Reverse Engineering of Design Patterns for High Performance Computing

Nija Shi
shini@cs.ucdavis.edu

Ron Olsson
olsson@cs.ucdavis.edu

Department of Computer Science
University of California, Davis
Outline

• Introduction
• Current Approaches
• A Motivating Example
• Our Initial Prototype – PINOT
• Results
• Patterns for HPC
• Future Work
Introduction

• Why Reverse Engineering?
  – Legacy code
  – Program understanding
  – Insufficient documentation

• Current program understanding tools
  – Debugging tools
  – Program analysis tools:
    CSCOPE, SourceNavigator, SourceInsight, etc.

• Improve current program understanding tools
  – Bring program understanding to the design level
Introduction
# Representative Current Approaches

<table>
<thead>
<tr>
<th>Tools</th>
<th>Language</th>
<th>Techniques</th>
<th>Case Study</th>
<th>Patterns Identified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPOOL</td>
<td>C++</td>
<td>Database query</td>
<td>ET++</td>
<td>Template Method, Factory Method, Bridge</td>
</tr>
<tr>
<td>DP++</td>
<td>C++</td>
<td>Database query</td>
<td>DTK</td>
<td>Composite, Flyweight, Class Adapter</td>
</tr>
<tr>
<td>Vokac et al.</td>
<td>C++</td>
<td>Database query</td>
<td></td>
<td>Singleton, Template Method, Observer, Decorator</td>
</tr>
<tr>
<td>Antoniol et al.</td>
<td>C++</td>
<td>Software metric</td>
<td>Leda, libg++, socket,</td>
<td>Proxy, Adapter, Bridge</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>galib, groff, mec</td>
<td></td>
</tr>
<tr>
<td>SPQR</td>
<td>C++</td>
<td>Formal semantic</td>
<td></td>
<td>Decorator</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Star Office Calc, Writer.</td>
<td></td>
</tr>
<tr>
<td>FUJABA</td>
<td>Java</td>
<td>Fuzzy logic and Dynamic analysis</td>
<td>Java AWT</td>
<td>Bridge, Strategy, Composite</td>
</tr>
<tr>
<td>Heuzeroth et al.</td>
<td>Java</td>
<td>Dynamic analysis</td>
<td>Java Swing</td>
<td>Composite, Mediator, CoR, Visitor</td>
</tr>
<tr>
<td>KT</td>
<td>SmallTalk</td>
<td>Dynamic analysis</td>
<td></td>
<td>Composite, Visitor, Template Method</td>
</tr>
<tr>
<td>MAISA</td>
<td>OMT</td>
<td>Graph matching</td>
<td></td>
<td>Abstract Factory</td>
</tr>
</tbody>
</table>
Current Approaches

• Categorize into
  – Purely static approaches
  – Static and dynamic approaches
  – Other approaches, e.g., based on UML diagrams
Purely Static Approaches

• Method
  – Extract structural relationships (structural analysis)
  – For a pattern, check for certain structural properties

• Drawback
  – Relies only on structural relationships, which are not the only distinction between patterns
Static and Dynamic Approaches

• Method
  – Use structural analysis to narrow down search space
  – Use dynamic analysis to verify behavior

• Drawback
  – Requires good data coverage
  – Verifies program behavior but does not verify the intent
  – Complicates the task for detecting patterns for HPC
Other Approaches

• Method
  – Extract pattern instances at the design level
  – E.g., Using UML diagrams that contain both structural and behavioral diagrams

• Drawback
  – Depends on the availability of UML diagrams
  – Extraction of UML behavioral diagrams is still ongoing work by other researchers
A Motivating Example

Detecting the Singleton Pattern

- As found in FUJABA
- Common search criteria
  - private Singleton()
  - private static instance
  - public static getInstance()
- Problem
  - No behavioral analysis on getInstance()
- Solution?

```java
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == NULL)
            instance = new Singleton();
        return instance;
    }
}
```

Inaccurately recognized as Singleton
Pattern INference and recOvery Tool

- **PINOT**
  - A fully automated pattern detection tool
  - Designed to be faster and more accurate

- How PINOT works
Pattern INference and recOvery Tool

- **Implementation Overview**
  - A modification of Jikes (open source C++ Java compiler)
  - Analysis using Jikes abstract syntax tree (AST) and symbol tables
  - PINOT collects information of
    - Class/Interface hierarchies
    - Method invocations
    - Object creation
    - Read/Write access for member instances
  - Behavioral analysis in PINOT
    - Reaching assignment, used in Singleton Pattern
    - Extracting statecharts from method bodies, used in Flyweight Pattern
    - Analyzing execution paths, used in Chain of Responsibility Pattern (CoR)
  - PINOT considers related patterns
    - E.g., Strategy and State Patterns, Composite and CoR, etc.
    - Speed up the process of pattern recognition
Recognition of Singleton by PINOT

• Structural aspect
  – private Singleton()
  – private static instance
  – public static getInstance()

• Behavioral aspect
  – Analyze the behavior in getInstance()
    • Slice the method body for instance and analyze the sliced program
    • Check if instance is returned
    • Simulate the sliced program and check for write access on instance
    • Loops are not considered
Recognition of Chain of Responsibility (CoR) by PINOT

• Structural aspect

Diagram taken from http://www.dofactory.com/Patterns/PatternChain.aspx
Recognition of CoR by PINOT

• Behavioral aspect
  – Identify a `HandleRequest()` in `Handler`
  • In `HandleRequest()`, check for delegation to `successor.HandleRequest()`
Recognition of Flyweight by PINOT

- Structural aspect

Diagram taken from http://www.dofactory.com/Patterns/PatternChain.aspx
Recognition of Flyweight by PINOT

• Behavioral aspect
  – Identify FlyweightFactory
    • Identify the flyweight pool instance flyweights
      (check for a collection object)
    • Slice GetFlyweight(key) for flyweights
    • Translate the sliced program to a statechart
    • Verify the statechart
# Results on Patterns Detected

<table>
<thead>
<tr>
<th>Package</th>
<th>JavaAWT</th>
<th>JHotDraw</th>
<th>Apache Ant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singleton</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CoR</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bridge</td>
<td>2</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>Strategy</td>
<td>11</td>
<td>36</td>
<td>8</td>
</tr>
<tr>
<td>State</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Composite</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Mediator</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Flyweight</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Results on Execution Time

<table>
<thead>
<tr>
<th>Package</th>
<th>Number of classes</th>
<th>KLOC</th>
<th>Total Files</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaAWT</td>
<td>485</td>
<td>142.8</td>
<td>345</td>
<td>5.05</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>464</td>
<td>71.7</td>
<td>484</td>
<td>6.23</td>
</tr>
<tr>
<td>Apache Ant</td>
<td>526</td>
<td>72.4</td>
<td>232</td>
<td>3.64</td>
</tr>
</tbody>
</table>

PINOT ran this on a 2.8 GHz machine.

FUJABA took 22 minutes on a Pentium III 933MHz machine.
## Results on Java AWT

<table>
<thead>
<tr>
<th>Pattern</th>
<th>PINOT</th>
<th>FUJABA</th>
<th>Pattern Stories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singleton</td>
<td>Toolkit, GraphicsEnvironment, ColorModel</td>
<td></td>
<td>Toolkit</td>
</tr>
<tr>
<td>CoR</td>
<td>Component and Container</td>
<td></td>
<td>Component and Container</td>
</tr>
<tr>
<td>Bridge</td>
<td>Component and ComponentPeer, MenuComponent, MenuComponentPeer</td>
<td>Component and ComponentPeer</td>
<td>Component and ComponentPeer</td>
</tr>
<tr>
<td>Strategy</td>
<td>Container and LayoutManager and 10 other instances</td>
<td>Component and ComponentPeer, Container and LayoutManager</td>
<td>Container and LayoutManager</td>
</tr>
<tr>
<td>Composite</td>
<td>Component and Container</td>
<td>Component and Container</td>
<td>Component and Container</td>
</tr>
<tr>
<td>Mediator</td>
<td>Component and Container, MediaTracker, MediaEntry</td>
<td></td>
<td>Component and Container</td>
</tr>
</tbody>
</table>
Double-checked Locking

- Double-checked Locking Pattern (DCL)
  - Designed for optimization
  - Used with Multi-threaded Singleton pattern
  - PINOT checks whether DCL is applied when a Singleton pattern is detected

```java
public class Singleton {
    private Singleton()
    private static Singleton instance;

    public static Singleton getInstance()
    {
        if (instance == null)
        {
            synchronized(Singleton.class)
            {
                instance = new Singleton();
            }
        }
        return instance;
    }
}
```
• **java.awt.Toolkit**
  – Java AWT is a Multi-threaded GUI toolkit
  – **Toolkit** is declared abstract
  – Each JDK has one subclass of **Toolkit**
    – `sun.awt.motif.MToolkit` subclasses **Toolkit**, provided for Solaris/Linux
  – `sun.awt.motif.MToolkit` is created at the first call to `getDefaultToolkit()`
  – `getDefaultToolkit()`
    • takes the role of `getInstance()` in the Singleton pattern
    • is synchronized at declaration, does not use the DCL
    • is invoked
      – directly by users:
        » `Toolkit.getDefaultToolkit().beep();`
      – indirectly by multiple AWT threads: for displaying
        » `Component.addNotify(); // by Container`
  – DCL can benefit Java AWT applications
Patterns for HPC

• Patterns for HPC
  – Patterns for Concurrent and Networked Objects
    By Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann
  – Patterns for Parallel Programming
    By Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill
Patterns for HPC

• Properties
  – Structural
    • Defined in class declarations
    • Syntax-based analysis
    • E.g., Monitor Object Pattern, Thread-safe Interface
  – Behavioral
    • Embedded in method bodies
    • Semantic-based analysis
    • E.g., Reactor, DCL, Leader/Followers Patterns
Patterns for HPC

• Semantic-based analysis
  – Techniques to analyze behavior
    • Recognize the Java concurrent programming model
      – Locking, synchronization, communication mechanisms
      – The `java.util.concurrent` package in Java 1.5
      – Patterns implemented at the syntax level:
        » Scoped-Locking Pattern is provided in `synchronized`
        » Thread-safe Interface is provided in `synchronized` and `java.util.concurrent.locks.ReentrantLock`

• Program slicing
• Static analysis
• Extraction of finite-state machines

– Consider related GoF patterns
  • E.g., Decorator and Thread-safe Wrapper Facade patterns in `java.util.Collections.synchronizedMap()`
Future Work

• Formalize our approach
• Complete the 23 GoF patterns
• Add common concurrent and parallel patterns into PINOT
• Provide a language for users to describe patterns (e.g., UML diagrams)
• Train PINOT to learn new patterns or recognize more implementation variations