

# 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



With contributions by Jonathan Aldrich

Wesley Coelho

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



# Software loses structure over time

---

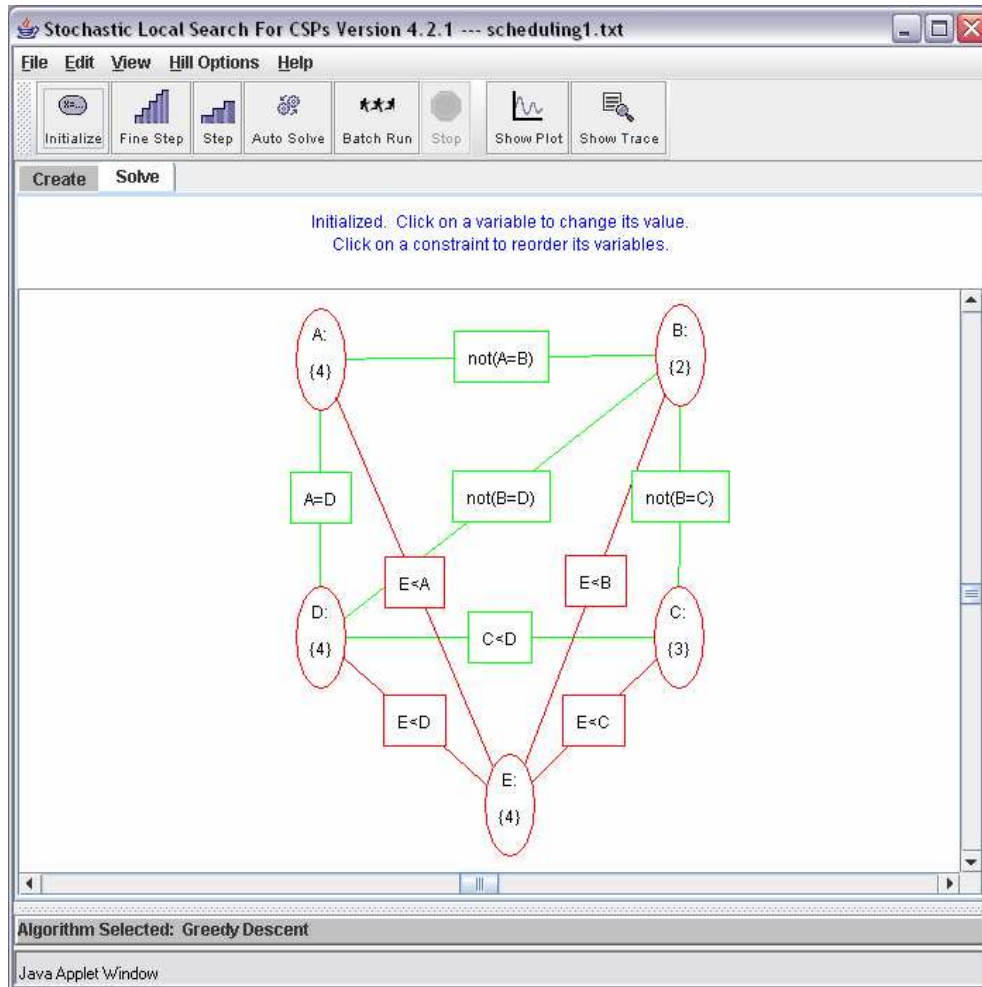
- External documentation often out of date
- Developers rarely consult external documentation (if it exists)
- Developers unaware of architectural intent not described in code
- Developers introduce violations
- Architectural violations lead to
  - Brittleness,
  - Inadaptability, ...

# The case study subject system

---

- No external documentation
- Developer turnover
- Junior developers
- Evolved over several years
- Violations of the architecture
  - Lack of coherence and clarity of form
- Representative of legacy applications
  - Without careful maintenance and evolution

# Case Study Application: HillClimber



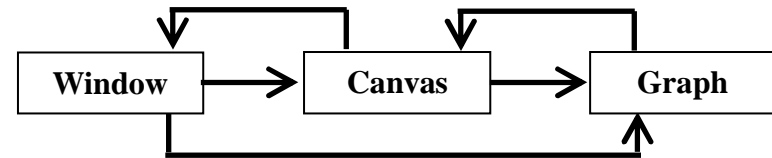
- Graphically demonstrates AI algorithms
- Created and maintained by undergraduate interns
- Over 16,000 lines of code in 80 classes
- Large enough to exhibit complex design issues

# HillClimber's architecture over time

---



## Shared Framework



## HillClimber: Today

- Part of a collection of applications using a shared framework
- Evidence of loss of structure, e.g.:
  - *Window* component requiring services from *Graph* component is a violation
  - Components provide services to or require services from most other components
  - Complex communication patterns

# Goal of the case study

---

- One-time refactoring is temporary
- Loss of architectural information is a key contributor to architectural violations
- Address a root cause of violations
  - Specify the architecture in the code directly
- Re-engineer the implementation
  - To closely match idealized architecture
  - In a form that may prevent future architectural violations

# Specify architecture in code

---

- ArchJava
  - Extension of Java programming language
  - Backwards compatibility critical for re-engineering
- Architecture specified in code
  - Components, Ports, Connections, ...
- Enforces **communication integrity**
  - Two components can communicate only if they are connected in the architecture
- Extract as-implemented C&C view

# ArchJava Example

---

```
public component class HillEngine {
```

```
public component class HillEngine{
```

```
requires boolean isInline();  
... }
```

```
public port engine{
```

```
provides void setDelay(int dt);
```

```
provides int getDelay();
```

```
}
```

```
...  
}
```

```
connect engine, graph.engine;
```

```
public HillEngine() {
```

```
...
```

```
// Note: Do most of initialization in init()
```

```
}
```

```
public void init() {
```

```
...
```

```
}
```

```
public void step() {
```

```
...
```

```
if (!canvas.isInline() ) {
```

```
    window.setButtonsSolved(true);
```

```
...
```

```
}
```

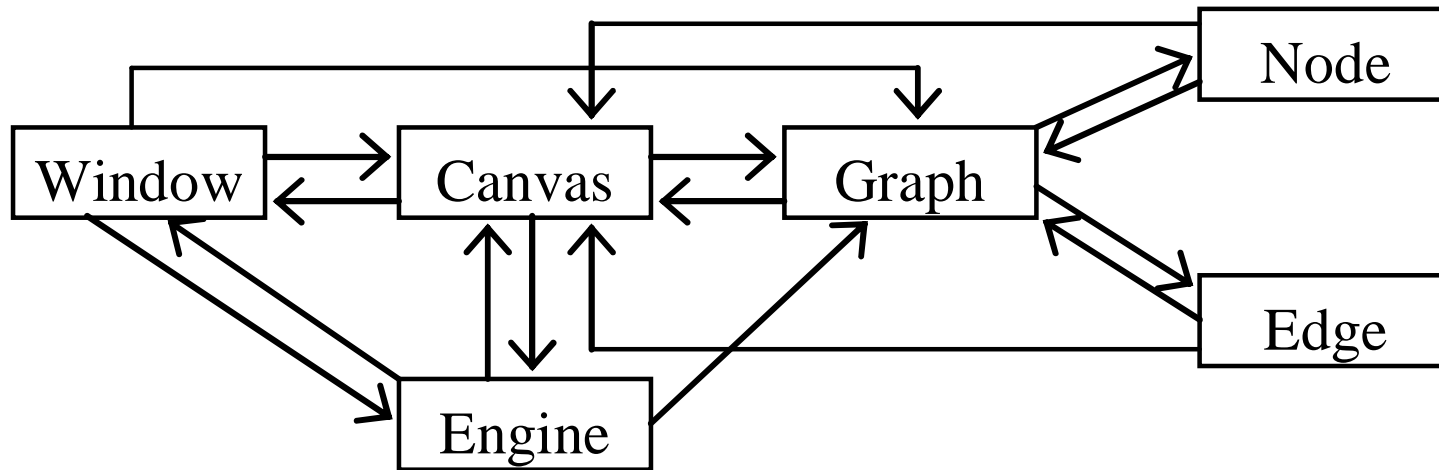
# Architecture-based re-engineering

---

1. Identify the source architecture
2. Identify the target architecture
3. Refactor the original program (in Java)
4. Re-engineer to ArchJava
5. Check against the target architecture

# Step 1: Identify the source architecture

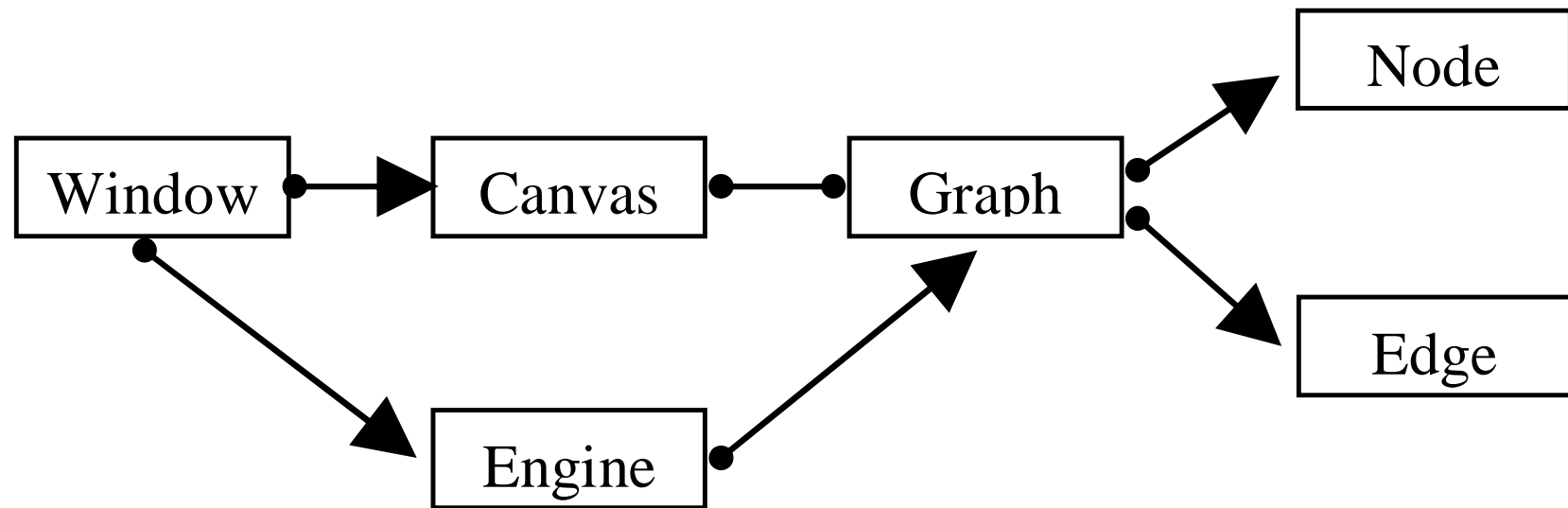
---



- Extracted using manual call-graph analysis
- Changes to one component affect several components
- Communication between components follows a nearly arbitrary pattern

## Step 2: Identify the target architecture

---



- Uses same architectural decomposition to avoid significant rework
- Less complex communication patterns
  - Graph and Engine independent of UI

## Step 3: Refactor the original program

---

- Avoid enforcing degraded architecture
- Re-engineer incrementally, while maintaining a running system
- Prepare the system for re-engineering
  - ArchJava imposes constraints on implementation
  - Limited tool support for refactoring once in ArchJava
- May involve significant restructuring if current implementation does not match the target architecture well

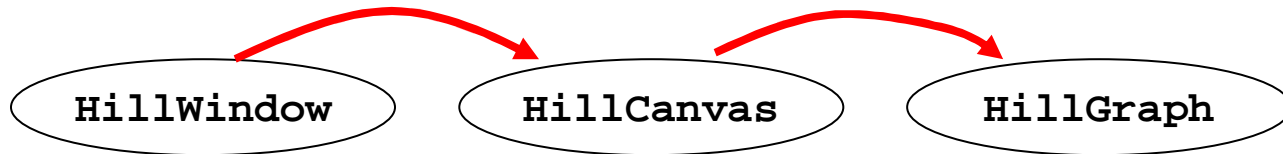
# Refactoring Examples

---

- Rename identifiers for comprehension
  - ArchJava code  $\equiv$  architectural documentation
- Encapsulate fields
  - Required for communication integrity
- Remove “navigation code”
  - Code that traverses a series of object links before calling a method on the final object

# Example: Remove Navigation Code

---



```
getCanvas().getGraph().setLineWidth(...)
```

- Symptom of misplaced behavior that
  - "Do not talk to strangers" – Law of Demeter
- Illegal in ArchJava because it involves passing component references

# Example: Refactoring HillEngine

---

```

public class HillEngine {
    public HillCanvas canvas;

    public int delayTime = 100;

    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 (true) {
            ((HillWindow) canvas.parent).setButtonsSolved(true);
        }
        ...
    }
}

```

```

private int delayTime = 100;

public int getDelayTime(){
    return delayTime;
}

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 HillWindow getWindow(){return window; }
public void setWindow(HillWindow window) {
    this.window = window; }

public void step(){
    if( !canvas.isInline() )
        window.setButtonsSolved(true);
    ...
}

```

Before: Original Java HillEngine class.

After: Refactored HillEngine Java class. 16

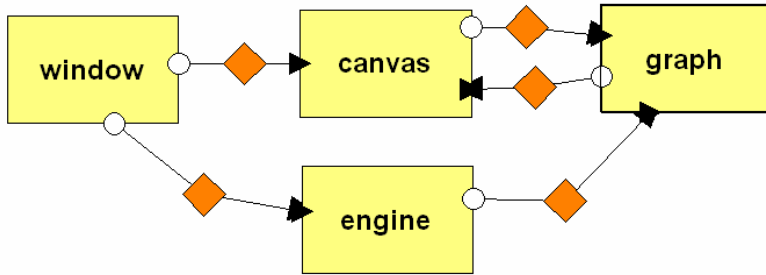
## Step 4: Re-engineer to ArchJava

---

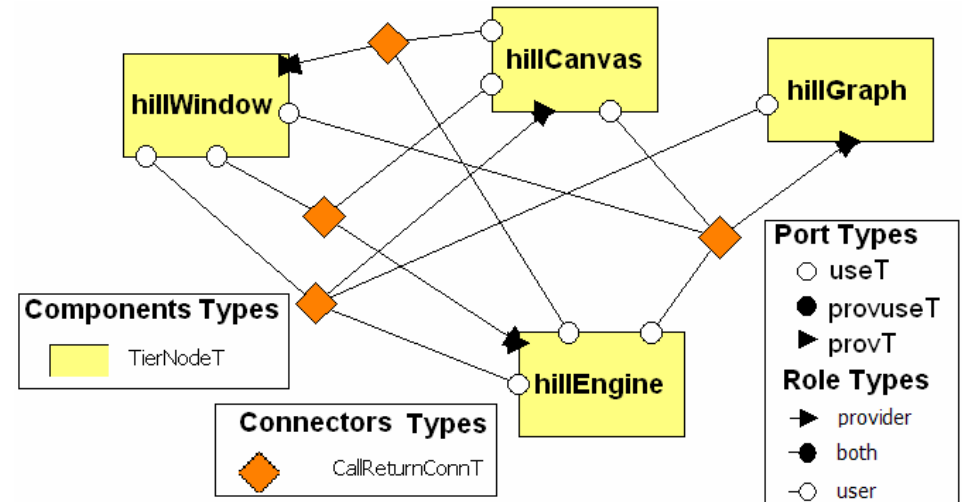
- Switch to ArchJava IDE
- Incrementally apply ArchJava constructs to describe and enforce architecture
  - Change class to component class
  - Convert instance variable to port
  - Change field link to connection
  - Convert program from reference-passing to establishing connections
    - Eliminate constructors or methods that take arguments of component type

# Step 5: Check against target architecture

---



**Target architecture**



**Automatically recovered intermediate architecture**

- Automatically generated C&C snapshots
  - Visualize object sharing issues
  - Expose “unwanted” control flow
  - Visualize disconnected ports (statically)

# Re-engineering result

---

- ArchJava effective in documenting and enforcing architectural structure
- Re-engineering effort involved
  - Approximately 2-3 days
- Learned several lessons
  - Hints for tool builders and language designers
  - Tradeoffs when using ArchJava

# Hints for Tool Builders

---

- Keep it iterative
  - Inability to go back to pure Java and perform more refactoring was limiting
  - E.g., could not go back to pure Java to refactor missed unencapsulated fields
- Keep it incremental
  - “Legacy mode”: a component class can extend a “regular” class
  - Can add ports to non-component classes
  - Useful for maintaining a running version
  - Turning a class into a component class can suddenly generate many ArchJava compile errors

# Hints for Tool Builders (Continued)

---

- Tolerate incompleteness
  - Required and provided port functionality must be completely specified
  - Temporarily disable checks for required/provided functionality to help maintain a running system
- Automate as much as possible
  - Many re-engineering tasks we performed could be automated
    - “Convert to component class” refactoring
  - Many of the tools we used were not satisfactory
  - Tools critical for dealing with larger code bases

# Tradeoffs when using ArchJava

---

- Fundamental language design issue
  - Object-oriented programming and design patterns rely heavily on passing references
- Static checking vs. runtime checking
  - Runtime exceptions possible
  - Extensive testing of re-engineered program
- What we could not express
  - Specify port uni-directionality
  - Relax communication integrity
  - Full communication integrity
- Missing debugging and refactoring support

# Case Study Limitations

---

- Need further study to answer
  - Is re-engineered program easier to understand and evolve?
  - Does documenting architectural intent help maintainers avoid architectural violations?
  - Will maintainers encounter expressiveness limitations of ArchJava?
- Current level of tool support for ArchJava limits its use in production

# Related Work

---

| <b>Characteristic</b>       | <b>This Case Study</b> | <b>Previous Case Studies</b> | <b>Legacy Applications</b> |
|-----------------------------|------------------------|------------------------------|----------------------------|
| <b>Programmers</b>          | Many                   | One                          | Many                       |
| <b>Programmer Expertise</b> | Novice                 | Expert                       | Junior                     |
| <b>Turnover</b>             | High                   | Low                          | High                       |
| <b>Lines of code</b>        | 16 K                   | ~ 8 K                        | 10 K – 10+ M               |
| <b>Platform</b>             | Java                   | Java                         | Many                       |
| <b>Middleware</b>           | None                   | None                         | Often                      |

## In summary

---

- Showed that a legacy system can be incrementally re-engineered to
  - Improve its structure
  - Document and enforce its architecture
- Presented a process for architecture-based re-engineering
- Provided hints for tool builders and language designers
- Already lead to improvements in ArchJava

# Questions?

---