Design Pattern Detection
A design pattern systematically names, explains and evaluates an important and recurring design problem and its solution.

Good designers know not to solve every problem from first principles.
  - They reuse solutions.

This is very different from code reuse.
Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context
Essential Elements of a Design Pattern

- **Name**
  - Naming a pattern increases our design vocabulary

- **Problem**
  - When to apply the pattern

- **Solution**
  - Elements that make up the design, their relationships, responsibilities, and collaborations

- **Consequences**
  - Results and trade-offs of applying the pattern
How Design Patterns Solve Design Problems

• Finding appropriate objects
• Determining object granularity
• Specifying object interfaces
• Specifying object implementations
• Putting reuse mechanisms to work
  • Inheritance vs. Composition
  • Delegation
• Designing for change
Pattern Benefits

- Enable large scale reuse of software architectures
- Explicitly capture expert knowledge and design trade-offs
- Help improve developer communication
- Help ease the transition to OO methods
Pattern Drawbacks

- Patterns do not lead to direct code reuse
- Patterns are often deceptively simple
- You may suffer from pattern overload
- Patterns must be validated by experience and debate rather than automated testing
- Integrating patterns into a process is human intensive rather than a technical activity
Pattern Description Template

- Name
- Intent
  - What does the pattern do? What problems does it address?
- Motivation
  - A scenario of pattern applicability
- Applicability
  - In which situations can this pattern be applied
- Participants
  - Describe participating classes/objects
Collaborations
  - How do the participants carry out their responsibilities?

Diagram
  - Graphical representation of the pattern

Consequences
  - How does the pattern support its objectives?

Implementation
  - Pitfalls, language specific issues

Examples
Classification

- **Structural**
  - Deal with decoupling interface and implementation of classes and objects

- **Behavioural**
  - Deal with dynamic interaction among collections of classes and objects

- **Creational**
  - Deal with initializing and configuring collections of classes and objects
Detecting design patterns

- A difficult task
- Patterns are primarily a literary form
- No rigorous mathematical definitions
- Automatic detection beyond the state of the art of Artificial Intelligence
- Instead, detect the artifacts of implementing the solution of the design pattern
- Purely structural patterns are easier to detect
- Purely behavioural patterns are much harder
- Most patterns are somewhere in the middle
A template solution needs to be both

**Distinctive**
- The static structure is not likely to be represented in a design that does not use the pattern

**Unambiguous**
- Can only be done in one way (or in a small number of variants)

- An object adapter is unambiguous but not distinctive
Object Adapter Static Structure

Client

Target
request()

Adapter
request()

Adaptee
specificRequest()

Client

Adapter

Adaptee

Client

Target

Adapter

Adaptee

Client

Target

Adapter

Adaptee

Client

Target

Adapter

Adaptee
A Decorator is sometimes referred to as a degenerate Composite.

The static structure of the two patterns is very similar.

The dynamic behaviour is also the same.

Static difference: A Composite contains a collection of Components, while a Decorator contains only one.

Intent difference: The Composite pattern groups components into a whole. The Decorator patterns enhances the responsibility of a component.
Both patterns allow flexible choice from a set of alternatives

In their simple variants, the static structure and the dynamic behaviour are exactly the same

The difference: Choosing a particular behaviour (State) vs. choosing a particular algorithm (Strategy)
Both static and dynamic analysis are necessary in order to detect patterns

Static analysis
- The static structure of the pattern has to match a subgraph of the static structure of the software system

Dynamic analysis
- Message passing during run-time has to match the message flow that implements the behaviour of the pattern
Design Pattern Instances

- Each design pattern has a fixed set of roles, e.g. in the Adapter pattern, there is a Client, a Target, an Adapter, and an Adaptee
- Every detection technique attempts to discover instances of the design pattern in the software system being examined
- A design pattern instance is a set of classes that match the roles
Design Pattern Detection Research Issues

- False positive elimination
  - The precision of most published approaches is quite poor, often below 50%

- Dealing with Variants
  - Patterns are conceptual. Their implementation may vary considerably depending on the specific context

- Counting instances
  - Different detection approaches do it differently
Detecting Design Patterns in Java software with PDE

- PDE is a tool that collects static and dynamic facts from a system written in Java and detects design patterns in it.
- It will be installed on indigo by the end of the month.
- A possible course project is to apply PDE to an open source system and evaluate the results.
Every pattern has a static definition, e.g.
uses client target
inherits adapter target
uses adapter adaptee

Javex and grok are used to extract static facts such as
uses ClassA ClassB
inherits ClassC ClassB
uses ClassC ClassD

QL matches the static definition to the static facts
Every pattern has a dynamic definition in XML

```xml
<entry className="adapter"
calledByClass="client"
thisObject="object1"
nextCallInSubtree="yes">
  <entry className="adaptee"
calledByClass="adapter"
calledByObject="object1"
thisObject="object2">
  </entry>
</entry>
```
Probekit is used to collect dynamic facts such as:

```
<entry
calledByClass="ContactAdapter"
calledByMethod="setTitle"
calledByObject="ContactAdapter@145"
className="ChovnatlhImpl"
methodName="cherPatlh"
thisObject="ChovnatlhImpl@110">
```

If the dynamic facts do not match the dynamic definition the candidate instance is deemed a false positive.
Results with sample pattern implementations

- PDE detects 22/23 patterns
  - Except Facade, all patterns are detected
  - Facade is more an architectural design pattern

- PINOT detects 17/23
  - Pattern definitions are hard coded

- FUJABA detects 14/23
  - Behavioral patterns hard to detect

<table>
<thead>
<tr>
<th>Pattern</th>
<th>PDE</th>
<th>PINOT</th>
<th>FUJABA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creational</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Builder</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Factory Method</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Prototype</td>
<td>✓</td>
<td>✓</td>
<td>#</td>
</tr>
<tr>
<td>Singleton</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Structural</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adapter</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bridge</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Composite</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Decorator</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Facade</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Flyweight</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Proxy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Behavioral</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chain of Resp.</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Command</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Interpreter</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Iterator</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Mediator</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Memento</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Observer</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>State</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Strategy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Template Method</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Visitor</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Sum</td>
<td>22/23</td>
<td>17/23</td>
<td>14/23</td>
</tr>
<tr>
<td>Method</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factory Method</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Adapter</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bridge</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Builder</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chain of Responsibility</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Composite</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decorator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flyweight</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Interpreter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mediator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memento</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prototype</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proxy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Singleton</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Strategy</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Template Method</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Visitor</td>
<td>3</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>