

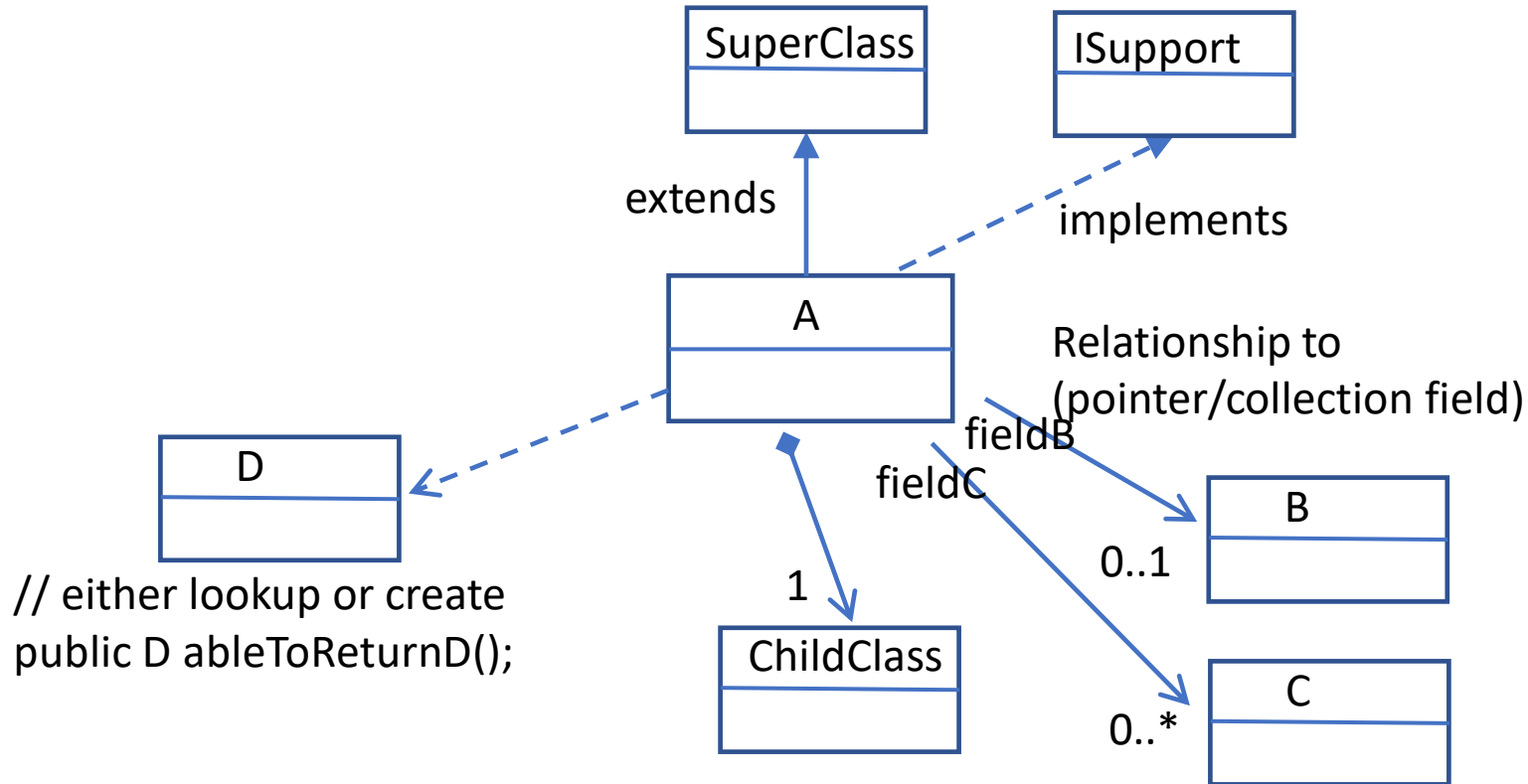
Design Patterns Glossary

arnaud.nauwynck@gmail.com

Reminder UML

Class A extends SuperClass

Class A implements ISupport



// either lookup or create
public D ableToReturnD();

// reference to 0 or 1 B ... nullable pointer
private B fieldB;

// reference to any number of C
private List<C> fieldC = new ArrayList<>();

// A has strong aggregation (ownership) on Child
// when destroying A, Child are destroyed
private ChildClass child = new ChildClass();

Fundamental Principle : SOLID

<https://en.wikipedia.org/wiki/SOLID>

Single

Open-closed

Liskov substitution principle

Interface segregation

Dependency inversion

Inheritance Principle

Do not inherit code...

only abstract method

(can not inherit from multiple classes)

Use Delegation instead of Inheritance

3 Categories of Design Patterns

Creational Patterns

Structural Patterns

Behavioral Patterns

Part 1 : Creational Patterns

Abstract Factory (Kit)

Builder

Factory Method

Prototype

Singleton

Not in Gof:

Dependency Injection

Creational Patterns..(Wikipedia)

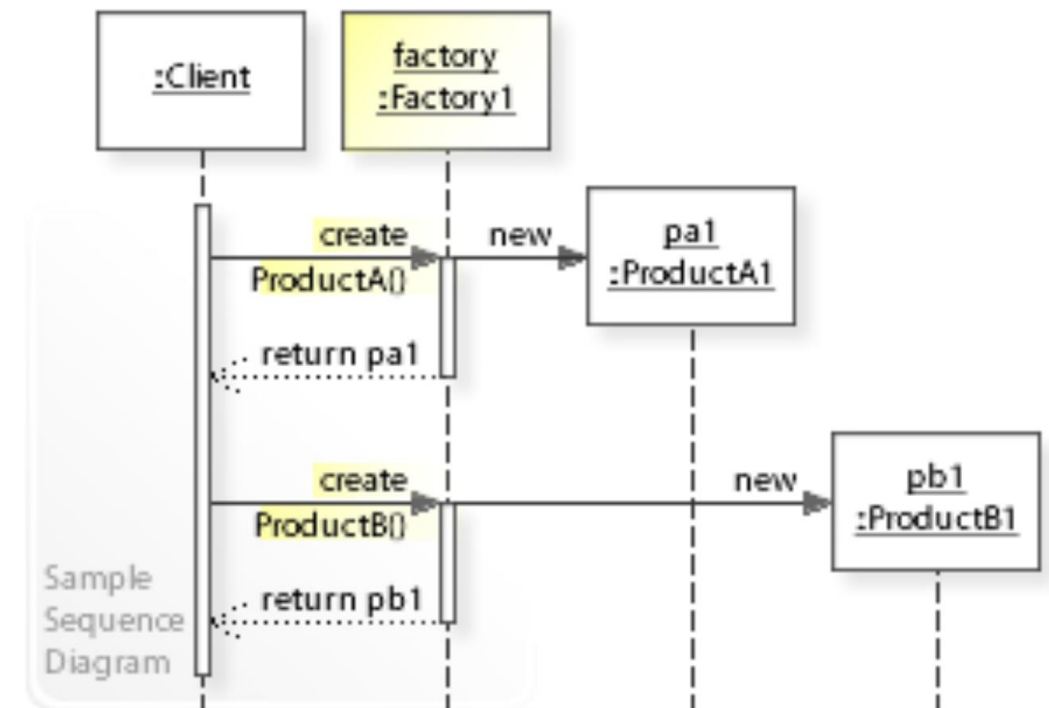
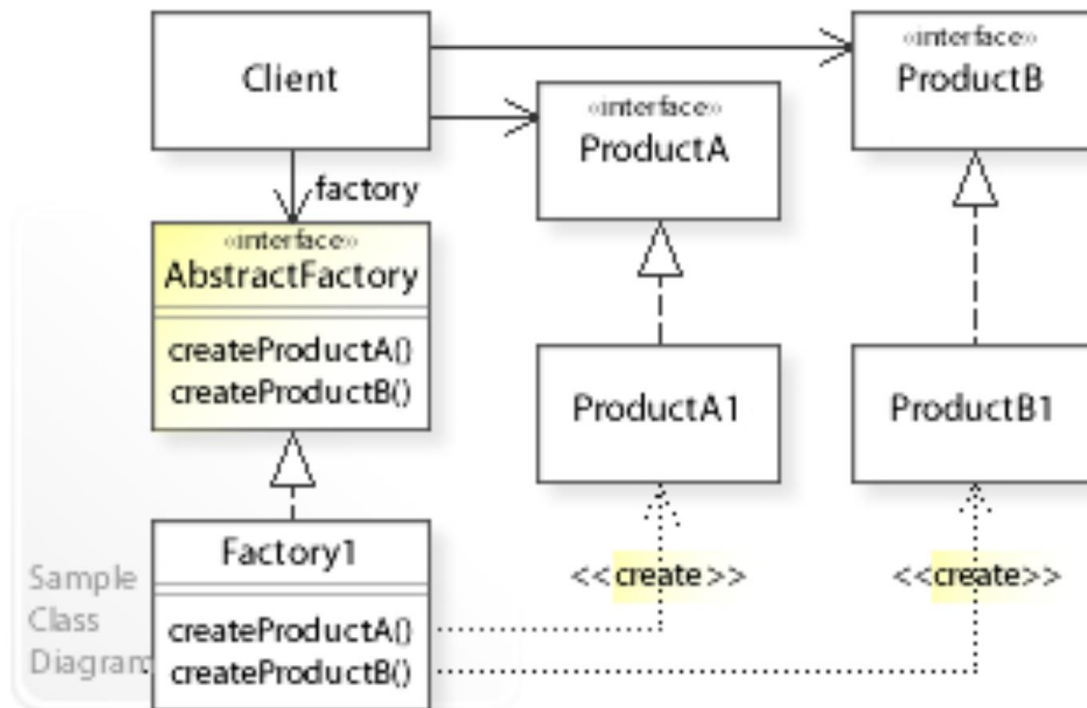
https://en.wikipedia.org/wiki/Software_design_pattern

[Creational patterns](#) [\[edit \]](#)

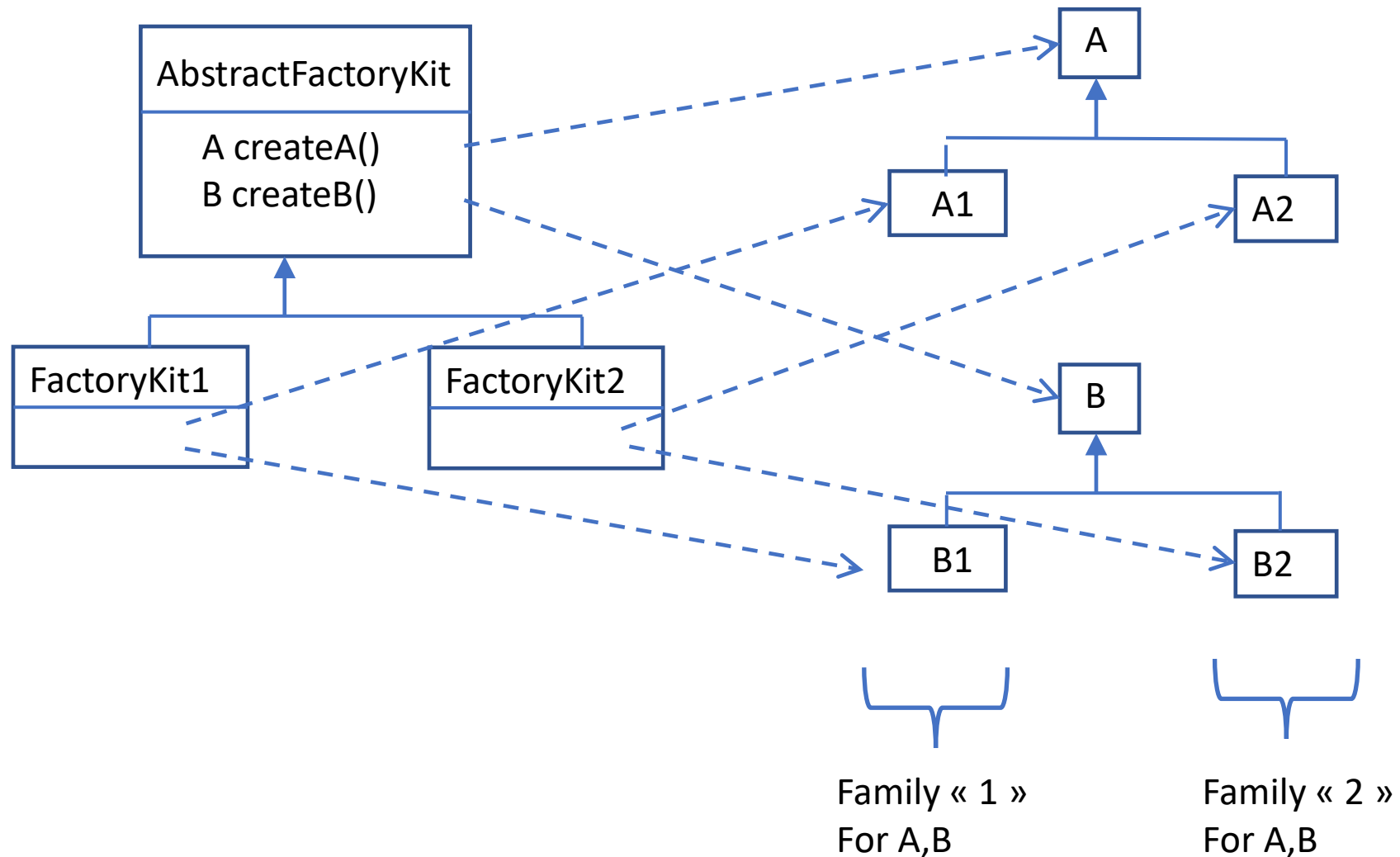
Name	Description	In <i>Design Patterns</i>	In <i>Code Complete</i> ^[14]	Other
Abstract factory	Provide an interface for creating <i>families</i> of related or dependent objects without specifying their concrete classes.	Yes	Yes	—
Builder	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.	Yes	No	—
Dependency Injection	A class accepts the objects it requires from an injector instead of creating the objects directly.	No	No	—
Factory method	Define an interface for creating a <i>single</i> object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.	Yes	Yes	—
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern.	No	No	PoEAA ^[15]
Multiton	Ensure a class has only named instances, and provide a global point of access to them.	No	No	—
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.	No	No	—
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum.	Yes	No	—
Resource acquisition is initialization (RAII)	Ensure that resources are properly released by tying them to the lifespan of suitable objects.	No	No	—
Singleton	Ensure a class has only one instance, and provide a global point of access to it.	Yes	Yes	—

Abstract Factory Kit

Provide an interface for creating *families* of related or dependent objects without specifying their concrete classes.



Abstract Factory Kit


























Example AbstractFactory Kit

In java awt ...

« java.awt.Toolkit »

... to support different
Systems:

Unix, Windows, Mac, ..

```
✓  Toolkit
  •  Toolkit()
  •  createDesktopPeer(Desktop) : DesktopPeer
  •  createButton(Button) : ButtonPeer
  •  createTextField(TextField) : TextFieldPeer
  •  createLabel(Label) : LabelPeer
  •  createList(List) : ListPeer
  •  createCheckbox(Checkbox) : CheckboxPeer
  •  createScrollbar(Scrollbar) : ScrollbarPeer
  •  createScrollPane(ScrollPane) : ScrollPanePeer
  •  createTextArea(TextArea) : TextAreaPeer
  •  createChoice(Choice) : ChoicePeer
  •  createFrame(Frame) : FramePeer
  •  createCanvas(Canvas) : CanvasPeer
  •  createPanel(Panel) : PanelPeer
  •  createWindow(Window) : WindowPeer
  •  createDialog(Dialog) : DialogPeer
  •  createMenuBar(MenuBar) : MenuBarPeer
  •  createMenu(Menu) : MenuPeer
  •  createPopupMenu(PopupMenu) : PopupMenu
  •  createMenuItem(MenuItem) : MenuItemPeer
  •  createFileDialog(FileDialog) : FileDialogPeer
  •  createCheckboxMenuItem(CheckboxMenuItem)
```

Windows implementation:

class Wtoolkit extends Toolkit {

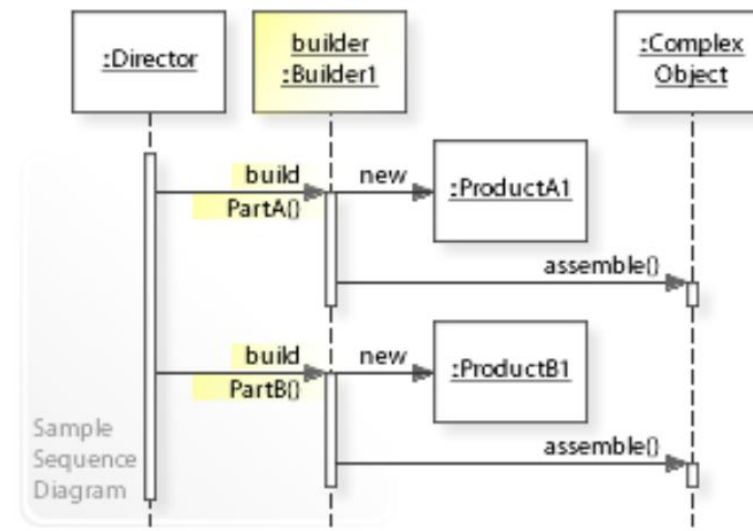
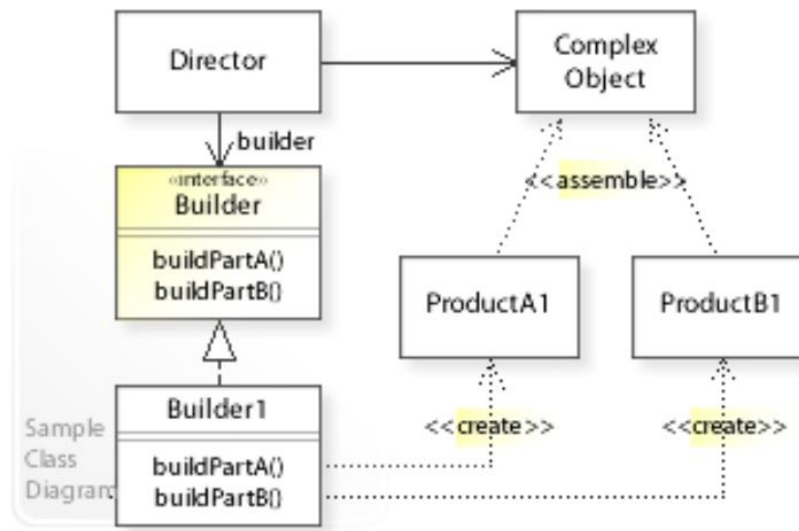
```
@Override
public ButtonPeer createButton(Button target) {
    ButtonPeer peer = new WButtonPeer(target);
    targetCreatedPeer(target, peer);
    return peer;
}

@Override
public TextFieldPeer createTextField(TextField target) {
    TextFieldPeer peer = new WTextFieldPeer(target);
    targetCreatedPeer(target, peer);
    return peer;
}

@Override
public LabelPeer createLabel(Label target) {
    LabelPeer peer = new WLabelPeer(target);
    targetCreatedPeer(target, peer);
    return peer;
}
```

Builder

Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.



@Builder in Lombok

temporary object for building an immutable one

```
class ImmutableA {  
    private final int field1, field2 ....
```

@Builder

```
class ImmutableA {  
  
    private final int field1;  
    private final int field2;  
    private final int field3;  
  
}
```



```
    protected ImmutableA(ImmutableABuilder b){  
        this.field1 = b.field1;  
        this.field2 = b.field2;  
        this.field3 = b.field3;  
    }  
    public static Builder builder() {  
        return new Builder();  
    }  
}
```



```
public static class Builder {  
    private int field1, field2, field3; // ← NOT final !!!  
  
    public Builder field1(int p) { this.field1 = p; return this;}  
    public Builder field2(int p) { this.field2 = p; return this;}  
  
    public ImmutableA build() {  
        return new ImmutableA(this);  
        // or equivalently new..(field1, field2, field3, ....);  
    }  
}
```

Sample @Builder usage

Fluent API for building objects...
Each value is self-described

Is much more obvious than

Does not compile for immutable

```
A a = A.builder()  
    .field1 ( 10 )  
    .field2 ( 5 )  
    .field3 ( 32 )  
    .field4 ( 8 )  
    .build();
```

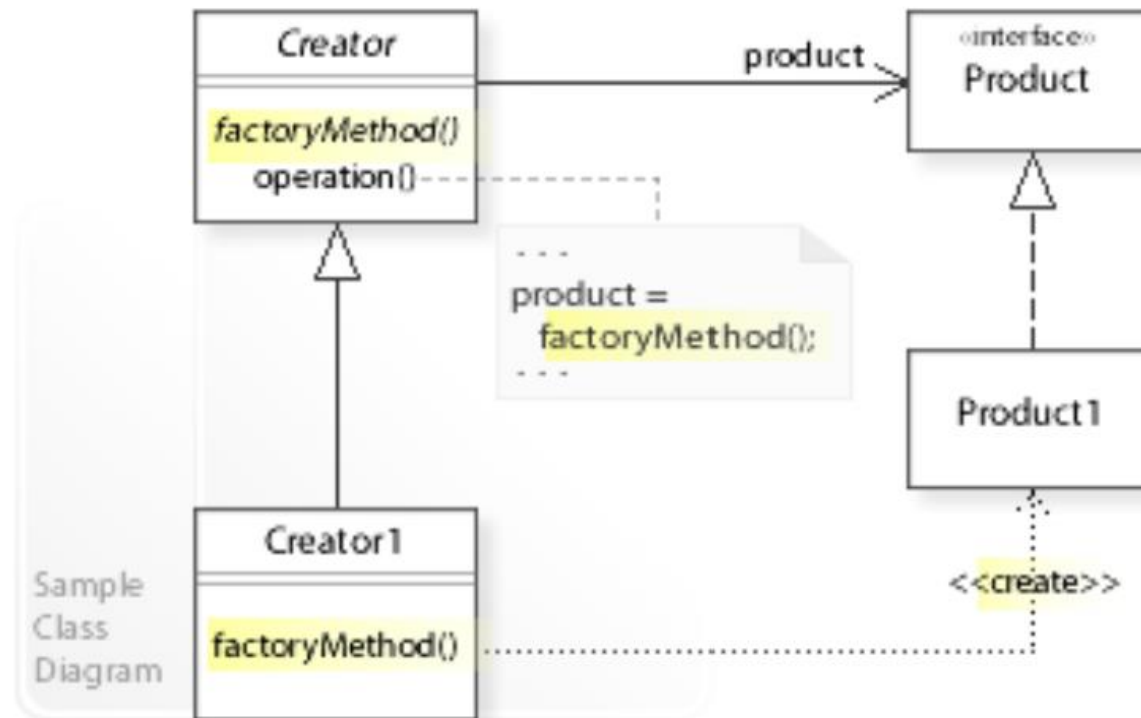
```
A a = new A(  
    10, // for field1  
    5,  
    32, // for field3  
    8 );
```

```
A a = new A(); // ERR .. missing params  
a.field1 = 10;  
a.field2 = 5;  
a.field3 = 32;  
a.field4 = 8;
```

Factory Method

Define an interface for creating a *single* object, but let subclasses decide which class to instantiate.

Factory Method lets a class defer instantiation to subclasses.



Synonyms

Supplier

Provider

Factory Method

Create Function

Generator

Allocator

Example

```
package java.util.function;

    * Represents a supplier of results.
    @FunctionalInterface
    public interface Supplier<T> {

        * Gets a result.
        T get();

    }
```

Instead of

```
abstract class A {

    protected abstract B createB();

    // other methods for using new Bs
}
```

« Delegate instead of Extends »

```
class A {

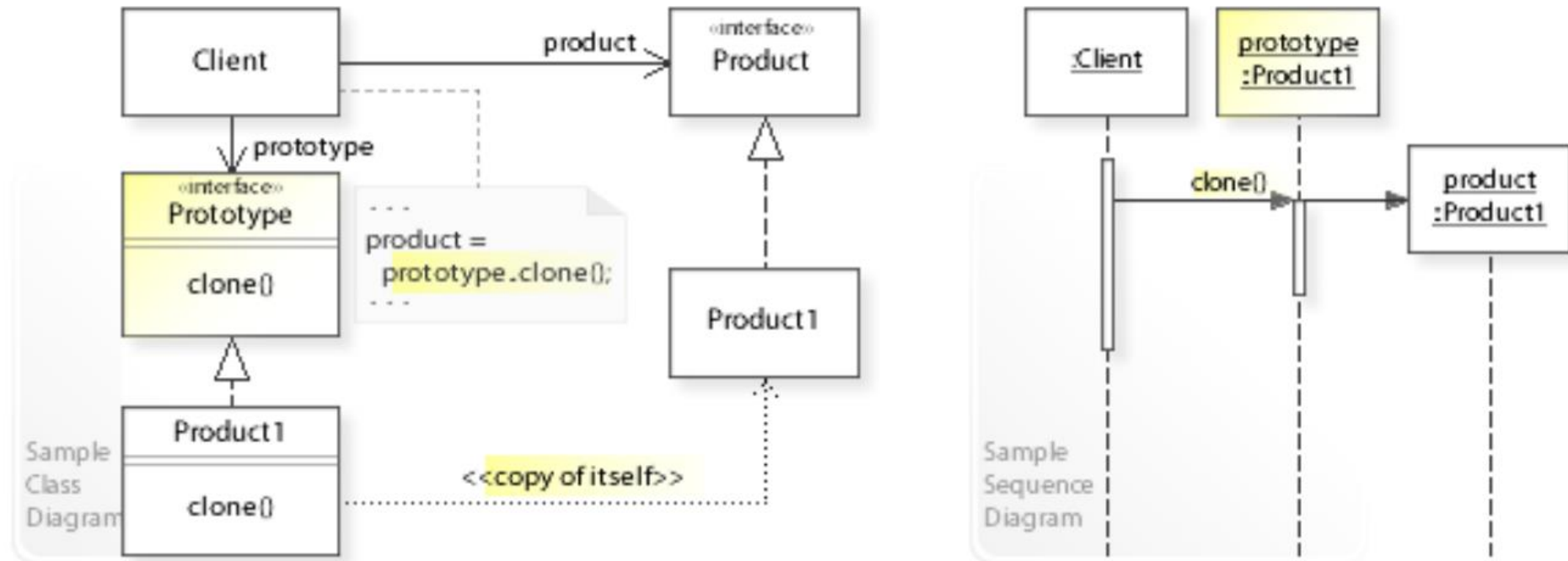
    protected final Supplier<B> bCreateFunction;

    public A(Supplier<B> bCreateFunction) {
        this.bCreateFunction = bCreateFunction;
    }

    // other methods for using new Bs
}
```


Prototype

Create object using « `existingObjectInstance.clone()` » instead of « `new Class()` »



Dependency Injection

A class accepts the objects it requires from an injector instead of creating the objects directly.

Synonyms :

- “DI”
- “IOC” Inversion Of Control
- Hollywood Principle: “Don’t call me, I will call You”

Example

Massively used in all modern frameworks
Example SpringFramework / SpringBoot

```
@Service
class SomeService {
}

@Configuration
class SomeServiceConfiguration {
    @Bean
    public AnotherService bean() {
        return new ...
    }
}
```

```
@Service
class MyService {
    @Autowired
    private SomeService field1;

    @Autowired
    private AnotherService field2;
}
```

Or using constructors + final...

```
@Service
class MyService {
    private final SomeService field1;
    private final AnotherService field2;

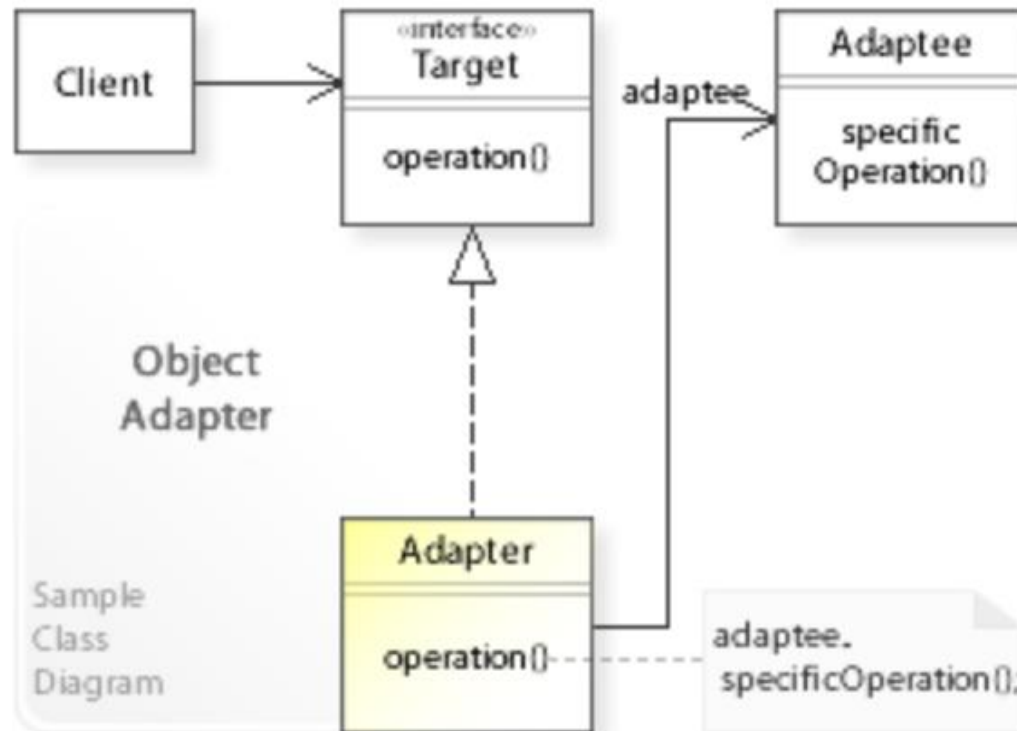
    @Autowired
    public MyService(
        SomeService field1, ....) {
        this.field1 = field1; this.field2 = field2;
    }
}
```

Part 2 : Structural Patterns

Name	Description	In <i>Design Patterns</i>	In <i>Code Complete</i> ^[14]	Other
Adapter , Wrapper , or Translator	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.	Yes	Yes	—
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.	Yes	Yes	—
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.	Yes	Yes	—
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.	Yes	Yes	—
Delegation	Extent a class by composition instead of subclassing. The object handles a request by delegating to a second object (the delegate)	—	—	—
Extension object	Adding functionality to a hierarchy without changing the hierarchy.	No	No	Agile Software Development, Principles, Patterns, and Practices ^[16]
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.	Yes	Yes	—
Flyweight	Use sharing to support large numbers of similar objects efficiently.	Yes	No	—
Front controller	The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.	No	No	J2EE Patterns ^[17] PoEAA ^[18]
Marker	Empty interface to associate metadata with a class.	No	No	Effective Java ^[19]
Module	Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.	No	No	—
Proxy	Provide a surrogate or placeholder for another object to control access to it.	Yes	No	—
Twin ^[20]	Twin allows modeling of multiple inheritance in programming languages that do not support this feature.	No	No	—

Adapter

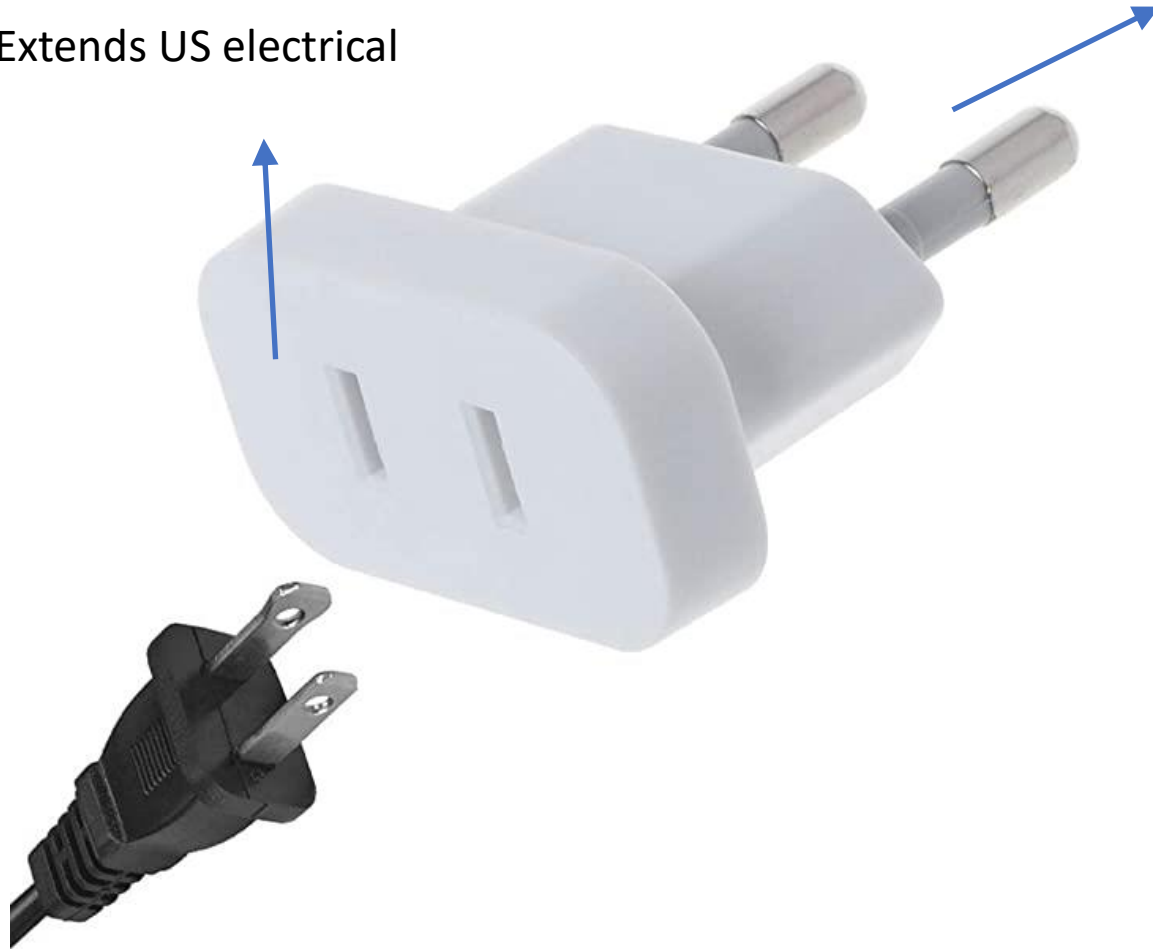
Convert the interface of a class into another interface clients expect



Example ... travelling

Extends US electrical

Delegate to European
electrical



Example ... in compiler

```
// everything that can be wrapped in « { } »  
// and appended by « ; »  
// statement have NO type (i.e. void)  
abstract class Statement {  
}
```

```
// call an expression, to adapt it as a Statement  
class ExprStatement extends Statement {  
    private Expression expression;  
}
```

```
// everything that can be wrapped in « ( ) »  
// statement have a type  
abstract class Expression {  
}
```

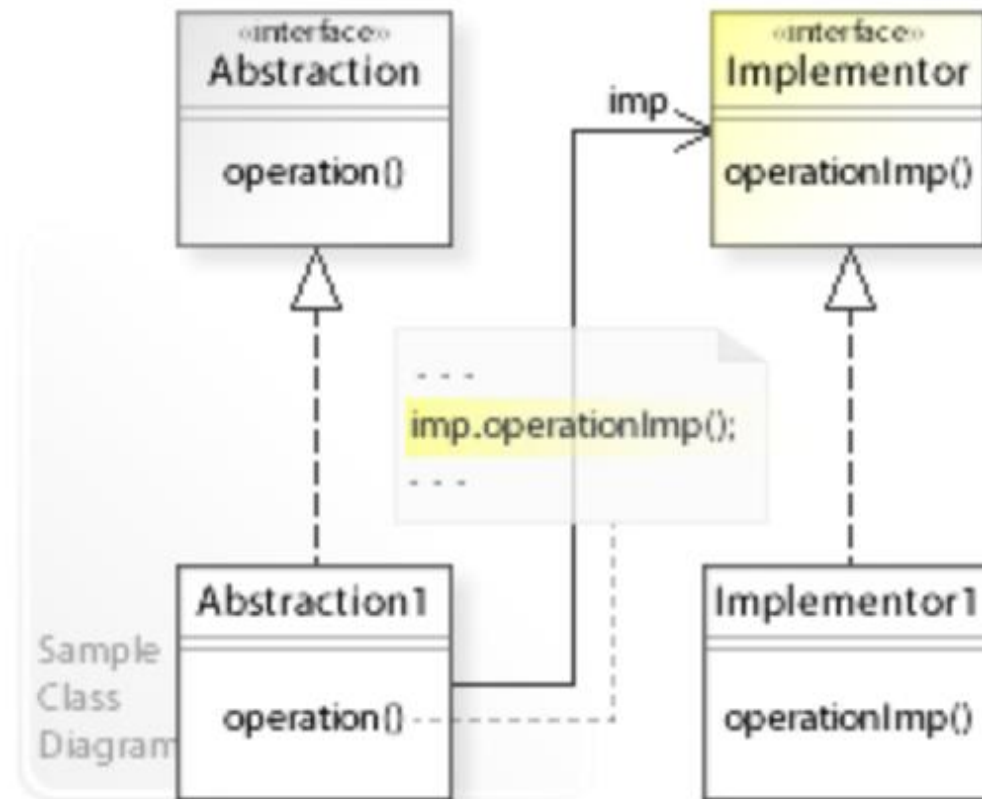
$f(x) + f(y)$ is an Expression

```
{  
f(x) ; .... Is a sequence of Statements  
f(y) ;      made of expressions (function call)  
}
```

1

Bridge

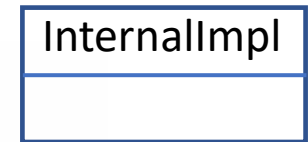
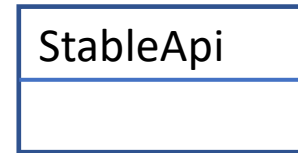
Decouple an abstraction from its implementation allowing the two to vary independently.



Synonym

shock absorber

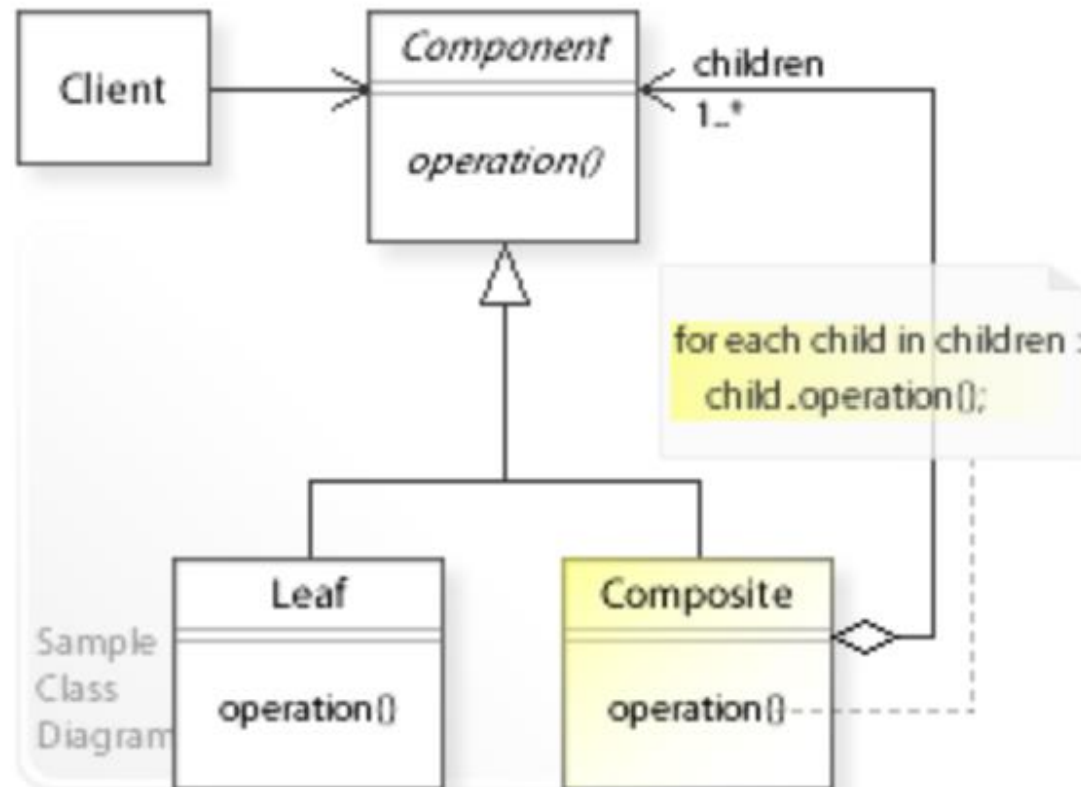
Isolation



May changes
.. Should not impact

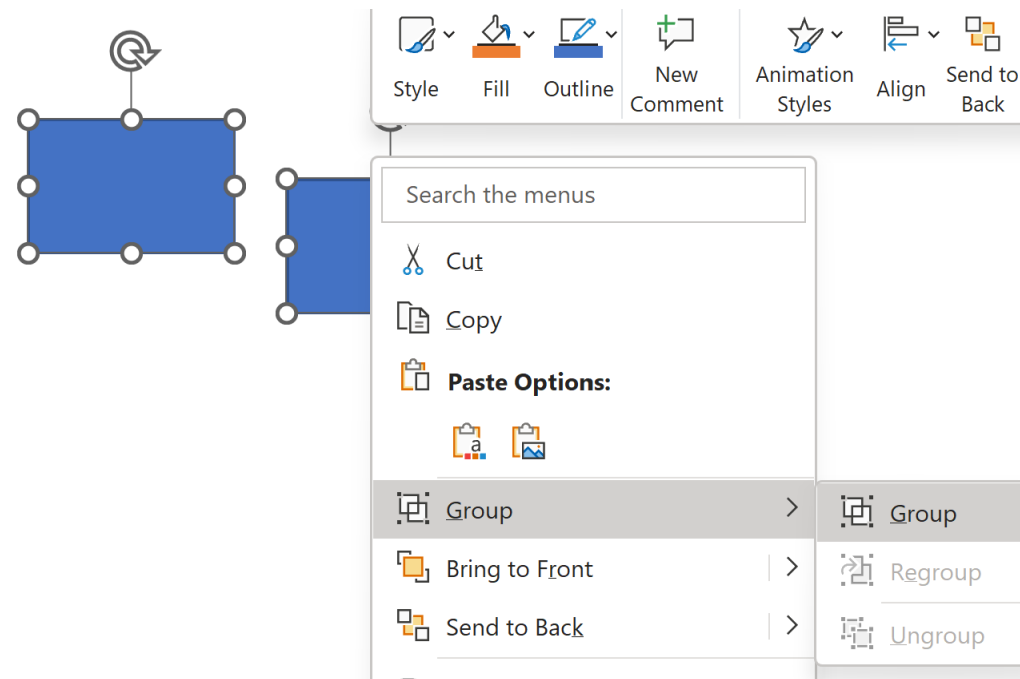
Composite

Compose objects into tree structures to represent part-whole hierarchies.
Composite lets clients treat individual objects and compositions of objects uniformly.



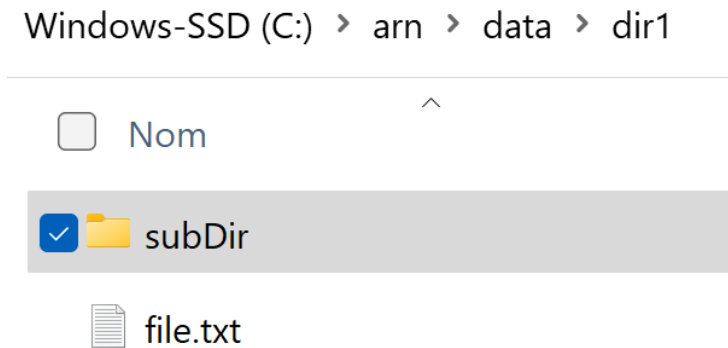
Example of Composite:

a « Group » of graphical elements
is itself a graphical element



On the « group »,
you can select, copy, translate, rotate,
change transparency, bring to front,...
(all commands)

Examples of Composite

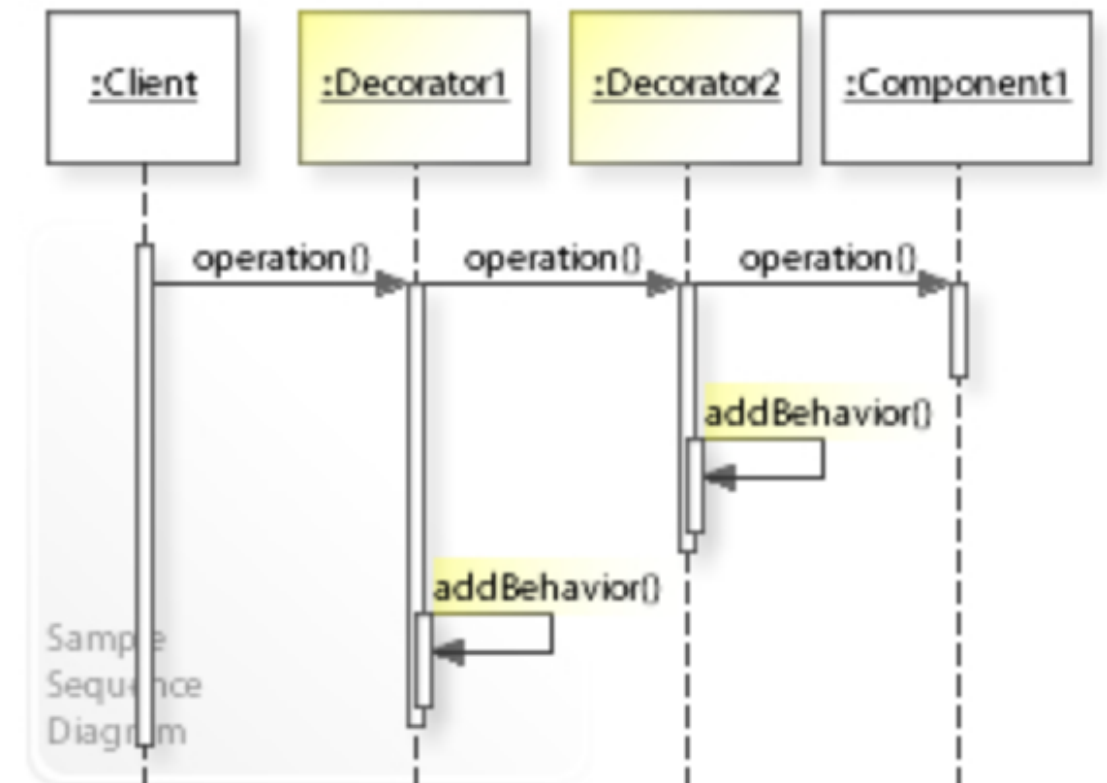
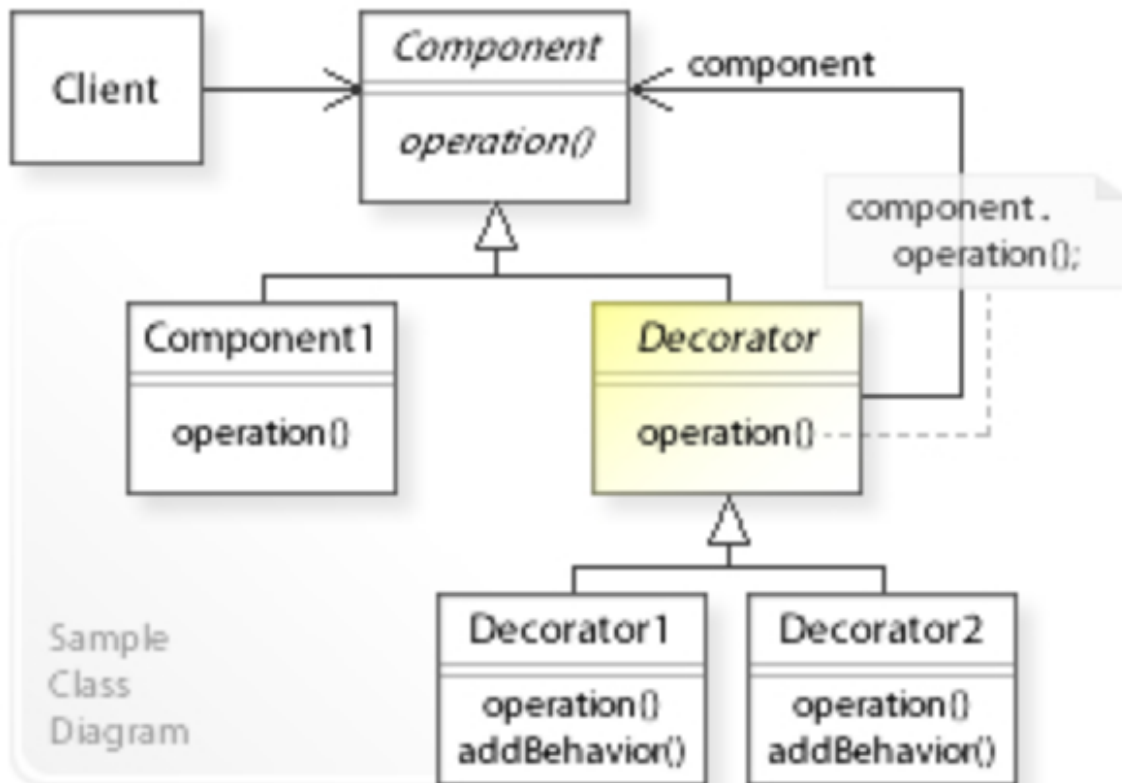


A Directory is a Composite of (sub-)directories and files

A directory is itself a filesystem element (=Inode), like file

Decorator

Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.



Example Decorator

On graphical Element,

A « Border » graphical element -> add a border decoration to the underlying element

A « Shadowed » graphical element -> add a shadow decoration to the underlying element

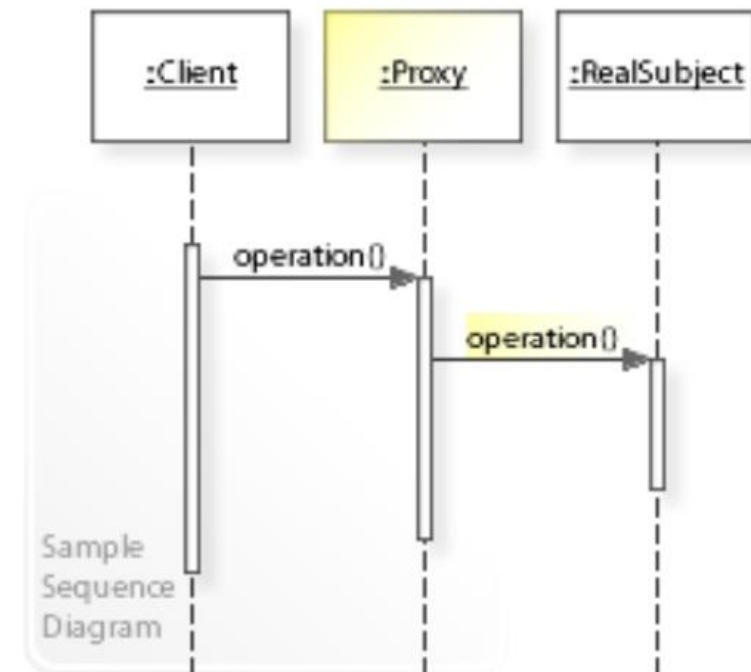
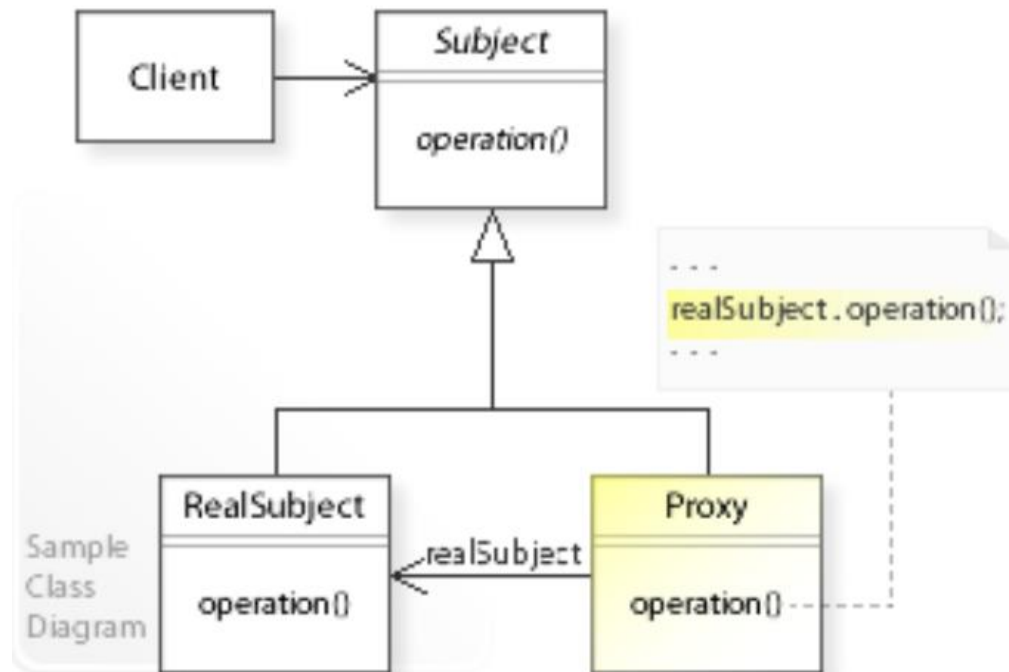
On Aspect-Oriented-Programming,

A « Logged » method -> add a log, to the execution of a method

A « Timed Metric » method -> increment a statistic counter with the execution time

Proxy

Provide a surrogate or placeholder for another object to control access to it.



Synonyms

Indirection

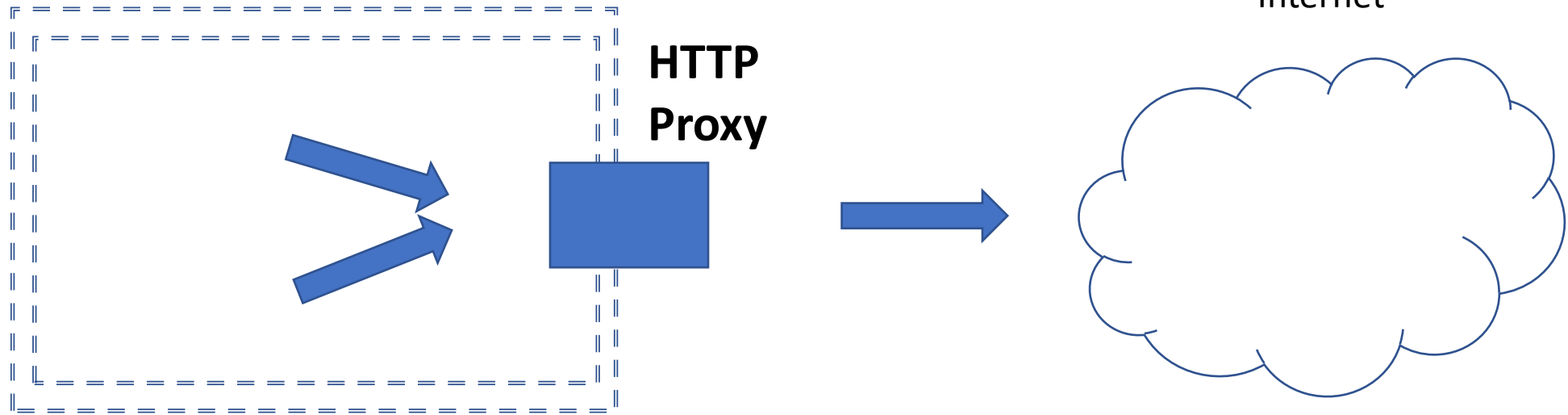
Intermediate

Cache

Access through

Example http Proxy in networks..

Firewall
on Intranet company network



```
$ curl -p myproxy:8080 https://www.google.fr
```

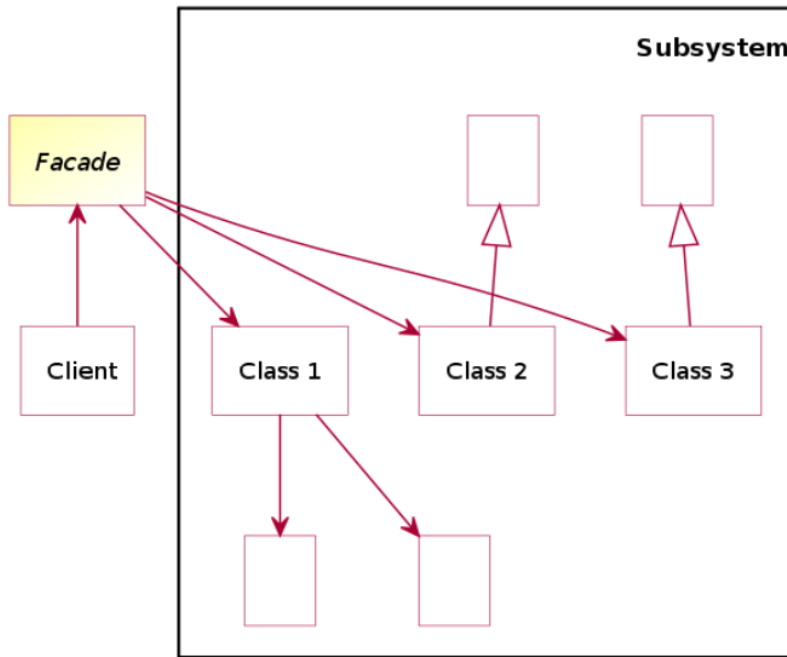
⇒ 1/ TCP connect to myproxy

+ send http request to myproxy, with extra http Header for « host »

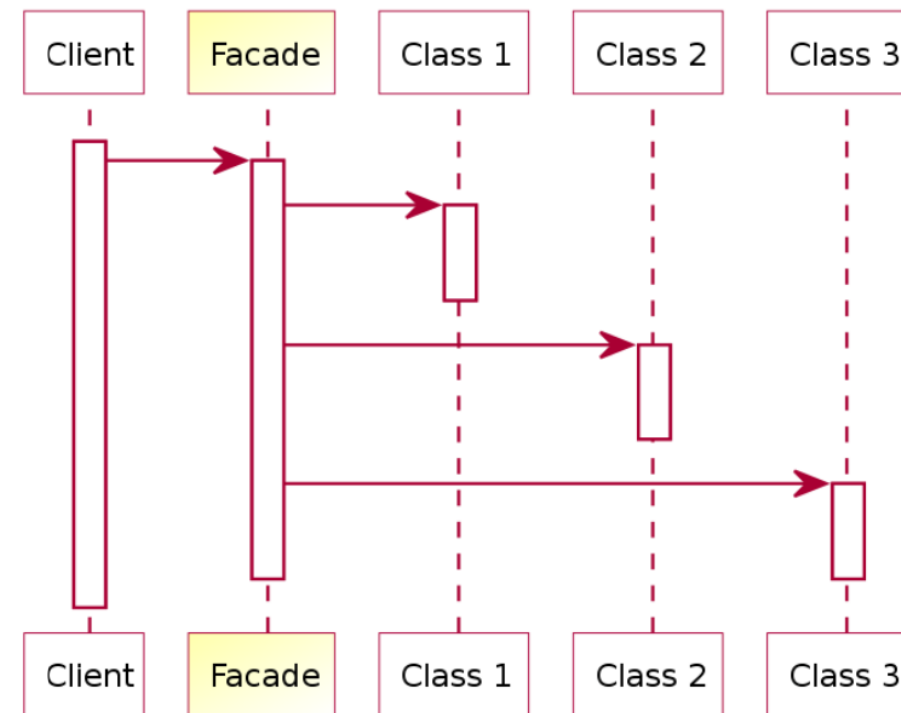
⇒ 2/ the proxy rewrite another HTTP Request to host

Facade

Provide a unified interface to a set of interfaces in a subsystem.
Facade defines a higher-level interface that makes the subsystem easier to use.

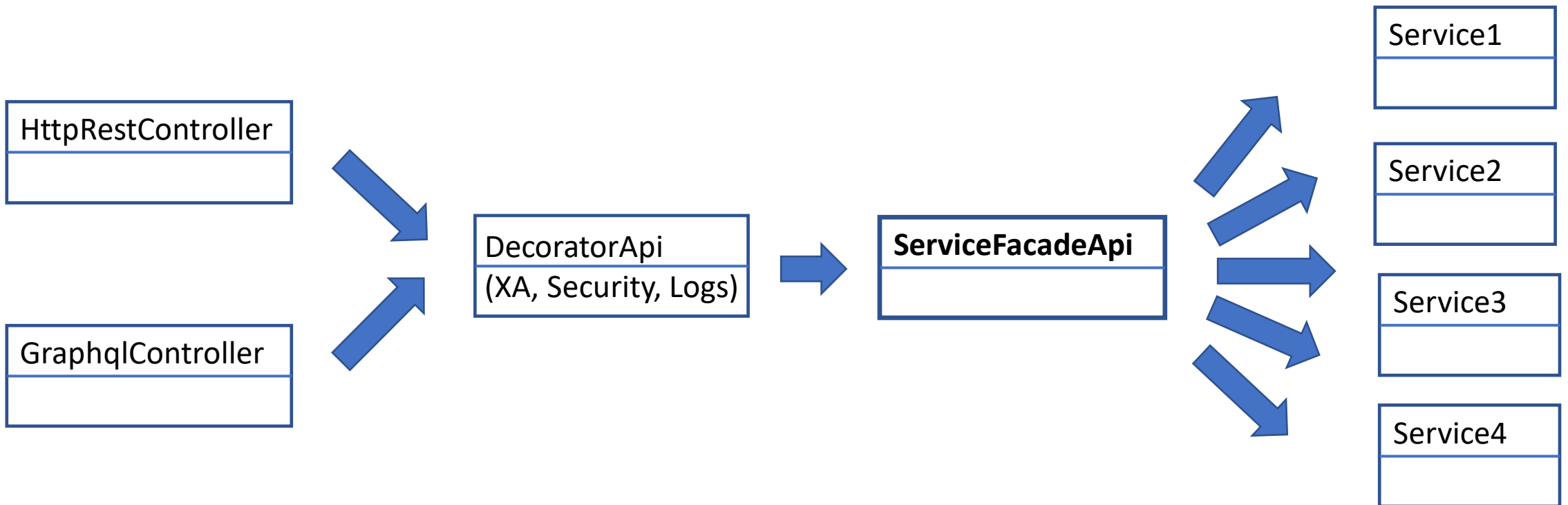


Sample class diagram



Sample sequence diagram

Examples ... Network protocol layers

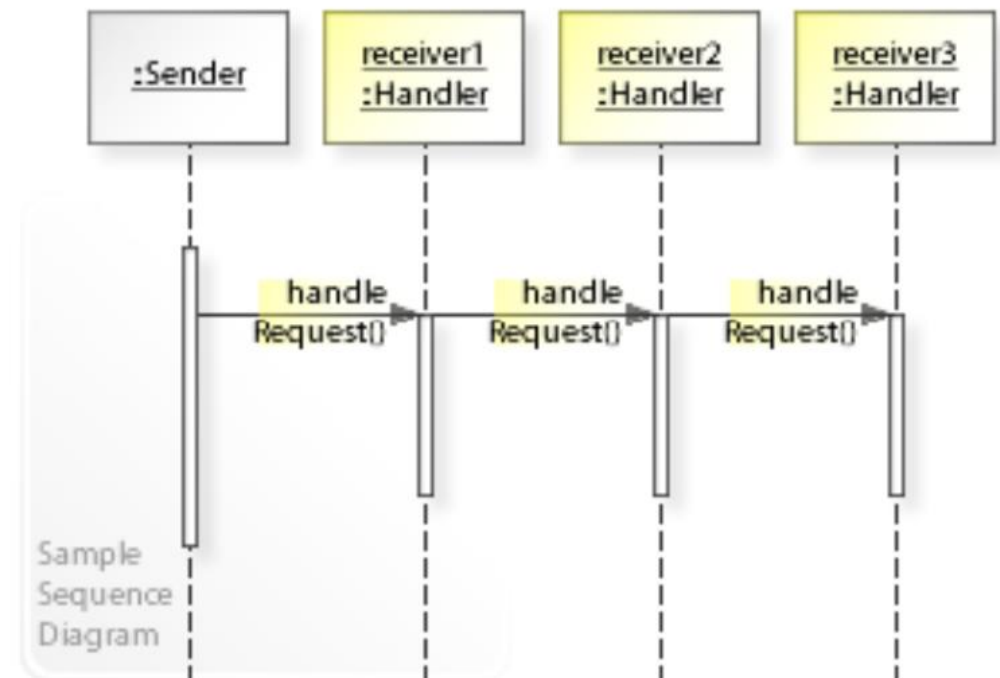
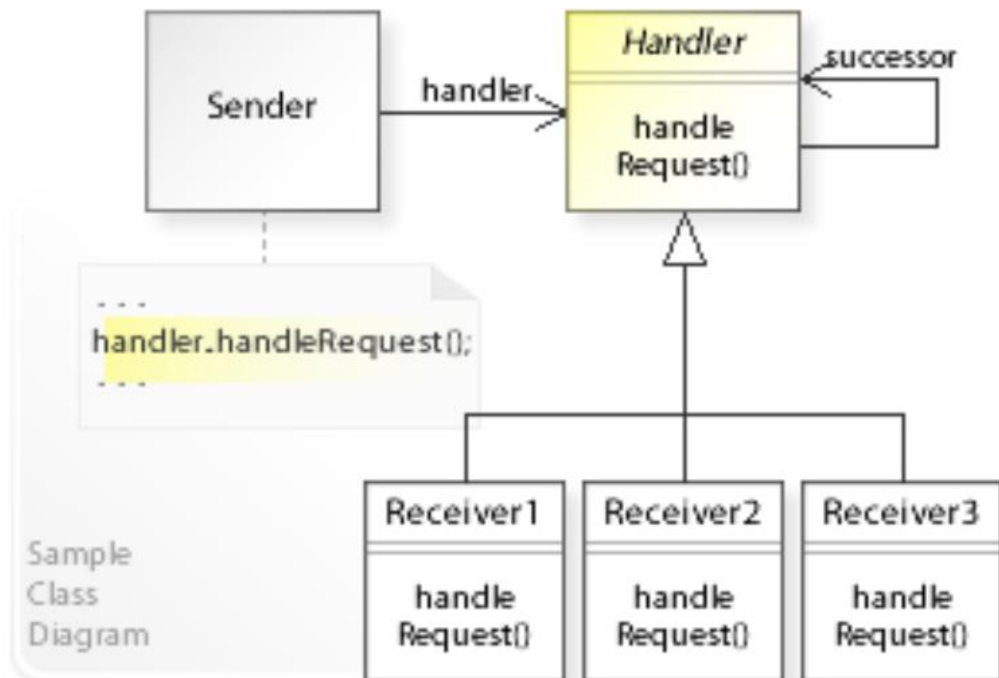


Part 3 : **Behavioral** Patterns

Name	Description	In <i>Design Patterns</i>	In <i>Code Complete</i> ^[14]	Other
Blackboard	Artificial intelligence pattern for combining disparate sources of data (see blackboard system)	No	No	—
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.	Yes	No	—
Command	Encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests. It also allows for the support of undoable operations.	Yes	No	—
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.	Yes	No	—
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	Yes	Yes	—
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it allows their interaction to vary independently.	Yes	No	—
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.	Yes	No	—
Null object	Avoid null references by providing a default object.	No	No	—
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.	Yes	Yes	—
Servant	Define common functionality for a group of classes. The servant pattern is also frequently called helper class or utility class implementation for a given set of classes. The helper classes generally have no objects hence they have all static methods that act upon different kinds of class objects.	No	No	—
Specification	Recombinable business logic in a Boolean fashion.	No	No	—
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.	Yes	No	—
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.	Yes	Yes	—
Template method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.	Yes	Yes	—
Visitor	Represent an operation to be performed on instances of a set of classes. Visitor lets a new operation be defined without changing the classes of the elements on which it operates.	Yes	No	—
Fluent Interface	Design an API to be method chained so that it reads like a DSL. Each method call returns a context through which the next logical method call(s) are made available.	No	No	—

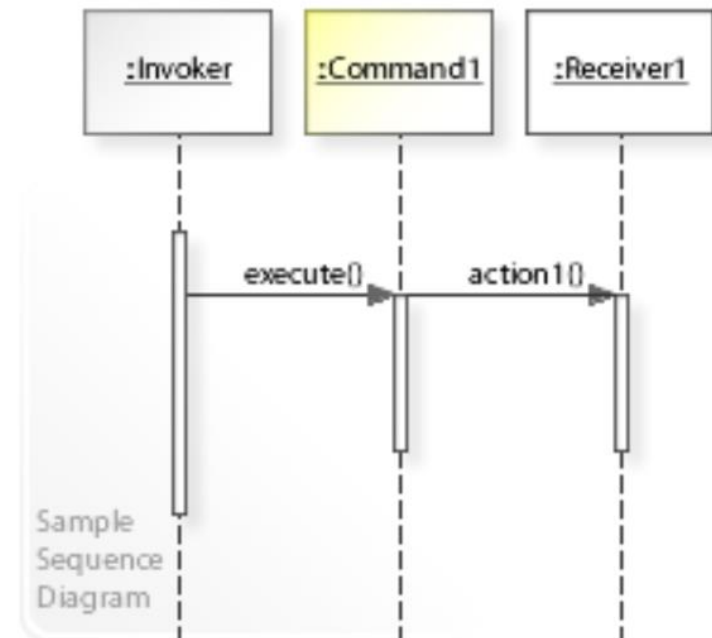
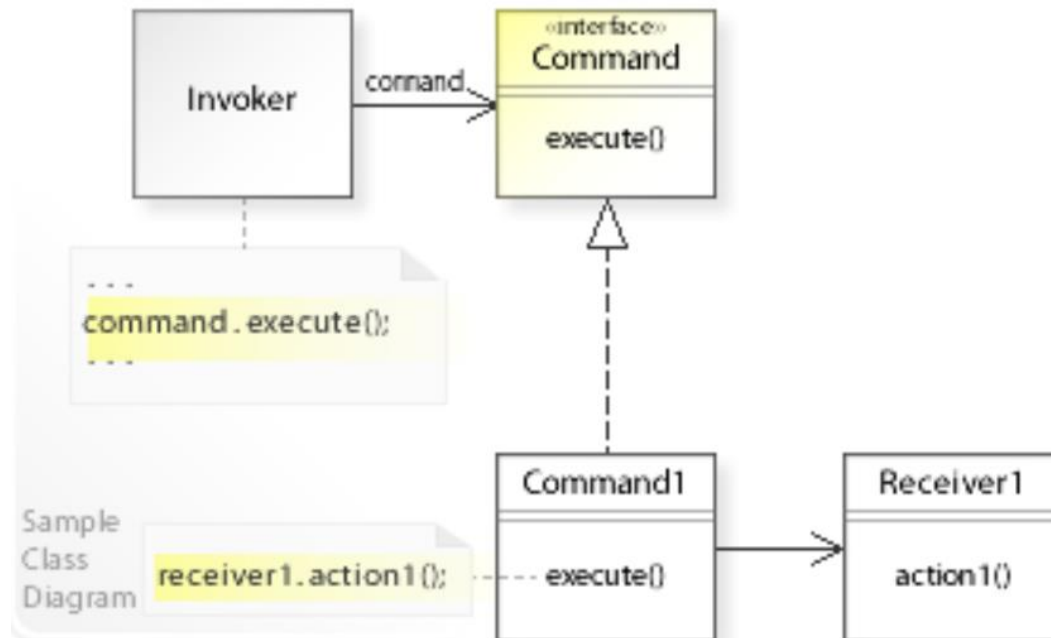
Chain of Responsibility

Avoid coupling the sender of a request to its receiver
by giving more than one object a chance to handle the request.
Chain the receiving objects and pass the request along the chain until an object handles it.



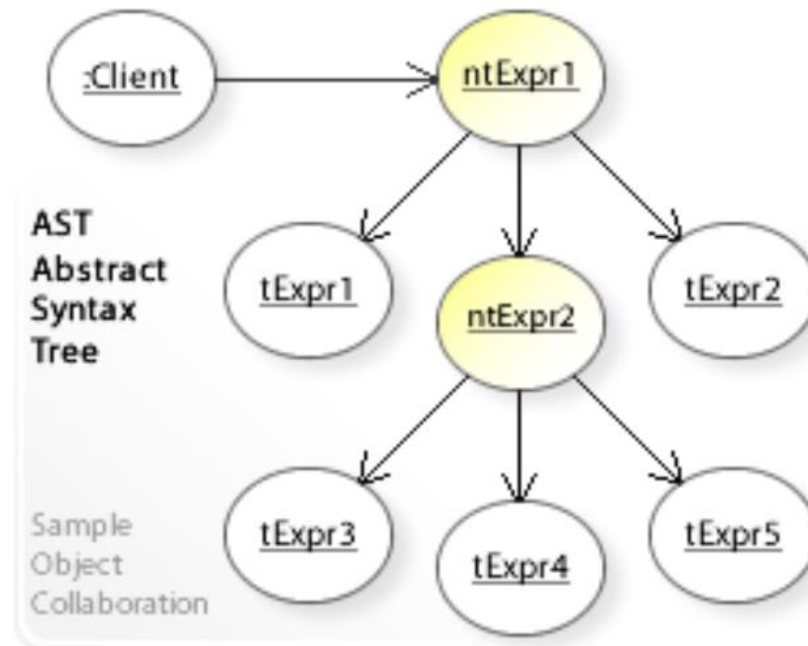
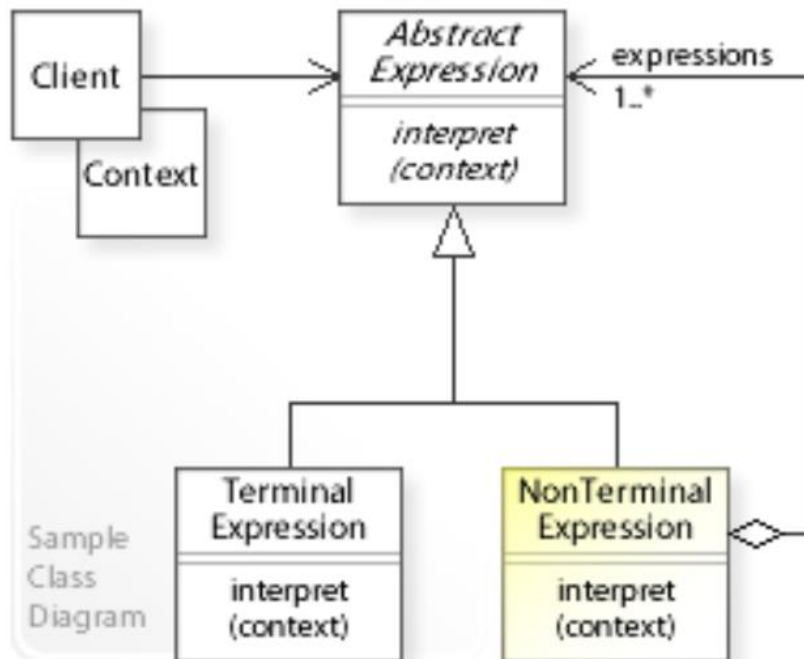
Command

Encapsulate a request as an object,
thereby allowing for the parameterization of clients with different requests,
and the queuing or logging of requests.
It also allows for the support of undoable operations.



Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Example of Interpreter : Math Expression

```
expression ::= plus | minus | variable | number
plus ::= expression expression '+'
minus ::= expression expression '-'
variable ::= 'a' | 'b' | 'c' | ... | 'z'
digit = '0' | '1' | ... | '9'
number ::= digit | digit number
```

```
abstract class Expression {}
```

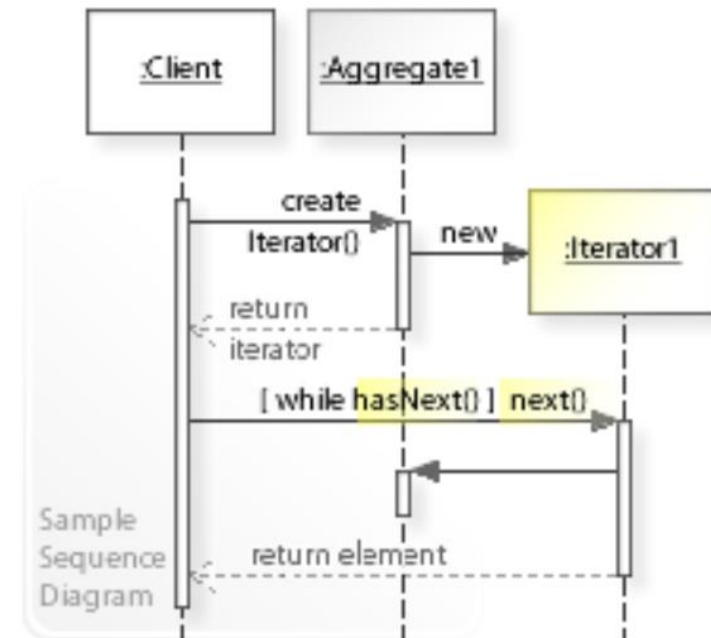
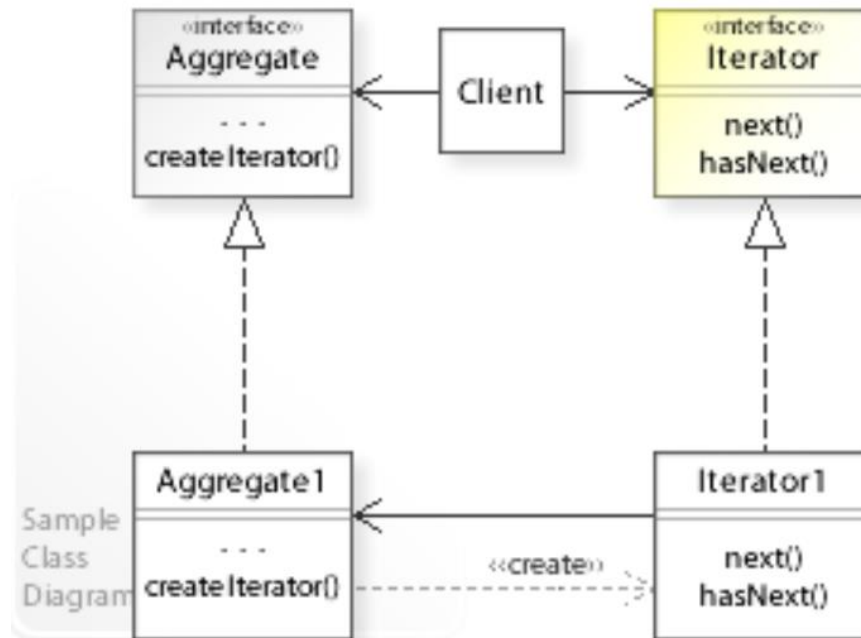
```
class NumberExpression extends Expression {
    double value;
}
```

```
class VariableExpression extends Expression {
    String name;
}
```

```
class BinaryOperationExpression extends Expression {
    Expression leftOperand;
    String operator;
    Expression rightOperand;
}
```

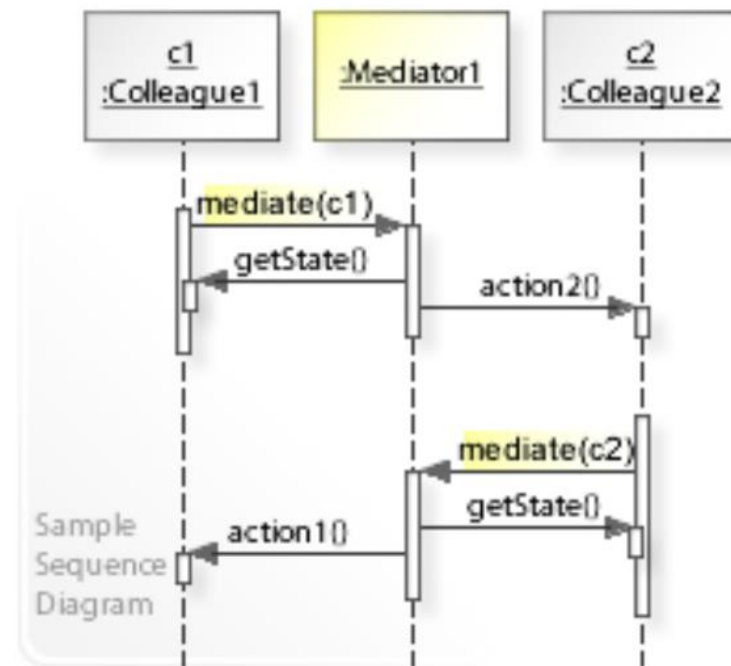
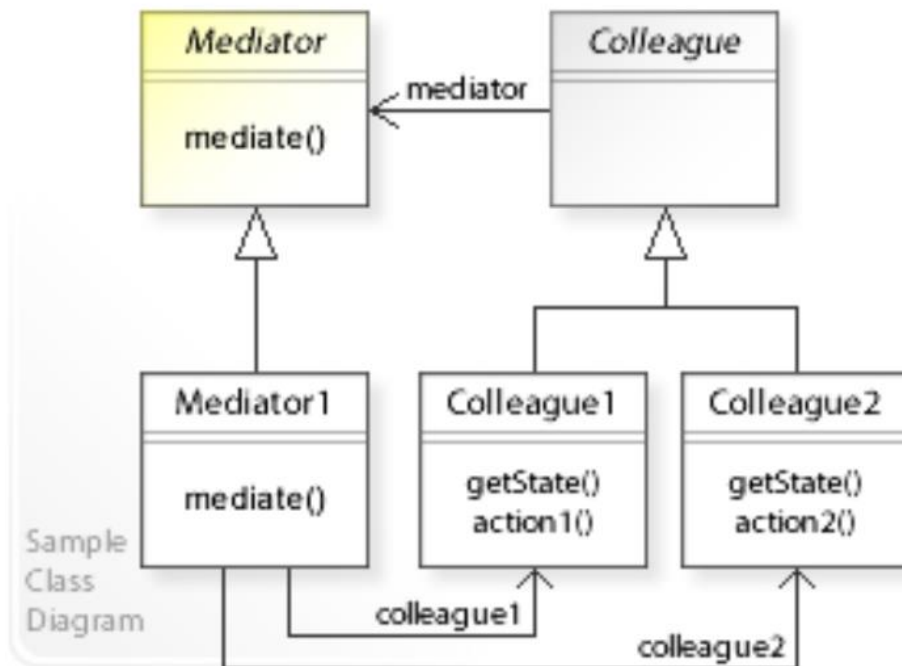
Iterator

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



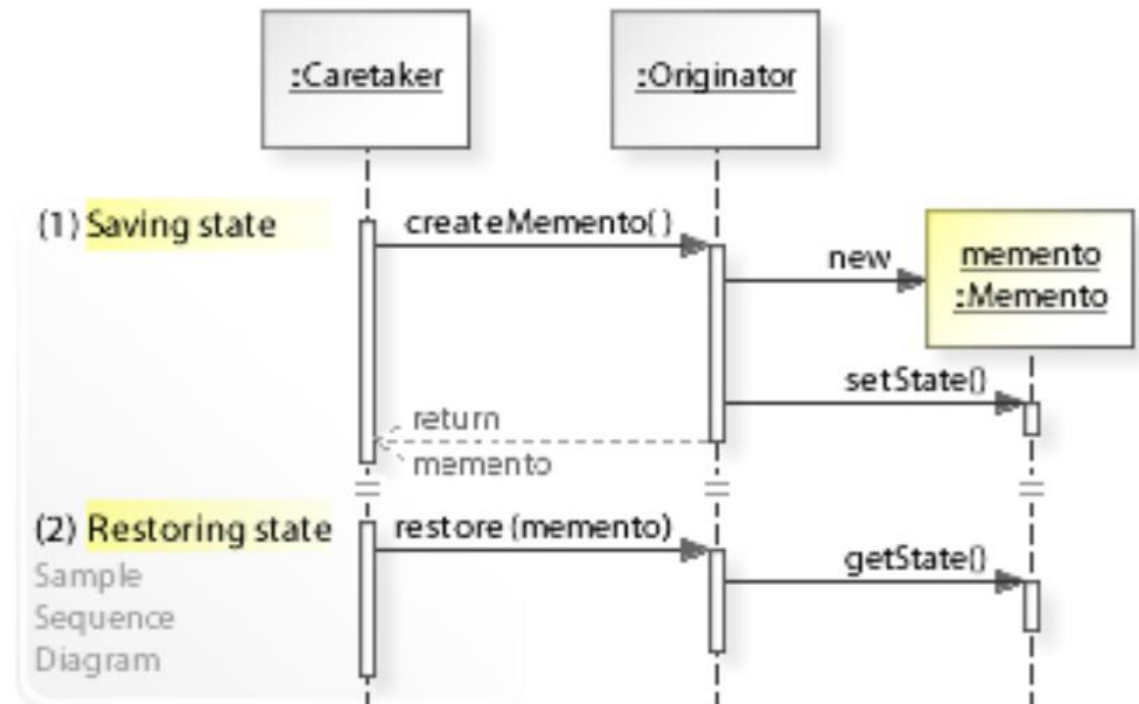
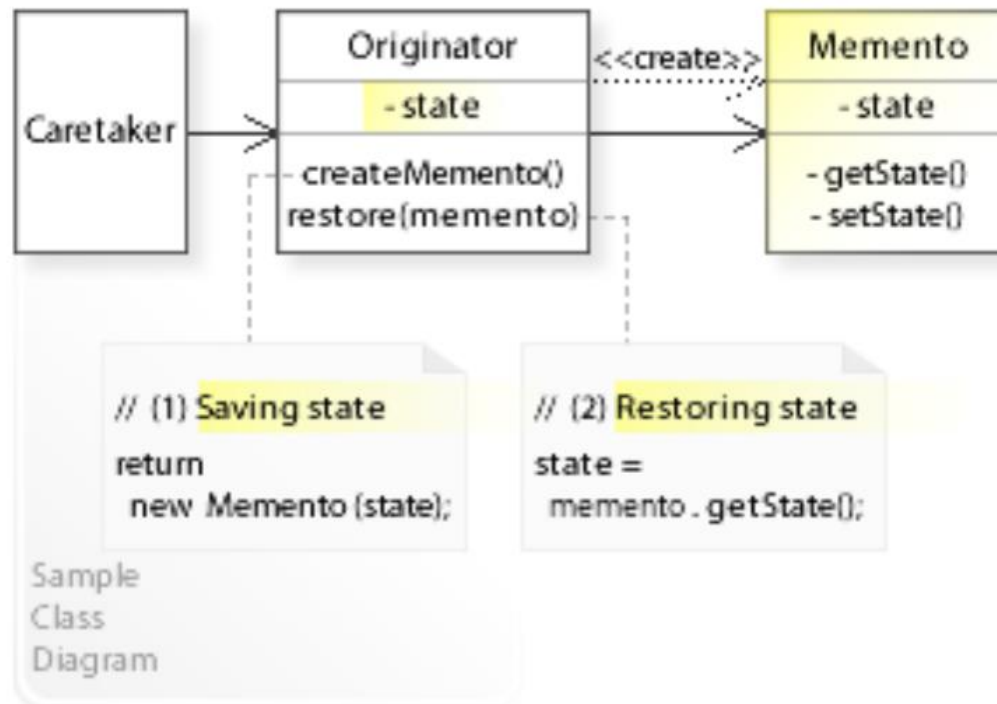
Mediator

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it allows their interaction to vary independently.



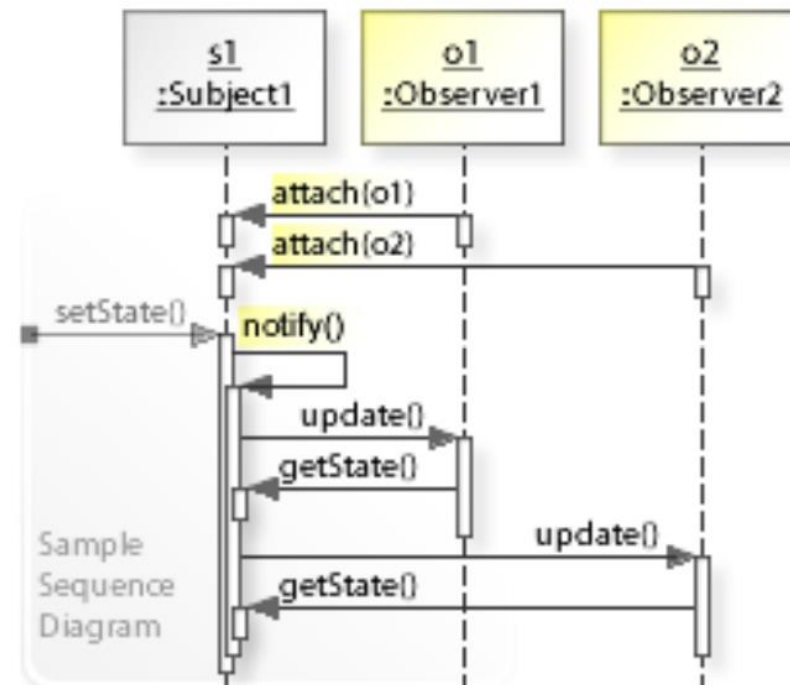
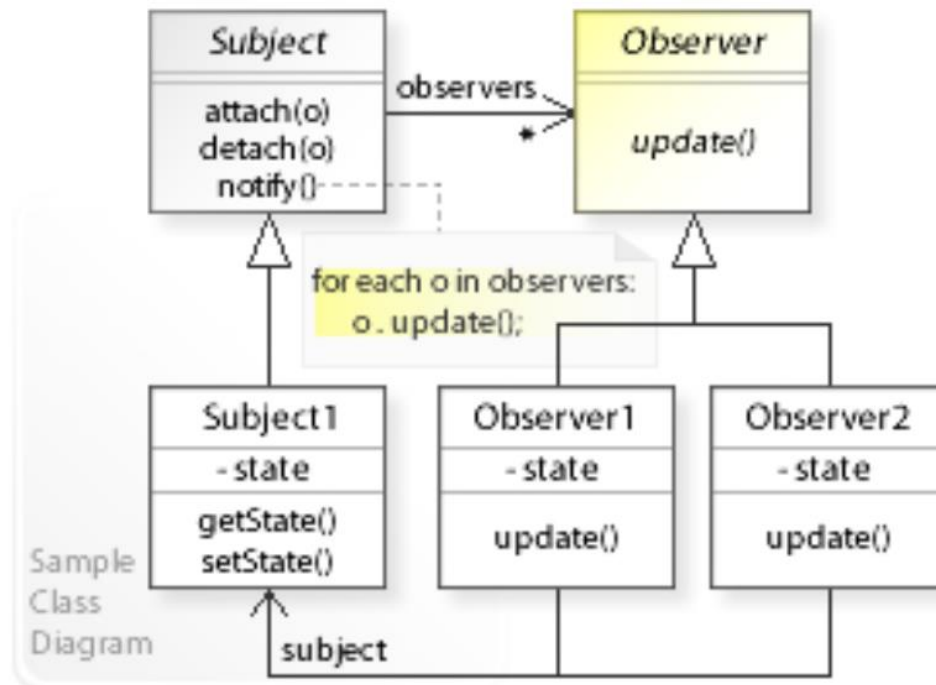
Memento

Without violating encapsulation,
capture and externalize an object's internal state
allowing the object to be restored to this state later.



Observer

Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.



Synonyms:

Publish & Subscribe

Notifier

Model – View (- Controller)

Event Sender

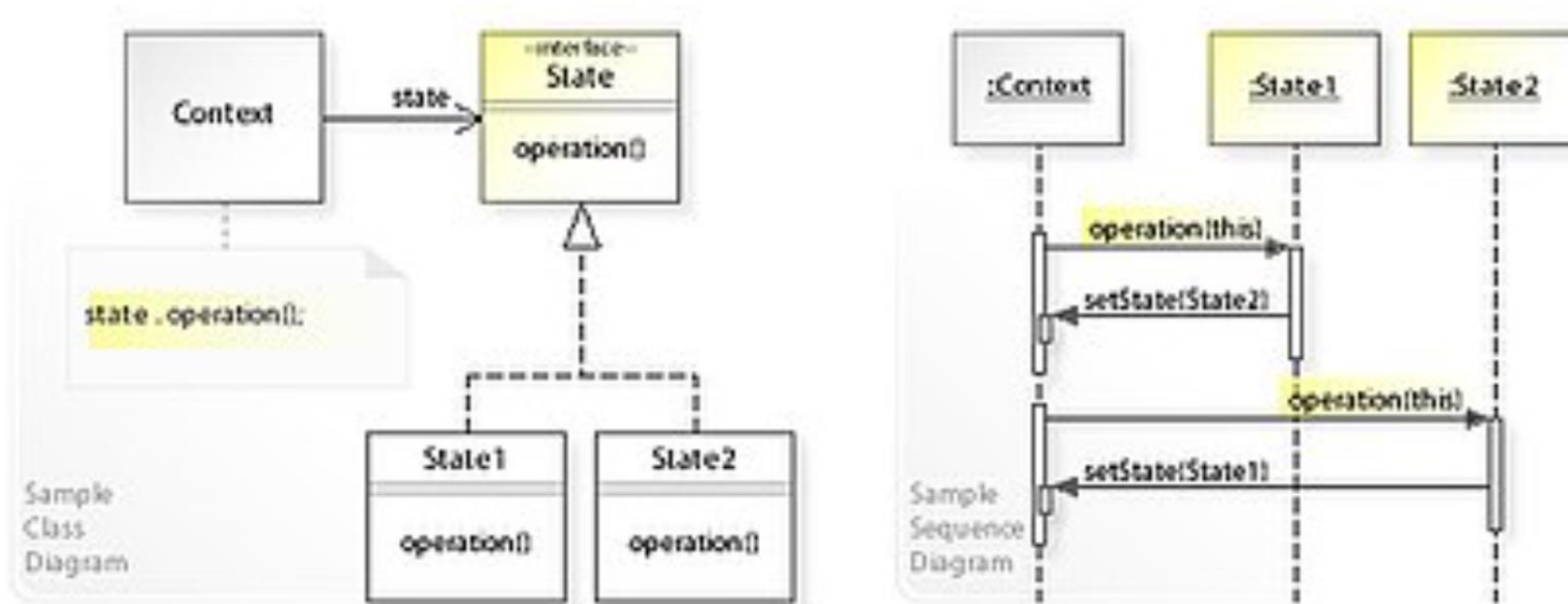
Message Sender

Emitter

Reactive Object

State

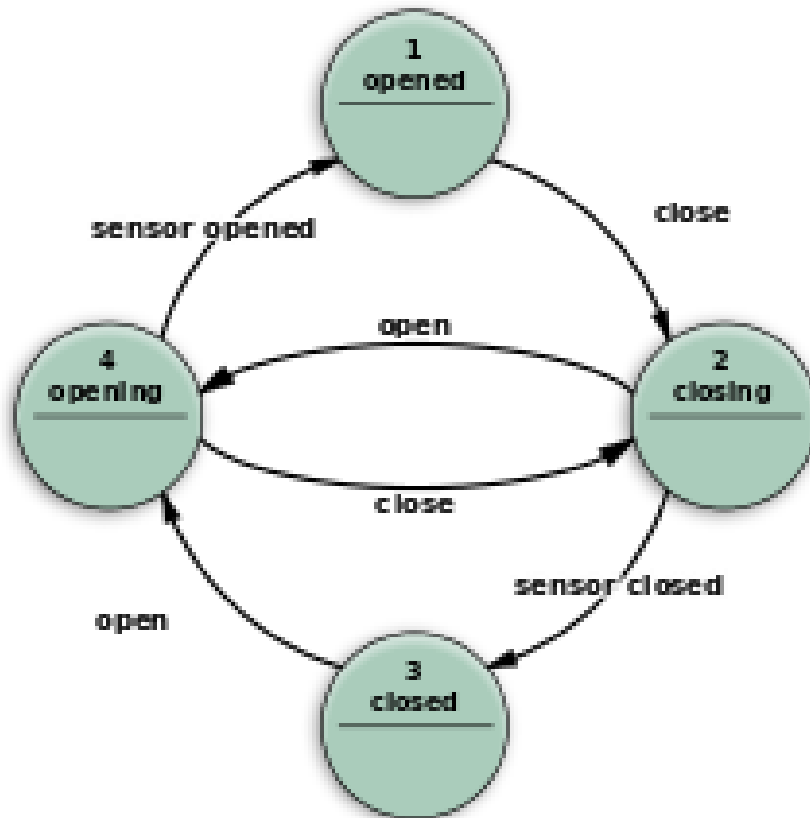
Allow an object to alter its behavior when its internal state changes.
The object will appear to change its class



Example State

1 class per state Automaton

(Deterministic) Finite State Machine ((D)FSM, FSA)



Abstract class State {

...

}

class Opened extends State {

}

class Opening extends State {

}

class Closing extends State {

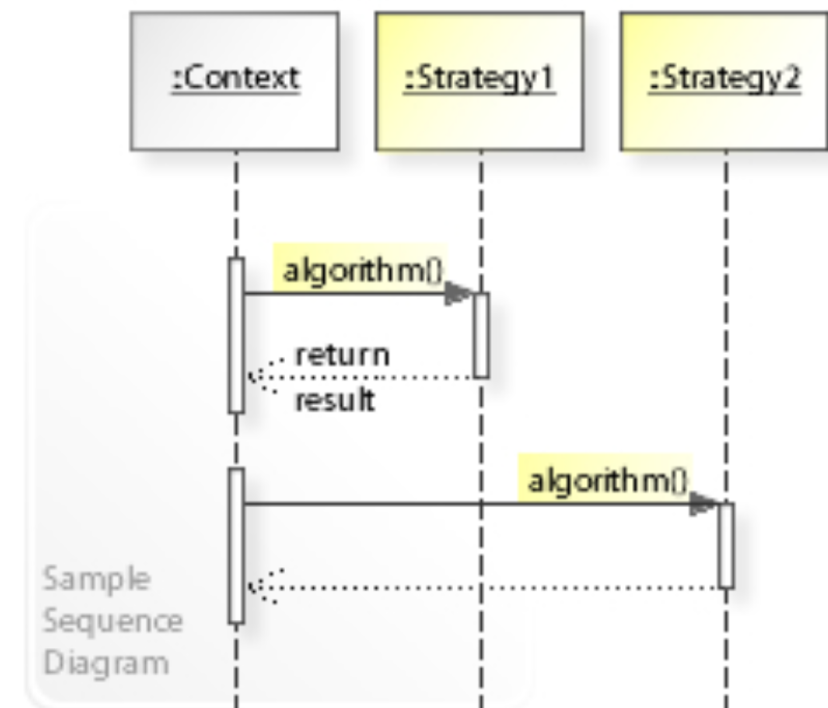
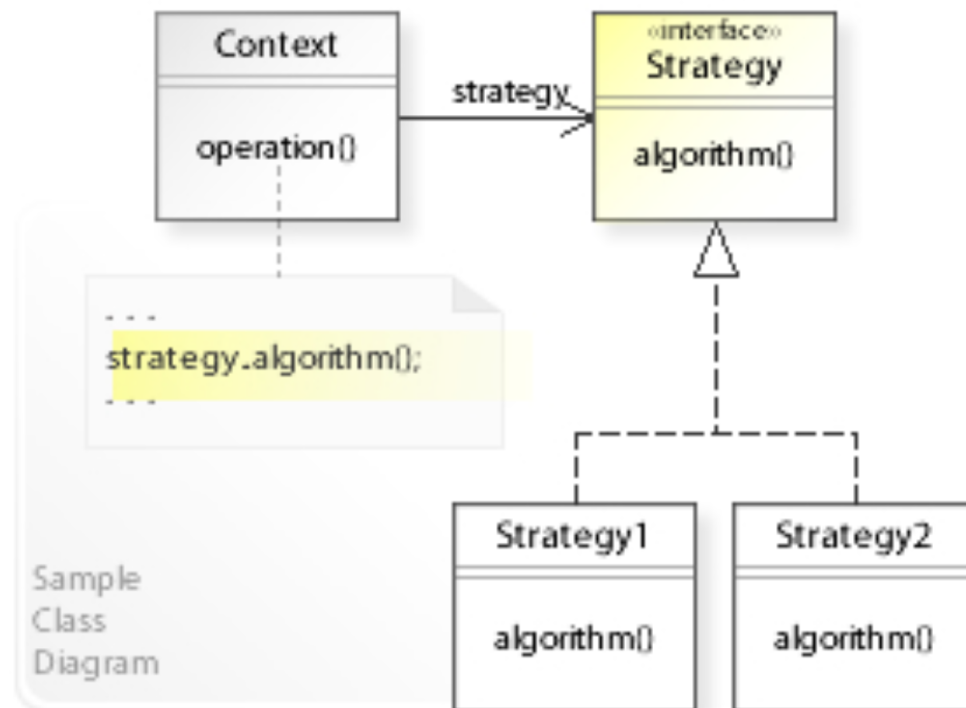
}

class Closed extends State {

}

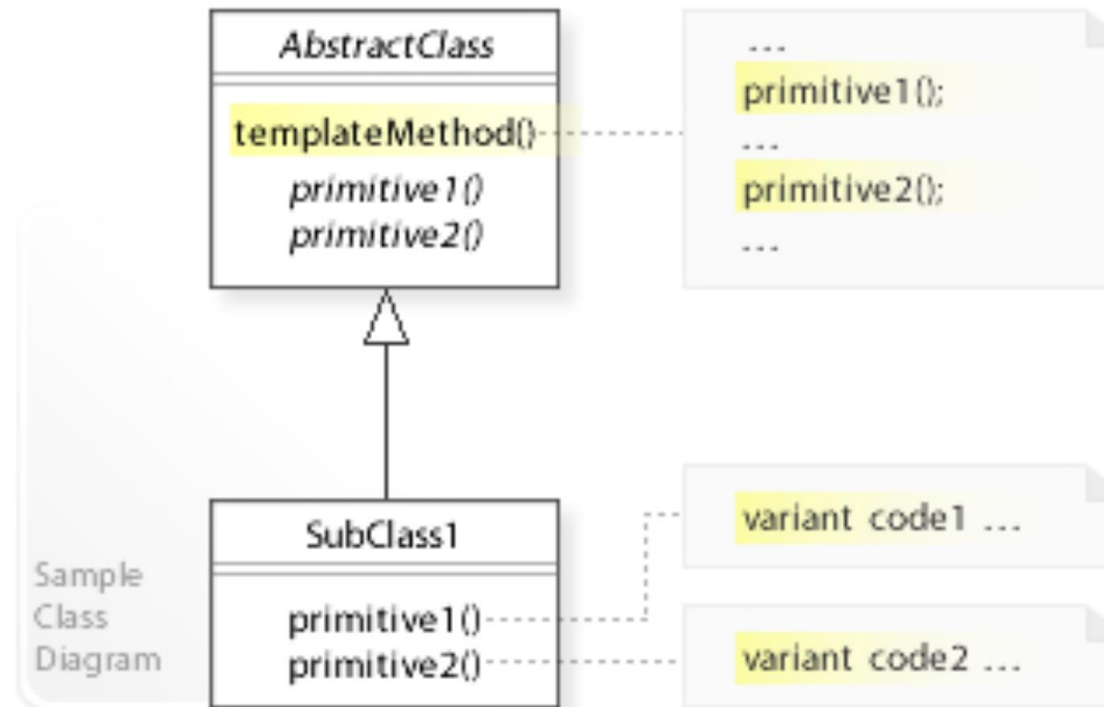
Strategy

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



Template Method

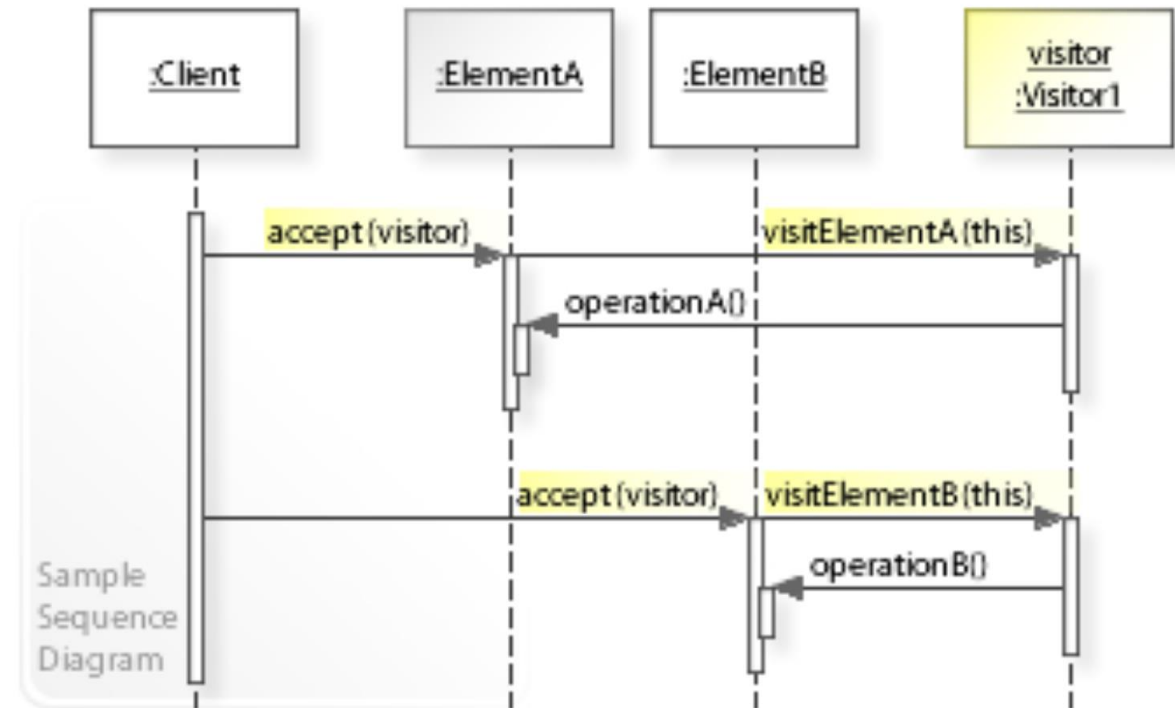
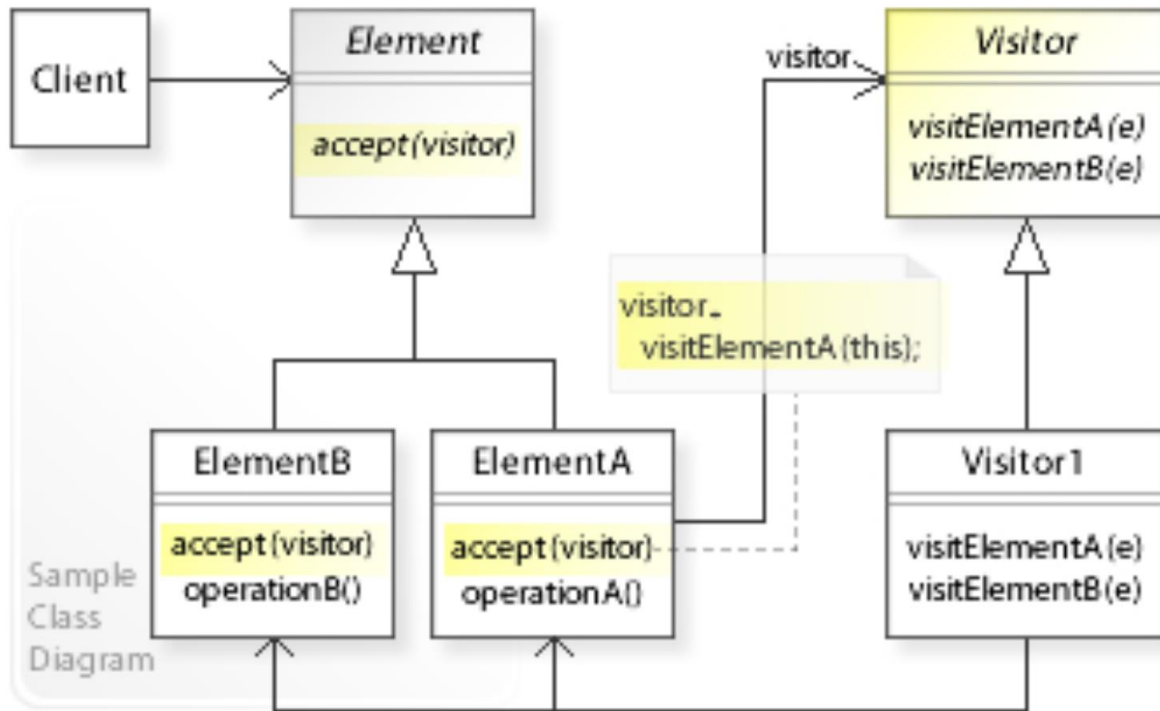
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Visitor

Represent an operation to be performed on instances of a set of classes.

Visitor lets a new operation be defined without changing the classes of the elements on which it operates.



Example Visitor

Used in almost all Compilers, AST, Interpreter
(example javac, Eclipse AST)
For all Compile Phases

```
abstract class ASTNode {  
    public abstract void accept(Visitor v);  
}  
  
abstract class FooNode extends ASTNode {  
    @Override public void accept(Visitor v) {  
        v.visitFoo(this);  
    }  
}
```

```
abstract class Visitor {  
    public abstract void visitFoo(FooNode p);  
    public abstract void visitBar(BarNode p);  
    public abstract void visitBaz(BazNode p);  
}
```

Using Visitor ..

```
class XXXVisitor extends Visitor {  
    private int xxxResult;  
  
    @Override  
    public void visitFoo(FooNode p) {...}  
  
    @Override  
    public void visitBar(BarNode p) {...}  
  
    @Override  
    public void visitBaz(BazNode p) {...}  
  
}
```

```
ASTNode node = ...
```

```
XXXVisitor visitor = new XXXVisitor();  
node.accept(visitor);  
int xxxResult = Visitor.getXXXResult();
```

Note on Visitor

Method « `accept()` » => does « **switch** »
method « `visit()` » => does the « **case** »,
usually calling **recursing** «so visiting tree »

Very efficiently : using virtual table of abstract class

... much more efficient than

```
if (x instanceof Foo) {  
    caseFoo((Foo) x);  
} else if (x instanceof Bar) {  
    caseBar((Bar) x);  
} ...
```

Fluent Interface

Design an API to be method chained so that it reads like a DSL.

Each method call returns a context through which the next logical method call(s) are made available.

No UML diagram ... depends of the DSL

Example Fluent Interface: lombok @Builder

```
@Builder
class A {
    private int field1, field2;
}
// ➡
class Builder {
    private int field1, field2;

    public Builder field1(int p) {
        this.field1 = p;
        return this;
    }
    public Builder field2(int p) {
        this.field2 = p;
        return this;
    }
    public A build() {...}
}
```

```
A a = A.builder()
    .field1 ( 10 )
    .field2 (20 )
    .build();
```

Example Fluent interfaces... Springboot conf

<https://spring.io/blog/2019/11/21/spring-security-lambda-dsl>

```
@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/blog/**").permitAll()
                .anyRequest().authenticated()
            .and()
            .formLogin()
                .loginPage("/login")
                .permitAll()
            .and()
            .rememberMe();
    }
}
```

SpringBoot also use DSL + Callback

...simpler no push / pop context (cf « .and() »)

```
@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests(authorizeRequests ->
                authorizeRequests
                    .antMatchers("/blog/**").permitAll()
                    .anyRequest().authenticated()
            )
            .formLogin(formLogin ->
                formLogin
                    .loginPage("/login")
                    .permitAll()
            )
            .rememberMe(withDefaults());
    }
}
```

How many design patterns in previous Slide?



HINTS ... Design pattern...

- It uses Dependency Injection
- DSL
- Class name is suffixed « Adapter », but is it?
- Maybe also a « Template method»
- « http » param is a Builder, also « authorizeRequests », « formLogin »?
- Using Functional interface, to delegates
- Authorizations use chain of responsibility
- Some authorization are adapter/bridges using ant patterns
- rememberMe looks like a Memento
- Etc...

Questions ?