Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 240–264
The Journal of Systems and Software
www.elsevier.com/locate/jss

A case study in re-engineering to enforce architectural control flow and data sharing ☆

Marwan Abi-Antoun [a,*], Jonathan Aldrich [a], Wesley Coelho [b]

[a] *Institute for Software Research Intl (ISRI), Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, United States*
[b] *Department of Computer Science, University of British Columbia, Vancouver, BC, Canada V6T 1Z4*

Received 17 January 2006; received in revised form 5 October 2006; accepted 14 October 2006
Available online 30 November 2006

Abstract

Without rigorous software development and maintenance, software tends to lose its original architectural structure and become difficult to understand and modify. ArchJava, a recently proposed programming language which embeds a component-and-connector architectural specification within Java implementation code, offers the promise of preventing the loss of architectural structure. AliasJava, which can be used in conjunction with ArchJava, is an annotation system that extends Java to express how data is confined within, passed among, or shared between components and objects in a software system.

We describe a case study in which we incrementally re-engineer an existing Java implementation to obtain an implementation which enforces the architectural control flow and data sharing. Building on results from similar case studies, we chose an application consisting of over 16,000 source lines of Java code and over 90 classes. We describe our process, the detailed steps involved (some of which can be automated), as well as some lessons learned and perceived limitations with the languages, techniques and tools we used.
© 2006 Elsevier Inc. All rights reserved.

Keywords: AliasJava; ArchJava; Ownership domains; Re-engineering

1. Introduction

The architecture of a software system is commonly described in documentation artifacts produced and maintained independently from source code implementations. Studies have shown that developers rarely consult external documentation (LaToza et al., 2006) before making code changes. Over time, the implemented system's design drifts from its original architecture. Eventually, the architectural specification becomes too inaccurate to be used, leading to further degradation of the system structure.

Missing or un-enforced architectural information is a key factor which contributes to architectural problems including architectural drift (Jaktman et al., 1999), i.e., "a lack of coherence and clarity of form which may lead to architectural violation and increased inadaptability of the architecture" (Perry and Wolf, 1992) and architectural erosion, i.e., "violations in the architecture that lead to increased system problems and brittleness" (Perry and Wolf, 1992).

Re-engineering, i.e., "the examination and alteration of a software system to reconstitute it in a new form and the subsequent implementation of the new form" (Chikofsky and Cross, 1990), is one way to correct architectural drift and erosion. Re-engineering a system can extend its lifetime and delay the introduction of a new system built from scratch, thus resulting in cost savings.

In this paper, we describe a case study in-the-small of a new kind of re-engineering to improve and to assure system structure. The assurance guarantee is provided by ArchJava

☆ This is a revised and expanded version of the following paper: Abi-Antoun, M. and Coelho, W. 2005. A case study in incremental architecture-based re-engineering of a legacy application. In: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA), Pittsburgh, Pennsylvania, pp. 159–168.
* Corresponding author. Tel.: +1 412 268 1964.
E-mail address: mabianto@cs.cmu.edu (M. Abi-Antoun).

and AliasJava, two programming language extensions that can enforce architectural control flow and data sharing at the implementation level. ArchJava, a recently proposed programming language, embeds a component-and-connector runtime architectural specification within Java implementation code. AliasJava, which can be used in conjunction with ArchJava, is an annotation system that extends Java to express how data is confined within, passed among, or shared between components and objects in a software system. Using the re-engineering paradigm of abstraction, transformation and re-implementation (Jacobson and Lindström, 1991), we extract the architectural intent in the form of a target architecture and reconstitute the implementation in a new form which makes explicit the control flow and data sharing architecture in code in the belief this will limit future architectural drift and erosion.

This paper makes several contributions. We refine some of the principles and build on results from similar case studies (Aldrich et al., 2002a,b,c) using a subject system that exhibits the following characteristics: (a) it was developed and maintained by several different programmers over the course of several years unlike previous case studies where the subject system was developed and maintained by a single developer (e.g., the Aphyds and Taprats systems in Aldrich et al., 2002b); (b) it is a realistic code base developed and maintained by novice programmers unlike previous subject systems (e.g., the Taprats system in Aldrich et al., 2002b) intended for an object-oriented design competition; (c) it is larger in source lines of code than the similar case studies that have been previously attempted; and (d) unlike previous case studies which have reasoned about communication through control flow (e.g., Aldrich et al., 2002a,b) or data sharing (e.g., Aldrich et al., 2002c; Haechler et al., 2005), we reason about both in the same implementation. To the best of our knowledge, it is the largest case study to date that evaluated an ownership type system on a real object-oriented implementation (another one is described in Haechler et al. (2005)). Finally, as noted in Stevens et al. (1998), re-engineering expertise is lacking; we hope that by documenting the difficulties likely to be encountered and the lessons we learned, we can provide insight into how this activity can be better supported by future languages, techniques and tools.

The paper is organized as follows. Section 2 introduces the subject system. Section 3 discusses the goals of the case study. Section 4 discusses the re-engineering activity in detail and attempts to generalize from our experience, concluding with a discussion of the effort involved and some limitations of this case study. Section 5 discusses some related work. Finally, Section 6 discusses some lessons learned as well as some perceived limitations of the tools, techniques and languages we used.

2. The case study application

The subject system in our case study, HillClimber, is part of CISpace, a collection of Java applications that graphically demonstrate artificial intelligence algorithms. CIspace applications are used as educational tools in undergraduate artificial intelligence courses at several universities and are created and maintained by undergraduate student interns during summer terms at the University of British Columbia (UBC). For several years, new students have contributed to the applications using only the source code as documentation of the systems' design. Predictably, some developers made modifications that were not consistent with the original architecture of the system. These applications provide an example of how the loss of

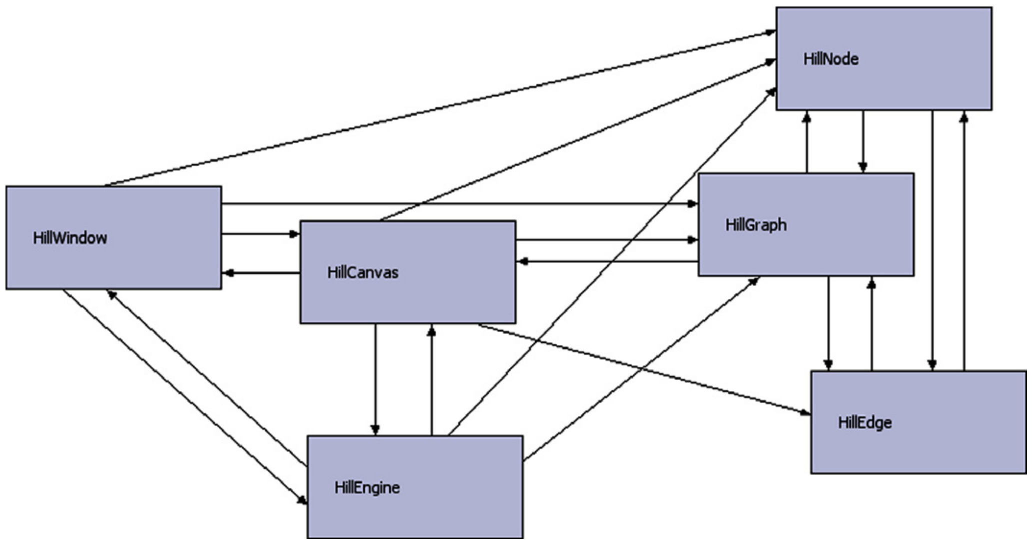

Fig. 1. Module view generated by the LDM tool showing the dependencies between the main classes in the HillClimber application. The boxes correspond to Java classes. The LDM tool uses a standard notion of dependency: class A depends on class B if there are explicit references in A to syntactic elements of B and is shown as an arrow from A to B.

architectural information progressively leads to the degradation of program structure.

The case study subject system, HillClimber, demonstrates stochastic local search algorithms for constraint satisfaction problems. Though HillClimber is relatively small, it contains complex design issues representative of object-oriented applications. In the original design of HillClimber, the application *window* uses a *canvas* to display *nodes* and *edges* of a *graph* in order to demonstrate the algorithms provided by the *engine*.

This simple structural intent was not documented but existed implicitly in the source code. Over time, this intent was damaged by modifications performed by developers who were unaware of it and therefore unable to preserve it. Fig. 1 illustrates the structure after several years of modification by new developers: it shows how communication between components now follows arbitrary paths with little structure. This results in source code that is more difficult to understand and modify, less reusable, more complex and more error prone, all characteristics of architectural drift and erosion (Jaktman et al., 1999).

3. Architecture-based re-engineering

Documenting the software architecture is not enough if important architectural information is lost when moving from design to implementation and later on during evolution. There are several factors that contribute to disparities between architectural specification and the actual implementation. During software development and evolution, developers often do not consult external architecture documentation, even if it exists and is reasonably up-to-date. New developers may also make changes that violate the architecture because they are unaware of the underlying architectural intent.

3.1. Enforcing architectural structure in code

Programming languages that can describe architecture at the implementation level offer a promising solution to these problems. Specifying the architecture in source code has several advantages: first, there is no need to maintain a separate artifact. Second, developers will be more aware of the architecture because it is explicitly documented within the source and not in external documents. Third, the runtime architecture is enforced statically using a type system that enforces the communication integrity property (Luckham and Vera, 1995; Moriconi et al., 1995), which means that two components in the implementation may communicate at runtime only if they are connected in the architecture. Finally, the soundness of the type system guarantees that violations of the architecture do not exist.

Enforcing communication integrity is challenging in programming languages that support references, objects and first-class functions. Previous systems have made serious compromises in order to enforce communication integrity. Some eliminate implicit communication mechanisms entirely (Chapman, 2001; ITU-T, 1999), an approach which works well for static systems such as embedded control circuits, but which is inapplicable to more dynamic systems and is unlikely to be accepted by practitioners used to the flexibility of object-oriented languages like Java. Others postpone conformance checks until runtime (Madhav, 1996) when architectural violations may cause user-visible faults. Finally, some support only simple architectural models (Lam and Rinard, 2003; Murphy et al., 2001) such as module views, giving up many of the benefits of architectural analysis.

ArchJava. ArchJava (Aldrich et al., 2002a) is a recently proposed implementation language that embeds a Component-and-Connector (C&C) architectural specification within Java implementation code. ArchJava extends the Java language with component classes which describe component types that are part of an architecture, component instances, connections which allow components to communicate, ports which are the endpoints of connections, as well as other standard features of architecture description languages such as required and provided methods on ports. The ArchJava type system statically enforces communication integrity, i.e., that communication among components is consistent with the explicit architecture. Without additional annotations, ArchJava only enforces the control flow architecture, i.e., communication through method calls. Furthermore, the aliasing and ownership annotations of AliasJava, a language described later, can be added to an ArchJava program to enforce in addition data sharing constraints, thus enforcing full communication integrity.

We now discuss how ArchJava compares to existing approaches.

Architecture Description Languages. A number of architecture description languages (ADLs) have been defined to describe, model, check and implement software architectures (Medvidovic and Taylor, 2000). The C2 system provides a framework for implementing software architectures but does not automatically ensure that the code instantiating the framework respects architectural constraints (Medvidovic et al., 1996). The SADL system (Moriconi et al., 1995) formalizes architectures in terms of theories, providing a framework for proving that communication integrity is maintained when refining an abstract architecture into a concrete one, however it does not provide automated support for enforcing communication integrity. The Rapide system includes a tool that dynamically monitors the execution of a program, checking for communication integrity violations (Madhav, 1996). The Rapide papers also suggest that integrity could be enforced statically if system implementers follow style guidelines, such as never sharing mutable data between components (Luckham and Vera, 1995). However, guidelines which include forbidding shared data prohibit many useful programs and are not enforced automatically.

Module Systems. Module systems and module interconnection languages (MILs) support system composition from separate modules (Prieto-Diaz and Neighbors, 1986). ArchJava differs from module systems in that the former make data and control flow explicit through architectural connections, while the latter use import and export connections primarily to make names and types defined in one module visible to client modules (Medvidovic and Taylor, 2000). Advanced module systems have rich facilities for defining, manipulating and controlling access to types. These facilities support encapsulation, for example by restricting the definition of a type or a function name to within a single module. However, module systems do not provide mechanisms for controlling shared objects or functions, and thus do not enforce architectural conformance.

Enforcing Design. Lam and Rinard's (2003) system is similar to ArchJava since it also involves a type system for describing and enforcing design; however, their designs describe communication between subsystems (corresponding to ArchJava's components) that is mediated through shared objects that are labelled with tokens. Lam and Rinard's system does not model architectural hierarchy, and the set of subsystems and tokens is statically fixed rather than dynamically determined as in ArchJava; furthermore, their system does not describe data sharing as precisely. Similarly, the Reflexion Models system (Murphy et al., 2001) supports design structure using an analysis in order to find inconsistencies between an architectural model and source code. The Reflexion Models analysis based on call graph construction is more lightweight than ArchJava's type system but does not support hierarchical, instance-based runtime architectures or precise data sharing constraints.

CASE Tools. Several Computer-Aided Software Engineering (CASE) tools support the SDL language (ITU-T, 1999) which allows developers to describe architectural structure within the implementation of an embedded system. The language enforces architectural conformance but only by prohibiting shared references between components. The SPARK system takes a similar approach, supporting a subset of Ada without references in order to rigorously guarantee information flow properties (Chapman, 2001). The prohibition of references is reasonable and even desirable for the telecommunications and other embedded systems for which SDL and SPARK were designed but is inappropriate for the dynamic object-oriented applications that ArchJava can handle. Unlike these CASE tools, ArchJava can enforce architectural conformance in the presence of shared objects and references.

3.2. Re-engineering

Re-engineering is rebuilding a software system or component to suit some new purpose.

New Purpose. In our case study, we reconstitute the subject system in a new form, one that encodes architectural design information in the source code using the language features of ArchJava and AliasJava. We are in fact switching to another language (although one that is close to Java) and the explicit new purpose is better maintainability by precisely and explicitly expressing and enforcing the control flow and the data sharing architecture of the system.

The goal is to see the effect of re-engineering with a new language as a target, and with design information explicitly encoded in the source text. The soundness of the ArchJava and AliasJava type systems together with an effective change management policy can enforce the architecture and prevent the loss of architectural structure in the re-engineered program more easily than had the program been left as a regular Java program. The communication integrity property requires explicitly declaring any new control flow or data sharing communication as they are added to the implementation. Code reviews or inspections assisted with tool support (e.g., Abi-Antoun et al., 2006) could check that the changes are allowed with respect to the system's specified runtime architecture.

Architecture-based re-engineering. Using the re-engineering paradigm of abstraction, transformation (or reasoning about changes at a higher abstraction level) and re-implementation (Jacobson and Lindström, 1991), we extract the architectural intent in the form of a target architecture and reconstitute the implementation in a form that we believe will limit future architectural drift. Using architectural information as the higher-level information to reason about the existing code as well as the target code qualifies the activity as re-engineering. In addition, the abstraction and modification steps help avoid ending up with the same eroded architecture that's well enforced.

There are multiple architectural views of a given system (Clements et al., 2003). For example, a module view showing the static source code organization is useful for allocating tasks to developers but does not directly bear on the runtime behavior of the system. On the other hand, a runtime architectural view described using an Architecture Description Language (ADL) (Medvidovic and Taylor, 2000) can be useful for predicting the system's behavior and other quality attributes such as performance and reliability. A runtime architectural view shows a system in terms of its components (e.g., Client, Server, Database), connectors (e.g., Database Write connector) and their interactions. Runtime views help with understanding the computational model for the system, i.e., how data and control flow through the running system, the protocols through which components interact and runtime dependencies between otherwise independent components. For the subject system, the quality we are concerned with improving is the architectural runtime structure.

Since architecture is determined during design of a system, one might expect that this is the ideal time or even the only possible time to encode these high-level design decisions. However, backward compatible programming languages that enforce architectural structure can be used on an existing implementation in order to recover, formalize, and enforce the architecture. Backwards compatibility between the destination (ArchJava) and the source implementation language (Java) in this case allows an incremental re-engineering approach by transforming the system in small increments and always have a running version, as in taking "chicken little steps" (Brodie and Stonebraker, 1995) to avoid the complexity and the risks of big bang re-engineering. Furthermore, ArchJava only enforces the constraints that are explicitly specified: a Java program is a legal ArchJava program, without the benefit of having the ArchJava type system enforce any communication integrity. So, judiciously used, ArchJava can help make a program easier to understand by making its architecture explicit.

3.3. ArchJava

We now illustrate ArchJava with an example. Fig. 2 shows the architecture of a simple graphics pipeline. The `generate` component stores the current scene and generates shapes to be displayed. These shapes are passed on to the `transform` component which stores the current transformation, applies it to each shape in turn and then passes them on to the `rasterize` component to be displayed. ArchJava can enforce an architectural invariant of the pipeline architectural style (Garlan and Shaw, 1993) that the components are arranged in a linear sequence with each component getting information from its predecessor and sending it on to its successor.

Components. Component types are defined in ArchJava using component classes. A component instance of a component type in ArchJava is a special kind of object instantiated using the `new` keyword, that unlike an ordinary Java object, has its communication patterns declared explicitly in code. Fig. 3 shows the code that defines the `GraphicsPipeline` and `Transform` component classes. We assume that `Generate` and `Rasterize` are component classes defined elsewhere, and `Trans3D` and `Shape` are ordinary Java classes that are not architecturally relevant. The `GraphicsPipeline` class contains three fields, one for each component in the pipeline.

Ports. Components communicate through explicitly declared ports. A port is a communication endpoint declared by a component. For example, the `Transform` component class declares an `in` port that receives incoming shapes and an `out` port that passes transformed shapes on to the next component. Each port declares a set of required and provided methods. A *provided* method is implemented by a component and is available to be called by other components connected to the component through one of its ports. Conversely, each *required* method is provided by some other component connected to a port. For example,

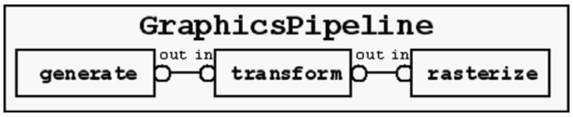

Fig. 2. The `GraphicsPipeline` is a pipeline made up of three subcomponents: the `generate`, the `transform` and the `rasterize` subcomponents.

```
public component class GraphicsPipeline {
  protected /* owned */ Generate generate = ... ;
  protected /* owned */ Transform transform = ... ;
  protected /* owned */ Rasterize rasterize = ... ;

  connect pattern Generate.out, Transform.in;
  connect pattern Transform.out, Rasterize.in;

  public GraphicsPipeline() {
    connect(generate.out, transform.in);
    connect(transform.out, rasterize.in);
  }
}
public component class Transform {
  protected /* owned */ Trans3D currentTransform;

  public port in {
    provides void draw(/* unique */ Shape s);
  }
  public port out {
    requires void draw(/* unique */ Shape s);
  }
  void draw(/* unique */ Shape s) {
    currentTransform.apply(s);
    out.draw(s);
  }
}
```

Fig. 3. ArchJava for the `GraphicsPipeline` and `Transform` component classes. We assume that `Generate` and `Rasterize` are component classes defined elsewhere, and `Trans3D` and `Shape` are ordinary Java classes that are not architecturally relevant. The `GraphicsPipeline` class contains three fields, one for each component in the pipeline. AliasJava annotations which will be added later are shown as comments, e.g., `/* owned */` or `/* unique */`.

the `draw` method's implementation transforms its shape argument and then calls the required method `draw` on the `out` port. As the example shows, a component can invoke one of its required methods by sending a message to the port that defines the required method.

Connections. In ArchJava, developers declare the connection patterns that are permitted in the architecture at runtime. The declaration `connect pattern Generate.out, Transform.in` permits the graphics pipeline component to make connections between the `out` port of its `generate` subcomponents[1] and the `in` port of its `transform` subcomponents. The connect patterns declared in `GraphicsPipeline` constrain its subcomponents to communicate in a linear sequence, fulfilling the constraint of the pipeline architectural style. Once connect patterns have been declared, concrete connections (using the `connect` keyword) can be made between components. For example, the constructor for `GraphicsPipeline` connects the `out` port of the `transform` component instance to the `in` port of the `rasterize` component

[1] Note: the term subcomponent indicates composition, whereas the term component subclass would indicate inheritance.

instance. This connection binds the required method draw in the out port of transform to a provided method with the same name and signature in the in port of rasterize (not shown). Thus, when transform invokes draw on its out port, the corresponding implementation in rasterize will be invoked.

3.4. AliasJava

One of the major challenges in enforcing software architecture is the sharing of data between components in an architecture. This sharing is not explicit in object-oriented languages but instead is implicit in the structure of references created at runtime. ArchJava uses the AliasJava system (Aldrich et al., 2002c; Aldrich and Chambers, 2004) to make this sharing structure explicit, thereby allowing architects to constrain communication through shared data and ensuring that the implementation conforms to those sharing patterns.

AliasJava is an annotation system that extends Java to express how data is confined within, passed among, or shared between components and objects in a software system. Developers can express controlled aliasing through ownership domains and the lack of aliasing through uniqueness using annotations on reference types. These annotations can be used jointly with or separately from the ArchJava features discussed earlier.

Ownership Domains. AliasJava controls aliasing relationships in object-oriented programs by dividing objects into conceptual groups called *ownership domains* and allowing architects to specify high-level policies that govern references between ownership domains. AliasJava supports abstract reasoning about data sharing by assigning each object in the system to a single ownership domain. There is a top-level ownership domain denoted by the keyword shared. In addition, each object can declare one or more domains to hold its internal objects, thus supporting hierarchical specifications. Fig. 4 uses a Sequence abstract data type to illustrate the ownership model used in AliasJava. The Sequence object is part of a top-level owner ownership domain. Within a Sequence object, the iters ownership domain is used to hold iterator objects that clients use to traverse the sequence, and the owned ownership domain is used to hold the Cons elements in the linked list that is used to represent the sequence.

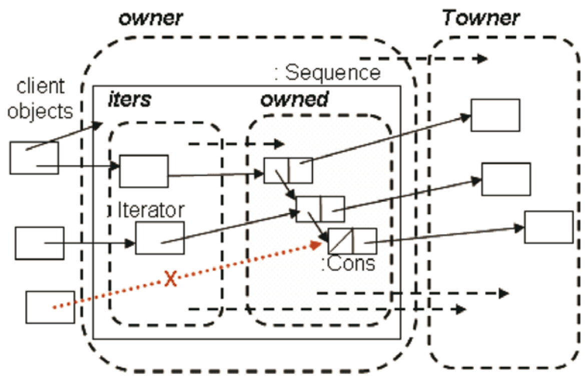

Fig. 4. A conceptual view of the ownership and aliasing model in AliasJava. The rounded, dashed rectangles represent ownership domains, with a gray fill for private domains and no fill for public domains. Solid rectangles represent objects. The top-level shared domain contains the highest-level objects in the program. Each object may define one or more domains that in turn contain other objects. Dashed arrows represent link permissions between domains.

Domain Permissions. Objects within a single ownership domain can refer to one another but references can only cross domain boundaries if the programmer specifies an architectural link between the two domains when they are created. Each object can declare a policy describing the permitted aliasing among objects in its internal domains and between its internal domains and external domains. AliasJava supports two kinds of policy specifications:

- A link from one domain to another, denoted with a dashed arrow in Fig. 4, allows objects in the first domain to access objects in the second domain.
- A domain can be declared public. Permission to access an object automatically implies permission to access its public domains.

Fig. 5 shows how the Sequence Java code can be annotated with aliasing information to model the constraints expressed in Fig. 4. The first two lines of code within the Sequence class declare the owned domain, the iters domain, and a link from the iters domain to its owned domain, allowing the iterators to refer to objects in the linked list. The iters domain is public, allowing clients to access the iterators, but the owned domain is private so clients must access the elements of Sequence through its iterator interface rather than traversing the linked list directly. In addition to the explicit policy specifications mentioned above, AliasJava includes the following implicit policy specifications:

(1) An object has permission to access other objects in the same domain.
(2) An object has permission to access objects in the domains that it declares.

The first rule allows the different Cons elements in the linked list to access each other, while the second rule allows the sequence to access its iterators and linked list. Any reference not explicitly permitted by one of these rules is prohibited according to the principle of least privilege. It is crucial that there is no transitive access rule: for example, even though clients can refer to iterators and iterators can refer to the linked list, clients cannot access the linked list directly because the sequence has not given them permission to access the owned domain. Thus, the policy specifications allow developers to specify that some objects are an internal part of an abstract data type's representation, and the compiler enforces the policy, ensuring that this representation is not exposed.

```
class Sequence<Towner> assumes owner -> Towner {
  domain owned;
  public domain iters;
  link owned -> Towner;
  link iters -> Towner, iters -> owned;
  owned Cons<Towner> head;
  void add(Towner Object o) { head = new Cons<Towner>(o,head); }
  iters Iterator<Towner> getIter() {
    return new SequenceIterator<Towner, owned>(head); }
}
class Cons<Towner> assumes owner -> Towner {
  Cons(Towner Object obj, owner Cons<Towner> next)
    { this.obj=obj; this.next=next; }
  Towner Object obj;
  owner Cons<Towner> next;
}
```

Fig. 5. AliasJava code for the `Sequence` abstract data type. Sequence holds a linked list as its internal representation in the private `owned` domain. A public `iters` domain holds its iterators. Link declarations specify that iterator objects in the `iters` domain have permission to access objects in the `owned` domain; outside objects cannot directly access `Cons` elements. Both domains can access the domain parameter `Towner`.

Domain Parameters. Components can share objects with connected components by binding domain parameters to actual ownership domains, allowing both components to access the objects in these domains. Thus ownership domain parameters generalize the concept of shared variable connectors introduced in SADL (Moriconi et al., 1995) to allow richer forms of object-oriented interaction between components.

In Fig. 5, the `Sequence` class is parameterized by the domain parameter `Towner` using syntax similar to Java version 1.5 for generics. The `head` field is of type `owned Cons⟨Towner⟩`, denoting a `Cons` linked list element that resides in the `owned` domain and holds an object that resides in the `Towner` domain. The `add` member function constructs a new `Cons` element for the object passed as argument and adds it to the head of the list. Skipping ahead to the definition of the `Cons` element class, we see that it is also parameterized by the domain parameter `Towner`. The class contains a field `obj` holding an element in the list, along with a `next` field referring to the next `Cons` element (or null, if this is the end of the list). The `next` field has type `owner Cons⟨Towner⟩`, indicating that the next element in the list has the same owning domain as the current element (i.e., all the elements are part of the `Sequence`'s `owned` domain).

Domain Hierarchy. The set of named ownership domains each object declares are nested within the domain that owns the object, so ownership defines a forest of trees where each parent owns its children and the roots of the tree are unique. Unique objects may be assigned to an ownership domain, attaching one ownership tree as a subtree of another. All connected components must be part of an ownership domain declared by the component making the connection. Fig. 3 shows the additional ownership and aliasing annotations as comments on the previous GraphicsPipeline ArchJava example. In the `Graphics-Pipeline` component class, the fields types are annotated with the implicit ownership domain `owned`, meaning that `generate`, `transform`, and `rasterize` are subcomponents of the `GraphicsPipeline` component instance that owns them.

Unique Data. While ownership is useful for representing persistent aliasing relationships, it cannot capture the common scenario of an object that is passed between objects without creating persistent aliases. Objects to which there is only one reference, including newly created objects, are annotated `unique` in AliasJava. Unique objects can be passed from one ownership domain to another as long as the reference to the object in the old ownership domain is destroyed when the new reference is created.

The pipeline architectural style (Garlan and Shaw, 1993) prohibits data sharing between components. To enforce this architectural invariant in the `GraphicsPipeline` example, using AliasJava, the `Shape` objects are annotated as `unique` to ensure that shapes are handed off from one component to another, i.e., that no component may retain a reference to a shape object after it passes it on to the next component. This invariant allows the developers of each component to assume they have exclusive access to the shape they are manipulating.

Lent Data. AliasJava allows one ownership domain to temporarily lend an object to another ownership domain with the constraint that the second ownership domain will not create any persistent references to the object. For example, annotating a method parameter as `lent` indicates that it is a temporary alias. A `unique` object can be passed to a method as a `lent` argument even without destroying the original unique reference. The method can pass on the object as a `lent` argument to other methods but cannot return it or store it in a field. Using `lent`, an `owned` object can also be temporarily passed to an external method for the duration of a method call, without any risk that the outside component might keep a reference to that object. Thus, the `lent` annotation preserves all of the reasoning about the `unique` object, but adds practical expressiveness to AliasJava.

Shared Data. Objects marked with the `shared` annotation may be aliased globally. Unfortunately, little reasoning can be done about `shared` references, except that they may not alias non-`shared` references. However, `shared` references are essential for interoperating with existing runtime libraries, legacy code and static fields, all of which may refer to aliases that are not confined to the scope of any object instance.

In the next section, we show how we used the constructs of ArchJava and AliasJava to precisely specify and enforce the control flow and data sharing architecture of the HillClimber subject system.

4. The HillClimber case study

In the case study, we re-engineered the system using the following activities:

(1) Identifying the source architecture.
(2) Identifying a target architecture.
(3) Analyzing the original program in Java to look for known problem areas.
(4) Restructuring the original program in Java.
(5) Re-engineering the original program to ArchJava to express the architectural control flow.
(6) Periodically checking against the target architecture.
(7) Annotating the program with aliasing and ownership to express the architectural data sharing using AliasJava.

Although we list an overall sequence of activities, the re-engineering process was iterative. For instance, after we started re-engineering the code (Step 5), we realized that we had not been aggressive enough in re-structuring the original program (Step 4), so we had to go back and make additional changes. In addition, many program changes may be required to reach the desired target architecture or the target architecture itself may need to be changed (Step 2). Expressing the software architecture in ArchJava and AliasJava highlighted several refactoring opportunities (Steps 4 and 5) by making the control flow and the data sharing between components explicit. We next describe each activity in turn, and for each step, we will point out any tools that were used to facilitate the re-engineering process.

4.1. Identifying the source architecture

The first step in re-engineering HillClimber was to determine its current architecture and use this as a basis for developing a target architecture to be explicitly specified and enforced using ArchJava. Although HillClimber has been maintained by several developers over several years, there were no artifacts other than source code that document its design. Furthermore, source code comments were sparse and sometimes out of date. It was therefore necessary to recover the source architecture by analyzing the source code.

The HillClimber application is one of several CIspace applications that share a common `graphFramework` package that includes abstract implementations of key classes. Classes that are shared among several CIspace applications include the `GraphCanvas`, `Graph`, `Node` and `Edge` classes (see Fig. 6). In HillClimber, these are subclassed respectively as `HillCanvas`, `HillGraph`, `HillNode` and `HillEdge`. In the HillClimber application, another two classes, `HillWindow` and `HillEngine`, are tightly coupled with the key `graphFramework` classes. These classes implement the core functionality of the HillClimber application and exhibit complex communication patterns. The HillClimber application is of sufficiently manageable size that the source architecture could be recovered by

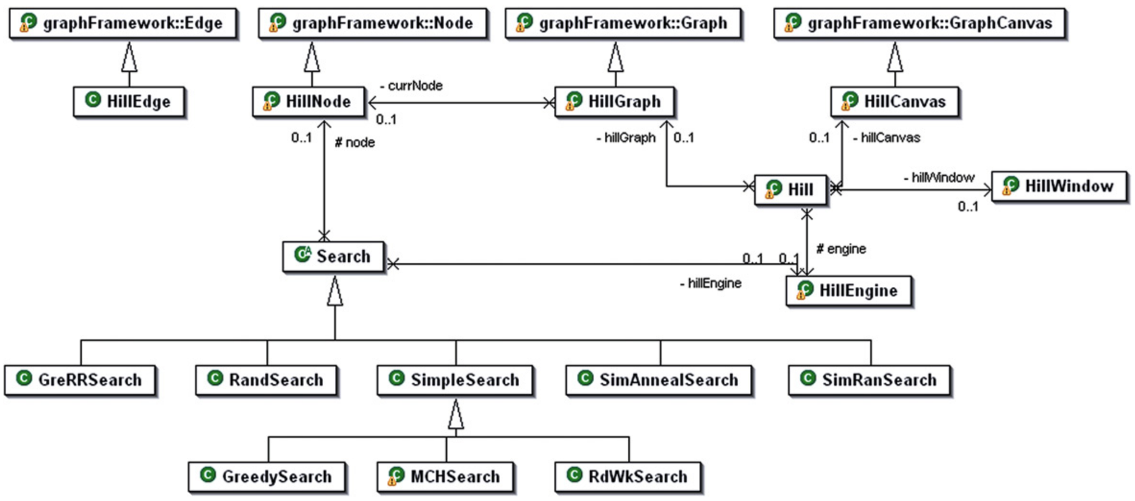

Fig. 6. UML class diagram retrieved from the original Java implementation using the EclipseUML tool [Omo05], illustrating the inheritance between the classes in the `graphFramework` package and the HillClimber-specific classes.

manual inspection: given the key classes, the relationships between them can be discovered by recording communication between components, such as method calls. Fig. 1 was generated by the LDM tool that analyzes dependencies between classes (Sangal et al., 2005). Fig. 1 shows the key components and the arrows indicate the discovered presence of communication between them; one can see that most components were communicating with most other components.

4.2. Identifying the target architecture

Developing the target architecture involves the following steps: (a) identifying the architectural styles in use, if any; (b) identifying the top-level components in the source architecture and reusing those components in the first iteration of the target architecture; (c) identifying which elements of the architecture are static and which elements are dynamic; and (d) determining the desired communication pattern between the identified top-level components.

For a graphical application such as HillClimber, we could have made the communication patterns between components more loosely coupled and conform to the implicit invocation architectural style (Garlan and Shaw, 1993). However, to avoid a significant departure from the source architecture and the resulting code rework, we chose roughly the same top-level components as in the source architecture.

As was done in similar case studies (Aldrich et al., 2002a,b), it is often ideal to have the original developers draw the idealized architecture. In this case study, the third co-author was a former developer and maintainer on the CIspace project, so he posited the target architecture based on his previous knowledge of the architecture of the underlying framework. It will become apparent later when we compare the target architecture to the actual architecture, that the target architecture left out many legitimate communication between components. This supports the hypothesis that developers have a conceptual model of their architecture that is mostly accurate, but that this model may be a simplification of reality (Aldrich et al., 2002a).

Unlike the original architecture, we desired in the target architecture a simplified, minimal communication pattern with loosely coupled components to improve component reusability and ease future maintenance of the system. We removed unneeded communication paths. For example, the `engine` component drives changes to the `graph` and it is not necessary for it to communicate directly with the `canvas`, whose function is to display the state of the `graph` component. Similarly, the `window` component that implements the user interface does not need to communicate directly with the `graph`.

We documented the target architecture using the Acme Architecture Description Language (Garlan et al., 2000) and its available tool support, AcmeStudio (Schmerl and Garlan, 2004). Fig. 7 shows the top-level components in the desired target architecture. Unlike Fig. 1 which shows the static organization of the source code in terms of classes, Fig. 7 shows the runtime structure of the system in terms of component instances.

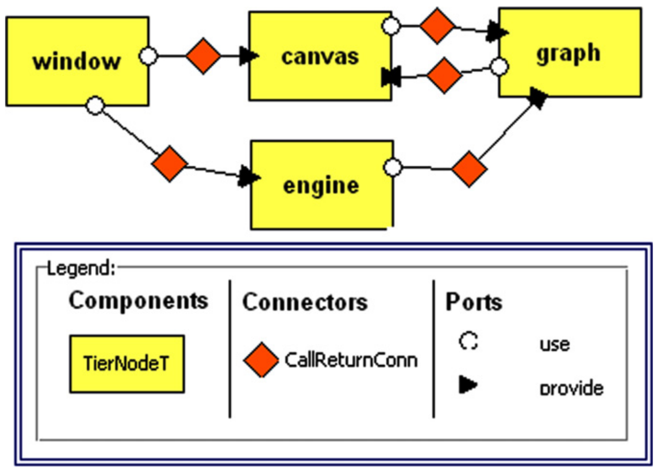

Fig. 7. C&C view showing the desired target runtime architecture. The assigned port types encode the control flow. Components `node` and `edge` are not shown.

In ArchJava, each component instance corresponds to an instance of a single component class: `window`, `canvas`, `engine` and `graph` are instances of `HillWindow`, `HillCanvas`, `HillEngine` and `HillGraph` respectively. Furthermore, components `window`, `canvas`, `engine` and `graph` are singleton static instances. In the desired target architecture, port types are used to encode the directionality of the communication: e.g., a connection from a port of type `use` on component `window` to a port of type `provide` on component `engine` means that `engine` provides a service to `window`, or alteratively `window` requires a service from `engine`, i.e., control flows from `window` to `engine`.

4.3. Analyzing the original program

The goal of this step is to consider the structural properties of the original program that require refactoring in preparation for re-engineering. For example, some object-oriented implementations are difficult to re-engineer to ArchJava, because many object-oriented patterns involve passing object references and ArchJava restricts passing component references in order to enforce the communication integrity property.

Top-level Elements. The previously identified top-level elements are those that will become component classes. At this point, it is important to consider whether the top-level elements being turned into component classes are conceptually part of the architecture rather than just data structures. Using component classes as data structures is bound to be awkward because component classes are not really intended to fill that role: data structures demand flexibility and ArchJava's component classes impose more rigid constraints such as they cannot be passed as references or stored in arrays.

In HillClimber, we initially decided to turn the following Java classes into ArchJava component classes: `HillWindow`, `HillCanvas`, `HillEngine` and `HillGraph`. We considered `HillNode` and `HillEdge` as data structures; similarly, we left many classes (e.g., `Set`, `IntList`) as ordinary Java classes because they are not part of the architecture.

Object Sharing. We studied the sharing of objects. As discussed in (Aldrich et al., 2002b), ArchJava does not allow a component to be shared by two container components. Thus, structures that are shared between components should be left as ordinary objects, unless the sharing can be easily replaced with method calls through the container component's port.

In HillClimber, there were many objects that were shared between various component instances, so their classes were left as ordinary classes. In particular, `HillNode` and `HillEdge` objects are instantiated by the `HillCanvas` object if they are created interactively by the user or by the `HillGraph` object if the graph is loaded from a persistent representation. Similarly, objects of class `Constraint` which implements all the functionality related to constraints, are shared between objects of type `HillCanvas`, `HillEdge` and a variety of other dialogs, so they were left as ordinary classes.

Initialization Order. It is important to discover the order in which the top-level elements are initialized. This information is needed when constructing the static architectural instances and their static connections: e.g., in HillClimber, `HillCanvas` had to be initialized before `HillGraph`.

Communication Patterns. At this point, one should get a rough idea of the extent to which the original code violates communication integrity rules. For instance, if there are many cases of passing around component objects or interfaces, this is an indication that significant work may be needed to convert the design to one that can be implemented in terms of ports and connections. In HillClimber, this was indeed the case: references of type `HillWindow`, `HillCanvas`, `HillEngine`, and `HillGraph` were being passed as constructor arguments or as method arguments.

When examining communication patterns, we identified *navigation code* (Demeyer et al., 2002), i.e., code that traverses a series of object links before calling a method on the final object, a well known symptom of misplaced behavior that violates the Law of Demeter (Lieberherr and Holland, 1989) and leads to unnecessary dependencies between classes. We used simple pattern matching to identify some navigation code as explained in Demeyer et al. (2002). More advanced techniques such as approaches based on aspects (Lieberherr et al., 2003) have also been proposed.

In HillClimber, we found many occurrences of navigation code in the component classes, e.g., class `HillWindow` included code such as the following: `getCanvas().getGraph().setLineWidth(...)`.

Encapsulation. We looked for fields that are not encapsulated. In a modern integrated development environment, fields shown in a tree hierarchy are color-coded based on visibility, making them easy to identify. In HillClimber, as can be seen in Fig. 8, there were many un-encapsulated fields.

```
public class HillEngine {
  public HillCanvas canvas;
  private HillGraph graph;
  public int dt = 100; // delay time

  public HillEngine(HillGraph graph, HillCanvas canvas) {
    this.graph = graph;
    this.canvas = canvas;
    // Default heuristics
    stepCount = 0;
    searchAlgs = new Search[8];
    searchAlgs[0] = new RandSearch(this);
    ...
  }
  public void step() {
  ...
    if (!canvas.inline) {
       ((HillWindow)canvas.parent).setButtonsSolved(true);
  ...
  }
  ...
}
```

Fig. 8. Original Java implementation of the HillEngine class. Note the un-encapsulated fields and the *code navigation* in the implementation of the `step` method.

Inheritance. Having identified the component classes, we verified that an ordinary class would not have a superclass that is a component class. ArchJava allows component classes to extend ordinary classes, so that legacy libraries could invoke the inherited methods of components through references to the appropriate superclass.

Initially, we wanted to keep all the classes in the `graphFramework` as ordinary classes. However, after we converted the subclasses to component classes, we noticed that the inherited methods of some of these components were being invoked arbitrarily through their inherited interfaces, threatening communication integrity. In particular, we discovered that the subclasses `HillGraph` and `HillCanvas` were communicating through their respective base classes `GraphCanvas` and `Graph`: several methods in the `Graph` classes called a `getCanvas()` method which returned a reference to the canvas object. We refactored the program to have an explicit `canvas` port on the base `Graph` class, while remaining an ordinary class. The `canvas` port was also accessible in the `HillGraph` class as an inherited port.

Using the inheritance feature in ArchJava requires additional considerations: ArchJava's sound type system rules prevents a subtype from requiring more methods than its super-type; otherwise, component substitutability would break. In HillClimber, we had to add several required methods to the `graphPort` port in the `GraphCanvas` class although these methods were only needed in its subcomponent class `HillCanvas`.

Singleton Objects. We studied how the classes that are to become component classes are being instantiated. For some components, it may be preferable to make them singleton instances and use static connections because ArchJava offers a more straightforward way to implement singleton components. In HillClimber, `HillWindow`, `HillEngine`, `HillCanvas`, and `HillGraph` are singleton instances.

Object Re-initialization. We examined methods that perform re-initialization of objects to study if they release and reallocate new objects or if they reuse existing objects by resetting their state. We wanted to let the core components in the HillClimber system form a static architecture with component instances that persisted for the entire execution of the program. Our rationale was that this architecture would be simpler to reason about than a completely dynamic architecture since architectural connections would be set up at startup time. We modified the program to re-initialize the same `HillGraph` instance instead of reallocating a new instance each time a graph was loaded.

4.4. Restructuring the original program

The goal of this step is to restructure or refactor the original program in Java before re-engineering the program into ArchJava.

Converting a program to ArchJava may involve significant restructuring if the implementation does not match the target architecture well. Restructuring will be inevitable because many object-oriented patterns rely on passing references and ArchJava restricts that to enforce communication integrity. We attempted to proactively restructure all the potential trouble areas in the original program. However, it was hard to determine when to stop; since additional refactoring was likely to happen during the actual conversion to ArchJava, we delayed many of the difficult refactorings until they were necessary. In addition, restructuring the original program helped familiarize us with the code base, as in the "refactor to understand" re-engineering best practice (Demeyer et al., 2002).

We made heavy use of the built-in support for refactoring in the Eclipse Object development environment (Object Technology International, 2003) to avoid introducing defects during this stage. In addition, a recommended best practice is to have an extensive set of unit tests (Fowler et al., 1999) and run the unit tests after each refactoring. Since HillClimber is primarily a GUI application, we performed functional tests after each non-trivial refactoring. Some of the important refactorings are discussed next.

Renaming. When enforcing architectural structure in code and recovering architectural structure from code, program identifiers become important since the ArchJava implementation will also serve as architectural specification. Objects corresponding to component instances were assigned meaningful variable names to more clearly convey the architectural intent, e.g., we used `edgeDialog` instead of `dlg`.

Other practical considerations included checking that none of the identifiers used in HillClimber conflicted with new keywords introduced by the ArchJava language extension (such as `connect`, `port`, etc.). Similarly, since ArchJava requires the Java programming language version 1.5, we had to check that the code would compile with Java 1.5. Refactoring tools can greatly assist in renaming by performing capture-avoiding substitutions.

Encapsulation. All fields on classes that are intended to become component classes should be encapsulated and be accessible only through accessor and modifier methods, i.e., no fields should be public, static, transient, or volatile. ArchJava will consider as illegal any non-private and non-protected component fields in order to enforce communication integrity. In HillClimber, we encapsulated fields in all classes, including superclasses in the `graphFramework` package. Even if a class is not directly intended to become a component class, one of its subclasses can be turned into a component class.

Unnecessary Code. Eliminating unnecessary code is useful at this point. For instance, we eliminated fields of reference type that were never assigned to or read from locally, as well as local variables of reference type that were not in use. When annotating the program with AliasJava, every reference type will need to be annotated, assuming the default is not suitable. Annotating unused fields and variables will cause unnecessary extra work. Fortunately, the Eclipse Java compiler (Eclipse Java Development Tooling) has many options related to detecting unnecessary code. For instance, in HillClimber, `HillEngine` had declared a field of type `OptionsDialog` that was not in use. We also found several local variables mainly of type `String` that were not in use. Of course, we did not eliminate unused code such as unused method parameters that might have been there in order to make the system more flexible or extensible.

Arguments of Type Component Class. In most cases, ArchJava does not allow constructor or method arguments to have the type of component classes. A common object-oriented pattern involves passing of the communicating objects as argument to a non-default constructor to ensure that the references are set correctly. We found it useful to temporarily replace this pattern with explicit calls to setters and getters; once the program is converted to ArchJava, the setters will be converted into ArchJava connect statements, as getters and setters taking component types will be illegal. See Fig. 9 for how the `HillEngine` constructor was refactored.

Initialization Code. Object-oriented programmers often perform all the initialization aggressively in a given class's constructor. However, initialization code in the constructor should not call any port methods since those ports would still be unconnected in the constructor. The ArchJava compiler statically warns about possibly unconnected ports in a constructor.

This seems to be a common pattern in component programming. For instance, in Microsoft's ATL library for COM component programming (Microsoft Active Template Library), a `FinalConstruct()` method is provided, where the rest of the initialization can be completed, such as aggregating other objects and the library guarantees that `FinalConstruct()` is called after the constructor.

In HillClimber, there were many such instances. We followed the same pattern in all cases: we kept the constructor minimal, and moved the initialization code into a separate `init()` method. Object instantiation sites had to be changed to make sure that the new `init()` method was called on all instantiated objects, and called only after all the static and dynamic connections had been established. Unfortunately, we had little tool support for this type of refactoring so additional care had to be exercised to avoid introducing defects into the program with this refactoring.

Constructor Calls to Overridable Methods. Constructors must not call overridable methods: the superclass constructor runs before the subclass constructor, so the overriding method in the subclass will get invoked before the subclass constructor has run. If the overriding method depends on initialization performed by the subclass constructor, then the method will not behave as expected (Bloch, 2001). If these calls remain when the program is converted to ArchJava, runtime exceptions occur if the overriding methods depend on ports having already been connected. Currently, ArchJava does not statically warn about calls to overridable methods[2] which may be accessing ports. A sophisticated linear type system to check for all disconnected ports would be required for static checking and is not currently implemented.

We actually found one problematic instance in the HillClimber application: the constructor of the `graphFramework.Node` class was calling a virtual `updateSize()` method, which was overridden inside the subclass `HillNode` and where it was accessing the `canvas` port. Fortunately, there exists tool support to look for these kinds of errors: for instance, running the tool EclipsePro Audit (Instantiations, 2006) on the original Java implementation uncovered around 70 instances where the constructor invoked a non-final method. However, not all of the flagged style violations were errors.

Navigation Code. Previous ArchJava case studies (Aldrich et al., 2002a) explained how navigation code is often a significant problem when converting to ArchJava: being proactive and eliminating as much as possible of it will be effort well spent. In HillClimber, for instance, we replaced `((HillWindow)canvas.parent).setSolved(true);` by declaring a field `window`, making sure that the field is initialized and changing the call to `window.setSolved(true)`.

Port Types as Interfaces. This refactoring is not essential, but we found it helpful. For many of the HillClimber classes that were being converted into component classes, we extracted into interfaces all the public methods available on the class including methods inherited from the base

```
public class HillEngine {
  ...
  private HillWindow window;
  private int dt = 100;  // delay time

  public HillEngine() {
    // Default heuristics
    stepCount = 0;
    searchAlgs = new Search[8];
    RandSearch randSearch = new RandSearch();
    // TODO: Convert this to connect statement
    randSearch.setWindow(window);
    ...
  }
  public void setDt(int dt) { this.dt = dt;}
  public int getDt() { return dt; }

  // TODO: Remove this once in ArchJava
  public HillWindow getWindow(){return window; }
  public void setWindow(HillWindow window) {
        this.window = window; }
    ...
}
```

Fig. 9. Refactored HillEngine Java class.

[2] By default, all public or protected methods in Java are virtual.

classes. These interfaces were useful for adding the correct required and provided method signatures on the ports as they were being declared since ArchJava does not currently support explicit port types.

4.5. Expressing the architectural control flow

From this point onwards, we switched to the ArchJava environment and could no longer use the Eclipse Java development environment. The first step was to simply rename the `*.java` files to `*.archj` and recompile using the ArchJava compiler. As long as no program identifiers are using any reserved keywords, the ArchJava development environment will compile HillClimber without further modification or error, but at this point, the ArchJava type system would not be enforcing any communication integrity.

ArchJava annotations were added incrementally to convert key communication relationships from standard method invocations to the port communication construct as described below. See Fig. 10 for the resulting ArchJava code.

```
public component class HillEngine {
  // Ports
  public port /* HillCanvas */ canvas  {
    requires boolean isInline();
    ... }
  public port /* HillGraph */ graph  {
    requires int numEdges();
    ... }
  public port /* HillWindow */ window  {
    requires void setButtonsSolved(boolean solved);
    ... }
  private port /* HillWindow */ p_window {
    provides Applet getApplet() {
    return window.getApplet(); }
    ... }
  // Glue internal port to external port
  private port /* HillGraph */ p_graph {
    provides int numEdges() {
      return graph.numEdges();}
    ... }
  public port /* HillEngine */ engine {
    provides void setDt(int dt);
    provides int getDt();
    ... }
  // Child components
  private final RandSearch randSearch = new RandSearch();
  // Static connections
  connect engine, randSearch.engine,  ...
  connect p_graph, randSearch.graph, ...
  ...
  public HillEngine() {
    ...
    // Note: Do most of initialization in init()
  }
  public void init() {
    ...
  }
  public void step() {
    ...
    if (!canvas.isInline() ) {
         window.setButtonsSolved(true);
    ...
  }
  ...
}
```

Fig. 10. Re-engineered HillEngine component class in ArchJava.

- **Changing an ordinary class into a component class**: the change itself was simple but often required many additional changes to pass communication integrity checks, e.g., there can no longer be any method parameter having a type corresponding to a component class.
- **Adding a port to a component class**: we followed the following guidelines when adding ports to ease future program understanding.
  - We defined unidirectional ports in order to clarify the directionality of the communication, i.e., a given port exposes only provided methods or only required methods and never both.
  - We created separate ports on each component to keep the "interface" (i.e., the set of methods provided or required) of each port as narrow as possible.
  - We named each port to correspond to that of the component instance it is intended to require services from or provide services to.
  - In the case of inherited ports, we defined all the required methods on the port of the base component class and added any provided methods on the derived component class. This is actually required by ArchJava's component substitutability rules.
- **Changing a field link into a connection:** in many cases, this simply involved converting an instance variable to a port. If the port is given the same name as the deleted instance variable, and if there are no direct calls to public fields (this is where encapsulating all the fields pays off), method call receivers need not be modified because the port call syntax will be identical. For each method that is called on a given port, we declared its signature as a required method on the new port (see Fig. 10).
- **Adding static connections:** the newly created ports on static component instances were connected using static connections.
- **Using dynamic constructs:** for dynamic component instances, we used ArchJava's dynamic constructs (connect patterns and expressions (Aldrich et al., 2002a)). As discussed previously, in many cases, we followed consistently the pattern of instantiating a component, connecting its ports, then calling an initialization method to complete the component's initialization as shown in the code snippet below:

```
FontDialog fd = new FontDialog( (java.awt.
Frame)this.parent );
connect(fd.canvas, canvas);
fd.init();
```

4.6. Checking against the target architecture

To guide the re-engineering activity toward the target architecture, we periodically checked the implementation against the target architecture by recovering an up-to-date architectural component-and-connector (C&C) view from the implementation using available ArchJava tool support (Abi-Antoun et al., 2006). The recovered C&C views contained purely structural information such as components, ports and their connections. Architectural styles and types were manually supplied since ArchJava does not currently represent that information. For instance, different port types were manually assigned based on whether a port exposed only provided methods, exposed only required methods or exposed both provided and required methods (see Fig. 11).

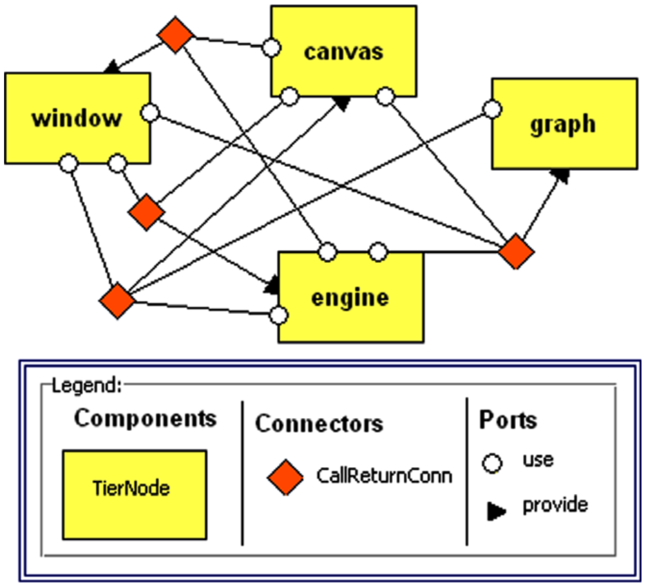

Fig. 11. The re-engineered HillClimber as-built runtime architecture.

The C&C views were useful for quickly assessing the current state of the implementation and determining how far it was from the desired target architecture. These snapshots helped produce a cleaner design since exposing the control flow information highlighted spaghetti style connections. Finally, the extracted C&C views helped with some of the code changes by enabling us to quickly identify the ports which were not connected.

Toward the end of the case study, we discovered that the original target architecture was too optimistic, and that there were good reasons for the additional runtime dependencies. For instance, the observed runtime dependency in Fig. 11 between component `engine` and component `canvas` was the union of what the given component `engine`, as well as any of it subcomponents, required. Since one of `engine`'s subcomponents required access to functionality from the `canvas`, that automatically forced the `engine` component to require functionality from the `canvas` component.

4.7. Expressing the architectural data sharing

Thus far, ArchJava provides a partial communication integrity guarantee (Aldrich et al., 2002a), enforcing the control flow architecture, i.e., communication through method invocation. The goal of this step is to add alias and ownership annotations to the program to provide a more complete assurance of communication integrity, and include communication through shared data.

Overall strategy. Annotating a program with AliasJava is also an iterative process, with the following overall sequence of steps:

(1) Determine the non-default ownership domains, i.e., domains other than shared and owned.
(2) Map each architecturally relevant object, i.e., each instance of a component class, to the appropriate ownership domain.
(3) Map the ownership domains and the links between them to domain declarations in the program, and to annotations on the architectural object instances: this requires propagating the appropriate domain parameters.
(4) Annotate any remaining objects with the most precise appropriate domain to pass the shared data communication integrity checks: in many cases, the lent defaults on method parameters and on local variable declarations are appropriate; for many non-architecturally relevant objects, one of the default domains is often appropriate (e.g., owned or shared); otherwise, one of the domain parameters is used.
(5) In the process of annotating object references, study the need to use more specific domain parameters. If a domain parameter does not really seem appropriate for the specific class, such as when a class mainly involved with business logic needs to access a domain parameter for user interface objects, identify the corresponding code as needing refactoring.

Ownership Domains. For HillClimber, we wanted to roughly separate the runtime component instances of the application into a tiered architecture corresponding to the Model View Controller (MVC) pattern (Krasner and Pope, 1988). We created a data domain corresponding to the model tier, a logic domain corresponding to the logic tier, and a user domain corresponding the user interface tier with the graphical user interface components. We wanted to place objects of type HillGraph, HillNode, and HillEdge in the same ownership domain data. This also required putting objects of their respective base classes Graph, Node, and Edge in the same domain data. With these ownership domain annotations, we wanted to make sure that the graph and its associated nodes and edges were only modified by other components in the same data domain and its owner, which is important for preserving the integrity of the graph structure. We also wanted to put the algorithmic objects in their own domain, so we put objects of type HillEngine in the logic ownership domain. See Fig. 13 for the corresponding domain declarations in the top-level Hill component.

Domain Parameters. To satisfy the rules of the AliasJava language, we propagated the top-level domains as domain parameters to the related classes, including their base classes, as needed. See Fig. 12 for the usage of the ui, logic and data domain parameters in the HillEngine component class.

Default Domains. In HillClimber, we marked all instances of java.lang.String with the global shared domain. For many of the user interface widgets, we marked the corresponding objects as owned. In a few cases where we considered aliasing to be innocuous and to avoid the need to explicitly pass the ui domain parameter, we marked the corresponding objects as shared.

Annotations as Metrics. Many re-engineering approaches encourage the use of metrics to measure the quality of the rework (Simon et al., 2001). Adding the aliasing annotations to the re-engineered program expressed in ArchJava highlighted additional refactoring opportunities that cannot be easily noticed by looking at the control flow communication. While adding the aliasing annotations to the re-engineered program, we used the list of domain parameters added to a given class as a measure of the quality of the design. If unexpected domain parameters appeared, they often revealed unwanted data sharing relationships which led us to refactor the program, thereby possibly affecting the control flow communication as well.

For instance, during an early iteration, after we parameterized the GraphCanvas component class by the ui and the data domains, we found ourselves passing the ui domain parameter to the Graph class, the base class for HillGraph. We were surprised that the Graph base component class needed the ui domain since we were expecting to only pass it the data domain parameter. It turned out that Graph had a reference to GraphCanvas, so Graph only needed the ui domain parameter to properly annotate its GraphCanvas reference. This revealed that the subclasses HillGraph and HillCanvas were communicating through their respective base classes Graph and GraphCanvas as shown in the code snippet in Fig. 14. We ended up eliminating the canvas reference to the GraphCanvas from the Graph base component class, and replacing it with an explicit canvas port, which is inherited by the HillGraph class. This was not hard to do since all the methods needing the canvas reference were simple wrapper methods.

Similarly, we were surprised to have to pass the data domain parameter to a simple dialog class FontDialog, because the dialog had a reference declared with its most specific type GraphCanvas. In some cases, it would have been possible to generalize the type of the reference, and make it of type java.awt.Frame. In this case, since FontDialog needed to access some of the functionality of the GraphCanvas (which was parameterized with domains ui and data), we would have had to create an interface which was specific to GraphCanvas but which did not require the domain parameters. However, in the general case, it may not be possible to avoid passing the domain parameters around between closely related classes. For FontDialog, we ended up replacing the reference with a canvas port.

Annotation Statistics. We built a simple program analysis to obtain statistical information on the AliasJava annotations in use. Clearly, having a majority of the objects in

```
public component class HillEngine<ui, logic, data> {
    private ui PlotFrame<ui,logic> pFrame;
    // Array of search algorithms
    private logic Search<logic,data>[logic] searchAlgs=new Search<logic,data>[8];
    private logic AutoSolve autoSolve;
    private logic BatchRun batchRun;
    private final ui TraceDialog<data> traceDialog = new TraceDialog<data>();
    private logic Vector<logic> batchSteps;
    private final logic RandSearch<logic,data> randSearch =
                        new RandSearch<logic,data>();
    ...
    // Connect patterns
    connect pattern engine, PlotFrame.engine;
    connect pattern engine, AutoSolve.engine;
    connect pattern engine, BatchRun.engine;

    // Connect expressions
    connect engine, randSearch.engine, ...;

    // Ports
    public port /* HillCanvas */ canvas
    {
        requires boolean isInline();
        requires void repaint();
        ...
    }
    public port /* HillGraph */ graph
    {
         requires data Enumeration<data> getNodes();
         requires int numNodes();
         ...
    }
    public port /* HillWindow */ window
    {
        requires shared Applet getApplet();
        requires shared Point getLocation();
        ...
    }
    public HillEngine() {
    }
    public void init() {
        traceDialog.init();
    }
    ...
}
```

Fig. 12. HillEngine component class in ArchJava with AliasJava annotations.

the program marked as `shared` would have reduced the value of an annotation system. Overall, the field annotations were broken down as follows: 45% as `owned`, 34% as `shared`, 10% as `data`, 6% as `logic`, 4% as `ui` and 1% as other annotations. Variable and method parameter declarations were broken down as follows: 69% as `lent`, 14 % as `shared`, 16% as `data` and 1% as other annotations. In AliasJava, the default annotation for object fields is `owned`, and the default annotation for local variables and method parameters is `lent`, so if these values are appropriate, no annotation is necessary. Finally, as we mentioned earlier, we could have reduced further the number of `shared` annotations by explicitly passing an explicit `ui` domain parameter.

Data Sharing Visualization. Typical ADL C&C views are control flow centric, so we made a stylized use of C&C views to show data sharing using ports for ownership domains and shared data connectors for shared data communication, shown in Fig. 15. Furthermore, current ADLs do not easily support multiple projections (Clements et al.,

```
public component class Hill {
  domain data;
  domain userTier;
  domain logicTier;

  private final userTier HillWindow<userTier,logicTier,data> window =
                        new HillWindow<userTier, logicTier,data>(null);
  private final userTier HillCanvas<userTier,data> canvas =
                        new HillCanvas<userTier, data>(window, false);
  private final data HillGraph<logicTier,data> graph =
                        new HillGraph<logicTier,data>();
  private final logicTier HillEngine<userTier,logicTier,data> engine =
                        new HillEngine<userTier, logicTier, data>();

  connect canvas.graphPort, graph.graphPort, engine.graph, window.graph;
  connect canvas.canvas, graph.canvas, engine.canvas, window.canvas;
  connect engine.engine, canvas.engine, window.engine;
  connect canvas.window, window.window, engine.window;

  public Hill()
  {
      canvas.init();
      graph.init();
      window.init();
      engine.init();
  }
  public static void main( String[] args )
  {
     Hill system = new Hill();
  }
}
```

Fig. 13. The top-level component class in ArchJava with AliasJava annotations.

```
Before:
public class Graph<data> ...
    // canvas containing this graph
    private shared GraphCanvas<data> canvas;

    public shared GraphCanvas<data> getCanvas() {
        return canvas;
    }
    public boolean isInSolveMode() {
        return getCanvas().getMode() == GraphConsts.SOLVE;
    }
...
After:
public component class HillGraph<logic, data> extends Graph<data> {
...
    public port canvas // This replaces getCanvas() from the base class
    {
        requires boolean isSolveMode();
        ...
    }
...
}
```

Fig. 14. Earlier version of the `Graph` class exhibiting communication with the `GraphCanvas` class. The reference to GraphCanvas was eliminated from the base class, and replaced with a port canvas in the `HillGraph` sub-class.

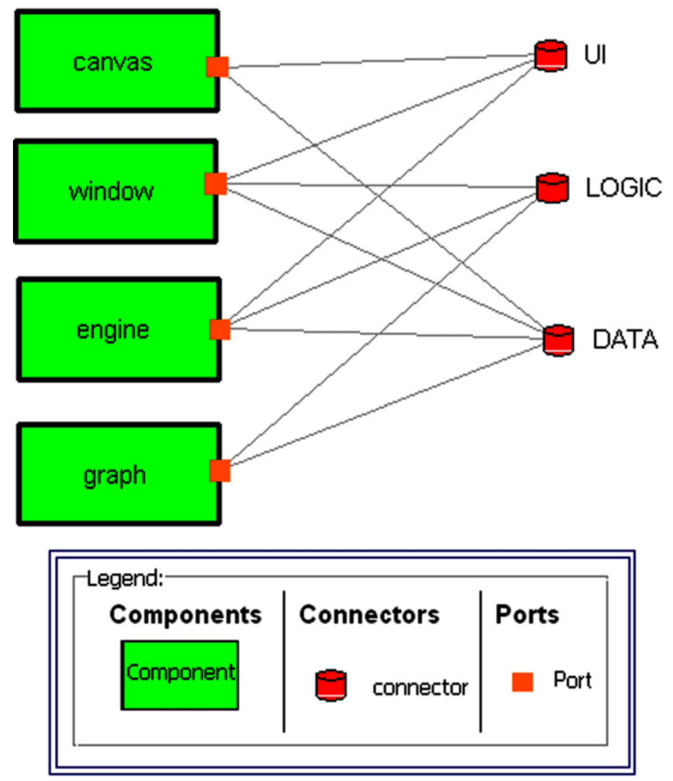

Fig. 15. A visualization of the runtime data sharing architecture of the re-engineered HillClimber application using a C&C view. Each component has one port to indicate that it shares data with other components. A shared data connector is defined for each ownership domain, i.e., the DATA, LOGIC, and UI connectors shown. Two components that are connected to a shared data connector communicate through shared data.

2003) of the same view: we required a *control flow* projection and a *data sharing* projection of the same C&C View but we ended up with two separate C&C views which have to be maintained independently and thus are bound to diverge.

4.8. Summary

In summary, we used ArchJava and AliasJava to re-engineer a system of non-trivial size to enforce full communication integrity. Making both the control flow and the data sharing explicit in the implementation enabled us to reason about the implementation and we believe this will limit future architectural drift and erosion.

The soundness of the ArchJava and AliasJava type systems together with an effective change management policy can facilitate enforcing the architecture to prevent the loss of architectural structure in the re-engineered program, more easily than in the original Java program. The communication integrity property requires explicitly declaring any new runtime connections as they are added to the implementation. For instance, if a developer is adding additional control flow to the program, she will need to create a new port, add a new required or provided method to an existing port, add a new connect pattern, or add a new connect statement. Similarly, if a developer is changing the data sharing architecture, he will need to declare additional domain parameters and pass additional domain arguments to object allocation sites. Code reviews or inspections could look for such changes prior to check-in to see if the additional communication patterns are allowed with respect to the system's runtime architecture. Ideally, one would have tool support to compare the changed architecture to the system's desired architecture (Abi-Antoun et al., 2006) and only allow certain changes. In the original Java program, changing the runtime architecture may be as simple as passing a reference to an object which makes it harder to determine which code modifications are changing the system's runtime architecture.

4.9. Effort estimates

With the current level of tool support, re-engineering a large program to use ArchJava and AliasJava may require significant effort. A careful log of the number of hours involved in the various steps was not kept since the case study was conducted in different phases. There was also a large number of interruptions to fix various problems encountered in the tools, and significant time spent on tool setup and learning. When adding the AliasJava annotations, we did not separately track the time spent annotating the parts of the Java Standard Library that were in use as we should have, since this is a one time investment that can be amortized over several re-engineered systems. The entire case study, i.e. the steps described above including restructuring, re-engineering to ArchJava and annotating with AliasJava could probably be performed in less than 40 hours excluding the time spent to learn ArchJava and Alias Java and assuming a fully annotated Java Standard Library.

Expressing the control flow architecture in ArchJava required significant source code restructuring to make the code conform to the intended architecture. However, adding the AliasJava annotations required changing many more lines of code than expressing the control flow architecture did. The metrics for the changed lines of code were computed using the GNU diff (MacKenzie et al., 2003) and the GNU diffstat utilities. All numbers reported by diffstat are measured in lines of code and ignore all blanks. As a reminder, the original Java program consisted of 16,000 lines of code and 92 classes contained in 82 files.

Restructuring the Java program. Re-structuring the program in Java did touch quite a few files. In total, there were 61 files changed. In those files, there were 201 line insertions, 504 line deletions and 1702 line modifications. In summary, the changes affected 15% of the total lines of code. In this case, most of the code changes were performed automatically by the Eclipse refactoring tool support.

Re-engineering to ArchJava. Re-engineering the restructured Java program to ArchJava to express the control flow architecture affected 43 files: there were 678 line insertions, 243 line deletions and 1,503 line modifications, affecting about 15% of the lines of code in the program. In this case, most of the changes to the code were performed manually. The changes are predominantly additions and modifications of lines of code; some of the deletions in this step simply undo some of the temporary additions during the restructuring step such as the intermediate accessors and modifiers discussed in Section 4.4. There are many different architectural views of a Java program that could be expressed using ArchJava, and the choice of which one to use relies critically on human architectural knowledge and judgment. Thus we do not believe that this transformation can be done completely automatically. However, tools e.g., TXL (Cordy, 2006) have been developed to ease the development of custom refactoring transformations and could be useful in capturing some of the Java to ArchJava restructuring operations once the architectural structure has been decided.

Adding AliasJava annotations. As predicted, adding the AliasJava annotations to the re-engineered ArchJava program to express the data sharing architecture affected a large number of files. There were 84 files changed, 29 line insertions, 12 line deletions and 3475 line modifications (excluding annotations to the Java standard library). In this case, most of the changes to the code were performed manually, since ownership annotation inference remains an open research problem (discussed later).

By far, most of the differences are changes to existing lines of code: in fact, over 20% of the lines of code of the re-engineered ArchJava program had to be modified. The observed additions and insertions are mostly incidental since we had to make some minor changes to the code because we were using an older annotated version of Java Standard Library which had been partially annotated in previous case studies. For instance, we replaced uses of `StringBuilder` with regular string concatenation. Also, in the process of annotating local variables, we discovered that some variables were not in use, so we deleted their declarations. Finally, we had to replace some legal Java syntactic constructs to work around certain limitations or bugs of the AliasJava compiler, e.g., replace array initializer syntax with the equivalent longer syntax.

4.10. Case study limitations

We did not empirically evaluate that the re-engineered ArchJava HillClimber implementation is actually easier to understand and evolve than the original Java implementation. That would require having junior programmers (e.g., UBC summer interns) co-evolve the ArchJava implementation in parallel with the original Java implementation to see if the system architecture is preserved better than if it has been left in pure Java and to see if the maintainers run into some of the expressiveness issues in ArchJava discussed in Section 6.2. However, the current level of tool support for ArchJava does not make this an attractive proposition and co-evolving the two implementations would constitute double work. Finally, this case study did not address the complex issues that are likely to arise when re-engineering an application that relies heavily on middleware (e.g., Enterprise Java Beans; Sun Microsystems, 2006). Additional case studies are an important element of future work in this area.

5. Related work

We previously discussed in Section 3.1 some of the approaches related to ArchJava and AliasJava. This section overviews related work in the area of re-engineering and similar case studies.

Re-engineering. Many published re-engineering case studies (Britcher, 1990; Gannod et al., 1998; Jacobson and Lindström, 1991) illustrate best practices for re-engineering and deal with legacy systems that are much larger than our subject system. We described a case study in-the-small which emphasized using a recently proposed language that enforces communication integrity checks. However, the current level of tool support for the research languages we used may limit the viability of the re-engineered system.

Re-engineering often deals primarily with module views (Schwanke and Platoff, 1993) and aims to improve the static system structure. In our approach, we used mainly instance-level runtime architectural views to visualize and reason about the subject system (Figs. 11 and 15), although we did consider static module view (Fig. 6) to take into account ArchJava and AliasJava's rules concerning inheritance. The research community is increasingly using architectural information during the re-engineering process, e.g. (Krikhaar et al., 1999; Riva et al., 2004; Tran et al., 2000). Our approach is similar to the architectural improvement proposed by Krikhaar et al. (1999): during an architecture impact analysis phase, structural changes are made to the architectural model to determine the resulting architecture; then, if the resulting architecture is desirable, the structural changes are applied to the system to obtain the new architecture, during a transformation phase. Similarly, several case studies (Cha et al., 2003; Chu et al., 1999; O'Cinneide and Nixon, 1999; Ping et al., 2003) emphasized the application of design patterns (Gamma et al., 1994) in order to clarify the design. Although component-and-connector architectural views are at a higher level of abstraction than UML diagrams illustrating design patterns, we used a common pattern for graphical user interface applications, the Model View Controller (MVC) design pattern to clarify the architectural intent when adding the ownership annotations.

Refactoring. Most of our program transformations have been previously discussed in the large body of refactoring literature (see Mens and Tourwé, 2004 and references therein). Refactoring typically implies that the destination language is exactly the same as the source language, whereas our work attempts to see the effect of re-engineering with a new language as a target and with design information explicitly encoded in the source text (another aspect missing from refactoring). We attempted to underline this difference in the paper by distinguishing an early restructuring step (Step 4 in Section 4) from the overall re-engineering process which also includes reconstituting the system in the ArchJava language with AliasJava annotations (Steps 5 and 7 in Section 4) to enforce the architecture in code.

Some refactorings, e.g., checking for duplicated code (Balazinska et al., 2000), were not applicable for HillClimber, perhaps due to its relatively small size. The refactoring community has proposed using metrics (Simon et al., 2001) or visualization (Emden et al., 2000) to identify "code smells" (Fowler et al., 1999) or to determine the effect of refactoring on the maintainability of the program (Kataoka et al., 2002). Our approach relies on language features to provide assurance of architectural conformance as well as highlight refactoring opportunities: we augment the program with architectural control flow and data annotations, use the annotations to reason about the runtime structure of the program and modify the annotated program based on what those annotations tell us. Finally, we visualize the instance-based runtime architectural view at a slightly higher level of abstraction than detailed design.

Other Related Case Studies. Unlike previously published case studies involving adding control flow or aliasing annotations to existing applications (Aldrich et al., 2002a,b; Müller and Poetzsch-Heffter, 2000), we were more methodical about having an explicit restructuring step in preparation for the re-engineering step. We explained each step in detail, as a starting point to maybe generalize the steps into an "architectural pattern language" (Goedicke and Zdun, 2002) to re-engineer an object-oriented implementation into an ArchJava implementation. A previously published AliasJava case study illustrated adding aliasing and ownership annotations to an existing Java application, Aphyds (Aldrich et al., 2002c). However, the Aphyds case study consisted of annotating only a self-contained part of the application with no user interface and affected fewer lines of code than this case study. Previous case studies have reasoned separately about communication through control flow (Aldrich et al., 2002a,b) or shared data (Aldrich et al., 2002c; Haechler et al., 2005). The HillClimber case study illustrated the benefit of reasoning about both in the same implementation as they can affect each other: adding the AliasJava annotations to the ArchJava implementation revealed additional "code smells" that were not noticed earlier simply by looking at the control flow architecture and led to additional restructuring. A case study using the Universes ownership type system (Müller and Poetzsch-Heffter, 2000) on an industrial software application is documented in Haechler et al. (2005). Although the subject system in the Universes case study is larger than HillClimber (around 55,000 lines of code), only a portion of the system was annotated. In the Universes case study, the author performed much of the restructuring in the process of adding the aliasing annotations. In many cases, adding ownership annotations generated many compile errors that took time to resolve. Finally, the Universes case study used mainly design-level object interaction views and only focused on the data sharing perspective. In the HillClimber case study, we used architectural views and reasoned about both control flow and data sharing.

6. Lessons learned

In this section, we describe some of the lessons we learned during the case study, in the belief that a wish list will provide impetus for improving the languages, techniques and tools we used. Many of these lessons may already be known, but unfortunately, it seems that some of them were ignored in the design of some of the more recent languages and tools that we used in this case study. We re-emphasize them for the next generation of languages and tools so they can handle larger systems.

6.1. Hints for language and tool designers

Keep It Iterative. The activities that seemed to make the process harder were precisely the ones that interfered with the iterative nature of the process: e.g., once a program is converted to ArchJava, refactoring support available for a Java program is no longer available, even if the ArchJava program still has many classes that are still plain Java classes. For instance, after we started migrating the code to ArchJava, we discovered that we had forgotten to encapsulate several fields in the base classes in the `graphFramework` package. Even though none of the classes in that package had been converted then to component classes or declared ports, we could no longer use refactoring tool support. Fowler offers an additional insight: "[...] irreversibility [is] one of the prime drivers of complexity. And agile methods [...] contain complexity by reducing irreversibility" (Fowler, 2003). Turning a Java program into an ArchJava program is an irreversible transformation.

Keep It Incremental. Having the ability to incrementally convert the program to ArchJava was extremely valuable. For instance, turning a class into a component class can suddenly generate many ArchJava compile errors (e.g., if that type was used as a constructor argument). However, there was always an easy workaround: non-component classes can have ports as well. So in some cases, we resorted to first adding ports and then converting the class to a component class after we better understood the dependencies. However, the legacy mode was not always perfect: currently, one can add ports to an ordinary class, but one cannot add connect expression except for a component class. If one is not careful and does not connect the inherited ports when instantiating the subclasses, these ports may remain disconnected and produce runtime exceptions.

Tolerate Incompleteness. Even development environments are moving towards tolerating incompleteness. For example, the Eclipse Java Development Tooling allows running and debugging code which still contains unresolved errors. The ability to temporarily tolerate incompleteness and errors is even more critical for a language such as ArchJava. During a re-engineering activity, mixing the two concerns (i.e., the implementation and the architecture) is hard if one wants to first codify the desired architecture, yet maintain a running system. Some Architecture Description Languages require declaring ports but do not require declaring the provided and required functionality on the ports. ArchJava always requires both. ArchJava currently elegantly supports architectural design with abstract components and ports, which allow an architect to specify and typecheck an architecture before beginning program implementation. However, it does not easily support the ability to incrementally enforce architectural conformance checking. Some possible options could include having different warning levels, or having a setting to relax some of the checks for required and provided functionality, at least temporarily.

Automate as Much as Possible. Although automation has been a major, long-term goal of software engineering research and practice to aid engineers in development and evolution tasks, we realized that the tool support is still rudimentary in many cases. Although we were able to use standard refactoring tools for many of the simple program transformations, we think it would have been helpful to have a set of tools to further automate the process of restructuring the original program. Identifying program code in need of refactoring is still mostly a manual exercise. However, we believe that for this kind of re-engineering, where the rules are relatively well known, automated support would be particularly useful. For instance, a tool could take a list of the intended component classes, look for known problems (such as public fields of those types or constructor arguments of those types), suggest a list of refactoring (e.g., encapsulate fields), automatically construct some of the refactorings, and finally give the user the opportunity to preview the proposed changes and accept or reject them.

6.2. Perceived ArchJava limitations

There are important limitations to the currently available tool support for the ArchJava language that affect the viability of the re-engineered HillClimber application. The ArchJava development environment offers only basic features and does not provide support for debugging and refactoring. However, there are more fundamental issues that we encountered during this case study that we would like to see addressed in future versions of ArchJava.

Separate Inheritance from Subtyping. ArchJava's type system, following the conventional type systems of implementation languages, focuses on implementation-level substitutability. A crucial characteristic of conventional type systems is that all the external services a component requires are stated in its type, along with a subset of the services that the component provides. This characteristic ensures that when the type of some component is given, any component implementation that conforms to the type can be used in the actual system without violating basic rules of component composition.[3] From an architectural information hiding standpoint, and in keeping with having interfaces that are as narrow as possible, we thought that it was counterintuitive to push the declarations of all required methods into the base class, although some were only really needed in the subclasses. The problem is that inheritance and subtyping need to be separated, so one can inherit to get reuse without subtyping. ArchJava should separate these two to get substitutability of subtypes but not necessarily of subclasses (where it may not be needed), and to get as much information hiding as is needed.

Runtime Exceptions. ArchJava is designed on the premise that if a program type checks successfully, the match between the implementation and specified architecture is guaranteed. Unfortunately, certain classes of architectural errors are only caught at runtime. For instance, since a component instance can still be freely passed between components as an expression of type `java.lang.Object`, an exception is thrown if an expression is downcast to a component type outside the scope of its parent component instance. Similarly, a runtime exception is thrown when accessing a port that is not connected. Extensive testing is still needed to verify that no serious defects are introduced into a program when re-engineering it into an ArchJava program. To some extent, the risk of introducing such defects that did not previously exist may lower the value of the re-engineering activity.

Missing Port Types. ArchJava does not have explicit port types. We resorted to using comments next to the port name to specify the type of the port. The absence of port types in ArchJava imposed some amount of code duplication for declaring required methods. On the other hand, this allowed each component to expose a narrower interface in a given port, by only requiring or providing methods that it actually uses or implements directly. Having port types would also enable a feature similar to the C# explicit interfaces (Wiltamuth and Hejlsberg, 2002). In HillClimber, we had to rename some methods to avoid having a clash between two provided methods of the same name. ArchJava currently supports supplying a provided method body inline in the port declaration, but this may be unwieldy for lengthy methods. An alternative syntax using port types and explicit interfaces is proposed in Fig. 16.

Missing Port Directionality. ArchJava does not currently allow developers to express that a port can have only required methods or only provided methods. There is no way to distinguish cases where this happens by chance or where the architect's intent is that the port is unidirectional.

Missing Strategy for Component Construction. As discussed earlier, it would be helpful if ArchJava provided a better mechanism for completing the initialization of a

[3] This distinction in the way required and provided services are handled is known as the contravariant subtyping principle.

```
public component class HillEngine {
    public ICanvasPort canvasPort  {
       provides void doThis();
    }
    public IEnginePort enginePort {
       provides void doThis();
    }
    // Constructor
    public HillEngine()
    {
    }
    // Provided method implementations
    public void IEnginePort.doThis() {
        ...
    }
    public void ICanvasPort.doThis() {
        ...
    }
}
```

Fig. 16. Proposed syntax using port types and explicit interfaces.

component, and guarantee that the initialization method would always be called. One option is to use a Final Constructor as in some component systems as discussed earlier. More generally, ArchJava should have some kind of strategy that creates subcomponents as they are needed, rather than at a particular fixed point which could be either too soon or too late.

Relaxing Architectural Constraints. In many real world architectures, it is often necessary to make exceptions to architectural constraints. For example, in a layered architecture with strict performance requirements, it may be necessary to tunnel between layers so that calls skip one or more layers in order to follow more direct routes, as explained in Griswold and Notkin (1995). Embedded architecture description languages such as ArchJava currently have no mechanism for handling these exceptions. For instance, even type-safe programming languages such as C# allow programmers to mark code blocks as unsafe (Wiltamuth and Hejlsberg, 2002) and perform low-level operations that are normally not available.

Tightening Architectural Constraints. Even when architecture is specified in source code and enforced by the compiler, there are still methods of circumventing the imposed architectural structure. For example, components that are intended to be isolated from each other could still communicate via shared files or network messages.

Maintaining the Architecture in Code. Although architecture descriptions in source code reduce the need to maintain external documentation, developers still need to devote resources to maintaining the embedded architectural description. In a degenerate case, it is possible to circumvent architectural constraints by making the corresponding objects non-architecturally relevant (i.e. using ordinary Java classes), and internal to a single architecture-level component, for example. Therefore, effort must be devoted to ensuring that the embedded architecture specification is appropriate, complete and up-to-date. Having the ability to externally visualize the architecture can prevent such scenarios.

6.3. Perceived AliasJava limitations

Manually adding aliasing and sharing annotations to a program of non-trivial size taught us a few lessons for ownership systems.

Incrementally and Partially Specifying Annotations. As they stand today, aliasing annotations cannot always be added by small increments: in many cases, adding annotations to the entire system is a prerequisite for analyzing a specific component or a specific data sharing aspect. This makes it relatively challenging to get started on a large code base. The subject system of a previous AliasJava case study (Aldrich et al., 2002c) cleanly adhered to the Model View Controller (MVC) architectural style. Having such a design enabled us to annotate only the model part of the architecture: but even there, the model component and all its subcomponents had to be completely annotated.

In HillClimber, it was not as easy to perform an incremental or partial annotation of the program since the design is much more tightly coupled. According to the Structural Analysis for Java tool (alphaWorks, 2004), changing the `graphFramework.Graph` class could potentially affect 53 out of the 92 classes (i.e., 57%). And indeed, making the class `graphFramework.Graph⟨data⟩` take a domain parameter affected many classes. Similarly, annotating the other top-level classes meant that almost the entire program had to be touched, as shown by the changed lines of code measurements discussed earlier. A similar lesson was drawn in Haechler et al. (2005), even though that case study was also attempting to annotate only a part of the program.

Inferring Annotations Automatically. Inferring annotations automatically is a problem that is currently receiving quite a bit of attention (Aldrich et al., 2002c). Based on the previous lessons, we think that this attention is warranted in order to make using aliasing and ownership annotations practical for everyday use by programmers.

Tool support that even partially infers and automates adding annotations would be extremely useful: e.g., if domain parameters are added to a class, have a tool make all the derived classes also take at least those domain parameters. Similarly, if a domain parameter is added to the class, the same domain parameter could be added to any class that instantiates the parameterized class. In the absence of practical inference of annotations and tool support, it is a lot harder to follow an incremental approach when annotating a program.

Supporting Legacy Mode. AliasJava does not allow implementations of interface methods or overrides of abstract methods to change the ownership annotation. Allowing subtypes to be annotated in ways that are incompatible with their supertypes breaks substitutability and is unsound. However, two applications instantiating a framework, library or third-party component may need to annotate the same methods in incompatible ways depending on their usage.

For instance, AliasJava annotates the `this` receiver by default with `lent`. However, application code implementing abstract methods or interface methods (e.g., from a user interface framework) can do arbitrary things with the `this` pointer such as storing it or passing it to other components. So when annotating legacy GUI code, the `lent` default may not be appropriate. We think this problem can be solved in AliasJava by adding some kind of parameter to the superclass, e.g., by making each method parametric in the ownership type of its receiver.

Adding Annotations to Libraries. Adding annotations to libraries or third-party components is often not an option if the source code is not available. And even when the source code is available, annotating a library should be avoided, as it will have to be redone for each new version of the library. Finally, two applications may require two incompatible annotations of the same library. A similar issue with another ownership system was aptly called the "method needed twice" problem in Haechler et al. (2005). A feasible solution that is currently not implemented is to allow annotations to be added to existing code through external files. This would solve the unavailable library code problem as well as the library evolution problem, as one would only have to re-annotate the modified interfaces of the library.

In HillClimber, we ran into this problem: we had a `graphFramework.GraphApplet` class which extends the library `java.applet.Applet` class. For the HillClimber application class `GraphApplet`, we wanted the overridden method `init` to have a `shared this` annotation, whereas the AWT `Applet init` method had a `lent` (the default) annotation for `this`.

Supporting Common Idioms. The AliasJava annotations were generally perceived as too heavyweight for many relatively common programming idioms. For instance, in many object-oriented applications, it is common to have a child object point back to its parent object. When ownership parameters are added, if a class `A` has a reference to a class `B` with domain parameters $\langle \alpha, \beta, \gamma \rangle$, then class `A` has to take domain parameters α, β, γ in addition to any domain parameters of its own δ, ϵ, resulting in class `A` with domain parameters $\langle \alpha, \beta, \gamma, \delta, \epsilon \rangle$. However, the first three domain parameters may only be used to properly annotate the reference to type `B`. This quickly can lead to lengthy domain parameter lists and can be counterintuitive if some of the domains are intended to be private. Having shorthand annotations such as `owner` used in the Sequence example (used in Fig. 5) does help a little.

Supporting Additional Annotations. The design of the AliasJava annotation system focused on precisely specifying the aliasing relationships between objects in the system. As a result, it does not include a few annotations that are used in some of the related work. For instance, package-based confinement (Bokowski et al., 1999) provides a middle ground between the global shared domain and domains that are local to an object. AliasJava supports `shared`, which indicates the worst case of a globally aliased reference. One typically wants a certain object to be shared between a small number of classes, not globally, without having to create many domain parameters for fine grained sharing. For instance, read-only annotations (Boyland et al., 2001) can express useful additional invariants about a system. Similarly, external uniqueness (Clarke and Wrigstad, 2003) may reduce the burden of annotating internal objects. These features could be added to a future version of the AliasJava language in a natural way. Additional case studies with these combined language features would be needed to evaluate their practicality.

Since the case study was first conducted and in light of the lessons learned, we made various enhancements to ArchJava and AliasJava to address some of the adoptability challenges discussed above. We converted many compilation errors to warnings which makes it easier to maintain a running system that can be functionally tested while the re-engineering is underway. To support incremental and partially specifiable annotations, we re-implemented the AliasJava system as Java 1.5 annotations as opposed to a language extension (Abi-Antoun and Aldrich, 2006). Having the annotated program remain a legal Java 1.5 program enables the use of all the tool support available for Java programs, and makes it easier to justify the claim that programs are easier to evolve with AliasJava annotations than without. Finally, expressing AliasJava using annotations could make it easier to extend the language in a non-breaking way. However, ArchJava is not yet available as an annotation-only system.

7. Conclusion

Architectural specifications are often not sufficiently maintained along with the actual implementation. Languages such as ArchJava effectively enforce architectural structure in source code and promise to help prevent the loss of architectural information, and the resulting architectural drift and erosion. Although such languages are best applied during the initial development phases, they can be applied to existing systems to re-engineer, document and enforce the desired structure.

By eliciting and refining some of the underlying re-engineering principles such as those outlined in Aldrich et al. (2002b), we hope to make the re-engineering activity less daunting, less painful and less error prone. We also pointed out several limitations of the languages and the tools we used that will need to be overcome before they can be used effectively in production software development.

Acknowledgements

We thank Thomas LaToza, Bradley Schmerl and the anonymous reviewers for comments on this material. We also thank Alan Mackworth for granting us the permission to use the HillClimber code base and publish details of the case study. This work was supported in part by NASA cooperative agreements NCC-2-1298 and NNA05CS30A, NSF grant CCR-0204047, a 2004 IBM Eclipse Innovation Grant, the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems" and was performed as a joint research project in Strategic Partnership between Carnegie Mellon University and Jet Propulsion Laboratory.

References

Abi-Antoun, M., Aldrich, J., 2006. JavaD: Bringing Ownership Domains to Mainstream Java. Technical Report CMU-ISRI-06-110.

Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., Garlan, D., 2006. Differencing and merging of architectural views. In: Proc. 21st IEEE International Conference on Automated Software Engineering (ASE'06), pp. 47–58.

Aldrich, J., Chambers, C., 2004. Ownership domains: separating aliasing policy from mechanism. In: Proc. European Conference on Object-Oriented Programming (ECOOP), pp. 1–25.

Aldrich, J., Chambers, C., Notkin, D., 2002a. ArchJava: connecting software architecture to implementation. In: Proc. International Conference on Software Engineering (ICSE), pp. 187–197.

Aldrich, J., Chambers, C., Notkin, D., 2002b. Architectural reasoning in ArchJava. In: Proc. European Conference on Object-Oriented Programming (ECOOP), Lecture Notes In Computer Science, vol. 2374, pp. 334–367.

Aldrich, J., Kostadinov, V., Chambers, C., 2002c. Alias annotations for program understanding. In: Proc. of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pp. 311–330.

Balazinska, M., Merlo, E., Dagenais, M., Lague, B., Kontogiannis, K., 2000. Advanced clone-analysis to support object-oriented system refactoring. In: Proc. IEEE Working Conf. Reverse Engineering (WCRE), p. 98.

Bloch, J., 2001. Effective Java. Addison-Wesley.

Bokowski, B., Vitek, J., 1999. Confined types. In: Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pp. 82–96.

Boyland, J., Noble, J., Retert, W., 2001. Capabilities for sharing: a generalization of uniqueness and read-only. In: Proc. European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science, vol. 2072, pp. 2–27.

Britcher, R., 1990. Re-engineering software: a case study. IBM Systems Journal 29 (4), 551–567.

Brodie, M.L., Stonebraker, M., 1995. Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach. Morgan-Kaufmann Publishers.

Cha, J.E., Kim, C.H., Yang, Y.J., 2003. Architecture based software reengineering approach for transforming from legacy system to component based system through applying design patterns. In: Software Engineering Research and Applications, Lecture Notes in Computer Science, vol. 3026, pp. 266–278.

Chapman, R., 2001. SPARK – a state-of-the-practice approach to the Common Criteria implementation requirements. In: Proc. International Common Criteria Conference, July.

Chikofsky, E., Cross, J., 1990. Reverse engineering and design recovery: a taxonomy. IEEE Software 7 (1), 13–17.

Chu, W.C., Lu, C.W., Shiu J.P., He, X., 1999. Pattern-based software reengineering: a case study. In: Proc. Sixth Asia Pacific Software Engineering Conference (APSEC '99), pp. 300–308.

CISpace: Tools for learning Computational Intelligence. Available from: <http://www.cs.ubc.ca/labs/lci/CIspace>.

Clarke, D.G., Wrigstad, T., 2003. External uniqueness is unique enough. In: Proc. European Conference on Object-Oriented Programming (ECOOP), pp. 176–200.

Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., 2003. Documenting Software Architecture: View and Beyond. Addison-Wesley.

Cordy, J.R., 2006. The TXL source transformation language. Science of Computer Programming 61 (3), 190–210.

Demeyer, S., Ducasse, S., Nierstrasz, O., 2002. Object-Oriented Reengineering Patterns. Morgan Kaufmann Publishers.

Eclipse Java Development Tooling (JDT) core. Available from: <http://dev.eclipse.org/viewcvs/index.cgi/jdt-core-home/main.html?rev=1.97>.

van Emden, E., Moonen, L., 2000. Java quality assurance by detecting code smells. In: Proc. Working Conference on Reverse Engineering (WCRE), pp. 97–106.

Fowler, M., 2003. Who needs an architect. IEEE Software 20 (5), 11–13.

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. Refactoring: Improving the Design of Existing Programs. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Gannod, G.C., Sudindranath, G., Fagnani, M.E., Cheng, B.H.C., 1998. PACKRAT: a software reengineering case study. In: Proc. Working Conference on Reverse Engineering (WCRE), pp. 125–134.

Garlan, D., Shaw, M., 1993. An introduction to software architecture. In: Ambriola, V., Tortora, G. (Eds.), Advances in Software Engineering and Knowledge Engineering I. World Scientific Publishing Company.

Garlan, D., Monroe, R., Wile, D., 2000. Acme: architectural description of component-based systems. In: Foundations of Component-Based Systems. Cambridge University Press, pp. 47–67.

Goedicke, M., Zdun, U., 2002. Piecemeal legacy migrating with an architectural pattern language: a case study. Journal of Software Maintenance: Research and Practice 14 (1), 1–30.

Griswold, W.G., Notkin, D., 1995. Architectural tradeoffs for a meaning-preserving program restructuring tool. IEEE Transactions of Software Engineering 21 (4), 275–287.

Haechler, Thomas. Applying the Universe type system to an industrial application: case study. Master Project Report, Department of Computer Science, Swiss Federal Institute of Technology, 2005.

IBM alphaWorks, Structural Analysis for Java tool 2004. Available from: <http://www.alphaworks.ibm.com/tech/sa4j>.

Instantiations, Inc. 2006. EclipsePro Audit tool. Available from: <http://www.instantiations.com/eclipsepro>.

ITU-T. 1999. Recommendation Z.100, Specification and Description Language (SDL). Geneva, Switzerland, November.

Jacobson, I., Lindström, F., 1991. Reengineering of old systems to an object-oriented architecture. In: Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pp. 340–350.

Jaktman, C.B., Leaney, J., Liu, M., 1999. Structural analysis of the software architecture – a maintenance assessment case study. In: Proc. TC2 First Working IFIP Conference on Software Architecture (WICSA1), pp. 455–470.

Kataoka, Y., Imai, T. Andou, H., Fukaya, T., 2002. A quantitative evaluation of maintainability enhancement by refactoring. In: Proc. International Conference on Software Maintenance (ICSM), pp. 576–585.

Krasner, G.E., Pope, S.T., 1988. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming 1 (3), 26–49.

Krikhaar, R., Postma, A., Sellink, A., Stroucken, M., Verhoef, C.A., 1999. Two-phase process for software architecture improvement. In: Proc. IEEE International Conference on Software Maintenance (ICSM), pp. 371–380.

Lam, P., Rinard, M.A., 2003. Type system and analysis for the automatic extraction and enforcement of design information. In: Proc. European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science, vol. 2743, pp. 275–302.

LaToza, T.D., Venolia, G., DeLine, R., 2006. Maintaining mental models: a study of developer work habits. In: Proc. IEEE International Conference on Software Engineering (ICSE), pp. 492–501.

Lattix Inc's Dependency Manager (LDM) tool. Available from: <http://www.lattix.com>.

Lieberherr, K., Holland, I., 1989. Assuring good style for object-oriented programs. IEEE Software 6 (5), 38–48.

Lieberherr, K., Lorenz, D.H., Wu, P., 2003. A case for statically executable advice: checking the law of demeter with AspectJ. In: Proc. 2nd International Conference on Aspect-Oriented Software Development (AOSD '03), pp. 40–49.

Luckham, D.C., Vera, J., 1995. An event based architecture definition language. IEEE Transactions of Software Engineering 21 (9), 717–734.

MacKenzie, D., Eggert, P., Stallman, R., 2003. Comparing and Merging Files with GNU Diff and Patch. Network Theory Ltd.

Madhav, N., 1996. Testing Ada 95 programs for conformance to rapid architectures. In: Proc. Reliable Software Technologies – Ada Europe 96.

Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. IEEE Transactions of Software Engineering 26 (1), 70–93.

Medvidovic, N., Oreizy, P., Robbins, J.E., Taylor, R.N., 1996. Using object-oriented typing to support architectural design in the C2 style. In: Proc. Fourth ACM Symposium on the Foundations of Software Engineering, pp. 24–32.

Mens, T., Tourwé, T., 2004. A survey of software refactoring. IEEE Transactions on Software Engineering 30 (2), 126–139.

Microsoft Active Template Library (ATL) for COM. Available from: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_atl_CComObjectRootEx.asp>.

Moriconi, M., Qian, X., Riemenschneider, R.A., 1995. Correct architecture refinement. IEEE Transactions on Software Engineering 21 (4), 356–372.

Müller, Peter, Poetzsch-Heffter, Arnd, 2000. Universes: a type system for controlling representation exposure. In: Poetzsch-Heffter, A., Meyer, J. (Hrsg.): Programmiersprachen und Grundlagen der Programmierung, 10. Kolloquium, Informatik Berichte 263, 1999/2000.

Murphy, G.C., Notkin, D., Sullivan, K.J., 2001. Software reflexion models: bridging the gap between design and implementation. IEEE Transactions on Software Engineering 27 (4), 364–380.

Object Technology International, Inc. Eclipse Platform Technical Overview, 2003. Available from: <http://www.eclipse.org/whitepapers/eclipse-overview.pdf>.

O'Cinneide, M., Nixon, P., 1999. A methodology for the automated introduction of design patterns. In: Proc. IEEE International Conference on Software Maintenance (ICSM), pp. 463–472.

Omondo EclipseUML. Available from: <http://www.omondo.com>.

Perry, D.E., Wolf, A.L., 1992. Foundations for the study of architecture. ACM SIGSOFT Software Engineering Notes 17 (4), 40–52.

Ping, Y., Kontogiannis, K., Lau, T.C., 2003. Transforming legacy Web applications to the MVC architecture. In Proc. Eleventh Annual International Workshop on Software Technology and Engineering Practice (STEP'04), pp. 133–142.

Prieto-Diaz, R., Neighbors, J., 1986. Module interconnection languages. Journal of Systems and Software 6 (4), 307–334.

Riva, C., Selonen, P., Systa, T., Xu, J., 2004. UML-based reverse engineering and model analysis approaches for software architecture maintenance. In: Proc. IEEE International Conference on Software Maintenance (ICSM), pp. 50–59.

Sangal, N., Jordan, E., Sinha, V., Jackson, D., 2005. Using dependency models to manage complex software architecture. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pp. 167–176.

Schmerl, B., Garlan, D., 2004. AcmeStudio: supporting style-centered architecture development. In: Proc. IEEE International Conference on Software Engineering (ICSE), pp. 704–705.

Schwanke, R.W., Platoff, M.A., 1993. Cross references are features. In: Machine Learning: From Theory to Applications, pp. 107–123.

Simon, F., Steinbrückner, F., Lewerentz, C., 2001. Metrics based refactoring. In: Proc. 5th European Conference on Software Maintenance and Reengineering (CSMR 2001), pp. 30–38.

Stevens, P., Pooley, R., 1998. Systems reengineering patterns. In: Proc. 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 1998, pp. 17–23.

Sun Microsystems. 2006. Enterprise JavaBeans. Available from: <http://java.sun.com/products/ejb/docs.html>.

Tran, J.B., Godfrey, M.W., Lee, E.H.S., Holt, R.C., 2000. Architectural repair of open source software. In: Proc. 8th International Workshop on Program Comprehension (IWPC), pp. 48–59.

Wiltamuth, S., Hejlsberg, A. C# Language Specification. Standard ECMA-334, 2nd edition, 2002.