A Appendix-A

A.1 Catalogues of Design Patterns

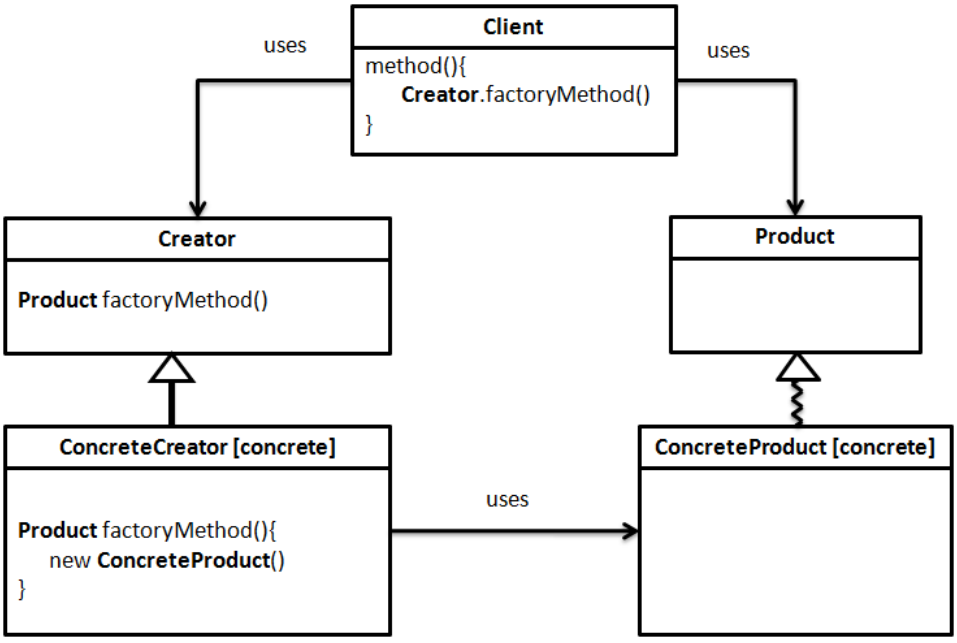
Below is the definition for each design pattern using the FINDER notation, followed by a description of the rules. These definitions have been created using our notations described in Chapter 3.

A.1.1 Factory Method

The Factory Method design pattern consists of five roles: Client, Creator, ConcreteCreator, Product, and ConcreteProduct, such that:

1. ConcreteCreator must inherit from Creator.
2. ConcreteProduct must inherit from Product, such that the inheritance is collapsible.
3. ConcreteCreator must be concrete.
4. ConcreteProduct must be concrete.

Figure A.1: Factory Method



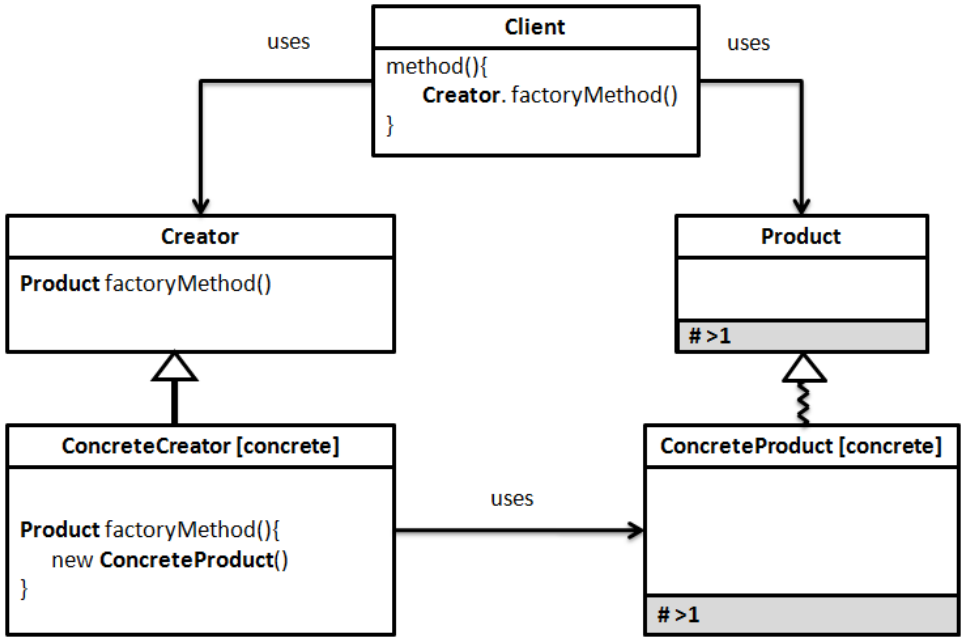
5. Creator must contain a method (*factoryMethod*) that returns a Product.

6. The implementation of the method (*factoryMethod*) in ConcreteCreator must create a new object of type ConcreteProduct.

7. Client must contain a method that calls the method (*factoryMethod*) in Creator.

A.1.2 Abstract Factory

Figure A.2: Abstract Factory



The Abstract Factory design pattern is a collection of Factory Method design patterns. Therefore, its model is similar to the Factory Method model described before. Moreover, there should be at least two Factory Method design pattern instances. This requires having more than one ConcreteProduct, and more than one Product. The Abstract Factory design pattern consists of five roles: Client, Creator, ConcreteCreator, Product, and ConcreteProduct, such that:

1. ConcreteCreator must inherit from Creator.

2. ConcreteProduct must inherit from Product, such that the inheritance is

collapsible.

3. ConcreteCreator must be concrete.

4. ConcreteProduct must be concrete.

5. Creator must contain a method (*factoryMethod*) that returns a Product.

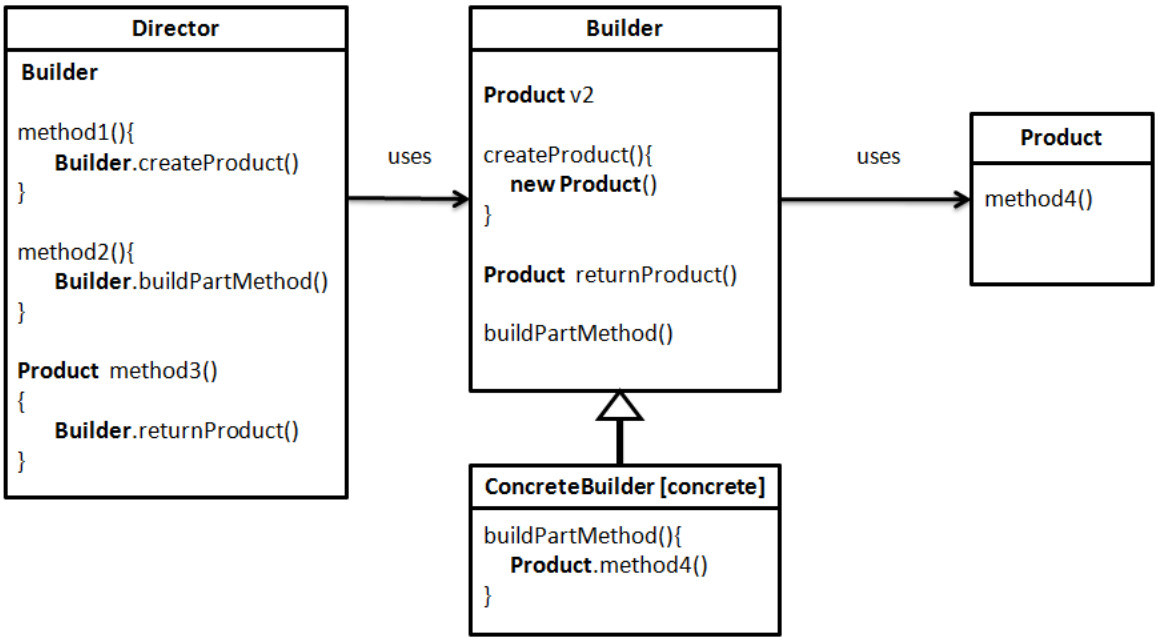
6. The implementation of the method (*factoryMethod*) in ConcreteCreator must create a new object of type ConcreteProduct.

7. Client must contain a method that calls the method (*factoryMethod*) in Creator.

8. Post processing rule: For each Abstract Factory anchor instance, there must be at least two different Product and ConcreteProduct candidates.

A.1.3 Builder

Figure A.3: Builder



The Builder design pattern consists of four roles: Director, Builder, Concrete-Builder, and Product, such that:

1. Builder must contain a field of type Product.

2. Builder must contain a method (*createProduct*) that creates a new object of type Product.

3. Builder must contain a method (*returnProduct*) that returns a Product.

4. Builder must contain a method (*buildPartMethod*), such that the implementation of (*buildPartMethod*) in ConcreteBuilder calls a method in Product.

5. ConcreteBuilder must inherit from Builder.

6. ConcreteBuilder must be concrete.

7. Director must contain a reference to Builder, either by having a field of type Builder, or having Builder passed to it through method parameters or by calling another method that returns a Builder.

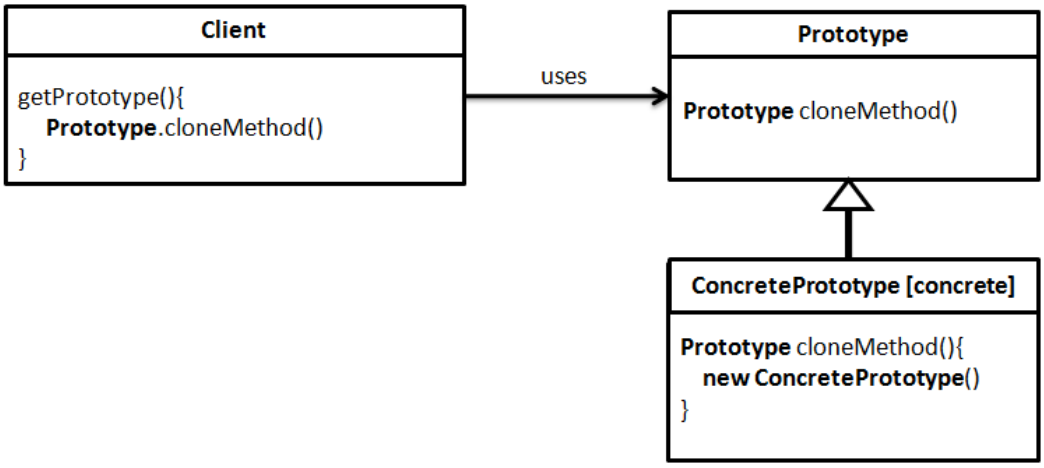
8. Director must contain a method that calls the method (*createProduct*) in Builder.

9. Director must contain a method that calls the method (*buildPartMethod*) in Builder.

10. Director must contain a method that returns a Product and calls the method (*returnProduct*) in Builder.

A.1.4 Prototype

Figure A.4: Prototype



The Prototype design pattern consists of three roles: Client, Prototype, and ConcreteProduct, such that:

1. ConcretePrototype must inherit from Prototype.

2. ConcretePrototype must be concrete.

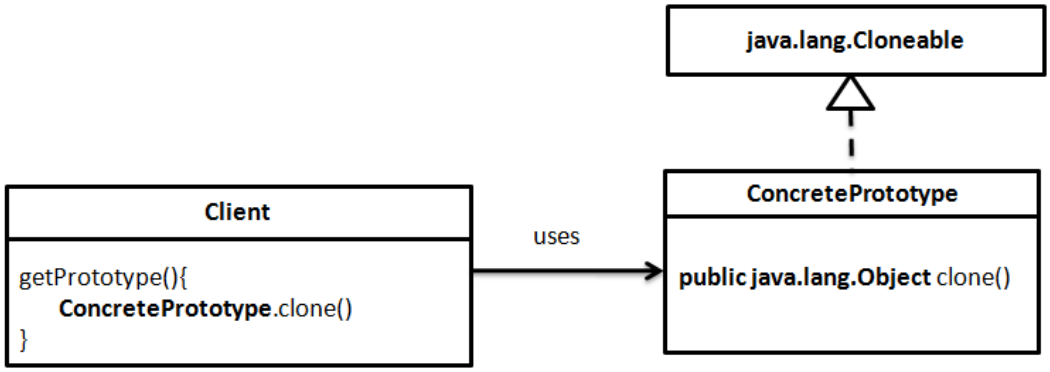
3. Prototype must contain a method (*cloneMethod*) that returns a Prototype.

4. The implementation of the method (*cloneMethod*) in ConcretePrototype must create a new object of type concretePrototype.

5. Client must contain a method *getPrototype* that calls the method (*cloneMethod*) in Prototype.

A.1.4.1 Prototype - Cloneable

Figure A.5: Prototype - Cloneable



The Prototype design pattern can also be implemented using the java.lang.Cloneable interface provided by Java. Java uses the clone() method of Object class to copy the state of one object to the other. Figure A.5 shows the definition used to detect the implementation of Prototype using the Cloneable interface.

The Prototype design pattern consists of three roles: Client, ConcretePrototype, and java.lang.Cloneable, such that:

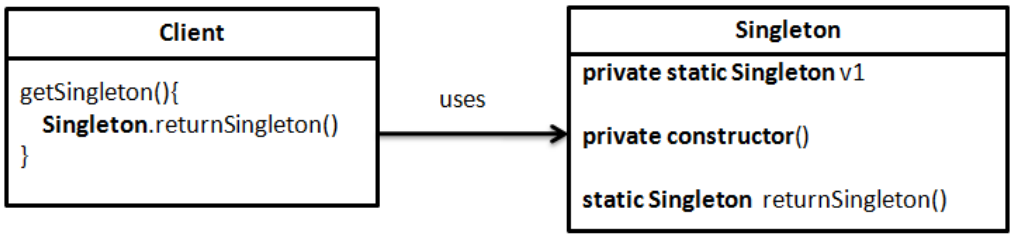
1. ConcretePrototype must implement from java.lang.Cloneable.

2. ConcretePrototype must contain a public method (*clone*) that returns a java.lang.Object.

3. Client must contain a method that calls the method (*clone*) in ConcretePrototype.

A.1.5 Singleton

Figure A.6: Singleton



The Singleton design pattern consists of two roles: Client and Singleton, such that:

1. Singleton must contain a private constructor.

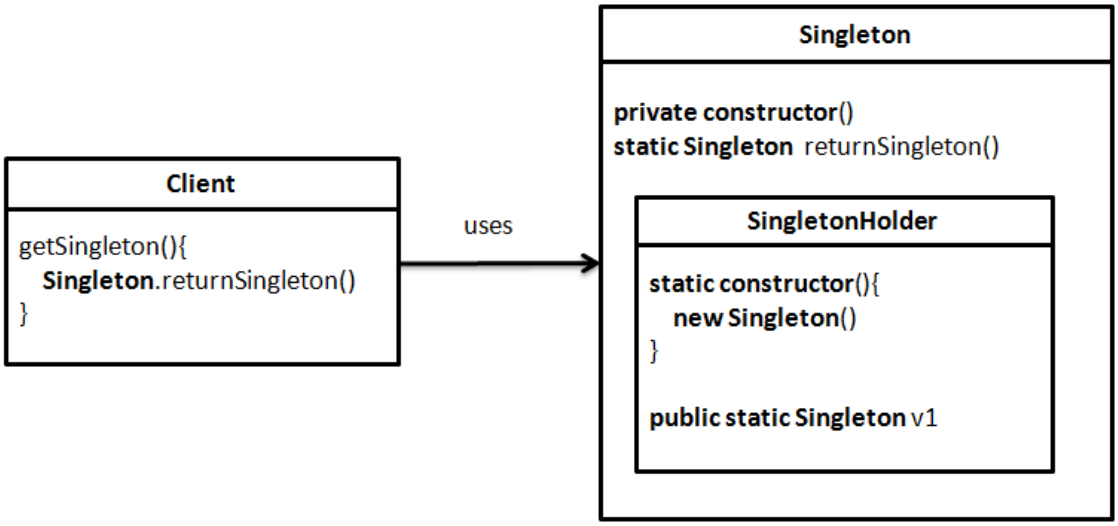
2. Singleton must contain a private static field of type Singleton.

3. Singleton must contain a static method (*returnSingleton*) that returns a Singleton.

4. Client must contain a method that calls the method (*returnSingleton*) in Singleton.

A.1.5.1 Singleton - Bill Pugh Solution

Figure A.7: Singleton - Solution of Bill Pugh



Another implementation of the Singleton design pattern was provided by Bill Pugh, a computer science researcher from the University of Maryland[17].Figure A.7 shows the definition used to detect the implementation of Singleton using the Bill Pugh solution.

The Singleton design pattern consists of three roles: Client, Singleton, and SingletonHolder, such that:

1. Singleton must contain a private constructor.

2. Singleton must contain a static method (*returnSingleton*) that returns a Singleton.

3. Singleton must contain SingletonHolder.

4. SingletonHolder must be static.

5. SingletonHolder must contain a static constructor that creates a new object of type Singleton.

6. SingletonHolder must contain a static public field of type Singleton.

7. Client must contain a method that calls the method (*returnSingleton*) in Singleton.

A.1.6 Adapter

There are two types of the Adapter design pattern: the Object Adapter and the less commonly used Class Adapter.

A.1.6.1 Object Adapter

Figure A.8: Object Adapter

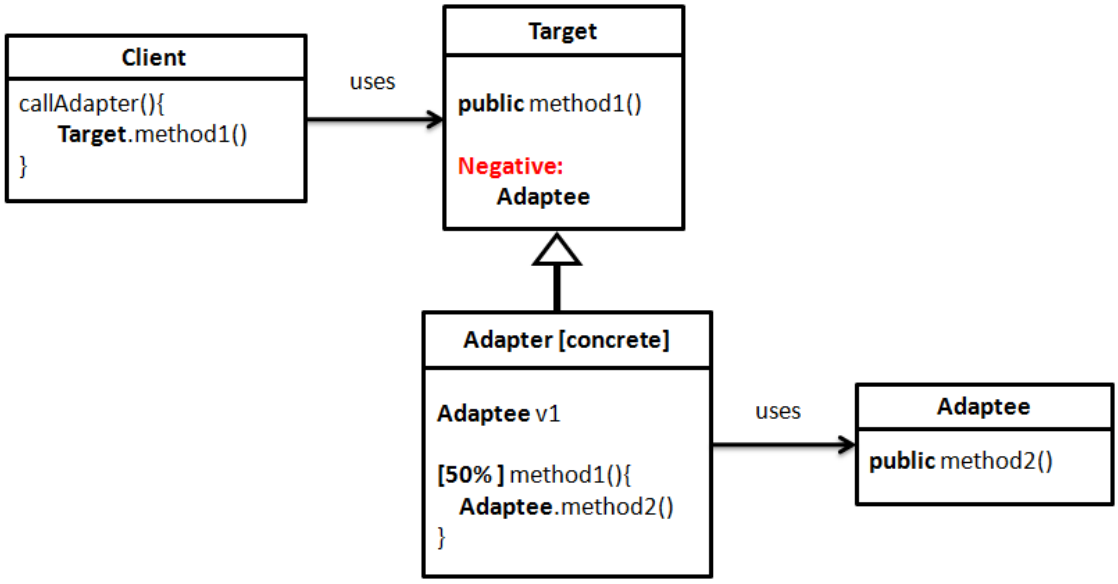


Figure A.8 shows the Object Adapter design pattern model. Object Adapter design pattern consists of four roles: Client, Target, Adapter, and Adaptee, such that:

1. Adapter must inherit from Target.

2. Adapter must be concrete.

3. Adapter must contain a field of type Adaptee.

4. Target must contain a public method that is called by Client.

5. Target must not contain a reference to Adaptee, either by having an object of type Adaptee, or having Adaptee passed to it through method parameters or by calling another method that returns an Adaptee.

6. In Adapter, the implementation of at least 50% of the methods inherited from Target must call a method inherited from Adaptee.

A.1.6.2 Class Adapter

Figure A.9: Class Adapter

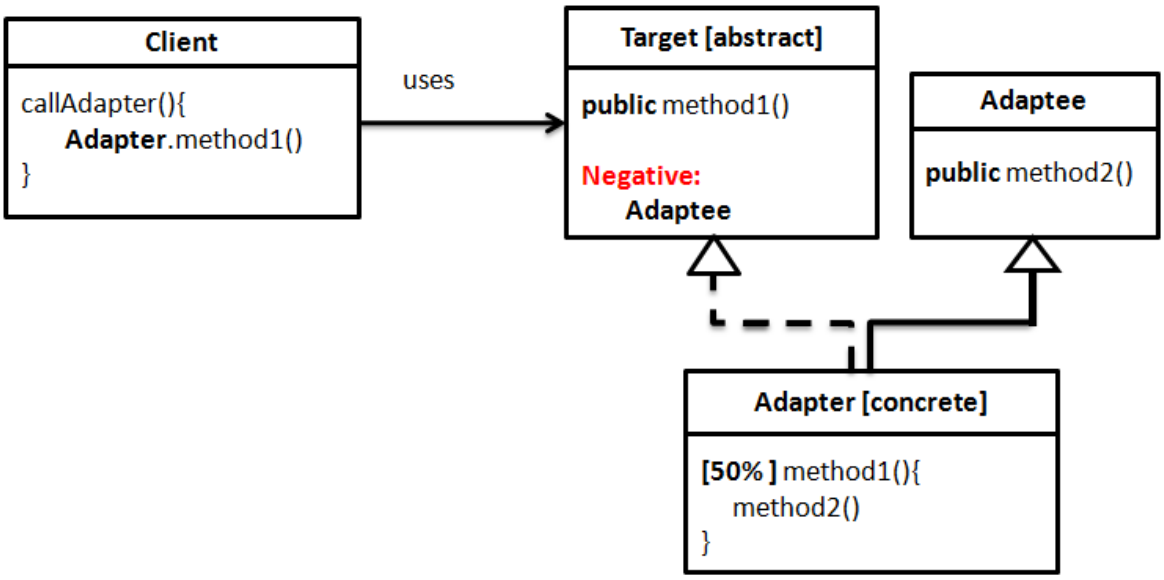


Figure A.9 shows the Class Adapter design pattern model. Class Adapter design pattern consists of four roles: Client, Target, Adapter, and Adaptee, such that:

1. Adapter must implement Target.

2. Adapter must inherit from Adaptee.

3. Adapter must be concrete.

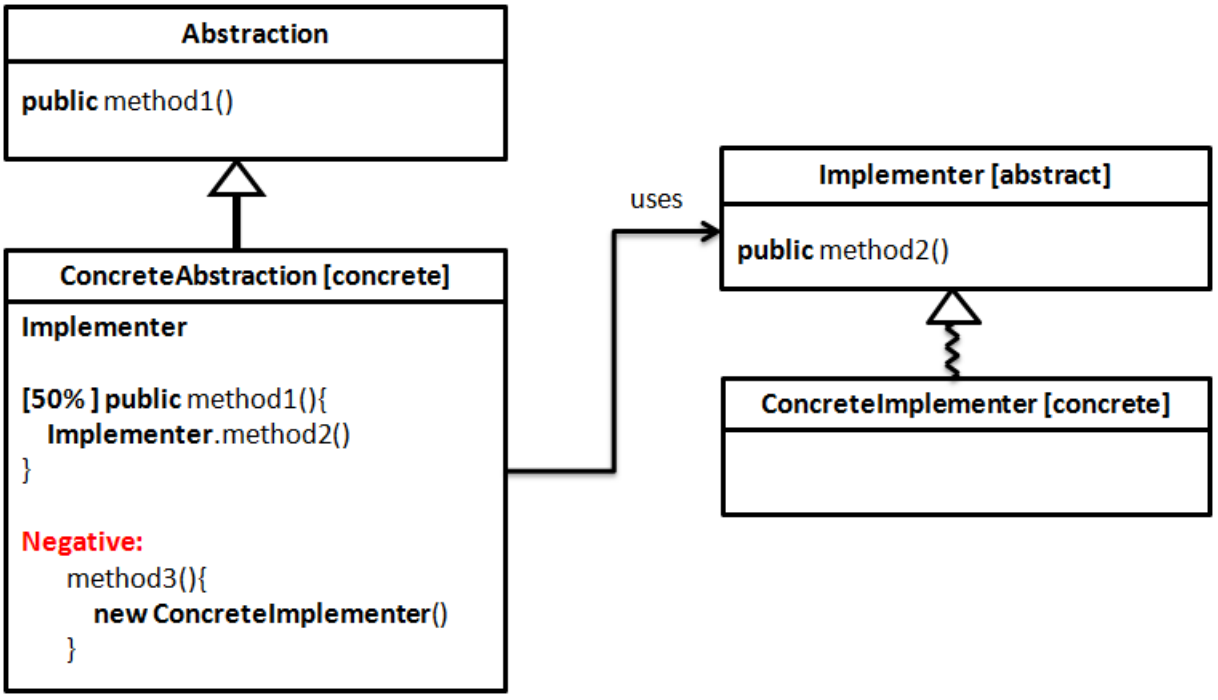
4. Target contains methods that are called by Client.

5. Target must not contain a reference to Adaptee, either by having an object of type Adaptee, or having Adaptee passed to it through method parameters or by calling another method that returns an Adaptee.

6. In Adapter, the implementation of at least 50% of the methods inherited from Target must call a method inherited from Adaptee.

A.1.7 Bridge

Figure A.10: Bridge



The Bridge design pattern consists of four roles: Abstraction, ConcreteAbstraction, Implementer, and ConcreteImplementer, such that:

1. Implementer is abstract.

2. ConcreteImplementer must inherit from Implementer, such that the inheritance is collapsible.

3. ConcreteImplementer must be concrete.

4. ConcreteAbstraction must inherit from Abstraction.

5. ConcreteAbstraction must be concrete.

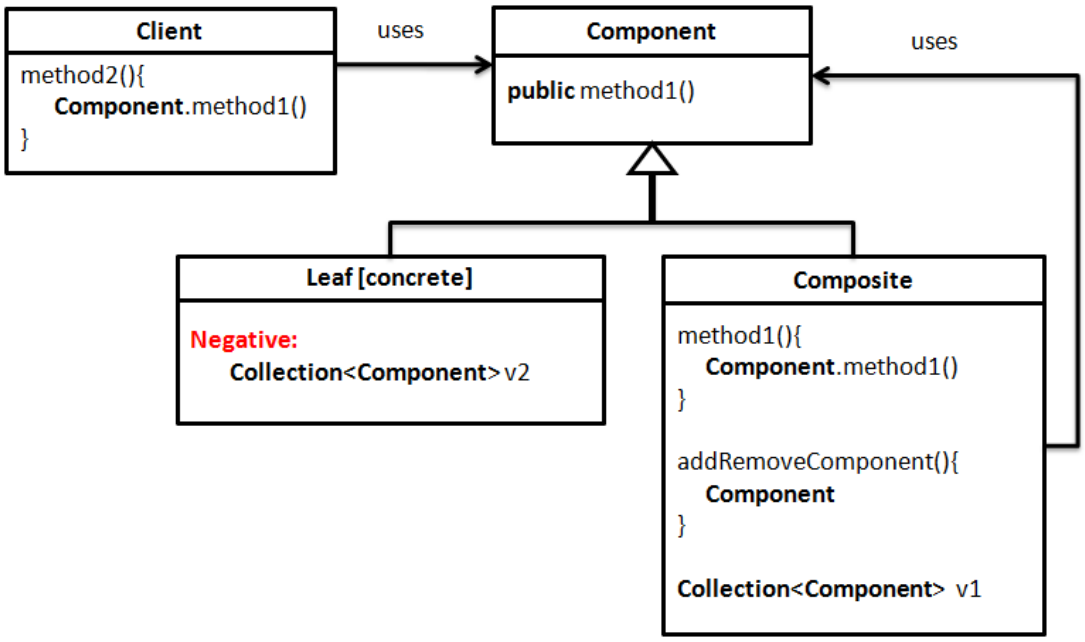
6. ConcreteAbstraction must contain a reference to Implementer, either by having a field of type Implementer, or by getting Implementer passed to it through method parameters or by calling another method that returns an Implementer.

7. ConcreteAbstraction must not contain a method that creates an object of type ConcreteImplementer.

8. In ConcreteAbstraction, the implementation of at least 50% of the public methods inherited from Abstraction must call a public method in Implementer.

A.1.8 Composite

Figure A.11: Composite



The Composite design pattern consists of four roles: Client, Component, Composite, and Leaf, such that:

1. Component must contain a public method (*method1*) that is called by Client.

2. Composite must inherit from Component.

3. The implementation of the method (*method1*) in Composite must call the method (*method1*) on a reference of type Component.

4. Composite gets Component passed to it through method parameters, or by calling another method that returns a Component.

5. Composite contains a collection of Components.

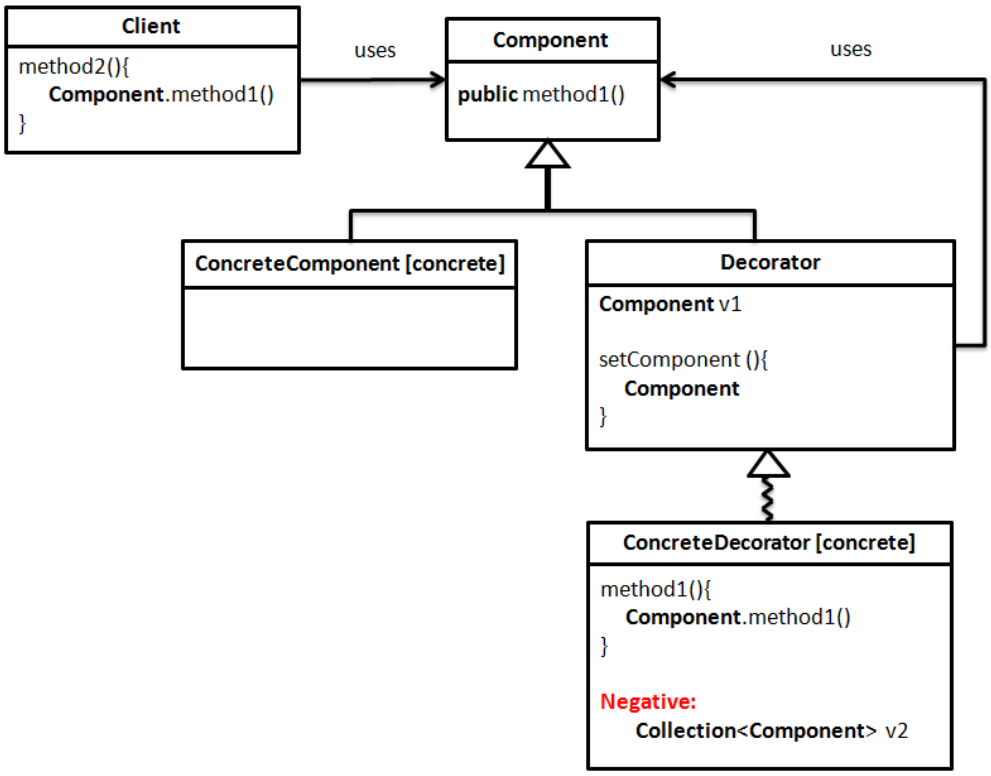
6. Leaf must inherit from Component.

7. Leaf must be concrete.

8. Leaf must not contain a collection of Components.

A.1.9 Decorator

Figure A.12: Decorator



The Decorator design pattern consists of five roles: Client, Component, ConcreteComponent, Decorator, and ConcreteDecorator, such that:

1. Component must contain a public method (*method1*) that is called by Client.

2. Decorator must inherit from Component.

3. Decorator must contain a field of type Component.

4. Decorator gets Component passed to it through method parameters, or by calling another method that returns a Component.

5. ConcreteDecorator must inherit from Decorator, such that the inheritance is collapsible.

6. ConcreteDecorator must be concrete.

7. ConcreteDecorator must not contain a collection of Components.

8. The implementation of *method1* in ConcreteDecorator calls the method (*method1*) on a reference of type Component.

9. ConcreteComponent must inherit from Component.

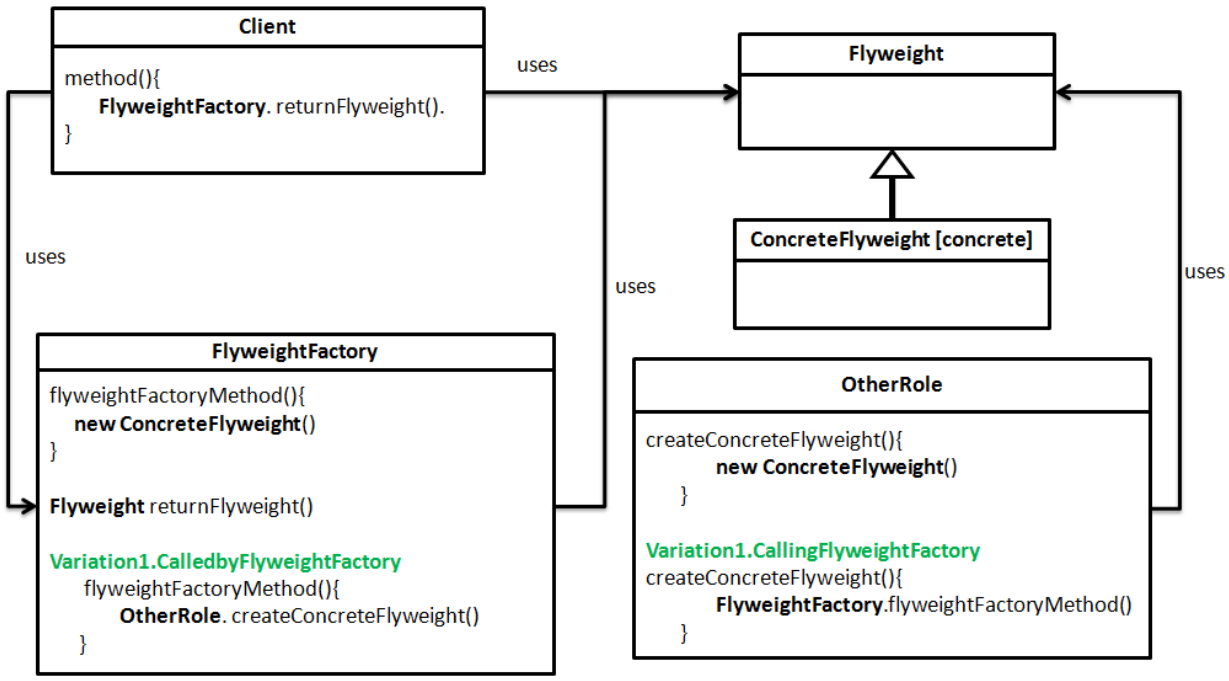
10. ConcreteComponent must be concrete.

A.1.10 Facade

The Facade pattern provides a simplified interface to a large set of information, and hides other unwanted information. To detect this pattern, one needs to have access to information related to the evolution of the system's architecture. The Facade pattern is more of an architectural pattern rather than a design pattern, which makes it hard to define this pattern statically with our approach. Moreover, the Facade design pattern defines only class level interaction between participants. Hence, no fine-grained rules could be set for the detection process.

A.1.11 Flyweight

Figure A.13: Flyweight



The Flyweight design pattern consists of four roles [7]: Client, FlyweightFactory, Flyweight, and ConcreteFlyweight, such that:

1. ConcreteFlyweight must inherit Flyweight.

2. ConcreteFlyweight must be concrete.

[7]A fifth role called OtherRole is shown in the definition in Figure A.13. OtherRole does not refer to an actual role in the Flyweight design pattern and is not part of the design pattern detected roles. It refers to a role that uses the Flyweight design pattern. Therefore, OtherRole is not reported in the design pattern candidates

3. FlyweightFactory must contain a method (*flyweightFactoryMethod*) that creates a new object of type Flyweight.

4. FlyweightFactory must contain a method (*returnFlyweight*) that returns a Flyweight.

5. Client must contain a method that calls the method (*returnFlyweight*) in FlyweightFactory.

6. OtherRole contains a method (*createConcreteFlyweight*) that creates a a new object ConcreteFlyweight.

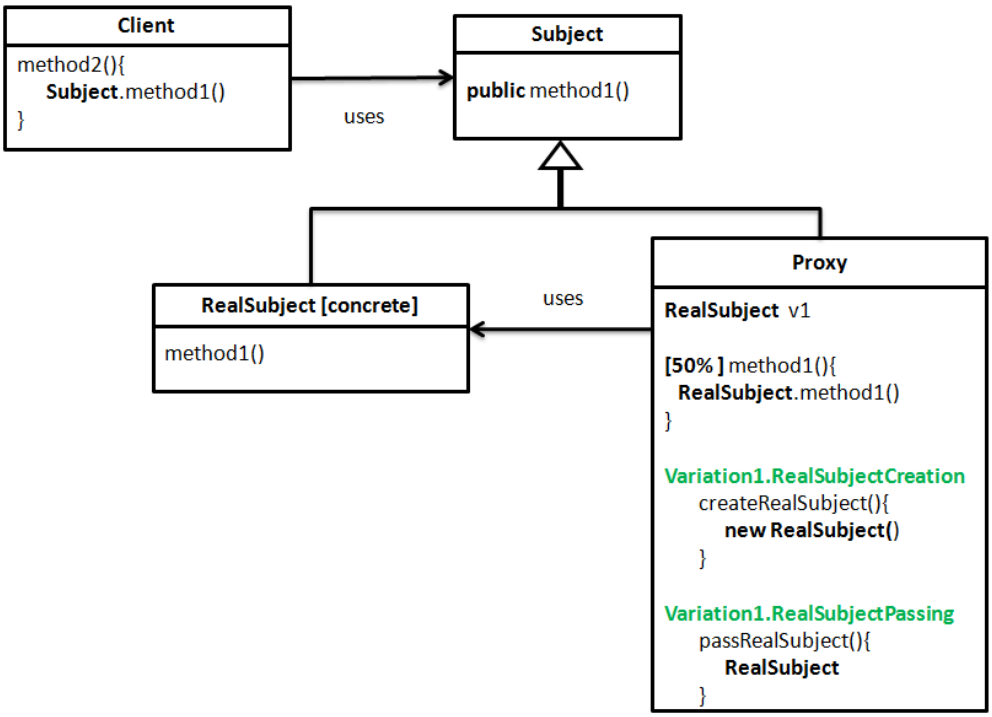
7. Variation1:

    - *CalledbyFlyweightFactory*: The method (*createConcreteFlyweight*) is called by the method (*flyweightFactoryMethod*) in FlyweightFactory.
    - *CalledbyFlyweightFactory*: The method (*createConcreteFlyweight*) calls the method (*flyweightFactoryMethod*) in FlyweightFactory.

A.1.12 Proxy

There are two types of the Proxy design pattern: Proxy and Proxy2. The Proxy

Figure A.14: Proxy



design pattern consists of four roles: Client, Subject, RealSubject, and Proxy, such that:

1. Subject must contain a public method that is called by Client.

2. RealSubject must inherit from Subject.

3. RealSubject must be concrete.

4. Proxy must inherit from Subject.

5. Proxy must contain a field of type RealSubject.

6. The implementation of at least 50% of the Subject methods in Proxy calls the methods implementation of Subject methods in RealSubject.

7. Variation1:

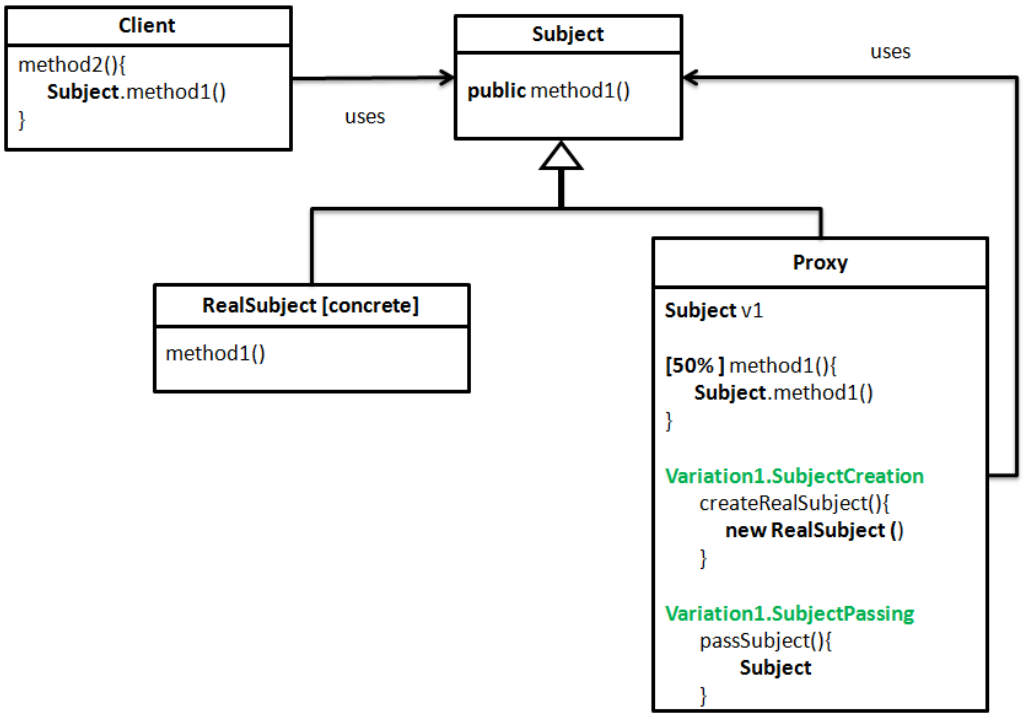
   - *RealSubjectCreation*: Proxy must contain a method that creates a new object of type RealSubject.
   - *RealSubjectPassing*: Proxy must get a RealSubject passed to it through method parameters, or by calling another method that returns a RealSubject.

A.1.12.1 Proxy2

In Proxy2, the Proxy role has an association to the Subject role instead of RealSubject. This kind of Proxy variation has been reported by Gnter Kniesel and Alex Binun from University of Bonn. Figure 5.1 shows the UML diagram of Proxy2. More information about this variation is available at `http://java.uom.gr/~nikos/pattern-detection.html`.

The Proxy2 design pattern consists of four roles: Client, Subject, RealSubject, and Proxy, such that:

Figure A.15: Proxy2



1. Subject must contain a public method that is called by Client.

2. RealSubject must inherit from Subject.

3. RealSubject must be concrete.

4. Proxy must inherit from Subject.

5. Proxy must contain a field of type Subject.

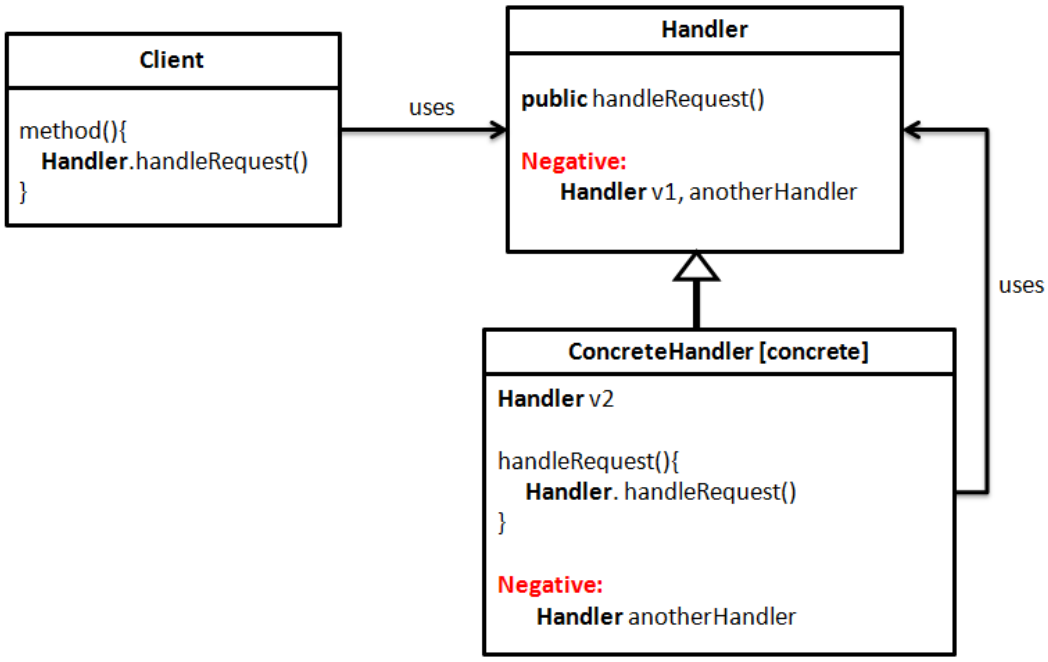
6. The implementation of at least 50% of the Subject methods in Proxy calls the abstract methods of Subject.

7. Variation1:

    - *RealSubjectCreation*: Proxy must contain a method that creates a new object of type RealSubject.
    - *SubjectPassing*: Proxy must get a Subject passed to it through method parameters, or by calling another method that returns a Subject.

A.1.13 Chain of Responsibility

Figure A.16: Chain of Responsibility



The Chain of Responsibility design pattern consists of three roles: Client, Handler, and ConcreteHandler, such that:

1. Handler must contain a public method (*handleRequest*) that is called by Client.

2. Handler must not contain more than one field of type Handler.

3. ConcreteHandler must inherit from Handler.

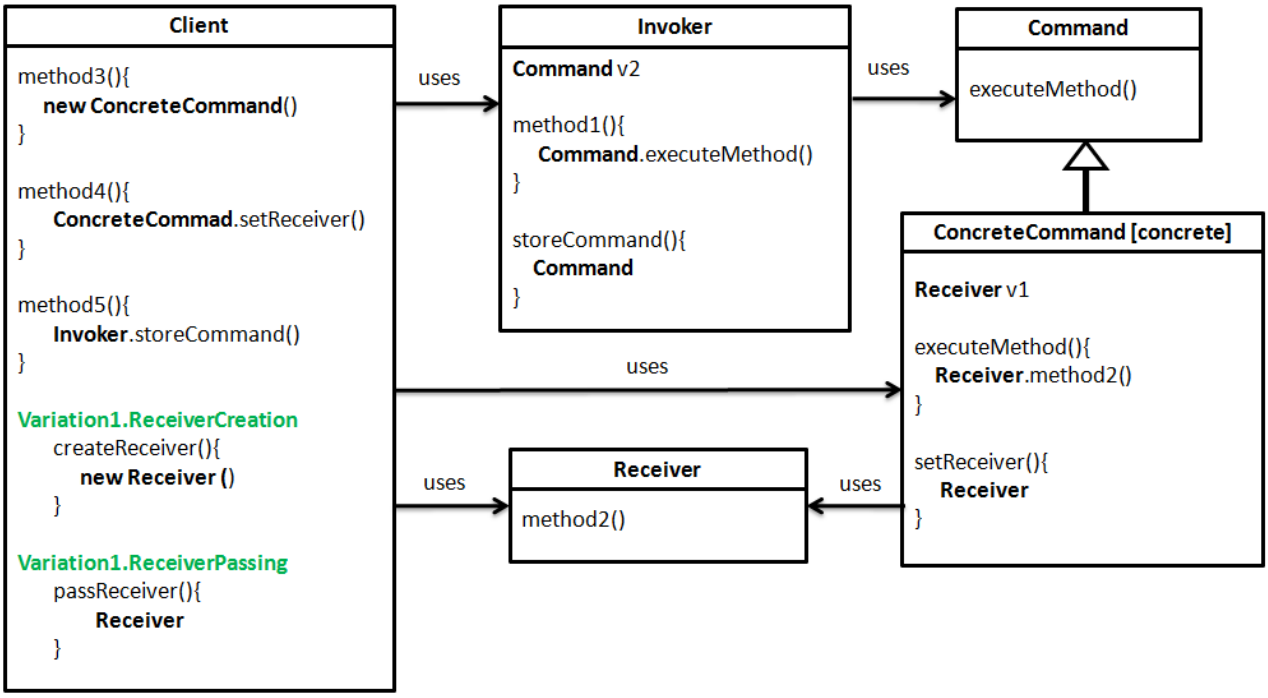
4. ConcreteHandler must be concrete.

5. The implementation of *handleRequest* in ConcreteHandler must call the method *handleRequest* on a reference of type Handler.

6. ConcreteHandler must contain one field of type Handler.

A.1.14 Command

Figure A.17: Command



The Command deign pattern consists of five roles: Client, Receiver, Invoker, Command, and ConcreteCommand, such that:

1. ConcreteCommand must inherit from Command.

2. ConcreteCommand must be concrete.

3. Command must contain a method (*executeCommand*) that is called by Invoker.

4. The implementation of the method (*executeCommand*) in ConcreteCommand calls a method in Receiver.

5. ConcreteCommand must contain a field of type Receiver.

6. ConcreteCommand must contain a method (*setReceiver*) that gets Receiver passed to it through method parameters, or by calling another method that returns a Receiver.

7. Invoker must contain a field of type Command.

8. Invoker must contain a method (*storeCommand*) gets Command passed to it through method parameters, or by calling another method that returns a Command.

9. Client must contain a method that creates a new object of type ConcreteCommand.

10. Client must contain a method that calls the method (*setReceiver*) in ConcreteCommand.

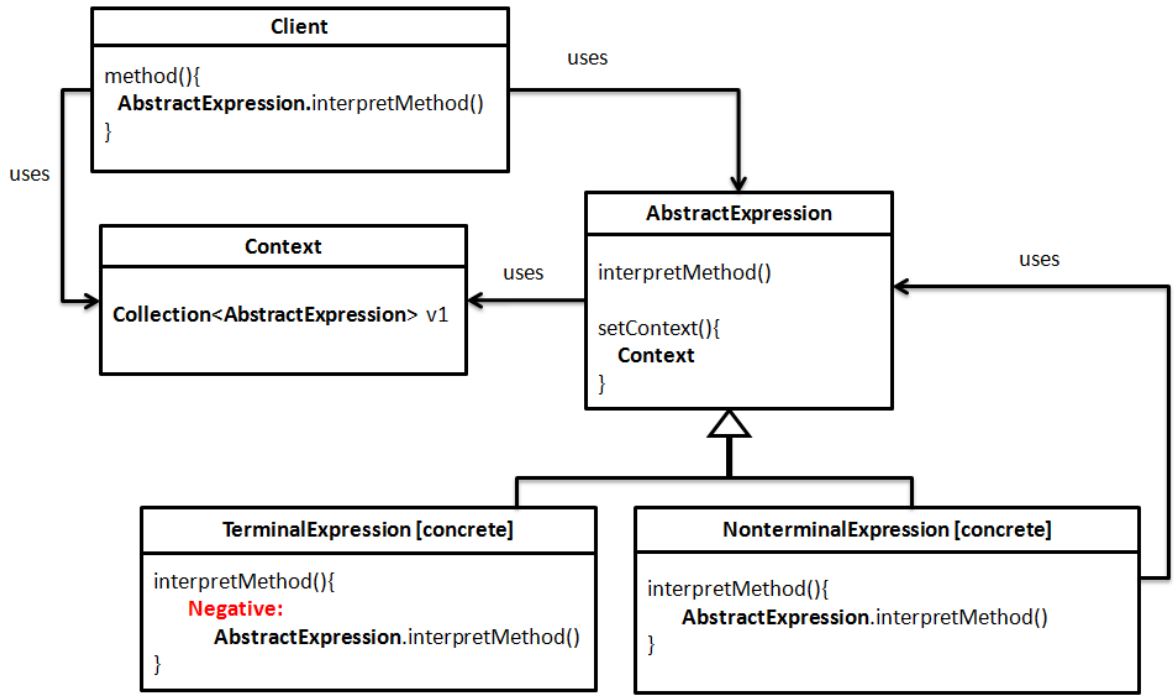
11. Client must contain a method that calls the method (*storeCommand*) in Invoker.

12. Variation1:

- *ReceiverCreation*: Client must contain a method that creates a new object of type Receiver.
- *ReceiverPassing*: Client must get a Receiver passed to it through method parameters, or by calling another method that returns a Receiver.

A.1.15 Interpreter

Figure A.18: Interpreter



The Interpreter design pattern consists of five roles: Client, Context, AbstractExpression, NonterminalExperssion, and TerminalExpression, such that:

1. AbstractExpression must contain a method (*interpretMethod*) that is called by Client.

2. AbstractExpression must get Context passed to it through method parameters, or by calling another method that returns a Context.

3. Context must contain a collection of AbstractExpression.

4. TerminalExpression must inherit from AbstractExpression.

5. TerminalExpression in concrete.

6. NonterminalExpression must inherit from AbstractExpression.

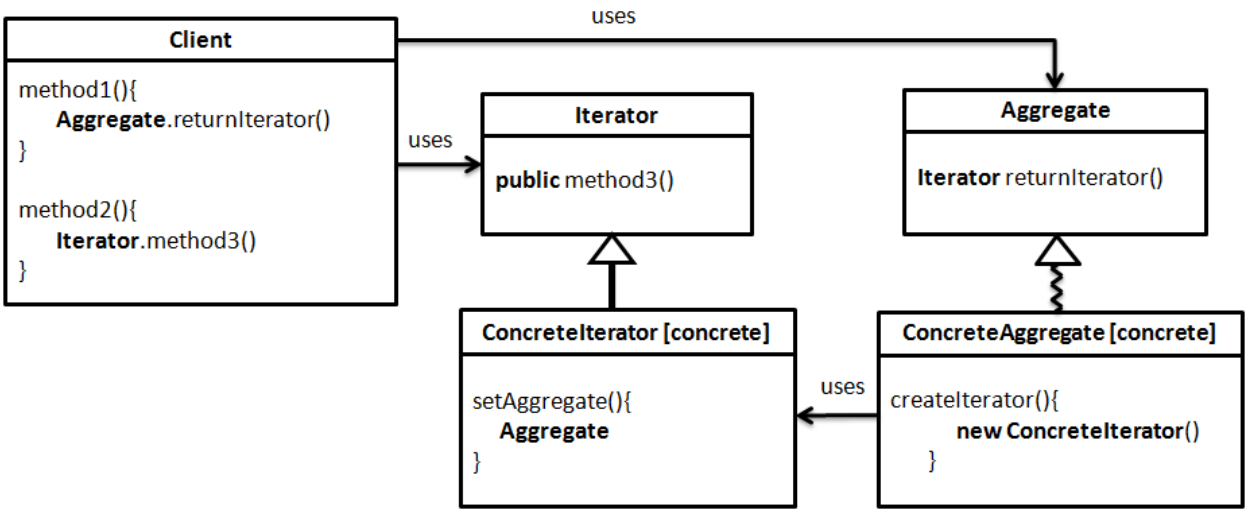
7. NonterminalExpression must be concrete.

8. The implementation of the method (*interpretMethod*) in NonTerminalExpression must call the method (*interpretMethod*) on a reference of type AbstractExpression.

9. The implementation of the method (*interpretMethod*) in TerminalExpression must not call the method (*interpretMethod*) on a reference of type AbstractExpression.

A.1.16 Iterator

Figure A.19: Iterator



The Iterator design pattern consists of five roles: Client, Iterator, ConcreteIterator, Aggregate, and ConcreteAggregate, such that:

1. Aggregate must contain a method (*returnIterator*) that returns Iterator.

2. Client must contain a method that calls the method (*returnIterator*) in Aggregate.

3. CocnreteAggregate must inherit from Aggregate, such that the inheritance is collapsible.

4. ConcreteAggregate must be concrete.

5. ConcreteAggregate must contain a method that creates a new object of type ConcreteIterator.

6. Iterator must contain a public method that is called by Client.

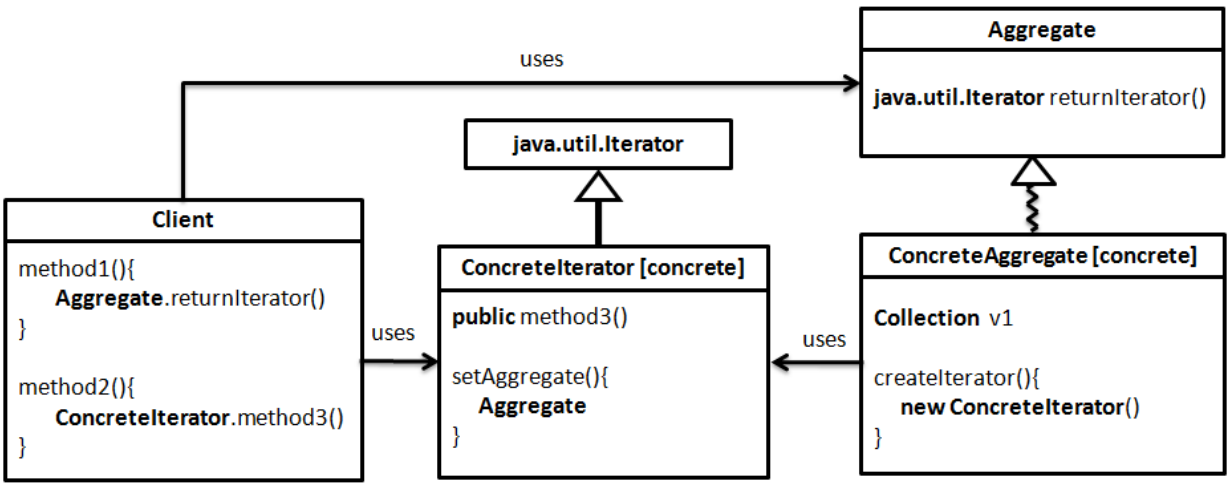
7. ConcreteIterator must inherit from Iterator.

8. ConcreteIterator must be concrete.

9. concreteIterator must get Aggregate passed to it through method parameters, or by calling another method that returns an Aggregate.

A.1.16.1 Iterator - Java Implementation

Figure A.20: Iterator - Using Java Iterators



The Iterator design pattern can be also implemented using the java.util.Iterator interface, which provides an iterator over a collection. Figure A.20 presents the implementation of Iterator design pattern using java.util.Iterator. There are four roles: java.util.Iterator, ConcreteIterator, Aggregate, and ConcreteAggregate, such that:

1. Aggregate must contain a method (*returnIterator*) that returns java.util.Iterator.

2. Client must contain a method that calls the method (*returnIterator*) in Aggregate.

3. CocnreteAggregate must inherit from Aggregate, such that the inheritance is collapsible.

4. ConcreteAggregate must be concrete.

5. ConcreteAggregate must contain a method that creates a new object of type ConcreteIterator.

6. ConcreteIterator must inherit from java.util.Iterator.

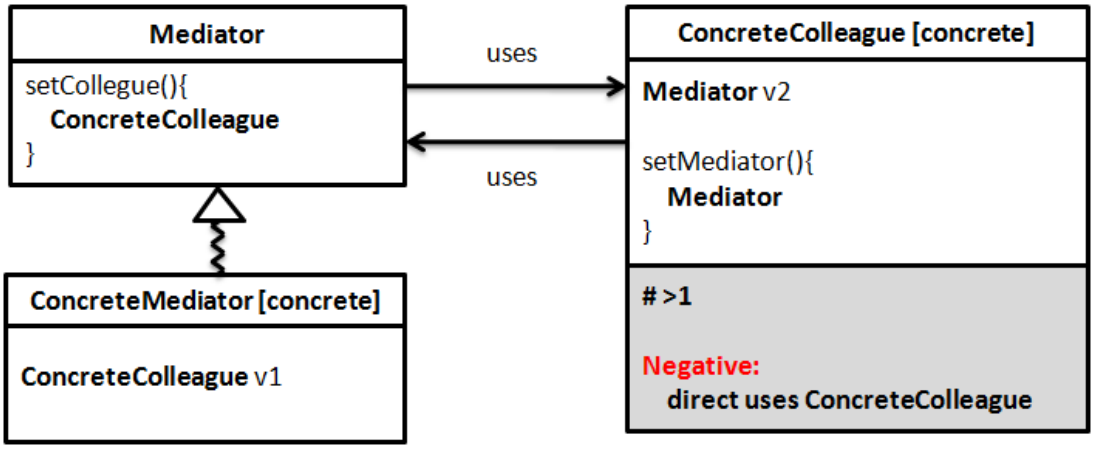
7. ConcreteIterator must contain a public method that is called by Client.

8. ConcreteIterator must be concrete.

9. concreteIterator must get Aggregate passed to it through method parameters, or by calling another method that returns an Aggregate.

A.1.17 Mediator

Figure A.21: Mediator



The Mediator design pattern requires that there are more than one ConcreteColleague communicating with each other using the ConcreteMediator. Therefore, at least two ConcreteColleagues must be detected. The Mediator design pattern consists of three roles: Mediator, ConcreteMediator, and ConcreteColleague, such that:

1. Mediator must get ConcreteColleague passed to it through method parameters, or by calling another method that returns a ConcreteColleague.

2. ConcreteMediator must inherit from Mediator, such that the inheritance is collapsible.

3. ConcreteMediator must be concrete.

4. ConcreteMediator must contain a field of type ConcreteColleague.

5. ConcreteColleague must be concrete.

6. ConcreteColleague must contain a fields of type Mediator.

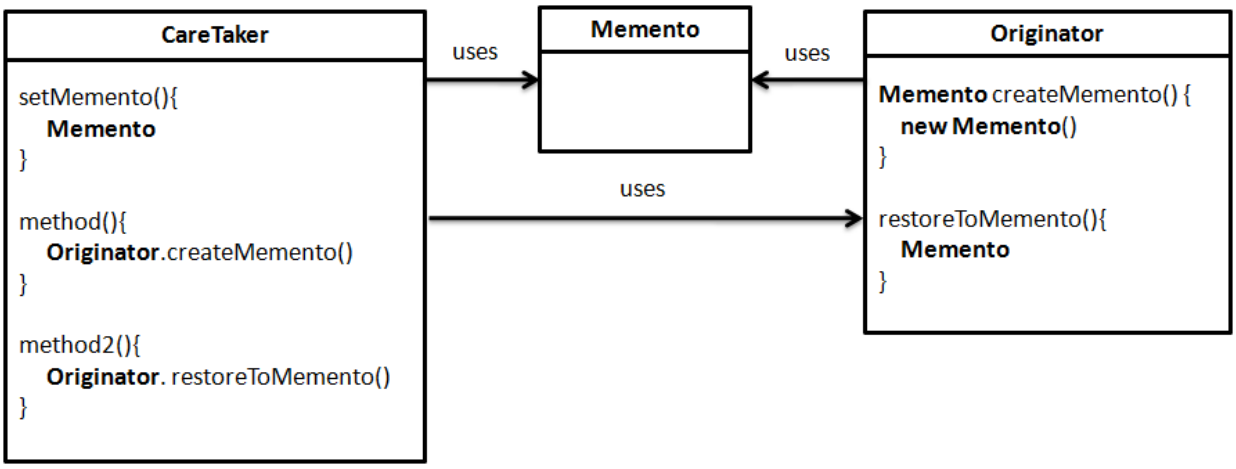
7. ConcreteColleague must get Mediator passed to it through method parameters, or by calling another method that returns a Mediator.

8. Post processing rule: There must at be least two ConcreteColleague candidates in the same Mediator anchor instance.

9. Post processing rule: ConcreteColleague must not directly use another ConcreteColleague .

A.1.18 Memento

Figure A.22: Memento



The Memento design pattern consists of three roles: CareTaker, Memento, and Originator, such that:

1. Originator must contain a method (*createMemento*) that creates a new object of type Memento, and returns a Memento.

2. Originator must get Memento passed to it through method parameters, or by calling another method that returns a Memento.

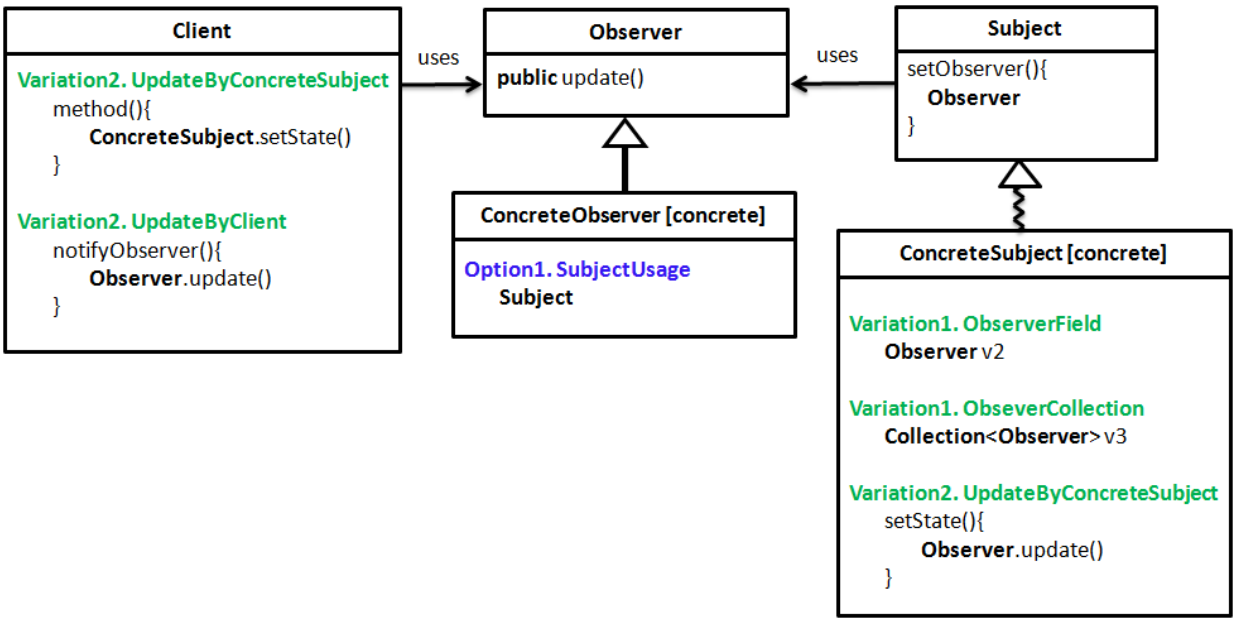
3. Originator must contain a method that calls the method (*restorToMemento*) in CareTaker.

4. CareTaker must contain a method (*restorToMemento*) that gets Memento passed to it through method parameters, or by calling another method that returns a Memento.

5. CareTaker must contain a method that calls the method (*createMemento*) in Originator.

A.1.19 Observer

Figure A.23: Observer



The Observer design pattern consists of four roles: Observer, ConcreteObserver, Subject, and ConcreteSubject, such that:

1. Subject must contain a method (*setObserver*) that gets Observer passed to it through method parameters, or by calling another method that returns an Observer.

2. ConcreteSubject must inherit from Subject, such that the inheritance is collapsible.

3. ConcreteSubject must be concrete.

4. ConcreteObserver must inherit from Observer.

5. ConcreteObserver must be concrete.

6. Variation1:

   - *ObserverField*: ConcreteSubject must contain a field of type Observer.
   - *ObserverCollection*: ConcreteSubject must contain a collection of type Observer.

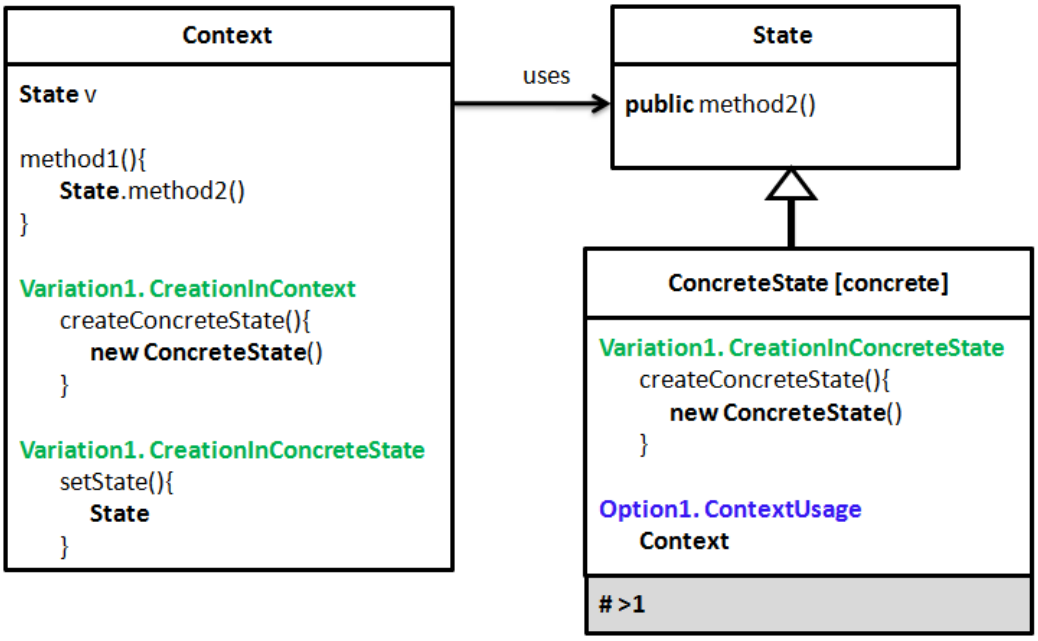
7. Variation2:

   - *UpdateByClient*: Client must contain a method that calls a public method (*update*) in Observer.
   - *UpdateByConcreteSubject*: Client must contain a method that calls a method (*setState*) in ConcreteState, such that the method (*setState*) calls a public method (*update*) in Observer.

8. Option1 (*SubjectUsage*): ConcreteObserver must contain a reference to Subject either by having a field of type Subject, or by getting Subject passed to it through method parameters, or by calling another method that returns a Subject.

A.1.20 State

Figure A.24: State



The State design pattern allows an object to alter its behavior when its internal state changes. Therefore, we require that at least two ConcreteState candidates are detected. The State design pattern consists of three roles: Context, State, and ConcreteState, such that:

1. Context must contain a field of type State.

2. Context must contain a method that calls a public method in State.

3. ConcreteState must inherit from State.

4. ConcreteState must be concrete.

5. Variation1:

   - *CreationInContext*: Context must contain a method that creates a new object of type ConcreteState.
   - *CreationInConcreteState*: ConcreteState must contain a method that creates a new object of type ConcreteState, and Context must get State passed to it through method parameters, or by calling another method that returns a State. The *CreationInConcreteState* variation is shared among two roles, Context and ConcreteState. Therefore, the two rules share the same group number and rule name. For this variation to be satisfied, both parts need to be true.

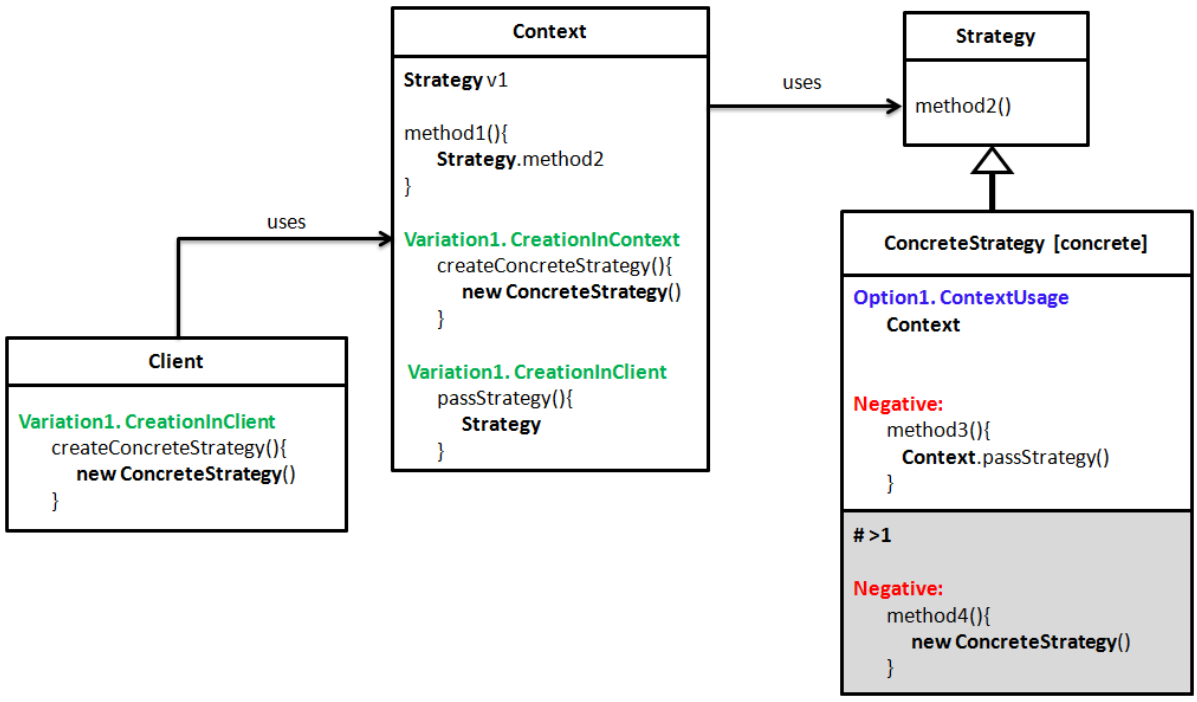
6. Option1 (*ContextUsage*): ConcreteState must contain a reference to Context by having a field of type Context, or by getting Context passed to it through method parameters or by calling another method that returns a Context.

7. Post processing rule: There must be least two ConcreteState candidates in the same State anchor instance.

A.1.21 Strategy

. The Strategy design pattern defines a family of algorithms (strategies), encap-

Figure A.25: Strategy



sulates each one, and makes them interchangeable. Therefore, we require that at least two ConcreteStrategy candidates are detected. The Strategy design pattern consists of four roles: Strategy, ConcreteStrategy, Context, and Client, such that:

1. Context must contain a field of type Strategy.

2. Context must contain a method that calls a public method in Strategy.

3. Variation1:

    - *CreationInContext*: Context must contain a method that creates a new object of type ConcreteStrategy.
    - *CreationInClient*: Client must contain a method that creates a new object of type ConcreteStrategy, and Context must get Strategy passed to the method (*passStrategy*) through method parameters, or by calling another method that returns a Strategy. The *CreationInClient* variation is shared among two roles, Context and Client. Therefore, the two rules share the same group number and rule name. For this variation to be satisfied, both parts need to be true.

4. ConcreteStrategy must inherit from Strategy.

5. ConcreteStrategy must be concrete.

6. Option1 (*ContextUsage*): ConcreteStrategy must contain a reference to Context by having a field of type Context, or by getting Context passed to it through method parameters or by calling another method that returns a Context.

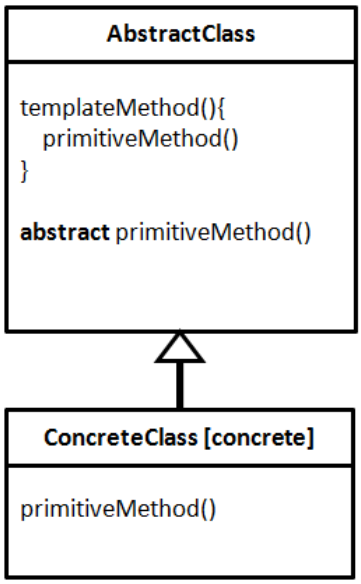
7. ConcreteStrategy must not contain a method that calls the method (*passStrategy*) in Context.

8. Post processing rule: There must at be least two ConcreteStrategy candidates in the same Strategy anchor instance.

9. Post processing rule: For every Strategy anchor instance, ConcreteStrategy candidates must not contain a method that creates a new object of type ConcreteStrategy.

A.1.22 Template Method

Figure A.26: Template



The Template Method design pattern consists of two roles: AbstractClass and ConcreteClass, such that:

1. ConcreteClass must inherit from AbstractClass.

2. ConcreteClass must be concrete.

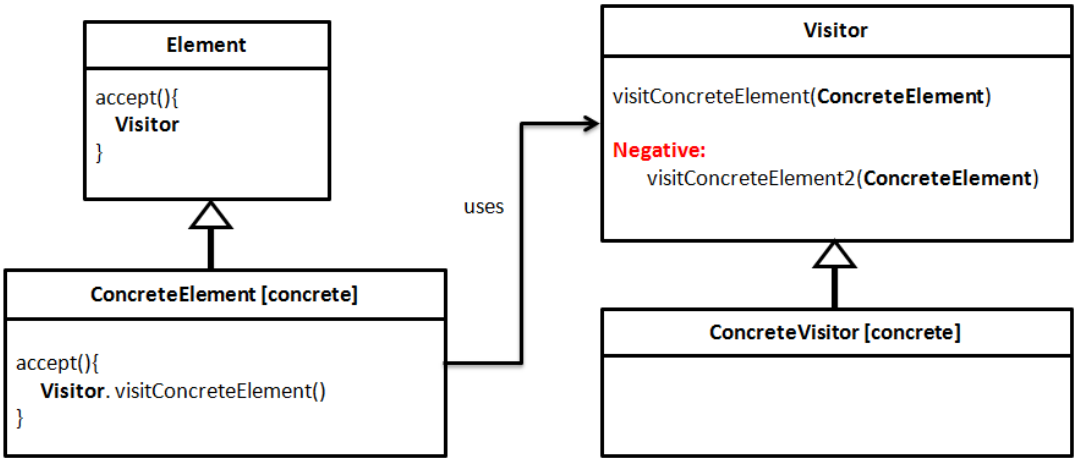
3. AbstractClass must contain an abstract method (*primitiveMethod*).

4. AbstractClass must contain a method (*templateMethod*) that calls the method (*primitiveMethod*).

5. ConcreteClass must override the method (*primitiveMethod*) in AbstractClass.

A.1.23 Visitor

Figure A.27: Visitor



The Visitor design pattern consists of four roles: Element, ConcreteElement, Visitor, and ConcreteVisitor, such that:

1. Visitor must contain a method (*visitConcreteElement*) that has a parameter of type ConcreteElement.

2. Visitor must not contain more than one method, i.e. *visitConcreteElement2* that has a parameter of type ConcreteElement.

3. ConcreteVisitor must inherit from Visitor.

4. ConcreteVisitor must be concrete.

5. Element must contain a method (*accept*) that gets Visitor passed to it through method parameters, or by calling another method that returns a Visitor.

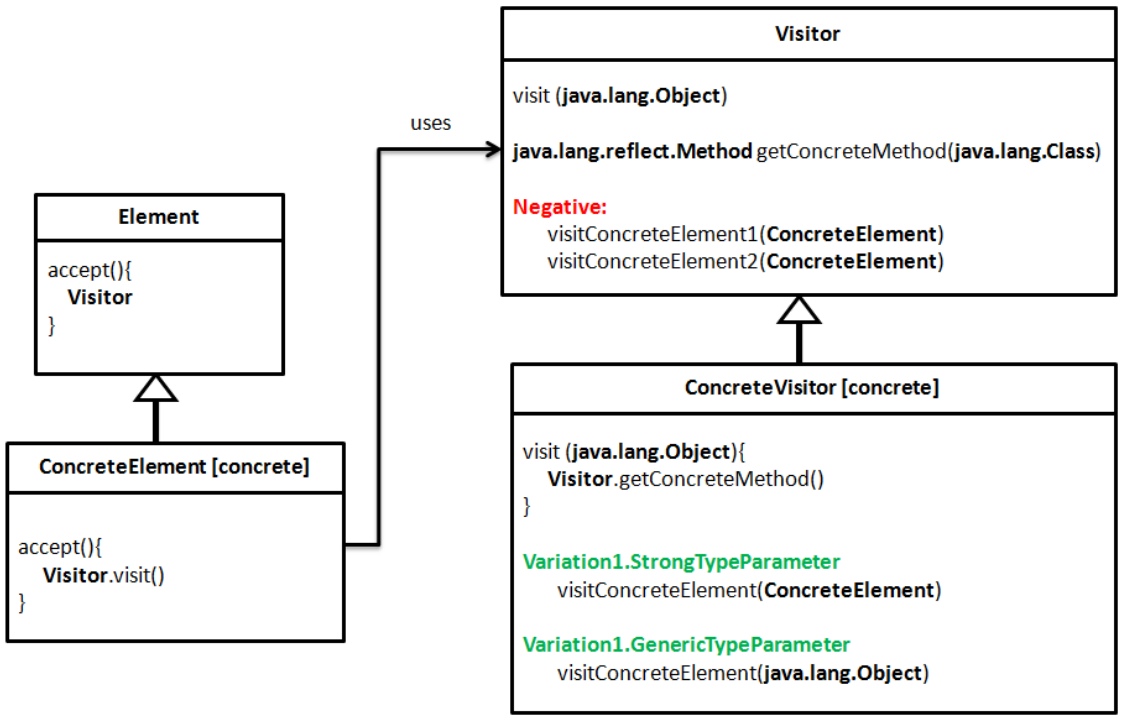
6. ConcreteElement must inherit from Element.

7. ConcreteElement must be concrete.

8. The implementation of the method *accept* in ConcreteElement must call the method *visitConcreteElement* in Visitor.

A.1.23.1 Visitor - Reflective

Figure A.28: Visitor - Reflective



The Visitor design pattern can be implemented using reflection in Java. This implementation is called Reflective Visitor. The ReflectiveVisitor design pattern consists of four roles: Element, ConcreteElement, Visitor, and ConcreteVisitor, such that:

1. Visitor must contain a method (*visit*) that has a parameter of type java.lang.Object.

2. Visitor must contain a method (*getConcreteMethod*) that has a parameter of

type java.lang.Class passed to it, and returns java.lang.reflect.Method.

3. Visitor must not contain more than one method, i.e. *visitConcreteElement1* and *visitConcreteElement2*, that have a parameter of type ConcreteElement.

4. ConcreteVisitor must inherit from Visitor.

5. ConcreteVisitor must be concrete.

6. The implementation of the method (*visit*) in ConcreteVisitor must call the method (*getConcreteMethod*) in Visitor.

7. Element must contain a method (*accept*) that gets Visitor passed to it through method parameters, or by calling another method that returns a Visitor.

8. ConcreteElement must inherit from Element.

9. ConcreteElement must be concrete.

10. The implementation of the method *accept* in ConcreteElements must call the method *visitConcreteElement* in Visitor.

11. Variation1:

    - *StrongTypeParameter*: ConcreteVisitor must contain a method has a parameter of type ConcreteElement.

- *GenericTypeParameter*: ConcreteVisitor must contain a method has a parameter of type java.lang.Object.