, the migration process is not improved very much. I would consider type inference engines a building block for more sophisticated tools, but their stand-alone use is not beneficial to the migration process.

Syntax Conversion The mere syntax conversion is simple compared to the other tasks because the Smalltalk syntax is simple; the whole syntax can be summarized on a single page. Most Smalltalk language features, e. g. class definitions and controls structures, are based on the message passing paradigm. These features do not require special language constructs as in many other languages. Message-passing can be translated to method invocation in Java.

A few difficulties exist. [Boyd, 1997] and [Eßer et al., 2001] discuss these in detail. A convenient table comparing the constructs of both languages is given in [Fussell, 1997a]. The most important difficulty are blocks. Smalltalk blocks are mainly used for control structures, for iterations, and for continuations. The conversion of control structures is straightforward, for iterators the code has to be adapted to use the Java-style iterators. Most blocks which are used as continuations can be mapped to anonymous inner classes in Java [Boyd, 1997]. However, the semantics of block closures are complex and blocks can be used in ways that do not fit into the three categories mentioned here. In these cases blocks cannot be converted automatically and there is no alternative to rewrite the code manually, see section 6.3.1 on page 101.

Some Smalltalk features cannot be converted to Java. Code using them needs to be redesigned. An example is the `become:` method. In fact it is—as most Smalltalk features—implemented as a method (in `Object`) and could be considered a feature of the library. Here, the boundary between language and library is vague. `become:` works like a special assignment statement. It changes all references to the receiver to references to the argument. A similar feature is not available in Java.

New Java features like packages and interfaces can generally be ignored in the conversion. Some of them are enforced by the Java language and thus have to be addressed. Exception handling works in a similar way on both platforms. However, the conditions under which exceptions are thrown are different and the hierarchy of exception classes is different as well. At the syntax level exceptions are thrown with the keyword `throws` and caught with `catch`. The Java compiler requires thrown exceptions to be declared in the method signature and exceptions thrown by other methods to be caught or rethrown. The point is that the Java language enforces these conventions. For the syntax conversion this means

that catching and throwing of exceptions has to be added to the code including the correct typing of exceptions.

Syntax converters can be implemented in many ways. e. g. the ORISABase converter is implemented as a stand-alone converter in Prolog [Elsner, 1998]. It requires Smalltalk code to be filed-out from the image. The `.st`-file is used as input for the converter. The MOVE/S2J converter is implemented in the original Smalltalk environment and works like a file-out filter for saving code in a different format. The advantage of the latter approach is that the implementation can use the other tools of the Smalltalk environment, e. g. to search for senders or implementors of messages. One could also imagine to implement the converter as an import filter of a Java IDE but to the best of my knowledge this has not been done. In this case one would loose the benefit of using the existing Smalltalk tools for the implementation of the converter.

All implementation alternatives convert the Smalltalk code as a whole and produce a syntactically almost correct Java program—with the few limitations noted above. The converted program will not be semantically correct, see also section 2.4.1.3.

The important issue for the migration process is that the Java program does not work at this stage. Not in parts and not as a whole. A developer now has to work bottom-up to fix all problems. I will discuss this situation in more detail at the end of the next section.

2.4.1.3 Library Conversion

The library conversion is the most difficult part of a migration. See section 2.1.2.3 for details. The library conversion uses a mapping from Smalltalk classes to Java classes which provide a similar functionality, if available.

Developing such a mapping is a large undertaking in itself, given the size of the Smalltalk library. A mapping should also include a mapping from Smalltalk messages to Java methods, if available. Such mappings are specific to a Smalltalk dialect and only few portions can be reused for other Smalltalk dialects.

The hard part of the library conversion is this: to make the migrated code work, it has to be adapted to the Java class library. I call this *rebinding* of the application. The rebinding is similar to refactoring [Fowler, 1999] in that it consists of making small and well-defined changes to the code. The difference is that rebinding alters the behavior of the code to make it work with the target library.

The rebinding can cause a lot of hand-work at many places in an application. To reduce this effort it can be useful to combine rebinding with emulation and isolation of parts of the class hierarchy. e. g. MOVE/S2J uses both options. This tool includes a package which reimplements the Smalltalk collection class hierarchy in Java (`de.gebit.trend.collection`). Another package (`de.gebit.smalltalk.s2j.gui`) emulates Smalltalk GUI widgets in Java. The migrated application will not use the native Java toolkit directly (AWT or Swing) but via the provided package. The benefit of emulation and isolation is that fewer changes to the code are necessary. This will ease the migration but the Java code still has a Smalltalk “smell”.

The class mapping—in addition to a syntax conversion—is a major step in a migration process. However a lot of hand-work remains because the mapping is only an approximation and the rebinding of the client code cannot be automated.

2.4.2 General Purpose Development Tools and IDEs

General purpose tools are general in the sense that they are not designed to support migration specifically, unlike the tools presented in the previous section. After using these specific tools, a developer will face the challenge to fix a large amount of little problems in the generated Java code. At this point, text processing tools like `sed`, `grep`, `find`, `awk`, `perl`, etc. are useful for searching and manipulating the Java files. Powerful text editors like Emacs provide similar functionality. Since the use of such text processing tools would be ad hoc and unpredictable, I focus here on two other kinds of tools: testing tools and IDEs.

2.4.2.1 Testing Tools

The migration processes I presented in section 2.3 all address testing. The rationale is that black-box tests can demonstrate the functional equivalence of the original and the migrated system. Testing needs to be automated to be useful and requires tools for managing test data and running test cases. White-box testing, i. e. testing at a level below the system boundary, is less useful because it does not allow for the comparison of the two systems.

Testing tools can only test code if the code can be executed. However, after syntax conversion and class mapping the generated code will need to be fixed before it can be executed. Initially, all test cases would simply fail because the target system would not even run. The next step would be to make at least one test case fail in a meaningful way by executing a skeleton of the target system. A developer would have to work bottom-up to get such a skeleton working. From there on, the developer's micro process can continue test case by test case. The main limitation of this process is the initial step toward the executable skeleton. Since the skeleton can turn out to be a major part of the overall system, the benefits of testing are delayed and process support in the early migration phase is missing.

2.4.2.2 Integrated Development Environments

Of course, one cannot perform a migration using only the specific migration tools. The barest minimum a developer needs to use is a Java complier. Hardly anyone would today use only a command-line compiler and a simple text editor since powerful IDEs are available for Java.

IDEs have no special support for a migration since they are geared towards forward engineering. They have wizards and template mechanisms to create new program elements quickly. It is possible to extend IDEs with additional tools (plug-ins) to support migration, e. g. export filters, as described in section 2.4.1. Another obvious limitation of most IDEs is that they focus on a single language, like Smalltalk or Java. The consequence for a migration project is that one will hardly find an IDE which supports both the source and the target development platform.

For a Smalltalk-to-Java migration, a Java IDE will support the forward engineering part of the migration process, i. e. working with the Java code. Modern IDEs are very powerful. I will not review their merits here because their benefits are well-known. An important point to note is that modern IDEs provide excellent support for agile development processes. The tight integration of tools and especially incremental compilation allow for short turn-around

cycles between different coding activities such as editing, browsing, compiling, executing, and debugging code. This level of process support sets a standard against which migration processes and their tool support have to be measured.

A sensible developer will not only use a Java IDE but also the Smalltalk IDE. The Smalltalk programming environment provides many useful features for the analytical tasks of a migration.

The Smalltalk class browsers help in understanding the code base. A developer can use the cross-referencing features, e. g. senders of a message, to navigate through the code effectively. Inspectors help to analyze and modify individual objects, object structures, and the system state as a whole. Debuggers help understanding code by observing an execution. A developer can halt and resume an execution. He can experiment with the code to verify his understanding. Executing the original system is also key to gathering runtime data for further analysis. More details about features of the Smalltalk programming environment can be found in section 3.5.3.4.

Both the Smalltalk and the Java IDEs provide essential tool support needed for a migration. Together with the specific migration tools they comprise a useful tool-box. However, working with this heterogenous tool-set is not productive because the migration process is not supported seamlessly. I describe this problem in the next section.

2.5 Problem Statement

The previous sections have discussed various aspects of migration, processes, and tool support. Here I summarize the problem of missing process support (2.5.1). I contrast this problem with a vision of a well-supported piecemeal migration process (2.5.2) and conclude with the statement of the exact problem this dissertation addresses (2.5.3).

2.5.1 Missing Process Support

Let's look at how a developer would perform a migration (2.1) following the source-to-source migration path (2.2), using an incremental process (2.3), based on the available tools (2.4). The process starts with exporting the Smalltalk code and converting the syntax to Java. The syntax conversion can be part of the export process if the converter is implemented as a plug-in to the Smalltalk environment, or it can be a separate step if the converter is a stand-alone tool.

The migration tools generate a set of Java files with a large number of problems due to their limitations (2.4.1). The developer now has to face the challenge to make the Java code work. If the developer decides to develop an automated test suite along with the migration—as I would recommend (2.4.2.1)—the first test cases should be implemented now. The test cases will structure the developer's further work.

With or without a test suite, the next step is to fix the Java code bottom-up towards a running skeleton or kernel of the application that can be "fleshed out" in later iterations. The first test case mentioned above can then be run against this skeleton. At this early stage the

test will probably fail due to semantic errors in the code. The important point is to *run* the test case.

Working with an executable application will significantly improve the developer's productivity because he can—in addition to static information—use dynamic information about the program. The Java IDE provides many features to obtain this dynamic information (debugger, breakpoints, single-stepping, inspectors, etc.). Also additional tools like profilers can be used.

In order to fix the problems in the generated Java code, the developer will often refer to the original Smalltalk system and use the Smalltalk IDE for program understanding. Since the Smalltalk and the Java IDE are separate environments, frequent switching between them is tedious. e. g. browsing and searching for references to program elements has to be done in both environments separately. If the environments were integrated in some way, such tasks could be performed with less effort.

This kind of integration is one aspect of the environment integration which is the major theme of the remaining chapters. The other aspect is the integration of the code of both platforms, see section 2.5.3. There is yet another kind of integration not to be confused with these two aspects. This is the integration of the Smalltalk development tools and a developer's application within the Smalltalk environment (virtual image), see section 3.5.3.4 for a description of the architecture of the Smalltalk programming environment.

Making the initial skeleton work is hard not only because the migration tools produce imperfect Java code. The major difficulty is to find a group of classes that can serve as the skeleton because for each candidate class all collaborator classes have to be included in the skeleton as well. Transitively, this can include a large number of classes and make the first step towards the executable skeleton difficult. e. g. let's consider some persistent domain object. The domain object could not be used unless the persistence mechanism would be available on the target platform. However, that mechanism is—of course—still available in Smalltalk. If there were a possibility for the migrated Java domain object to use the old persistence mechanism, the initial executable skeleton could be reached much faster. Since there is no such possibility, the developer has to work with Java code which he cannot execute in a meaningful way over a long period of time.

This example shows that the migration process is not well supported by the available tools. Before stating the problem in detail in section 2.5.3, I describe in the next section how better tool support would result in a more productive process.

2.5.2 Vision: Piecemeal Migration

In this section I describe the vision of a different migration process. The purpose is to show what is on the wish-list for current tools. The process is a vision in the sense that I omit the description of the tools that would be necessary to support it. This will be the topic of my thesis in chapter 4. The vision addresses the problem of working with the "broken" Java code after some converters have been applied. I assume the same use of a test suite as discussed in the previous section.

The process starts with the development of an initial test case. The developer selects a class that he wants to migrate. By some means he transfers the class to the other IDE. The

transfer includes an automatic application of some specific migration tools. These tools convert at least the syntax to Java and possibly perform more advanced transformations. Due to the limitations of the migration tools (2.4.1) the Java IDE will possibly flag some compilation problems. The developer fixes these problems and compiles the code. At this point the code can be executed but may produce wrong results. This would be indicated by running the test suite.

In this vision the migrated Java code gets executed together with its original Smalltalk collaborators. Most classes are related and invoke the other classes' services in their own methods. A major difficulty described in the previous section is that a developer has to fix a class' collaborators transitively. Here, the collaborators do not need to be fixed because the original Smalltalk classes are used.

The point is that the system the developer works with is a *hybrid* system consisting of original and migrated code. The migration process is improved by not requiring the transitive migration of collaborator classes. The migration can be done in small steps, one class at a time. The hybrid system is executable at the end of each small iteration. Errors can be found easily because the unit of work in each iteration is small. Automated tests can be run against the executable system and serve as regression tests.

2.5.3 The Exact Problem

Parallel to the line of thought since chapter 1 I formulate the problem statement from the general level of migration to the level of programming.

Problem Statement

1. Migration is difficult because productive migration processes are missing.
2. Migration processes are not productive because tools do not supported them well.
3. Tools do not support a migration well because a developer has to work with isolated tools and code bases.
4. Integration of the tools and code bases is difficult because it has to address both the programming and the runtime environments.

To address the last point I review tool integration technology in the next chapter. My approach to this problem will be presented in my thesis statement in chapter 4. The solution will be fleshed out in chapters 5 and 6.

Chapter 3

Tool Integration in Programming Environments

In this chapter I review research and technologies in the area of tool integration. I start with a definition of tool integration (3.1). The definition depends on the concept of a *programming environment* (3.2). I present various aspects of programming environments including a brief history, elements, models, and a classification of environments. The three important tool integration mechanisms are data, control and view integration (3.3). Tool integration architectures (3.4) are at a higher level than these mechanisms. I continue with standards that have been established in this domain and concrete implementations including research prototypes and commercial products (3.5). Smalltalk is not just a language but also a programming environment. It has some unique features, especially in the area of tool integration (3.5.3.4).

3.1 Definitions and Terminology

In this section I define the terms *tool*, *tool integration*, and *environment*. Especially the term environment needs clarification because it is very generic and used for different purposes in different domains.

The only widely known acronyms from this field seem to be IDE and CASE. When researching the literature I came across a vast number of acronyms such as SDE, PSE, IPSE, PSEE, SEE, SPU. I will explain these acronyms below in section 3.2.1. These acronyms usually emphasize a specific aspect of an environment. Since most of them are not used today any more I map them on a generic definition of an environment and mention the specific aspects separately.

The terminology used in this work is defined as follows:

- A *tool* is a program used for software development which operates on another program or on a higher-level representation of another program.
- *Tool integration* is the act of combining tools into an environment.
- An *environment* is a set of tools that work together.

Environments fall into two main categories, exemplified by the terms IDE and CASE mentioned above. The focus of the first is the act of programming, usually of a single developer. These environments are universally called *development environments*. The focus of CASE tools is the task of modeling which is usually part of a larger software engineering effort. Environments which address the overall aspects of software engineering are called project support environments or software engineering environments. Such environments address a much wider range of tasks than development environments. Since the focus of this work is the integration of programming environments I will touch on software engineering environments only briefly.

Special kinds of environments are runtime—or execution—environments. When referring to these I always use the qualified term and never subsume them under the general term environment. In this chapter the term environment generally refers to development and software engineering environments and deliberately excludes runtime environments. This convention is not the most elegant classification but it reflects the common usage of the terms and shows that they stem from different domains.

The distinction between environments in general and runtime environments is orthogonal to the categorization of development and software engineering environments above. This distinction here is also more general since it applies to all programs in that every program is written at “development-time” and executed at run-time.

It is important to have this distinction in mind since development environments can be integrated with runtime environments as well. For an example how this is achieved in Smalltalk see section 3.5.3.4.

The obvious integration is that every development environment *executes* in a runtime environment of some sort. The interesting aspect is that this runtime environment can also be integrated with the runtime environment of the software that is being developed. There are two major examples that I will present later: the Smalltalk programming environment, which heavily relies on reflection and being interpretative, and debugger interfaces which integrate an IDE with a runtime environment.

I use the term *programming environment* for a development environment together with an execution environment. Often a class library is included as well.

3.2 Environments

Environments come in all sizes and shapes. Environments as defined in the previous section include programming environments with a narrow scope and software engineering environments with a wide scope. Section 3.2.1 gives a brief history of environments and sets the stage for more detailed aspects later. The history exposes the major developments, milestones, actors, market concerns and other topics. In later sections I will frequently refer to this brief history.

Section 3.2.2 enriches this set of elements by describing models of environments. These models are interesting because they already define some kind of integration for the tools contained in an environment.

The classification in section 3.2.2 provides a new perspective on the history presented in the first section of this chapter. This classification organizes the concepts which have been

introduced in the rather subjective historical outline of the field. As before, these classifications cover tool integration among all other aspects of environments. These criteria will be useful in the remaining sections as I proceed to more and more detail.

3.2.1 History

This review is mainly based on the review included in [Ossher et al., 2000]. The authors differentiate between early environments targeted for programming (3.2.1.1) and environments with a wider scope targeted at software engineering in general (3.2.1.2). I add another section on environments that try to improve specific tasks or provide innovative solutions to common problems (3.2.1.3). A good, though dated, overview of the literature provide [Brown and Penedo, 1992]. A more up-to-date overview of integration in software systems in general, i. e. not focused on tools, can be found in [Stavridou, 1999].

3.2.1.1 Programming Support Environments

Programming Support Environment (PSE) is the historic term for programming environments that are generally called Integrated Development Environments (IDE) today. The term *environment* for a collection of software tools was first used around 1980. This has to been seen in the context of the emerging personal computers.

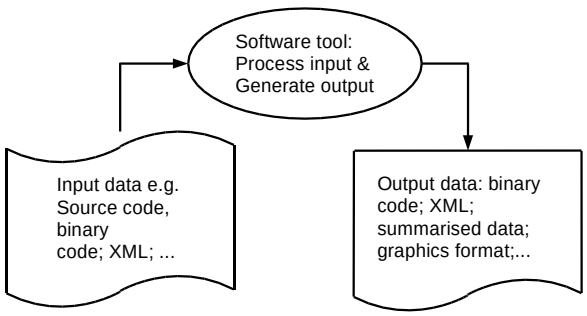

Figure 3.1: Typical structure of environments [Grundy and Hosking, 2001], batch input/output-driven

However, the first programming environment—although it was not called that—was the Unix *operating* system in 1971. Unix was labelled a programming environment much later, as in the title of the book by Kernighan and Pike [Kernighan and Pike, 1984]. The power of Unix came from the sharing of files between simple tools and from the pipes and filters architecture for combining tools. This was the first data integration approach, see also figure 3.1. More details about data integration can be found in section 3.3.1.1.

Control integration between tools was manual. The pipes and filters architecture allowed for easy combination of tools and one invocation of a tool passed its output to the next tool.

In this scenario, a developer had to take care to invoke tools in appropriate ways. One of the first automated tools was `make` for automating the build process. It was configured using a makefile and shared data with other tools via files as well. This kind of tool integration was straight-forward but limited to batch processing.

The sharing of files caused a number of problems. The solution was to store and exchange data using a database. This database is usually known as a repository. Figure 3.1 shows the typical structures of environments at that time. Note that these structures apply to single-user environments—I will discuss the use of a database in a multi-user scenario in the next section.

This was an improvement to data integration but not to control integration. A major step to improve control integration was the Field environment by Steven Reiss [Reiss, 1990]. Tools were connected with each other using message passing. The format of the the messages was predefined and permitted free-form fields for arbitrary data. Messages were delivered via a message server called Msg. Tools wishing to receive messages had to register with the message server. Message passing was easy to implement and the tools were only loosely coupled. This mechanism not only provided a great improvement to control integration but could also be used for simple data integration. Reiss designed and implemented a second environment—Desert—in which tools could exchange *fragments* of data using messaging [Reiss, 1996]. More details about control integration can be found in section 3.3.1.2.

Another important issue is presentation integration, especially from the time on when graphical user interfaces were available for programmers. The PCTE environment (3.5.3.2) of the early 1980's was targeted towards the X Windows GUI system that was available on Unix workstations. Due to many inconsistencies in the GUI domain—different operating systems, style guides, GUI toolkits, programming languages—presentation integration is still a difficult problem today. A limited success could only be achieved in settings were a major company dictated the platform and tool developers voluntarily adopted this regimen, e. g. Microsoft VisualStudio and Eclipse. More details about presentation integration can be found in section 3.3.1.3.

So far I only mentioned *programming* environments. The use of a database for sharing data already suggested that it was desirable to use environments on a larger scale and support other software engineering tasks than programming. This implies also the use of environments by multiple users.

3.2.1.2 Software Engineering Environments

Software Engineering Environments (SEE) extend the scope of programming environments beyond the core programming activity. I will first present some environments that are extended to activities like design and testing. Another category of SEEs not only addresses a wider set of activities but the development process itself. I will mention these process-centered software engineering environments (PSEE) at the end of this section.

Supporting a wider range of tasks in an environment has two consequences: 1. the number of tools increases and 2. the number of users of the environment increases.

Environments have been extended to support any relevant task, both on the development level, but also on the project management and quality assurance level. The support for

development tasks includes tools for requirements gathering, design, GUI construction, database design and management, testing, code analysis, deployment, bug tracking and many others. A full list is given in [Sharon and Anderson, 1997].

The integration of these tools into an SEE involves all three aspects—data, control, and presentation-integration. Just considering data integration one can only state a limited success. For example tracing a bug backwards through the tool chain to the test tool and then to the requirements tool shows how many parties are involved in this simple task. The tools involved in this example would have to agree on a common data model for the data that is being exchanged. However, no such standard could be established so far. All SEEs that exist today a developed by a single organization and do not require a public standard.

On area in which standards exist is the exchange of data between design tools and programming environments. I will refer to these design tools as CASE tools as it is common usage. Some authors call every software tool a CASE tool since it is computer-aided software engineering, in the strict sense. CASE tools gained much interest in the early 1990's. As the market grew many companies started to develop CASE tools. The companies soon had to find out that their potential customers were using other tools for software engineering and that their CASE tools were not compatible with them. The acceptance of CASE tools depended very much on the ability to exchange data between them and other environments. Standards like CDIF emerged that provided simple data integration using files. I provide more details on data integration in section 3.3.1.1 and on exchange formats in section 3.5.1.2.

The idea of using a common database, or repository, was not as successful as the CDIF standard. Environment standards such as PCTE defined the basic infrastructure for an environment, even including an object management system, but left the semantics of the data open. User-defined schemas were considered a feature, not a limitation. The lack of a public standard schema for repositories prevails as developers still exchange UML models using files. Even if a repository is used and the data is exchanged at the level of files, only coarse-grained data integration is achieved.

An improvement of using a repository over using files was that a repository is able to handle multiple users and that it stored the data in a single place without duplication. Many achievements from the database field benefited SEEs, e. g. locking, transaction, and replication.

As multiple users were using the SEEs, the question arose how to present the repository data to different users in appropriate ways. e. g. requirements analysts and test engineers would be interested in different aspects of domain objects stored in the repository. A whole research trend developed around the idea of multiple views of the same data [Grundy and Hosking, 2001], see also figure 3.2. Example systems include Viewpoints, Statemate, and Rational Rose. These views are quite similar to views in a relational database. Supporting multiple views gets hard when a GUI presents these views in parallel unsung multiple windows, and furthermore when multiple users do this at the same time.

Another research trend focused on process-centered SEEs (PSEE). This trend dates back to Osterweil's landmark paper [Osterweil, 1987] in which he proposes to treat a software process simply as software. In these environments the meta-model not only includes the artifacts of software engineering but also the process itself. A PSEE has tools to *enact* and manage a process. This includes all the management tasks as assigning people to tasks, etc. One example of a commercial tool is Innovator which I used at DaimlerChrysler. The

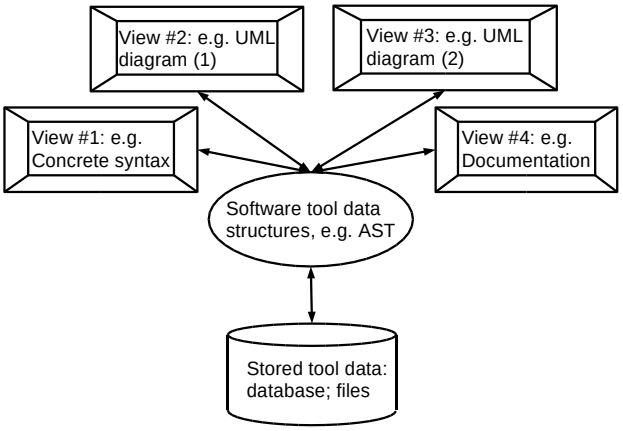

Figure 3.2: Typical structure of environments [Grundy and Hosking, 2001], interactive, multiple views, incremental persistency

standard development process of DaimlerChrysler, the “Handbuch der Systemgestaltung” (HBSG), was defined in Innovator to assist projects in following the process. In this case—and in general—the acceptance of process-support in SEEs is limited. Today there are light-weight processes like eXtreme Programming which propose to use both simple and powerful tools for programming but not for communication. The agile processes instead focus on face-to-face communication, e. g. pair programming, because it is the most effective. However, this research direction is still very active since many organizations hope to reach higher levels on the SEI CMM model by using PSEEs. .

The PSEE research resembles research into general purpose work-flow systems. Such systems have process modeling and management tools. However, I believe that the software process is far to complex and ad hoc for formalization and automatic enactment. These work-flow systems seem to be adequate for less intellectual and more administrative tasks—which software engineering is not. Before software processes can be executed by a tool the materials of software engineering must be understood better and agreed upon in the form of a public standard model. The lack of this model is one of the problems with using repositories mentioned above. PSEE are—to use the terms of Brown and McDermid—tackling the wrong problem, or the right problem at the wrong level.

The overall trend in SEEs is towards more and more “outward” integration. The topic of “inward” integration has been covered well by the PSE research and powerful IDEs are available today. By outward integration I mean integration of tools into their context, i. e. into the software engineering team and organization, their methodology, their standards and their possibly informal working model, into new working conditions such as tele-commuting and open source development on the Internet.

3.2.1.3 Specific Envirionments

In this section I review specific environments that focus on special tasks of software engineering or provide special solutions to common problems. One source for recent trends in environments is [Grundy and Hosking, 2001].

Component-based environments are build using a component model such as COM, OpenDoc or JavaBeans [Grundy et al., 1998, Grundy et al., 1999]. Component-based tool integration solves the basic problem of how to invoke services from other tools and how to share data—as components. However, given the experience of PCTE I believe that this direction also is limited to the syntactic level of tool integration. If tools agree to adhere to COM or JavaBeans standards nothing is said about the interfaces they provide. COM and JavaBeans provide in this scenario the same functionality that PCTE's object management system provided. As of today, only few environments are implemented using components and it has to be seen if this approach will be successful. e. g. the Java IDEs Eclipse and NetBeans do not use JavaBeans as an extension mechanism but rather define their own plug-in model that tool implementor have to adhere to. The ubiquitous plug-in interfaces show that general component models are of limited us and that specific interfaces for domain specific plug-in are necessary.

Some environments use the Internet as basic networking infrastructure to address new working models of developers and distributed teams, e. g. [Kaiser et al., 1997]. Using HTML for user interfaces, HTTP for communication, or Java Applets is at best at the carrier level of tool integration.

A combination of component-based and web-based environments uses SOAP for the communication between client and server tools. This may ease the implementation of such tools and solve some problems that distributed teams have but it does not improve tool integration in a significant way—it gets rather harder.

Web-based environments do not necessarily have to follow the client-server model. It has also been attempted to implement distributed environments which can also operate independently. Synchronization between different locations can be performed on demand.

Agent-based environments use agents to perform some work inside an environment. In my understanding an agent is an autonomous process performing a task for the user. I would not imply the use of any AI techniques. The agents can either run inside the environment, or inside the tools that populate the environment. In either case, the infrastructure provided by the environment or tool has to specify or conform to some standards for agent execution. Or, the agents are coded directly into the tool. Some tools provide scripting which would allow an agent extension. An example of this Instantiations' CodeStudio Pro which integrates IBM VisualAge/Java with IBM WebSphere Application Developer (Eclipse). Another example is the system of Babst and Krone [Bapst and Krone, 1997] which has agents for coordinating a mulit-paradigm programming environment.

Reflective environments are environments which contain a representation of itself. This is also called a *self-representation*. An example of a reflective programming environment is Smalltalk. In Smalltalk all components that make up a program—classes, methods, source code, bytecode—are represented as objects. For more details see section 3.5.3.4. Reflection is not just a sophisticated feature of a high-level language but a feature of an environment. Reflection is also present e. g. in C++ in form of run-time type information (RTTI), and even in C it is possible to provide reflective facilities [Kowalski et al., 1991].

Meta-environments are environments used to create other environments. They should not be confused with reflective environments which allow meta-programming at the level of the self-representation of the environment. Meta-environments were developed to ease the creation of environments because this can be a challenging task in itself. More than 60 meta-environments are listed in [Karrer and Scacchi, 1994] but most of them are rather

frameworks to build customized frameworks. A meta-environment in the strict sense uses a specification or meta-model to create a programming environment for use by developers. In this scenario tool integration can be achieved rather easily because everything is under the control of the generator. However, integrating foreign tools is difficult because the generated code would have to be adapted, or the generator would have to be extended to provide stubs and skeletons for foreign tools. Most of the work in this areas is of very theoretical nature and references to widely accepted tools are not made.

Maintenance and reengineering environments address specific maintenance tasks. The idea to support maintenance with specific environments is as old as the idea of programming environments [Feldman, 1979]. Basically, the tool integration issues are the same for this kind of environments as for forward engineering environments. A difference is the kind of data being exchanged. The more analytical tasks of maintenance require different information to be shared. As in forward engineering no widely accepted standard for data exchange could be established. There were some old, and there are new efforts to standardize file formats but it has to be seen if they are successful. The reengineering environments went a similar route like the SEEs. When interest in process support grew, some research has been conducted as to how support the maintenance process, e. g. [Belkhatir and Melo, 1993]. In reengineering techniques from artificial intelligence and program transformation are easier to apply than in other domains. This is reflected in the tools that need to be integrated and the data they exchange.

There are too many different kinds of specific environments to list them all here. There are environments supporting multiple programming paradigms [Meyers and Reiss, 1995], environments addressing software architecture [Christensen, 1999], environments using virtual reality for visualization [Stotts and Purtilo, 1994].

To conclude the historic review I would like to state what I consider state-of-the-art in today's software development environments. One has a choice between highly sophisticated environments like Smalltalk with poor external integration and on the other hand simpler—though still powerful—loosely coupled file-based tools as in typical Java IDEs. However, to attempt both high sophistication and tight integration has not been successful so far.

3.2.2 Classification and Models of Programming Environments

The history of environments in section 3.2.1 presented a great number of different environments. The purpose of this section is to clarify the various aspects that were introduced earlier. When researching the literature I found several classifications and reference models of environments. Each of them highlights specific aspects. Since my focus is tool integration I include two models of environments that each expose some interesting aspect of tool integration. These classifications and models deal also with a number of other issues which I omit unless they are related to tool integration. The next section (3.3) on tool integration mechanisms contains another classification which focuses on tool integration.

3.2.2.1 Three Components and Four Classes of Models (Perry/Kaiser)

The model by Perry and Kaiser [Perry and Kaiser, 1991] contains a general model consisting of three components: policies, mechanisms, and structures. Based on this general

model four classes of models are defined: individual, family, city, and state model.

The following three components make it easy to separate different concerns:

1. Policies are the rules, guidelines and strategies imposed on the programmer by the environment
2. Mechanisms are the visible tools and tool fragments
3. Structures are the underlying objects and object aggregations on which mechanisms operate

The authors use these three components to classify environments by means of a sociological metaphor based on scale:

- The individual model
- The family model
- The city model
- The state model

These models map to the single-user programming environment, the small team software engineering environment, the large organization software engineering environment, and a new larger than anything environment which has to be investigated further.

For each of the four models the authors present several instances. e. g. for the the individual model—which is relevant for the Smalltalk and Java environments I refer to in this work—the authors present four groups of environments: toolkit environments, interpretive environments, language-oriented environments, and transformational environments. Unix is the premier example of a toolkit environment, see also section 3.4.1. Smalltalk and, to a lesser degree, Java environments are examples of interpretive environments.

3.2.2.2 ECMA Reference Model

The Reference Model for Frameworks of Software Engineering Environments defined by ECMA [ECMA, 1993] is a byproduct of the PCTE standardization, see section 3.5.3.2, but it is not specific to PCTE. The purpose of the reference model is to provide a basis for the comparison of environments. It defines a common terminology for features of software engineering environments and describes their functions. The reference model does not endorse a specific architecture for environments although figure 3.3 looks like an architecture at first sight. However, the “toaster” model is only a high-level grouping of features described in the model. Various reference model like this had been defined and even more *mappings* of existing environments against these models have been made, e. g. [Zelkowitz, 1993].

The model defines seven large service areas: object management service, process management service, communication service, operating system process service, user interface service, policy enforcement service, and framework administration service. Each of this service areas defines about a dozen sub-services. For example the object management

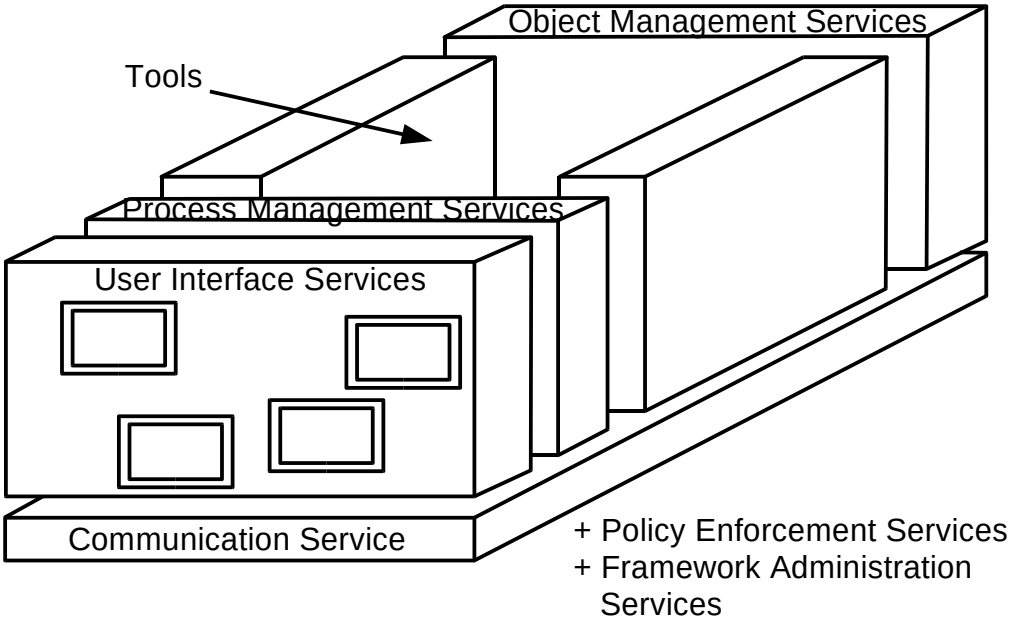

Figure 3.3: The EMCA Reference Model (“toaster” model)

service contains 21 services: metadata service, data storage service, relationship service, name service, distribution and location service, data transaction service, concurrency service, operating system service, archive service, derivation service, replication and synchronization service, access control and security service, function attachment service, common schema service, version service, composite object service, query service, state monitoring and triggering service, data subsetting service, data interchange service.

3.3 Tool Integration Mechanisms

In this section I describe the three main tool integration mechanisms: data, control, and presentation integration. The second part of this section describes a classification of tool integration mechanisms consisting of five levels.

3.3.1 Data, Control, and Presentation Integration

The three main integration mechanisms for tool integration are data, control, and presentation integration. This classification is based on the work by Wasserman [Wasserman, 1989] and Thomas and Nejmeh [Thomas and Nejmeh, 1992]. I will first give an overview of the classification and then go into more details in sections 3.3.1.1 to 3.3.1.3.

Wasserman lists five types of integration. Three of them—data, control, and presentation integration—are cited most often. Platform integration is not a big issue today since platform independence between the major workstation operating systems can be achieved relatively easy. To include process integration in 1989 was far-sighted by Wasserman. Process support became a major research trend which is still active today. The five types of tool integration are, as listed in [Brown and Penedo, 1992]:

1. Platform integration, i. e. the set of system services that provides network and operating systems transparency to applications.
2. Presentation integration, i. e. tools that share a common look and feel from the user's perspective.
3. Data integration, i. e. support for data sharing across tools.
4. Control integration, i. e. support for event notification among tools and the ability to activate tools under program control.
5. Process integration, i. e. support for a well-defined software process.

Based on the work by Wasserman [Thomas and Nejmeh, 1992] provide definitions for tool integration concepts. Their unique approach is to define tool integration as a *relationship* between tools. To the aspects data, control, presentation and process integration they add a number of *properties* that can be used to analyze a given integration approach. Each attribute is also phrased as a question to simply checking for a specific property, see table 3.1.

[Brown et al., 1992] propose yet another classification which they call multi-level classification. It consists of the three levels 1. mechanisms, 2. end user services, and 3. process. This classification is less useful for me because I work only at the mechanism level. It is of more use for research into PSEEs [Stavridou, 1999, p. 12], see 3.2.1.2.

Figure 3.4 shows the different integration mechanisms. Each of the subfigures shows two tools, one on the left and another on the irght. The different integration mechanisms are shown in the middle.

The following three sections focus on the three major integration mechanisms. After that I present a classification which is then combined with the threefold classification.

3.3.1.1 Data Integration

The oldest data integration mechanism is the exchange of files, see figure 3.4 (a). Although it is the oldest mechanism, it is powerful and well-accepted. Current forms are labelled single-source or round-trip tools in which tools synchronize their data with the file system whenever necessary.

Integration through a repository is historically the second data integration mechanism. Examples include file-based version control systems, relational and object-oriented databases. [Stavridou, 1999, p. 10] lists four characteristics of repositories: a data storage mechanism,

Data Integration	Interoperability	How much work must be done for a tool to manipulate data produced by another?
	Nonredundancy	How much data managed by a tool is duplicated in or can be derived from the data managed by the other?
	Data consistency	How well do two tools cooperate to maintain the semantic constraints on the data they manipulate?
	Data exchange	How much work must be done to make the nonpersistent data generated by one tool usable by the other?
	Synchronization	How well does a tool communicate changes it makes to the values on nonpersistent, common data?
Control Integration	Provision	To what extent are a tool's services used by other tools in the environment?
	Use	To what extent does a tool use the services provided by other tools in the environment?
Presentation Integration	Appearance and behavior	To what extent do two tools use similar screen appearance and interaction behavior?
	Interaction paradigm	To what extent do two tools use similar metaphors and mental models?
Process Integration	Process step	How well do relevant tools combine to support the performance of a process step?
	Event	How well do relevant tools agree on the events required to support a process?
	Constraint	How well do relevant tools cooperate to enforce a constraint?

Table 3.1: Tool integration types with elaborated properties and questions

an interface to persistent data, a set of schemata or information models, and a set of operations on that data.

Integration using APIs is more than accessing data through a common interface. This is already addressed by abstract data types. API-based data integration includes standardizing the interface to be used by a large number of tools. It may also include a standardization of the data formats and semantics. As I will show later (PCTE in section 3.5.3.2) only the former has been standardized in some cases.

Message passing was originally envisioned as a control integration mechanism. I will present it in the next section in more detail. Message passing can also be extended for data integration using it to send fragments of data [Reiss, 1995].

3.3.1.2 Control Integration

Control integration can obviously be achieved by direct (procedure) calls. More sophistication is reached by a standardized API for invoking tools. This allows for different architectures such as master/slave or framework architectures into which plug-ins can be embedded, see section 3.4.

The main limitation of direct calls is tight coupling and platform dependence. Cooperating tools have to share the same execution environment. This implies sometimes also the use of the same programming language for the implementation of the tools.

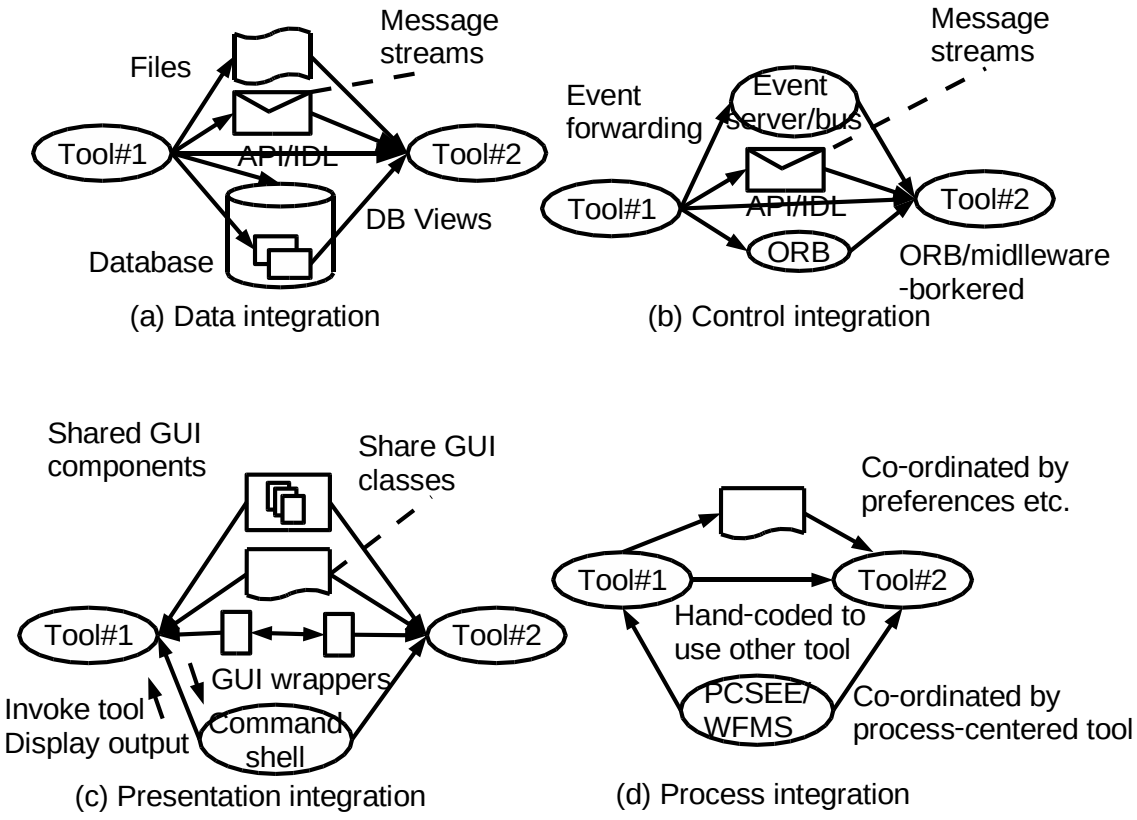

Figure 3.4: Examples of tool integration strategies [Grundy and Hosking, 2001]

The use of message passing was a milestone in tool integration [Reiss, 1990]. The message passing mechanism uses a central message server to which tools send messages intended for other tools. Any tool can register interest with the message server and request message broadcasting to itself. As mention earlier, message passing can also be extended to support data integration.

Figure 3.4 differentiates between message streams , a message broker, and an event server. Message streams are connections between tools that are similar to the pipes and filters architecture described in section 3.4.1 and are used for the delivery of multiple messages in a row.

Message brokers can deliver messages synchronously and asynchronously. The difference to an event server is that a broker must know the recipient of a message while an event server broadcasts messages to receivers that have registered interest in specific messages. In the latter case a sender does not know which tools will eventually receive the event notifications.

3.3.1.3 Presentation Integration

Since command line user interfaces are fairly outdated, the only two relevant presentation integration mechanisms are 1. using the same GUI toolkit and 2. (re-)using the same window elements, e. g. a class tree view or a highlighting editor. For migration between two

programming environments it would be too ambitious to integrate the GUIs of both environments. For this task it should be sufficient to have data and control integration.

3.3.2 Levels of Tool Integration

Brown and McDermid define five levels of tool integration [Brown and McDermid, 1992]. They claim that much concurrent work on IPSEs is based on an inappropriate view of integration and that infrastructures like PCTE, see section 3.5.3.2, have limited value because of this. The five levels of tool integration are:

1. Carrier Level—Composing tools by enforcing a single, consistent file format.
2. Lexical Level—Sharing a common understanding of the lexical conventions of structures which are shared between tools.
3. Syntactic Level—Agreeing on a set of data structures, and rules for the formation of those structures, between a set of tools.
4. Semantic Level—Augmenting a common understanding of the data structures used by tools with a common definition of the semantics of those structures.
5. Method Level—Determining a common model of the software development process in which all tools are compatible.

The important level to note is the semantic level. The authors argue that most existing tool integration mechanisms operate only at lower levels and that this is not sufficient.

[Cuthill, 1994] combines this five-level classification with the data, control, and presentation integration distinction in table 3.2. Note that the fifth level, the method level, has been omitted because an object-oriented method is generally assumed. Also, in the context of a Smalltalk to Java migration no significant difference would occur at the method level.

Integration category	**Carrier**	**Lexical**	**Syntactical**	**Semantics**
Data	File transfer	Using shared files	Use of the same database/object base	Use of the same metadata
Control	Remote procedure call	Triggers	Message servers	Agreed messages
Presentation	Use of the same window system	Use of the same window manager	Use of the same toolkit	Standard semantics for toolkit

Table 3.2: Integration levels and capabilites[Cuthill, 1994]

3.4 Tool Integration Architectures

Tool integration architectures describe integration at a higher level than the mechanisms presented in section 3.3. A tool integration architecture also addresses non-technical aspects, e. g. the integration standards supported by a group of tool vendors, or company-wide standards for a development process.

Architectures embody high-level design decisions. Once an architecture has been implemented the design rationale is often lost in a vast amount of technical detail. A good way to look at the architectures that I present here is to ask *why* it has been chosen, i. e. what problem does it try to solve.

Note that in this section the term architecture refers to *tool integration* and not to environments in general. Some research has been done in the area of *environment* architectures. Today a—possibly distributed—object-oriented architecture is generally assumed. Especially Smalltalk and Java programming environments that I will look into later have an object-oriented architecture. Therefore, I consider only tool integration architectures but not environment architectures. As I will show later I consider the programming environments involved in a migration pretty much as black-box components—with some special requirements listed in section 4.2 on page 63.

The term *component* also needs special attention. When talking about architectures the term component refers to a large computational unit. This use of the term should not be confused with a component in the sense of a component model such as COM or JavaBeans. There has been some research in the direction of component-based environments, e. g. [Lin and Reiss, 1993, Grundy et al., 1998], but since these kinds of environments are 1. not a reality today and 2. not relevant for a migration I do not consider them further. At the architectural level *component* refers to complete tools, e. g. editors, compilers, and repositories. As architectural components they can be components in the sense of a component model but they do not have to be.

The four types of architectures that I present here can be distinguished by the *level of agreement* between cooperating tools. In the first architecture the tools agree on nothing more than to use *pipes* (and/or files) to cooperate. A *workbench* is an environment created top-down, usually from a single vendor, into which tool suppliers can integrate their tools—provided they adhere to the interfaces defined by the workbench supplier. There is no voluntary agreement between tools. [Wallnau and Feiler, 1991] call this architecture *me-centered*. The next two architectures are examples of what Wallnau and Feiler call *me-too* tool integration architectures. Tool *coalitions* agree on a narrow interface—implemented using one of the mechanisms from the previous section—which is specific to the tools involved in the coalition. A *federation* is an extension of a coalition to a wider set of tools. In a federation tool implementors agree on public standards and tools that implement a standard interface can be replaced by each other.

Using Eric Raymonds metaphor of the cathedral and the bazaar, the workbench would be the cathedral and the tool federation the bazaar. The coalition could metaphorically be called a Mafia organization.

3.4.1 Pipes and Filters Architecture

The pipes and filters architecture has its roots in Unix [Kernighan and Pike, 1984]. In section 3.2.1.1 I mentioned already that Unix pioneered the concept of a programming environment long before the term environment was established. The pipes and filters model for tool integration was only much later in [Shaw and Garlan, 1996] conceived as an architecture. Also, for example the famous C3 project at Chrysler [Beck, 1999] used the metaphors of stations and bins for its architecture—very similar to pipes and filters.

As an architecture it provides loose coupling between tools. By sharing files and/or data via pipes it is mainly a data integration solution. Piping output from one tool as input to another tool is a simplistic way of control integration. Although it is efficient it does not provide for much variation in the ways tools can interact. Error handling and integration of multiple tools simultaneously is also limited.

The main limitation of this architecture is that the structure of the shared data is not standardized. Commonly, a convention of one data item per line—whatever that is—is used, e. g. for `ls`, `sort`, `sed`. This limits the integration of tools to the lexical level as defined in section 3.3.2.

3.4.2 Workbench Architecture

The main characteristic of a workbench—sometimes called framework or platform—is that it provides an environment into which tools can be embedded. I will give two examples in section 3.5.3.2 (PCTE) and section 3.5.3.3 (Eclipse). Figure 3.3 gives a good visual impression of what a workbench architecture looks like.

The workbench provides a set of common services that an integrated tool not only can but should use. Tools embedded this way *plug-in* to the architecture and can, at least theoretically, be replaced by other plug-ins that provide the same functionality. For example in the Eclipse platform the version control component is pluggable. Eclipse has support for CVS built in and plug-ins for many other version control systems exist.

The workbench architecture provides fine-grained tool integration. The architecture defines interfaces for tools to use and achieves a high level of semantic tool integration. However, integration of tools that have been developed prior to or independently of the workbench architecture are difficult to integrate.

3.4.3 Coalition Architecture

The coalition and the federation (3.4.4) architecture are different from the previous two architectures in that they try to provide integration for *existing* tools. The pipes and filters architecture of the Unix operating system grew a huge number of tools around the kernel, the workbench architecture assumes tool implementors to follow the defined architecture and write plug-ins for it.

On the other hand coalitions and federations assume tools to preexist already and seek for a way how to adapt them into a greater whole. [Wallnau and Feiler, 1991] note these efforts started in the context of a large number of fully developed CASE tools. At that time it was discovered that those CASE tools were not inter-operable with development environments which inhibited their market acceptance.

The problem with CASE tool integration is that not only syntactic but also semantic integration, see section 3.3.2, is mandatory. Agreeing on semantics is difficult. A vendor can relatively easy define a corporate standard and hope for buy in of tool suppliers. The situation is more complicated with multiple vendors; industry standards are beyond that.

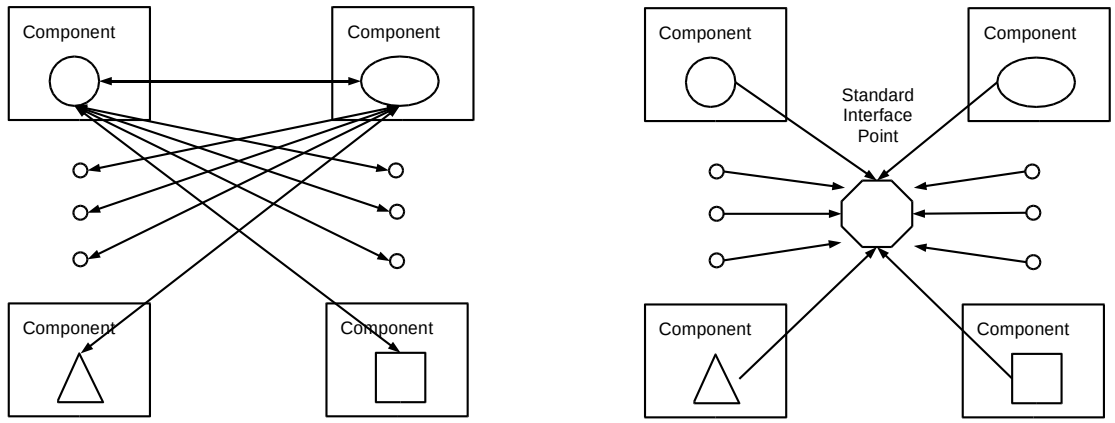

Figure 3.5: Point-to-point integration and integration through standards [Stavridou, 1999, p. 15–16]

A coalition is an agreement between tool vendors on interfaces between tools of the coalition members. Figure 3.5 shows on the left the components of the coalition architecture which are integrated with each other in a point-to-point fashion.

Figure 3.6 shows more detail and that the coalition tools also integrate with the host platform. Wallnau and Feiler postulate an evolution from coalitions to federations which—in hindsight—still has to happen [Wallnau and Feiler, 1991].

Two examples of coalitions are the CASE Communique group and the CASE Interoperability Message Set. For more details see section 3.5. More focused examples are the integration of IDE's Software Through Pictures with FrameMaker using the LiveLinks mechanism, or the integration of MID's Innovator with Microsoft Word.

3.4.4 Federation Architecture

Federations generalize the point-to-point integration of coalitions by integrating through standard interfaces.

There are two kinds of federations, depending on a data or control-centric view. With a data integration focus the standards apply to the schemas of the exchanged data. This would be equivalent to the database schema of an object management system or the meta-model of a repository.

The second kind of federation has a focus of control integration. Since it is not possible to use direct in-process calls from one tool to another, control is passed indirectly through message passing. In that case the standard applies to the message protocol and the syntax and semantics of the messages.

The CASE Interoperability Message Set at first looks like a standard but considering the companies involved I would rather count it as a coalition.

[Wallnau and Feiler, 1991] note that at that time federations were not yet existing. Looking at federations 10 years later I have to attest that none of the proposed federation architectures could be successfully implemented as tool federations.

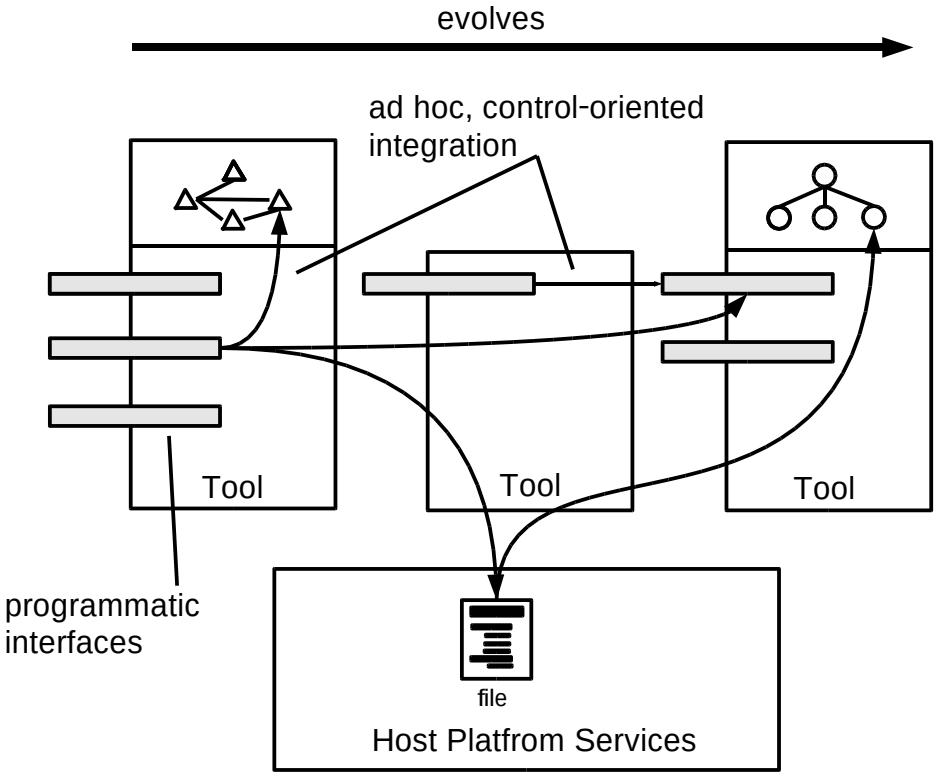

Figure 3.6: Coalition environment [Wallnau and Feiler, 1991, p. 5]

However, during this period another standard had been established and gained wide-spread acceptance. This standard is quite similar to the idea of a tool federation architecture: CORBA. The CORBA standard defined the basic format of messages and for vertical domains it defined specific protocols.

The similarity can be seen in figure 3.7 which shows the horizontal and the vertical service domains.

CORBA was a success for application integration but not for tool integration. The language-independent architecture created too much overhead and additional work to use it for fine-grained tool integration. Nevertheless, there was an effort to define a vertical CORBA service for tool and repository access, the CORBA Meta Object Facility (MOF). The MOF standard is quite complex and did not receive much acceptance by tool vendors. Some implementations exist, mostly by participants of the MOF standardization process. However, a companion standard of the MOF, the XML Metadata Interchange (XMI) format defines a file-based interchange format. XMI is for example used to exchange UML models and has wide-spread support.

3.5 Standards and Implementations

So far I presented many concepts and ideas related to tool integration. The purpose of this section is to present *applications* of these concepts and ideas. I will not introduce any new

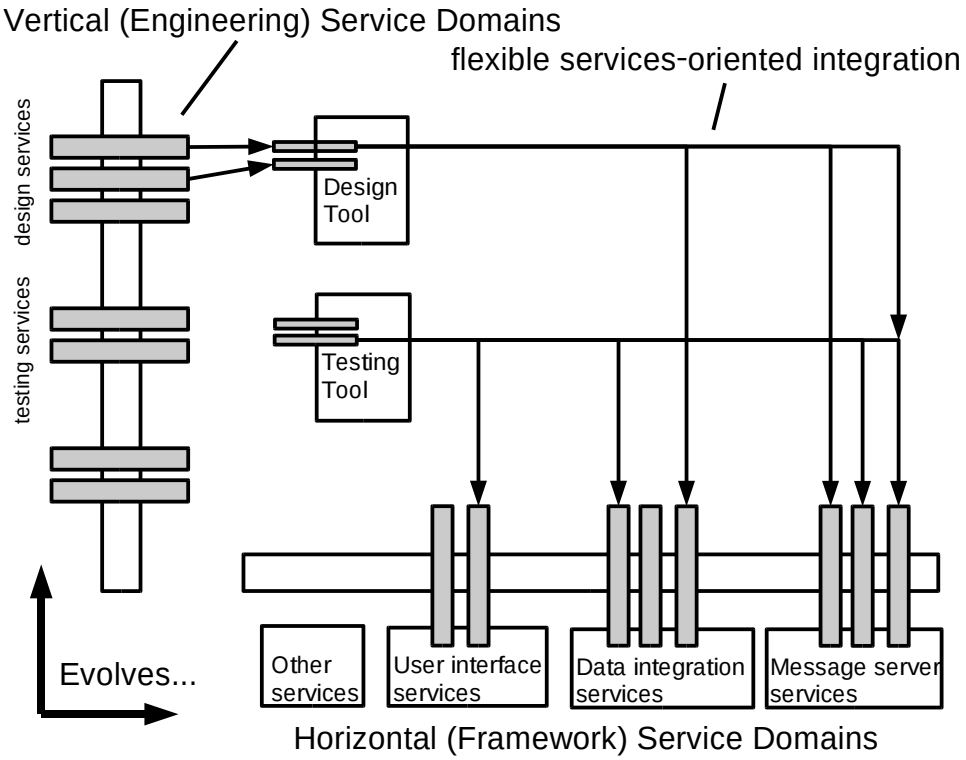

Figure 3.7: Federated CASE environment [Wallnau and Feiler, 1991, p. 6]

topics here—the only interesting aspect of the material of this section is that it uses in some way the integration approaches discussed so far.

This section groups two themes: standards and implementations. Standards are always defined when a number of vendors find it beneficial—for some reason—to cooperate. Especially the CASE tool hype in the early 1990's led to a definition of some standards. Standards were defined both at the mechanism (3.3) and the architecture level (3.4), and use various implementation approaches.

Sometimes generic middleware such as COM or CORBA is also subsumed under the label of tool integration standards. It is important to note that these generic standards reach beyond tool integration. However, there are some interesting aspects in using generic middleware to integrate tools.

The second and third subsection present some concrete implementations of tool integration frameworks and environments. I believe that a thorough review has to include commercial implementations because they ultimately demonstrate the practicality of a tool integration approach. It would be impossible to list all implementations ever made and—considering the repetition of ideas—it would not add much to the understanding of the the key ideas. I include some implementations mentioned in section 3.2.1 on the history of environments and highlight the interesting aspects in each case.

3.5.1 Standards

Many of the standards that have been attempted are data-oriented in the sense of data integration, see section 3.3.1.1. Another kind of standards addresses a message passing style of tool integration. Some of the standards are abstract, others are backed up by a reference implementation. Often a company is behind a standard and tries to win more supporters by making it freely available.

3.5.1.1 Repository Standards and Standard Schemas

Repository-based integration is one of the oldest forms of tool integration. Proprietary standards for repositories include IBM AD/Cycle and DEC CDD/Plus. Some authors consider PCTE to be data-centric and would also count it as a standard for a data store. Similar to repository standards are common data schemas, e. g. the IBM Information Model or NIST IRDS.

Platform migration would benefit from a repository standard or a standard data schema if the development environments of the source and target platform would use such a standard repository or schema. However, this is not the case.

3.5.1.2 Exchange Formats

Many different exchange formats exists, e. g. see [Kienle et al., 2000]. Some formats address specific domains such as the exchange of graphs, syntax trees, diagrams, or models. A few exchange formats also address the meta-level, i. e. they allow for the definition of the elements of the data files. A recent trend has been to use XML for everything that has been done before without XML.

The CASE Data Interchange Format (CDIF) is a well-established format for the exchange of data—not metadata—between CASE tools. It does not address the exchange with tools other than CASE tools. With the popularity of object-oriented technology and UML it has been superseded by XMI.

The XML Metadata Interchange Format (XMI) is not a simple data exchange format but a *meta*-data exchange format. This implies its usability for data exchange. The OMG XMI standard defines two canonical generation mechanisms for both metadata and data. For a given domain—XMI has its widest use for the exchange of UML models—first a DTD (XML Schema is being worked on) is *generated* for the domain elements. In the case of UML this includes use cases, classes, objects, relationships, etc. A second step defines the generation of data files compatible with the DTD generated in the first step.

For the exchange of graphs several formats have been proposed. These formats are similar to formats for vector graphics in that they define nodes and vertices. However, the graphs used for tool integration are usually not diagrams but internal data structures, e. g. dependency and call graphs. The latest effort in this area is to define an XML-based format called Graph Exchange Language (GXL).

It has been found that reengineering has specific requirements not addressed by generic standard formats. Several attempts were made to define exchange formats for reengineering tools, e. g. for the exchange of syntax trees. For reengineering it is mandatory to

maintain all information so that the code can be regenerated. A typical parser of a compiler would eliminate much information, e. g. comments, literals, etc., and an exchange format for such syntax trees would not include these kinds of data. One example of a recent exchange format for reengineering tools is the FAMOOS Information Exchange Model (FAMIX) [Demeyer et al., 1999].

To perform a Smalltalk-to-Java migration all these exchange formats are of limited use. First, the native formats involved are Smalltalk and Java source code. There is no unified representation of both. Second, a higher-level representation such as UML could model the classes and methods of a program but would not be able to include code-level details. Also, the ability to query the code for senders and implementors of messages would be limited—or it would have to be reimplemented on the higher level. As I will explain later I want to use dynamic information about both the tools and the migrated program which could not be captured in a file-based format.

3.5.1.3 Message Protocol Standards

Two specific message protocols for tool integration have been mentioned already in sections 3.4.3 and 3.4.4: the CASE Communique and the CASE Interoperability Message Set (IMS). The former group was led by Hewlett-Packard, IBM, Informix, and Control Data; the latter by Digital, Silicon Graphics, and Sun Soft. The implementation of the first group was based on HP's SoftBench, the implementation of the second group was related to ToolTalk. Finally the work was merged and submitted as a joint draft to ANSI X3H6.

The X3H6 standard was never finalized. At the same time the CORBA standard was developed and the similarities between the two were obvious. Although I did not research the historical details, my impression is that much of the tool integration work was submitted to the CORBA standardization process.

Much later, in 1999, a tool integration standard based on CORBA was defined: the CORBA Meta Object Facility (MOF). This happened at a time when support for CORBA was already decreasing. The MOF was primarily geared towards tool access to repositories but many tools lacked CORBA interfaces. A companion of the MOF is OMG's XML Meta Data Interchange (XMI) standard mentioned earlier. Using the same concepts as the MOF, XMI can be seen as a file-based variant of the MOF.

3.5.1.4 Component Models and Middleware

Standards that we today consider generic middleware (e, g. RPC, CORBA) and component models (e. g. OpenDoc, COM) have been viewed by many authors in the 1990's as tool integration technologies. For example, the similarity can be seen in standards like PCTE which included services for object management, transactions, etc. Such services are typical features of middleware of component containers. They can, of course, be used for tool integration but then they operate only the syntactic level of tool integration, see section 3.3.2. The semantic level is addressed only by the standardization efforts described in the previous section.

3.5.2 Tool Integration Frameworks

Tool integration frameworks have developed a life of their own separate from a specific environment. The first thing to note is that there are only few tool integration frameworks. It seems that they are conceptually too close to generic middleware to justify an existence of their own. The tool integration frameworks presented here are all based on message passing. They can be seen in line with the architecture of the Field environment, see section 3.5.3.1, and demonstrate the popularity of the message passing approach.

3.5.2.1 Software Bus (Polylith)

The Software Bus is an abstract concept. It was proposed by [Purtilo, 1994] and realized in the Polylith implementation. The major difference to the other message passing approaches is that the bus is not centralized. Tools communicate point-to-point via named bus channels [Barrett, 1994, p. 12]. The bus architecture is very similar to the CORBA architecture and Polylith can be seen as a predecessor to CORBA.

3.5.2.2 Broadcast Message Server (HP SoftBench)

The Broadcast Message Server (BMS) is part of the Hewlett-Packard SoftBench product [Cagan, 1990]. SoftBench includes an editor, a static analyzer, a debugger, a program builder, and a mail tool. BMS is a direct offspring of the Field environment, see section 3.5.3.1, and is at the heart of SoftBench—like the Msg server in Field. The major addition of BMS compared to Field is a message *protocol*.

3.5.2.3 ToolTalk

ToolTalk, which is used by Sun Soft in the SPARCworks IDE, is Sun's corresponding product to BMS/SoftBench. It is described as a network spanning, inter-application message system and was initially implemented on top of Sun Soft's ONC RPC. The design of ToolTalk is more object-oriented than BMS.

3.5.3 Environment Implementations

In this section I present a few typical programming environments. The focus is always on the tool integration strategies. The environments include Field and PCTE for the historical perspective, Smalltalk and Eclipse (as a typical Java IDE) because of their relevance for a migration. SoftBench was already mentioned in section 3.5.2.2.

3.5.3.1 FIELD

Field, the Friendly Integrated Environment for Learning and Development, integrates various UNIX tools and provides a number of tools for software visualization. The major contribution of Field is the message passing approach to tool integration [Reiss, 1990]. Field has

a central message server called Msg. Tools register with the server if they want to receive messages. All messages are sent to the Msg server and then broadcast to all tools that are interested in the messages. Reiss showed that this approach is fairly simple to implement and provides a huge benefit. He wrote a custom socket protocol to connect the tools with the Msg server and used formatted plain text messages.

Previous tool integration approaches focused on data integration while Field focused on control integration. Data integration was limited in Field at first and was improved in Reiss' Desert environment which uses message passing to share fragments of data between tools [Reiss, 1996]. Another interesting aspect of Desert is that it provides an interface to ToolTalk, see section 3.5.2.3. Further features include asynchronous messages, filtering, and auto-invocation of tools.

Field played an important role in the development of commercial products. Many companies took Field's architecture as a blueprint for their own implementations and thus demonstrated the power of the message passing approach. e. g. DEC's Fuse environment was based on Field. Message passing was used in SoftBench, IBM Workbench/6000 and Sun Soft's ToolTalk.

3.5.3.2 Portable Common Tool Environment (PCTE)

The Portable Common Tool Environment (PCTE) started as an ESPRIT project in 1983 to specify and prototype *public tool interfaces* for software engineering environments. The first publication in 1986 (version 1.4) included C language definitions for the tool interfaces (Ada interfaces in 1987). The Independent European Programme Group (IEPG) of NATO split from the ESPRIT effort to extend the specification for military use and create its own derivate called PCTE+. The major additions were in the areas of operating system independence and security. e. g. PCTE+ specified 11 kinds of locks and 20 types of access rights [Cuthill, 1994]. In 1990 all PCTE efforts were jointly submitted to the European Computer Manufacturers Association (ECMA) for standardization. The ECMA PCTE standard was published in 1990 and bindings for C and Ada followed in 1991. In 1994 ECMA PCTE was also accepted as ISO/IEC 13719 standard and a second edition has been released since then. I refer to the 4th edition of the ECMA standard [ECMA, 1997].

PCTE is a specification, not an implementation. The specification is language-neutral but—at least originally—not platform neutral. The specification was decidedly compatible with Unix System V operating systems to ease tool migration. Platform dependencies were reduced by the PCTE+ branch. The abstract PCTE specification of tool interfaces defines a programmatic interface of services that tools can invoke. PCTE is abstract in the sense that the tool interfaces are not targeted towards a specific programming language. The specification is accompanied by concrete language mappings of the interfaces to the C, Ada, and IDL.

The PCTE specification is split into two volumes. Volume 1 includes six main service areas. Volume 2 deals with the user interface services. This part is based on the X Windows system and only relevant for UI programmers. Since I'm interested in tool integration I will focus on the services specified in volume 1.

The six service areas of PCTE are Object Management Service (OMS), Execution (EXE), Inter-Process Communication (IPC), Activities (ACT) (concurrency control), Communica-

tion (COM) (file input/output), and Distribution (DIS). PCTE+ added Security (SEC), Notify (NFT), and Accounting (ACC) [Boudier et al., 1989].

In PCTE all data used by tools was treated as objects. These objects are managed by the OMS. PCTE objects like files, pipes and processes are large-grained objects. The specification initially fell short on support for fined-grained objects and object-orientation in general. These limitations have been addressed by a *fine-grain data extension* and the *object-oriented extension* (ECMA-227 and ECMA-255) in 1995 after ECMA and ISO have standardized PCTE. Both extensions are included in the fourth edition of the standard to which I refer.

PCTE supports user defined schemas for the data of the tools. Schemas can be defined using a entity-relationship-attribute data model. Some schemas are predefined and available by default, e. g. for the objects defined by PCTE. The standard chose a self-representation approach in which not only the tool data but also the environment objects are treated as objects in the sense of the OMS. User defined schemas provide flexibility on one hand but on the other hand this leaves much details open. PCTE provides the platform for the exchange and management of tool data but the semantics are left open. [Cuthill, 1994, p. 192] argues that PCTE reaches the level of syntactic tool integration but not semantic integration as defined by Brown and McDermid [Brown and McDermid, 1992], see table 3.2.

The support for control integration in PCTE is limited. As [Cuthill, 1994] states, “triggers, message queues, and pipes provide limited control integration that is available only to tools running as PCTE processes, which limits their usefulness.” This limitation excludes all *foreign tools* from participation in control integration. A foreign tool that needs to exchange files with PCTE would be integrated by representing the file as a PCTE object. The object would only contain the metadata about the file, e. g. the file name, but not the contents. The foreign tool then would have to be started using an OS process; after termination the PCTE environment can collect the returned data in the tool’s output file. See [Long and Morris, 1993] for more details on using foreign tools in PCTE.

Another criticism of PCTE is that it includes to much operating system functionality. e. g. process management is a typical operating system functionality, object management a database functionality and user interfaces are addressed—at that time—by the X Windows system.

An interesting result of the ECMA specification was that, as a byproduct, a reference model was produced, see 3.2.2.2. The reference model is not specific to PCTE.

Many implementations of PCTE were attempted. At first the PCTE community was split between the original PCTE, PCTE+, and ECMA PCTE. The ECMA standard was released in 1990 [Long and Morris, 1993] note that only few implementations have been realized. To date there are no established implementations available.

However, many other attempts to build environments used the experience from PCTE. One interesting project is documented by [Gautier et al., 1995]. Their experience was that tool integration in PCTE was rather difficult and their way around it was to make their tools scriptable using an embedded Tool Command Language (TCL) interpreter. Their tools were able to communicate by sending script to one another. I will return to this idea later.

3.5.3.3 Eclipse

Eclipse is an open source Java IDE based on the IBM WebSphere Application Developer product. It is often called a workbench or a platform because the scope of Eclipse is wider than being a Java IDE. Environment services are split into platform services and language-specific services, e. g. the Java Development Tooling (JDT). Eclipse is –what I would call— a second generation Java IDE. The first Java IDEs were direct spin-offs of C++-IDEs and were usually written in another language than Java because Java was not mature enough at that time.

Eclipse, and for example NetBeans, are Java IDEs written in Java. This allows a developer who is proficient with Java to write his own extensions for Eclipse. To support this Eclipse can host plug-ins and has special tools to support plug-in development.

The architecture of Eclipse is that of a workbench as described in section 3.4.2, i. e. Eclipse is a monolithic Java application that provides hotspots for extensions. Eclipse plug-ins must adhere to the Eclipse plug-in API and if they do so, they are well integrated into the platform. The designers of Eclipse describe five levels of integration for tools: 1. none, 2. data, 3. invocation, 4. API, and 5. user interface [Amsden, 2001]. Though rigid, the fifth level of user interface integration is reached, as described in section 3.3.1.3.

However, if foreign tools cannot be adapted to execute within Eclipse, they are hard to integrate.

3.5.3.4 Smalltalk

Smalltalk is both a language and an environment as the title of the landmark book by Goldberg and Robson [Goldberg, 1984] suggests. The Smalltalk environment is often over-looked, especially when talking about migration. Smalltalk is not only just another language like Cobol, C, or Pascal—the environment has to be considered as well.

In this section I focus on two issues. The first focus is tool integration in the Smalltalk environment. To explain this it is necessary to describe some aspects of the language and the development/runtime environment. Smalltalk has some unique features that can be problematic in a migration. On the other hand, my second focus is the potential to actively use these advanced features to help in a migration.

Smalltalk programs run inside a Virtual Image that is executed on top of a Virtual Machine (VM). The image is saved when a developers exits the Smalltalk environment. All objects in the image are restored when the environment is reopened. This includes instances of all classes, and obviously also all windows. Smalltalk has a meta-level architecture. This means that the elements of a Smalltalk program like classes, methods, etc. are present in the image as objects. Unless this is used for meta-programming—it is not mandatory—I prefer to call this the *self representation* of programs in Smalltalk.

Tool integration is very simple in this scenario. Since all Smalltalk code is (self-)represented in the image in the form of meta-objects, tools can directly use these objects as they would use any other objects. There is no difference between “normal” objects and meta-objects.

The definition of Smalltalk programs is imperative and uses the objects present in the image. This should not be confused with saying a language like Prolog is declarative.

Smalltalk programs are imperative but their definition is imperative too. This means that a Smalltalk program is not a static text that is compiled into bytecodes but a program itself. The definition of a Smalltalk program that defines e. g. a class with instance variables and methods consists of Smalltalk code that calls the compiler with pieces of the class definition and asks it to create a new class. Then the compiler is called with other pieces of the program definition and asked to create corresponding methods. A consequence of this style of program definition is that the meaning of a program can never be determined statically. The meta-objects in the image might have been modified to support a different exception handling mechanism or transparent persistence. Then the loaded code would use these modified features, otherwise not. The Smalltalk ANSI Standard is the first approach to define a declarative syntax for Smalltalk [Wirfs-Brock, 1996].

Smalltalk exists in a number of dialects. The dialects are different enough that applications can not easily be ported from one to another. One difficulty are different source code formats for file-outs. These formats differ because the dialects provide different language features, e. g. class instance variables—or not, and these differences are reflected in the file format. A second difficulty is the imperative definition of Smalltalk programs. To have the same meaning in different dialects, this requires a program to use only those objects for its definition that are common among dialects. However, the object models of Smalltalk dialects are not necessarily the same.

In Smalltalk there is no difference between runtime and “development” time (or compile time). Development takes place in the image that is also used for deploying a Smalltalk application. Only the development tools are removed from the image prior to the deployment.

Let me summarize what this means for tool integration and for migration. Tool integration is very easy for tools that “live” in the image but hard for external tools. For a migration this means that it is best to work with Smalltalk code inside the image rather than with an external tool. Outside the image Smalltalk code is harder to analyze and change because the full meaning of the code is lost due to the imperative definition.

3.6 Summary

I presented data, control, and presentation integration mechanisms (3.3.1) and argued that presentation integration is not strictly required for a migration; data and control integration are essential. There are five levels of tool integration (3.3.2) but most integration mechanisms, implementations, and standards do not reach the level of semantic tool integration.

None of the integration standards was successful. Only technologies which addressed integration in a generic, not a tool-specific way, were successful, e. g. CORBA. The reasons for the failure of tool integration technologies are hard to analyze. In my opinion it is a mixture of too little ease of use and too much focus on the tool domain. However, many useful concepts have been established.

State-of-the-art environments provide tight integration of tools that run inside an environment but integration with external tools is generally quite limited. This is the case especially for Smalltalk and Java environments. For these environments, standards play a marginal role—the design of the environments is usually proprietary. Development environments

are much more open today and provide—proprietary— extension mechanisms to integrate third party tools.

An interesting move was that of IBM from VisualAge/Java to Eclipse/WebSphere Application Developer. The monolithic, repository-based environment was discarded in favor of a file-based model (using CVS). Also several CASE tools moved away from repositories and instead use files, e. g. Together (“round-trip engineering”). These single source tools are a step back in history to the beginnings of the Unix environment. However, this approach works well and given the fast workstations today, frequent synchronizations with the file system will hardly be an issue.

Chapter 4

Thesis

This chapter is the central point of this dissertation. In chapter 2 I identified the problem of insufficient tool support for software migration. The root cause is that the tools a developer has to use are not well integrated (2.5.3). In chapter 3 I reviewed tool integration technologies and concluded that none of these technologies can be applied in the migration scenario (3.6).

In this chapter I state my thesis (4.1) and list the constraints of my work (4.2).

4.1 Thesis Statement

First I state my thesis. Below I explain what each keyword means.

> *Programming Environments can be integrated by message passing. This integration supports a piecemeal platform migration process.*

Migration Migration is the act of transferring an application from one *platform* to another so that the migrated application exhibits exactly the same behavior as before (2.1).

One can choose between different *migration paths* (2.2). The source-to-source migration path is the most desirable because developers will be able to understand and maintain the migrated system.

Platform The platform of an application consists of four components: a language, a class library, tools (development environment, IDE), and a runtime environment.

The platforms I consider are Smalltalk and Java (2.1.1 and 2.1.2). Note that both languages are interpreted and therefore the interpreter, the virtual machine, is part of the platform.

Support Tool support relates development processes and tools. A development process requires tools which support it, and tools provide functionality that supports a process. Where tool support is missing, a developer has to adapt his process to the process which his tools support.

Migration Process A migration project can follow a waterfall, an iterative, or some other kind of process (2.3). I concluded that the typical tool-set does not support any productive migration process well (2.5). A productive process would be a *piecemeal* migration process.

Piecemeal Process A piecemeal[1] migration (2.5.2) is performed by migrating an application piece-by-piece. Rather than rebuilding the migrated application from scratch, each piece of the application is *replaced* by the migrated piece. Figure 4.1 shows the piecemeal migration where migrated pieces are integrated into the original application.

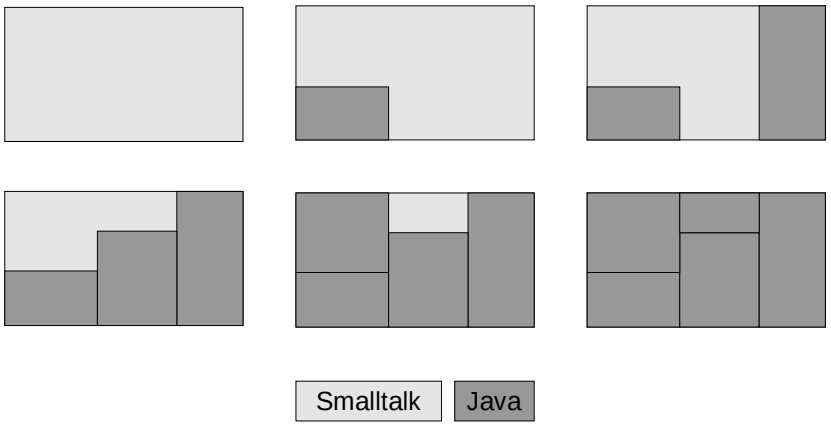

Figure 4.1: Piecemeal Migration

The benefit of a piecemeal migration is that the hybrid system is executable at all times during the migration. It is available for inspection, debugging, testing, and other activities. This improves a developer's productivity compared to rebuilding the application from scratch.

However, existing tool-sets do not support a piecemeal migration because the tools (2.4) are not well integrated (2.5.3). Providing this support is the topic of my thesis.

Integration With existing tools it is hard to perform a piecemeal migration because the tools are not well integrated. The Java and Smalltalk programming environments provide some integration already, see figure 4.2. The development environments (IDEs) are well integrated with the source code on each platform (S1 and J1). The Smalltalk development environment is perfectly integrated with the Smalltalk runtime environment—there is no difference between them. In Java the integration of development and runtime environment is achieved by the Java Platform Debugger Architecture (JPDA) [Sun Microsystems, 2002]. Migration tools can be integrated using a file interface (S2, J2).

What is missing is the integration of the two development environments and the two runtime environments (A, B). In my thesis I claim that it is possible to integrate the two platforms at both levels using message passing. Message-passing is a tool integration mechanism

[1]The name piecemeal migration is inspired by Richard Gabriel [Gabriel, 1996] who described two desirable characteristics of software: habitability and piecemeal growth. He illustrates these qualities using the metaphor of a comfortable farmhouse that evolves over time. Furthermore this name avoids confusion with other incremental processes.

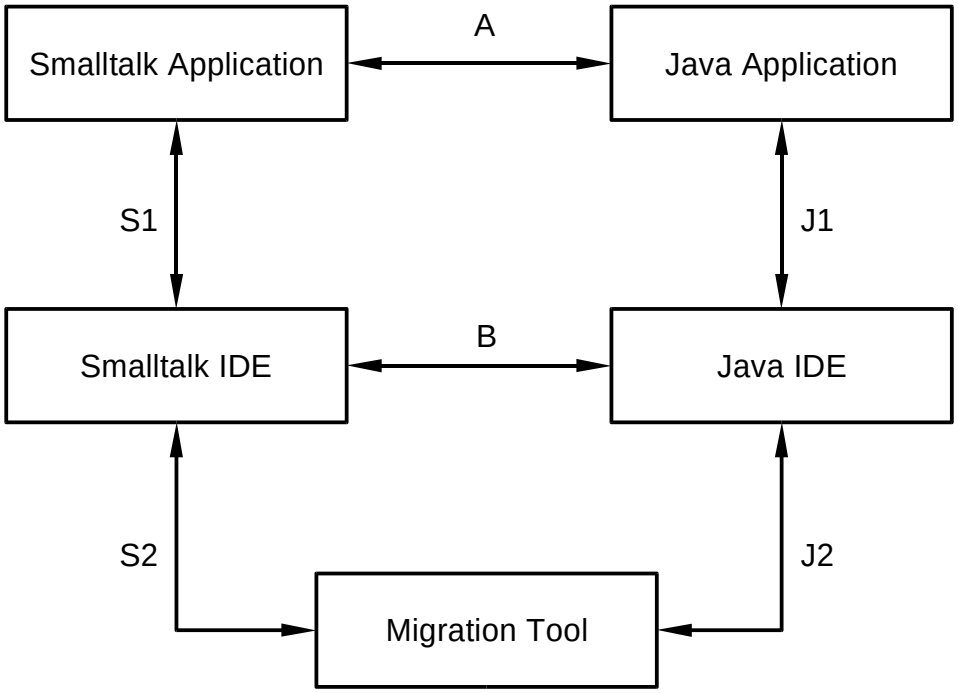

Figure 4.2: What to integrate

designed for the integration of tools *within* a development environment, see sections 3.3.1.2 and 3.5.3.1.

The new aspects of my integration approach are that two *integrated* development environments (IDEs) are integrated at a new level with each other, and that both the development and the runtime environment are integrated (A, B). In fact, the environment integration does not need to differentiate between the two levels of integration. The integration can be as seamless as in Smalltalk. Figure 4.3 shows the two programming environments and a single line representing the integration.

4.2 Constraints

There are two kinds of constraints: the requirements for the problem space (as described in section 2.5), and the aspects of the problem which my solution (5 and 6) does not address.

Constraints on the Problem My focus is the migration from Smalltalk to Java. The programming environments of these platforms have a number of features that my solution will be based upon. My thesis requires the existence of programming environments with the following characteristics.

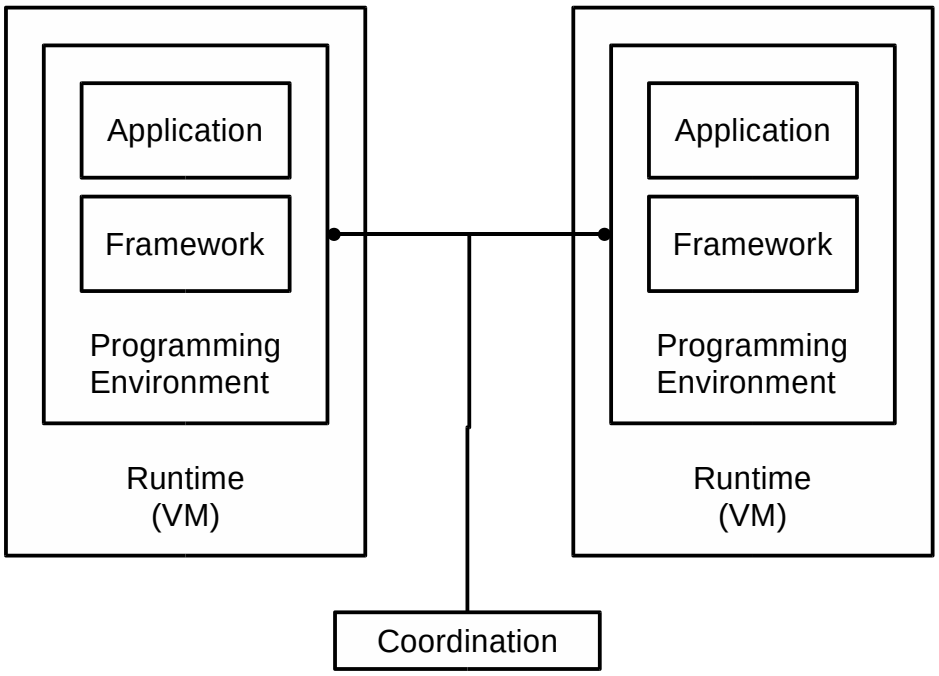

Figure 4.3: New tool integration

Openness A programming environment has to provide some kind of extension mechanism (“plug-ins”). It is not sufficient to have tool integration within an IDE if the integration mechanism is not open for third parties to add features.

Interpreter Because Smalltalk and Java are both interpreted languages I did not research the possibility of applying my solution to statically compiled languages like C/C++. Reiss [Reiss, 1990] implemented message passing for an IDE (FIELD) written in C/C++. His work is similar to my integration of the development environments (B in figure 4.2). However, I believe that the integration of the *execution* environments (A in figure 4.2) would not be flexible enough in a statically compiled language. Since I did not investigate this possibility, I limit my thesis to interpreted environments.

Hot-Swapping I require the execution environments to be able to reload modified code at runtime. This is true for most interpreters but not necessarily. Hot-swapping allows a developer to fix bugs when the application is halted, e. g. at a breakpoint, *without* having to restart the execution after modification of the application.

Reflection Because my solution will make some changes to the development environments and many changes to the application’s code, I require the environments to have a self-representation of itself (3.2.1.3).

Availability of Specific Tools My research is focused on the integration of the programming environments. My goal was not to implement yet another specific migration tool. Many small and useful tools exist, e. g. for syntax conversion, type analysis, pro-

gram understanding. I assume that a developer will have some of these tools (or he might write or port them himself).

Note that I do not require the programming environments to be Smalltalk or Java environments. I list here only the characteristics I rely on. The proposed solution can be applied to other platforms meeting these requirements, e. g. LISP/CLOS.

Constraints on the Solution Because the focus of my work is the migration from Smalltalk to Java, my thesis does *not* address the following aspects.

Team Development The two integrated programming environments are to be used by a single developer.

Versioning For versioning of code in the development environments my solution relies on the versioning functionality of the respective IDEs. There is no support for integrated versioning across platforms.

Schema Evolution Whenever a class is modified which has instances, then the old instances will be out-of-sync with the current version of their class. This is a common problem of dynamic code modifications. In Java this results in a “cannot reload class” error. If that happens the application has to be restarted.

Native Code The piecemeal migration process does not address migrating native code—only Smalltalk to Java.

Multi-Threading An application can be multi-threaded. However, a developer is expected to work only with one thread at a time.

Chapter 5

An Integration Architecture for Programming Environments

The *integration architecture* is the infrastructure for the integration of two programming environments. It facilitates the piecemeal migration process that I will describe in detail in chapter 6.

Here I present how the integration architecture works. The presentation starts with a design rationale which takes into account the constraints of the problem (5.1). I give a high-level overview of the architecture (5.2) and continue with details on the components used (5.3). The second half of this chapter consists of a description of the implementation of a prototype that I developed to demonstrate the feasibility of the architecture (5.4).

5.1 Design Rationale

The design rationale explains *why* the design presented here has been chosen. Understanding the rationale helps in understanding the final outcome of a design process. The integration architecture embodies several design decisions and the design rationale explains the motivation for these decisions.

First of all, the design of the integration architecture is limited by the constraints of the problem domain (4.2). The main constraint is that of limited resources. In chapter 2 I argued that migration projects are difficult and expensive—also the development of migration tools. The integration architecture attempts to be *simple* so that it does not require a large effort to implement it.

A second design goal is to *reuse* existing functionality where possible. Modern programming environments provide a wealth of features. Many of them create new opportunities for implementing migration tools. The integration architecture attempts to seize these opportunities by using both powerful concepts and existing implementations.

5.1.1 Choice of Platforms

My research started by investigating the migration from Smalltalk to Java in general. Because I planned to demonstrate the feasibility of the integration architecture by implementing a prototype, I had to choose concrete platforms.

For the source platform I selected VisualWorks Smalltalk. It is the most popular Smalltalk dialect and a direct descendant of the original Smalltalk-80 system. The version of VisualWorks is itself an issue. Earlier versions provided better support for refactoring (versions 2–2.5). This would have been useful for changing the original code. However, the introduction of namespaces broke the refactoring tools (version 3) and full support for refactoring has been included only in the latest version (version 7).

For the target platform I use the Java Development Kit 1.4 (JDK). Although I started this research when version 1.0 was released, several major improvements have been made since then. JDK 1.1 provided better reflection capabilities, JDK 1.3 introduced support for dynamic proxies, and the current JDK 1.4 includes the Java Platform Debugger Architecture (JPDA) which allows changing classes at runtime (hot-swapping) [Sun Microsystems, 2002]. It is interesting to note that the versions of the platforms make a big difference not only for an application developer but also for a tool developer. Both benefit from improved functionality of a platform.

As I will explain later, I connect the two programming environments via an *Integrator* component that I put in between the two environments. For this Integrator I use Dolphin Smalltalk, not VisualWorks. I use a different Smalltalk dialect here for two reasons. 1. Since I implement the migration-specific tool support (class mapping, syntax conversion, etc.) in another Smalltalk dialect I show that this functionality is independent of the source environment and does not rely on residing in the same environment. 2. By separating the Integrator from VisualWorks I achieve more generality of the solution. The VisualWorks part of the prototype is in fact quite small compared to the Dolphin Smalltalk part. This allows me to easily adapt my implementation to another source platform, e. g. IBM Smalltalk.

5.1.2 Blind Alleys

It was clear early on that I needed a way to communicate between the two programming environments. I investigated the use of communication facilities, e. g. the Object Management Group's (OMG) Meta Object Facility (MOF). The intended use of the MOF is to provide a CORBA-based communication infrastructure for heterogeneous tool-sets consisting of IDEs, CASE tools, repositories, testing tools, database tools, etc.

To date there are only a small number of products which support MOF. It turned out that using CORBA—while it may be the most universal communications technology—was too difficult to handle and provided solutions for problems that did not exist in this context. e. g. simply the presence of CORBA services for naming, life-cycle management, security, persistence, etc. made using CORBA difficult—even without using all these services.

A very similar standard, XML Metadata Interchange (XMI), gained much more popularity. XMI is a generic approach to define DTDs for XML-files that hold the data of modeling tools, i. e. metadata. XMI is not intended as a replacement for MOF since it is not a communications mechanism. Of course it would be possible to exchange XMI data using

some middleware but this is beyond the scope of the XMI specification. The intended use of XMI is to exchange the data using files.

Another disadvantage which applies both to MOF and XMI is that they are geared towards metadata of modeling tools. What I need to exchange in my integration architecture is metadata about run-time environments and running programs.

Although I planned to use CORBA and XMI in my implementation, I finally decided against it. The main drawback of CORBA is that it requires static interfaces described in the Interface Definition Language (IDL). It supports dynamic invocations as well, but those do not allow for the easy execution of scripts which are not bound to a class. The reason not to use XMI is that I do not exchange metadata a lot, especially not different kinds of metadata. The metadata that I exchange—classes—is easier to handle implicitly in the code because their format does not change.

Yet another technology that I intended to use and dropped is XML—and SOAP. XML seemed to be a good candidate for transmitting data between Smalltalk and Java. The Simple Object Access Protocol (SOAP) specifies the format of messages for transmitting data. Even more interesting is that SOAP provides an encoding of different data types in XML. A Smalltalk implementation of SOAP maps Smalltalk types to the SOAP XML encoding, and a Java implementation of SOAP can decode the data into Java data types. What SOAP does not provide is a transport layer. SOAP messages can transmitted over HTTP, SMTP or other transports. However, my prototype contains a `SyntaxConverter` which can not only convert classes but also data from a Smalltalk to a Java representation. This, and the missing transport, are the reasons why I chose not to use XML and SOAP in the implementation of the prototype.

5.1.3 IDE Features Used

The IDEs have to provide an incremental compiler that allows the modification of code at runtime. I use meta-programming to make changes to the code (code instrumentation). The incremental compiler avoids the problem of restarting the system after each code modification.

The IDEs have to be somewhat open, e. g. by providing an Open API as in NetBeans, a Tool Integrators API as in VisualAge/Java, a plug-in architecture as in Eclipse, or by having a white-box architecture (like Smalltalk). The openness of the IDEs does not provide an integration mechanism to be used between IDEs but allows me to put one in. IDEs provide a mechanism to integrate other tools into them. My idea is to use this mechanism to open the IDEs even more so that a true integration between peer IDEs becomes possible.

The IDEs have to support the interpretation of code. This means that code can be executed dynamically. This is a little bit more than an incremental compiler. It means that an IDE can load (compiled) code dynamically into the current execution environment and run it. For Java this is tricky—JDK 1.4's debugger architecture permits this with some limitations. A Java interpreter can be run together with the application on the same VM to remedy this problem.

The development environment and the runtime environment should—ideally—be the same. This is the case in Smalltalk, but not in Java. In Java, my prototype uses some workarounds to make the development and the runtime environment appear as one (5.4.2.2).

5.1.4 Communications Mechanism

Above I stated that CORBA provides many services that I do not need but make using it complicated. My alternative presented below could of course be implemented using CORBA. My point is only that it is not worth the effort.

My requirements for the communication mechanism are as follows. I require two connections, one from Smalltalk to the Integrator (also Smalltalk), and one from the Integrator to Java. Both connections have to be bidirectional so that both parties can send a message to the other. A response for every message is mandatory—although a client can choose not to wait for a response.

The implementation of my prototype uses sockets to send text messages between the components. The messages consist of any syntactically correct Smalltalk or Java expression that the receiver can evaluate. This approach naturally supports sending of simple parameter values and sending of command scripts which are independent of a class. Return values are Smalltalk or Java expressions as well. Although this approach may seen simplistic at first it is very powerful. It frees the tools from dealing with static interfaces which is a large benefit because during a migration the classes on both platforms are changed frequently. However, this communication mechanism is not suited for developing applications where security, transactions, etc. are required.

5.2 High-Level Overview

The main idea is that two programming environments are connected so that during a migration a developer can work on the whole code base consisting of original and migrated code in an integrated way.

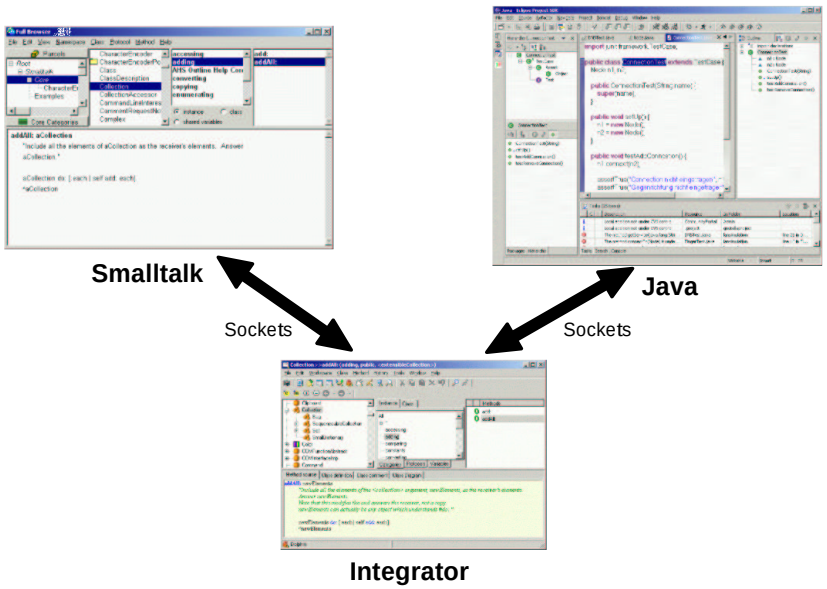

Figure 5.1: Two programming environments are connected

The architecture consists of three major components: the two programming environments of the source and the target platform, and an additional tool called the *Integrator*.

The programming environments represent both the development and the run-time environment of the application. They also provide a wealth of functionality like parsers, code analyzers, etc.

Both IDEs are open. This openness is used to embed a small component, a *Connector*, into both IDEs. A Connector provides the ability to communicate with the Integrator. The Java Connector is embedded into Eclipse using its plug-in architecture. In Smalltalk the code is so simple that it can be run directly from a workspace. Note that Smalltalk is a white-box framework—no explicit interfaces are necessary.

The Integrator is the component where migration-specific functionality is located. This includes loading code from the original environment, syntax conversion to the target language (possibly other transformations), and installing converted code in the target environment.

The Connectors communicate with the Integrator over sockets. The communication is typically initiated from the Smalltalk environment or the Integrator to issue some commands. There are different kinds of messages such as commands, events, parameter values, classes to be migrated but all messages are free-format. The only requirement is that the message is correct Smalltalk or Java so that the receiver can evaluate it. Messages may include other encodings, e. g. XML or BASE64, within the correct syntax. A script is free to choose whatever return format it wants to produce. The result can be binary data, strings, XML, or anything else. The important thing to note here is the flexibility of this approach: it is not necessary to define an API for the IDEs. On the contrary, the IDEs are completely accessible from the Integrator.

All messages are routed through the Integrator. In some cases the Integrator forwards a message to the receiver without changing anything. Those cases could be optimized by sending the messages directly from the Smalltalk Connector to the Java Connector. However, sending everything through the Integrator is more simple without adding too much overhead. It allows the Integrator to perform some useful conversions and house-keeping of the messages it sees.

.

5.3 Building Blocks

I used and reused a number of building blocks to develop the integration architecture. First and foremost, this includes the two programming environments. Here I give more details about the other components that I used.

5.3.1 VisualWorks, Eclipse, and Dolphin Smalltalk

The choice of the three products, VisualWorks, Eclipse, and Dolphin, was motivated in the design rationale earlier (section 5.1). In chapter 3 I concluded that migrations are difficult because of the dependencies of an application to the class library of a platform.

My implementation uses VisualWorks, Eclipse (JDK 1.4), and Dolphin Smalltalk and thus is equally dependent on their base classes. In each environment I use the `Socket` classes intensively. Below I explain which other components I use.

5.3.2 Rdoit

In Smalltalk code can be written in a workspace, selected, and executed by selecting `doit` from the context menu. The selected part of a workspace is then evaluated by the Smalltalk compiler. Rdoit provides a remote `doit` over sockets. It was first written by Bill Voss for Smalltalk-80 in 1991 and intended for accessing Smalltalk remotely. I reimplemented `rdoit` for VisualWorks and Java—the reuse here is of conceptual nature, see figure 5.3. Using `rdoit` I can script an IDE from outside. I prefer to use the term scripting because it is more widely understood than `rdoit`.

5.3.3 Refactoring Browser

The Refactoring Browser (RB) provides a number of useful features—both concrete and conceptual. The RB was initially developed by Don Roberts and John Brant (University of Illinois at Urbana Champaign) for VisualWorks 2.x and was maintained until 3.0. It did not work with later versions of VisualWorks that introduced namespaces. In Version 7 it is included again and supports namespaces. The RB is open source and has been ported to many other Smalltalk dialects. There are also similar refactoring tools for Java. e. g. Eclipse, NetBeans, IntelliJ, and IDEA support some refactorings. However, refactoring is much harder to do in Java because of static typing; the tools support fewer refactorings.

The first application of the RB in a migration is, of course, to use the refactorings directly. They may be invoked interactively using the browser or programatically.

The RB uses its own representation of a Smalltalk program, the RB's *program model*. This is similar to an abstract syntax tree but differs in some ways. As one always needs the right tool for a job, the program model has special data needed for refactoring. Other than an abstract syntax tree (AST) of a compiler, the RB does can not ignore comments and does not optimize any constant expressions. Both reveal a programmer's intent and must not be removed after a refactoring.

The purpose of the program model is to allow for analysis and modification of a program without yet changing the real code in the image. Once it has been verified that a refactoring can be performed safely, the modified code will be installed in the image.

The RB has two interesting support tools. The first is the rewrite tool. The rewrite tool works on the program model and changes the program according to a predefined set of rules. Each refactoring has its own rule-set. There are also rules for code formatting. I use the rewrite tool for the syntax conversion to rewrite Smalltalk code in Java.

The other interesting tool is SmallLint. This is a typical lint tool for Smalltalk. Given the program model of an application it checks conformance to predefined structural conditions and warns the programmer if they are violated. The use in migration could be to find code dependencies and identify classes that have to be migrated together.

5.3.4 BeanShell

BeanShell is a Java *source* interpreter (www.beanshell.org). It should not be confused with a virtual machine which is a bytecode interpreter. BeanShell interprets Java source code at runtime without going through a compilation process. The result is that scripts can be written in Java and executed dynamically. BeanShell provides a Smalltalk `doit`—or a LISP `eval`—for Java.

BeanShell is used in the Java Connector, see section 5.4.2.2, to provide the same functionality as `Compiler evaluate: aScript` in Smalltalk. An important feature of BeanShell is that it can evaluate Java expressions outside of a class. This makes it very useful for scripting.

5.3.5 Java Platform Debugger Architecture

The Java Platform Debugger Architecture (JPDA) [Sun Microsystems, 2002] was significantly updated in JDK 1.4. It has now support for hot-swapping classes. This means that a class which has been loaded by a class loader already can be recompiled and *reloaded*. The old class is then swapped with the new class. This is a standard feature of Smalltalk since the early beginnings but in Java it has been added only in the current 1.4 release.

Hot-swapping allows dynamic code instrumentation. Java code can be changed at runtime without having to restart an application. To be more precise, this could be called debug-time instead of run-time because the application has to run a virtual machine (VM) in debug-mode for hot-swapping to work.

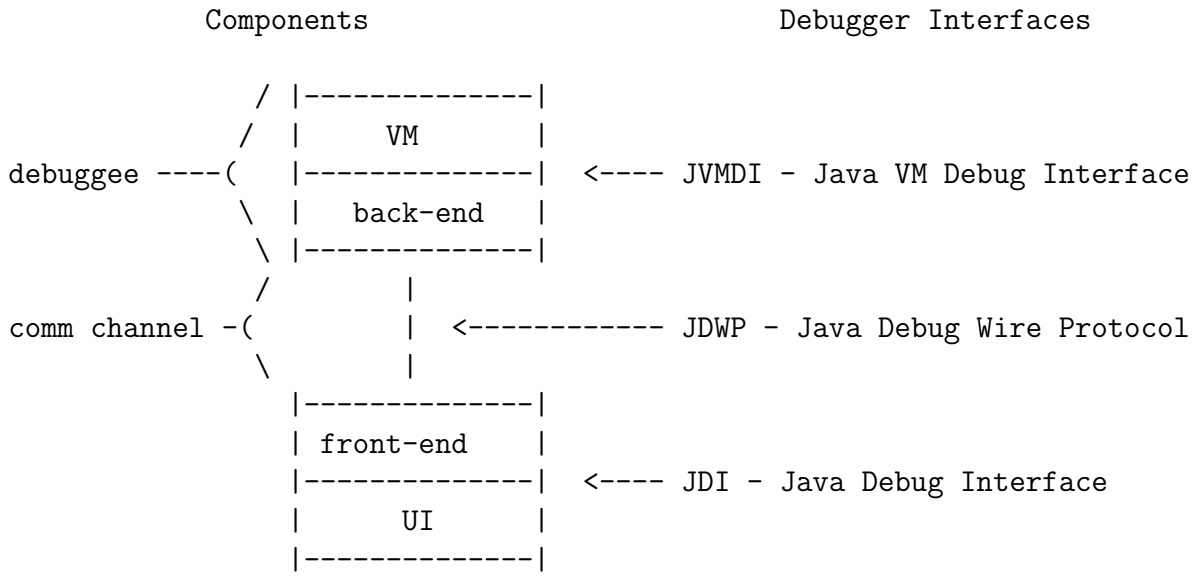

Figure 5.2: The Java Platform Debugger Architecture (JPDA) [Sun Microsystems, 2002]

The downside of the JPDA is that it always involves running two Java VMs. Hot-swapping works only for the debuggee, see figure 5.2. The debugger runs on a separate VM and communicates with the debuggee using the Java Debug Wire Protocol (JDWP).

The JPDA specification only defines interfaces for the involved components. The JDK 1.4 includes a sample client for JDWP. A stable implementation is included e. g. in Eclipse.

Using JPDA makes the integration architecture a bit more complicated than shown in figure 5.1. In fact the Java programming environment consists of two Java VMs running as a debugger/debuggee tandem. The debugger VM is the VM where the Eclipse IDE runs, and the debuggee VM is where the application runs. Using Eclipse for this tandem provides a stable and proven solution. However, the Integrator has to issue commands at first to the debugger process which in turn passes them on to the debugger using the JPDA implementation of Eclipse.

5.4 The Prototype

This section describes interesting aspects of my architectural prototype. It presents the major components, the Integrator (5.4.1) and the Connectors for Smalltalk and Java (5.4.2). The description of their collaboration (5.4.3) provides the full picture of how everything works together. Finally, I outline how to install, start, and configure the integrated programming environments (5.4.4).

I use the following naming conventions for the code and the diagrams shown in later sections. Since I'm dealing with migration I regularly have Smalltalk classes and their migrated counterparts. My naming convention ensures that the naming is consistent and that one can easily tell on which platform a class lives. Typically, I will have a class, say `Foo`, and migrate it to Java. I prefix the class name in Java with a `J` and consequently call the migrated class `JFoo`. If there could be any confusion with AWT or Swing classes like `JButton` then I suggest modifying this convention, e. g. by using underscores. Because there are no chances for a confusion in this document, I use the simple `J` prefix. Similarly, I use the `S` prefix to emphasize that a class is a Smalltalk class. However, I have to live with existing class names of the base classes. In cases where it does not make sense to change the name of base classes, I omit the prefix. If I omit a prefix in such a case I will always include the prefix in related classes. The premier example of a related class is a proxy class. e. g. if I generate a proxy for `Foo`, the proxy would normally be called `PFoo`. Since I omitted the prefix in `Foo` I include the `S` in the name `PSFoo`.

5.4.1 The Integrator

The Integrator is the central component of the integrated programming environments. Its main responsibility is to provide a bridge between the programming environments of the source and the target platform. Besides this, the Integrator is the home of all migration-specific functionality.

To provide the bridge between Smalltalk and Java, the Integrator creates connections to and accepts connections from each platform. The Integrator is multi-threaded; at the minimum two server sockets are waiting for incoming connections. For each call a separate thread is used, very much like in a web server.

All communication from Smalltalk to Java, and vice versa, is routed through the Integrator. This approach is similar to the `Msg` server [Reiss, 1990]. However, since I integrate

only two platforms, it does not make sense to use a technique like Reiss' selective broadcasting. This technique was useful for Reiss because he integrated more than two tools. Because I integrate exactly two programming environments, the destination of a message is always clear—it is the other environment. Routing messages through the Integrator is useful for performing migration-specific tasks. For example, when migrating a class to Java the source code for the class is sent through the Integrator and converted to Java at this point. Or, when sending parameter values, these values are converted to their Java representation, see below. If the Integrator passes messages without modifying them, then the routing could be optimized, e. g. see figure 5.7. In this example the Smalltalk Connector could call the Java Connector directly, and also return the result directly. To keep the implementation of the prototype simple I did not attempt to detect such cases.

The Integrator contains two components which provide migration-specific functionality: the `ClassMapper` and the `SyntaxConverter`. See section 2.4.1 on page 24 for more details on other migration-specific tools that could be plugged-in. The `ClassMapper` maps class names and method names of Smalltalk classes to corresponding Java classes and methods, see section 2.2.5 on page 21. Every time the `ClassMapper` cannot provide a mapping I investigate the case and add a mapping manually. It will be used automatically from then on. The `SyntaxConverter` is based on the Refactoring Browser. The Refactoring Browser provides a rewrite tool that allows one to define rewrite rules which are applied to syntax trees. In normal operation the Refactoring Browser first checks the preconditions for the application of a selected refactoring. If the preconditions apply, the rewrite tool refactors the code. In the implementation of my prototype I use the rewrite tool to rewrite the Smalltalk code as Java.

Two other important features are the generation of proxy classes and the conversion of values. Both are similar to the syntax conversion mentioned above—with some modifications. The Integrator can generate proxy classes for Smalltalk classes. This is similar to migrating classes, only without migrating the implementations of methods. The methods instead are generated to call the proxy's subject. The environment integration uses call-by-reference between the two environments per default, but for base types it uses call-by-value. Passing objects by value copies the values from one platform to the other. The copied values do not have the same identity as the original values. For base types, this is the case in Smalltalk and Java anyway. Passing them by reference would be wasteful. The Integrator converts the literal representation of base types between Smalltalk and Java. Most base types look similar on both platforms, e. g. `Sting`s and `Integer`s, but each platform provides different ways for special elements such as control characters or numbers expressed using different radices. The Integrator can also convert base types back from Java to Smalltalk—a feature required in some cases (5.4.3.3). Yet another minor functionality is logging of class and method name mappings. If the Integrator detects a name collision it prompts for a resolution. The actual mappings are remembered by the Integrator. This information is required for the generation of proxies for migrated classes.

The use of the Integrator will be described in detail in chapter 6 on the application of the integrated programming environments.

5.4.2 The Connectors

The integration architecture uses two Connectors, one in each programming environment. Because of platform differences, the Smalltalk and the Java Connectors have different implementations for the same features. First, I describe the commonalities, then the platform-specific aspects.

A Connector is a plug-in into an IDE. At the minimum a connector has one thread which waits for incoming requests from the Integrator. Outgoing requests are executed within the thread of the caller. The main functionality of a Connector is responding to incoming requests. The messages sent between a Connector and the Integrator are scripts, i. e. executable code. These scripts are evaluated upon reception and produce a return value which is a script itself. For simple return values, the script trivially evaluates to the return value, e. g. an `Integer` or `String`. The evaluation of the script can be seen in the Smalltalk code in the next section (`Compiler evaluate:`).

If a request references other objects, then some housekeeping needs to be done so that messages can be dispatched to the referenced objects. Messages to classes can be invoked directly because classes are referenced by name. Instances usually do not have names and the Connectors provide a `Registry` where anonymous instances can be remembered. For each object to be kept in the `Registry`, a Connector generates an object id (e. g. the hash-code if that is always available) and adds a mapping from the object id to the instance in the `Registry`. Proxies on the opposite platform know the object id of their subject. Calls to a subject include this object id which is resolved in the Connector.

5.4.2.1 The Smalltalk Connector

The Smalltalk Connector is extremely simple. It is very similar to rdoit (5.3.2). The Smalltalk Connector consists *only* of the code snippet shown in figure 5.3 which can be executed in a workspace. After running this code the Connector is ready for responding to incoming requests. During start-up of the integrated programming environments the Integrator will send further utility classes, e. g. the `Registry` mentioned above, to the Connector and *install* them.

There are two important utility classes: the `Invoker` and the `Forwarder`. I will describe the `Invoker` in the next section on the Java Connector because it requires a more difficult implementation in Java. The `Forwarder` is the component which dispatches requests to Java via the Integrator. Besides dispatching requests it has some convenience methods which compose meaningful commands for the Integrator. An example for this is `Forwarder migrate: aClass`. The Forwarder attaches the source code to this request and sends it to the Integrator. The Integrator then performs the migration of `aClass`.

5.4.2.2 The Java Connector

The functionality of the Java Connector is the same as that of the Smalltalk Connector. The Java Connector listens on a TCP/IP port for incoming scripts, executes them, and returns the result to the caller. However, the implementation in Java is more complicated because some approaches do not work out-of-the-box as in Smalltalk.

```
| socket childSocket connection stream |
socket := SocketAccessor newTCP.
socket listenFor: 1.

childSocket := socket accept.

connection := ExternalConnection new
  input: childSocket;
  output: childSocket.
stream := connection readAppendStream.
stream lineEndTransparent.

[
  script := stream throughEmptyLine.
  Transcript nextPutAll: line; cr; endEntry.
  ((line at: 1) = $.) ifTrue: [^nil]
  stream nextPutAll: (Compiler evaluate: script) printString; cr.
] repeat.

stream close.
```

Figure 5.3: The Smalltalk Connector

First, Java does not support the dynamic execution of Java code, i. e. there is no equivalent to `Compiler evaluate: aScript`. I work around this limitation by using the BeanShell interpreter. It provides a source interpreter for Java that does not require compiling scripts to bytecode. Another problem which BeanShell solves is the limitation of the Java compiler that it can only compile code that is part of a class—the smallest compilation unit is a class. This is also a limitation of the Java runtime; it can only execute code which is part of a class. In BeanShell the evaluation of a script looks like `(new bsh.Interpreter()).eval(aScript);`. In this example the interpreter is transient. It will be garbage collected after returning a result. The Java Connector keeps a reference to one BeanShell interpreter and reuses it across different evaluations.

Second, the Java IDE (Eclipse) is not as transparent as the VisualWorks Smalltalk development environment. In Smalltalk every class is held in memory, the virtual image, and any object can access any other object. Java has a file-based class model. A Java virtual machine (JVM) supports loading classes from different sources, e. g. from local disks, jar files, and web servers. Java classes are loaded by a class loader and this class loader protects the local machine depending on where a class was loaded from (sand-box). Classes loaded from a web server have lower permissions than classes loaded from the local disk. The point is that the JVM creates a hierarchy of class loaders—a tree—and classes loaded in parallel branches are not visible to each other. This is a powerful feature because it provides security and supports the use of different versions of a class in different branches of the class loader hierarchy. The Eclipse IDE uses this hierarchy to shield plug-ins from each other and to shield the basic workbench from plug-ins. If a plug-in requires access to classes of the workbench, the plug-in developer has to list these dependencies in the `plugin.xml` configuration file. As a consequence of this security approach, the Java Con-

nector is implemented as a regular plug-in into Eclipse. To make it accessible from the user interface, the Java Connector plug-in adds a pull-down menu to the menu bar for starting and stopping the server thread—the equivalent to the code in figure 5.3.

Third, the Java development environment and the Java runtime environment are not integrated as seamlessly as in Smalltalk. In Smalltalk there is no difference between them—application code is executed within the same virtual machine as the development environment. Although the same approach could be taken for a Java programming environment, a specific limitation of the Java VM prevents this. Exception handling in Java produces a stack trace *and* unwinds the call stack. In Smalltalk an exception raises a debugger window which presents the call stack. The difference is that in Smalltalk the call stack is not unwound. The interesting aspect is that in Smalltalk the root cause for the exception can be "repaired". After that, the execution of a program can be resumed. e. g. for out-of-memory, file-not-found, division-by-zero and other exceptions a developer can find the problematic code in the call stack, make changes in the operating system or the call stack to prevent the exception from occurring again, and then resume the execution. Because the call stack is unwound in Java, the virtual machine has to be restarted. If there were the same kind of integration as in Smalltalk for the Java development and runtime environment, then every crash of an application would require a restart of the development environment. Obviously this is far from useful and Java-based programming environments take a different approach. A Java IDE executes application code in a second VM which will be restarted after a crash—the VM of the IDE is unaffected by this. In early versions of Java a restart of the application VM was required also after classes were changed. The Java class loader was not able to detect changes and reload classes. Java 1.4 improved this by providing a standardized interface for accessing a runtime VM through a debugger, the Java Platform Debugger Architecture (JPDA) [Sun Microsystems, 2002].

The Java Connector receives two kinds of messages from the Integrator: 1. messages from the Smalltalk part of an application to the Java part, and 2. messages to the IDE. These two kinds of messages correspond to the lines A and B in figure 4.2 on page 63. Messages to the IDE do not need to be dispatched further and are evaluated directly by the Java Connector. Messages to the Java application need to be dispatched to the second Java VM where the application runs. This is shown in figure 5.4. In this design I moved the `Invoker` from the IDE's VM (left) to the application VM (right). The diagram shows first how a `new` message is dispatched to create a new instance of a class. This instance is registered by the `Invoker` so that it can be referenced later using an object id. The lower half of the diagram shows how a message is sent to this instance. The Java Connector dispatches the message to the Invoker. Technically the connection between the two Java VM uses the Java Debug Wire Protocol (JDWP). The Eclipse IDE provides a complete implementation of JPDA so that the use of JDWP is transparent to the Java Connector.

5.4.3 Collaboration

This section describes the collaboration of the Integrator, the Smalltalk Connector, and the Java Connector. I use three use cases to show how everything works together. The first use case is migrating a class, the second is instantiating a migrated class, and the third is invoking methods on an instance of a migrated class.

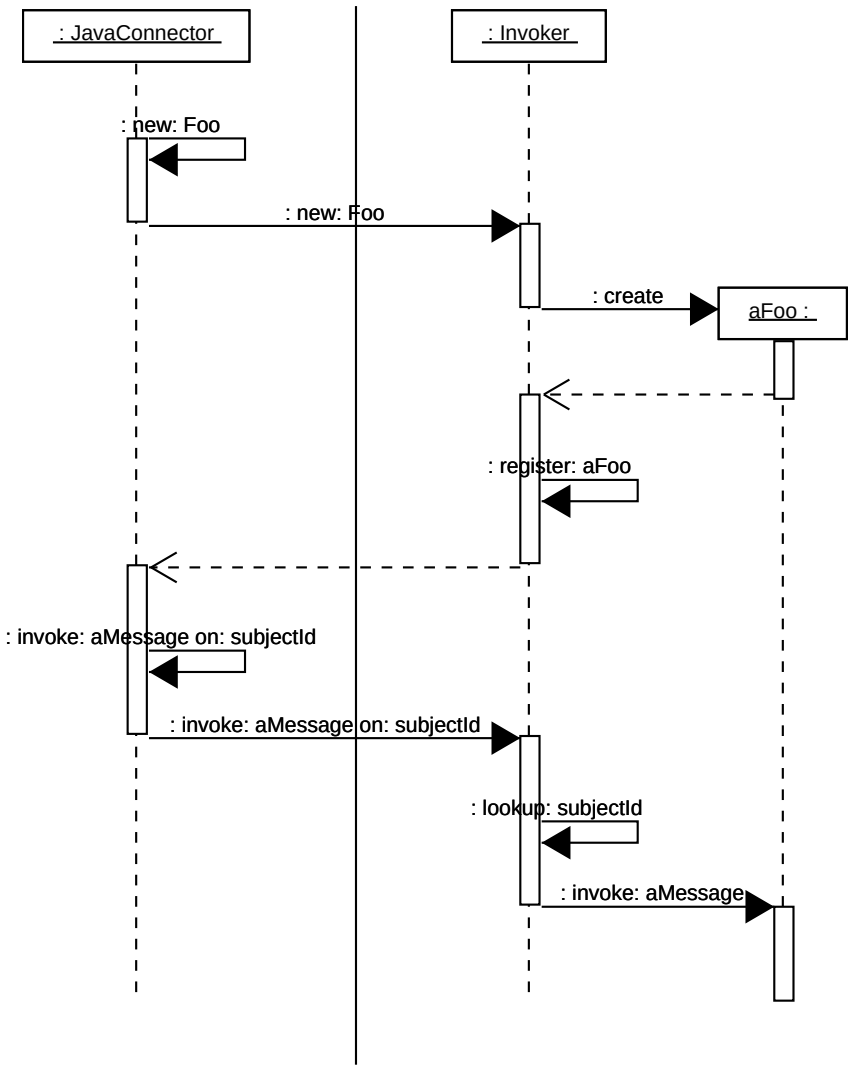

Figure 5.4: Integrating the Java VM of the IDE with the VM of the application

The UML diagrams in this section show the Smalltalk Connector on the left, the Integrator in the middle, and the Java Integrator on the right. Solid arrows are calls and dashed arrows are returns. For returns and instance creation messages I omit the label where no confusion can occur. The diagrams do not show the Java runtime integration that I presented in the previous section (5.4.2.2). One needs to imagine figure 5.4 being appended to the right of each diagram.

5.4.3.1 Migrating a Class

The migration starts with sending the message `migrate` to a class. The Integrator has sent a command to the Smalltalk Connector to install a `migrate` method in `Behavior` so that all Smalltalk classes understand this message. The Smalltalk Connector converts the class into the standardized Smalltalk Interchange Format (SIF). This format is standardized by ANSI in the X3J20 Smalltalk standard and it is used for exchanging Smalltalk code in `.st`-files. This SIF representation of a class is forwarded by the Smalltalk Connector to the

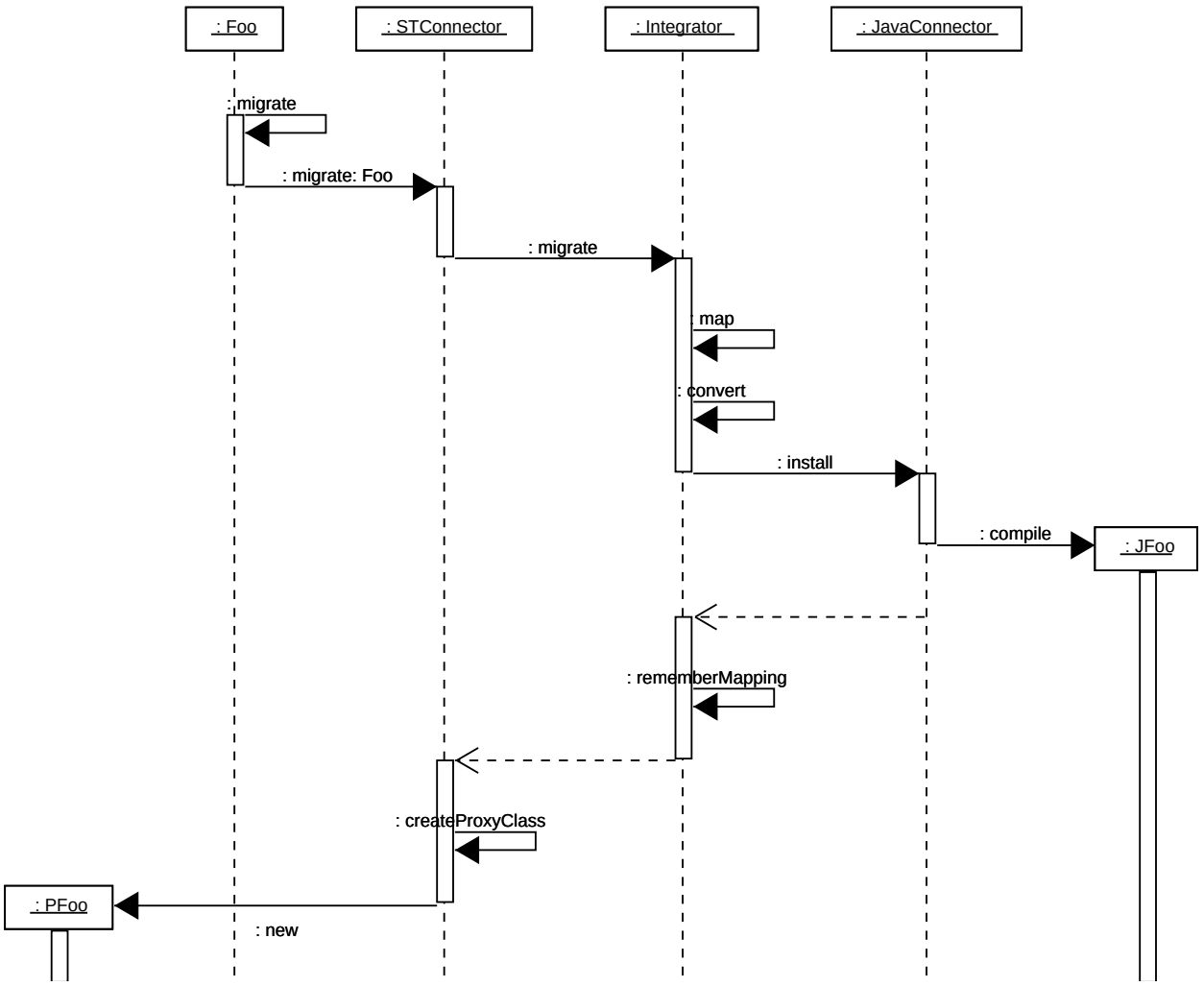

Figure 5.5: Collaboration of Connectors and Integrator

Integrator where the migration to Java will be done. The Integrator first maps the class and method names according to the mappings of the `ClassMapper` to Java names. Then the `SyntaxConverter` converts the class to Java. This Java class is sent to the Java Connector for installation. The Java IDE compiles the class and success or failure is reported back to the Smalltalk Connector. After a successful compilation, the Smalltalk Connector creates a proxy class for the migrated class.

Note that a whole class is migrated at a time because a class is the smallest compilation unit in Java. Also note that unlike instances (as described in the following two subsections), the migrated class is not registered in the Java Connector. It is not necessary to register classes because they can be referenced by name.

The generated proxy class can exist in parallel with the original class—as shown in the diagram—or hide the original class. Keeping the original class and the proxy class is useful for running test cases against original and migrated code. The point is to which class the name of the original class refers. I discuss the advantages and disadvantages of the two alternatives in section 6.2.2.2.

Generating the proxy class can involve some house-keeping. Section 5.4.3.3 below de-

scribes a situation where it is necessary to create a proxy for a Smalltalk class on the Java side, e. g. `PJFoo` for the Smalltalk class `Foo`. When generating the Smalltalk proxy class `PFoo` for `Foo`, the Java proxy needs to be removed. The Smalltalk Connector keeps a log of classes that have Java proxies and performs the house-keeping automatically.

5.4.3.2 Instantiating a Migrated Class

Once a class has been migrated and a proxy class has been generated for it, instances of this proxy class can be created. Instantiating a proxy class starts with sending it the message `new`—as with any other class, see figure 5.6. The proxy class tells the Smalltalk Connector to request instantiating an instance of the migrated subject class. The request includes the class name of the proxy class. The Integrator replaces this name with the name of the target class and passes the request on to the Java Connector. The Java Connector creates a new instance of the target class and registers it with its local `Registry`. The registration produces an object id for the new instance which is returned to the caller. Then the proxy class creates an instance of itself and initializes the subject of the new instance with the object id that was returned from the Java side.

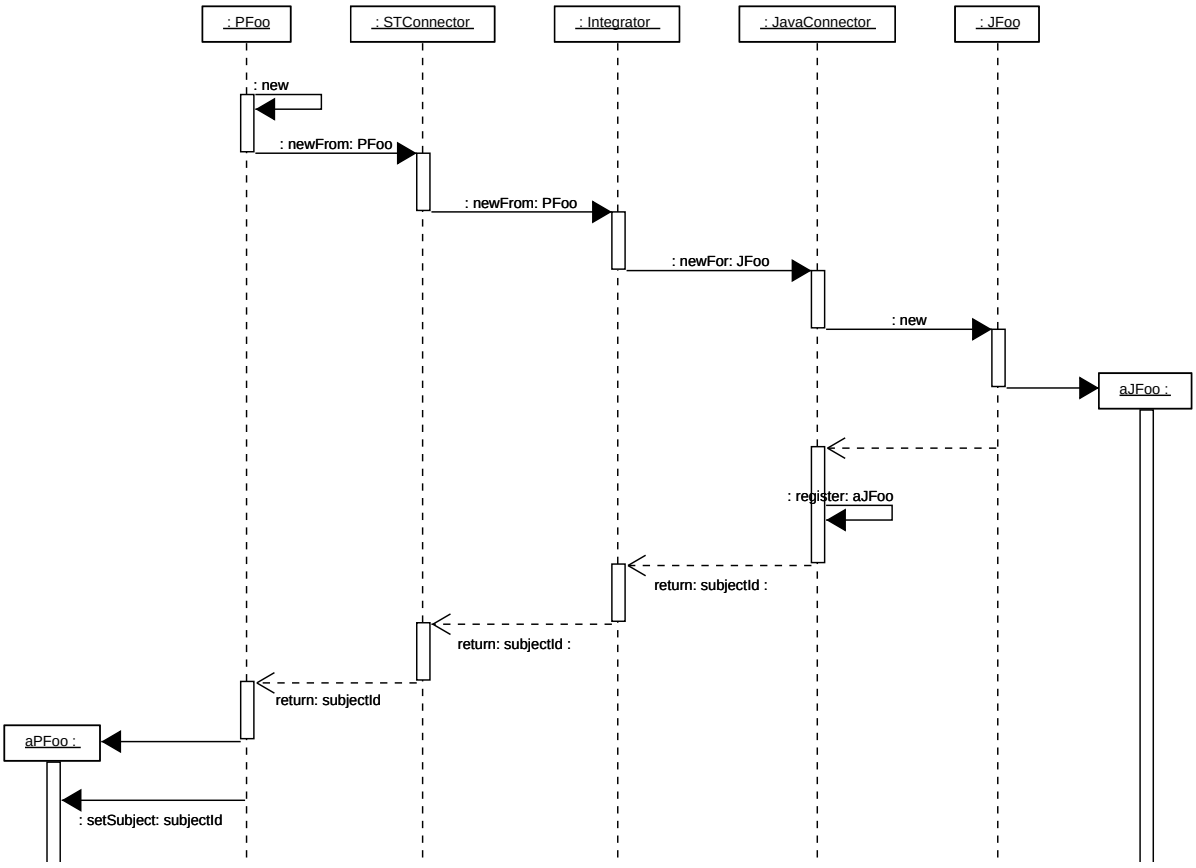

Figure 5.6: Instantiating a migrated class

The proxy class has to take care not to create more than one instance with the same subject. Having multiple proxy instances for the same subject would violate the identity of the subject. If exactly one proxy instance exists for a subject, then the proxy instance can

be used to compare the identity of the proxy. Making a copy of the proxy requires making a copy of the subject as well. To achieve this I override the `new` method in the proxy class and track the subject ids for which an instance has been created. To prevent calling `basicNew` I override it as well to produce an error message when called.

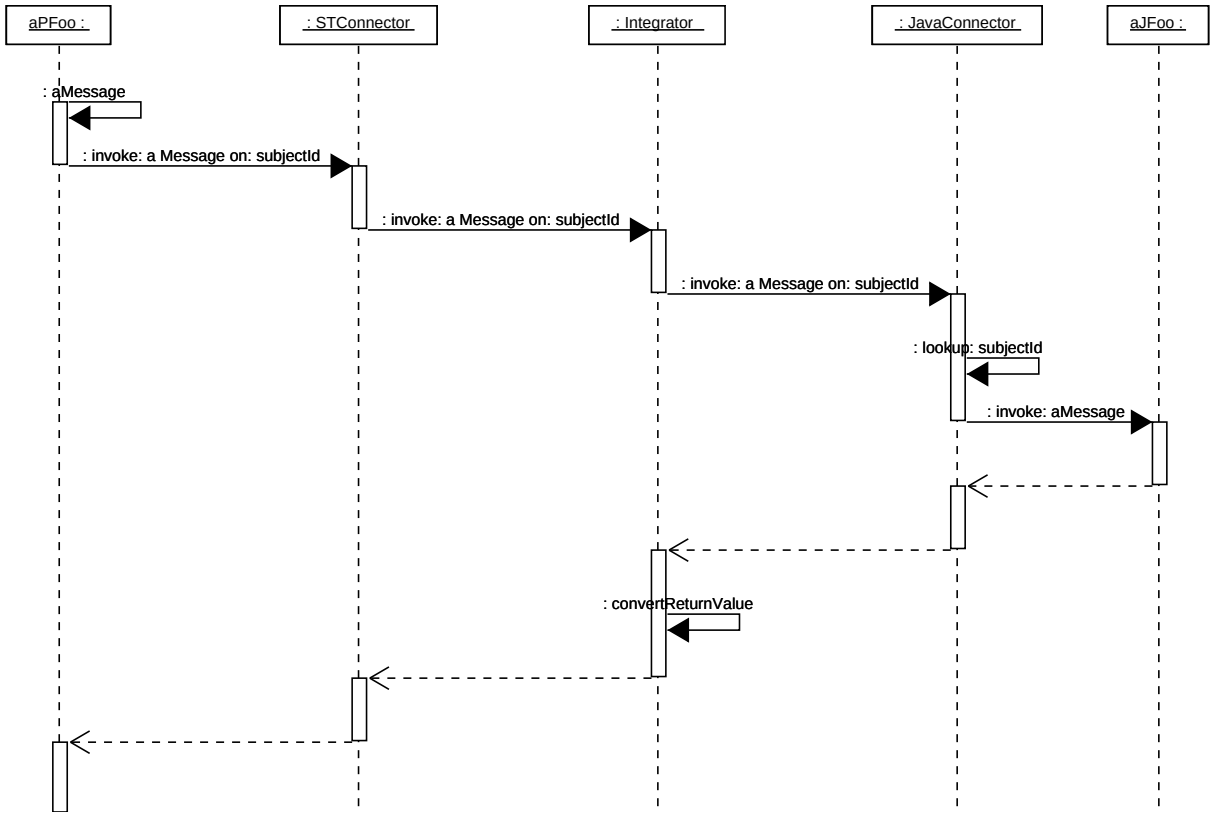

Figure 5.7: Sending a message from a proxy to its subject

5.4.3.3 Calling Migrated Code

Once an instance of a proxy class has been created it is possible to send messages to it. The simplest form of a message is a unary message, see figure 5.7. The proxy passes all messages to the Smalltalk Connector and includes the object id of its subject in the request. The Java Connector looks for the subject in its `Registry` and dispatches the message to it. The return value is passed back to the proxy through the Integrator. Because values are sent as text, the Java representation of a value might not be understood by Smalltalk. In most cases one will not notice a difference, but e. g. the representation of control characters is different. The Integrator converts the literal representation of the return value from the Java representation to the Smalltalk representation. Note that at this point I deal with a *backwards* migration—of data though. As the next case will show, a forward migration of data is necessary as well. This functionality, however, is available in the `SyntaxConverter` already. The backwards conversion is a separate implementation.

The data migration is similar to what a CORBA Object Request Broker (ORB) does when it marshals values, or what a SOAP client does when it encodes values. Both technologies

could be used here but since they are far more general—they address many more languages besides Smalltalk and Java—they are also much more complicated to use. I listed both in section 5.1.2 as blind alleys because I found it easier to implement the forward and backwards conversion for a dozen of base types myself.

The next kind of messages are messages with parameters. I will describe three cases: 1. simple parameters (base types), 2. complex parameters (objects with identity), and 3. proxies as parameters. Figure 5.8 shows sending a message with a base type as parameter, e. g. a `String`. As before, the message is forwarded to the Integrator. The Integrator converts the parameter values to their Java representation and forwards the message to the Java Connector. The return value is passed back as in the previous diagram (5.7).

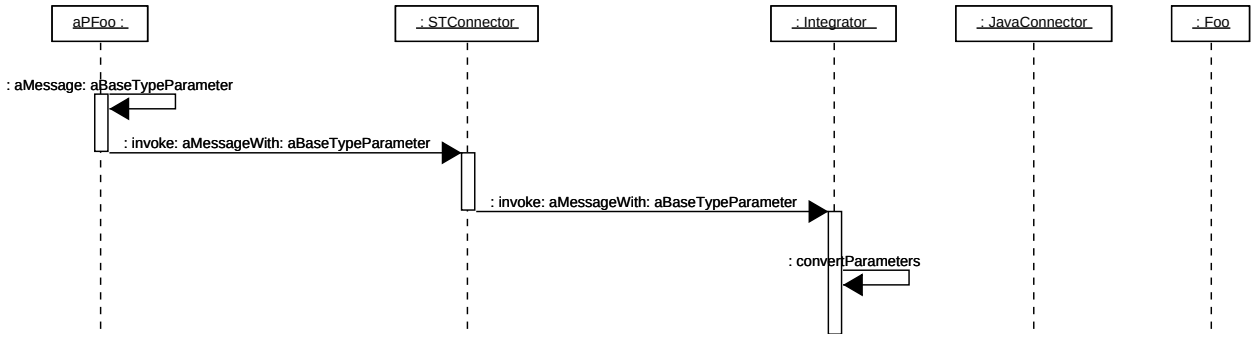

Figure 5.8: Sending a message with a base type parameter (returns not shown)

Messages with complex parameters, see figure 5.9, are different from the previous case because complex objects have to be passed by reference, not by value. If complex objects were passed by value, a copy would be sent with the message and the identity of the complex object would not be preserved. To call an object by reference, it is necessary to create a reference. For example the message `addDependent: self` contains the complex parameter `self`. Instead of passing `self` to Java, a Java proxy for `self` will be generated. The message is changed to refer to that Java proxy for `self` instead of `self` directly. If the parameter is accessed later in Java, in fact the Java proxy will be used and all messages will be routed back to the Smalltalk side.

The sending of the message starts by forwarding them to the Smalltalk Connector. At this point the Smalltalk Connector sees the complex parameter. It checks whether this object has a reference, i. e. a Java proxy, already. If so, the complex object had been registered earlier and its object id can be retrieved from the `Registry`. When the message is passed on to the Java Connector, this object id is used instead of the reference to the object itself. If there is no Java proxy yet, the Smalltalk Connector sends a command to the Integrator to create a proxy for the complex object. After successful creation of the Java proxy, the complex object is added to the `Registry`. Note that the Smalltalk Connector does not know anything about the Java proxy. It only takes the registration of an object in the `Registry` as an indicator that this object must have some Java proxy.

Messages can also include a proxy as a parameter. A proxy is a complex type but the previous solution for complex types would not make sense for proxies. Since the proxy refers to an object on the opposite platform, a proxy can be replaced in a message simply by a reference to its subject.

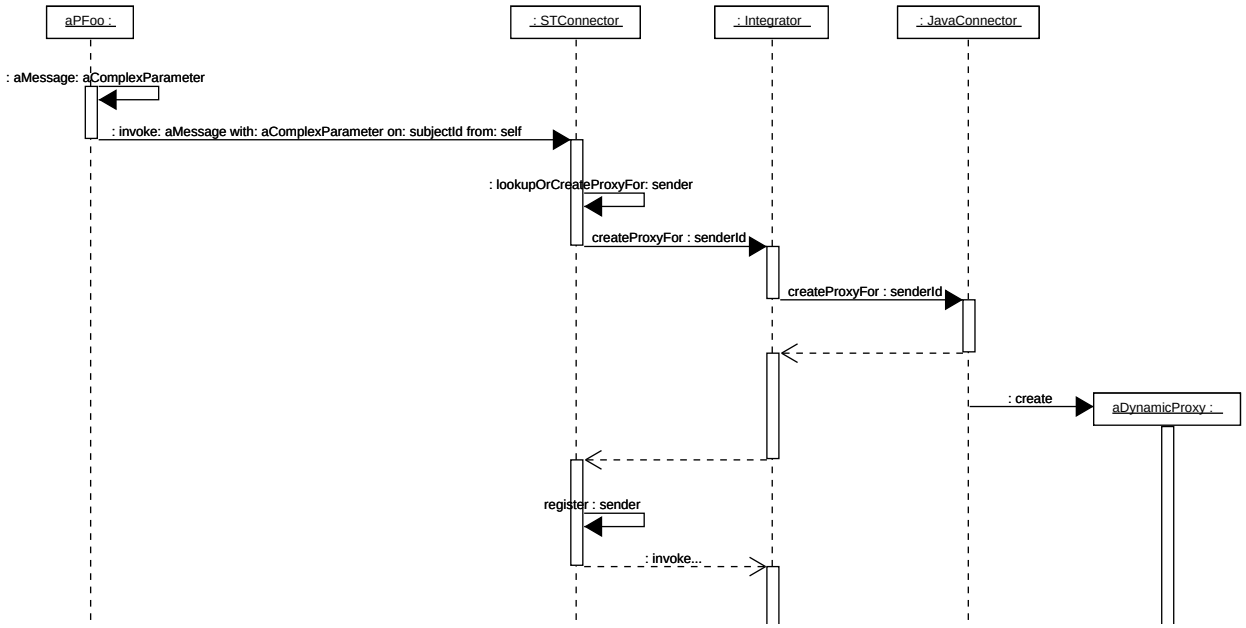

Figure 5.9: Sending a message with a complex parameter (returns not shown)

So far I only discussed sending messages from Smalltalk to Java. Messages from Java to Smalltalk are sent exactly in the same way.

5.4.4 Installation and Startup

To install the Smalltalk Connector into VisualWorks the code shown in figure 5.3 can be pasted into a workspace and executed (`doit`). The Smalltalk Connector listens per default on port 3030. Log messages are written to the Transcript.

To install the Java Connector the plug-in archive `connector.jar` has to be unzipped in the `plugins` subdirectory of an Eclipse installation. Eclipse automatically detects and activates the plug-in on startup. The user interface will have an additional pull-down menu with two entries for starting and stopping the Java Connector. Per default the Connector listens on port 6060. Messages are logged in the Eclipse log window.

The Integrator is written in Dolphin Smalltalk and is intended to be used in the Dolphin development environment. The file `integrator.st` needs to be filed-in into Dolphin Smalltalk. The Integrator listens on two ports, 4040 for the Smalltalk connector and 5050 for the Java connector.

The integrated environments are started by starting VisualWorks, Eclipse, and the Integrator. In VisualWorks the code in the workspace has to be executed. In Eclipse the Java Connector has to be started through the pull-down menu. Then the Integrator can be started by executing (`doit`) `Integrator start` in a workspace. The Integrator will establish a connection to both Connectors and check that their status is ok. To complete the setup of the Smalltalk Connector the Integrator installs some helper classes (`Invoker`, `Forwarder`, `STConnector`) in VisualWorks by sending them to the Smalltalk Connector.

Chapter 6

Applying the Integrated Programming Environments to Platform Migration

In this chapter I show how to perform a *piecemeal* migration using the integrated programming environments. The chapter has four sections which proceed from a high level to more and more detail, see figure 6.1. The first section describes the iterative migration *macro process* consisting of three *phases* (6.1). The second section provides more details about how a developer uses the integrated environments at the *micro process* level (6.2). The third section discusses specific problems that a developer will face—and solutions to these problems (6.3). The chapter with a description of the results (6.4).

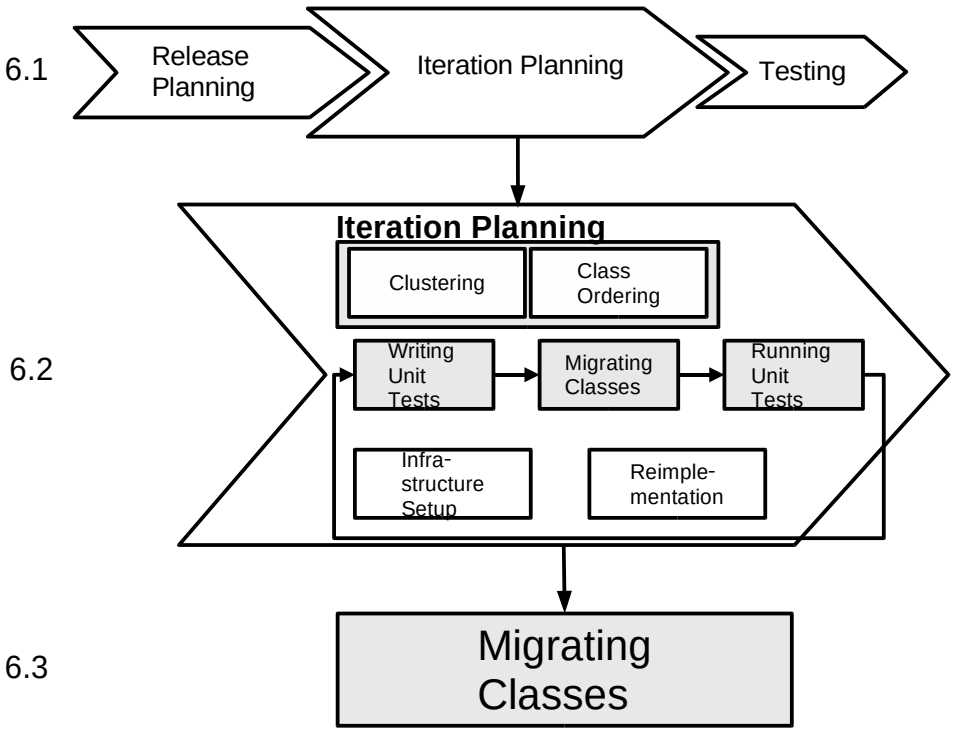

Figure 6.1: Piecemeal migration process at different levels of detail

6.1 Macro Process

The macro process follows two principles (6.1.1) which I describe first. It has three phases (6.1.2) and two kinds of iterations (6.1.3).

The piecemeal migration process has many similarities to eXtreme Programming (XP) [Beck, 1999] but also differs in some respects. Unit testing and continuous integration are slightly different; there is no emergent design and no multi-user coordination. The most striking similarities are the test-first approach and short iterations.

6.1.1 Principles

When designing the process I discovered that I put in a number of well-established “ingredients” that I want to list first. These principles apply to the macro process. I also list principles that apply to the micro process in section 6.2.1.

6.1.1.1 Process follows Architecture

The first principle states that the migration process shall be aligned with the architecture of the existing Smalltalk application. This principle is similar to Conway’s Law which states that “organizations which design systems are constrained to produce systems which are copies of the communication structures of these organizations” [Conway, 1968]. In other words, the organization of the software and the organization of the software team will be congruent. My variation of this law is that the migration process should be congruent with the architecture of the existing system.

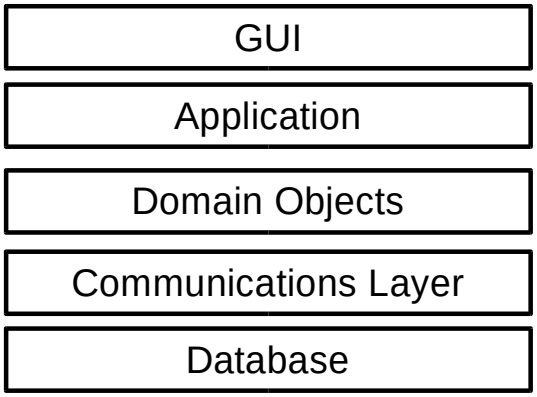

Figure 6.2: Layered architecture of a typical Smalltalk application

The architecture of a typical Smalltalk application consists of several layers, see figure 6.2. The user interface and the domain objects are typically connected using the Model-View-Controller (MVC) design pattern. These three elements are located in different layers of the architecture. The models usually map well to the domain objects, the controllers to the

application models, and the views to the user interface. Furthermore, the architecture is frequently designed as two-tier architecture with a fat client containing the user interface and the domain objects, and a database server. Smalltalk applications rarely use an application server and in most cases the communications layer consists only of a database interface.

There are two possibilities to design a process so that it complies with this principle. The piecemeal migration process uses both alternatives for two kinds of iterations. The first alignment of the process is according to the horizontal layering. The macro process proceeds—iteratively—through the breadth of the system functionality. The second alignment is orthogonal to the layering. The micro process proceeds “vertically” through the system covering a slice of the system in each iteration.

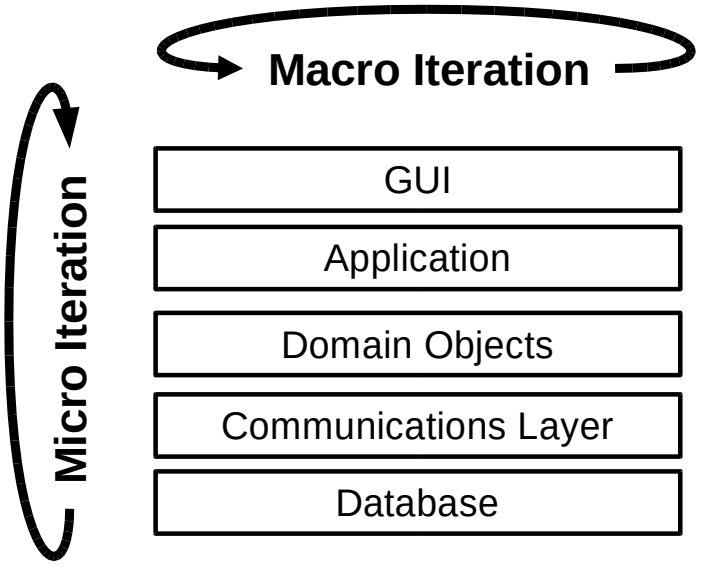

Figure 6.3: Two kinds of iterations

6.1.1.2 Unit-of-Work is a Layer’s Main Abstraction

The second principle states that a layer’s main abstraction should be the unit-of-work for structuring the process at that layer.

Use cases describe the functionality of the existing system at the highest level, see figure 6.4. These use cases are refined into a hierarchy which is similar to a traditional functional decomposition. The rationale here is to work with a description of the system which is both easily accessible, e. g. from the user documentation, and is comprehensible by the end users and/or the customers of the project.

I use use cases for the process description although I took many elements from XP; using XP’s user stories would seem natural. The difference between user stories and use cases is not big. The point is that both are a description of functionality at the user level. User stories are less formal and leave much to be discovered by talking with the customer in

person. Use cases have the advantage that UML describes *extends* and *uses* relationships between them. These relationships are useful for analyzing the dependencies between use cases and structuring the iterations according to these dependencies. Note that use cases are used as an input for the functional tests, see section 6.1.1.5 below.

The main abstraction at the user interface layer are individual windows, dialogs, wizards, etc. Each of these can be identified easily and separated from the rest of the system. Classes and groups of classes—if they are tightly coupled—are the unit-of-work at the application and model level. At the database layer the main abstractions are tables and transactions. According to the unit-of-work principle one should avoid working with less or more than complete tables or transactions.

6.1.1.3 Orthogonal Iterations

As mentioned earlier, the migration process has two kinds of iterations: One at the macro level and another one at the micro level.

This principle states that the two levels of the process should proceed in orthogonal directions through the system architecture. In this way the whole system will be covered. The vertical path of the micro process through the layers is shown in figure 6.4.

The macro process works through a set of hierarchically decomposed use cases. The micro process starts with one or more use cases and aims to migrate the functionality described by the use cases. The point here is that a use case describes an end-to-end flow of control through the system since almost every functionality in a typical two-tier Smalltalk system requires database access.

6.1.1.4 Focus on Migrating Smalltalk Code

This principle states that one should not attempt to do seemingly related tasks together with a migration which in fact do not deal with *migrating Smalltalk code*.

One example, where an element of a Smalltalk application is in fact not Smalltalk, is the user interface. By user interface I refer to the widgets of a window as represented by the operating system (or an emulation). These widgets represent the view component of the Model-View-Controller design pattern. This part of the user interface is typically designed using a GUI painter. There are several ways a user interface can be stored and different Smalltalk dialects have their own approach: serializing the widgets, generating code, or a declarative description. The VisualWorks dialect of Smalltalk uses the third approach. The description of a user interface is called a `windowSpec` and is stored as a class method of the application window. Once a window is opened, the `windowSpec` is interpreted by a UI builder which constructs the UI from the description. The reason for doing it this way is to achieve platform independence. Each platform (Windows, Unix, MacOS) has its own builder.

The problem with trying to migrate a VisualWorks user interface description (`windowSpec`) is that this specification is a separate language from Smalltalk. This principle states that `windowSpecs` are not within the scope of a Smalltalk migration. Another example of this case are stored procedures in a database.

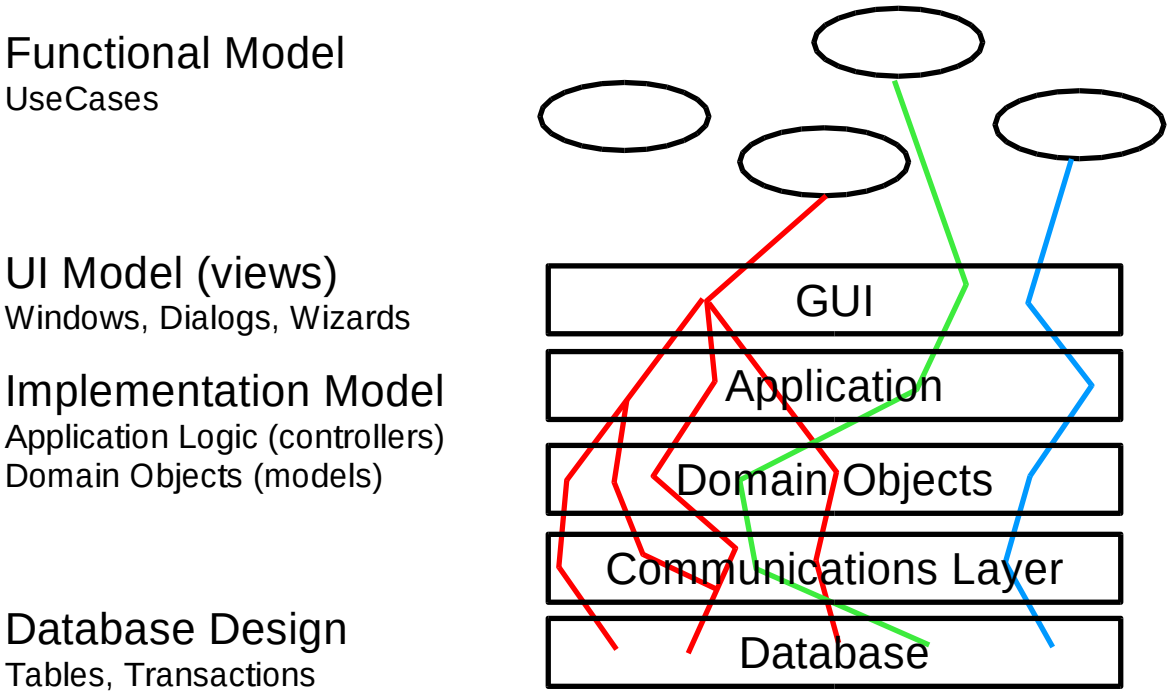

Figure 6.4: Vertical slices through a layered application

There are different alternatives to solve this limitation. One is to reimplement the UI using a GUI painter on the target platform. This has the great advantage that the GUI can be developed according the target platform's style. Another alternative is to convert the `windowSpec` into another declarative language which can be interpreted on the target platform. e. g. the XML User Interface Language (XUL) could be used since a builder is available for Java. Yet another alternative is to reimplement the VisualWorks UI builder in Java. A prototypical implementation is described in [Eßer et al., 2001].

Another goal that should not be mixed with a migration is rearchitecting of the existing system. In most cases there are new requirements on the waiting list for the developers but the old platform somehow inhibits their implementation. Then the motivation might be to migrate to Java so that the new requirements can be implemented more easily. With the ubiquity of the Internet a frequent requirement is to web-enable an existing Smalltalk system. This means to create a web-based user interface instead of the fat client architecture. However, this requires a significant change of the system architecture.

This principle suggests that in this case the migration should be finished—possibly omitting the UI code—before rearchitecting the system.

6.1.1.5 Continuous Testing

This principle states that testing should be practiced at both the macro and the micro process levels *frequently*. Continuous testing means to test after every non-trivial change to

the system. Testing can be time-consuming—and continuous testing even more so. Therefore, it is desirable to develop a test suite which can be run automatically.

At the macro process level *functional tests* test a system from a users perspective. Functional tests can be difficult to develop if an application has a graphical user interface. Testing tools like mouse-click and key-stroke recorders can be used; for a web-based user interfaces e. g. HTTPUnit would be applicable. Generally, there is a reluctance to invest in test cases for an existing system. The point here is that 1. a small number of functional tests plus some variations will serve the purposes of the piecemeal migration approach and 2. being able to run the functional tests against both the original and the migrated system is an excellent way to demonstrate the functional equivalence of both systems.

The functional tests can be developed much easier than in forward engineering because in a migration project one has an existing application already. The aspect of test-first *design*, as promoted e. g. by XP, is not relevant for a migration because the design should be maintained where possible. This should not be confused with a test-first migration where the test cases are developed before performing a migration, see section 6.2.1.1 about testing at the micro process level. A secondary benefit of developing functional tests first is that they are also useful in structuring the process into iterations.

Unit tests are used for testing in the micro process. See section 6.2.2.1 for more details on creating unit tests to be used in a migration.

Both functional and unit tests can be run against the original and the migrated system, see section 6.2.2.3.

6.1.2 Phases

The migration macro process, figure 6.5, has three phases: planning, migrating, and testing. All three phases comprise one iteration. Note that although these three phases look like a waterfall process, they are *logical* phases. It is acceptable to complete a phase only partially in one iteration, if the unfinished work is continued in a later iteration. Iterations are discussed in more detail in section 6.1.3.

The first and the last phase may involve a customer or end user for describing the use cases, and for developing and executing the functional tests. The migration phase is the phase in which a developer works on his own and applies the integrated programming environments. The description of the first and third phase is included here for completeness so that the reader can understand the application of the tool integration solution in the context of a migration project. However, I will focus on the migration phase in section 6.2 and keep the description of the other phases brief.

6.1.2.1 Planning Phase

The planning phase consists of three activities. Two of them are managerial in nature and grouped together as release planning. This is similar to release planning in XP where a set of user stories is developed.

The first activity is to create a use case model of the existing Smalltalk application. This could be done from the user or the internal system documentation. Or more directly, by

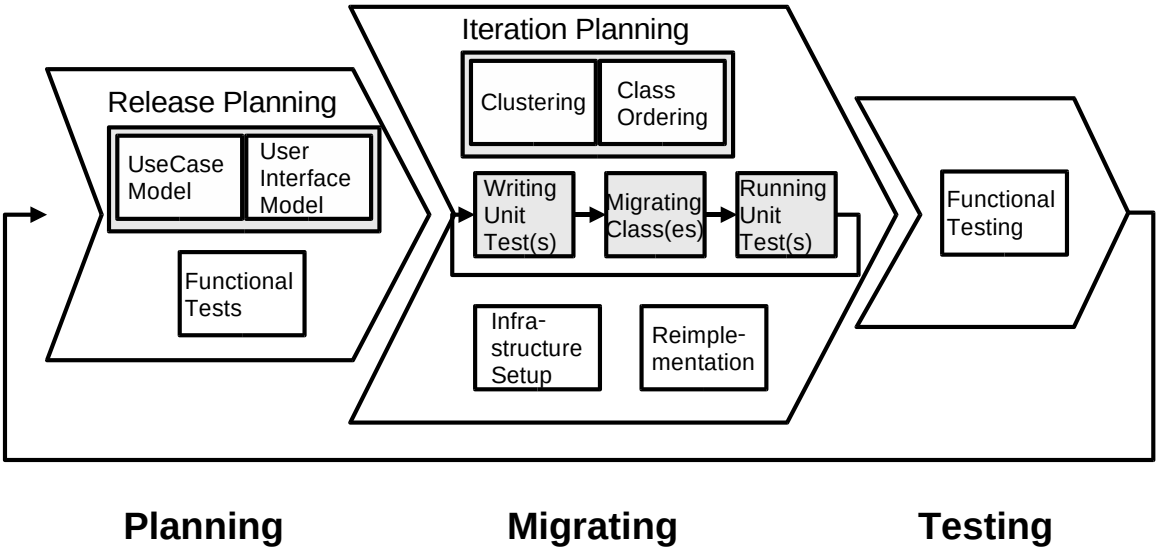

Figure 6.5: The migration process consisting of three *phases* and eleven *activities*

trying out the system and speaking with end users. The goal of this activity is to structure the use case model hierarchically so that each use case has a moderate size. Use cases which are too large inhibit planning the iterations. If a use case is too large it should be split into smaller use cases. The use cases *drive* the migration process [Jacobson et al., 1992].

The second activity is to create a model of the user interface. This model shows which windows, dialogs, wizards, etc. an application has. The purpose of this overview is managerial because it makes it easier to detect gaps in the coverage of the system functionality. It has no technical purpose for the migration.

The use cases developed in the first activity are the basis for the functional tests. If possible, the functional tests should be automated. If it is not feasible to automate the functional tests, they have to be run by hand. The functional tests will be used in the third phase when the migration is finished. Although they could be developed only then, having them ready early in the process also serves the purpose of program understanding. In any case the developer will have to spend a considerable amount of time on understanding the program anyway. It would be wasteful not to document this—functional tests are one way of doing this which is also useful later for testing the migrated system.

6.1.2.2 Migration Phase

This section gives only a brief overview of the migration phase. More details can be found in the next section 6.2. The purpose of this section is to explain the relationships to the other two phases of the macro process and to describe the activities which are not part of the micro-iterations (6.2.2).

The input for the migration phase from the previous phase is a set of use cases which have been scheduled for the current iteration. The output of the migration phase is a set of

migrated classes needed for the current set of use cases. All unit tests for these classes need to pass—if the functional tests pass for these classes will be determined in the next phase (6.1.2.3).

The migration phase includes two rather organizational activities—like the planning phase—which are grouped as *iteration planning*: clustering and class ordering, see figure 6.5.

Clustering Clustering refers to defining a working set for the current use cases. The working set includes all classes which have to be migrated in this iteration and excludes all base classes that deliberately will not be migrated. The clustering also describes groups of classes that are tightly coupled and possibly have to be migrated together in one micro-iteration.

Class Ordering The class ordering defines an efficient path through the migration effort. Some classes will have stronger dependencies on each other or on base classes. Although all dependencies have to be resolved during the migration, the class ordering defines a work-flow that makes working through these dependencies easier.

Note that the next three activities are all part of the micro-iterations. They will be covered in detail in section 6.2.2.

Writing Unit Tests Writing unit tests, see section 6.2.2.1, is the beginning of an iteration at the micro process level. Writing the tests first is inspired by the test-first programming approach in XP. The idea behind it is to structure the process by concrete goals, i. e. a test case, which fails initially. All further work in an iteration then focuses on making such a test case pass.

Writing the tests can be challenging because the existing code might have a structure that is not easy to test. [Feathers, 2002] describes some steps to make legacy code more testable. Note that this possibly requires refactoring existing code or adding code to expose classes in order to make them more testable. This refactoring happens prior to working on the migration itself.

Migrating Classes The migration of classes is done in several steps, see figure 6.6. Each step is supported by the Integrator. This activity is described in more detail in section 6.2.2.2.

Running the Unit Tests Running the tests is a very short but nevertheless important activity. The tests that were created two steps earlier are executed. Since all work in the previous step was concentrated on making all the test cases work there should be no problems.

Infrastructure Setup The migration phase contains two additional activities which are not directly related to the migration, but necessary to support it. The infrastructure setup activity subsumes all tasks related to installation, configuration, and implementing base functionality in the migrated system on the target platform. Usually during the first iterations of a project a lot of the required infrastructure is missing. The additional effort is not necessary in later iterations. This special situation is discussed in more detail in section 6.1.3.3.

Reimplementation It is not always possible or desirable to migrate every part of the source system. Sometimes license restrictions prohibit the migration of code. Another very striking example is the GUI. See section 6.1.1.4 for more details on this issue. In VisualWorks the GUI is constructed using a GUI builder which creates a description of the GUI in its own language (different from Smalltalk). In such a case I suggest reimplementing the GUI using the GUI builder of the target platform. Of course, the goal is to reimplement as few components as possible. My point here is, that—when necessary—this is the place in the process where to do it.

6.1.2.3 Testing Phase

I include the testing phase here only to demonstrate the overall process—there is nothing migration-specific in this phase. The description of this phase is simplified. In a real project many more activities would have to be included in the process, e. g. more kinds of tests like stress and load tests, updates to the documentation, preparation of the installation and deployment.

Functional testing is at the same level of abstraction as the first phase which dealt with capturing the functionality in use cases. From these use cases the functional tests were derived. This testing activity solely consists of running these functional tests. If the tests are not automated they have to be run manually. As in the first phase, the end users/customers can be involved.

The only interesting point to be made is that the integrated programming environments support the execution of the test suite against both the original and the migrated system. This is done at the micro process level for the unit tests, for the functional tests it is done here in this phase. Running the *functional* tests against both the original and the migrated system demonstrated the functional equivalence of the systems.

6.1.3 Iterations

The migration process operates at two levels, the macro and the micro level. It has iterations at both levels. Accordingly, these are called *macro iterations* (6.1.3.1) and *micro iterations* (6.1.3.2). The first iteration of a project always requires more effort than later iterations (6.1.3.3).

6.1.3.1 Macro Iterations

Macro iterations represent the main structure of the migration process. The goal of each macro iteration is to produce a release of the migrated system which is fully functional—for the selected use cases defined in the release planning. The system will not be complete, of course, and deployment would not make sense. But the functionality is visible to the end users so that they can give feed-back. The duration of a macro iteration should be between one and four weeks. This is in line with the length of iterations in XP.

6.1.3.2 Micro Iterations

Micro iterations are the interesting aspect of the process for the work presented here. The micro process describes the activities of an individual developer. The iterations at this level are much shorter than the macro iterations. They can last from five minutes to eight hours. The ideal duration would be between 15 minutes and one hour. A micro iteration should not last more than a work day since it is harder to catch up on the next day. I would recommend splitting a large iteration into smaller iterations and perform them sequentially. The advantage is that after each iteration the set of passing tests is a check point for the developer. An iteration which takes many hours breaks the code for a long time. Then the risk of introducing several problems into the code without detecting them is much higher. The rationale is to get to the next test run as quickly as possible.

The activities of the micro iterations are described in more detail in 6.2.2.

6.1.3.3 The First Iterations

The first iterations of a project are special in the distribution of work between migrating use cases and setting up infrastructure. For example a simple use case like logging into the system requires access to the data base layer. Although a use case like this looks quite simple at first, it implies an end-to-end execution path through the system. Setting up the data base, the communications layer has to be done before the application logic can be written and run. This kind of setup work is grouped into the infrastructure setup activity (6.1.2.2).

It is known from forward engineering XP projects that these additional tasks in the first iterations of a project can be quite time-consuming. Of course, as a project proceeds, most of the infrastructure will be in place and no additional setup will be required. This situation is similar to brining a car up to speed. Once the cruising speed has been reached, the trip can continue comfortably.

6.2 Micro Process

The micro process is part of the migration phase described at a high level earlier (6.1.2.2). The migration phase has at its core three activities which comprise the micro process (figure 6.5). These activities are repeated as micro iterations (6.1.3.2).

6.2.1 Principles

The micro process is—like the macro process—based on principles. Again, these principles are inspired by XP.

6.2.1.1 Test-First Migration

Test-first migration means that test cases have to be written prior to migrating code. This test-first approach is similar to XP, but it serves a completely different purpose here. In XP the test-first approach is a way of designing software (test-first design), see also section 6.1.1.5. The idea is that the design somehow emerges from programming while test cases keep a developer focused on the task at hand, and continuous refactoring improves the design. Here the design of the Smalltalk application already exists and it should be preserved as much as possible. Changes are appropriate where the Smalltalk design needs to be adapted to the style of the Java platform. The purpose of test-first migration is to have test cases available in the process early on. These test cases serve two tasks: they help the developer to focus on the current use case, and they allow for the comparison of the behavior of the original and the migrated system.

Creating tests, as described below in section 6.2.2.1, can be difficult. Sometimes the code is not easy to test and needs to be refactored to be more testable. Initially, a developer will have to spend quite some time on understanding the original system. The test cases are, despite the additional effort, a valuable resource which document the understanding that a developer has gained about the system. This information is not available in any other way, and it has the advantage that it is—as a test case—executable.

6.2.1.2 Never Break the Code for a Long Time

The code of the system should be executable at all times—on both platforms. The original system will be executable, obviously. My point is that during the migration the hybrid system consisting of both Smalltalk and Java code shall be executable. Of course, it is impossible to adhere to this rule 100% but it serves well as a guideline. The rationale is that test cases can provide the developer with valuable feedback about the behavior of the code. Passing test cases mean that everything is fine. For being able to run the test cases it is mandatory that the code is executable. This allows a developer to get feedback from the code at regular intervals.

There are different opinions in XP about how long a break in the code is acceptable. Usually short intervals as little as five minutes allow a developer to work faster because his flow of work is interrupted less often by searching for bugs. However, there will be situations where large breaks cannot be avoided. Then one should try to limit the break to eight hours. The rationale is—as in XP—that a developer looses the context when he returns to work on the next day. It is better to split large tasks which can be finished in one day. This results in passing test cases (“green bar”) at the end of each day.

Daily builds are similar to this principle but not quite the same. A build is a prerequisite for running test cases because only a built system is executable and can be tested. Without test cases, a build simply checks for compilation problems. Note that in modern IDEs

the build process has been greatly simplified by incremental compilers which recompile all dependent compilation units automatically. Both IDEs used for the prototype have such incremental compilers. Often the goal of a daily build is not the build itself but the daily integration of code developed by different team members. The daily integration is a prerequisite of the daily build. These frequent integrations help avoiding integration problems ("integration hell") after several developers have independently worked on a single code base for many days. However, the point is that frequent testing is the key to success—and this requires an executable system.

The second rationale for this principle is that it is easier to understand the source system if one can observe the execution (setting breakpoints, watching variables, inspecting object hierarchies, etc.). If bugs are introduced during the migration process they can be discovered easier if a developer can analyze the running system. Therefore, the code should not be broken for a long time.

6.2.2 Micro Iteration Activities

A micro iteration (6.1.3.2) contains three activities. They follow a V-model: the first and the third activity deal with writing and running tests. The second activity consists of performing the actual migration. Note that the structure of the micro process is similar to the structure of the macro process. The macro process iterations are framed by *functional* tests, the micro process iterations are framed by *unit* tests. This structural similarity of a process is sometimes called *recursiveness*.

6.2.2.1 Writing Unit Tests

Usually there are no test cases for legacy systems and developing tests is an extra effort for a migration project. I argued many times that testing is crucial for a migration project. Therefore, the goal for writing the unit tests is to minimize the additional effort while still getting the whole set of benefits from the tests. It certainly would be wasteful to require a test case (a subclass of `TestCase` with several test methods) for every class in the original system. Since there are many strategies for writing test cases, such as top-down, inside-out, bottom-up, ad-hoc, etc., the question is how to write the *right* test cases?

When looking for the right test cases it is useful to think in terms of inflection points and test coverings, two concepts introduced by [Feathers, 2002]. Test coverings are tests which are technically similar to unit tests but serve a different purpose. They are introduced at *inflection points*. An inflection point "is a narrow interface to a set of classes. If anyone changes any of the classes behind an inflection point, the change is either detectable at the inflection point, or inconsequential in the application." A test covering is a set of tests that introduces an invariant in the system. It allows for the modification of the covered classed without writing individual tests for the covered classes. Finding good inflection points is a purely intellectual activity for which to date no tool support exists. I discovered some good inflection points by writing tests in a bottom-up manner. In the end I found that I could remove the lower-level tests that I created on the bottom-up path—it was sufficient to keep the higher-level test. Experience will help avoiding such extraneous efforts.

Sometimes the functionality to be tested is not easy to test. To make it more testable it needs to be refactored so that it is exposed better. Also, it might be necessary to add test harnesses which provide and collect meaningful test data different from the data structures used by the application internally. If a use case is too large for one iteration, it is suggested to split it into smaller ones. Then, test cases are useful for regression testing the required refactorings.

When writing test cases a developer has to choose on which platform he wants to write the tests. Both the source and the target platform can be used. In my experience it is a good idea to write the first tests in Smalltalk because one wants to capture the functionality of the existing system first. This is easier to do in Smalltalk since the original application is a Smalltalk application. Once more and more classes are migrated to Java the coupling between the test cases and the classes under test will indicate that it would be better to have the test cases on the target platform. Since the whole environment integration is about migration from Smalltalk to Java, the Smalltalk test cases can easily be migrated to Java. Usually the structure of test cases is quite simple so that very little manual intervention can be expected for the migration of the test cases to Java.

The test cases are written using the normal tools of the IDEs and using the classes provided by the SUnit and JUnit testing frameworks. As mentioned above, each test case is a subclass of the class `TestCase` which is provided by the framework on both platforms. To each test case class several test methods are added to perform the actual tests. The `setUp` and `tearDown` methods can be used as usual to create and clean up the test fixtures.

6.2.2.2 Migrating Classes

The migration of classes is the core feature provided by the integrated programming environments. This migration activity demonstrates most clearly how the integration architecture presented in chapter 5 supports the piecemeal migration process. First, I describe the six main steps for migrating classes. I continue with some notes on details which are not shown in figure 6.6, e. g. the use of test cases. Finally, I give a concrete example of migrating two classes and developing test cases for them. I make frequent references to chapter 5 which describes the implementation of my prototype.

1. In the beginning both programming environments are started (5.4.4). The Smalltalk application resides on the source platform and the target platform is “empty”.
   Both platforms contain a Connector which connects them to the Integrator; the Integrator is not shown yet because it will be used only later.

2. A developer works on a small subset of the application at a time, i. e. a set of use cases as defined by the release planning activity. In this example the current use case contains four classes, see the dark grey classes on the left. These classes are the unit-of-work for the current macro-iteration, the number of classes for the micro-iterations will be smaller.

3. The developer chooses which classes to migrate (light grey on the left) and sends them through the Integrator to the Java side (dark grey on the right). The original Smalltalk classes are replaced by proxies which forward all requests to the migrated classes. Note that these changes are reflected immediately in the runtime

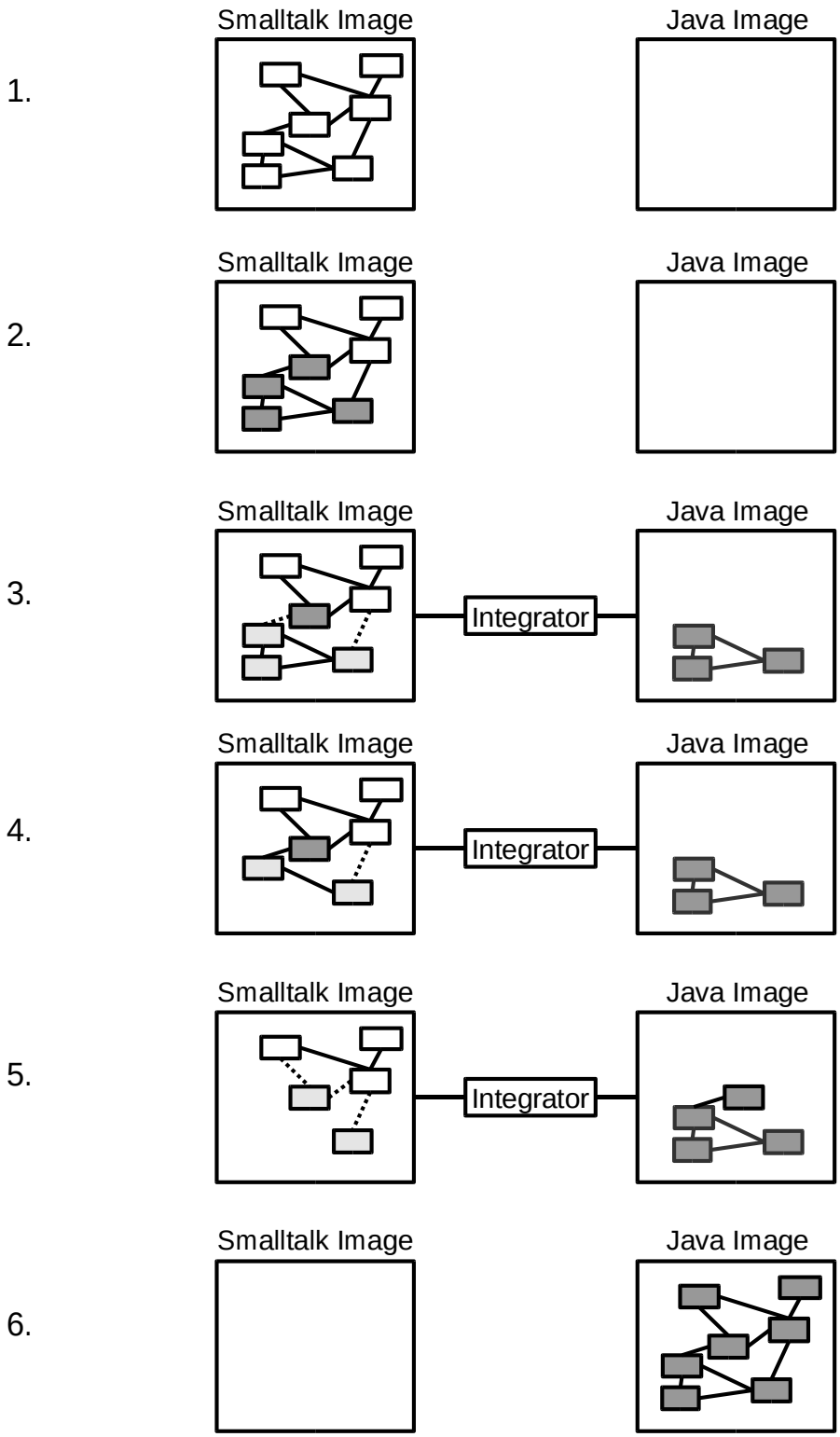

Figure 6.6: The six main steps for migrating classes

environment—no restarting is necessary. In this step the Integrator performs the syntax conversion of the transferred classes.

4. The original code of the classes that were replaced by proxies gets disabled. (It might be used later to run unit tests.)
   Step 3 is repeated until all classes that belong to the current unit-of-work are handled.

5. Steps 2–4 are repeated for more units of work. At this point all selected classes are migrated to the Java platform. Note that the structure of the classes (the boxes and lines on the left and right) are the same. If the migrated classes would require some adaption to the Java platform, then the classes on the right would look different—more simple, or more complicated.

6. In the end there will be no classes (and no proxies) left on the source platform. All code is now migrated to the target platform. The Connectors of the IDEs can be shut down because the integration is no longer required.

The process in figure 6.6 is simplified. The following details are not shown: First, the diagram shows a Java *image* on the right. This is an abstraction of the real situation because in Java the development and the runtime environment are not as well integrated as in Smalltalk. I describe the real situation at the end of section 5.4.2.2. Another simplification of this kind is that no base classes are shown. Of course, the Smalltalk and the JDK base classes exist on both sides.

Second, the diagram only shows classes—not instances. I assume that no instances of a class exist when it is migrated. Of course, instances of other classes can and will exist. The reason for making this assumption is that during the migration the original class will be changed. Instances of a modified class can not survive this change because the class structure is changed. There are safe changes (adding methods, changing methods), but changing a class into a proxy for the very same class is not safe. If instances of a previous version of a class should exists, the Java runtime environment will report an error and then needs to be restarted. The Smalltalk environment can be cleaned from those instances by finding all instances, i. e. sending the `allInstances` message to the old class, and calling the desctructor of these instances.

Third, the diagram does not show any test cases and how to apply them. The following example will explain this. Let's assume we have a `Point` class; each point stores its x and y coordinates. Furthermore, let's assume we have a `Rectangle` class which stores two points for its corners. Now, let's migrate first the `Point` class, and then the `Rectangle` class.

- Write a test case for `Point`, e. g. also for some arithmetic with points. The test should pass for the Smalltalk class.

- Select `migrate` from the context menu of `Point`, or invoke `Point migrate` from a workspace. This will create a `PPoint` class, a proxy for the migrated class which is called `JPoint`.

- Tell the test case to use the migrated class by sending it the message `TestPoint use: #PPoint insteadOf: #Point`. An alternative to this is to create a second test case which uses the `PPoint`; this would be called `TestPPoint`.

- Execute `TestPoint`, or `TestPPoint` for the second option. If the migration went well, the test will pass immediately. It is not unlikely for the test to fail because of some small problem. Fix it to make the test case pass. Now the `Point` has been migrated successfully. At this stage both the original `Point` and the proxy `PPoint` coexist. Since the tests have passed it is save to hide the original `Point` by renaming `PPoint` to `Point`. Note that the change is reflected immediately in the image.

- Continue with `Rectangle`: Write a test case for `Rectangle`, i. e. `TestRectangle`. The test should pass. Note that it will use the proxy automatically if the original class has been renamed.

- Migrate `Rectangle` as above; this will create a `PRectangle`. Make the test case use the proxy as above. Make the test case pass.

6.2.2.3 Running the Unit Tests

The unit tests are executed as usual: `SUnit` and `JUnit` provide a method `runAllTests` which can be executed from the IDEs on both platforms. A convenience method, as mentioned in the `Point` example above, is added to `TestCase` so that a test can be changed easily to use a different class (`TestPoint use: #PPoint insteadOf: #Point`). The Connectors provide another convenience method to run the test cases on both platforms together—without this a developer would have to start the tests separately in each IDE. For example in Smalltalk one can send the following message: `STConnector runAllTests`. This will run all `Sunit` tests locally, and send a message to the Java IDE to execute all `JUnit` tests as well. One limitation is that the `TestResult`s are not aggregated. The results should be checked in the respective user interface instead.

Note that, as described in the previous section (6.2.2.2), the tests can be run against both the original and the migrated code, i. e. SUnit tests can be used to test Java code, and JUnit tests can be used to test Smalltalk code.

6.3 Specific Migration Problems

In this section I discuss specific migration problems and show how these can be solved more easily *with* the integrated programming environments than without.

[Eßer et al., 2001] *assess* the differences between Smalltalk and Java (2.1.2). For each migration problem they list a severity and a frequency. From this list I omitted all problems which have a 'low' rating in either category. Many differences have a 'low' rating because of the similarities of the platforms (2.1.1). However, a 'low' rating still indicates that there is a problem. Furthermore, I omitted two problems which are not addressed by the integrated programming environments: the use of meta-classes and the use of extensions. In both cases a manual redesign of the Java application is unavoidable. Then it does not matter whether the programming environments are integrated or not. Three hard migration problems remain from Eßer et al.'s list, see table 6.1: blocks (6.3.1), typing (6.3.2), and library mismatch (6.3.3).

Problem	Severity	Frequency
Blocks	medium	high
Types	high	throughout
Library Mismatch	medium	high

Table 6.1: The three most difficult migration problems

The next three sections share a common line of thought. First, I describe the migration problem. Second, I discuss how to solve it without integrated tools, and then how to solve it with integrated tools. The first section also contains a detailed comparison of the migration process. It demonstrates *why* working with integrated tools is much better. This comparison applies to all three problems and is not repeated for the second and third problem.

6.3.1 Blocks

Blocks (`[ statements ]`) are objects in Smalltalk. They encapsulate statements and delay their execution until a `value` message is sent to the block. Java does not have an equivalent construct. Java has several other constructs which can emulate the behavior of Smalltalk blocks, depending on how a block is used.

A block can reference variables which are visible in the enclosing scope. This includes local variables and parameters of the current method, as well as instance variables of the object which the current method belongs to. The unique feature of blocks is that the block will keep the context of the referenced variables—even if the defining method returns and the call stack is unwound. Such blocks are technically `FullBlock`s. If a block does not require a context, e. g. `[ :x | x + 1]`, the block is a `CleanBlock`.

There are three major uses of blocks:

1. Blocks in control structures (e. g. `<expression> ifTrue: [ statements ]`)
2. Blocks as iterators (e. g. `aCollection do: [ statements ]`)
3. Blocks as call-backs or continuations (e. g. `aSortedCollection sortBlock: [ statements ]`)

The first and the second case are straight-forward to convert to Java. The `Iterator` in Java will be more verbose than in Smalltalk but it is conceptually the same construct and the right way to go. The third case can often be handled by creating inner classes which maintain the block context. However, [Engelbrecht and Kourie, 1998] note that not all `FullBlock`s can be emulated by inner classes because scoping rules for inner classes do not match exactly the scoping of block contexts. Furthermore, it is not possible to decide if an emulation with an inner class will be save or not.

This has a number of consequences for the migration process. First, a developer has to *decide* between the three cases. Second, a good heuristic will get most of the cases *probably* right, but not 100%. Third, the problem is, that a certain amount of the decisions will be wrong (“bugs”) and need to be revised later. e. g. the `sortBlock` in the third example looks like a continuation, but later—when the collection will be sorted—the block will be

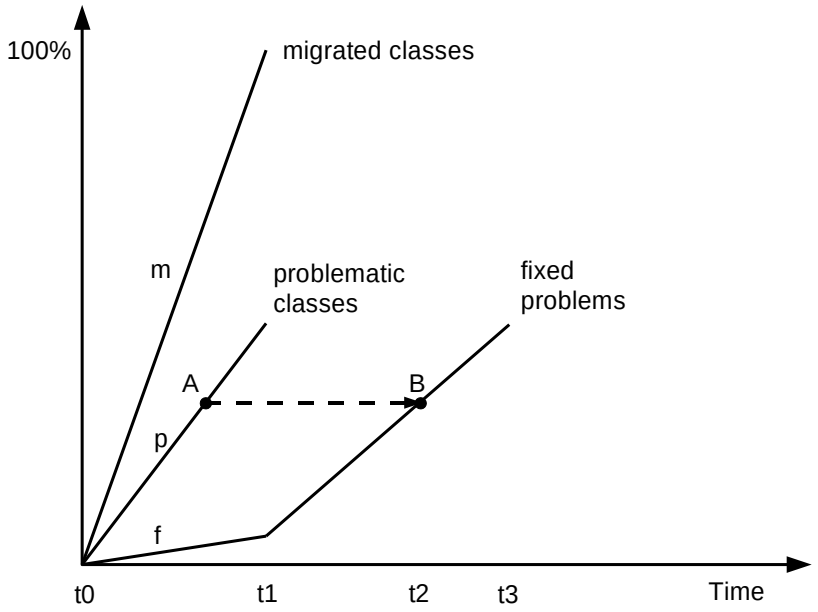

Figure 6.7: Resolution of migration problems without environment integration

used in an iterator construct as in the second example. The point is that one needs to live with this imperfections and carry a number of bugs through the migration process.

Figure 6.7 shows this in more detail. The purpose of the diagram is to show the rate of migrated classes together with the unresolved and the resolved problems. The diagram only shows the trend of these rates, not exact values. I assume that a developer uses a semi-automatic migration tool to convert an application class-by-class. Whenever the tool indicates a problem the developer tries to resolve it—if possible—immediately. Line m shows the migration of classes from 0% to 100%. Because not all classes can be migrated perfectly, a certain number of problems is introduced into the system (line p). When all of the classes are migrated, some of the problems have been resolved by the developer's intervention already (line f at time t1). However, a large number of problems remain to be fixed until the migrated application is fully functional (line f at time t3).

My point is that fixing the problems (from time t1 to t3) is very unproductive because the application is not executable. A developer needs to bootstrap the application starting with a core of classes and enriching it incrementally. Also, to understand the original application he needs to switch between the two development environments frequently. I described this situation earlier in section 2.5 on page 29.

I will now compare this development process with the piecemeal process shown in figure 6.8. I assume, as above, that the developer uses the same semi-automatic migration tools and that he introduces the same number of problems into the system (lines m and p).

Because the integrated programming environments support the piecemeal migration process well, the developer has a number of powerful options to resolve problems shortly after introducing them. The key feature of the environment integration is that the whole application is executable as a hybrid Smalltalk/Java system. Resolving problems early keeps the bug rate low and makes localizing and fixing bugs much easier than with a large number of bugs to carry around.

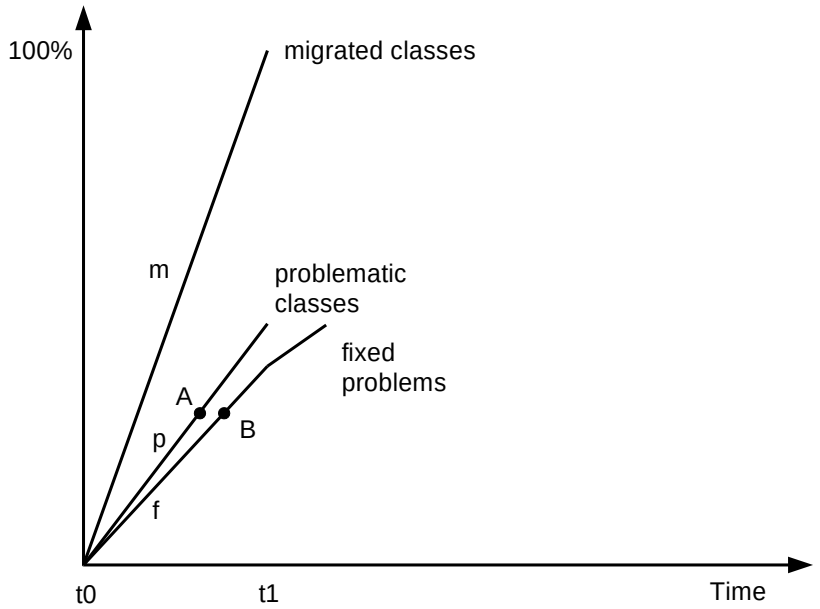

Figure 6.8: Resolution of migration problems with environment integration

- Because the full application is executable, it can be tested using automated tests (functional and unit tests). There are no missing classes as in the bootstrap process above. Automated tests make it easy to find bugs. A test case can be written to demonstrate the bug and later serve as a regression test. In the bootstrap process it would not make sense to write tests at all because they could not be executed. Unit tests could be written but this would result in a large overhead. I argued in section 6.2.2.1 that it is better to write functional tests. One should not attempt to cover the whole application with unit tests—that would be wasteful. Instead, writing test cases which cover whole execution paths though the system, e. g. as shown in figure 6.4, are more useful.

- Another benefit of testing is that tests can be used for program understanding. A test case can encode a developer's hypothesis and will validate this hypothesis every time the test is run. Of course, if these test cases were written to check hypotheses about Smalltalk classes initially, they can be reused for testing the migrated Java classes later on.

- Call-back problems (third use of blocks above) can be detected only in the running system because the problem does not occur in the class which created the block, but in the class that fails to perform the call-back. Those problems are much easier to detect in a debugger—which is only possible in the executable system as supported by the integrated programming environments.

- Writing SUnit tests without environment integration would be wasteful because theses tests could not be used to test Java classes. This is possible in the integrated environments and makes the overhead of writing test cases more economical. Another benefit of the test cases is that they support refactoring by giving the developer confidence in not breaking the code by aggressive refactoring.

- The semi-automatic migration tools mentioned in the two processes may include dynamic analyses of the Smalltalk system. In the integrated scenario it is possible to repeat those analyses with the hybrid Smalltalk/Java system to improve their results. The results are also more interesting because they involve increasingly more migrated Java classes which will be part of the final Java application.

- Assertions can be used in the hybrid system to encode a developer's assumptions about the application and decisions he made during the syntax conversion. This helps a lot in detecting mistakes because invalid hypotheses will be indicated immediately (during a test run). In the case of blocks one can add an assertion directly to a block which verifies that this block is, e. g., a `CleanBlock`.

The point of all this is that problems can be fixed shortly after introducing them. Figure 6.7 shows that without environment integration, a problem exists for a long time in the code (from point A to point B). Figure 6.8 shows that the resolution, point B, is reached much earlier. This has two benefits: 1. The bug rate, i. e. the vertical distance between lines p and f, is much smaller and thus allow a developer to work more productively. 2. Most of the problems will be fixed at time t1 already; only a small fraction remains to be fixed later. In other words, the integrated programming environments do not improve the output of specific migration tools, but make resolving the *unavoidable* migration problems a lot easier.

6.3.2 Typing

In this and the next section I do not repeat the comparison of the processes (figures 6.7 and 6.8) because the same issues apply here as well. I will first describe the problem and then how to deal with it more easily in the tool-supported piecemeal migration process.

Variables, method parameters, and return values are statically typed in Java. A—possibly polymorphic—type needs to be declared at compile-time. Smalltalk variables, parameters and return values are untyped; only the referenced objects have a type, i. e. a reference to a class at run-time. A variable may refer to instances of different classes at different times. Thus the type of a variable needs to include all potential classes that may be referenced by that variable.

For the syntax conversion from Smalltalk to Java it is mandatory to include type information in the generated source code. [Eßer et al., 2001] list three alternatives, each with its advantages and disadvantages. Only one alternative produces maintainable Java code—one of my requirements listed in section 2.2.5 on page 21. This alternative is to use the correct types in Java—the problem is only how to obtain this type information. I agree with Eßer et al. that a static type analysis can not be done efficiently for an application of typical complexity.

Instead, a number of heuristics can be used to obtain "good enough" type information. The static sources for type information include the naming of variables and comments. This assumes that a consistent naming and coding convention has been used throughout the code. e. g. from `anOrder add: anItem` one can guess that the involved types are probably `Order` and `Item`. Another source of type information is a dynamic type analysis. A complete dynamic type analysis would not be feasible because all potential execution

paths would have to be analyzed. An alternative to a complete analysis is a partial analysis which also fits well into an iterative process. Test cases can provide repeatable execution paths through an application, and dynamic observation can collect the classes of variables, parameters, and return values. For parameters and return values the type information can be collected using John Brant's Method Wrappers. Observing the variable assignments within a method can be done using reflection.

As above, the developer will create imperfect Java code at first. The heuristics provide type information which will be right in many cases, but not all. As above, this imperfect type information creates a number of problems in the migrated code. As above, fixing these problems is a lot easier when working with the integrated environments because the hybrid Smalltalk/Java application is executable. The dynamic techniques described here and most of the techniques described in the previous section can be used to improve the type information: testing, debugging, assertions, etc. The point is, as in the previous section, that with the integrated programming environments and the piecemeal migration process it is easier and faster to stabilize the migrated application.

6.3.3 Library Mismatch

Library mismatch is the problem that the class mapping (6.1.2.2) is only an approximation. Smalltalk classes are mapped to Java classes which provide roughly the same functionality but possibly in a different way. [Eßer et al., 2001] have studied the `Collection` and the `Magnitude` hierarchies in depth. They conclude that in most cases manual rework of the generated Java code is necessary.

The mapping is imperfect for almost every class. I will give some examples to show how trivial these problems appear to be at first, and how difficult a fully automated conversion would be. For example the `Date` and `Time` classes do not have an exact counterpart in Java. Java provides a `Date` and a `Calendar` class. The problem is that the factoring of the functionalities is different. In Smalltalk times and dates are clearly separated, in Java `Date` combines a date and a time. Fixing this problem requires refactoring the client classes of `Date` and `Time`. This is not supported by existing migration tools and has to be done manually.

Another example are arrays. Arrays expose two problems: The first problem is that Java provides arrays as part of the language—these arrays are not objects—and the class `Array`. Depending on how arrays are used in Smalltalk, it makes more sense to use the `Array` class, or the built-in arrays. This needs to be decided on a case-by-case basis.

The second aspect of arrays is that they use a different convention for accessing elements. Smalltalk arrays access the first element at index 1, Java arrays at index 0. This looks like a simple case but in fact it is not. Simply decrementing the index when accessing the array will produce not the desired result. e. g. if a method receives an index as a parameter, which is decremented already by the caller, then decrementing the index again will reference the wrong position. But another caller might invoke the method without decrementing the parameter first and the callee is expected to decrement it. The big problem of this kind of "off-by-one" bugs is that they are hard to track. The tracking gets even more difficult if indices are calculated from other indices. The compiler will not report any compilation

problems. Only at runtime a careful observation of inputs and outputs will detect these bugs.

Again, my point is that these migration problems can be resolved much easier using the dynamic techniques described in sections 6.3.1 and 6.3.2—which is only possible using the integrated programming environments.

6.4 Results

Finally, I describe what actually has been migrated using the proposed tool support. I cannot present a reference project because I left the initial sponsor of this work and previous employer of mine, DaimlerChrysler, before the prototype was ready to use. Instead, I demonstrate the feasibility of the integration architecture with a *test suite* which covers the important aspects of the implementation. This is quite similar to a company which develops a compiler and starts selling it; customers would not ask “did you write program X” using the compiler, but rather “does it work?” The developers will most certainly have assured themselves of this fact by developing a test suite for the specified features. This is also my approach.

A reference project is a large effort and it’s results are questionable. From my experience at DaimlerChrysler I can report two numbers to give an impression of the size of a typical migration project: a Smalltalk-to-Smalltalk migration project, mentioned on page 6, was estimated to cost several millions of dollars. In 2000 I was personally involved in the migration of a Smalltalk security framework to Java which had a total cost of about DM 600.000. Even reducing the size by a factor of 10 would require a major effort. The second issue is that typical Smalltalk applications contain a lot of redundancy. e. g. the user interface logic and the data access logic are repeated in a great number of classes—only the specific application logic differs. A more sensible approach is to trim the code size and consider the equivalence classes of important migration problems, e. g. as described in [Eßer et al., 2001]. I claim that the test suite I present below is a *representative* selection of migration problems one will discover in typical applications.

The test suite consists of 40 test classes with, on average, about 10 test methods each. The tests are implemented as subclasses of the `TestCase` base class supplied by the SUnit and JUnit testing frameworks. I developed the tests in parallel with the prototype following a test-first approach like in eXtreme Programming. The resulting tests are *chains* of test methods starting with trivial initial test and proceeding to the hard cases. e. g. when developing `TestBlocks`, see section 6.3.1, I started with a trivial empty block and then proceeded to more difficult cases like blocks with local variables and blocks with parameters, see table 6.2. Not all test methods in such a sequence of test methods are of the same value; the trivial test methods developed initially are obviously less valuable than those developed later on. The point is that a test-first development approach naturally leads to increasingly harder and therefore more interesting test cases. Each test class shown in table 6.2 has at least one—most classes have several—chains of interesting test methods.

The test cases differ from typical SUnit and JUnit tests in a few respects due to their distributed nature and the data they deal with:

Test class	Functionality tested by the test methods
TestScripting	evaluating scripts, returning results, both Smalltalk and Java (using BeanShell)
TestJPDA	Java-Java integration between Eclipse and the application VM (5.4.3)
TestCommunication	sending and receiving scripts to/from the Integrator and the opposite Connector, returning values, encoding/conversion of data
TestProxyGeneration	generating a proxy for a given class, both directions
TestClassMapping	mapping of class and method names (2.4.1.3)
TestSyntaxConversion	syntax conversion of syntactic elements, largest test class with 35 test methods
TestClassMigration	migrating an empty class, a class with class/instance variable declarations, with class/instance methods, with inheritance (5.4.3.1)
TestInstanceCreation	creating instances of migrated classes (5.4.3.2)
TestInvocation	calling remote instances using proxies, no/simple/complex parameters, conversion of parameters, proxy generation for parameters (5.4.3.3)
TestReverseInvocation	same as before for opposite direction, extra: conversion of parameter representation from Java to Smalltalk
TestTypes	simple run-time type analysis by type observation using Method Wrappers (6.3.2)
TestDoesNotUnderstand	detecting uses of the “does-not-understand”-idiom using Method Wrappers
TestBlocks	migrating empty block, block with parameter(s), block with local variables(s), full block (as inner class), block as parameter, block as return value
TestVariableAccess	classes accessing (inherited) instance, class, class instance, pool, and global variables
TestEquality	comparing instances for equality and identity, redefing `hashCode`
TestCopying	shallow and deep copying of complex objects with proxies, using `Cloneable` in Java
TestTests	running hybrid Smalltalk and Java tests
TestExceptions	detecting undeclared exceptions, handling declared exceptions, declaring thrown exceptions, extending the Class Mapping to exceptions
TestPerform	dynamic invocation of methods using reflection
TestIndexedClasses	migrating classes where instance variables are accessed by an index rather than a name
TestPools	migrating contents of pool dictionaries
TestGlobals	migrating global variables
TestAccessModifiers	creating appropriate `public` and `private` modifiers for instance variables and methods based on category information and “senders” analysis
TestPackaging	generating Java packages based on Smalltalk category information
TestRefactoringToProxy	refactoring of clients to use a proxy class instead of an original class
TestReplaceByProxy	hiding and/or removing of original class and replacing it by a proxy class

Table 6.2: Extract of test cases from the test suite

First, a test case, e. g. the test class `TestEquality`, consists of both a Smalltalk and a Java part. The Smalltalk part creates two instances of a class `C` and a copy of such an instance. Then it compares these instances for equality. When `C` is migrated to Java, the equality tests are repeated with instances of the migrated Java class. Technically, `TestEquality` has two implementations, one in Smalltalk and one in Java. The Java tests extend the Smalltalk tests; together both comprise a test with two physical parts which belong together logically. In this example I don not use the proxy mechanism to execute the Java tests from Smalltalk because it is more simple to write the test in Java. Also, the execution of the tests is faster without the Smalltalk-to-Java communication overhead.

Second, the test data of the test cases is not data in the sense of many different kinds of values but *source code*. Typical S/JUnit tests override the framework methods `setUp` and `tearDown` to create test data and clean it up after the test execution. My tests handle

test data in two ways: Simple tests contain the source code as simple methods of the test class, e. g. `TestBlocks` has a *test* method `testParameterizedBlock`. The corresponding test *data* method is `methodWithParameterizedBlock`. Tests which require the modification of an original class (or classes) *create* the original class(es) in the `setUp` method. e. g. in the example given in section 6.2.2.3 the class `Rectangle` needs to be changed to use a proxy once the class `Point` has been migrated and `PointProxy` is available. In such cases the modified test data has to be removed after the test execution in the `tearDown` method so that the next run can start with fresh test data. More complex tests, e. g. as shown in figure 6.6, involve 3-5 test data classes and are handled in the same way. The creation and removal of classes and methods is an expensive operation; it is desirable to reuse test data as much as possible, e. g. by writing fewer but longer test methods.

Third, the migration tests operate at two levels rather than one: at the level of classes *and* at the level of instances. Tests generally execute the code under test and compare the obtained results with the expected values. In that sense, the migration tests compare the migrated Java code with the expected Java code (class level). In addition to that, the tests also check instances of the original and the migrated classes (instance level): they *execute* the original and the migrated code to verify if they produce the same—or equivalent—results.

This test suite plausibly demonstrates that the prototypical implementation of the integration architecture works.

Chapter 7

Summary and Conclusions

7.1 Summary

In chapter 1 I motivated the problem of migrating applications. Platforms have their own life-cycle and when they get old, it is desirable to migrate applications to newer platforms. Chapter 2 presented the issues related to a migration. One can choose between different migration paths and migration processes. Some processes are more productive than others. However, only a process which is supported appropriately by tools can be used.

In section 2.5 I stated the problem that there is no productive migration process with adequate tool support. Various tools can be used to perform a migration but they are not integrated well. If these tools were integrated better, a migration process could be more productive.

I reviewed integration technology in section 3. Existing tool integration mechanisms have been designed to integrate tools within an integrated development environment (IDE), or to integrate heterogenous tools—such as IDEs and CASE tools. Only the tight integration within IDEs supports a productive development process. The loose data integration of heterogenous tools is not sufficient. All existing integration mechanisms have two limitations: they are designed for integration *within* some environment, and they do not address the integration of the development and the runtime environment well, e. g. as in Smalltalk (3.5.3.4). I assume that a developer will use the two IDEs of the source and the target platform. Each of these IDEs uses its own integration approach within its environment. What is missing is 1. the integration *between* the two development environments and 2. the integration of the two runtime-environments.

My thesis (4) is that the programming environments of the source and the target platform can be integrated using message passing; this integration of the environments supports a *piecemeal* platform migration process.

Chapter 5 describes an integration architecture for programming environments based on message passing. The implementation uses Smalltalk and Java as source and target platforms. For these platforms I have implemented a prototype of the integration architecture. In chapter 6 I showed how to use the integrated IDEs in a piecemeal migration process. The interesting aspects of the integration architecture and the piecemeal migration process are summarized in the next section (7.2).

7.2 Contributions

The two main contributions of my work are the integration architecture for programming environments and the incremental migration process. Both depend on each other—the tool support facilitates the process, and the process requires the tool support. The main line of thought is (see also section 2.5.3):

1. The integration architecture (5) addresses the integration of the development environments and the runtime environments of both platforms.
2. The integration of the programming environments of the source and the target platforms allow a developer to work with a unified code base.
3. This new kind of integration between the programming environments supports a productive, *piecemeal* migration process (6).
4. The new level of tool support provided by the integrated programming environments makes a migration easier and less risky.

7.2.1 Integration Architecture

The integration architecture is based on the idea of message passing [Reiss, 1990]. The tools in Reiss' Field environment communicate with each other by sending messages in a fixed text format over raw sockets to a central message server. This server forwards the messages to designated recipients.

My integration architecture extends message passing in several ways and adds new solutions to the integration technologies reviewed in chapter 3. First, my integration architecture integrates tools at a new level. I integrate programming environments which are integrated within themselves by some means already. Usually, tool integration happens within some kind of environment, e. g. an IDE, but in my case the two programming environments do not share such an encompassing environment. Their architectures are given and it is not desirable to spend a lot of effort on adapting them to a new encompassing environment. Instead, the integration architecture is adaptable to the programming environments.

Second, the integration architecture uses a number of powerful features of the programming environments, especially their openness. Today, programming environments usually provide an extension mechanism for adding functionality (plug-ins). My integration architecture uses this openness to add a Connector component to each environment. The purpose of the Connector is to communicate with the other environment via the Integrator component. The usual way to use an environment's plug-in mechanism is to make new functionality available inside an environment. On the other hand, in my integration architecture the Connector component is used to expose the environment's functionality to the outside world.

Third, the integration architecture relies on and exploits the reflectiveness of the programming environments (3.2.1.3 and 3.5.3.4). A reflective environment contains a representation of itself (self-representation). In a reflective environment both the tools and the code base are represented as objects. Reflection is a key feature for supporting the

piecemeal migration process which requires frequent changes of the application's code. These changes are made to the self-representation of the application, not to files containing source code. This way, a developer can modify the running system without having to restart the application.

Fourth, the integration architecture uses a flexible message format based on the idea of scripts. Messages are sent as plain text with the only requirement that the contents of a message have to be syntactically correct. There is no fixed message format like in Field. This design relies on the environments being interpretive. It would not be possible using a statically compiled language. These messages are used both for program elements (classes or methods) and for data.

Fifth, the integration architecture integrates both the development environments and the runtime environments. This kind of integration can be found e. g. in every Smalltalk environment. Development tools and application reside in the same environment—there is no difference between them. In my integration architecture I integrate the Smalltalk tools with the Java tools, and the Smalltalk part of an application with the Java part, see figure 4.2 on page 63.

7.2.2 Piecemeal Migration Process

The piecemeal migration process emphasizes short development cycles and immediate feedback from coding activities. This emphasis is inspired by eXtreme Programming. The feedback cycle is accomplished by testing (optional) and by keeping the application always executable—which is supported by the tool integration architecture. Conceptually, the process is based on well-established elements. It uses a two-level description (macro and micro process), phases to group related tasks, and an incremental development strategy.

The distinguishing feature of the migration process is that it is *piecemeal* (2.5.2 and 4.1). Today incremental approaches are common practice in forward engineering. Modern IDEs support incremental development well by providing features such as incremental compilers. My integration architecture supports an incremental process also for migrations. To distinguish the incremental migration process from forward engineering I call it *piecemeal* migration process.

The key aspect of the piecemeal migration process is that a developer transforms the original application during the migration piece-by-piece. Rather than rewriting the target application from scratch, or converting all code to the target platform and fixing the problems from scratch, a developer migrates one piece of the application at a time. Each migrated piece will be integrated with the original application so that the overall system is executable—and testable to provide feedback.

The benefit of this approach is that the unavoidable manual work can be done in a productive way. The increased productivity and the short feedback cycles lower the risk associated with migration projects and make them easier to plan.

7.3 Future Research

When designing—and later implementing—the integration architecture I came across several questions that would be worthwhile to investigate further.

Cutting a arbitrary distribution boundary into an application A system design usually has a well-defined boundary between distributed nodes. During the piecemeal migration the developer works with a hybrid distributed system. The distribution boundary is changed in every iteration. If application classes have inheritance relationships to framework classes, it can be hard to find the right unit-of-work for a short iteration. Ideally, tightly coupled classes would be migrated in one iteration but if the cluster of coupled classes contains many classes, the iterations would have to be longer. My design does not support distributed inheritance between classes of different platforms—and I would not suggest to attempt this. An interesting question is how to cluster the application classes into units-of-work so that the units are small enough to allow for short iterations.

Specifying the differences between platforms. The differences between Smalltalk and Java are many. Even though the languages are similar, the class libraries are quite different. Since there is no formal specification of the behavior of the library classes, it is hard to apply formal methods. It would be interesting to investigate the possibility to formally describe the *difference* between two platforms even if there is no formal specification for them. One approach could be a comparison of ports of an application.

Describing the cross-platform refactorings. If there were a formal description of the differences between two platforms, a tool could suggest to a developer what to do with some piece of code—and possibly apply a refactoring automatically. Refactorings preserve the semantics of a program. e. g. the Smalltalk refactoring browser supports a number of refactorings at the language level. This is possible because the semantics of the Smalltalk syntax are much clearer than the behavior of library classes. In a migration, refactorings should preserve the semantics as well—at least at a high level. At lower levels it might be necessary to adapt an application to the style of the target platform. e. g. changing the exception handling from Smalltalk to Java would not be semantics preserving at lower levels. The interesting question is how to describe the refactorings together with pre- and post-conditions without a precise notion of the semantics of the class libraries.

Metrics Migration projects are risky and expensive. It is desirable to have some metrics which roughly estimated the effort to be expected by a migration. Such metrics could also be used to decide between different target platforms. If the two previous questions were answered, one could investigate the required efforts in business terms, e. g. person months.

7.4 Conclusions

Migration is a difficult topic and I expect it to become even more difficult in the future. Innovation and market forces will *always* create new platforms. Also the number of program-

ming languages will increase. e. g. aspect-oriented programming (AOP) promotes the idea of separation of concerns and using a domain-specific language for each concern. The Java platform has demonstrated that the libraries will grow as well, e. g. it includes APIs for domains such as telephony and smart-cards, among others. The growth of languages and platforms together with two other reasons make me believe that migrations will become harder. First, each person has a limited mental capacity. For me it was quite challenging to become compentent both in Smalltalk and Java at the same time—a necessary precondition for migration. The growth of platforms will exceed what an individual can master, but a migration requires mastery of both platforms. Not being able to staff migration projects with the right people increases the risk of migration projects and the reluctance to attempt them at all. My second reason is that I do not see any formal approach yet. New platforms, e. g. Microsoft's .NET, are in no way easier to maintain because their semantics are as informal as the semantics of Smalltalk and Java. Of course, no vendor has an interest to lower the exit barrier for its customers. My point is that tool support for migration is essential but the informality of the platforms prohibits formal approaches. This raises some interesting questions which I sketched in the previous section. I expect the amount of legacy applications to grow and to become much more of a burden for many organizations. Legacy systems are defined as systems that resist change. I would add that legacy systems also resist migration.

My thesis is a pragmatic approach to deal with the manual work which will remain until the platforms permit the application of formal methods.

Bibliography

[Agesen, 1995] Agesen, O. (1995). *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Stanford University.

[Amsden, 2001] Amsden, J. (2001). Levels of Integration—Five Ways You Can Integrate with the Eclipse Platform. Object Technology International. Available online at `http://www.eclipse.org/articles`.

[Architur, 2001] Architur (2001). Talks2 whitepaper. Technical Report, Architur GmbH. Available online at `http://www.architur.de`.

[Armour, 2000] Armour, P. G. (2000). The Business of Software: The Case for a New Business Model. *Communications of the ACM*, 43(8).

[Bapst and Krone, 1997] Bapst, F. and Krone, O. (1997). A Coordination Kernel for Coupling Heterogeneous Programming Environments. Technical Report 97-03, Institute of Informatics, EIF Fribourg.

[Barrett, 1994] Barrett, D. (1994). Building a Demo: A Comparison of Three Software Integration Mechanisms. Technical Report UM-94-01, Arcadia Consortium. `http://www.blazemonger.com/publications.html`.

[Beck, 1997] Beck, K. (1997). *Smalltalk Best Practice Patterns*. Prentice Hall.

[Beck, 1999] Beck, K. (1999). *eXtreme Programming Explained: Embrace Change*. Addison–Wesley.

[Belkhatir and Melo, 1993] Belkhatir, N. and Melo, W. L. (1993). Supporting Software Maintenance Processes in TEMPO. In *Proceedings of the Conference on Software Maintenance*. IEEE Computer Society Press.

[Bothner, 1996] Bothner, P. (1996). Translating Smalltalk to Java. Technical report, Cygnus Solutions. Available online at `http://www.bothner.com/~per/papers/smalltalk.html`.

[Boudier et al., 1989] Boudier, G., Gallo, F., Minot, R., and Thomas, I. (1989). An Overview of PCTE and PCTE+. *ACM SIGSOFT Software Engineering Notes*, 13(5).

[Boyd, 1997] Boyd, N. (1997). Towards Smalltalk and Java Language Integration. Technical Report. Available online at `http://home.adelphia.net/~nikboyd/papers/sttojava/`.

[Boyd, 1999] Boyd, N. (1999). Introducing Jist. Technical Report. Available online at `http://home.adelphia.net/~nikboyd/papers/jist/`.

[Boyd, 2000] Boyd, N. (2000). Smalltalk over Java: An Introduction to Bistro. Technical Report. Available online at `http://home.adelphia.net/~nikboyd/papers/bistro/intro/`.

[Brown et al., 1992] Brown, A. W., Feiler, P. H., and Wallnau, K. C. (1992). Understanding Integration in a Software Development Environment. Technical Report CMU/SEI-91-TR-031, Software Engineering Institute, Carnegie Mellon University.

[Brown and McDermid, 1992] Brown, A. W. and McDermid, J. A. (1992). Learning from IPSE's Mistakes. *IEEE Software*, 9(2):23–28.

[Brown and Penedo, 1992] Brown, A. W. and Penedo, M. H. (1992). An Annotated Bibliography on Integration in Software Engineering Environments. *Software Engineering Notes*, 17(3):47–55.

[Cagan, 1990] Cagan, M. R. (1990). The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett–Packard Journal*, 41(3):37–47.

[Chikofsky and Cross, 1990] Chikofsky, E. J. and Cross, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17.

[Christensen, 1999] Christensen, H. B. (1999). The Ragnarok Software Development Environment. *Nordic Journal of Computing*, 6(1):4–21.

[Conway, 1968] Conway, M. E. (1968). How Do Committees Invent. *Datamation*, 14.

[Cook, 1992] Cook, W. R. (1992). Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press.

[Cuthill, 1994] Cuthill, B. (1994). Making Sense of Software Engineering Environment Framework Standards. *StandardView*, 2(2):188–202.

[Demeyer et al., 1999] Demeyer, S., Ducasse, S., and Tichelaar, S. (1999). Why FAMIX and not UML? In *Proceedings of UML '99*.

[ECMA, 1993] ECMA (1993). Reference Model for Frameworks of Software Engineering Environments. Technical Report TR/55, ECMA.

[ECMA, 1997] ECMA (1997). Portable Common Tool Environment (PCTE): Abstract Specification. Standard ECMA-149, 4th edition, ECMA.

[Elsner, 1998] Elsner, G. (1998). Von Smalltalk zu Java—einfach konvertieren! In *Proceedings of Smalltalk und Java in Industrie und Ausbildung (STJA)*, Erfurt, Germany.

[Engelbrecht and Kourie, 1998] Engelbrecht, R. L. and Kourie, D. G. (1998). Issues in Translating Smalltalk to Java. In Koskimies, K., editor, *Compiler Construction*, number 1383 in LNCS. Springer-Verlag.

[Eßer et al., 2001] Eßer, M., Müller, M., and Barthel, S. (2001). Migration von Smalltalk nach Java. Internal report, FT2/SA, DaimlerChrysler AG.

[Feathers, 2002] Feathers, M. (2002). Working Effectively with Legacy Code. Object Mentor, Inc. Available online at `http://www.objectmentor.com`.

[Feldman, 1979] Feldman, S. I. (1979). A Program for Maintaining Computer Programs. *Software–Practice and Experience*, 9(4):255–265.

[Fowler, 1999] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[Fussell, 1997a] Fussell, M. (1997a). Java and Smalltalk Syntax Compared. ChiMu Corporation. Available online at `http://www.chimu.com/publications/JavaSmalltalkSyntax.html`.

[Fussell, 1997b] Fussell, M. (1997b). SmallJava: Using Language Transformation to Show Language Differences. ChiMu Corporation. Available online at `http://www.chimu.com/publications/smallJava/index.html`.

[Gabriel, 1996] Gabriel, R. P. (1996). *Patterns of Software*. Oxford University Press.

[Garau, 2001] Garau, F. (2001). Concrete Type Inference in Squeak. Licenciate Thesis. Available online at `http://typeinference.swiki.net/1`.

[Gautier et al., 1995] Gautier, B., Loftus, C., Sherratt, E., and Thomas, L. (1995). Tool Integration: Experiences and Directions. In *Proceedings of the International Conference on Software Engineering*, IEEE Computer Society Press.

[GEBIT, 2001] GEBIT (2001). MOVE/Smalltalk to Java (S2J) Online Documentation. Available online at `http://www.gebit.de/Loesungen/produkte/Migration.htm`.

[Gittinger, 1999] Gittinger, C. (1999). Die Unified Smalltalk/Java Virtual Machine in Smalltalk/X. In *Proceedings of NetObjectDays*, Erfurt. Germany. Available online at `http://www.netobjectdays.org/pdf/99/stja/ClausGittinger.pdf`.

[Goldberg, 1984] Goldberg, A. (1984). *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.

[Grundy et al., 1999] Grundy, J., Mugridge, W., and Hosking, J. (1999). Constructing Component-Based Software Engineering Environments: Issues and Experiences. In *Symposium on Constructing Software Engineering Tools*, Los Angeles. Available online at `http://www.cs.auckland.ac.nz/~john-g/publications.html`.

[Grundy and Hosking, 2001] Grundy, J. C. and Hosking, J. G. (2001). *Wiley Encyclopaedia of Software Engineering*, Chapter Software Tools. Wiley InterScience, 2nd edition.

[Grundy et al., 1998] Grundy, J. C., Mugridge, W. B., Hosking, J. G., and Apperley, M. D. (1998). Tool Integration, Collaboration and User Interaction: Issues in component-Based Software Architectures. In *Proceedings of TOOLS Pacific*, Melbourne, Australia. IEEE CS Press. Available online at `http://www.cs.auckland.ac.nz/~john-g/publications.html`.

[Jacobson et al., 1992] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press/Addison-Wesley.

[Kaiser et al., 1997] Kaiser, G. E., Dossick, S. E., Jiang, W., and Yang, J. J. (1997). An Architecture for WWW-based Hypercode Environments. In *Proceedings of the International Conference on Software Engineering*, IEEE Computer Society Press.

[Kapp, 1995] Kapp, A. C. (1995). Vergleich der Designarchitekturen zweier Smalltalksysteme: Konsequenzen der getroffenen Designentscheide. Master's thesis, Uni Basel.

[Karrer and Scacchi, 1994] Karrer, A. S. and Scacchi, W. (1994). Meta-Environments for Software Production. *International Journal of Software Engineering and Knowledge Engineering*, 3(1):139–162.

[Kernighan and Pike, 1984] Kernighan, B. W. and Pike, R. (1984). *The Unix Programming Environment*. Prentice Hall.

[Kienle et al., 2000] Kienle, H. M., Czeranski, J., and Eisenbarth, T. (2000). Exchange Format Bibliography. In *Workshop on Standard Exchange Format (WoSEF)*, pages 2–9, Limerick, Ireland. Available online at `http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus/papers/index.html`.

[Kowalski et al., 1991] Kowalski, T., Sequist, C., Ellis, B., Goguen, H. H., Puttress, J. J., Castillo, C. M., Rowland, J. R., Rath, C. A., Wilson, J. M., Vesonder, G., and Schmidt, J. L. (1991). A Reflective C Programming Environment. In *Proceedings of International Workshop on UNIX-Based Software Development Environments*. USENIX.

[Li, 1998] Li, J. (1998). *Static Program Analysis to Support the Software Maintenance of Object-Oriented Applications*. PhD thesis, Institut für Informatik, Uni Stuttgart.

[Lin and Reiss, 1993] Lin, Y.-J. and Reiss, S. P. (1993). An Object-Centered Approach to Designing Programming Environments. Technical Report CS-93-38, Brown University. Available online at `http://www.cs.brown.edu/publications/techreports/reports/CS-93-38.html`,.

[Long and Morris, 1993] Long, F. and Morris, E. (1993). An Overview of PCTE: A Basis for a Portable Common Tool Environment. Technical Report CMU/SEI-93-TR-1, Software Engineering Institute, Carnegie Mellon University.

[Meyers and Reiss, 1995] Meyers, S. and Reiss, S. P. (1995). A System for Multiparadigm Development of Software Systems. Technical Report CS-91-50, Brown University. Available online at `http://www.cs.brown.edu/publications/techreports/reports/CS-91-50.html`.

[Ossher et al., 2000] Ossher, H., Harrison, W., and Tarr, P. (2000). Software Engineering Tools and Environments: A Roadmap. In *Proceedings of the International Conference on Software Engineering*, pages 261–277, IEEE Computer Society Press.

[Osterweil, 1987] Osterweil, L. J. (1987). Software Processes Are Software Too. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society Press.

[Perry and Kaiser, 1991] Perry, D. E. and Kaiser, G. E. (1991). Models of Software Development Environments. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society Press.

[Purtilo, 1994] Purtilo, J. M. (1994). The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174.

[Reiss, 1995] Reiss, S. (1995). Fragments: A Mechanism for Low Cost Data Integration. Technical Report CS-95-13, Brown University. Available online at `http://www.cs.brown.edu/publications/techreports/reports/CS-95-13.html`.

[Reiss, 1990] Reiss, S. P. (1990). Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(4):57–66.

[Reiss, 1996] Reiss, S. P. (1996). Simplifying Data Integration: The Design of the Desert Software Development Environment. In *Proceedings of the International Conference on Software Engineering*, pages 398–407, IEEE Computer Society Press.

[Sharon and Anderson, 1997] Sharon, D. and Anderson, T. (1997). A Complete Software Engineering Environment. *IEEE Software*, 14(2):123–127.

[Shaw and Garlan, 1996] Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspective on an Emerging Discipline*. Prentice-Hall.

[Stavridou, 1999] Stavridou, V. (1999). Integration in Software Intensive Systems. *The Journal of Systems and Software*, 48(2):91–104.

[Stotts and Purtilo, 1994] Stotts, P. D. and Purtilo, M. (1994). Virtual Environment Architectures: Interoperability through Software Interconnection Technology. In *Proceedings of the Third Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 221–224.

[Sun Microsystems, 2002] Sun Microsystems (2002). Java Platform Debugger Architecture Documentation. `http://java.sun.com/products/jpda/doc/`.

[Thomas and Nejmeh, 1992] Thomas, I. and Nejmeh, B. A. (1992). Definitions of Tool Integration for Environments. *IEEE Software*, 14:29–35.

[Wallnau and Feiler, 1991] Wallnau, C. K. and Feiler, P. (1991). Tool Integration and Environment Architectures. Technical Report CMU/SEI-91-TR-11, Software Engineering Institute, Carnegie Mellon University.

[Wasserman, 1989] Wasserman, A. I. (1989). Tool Integration in Software Engineering. In Long, F., editor, *Software Engineering Environments*, number 467 in LNCS, pages 137–149. Springer-Verlag.

[Wege, 1998] Wege, C. (1998). Vergleich der Anwendungsarchitekturen von Smalltalk und Java im Hinblick auf Migration. Master's thesis, Universtiy of Tübingen. Appeared as Technical Report WSI-98-12 of the Wilhelm-Schickard-Institut für Informatik. Available online at `http://www.informatik.uni-tuebingen.de/bibliothek/wsi-reports/wsi-98-12.pdf`.

[Wirfs-Brock, 1996] Wirfs-Brock, A. (1996). A Declarative Model for Defining Smalltalk Programs. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press.

[Zelkowitz, 1993] Zelkowitz, M. V. (1993). Use of an Environment Classification Model. In *Proceedings of the International Conference on Software Engineering*, pages 348–357. ACM Press.