A Case Study in Incremental Architecture-Based Re-engineering of a Legacy Application

Marwan Abi-Antoun
Institute for Software Research Intl (ISRI), Carnegie Mellon University
mabianto@cs.cmu.edu

Wesley Coelho
Department of Computer Science, University of British Columbia
coelho@cs.ubc.ca

Abstract

Without rigorous software development and maintenance, software tends to lose its original architectural structure and become more difficult to understand and modify.

ArchJava, a recently proposed implementation language which embeds a component-and-connector architectural specification within Java implementation code, offers the promise of preventing the loss of architectural structure.

We describe a case study in which we incrementally re-engineer an existing implementation with an eroded architecture to obtain an ArchJava implementation that more closely matches an idealized architecture. Building on results from similar case studies, we chose an application consisting of over 16,000 source lines of Java code and 80 classes that exhibited many characteristics of real-world legacy applications. We describe our process, some lessons learned, as well as some perceived limitations with the tools, techniques and languages we used.

1. Introduction

A legacy system is defined as one that significantly resists modification and evolution to meet new and constantly changing business requirements, regardless of the technology from which it is built [BS95]. Such characteristics are partly due to architectural problems, including *architectural erosion*, i.e., "violations in the architecture that lead to increased system problems and brittleness" and *architectural drift*, i.e., "a lack of coherence and clarity of form which may lead to architectural violation and increased inadaptability of the architecture" [PW92].

Missing or un-enforced architectural information is a key factor which contributes to architectural drift and erosion [JLL99]. The architecture of a software system is commonly described in documentation artifacts produced and maintained independently from source code implementations. Over time, the implemented system's design begins to drift from its original architecture. Eventually, the architectural specification may become too inaccurate to be used, leading to further degradation of the system structure.

Re-engineering is one way to correct architectural drift and erosion, 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" [CC90]. Re-engineering a legacy system can extend its lifetime, and delay the introduction of a new system built from scratch, resulting in cost savings.

In this paper we describe a case study in which a legacy system was re-engineered using ArchJava, a programming language that enforces architectural structure at the implementation level. Using the re-engineering paradigm of *abstraction*, *transformation* and *re-implementation* [JL91], we extract the architectural intent in the form of a target architecture and reconstitute the implementation in a form that we hope will limit future architectural drift.

This paper makes several contributions. We refine some of the design principles illustrated in [ACN02b] and build on results from similar case studies in [ACN02a, ACN02b]. We chose an application that exhibits more of the characteristics of legacy systems: a) it was developed and maintained by several different junior programmers over the course of several years, unlike previous case studies where the subject system was developed and maintained by a single developer (Aphyds and Taprats in [ACN02b]); b) it is a realistic code base developed and maintained by novice programmers unlike other subject systems (e.g., the Taprats system in [ACN02b]) intended for an object-oriented design competition; c) it did not have its original architecture drawn by the original developers and we could not talk to the maintainers; and d) it is larger in size (source lines of code) than the similar case studies that have been previously attempted. Finally, as

noted in [SP98], re-engineering expertise is lacking; so 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. Section 5 discusses some lessons learned as well as some perceived limitations of the tools, techniques and languages we used. Finally, we discuss some limitations of this case study in Section 6, and conclude.

2. The Case Study Application

For our case study, we used a legacy system called HillClimber, which 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; they 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 architectural information progressively leads to the degradation of program structure.

Our case study application, HillClimber, demonstrates stochastic local search algorithms for constraint satisfaction problems. Though it is relatively small, HillClimber is large and mature enough to contain complex design issues. Furthermore, HillClimber is a representative object-oriented application.

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*. Over time, this simple structural intent (which was not documented but existed implicitly in the source code, as is often the case) was damaged by modifications performed by developers who were unaware of this intent and therefore unable to preserve it. Figure 1 illustrates the structure after several years of modification by new developers. Clearly, the architectural intent has been lost: 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 [JLL99].

3. Architecture-Based Re-Engineering

There are many benefits to having a documented software architecture [CBB+03], ideally, using architecture description languages [MT00]. However, having a documented software architecture is often not enough: when developers move from design to implementation, architectural information is often lost.

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 independent architecture design documentation, even if it exists and is reasonably up-to-date. New developers may also change or violate the intended architecture because they are unaware of the underlying architectural intent.

3.1 Enforcing Architectural Structure in Code

Programming languages that support architecture at the implementation level offer a promising solution to these problems. By introducing syntax for describing architecture, high-level design decisions can be integrated with source code. There are several advantages to this approach:

- Since the architecture specification is captured in the source code, there is no need to maintain a separate artifact. The source code and its documentation will always be up-to-date.
- Developers will be more aware of the architecture because it is explicitly documented within the source and not in external documents.
- The architecture can be enforced using a type system. In this case, the type system is extended to prove that violations of the architecture do not exist, enforcing communication integrity [LV95], which means that two components in the implementation may communicate only if they are connected in the architecture.

3.2 Re-engineering

Since architecture is determined during design of a system, one would think 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 also be used on existing systems in order to recover, formalize, and enforce the architecture, i.e., re-engineer a legacy system to more effectively capture its architecture.

Using the re-engineering paradigm of *abstraction*, *transformation* (or reasoning about changes at a higher

abstraction level) and *re-implementation* [JL91], we extract the architectural intent in the form of a target architecture and reconstitute the implementation in a form that we hope will limit future architectural drift. Using higher-level information to reason about the existing code as well as the target code (in this case, architectural information), truly qualifies the activity as re-engineering. The abstraction and modification steps help avoid ending up with the same tightly coupled architecture that's very well enforced!

[Bri90] eloquently describes some advantages of re-engineering a legacy system instead of a complete rewrite: "re-engineering takes advantage of the profound effects of evolution. It preserves the functional behavior of a system that had been specified, designed, implemented, repaired, enhanced, verified, validated, and most importantly, used over years, while improving its quality". For the purpose of this discussion, the quality we are concerned with improving is the architectural structure of the system.

For a re-engineering case study such as this one, backwards compatibility with the current implementation language is essential. Incremental re-engineering, described in [BS95] as "take chicken little steps" avoids the complexity and the risks of big-bang re-engineering by transforming the system in small increments and always having a running version.

3.3 ArchJava

ArchJava [ACN02a] is a recently proposed implementation language that embeds a component-and-connector (C&C) architectural specification within Java implementation code. ArchJava's backward compatibility with Java and other software evolution features make it a suitable language for re-engineering existing real-world systems.

ArchJava allows architecture to be specified in source code by allowing programmers to define components, ports, and the connections between them. The ArchJava compiler enforces communication integrity, i.e., that communication among components is consistent with the explicit architecture.

4. Architecture-Based Re-engineering

For our case study, we performed the following activities to re-engineer the application:

- Identify the source architecture
- Identify a target architecture
- Analyze the original program (in Java)
- Restructure the original program (in Java)
- Re-engineer the original program (to ArchJava)
- Periodically check against the target architecture.

The re-engineering process was iterative. For instance, after we started re-engineering the code, we realized that we had not been aggressive enough in restructuring the original program, so we had to make additional changes. Many of the steps above may need to be repeated several times, since many changes may be required to reach the desired target architecture.

4.1 Identify 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 are no artifacts other than source code that document its design. Furthermore, source code comments are 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 components. Components that are shared among several CIspace applications include the `Window`, `GraphCanvas`, `Graph`, `Node` and `Edge` classes (See Figure 3). In the HillClimber application, another component, `HillEngine`, is tightly coupled with the key `graphFramework` components. These components 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 manual inspection. We also used a tool to generate UML class diagrams from an existing imple-

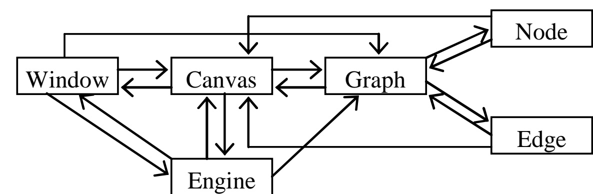

Figure 1: The Source Architecture.

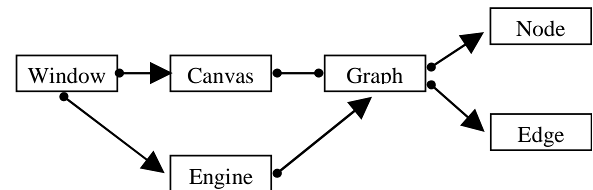

Figure 2: The Target Architecture: the arrows indicate unidirectional connections; i.e., the source component (circle end) requires the services of the target component (arrowhead end) and the target component does not initiate communication with the source component. Two circle ends indicate bi-directional communication.

mentation. However, such tools do not completely eliminate the need for a manual code inspection. For example, in Figure 3, the tool we used missed the obvious association between `HillGraph` and `HillNode` since `HillGraph` was using an untyped container to maintain references to `HillNode` objects. Some tools claim to not suffer from this limitation; however, one we tried (Womble [JW99]) did not scale to handle the HillClimber application.

Given the key components, the relationships between them were discovered by manually investigating the source code. Communication between components, such as method calls, was recorded. Figure 1 shows the key components and the arrows indicate the discovered presence of communication between them; one can see that communication between components follows a nearly arbitrary pattern, with most components communicating with most other components.

4.2 Identify the Target Architecture

Developing the target architecture involves the following steps: (a) Identify the architectural styles in use, if any; (b) Identify the top-level components in the source architecture and re-use those components in the first iteration of the target architecture; (c) Identify which elements of the architecture are static and which elements are dynamic; (d) Determine 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 conform to the Model-View-Controller (MVC) architectural style [KP88]. 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.

Unlike the original architecture, we desired 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 contents of the *Graph* component. Similarly, the *Window* component that implements the user interface does not need to communicate directly with the *Graph*. The resulting target architecture is shown in Figure 2.

4.3 Analyze 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.

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 (e.g., cannot be passed as references, cannot be stored in arrays, etc.) In HillClimber, we initially decided to turn the following Java classes into ArchJava component classes: `HillWindow`, `HillCanvas`, `HillEngine`, `HillGraph`, `HillNode` and `HillEdge`.

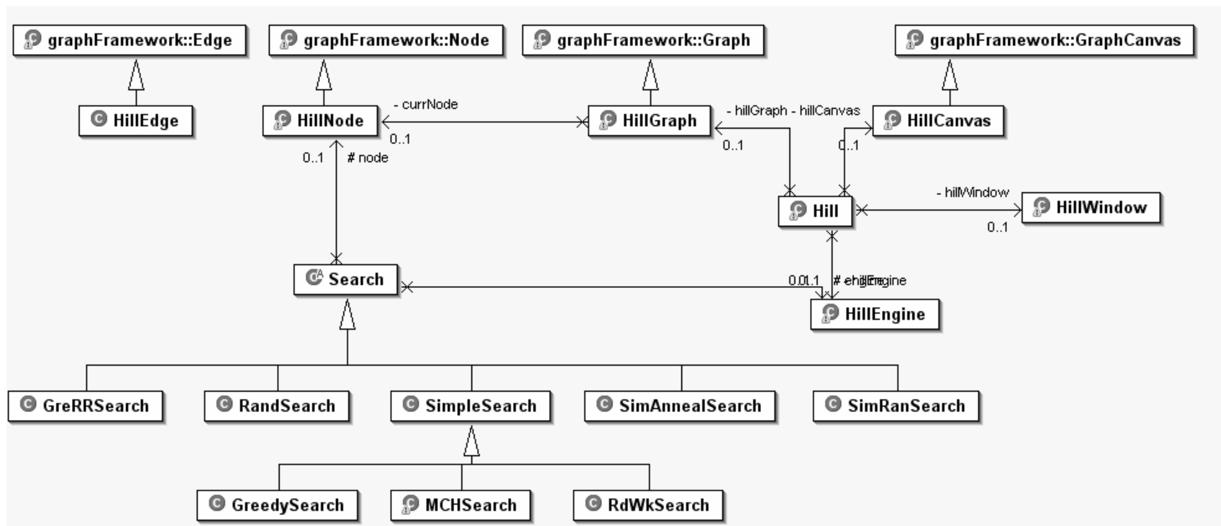

Figure 3: UML diagram retrieved from the original Java implementation using the EclipseUML tool [Omo05].

Object Sharing. Study the sharing of objects. As discussed in [ACN02b], 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, `HillNode` and `HillEdge` are shared between `HillCanvas` and `HillGraph`, so in the end, they were left as ordinary classes, but they do declare ports.

Initialization Order. It is important to discover the order in which the top-level elements are initialized. This information will be essential 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, identify "Navigation Code" [DDN02], i.e., code that traverses a series of object links before calling a method on the final object. It is a well-known symptom of misplaced behavior that violates the Law of Demeter [LH89], leading to unnecessary dependencies between classes. We used simple pattern matching to identify some navigation code as explained in [DDN02]. We did not use more advanced techniques, such as approaches based on aspect-oriented programming (e.g., [LLW03]). 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. Study how well fields are encapsulated. In a modern integrated development environment, fields shown in a tree hierarchy are color-coded based on visibility, making them easier to identify. It would have been helpful to also have tool support to query the implementation for such anomalies. In HillClimber, as can be seen in Figure 4, there were many un-encapsulated fields.

Inheritance Hierarchy. Having identified the component classes, verify that an ordinary class will not have a superclass that is a component class.

```
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);
  ...
  }
  ...
}
```

Figure 4: Original Java implementation of the `HillEngine` class. Note the un-encapsulated fields, and a method implemented using "code navigation."

ArchJava allows component classes to extend regular classes and interfaces, so that legacy libraries could invoke the inherited methods of components through references to the appropriate superclass. This enabled us to avoid converting classes from the `graphFramework` package to components. However, we noticed that the inherited methods of some these components were being invoked arbitrarily through their inherited interfaces, threatening the desired communication integrity. However, ArchJava guarantees that the new HillClimber-specific methods introduced in the HillClimber components can only be called through declared connections in the architecture.

Object Construction and Destruction. Study how the classes that are to become component classes are being instantiated. For some components, it may be preferable to make them static instances and use static connections because ArchJava offers a more straightforward way to implement static components.

In HillClimber, `HillWindow`, `HillEngine`, `HillCanvas`, and `HillGraph` are static instances, whereas `HillNode` and `HillEdge` are dynamic instances.

Study the construction of objects: of particular interest are non-default constructors. Component constructors may not have arguments of component type**.** Also, examine methods that perform re-initialization of objects; do they release and reallocate new objects, or do they reuse existing objects, by resetting their state? In HillClimber, a `HillGraph` instance was being reallocated, so we preferred to make it a static instance instead, with static connections to the other components.

4.4 Restructure 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. Refactoring will be inevitable because many object-oriented patterns rely on passing references, and ArchJava restricts that. We attempted to refactor proactively all the potential trouble areas in the original program. However, it is hard to determine when to stop; since additional refactoring was likely to happen during the actual conversion to ArchJava, we delayed many if the difficult refactorings until they were necessary.

Refactoring the original program was also helpful for becoming familiar with the code base, as in the "*refactor to understand"* re-engineering best practice [DDN02]. We made heavy use of the built-in support for refactoring in the Eclipse [Ecl03] development environment to avoid introducing defects during this stage. However, we did not follow the best practice of first having an extensive set of unit tests [FBB+09] and rerunning the unit tests after each refactoring. Some of the important refactorings are discussed next.

Rename. When enforcing architectural structure in code, and recovering architectural structure from code, names become important, since the ArchJava implementation will also serve as architectural specification: for example, we assigned component instances more meaningful names that clearly convey the architectural intent, e.g., use `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.

Encapsulate. 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. Furthermore, all fields of superclasses, even if the ones not intended to become component classes, should be encapsulated as well, as one may want to expose implementation methods from a super class as a provided functionality on the component sub-class. ArchJava will consider as illegal any non-private and non-protected component fields: this is one way that ArchJava enforces communication integrity. In HillClimber, we encapsulated many fields in many classes (including many in `graphFramework`).

Eliminate Constructor Arguments of Type Component Classes. ArchJava does not allow for constructor arguments having types of component classes. A common object-oriented pattern involves passing of the

```
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 stmt
    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; }
    ...
}
```

Figure 5: Refactored HillEngine Java class.

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 in order to facilitate converting the program to ArchJava. In ArchJava, getters and setters taking component types will be illegal, and these setters will have to be converted into ArchJava `connect` statements). See Figure 5 for how the `HillEngine` constructor was refactored.

Split Initialization Code from Constructor Code. Object-oriented programmers often perform all the initialization aggressively in a class constructor. However, care should be taken to not have initialization code in the construction that relies on calling port methods, since those ports would still be unconnected in the constructor. The ArchJava compiler will statically warn about possibly unconnected ports in a constructor. In HillClimber, there were many such instances. We followed the same pattern in all cases: we kept the constructor minimal, and moved initialization code into a separate `init()` method. This seems a common pattern in component programming: e.g., in Microsoft's ATL library for COM component programming [ATL], 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 this case, we had to be very careful to not introduce new defects into the program, by making sure that the `init()` method is actually called on all instantiated objects. Unfortunately, we had little tool support for this type of refactoring.

Eliminate Constructor Calls to Overridable Methods. [Blo01] explains why 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 de-

pends on initialization performed by the subclass constructor, then the method will not behave as expected. 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. To make things worse, by default (i.e., non-final) all public or protected methods in Java are virtual. Currently, ArchJava does not statically warn about calls to overridable methods 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 such 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.

Eliminate Navigation Code. [ACN02a] reports how navigation code is often a significant problem when converting to ArchJava: being proactive and eliminating as much as possible of it will not be wasted effort. 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).`

Extract Interfaces. This refactoring is not essential, but we found it helpful. For some HillClimber classes, (e.g., `HillEngine`), we extracted all the public methods available on a class, including methods inherited from the base classes, into interfaces.

4.5 Re-Engineer the Program

From this point onwards, we had to switch over to the ArchJava environment, and could not use the Eclipse Java development environment anymore. Since ArchJava is backwards compatible with the Java programming language, the first step when re-engineering to ArchJava is to 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 be able to compile HillClimber without further modification or error, but at this point, the ArchJava typesystem will not be enforcing any communication integrity.

ArchJava can be applied incrementally to convert key communication relationships from standard method invocations to the port communication construct. Several relationships between objects were converted to ArchJava using the re-engineering patterns described in great detail in [ACN02b], summarized here:

```
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);
    ...
  }
  ...
}
```

Figure 6: HillEngine component class in ArchJava.

- Change class to component class: this often requires many additional changes to pass communication integrity checks.
- Change a field link into a connection: in many cases, this is simply converting an instance variable to a port. If the port is given the same name as the deleted instance variable, statements that previously called the server object will not need to be modified because the port call syntax will be identical. This is only possible if there are no direct calls to public fields (this is where encapsulating all the fields pays off). For each method that was previously called on the callee, we declare its signature as a required method in the new port that now represents the provider (See Figure 6).
- Move creation to container component: this pattern required the most changes to the architecture. In retrospect, this step should have been addressed during the re-structuring step, [ACN02b] discusses in greater detail how this can problem be addressed.
- Finally, we used ArchJava's dynamic constructs (connect patterns and expressions) discussed in [ACN02a] for `HillNode` and `HillEdge`.

4.6 Check against the Target Architecture

To guide the re-engineering activity toward the target architecture, we periodically checked the state of 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 [AAG05].

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, assigning port types (based on whether a port only provides methods, only requires methods, or does both) was used to check the directionality of the communication (See Figure 7).

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. The snapshots helped produce a cleaner design, since exposing the control flow information can highlight spaghetti style connections. Second, they helped the actual migration effort: as long as the code compiled, the extracted C&C view quickly showed which ports were not connected (including those on dynamic instances). Finally, they helped visualize object sharing issues (e.g., by having `HillNode` and `HillEdge` instances appear as contained by both `HillCanvas` and `HillGraph`).

5. Lessons Learned

In this section, we describe some of the lessons we learned during the case study, in the hope that a wish list will provide impetus for improving currently available tool support.

5.1 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 to component classes or declared ports, we could not 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" [Fow03]. Turning a Java program into an ArchJava program is an irreversible transformation.

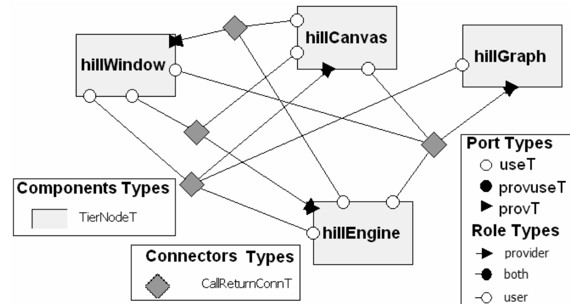

Figure 7: Intermediate recovered C&C view: port types (manually assigned) encode the directionality.

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.

Tolerate Incompleteness. Even development environments are moving towards tolerating incompleteness. For example, the Eclipse Java Development Tooling [JDT] 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. It would have been helpful to have a set of tools to further automate the process of re-structuring 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, identify navigation code), automatically construct some of the refactorings, and finally give the user the opportunity to preview the proposed changes and accept or reject them.

5.2 Perceived ArchJava Limitations

There are 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.

Runtime Exceptions. ArchJava is designed on the premise that if the type checker terminates 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.

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 imposes some amount of code duplication for declaring required methods. On the other hand, this allows having a narrower interface available to a port.

Missing Port Directionality. ArchJava does not currently allow developers to restrict the architectural intent that can be expressed with the component and port model, e.g., to express that a port can only have 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 Final Constructor. As discussed earlier, it would be helpful if ArchJava provided a better mechanism for completing the initialization of a component, and guarantee that the initialization method would always be called.

Missing Explicit Interfaces. Having a feature similar to C# explicit interfaces [C#02] would be useful, for similar reasons:

- Explicit interface member implementations allow interface implementations to be excluded from the public interface of a class when a class implements an internal interface that is of no interest to a consumer of that class.
- Explicit interface member implementations allow disambiguation of interface members with the same signature; otherwise, it would be impossible to have different implementations of interface members with the same signature and return type, or with the same signature but with different return types.

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 [GN95]. Embedded architecture description languages such as ArchJava currently have no mechanism for handling these exceptions. For instance, some programming languages, such as C#, allow programmers to mark code blocks as **`unsafe`** [C#02] 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 unrelated could still communicate via shared memory, shared files, or network messages. Although architecture descriptions in source code reduce the need to maintain documentation, developers are still required to devote resources to maintaining the embedded architectural description. In a degenerate case, it is still possible to circumvent architectural constraints by defining an entire program as internal to a single architecture-level component, for example. Therefore, effort must be devoted to ensuring that the coded architecture specification is appropriate and up-to-date. Having the ability to externally visualize the architecture can help avoid such scenarios.

6. Case Study Limitations

This case study did not demonstrate that the re-engineered ArchJava HillClimber implementation is actually easier to understand and evolve than the original Java implementation. It would be ideal if we could

have junior programmers (e.g., UBC summer interns) co-evolve the ArchJava implementation to see if the system architecture is preserved better than it would have been if left in pure Java, and to see if the maintainers are able to avoid the architectural violations discussed in the latter part of Section 5.2. However, the current level of tool support for ArchJava does not make this an attractive proposition.

In addition, 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 [EJB]). Additional case studies are an important element of future work in this area.

7. Conclusions

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 [ACN02b], we hope to make the re-engineering activity seem 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.

8. Acknowledgements

We thank Jonathan Aldrich for his helpful hints and technical support on the ArchJava compiler during the case study, and for detailed comments that significantly improved the paper. We also thank Alan Mackworth for granting us the permission to use the HillClimber code base and publish details of the case study.

9. References

[AAG05] Abi-Antoun, M., Aldrich, J., Garlan, D., Schmerl, B., Nahas, N., and Tseng, T. Improving System Dependability by Enforcing Architectural Intent. WADS, 2005.

[ACN02a] Aldrich, J., Chambers, C. and Notkin, D. ArchJava: Connecting Software Architecture to Implementation. In Proc. ICSE, 2002.

[ACN02b] Aldrich, J., Chambers, C. and Notkin, D. Architectural Reasoning in ArchJava. In Proc. ECOOP, 2002.

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

[Blo01] Bloch, J. Effective Java. Addison-Wesley. 2001.

[Bri90] Britcher, R. Re-engineering Software: A Case Study. In IBM Systems Journal, vol.29, No.4, 1990.

[BS95] Brodie, M. L., and Stonebraker, M. Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach. Morgan-Kaufman Publishers, 1995.

[C#02] Wiltamuth, S. and Hejlsberg, A. C# Language Specification. Standard ECMA-334, 2nd edition, 2002.

[CBB+03] Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. Documenting Software Architecture: View and Beyond, Addison-Wesley, 2003.

[CC90] Chikofsky, E. and Cross, J. Reverse Engineering and Design Recovery: A Taxonomy. In IEEE Software, 1990.

[DDN02] Demeyer, S., Ducasse, S., and Nierstrasz, O. Object-Oriented Reengineering Patterns, Morgan Kaufmann Publishers, 2002.

[Ecl03] Object Technology International, Inc. Eclipse Platform Technical Overview, 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf

[EJB] Sun Microsystems. Enterprise JavaBeans. http://java.sun.com/products/ejb/docs.html

[FBB+99] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D. Refactoring: Improving the Design of Existing Programs. Addison-Wesley, 1999.

[Fow03] Fowler, M. Who needs an architect? In IEEE Software, 20(5): p. 11, 2003.

[GN95] Griswold, W. G., Notkin, D. Architectural Tradeoffs for a Meaning-Preserving Program Restructuring Tool. In IEEE Transactions of Software Engineering, 1995.

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

[JL91] Jacobson, I., and Lindström. F. Reengineering of old systems to an object-oriented architecture. In Proc. OOPSLA, Vol. 26, No. 11, 1991.

[JLL99] Jaktman, C. B., Leaney, J. and Liu, M. Structural Analysis of the Software Architecture - A Maintenance Assessment Case Study. In Proc. of WICSA1, 1999.

[JW99] Jackson, D. and Waingold, A. Lightweight Extraction of Object Models from Bytecode. Proc. ICSE 1999.

[KP88] Krasner, G.E. and Pope, S.T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. In JOOP 1(3), 1988.

[LH89] Lieberherr, K. and Holland, I. Assuring Good Style for Object-Oriented Programs. In IEEE Software, 1989.

[LLW03] Lieberherr, K., D.H. Lorenz, and P. Wu, A Case for Statically Executable Advice: Checking the Law of Demeter with AspectJ. In Proc.AOSD, 2003.

[LV95] Luckham, D.C., and Vera, J. An Event Based Architecture Definition Language. In IEEE TSE 21(6), 1995.

[MT00] Medvidovic, N., and Taylor, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. In IEEE TSE 26(1), 2000.

[Omo95] Omondo EclipseUML. http://www.omondo.com/

[PW92] Perry, D. E., and Wolf, A. L. Foundations for the Study of Architecture. ACM SIGSOFT Software Engineering Notes, 17(4), pp. 40–52, 1992.

[SP98] Stevens, P., Pooley, R. Systems Reengineering Patterns. In Proc FSE, 1998.