An Extensible Framework to Discover Design Pattern

Cong Liu, Boudewijn van Dongen, Nour Assy, Wil van der Aalst
c.liu.3@tue.nl
March 16, 2017
Outline

• Background

• Definition of design pattern

• Existing design pattern discover techniques

• Our framework
The use of design patterns leads to the construction of well-formed, maintainable and reusable software systems.

Discovering design patterns in software systems
- help to understand the original design decision
- support software evolution and change easily.
What is design pattern

• A design pattern abstracts a reusable object-oriented design that solves a common recurring design problem in a particular context. (GoF, 1995)

• It describes the roles, responsibilities, and collaborations of participating classes, methods and objects.

<table>
<thead>
<tr>
<th>Catalog</th>
<th>Number</th>
<th>Design Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creational Patterns</td>
<td>5</td>
<td>Abstract Factory, Factory Method, Builder, Prototype, Singleton</td>
</tr>
<tr>
<td>Structural Patterns</td>
<td>7</td>
<td>Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy</td>
</tr>
<tr>
<td>Behavioral Patterns</td>
<td>11</td>
<td>Observer, Command, Chain of responsibility, Interpreter, Iterator, Visitor, Strategy, Mediator, Template Method, State, Memento</td>
</tr>
</tbody>
</table>
Observer pattern

one-to-many dependency between objects so that when one object changes state, all its dependents are automatically notified.

OP=(Subject, Observer, notify, update, register, unregister)
Observer pattern specification

An observer pattern is specified as \((Rset^o, Cons^o)\)

(1) \(Rset=\{\text{Subject, Observer, notify, update, register, unregister}\}\) is the set of roles; and

(2) \(Cons\) is the set of constraints defined on the roles.

\[
\begin{align*}
notify & \in M(Subject) \\
update & \in M(Observer) \\
\text{register, unregister} & \in M(Subject) \\
\text{Observer} & \notin P(\text{notify}) \\
\text{Observer} & \in P(\text{register}) \\
\text{Observer} & \in P(\text{unregister}) \\
\text{update} & \in I(\text{notify}) \\
\end{align*}
\]

\[
\begin{align*}
T(\text{register}, i) & < T(\text{unregister}, i) \\
T(\text{register}, i) & < T(\text{notify}, i) \\
I^*(\text{notify}) & \Rightarrow (\text{update}) \\
\end{align*}
\]

Depends on runtime invocation
Observer pattern instance

```
public class Member{
    public void receive(){}
}
```

```
public class MailingServer{
    public List<Member> members;
    public void addMember(Member member){
        members.add(member);
    }
    public void removeMember(Member member){
        members.remove(member);
    }
    public void notifyMembers(){
        for(Member member : members){
            member.receive();
        }
    }
}
```

<table>
<thead>
<tr>
<th>Roles</th>
<th>Code elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>MailingServer</td>
</tr>
<tr>
<td>Observer</td>
<td>Member</td>
</tr>
<tr>
<td>notify</td>
<td>notifyMembers</td>
</tr>
<tr>
<td>update</td>
<td>receive</td>
</tr>
<tr>
<td>register</td>
<td>addMember</td>
</tr>
<tr>
<td>unregister</td>
<td>removeMember</td>
</tr>
</tbody>
</table>

$notify \in M(Subject)$

$T(register, i) < T(unregister, i)$
Observer pattern instance invocation

- During the execution, the observer pattern instance can be invoked multiple times.

```java
public static void main(String []args) {
    MailingServer msA = new MailingServer();
    Member a= new Member();
    Member b= new Member();
    msA.addMember(a);
    msA.addMember(b);
    msA.notifyMembers();
    msA.removeMember(a);
    msA.removeMember(b);
    MailingServer msB = new MailingServer();
    Member m= new Member();
    msB.addMember(m);
    msB.notifyMembers();
    msA.removeMember(m);
}
```
### Execution data (method-level)

```java
public static void main(String[] args) {
    MailingServer msA = new MailingServer();
    Member a = new Member();
    Member b = new Member();
    msA.addMember(a);
    msA.addMember(b);
    msA.notifyMembers();
    msA.removeMember(a);
    msA.removeMember(b);

    MailingServer msB = new MailingServer();
    Member m = new Member();
    msB.addMember(m);
    msB.notifyMembers();
    msB.removeMember(m);
}
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Method</th>
<th>Class</th>
<th>Obj</th>
<th>C. M</th>
<th>C. C</th>
<th>C. O</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>addMember</td>
<td>MailingServer</td>
<td>1</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e2</td>
<td>addMember</td>
<td>MailingServer</td>
<td>1</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e3</td>
<td>notifyMember</td>
<td>MailingServer</td>
<td>1</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e4</td>
<td>receive</td>
<td>Member</td>
<td>2</td>
<td>notifyMember</td>
<td>MailingServer</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>e5</td>
<td>receive</td>
<td>Member</td>
<td>3</td>
<td>notifyMember</td>
<td>MailingServer</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>e6</td>
<td>removeMember</td>
<td>MailingServer</td>
<td>2</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e7</td>
<td>removeMember</td>
<td>MailingServer</td>
<td>2</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e8</td>
<td>addMember</td>
<td>MailingServer</td>
<td>4</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e9</td>
<td>notifyMember</td>
<td>MailingServer</td>
<td>4</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e10</td>
<td>updateS</td>
<td>Member</td>
<td>5</td>
<td>notifyMember</td>
<td>MailingServer</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>e11</td>
<td>removeMember</td>
<td>MailingServer</td>
<td>4</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pattern, instance, invocation

- A **design pattern** ➞ a tuple of roles and constraints

- A **design pattern instance** ➞ a tuple of participating classes and methods each acting a certain role, i.e., one implementation of the design pattern.

- A **design pattern instance invocation** ➞ an independent execution of the pattern instance.
Outline

• Background

• Definition of design pattern

• Existing design pattern discover techniques

• Our framework
Design pattern discovery/detection

- **Static analysis:**
  - Source code
  - Static Analysis
  - Candidate Pattern Instances

- **Dynamic analysis:**
  - Execution Data
  - Dynamic Analysis
  - Candidate Pattern Instances
Static analysis example

Source code

<table>
<thead>
<tr>
<th>Role</th>
<th>Code elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>MailingServer</td>
</tr>
<tr>
<td>Observer</td>
<td>Member</td>
</tr>
<tr>
<td>notify</td>
<td>notifyMembers</td>
</tr>
<tr>
<td>update</td>
<td>--</td>
</tr>
<tr>
<td>register</td>
<td>--</td>
</tr>
<tr>
<td>unregister</td>
<td>--</td>
</tr>
</tbody>
</table>

DPD

Missing roles
Hybrid analysis

Unable to extend to new patterns as constraints are hard-coded in the program.

interleaved invocations of pattern instance.
Limitations

• Hard to extend to new design patterns, as the constraints are hard-coded in the program.

• Some constraints are defined on invocation level, e.g., $T(\text{register},i) < T(\text{unregister},i)$. No explicit support for invocation identification before checking.

• Existing tools return incomplete pattern instance, manual efforts are required to understand.
Outline

• Background

• Definition of design pattern

• Design pattern discover techniques

• Our framework
Aim of framework

• To provide flexible support for new patterns.

• To perform accurate constraint checking, i.e., supporting invocation identification before checking.

• To discover complete pattern instance, i.e., the missing roles.
Extensibility

Candidate Pattern Instances

Dynamic Analysis

Execution Data

How to support pattern X?

Pattern instances
Extensibility

Constraints of new pattern X.

Candidate Pattern Instances

Dynamic Analysis

Execution Data

Pattern instances
Invocation identification

• An invocation represents one run of the pattern instance and one software execution may trigger multiple invocations.

• For oo software, an observer invocation involves one subject object and its registered observer objects.

• For each execution, we build a graph for each subject object and its corresponding observer objects.
  - Vertices: class objects
  - Arcs: calling relation among them.
## Invocation identification

<table>
<thead>
<tr>
<th>ID</th>
<th>Method</th>
<th>Class</th>
<th>Obj</th>
<th>C. M</th>
<th>C. C</th>
<th>C. O</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>addMember</td>
<td>MailingServer</td>
<td>1</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e2</td>
<td>addMember</td>
<td>MailingServer</td>
<td>1</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e3</td>
<td>notifyMember</td>
<td>MailingServer</td>
<td>1</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e4</td>
<td>receive</td>
<td>Member</td>
<td>2</td>
<td>notifyMember</td>
<td>MailingServer</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>e5</td>
<td>receive</td>
<td>Member</td>
<td>3</td>
<td>notifyMember</td>
<td>MailingServer</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>e6</td>
<td>removeMember</td>
<td>MailingServer</td>
<td>2</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e7</td>
<td>removeMember</td>
<td>MailingServer</td>
<td>2</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e8</td>
<td>addMember</td>
<td>MailingServer</td>
<td>4</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e9</td>
<td>notifyMember</td>
<td>MailingServer</td>
<td>4</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e10</td>
<td>updateS</td>
<td>Member</td>
<td>5</td>
<td>notifyMember</td>
<td>MailingServer</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>e11</td>
<td>removeMember</td>
<td>MailingServer</td>
<td>4</td>
<td>Main</td>
<td>Test</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Invocation m

1. Call to 2
2. Call to 3

### Invocation n

4. Call to 5
Discovery

Using the constraints, data, incomplete pattern instance to discover the missing roles.

<table>
<thead>
<tr>
<th>Role</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>MailingServer</td>
</tr>
<tr>
<td>Observer</td>
<td>Member</td>
</tr>
<tr>
<td>notify</td>
<td>notifyMember</td>
</tr>
<tr>
<td>update</td>
<td>--</td>
</tr>
<tr>
<td>register</td>
<td>--</td>
</tr>
<tr>
<td>unregister</td>
<td>--</td>
</tr>
</tbody>
</table>

\[
\text{update} \in M(Observer)
\]

\[
M(Observer) = \{\text{receive}\}
\]

\[
\text{update} \leftrightarrow \{\text{receive}\}
\]
Using the constraints, data and complete candidate pattern instance to remove false positives.

<table>
<thead>
<tr>
<th>Role name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>MailingServer</td>
</tr>
<tr>
<td>Observer</td>
<td>Member</td>
</tr>
<tr>
<td>notify</td>
<td>notifyMember</td>
</tr>
<tr>
<td>update</td>
<td>updateM</td>
</tr>
<tr>
<td>register</td>
<td>registerM</td>
</tr>
<tr>
<td>unregister</td>
<td>unregisterM</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{notify} & \in M(\text{Subject}) \\
\text{update} & \in M(\text{Observer}) \\
\text{register, unregister} & \in M(\text{Subject}) \\
\text{Observer} & \not\in P(\text{notify}) \\
\text{Observer} & \in P(\text{register}) \\
\text{Observer} & \in P(\text{unregister}) \\
\text{update} & \in I(\text{notify}) \\
\end{align*}
\]

\[
T(\text{register}) < T(\text{unregister}) \\
T(\text{register}) < T(\text{notify}) \\
I(\text{notify}) \Rightarrow (update)
\]

\[
\begin{align*}
\text{notify} & \in M(\text{Subject}) \\
\text{update} & \in M(\text{Observer}) \\
\text{register, unregister} & \in M(\text{Subject}) \\
\text{Observer} & \not\in P(\text{notify}) \\
\text{Observer} & \in P(\text{register}) \\
\text{Observer} & \in P(\text{unregister}) \\
\text{update} & \in I(\text{notify}) \\
\end{align*}
\]

\[
T(\text{register},i) < T(\text{unregister},i)
\]
Summary of the Framework

Source code

Static Analysis

Incomplete Pattern Instances

Pattern specification

Instrumentation

Invocation Identification

Discovery

Complete Pattern Instances

Checking

Execution Data

Invocation Data

Validated Pattern Instances
To do list...

Invocation identification techniques for other patterns?

How to setup evaluation to support the claims and motivations?

Over-engineering problem detection?
Discussion and Feedback!!!