1-1-2002

CentiJ: An RMI Code Generator

Douglas A. Lyon
Fairfield University, dlyon@fairfield.edu

Peer Reviewed

Published Citation
Douglas A. Lyon,"CentiJ: An RMI Code Generator." Journal of Object Technology 1, no. 5 (2002): 117-148.

JOURNAL OF OBJECT TECHNOLOGY

Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, No. 5, November-December 2002

CentiJ: An RMI Code Generator

Douglas Lyon, Fairfield University, Fairfield, U.S.A.

The Perfect bureaucrat is the man
who manages to make no decisions
and escapes all responsibility.

– Justin Brooks Atkinson

Summary

The *CentiJ* system synthesizes Java source code that funnels invocations through an RMI (Remote Method Invocation) based transport layer for distributed computation. The technique generates *bridge pattern* code (i.e., interfaces and proxies) that automate the creation of virtual proxies for message forwarding.

We examine the tradeoffs between bridge implementations based on manual static delegation, automatic static delegation, and dynamic proxy classes. Advantages of the *CentiJ* technique include improved performance, type safety, transparency, predictability, flexibility and reliability.

We then look at various methods for solving the disambiguation problem that arises when delegates have conflicting method signatures. Disambiguation can be automatic, semi-automatic or manual. *CentiJ* can automatically create a class that alters the interface to the bridge (using the *adapter* pattern).

1 INTRODUCTION

Java technology supports distributed computing on heterogeneous networks via a technology called RMI and CORBA [Lea] [Jennings]. To do this, RMI provides a communication framework for distributed computation in a remote address space. RMI is a de factor standard for communication between systems written in different languages [Slominski]. Sorry to say, use of the RMI technology often requires significant programmer effort and the writing of extra source code. The *CentiJ* project eases this programmer burden.

CentiJ enables the remove invocation of existing, (i.e., *legacy*) code, **without changing it**. We funnel the communications through a *bridge* that is remotely invoked.

CentiJ: An RMI Code Generator, in Journal of Object Technology, vol. 1, no. 5, November-December 2002, pages 117-148.

All the communications are encapsulated in the *CentiJ* generated code so that the *CentiJ* programmer does not need to modify the code (provided it uses the bridge interface).

The contribution of *CentiJ* is that it reuses original implementations, without altering the source code and provides for a means of distributing the computations. *CentiJ* works for programs that were not written with distributed computation in mind.

The RMI Problem

The RMI problem can be broken down into two sub-problems. The first is called the *legacy bridge problem*. The second is called the *virtual proxy synthesizer* problem.

The legacy bridge problem may be stated simply as follows, given a large number of methods in a variety of classes, find a single interface to these methods and an implementing class so that there is a reuse of the implementations in the existing (i.e., legacy code). We are subject to the constraint that we cannot change the existing code. Further we may not even have the existing source code. The legacy bridge problem is solved by building code that implements the *bridge pattern*. The *bridge pattern* consists of an interface, or protocol of communication and an implementation of the communication. Any network layer protocol can use a bridge pattern. Legacy code is often fragile, hard to maintain, difficult to reverse engineer, unchangeable and sometimes poorly designed. Hence the constraint that we can not change the legacy code base. Subject to these constraints, *CentiJ* builds a bridge between new code and the legacy system. Thus providing a solution to the *legacy bridge problem.*

The *virtual proxy problem* is the second sub-problem solved by *CentiJ*. With the virtual proxy, the goal is to method-forward to an existing implementation. *CentiJ* uses inputs from the legacy bridge problem and generates code that can be invoked on a remote address space.

Thus, *CentiJ* solves the RMI problem by gathering the implementations of a set of classes and generating a virtual proxy.

Motivation

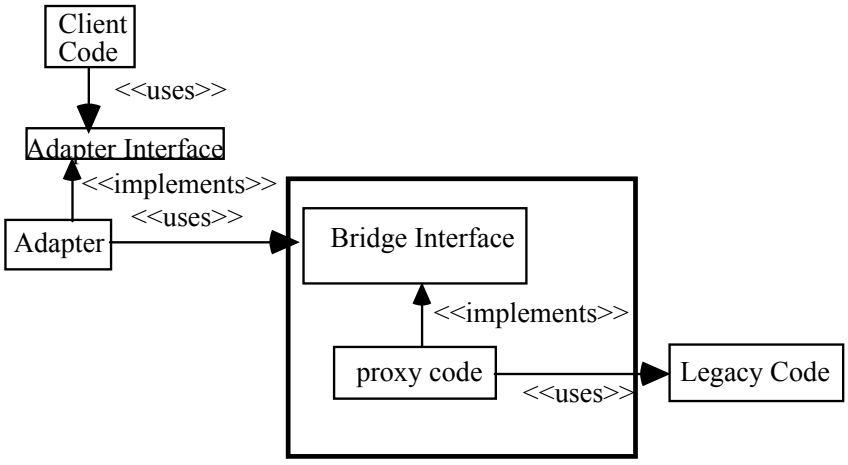
RMI code is typically written manually. This requires an extensive analysis of the existing code. Typically, a large number of dependencies between classes complicates the analysis [Korman]. The bridge pattern serves to regulate the communications between the legacy code and the new code. This regulation prevents changes from propagating to new code by holding the interface constant.

CentiJ generates source code from a collection of classes so that the system has an improved internal structure but identical external behavior (i.e., the code is refactored, but existing source is left unchanged). Refactoring is a key approach for improving object-oriented systems [Tichelaar]. Typical refactoring is performed by altering existing code. However, we are constrained from altering the legacy code and so *CentiJ* uses the bridge pattern as a means to *leave the legacy code unchanged* during refactoring. Some benefits

include: encapsulation of communication, location transparency, low cost of distributed programs and greater reliability in the generated code.

Approach

There are several ways to design a bridge. Figure 1 shows a simple bridge pattern that enables a stable, *bridge interface* to be referenced by an adapter. The adapter is, in turn, used by the client code.



Bridge Pattern

Figure 1. A Simple Bridge Pattern

For example, the *JDBC-ODBC* bridge, enables Java classes to have access to a stable interface for executing structured query language (SQL). The implementation of JDBC-ODBC bridge is often loaded dynamically and plays the role of the *proxy code* in Figure 1. The proxy code implements the bridge interface and delegates to the legacy relational database management system (RDBMS) for its implementation.

CentiJ uses delegation to build the bridge code. The *CentiJ* program generator adds a new *binding time*. There are now three binding times to consider:

1. Generation time: the time source code is output
2. Compile time: when the synthesized code is compiled
3. Run time: when the synthesized code is executed. [Cleaveland]

If we try to combine steps 1, 2 and 3, we loose type safety (an important feature of *CentiJ* code). This is because we cannot know about interface types before the code is

synthesized and compiled. An alternate design (based in *dynamic proxies*) is discussed in Section 2.2.

CentiJ's program generator uses the *reflection* API. The *reflection* API enables running Java programs to discover information about types (i.e., classes and interfaces) at *runtime*. *CentiJ* not only leaves existing legacy code unmodified, the source code is not needed, since all the information needed to generate the RMI code can be obtained via reflection.

Figure 2 shows the RMI architecture retrofitted with the bridge pattern shown in Figure 1.

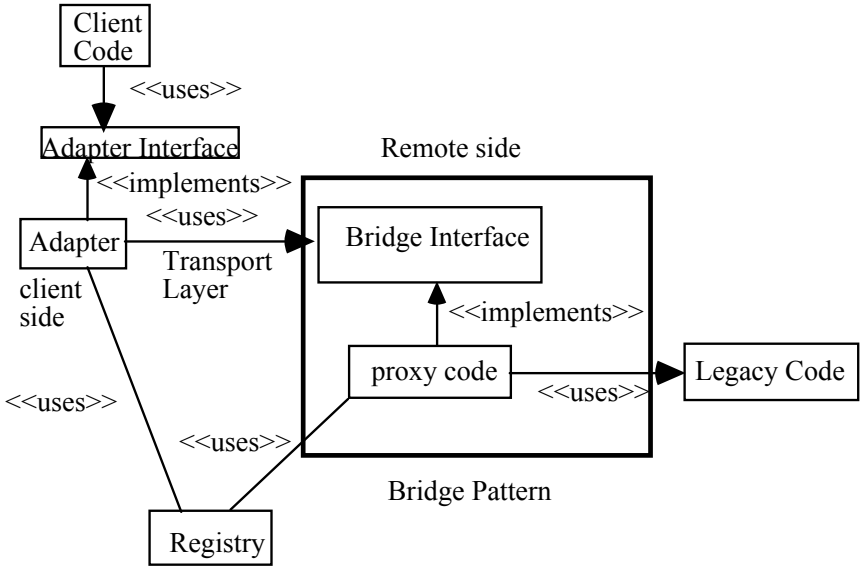


Figure 2. RMI and the Bridge Pattern

Figure 2 shows that the synthesized bridge code uses a *registry* in order to communicate its existence to the *client*. After the process of *discovery* (i.e., looking up the location of the server code) the *client* uses the network *transport* layer to communicate with the server. The registry acts as a broker, mapping reference to actual objects, registering and unregistering services. The RMI protocol supports a native SUN protocol as well as the Internet InterORB Protocol (IIOP) [Sun IIOP]. The client side of the RMI interface is called a *stub*. The server side of the *RMI* interface is called a *skeleton*. Invocation is generally unicast (i.e., point-to-point).

2 VARIOUS BRIDGE IMPLEMENTATIONS

This section examines the various implementations of the bridge pattern. The alternatives are based in *delegation*. We describe the two types of delegation, dynamic and static. Dynamic delegation is delegation that is performed at run-time using dynamic class loading. Static delegation is delegation that is performed at compile time. *CentiJ's* static delegation technique generates Java source code that must be compiled to be used.

We show how dynamic delegation is easy to implement, but also represents a poor software engineering approach. We then examine static delegation as a sound software engineering practice. Finally we examine the two kinds of static delegation, manual and automatic. The automatic technique is a unique contribution of *CentiJ*. We show how automatic bridge synthesis eases the creation of proxy classes and interfaces.

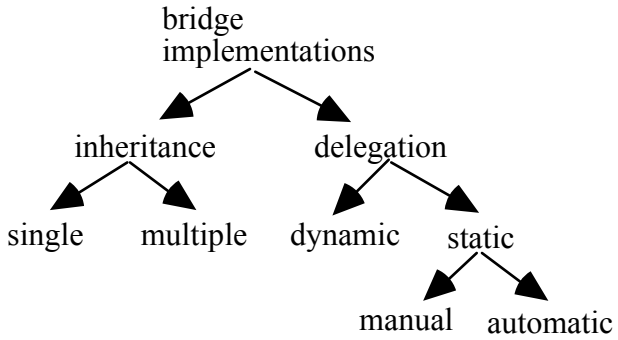


Figure 3. Various Bridge Implementations

Figure 3 shows the various bridge implementations that we will consider. While manual static delegation is the most common, type-safe implementation of a bridge, it is also the most labor intensive. The new mode of *automatic static delegation* alters the economics of static delegation so that it is both type-safe and low-cost. Bridges can also be built using inheritance. Sorry to say, inheritance methods for building bridges are difficult to implement for a single-inheritance type language (like Java). This is because classes that comply with the RMI framework typically subclass the *UnicastRemoteObject* as shown in Appendix A.

What is Delegation?

There is disagreement about what delegation is (and is not). According to one definition, delegation uses a receiving instance that forwards messages (or invocations) to its delegate(s). This is sometimes called a *consultation* [Kniesel]. This is the definition that we use in *CentiJ*.

Variations on delegation give rise to several *design patterns*. For example, if methods are forwarded without change to the interface, then you have an example of the

proxy pattern. If you simplify the interface with a subset of methods to a set of delegates, then you have a *facade pattern*. If you compensate for changes (i.e., deprecations) in the delegates, and keep the client classes seeing the same contract, then you have the *adapter pattern*. If you add responsibilities to the proxy class, then you have the *decorator pattern* [Gamma 1995]. Thus, we define *static delegation* as compile-time, type-safe, message forwarding from a proxy class to some delegate(s).

Compare this to the definition given by Lieberman [Lie 1986]. With Lieberman-delegation (i.e., *dynamic delegation*) the communications pattern is decided at run-time. Thus, compile-time checks are not performed and the message forwarding is not type-safe. In JDK1.3 dynamic delegation is more automatic (i.e., it is Lieberman-style). JDK 1.3 can build a proxy object from the reflection API called a *dynamic proxy class*.

Reflection enables a listing of methods and their signatures. These are used to *forward* invocations to the delegates contained by a *proxy* class. I call this *static proxy delegation*, in order to differentiate it from the *dynamic proxy classes* that have been introduced in JDK 1.3 [Sun 2000].

CentiJ refactors code in a *type-safe* way, without altering it. It is well known that improper refactoring can break subtle properties in a system. As a result, refactoring is generally followed by a testing phase [Katoaka]. By using an automatic code generation technique *CentiJ* reduces the need for extensive testing.

Before *CentiJ,* proxy classes were written manually using *manual static delegation*. In manual static delegation, an instance is passed to a proxy class as a parameter. A programmer writes *wrapper* code that *delegates* to the contained instance. The code that contains the wrapper code is called the *proxy* class. The code that contains the implementation code is called the *delegate*. For example:

```
Final class Movable {
      int x = 0;
      int y = 0;

      public void move(int _x, int _y) {
         x = _x;
         y = _y;
      }
}
```

To add a feature to the *Movable* class we cannot subclass it, because the class is *final*. We might be tempted to modify the *Movable* class, however, source code might not be available. For example, suppose we want a *MovableMammal*: To leave the existing code unchanged, we use manual static delegation:

```
class Mammal {
      public boolean isHairy() {
         return true;
      }
}
```

Our manually written delegation code follows:

```
public class MovableMammal {
      Mammal m;
      Movable mm;

      MovableMammal(Mammal _m, Movable _mm) {
         m = _m;
         mm = _mm;
      }

// this is message forwarding
      public void move(int x, int y) {
         mm.move(x,y);
      }
      public void isHairy() {
         return m.isHairy();
      }
}
```

CentiJ generates proxy classes, like the *MovableMammal* class, *automatically*. It even defines an *interface* to the *MovableMammal* so that the protocol of communication to the *MovableMammal* will always meet a minimum requirement. *CentiJ* generates code (and resolves name collisions) using the reflection API and either topological sorting or a GUI for programmer direction. This enables the automatic generation of bridge interfaces to large numbers of methods in a variety of classes. For example, the following code was generated automatically:

```
interface MammalMovableStub extends
      MammalStub, MovableStub {
}
 interface MammalStub {
      public boolean isHairy();
 }
 interface MovableStub {
      public void move(int v0,int v1);
 }
```

While it is true that the *MovableMammal* is neither *Movable* nor a *Mammal*, it is equally valid to describe the *MovableMammal* as a new reference data type that has all the implementations of its delegates (*Movable* and *Mammal*). This delegation technique is different than one based in specialization, but it is more practical in a single-inheritance language.

It has been asserted that refactoring must be language dependent because it must understand the language of the programs that it is manipulating. This is generally untrue since our system makes use of the reflection API and this API (or its equivalent) could be written for almost any language, in theory [Johnson].

Dynamic Delegation vs. Static Delegation

Delegation adds references to *helper* classes that can process the data, then delegate to the other classes for the implementation. Delegation has long been thought of as a generalization of inheritance (a point of view with which there is disagreement) [Aksit 1991] [Bracha].

Delegation has the disadvantage that:

1. The computational context must be passed to the delegate.
2. There is no straightforward way for the delegate to refer back to the delegating object [Viega].
3. The proxy class is coupled to the delegates.

With JDK 1.3, there is a new technique called *dynamic proxies* [Sun 2000]. Dynamic proxies have all the disadvantages of delegation and:

4. They are harder to understand than more static software.
5. Dynamic delegation is slower than static delegation.
6. The design has a counterintuitive class structure [Korman]
7. Type-safe dynamic delegation is impossible [Kniesel 98].

Point 7 requires some discussion. Dynamic proxies can't be compile-time checked for unresolved messages. In contrast, static delegation does provide compile-time checking of unresolved messages. This is a critical difference. Even inheritance will compile-time check unresolved messages. Thus, in the spectrum of type-safety, we have, in order of most-safe first:

1. Static delegation
2. Inheritance
3. Dynamic proxy classes

Inheritance is less type-safe than static delegation because shadowing is typically allowed, without warning, at compile time. Thus, some unexpected behavior can result. This raises the *problem of disambiguation*.

CentiJ solves the problem of disambiguation by topological sorting or by using a GUI that requires selection from the methods with conflicting signatures. *CentiJ* used to output ambiguous code, allowing the programmer to resolve ambiguities at compile-time. The last technique was found unreliable since the programmer was performing an error-prone activity.

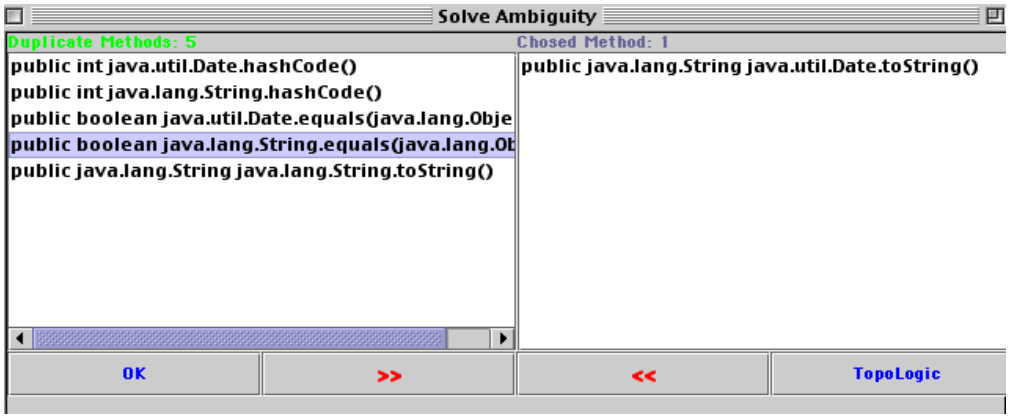


Figure 4. The Disambiguation GUI

Figure 4 shows the GUI presented to the programmer during the disambiguation process. Such a system runs counter to the delegation described by Kniesel. Kniesel has defined delegation as having automatic method forwarding (i.e., Lieberman delegation). We prefer to use the term *dynamic delegation*. The static method forwarding (which Kniesel says is not "true" delegation) is what I define as static delegation [Kniesel 99] [Kniesel 01]. Static delegation is type-safe, dynamic delegation is not. The methods invoked remain the same, but the change in behavior comes from a change in *implementation*.

Stroustrup tried an implementation of dynamic delegation in C++. He reported that every user of the delegation mechanism "suffered serious bugs and confusion". He says that the primary reasons are that functions in the proxy do not override functions in the delegate and functions in the delegate can not get back to the proxy (i.e., the *this* is in a different context). Stroustrup mentions a solution, by manually forwarding a request to another object (i.e., static delegation) [Stro 1994].

The static delegation of *CentiJ* alters behavior in a type-safe way, at run-time, by using polymorphic delegates.

Manual delegation has the disadvantage that:

1. Tedious wrappers need to be written for each method.
2. Manually writing forwarding methods is error-prone.
3. Programmers write arbitrary code in a forwarding method. This can give an object inconsistent interface.
4. Programmers must decide which message subset must be forwarded.

Automatic proxy synthesis overcomes these problems:

1. *CentiJ* does not generate arbitrary code.
2. The interface to the instances remains consistent.
3. The delegation is subject to in-line expansion and is more efficient than multiple inheritance or dynamic proxies
4. The mechanism for forwarding is obvious and easy to understand.
5. The proxy is coupled to the delegate in a more controlled manner than dynamic delegation.

6. Classes that use the proxy are presented with a stable bridge interface.
7. *CentiJ* lowers the cost of software maintenance via automatic code generation.

Problems that remain unsolved by *CentiJ* include:

1. Lack of straightforward way for the delegate to refer back to the delegating object [Viega].
2. The computational context must still be passed to the delegate [Kniesel].
3. The proxy class is fragile. If the interface to the delegate changes, the forwarding method in the proxy must change [Kniesel 1998].
4. A new binding time is needed with static proxy delegation (automatic or manual).

In comparison, dynamic proxy classes generate runtime errors, run slower and need no pre-compilation. We favor compile-time errors over runtime errors, and so find our technique superior in this regard. The trade-off is *pay now* or *pay later*.

Semi-automatic synthesis of delegation code addresses the time-consuming and error-prone drawback of manual delegation. It is also easier to understand. The basic issue is that a balance must be struck between code reuse and the fragility that arises from *coupling*, a measure of component interdependency. This balance is obtained by good object-oriented design (and is hard to automate!). *CentiJ* uses the following Stroustrup-suggested rules for code generation:

1. Ambiguities are illegal.
2. Only public methods are available
3. The methods are all public in the proxy class.
4. Subtyping is done with interfaces, not proxies.
5. Both proxy classes and interfaces are synthesized automatically.
6. Type checking is static.
7. Ambiguity resolution is static (i.e., done at code synthesis time).

CentiJ code, once compiled, never gets messages like “can’t find method” [Wand]. Appendix A and Appendix B show examples of automatically generated *CentiJ* code, and its use.

Summary of findings

We have reviewed different techniques for implementing the bridge pattern. We discussed using language extension to add delegation, language extension to add multiple inheritance, API extension to add delegation and API extension to add manual delegation. Approaches that use language extension fail for pragmatic reasons (lack of compatible tools, slow adoption, slow code, etc.). Approaches that use API extension are easier to deploy, in general, since they work with existing frameworks.

Generally, inheritance enables shared behavior. Some have argued that subtyping (i.e, the multiple-inheritance of interfaces in Java) is not inheritance. In fact, the bridge pattern divorces specification from implementation.

single inheritance	multiple inheritance	manual static delegation	automatic static delegation	dynamic proxies
Subclasses must inherit only a single implementation from a super class.	Subclasses must inherit only a single implementation from a super class.	The computational context must be passed to the delegate.	The computational context must be passed to the delegate.	The computational context must be passed to the delegate.
Inheritance compromises the benefits of encapsulation [Coad].	The topological sorting of the super-classes have been cited as a fruitful source of bugs [Arnold 1996].	There is no straightforward way for the delegate to refer back to the delegating object [Viega].	There is no straightforward way for the delegate to refer back to the delegating object [Viega].	There is no straightforward way for the delegate to refer back to the delegating object [Viega].
Inheritance hierarchy changes are unsafe [Snyder].	Inheritance compromises the benefits of encapsulation [Coad].	The proxy class is coupled to the delegates.	The proxy class is coupled to the delegates.	The proxy class is coupled to the delegates.
Conflicts between multiple parents are not reported. Ambiguity resolution has long been known as a problem with inheritance [Kniesel].	Inheritance hierarchy changes are unsafe [Snyder].	Tedious wrappers need to be written for each method.	The synthesis does not generate arbitrary code.	They are harder to understand than more static software.
	Conflicts between multiple parents are not reported. Ambiguity resolution has long been known as a problem with inheritance [Kniesel].	Manually writing forwarding methods is error-prone.	The interface to the instances remains consistent.	Dynamic delegation is slower than static delegation.
	Taxonomically organized data has become automatically associated with object-oriented programming [Cardelli].	Programmers write arbitrary code in a forwarding method. This can give an object an inconsistent interface.	The delegation is subject to in-line expansion and is more efficient than multiple inheritance.	The design has a counterintuitive class structure [Korman]
	Java has no built in support for multiple inheritance	Programmers must decide which message subset must be forwarded.	The mechanism for forwarding is obvious and easy to understand.	Type-safe dynamic delegation is impossible [Kniesel 98].
	Extensions to the language are generally incompatible with legacy code	The compilation of generated code, is required	Proxy is coupled to the delegate in a more controlled manner than automatic dynamic delegation.	
			Adapters possible	
			Lower the cost maintenance, improved reusability	
			The compilation of generated code, is required	

Figure 5. Summary of trade-offs

Figure 5 shows the trade-off summary for implementing the bridge pattern. A comparison is made between the various kinds of delegation with the various kinds of specialization. There are two kinds of specialization, single-inheritance and multiple-inheritance. There are two basic kinds of delegation, dynamic and static. The dynamic delegation is slow and makes type-safety impossible. The static delegation is fast, and type-safe. There are two kinds of static delegation, manual and automatic. The manual delegation requires programmers write method-forwarding code, a process that is both error-prone and

tedious. The automatic-static delegation has been shown to be an easy-to-deploy technique that generates proxy classes that are both type-safe and easy to understand.

3 RELATED WORK

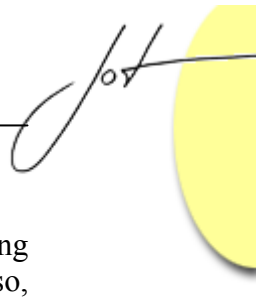
There are several projects that aim at making Java programs parallel. Once example is the *Do!* project [Launay]. The *Do!* project does not use a static refactoring of the code to help with distrubutions instead it uses special kinds of distributed *collections* to explictly express concurrency.

Another tool, *Orca* automated distribution decisions using a run-time system for placement and replication selection for remote jobs [Bal]. The *Ninja* project uses clusters of workstations, active proxies and low-level bytecode specialization for fine-grained parallelism. The *Pangaea* system uses a static source code analysis and a middleware back-end to distribute centralized Java programs. *J-Orchestra* takes the approach of fine-grained automatic parallelism using byte-code output from the Java compiler. *J-Orchestra*, *Do!*, *Orca, Ninja* and *Pangaea* do not attempt to perform any type of refactoring or code generation. Also they try to automate the decision for placing programs on other systems (a decision that is hard to automate). Their fine-grained approach to automating parallelism does not take into account the programmers' input (which often stems from specialized knowledge about the problem domain and code structure). [Tilevich] [Spiegel] [Spiegel 2000] [Gribble].

Means of automating RMI are not new. *JavaParty has* been around for some time (see http://wwwipd.ira.uka.de/JavaParty/tour.html). However, it requires that the language be modified. Further, it does not gather instances to build bridges as *CentiJ* does.

Tools for refactoring code automatically are not new [Opdy92b], [Opdy93a], [John93b] [Casais]. Language independent tools for refactoring code are not new either [Tichelaar]. Even the use of explicit and parametrical bindings to create type-safe inheritance is not new [Hauck]. Manual refactoring has long been recognized as an important component in extreme programming [Deursen]. All these refactoring techniques alter existing code, something *CentiJ* does not do.

Other source-code based tools for automatic refactoring include the Smalltalk Refactoring Browser [Roberts], the IntelliJ Renamer (http://www.intellij.com), which supports renaming of identifiers and the *Xref-Speller* (http://www.xref-tech.com/speller/) which supports set refactorings. The *Xref-Speller* is based in *emacs* macros and serves to perform a cross-reference analysis. This is useful for the renaming feature, (and its ability to generate cross-linked html code). The Daikon invariant detector reads source code and depends on instrumentation of the source code for full function (http://sdg.lcs.mit.edu/~mernst/daikon/). None of the afore mentioned tools automate bridge synthesis. This is also true for the class composition proposed by Harrison and Ossher [Harrison].



Casais has worked on automatic restructuring of the class hierarchy (by altering source code). The idea is that subclasses inherit everything from the superclasses. Also, *CentiJ* interfaces are used to capture information about the type hierarchy, not sub-classes.

Fanta and Rajlich have also worked on altering existing code, by moving functions around, expelling them from classes, refactoring properties and updating invocations to these elements. Moore has also worked on automatic refactoring and method restructuring. This work refactors expressions from methods. The Guru tool of Moore automatically refactors common code out of methods into abstract super-classes. For a programming language that lacks multiple inheritance (like Java) this effort can adversely affect how methods can be shared [Moore]. Casais claims that there may not be any case studies on the automatic reorganization of class hierarchies [Casais]. Thus, the question of how the code quality is changed by these systems remains open.

Our technique for static delegation requires that every instance be passed to a proxy-class, along with its execution context. Thus a programmer's updates in the protocol for communicating the means to pass parameters will have to be updated in the proxy class. This is the solution I took in *Java Digital Signal Processing* [Lyon 1998].

The *CentiJ* approach to automating the synthesis of bridge code is like the pre-processor approach of the *Jamie* system used by [Viega]. A problem with *Jamie* is that it extends the language by creating a macro-preprocessor. Also, *Jamie* uses *dynamic* delegation.

The LAVA language extends Java to provide for delegation. Kniesel says that current implementations of LAVA have an efficiency that is unacceptable [Kniesel 98] [Kniesel 99]. In comparison, *CentiJ* is fast. In fact, with in-lining enabled, there is *no performance degradation*.

Fisher and Mitchell provide a new delegation-based language [Fisher]. The primary advantage of the Fisher-Mitchell system is its ability to infer type, resolving method names at compile-time. Sorry to say, they had to devise a new language for this. In comparison, *CentiJ* works by API extension, rather than by creating a new language. An API extension is easier to deploy into an existing environment than a new language.

Delegation has been cited as a mechanism to obtain implementation inheritance via composition [Lie 1986], [Jz 1991]. Delegation was introduced in a prototype-based object model by Lieberman in 1986 [Lie 1986]. Lieberman indicated that delegation is considered safer than inheritance because it forces the programmer to select which method to use when identical methods are available in two delegate classes. Systems, like *Kiev,* extend the Java language so that it has multiple inheritance of implementation (http://www.forestro.com/kiev/kiev.html). Such language extensions are non-standard and unportable.

Reverse engineering programs, such as *Lackwit*, are able to discover inheritance relationships with greater ease than composition associations [O'Callahan]. That is

because the inheritance association implies a specialization semantic. On the other hand, composition association scales better than signle inheritance.

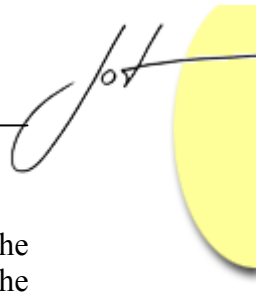
Message forwarding is an implementation sharing mechanism [Kniesel]. Experts have disagreed on this point, saying that delegation is a form of class-inheritance (since the execution context must be passed to the delegate). I take the opposite view, as class-inheritance type of sharing of context involves name sharing, property sharing and method sharing. Sharing via delegation is instance sharing. The semantics of instance sharing enable a control of the coupling between instances. This provides a mechanism for reuse without introducing uncontrolled cohesion (which increases brittleness in the code) [Bardou]. Tim Lavers published a technique for automatically generating RMI source code [Lavers]. It is very close to what *CentiJ* presents except that it does not gather the instances to build a bridge class, and makes use of dynamic proxy invocation. Also, it creates the stubs and skeletons via a dynamic invocation of RMIC (which is generally unreliable, and non-portable, in our experience). The reliabilty issue is raised by making use of platform dependent invocations to the operating system (hard-coding pathnames in the process).

In summary, all the refactoring systems reviewed in this section (except [Lavers]) not only need to read the source code, but they are like the *Elbereth* system in that they alter source code [Korman]. In the literature that we have reviewed, we have yet to find a means for automatically creating the bridges created by *CentiJ*. A macro system (or templates) would be a logical means of providing this ability, but, sorry to say, this would a require a modification of Java.

Methods for automatically generating adapters are not new. In fact, C++ has had a template feature for years [Stroustrup 1991]. Sorry to say, Java has no template feature, and one is needed. In answer to this need Veldhuizen created a Java pre-processor called *Lunar* [Veldhuizen 2000]. The goal of Lunar, however, is to post-process the Java into C, for the purpose of optimization of computation. Volanschi et Al. extend the Java language to implement specialization classes, as did Viroli et Al. [Volanschi] [Viroli]. Meyers et Al. have also proposed extending Java in order to add “generics” in Java (another name for templates). Sun has taken up the task of modifying Java to add generics in a draft version of their compiler. The language feature is said to be the second most often asked for language extension of Java. Sorry to say, the draft and specification are held as proprietary to the Java developer connection, and therefore cannot be disclosed here.

There are compelling arguments against altering the Java language in order to add generics. For one, it will make the language more complicated. Adding some API calls to generate source code is an easy to deploy technique and leaves Java compatible. Since Java is linked in runtime, templates will require code replication (like C++). This is just as easy to do with an API as it is with a pre-processor built into the compiler. Naturally, the generated code would be faster if generics were included as a part of the JVM, but speed was never a design goal of the Java language (as far as I know).

4 CONCLUSIONS



CentiJ does method forwarding across a transport layer in a manner invisible to the programmer. This helps to isolate client code from changes in the interfaces in the delegates. *CentiJ* can use a fully automatic system for resolving method ambiguity (via topological sorting).

Delegation with static binding enables inlining of code. Thus static delegation does not suffer from the performance degradation of dynamic delegation.

In brief:

1. Dynamic delegation is more automatic than static delegation.
2. Dynamic delegation is not type-safe, but static delegation is.
3. Automatic static delegation is almost as automatic as dynamic delegation, and just as type safe as static delegation.

The choice between static and dynamic delegation is a choice between safety and flexibility. [Agesen].

The following are some heuristics for the use of *CentiJ*:

- If polymorphism is needed, then use the automatically generated interface stubs, that *CentiJ* provides.
- If proxies are needed, then use *CentiJ* for generating proxies.
- If source code is unavailable, there may be little other choice.
- If source code is available, refactoring by hand may lead to better code, but may have an effect on a large number of client classes and require testing.
- If many programmers require a stable interface, then use the automatically generated bridge.
- In the case where the contracts shift in the delegates, allow the facade to become an adapter-facade-proxy, in order to protect the clients.

Deepening subclasses in order to add features is a fast way to create poor code that is very fragile. It is a poor way to introduce sub-typing. Only use subclasses if the class theoretic approach is appropriate to the domain, and then only if the taxonomic hierarchy is unlikely to change.

CentiJ decouples proxy delegation from subtyping. Benefits include:

1. An upwardly compatible extension.
2. Realistic performance.
3. A practically useful tool.
4. Inheritance restricted to subtypes only.
5. Name collisions resolved by topological sorting or programmer interaction.
6. No need for access to existing source code

In brief, automatic program generation of proxy classes provides a new way to refactor legacy code and alters the economics of implementation reuse in single inheritance type languages.

5 FUTURE WORK

More work is required to quantify the improvement accomplished by *CentiJ's* refactoring techniques. Metrics to quantify these improvements are elusive.

A next step in automation is the selection of which instance should be remotely invoked. Currently, we need a programmer for this. At present, fully-automatic systems can not overcome the limits of gnoseology in order to improve on the epistemology. A semantic network may help.

Casais claims that there may not be any case studies on the automatic reorganization of class hierarchies [Casais]. Thus, the question of how the code quality is changed by these systems remains open.

When the communication between agents is based on a modified interface we create what has been termed a *contract network protocol* [Davis and Smith]. A *contract network protocol* helps to isolate a system from deprecations in the delegate methods. Sun's repeated introduction of deprecation into its API's has become epidemic. To determine if a *contract network protocol* could help keep these deprecations from propagating to exsiting code is a topic of future research.

Given a registration mechanism (like the RMIRegistry) it should be possible to automate the program generation of the code for distributed computation. Of course a programmer will still be needed to decide where to segment the problem.

Distributed computation on an unreliable network is an open problem. Also difficult is to determine how to dynamically load balance the computations across such a network. We presently have a method for sorting machines by load and returning the least loaded machine from a list. Sorry to say, the method is based on a shell script and only works for Unix machines. This should be replaced by dynamically benchmarking the speed of a loaded JVM.

6 REFERENCES

[Agesen] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. "Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance". In *ECOOP '93 Conference Proceedings*, p. 247-267. Kaiserslautern, Germany, July 1993

[Aksit 1991] Mehmet Aksit, Jan Willem Dijkstra. "Atomic Delegation: Object-oriented transactions", *IEEE Software*, Los Alamitos, CA, IEEE Computer Society. March 1991.pps. 84-92.

[Arnold 1996] Ken Arnold and James Gosling. *The Java Programming Language*, Addison-Wesley, Reading, MA. 1996.

[Arnold 1998] Ken Arnold and James Gosling. *The Java Programming Language*, Second Edition, Addison-Wesley, Reading, MA. 1998.

[Bal] Bal, H.E., et al. "Performance Evaluation of the Orca Shared-Object Systems" ACM Trans. CS, Vol. 16, No. 1 (1998) pps. 1-40.

[Bardou] D. Bardou and C. Dony. "Split Objects: A Disciplined Use of Delegation Within Objects". In Proceedings of OOPSLA'96, Sans Jose, California. Special Issue of ACM SIGPLAN Notices (31)10, pages 122-137, 1996. http://citeseer.nj.nec.com/bardou96split.html

[Bracha] G. Bracha. "The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance". Ph.D. thesis, Department of Comp. Sci., Univ. of Utah, 1992

[Brant] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. "Wrappers to the Rescue". In Proceedings of ECOOP'98, July 1998. http://citeseer.nj.nec.com/189005.html

[Booch 1991] Grady Booch. *Object-Oriented Design*, Benjamin Cummings, Redwood Cits, CA. 1991.

[Cardelli] L. Cardelli, "Semantics of Multiple Inheritance", *Information and Computation,* 76 (1988) 138-164. http://citeseer.nj.nec.com/cardelli88semantics.html

[Casais] E. Casais, "Automatic reorganization of object-oriented hierarchies: a case study", *Object Oriented Systems,* 1 (1994), pp. 95-115

[Cleaveland] J. Craig Cleaveland, *Program Generators with XML and Java*. Prentice Hall, NJ. 2001.

[Coad] Peter Coad and Mark Mayfield. "Java-Inspired Design: Use Composition Rather than Ineritance", *American Programmer*, Jan. 1997, pps. 23-31.

[Compagnoni] Compagnoni, A. B., & Pierce, B. C. 1993 (Aug.). "Multiple Inheritance via Intersection Types". *Tech. rept. ECS-LFCS-93-275. LFCS*, University of Edinburgh. Also available as Catholic University Nijmegen computer science technical report 93-18. http://citeseer.nj.nec.com/compagnoni93multiple.html

[Cox] B. Cox. "Message/Object Programming: An evolutionary change in programming technology", *IEEE Software* (1)1. Jan 1982.

[Davis and Smith] R. Davis, and R.G. Smith, "Negotiation ias a metaphor for distributed problem solving", *Artificial Intelligence,* No. 20, pps. 63-109.

[Deursen] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. "Refactoring test code". In M. Marchesi, editor, Extreme Programming and Flexible Processes; Proc. XP2001, 2001. http://citeseer.nj.nec.com/vandeursen01refactoring.html

[Fanta] Fanta R., Rajlich V., "Reengineering Object-Oriented Code", in Proceedings of the International Conference on Software Maintenance, IEEE Computer

Society Press, Los Alamitos CA, 1998, pp. 238 - 246.
http://citeseer.nj.nec.com/fanta98reengineering.html

[Fisher] K. Fisher and J. C. Mitchell. "A Delegation-based Object Calculus with Subtyping". In Proc. of FCT, volume 965 of Lecture Notes in Computer Science, pages 42--61. Springer-Verlag, 1995. http://citeseer.nj.nec.com/104746.html

[Frank] Ulrich Frank. "Delegation: An Important Concept for the Appropriate Design of Object Models", *Journal of Object Oriented Programming*, June 2000. pps. 13-17,44

[Fraser] Timothy Fraser, Lee Badger, and Mark Feldman. "Hardening COTS Software with Generic Software Wrappers". In IEEE Symposium on Security and Privacy, May 1999. http://citeseer.nj.nec.com/fraser99hardening.html

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns*, Addison-Wesley, Reading, MA. 1995.

[Gribble] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Josheph, R. H. Katz, Z. M. Mao, S. Ross, and B. Zhao. "The Ninja Architecture for Robust Internet-Scale Systems and Services". Special Issue of Computer Networks on Pervasive Computing, 2000. (to appear). http://citeseer.nj.nec.com/gribble00ninja.html

[Har] S. Harbison. *Modula-3*. Prentice Hall, 1992.

[Hauck] F. J. Hauck: "Inheritance modeled with explicit bindings: an approach to typed inheritance"; *Proc. of the Conf. on Object-Oriented Progr. Sys., Lang., and Appl*. -- OOPSLA, (Washington, D.C., Sep. 26-Oct. 1, 1993); SIGPLAN Notices 28(10) , http://citeseer.nj.nec.com/hauck93inheritance.html

[Harrison] William Harrison, Harold Ossher and Peri Tarr, "Using Delegation for Software and Subject Composition", Research Report RC 20946, IBM Thomas J. Watson Research Center, August 1999. http://www.research.ibm.com/sop/soppubs.htm

[Jennings] Jennings, N., and Wooldridge, M. (2000) "Agent-Oriented Software Engineering". In Handbook of Agent Technology (ed. J. Bradshaw) AAAI/MIT Press. (to appear) http://citeseer.nj.nec.com/wooldridge99agentoriented.html

[John93b] Ralph E. Johnson and William F. Opdyke, "Refactoring and Aggregation", Object Technologies for Advanced Software - First JSSST International Symposium, Lecture Notes in Computer Science, Vol. 742, Springer-Verlag, 1993.

[Johnson] Ralph E. Johnson and William F. Opdyke. "Refactoring and aggregation". In S. Nishio and A. Yonezawa, editors, International Symposium on Object Technologies for Advanced Software, pages 264-278, Kanazawa, Japan,

November 1993. JSSST, Springer Verlag, Lecture Notes in Computer Science. http://citeseer.nj.nec.com/johnson93refactoring.html

[Jz 1991] Johnson and Zweig. Delegation in C++, *Journal of Object-Oriented Programming*, 4(11):22-35, November 1991.

[Kataoka] Yoshio Kataoka and Michael D. Ernst and William G. Griswold and David Notkin, "Automated Support for Program Refactoring using Invariants" http://citeseer.nj.nec.com/kataoka01automated.html.

[Kniesel] Günter Kniesel: "Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems". Technical report IAI-TR-94-3, Oct. 1994, University of Bonn, Germany. http://citeseer.nj.nec.com/kniesel95implementation.html

[Kniesel 98] Günter Kniesel: "Delegation for Java: API or Language Extension?". Technical report IAI-TR-98-5, May, 1998, University of Bonn, Germany. http://citeseer.nj.nec.com/kniesel97delegation.html

[Kniesel 99] Günter Kniesel, "Type-Safe Delegation for Run-Time Component Adaptation", In R. Guerraoui (Ed.): Proceedings of ECOOP99. Springer LNCS 1628. http://citeseer.nj.nec.com/kniesel99typesafe.html

[Kniesel 01] Günter Kniesel, private e-mail communications, kniesel@cs.uni-bonn.de.

[Korman] W. Korman and W. G. Griswold. "Elbereth: Tool support for refactoring Java programs". Technical report, University of California, San Diego Department of Computer Science and Engineering, May 1998. http://citeseer.nj.nec.com/korman98elbereth.html

[Lea] Doug Lea, *Concurrent Programming in Java*, *Design Principles and Patterns,* AW. 1997.

[Lie 1986] Henry Leiberman. Using prototypical objects to implement share behavious in object-oriented systms. In *Object-oriented Programming Systems, languages and Applications Conference Proceedings*, Pages 214-223.

[Launay] P. Launay, J.-L. Pazat. "A framework for parallel programming in Java". In HPCN'98, LNCS, April 1998 http://citeseer.nj.nec.com/launay97framework.html

[Lavers] Tim Lavers, "Java Tip 108: Apply RMI autogeneration", http://www.javaworld.com/javaworld/javatips/jw-javatip108.html

[Lyon 1998] Douglas Lyon and Hayagriva Rao. *Java Digital Signal Processing*, M&T Books, NY, NY. 1998.

[Lyon 1999] Douglas Lyon. *Image Processing in Java*, Prentice Hall, M&T Books, NY, NY. 1998.

[Lyon 2002] Douglas Lyon, The DocJava Home Page, http://www.docjava.com.

[Meyers] A. C. Myers, J. A. Bank, and B. Liskov: "Parameterized Types in Java", In: Proc. of 24th POPL, 132-145, 1997

[Moore] I. Moore. "Automatic inheritance hierarchy restructuring and method refactoring". In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications, pages 235-250, October 1996. SIGPLAN Notices, 31(10). http://citeseer.nj.nec.com/moore96automatic.html

[O' Callahan] O'Callahan,R., and Jackson, D., "Lackwit: A program understand ool based on type inference". In Proceedings of the 1997 International Conference on Software Engineering (ICSE'96) (Boston, MA, May 1997), pp. 338--348. http://citeseer.nj.nec.com/329620.html

[Opdy92b] William F. Opdyke, *Refactoring Object-Oriented Frameworks*, Ph.D. dissertation, University of Illinois, 1992. ftp://st.cs.uiuc.edu/pub/papers/refactoring/

[Opdy93a] William F. Opdyke and Ralph E. Johnson, "Creating Abstract Superclasses by Refactoring", Proceedings CSC'93, ACM Press, 1993.

[Postema] Margot Postema and Heinz W. Schmidt, *Reverse Engineering and Abstraction of Legacy Systems*, http://citeseer.nj.nec.com/151140.html

[Roberts] Don Roberts, John Brant, and Ralph Johnson. "A refactoring tool for Smalltalk" *Theory and Practice of Object Systems,* 3(4):253-63, 1997.

[Tichelaar] Sander Tichelaar and Stéphane Ducasse and Serge Demeyer and Oscar Nierstrasz, "A Meta-model for Language-Independent Refactoring", IEEE Proceedings ISPSE, 2000, http://citeseer.nj.nec.com/379788.html

[Snyder] Alan Snyder. "Encapsulation and Inheritance in Object-Oriented Programming Languages", Affiliation Software Technology Laboratory, Hewlett-Packard Laboratories, PO Box 10490, Palo Alto, CA, 94303-0971 http://citeseer.nj.nec.com/328789.html

[Spiegel] Andre Spiegel. Pangaea: An automatic distribution front-end for Java. In Fourth IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '99), in Proc. IPPS/SPDP '99, San Juan, Puerto Rico, USA, April 1999. IEEE. http://citeseer.nj.nec.com/spiegel99pangaea.html

[Spiegel 2000] Andre Spiegel, "Automatic Distribution in Pangaea", CBS 2000, Berlin, April 2000. See also http://www.inf.fu-berlin.de/~spiegel/pangaea/

[Slominski] Aleksander Slominski, M. Govindaraju, D. Gannon and R. Bramley, "SoapRMI C++/Java 1.1: Design and Implementation", pre-print http://citeseer.nj.nec.com/467360.html

[Stro 1987] Bjarne Stroustrup. "Multiple inheritance for C++". In Proceedings of the Spring '87 European Unix Systems User's Group Conference, Helsinki, Finland, May 1987. http://citeseer.nj.nec.com/stroustrup99multiple.html

[Stroustrup 1991] Bjarne Stroustrup. *The C++ programming Language*, Addison-Wesley, Reading, MA. 1991.

[Stro 1994] Bjarne Stroustrup. *The Design and Evolution of C++*, Addison-Wesley, Reading, MA. 1994.

[Sun IIOP] Sun Microsystems, "RMI over IIOP", http://java.sun.com/products/rmi-iiop

[Sun 2000] Tech Tips, "Using dynamic proxies to layer new functionalirty over existing code" May 30, 2000, http://developer.java.sun.com/developer/TechTips/2000/tt0530.html

[Sun 2001] "The JavaBeans Runtime Containment and Services Protocol specification" May 24, 2001, http://java.sun.com/products/javabeans/glasgow/#containment.

[Tempero] Ewan Tempero and Robert Biddle, "Simulating mulitiple inheritance in Java", The Journal of Systems and Software 55(2000) pps. 87-1000, Springer-Verlag.

[Tilevich] Eli Tilevich, "J-Orchestra: Automatic Java Application Partitioning", pre-publication http://citeseer.nj.nec.com/473381.html

[Vega] John Viega and Bill Tutt and Reimer Behrends, "Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages", CS-98-03, Microsoft Corporation, Feb., 1998, http://citeseer.nj.nec.com/3325.html

[Veldhuizen 2000] Todd L. Veldhuizen. "Just When You Thought Your Little Language Was Safe: ``Expression Templates'' in Java", GCSE, pps. 188-202, 2000. http://osl.iu.edu/~tveldhui/papers/2000/gcse00/index.html

[Viroli] M.Viroli and A. Natali, "Parametric Polymorphism in Java: an Approach to Translation Based on Reflective Features" ACM Conference on Object Oriented Programming: System, Languages and Applications, OOPSLA 2000, held in Minneapolis October 15-19, 2000

[Volanschi] Volanschi , E.-N., Consel, C., Muller, G., and Cowan, C. "Declarative specialization of object-oriented programs". In ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '97) (October 1997), pp. 286-300

[Wand] Mitchell Wand. "Type inference for record concatenation and multiple inheritance". In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92--97, Pacific Grove, CA, June 1989. http://citeseer.nj.nec.com/wand89type.html

Appendix A RMI Synthesis

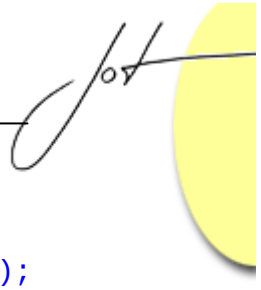
This section shows how RMI code is synthesized by *CentiJ*. The basic idea is that *CentiJ* must provide the bridge code with support for a transport layer protocol for remote invocation.

Before *CentiJ* a programmer "manually" created an adapter that conforms to the requirements of the RMI framework. *CentiJ* provides an automatic alternative for manual adapter synthesis. In RMI a *remote object* is one whose methods can be invoked from another Java Virtual Machine (JVM). Often, this is on a different host. An object of this type is describe by one or more *remote interfaces*. A remote interface subclasses the *java.rmi.Remote* interface. All methods in the subclass must throw a *java.rmi.remoteException* in its throws clause. As a simple server example, consider the *TimeServer*. The interface to the *TimeServer* extends the *java.rmi.Remote* interface, and all the methods throw *java.rmi.RemoteException* instances:

```
public interface TimeServerInterface extends java.rmi.Remote{
      String getTime() throws java.rmi.RemoteException;
}
```

The implementation of the *TimeServerInterface*, called the *TimeServer*, must subclass the *UnicastRemoteObject*. This means that it *cannot* subclass any other classes. This is because Java lacks multiple inheritance. Therefore the *TimeServer* must either provide implementations in its own method bodies or select those implementations via *delegation*.

```
public class TimeServer extends UnicastRemoteObject
      implements TimeServerInterface{

      private String name;

      public TimeServer(String s) throws RemoteException{
         super();
         name=s;
      }

      public String getTime() throws RemoteException{
         Date time=new Date();
         return time.toString();
      }

      public static void main(String args[]){
         System.setSecurityManager(new RMISecurityManager());

         try{
           TimeServer obj=new TimeServer("TimeServer");
            //Create the registry
            // and bind the Server class to the registry
            LocateRegistry.createRegistry(1099);
```



```
            Registry r= LocateRegistry.getRegistry();
            r.bind("TimeServer", obj);
            System.out.println("TimeServer bound in registry");
         }catch(Exception e){
            System.out.println("Error: "+e.getMessage());
            e.printStackTrace();
         }
      }
}
```

Now for the client:

```
import java.rmi.*;
import java.rmi.registry.*;
import java.net.URL;
import java.util.Date;

public class TimeClient {

      String ip = "192.168.1.95";

   public void run(){

         try{
            Registry r = LocateRegistry.getRegistry(ip);
            TimeServerInterface
         obj=(TimeServerInterface)r.lookup("TimeServer");

            System.out.println("remote time="+obj.getTime());
         }catch (Exception e){
            System.out.println("Error: "+e.getMessage());
            e.printStackTrace();
         }
         localTime=(new Date()).toString();

    }

      public static void main(String args[]){
         TimeClient t= new TimeClient();
         t.run();
      }
```

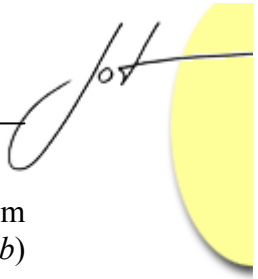
Consider how much extra work it took the programmer to generate the above code. *CentiJ* automates the generation of the above code so that the bridges are adapted to the RMI framework automatically.

For example, the RMI synthesizer created a container of *Student* elements, based on the *Vector* class. A new interface substitutes the *Student* class for the *java.lang.Object*, thus requiring that all elements stored in the *Vector* delegate be of *Student* type:

```
interface StudentVectorstub extends Remote {
      public static final String className = "StudentVector";
      public java.lang.String toString() throws
        RemoteException;
      public void copyInto(Student[] v0) throws
        RemoteException;
      public void trimToSize() throws RemoteException;
      public void ensureCapacity(int v0) throws
        RemoteException;
      public void setSize(int v0) throws RemoteException;
      public int capacity() throws RemoteException;
      public int size() throws RemoteException;
      public boolean isEmpty() throws RemoteException;
      public java.util.Enumeration elements() throws
        RemoteException;
      public boolean contains(Student v0) throws
        RemoteException;
      public int indexOf(Student v0) throws RemoteException;
      public int indexOf(Student v0,int v1) throws
        RemoteException;
      public int lastIndexOf(Student v0) throws
        RemoteException;
      public int lastIndexOf(Student v0,int v1) throws
        RemoteException;
      public Student elementAt(int v0) throws RemoteException;
      public Student firstElement() throws RemoteException;
      public Student lastElement() throws RemoteException;
      public void setElementAt(Student v0,int v1) throws
        RemoteException;
      public void removeElementAt(int v0) throws
        RemoteException;
      public void insertElementAt(Student v0,int v1) throws
        RemoteException;
      public void addElement(Student v0) throws
        RemoteException;
      public boolean removeElement(Student v0) throws
        RemoteException;
      public void removeAllElements() throws RemoteException;
}
```

Based on the pattern of the *Vector* class, the interface was altered at program generation time make the input and output be *Student* elements, rather than *Object* elements (i.e., this is a bridge + an adapter). Sometimes called *parametric polymorphism* [Viroli], altering the interface so that all the methods throw a *RemoteException* we adapt the class to the RMI framework. However, this means extensive wrappering is needed in the delegate. This is easy, and automatic, with *CentiJ*.

In the following example, we reuse the implementation of the *Vector* class to create a type-safe interface. To generate the interface of the adapter, the synthesizer performs a

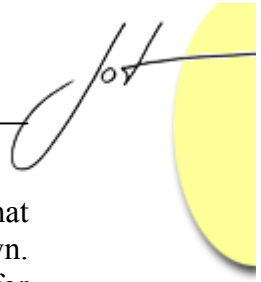


class substitution that searches for references to *java.lang.Object* types and replaces them with *Student* types. The implementation of the *Adapter* interface (i.e., *StudentVectorstub*) follows:

```
// automatically generated by the RMISynthesizer
public   class   StudentVector   extends   UnicastRemoteObject
implements StudentVectorStub {

// constructor:
public StudentVector (){
         try{
            //Create the registry
            // and bind the Server class to the registry
            LocateRegistry.createRegistry(1099);
            Registry r= LocateRegistry.getRegistry();
            r.bind(StudentVectorstub.className, obj);
         }catch(Exception e){
            System.out.println("Error: "+e.getMessage());
            e.printStackTrace();
         }
}

private java.util.Vector vector = new java.util.Vector();
     public java.lang.String toString() throws RemoteException {
          return vector.toString();
     }
     public void copyInto(Student[] v0) throws RemoteException {
          vector.copyInto(v0);
     }
     public void trimToSize() throws RemoteException {
          vector.trimToSize();
     }
     public void ensureCapacity(int v0) throws RemoteException {
          vector.ensureCapacity(v0);
     }
     public void setSize(int v0) throws RemoteException {
          vector.setSize(v0);
     }
     public int capacity() throws RemoteException {
          return vector.capacity();
     }
     public int size() throws RemoteException {
          return vector.size();
     }
     public boolean isEmpty() throws RemoteException {
          return vector.isEmpty();
     }
     public java.util.Enumeration elements() throws
             RemoteException {
          return vector.elements();
```

```
        }
        public boolean contains(Student v0) throws RemoteException {
             return vector.contains(v0);
        }
        public int indexOf(Student v0) throws RemoteException {
             return vector.indexOf(v0);
        }
        public int indexOf(Student v0,int v1) throws RemoteException {
             return vector.indexOf(v0,v1);
        }
        public int lastIndexOf(Student v0) throws RemoteException {
             return vector.lastIndexOf(v0);
        }
        public int lastIndexOf(Student v0,int v1) throws
                RemoteException {
             return vector.lastIndexOf(v0,v1);
        }
        public Student elementAt(int v0) throws RemoteException {
             return (Student ) vector.elementAt(v0);
        }
        public Student firstElement() throws RemoteException {
             return (Student ) vector.firstElement();
        }
        public Student lastElement() throws RemoteException {
             return (Student )vector.lastElement();
        }
        public void setElementAt(Student v0,int v1) throws
                RemoteException {
             vector.setElementAt(v0,v1);
        }
        public void removeElementAt(int v0) throws RemoteException {
             vector.removeElementAt(v0);
        }
        public void insertElementAt(Student v0,int v1) throws
                RemoteException {
             vector.insertElementAt(v0,v1);
        }
        public void addElement(Student v0) throws RemoteException {
             vector.addElement(v0);
        }
        public boolean removeElement(Student v0) throws
                RemoteException {
             return vector.removeElement(v0);
        }
        public void removeAllElements() throws RemoteException {
             vector.removeAllElements();
        }
   }
```

It is a simple matter to obtain the *StudentVectorClient* by passing the stub directly to the client software. However, every method in the proxy throws the *RemoteException*. As a



result, we wrapper the delegation to handle the exceptions locally. The assumption is that after the initial construction there should not be any *RemoteException* instances thrown. This is justifiable only if the internet connection is reliable. This assumption is valid for our networks. Distributed computation on an unreliable network is an open problem.

```
public class StudentVectorProxy {
      private StudentVectorStub vector = null;
      public StudentVectorProxy(String ip) {
         try {
            Registry r = LocateRegistry.getRegistry(ip);
            vector =
               (StudentVectorStub )
                  r.lookup(StudentVectorstub.className);
         } catch(Exception e) {
            System.out.println("Error: "+e.getMessage());
            e.printStackTrace();
         }
      }
      public java.lang.String toString() {
                try {
                 return vector.toString();
                catch (RemoteException e) {
                 return null;
               }
      }
      public void copyInto(Student[] v0) {
               try {
                 vector.copyInto(v0);
               }
               catch (RemoteException e) {
               }
      }
      public void trimToSize() {
               try {
                 vector.trimToSize();
               }
               catch (RemoteException e) {
               }
      }
      public void ensureCapacity(int v0) {
               try {
                 vector.ensureCapacity(v0);
               }
               catch (RemoteException e) {
               }
      }
      public void setSize(int v0) {
               try {
                 vector.setSize(v0);
```

```
            }
            catch (RemoteException e) {
            }
}
public int capacity() {
   try {
              return vector.capacity();
            catch (RemoteException e) {
              return null;
            }
}
public int size() {
   try {
              return vector.size();
            catch (RemoteException e) {
              return null;
            }
}
public boolean isEmpty() {
   try {
              return vector.isEmpty();
            catch (RemoteException e) {
              return null;
            }
}
public java.util.Enumeration elements() {
   try {
              return vector.elements();
            catch (RemoteException e) {
              return null;
            }
}
public boolean contains(Student v0) {
   try {
              return vector.contains(v0);
            catch (RemoteException e) {
              return null;
            }
}
public int indexOf(Student v0) {
   try {
              return vector.indexOf(v0);
            catch (RemoteException e) {
              return null;
            }
}
public int indexOf(Student v0,int v1) {
   try {
              return vector.indexOf(v0,v1);
            catch (RemoteException e) {
              return null;
            }
```

```
}
public int lastIndexOf(Student v0) {
   try {
             return vector.lastIndexOf(v0);
          catch (RemoteException e) {
             return null;
          }
}
public int lastIndexOf(Student v0,int v1) {
   try {
             return vector.lastIndexOf(v0,v1);
          catch (RemoteException e) {
             return null;
          }
}
public Student elementAt(int v0) {
   try {
             return (Student ) vector.elementAt(v0);
          catch (RemoteException e) {
             return null;
          }
}
public Student firstElement() {
   try {
             return (Student ) vector.firstElement();
          catch (RemoteException e) {
             return null;
          }
}
public Student lastElement() {
   try {
             return (Student )vector.lastElement();
          catch (RemoteException e) {
             return null;
          }
}
public void setElementAt(Student v0,int v1) {
          try {
             vector.setElementAt(v0,v1);
          }
          catch (RemoteException e) {
          }
}
public void removeElementAt(int v0) {
          try {
             vector.removeElementAt(v0);
          }
          catch (RemoteException e) {
          }
}
public void insertElementAt(Student v0,int v1) {
```

```
                try {
                  vector.insertElementAt(v0,v1);
                }
                catch (RemoteException e) {
                }
        }
        public void addElement(Student v0) {
                try {
                  vector.addElement(v0);
                }
                catch (RemoteException e) {
                }
        }
        public boolean removeElement(Student v0) {
           try {
                  return vector.removeElement(v0);
                catch (RemoteException e) {
                  return null;
                }
        }
        public void removeAllElements() {
                try {
                  vector.removeAllElements();
                }
                catch (RemoteException e) {
                }
        }

}
```

Appendix B Example of program generator invocation

In this section we describe an example of the *Proxy* class that is generated by the *DelegateSynthesizer* and the *ReflectUtil* class. The effect is to alter the interface to the delegates so that it is simpler to use, without having to change any of the existing code. For example, in order to use the *ReflectUtil* and the *DelegateSynthesizer* in the past, we would write:

```
public static void main(String args[]) {
   DelegateSynthesizer ds = new DelegateSynthesizer();
   ReflectUtil ru = new ReflectUtil(ds);
   ds.add(new java.util.Vector());
   ds.process();
   System.out.println(
      ds.getClassString());
}
```

Now we write:

```
public static void main(String args[]) {
   Proxy p = new Proxy();
   p.add(new Vector());
   p.process();
   System.out.println(
      p.getClassString());
}
```

The *Proxy* class contains all the methods of the *ReflectUtil* class and the *DelegateSynthsizer* class, with a different constructor than either of the two delegates. The constructor was coded by hand, and the class was renamed. Other than that, the code output by:

```
public static void main(String args[]) {
   DelegateSynthesizer ds = new DelegateSynthesizer();
   ReflectUtil ru = new ReflectUtil(ds);
   ds.add(ds);
   ds.add(ru);
   ds.process();
   System.out.println(
      ds.getClassString());
}
```

was all that was required to construct the *Proxy* class. We are now able to get an automatically generated interface, called the *ProxyStub* by executing:

```
public static void main(String args[]) {
```

```
        Proxy p = new Proxy();
        p.add(p);
        p.process();
        System.out.println(p.getInterfaces());
    }
```

This enables us to obtain the multiple-inheritance of typing that we would otherwise have missed if we used only delegation. The *Proxy* class can now implement the *ProxyStub*. In fact, the methods of any number of instances can be folded into a synthesized interface. After the code has been generated, the RMI Compiler (RMIC) is invoked on the synthesized code. Afterwards the class files are bundled into Java archives (JAR files) which are distributed to various servers.

About the author

After receiving his Ph.D. from Rensselaer Polytechnic Institute, **Dr. Lyon** worked at AT&T Bell Laboratories. He has also worked for the Jet Propulsion Laboratory at the California Institute of Technology. He is currently the Chairman of the Computer Engineering Department at Fairfield University, a senior member of the IEEE and President of DocJava, Inc., a small consulting firm in Connecticut.