# 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**

Technische Universiteit
**Eindhoven**
University of Technology

Where innovation starts

# Outline

- Background
- Definition of design pattern
- Existing design pattern discover techniques
- Our framework

# Background

The use of design patterns leads to the construction of well-formed, maintainable and reusable of software system.

Discovering design patterns in software systems

- help to understand the original design decision
- support software evolution and change easily.

# What is design pattern

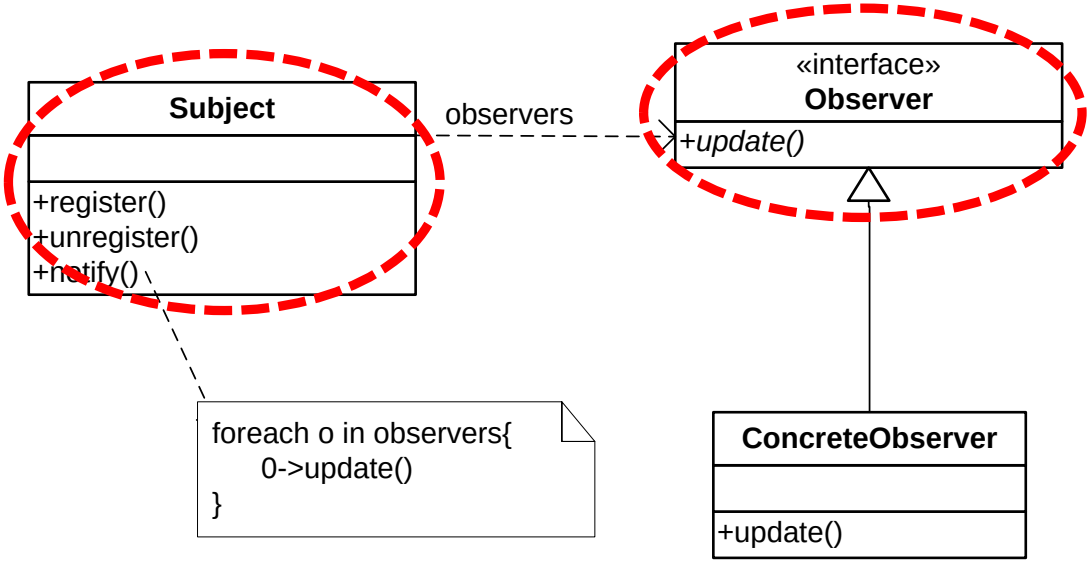
- A design pattern abstracts a reusable object-oriented design that solves a common recurring design problem in a particular context. (GoF, 1995)

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

| Catalog | Number | Design Patterns |
|---|---|---|
| Creational Patterns | 5 | Abstract Factory, Factory Method, Builder, Prototype, Singleton |
| Structural Patterns | 7 | Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy |
| Behavioral Patterns | 11 | ***Observer***, Command, Chain of responsibility, Interpreter, Iterator, Visitor, Strategy, Mediator, Template Method, State, Memento |

# 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={Subject, Observer, notify, update, register, unregister} is the set of roles; and

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

notify $\in$ **M**(Subject)

update $\in$ **M**(Observer)

register, unregister $\in$ **M**(Subject)

Observer $\notin$ **P**(notify)

Observer $\in$ **P**(register)

Observer $\in$ **P**(unregister)

update $\in$ **I**(notify)

**T**(register,i)<**T**(unregister,i)

**T**(register,i)<**T**(notify,i)

**I**$^*$(notify)➔(update)

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();}
    }
}
```

| Roles | Code elements |
|---|---|
| Subject | MailingServer |
| Observer | Member |
| notify | notifyMembers |
| update | receive |
| register | addMember |
| unregister | removeMember |

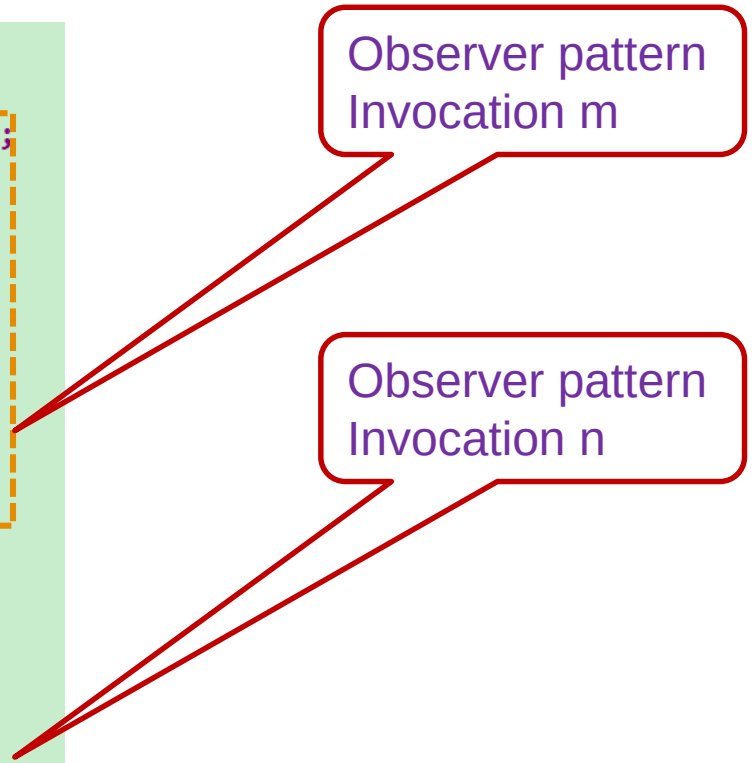
notify $\in$ **M**(Subject)

**T**(register, i)<**T**(unregister, i)

# Observer pattern instance invocation

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

```
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)

```
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 Mai
   Member m= new Member();

   msB.addMember(m);

   msB.notifyMembers();

   msA.removeMember(m);
}
```

| ID | Method | Class | Obj | C. M | C. C | C. O | T |
|---|---|---|---|---|---|---|---|
| $e_1$ | addMember | MailingServer | 1 | Main | Test | | |
| $e_2$ | addMember | MailingServer | 1 | Main | Test | | |
| $e_3$ | notifyMember | MailingServer | 1 | Main | Test | | |
| $e_4$ | receive | Member | 2 | notifyMember | MailingServer | 1 | |
| $e_5$ | receive | Member | 3 | notifyMember | MailingServer | 1 | |
| $e_6$ | removeMember | MailingServer | 2 | Main | Test | | |
| $e_7$ | removeMember | MailingServer | 2 | Main | Test | | |
| $e_8$ | addMember | MailingServer | 4 | Main | Test | | |
| $e_9$ | notifyMember | MailingServer | 4 | Main | Test | | |
| $e_{10}$ | updateS | Member | 5 | notifyMember | MailingServer | 4 | |
| $e_{11}$ | removeMember | MailingServer | 4 | Main | Test | | |

# Pattern, instance, invocation

Design level

- A **design pattern** ➔ a tuple of roles+constrains

Implementation level

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

Execution level

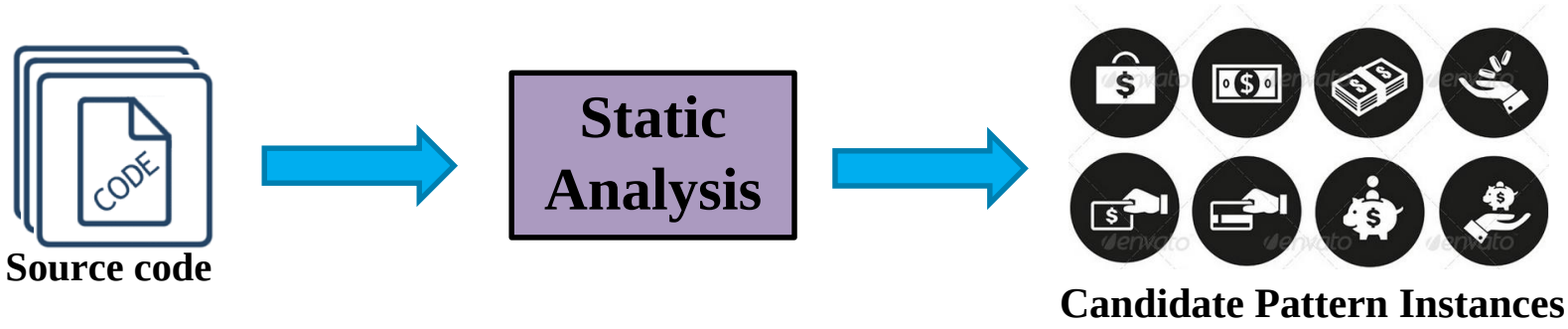
- A **design pattern instance invocation**➔ an independent execution of the pattern instance.

# Outline

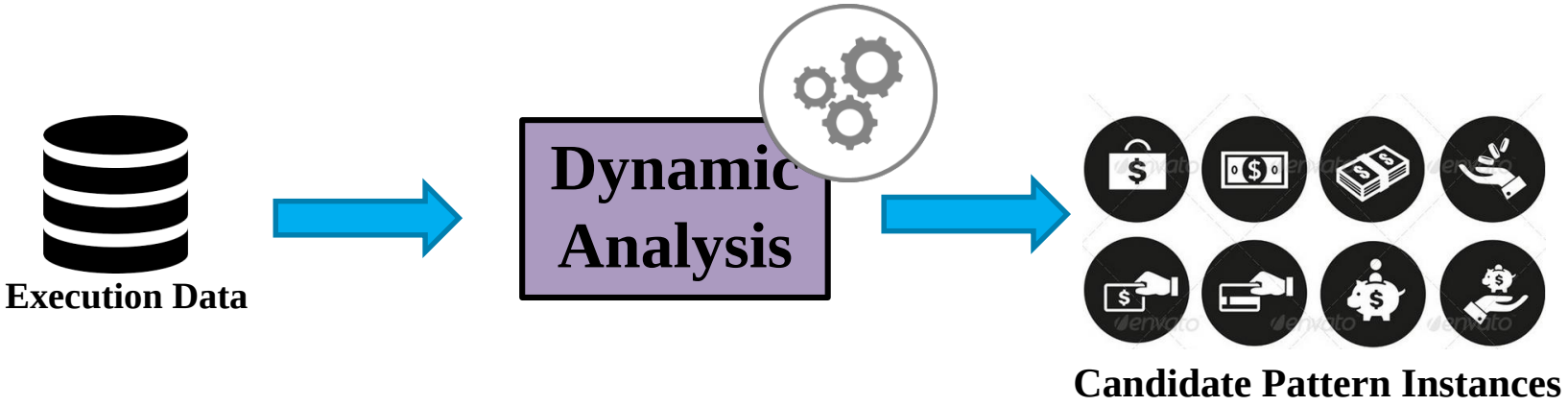
- Background
- Definition of design pattern
- Existing design pattern discover techniques
- Our framework

# Design pattern discovery/detection

- **Static analysis:**




- **Dynamic analysis:**

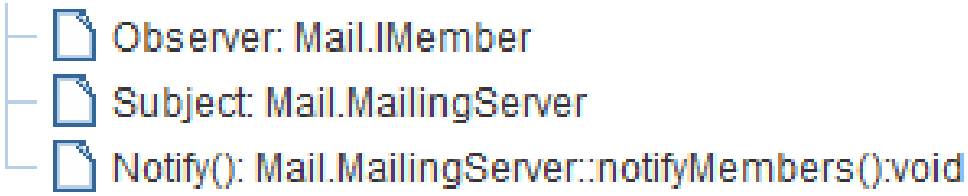


# Static analysis example

```
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();}
    }
}
```

**Source code**

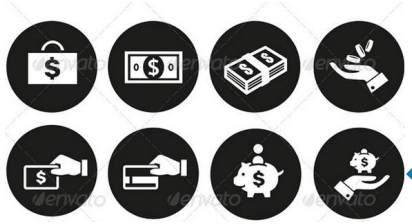
**DPD**




| Role | Code elements |
|---|---|
| Subject | MailingServer |
| Observer | Member |
| notify | notifyMembers |
| update | -- |
| register | -- |
| unregister | -- |

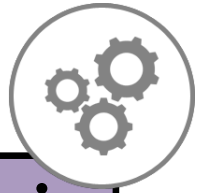
**Missing roles**

# Hybrid analysis



Candidate Pattern Instances

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



**Dynamic Analysis**

Pattern instances

**Execution Data**

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(register,i) < T(unregister,i). No explicit support for invocation identification before checking.
- Existing tools return incomplete pattern instance, manual efforts are required to understand.

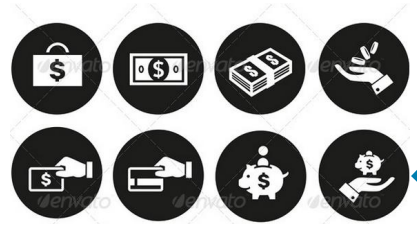
# Outline

- 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

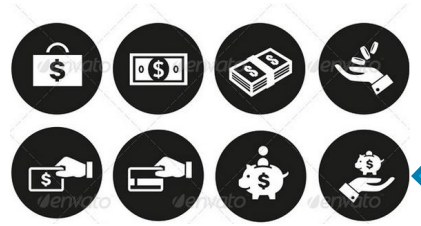
How to support pattern X?

Dynamic Analysis

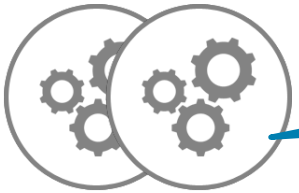
Execution Data

Pattern instances

# Extensibility



Candidate Pattern Instances



Constraints of new pattern X.

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

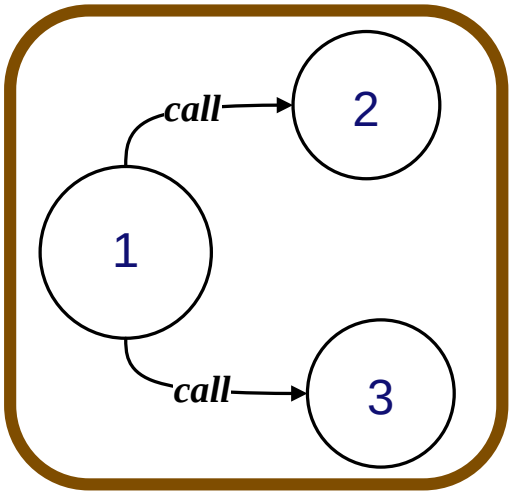
❑Arcs: calling relation among them.

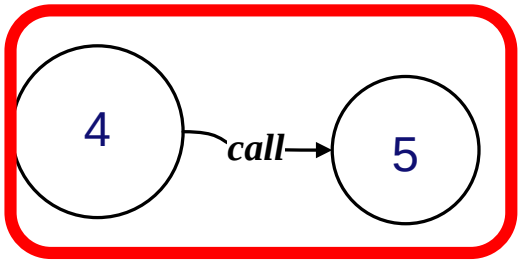
# Invocation identification

invocation m

| ID | Method | Class | Obj | C. M | C. C | C. O | T |
|---|---|---|---|---|---|---|---|
| $e_1$ | addMember | MailingServer | 1 | Main | Test | | |
| $e_2$ | addMember | MailingServer | 1 | Main | Test | | |
| $e_3$ | notifyMember | MailingServer | 1 | Main | Test | | |
| $e_4$ | receive | Member | 2 | notifyMember | MailingServer | 1 | |
| $e_5$ | receive | Member | 3 | notifyMember | MailingServer | 1 | |
| $e_6$ | removeMember | MailingServer | 2 | Main | Test | | |
| $e_7$ | removeMember | MailingServer | 2 | Main | Test | | |
| $e_8$ | addMember | MailingServer | 4 | Main | Test | | |
| $e_9$ | notifyMember | MailingServer | 4 | Main | Test | | |
| $e_{10}$ | updateS | Member | 5 | notifyMember | MailingServer | 4 | |
| $e_{11}$ | removeMember | MailingServer | 4 | Main | Test | | |

invocation n



# Discovery

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

| Role | Value |
|---|---|
| Subject | MailingServer |
| Observer | Member |
| notify | notifyMember |
| update | -- |
| register | -- |
| unregister | -- |

notify $\in$ **M**(Subject)
update $\in$ **M**(Observer)
register, unregister $\in$ **M**(Subject)
Observer $\notin$ **P**(notify)
Observer $\in$ **P**(register)
Observer $\in$ **P**(unregister)
update $\in$ **I**(notify)

update $\in$ **M**(Observer)

M(Member)={receive}

update $\leftarrow$ {receive}

| Role | Value |
|---|---|
| Subject | MailingServer |
| Observer | Member |
| notify | notifyMember |
| update | receive |
| register | addMember |
| unregister | removeMember |

| Role | Value |
|---|---|
| Subject | MailingServer |
| Observer | Member |
| notify | notifyMember |
| update | updateM |
| register | removeMember |
| unregister | addMember |

# Checking

Using the constraints, data and complete candidate pattern instance to remove false positives.

| Role name | Value |
|---|---|
| Subject | MailingServer |
| Observer | Member |
| notify | notifyMember |
| update | updateM |
| register | registerM |
| unregister | unregisterM |

notify ∈ **M**(Subject)
update ∈ **M**(Observer)
register, unregister ∈**M**(Subject)
Observer ∉ **P**(notify)
Observer ∈ **P**(register)
Observer ∈**P**(unregister)
update ∈ **I**(notify)

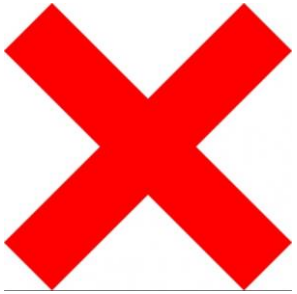
**T**(register)<**T**(unregister)
**T**(register)<**T**(notify)
**I**˚(notify)➔(update)

| Role name | Value |
|---|---|
| Subject | MailingServer |
| Observer | Member |
| notify | notifyMember |
| update | updateM |
| register | un-registerM |
| unregister | registerM |

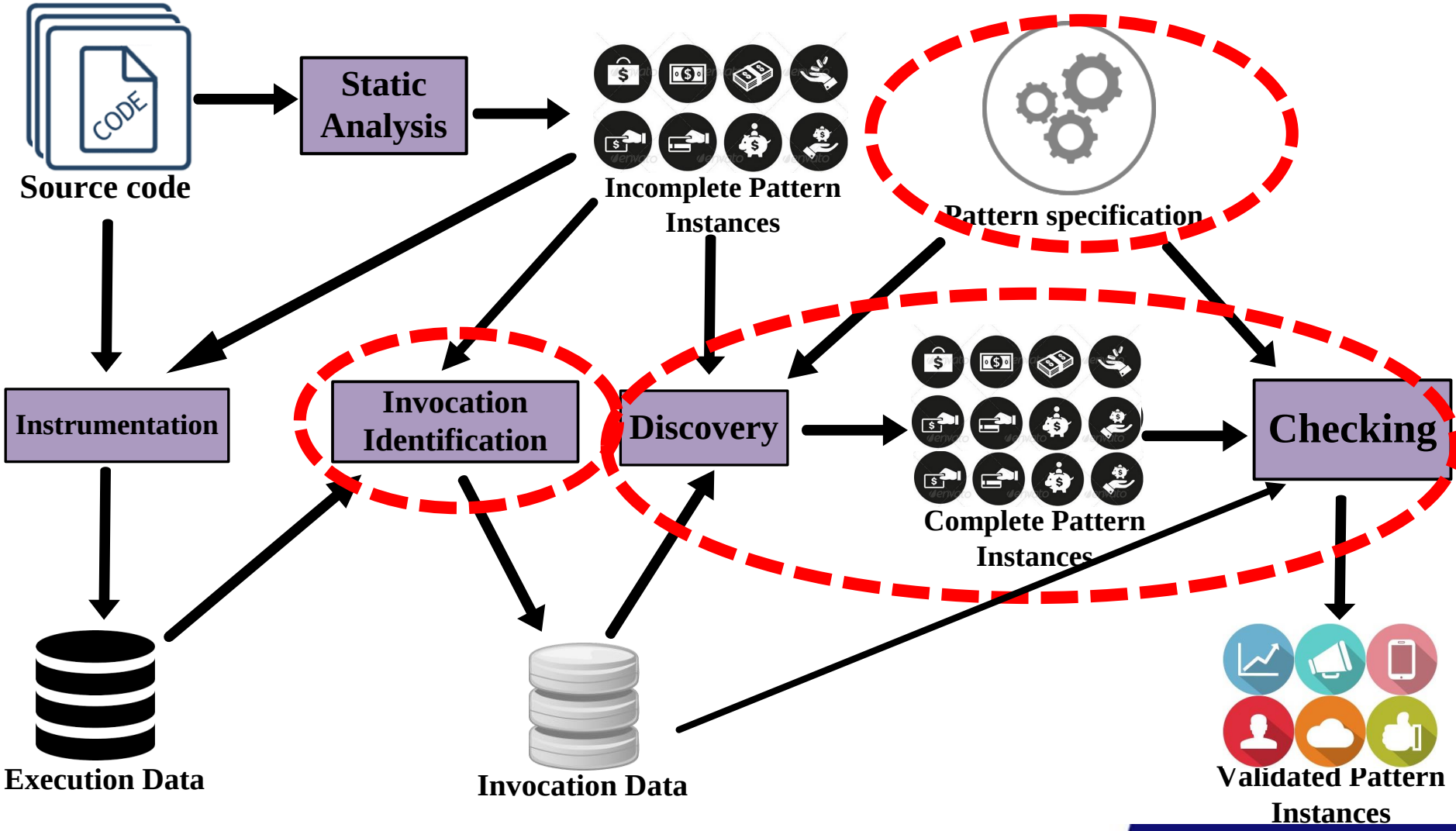
notify ∈ **M**(Subject)
update ∈ **M**(Observer)
register, unregister ∈**M**(Subject)
Observer ∉ **P**(notify)
Observer ∈ **P**(register)
Observer ∈**P**(unregister)
update ∈ **I**(notify)

**T**(register)<**T**(unregister)
**T**(register)<**T**(notify)
**I**˚(notify)➔(update)

**T**(register,i)<**T**(unregister,i)



# Summary of the Framework



# 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!!!