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

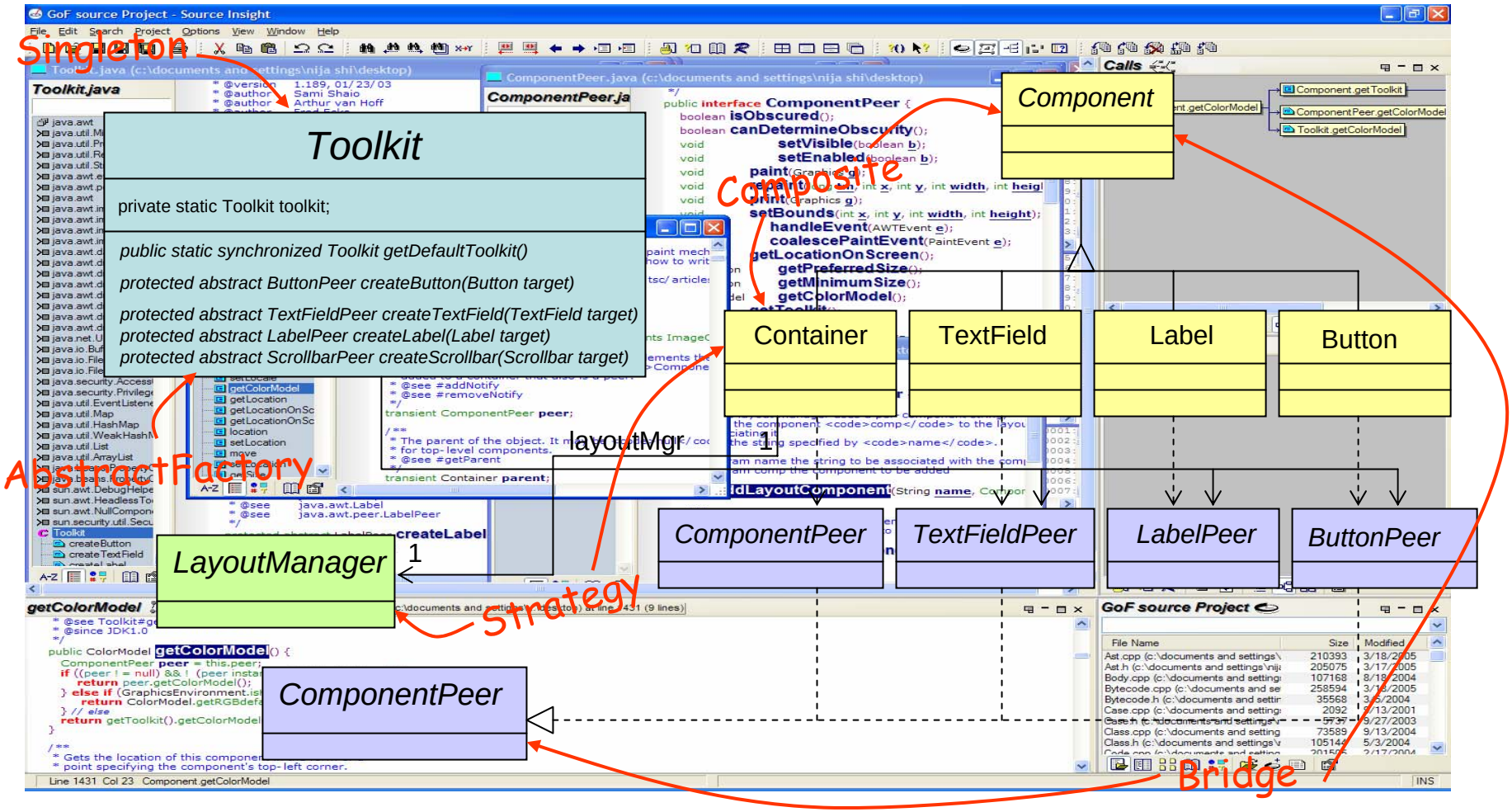
- Introduction
- Current Approaches
- A Motivating Example
- Our Initial Prototype – PINOT
- Results
- Patterns for HPC
- Future Work

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

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

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

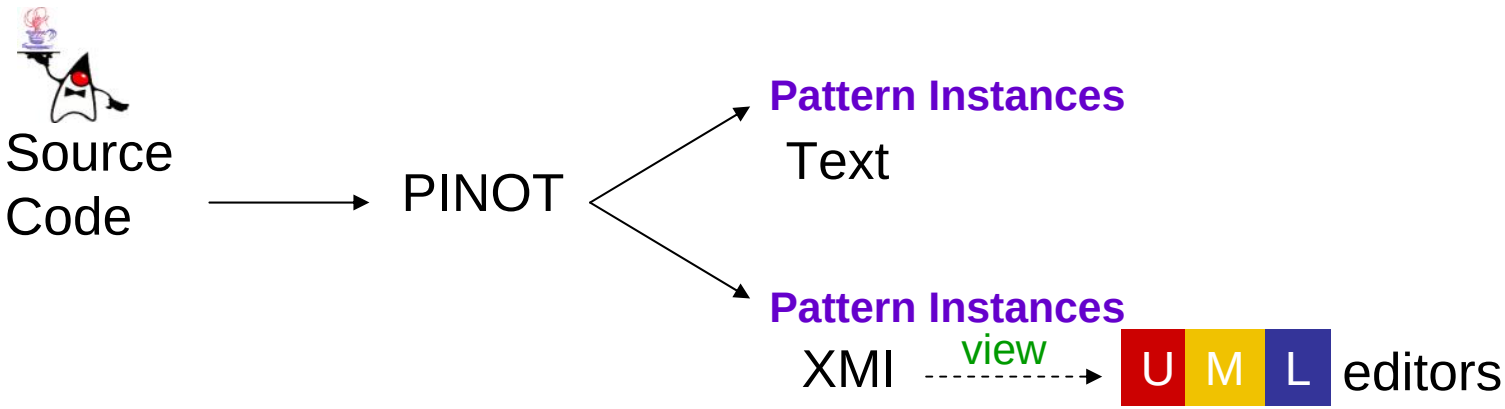
- As found in FUJABA
- Common search criteria
  - `private Singleton()`
  - `private static instance`
  - `public static getInstance()`
- Problem
  - No behavioral analysis on `getInstance()`
- Solution?

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

recall

- 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

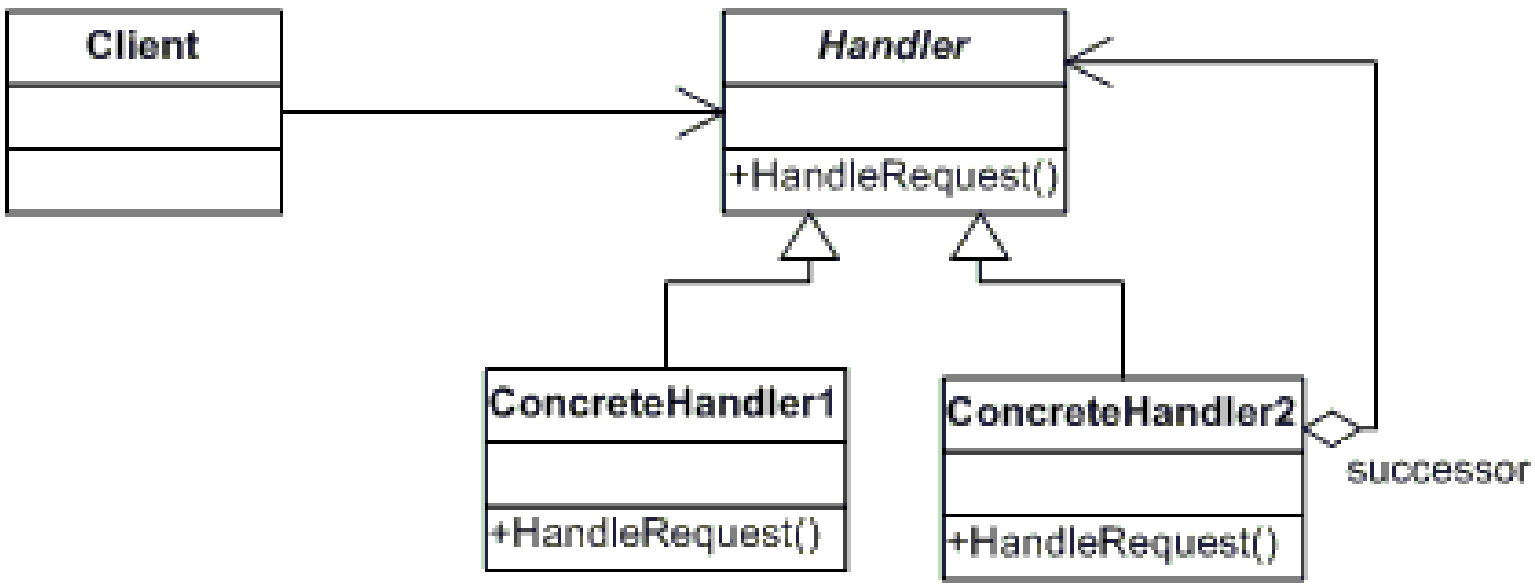
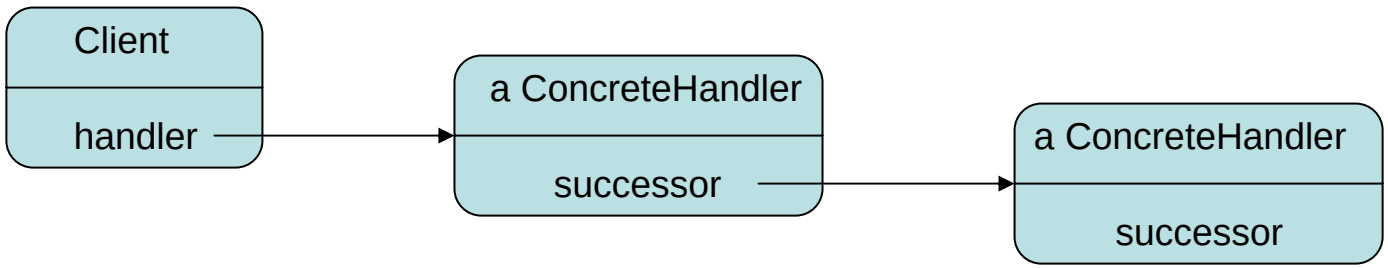


Diagram taken from http://www.dofactory.com/Patterns/PatternChain.aspx

Recognition of CoR by PINOT

- Behavioral aspect
  - Identify a `HandleRequest()` in `Handler`
    - In `HandleRequest()`, check for delegation to `sucessor.HandleRequest()`



Recognition of Flyweight by PINOT

- Structural aspect

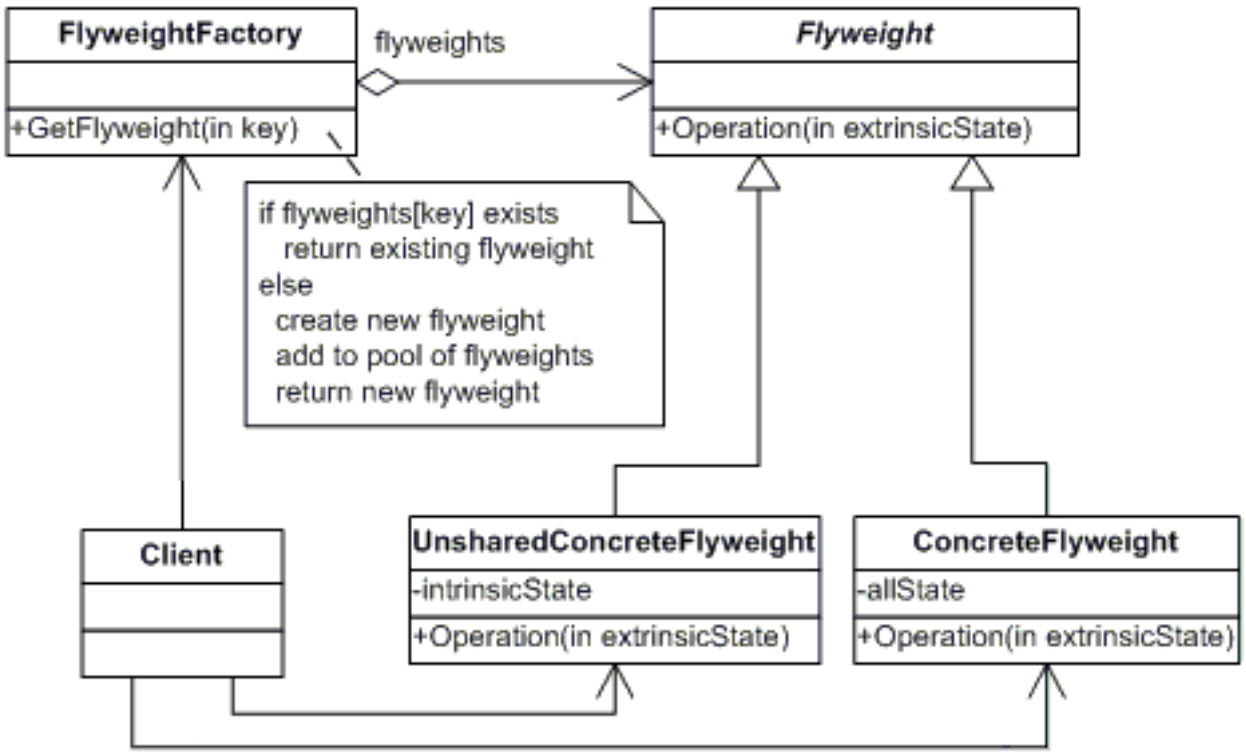


Diagram taken from http://www.dofactory.com/Patterns/PatternChain.aspx

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

	Packages		
	JavaAWT	JHotDraw	Apache Ant
Singleton	3	1	1
CoR	1	0	0
Bridge	2	24	2
Strategy	11	36	8
State	0	2	0
Composite	1	4	3
Mediator	2	0	0
Flyweight	0	0	2

Results on Execution Time

Package	Number of classes	KLOC	Total Files	Time (sec)
JavaAWT	485	142.8	345	5.05
JHotDraw	464	71.7	484	6.23
Apache Ant	526	72.4	232	3.64

PINOT ran this on a 2.8 GHz machine

FUJABA took 22 minutes on a Pentium III 933MHz machine

Results on Java AWT

	PINOT	FUJABA	Pattern Stories
Singleton	Toolkit GraphicsEnvironment ColorModel		Toolkit
CoR	Component and Container		Component and Container
Bridge	Component and ComponentPeer MenuComponent and MenuComponentPeer	Component and ComponentPeer	Component and ComponentPeer
Strategy	Container and LayoutManager and 10 other instances	Component and ComponentPeer Container and LayoutManager	Container and LayoutManager
Composite	Component and Container	Component and Container	Component and Container
Mediator	Component and Container MediaTracker and MediaEntry		Component and Container

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

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

  public static Singleton getInstance()
  {
    if (instance == null)
    {
      synchronized(Singleton.class)
      {
    }   if (instance == null)
    return instance;   instance = new Singleton();
  }   }
}
```

DCL

- 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