Tool Support for Software Migration Through Integration of the Programming Environments of the Source and Target Platforms

Frank Gerhardt
fg@acm.org

July 24th, 2001

Abstract: Specific tools to migrate an application from one platform to another are hard to find. The diversity of platforms to migrate from and to makes it difficult to create specific migration tools. The lack of specific tools causes developers to use general purpose tools in an isolated way. We propose to integrate the programming environments of two platforms so that a developer can work on a migration with general purpose tools in an integrated way. This integration permits a productive working style for platform migration that is similar to forward engineering (continuous integration, unit testing etc.).

Keywords: reengineering, application migration, programming environments, tool integration.

Contents

1 Introduction

This thesis is about *tool support for migrating applications* from one development platform to another. We begin with an introduction to software migration. In section 2 we focus on tool support for migration and identify the lack of tool support as our research problem. In section 3 we state our research goal. Our approach is presented in section 4.

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. We call such a bundle of a programming language and reusable code a *development platform*. For now we don't consider the tools of a development platform.

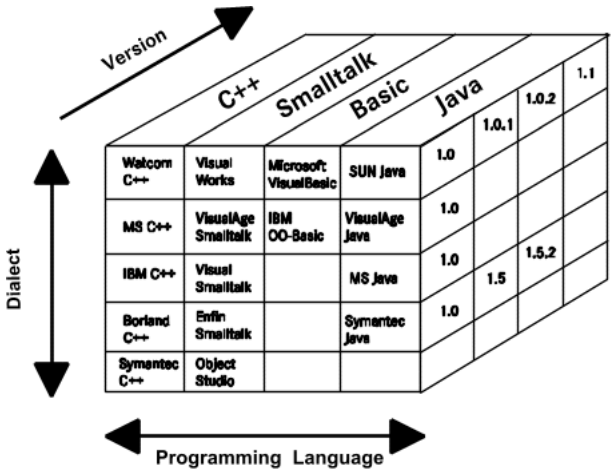

Figure 1: The dimensions of platform variety

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 and vendors try to be compatible with each other. But looking at the whole breadth of a development platform, only a narrow part is covered by a standard. Platforms differ in specific class libraries e.g. for graphical user interfaces.

Figure 1 shows three dimensions of platform diversity: 1. the different languages, 2. the dialects of the languages, and 3. the releases of implementations over time.

When a system is built a specific platform has to be chosen. 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 NT 4.0 SP5).

We see that there is a large number of possibilities (points in the three dimensions). As we will address the problems with migrating code from one development platform to another later, we will refer to the problem of having so many different source and target platforms.

The advantage or this diversity is that the choice for selecting the right tool for a specific task is great. The diversity is so big that there are often numerous possibilities even for specific problem domains. There isn't a problem if unrelated projects select different development platforms. However, the diversity of platforms causes problem in a corporate environment.

1.2 In a Corporate Environment

Large corporations build many systems over time. Large corporations usually have an IT strategy that defines which development platforms to use. For example Java is one the strategic platforms of many companies today.

Over time strategies change. Companies will find that their IT infrastructure consists of many different platforms. Each system they built had to decide for a specific platform (language, dialect, version).

Many of those systems were very successful and therefore are still in operation today. Since the business world changes over time, these systems have to be adapted to those changes by maintenance programmers.

To have a system that uses a specific platform in operation means for a company to keep this platform alive, to *maintain* this platform. While the vendor of a platform maintains the technical implementation, a company that uses this platform has to maintain the organisational 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.

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 tool for a specific problem. What can happen—over time—that a platform isn't the best choice any more and that one wants to migrate one or more applications to a new platform?

The problems that the use of platform can cause can lie in the platform (or product) itself or in the organisational "fit" of a platform.

1.3.1 Problems Related to the Platform

The problems with a development platform can be non-technical or technical.

The major non-technical problem is competition. Science progresses and new platforms emerge. One reason might simply be that there are better alternatives in the meantime. For example Java got a lot of attention recently and has improved it's 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 in this competition there are winners and losers. New platforms can cause old platforms to "die".

Other non-technical problems are more related to the vendor of a platform than 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 won't be maintained and enhanced as expected. One example for this situation if the disappearance of VisualSmalltalk from the market after there merger of its vendor with another Smalltalk vendor.

The number technical problems can be huge. 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 aren't. Sometimes the customer can help himself to work abound a deficiency but in many cases that would require too much effort.

1.3.2 Problems Related to the Organisation

The diversity of platforms and it's organisational consequences cause of number of problems for large organisations. In the end these problems are all about money, or *total cost of ownership* (TCO).

The problems fall into three categories: 1. the cost of having a platform, 2. the additional cost for having a possibly outdated platform, and 3. 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 have to be developed in-house. This increases the cost of an old platform compared to another, newer platform that will have 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. There is a higher cost for system 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 over-used might create problems themselves, e.g. a diversity of middle-ware products.

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

For all 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. For example, at DaimlerChrysler for *one* system the migration was estimated to cost several millions of Dollars. The risks involved are hard to assess.

To achieve the full benefit of abandoning a platform *all* system using this platform would have to be migrated: 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 high-tech too big.

1.4 Different Migration Paths

There are different kinds of migrations and it's important to understand which one addresses which problems. It is useful to differentiate these kinds of migrations by what one wants to retain, and what one wants to abandon.

The major differentiation is between migration paths to abandon the execution environment of a platform and paths to abandon the source language.

For each kind of migration paths we look at *compiled*, *interpreted*, and *compiled-interpreted* development platforms. An example for a compiled-interpreted platform is Java, where the source code first gets compiled to an intermediary byte code

which, in turn, is interpreted by a virtual machine. For software development there is not a big difference between an interpreted and a compiled-interpreted language; for a migration this differentiation allows one to "make the cut" at one more layer.

1.4.1 Retaining the Source Language

The first category of migrations addresses the next lower level from the source language. In the case of a compiled-interpreted language there are two next lower levels.

From an application perspective the goal is to keep the system running with minimal changes to the source code. In that case one wants to give up the underlying runtime platform and replace it with a new runtime platform. A migration of source code is not desired.

For a compiled language the migration can be done by a recompile (if supported) on the new runtime platform (e.g. operating system).

For an interpreted language a port of the interpreter needs be obtained for the new runtime platform. For example interpreters for Perl are available for many platforms and the migration is easy in this case.

For an interpreted-compiled language the situation is a bit more complex. Here the code is interpreted by an underlying interpreter (or VM), which runs on an underlying operating system (or directly on the hardware). If the goal is to abandon the lowest level (OS or hardware), the migration can be done by using the interpreter on a new runtime platform. To abandon the interpreter (or VM) in such a scenario will hardly be desirable because interpreter and language are usually tied together closely.

Although, there are two alternative solutions for the latter problem. 1. A cross-compiler can be used to compile byte-code for the new interpreter, e.g. Smalltalk source code can be cross-compiled for a Java virtual machine (this doesn't solve all problems). 2. If a universal virtual machine (UVM) is available both old and new byte-code can be executed by the same interpreter. This achieves inter-operability between old and new code but one is still tied into the old development platform for maintaining the old source code. This option is less desirable because one can't abandon an old development platform as a whole.

1.4.2 Migrating Away From the Source Language

The second category of migrations addresses the source code. This usually implies a new runtime platform as well, but not necessarily. Such a migration allows one to abandon the old development platform completely.

In this kind of migration the source code is translated to the programming language of the target platform.

There are two alternatives:

1. Emulation of the old platform on the new platform by a compatibility layer on top of the new platform.
2. Conversion/translation of the code to the style of the new platform.

An example of the first option is a translation from Cobol to C. Cobol data structures are mapped to C structures. Library calls are emulated in the compatibility layer. The whole systems reads like "Cobol in C".

At first sight this option looked like a clean migration to the target platform but the old runtime environment is still "there". It is hidden in the compatibility layer. For this reason one can conclude that this option could be considered to belong to the first category of migrations since it emulates the old environment in the new. It "misuses" the new platform to rebuild the old infrastructure.

The biggest disadvantage of this approach is that the code will be very hard to understand for somebody familiar with the target platform. Although, it might look familiar to an expert in the old platform. Also, the code might be slow because of the added compatibility layer.

In the second option the source code is translated to the *style* of the the target platform. In the example above this migration path would have resulted in a C system written in C style, C in C instead of Cobol in C.

Looking back at the problems caused by platform heterogeneity we can conclude that only this migration path towards the style of the target platform achieves the following benefits:

- The old platform can be abandoned completely since the 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 organisation to concentrate its efforts on the (strategic) target platform.
- Because there is one platform less and more focus on the (strategic) target platform it is easier to 1. transfer knowledge and people between projects and 2. reuse code between projects.
- The overall IT infrastructure is more flexible and easier to understand, the business can respond to change easier.

2 The Problem: Tool Support for Software Migration

This section introduces what has to be done in a migration in general and what tools can help in doing that. There are dedicated migration tools and general purpose

tools like IDEs etc. We show that given the diversity of development platforms it is unlikely to find a specific tool to perform a specific migration automatically. A developer will have to rely on the tools he already has and knows. In particular these are the IDEs of the source and target platform. The key problem identified here is that using two separate IDEs side-by-side is not a productive way to perform the migration.

2.1 Specific Migration Tools

A dedicated migration tool has to support the following tasks:

- Syntax conversion
- Identification of dependencies into the source platform
- Transformation of code to use the new platform (a "re-binding" of the dependencies)

Many tools available on the market focus on the first step and come in the form of code converters or translators.

Tools that address also the re-binding of decencies are very specific to the platforms they are designed for. The usually have some rule base that defines the mapping of features from the old platform to the new.

Tools like this have natural limitations. 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 code in terms of the target platform. Given the complexity of platforms like Smalltalk and Java it takes at least several months if not years to master each. Developers with these skills are difficult to find. Even more difficult is to automate this task by encoding this knowledge in a software tool.

The development of such a specific migration tool is expensive. First, because of the difficulty of the problem itself. Second, because of the standards that modern IDEs set. Developers expect a similar standard from a migration tool as well. E.g. today IDEs support incremental compilation. The implementation is hard, but not impossible. To support a similar feature in a migration tool is challenging.

Let's imagine such a tool for a moment. Let's assume it's a great piece of work and it works perfectly. Let's look at the market for this tool. The strange thing is that the more the tool gets used, the market shrinks. Because of this it is hard to develop a viable business model for the investment.

Conclusion: Specific migration tools for a given specific problem are rare (and often of bad quality).

2.1.1 Limitations of Specific Migration Tools

Specific migration tools often leave a developer where in a situation where 80% of the code had been converted successfully but the remaining 20% have to be fixed by hand. Then a developer has to use other tools still.

2.2 General Purpose Development Tools

There are a number of tools that can all be useful in a migration. First, of course, the IDEs of the source and the target platform. Often there a many additional tools available on both platforms that extend each IDE.

There are also external tools with a command-line or file interface, e.g graph visualisation tools that can display a system's structure encoded in some graph exchange file format.

2.2.1 What IDEs offer

For a migration a developer will certainly use an IDE for the target platform. Possibly he will also use the IDE of the source platform to understand the application better.

Many features of an IDE will turn out to be very useful in a migration. Code 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 analyse and modify individual objects 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. An incremental compiler helps compiling code quickly, even at runtime, so that one always has an executable system to work with.

However IDEs have no specific support for migration. They were never intended for this, of course.

2.2.2 The Working Style Supported by IDEs

Modern IDEs allow a developer to work in a very productive way.

Especially an incremental compiler permits short cycles between editing and running code. Frequent unit testing becomes possible. The inspectors and the debugger allow a program to be analysed not only statically—as in the class browser—but also dynamically. Executions can be stopped, the state of the system can be modified, code can be recompiled and the execution can be resumed.

Both the features mentioned in the previous section and the process outlined here are desirable also for a migration—especially when a migration has been only partially completed ("80% done", sec. 2.1.1) to fix the last problems.

2.2.3 Add-on Tools for IDEs

IDEs are often extended by add-on tools for specific tasks:

- type/code/data-flow analysis
- profiling
- dependency analysis (cross-referencing)
- refactoring [Fow99]
- visualisation
- metrics collection
- style checking, e.g. lint
- detection of code duplication
- quality assurance
- testing

Many tools like these have been developed as research prototypes or are commercially available. In providing tool support for migration it would not make sense to re-implement this functionality in a new A-to-B migration tool. Instead the functionality that is already available should be used.

2.3 Working with Two Separate IDEs Side-By-Side

In most cases the developer will not find a specialised migration tool. He will fall back on the tools he has already. This will include the old IDE because he knows it and the new IDE because he need to learn it anyway.

He might have (or write) a simple syntax converter.

Then he would export the code from the old IDE to files, convert everything, and import it into the new IDE.

Now the challenge is to get the migrated system running.

The challenge:

- lot's of compiler errors because the converter had to guess in many cases what the right translation could be
- lot's of semantic errors because the converter can't have a semantic understanding of all elements of the source platform. Note: no semantic specifications are available for most platforms.

The developer has to work bottom up. He has to compare the results produced by the code on the target platform with that of the old platform.

The developer can unit test the new code. But not together with the old code—only one platform at a time. It would be helpful if there were a way to run the unit test on both the old and the new platform to see if they produce the same results.

The overall working style in this situation is unsatisfactory compared to the working style in forward engineering (2.2.2).

3 The Goal: Combining the Tools of the Source and the Target Platform

3.1 Precise Statement of the Goal

The goal of this research is to support application migration by finding a (new) way to *integrate* the IDEs of the source and target platforms so that a developer can

- move code from one environment to the other 1. directly and 2. piece-by-piece,
- work on a single hybrid system that consists of original and (increasingly more) migrated code,
- can execute the hybrid system as a whole for understanding and testing.

In short: Enabling of the working style of forward engineering (2.2.2) in application migration.

Migration means a migration between two *development platforms* (1.1) such as Smalltalk or Java. Such a platform consists of a language and associated libraries

A (new) way means that the integration of IDEs isn't straightforward and can't be done in an ideal top-down design approach because IDEs are not designed to be integrated within another context. Although, they expose some openness in the form of APIs for plug-ins to integrate "smaller" tools into them. Modern object-oriented languages provide features like meta-programming; meta-data interchange standards (MOF, XMI) provide concepts for a data model for IDE integration. Combined, there is potential to use the restricted openness of IDEs to integrate them anyway in a *new* way—different from existing approaches to tool integration (3.3).

Integration means that the IDEs will work *together*. During the migration a developer will have a single consistent view of the (hybrid) system consisting of original and migrated code.

IDEs refers to modern IDEs that must provide some features to allow for the integration. Such constraints are listed in 3.2.

Working model means that a developer can work in a process similar to that in forward engineering (2.2.2): unit testing, short iterations, continuous integration.

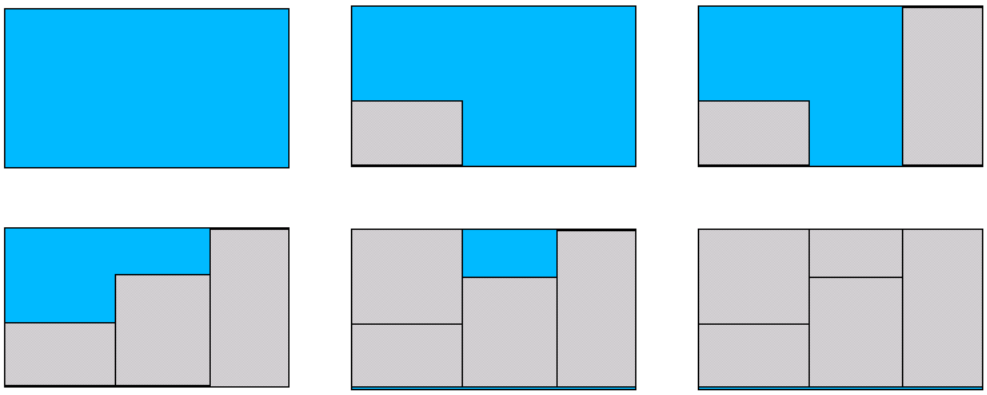

Figure 2: Piecemeal Migration

Let's look at what happens to an application during a migration, see figure 2. First, we start with the application on the old platform using the IDE for that platform. We select a part of the application and migrate this part to the new platform. We find the migrated code in the IDE of the new platform. The key point is that we maintain a single view of the system, consisting of the original parts and the migrated parts. Once a part is migrated, the migrated part *replaces* the original part in that single view. The integration mechanism ensures that the *hybrid* system consisting of code on the old and the new platform is consistent and executable as a whole.

3.2 Constraints

1. The IDEs and their architectures are given. It's not desirable (and usually not possible) to change the architecture of one of the IDEs. Hence the name integration architecture—because the architecture *integrates* IDEs.

2. The architecture should be minimal in the sense that it 1. relies on the tools that are available on both platforms already as much as possible and 2. does not introduce a new tool for the developer to learn (except for a user interface for the configuration of the integration mechanism—and not much more).

One exception is a syntax converter because it can usually not be found in an IDE—although an IDE contains building blocks for a syntax converter such as a parser.

In this work we presume that a (trivial) syntax converter is available. It might me a commercial or an in-house development. The syntax conversion is not a critical element of a migration. It is only the easy part.

3. The integration mechanism will be limited to object-oriented development platforms (in particular, the prototype will be implemented in Smalltalk and Java). These platforms provide the richest set of features in their IDEs and programming languages.

Object-oriented technology has been widely used and of the systems built in industry many are considered legacy systems and are candidates for a migration. This is especially the case for Smalltalk applications.

4. The proposed solution claims some level of generality—at some cost. The goal is not to provide a specific migration tool for a migration between two specific platforms ("points" in figure 1). For example we are aware of a specific rule-based migration tool for migrating VisualWorks Smalltalk code to Java. We argue that any migration tool can't provide a 100% migration and that some "handwork" will always be left. Furthermore we argue that for many platforms specific tools aren't available. Our approach attempts to help in those cases: make it easier to do the remaining—or possibly almost all—"handwork".

Specific tools will be better—if available. If not, our approach attempts to provide a productive migration path built on top the existing IDEs.

Specific tools will be better since we don't address product-specific details. A specific migration tool that addresses such details, e.g. in a rule base for mapping proprietary classes, will produce migrated code that is "closer" to the target platform than we can provide with the IDE integration and some syntax conversion. However, we argue that some handwork remains even when such a specific migration tool is used. It might be less handwork, but not zero.

The level of generality of our approach is one of the contributions of this research (4.5).

5. Standards should be used wherever applicable. This includes standards for meta-data interchange, middle-ware (XML-based protocols), tool integration (debugger interfaces).

6. Other restrictions of our approach are that the IDE integration mechanism does not address

- support for multiple developers,
- versioning of individual artifacts.

3.3 Related Work on Tool Integration

3.3.1 Repositories

Repositories deal with software artifacts. Therefore repositories have meta-models of the artifacts they deal with. The relationship to the integration architecture defined here is that during migration the tools have to deal with similar artifacts. Repositories are a source for such meta-models.

It has to be seen if those meta-models apply in the context of IDE integration because they usually don't deal with runtime information that might be important during a migration.

3.3.2 Meta-Data Interchange

The two IDEs are connected not only at the object-level but also at the meta-object level. Therefore meta-data needs to be exchanged.

Meta-data interchange formats are usually file-based. It has to be seen if they can be used for exchanging the required information between IDEs.

OMG'S Meta-Object Facility (MOF) is an interaction-based interchange mechanisms. This might provide further ideas for the integration of IDEs.

XMI is a mechanism to define meta-data interchange formats. It has to be seen if that can be applied to the (runtime) data required in the context of migration between IDEs.

Whatever interchange mechanism is used, it has to use a meta-model for the artifacts and tools involved. Such meta-models exist for modelling constructs (UML), and for programming languages. For IDE integration—in the sense of CORBA—this is like a vertical market domain.

3.3.3 Remote Debugging

Remote debuggers, e.g. the Java Platform Debugger Architecture (JPDA), define a way how a tool can attach to a running VM. This is quite similar to what the two IDEs do here—they attach to each other. Much work is done in the area today since in web software development the focus is much more on the server. To support this kind of architecture many IDEs support remote debugging of server applications.

It has to be seen if the standards that exist in this area provide useful ideas for the integration of IDEs.

3.3.4 Tool Integration

IDEs are per se *integrated*. The integration is done by the vendor of an IDE. Some vendors open their IDEs by providing a Tool Integrator API (e.g. IBM VisualAge

for Java). These mechanisms are, however, designed to integrate other tools within the enclosing IDE. They are not intended to integrate IDEs with each other.

IBM is attempting to integrate tools at a larger scope in the WebSphere Studio Workbench. This effort goes beyond the integration of development tools found in IDEs in that it includes other tools like HTML editors, modelling tools etc.

Although, these mechanism provide useful input for the design of a solution for tool integration in application migration.

3.4 How IDE Integration Improves Application Migration

The problem identified in section 2 is insufficient tool support for software migration. This research attempts to provide a solution by addressing the following aspects of the problem:

Built on top of strong tools (IDEs): In section 2.1 we argue that specific migration tool are often weak. Our approach builds on strong tools with lot's of useful features for a migration

Developer skills: The developer knows the IDEs already. It it not necessary to learn how to use a new specific migration tool. Our approach helps building an understanding of the whole system by providing an integrated view of original and migrated code.

Sound base for additional tools Specific migration tools can be developed easier on top of the integrated IDEs, e.g. syntax converter can rely on parser of source platform. A type analyser can use the cross-referencing features of the IDEs. Our approach lays a foundation for additional tools to use the available resources in an integrated way.

Testing: In our a approach a developer can work with an executable, though hybrid, system all the time. For understanding and proving hypotheses about the code on the old platform he can write test cases. These test cases can be run on the migrated code later. The integration mechanism ensures that this works. Alternatively the test cases can be migrated to the new platform easily. Having test cases greatly improves the confidence a developer can have in the changes he makes to the code (see [Bec99] for this argument in eXtreme Programming). Similar to the test-first design process in eXtreme Programming a developer can chose to work in a similar test-first migration process. He would first write a test-case to understand the existing code, perform the migration, and use the test-case to make sure that the migrated code produces the same results as the original code.

Incremental Migration: Relying on the incremental compilers of IDEs a developer can work on the migration in small steps (refactoring) without breaking the system for more than a couple of minutes.

Interactivity: The integration mechanism allows a developer to work with a "live" system in the debugger and inspectors. He can analyse and change the running system as in forward engineering *without* restarting the system—and the potential difficulty of putting the system into a specific state.

Consistency: The complete system code *and* state is available for understanding and testing at all times. In a non-integrated scenario a developer would work in one IDE to understand existing code, to analyse the system by running it in that IDE. In a second IDE he would make the system run on the new platform. Our approach allows the developer to work with a single consistent code base and system state.

Generality: Our approach is more general than specific migration tools. It will help in more cases but in less detail.

4 The Approach: An Integration Architecture for Programming Environments

The approach is as follows: To achieve the stated goal—improving tool support for application migration by integrating programming environments— an *integration architecture* is defined. The feasibility of this architecture will be demonstrated by the implementation of a prototype.

The first part of this section presents the design of the integration architecture. It explains the high-level overview and the design rationale. Then follows an example of usage and a description of the prototype.

4.1 High-Level Overview

This section describes the main idea: Two IDEs are connected by some middle-ware so that a developer can work on the whole code base involved in a migration in an integrated way.

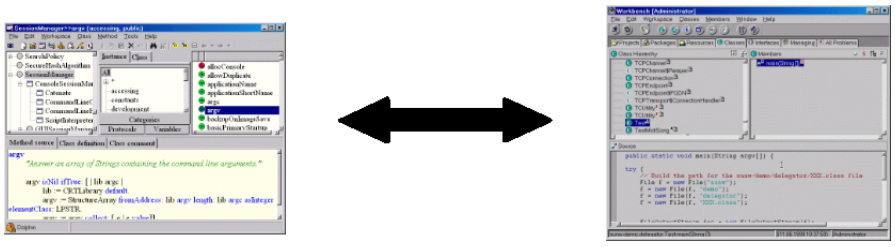

Figure 3: Two programming environments are connected through a messaging middleware

The two IDEs are connected by some (see below) middle-ware. To each IDE a *Connector* is added that acts like an object request broker. This component opens up the functionality and access to the code base for the other IDE. The Connector is added to the IDE by the IDE's extension mechanism (Open API [NetBeans], a Tool Integrator API [VisualAge], or by being white-box (like Smalltalk) or Open Source).

This integration of the IDEs combines the runtime object models of both sides. Code can be sent from one IDE to the other using the middle-ware. The transfer includes a trivial syntax conversion.

Proxies are used when a class or an object is removed from one IDE. These proxies can be added to the code *at runtime* by meta-programming [Zhe97].

Code that uses classes or objects that have been migrated will use the proxies instead. The location of original and migrated code is transparent.

See section 4.3 for an example of usage.

4.2 Design Rationale

The design of the proposed solution is limited by some constraints, see section 3.2.

The goal of this research is to improve the tool support for platform migration. The focus was not to build a specific product-to-product migration tool. However, even when looking at migration in a more general way the choice of platforms has to be limited to set the research goal within reach.

For our approach a platform has to provide a set of features so that the integration mechanism can work: object-orientation, meta-programming.

The IDEs have to provide a set of features too: an incremental compiler that allows the modification of code at runtime. We 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 openness of the IDEs doesn't provide an integration mechanism directly. IDEs provide a mechanism to integrate other tools into them. The idea was to use this mechanism to open the IDEs even more so that a true integration between peer IDE becomes possible. This approach is influenced by the CORBA Meta-Object Facility (MOF). Using MOF software tools can access a repository. The main drawback of MOF is that it requires CORBA with all its inherent complexities. Another influence is the idea to connect tools though message passing [Rei90].

In the context of software migration and IDEs many CORBA services aren't needed. It might also be difficult to get CORBA implementations for specific platforms. Even then, one would need an implementation of MOF in addition.

One of the design goal for the integration architecture is simplicity. Given the constraints of this domain the communication mechanism for the connection of IDEs has to be much simpler.

Another goal is the use of standards. For meta-data interchange XML-based standards are very popular. XML is also used for messaging. Therefore XML as a basis for the data interchange is a natural choice.

Another reason for XML is the availability of XML parser. They are available for virtually any platform. Often an XML parser is already part of an IDE—or it can be added easily.

SOAP is an XML-based communication protocol. It defines an encoding for commonly used data types (very similar in spirit to what CORBA did). SOAP implementations map these encodings to the types of the implementation language. Using SOAP solves the mapping problem of data types between different platforms.

4.3 Example of Usage

This section describes how a developer would use two integrated IDEs, see figure 4.

1. In the beginning both IDEs are started. The application is on the source platform, the target platform is "empty".

The two platforms are connected. Each IDE contains a Connector that connects it to the other IDE. The Connector uses and XML-based protocol to exchange data and meta-data with the the Connector of another IDE.

2. A developer works on a small subset of the application at a time. This "unit of work" could be a use case, or a unit in the sense of what is covered by a unit test.

3. The developer chooses to migrate some classed to the other platform. The Connector of the old IDE sends the source code of these classes to the new IDE where it is received by its Connector. The source text of these classes gets converted to the target language—this is done on either one of the two platforms.

On the source platform the original classes are replaced by proxies. This uses meta-programming. The code is compiled immediately to reflect to changes.

On the target platform the migrated coded is compiled as well. The proxies on the source platform are connected to the migrated code.

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 classed that belong to the current unit of work are handled.

5. Steps 2–4 are repeated for more units of work.

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.

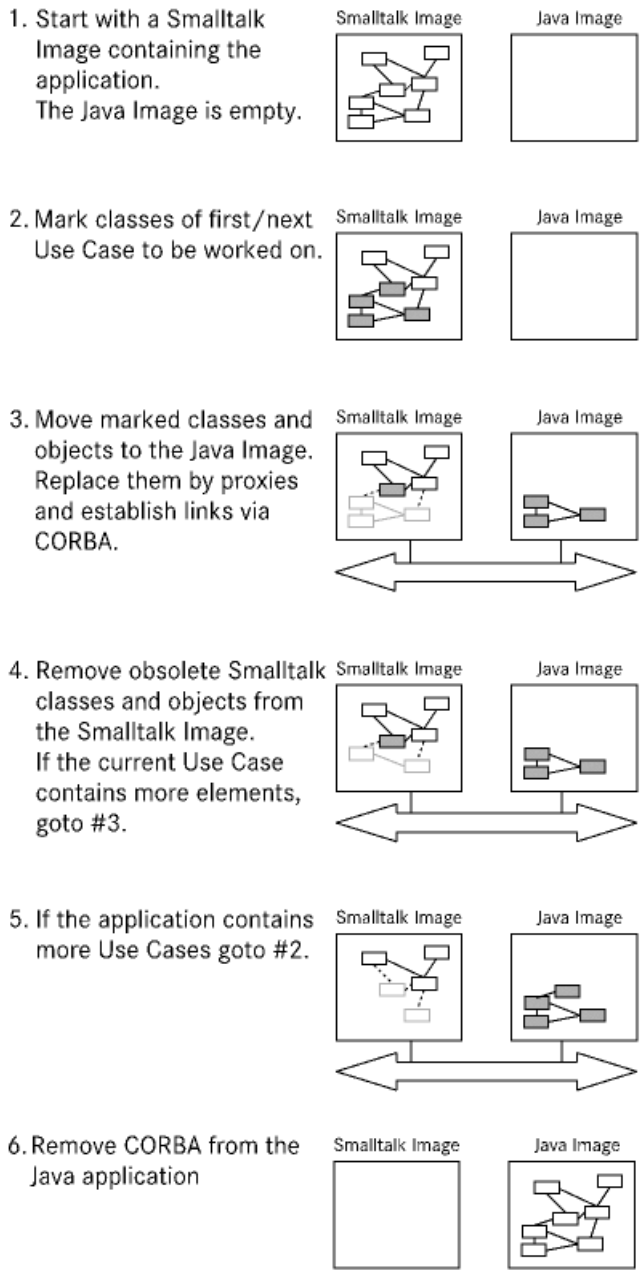

Figure 4: The main steps for a migration

4.4 The Prototype

The prototype is build using Smalltalk and Java. 1. Smalltalk and Java are both candidate platforms for a migration. 2. Both provide the required features to

support the proposed integration.

Figure 5 shows how the two runtime environments are connected through their meta-levels. The Connectors are designed to be "hidden" behind the meta-levels so that they don't interfere with the migrated application.

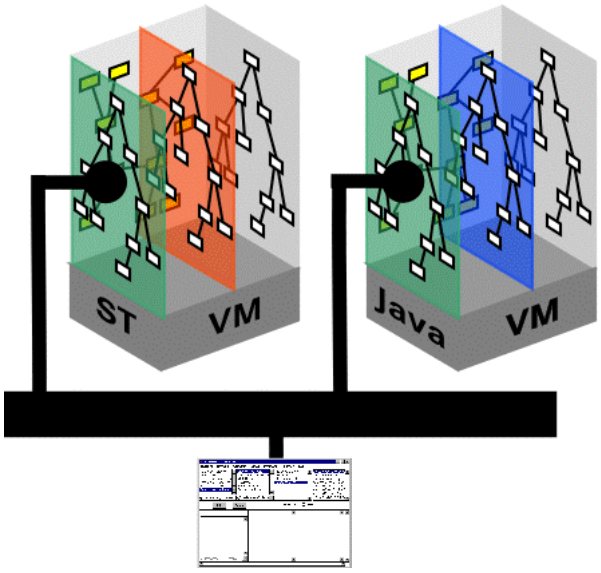

Figure 5: The two VMs are connected via their meta-level

4.5 Measurement of Contribution

There are three criteria to measure the contribution of this work:

1. The time a developer saves by working more productively in the integrated way compated to the non-integrated way.
2. How more easy it is to build tools on top of this architecture by using the functionality that has been made available by the integration.
3. Portability of the solution. How easily can the integration architecture be used for other platforms?

However, it will be hard to quantify the achieved benefits.

References

[Bec99] Kent Beck. *eXtreme Programming explained – embrace change*. Addison–Wesley, Reading, Mass., 1999. 3.4

[BK94] Frederic Bapst and Oliver Krone. A coordination kernel for coupling heterogeneous programming environments, 1994.

[Cou94] J. J. Courant. Softbench message connector: Customising software development tool interactions. *Hewlett-Packard Journal*, 45(3):34–39, 6 1994.

[Fow99] Martin Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley, 1999. 2.2.3

[GLST95] Bob Gautier, Chris Loftus, Edel Sherratt, and Lynda Thomas. Tool integration: experiences and directions. In *International Conference on Software Engineering*, 1995.

[OHT00] Harold Ossher, William Harrison, and Peri Tarr. Software engineering tools and environments: a roadmap. In *International Conference on Software Engineering*, pages 261–277, 2000.

[Rei90] S. P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57–66, 7 1990. 4.2

[Sta99] Victoria Stavridou. Integration in software intensive systems. *The Journal of Systems and Software*, 48(2):91–104, 1999.

[Yan92] Y. Yang. *Tool interfaces for software development*. PhD thesis, Department of Computer Science, The University of Queensland, 1992.

[YY96] Jun Han Yun Yang. Classification of and experimentation on tool interfacing in software development environments. In *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 1996.

[Zhe97] Ling Zheng. Dynamic program instrumentation: Implementation experience in HPUX, 5 1997. 4.1