A Graphical Notation for Design Pattern Detection

Haneen Dabain

Department of Computer Science and Engineering
York University

October 24, 2011

Table of contents

Design Pattern Detection

- Design patterns are well-known and frequently reused structures in the software development community.
- Design patterns enable large scale system reuse of software architectures.
- Design patterns help maintainers understand the system and the decisions made when designing and implementing the system.
- Design patterns help developer become proficient in the source code.
- Design patterns are often scattered in the source code of systems after implementation.

Design Pattern Detection

- Documentation of a system's design is often obsolete, if it exists at all.
- Design choices are thus lost.
- Analyzing systems manually to extract the design patterns can be time consuming and infeasible.
- Design pattern detection tools have been developed to automatically extract design pattern implementations.

Coarse-Grained Analysis vs. Fine-Grained Analysis

Coarse-Grained Analysis

Considers high-level information about system structure such as class inheritance and hierarchies, and class dependencies and interaction.

- Relatively fast.
- Scales well on large systems.
- The high-level information may not be sufficient to clearly differentiate between some design patterns.

Coarse-Grained Analysis vs. Fine-Grained Analysis

Fine-Grained Analysis

Captures finer details about the system's structure such as method signatures, method invocations, and field declarations.

- Provides more accurate results.
- May become infeasible when applied on large scale systems due to lack of memory or intensive CPU usage.

Design Pattern Detection

- Different tools produce different results and therefore vary in their performance.
- Results need to be evaluated to measure tools performance.

Detection Results Evaluation

- Precision and recall rates.
  - Precision rate: The percentage of true implementations in the recovered implementations (Accuracy).
  - Recall rate: The percentage of the actual true implementations that are correctly recovered (Completeness).
- Various studies have been conducted in order to evaluate and compare the results obtained by different tools.

Detection Results Evaluation

Challenges:

- Design pattern implementation variations.
- Different tools use different design pattern definitions.
- Evaluating results against different systems.
- No gold standard against which the tools are being evaluated.

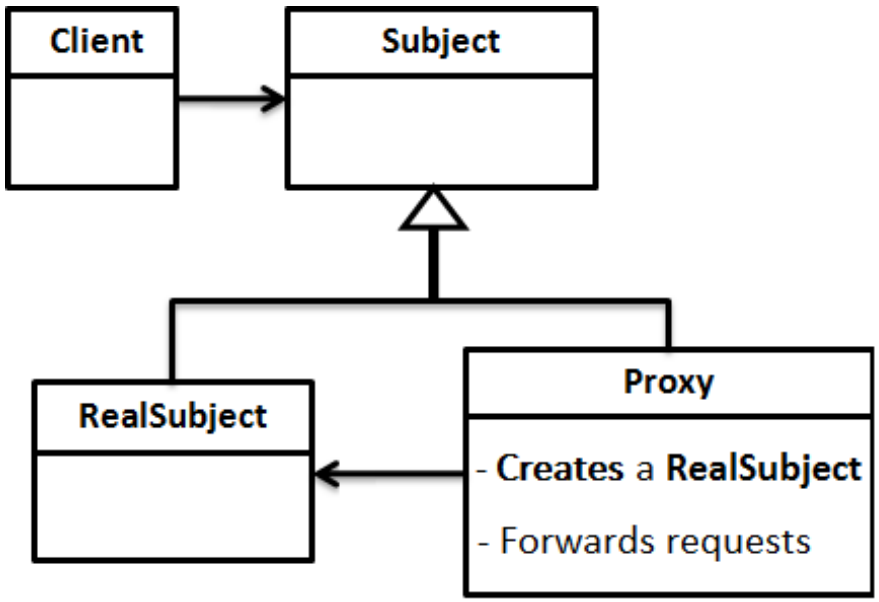
Detection Results Evaluation

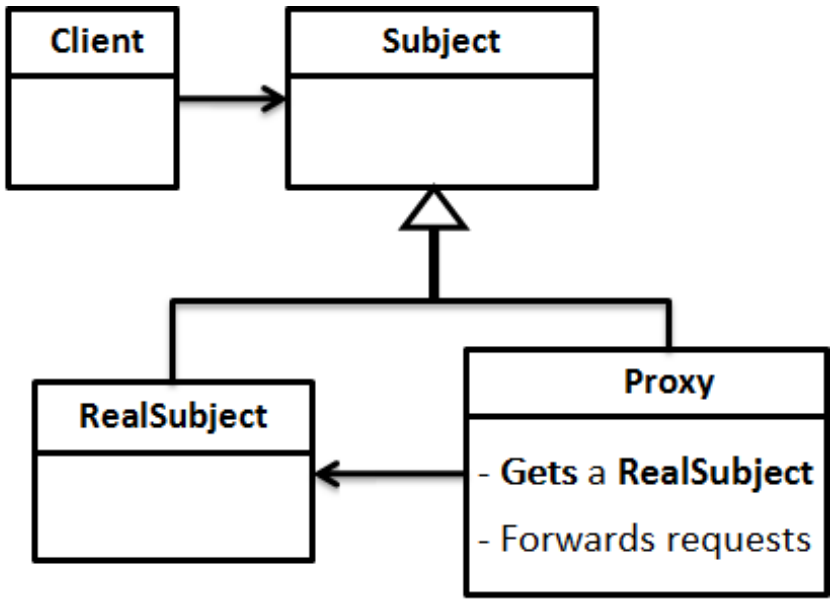
Challenges:

- **Design pattern implementation variations**.
- **Different tools use different design pattern definitions**.
- Evaluating results against different systems.
- No gold standard against which the tools are being evaluated.

Detection Results Evaluation

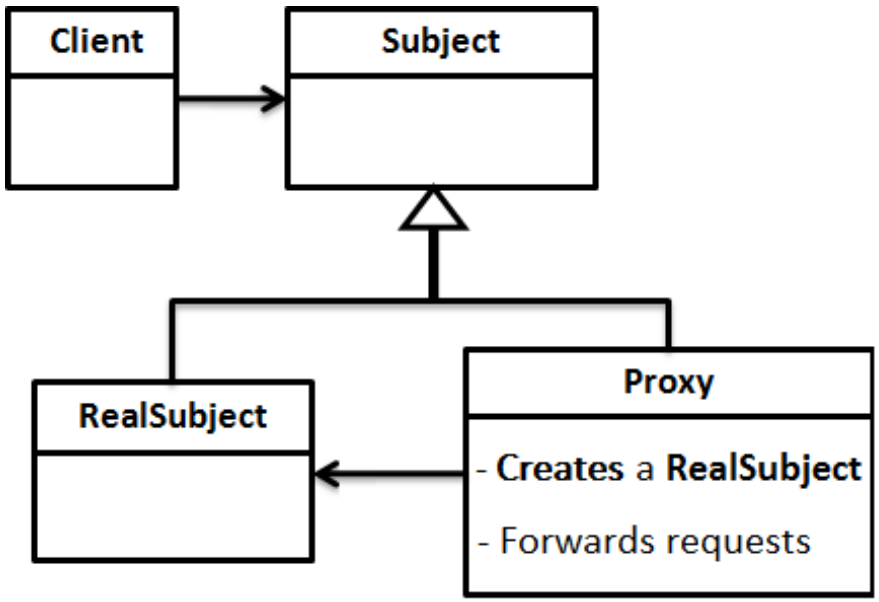
Example: Proxy design pattern

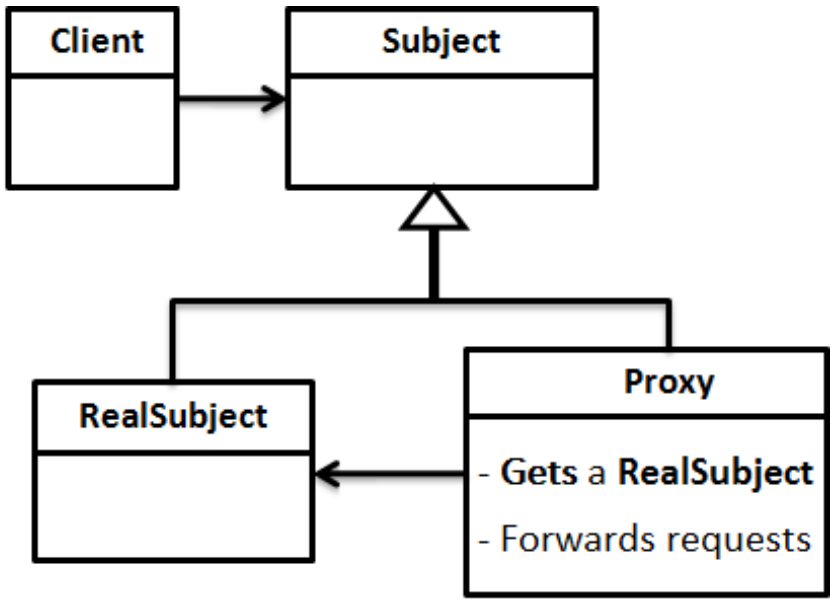




Detection Results Evaluation

Example: Proxy design pattern



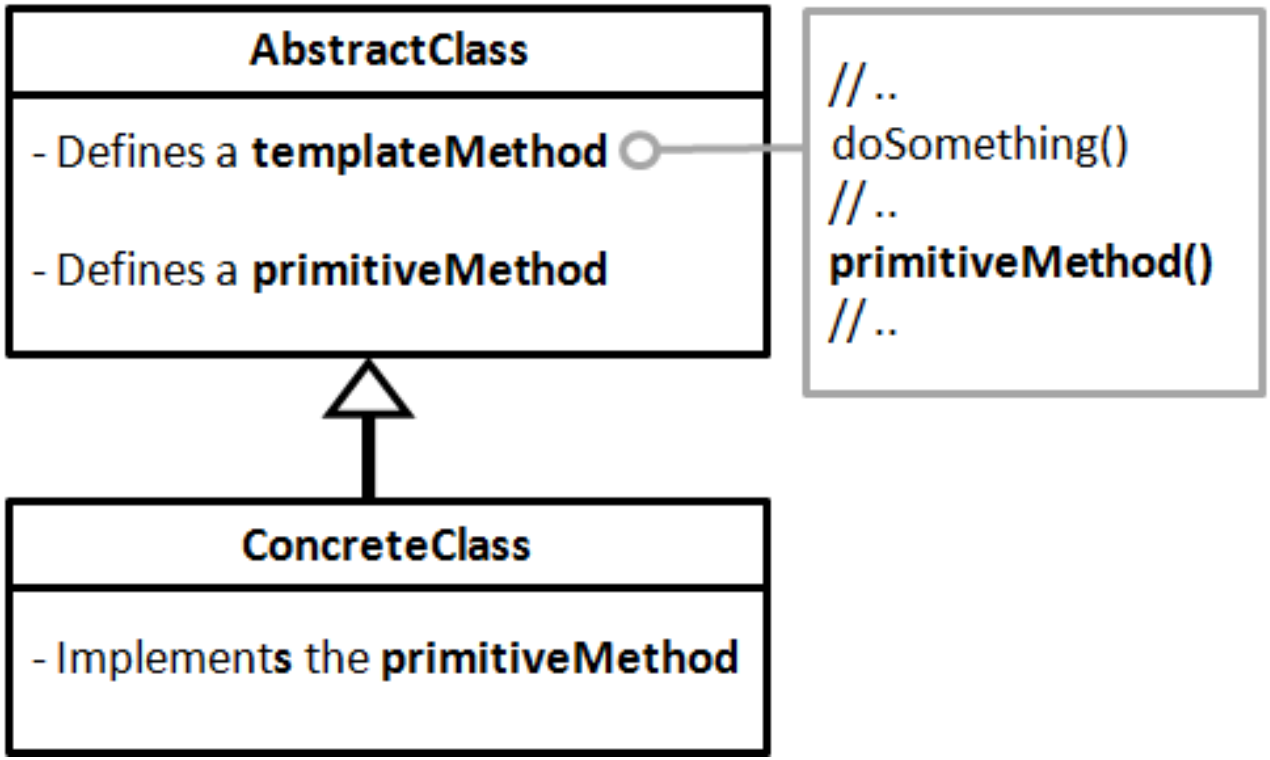


PINOT: Recovered

Not recovered!

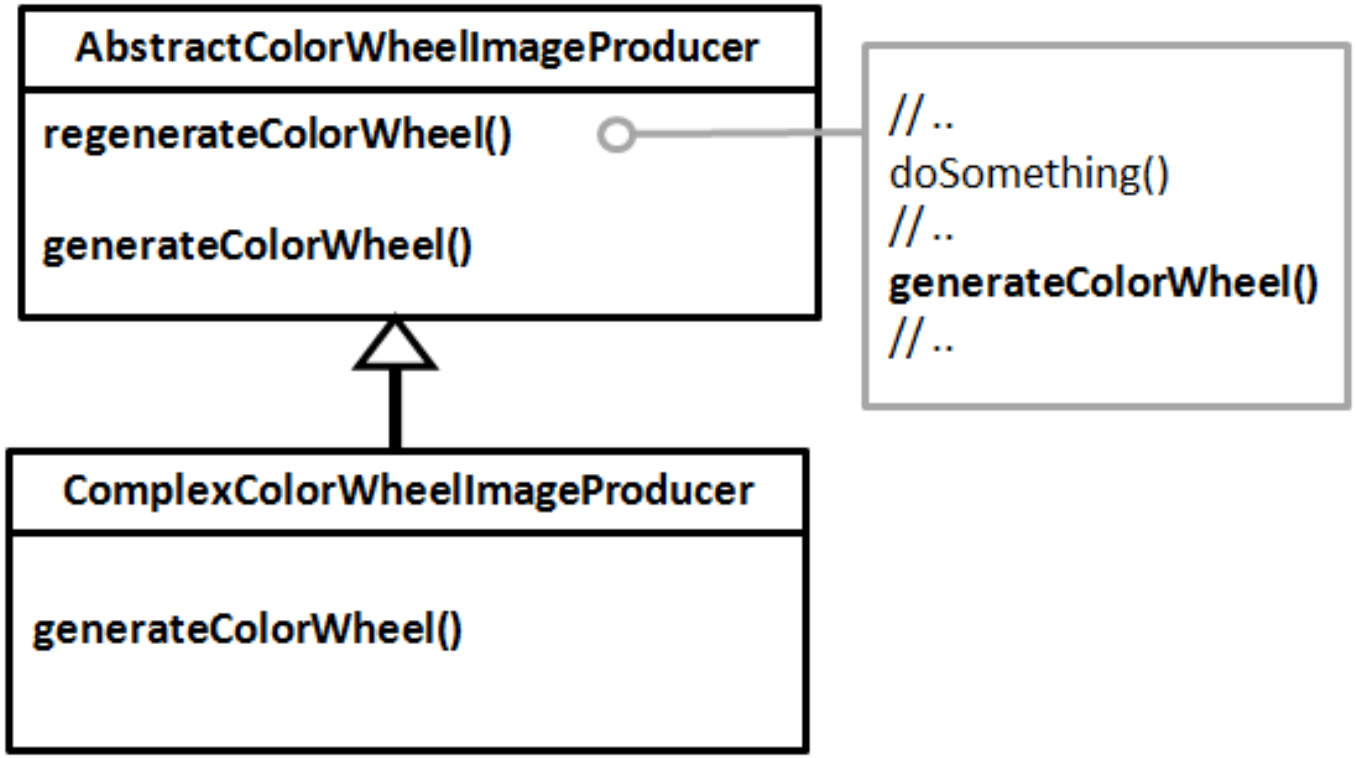
Detection Results Evaluation

Example: Template Method design pattern



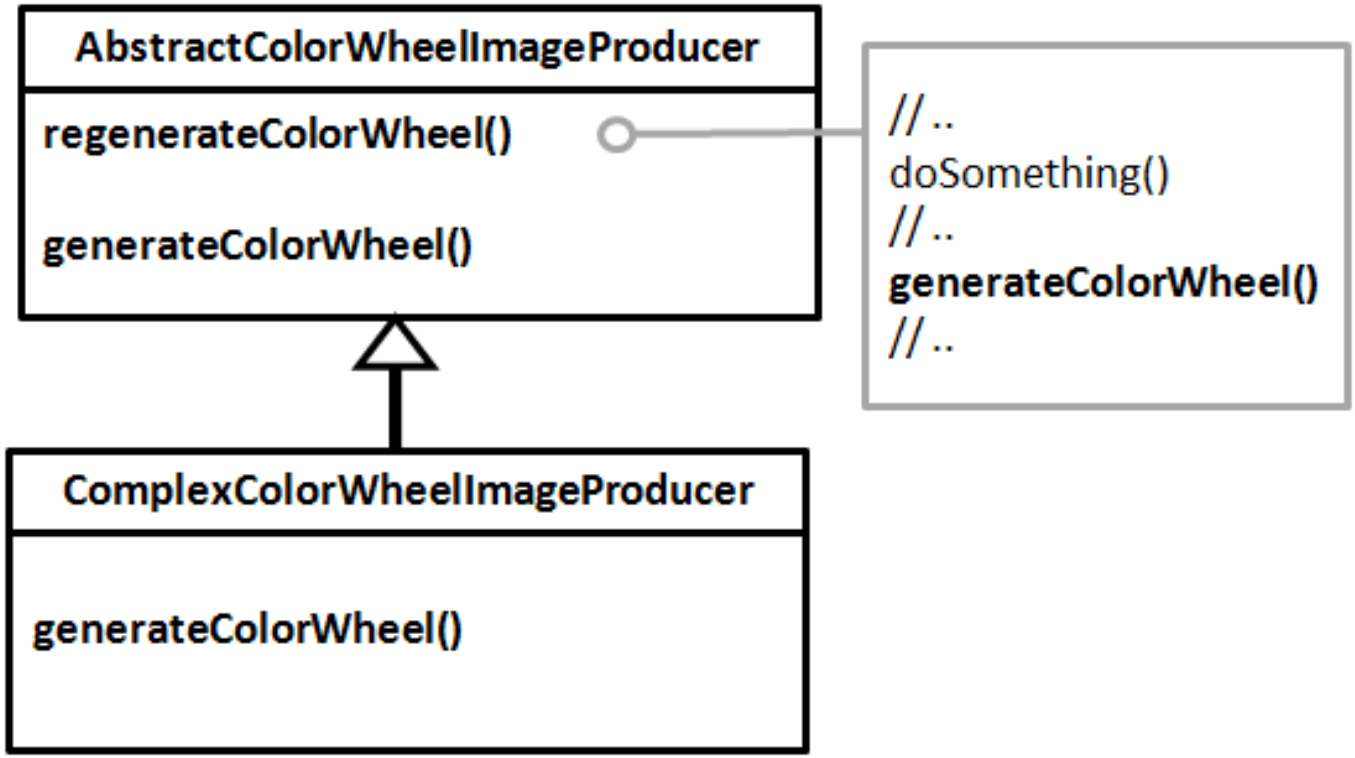
Detection Results Evaluation

Example: Template Method design pattern



Detection Results Evaluation

Example: Template Method design pattern



- SSA: Recovered

- PINOT: **Not recovered!**

Table of contents

Our Contribution

- The introduction of a graphical notation for describing design pattern definitions.
  - Describes the properties of design pattern implementations.
  - Supports design pattern variations.
- The introduction of design pattern definitions catalogue, such that these definitions:
  - Translate to a set of fine-grained rules that can be used for detecting design patterns.
  - Provide developers with roadmap for implementing design patterns.

Our Contribution

- The development of a tool that recovers design pattern implementations using fine-grained rules that correspond to design pattern definitions.
- The evaluation of the effectiveness of our approach in terms of results correctness and scalability.

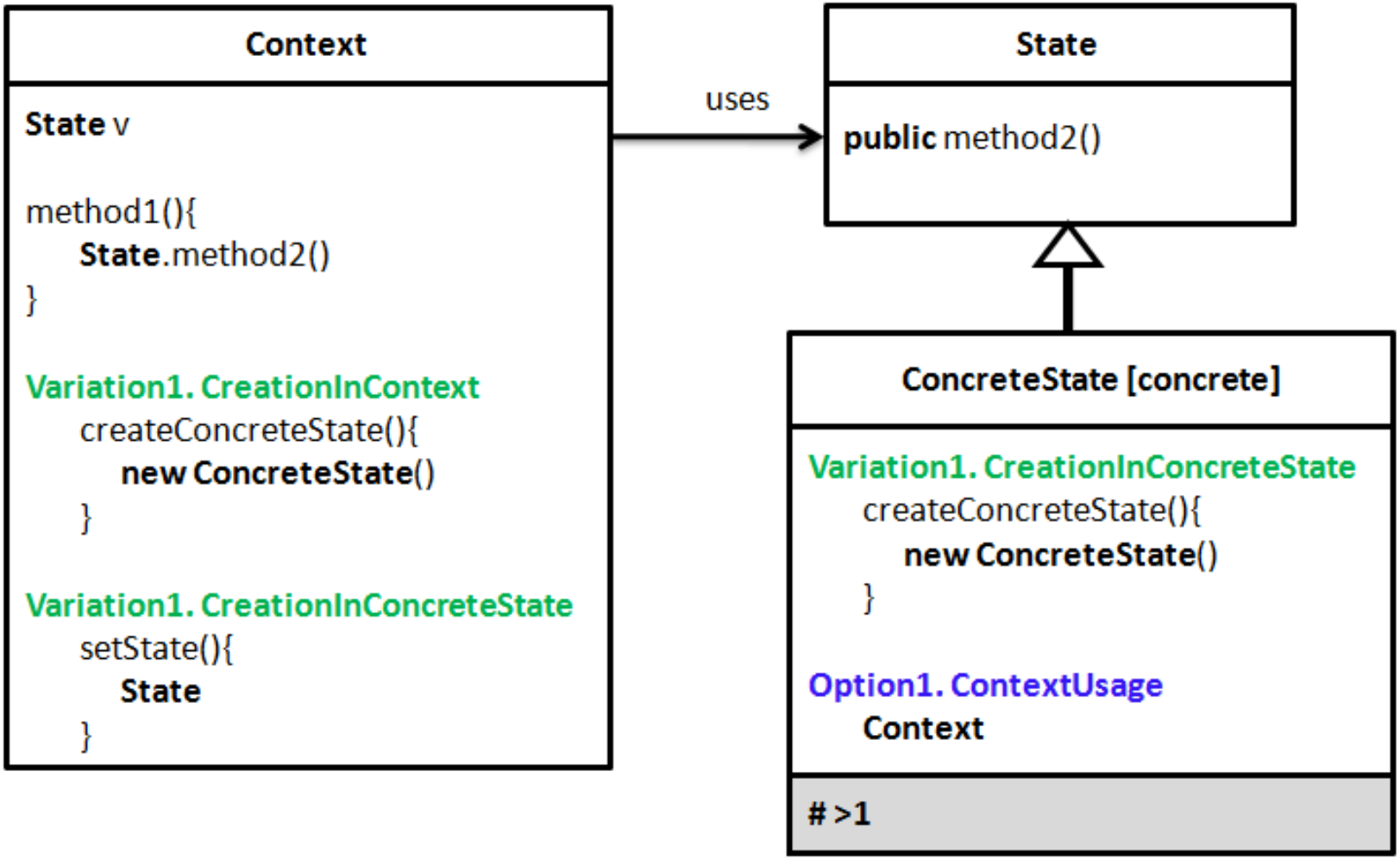
Table of contents

The FINDER Notation

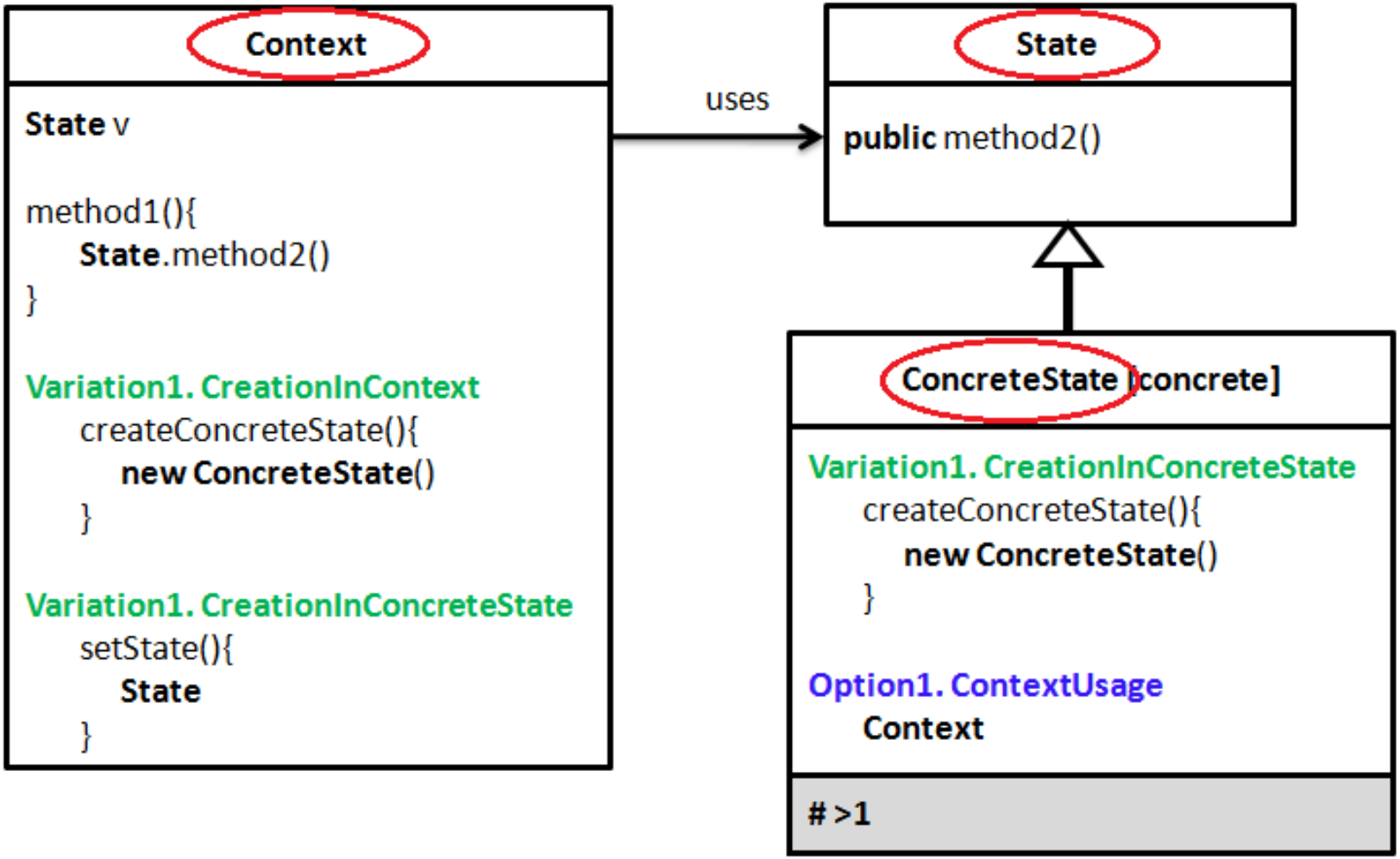
FINDER (**FIN**e-grained **DE**tection **R**ules)

UML-like diagrams describing fine-grained rules used for the detection of design pattern implementations.

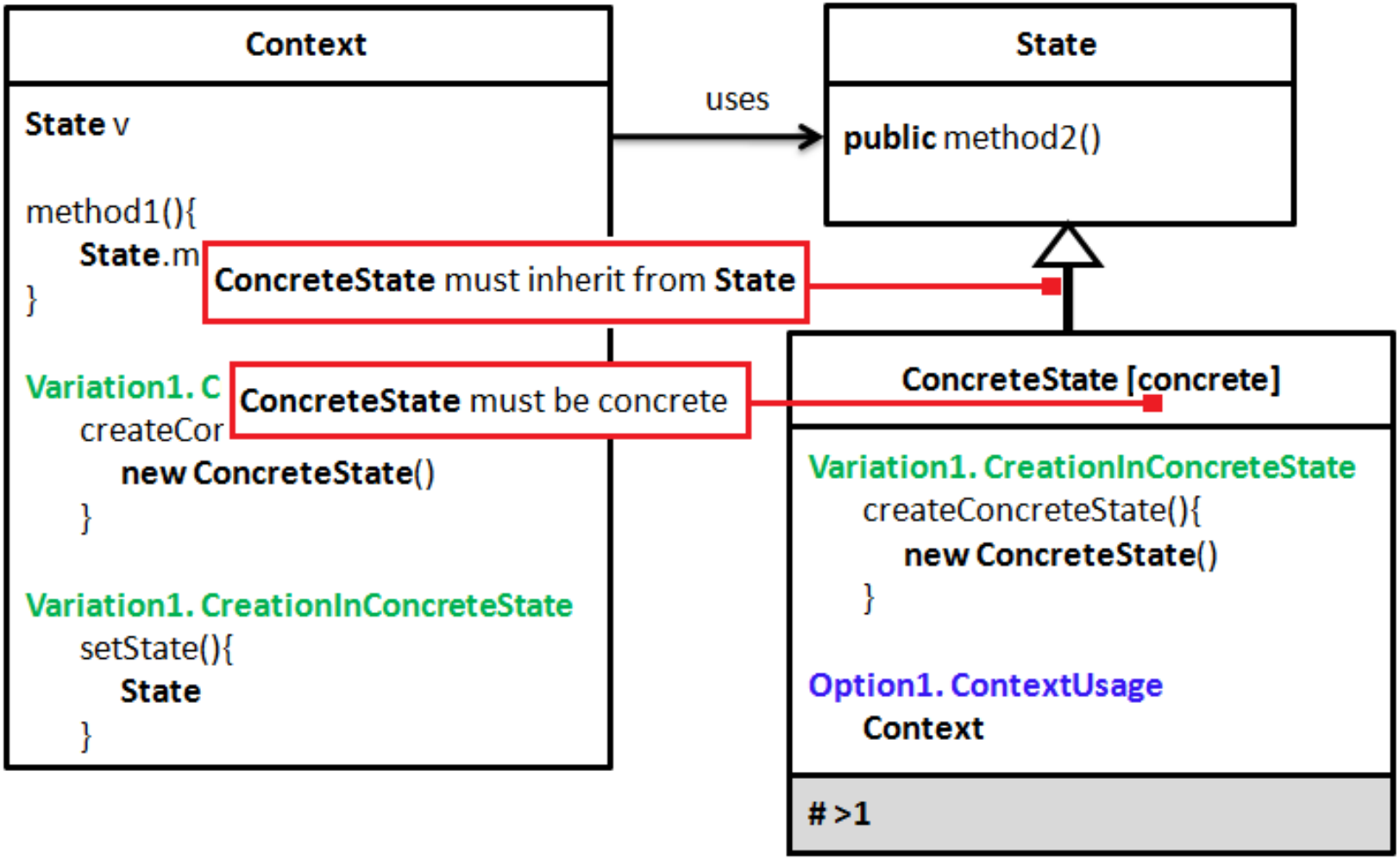
Example: The State Design Pattern FINDER Definition



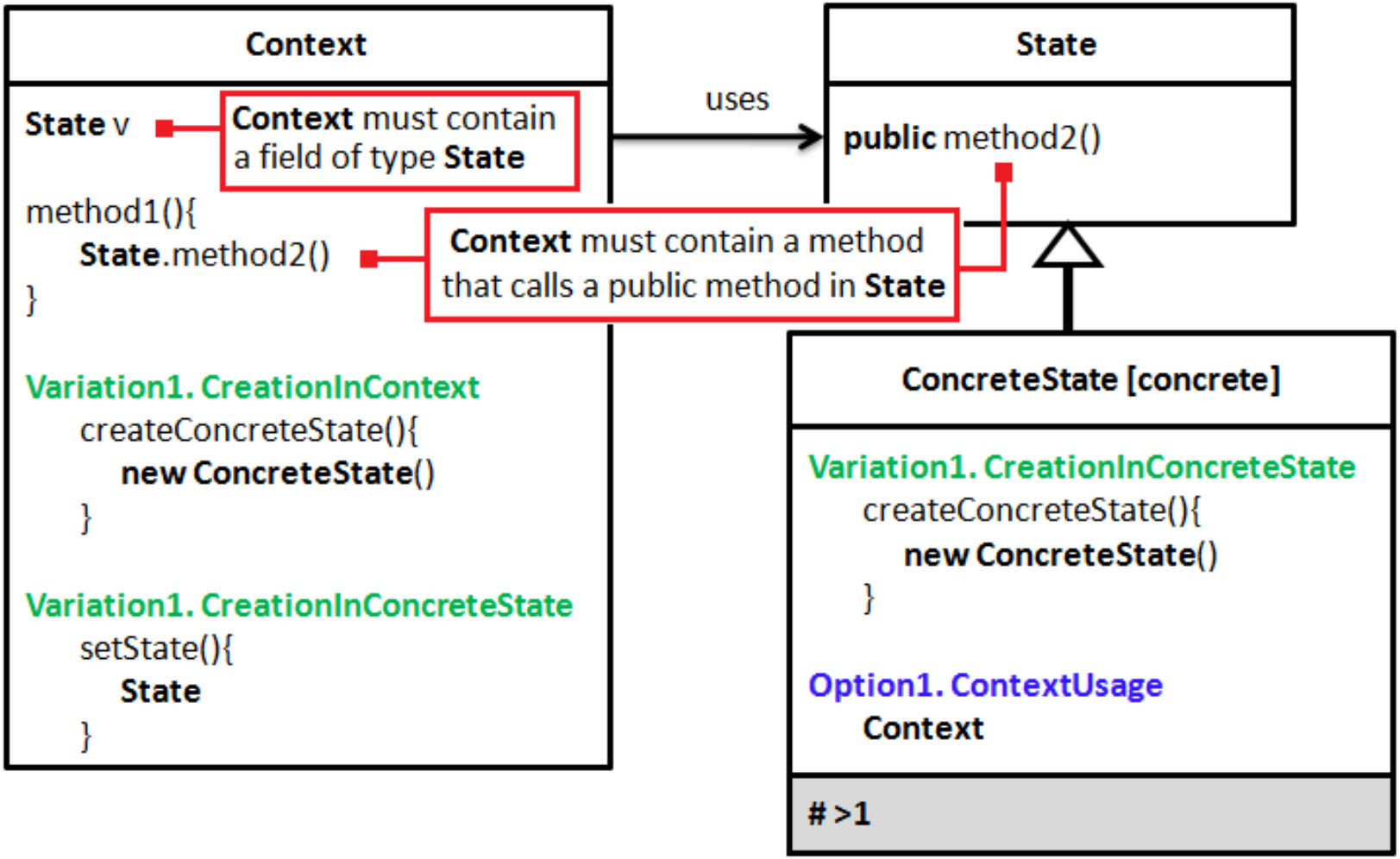
Example: The State Design Pattern FINDER Definition



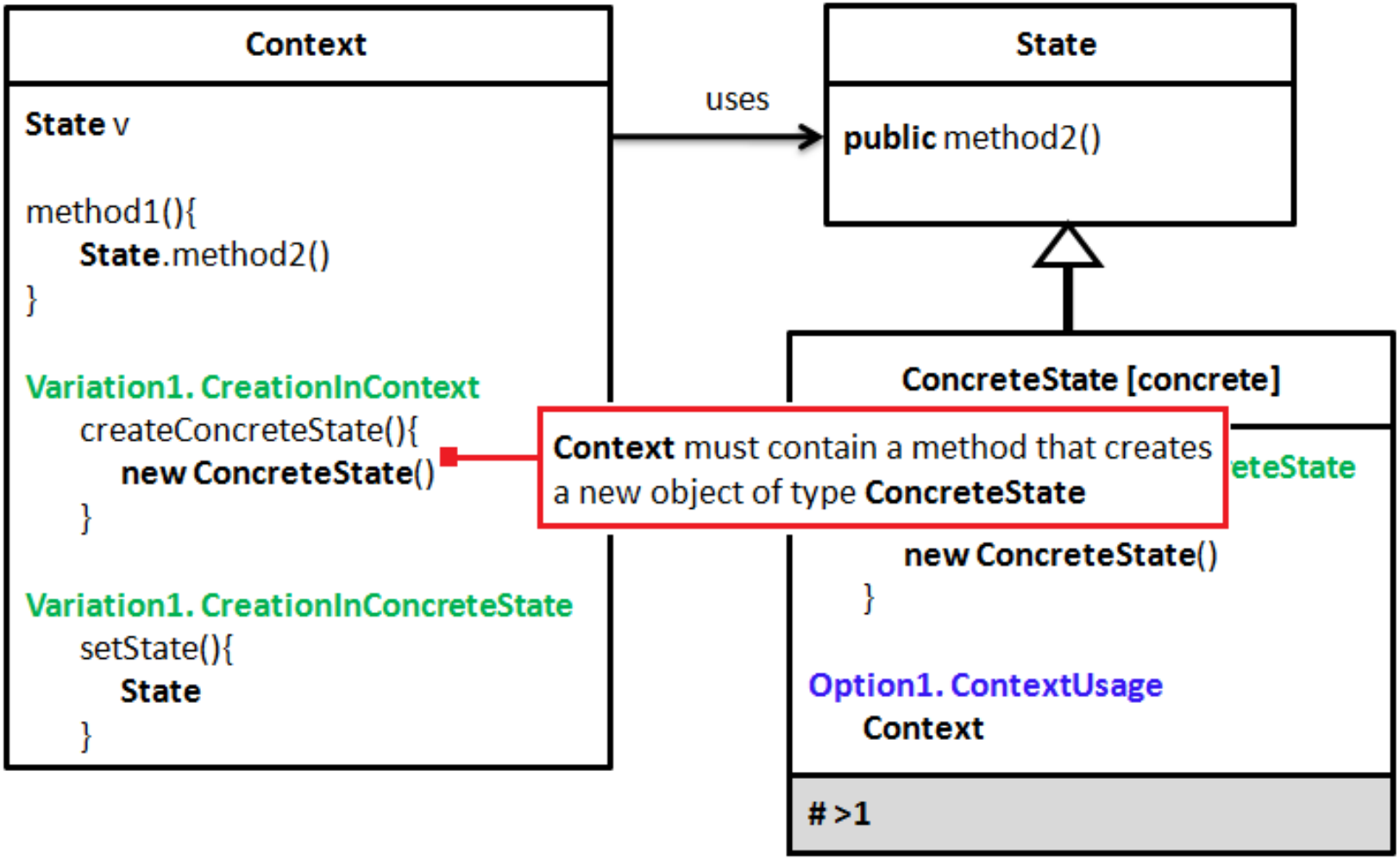
Example: The State Design Pattern FINDER Definition



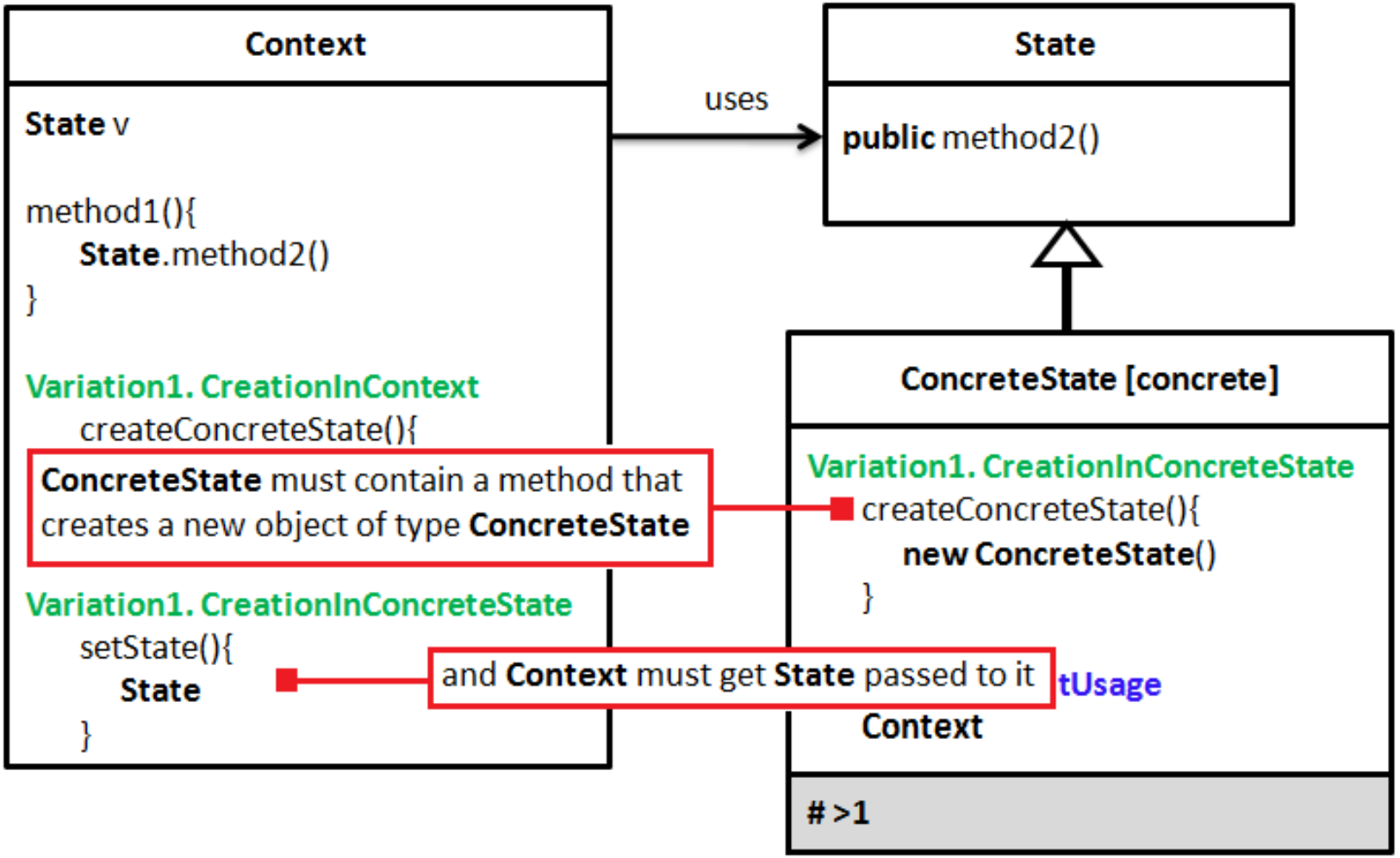
Example: The State Design Pattern FINDER Definition



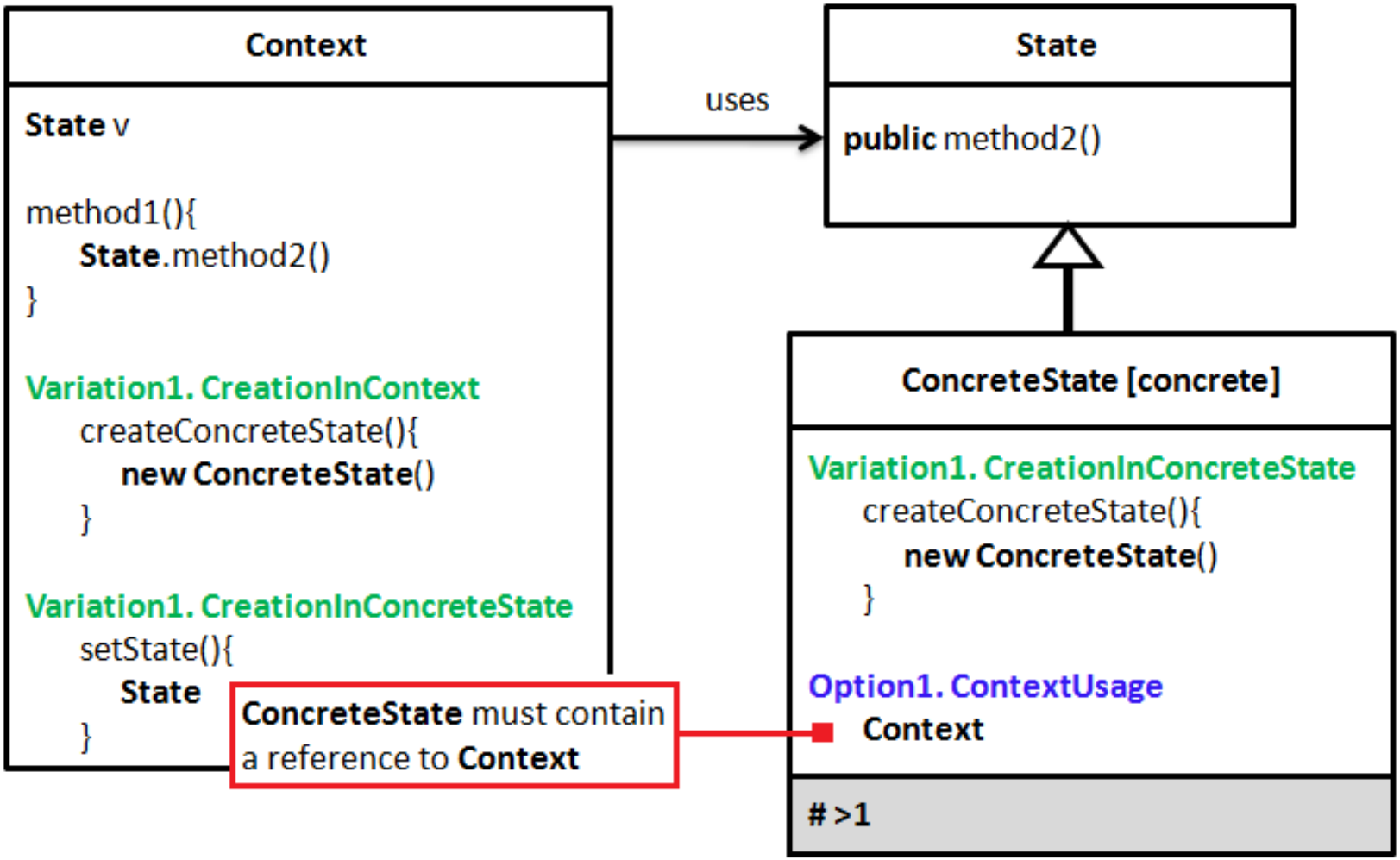
Example: The State Design Pattern FINDER Definition



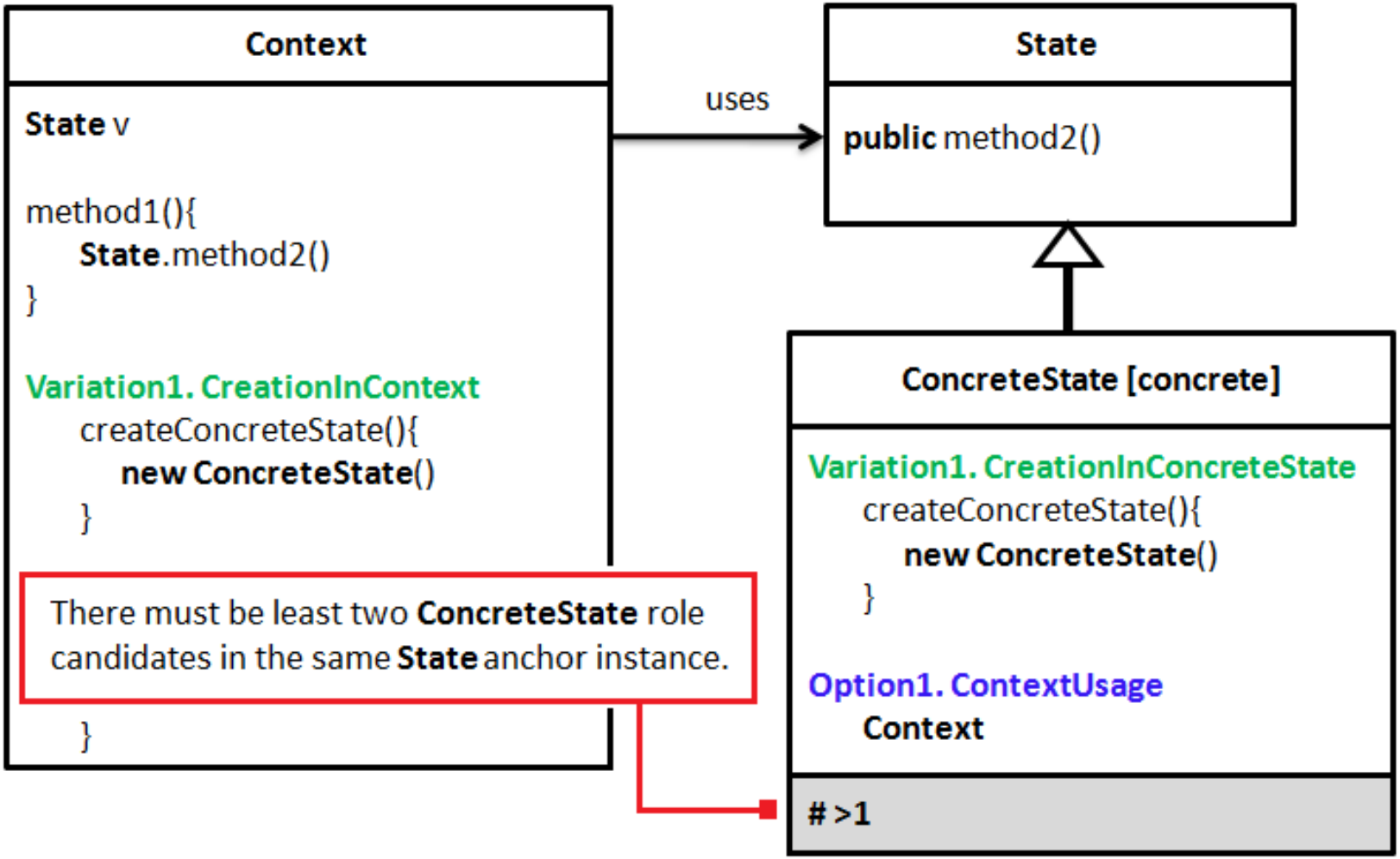
Example: The State Design Pattern FINDER Definition



Example: The State Design Pattern FINDER Definition



Example: The State Design Pattern FINDER Definition



How does the FINDER notation help?

- FINDER definitions provide developers with guidelines for implementing design patterns.
- Different tools can use the FINDER notation as a common language for representing their detections rules.
- The visual aspect will help understand fine-grained rules and their contribution to the design pattern implementations.
- Facilitates comparison between different design pattern definitions used by different tools

Table of contents

The FINDER Tool

Approach

- Static Fact Extraction:
  Create a factbase of fine-grained information about the system structure and components.

- Design Pattern Instance Recovery:
  Filter the factbase for components that match design pattern definition rules.

The FINDER Tool

Static Fact Extraction

- Javex and QL are used to produce a factbase that contains information about class hierarchies and attributes, class methods and fields, class interaction, and methods signature.

- Limitation:
  Javex is unable to capture the type of objects contained in collections.

VariableExplorer

The VariableExplorer Tool

- Collects information about collections, methods invoked on collections, and the parameters sent to those methods.
- This information is used to estimate the type of objects in collections.

VariableExplorer

AST (Abstract Syntax Tree)

- AST is a tree representation of the abstract syntactic structure of source code.
- Each element in the Java source file is represented as an ASTNode.
- ASTVisitor traverses ASTNodes and gathers information about their properties and fragments.

VariableExplorer

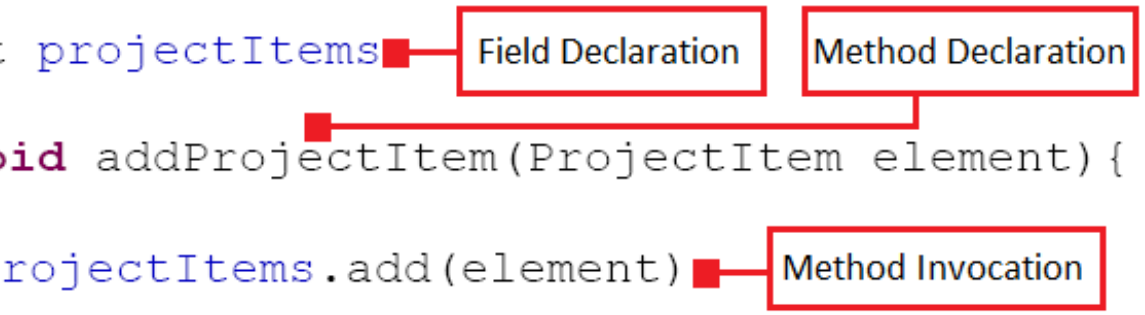
How it works

```
public class Project{

    ArrayList projectItems;

    public void addProjectItem(ProjectItem element){

        projectItems.add(element);

    }

}
```

VariableExplorer

How it works

```
public class Project{

    ArrayList projectItems

    public void addProjectItem(ProjectItem element){

        projectItems.add(element)

    }

}
```



VariableExplorer

How it works

```
public class Project{

    ArrayList projectItems;

    public void addProjectItem(ProjectItem element){

        projectItems.add(element);

    }

}
```

VariableExplorer

How it works

```
public class Project{

    ArrayList projectItems;

    public void addProjectItem(ProjectItem element){

        projectItems.add(element);

    }

}
```

VariableExplorer

How it works

```
public class Project{

    ArrayList<ProjectItem> projectItems;

    public void addProjectItem(ProjectItem element){

            projectItems.add(element);

    }

}
```

VariableExplorer

How it works

```
public class Project{

    ArrayList<ProjectItem> projectItems;

    public void addProjectItem(ProjectItem element){

        projectItems.add(element);

    }

}
```

Table of contents

AJP Implementations

Design Pattern Example	Factory Method	Abstract Factory	Builder	Prototype	Singleton	Object Adapter	Bridge	Composite	Decorator	Flyweight	Proxy	Chain of Responsibility	Command	Interpreter	Iterator	Mediator	Memento	Observer	State	Strategy	Template Method	Visitor
FactoryMethod	1																					
AbstractFactory	4	4																				
Builder			1																			
Prototype				1																		
Singleton					1																	
Adapter Object						1																
Bridge							2										2					
Composite						4	2	4														
Decorator								3	2			1										
Flyweight										2									2			
Proxy											1						2					
Chain of Responsibility												1										
Command	1					12	4						1									
Interpreter								10						8			2		6			
Iterator	1					3	1								1							
Mediator																3	4					
Memento																	1					
Observer																		3				
State																			2			
Strategy																				2		
Template Method								1													4	
Visitor						6		9														5

Recall rate: 100%, Precision rate of 78%

Additional Implementation Variations

Design Pattern Example	Factory Method	Factory Method Degenerate	Abstract Factory	AbstractFactory Degenerate	Builder	Prototype	Prototype Cloneable	Singleton	Singleton Bill	Object Adapter	Class Adapter	Bridge	Bridge Degenerate	Composite	Decorator	Decorator Degenerate	Flyweight	Proxy
FactoryMethod Degenerate		4		4														
AbstractFactory Degenerate		4		4														
Prototype Cloneable						1												
Singleton Bill								1										
Adapter Class										2								
Bridge Degenerate												2						
Decorator Degenerate															4			
Iterator Degenerate									1			1				1		
Iterator Java																		
Iterator Java Degenerate																		
Mediator Degenerate																		
Observer Degenerate																		
Visitor Reflective																		

...

Additional Implementation Variations - Continued

Design Pattern Example	Chain of Responsibility	Command	Interpreter	Iterator	Iterator Degenerate	Iterator Java	Iterator Java Degenerate	Mediator	Mediator degenerate	Memento	Observer	Observer degenerate	State	Strategy	Template Method	Visitor	Visitor Reflective
FactoryMethod Degenerate																	
AbstractFactory Degenerate																	
Prototype Cloneable																	
Singleton Bill																	
Adapter Class																	
Bridge Degenerate																	
Decorator Degenerate	2																
Iterator Degenerate					1												
Iterator Java						1					1						
Iterator Java Degenerate							1										
Mediator Degenerate									3								
Observer Degenerate											6						
Visitor Reflective																	6

Recall rate: 100%, Precision rate of 88%

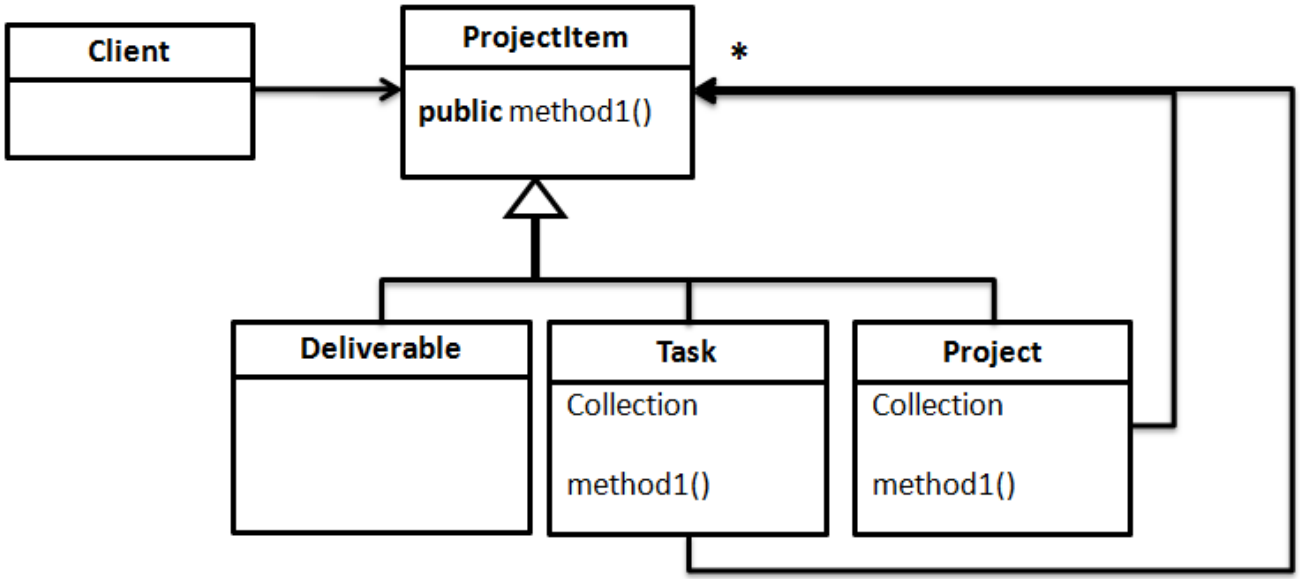
AJP Implementations

Design Pattern Options

Design Pattern Example	Observer	Observer with Subject	Strategy	Strategy with Context
Bridge	2			
Proxy	2			
Interpreter	2		6	
Mediator	4			
Observer	1	2		
Strategy			2	

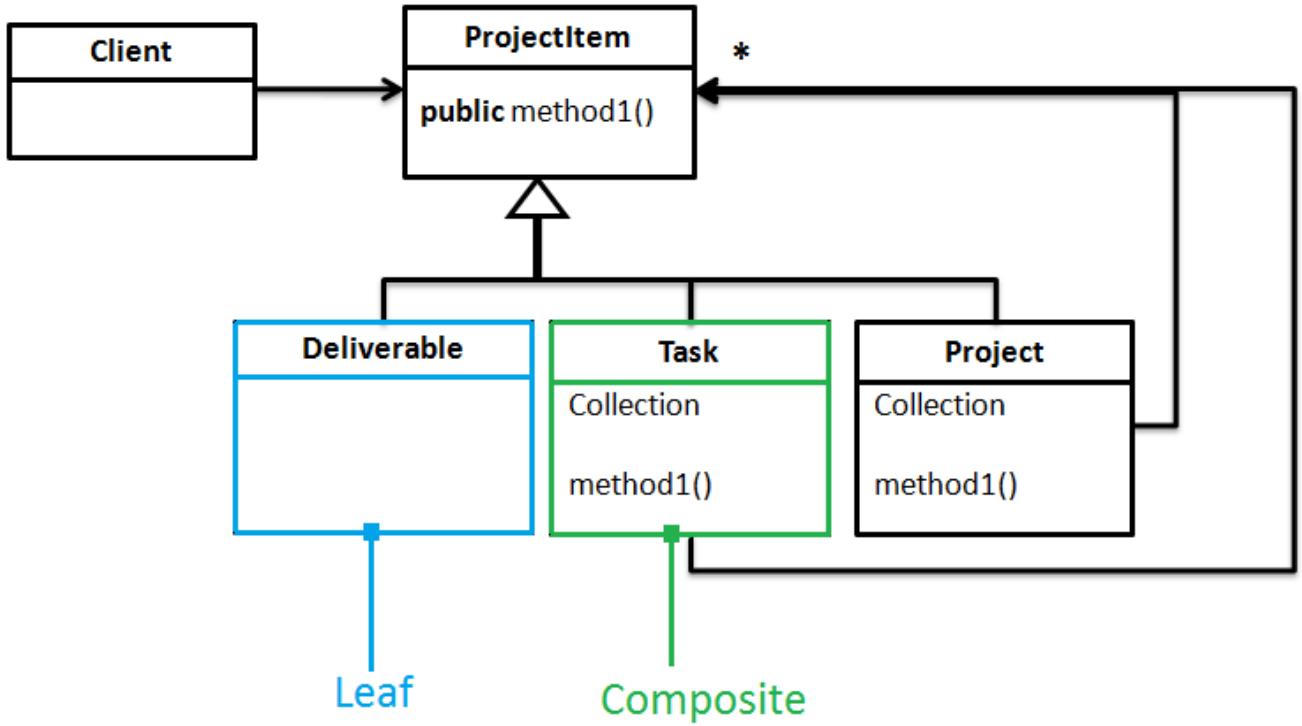
AJP Implementations

Composite Implementation



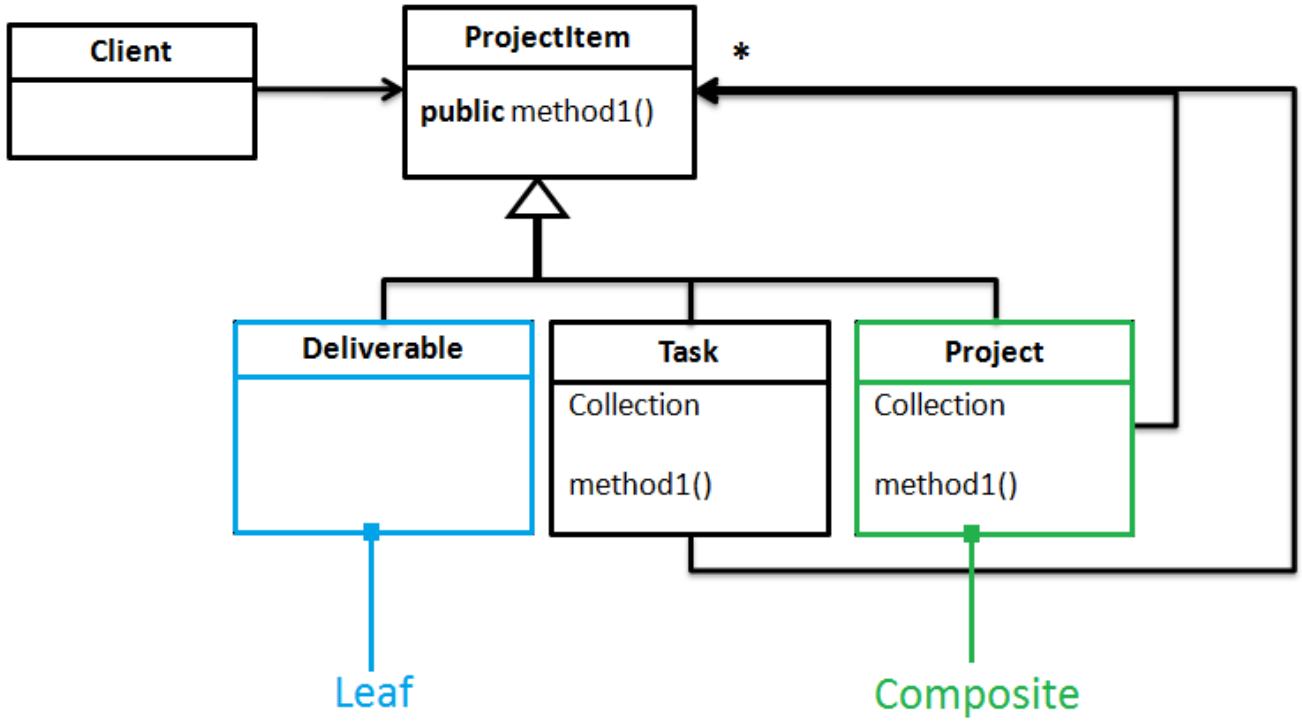
AJP Implementations

Recovered Composite Implementation



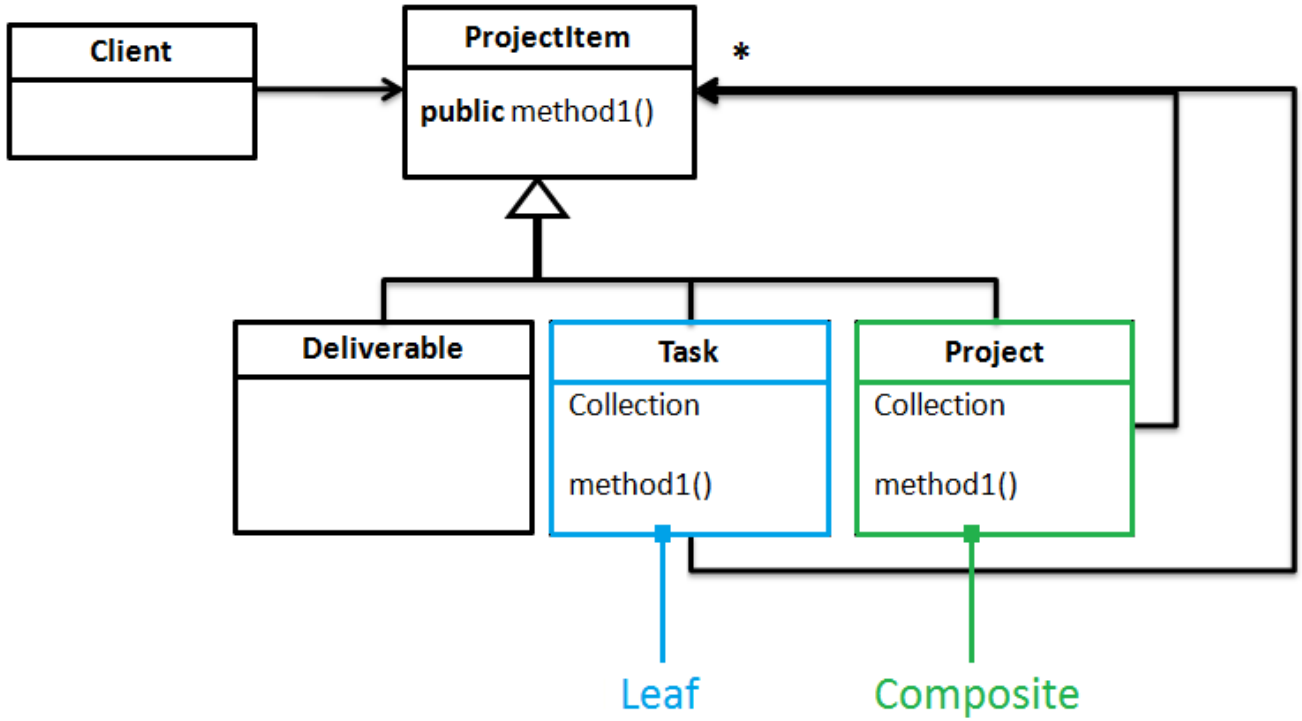
AJP Implementations

Recovered Composite Implementation



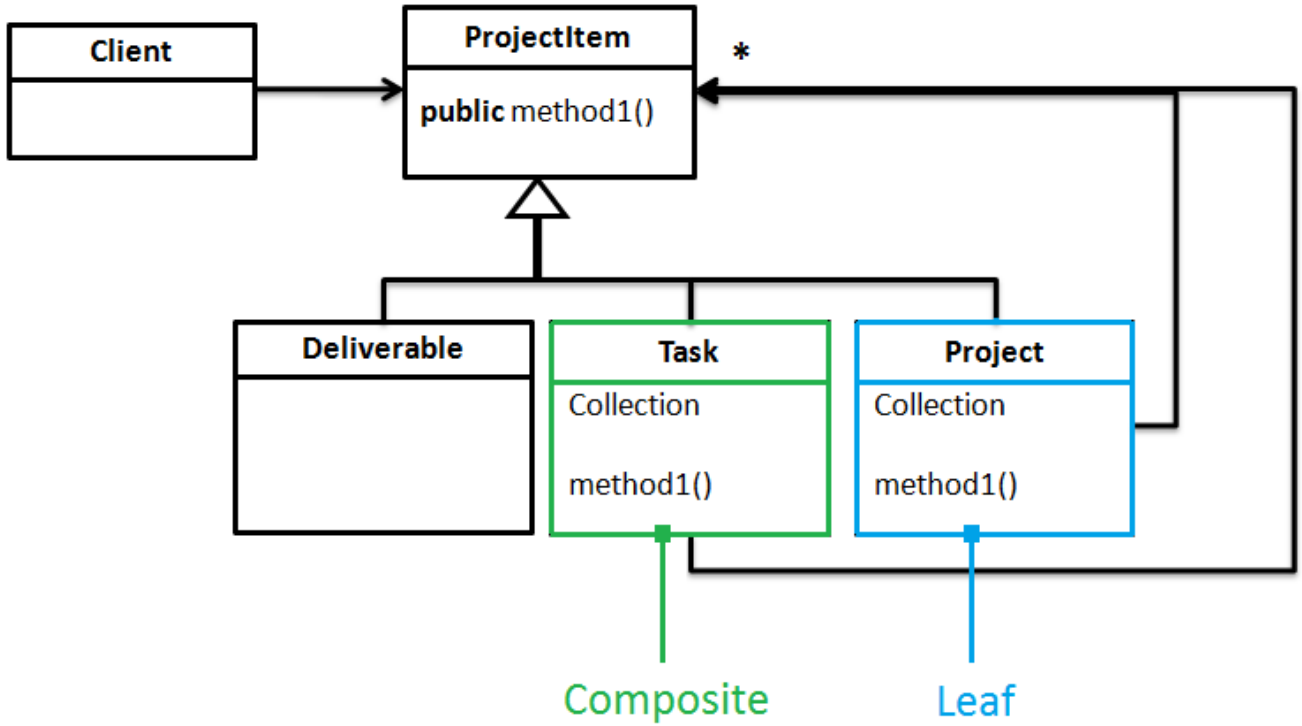
AJP Implementations

Recovered Composite Implementation without VariableExplorer



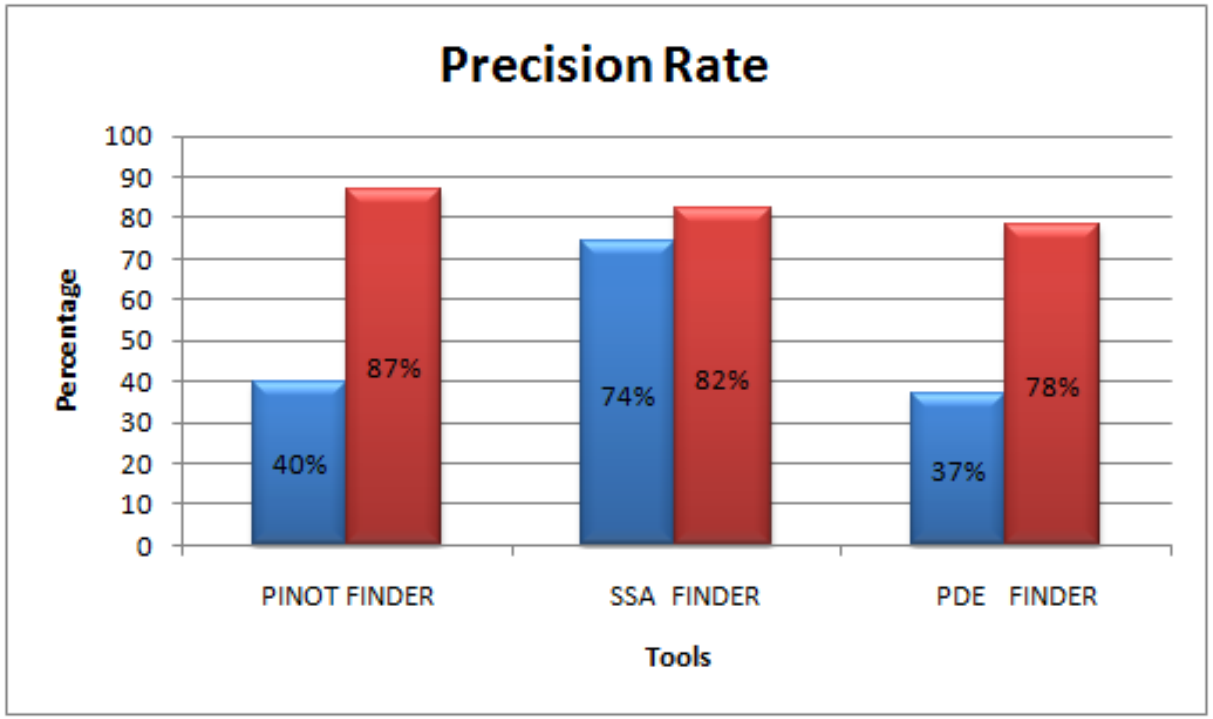
AJP Implementations

Recovered Composite Implementation without VariableExplorer



Experiments Results

AJP Implementations



Experiments Results

GTDFree and JHotDraw

FINDER was able to correctly recover several design pattern implementations, and that it scales well on large scale real life systems.

Table of contents

Conclusion

- FINDER notation describes fine-grained rules for design pattern detection, and helps understand fine-grained rules.
- FINDER definitions of the design patterns used as guidelines for building design patterns in Java.
- FINDER gave more accurate results than the tools it was compared against.
- FINDER scales well on large real life systems.

Table of contents

Future Work

- Conducting further analyses of design patterns implementation in Java systems to produce more refined and flexible fine-grained definitions.
- Creating FINDER definitions for new design patterns and adding them to our catalogue.
- Improving the performance of the FINDER tool, by optimizing the scripts used for factbase production and design patterns recovery.
- Performing further tests to better evaluate the performance of our tool.

Questions

Questions

Thank you!