An Introduction to Patterns

Robert B. France
Colorado State University

What is a Pattern? - 1

- Work on software development patterns stemmed from work on patterns from building architecture carried out by Christopher Alexander.
- Patterns are intended to capture the best available software development experiences in the form of problem-solution pairs.
- A pattern outlines solutions to a family of software development problems in a particular context.
- A pattern outlines a process for transforming problems targeted by the problem to solutions characterized by the pattern.

What is a Pattern? - 2

- Formally, a pattern consists of
  - A characterization of a family of problems
    - Determines the design problems that the pattern targets
  - A characterization of a family of solutions
    - Defines solutions for the design problems targeted by the pattern
  - A set of transformation guidelines
    - Guidelines for transforming a problem design to a solution characterized by the pattern

Example of a Software Development Pattern - The Model-View-Controller (MVC) Pattern

- **Context**: Developing user-interfaces (UIs)
- **Problem**: How to create a UI that's resilient to changes such as changes in look-and-feel windowing system, changes in functionality.
- **Factors**: changes to UI should be easy and possible at run-time; adapting or porting the UI should not impact the implementation of the core functionality.

Solution outline: Split application into 3 areas:

- The *Model* component: encapsulates core functionality; independent of input/output representations and behavior.
- The *View* components: displays data from the model component; there can be multiple views for a single model component.
- The *Controller* components: each view is associated with a controller that handles inputs; the user interacts with the system via the controller components.

Patterns Summary

- A pattern addresses a recurring software development problem that arises in a particular context, and outlines a solution for it.
- A pattern captures 'best practices' in software development (the intention!).
  - A pattern should be based on actual experiences in industry
- Patterns provide a common vocabulary for, and understanding of 'best practices'.
  - Developers can refer to a pattern by name (e.g., the Adapter pattern) and others familiar with the pattern will not need further description

Pattern Types

- **Requirements Patterns**: Characterize families of requirements for a family of applications
  - The checkin-checkout pattern can be used to obtain requirements for library systems, car rental systems, video systems, etc.
- **Architectural Patterns**: Characterize families of architectures
  - The Broker pattern can be used to create distributed systems in which location of resources and services is transparent (e.g., the WWW)
  - Other examples: MVC, Pipe-and-Filter, Multi-Tiered
- **Design Patterns**: Characterize families of low-level design solutions
  - Examples are the popular Gang of Four (GoF) patterns
- **Programming idioms**: Characterize programming language specific solutions

Gang of 4 (GoF) Design Pattern Classification

Purpose categories described in "Design Patterns:…" by Gamma, Helm, Johnson and Vlissides; Addison-Wesley

- Creational
  - Patterns that can be used to make object creation more flexible (e.g., Abstract Factory) or restrict creation activities (e.g., Singleton).
  - Help make a system independent of how objects are created, composed, and represented (i.e., allows one to vary how objects are created, composed, and represented)
- Structural
  - Concerned with creating flexible mechanisms for composing objects to form larger of structures
- Behavioral
  - Concerned with creating flexible algorithms and with assigning responsibilities to classes

Scope Criterion

Scope determines whether a pattern applies to classes or objects

- Classes
  - static relations
  - inheritance structures
  - Examples: Factory Method (creation); Adapter (structural); Interpreter (behavioral)
- Objects
  - object relations
  - Dynamic
  - Examples: Abstract Factory (creation); Bridge (structural); Iterator (behavioral)

Creational Design Patterns

Overview of Creational Patterns

- Separate object creation from representation.
- Flexibility in what gets created, who creates objects, how objects are created, and when objects are created.
- *Class Patterns*: use inheritance to vary class of instantiated object
- *Object Patterns*: delegate instantiation to another object.

Overview of creational class and object patterns

- Creational class patterns
  - defer creation to subclasses.
- Creational object patterns
  - defer creation to objects

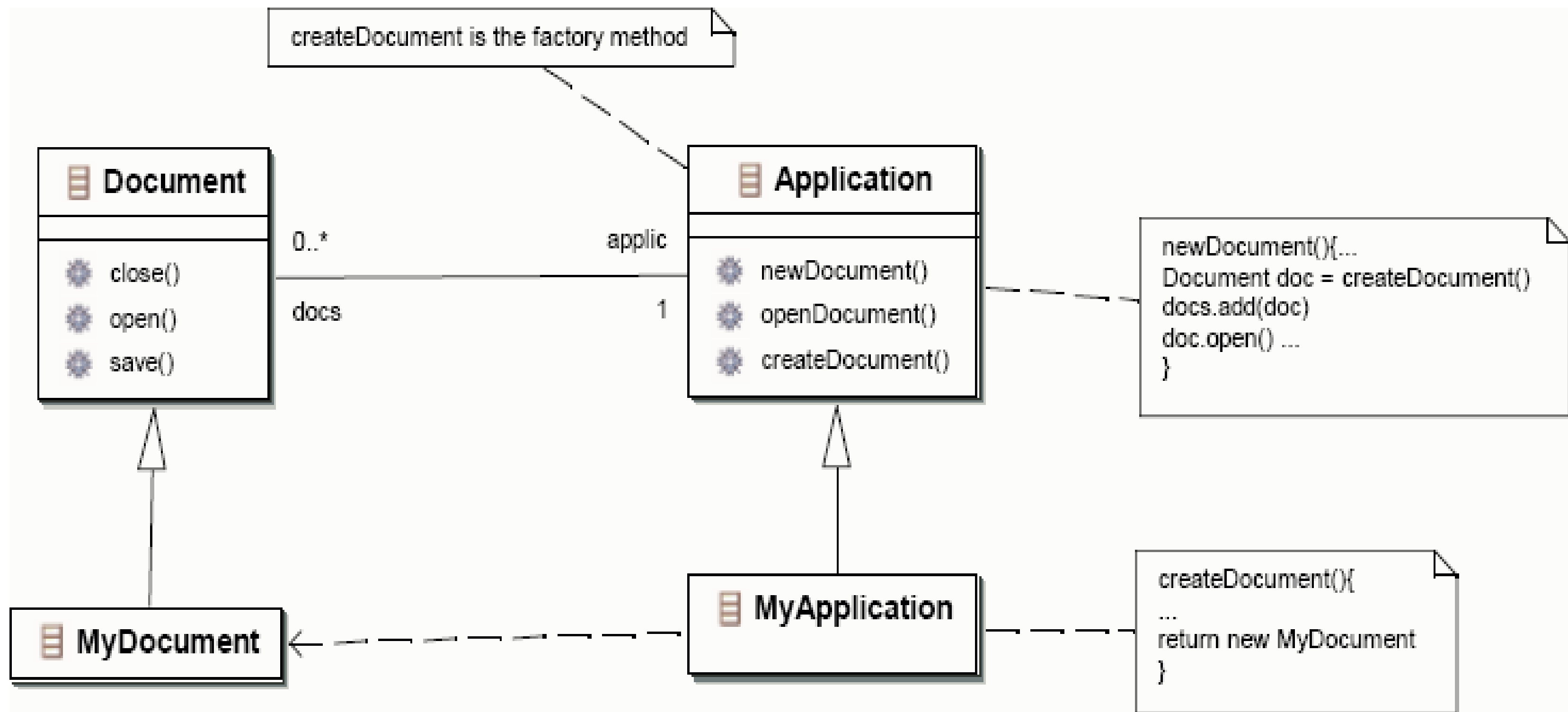
The Factory Method Pattern

- Defers object instantiation to subclasses: A factory method defines the interface of an operation that creates objects. The implementation of the operation is provided by subclasses.
- Abstract operation may implement a default implementation.
- Knowledge of what objects to create are encapsulated in subclasses.

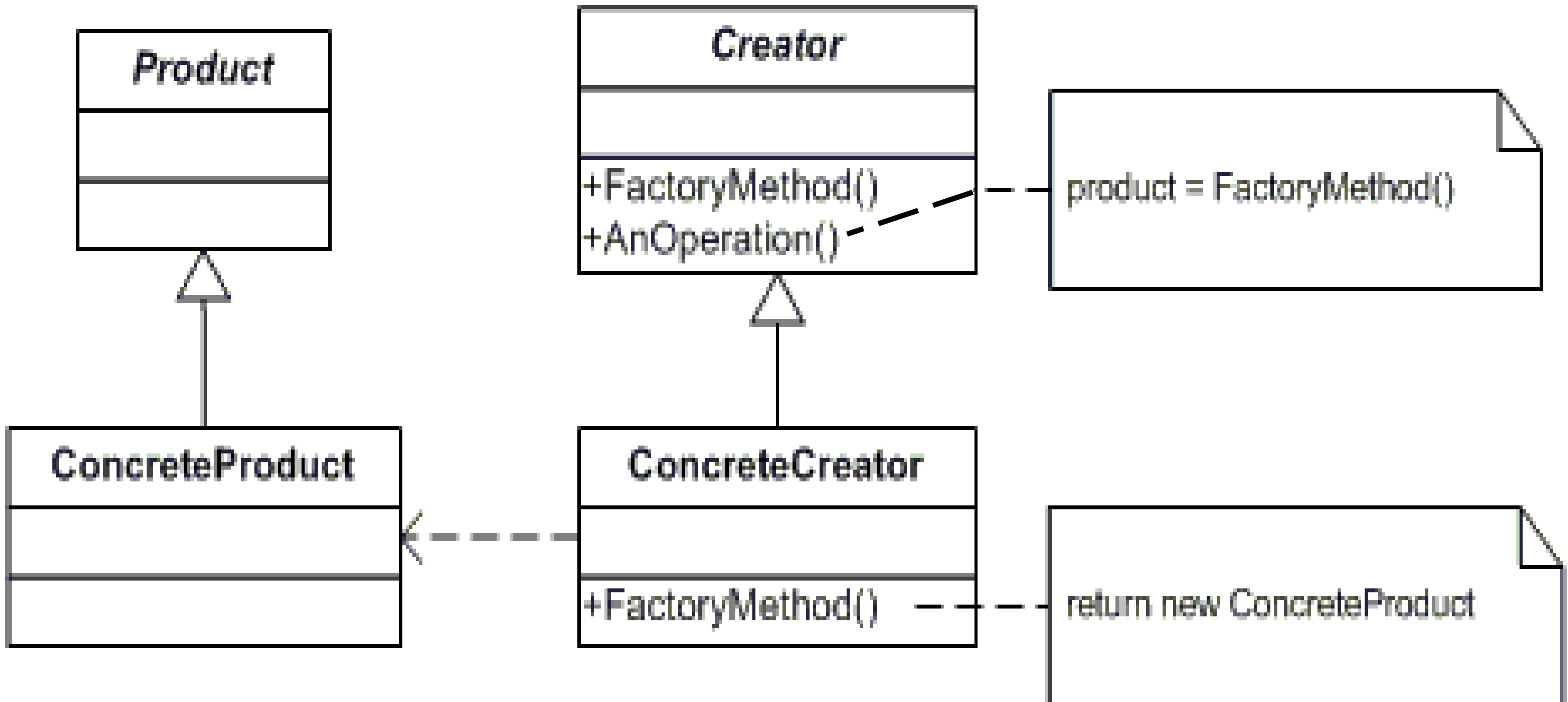
Application/Document Example

- Application class manages multiple documents of different types.
- Contains operations for manipulating documents.
- PROBLEM: Application knows when to create documents but does not know what types of documents to create.
- SOLUTION: Encapsulate knowledge of creation of concrete documents in subclasses and defer implementation to these subclasses.

Factory Method Solution



Solution Structure



Applicability

- Class cannot anticipate type of objects to create.
- Class wants its subclasses to determine type of objects to create.
- Class delegates object creation responsibility to a select set of subclasses.

Consequences

- Creation of a new product class may require creating a new Creator subclass.
- Gives subclasses a ‘hook’ for creating an extended version of an object.
- Factory methods can be called by clients.

Implementation

2 varieties

- Creator does not provide a default implementation: requires creation of concrete subclasses.
- Creator provides default implementation: subclasses can either inherit method as is or redefine it.

What design changes are accomodated?

- Changes to the type of products that can be created by a creator class
  - A new subclass can be added for each type of product that is needed.

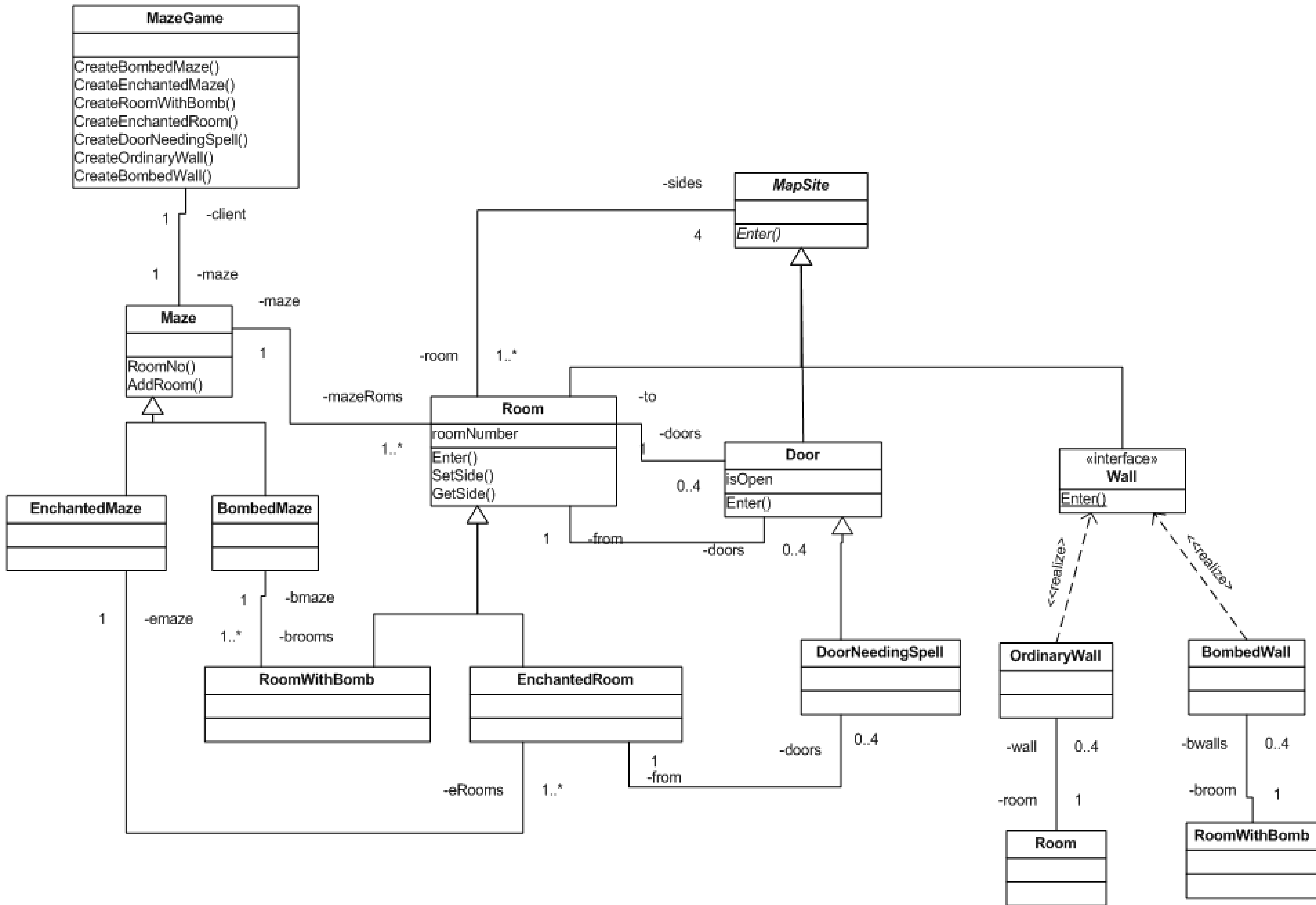
What changes are difficult to handle?

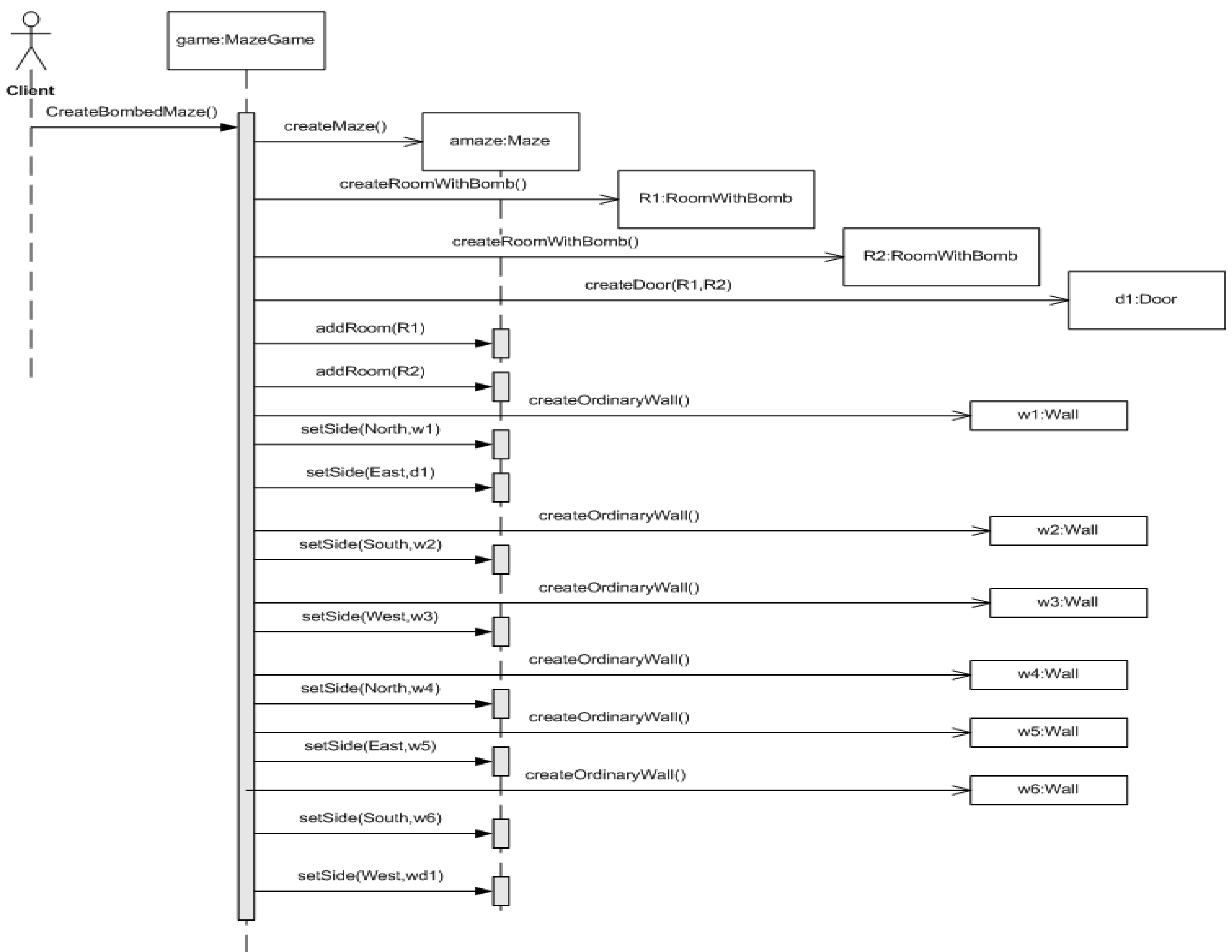
- One cannot add a new product to be created during run-time
- If the product consists of subparts one cannot vary the the types of subparts used.
  – Abstract factory or builder is needed for this purpose

Abstract Factory Pattern

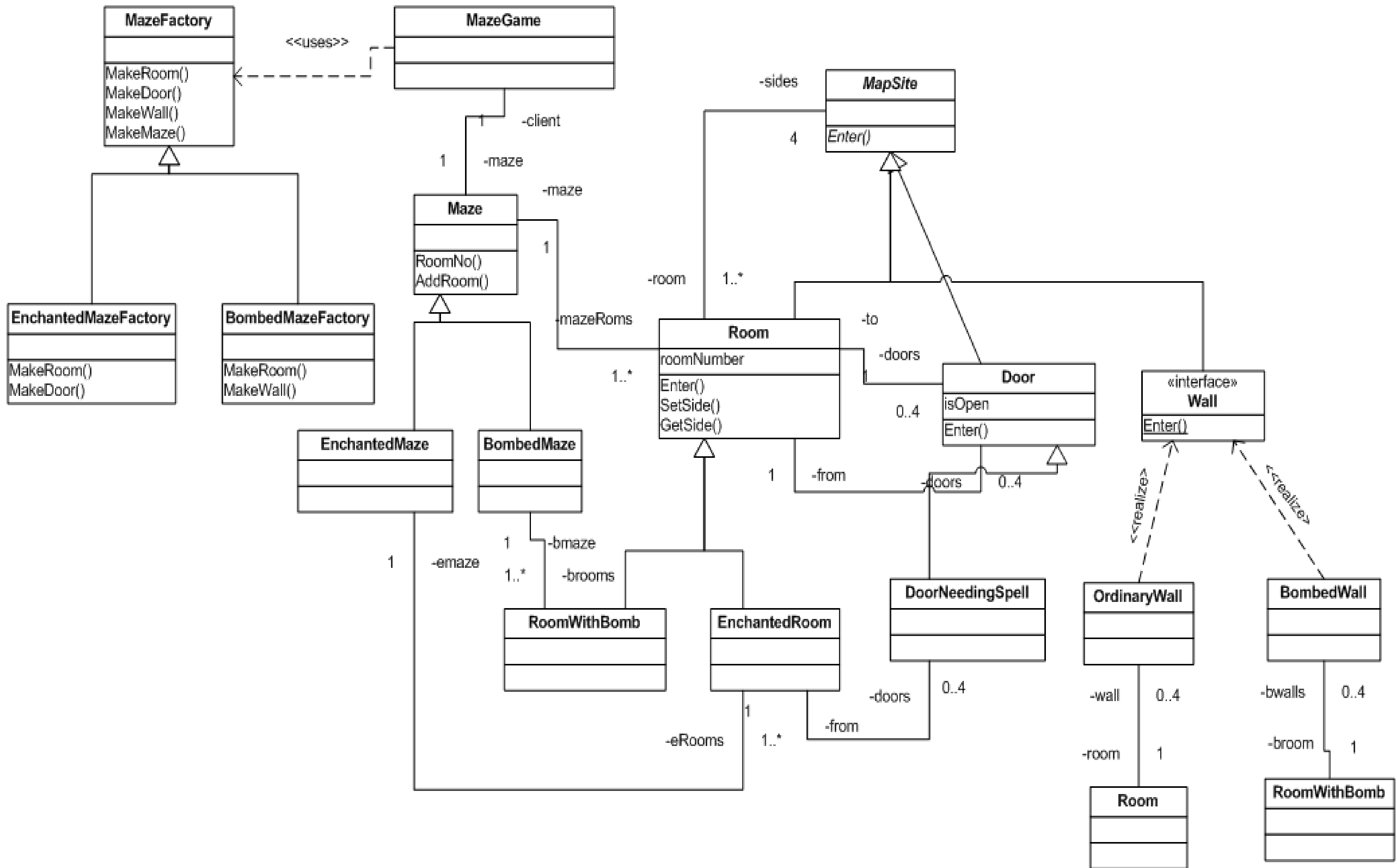
- Used for building composite objects.
- Isolates clients from implementation of components by providing an abstract interface.
- Enforces creation using only compatible components.

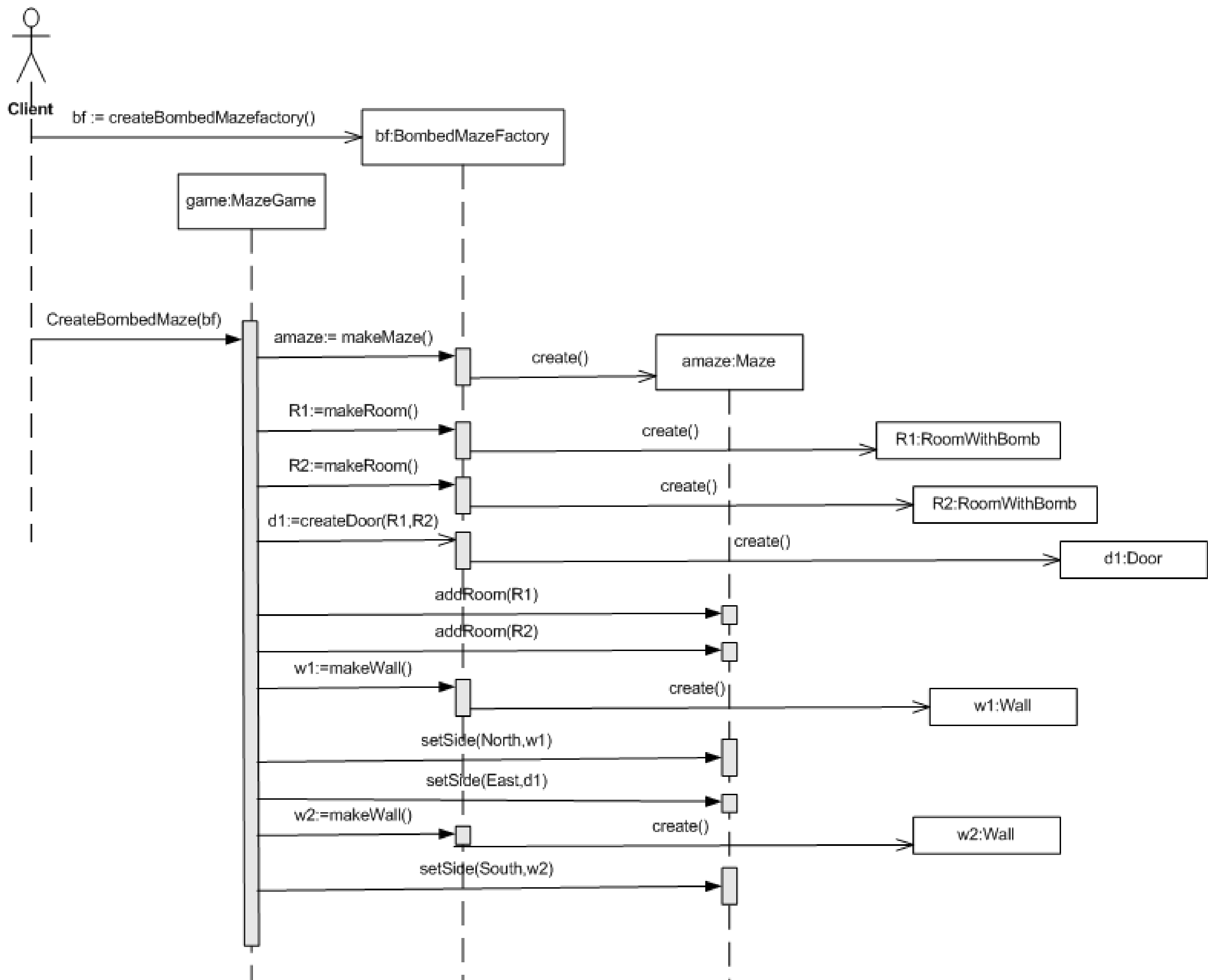
Maze Game Class Model (not all operations shown)



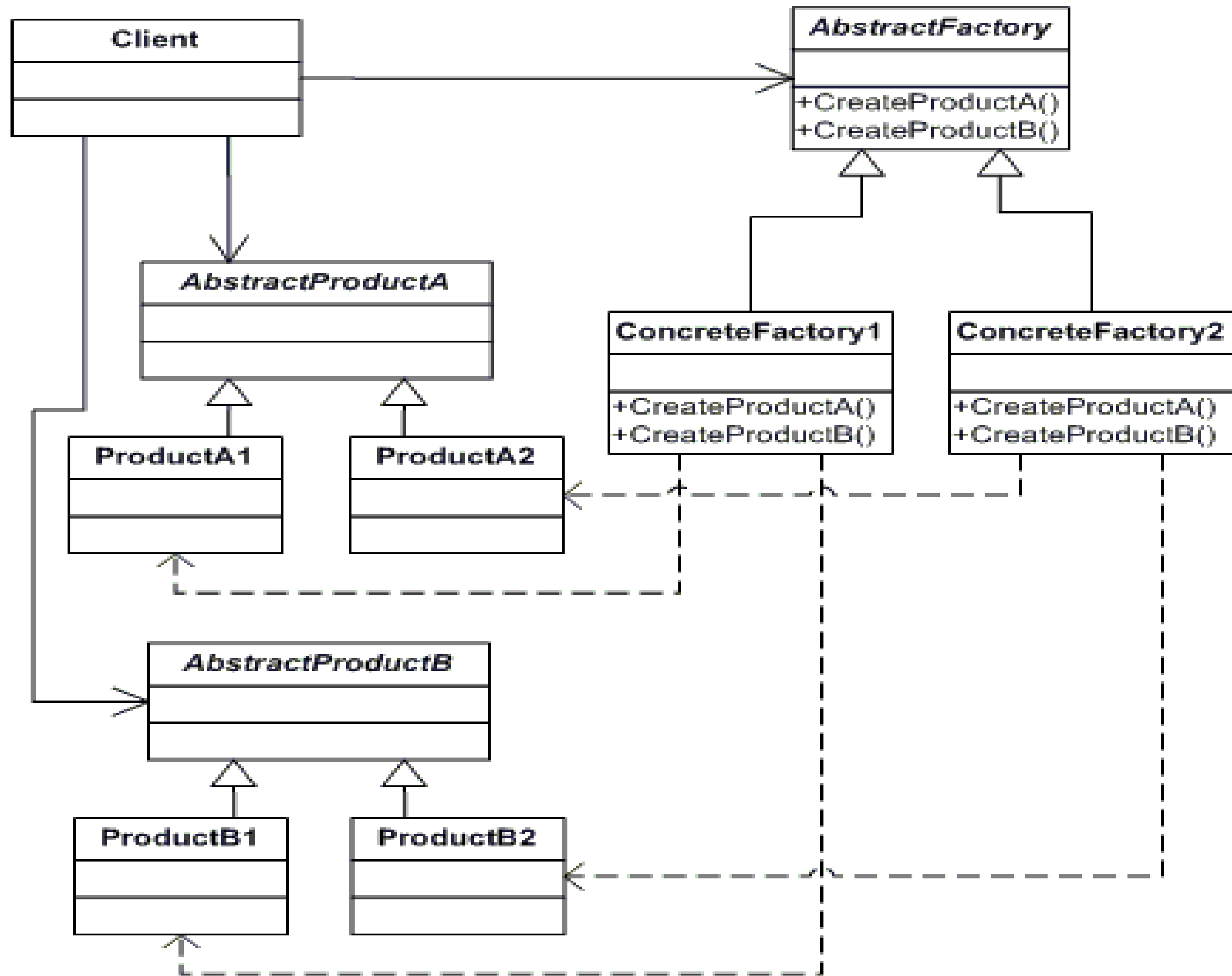

Client
game:MazeGame
CreateBombedMaze()
createMaze()
amaze:Maze
createRoomWithBomb()
R1:RoomWithBomb
createRoomWithBomb()
R2:RoomWithBomb
createDoor(R1,R2)
d1:Door
addRoom(R1)
addRoom(R2)
createOrdinaryWall()
w1:Wall
setSide(North,w1)
setSide(East,d1)
createOrdinaryWall()
w2:Wall
setSide(South,w2)
createOrdinaryWall()
w3:Wall
setSide(West,w3)
createOrdinaryWall()
w4:Wall
setSide(North,w4)
createOrdinaryWall()
w5:Wall
setSide(East,w5)
createOrdinaryWall()
w6:Wall
setSide(South,w6)
setSide(West,wd1)

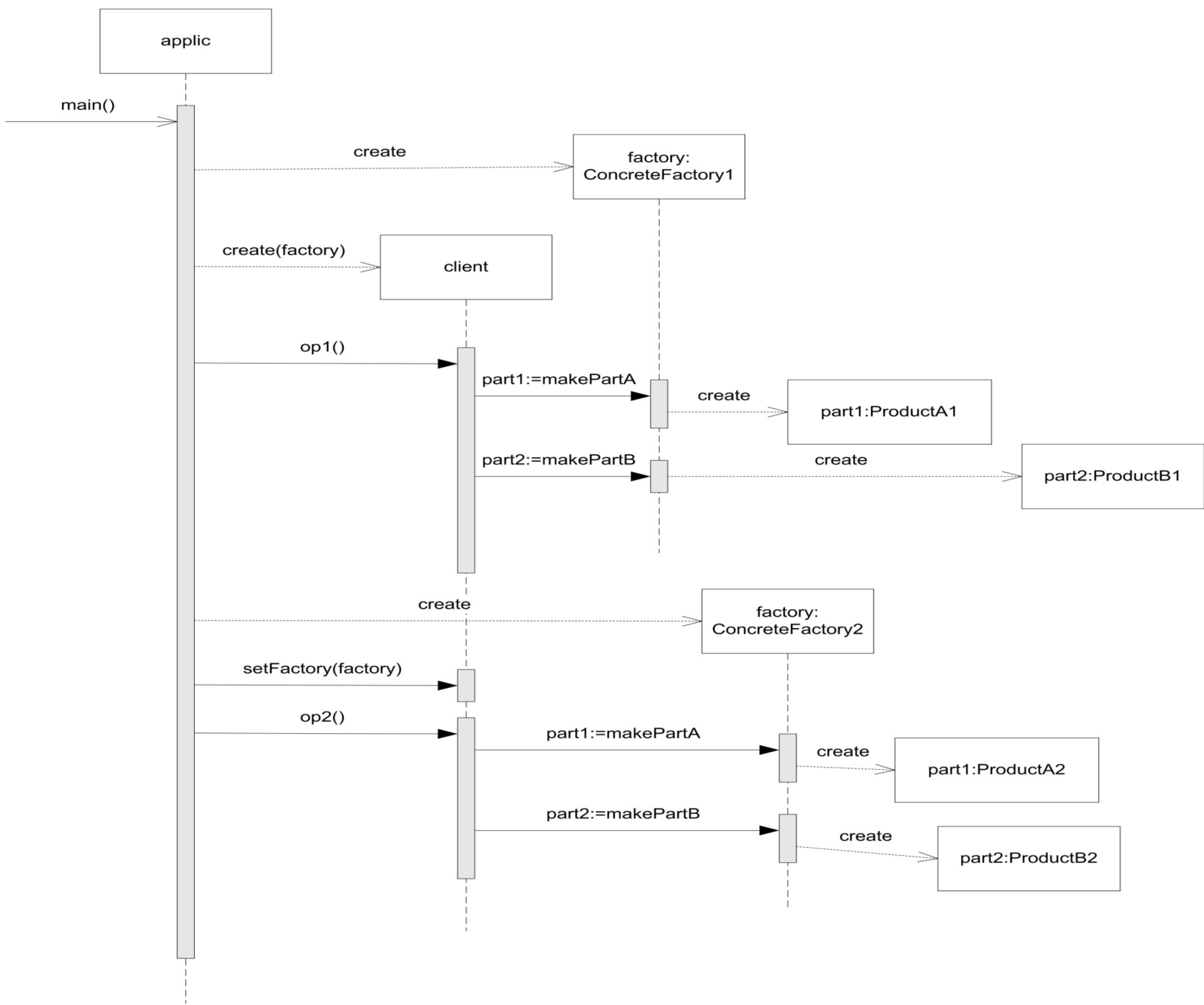
Maze Game with Abstract Factory

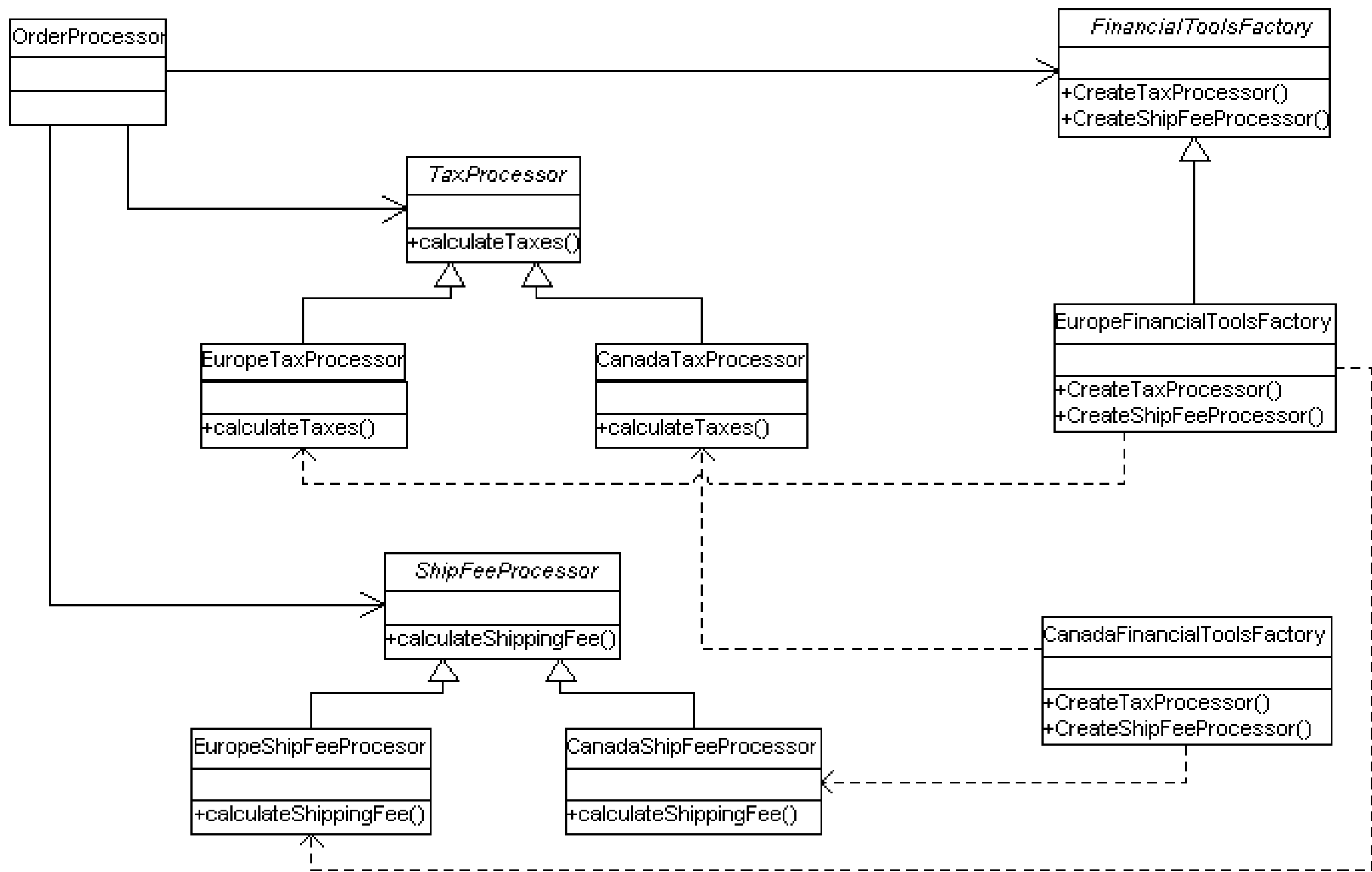


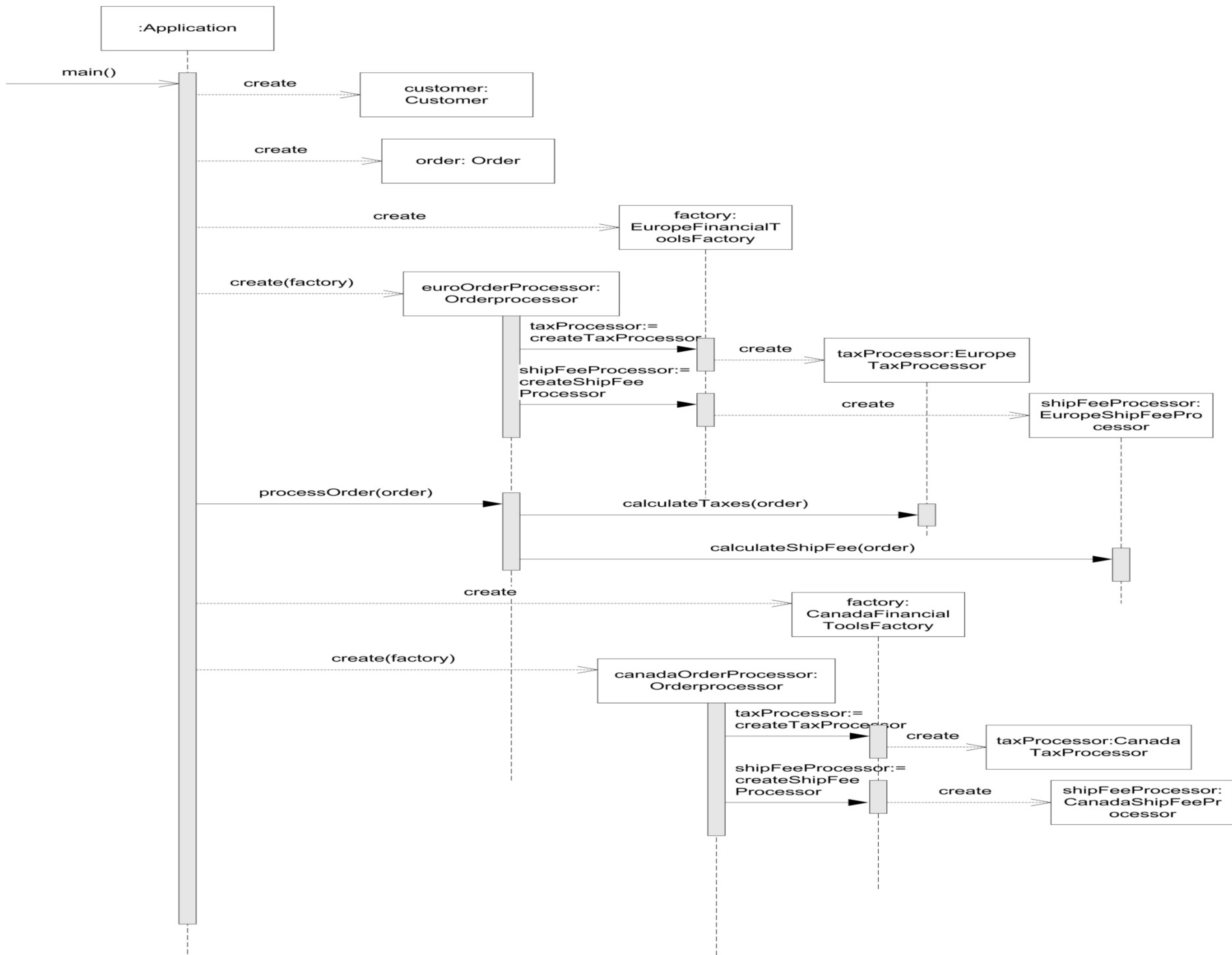

Client
bf := createBombedMazefactory()
bf:BombedMazeFactory
game:MazeGame
CreateBombedMaze(bf)
amaze:= makeMaze()
create()
amaze:Maze
R1:=makeRoom()
create()
R1:RoomWithBomb
R2:=makeRoom()
create()
R2:RoomWithBomb
d1:=createDoor(R1,R2)
create()
d1:Door
addRoom(R1)
addRoom(R2)
w1:=makeWall()
create()
w1:Wall
setSide(North,w1)
setSide(East,d1)
w2:=makeWall()
create()
w2:Wall
setSide(South,w2)

Solution Structure




applic
main()
create
factory:
ConcreteFactory1
create(factory)
client
op1()
part1:=makePartA
create
part1:ProductA1
part2:=makePartB
create
part2:ProductB1
create
factory:
ConcreteFactory2
setFactory(factory)
op2()
part1:=makePartA
create
part1:ProductA2
part2:=makePartB
create
part2:ProductB2


OrderProcessor
FinancialToolsFactory
+CreateTaxProcessor()
+CreateShipFeeProcessor()
TaxProcessor
+calculateTaxes()
EuropeTaxProcessor
+calculateTaxes()
CanadaTaxProcessor
+calculateTaxes()
EuropeFinancialToolsFactory
+CreateTaxProcessor()
+CreateShipFeeProcessor()
ShipFeeProcessor
+calculateShippingFee()
CanadaFinancialToolsFactory
+CreateTaxProcessor()
+CreateShipFeeProcessor()
EuropeShipFeeProcesor
+calculateShippingFee()
CanadaShipFeeProcessor
+calculateShippingFee()


:Application
main()
create
customer:
Customer
create
order: Order
create
factory:
EuropeFinancialT
oolsFactory
create(factory)
euroOrderProcessor:
Orderprocessor
taxProcessor:=
createTaxProcessor
create
taxProcessor:Europe
TaxProcessor
shipFeeProcessor:=
createShipFee
Processor
create
shipFeeProcessor:
EuropeShipFeePro
cessor
processOrder(order)
calculateTaxes(order)
calculateShipFee(order)
create
factory:
CanadaFinancial
ToolsFactory
create(factory)
canadaOrderProcessor:
Orderprocessor
taxProcessor:=
createTaxProcessor
create
taxProcessor:Canada
TaxProcessor
shipFeeProcessor:=
createShipFee
Processor
create
shipFeeProcessor:
CanadaShipFeePr
ocessor

Benefits

- Separates clients from implementation of created objects.
- Product family can be changed easily.
- Promotes consistency among products: enforces use of compatible parts.

Drawback

Difficult to introduce a new kind of product family.

- Interface fixes components that are to be created.

What design changes are accomodated?

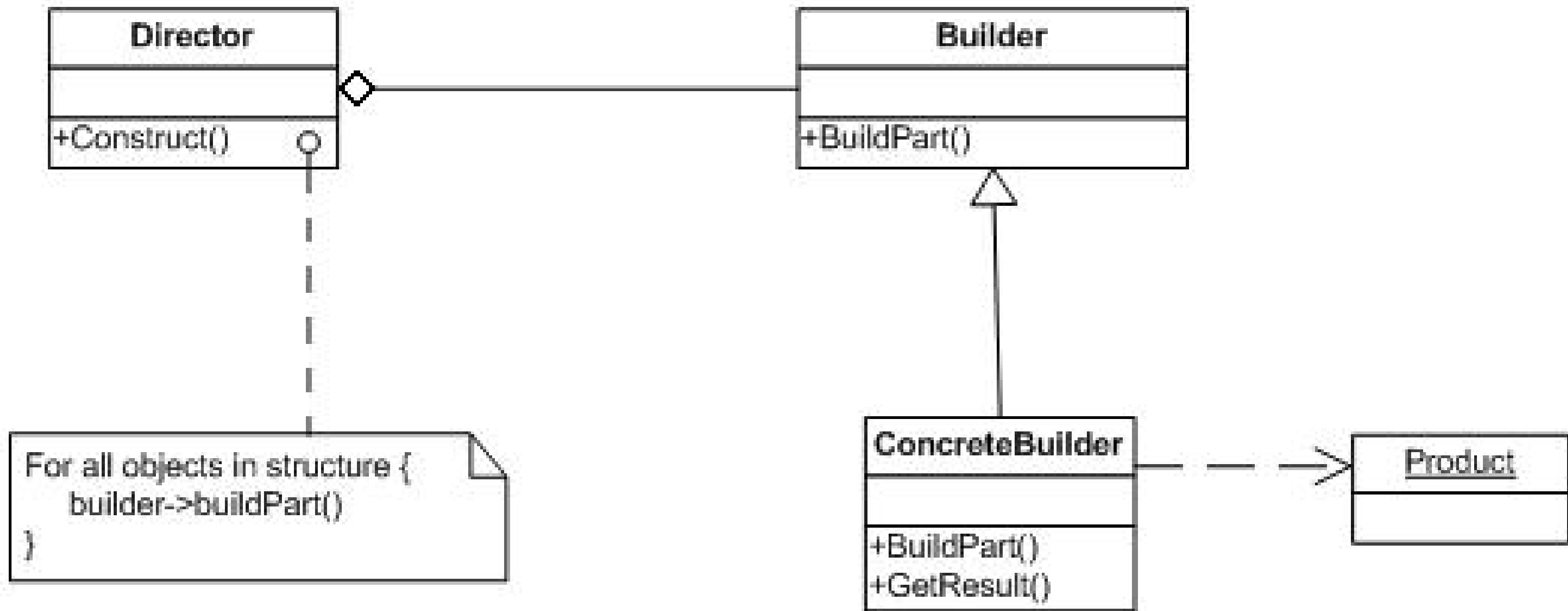
- Can create a new product family during run-time by passing in a specialized factory.
- A new class of members of the family can be introduced during design by creating a factory subclass.

What changes are difficult to handle?

- Cannot introduce a new product family with different parts easily
  - Interface fixes components that are to be created.

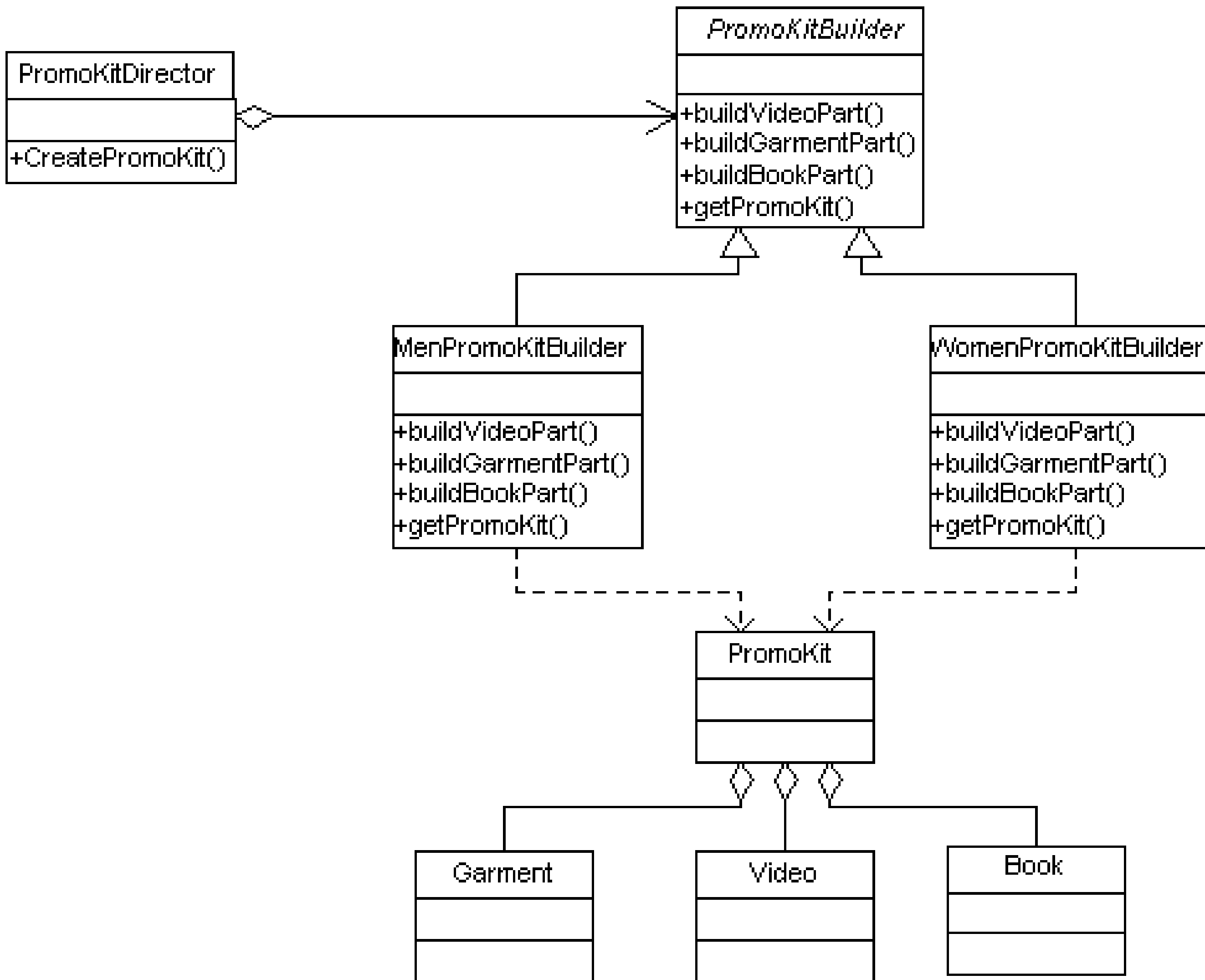
Builder Pattern

- Abstract construction steps of object structures so that different implementations of these steps can create different forms of objects.



Builder: Abstract interface for creating objects
Concrete Builder: Provides implementations of creation operations in Builder
Director: Builds an object structure using a Builder object

Example



```
public class PromoKitDirector {
public PromoKit createPromoKit(
   PromoKitBuilder builder) {
         builder.buildVideoPart();
         builder.buildGarmentPart();
         builder.buildBookPart();
         return builder.getPromoKit();
         }
}
```

```
// Integration with overal application
public class Application {
    public static void main(String[] args) {
    String gendre = "M";
    PromoKitDirector director = new PromoKitDirector();
    PromoKitBuilder promoKitBuilder = null;
    if (gendre.equals("M")) {
            promoKitBuilder = new MenPromoKitBuilder();
            }
            else if (gendre.equals("F")) {
                    promoKitBuilder = new WomenPromoKitBuilder();
                    }
                    else {     // ....       }
    PromoKit result = director.createPromoKit(promoKitBuilder);
    }
}
```

Benefits

- Can vary internal representation of object structure (product)
  - Builder provides abstract interface to objects that build complex structures (Director objects)
  - Same builder object can be used with different directors
- Allows control over construction
  - Director can control when operations in builder are called
  - The director retrieves the product from the builder only after it is finished

Drawback

What are the drawbacks?

What design changes are accomodated?

What changes can be made with little effort?

What changes are difficult to handle?

Singleton Pattern

Used to ensure that only one instance of a class exists.

- The class keeps track of the sole instance and does not permit instantiation if an instance already exists.
- This is done by hiding the constructor from clients (but not the subclasses), and defining a static operation that creates an instance if and only if there are no instances.

Solution Structure

Singleton

-instance : Singleton

-Singleton()
+Instance() : Singleton

MyShapeController Factory

```
// MyShapeControllerFactory.java
// MyShapeControllerFactory uses the Factory Method design
// pattern to create an appropriate instance of MyShapeController
// for the given MyShape subclass.
package com.deitel.advjhtp1.drawing.controller;

import com.deitel.advjhtp1.drawing.model.*;
import com.deitel.advjhtp1.drawing.model.shapes.*;

public class MyShapeControllerFactory {

// reference to Singleton MyShapeControllerFactory
   private static MyShapeControllerFactory factory;

   // MyShapeControllerFactory constructor
   protected MyShapeControllerFactory() {}
```

```
// return Singleton instance of MyShapeControllerFactory
   public static final MyShapeControllerFactory getInstance()
   {
      // if factory is null, create new MyShapeControllerFactory
      if ( factory == null ) {
         factory = new MyShapeControllerFactory();
   } // end if

      return factory;

   } // end method getInstance
```

Structural Patterns

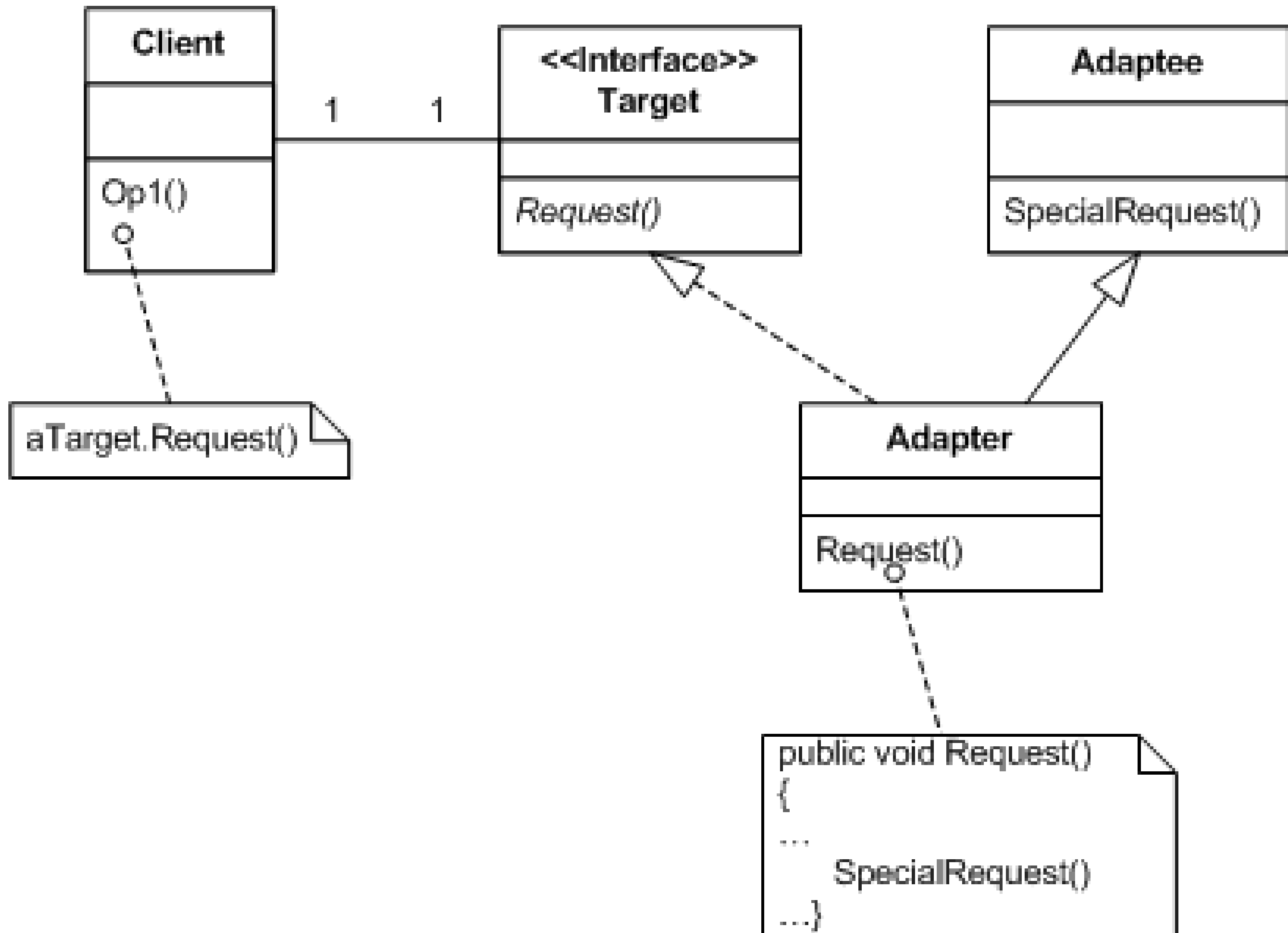
Adapter Pattern

- An adapter pattern converts the interface of a class into an interface that a client expects
- Adapters allow incompatible classes to work together
- Adapters can extend the functionality of the adapted class

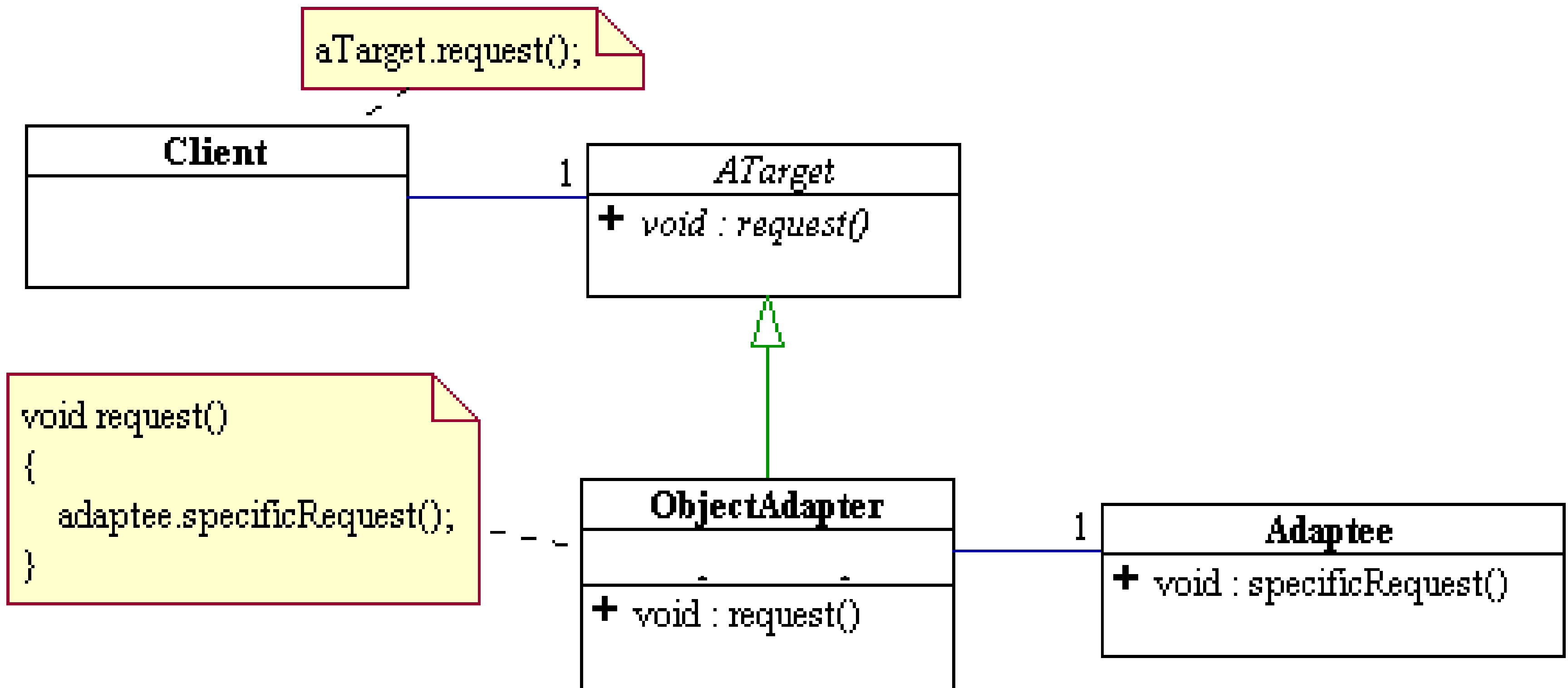
When to Use

- Need to adapt the interface of an existing class to satisfy client interface requirements
  - Adapting Legacy Software
  - Adapting 3rd Party Software

Class Adapter Pattern



Object Adapter Pattern



What changes are easily accomodated?

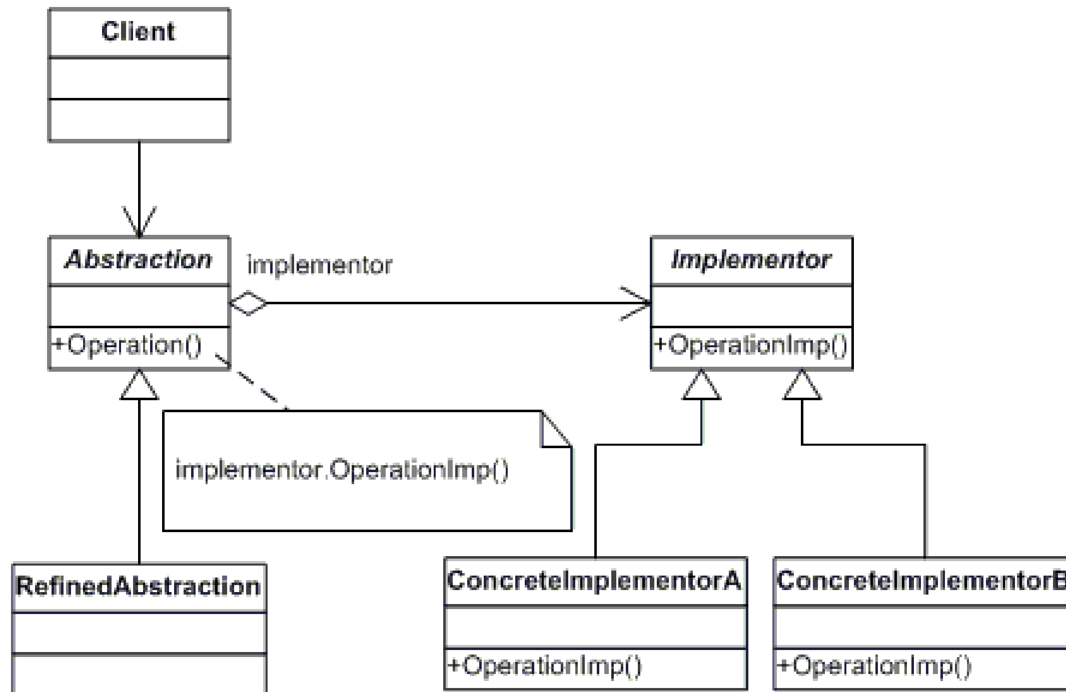
- One can use components with incompatible interfaces

What changes are difficult to handle?

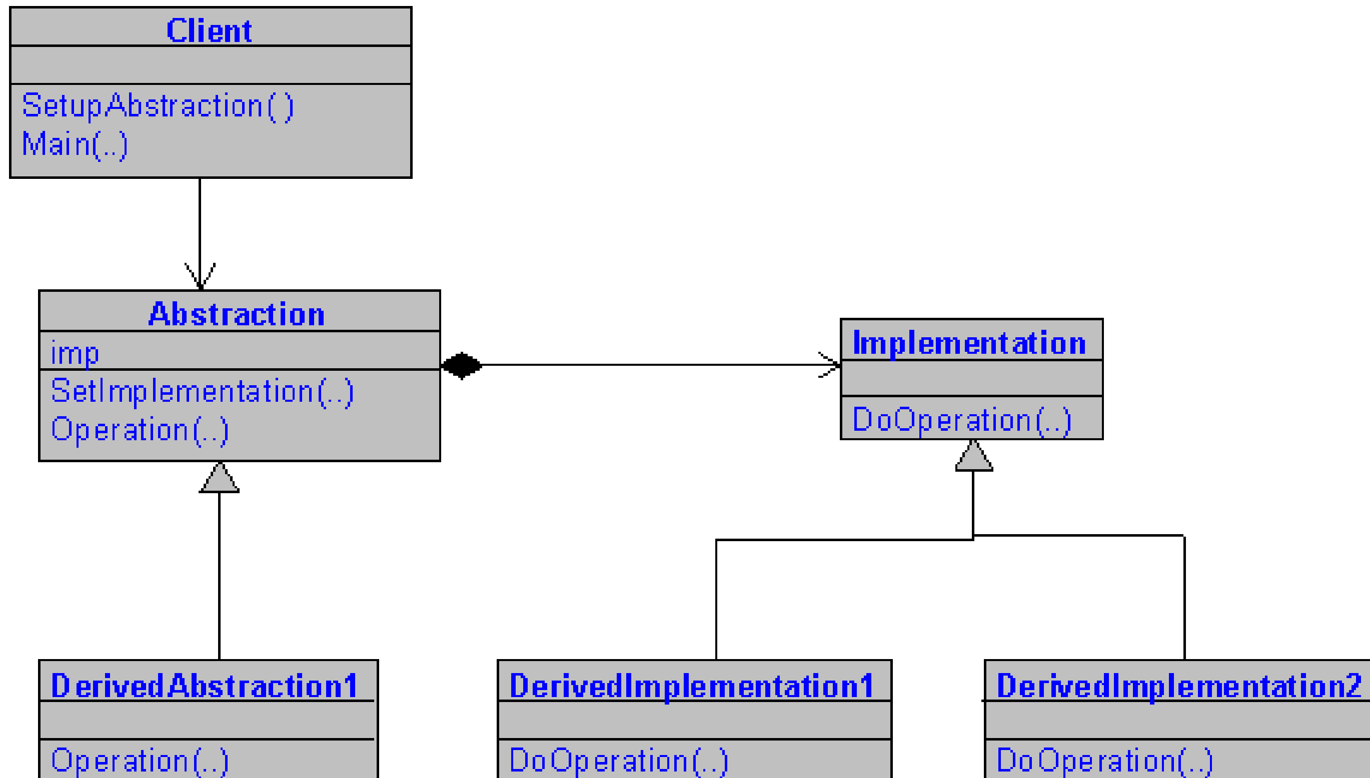
- When the adaptee provides only SOME of the functionality needed to handle a customer request, the additional functionality must be provided somewhere else (e.g., in the adapter)

Bridge Pattern

- Used to decouple implementations from abstraction when abstraction can have more than one implementation.



Bridge Pattern Example

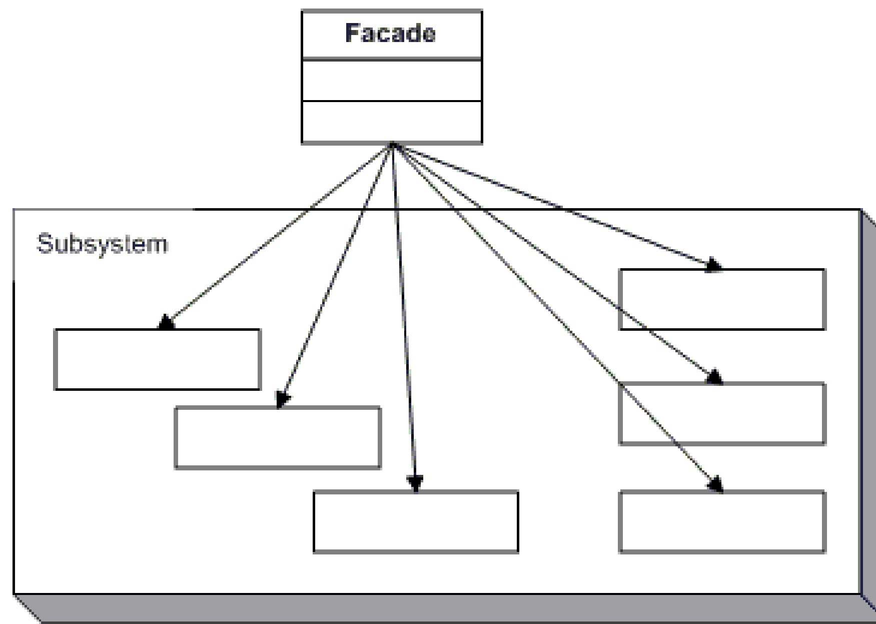


What changes are accomodated?

- One can vary the implementation associated with an abstraction during run-time.
- One can add new implementations during design by subclassing the implementation superclass

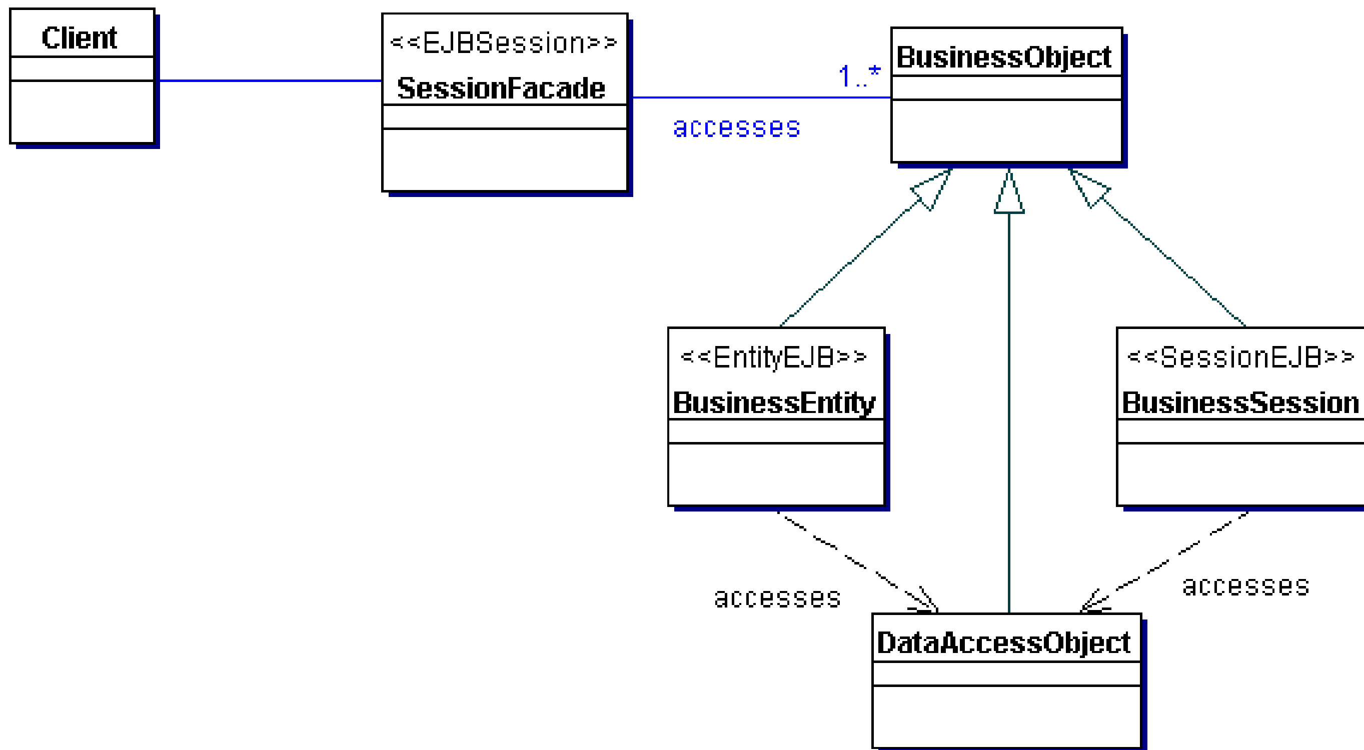
Façade Pattern

- Provides a unified interface to a set of interfaces in a subsystem
- Helps to minimize communication and dependencies across subsystems

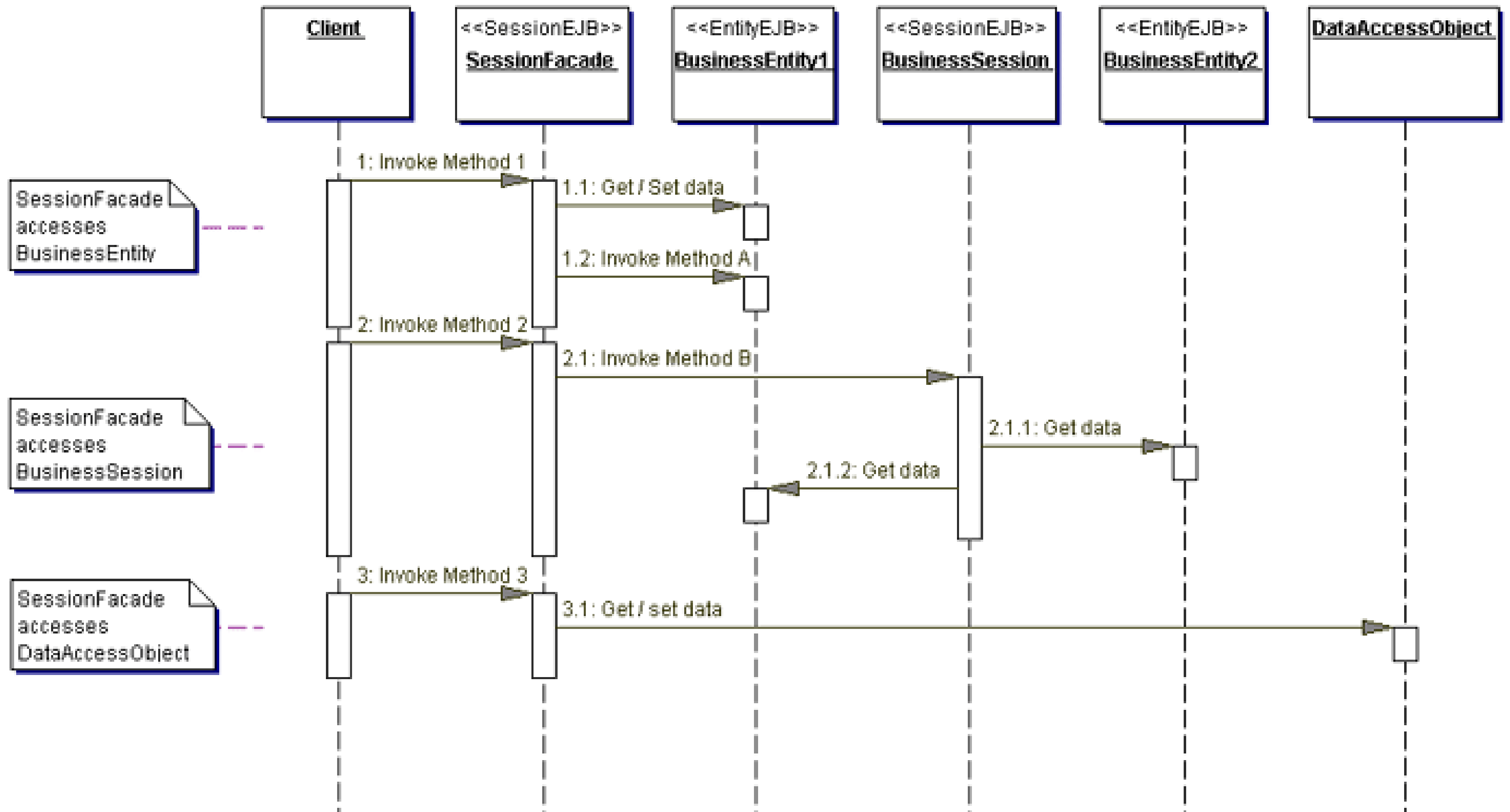


Façade Pattern Example

Use a session bean as a facade to encapsulate the complexity of interactions between the business objects participating in a workflow. The Session Facade manages the business objects, and provides a uniform service access layer to clients.



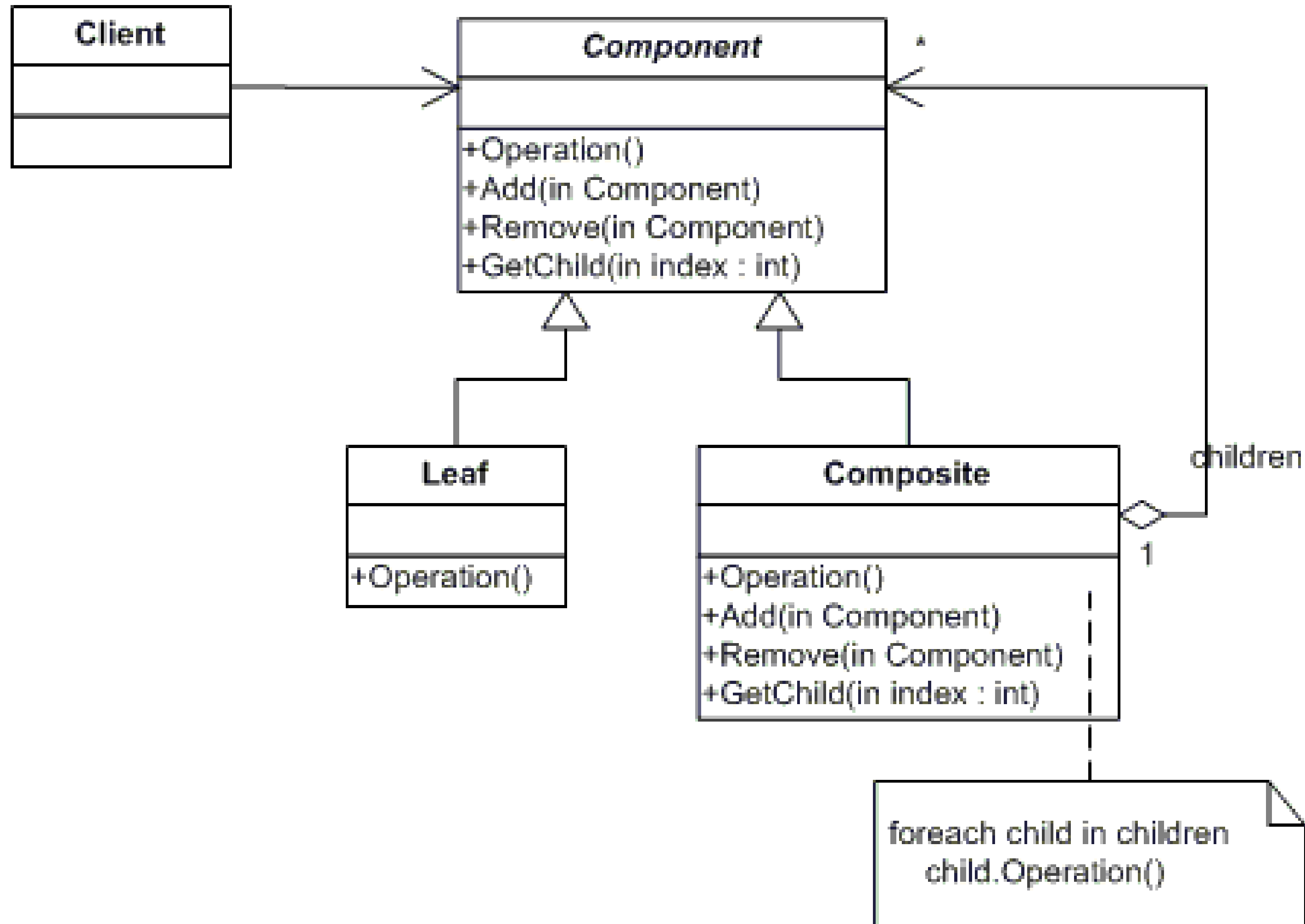
Façade Example – Sequence Diagram



Composite Pattern

- Use when one wants to treat elements in a composite structure uniformly

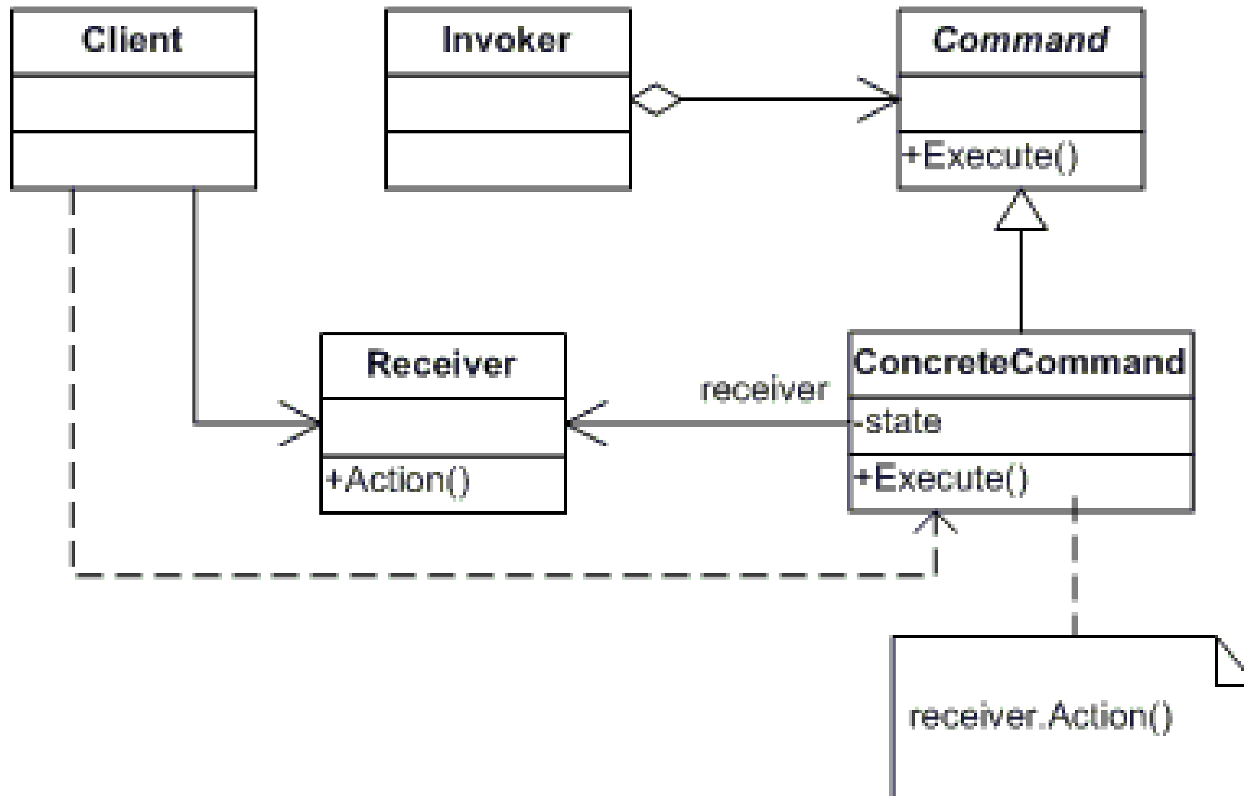
Composite Structure Example



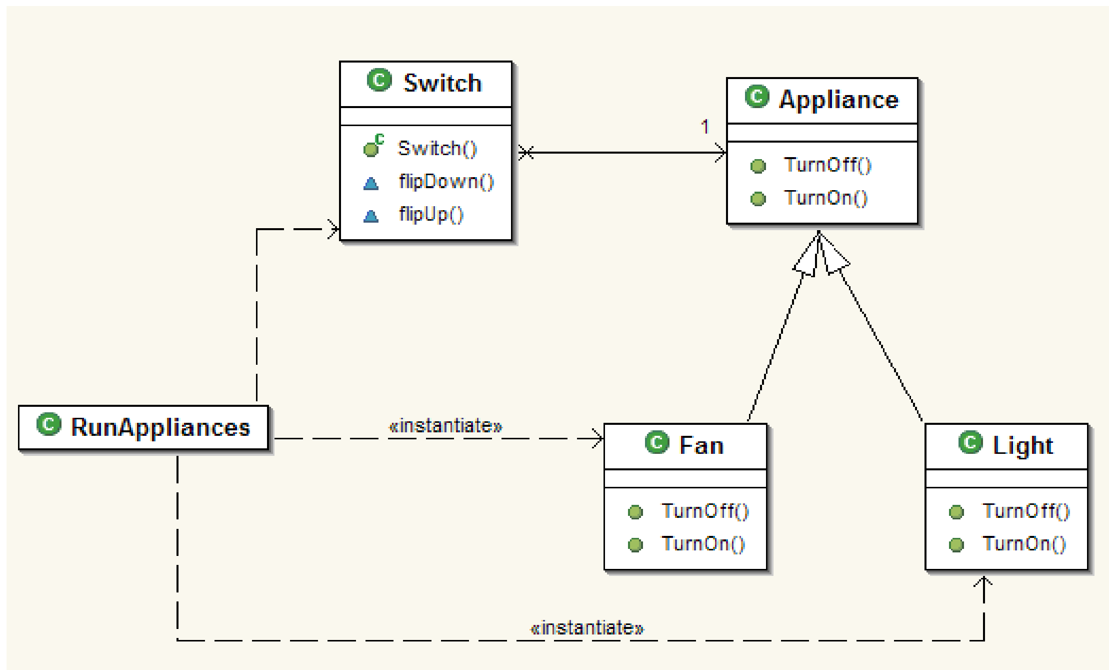
Behavioral Patterns

Command Pattern

- Encapsulates a request as an object so that clients can be parameterized with different requests



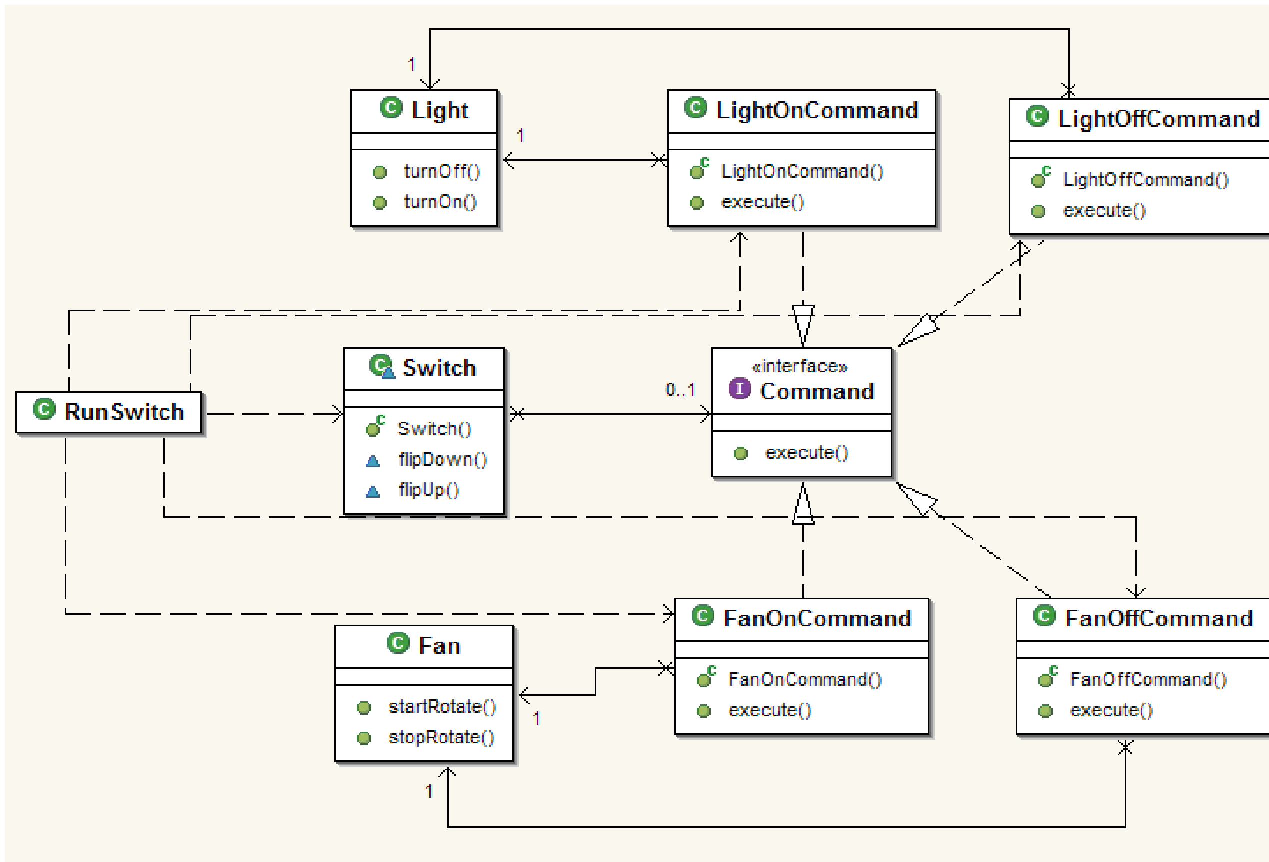
Using Polymorphism but not the Command Pattern



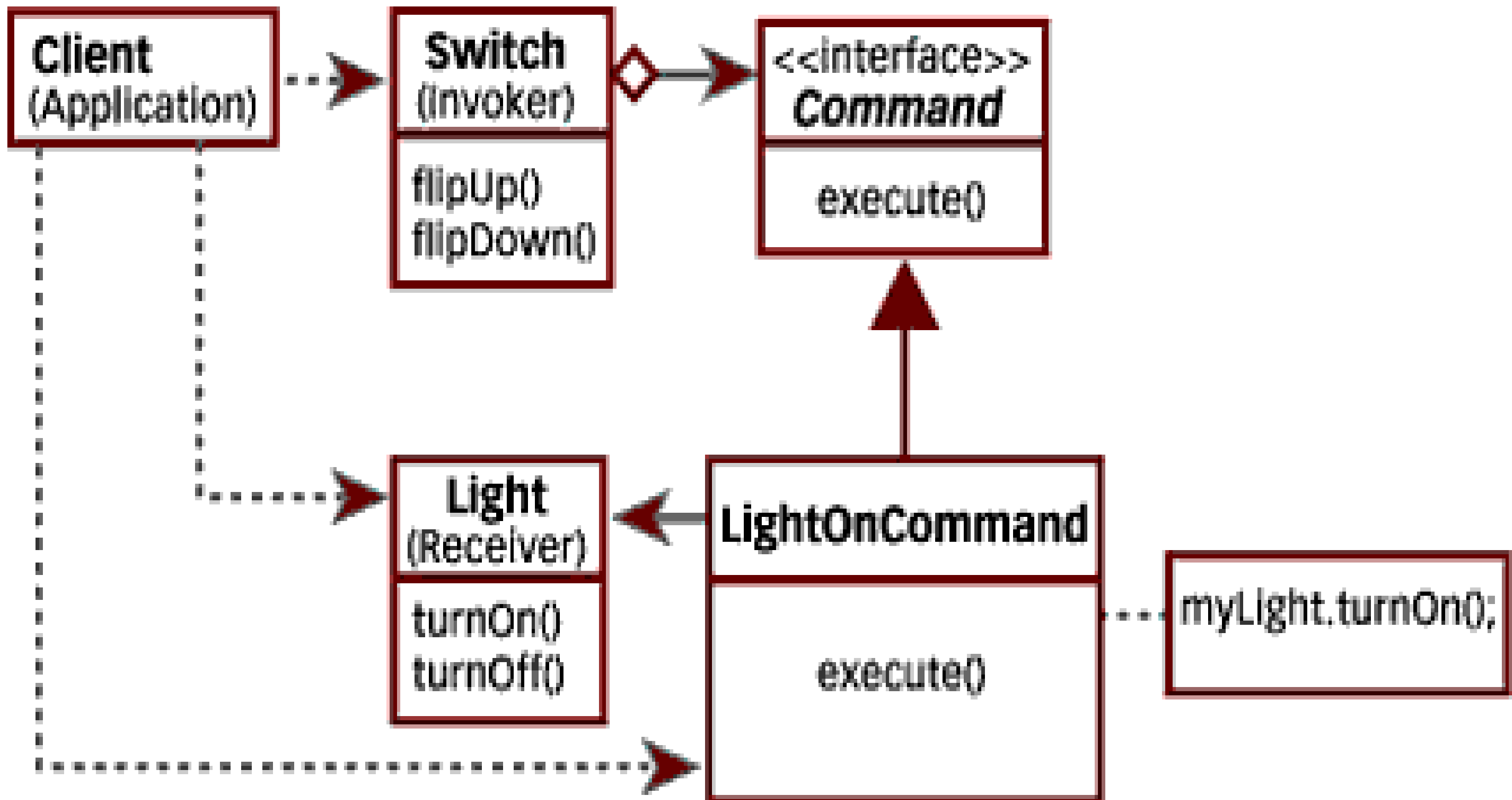
Thoughts on Non-Command

- What level of de-coupling is reached?
  - The switch doesn't have to call the individual classes turn on and off method, just a general appliance.
  - The switch still knows that it is attached to an appliance, this we should get rid of.
  - What happens if we want to change the command that is preformed on a light?

Command Pattern



A Website's model of the same concept.



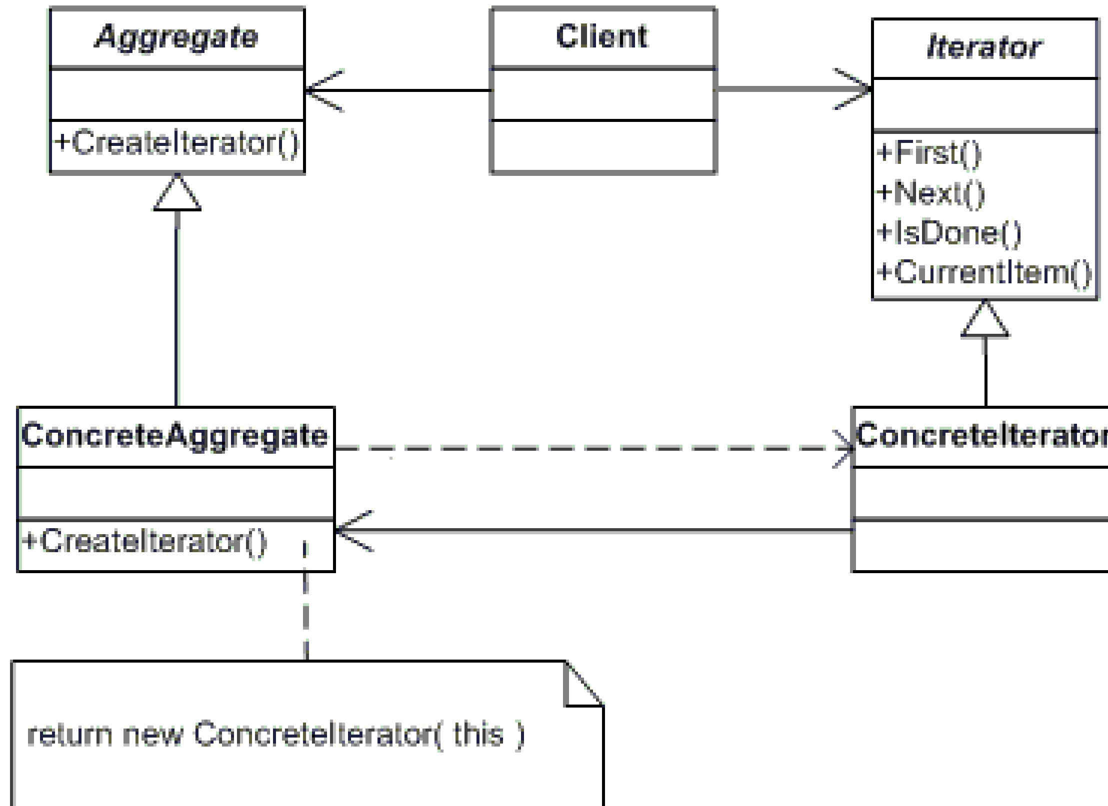
Thoughts on Command

- 4 More classes added, though all small.
- Level of Decoupling much greater.
  - The switch no longer knows anything about an appliance on the end, just that it can be turned on and off.
- Future flexibility much greater.
  - If we change a fan or a light, it does not have any side effects.

Higher Level Concepts

- Basically, command separates (or decouples) the invoker from the receiver by creating an interface in-between them.
- Allows the passing of methods as arguments. (Note that this is much easier to accomplish in C++ with function pointers)

Iterator Pattern

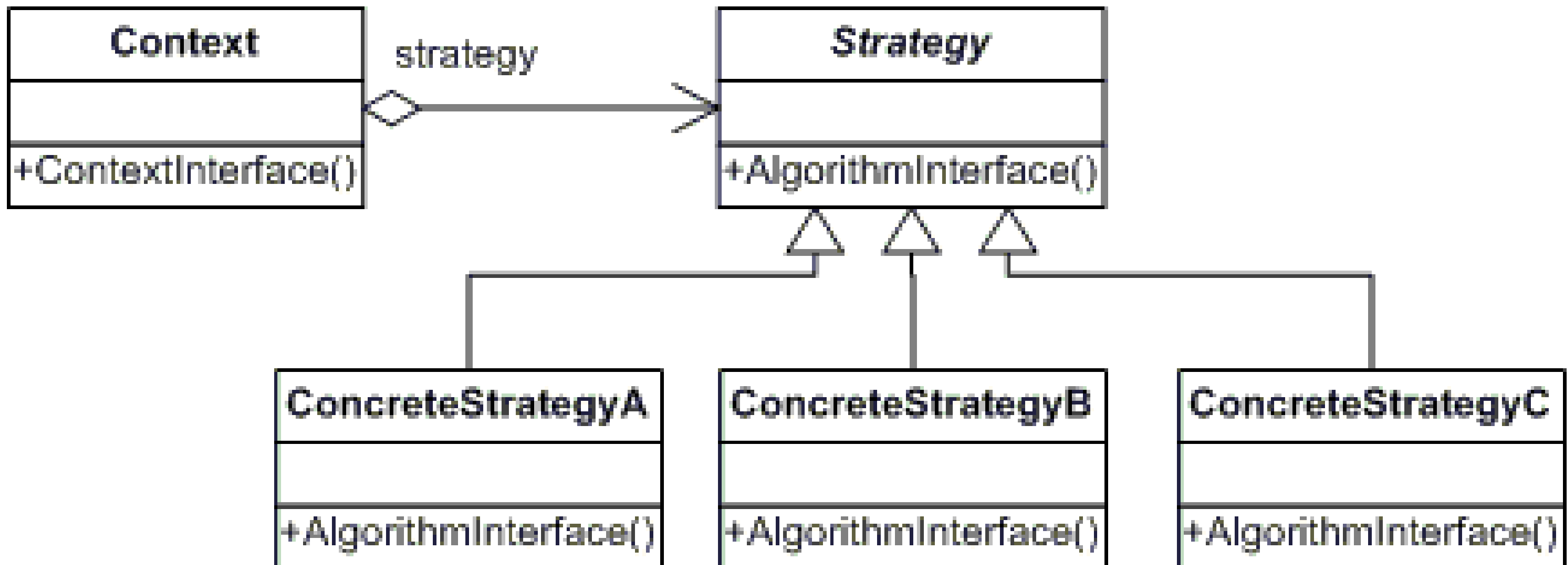


Participants

The classes and/or objects participating in this pattern are:

- **Iterator (AbstractIterator)**
  - defines an interface for accessing and traversing elements.
- **ConcreteIterator (Iterator)**
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- **Aggregate (AbstractCollection)**
  - defines an interface for creating an Iterator object
- **ConcreteAggregate (Collection)**
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Strategy Pattern



Participants

- **Strategy (SortStrategy)**
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
- **ConcreteStrategy (QuickSort, ShellSort, MergeSort)**
  - implements the algorithm using the Strategy interface
- **Context (SortedList)**
  - is configured with a ConcreteStrategy object
  - maintains a reference to a Strategy object
  - may define an interface that lets Strategy access its data.

Template Method