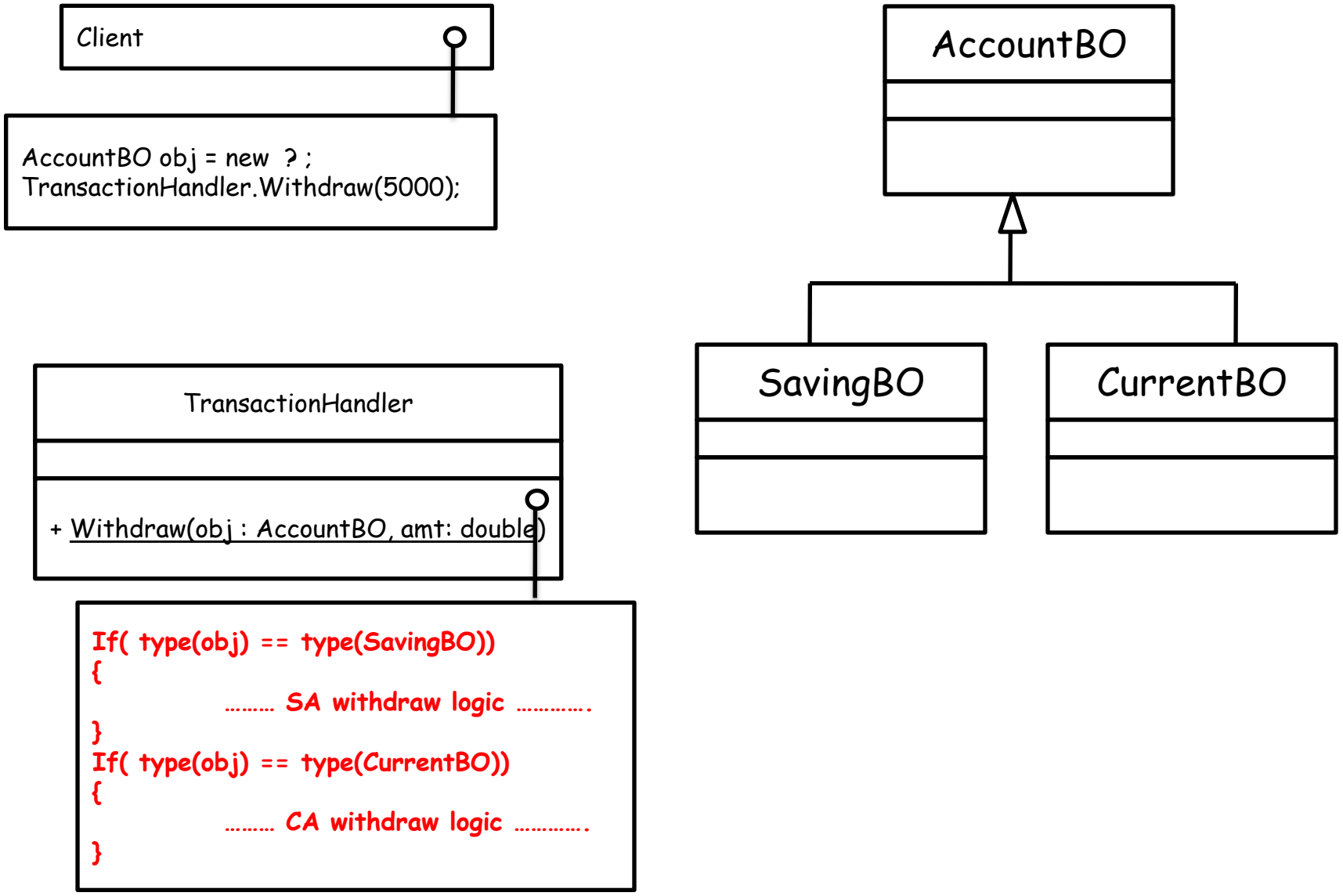
# Rule of thumb

Use design patterns only when you recognize situation which is causing you pain in future and only if there is no other simple OOP way to solve that problem

Design patterns are solutions to problems as they occur, not solutions to problems we think may occur

# Problem


Client
AccountBO obj = new ? ;
TransactionHandler.Withdraw(5000);
AccountBO
SavingBO
CurrentBO
TransactionHandler
+ Withdraw(obj : AccountBO, amt: double)
If( type(obj) == type(SavingBO))
{
…… SA withdraw logic …….
}
If( type(obj) == type(CurrentBO))
{
…… CA withdraw logic …….
}

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

# Open/Closed Principle (OCP)

- Open for extension.
  - We can extend the module with new behaviors.
- Closed for modification.
  - Extending the behavior of a module does not result in changes to the source or binary code of the module.

# Cyclomatic complexity

Cyclomatic complexity is a measure of how many paths of execution there are through the same code.

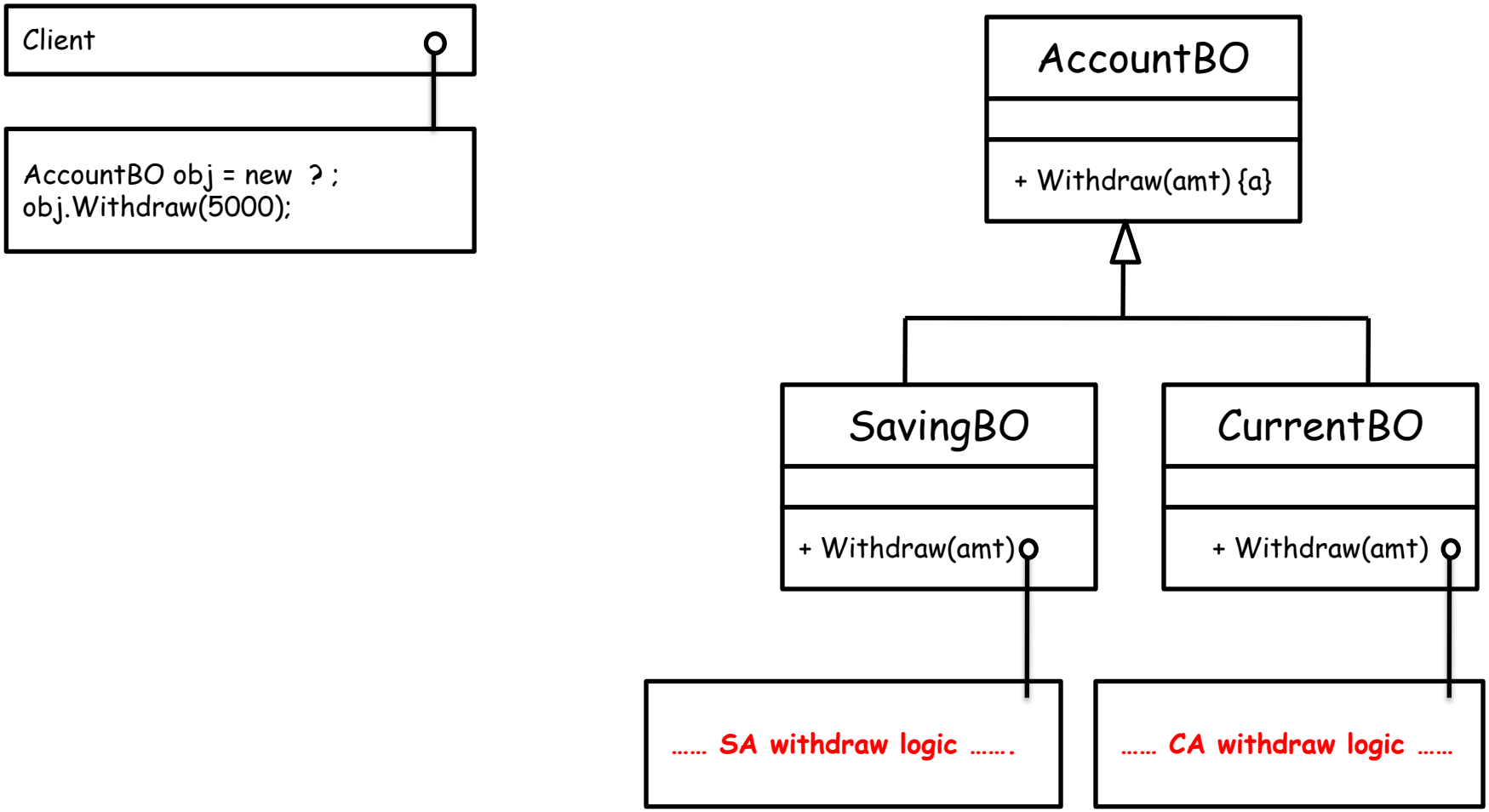
```
public void HelloWorld()
{
        Console.WriteLine("Hello World!");
}
```

Very low cyclomatic complexity. easy to achieve 100% code coverage.

```
public void HelloWorldToday()
{
    switch (DateTime.Now.DayOfWeek)
    {
        case DayOfWeek.Monday:
        Console.WriteLine("Hello Monday!");
        break;
        …..
```

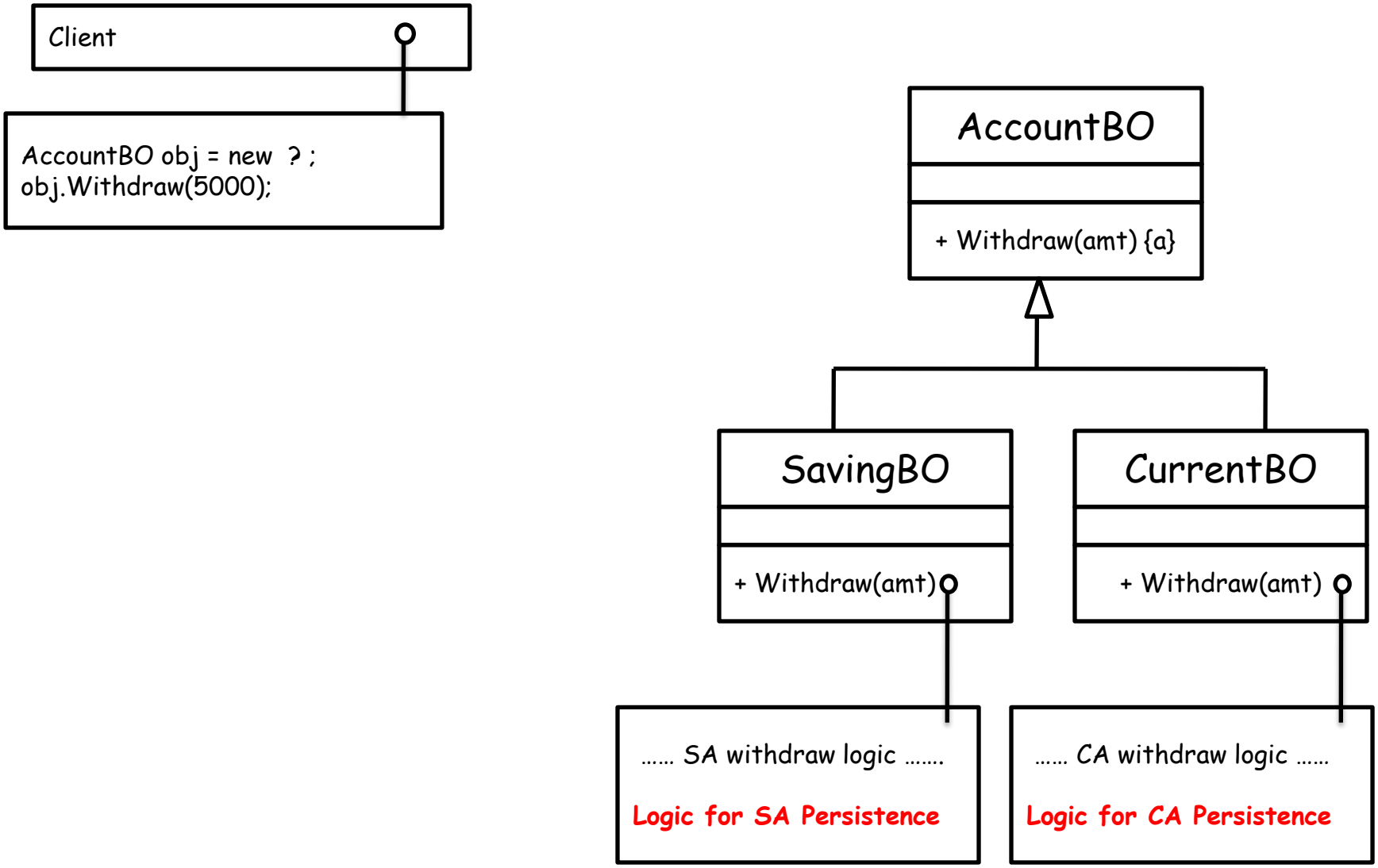
Cyclomatic complexity is much higher. you have to generate all the conditions needed

# Applying Polymorphism



Polymorphism is the characteristic of being able to change behavior depending on the object.

# Problem


Client
AccountBO obj = new ? ;
obj.Withdraw(5000);
AccountBO
+ Withdraw(amt) {a}
SavingBO
+ Withdraw(amt)
CurrentBO
+ Withdraw(amt)
…… SA withdraw logic …….
Logic for SA Persistence
…… CA withdraw logic ……
Logic for CA Persistence

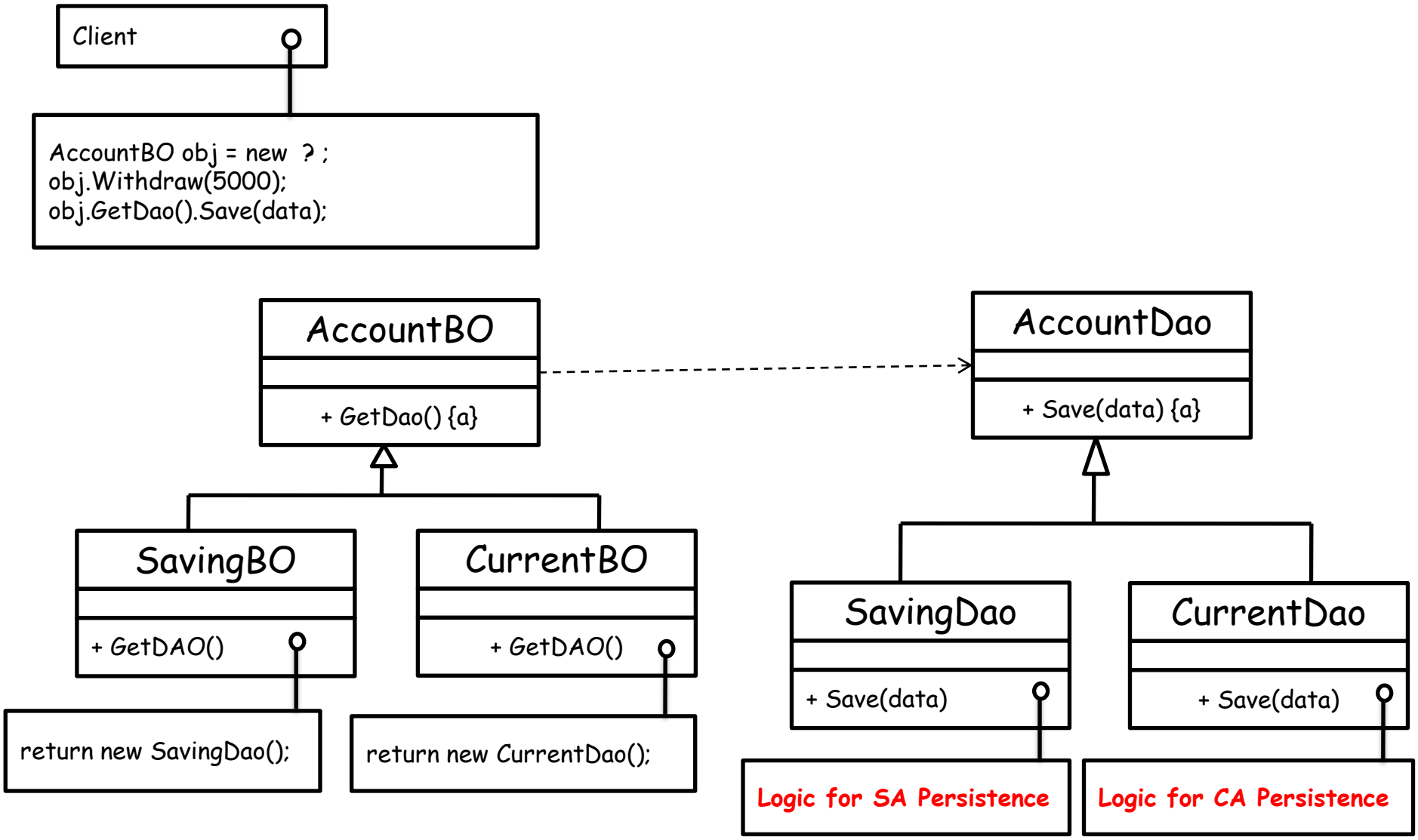
If a class assumes more than one responsibility, that class will have more than one reason to change.

Cohesion is a measure of how strongly related and focused the responsibilities of a class are.

How to achieve High Cohesion?

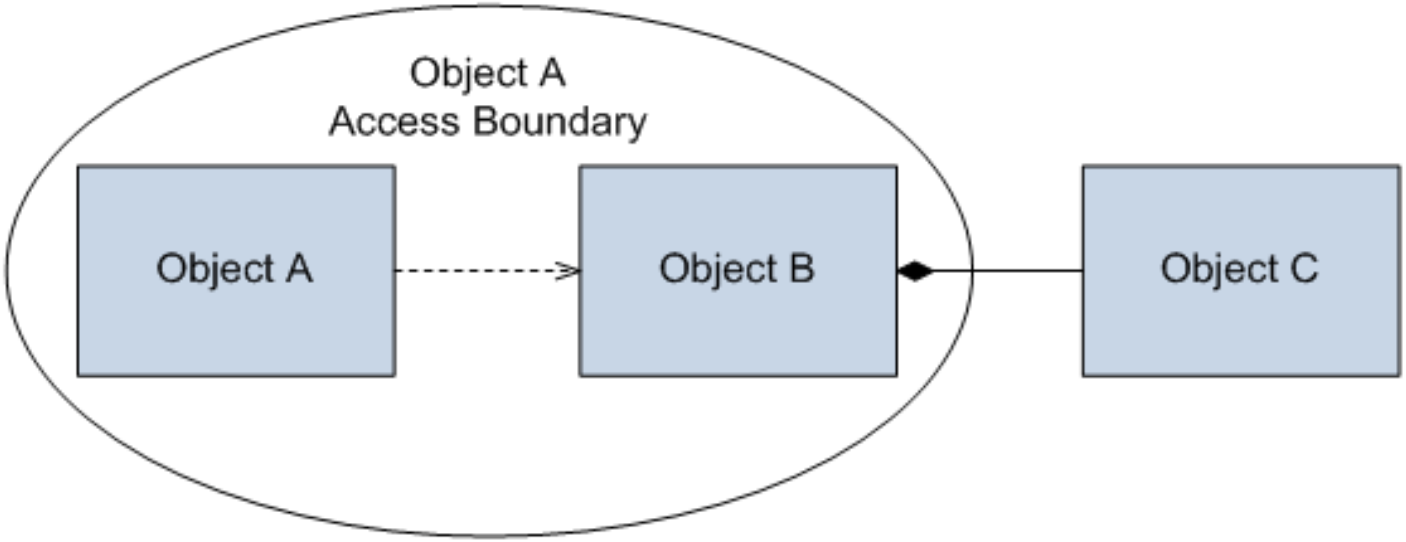
Follow the Single-Responsibility Principle

# Applying SRP



If the application is not changing in ways that cause the two responsibilities to change at different times, there is no need to separate them.

# Don’t Talk to Stranger

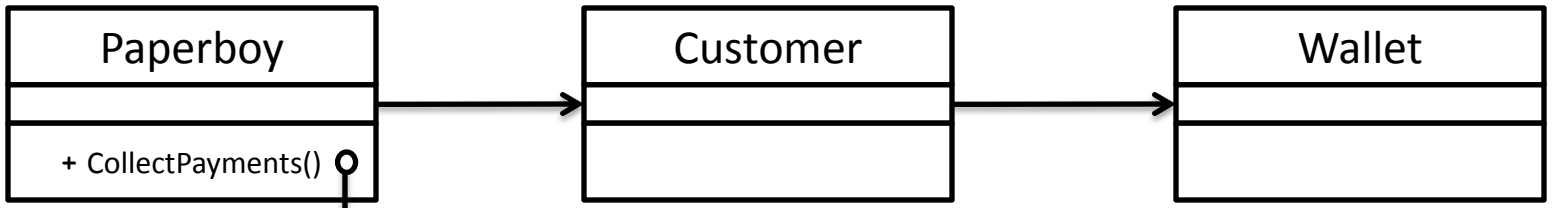



Object A has a dependency on Object B which composes Object C. Object A is permitted to invoke methods on Object B, but is not permitted to invoke methods on Object C.

# Law of Demeter

1. Your method can call other methods in its class directly
2. Your method can call methods on its own fields directly (but not on the fields' fields)
3. When your method takes parameters, your method can call methods on those parameters directly.
4. When your method creates local objects, that method can call methods on the local objects.
5. One should not call methods on a global object (but it can be passed as a parameter ?)
6. One should not have a chain of messages a.getB().getC().doSomething() in some class other than a's class.

# Law of Demeter

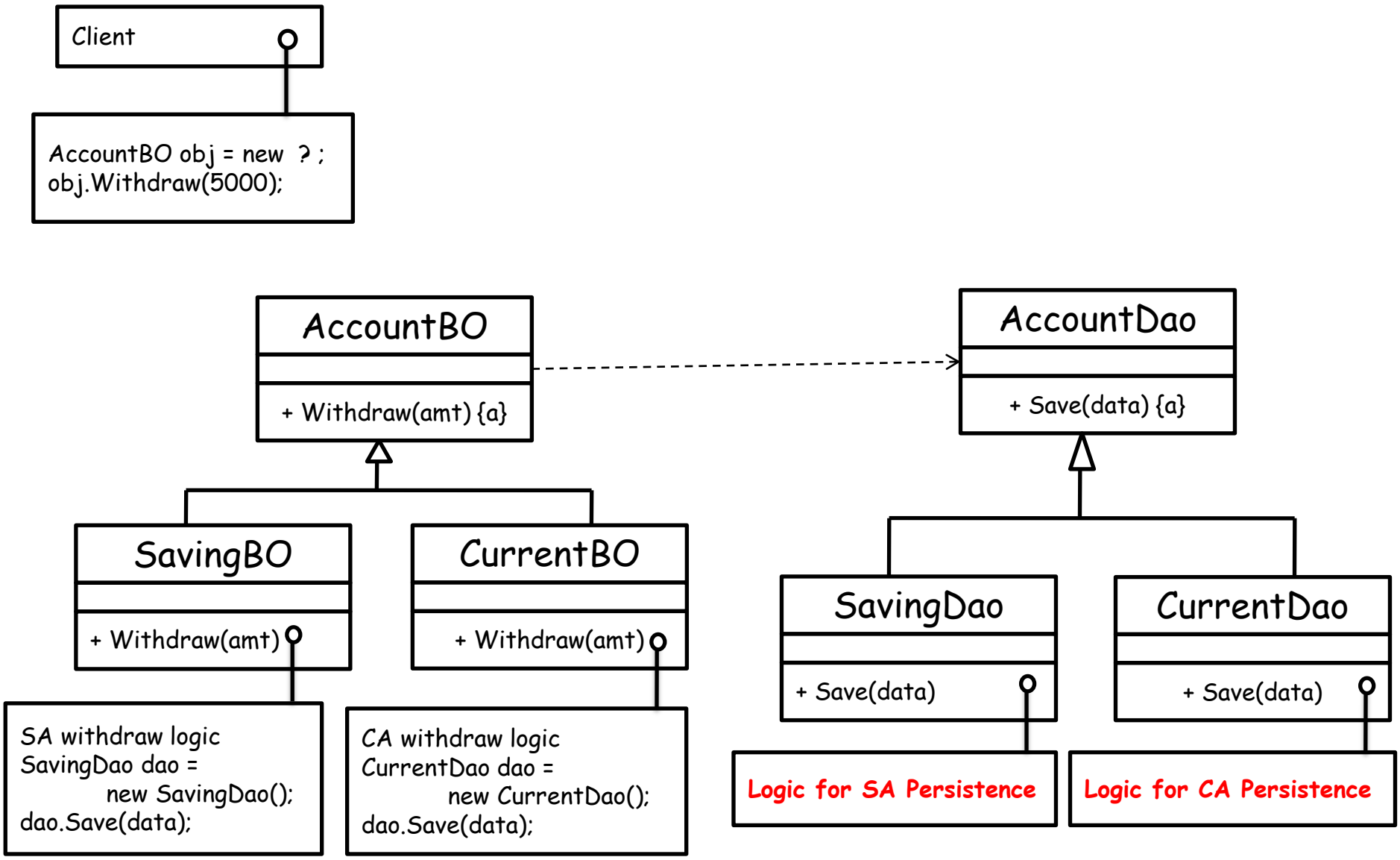



```
decimal payment = 2m;
foreach (Customer customer in Customers)
{
        if (customer.Wallet.Money >= payment)
        {
                customer.Wallet.Money -= payment;
                fundsCollected += payment;
        }
}
```

“Tell Don’t Ask” principle encourages you to tell an object to do something rather than rip data out of an object to do it in client code.

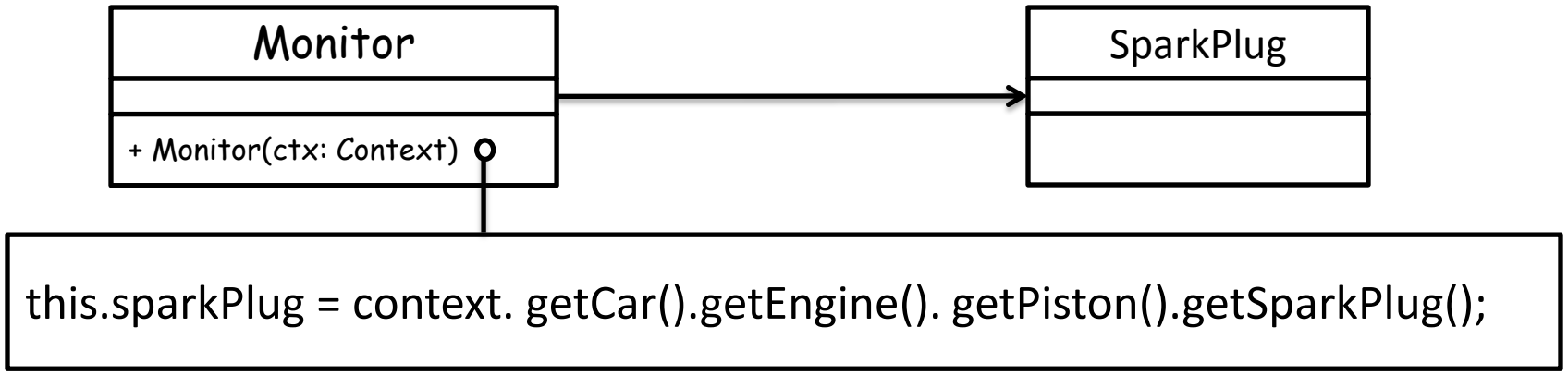
objectA tells objectB to do something, rather than asking objectB about its state so that objectA can make adecision.

# Applying LoD



The Law of Demeter, also know as the Principle of Least Knowledge

# Unit Testing Problem

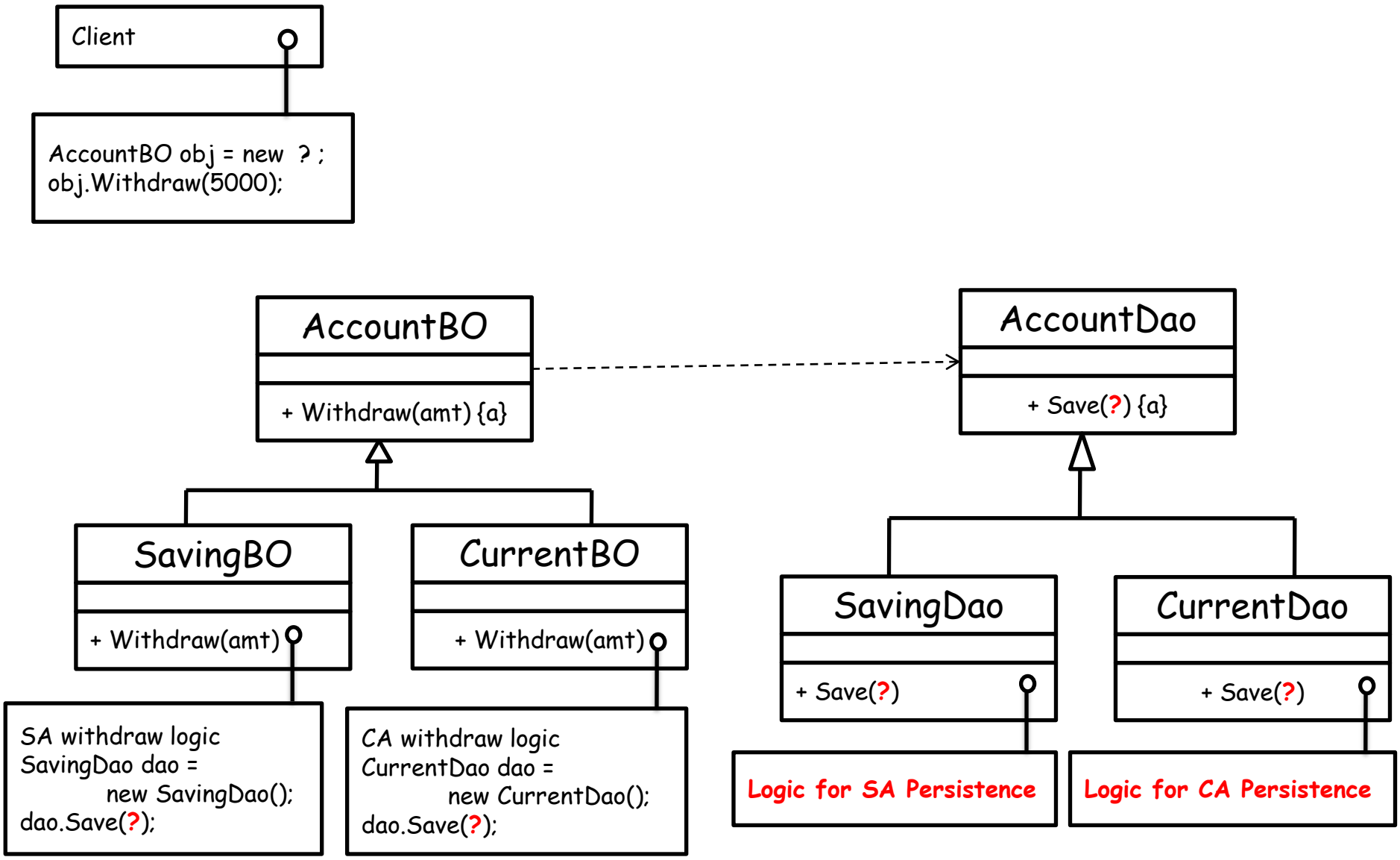


The Law of Demeter helps in designing your method calls to permit easy mocking

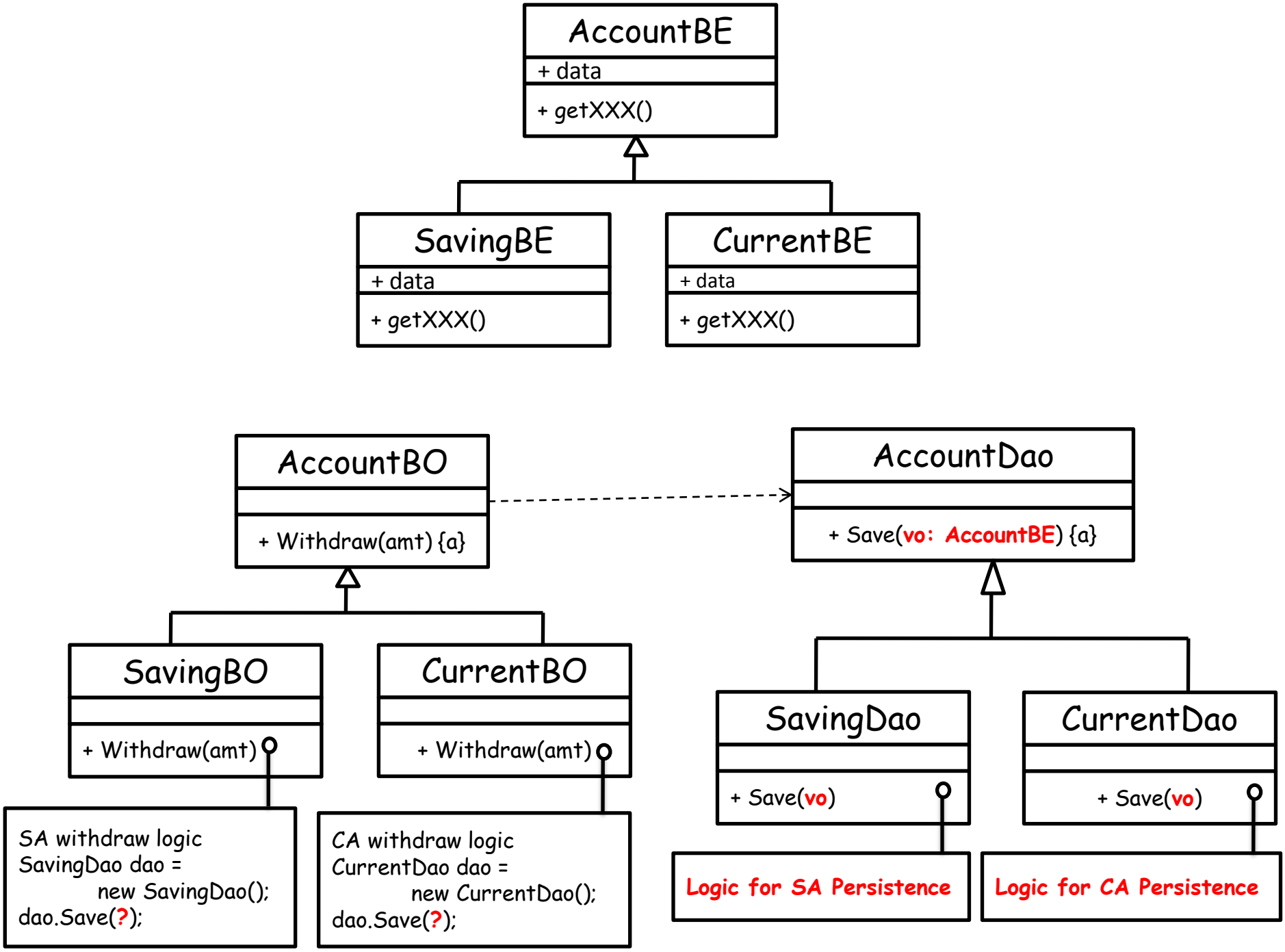
Law of Demeter violation is like a haystack where the code is desperately trying to locate the needle.

*no principle is a law, and all principles should be used when and where they are helpful.*

# Problem

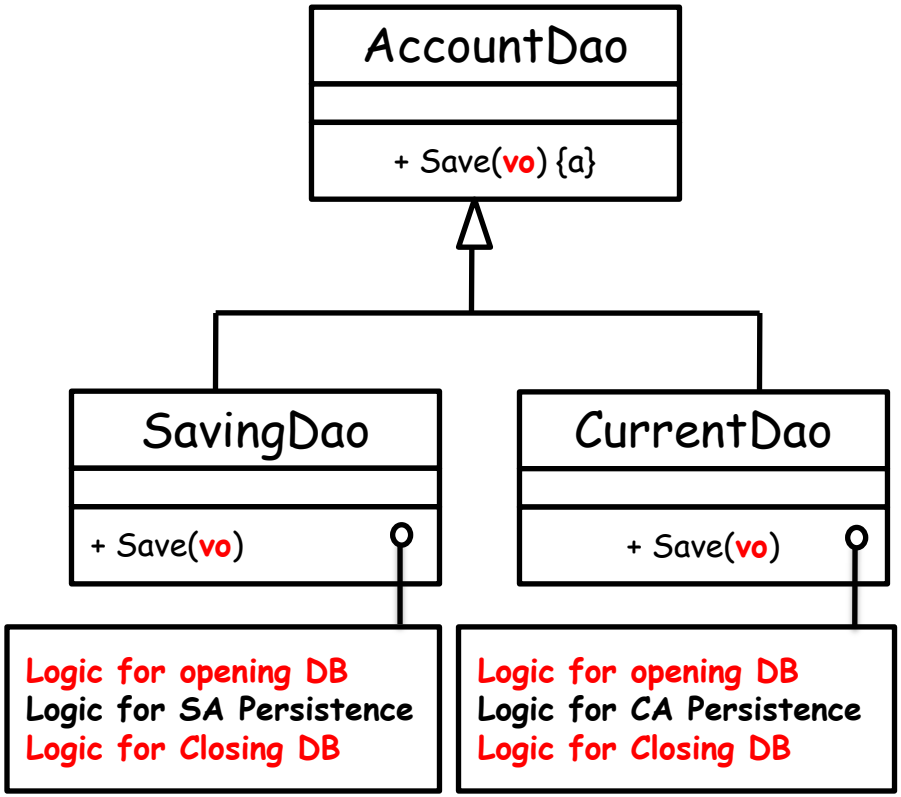

Client
AccountBO obj = new ? ;
obj.Withdraw(5000);
AccountBO
+ Withdraw(amt) {a}
AccountDao
+ Save(?) {a}
SavingBO
+ Withdraw(amt)
CurrentBO
+ Withdraw(amt)
SavingDao
+ Save(?)
CurrentDao
+ Save(?)
SA withdraw logic
SavingDao dao =
new SavingDao();
dao.Save(?);
CA withdraw logic
CurrentDao dao =
new CurrentDao();
dao.Save(?);
Logic for SA Persistence
Logic for CA Persistence

# Value Object Pattern (VO)


AccountBE
+ data
+ getXXX()
SavingBE
+ data
+ getXXX()
CurrentBE
+ data
+ getXXX()
AccountBO
+ Withdraw(amt) {a}
AccountDao
+ Save(vo: AccountBE) {a}
SavingBO
+ Withdraw(amt)
CurrentBO
+ Withdraw(amt)
SavingDao
+ Save(vo)
CurrentDao
+ Save(vo)
SA withdraw logic
SavingDao dao =
new SavingDao();
dao.Save(?);
CA withdraw logic
CurrentDao dao =
new CurrentDao();
dao.Save(?);
Logic for SA Persistence
Logic for CA Persistence

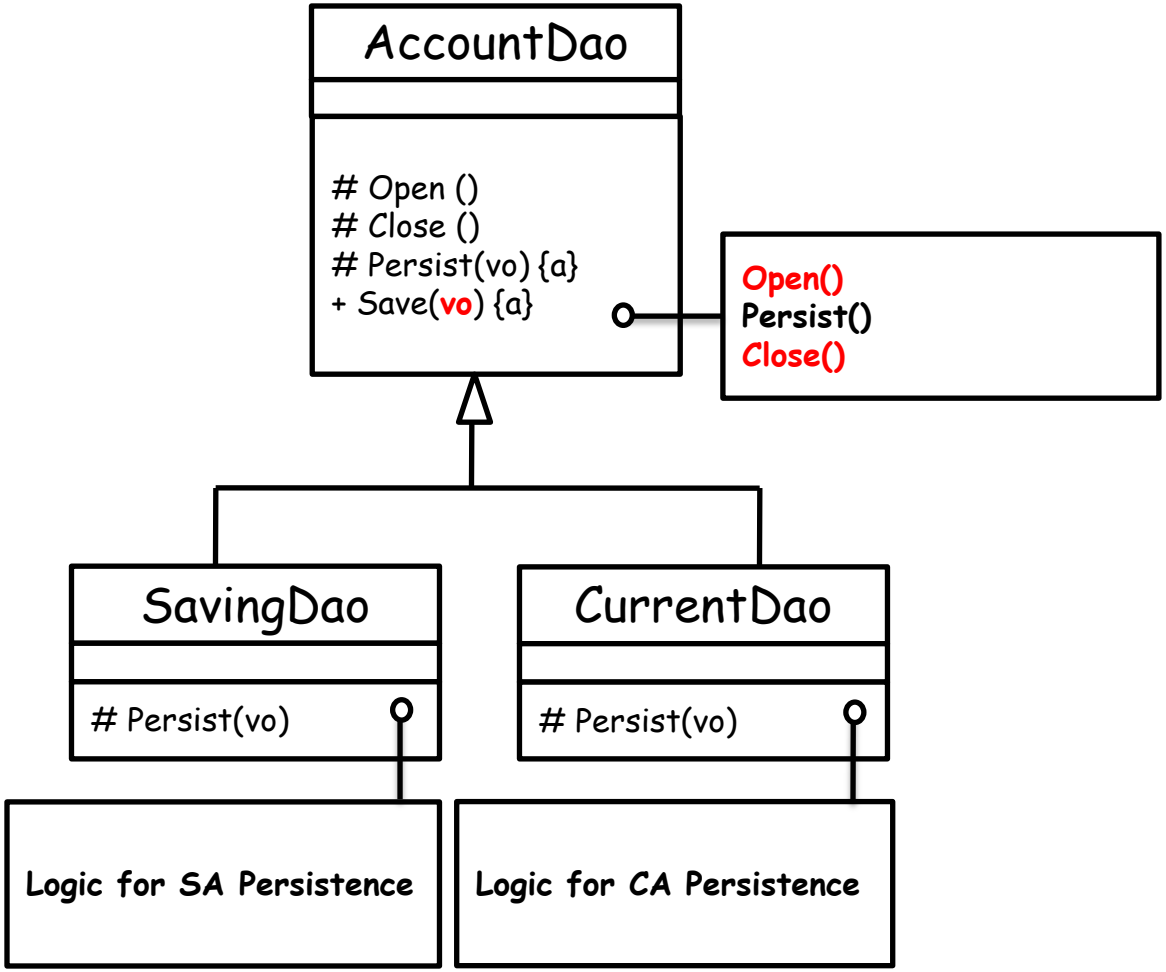
Data transfer object, formerly known as value objects, is used to transfer data between subsystems.

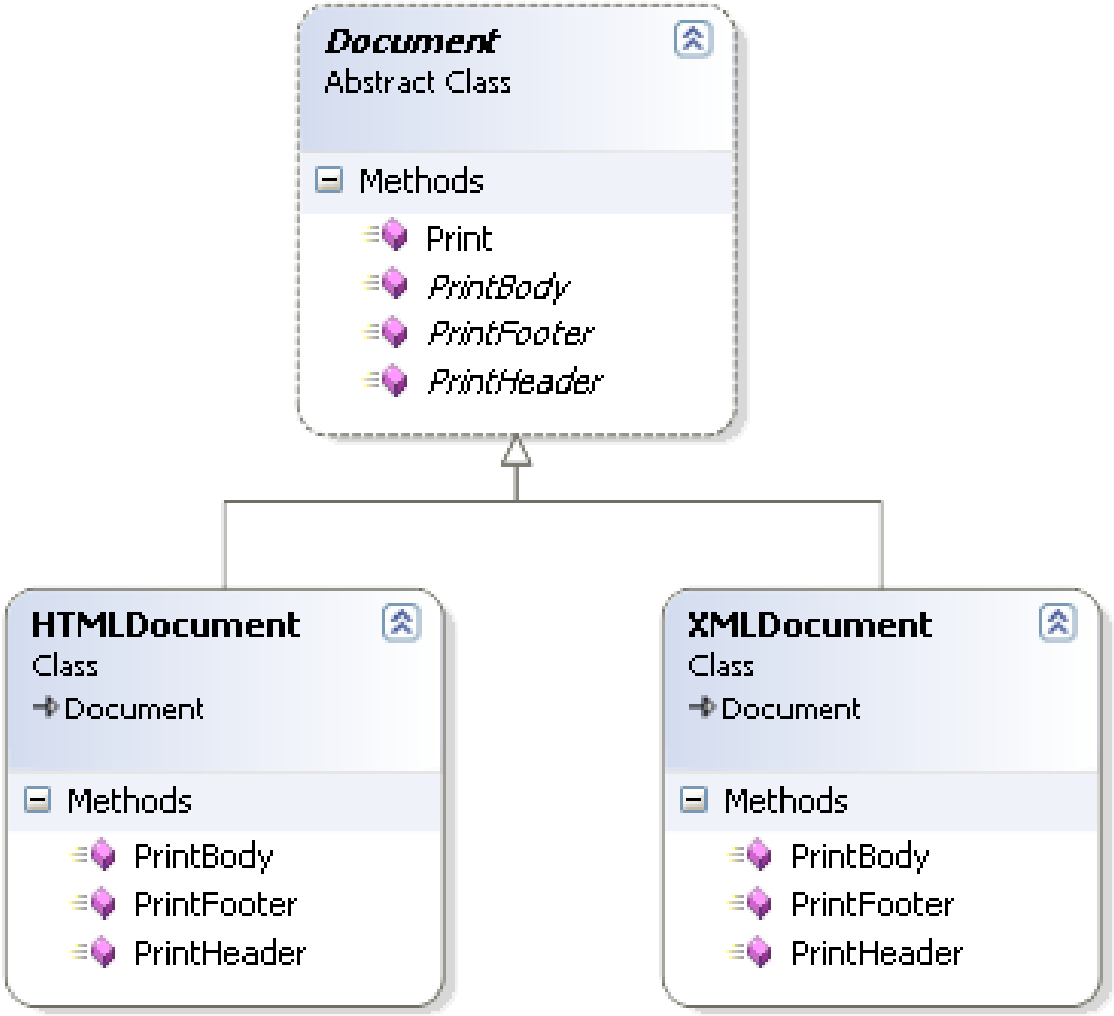
# Problem


AccountDao
+ Save(vo) {a}
SavingDao
+ Save(vo)
CurrentDao
+ Save(vo)
Logic for opening DB
Logic for SA Persistence
Logic for Closing DB
Logic for opening DB
Logic for CA Persistence
Logic for Closing DB

Don't Repeat Yourself (DRY) / Duplication is Evil (DIE) / Single Source of Truth is a principle aimed at reducing repetition of information of all kinds.

# Applying Template Method



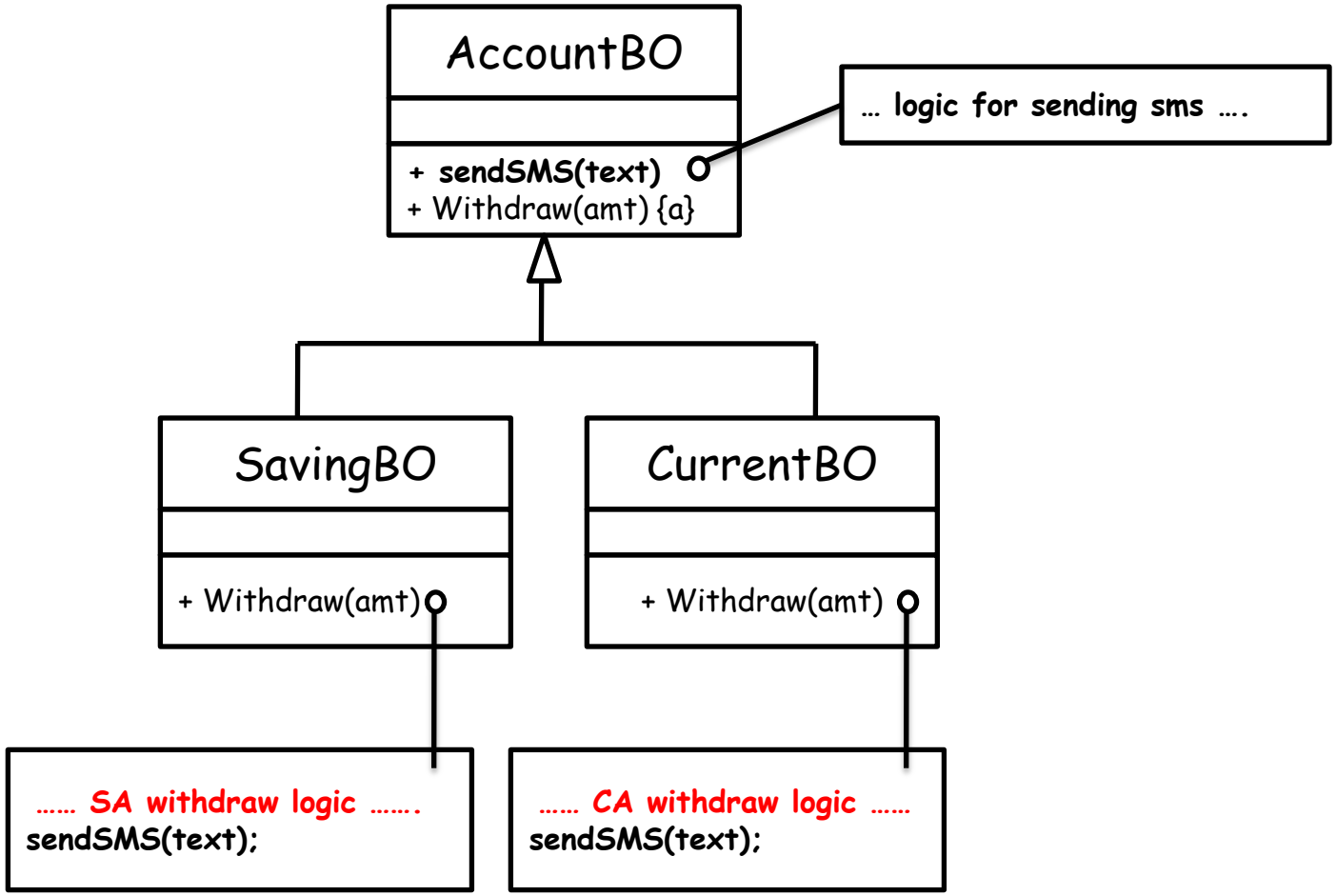

Document
Abstract Class
Methods
Print
PrintBody
PrintFooter
PrintHeader
HTMLDocument
Class
Document
Methods
PrintBody
PrintFooter
PrintHeader
XMLDocument
Class
Document
Methods
PrintBody
PrintFooter
PrintHeader

Template method defines skeleton of an algorithm. One or more steps can be overridden to allow differing behaviors while ensuring that the algorithm is still followed.

The Hollywood principle

"Don't call us, we'll call you“

# Problem

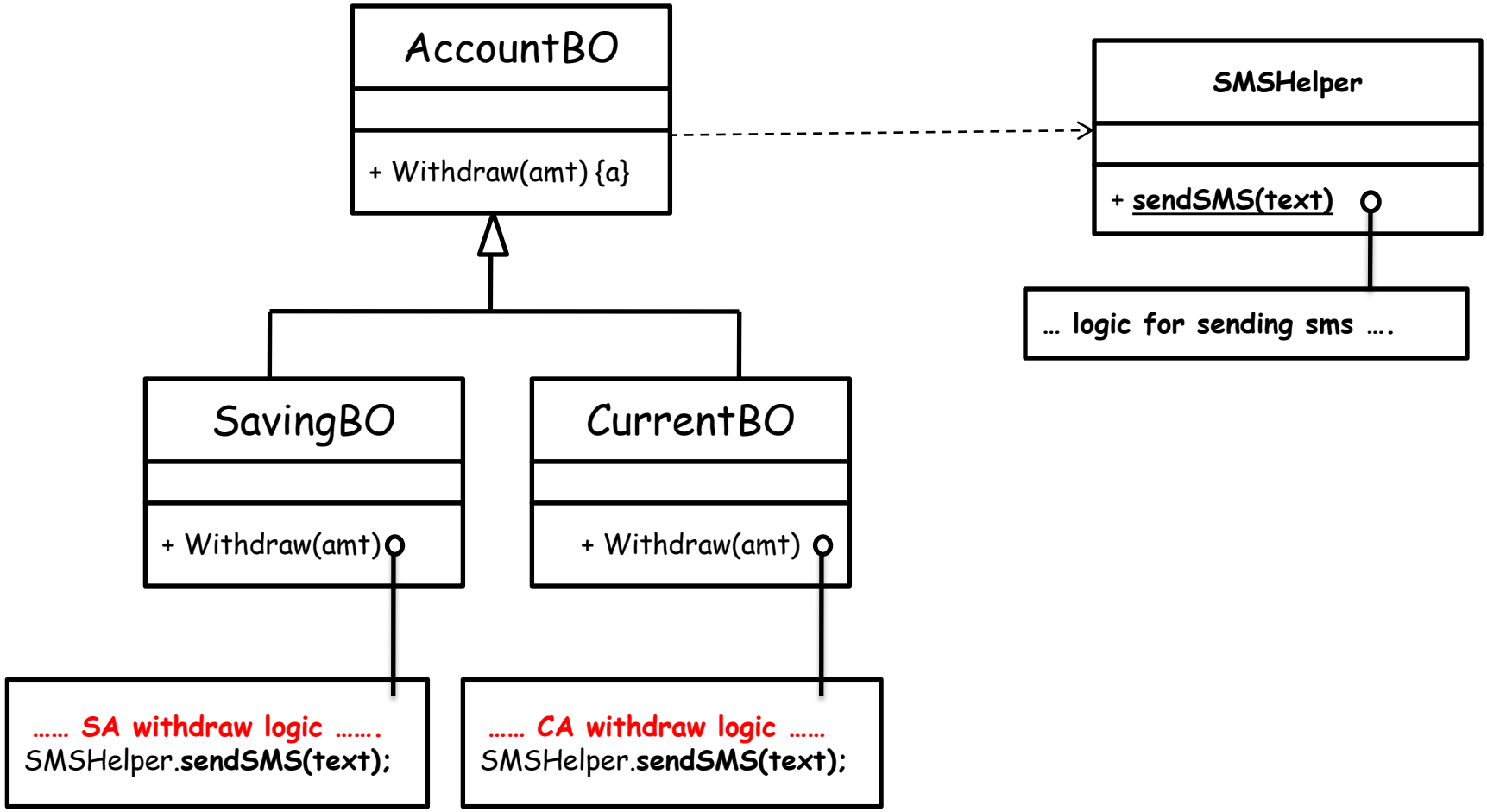

AccountBO
+ sendSMS(text)
+ Withdraw(amt) {a}
… logic for sending sms ….
SavingBO
+ Withdraw(amt)
CurrentBO
+ Withdraw(amt)
…… SA withdraw logic …….
sendSMS(text);
…… CA withdraw logic ……
sendSMS(text);

# Pure Fabrication

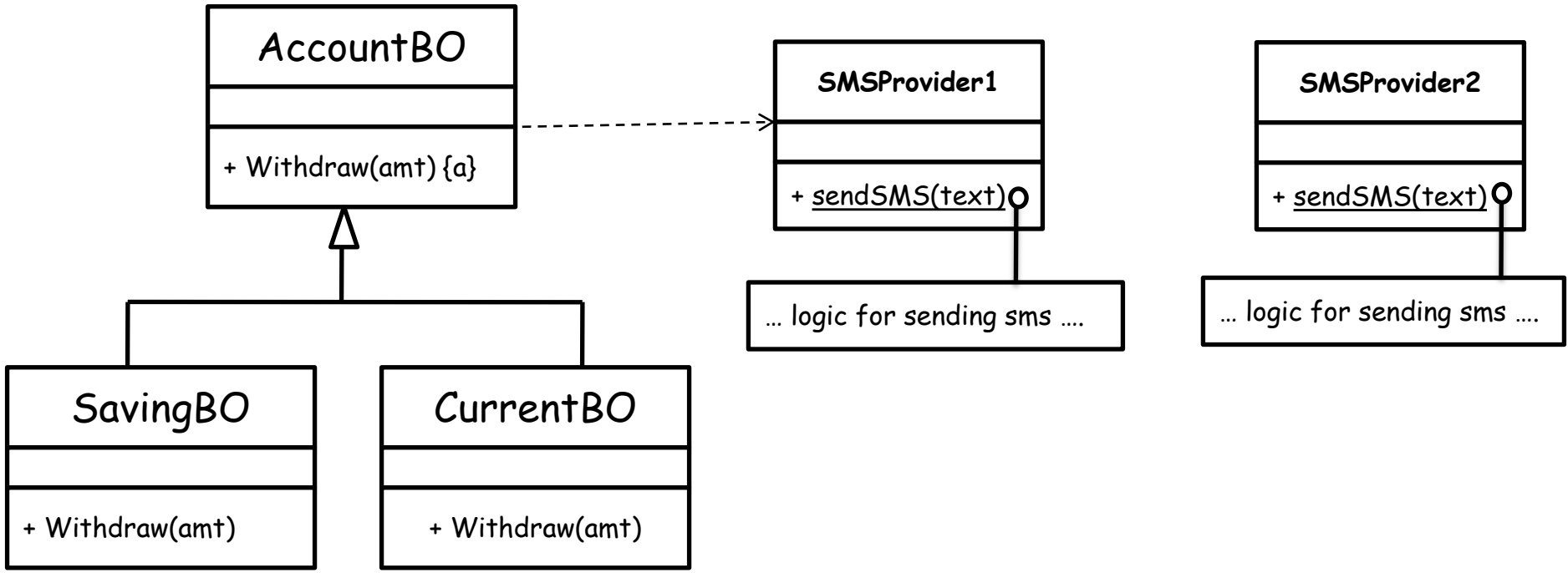
- Who is responsible when you are **desperate**, and do not want to violate high cohesion and low coupling?

- Assign a highly cohesive set of responsibilities to an **artificial** class that does not represent a problem domain concept - **something made up**, in order to support high cohesion, low coupling, and reuse.

# Pure Fabrication



# Problem

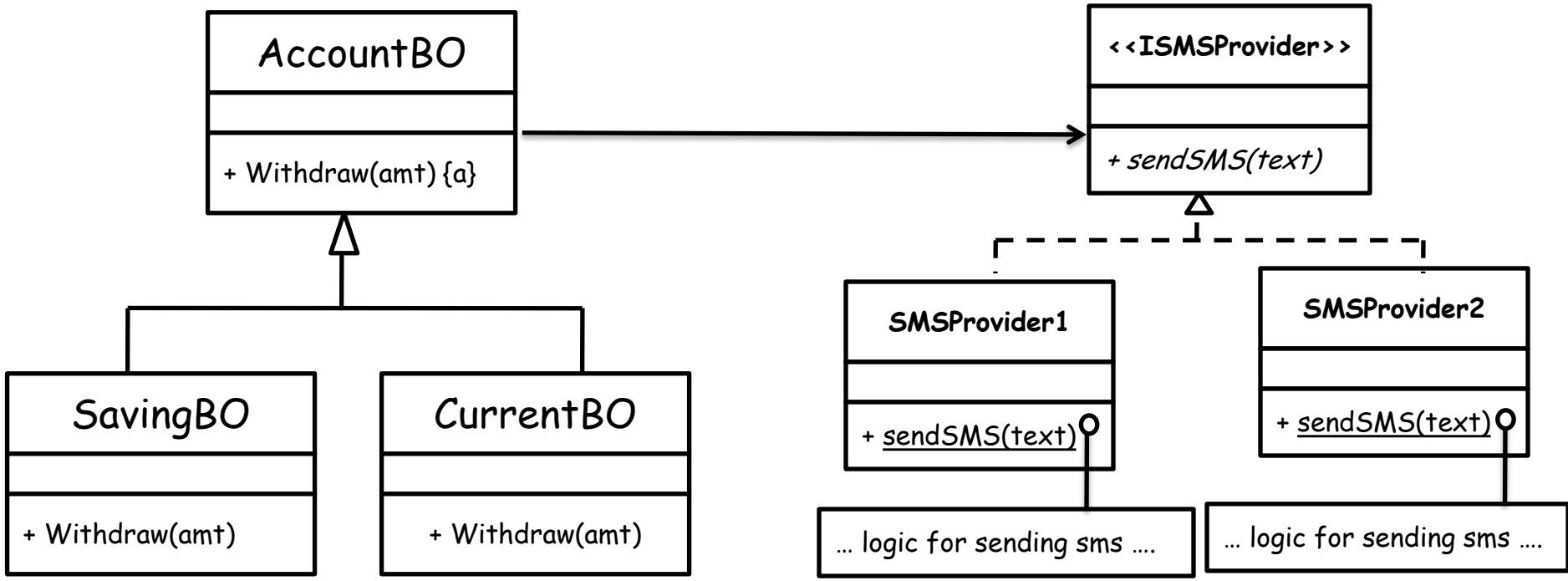

AccountBO
+ Withdraw(amt) {a}
SMSProvider1
+ sendSMS(text)
… logic for sending sms ….
SMSProvider2
+ sendSMS(text)
… logic for sending sms ….
SavingBO
+ Withdraw(amt)
CurrentBO
+ Withdraw(amt)

# Low Coupling

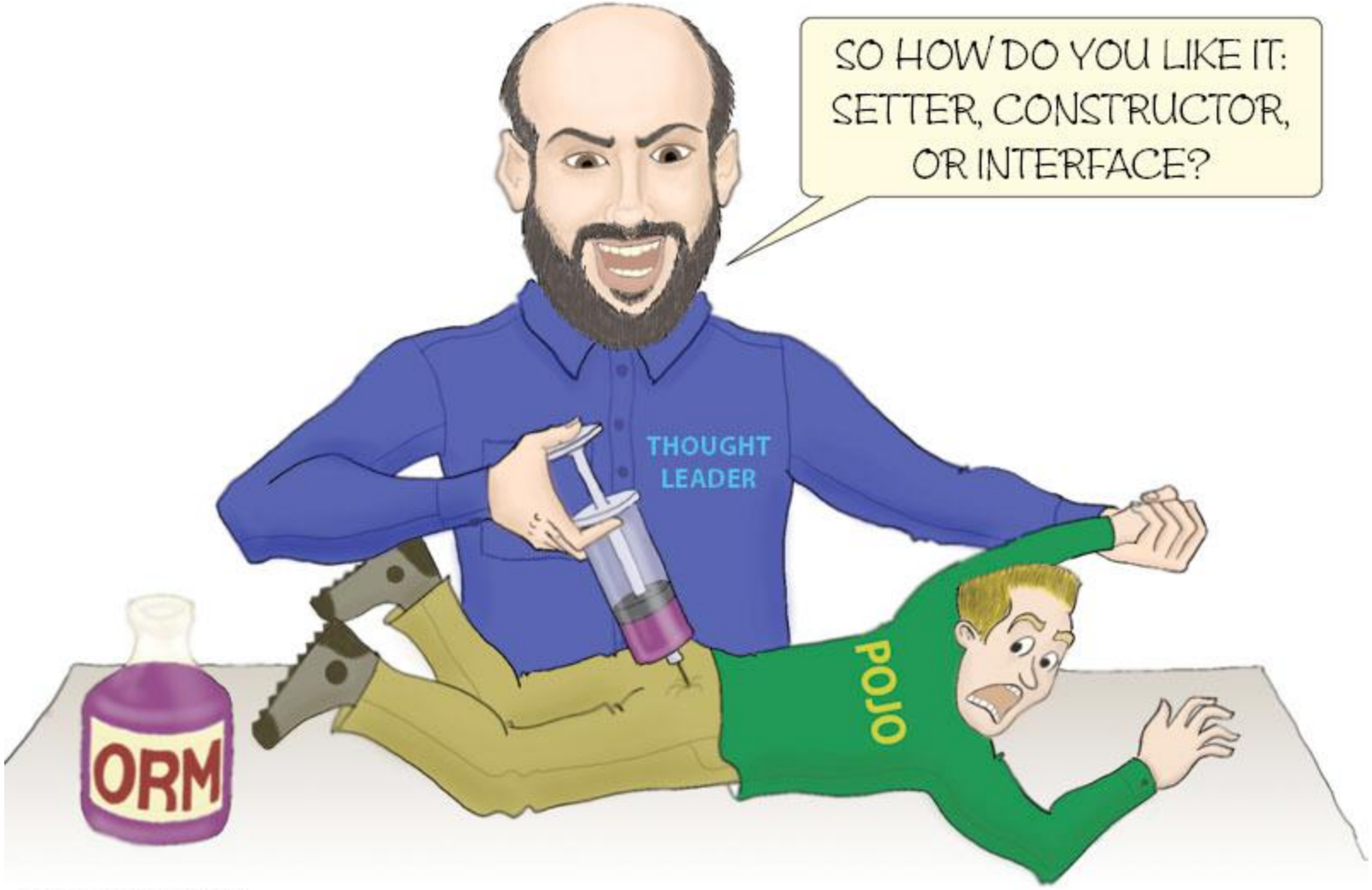
How to support low dependency and increased reuse?

Remove coupling between the two classes.

# Applying Interface Pattern



# Dependency Injection in Action



# Dependency Injection

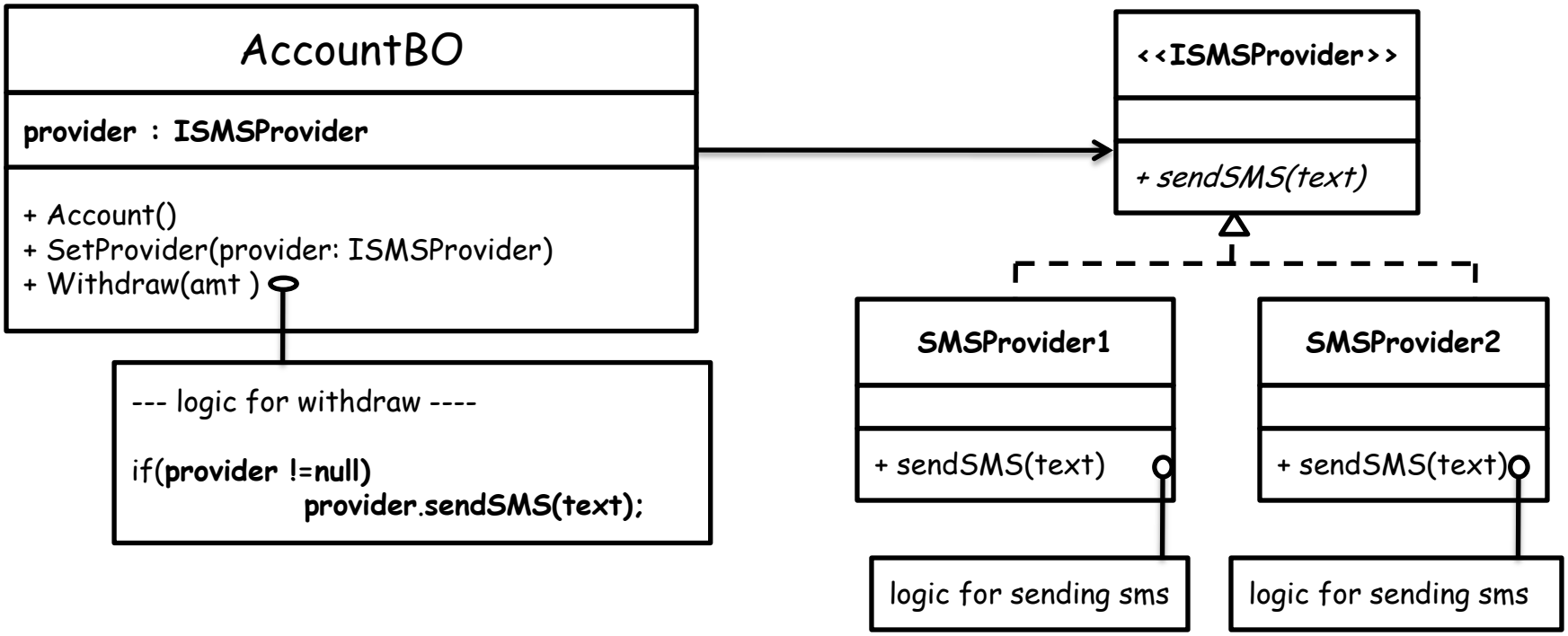
# Concrete Class Dependency

```
public class WithoutIoC
{
    private IDoSomething somethingDoer;

    public WithoutIoC()
    {
        somethingDoer = new SomethingSpecificDoer();
    }

    public void DoMyThing()
    {
        Console.WriteLine("In the beginnig was me.");
        somethingDoer.Something();
        Console.WriteLine("After my doer did it");
    }
}
```

# Allow dependency to be passed in

```
public class WithIoC
{
    private IDoSomething somethingDoer;

    public WithIoC(IDoSomething somethingDoer)
    {
        this.somethingDoer = somethingDoer;
    }

    public void DoMyThing()
    {
        Console.WriteLine("In the beginnig was me.");
        somethingDoer.Something();
        Console.WriteLine("After my doer did it");
    }
}
```

# Applying DI

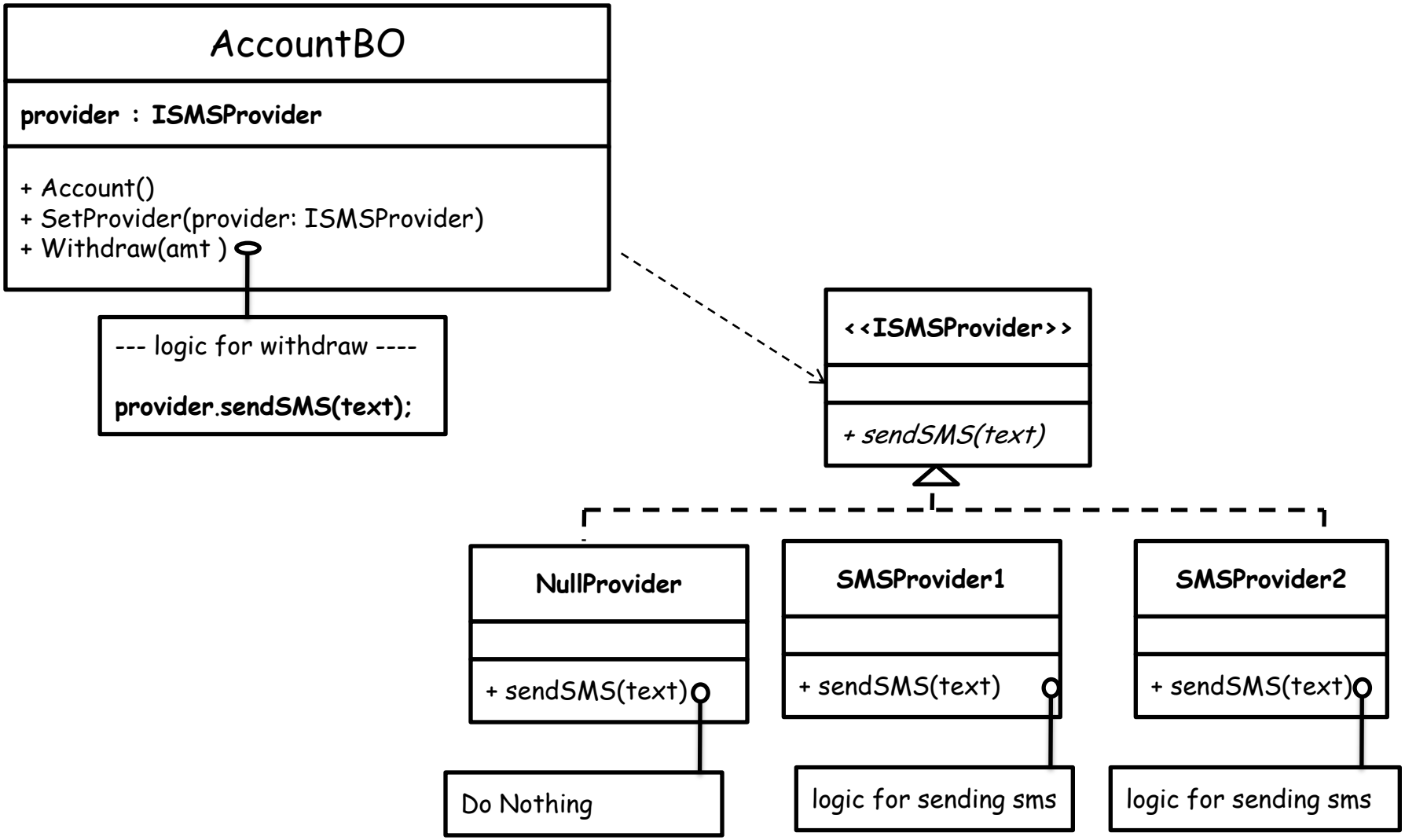

AccountBO
provider : ISMSProvider
+ Account()
+ SetProvider(provider: ISMSProvider)
+ Withdraw(amt )
--- logic for withdraw ----
if(provider !=null)
provider.sendSMS(text);
<<ISMSProvider>>
+ sendSMS(text)
SMSProvider1
+ sendSMS(text)
logic for sending sms
SMSProvider2
+ sendSMS(text)
logic for sending sms

Strategy Pattern Enables you to use different business rules or algorithms depending on the context in which they occur.

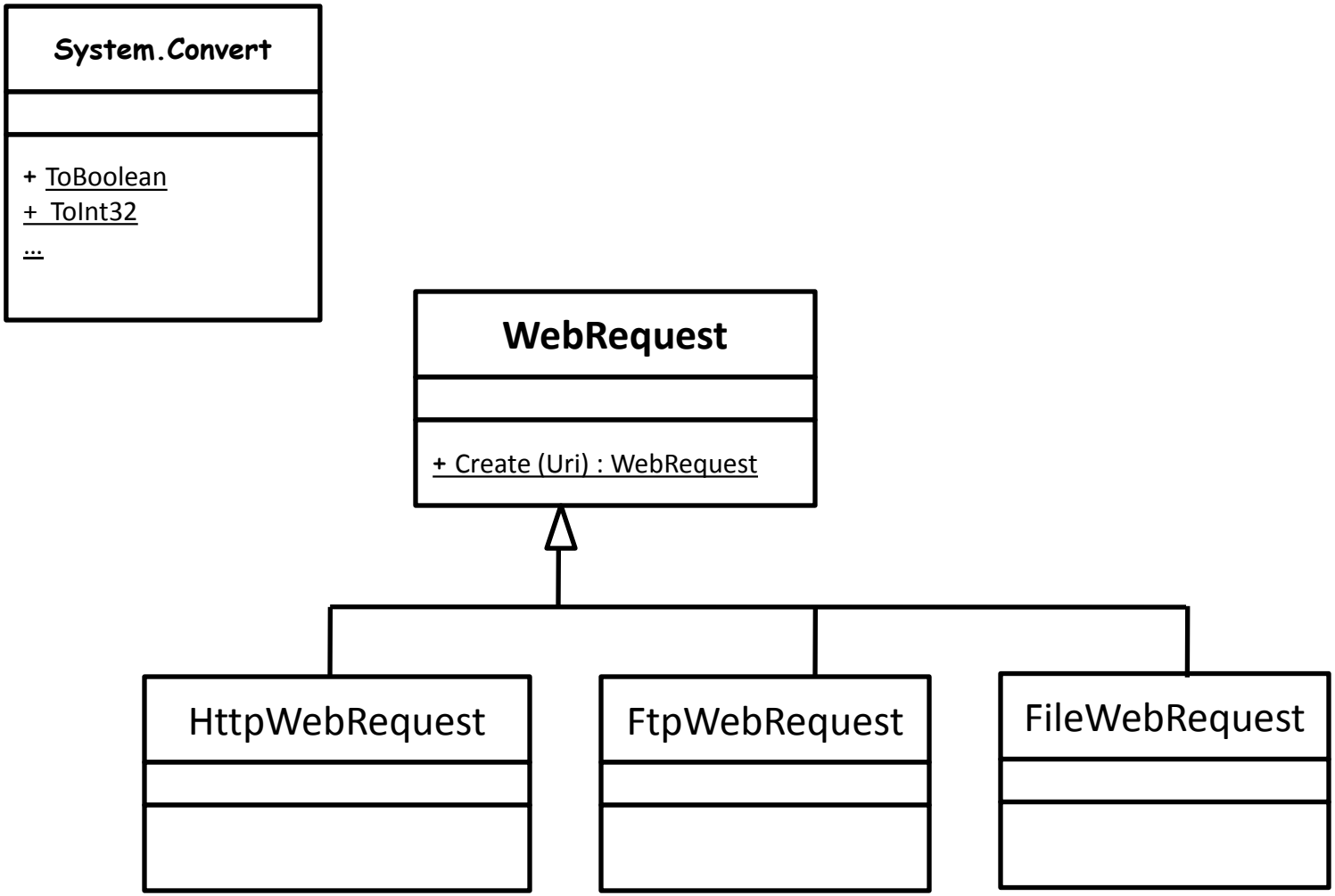
# Template v/s Strategy

# Strategy in .Net

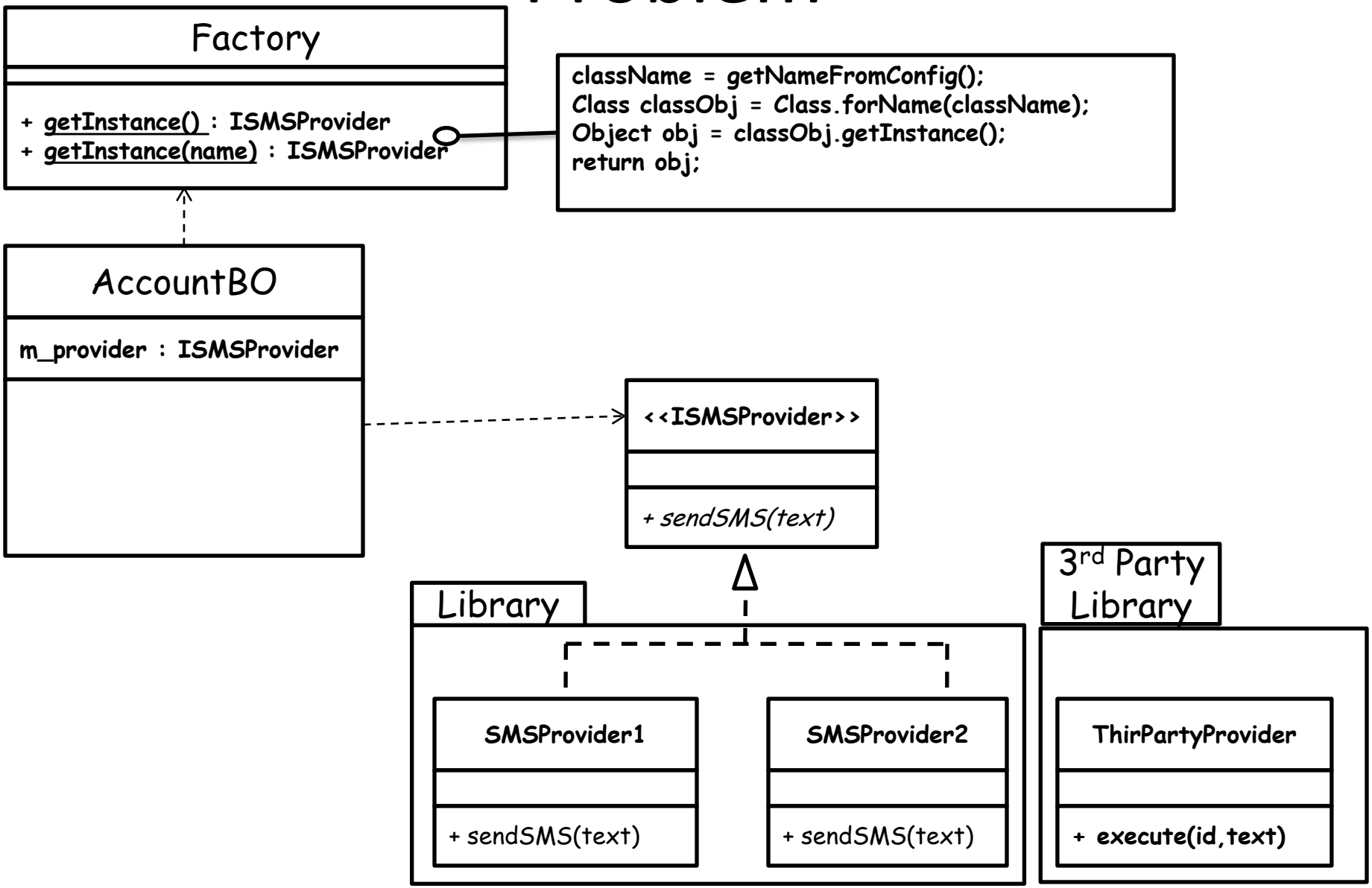
# Applying Null Object



# Factory Method in .Net



# Problem

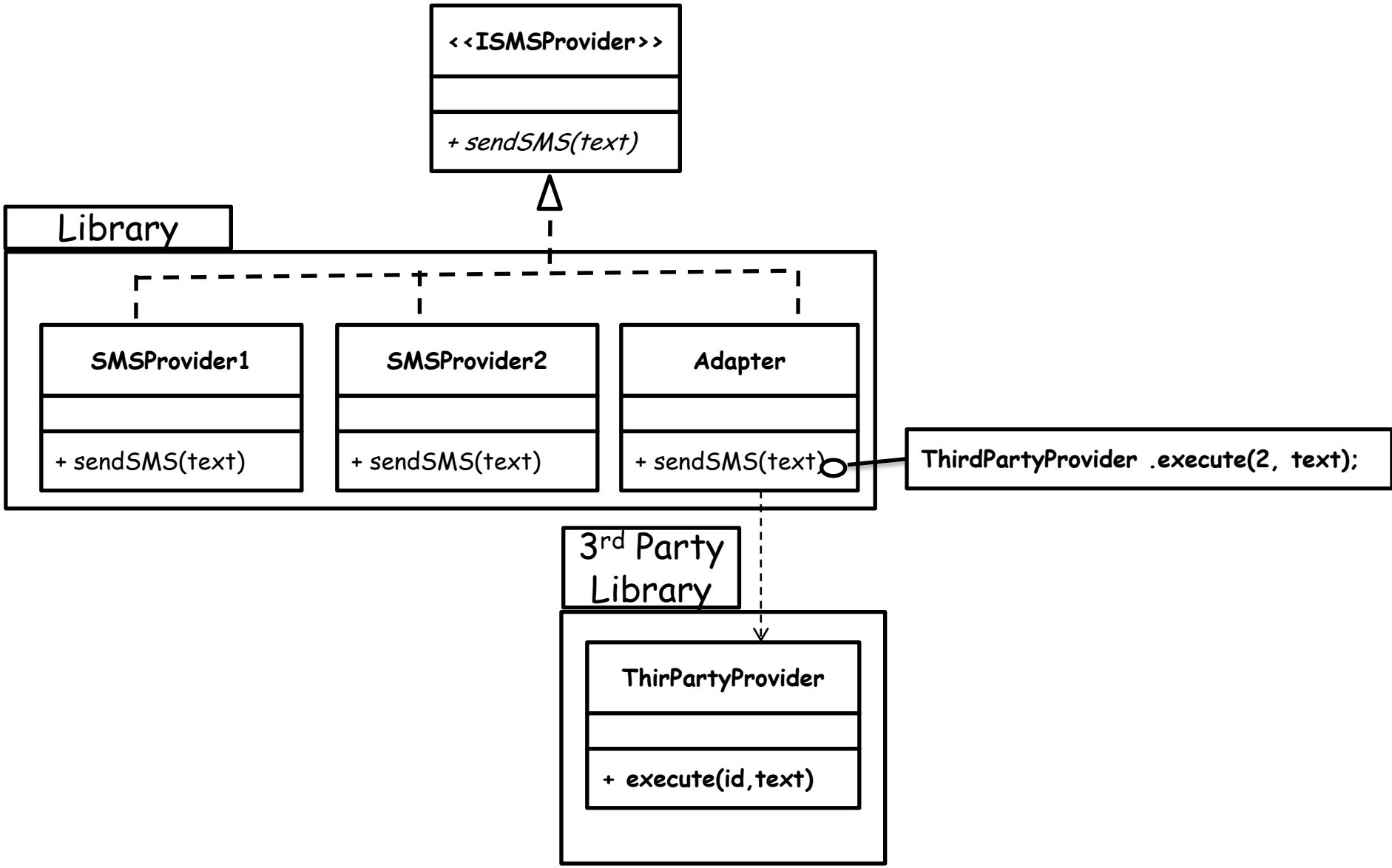

Factory
+ getInstance() : ISMSProvider
+ getInstance(name) : ISMSProvider
className = getNameFromConfig();
Class classObj = Class.forName(className);
Object obj = classObj.getInstance();
return obj;
AccountBO
m_provider : ISMSProvider
<<ISMSProvider>>
+ sendSMS(text)
Library
SMSProvider1
+ sendSMS(text)
SMSProvider2
+ sendSMS(text)
3rd Party Library
ThirPartyProvider
+ execute(id,text)

# Indirection

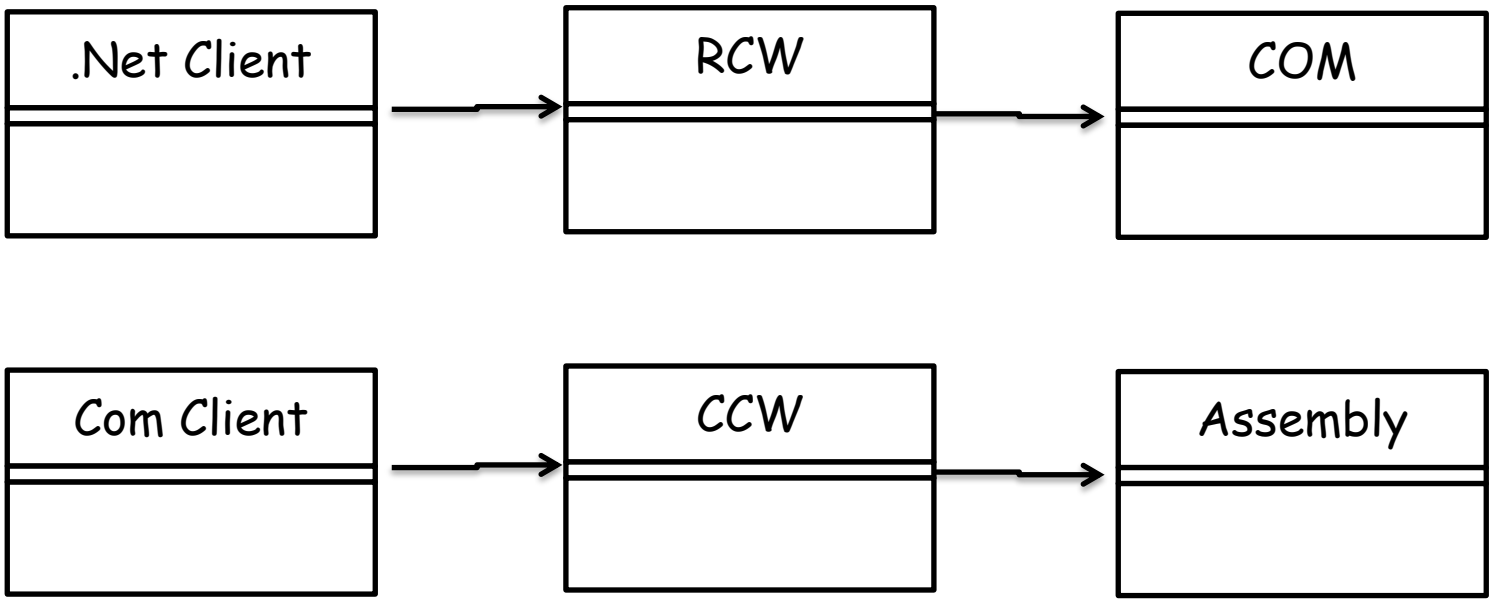
- Problem:  How do we assign responsibility to avoid direct coupling between two or more classes?

- Solution:  Assign responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.

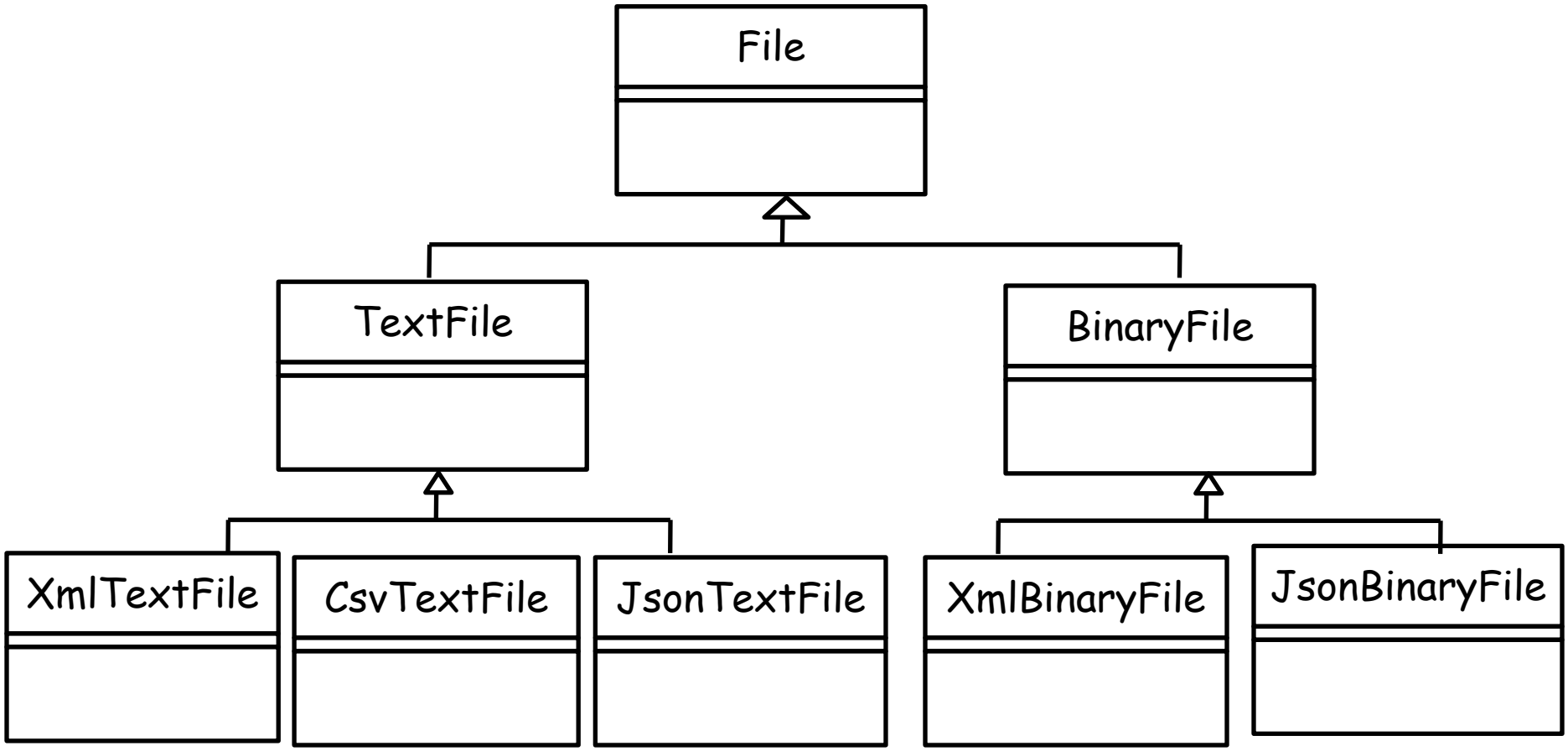
# Applying Adapter



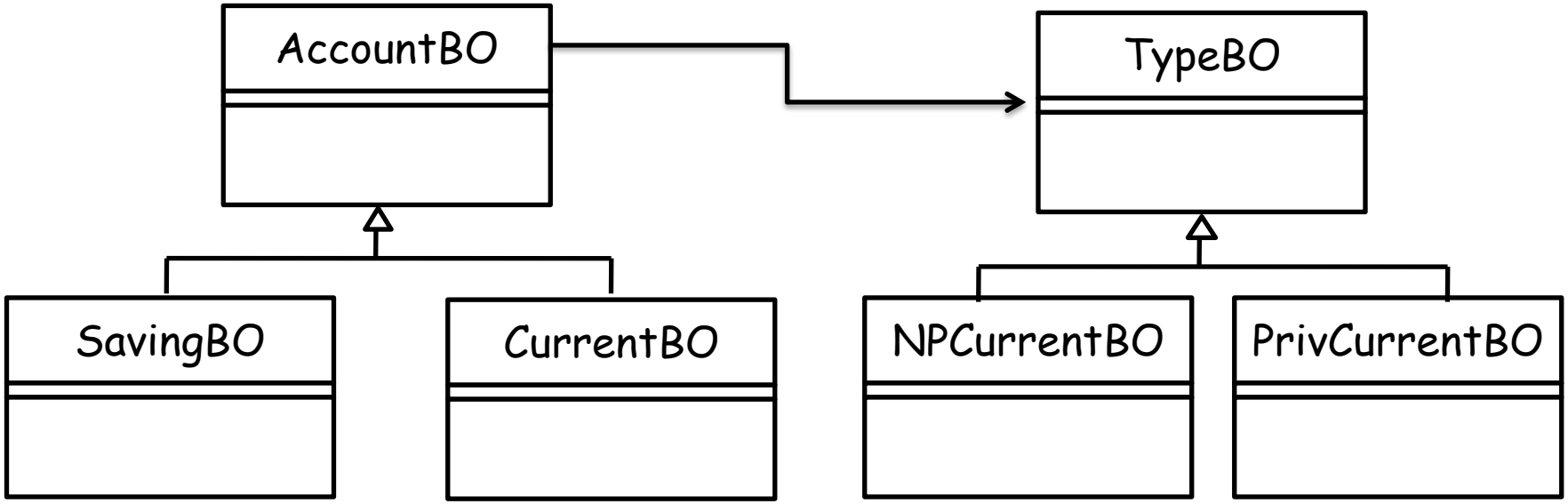
# Adapter in .Net



# Problem

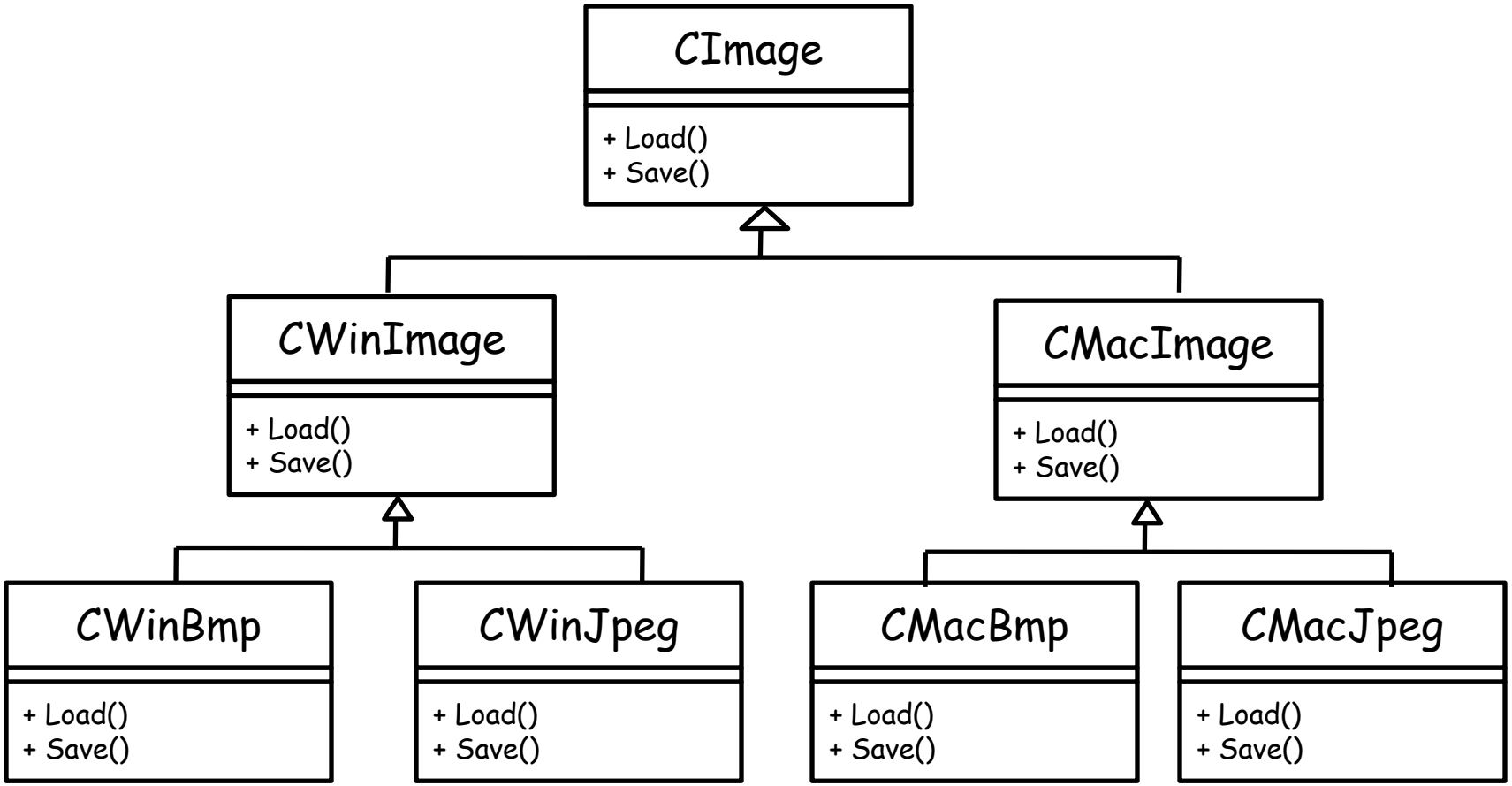

File
TextFile
BinaryFile
XmlTextFile
CsvTextFile
JsonTextFile
XmlBinaryFile
JsonBinaryFile

# Applying Bridge

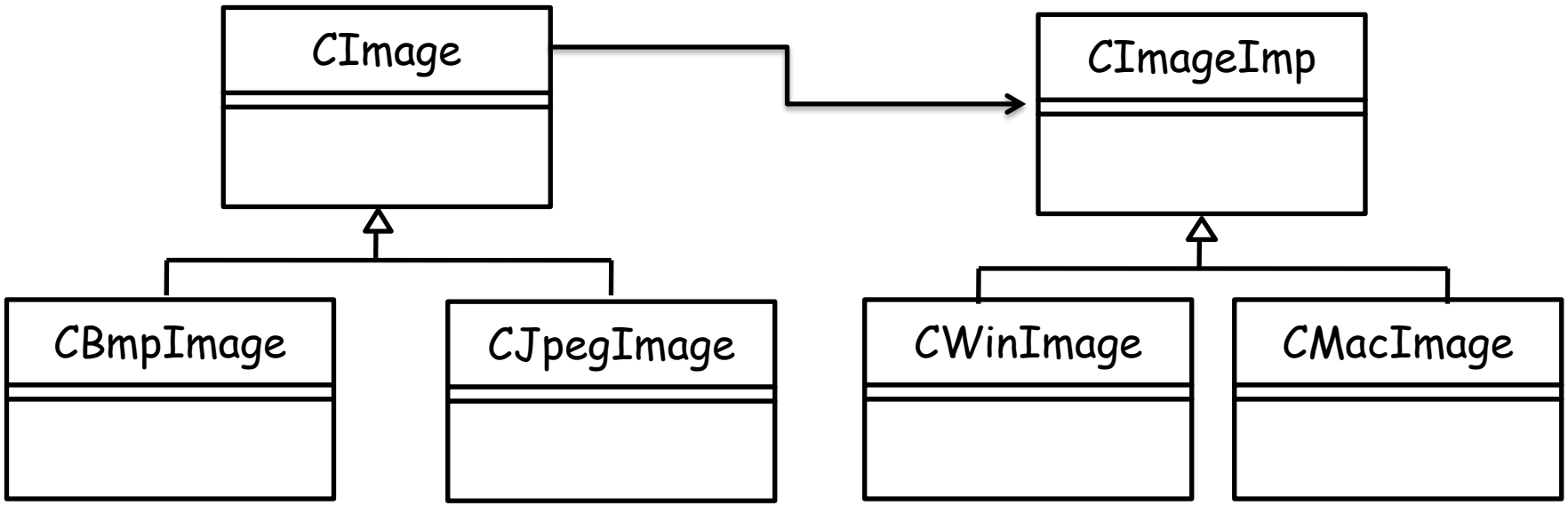


Decide if two orthogonal dimensions exist in the domain. These independent concepts could be: abstraction/platform, or domain/infrastructure, or front-end/back-end, or interface/implementation.

# Problem

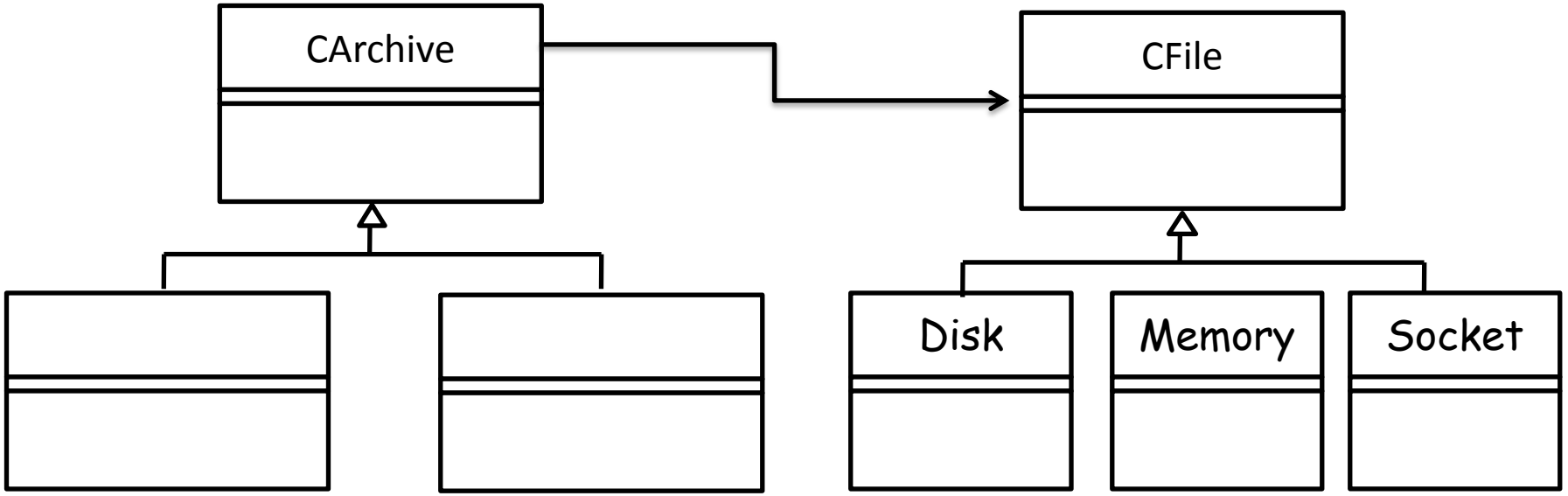

CImage
+ Load()
+ Save()
CWinImage
+ Load()
+ Save()
CMacImage
+ Load()
+ Save()
CWinBmp
+ Load()
+ Save()
CWinJpeg
+ Load()
+ Save()
CMacBmp
+ Load()
+ Save()
CMacJpeg
+ Load()
+ Save()

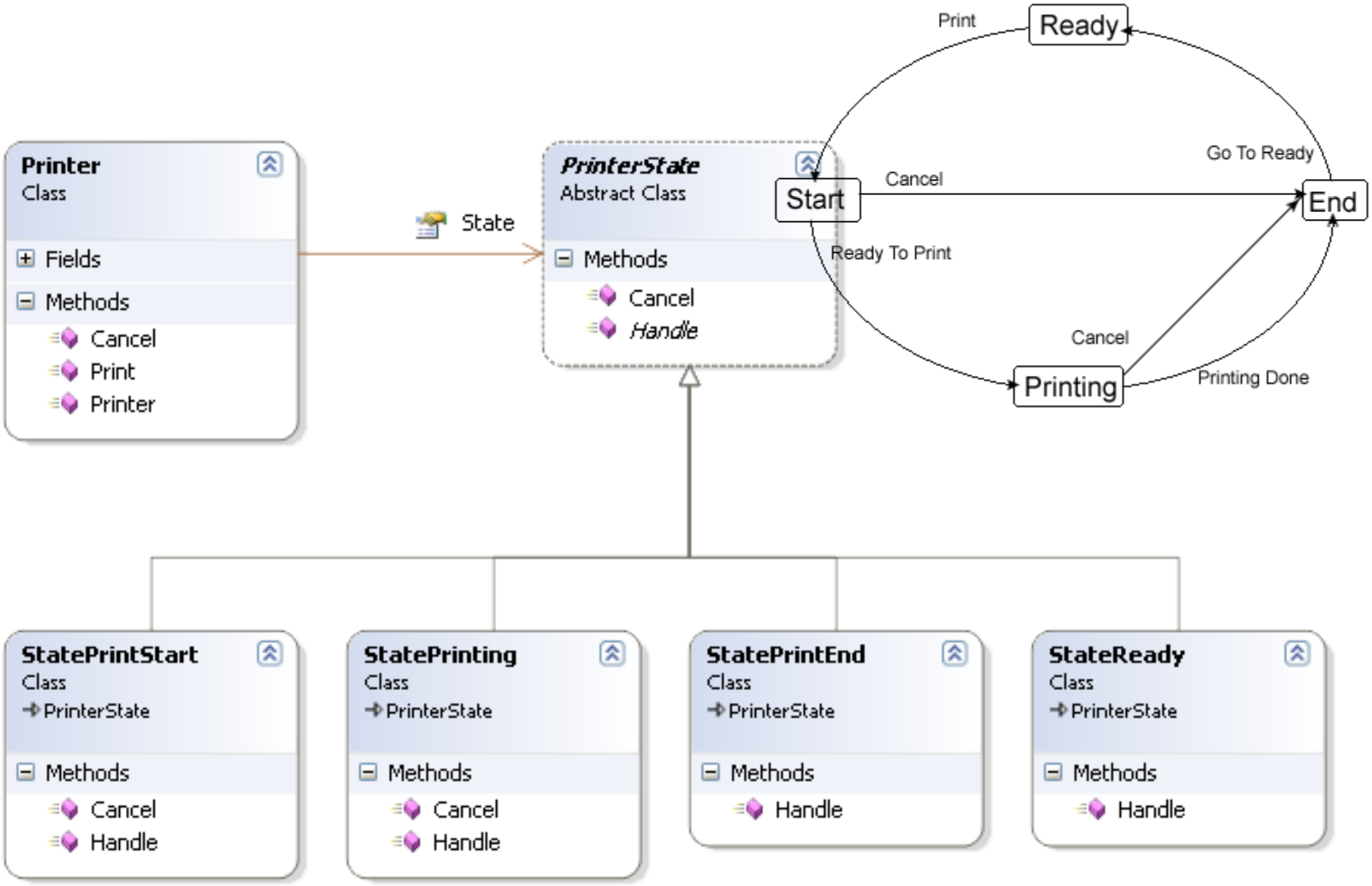
# Applying Bridge



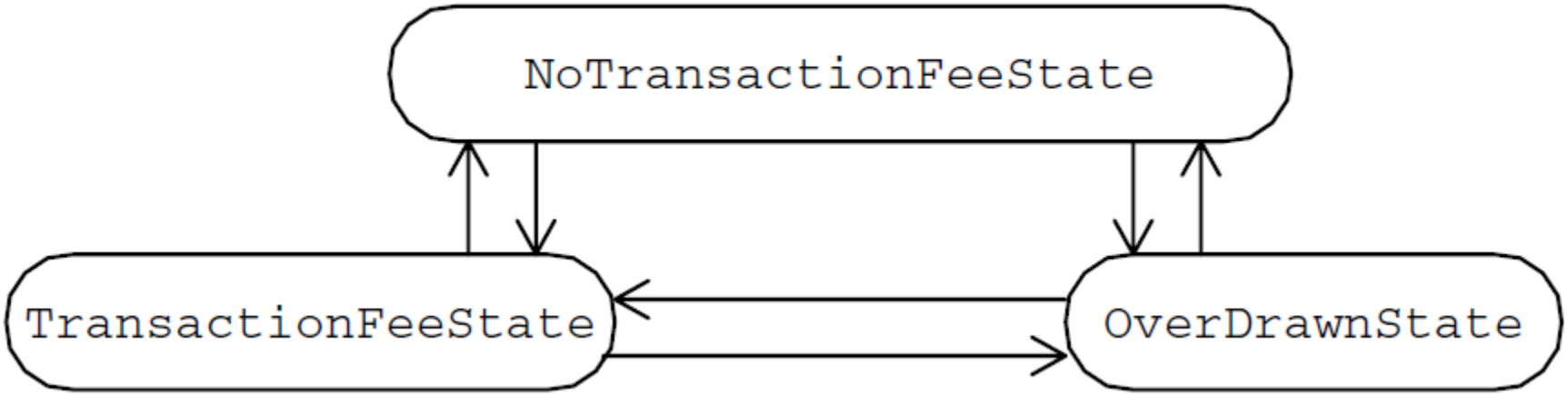

Strategy and Bridge are almost identical, differing mostly in intent only.

# Applying Bridge in MFC



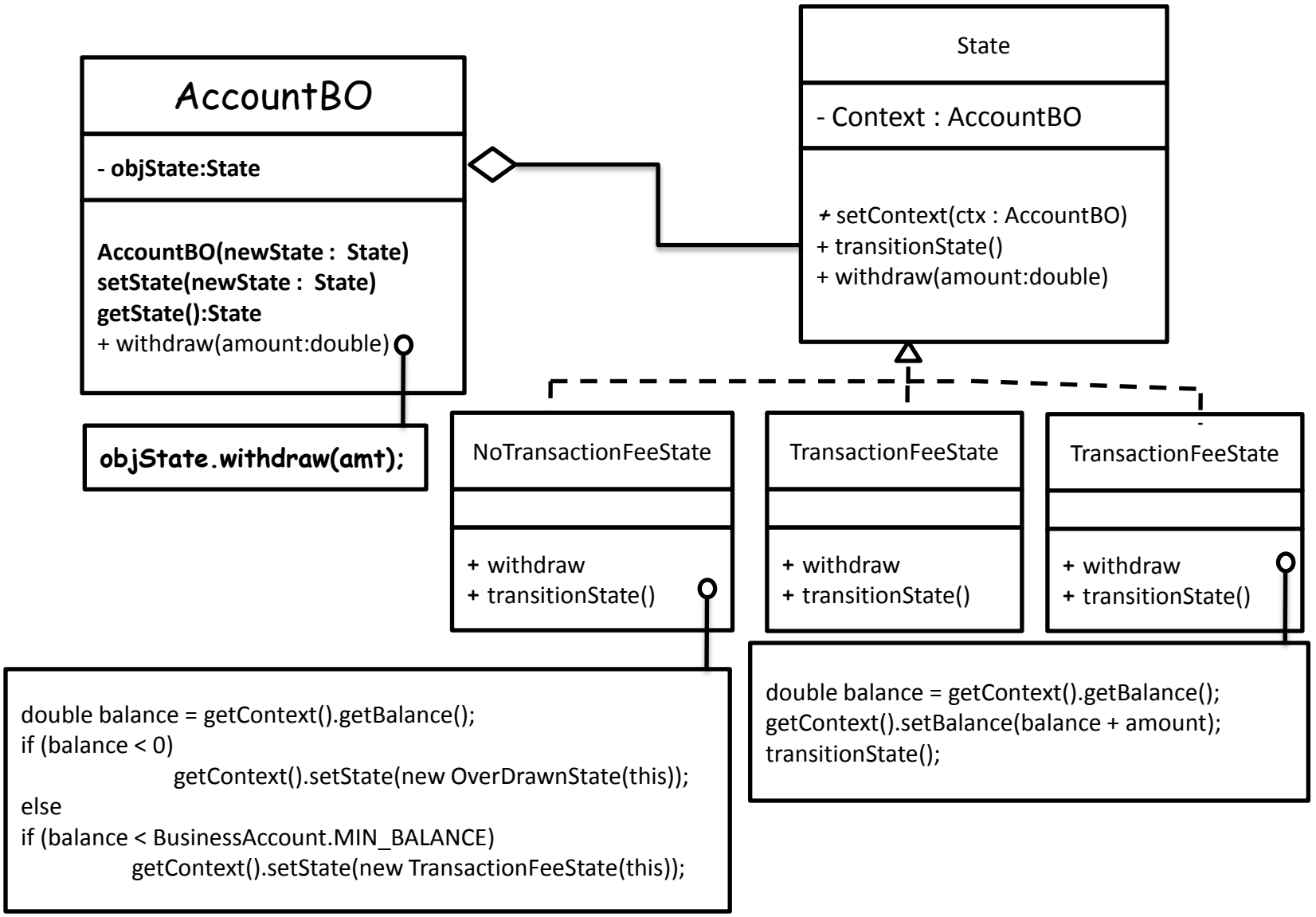

Printer
Class
Fields
Methods
Cancel
Print
Printer
State
PrinterState
Abstract Class
Methods
Cancel
Handle
Print
Ready
Go To Ready
Start
Cancel
End
Ready To Print
Cancel
Printing
Printing Done
StatePrintStart
Class
PrinterState
Methods
Cancel
Handle
StatePrinting
Class
PrinterState
Methods
Cancel
Handle
StatePrintEnd
Class
PrinterState
Methods
Handle
StateReady
Class
PrinterState
Methods
Handle

# Problem


NoTransactionFeeState
TransactionFeeState
OverDrawnState

| *From* | *To* | *What Causes the Transition* |
|---|---|---|
| No transaction fee state | Transaction fee state | A withdrawal that can make the balance positive but less than the minimum balance. |
| | Overdrawn state | A withdrawal that can make the balance negative. |
| Transaction fee state | No transaction fee state | A deposit that can make the balance greater than the minimum balance. |
| | Overdrawn state | A withdrawal that can make the balance negative. |
| Overdrawn state | No transaction fee state | A deposit that can make the balance greater than the minimum balance. |
| | Transaction fee state | A deposit that can make the balance positive but less than the minimum balance. |

# Applying State



- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the “handle/body” idiom.

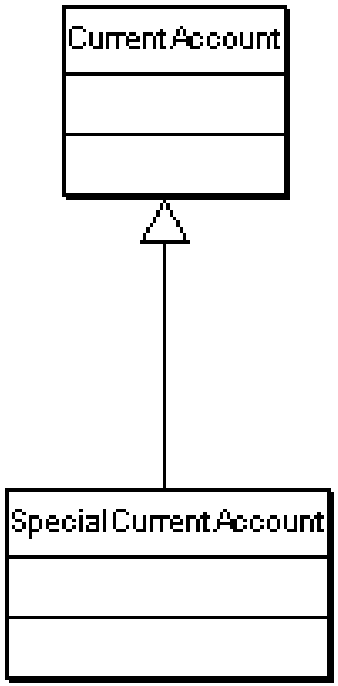
Strategy are initiated and controlled by the client.

# Strategy v/s Bridge

- With Strategy, the choice of algorithm is fairly stable. With State, a change in the state of the "context" object causes it to select from its "palette" of Strategy objects.

- Bridge admits hierarchies of envelope classes, whereas State allows only one

- A Strategy is the parameterized variation of behavior.
- The State pattern allows the dynamic variation of behavior.
- A **Bridge** is when you want to vary the implementation and interface separately.
- can think of a Bridge is a combination of a Proxy (or Adapter) and a Strategy.
- While the Strategy pattern is meant for *behavior*, the Bridge pattern is meant for *structure*.
- combined use of Strategy and Template Method is called Bridge.


CurrentAccount
SpecialCurrentAccount

# Liskov Substitution Principle (LSP)

- Subtypes must be substitutable for their base types.

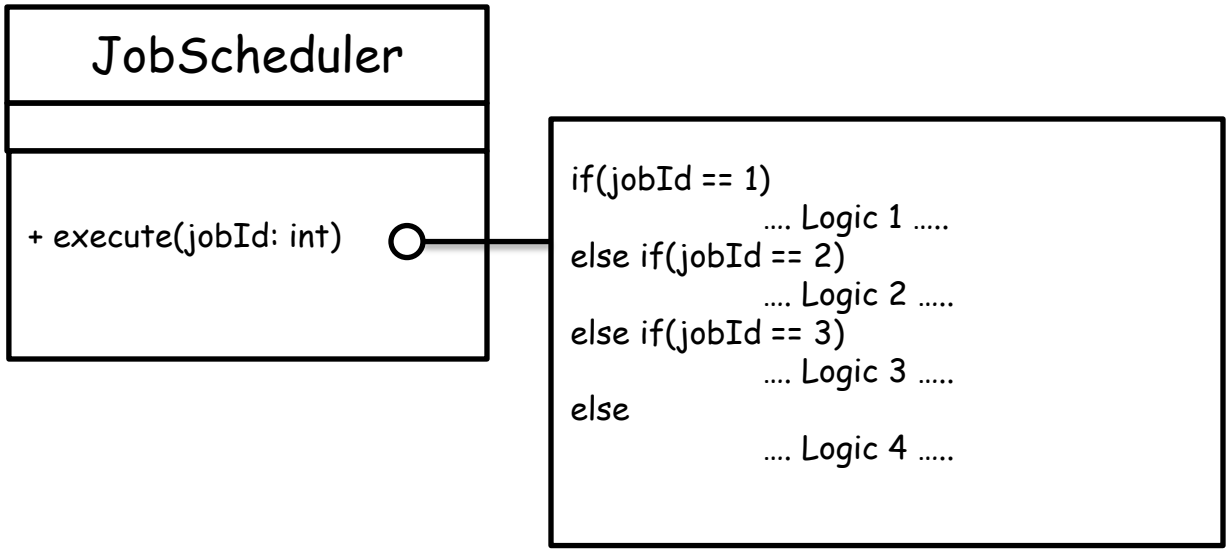
```
class T{}
class S : T{}

S o1;
T o2;

void P(T arg)
{
   // behavior
}
```

The behavior of P is unchanged when $o_1$ is substituted for $o_2$.

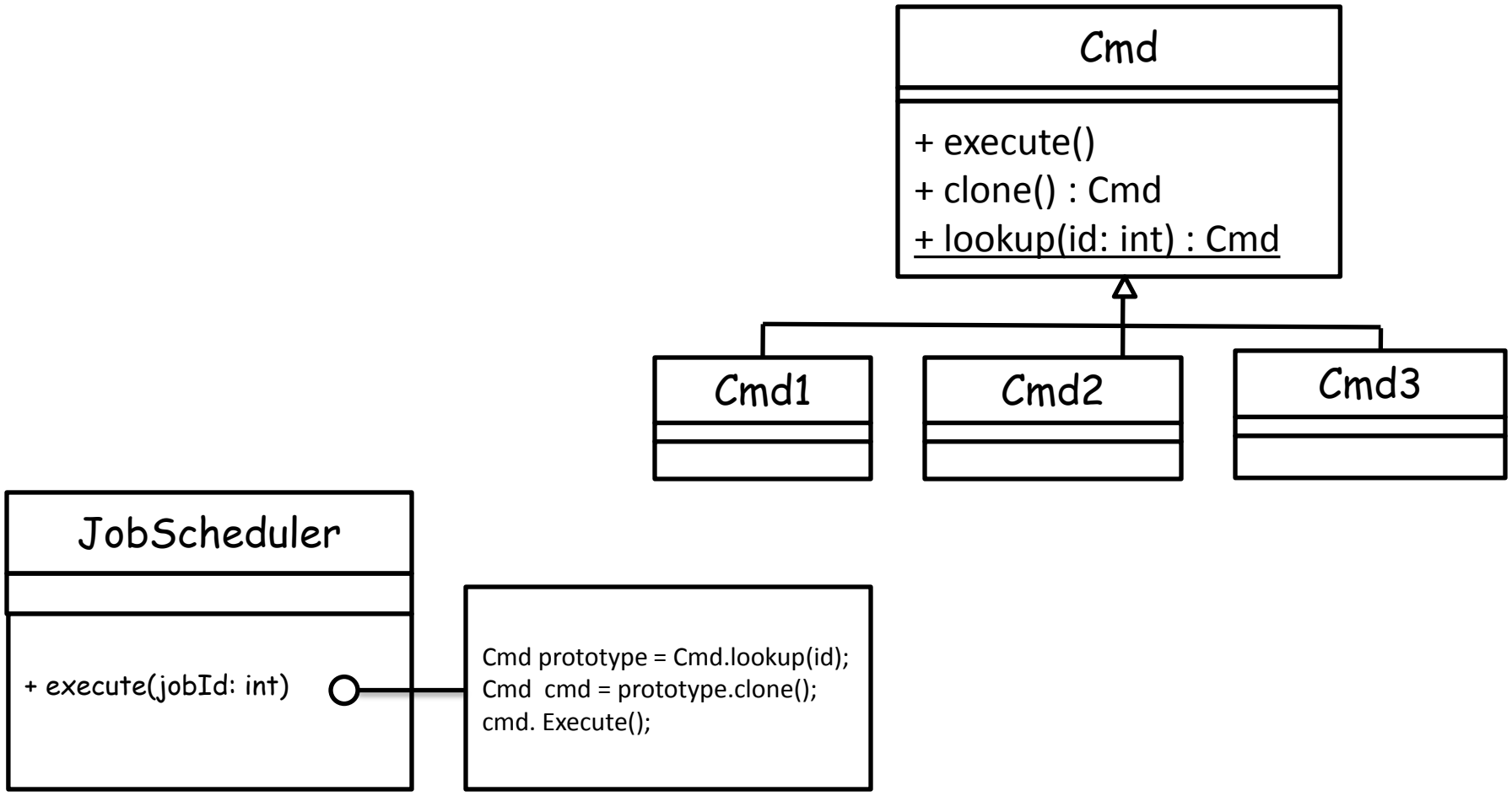
- In class hierarchies, it should be possible to treat a specialized object as if it were a base class object.

- Inheritance is one of the most abused concepts

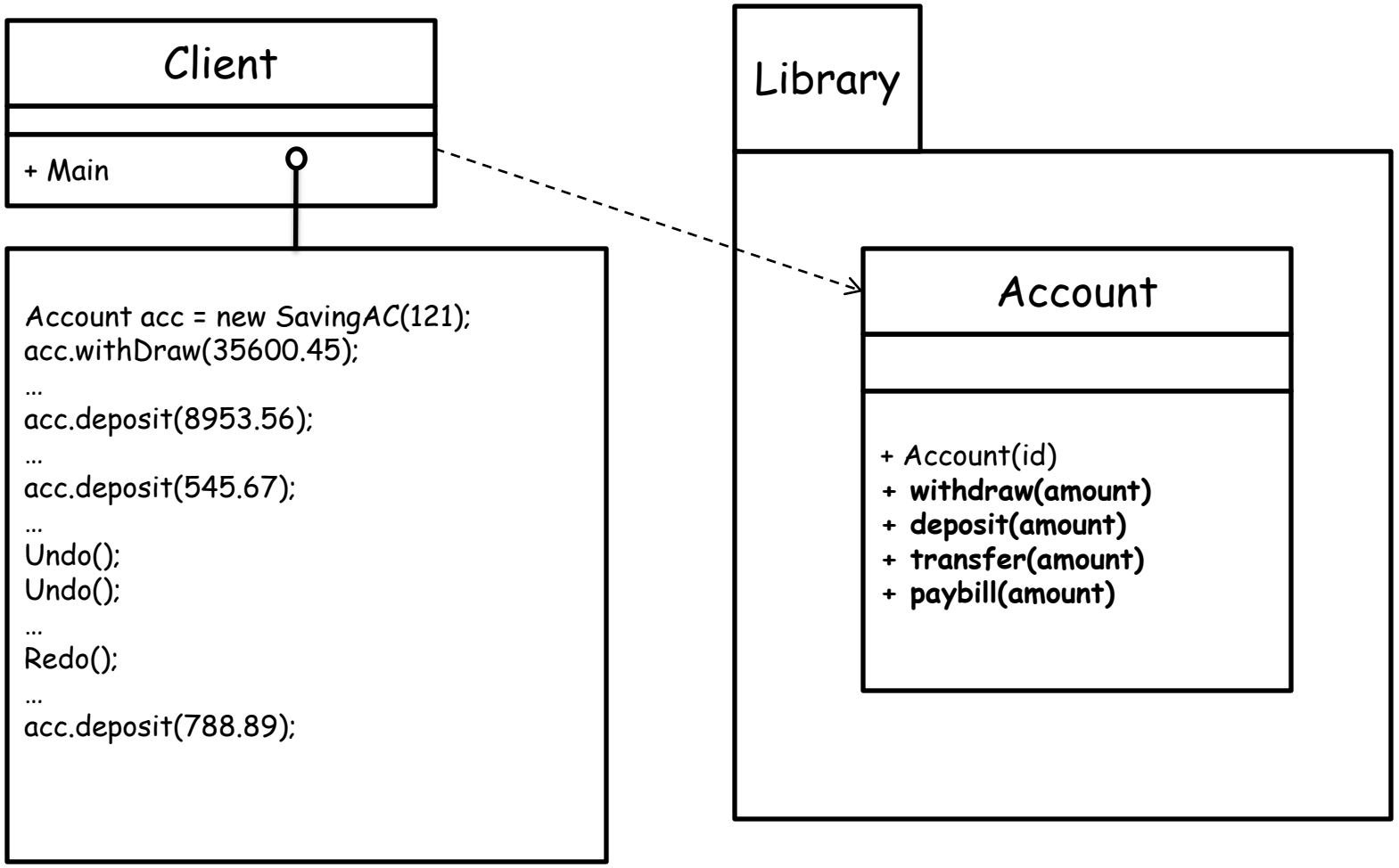
# Problem


JobScheduler
+ execute(jobId: int)
if(jobId == 1)
.... Logic 1 .....
else if(jobId == 2)
.... Logic 2 .....
else if(jobId == 3)
.... Logic 3 .....
else
.... Logic 4 .....

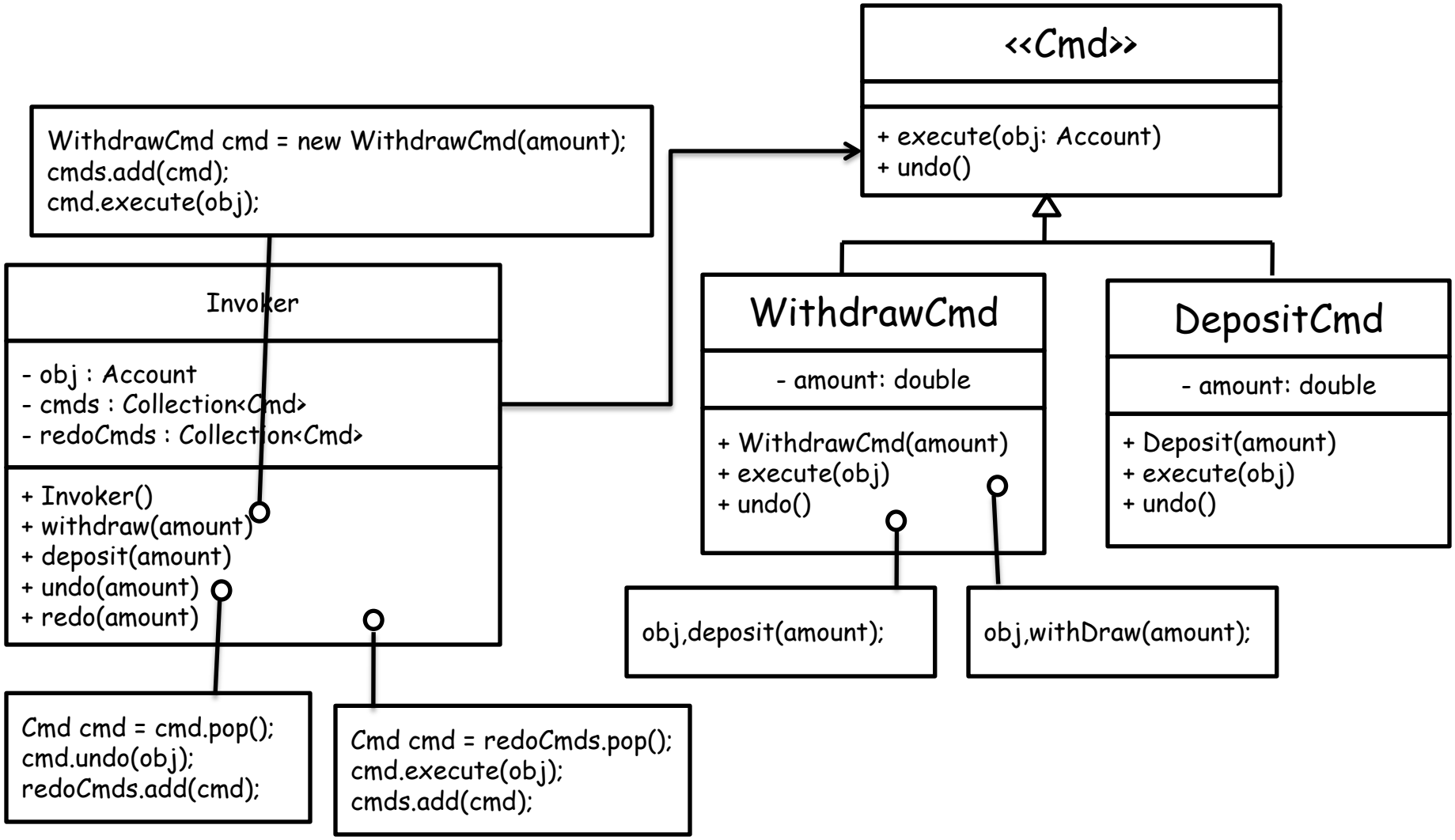
# Applying Command

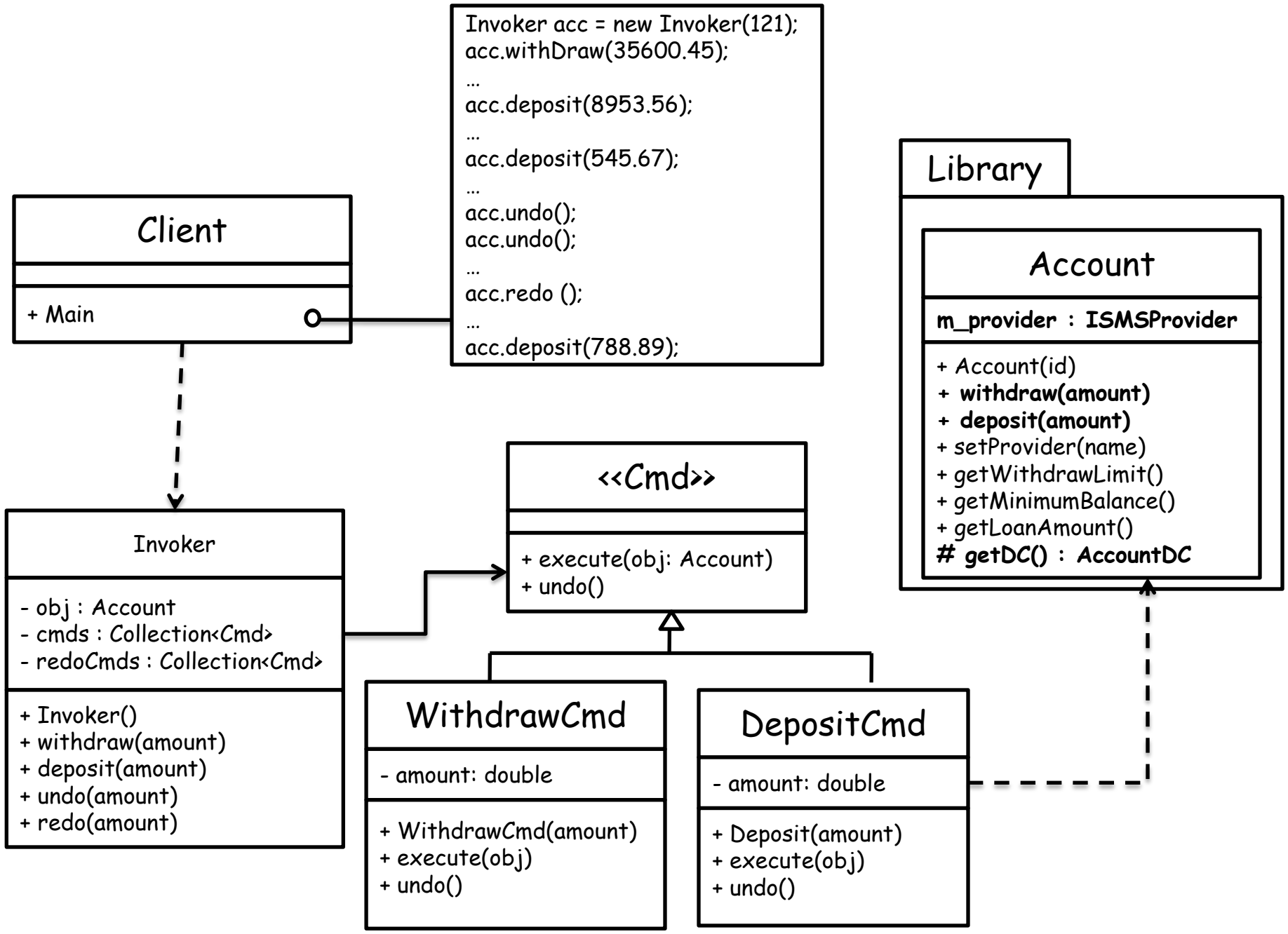


# Long running transaction Problem

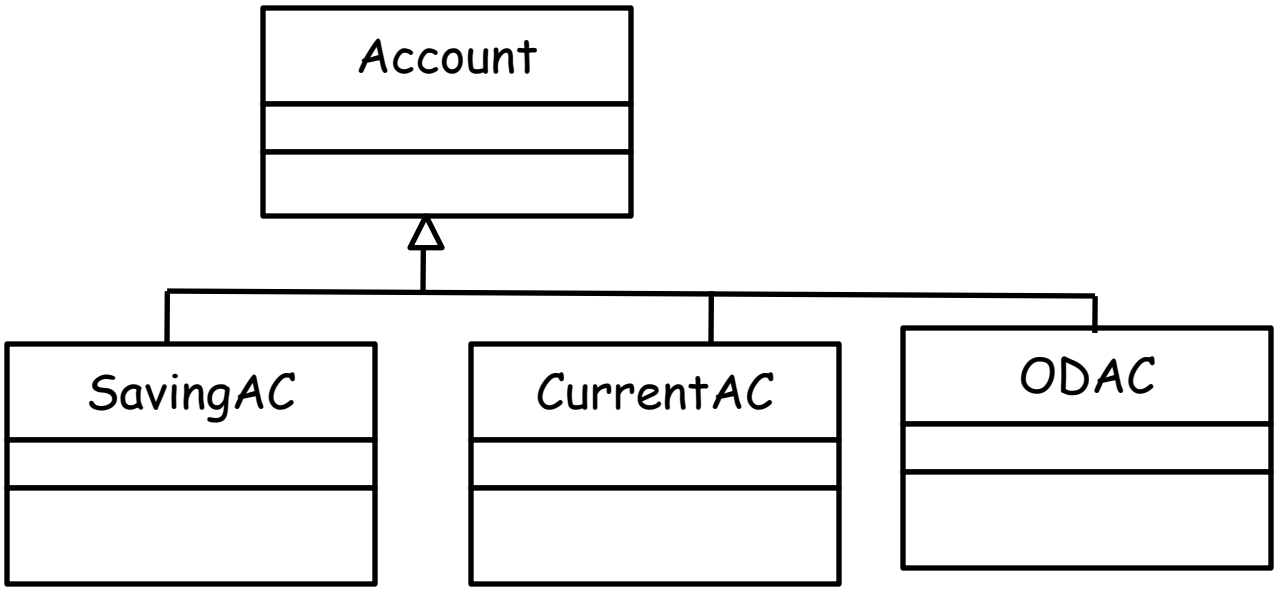


# Applying Command



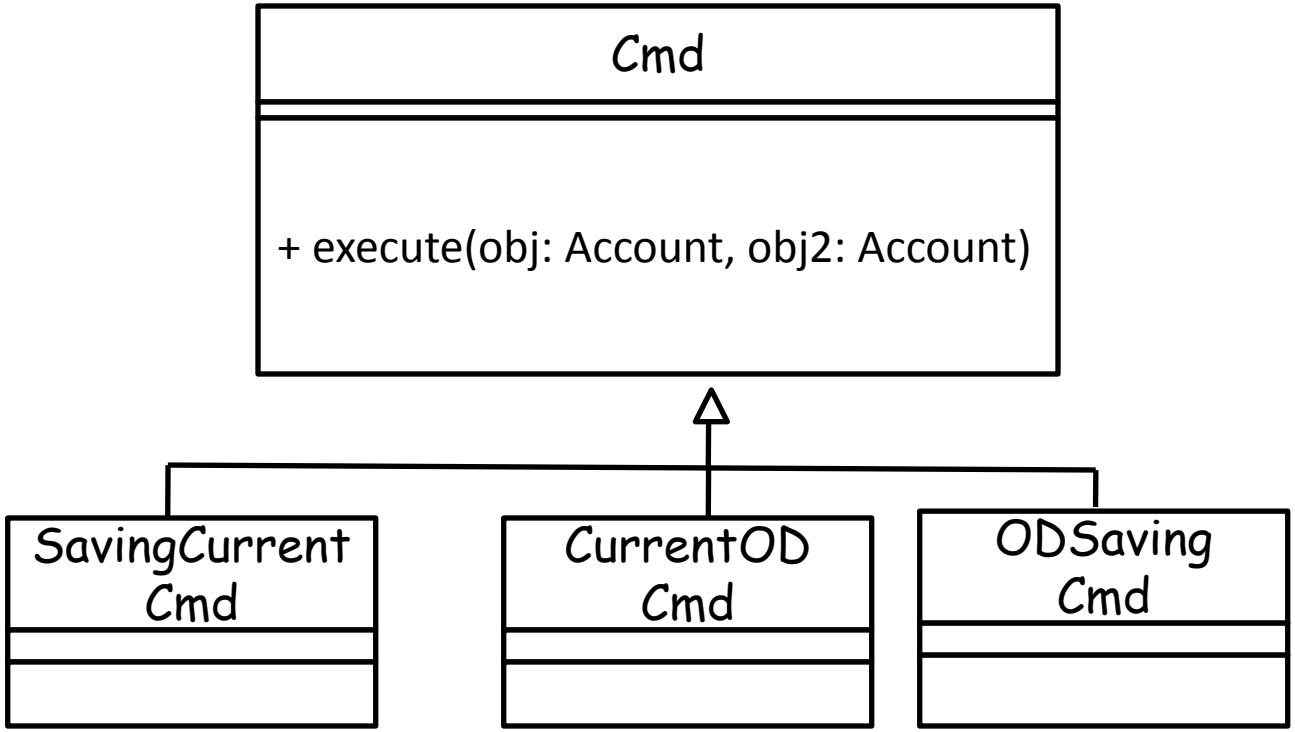

Invoker acc = new Invoker(121);
acc.withDraw(35600.45);
...
acc.deposit(8953.56);
...
acc.deposit(545.67);
...
acc.undo();
acc.undo();
...
acc.redo ();
...
acc.deposit(788.89);
Client
+ Main
Library
Account
m_provider : ISMSProvider
+ Account(id)
+ withdraw(amount)
+ deposit(amount)
+ setProvider(name)
+ getWithdrawLimit()
+ getMinimumBalance()
+ getLoanAmount()
# getDC() : AccountDC
<<Cmd>>
+ execute(obj: Account)
+ undo()
Invoker
- obj : Account
- cmds : Collection<Cmd>
- redoCmds : Collection<Cmd>
+ Invoker()
+ withdraw(amount)
+ deposit(amount)
+ undo(amount)
+ redo(amount)
WithdrawCmd
- amount: double
+ WithdrawCmd(amount)
+ execute(obj)
+ undo()
DepositCmd
- amount: double
+ Deposit(amount)
+ execute(obj)
+ undo()

# Dual Dispatching Problem



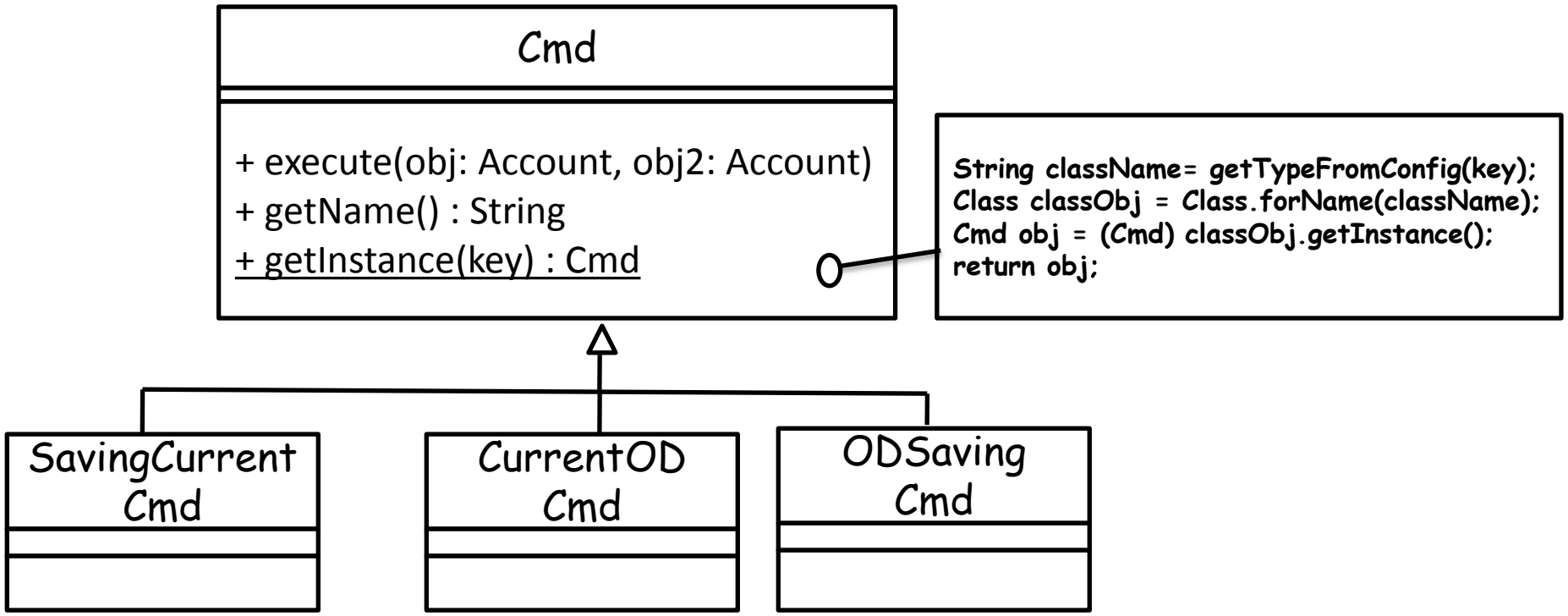

```
GetOffer(Account obj, Account obj2)
{
        If(typeof(obj)== typeof(SavingAC) && typeof(obj2) == typeof(CurrentAC))
                //…..
        If(typeof(obj)== typeof(ODAC) && typeof(obj2) == typeof(CurrentAC))
                //…..
        If(typeof(obj)== typeof(SavingAC) && typeof(obj2) == typeof(ODAC))
                //…..
}
```

# Applying Command



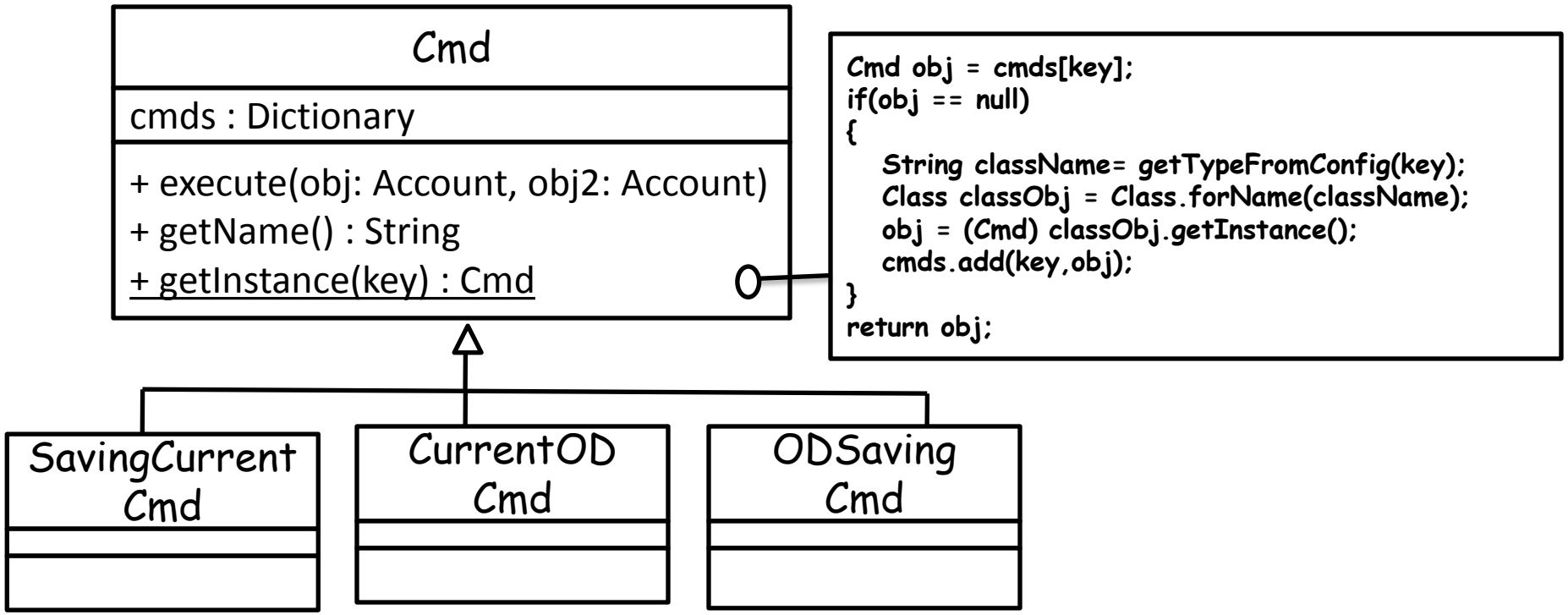

```
GetOffer(Account obj, Account obj2)
{
        Cmd  cmd = new ?;
        cmd. execute(obj,obj2);
}
```

# Applying Creator Method



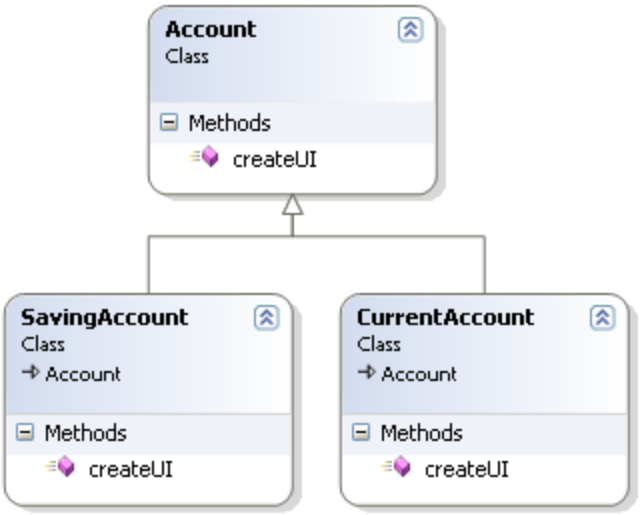

```
GetOffer(Account obj, Account obj2)
{
        String  key = obj.getName() + obj.getName();
        Cmd cmd= Cmd.getInstance(key);
        cmd. execute(obj,obj2);
}
```

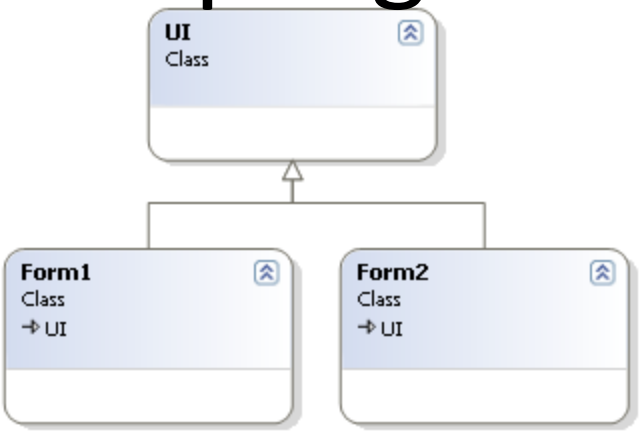
# Applying FlyWeight




```
GetOffer(Account obj, Account obj2)
{
        String  key = obj.getName() + obj.getName();
        Cmd cmd= Cmd.getInstance(key);
        cmd. execute(obj,obj2);
}
```

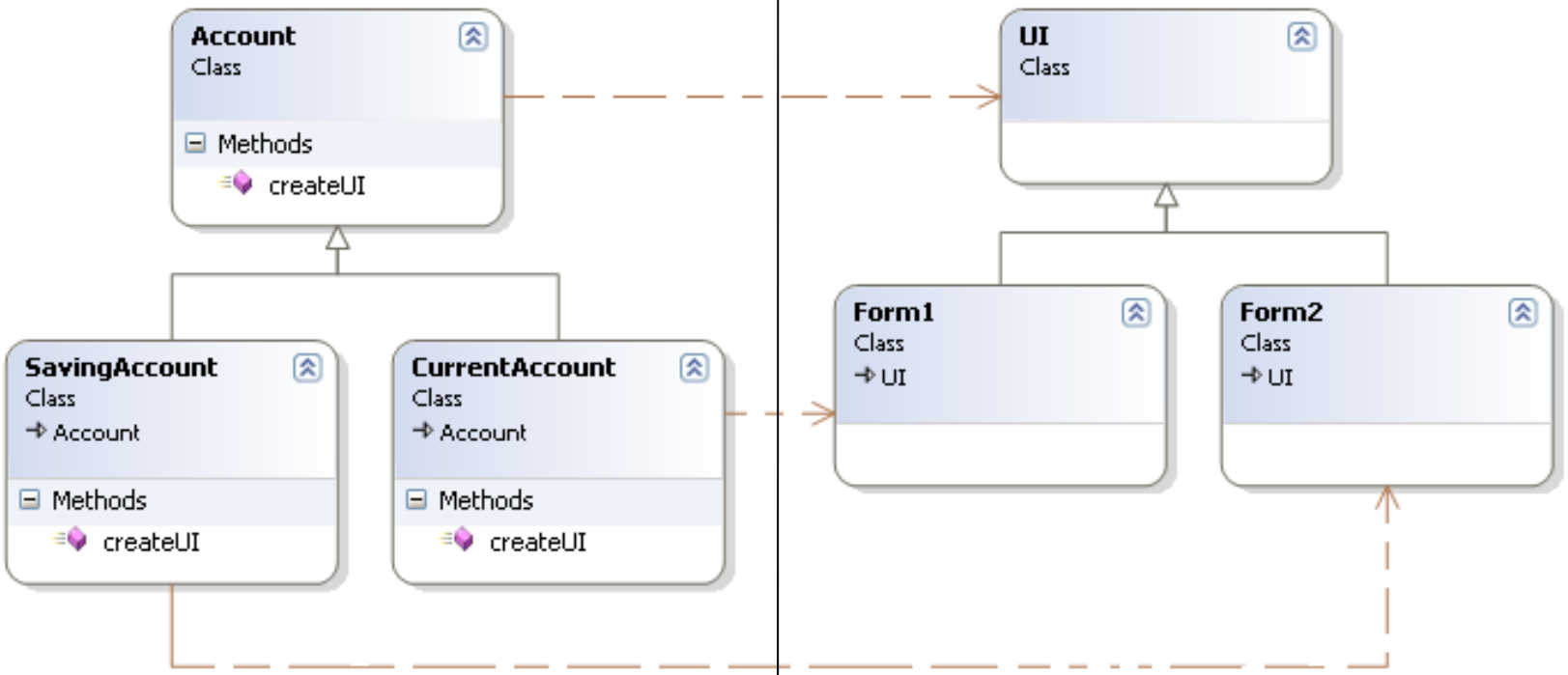
# Low Coupling




```
void showUI(Account obj)
{
        UI ui;
        if(type(obj) == type(Account))
                ui = new UI();
        if(type(obj) == type(SavingAccount))
                ui = new Form1();
        if(type(obj) == type(CurrentAccount))
                ui = new Form2();
        ui.render();
}
```

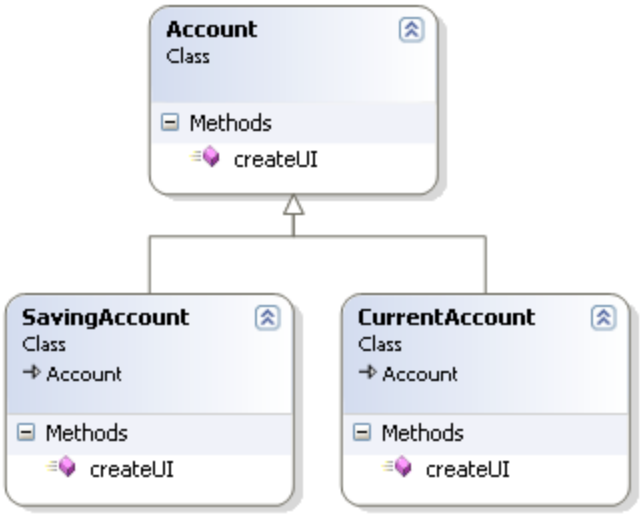
# Applying FactoryMethod

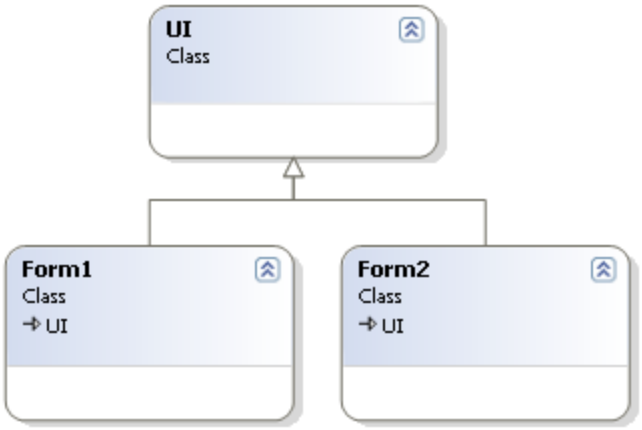



```
void createUI()
{
   return new Form1;
}
```

```
void showUI(Account obj)
{
        UI ui= obj.createUI();
        ui.render();
}
```

# Applying Static Polymorphism

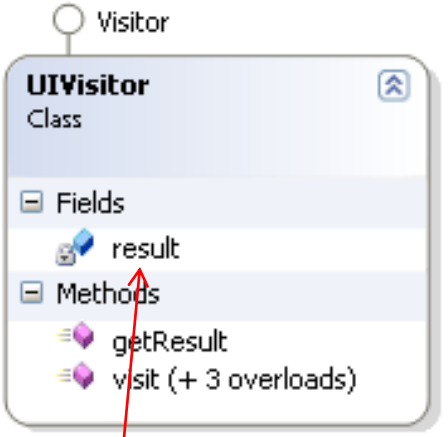



```
void ShowUI(Account obj)
{
        ....
}
void ShowUI (SavingAccount obj)
{
        ...
}
void ShowUI (CurrentAccount obj)
{
        ....
}
```

# Applying Visitor

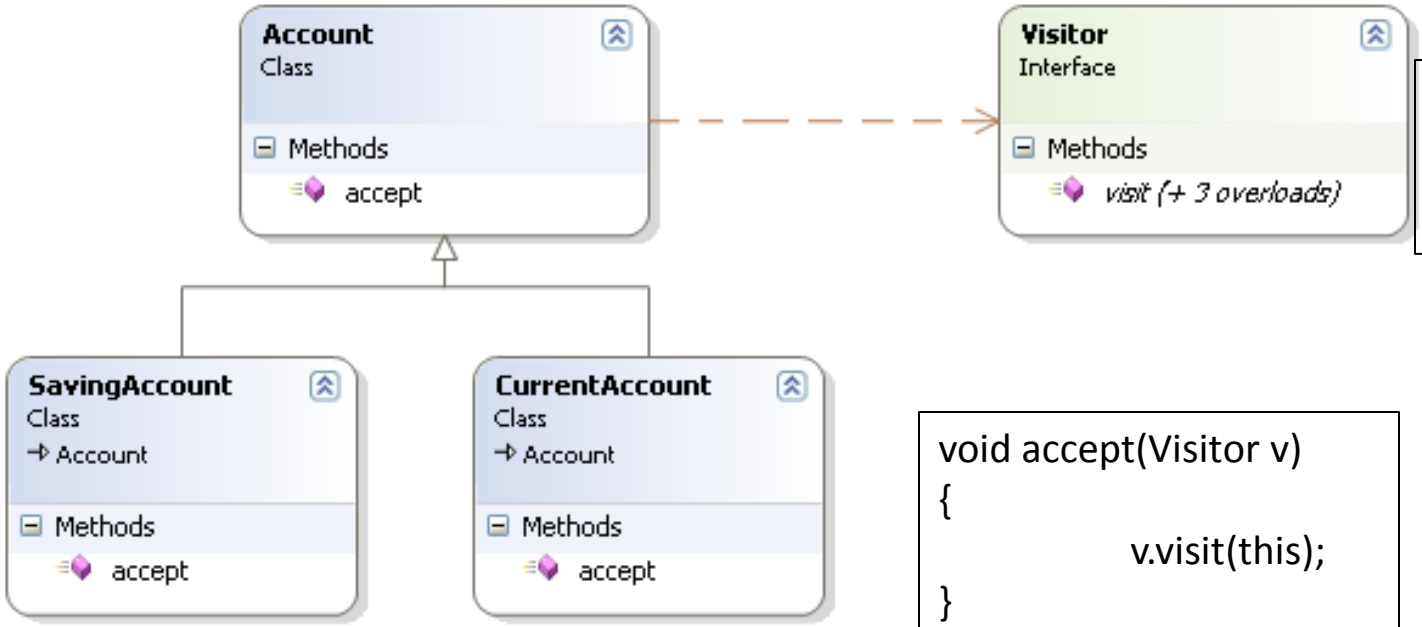

```
void visit(Account obj)
void visit(SavingAccount obj)
void visit(CurrentAccount obj)
```




```
showUI(Account obj)
{
        UIVisitor v=  new UIVisitor();
        v.Visit(obj);

        UI ui = v.getResult();
        ui.render();
}
```

```
void visit(Account obj){
        result = new UI();
}
void visit(SavingAccount obj){
        result = new Form1();
}
void visit(CurrentAccount obj){
        result =  new Form2();
}
```

# Applying Visitor

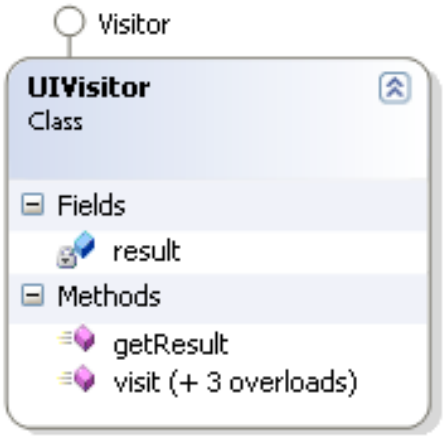



visit(Account)
visit(SavingAccount)
visit(CurrentAccount)

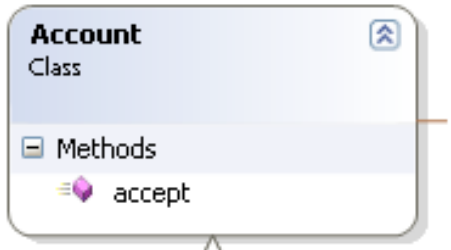
```
void accept(Visitor v)
{
        v.visit(this);
}
```

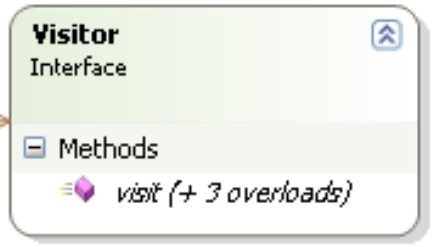
---

```
DoJob(Account obj)
{
        UIVisitor v=  new UIVisitor();
        obj.accept(v);
}
```

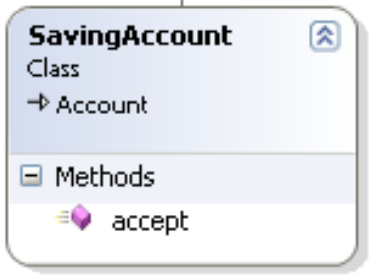


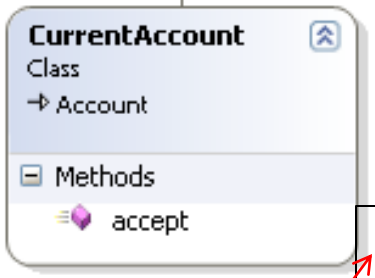

```
visit(Account){
        //.. code;
}
visit(SavingAccount){
        //.. code;
}
visit(CurrentAccount){
        //.. code;
}
```



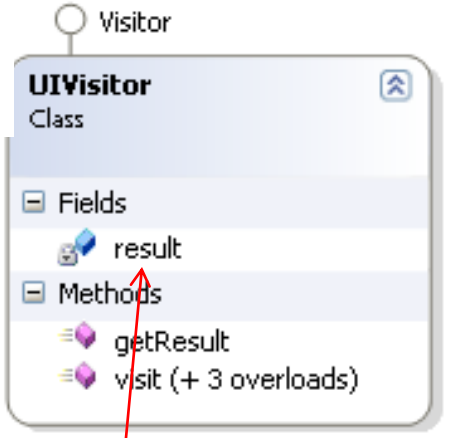




```
void visit(Account obj)
void visit(SavingAccount obj)
void visit(CurrentAccount obj)
```



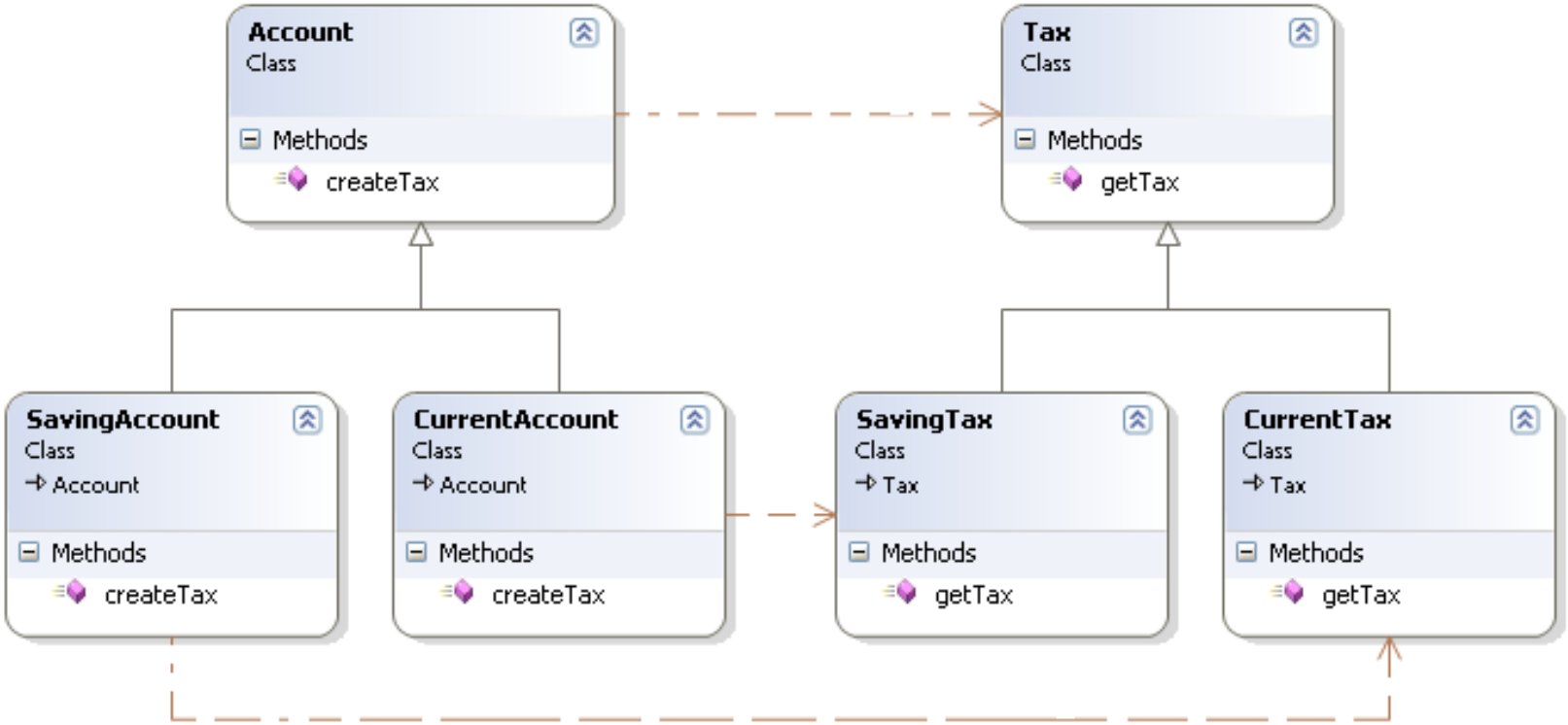




```
void accept(Visitor v)
{
        v.visit(this);
}
```



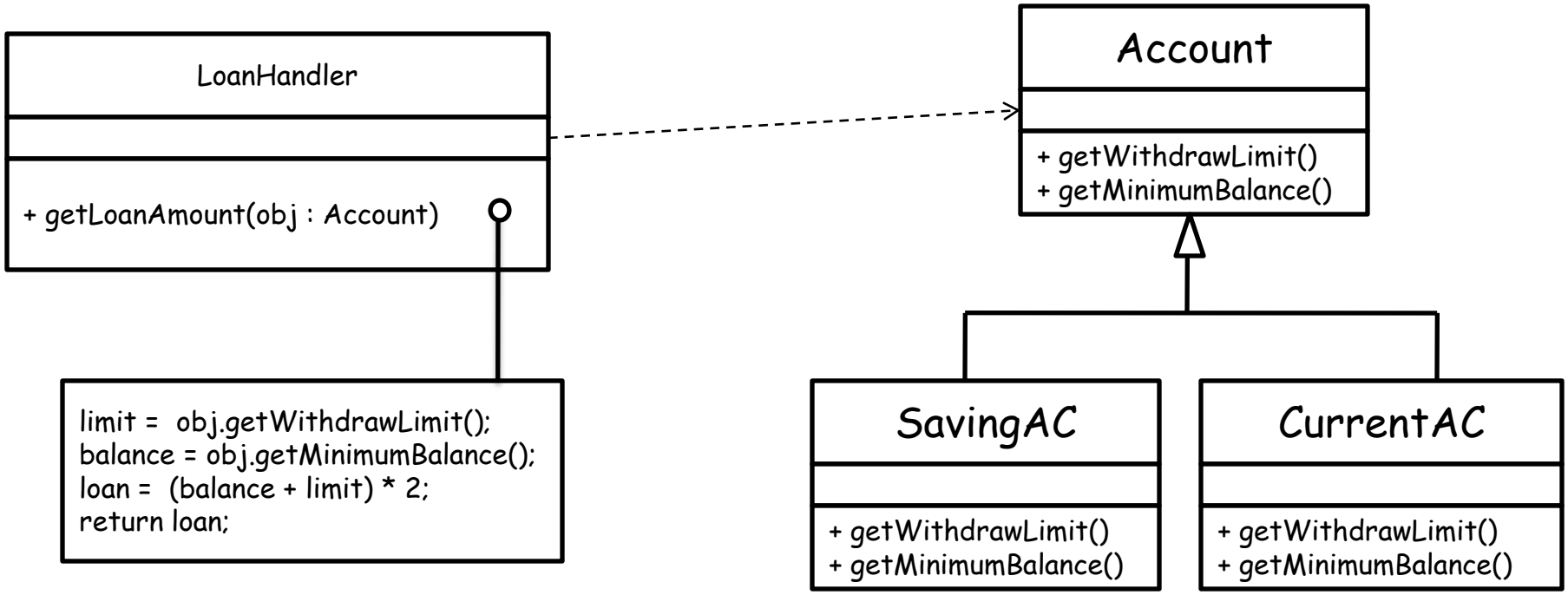

```
showUI(Account obj)
{
        UIVisitor v=  new UIVisitor();
        obj.accept(v);

        UI ui = v.getResult();
        ui.render();
}
```

```
void visit(Account obj){
        result = new UI();
}
void visit(SavingAccount obj){
        result = new Form1();
}
void visit(CurrentAccount obj){
        result =  new Form2();
}
```




```
void DoJob(Account obj)
{
        Tax tax = obj.createTax();
        tax.getTax();
}
```

# Problem

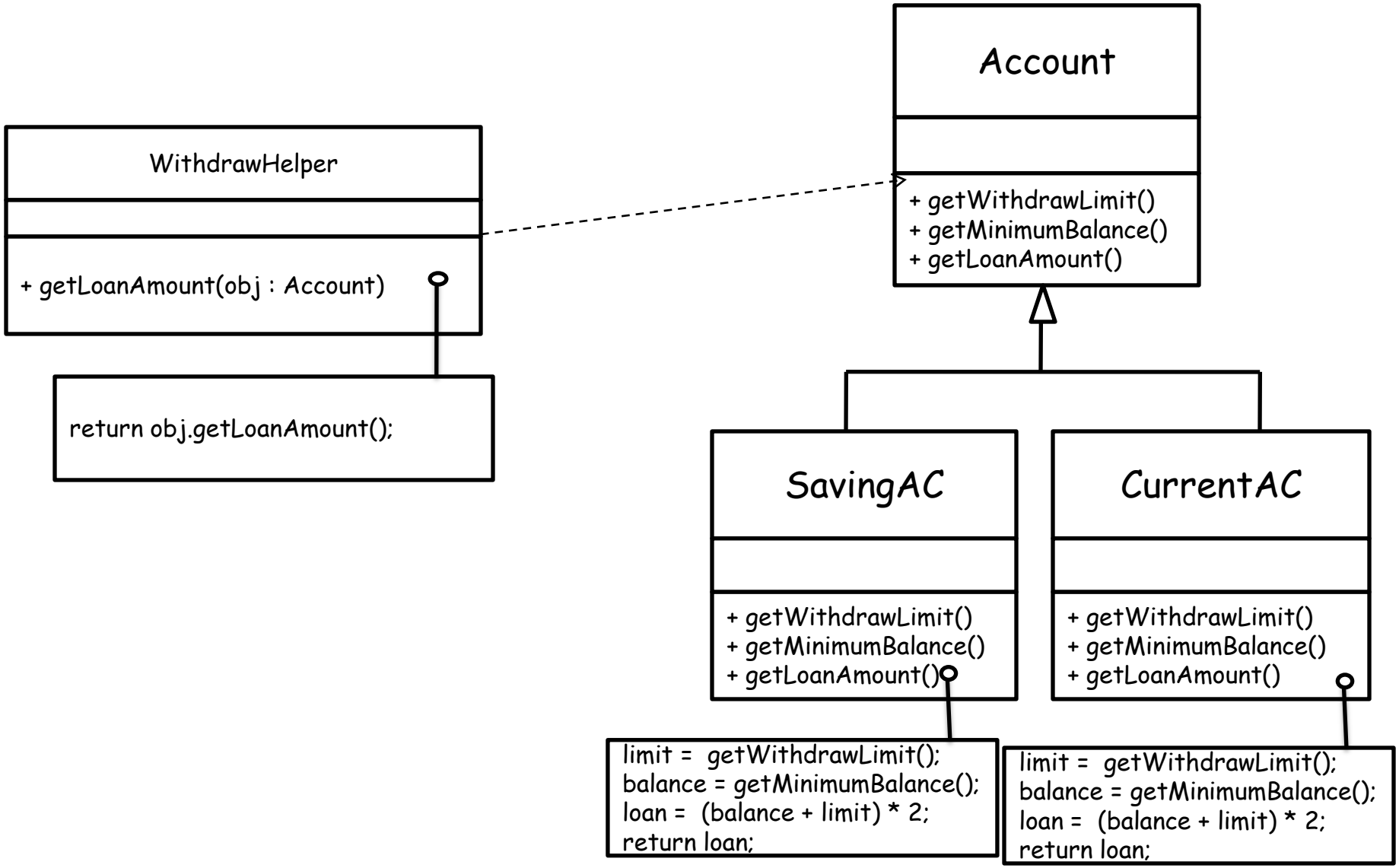

LoanHandler
+ getLoanAmount(obj : Account)
limit = obj.getWithdrawLimit();
balance = obj.getMinimumBalance();
loan = (balance + limit) * 2;
return loan;
Account
+ getWithdrawLimit()
+ getMinimumBalance()
SavingAC
+ getWithdrawLimit()
+ getMinimumBalance()
CurrentAC
+ getWithdrawLimit()
+ getMinimumBalance()

# Information Expert

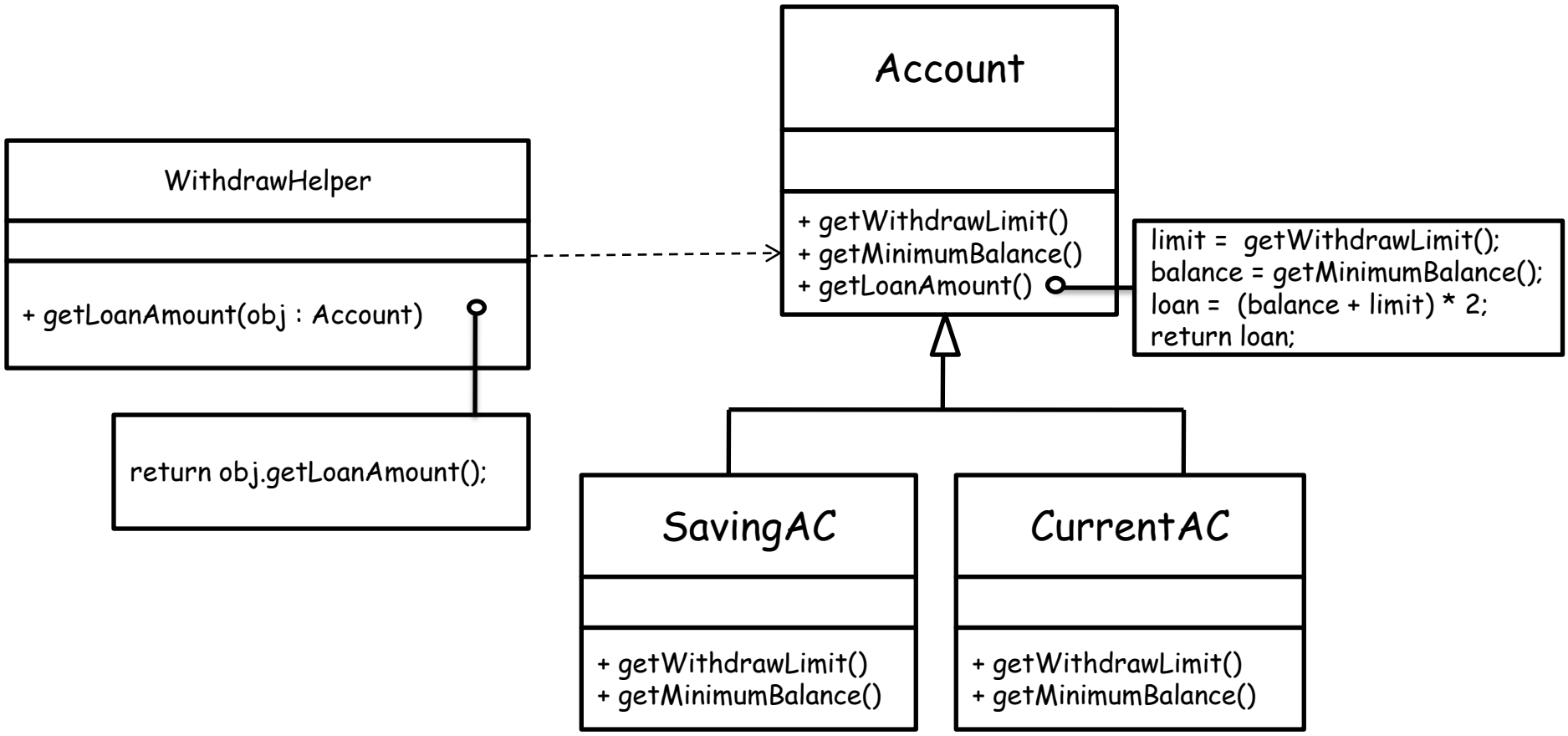
A system will have hundreds of classes. How do I begin to assign responsibilities to them?

Assign responsibility to the Information Expert – the class that has the information necessary to fulfill the responsibility.

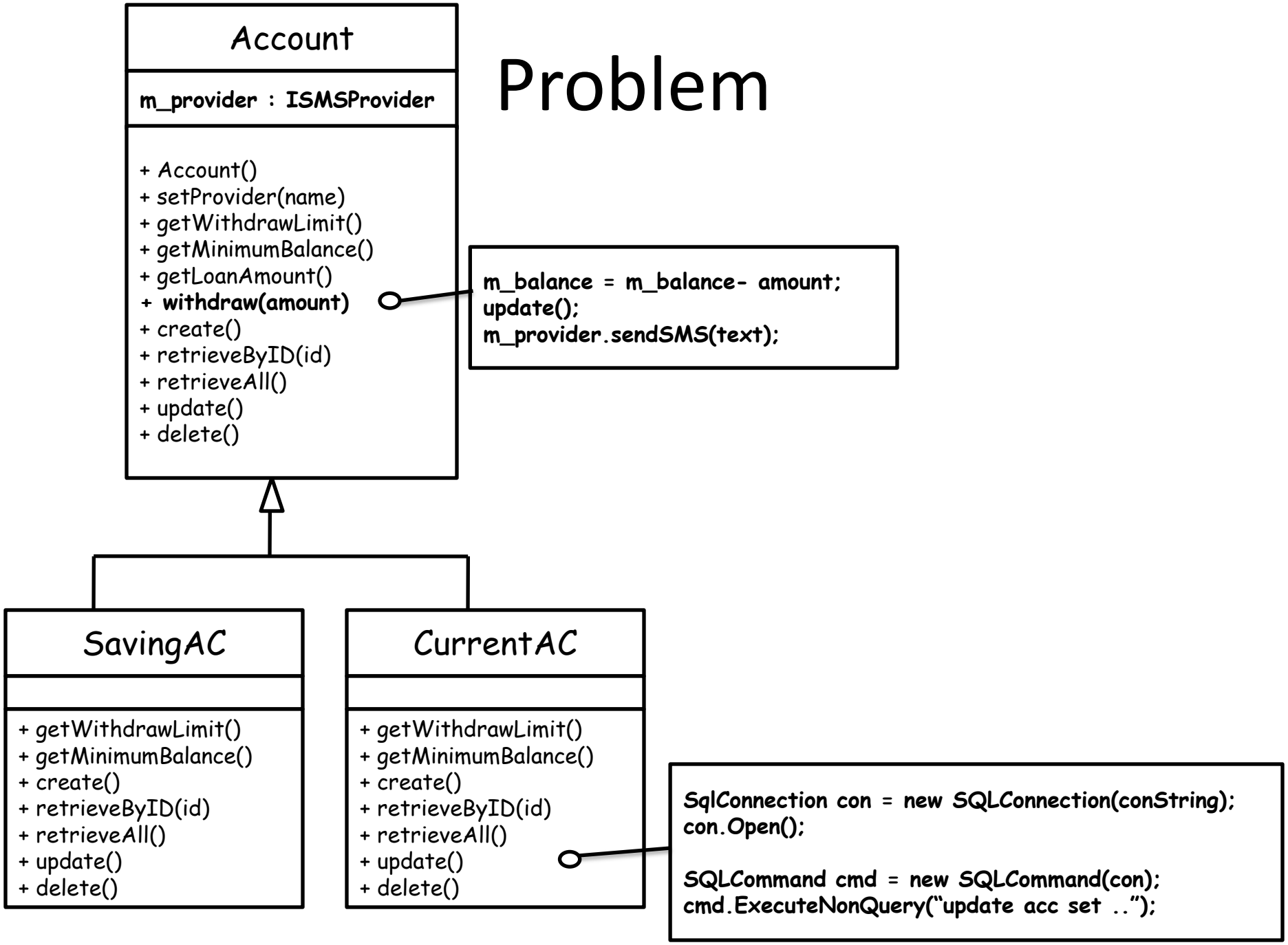
# Applying Information Expert

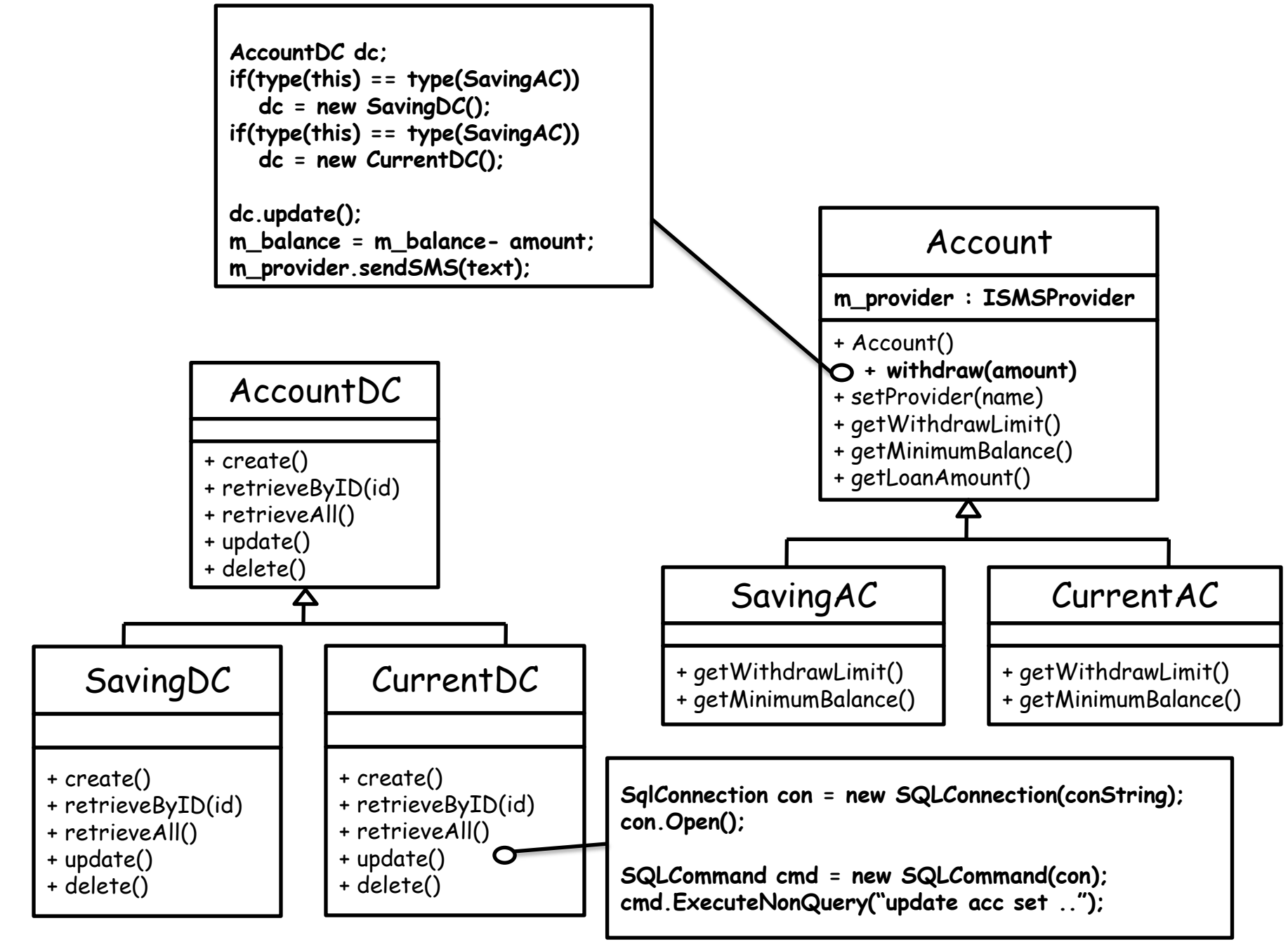


# Applying Template Method



# Problem


Account
m_provider : ISMSProvider
+ Account()
+ setProvider(name)
+ getWithdrawLimit()
+ getMinimumBalance()
+ getLoanAmount()
+ withdraw(amount)
+ create()
+ retrieveByID(id)
+ retrieveAll()
+ update()
+ delete()
m_balance = m_balance- amount;
update();
m_provider.sendSMS(text);
SavingAC
+ getWithdrawLimit()
+ getMinimumBalance()
+ create()
+ retrieveByID(id)
+ retrieveAll()
+ update()
+ delete()
CurrentAC
+ getWithdrawLimit()
+ getMinimumBalance()
+ create()
+ retrieveByID(id)
+ retrieveAll()
+ update()
+ delete()
SqlConnection con = new SQLConnection(conString);
con.Open();
SQLCommand cmd = new SQLCommand(con);
cmd.ExecuteNonQuery("update acc set ..");


AccountDC dc;
if(type(this) == type(SavingAC))
dc = new SavingDC();
if(type(this) == type(SavingAC))
dc = new CurrentDC();
dc.update();
m_balance = m_balance- amount;
m_provider.sendSMS(text);
Account
m_provider : ISMSProvider
+ Account()
+ withdraw(amount)
+ setProvider(name)
+ getWithdrawLimit()
+ getMinimumBalance()
+ getLoanAmount()
AccountDC
+ create()
+ retrieveByID(id)
+ retrieveAll()
+ update()
+ delete()
SavingAC
+ getWithdrawLimit()
+ getMinimumBalance()
CurrentAC
+ getWithdrawLimit()
+ getMinimumBalance()
SavingDC
+ create()
+ retrieveByID(id)
+ retrieveAll()
+ update()
+ delete()
CurrentDC
+ create()
+ retrieveByID(id)
+ retrieveAll()
+ update()
+ delete()
SqlConnection con = new SQLConnection(conString);
con.Open();
SQLCommand cmd = new SQLCommand(con);
cmd.ExecuteNonQuery("update acc set ..");

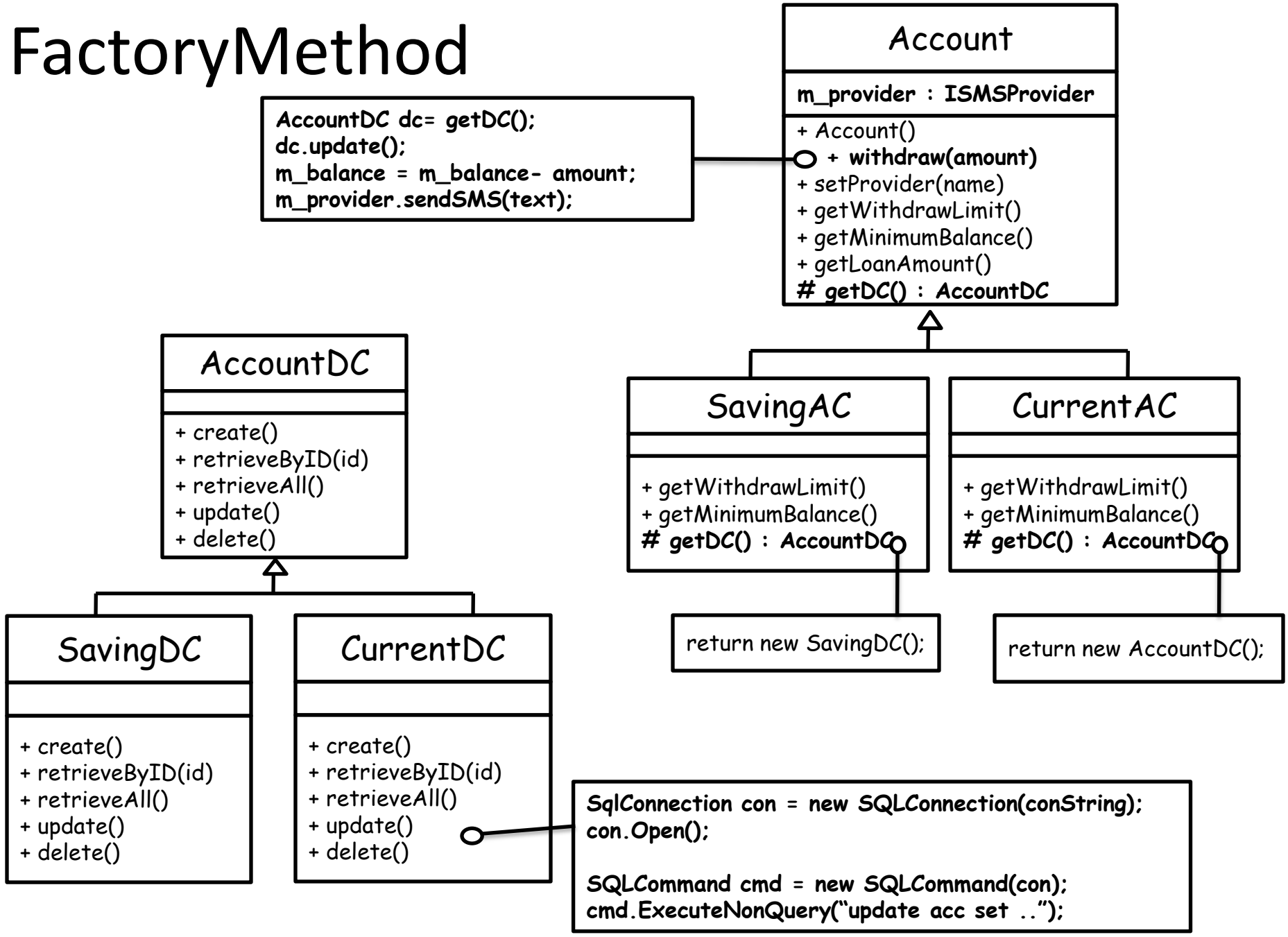
# Dependency Inversion principle

Abstractions should not depend upon details.
Details should depend upon abstractions.

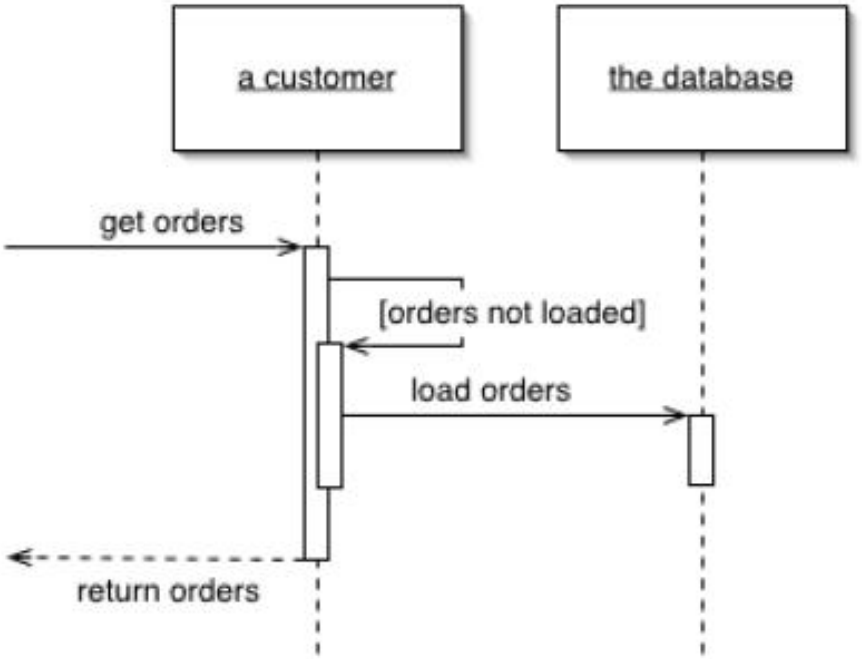
"Structured" methods of the 1970's tended towards a "top-down decomposition", which encouraged high-level modules to depend on modules written at a lower level of abstraction.

we must reverse the direction of these dependencies to avoid *rigidity, fragility* and *immobility*.

# Applying FactoryMethod

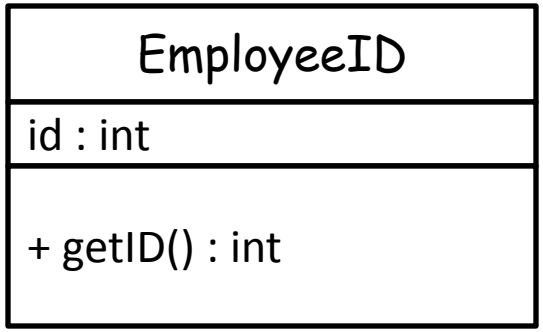


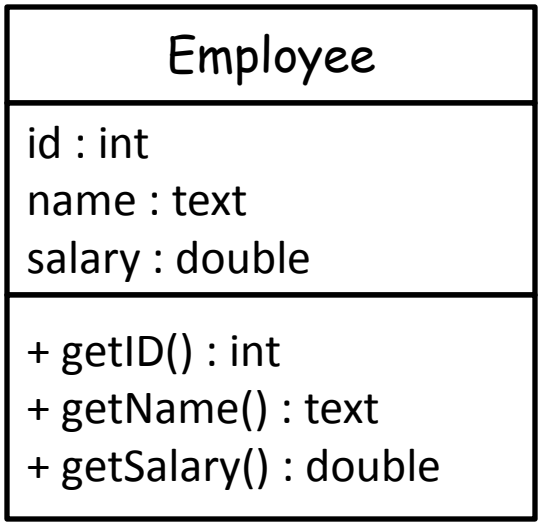
# Identity Map




Ensure each object only gets loaded once by keeping every loaded object in a map. Lookup objects using the map when referring to them.

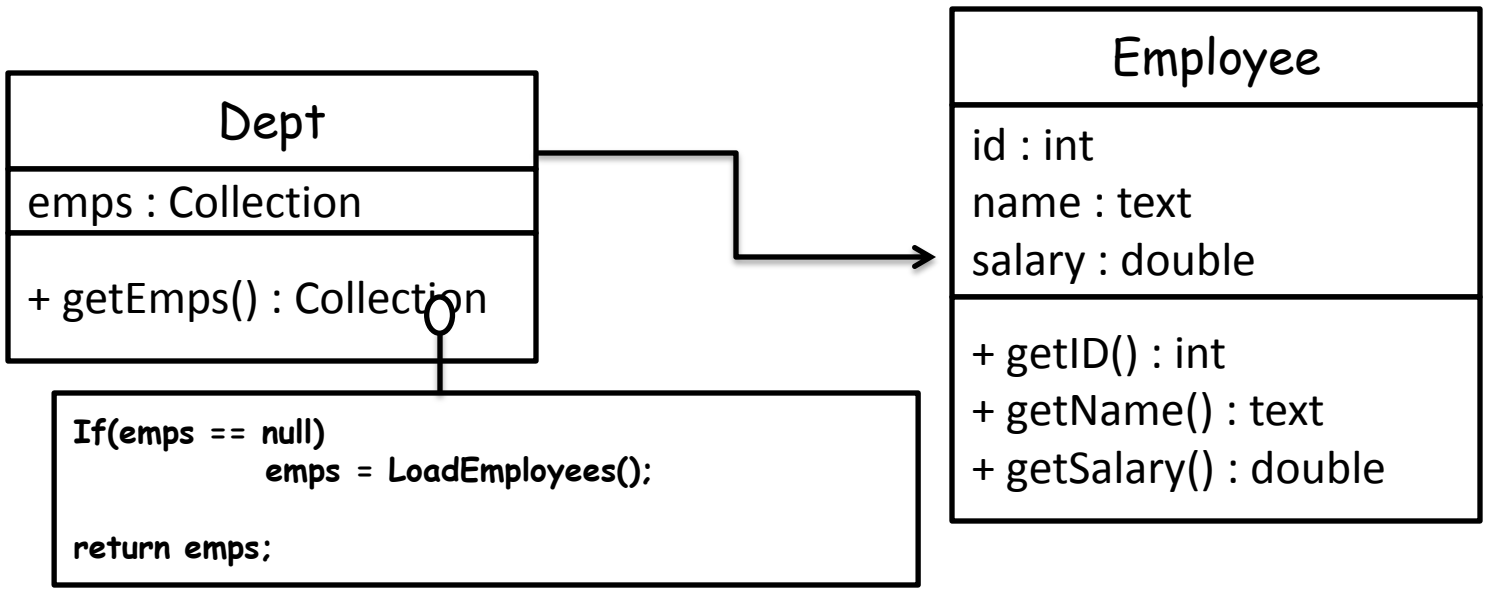
# Identity Field



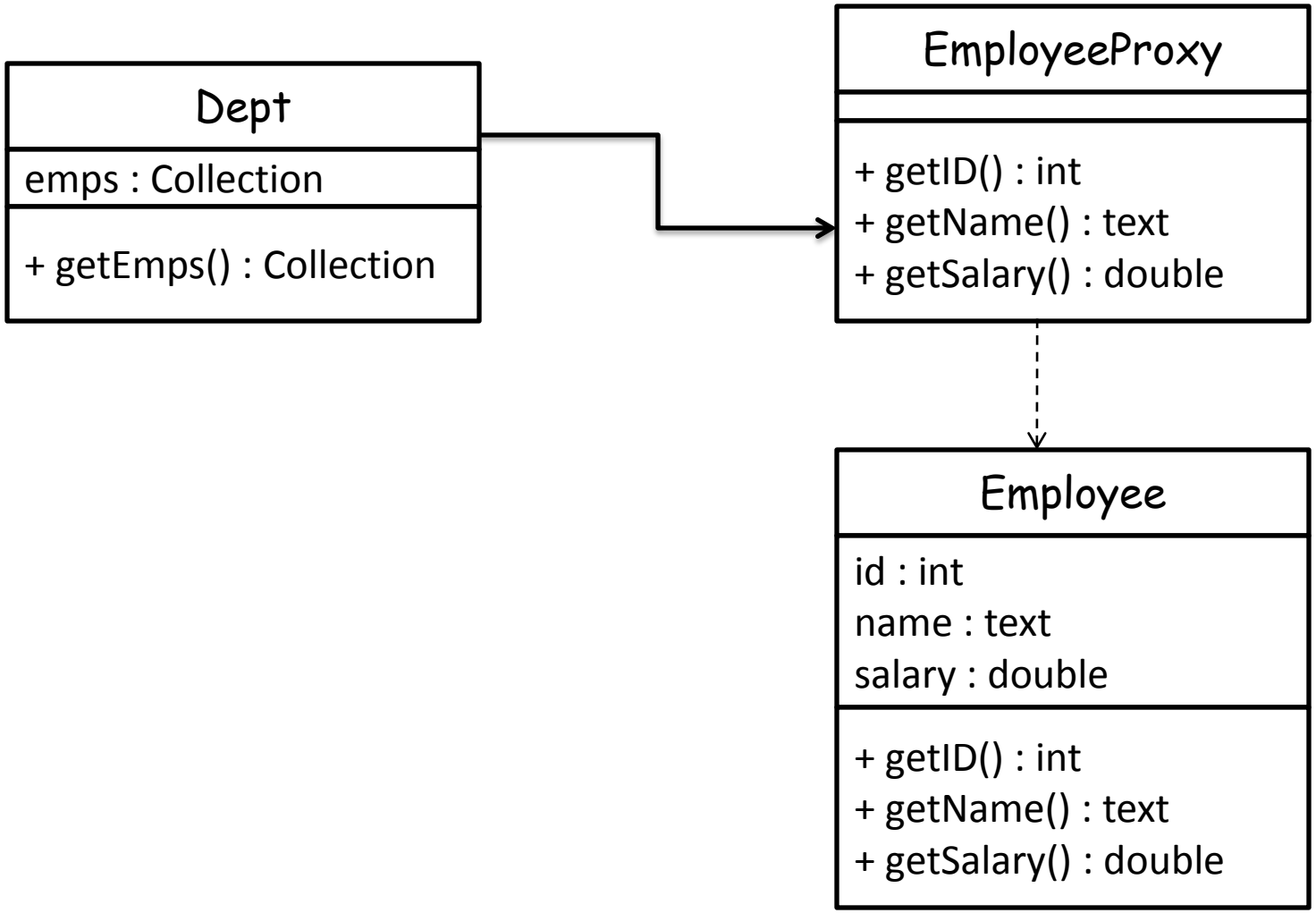

Save a database id field in an object to maintain identity between an in-memory object and a database row.

# Lazy Loading

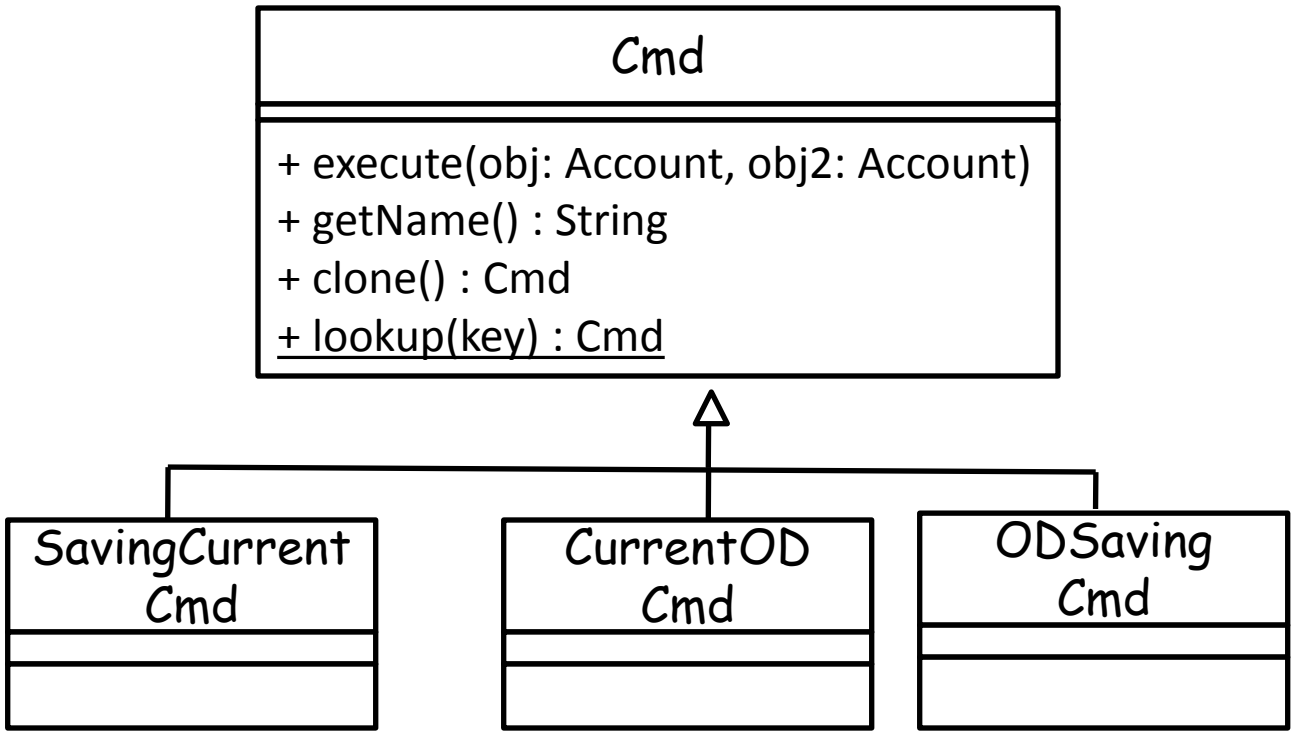



An object that doesn't contain all of the data you need, but knows how to get it.

# Lazy Loading with Proxy

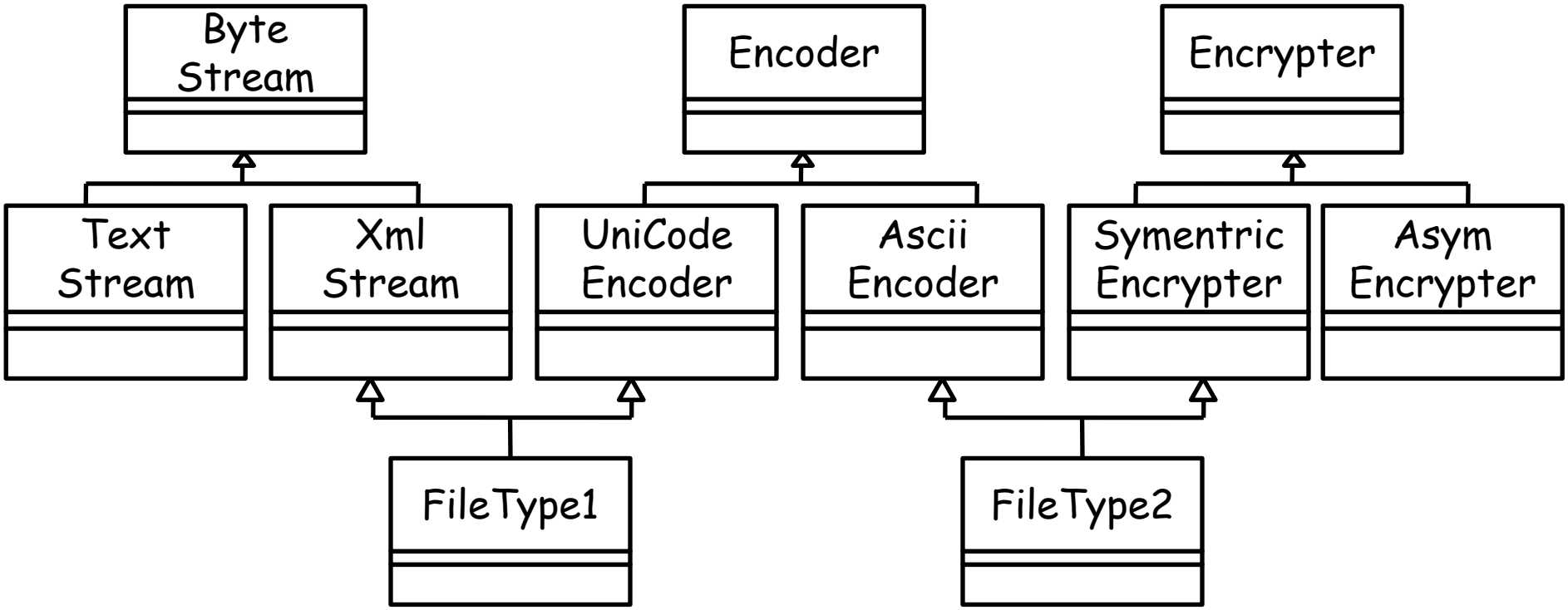


# Applying Prototype

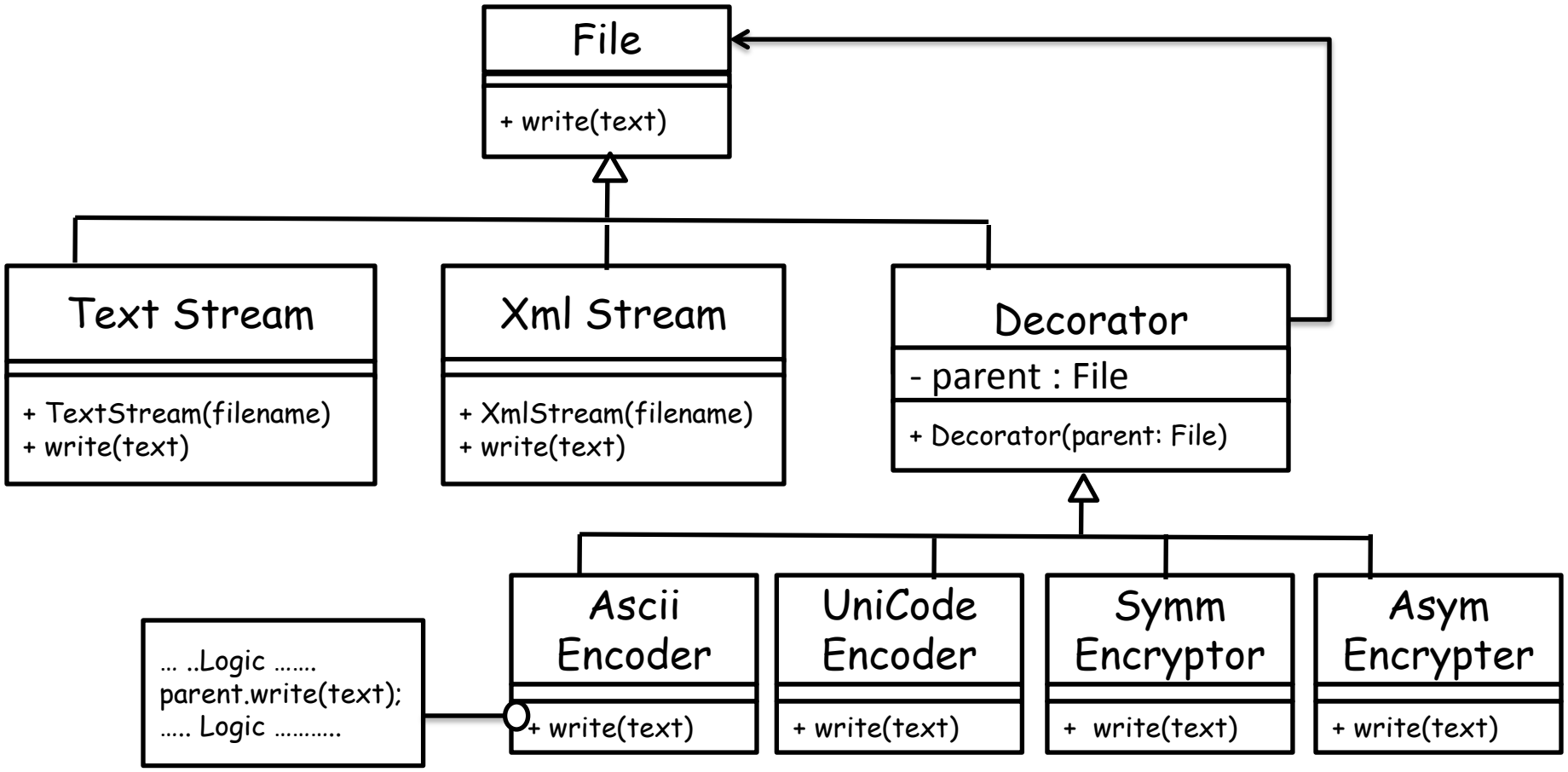



```
GetOffer(Account obj, Account obj2)
{
        String  key = obj.getName() + obj.getName();
        Cmd prototype = Cmd.lookup(key);
        Cmd  cmd = prototype.clone();
        cmd. execute(obj,obj2);
}
```

# Multiple Inheritance problem



# Applying Decorator



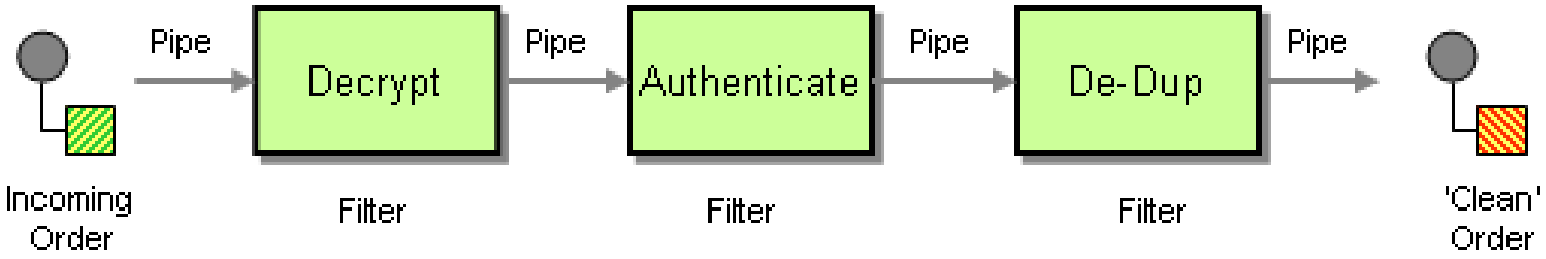

File file = new SymmEncrypter(new AsciiEncoder(new TextStream(“file.txt”)));
file.write(“Hello”);

# Adapter v/s Decorator

With a Decorator, the interfaces of the objects you're composing are the same, while the entire point of an Adapter is to allow you to change interfaces.

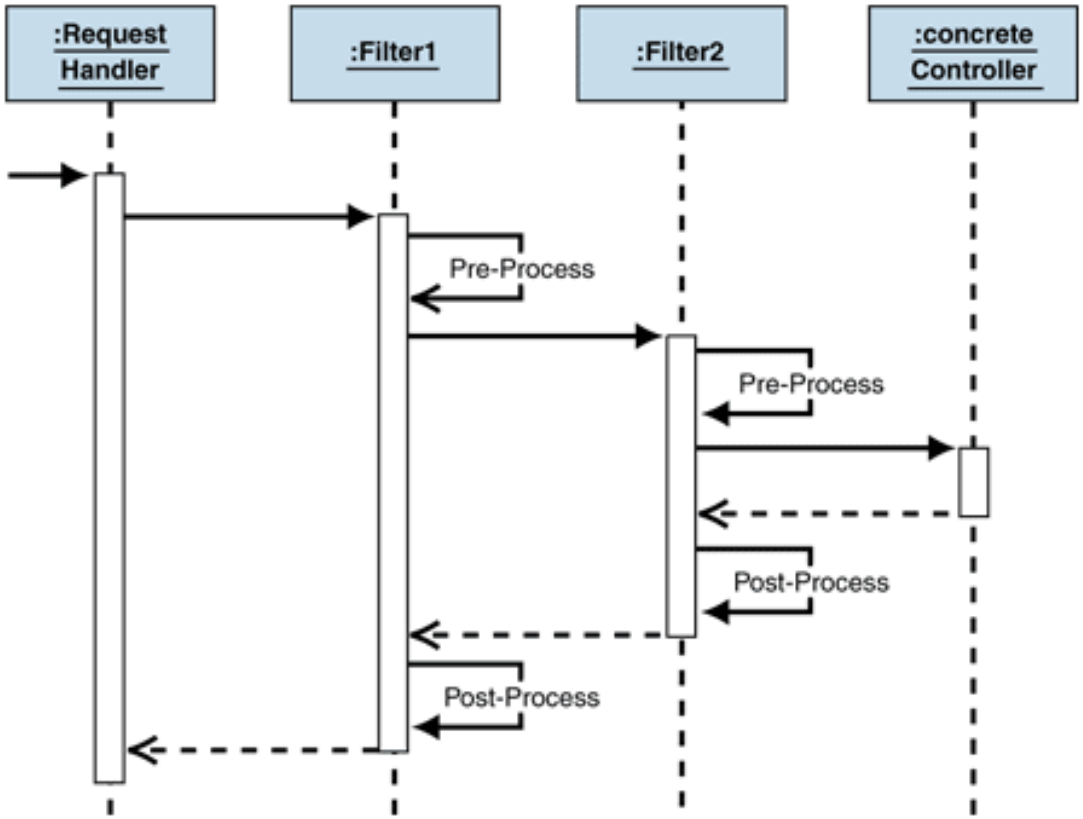
# Pipeline Pattern

**Pipeline** consists of a chain of processing elements , arranged so that the output of each element is the input of the next.




The concept is also called the **pipes and filters** [design pattern](#).

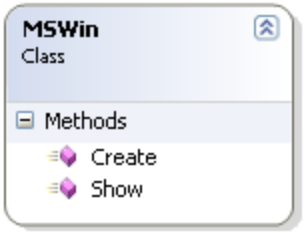
# Intercepting Filter

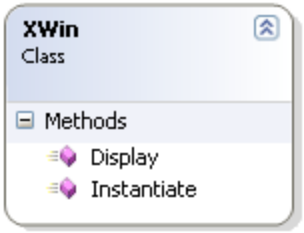


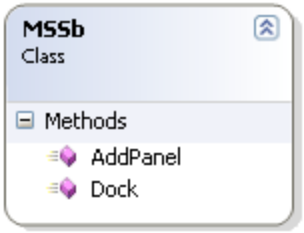

How do you implement common pre- and post-processing steps around Web page requests?

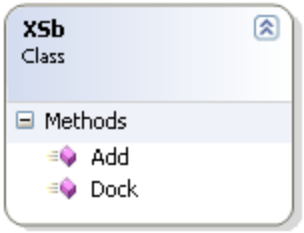
# Toggle between Implementation

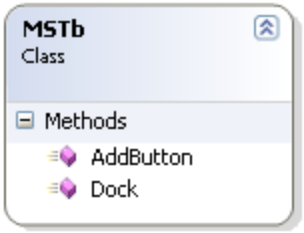
```
MSWin w = new MSWin();
w.Create();
w.Show();

MSTb tb = new MSTb();
Tb.AddButton();
Tb.AddButton();

MSSb sb = new MSSb();
sb.AddPanel();
sb.AddPanel();
sb.Dock();
```

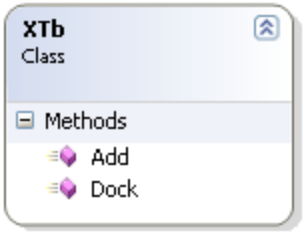


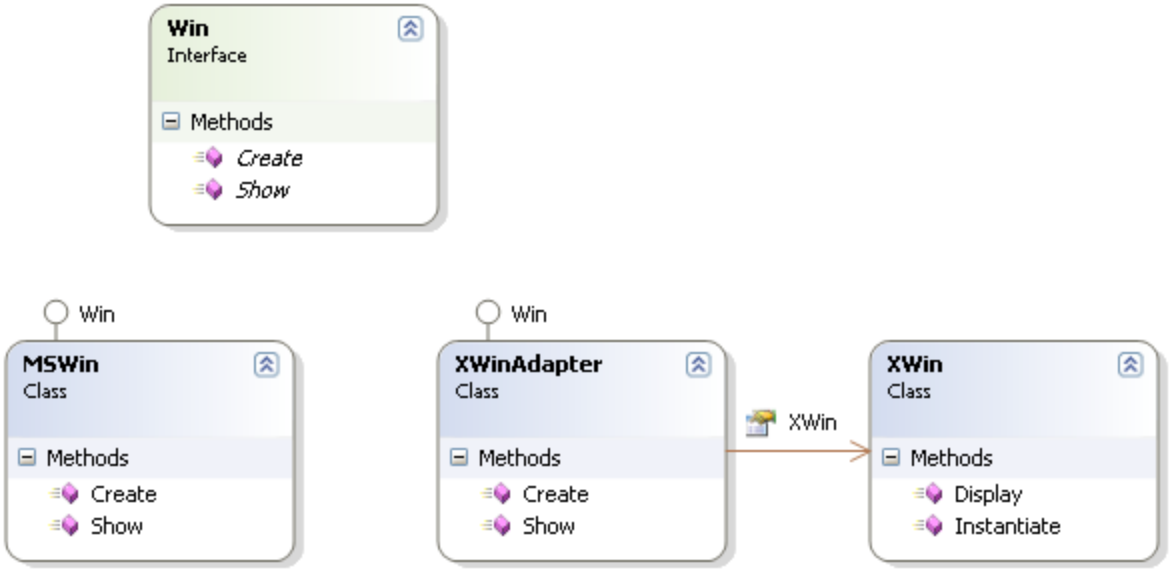



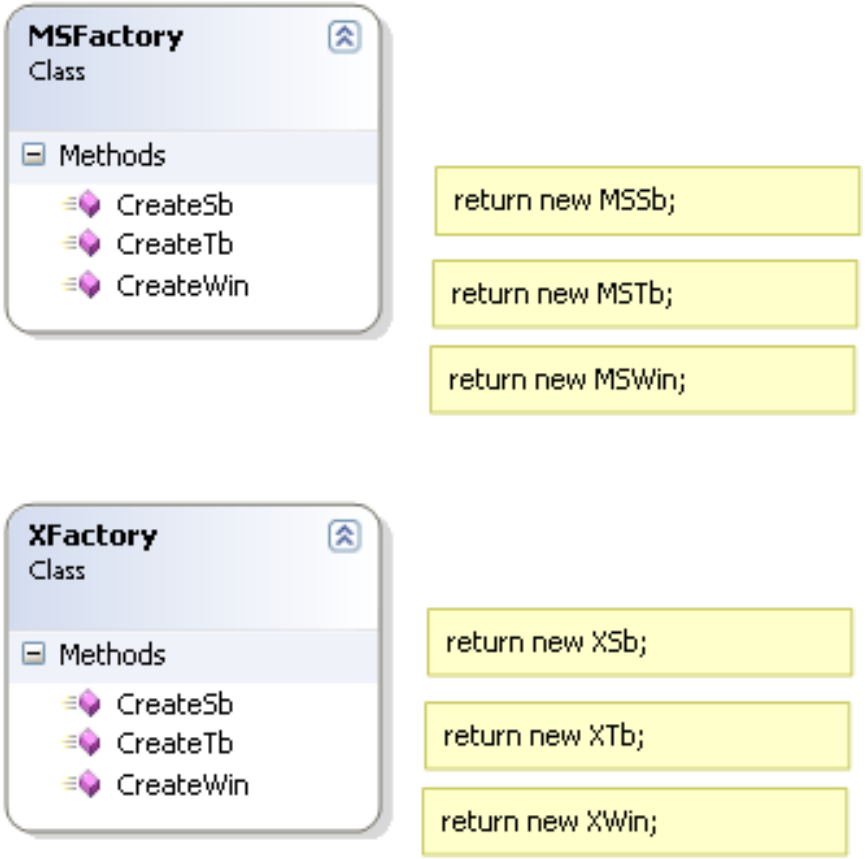
# Applying Adapter

```
Win w = new MSWin();
w.Create();
w.Show();

Tb tb = new MSTb();
Tb.AddButton();
Tb.AddButton();

Sb sb = new MSSb();
sb.AddPanel();
sb.AddPanel();
sb.Dock();
```



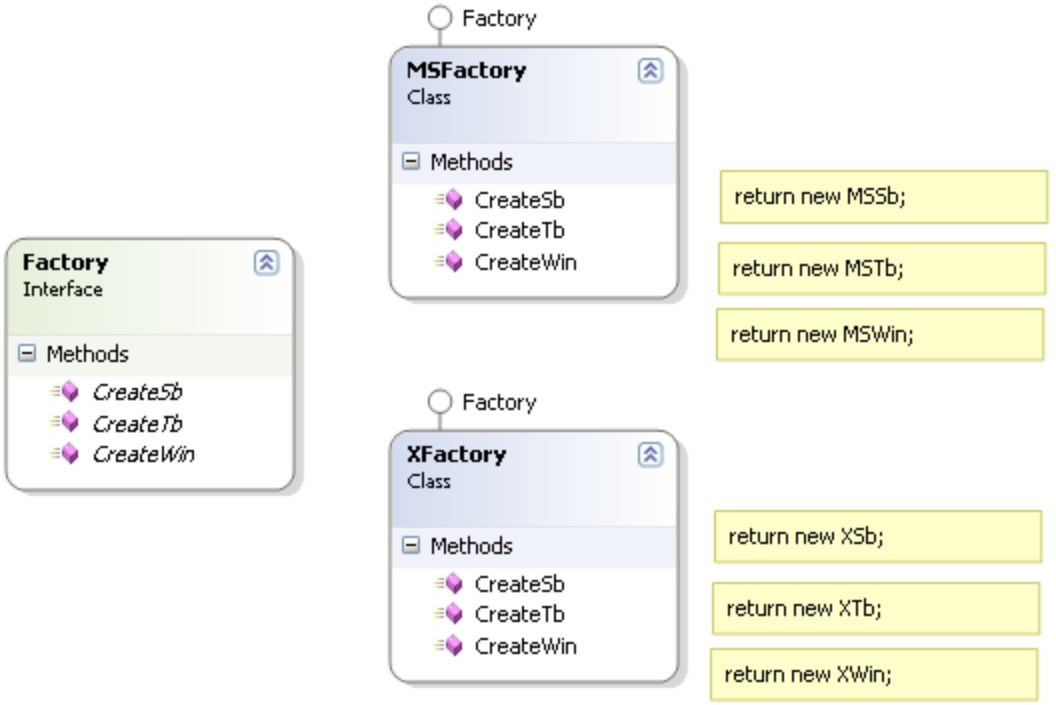
# Applying ClassFactory

```
MSFactory f = new MSFactory();
Win w = f.CreateWin();
w.Create();
w.Show();

Tb tb = f.CreateSb();
Tb.AddButton();
Tb.AddButton();

Sb sb = f.CreateTb();
sb.AddPanel();
sb.AddPanel();
sb.Dock();
```



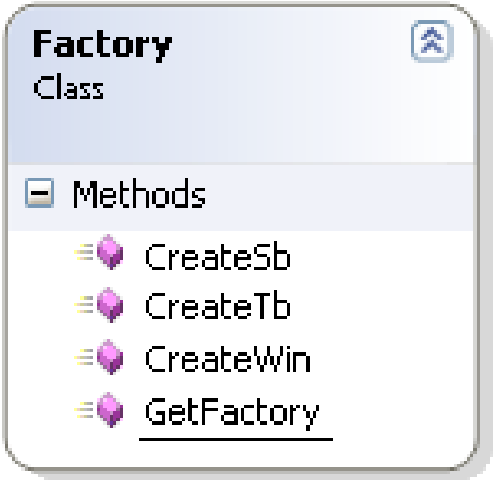
# Applying Abstract Factory

```
Factory f = new MSFactory();
Win w = f.CreateWin();
w.Create();
w.Show();

Tb tb = f.CreateSb();
Tb.AddButton();
Tb.AddButton();

Sb sb = f.CreateTb();
sb.AddPanel();
sb.AddPanel();
sb.Dock();
```



# Applying Creator Method

```
Factory f = Factory.GetFactory(1);
Win w = f.CreateWin();
w.Create();
w.Show();

Tb tb = f.CreateSb();
Tb.AddButton();
Tb.AddButton();

Sb sb = f.CreateTb();
sb.AddPanel();
sb.AddPanel();
sb.Dock();
```




```
if(type == 1)
            return new MSFactory;
if(type == 2)
            return new XFactory;
```

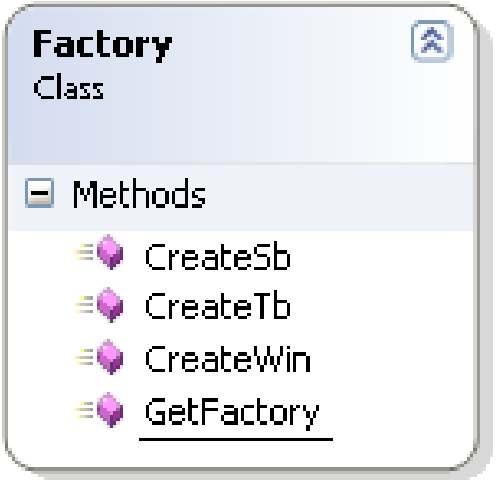
# Protected Variations

- Problem:  How do we design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?

- Solution:  Identify points of predicted variation or instability; assign responsibility to create a stable interface around them.
  - Reading parameters from an external source to change behavior of a system at run time, style sheets, metadata, etc

# Protected Variant

```
Factory f = Factory.GetFactory(GetConfig());
Win w = f.CreateWin();
w.Create();
w.Show();

Tb tb = f.CreateSb();
Tb.AddButton();
Tb.AddButton();

Sb sb = f.CreateTb();
sb.AddPanel();
sb.AddPanel();
sb.Dock();
```




```
Factory* f= lookup(type);
return f;
```

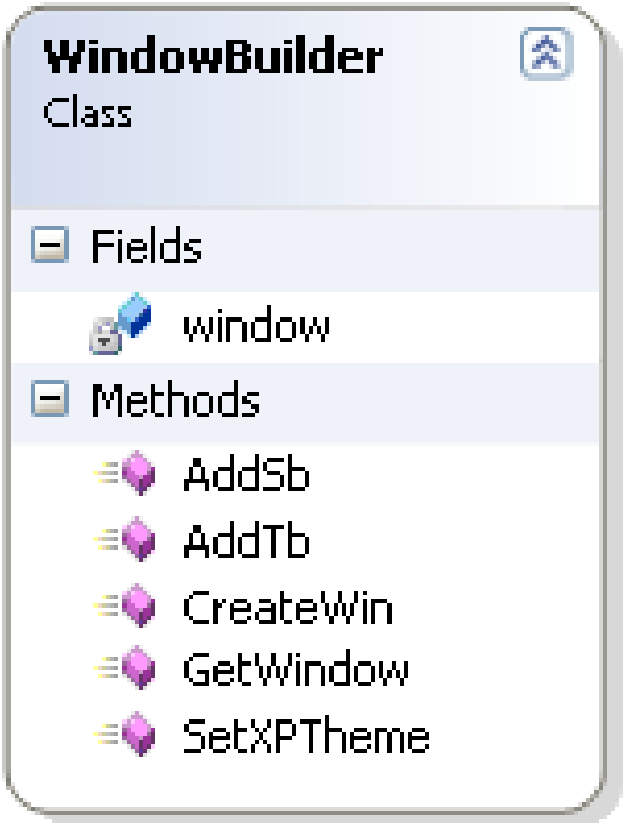
# Singleton



```
if(m_instamce == null)
            m_instance = new MSFactory;

 return m_instance;
```

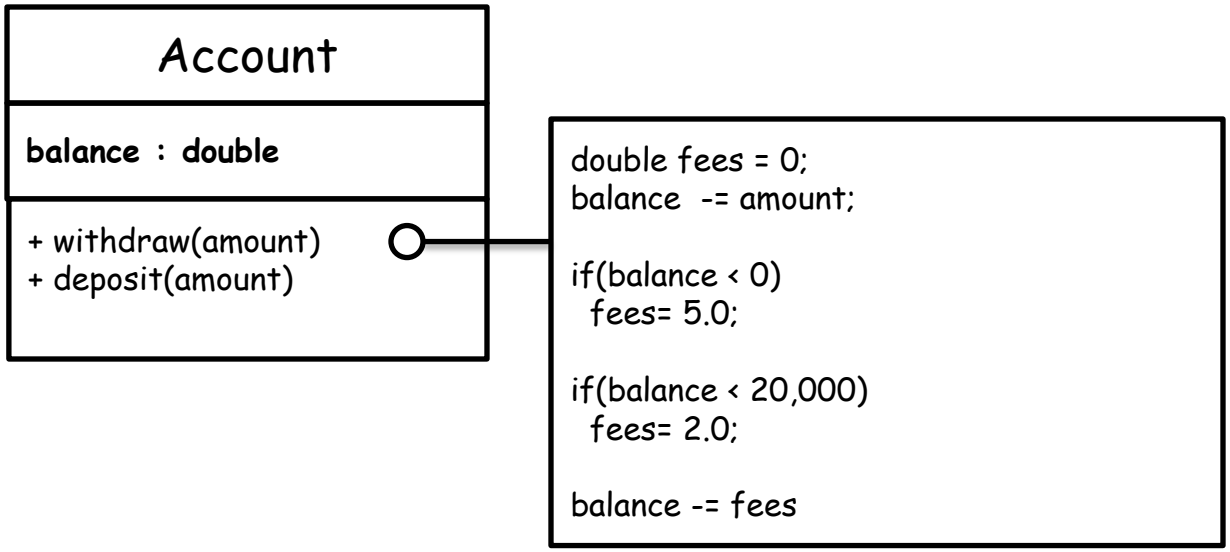
# Builder

```
WindowBuilder builder = new WindowBuilder();
builder.CreateWin();
builder.AddTb();
builder.AddSb();
builder.SetXPTheme();
Win w = builder.GetWindow();
```

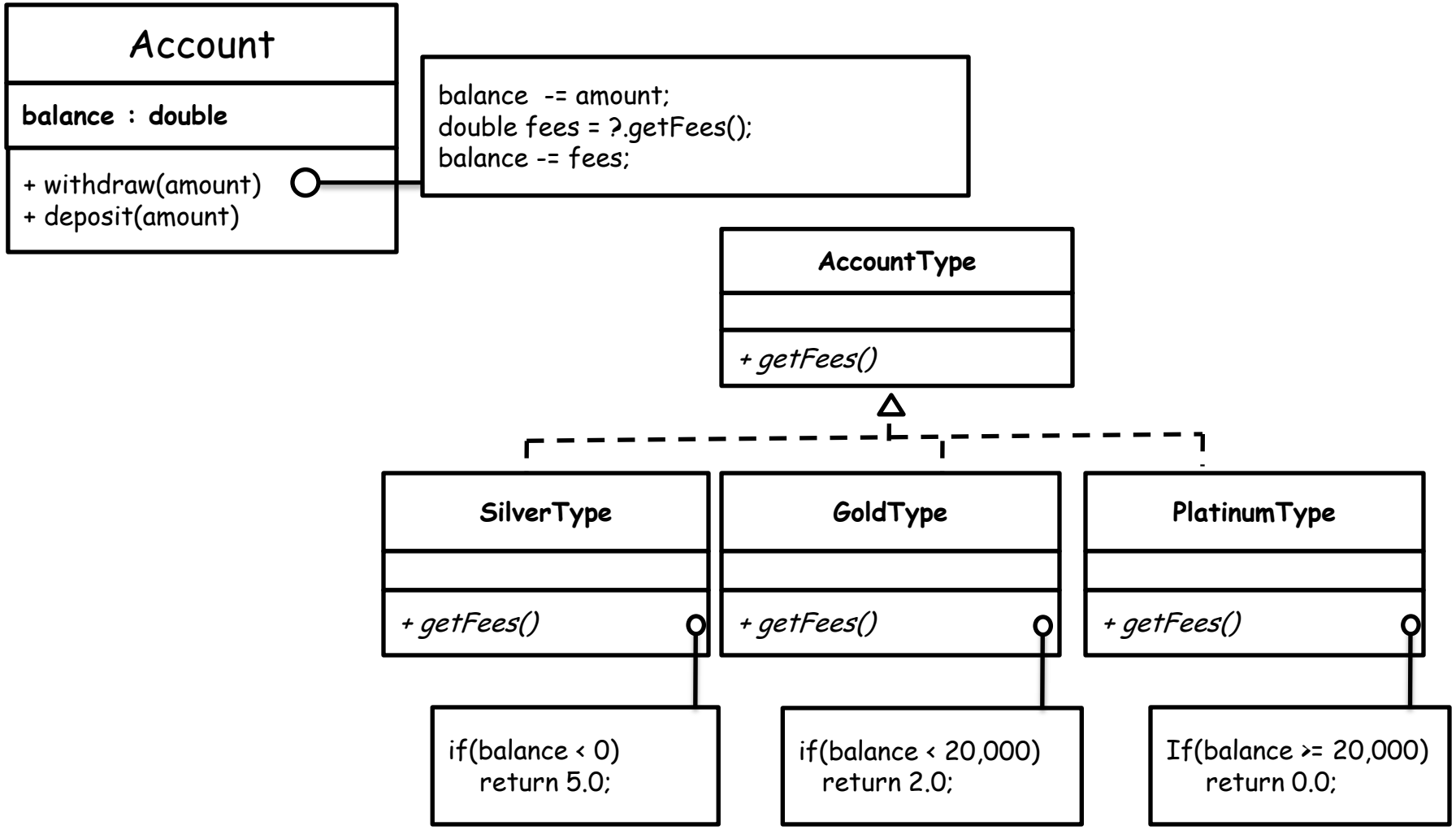



Abstract steps of construction of objects

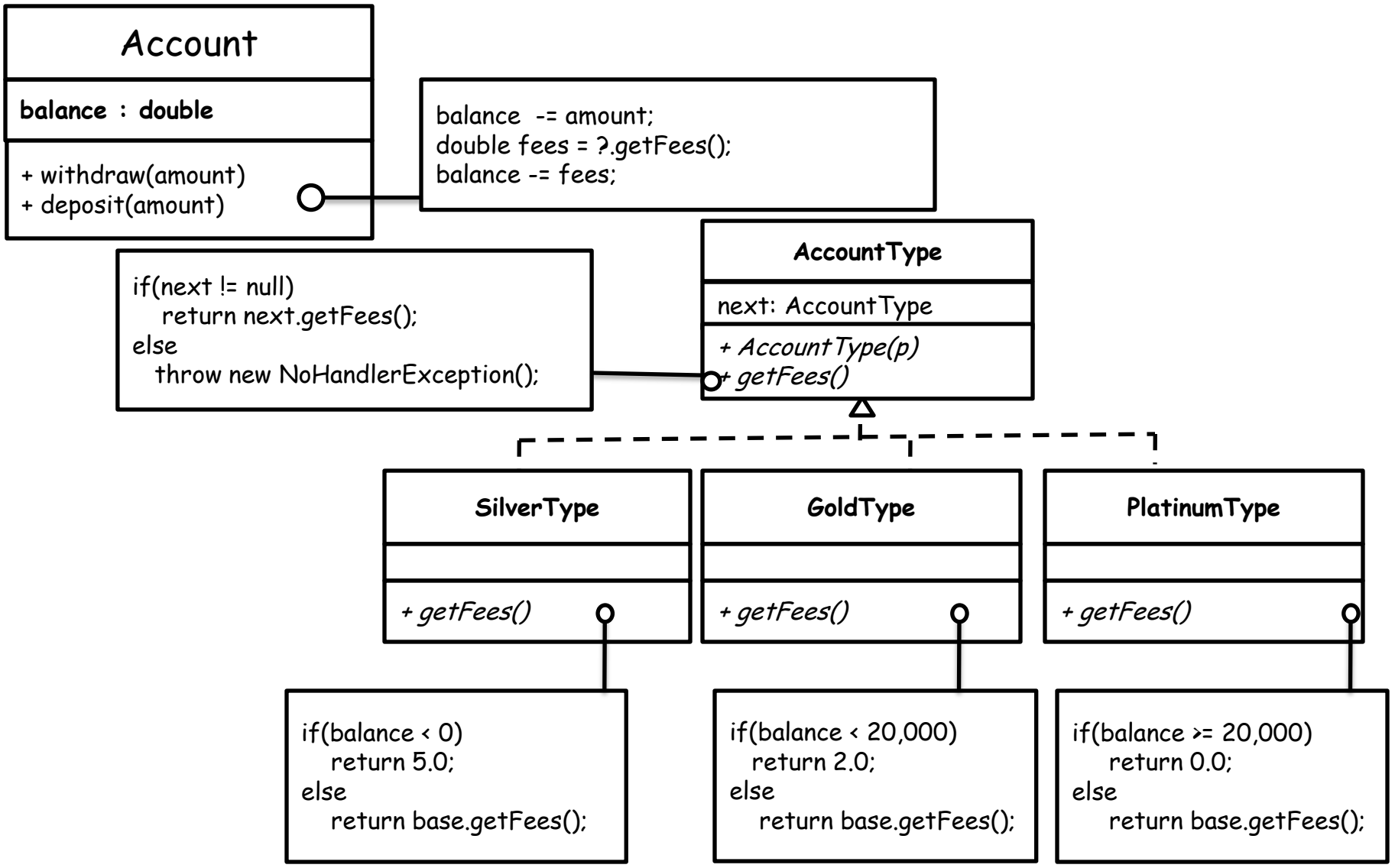
# Problem


Account
balance : double
+ withdraw(amount)
+ deposit(amount)
double fees = 0;
balance -= amount;
if(balance < 0)
fees= 5.0;
if(balance < 20,000)
fees= 2.0;
balance -= fees

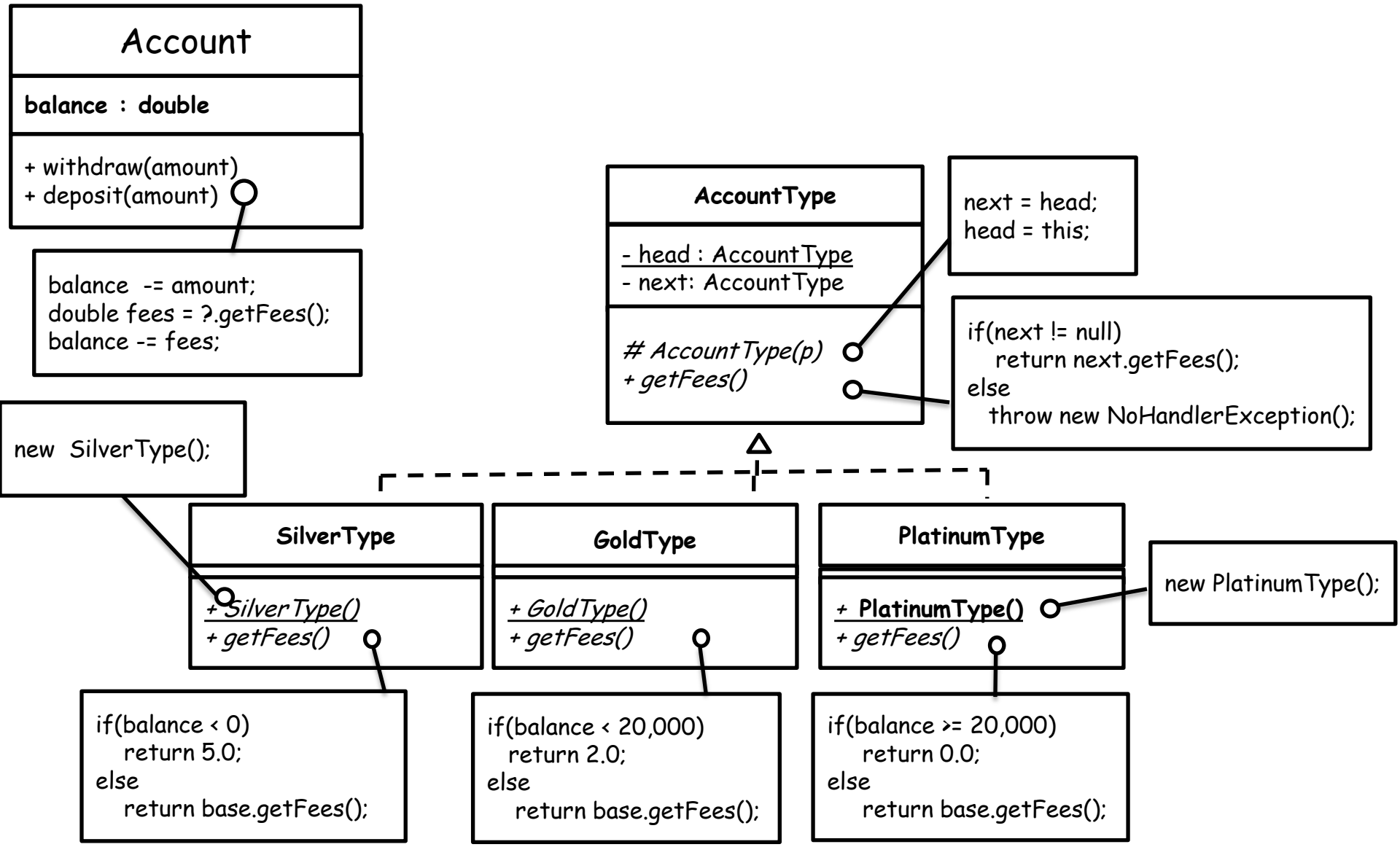
# Moving Conditions Horizontally



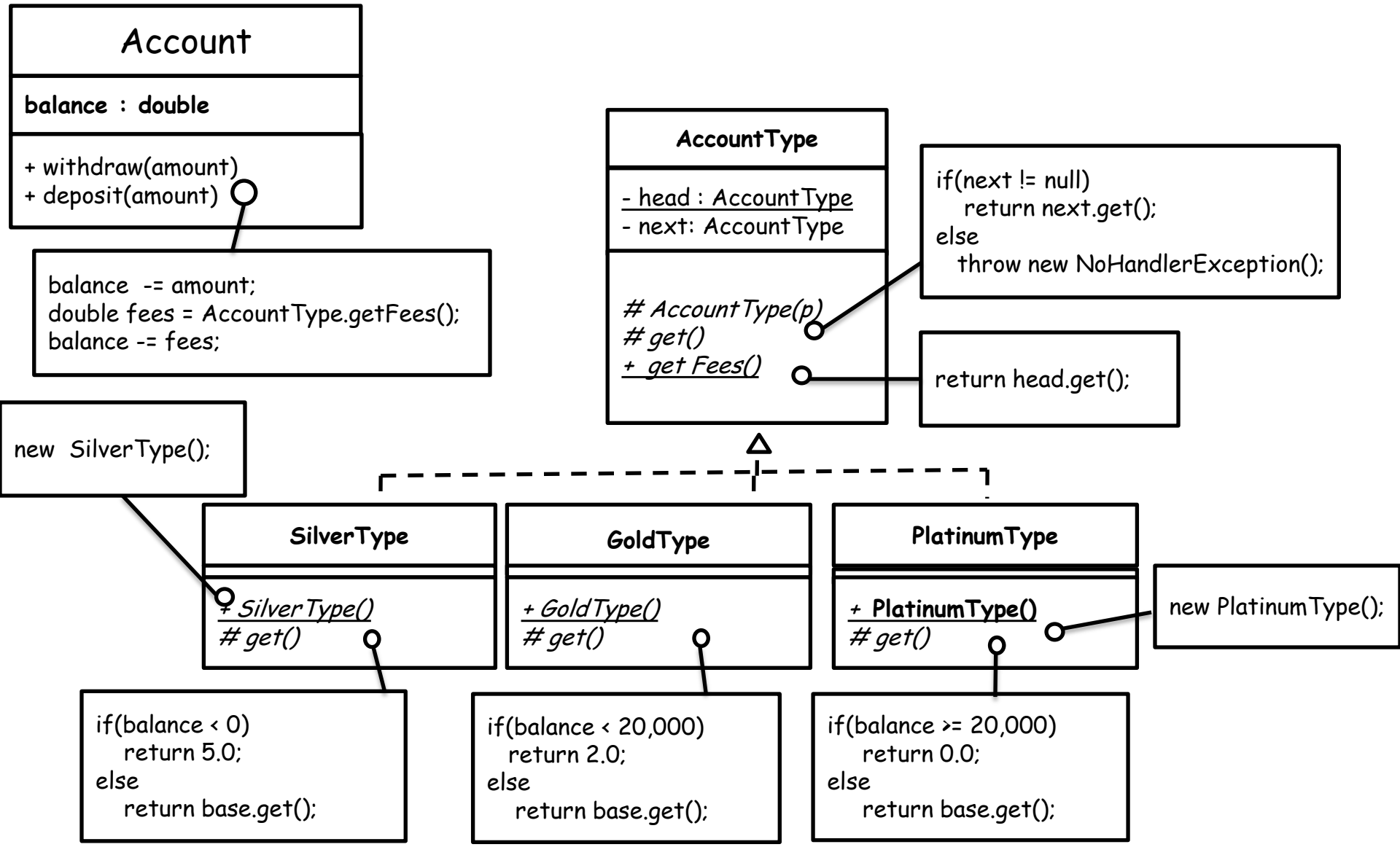
# Applying Chain of Responsibility



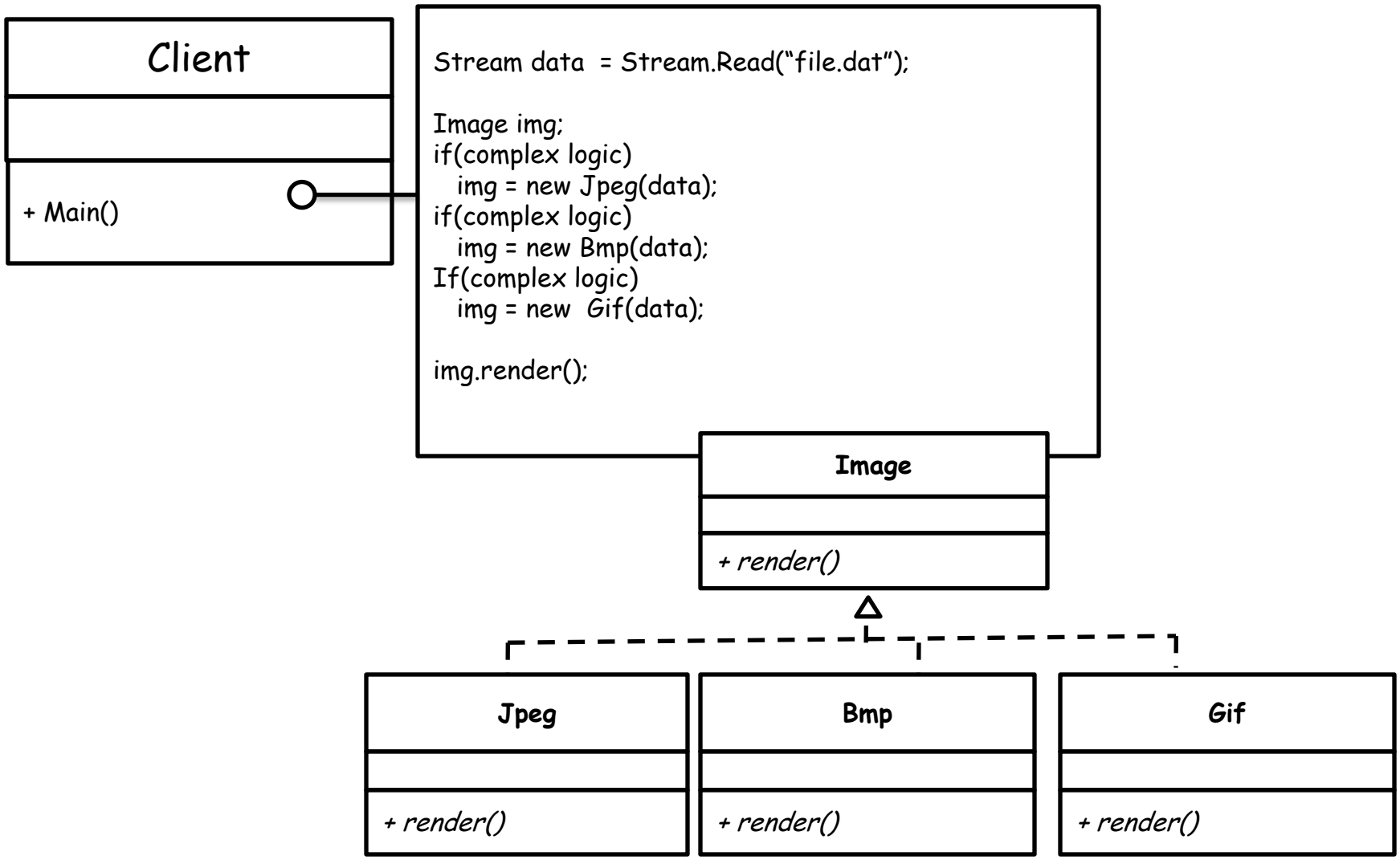
# Applying Static Constructor



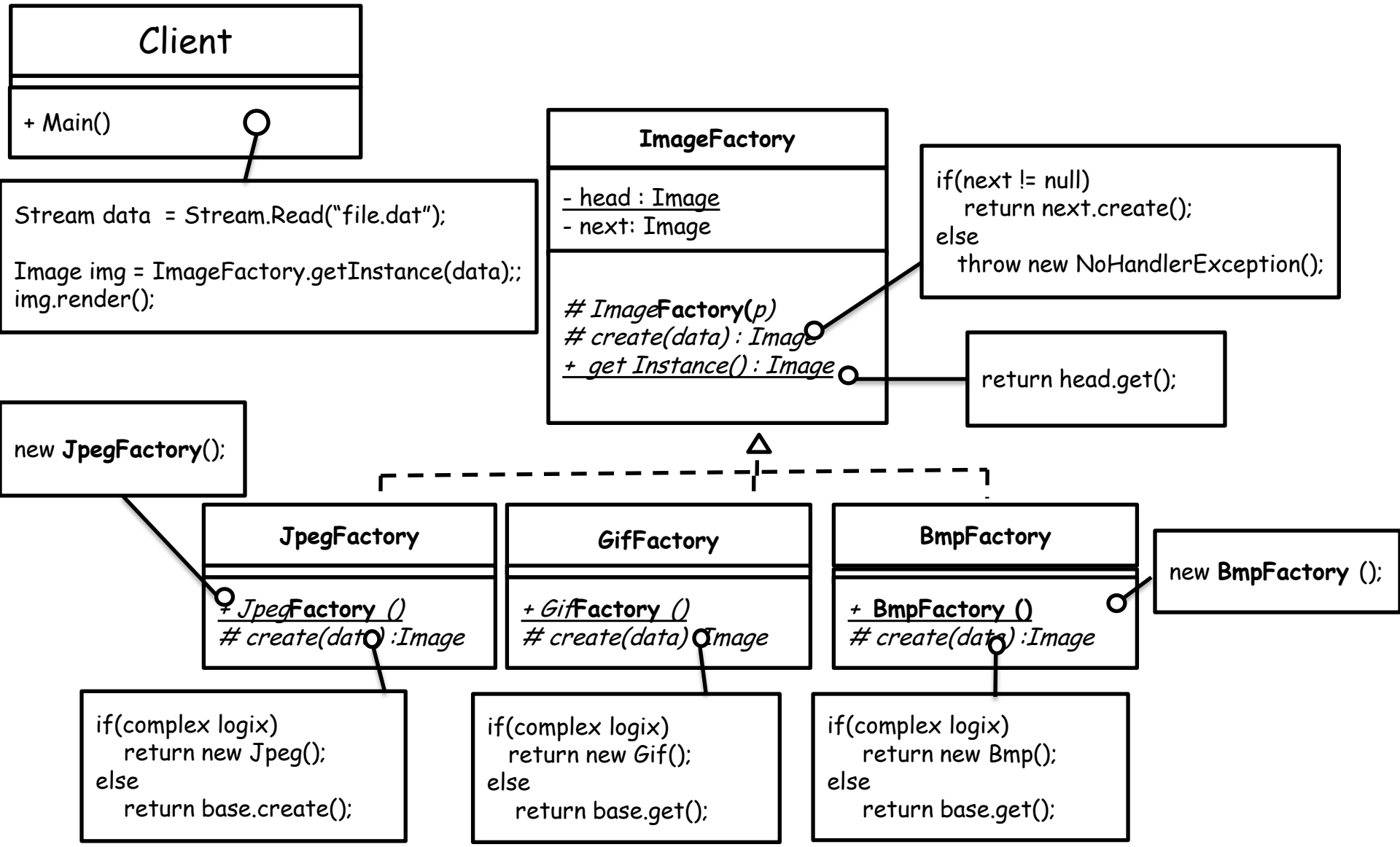
# Applying Creator Method



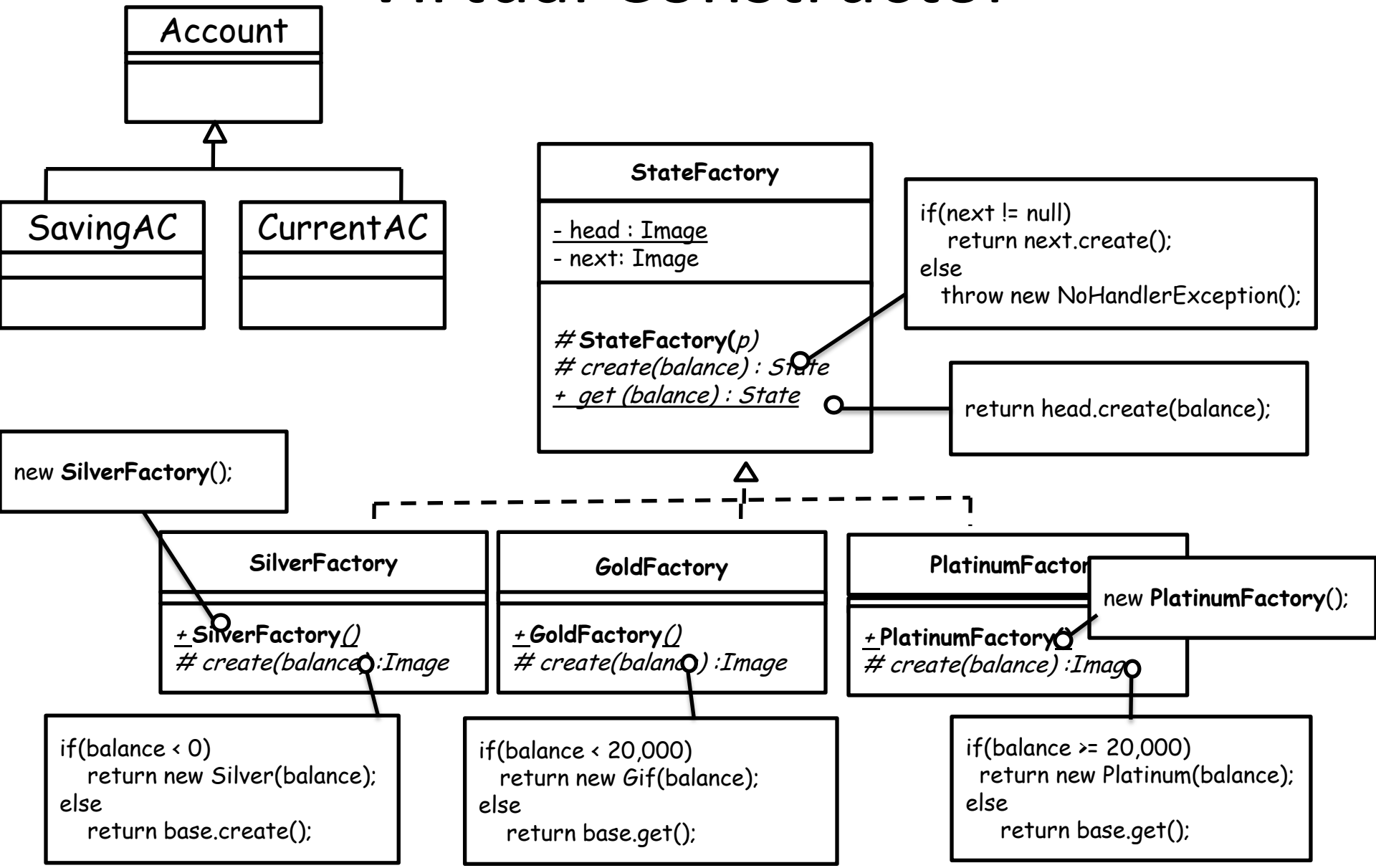
# Virtual Constructor



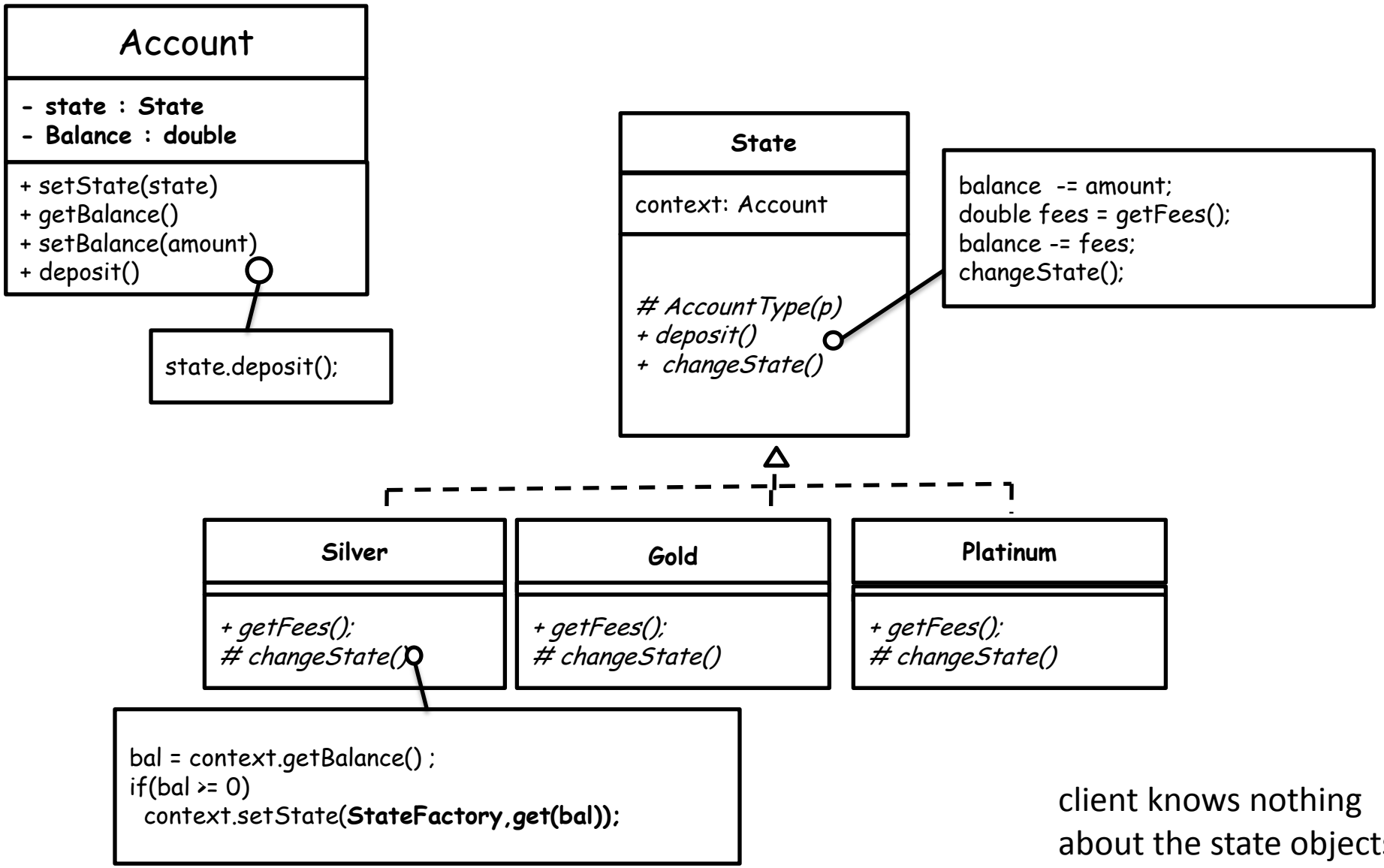
# Virtual Constructor



# Virtual Constructor



# Applying State




client knows nothing about the state objects

# Inversion of Control

IoC provides services through which a component can access its dependencies and services for interacting with the dependencies throughout their life. IoC can be decomposed into two subtypes:

1. Dependency Injection
2. Dependency Lookup.

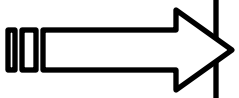
**Dependency Lookup:** With lookup a component must acquire a reference to a dependency. Dependency Lookup comes in two types:

1. Dependency Pull
2. Contextualized Dependency Lookup (CDL).

**Dependency Injection:** The dependencies are literally injected into the component by the IoC container. Injection has two common flavors:

1. Constructor Dependency Injection
2. Setter Dependency Injection.

# Dependency Lookup

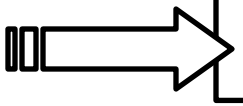


**Dependency Pull**
**Contextualized Dependency Lookup (CDL).**

Dependencies are pulled from a registry as required.

```
public static void main(String[] args) throws Exception {

        // get the bean factory
        BeanFactory factory = getBeanFactory();

MessageRenderer mr = (MessageRenderer)
                              factory.getBean("renderer");
       mr.render();
}
```

# Dependency Lookup



**Dependency Pull**
**Contextualized Dependency Lookup (CDL).**

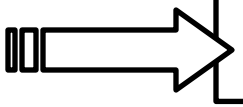
- Lookup is performed against the container that is managing the resource, not from some central registry.

- CDL works by having the component implement an interface.

- By implementing this interface, a component is signaling to the container that it wishes to obtain a dependency.

```
public interface ManagedComponent
{
        public void performLookup(Container container);
}
```

# Dependency Lookup



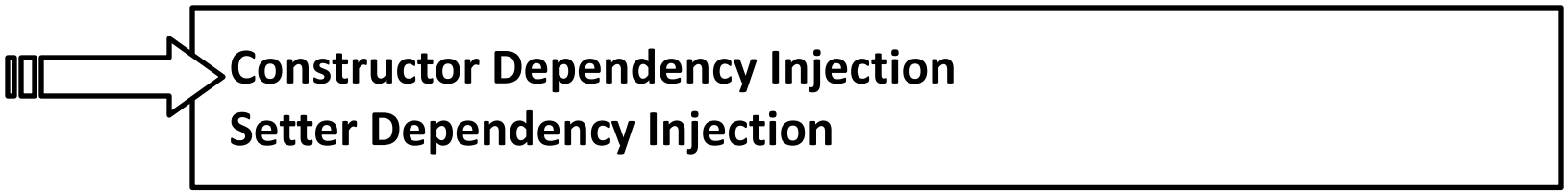
**Dependency Pull**
**Contextualized Dependency Lookup (CDL).**

- When the container is ready to pass dependencies to a component, it calls performLookup() on each component in turn.

- The component can then look up its dependencies using the Container interface.

```
public class MyBean implements ManagedComponent
 {
   private Dependency dep;

   public void performLookup(Container container){
       this.dep = (Dependency)
                    container.getDependency("myDependency");
   }
}
```

# Dependency Injection




The component declares a constructor or a set of constructors taking as arguments its dependencies, and the IoC container passes the dependencies to the component when it instantiates it.

Constructor injection is particularly useful when you absolutely must have an instance of the dependency class before your component is used.

```
public class MyBean
{
        private Dependency dep;

        public MyBean(Dependency dep)
        {
                this.dep = dep;
        }
}
```

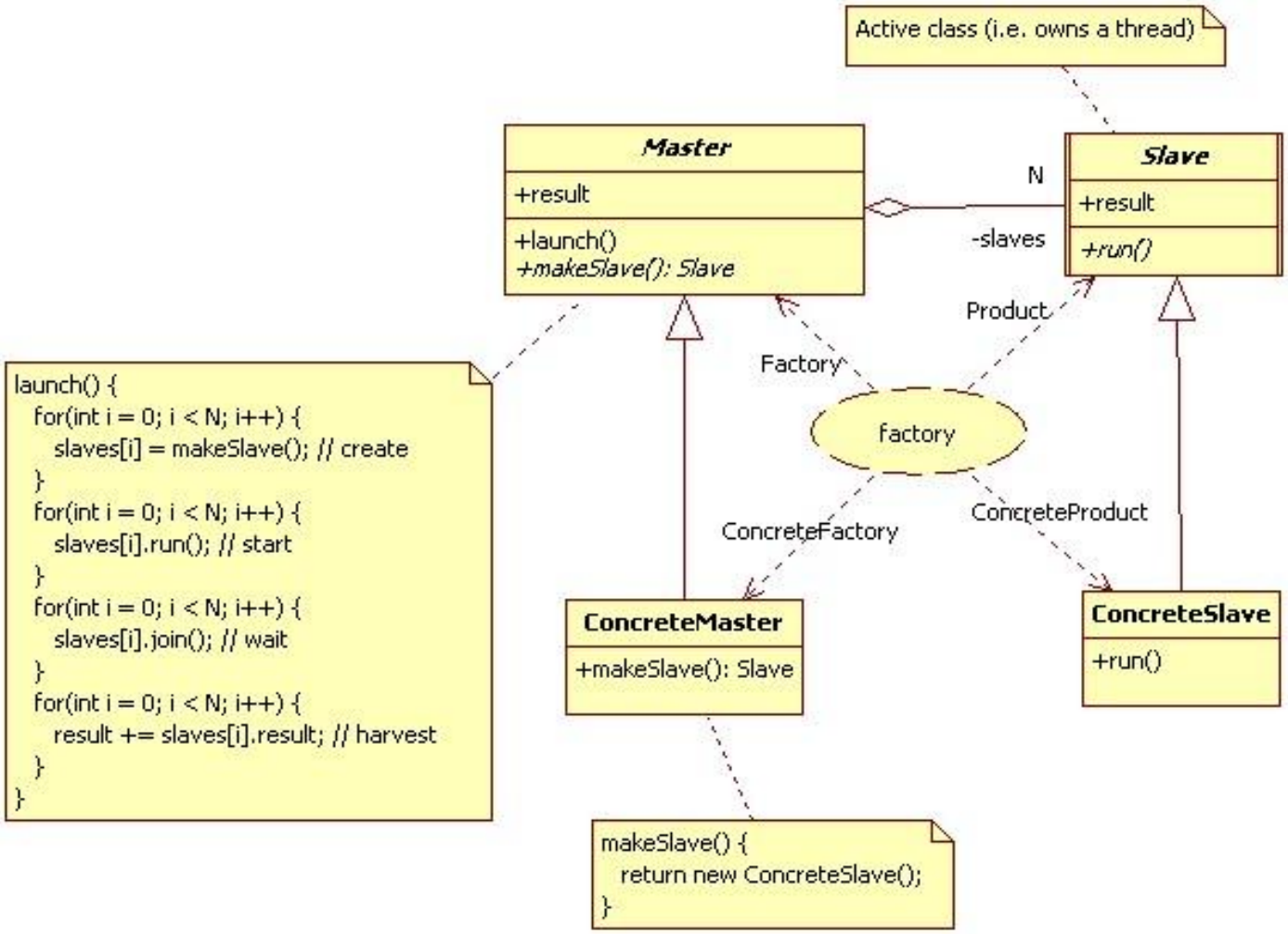
# Dependency Injection


The IoC container injects a component's dependencies into the component via setter methods.

In practice, setter injection is the most widely used injection mechanism, and it is one of the simplest IoC mechanisms to implement.

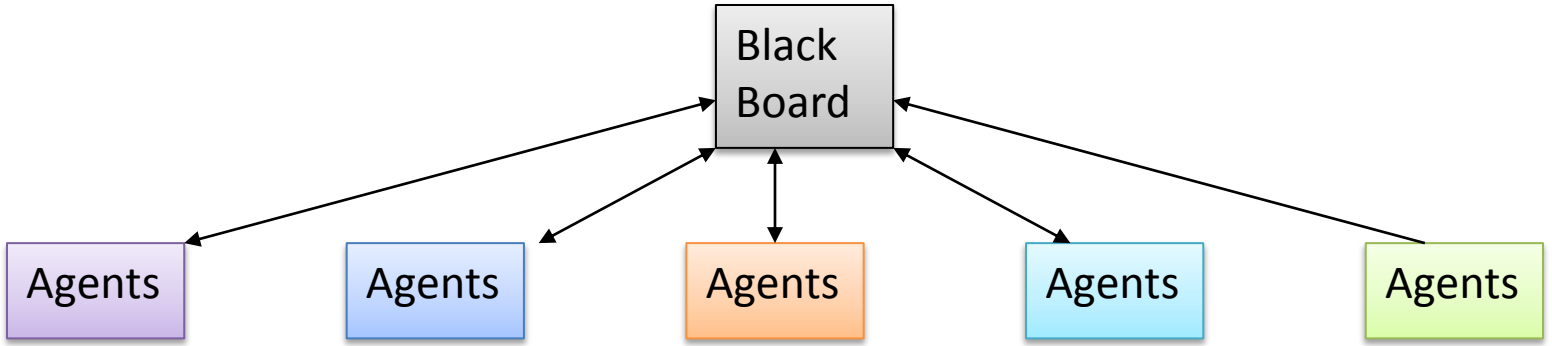
```
public class MyBean
{
	private Dependency dep;

	public void setMyDependency(Dependency dep) {
		this.dep = dep;
	}
}
```

# Master Slave Pattern



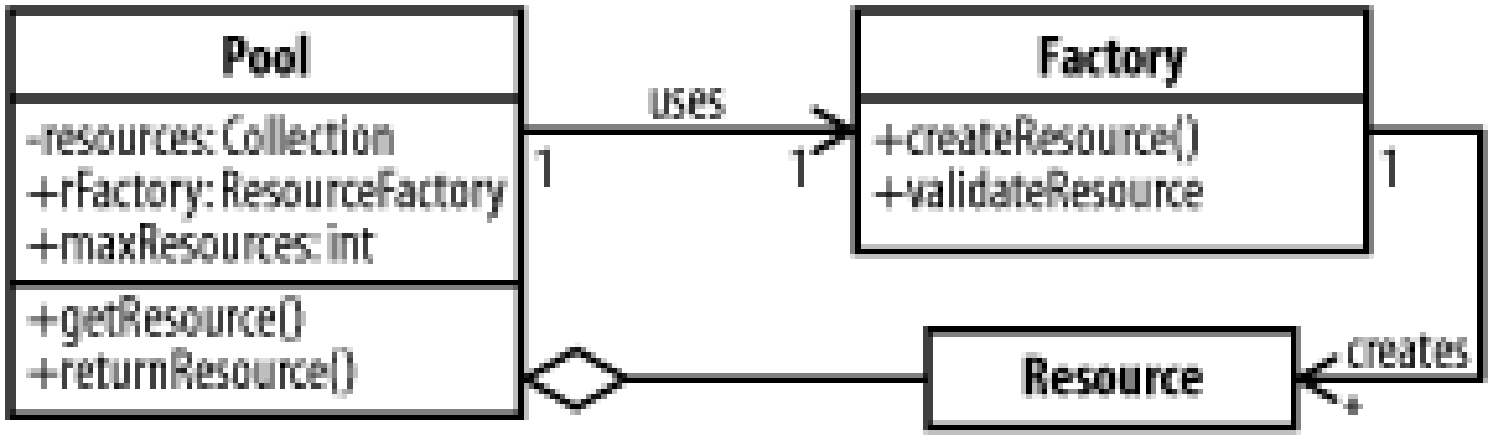
# Blackboard Pattern

A blackboard is a repository of messages which is readable and writable by all processes.



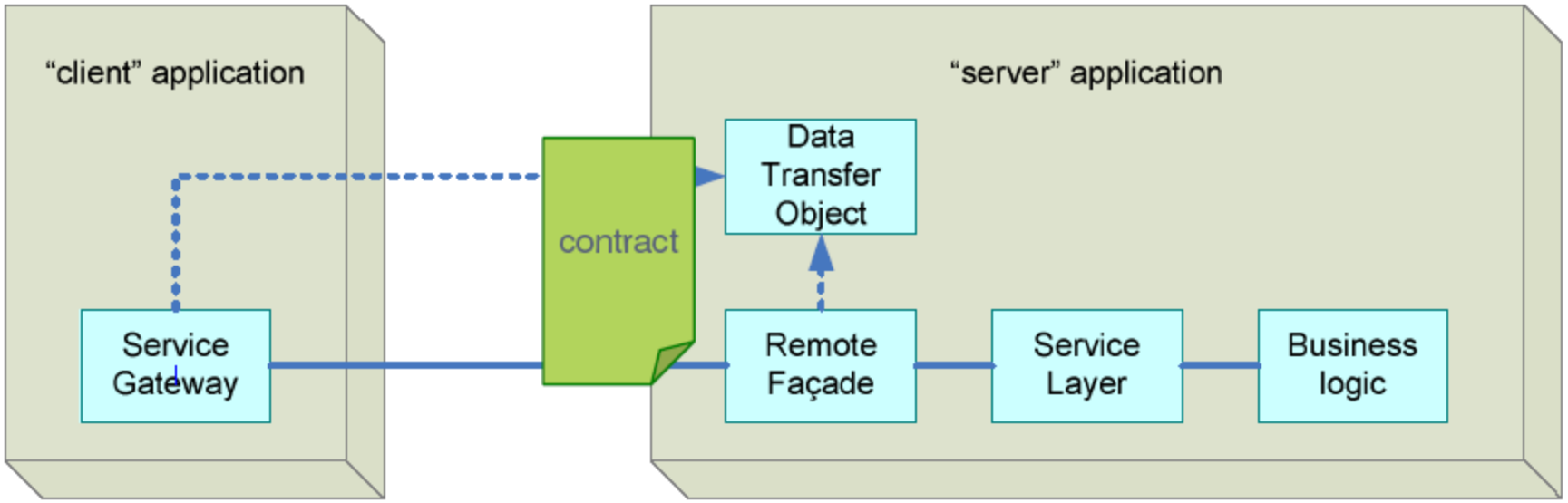

A process which posts an announcement to the blackboard has no idea whether zero, one, or many other processes are paying attention to its announcements.

# Resource Pool

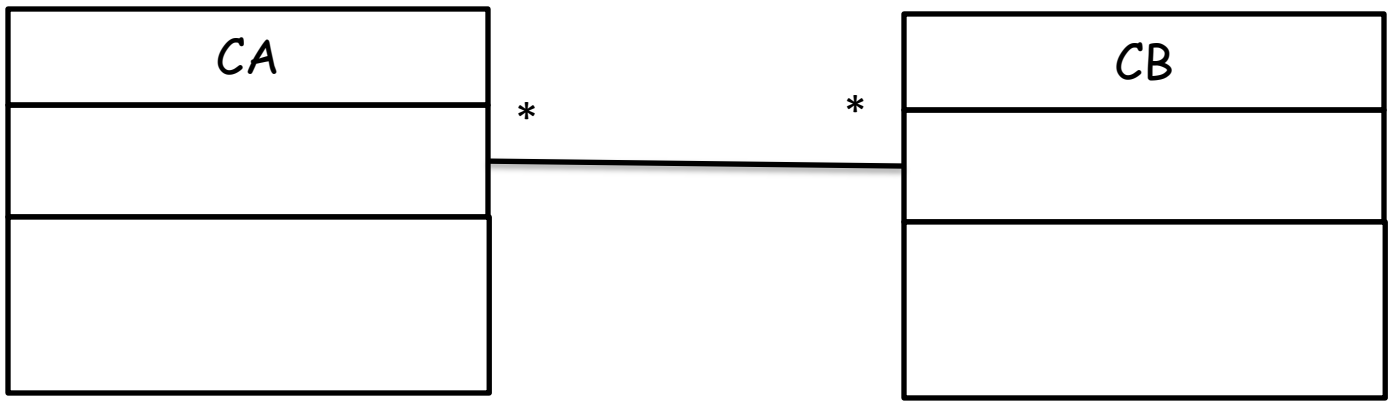



Pools show the most benefits for objects like database connections and threads that have high startup costs.

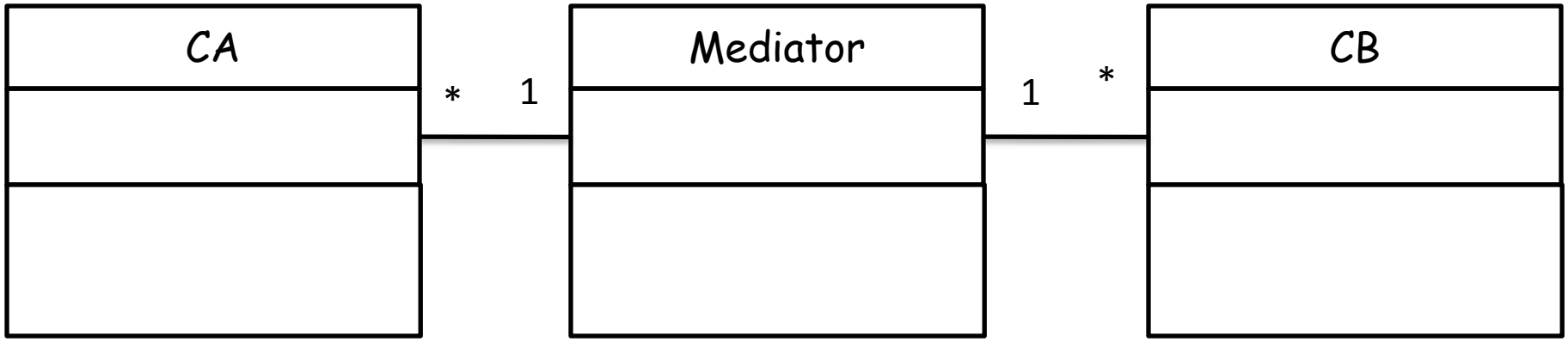
# Patterns for Distributed applications



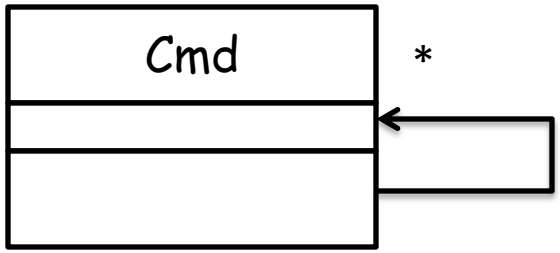
# Problem



# Applying Mediator

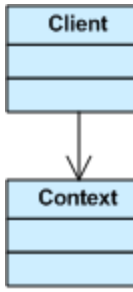


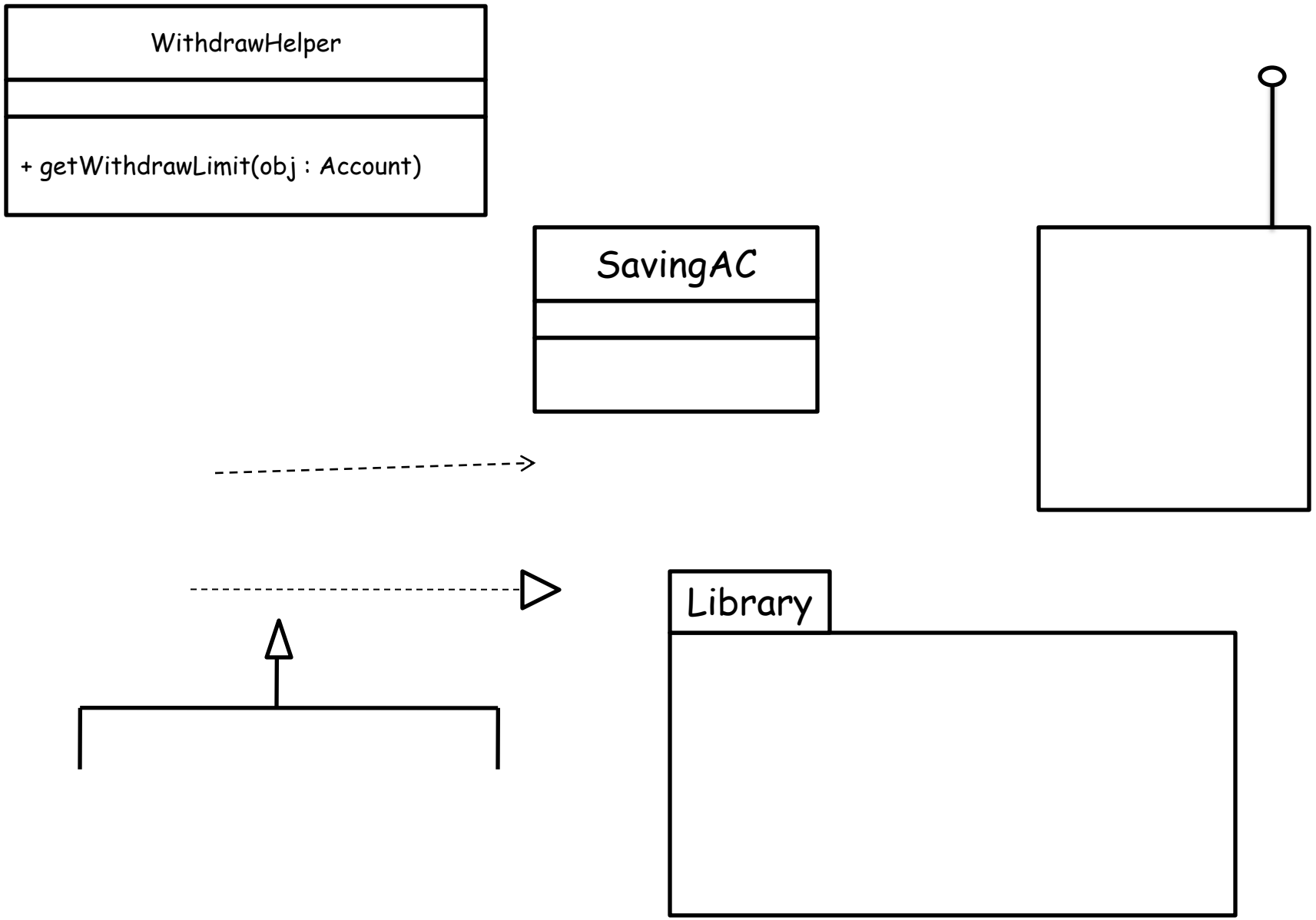
# Composite Pattern



# Interpreter

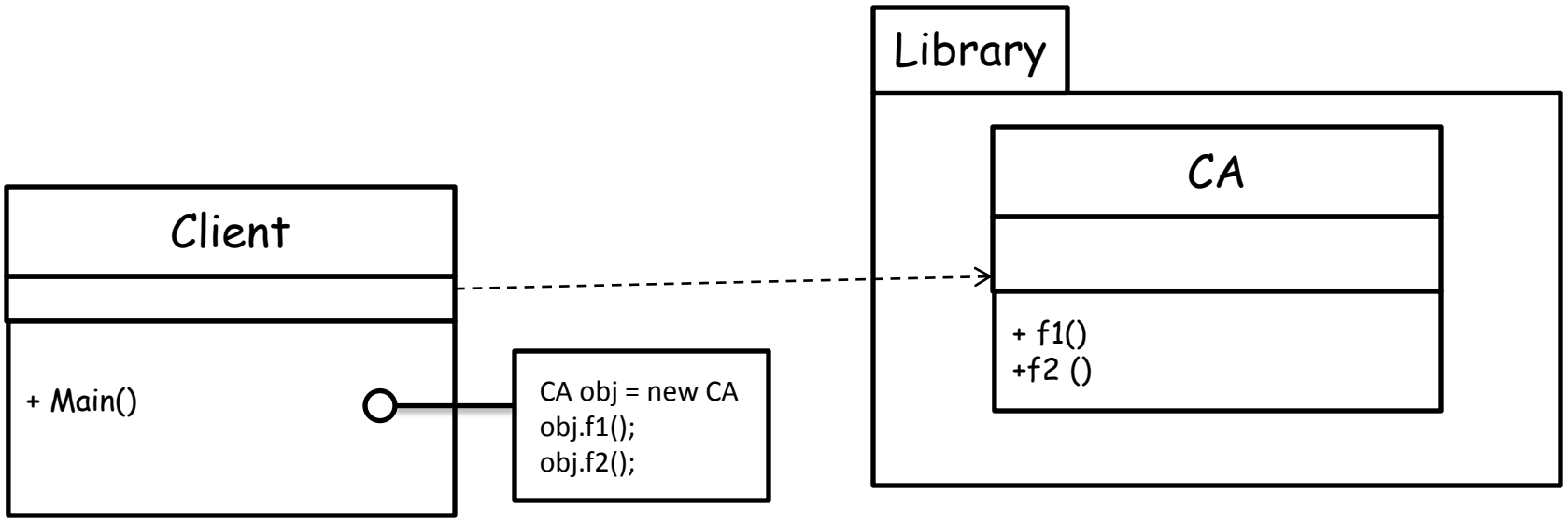
- Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight.
- The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).
- The pattern doesn't address parsing.
- Interpreter can be used in rules engines



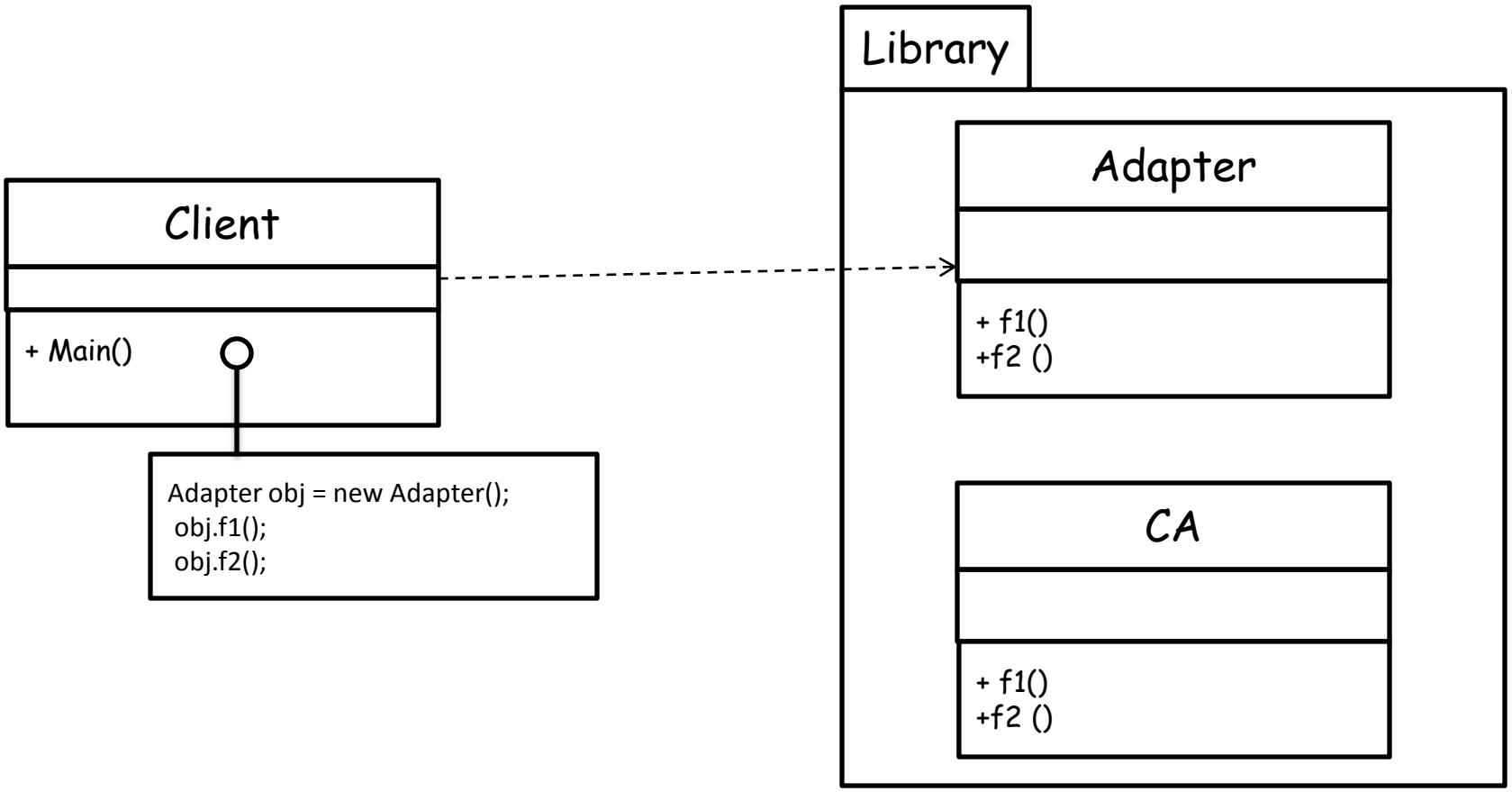

WithdrawHelper
+ getWithdrawLimit(obj : Account)
SavingAC
Library

# Business Rule Engine

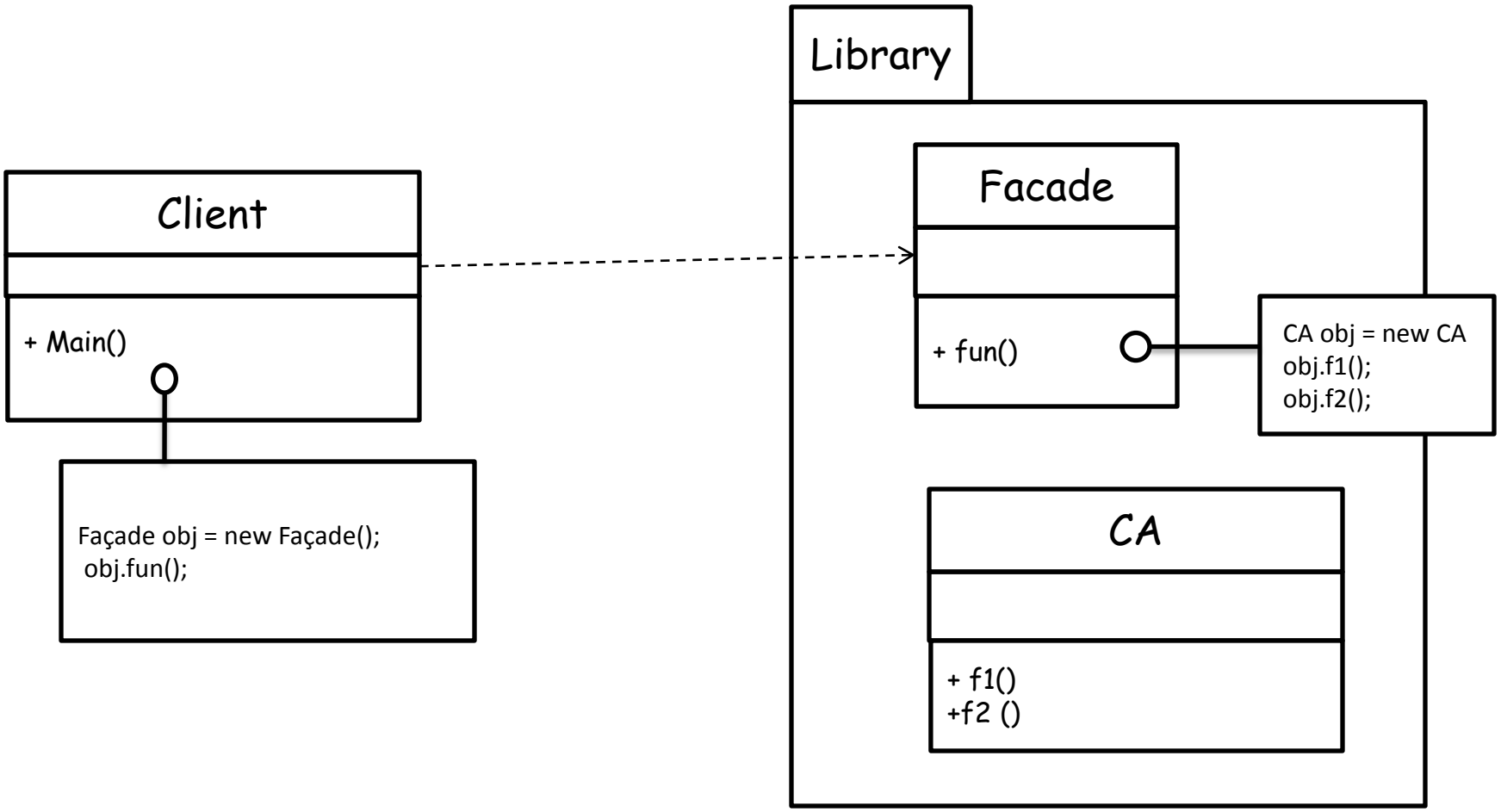
# Low Coupling problem



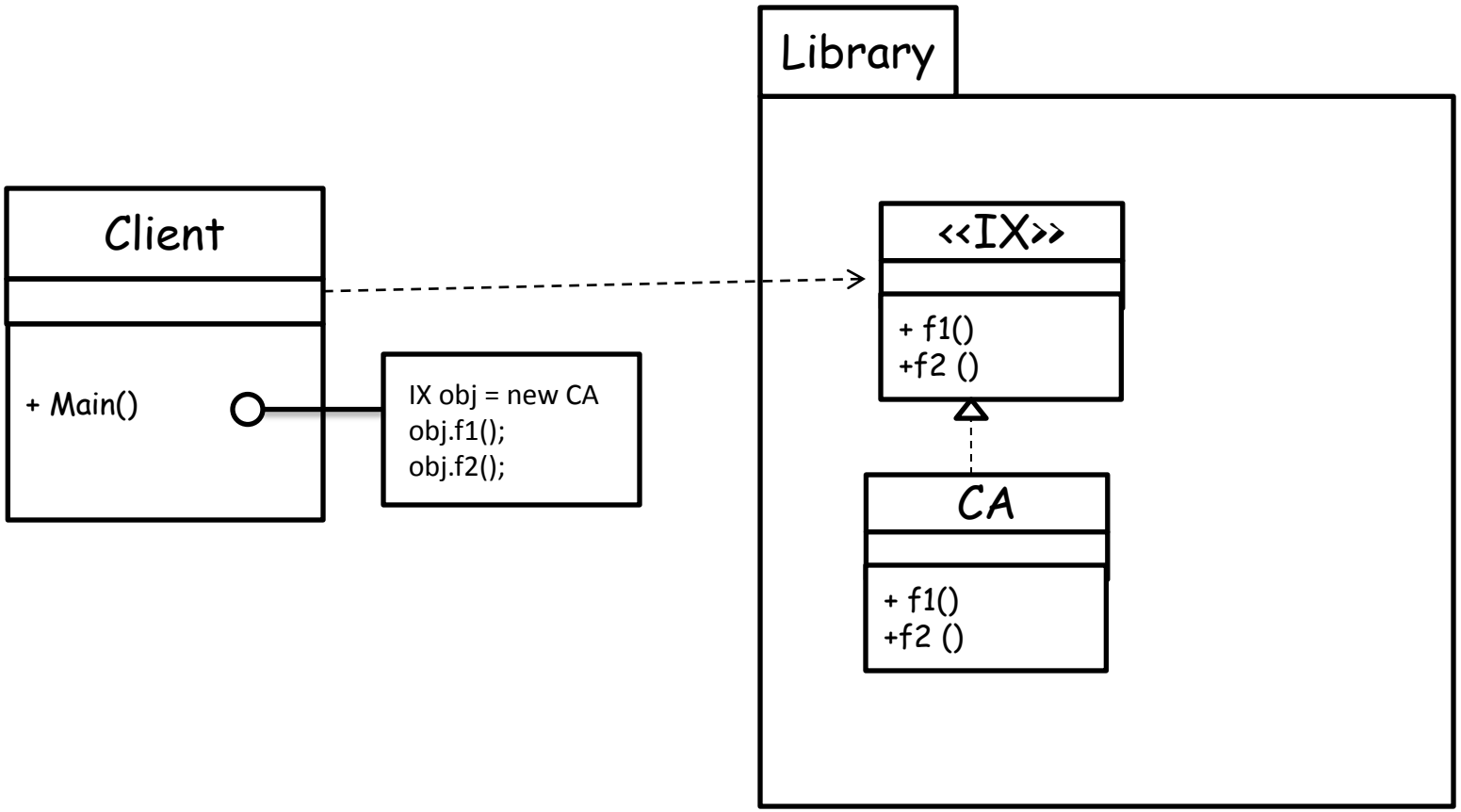
# Applying Adapter



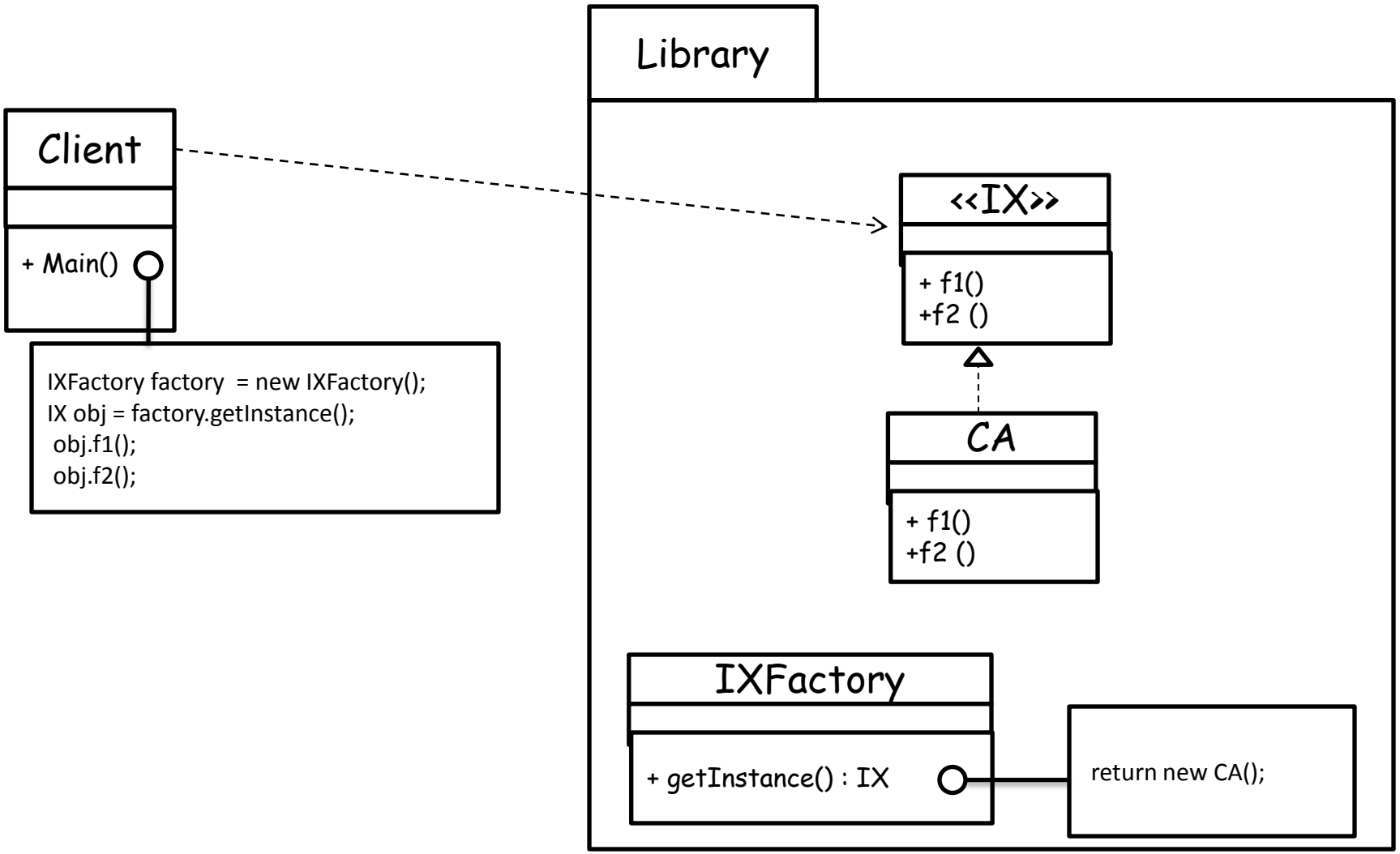
# Applying Facade



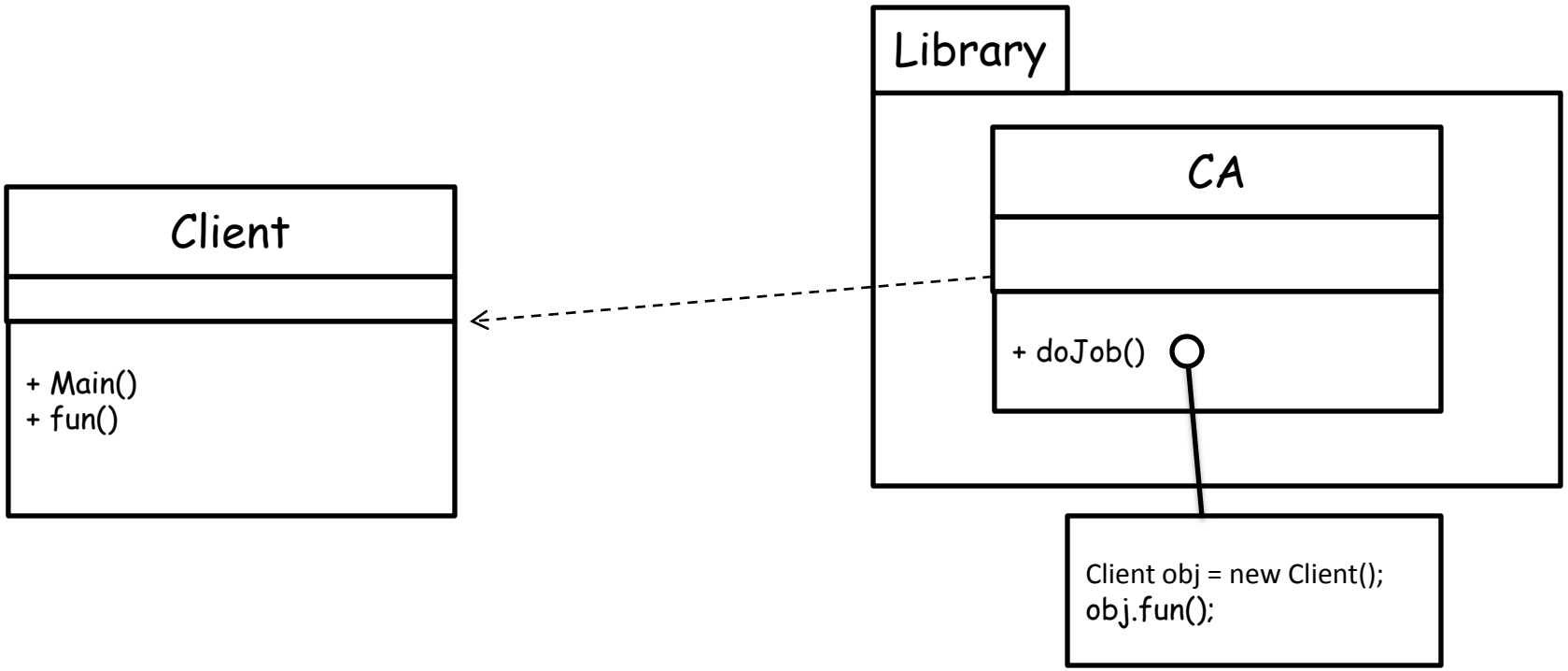
# Applying interface pattern



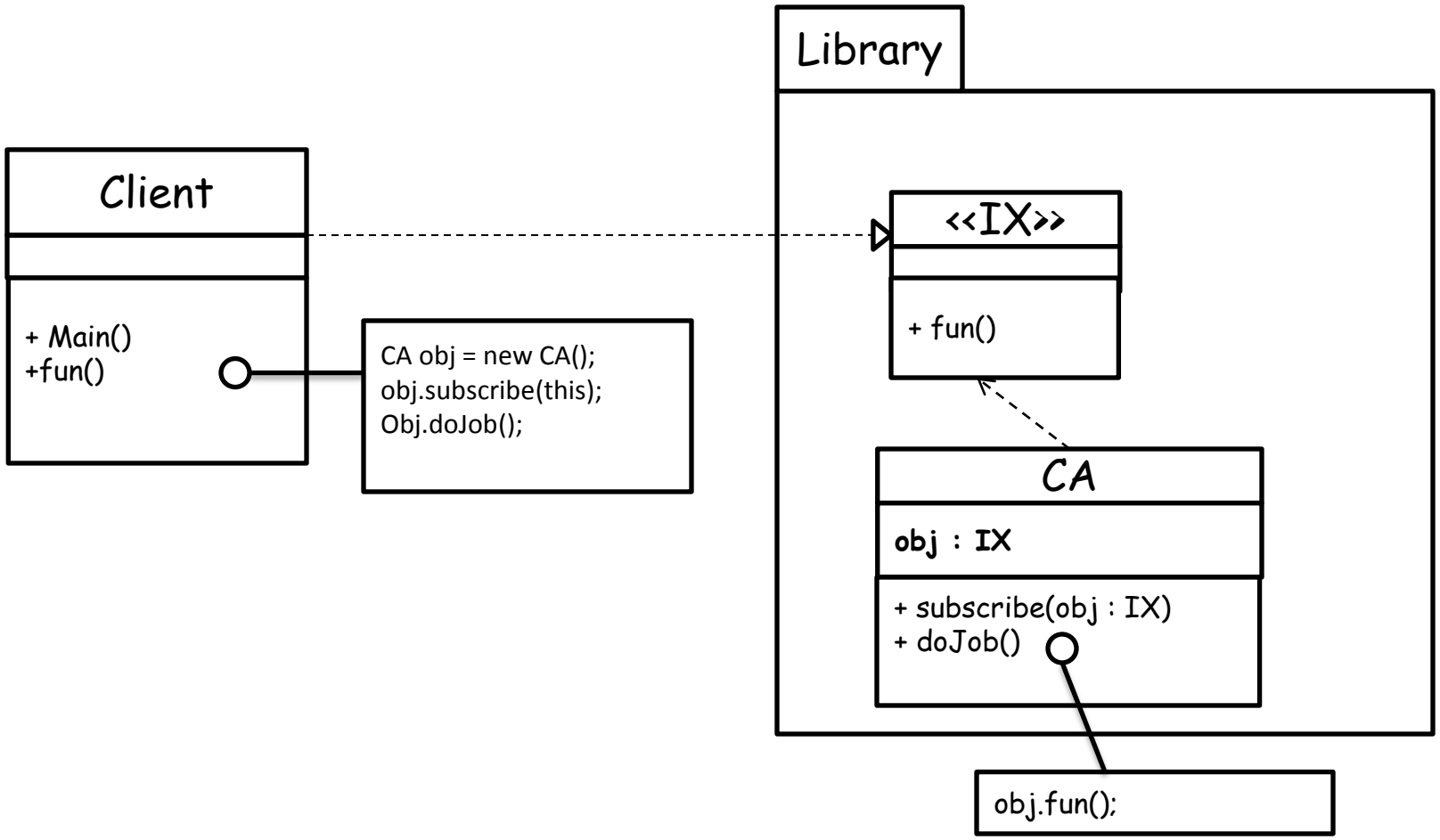
# Applying Class Factory



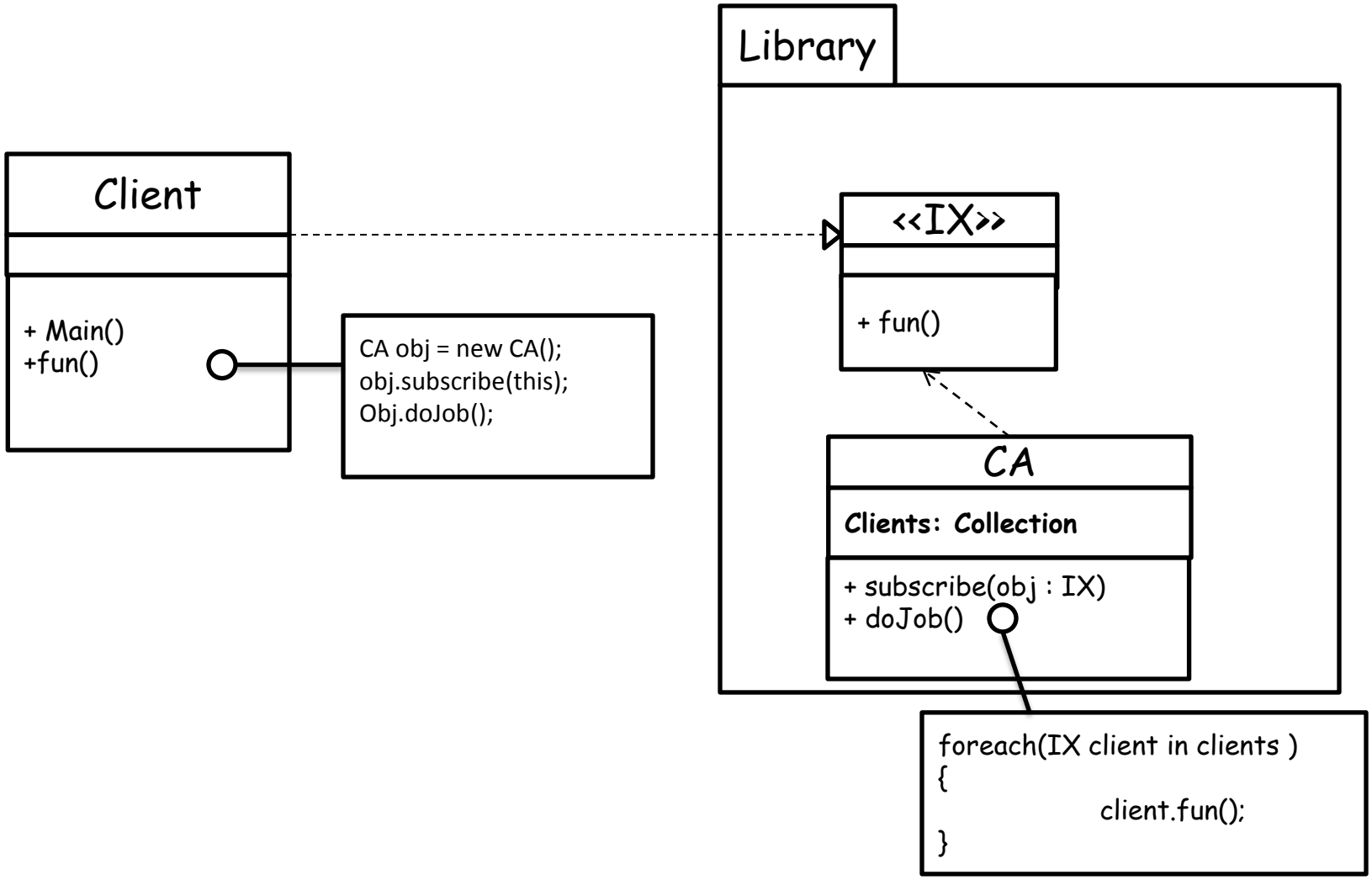
# Callback Coupling Problem



# Applying interface pattern



# Applying Observer



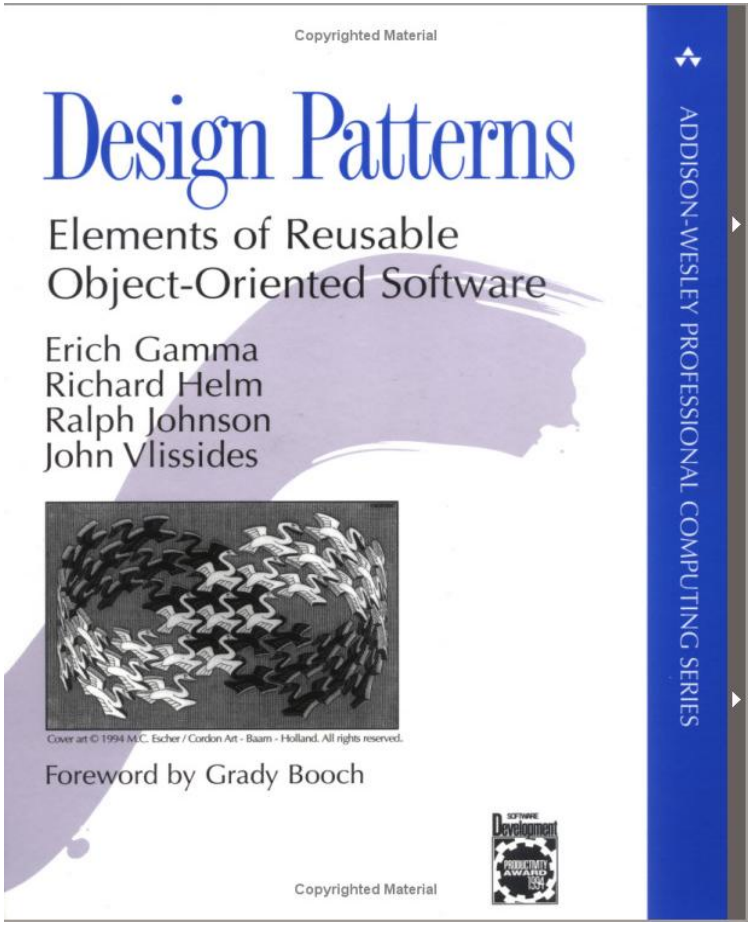
# Iterator in java

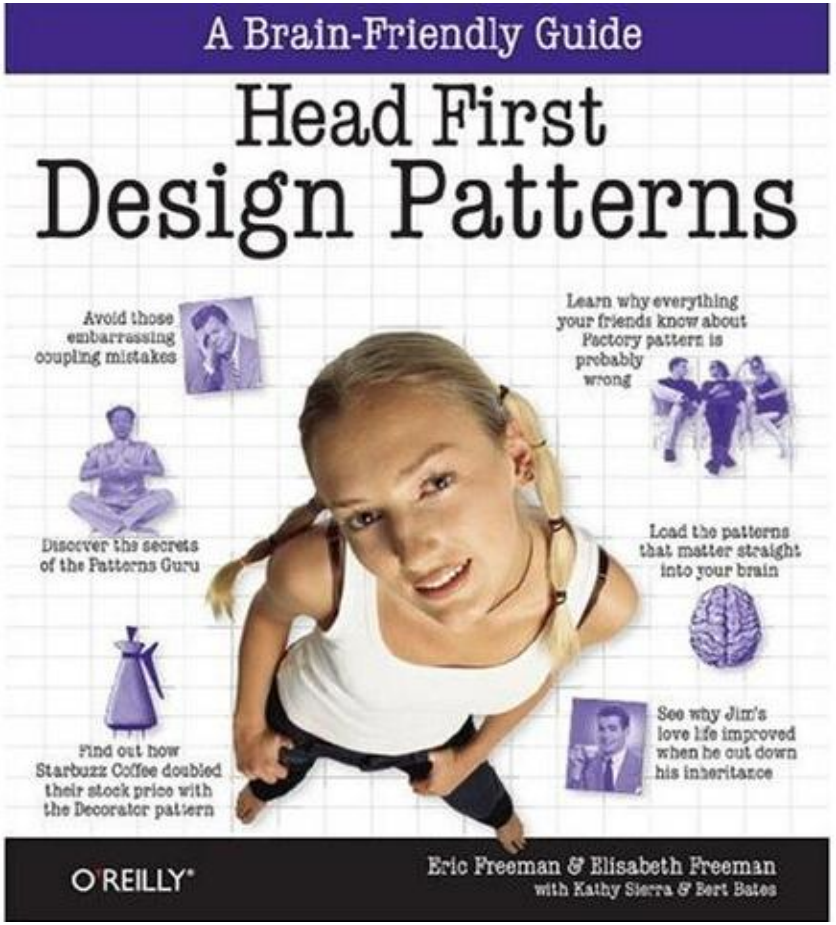
- You may have met iterators in Java

```
ArrayList<MyElement> myCollection
          = new ArrayList <MyElement> ();
. . .
//create an iterator
Iterator it = myCollection.iterator();
while (it.hasNext())
{
   //get next element
   MyElement element = it.next();
   //do some processing with this element
   element.doMethod();
}
```

# Recommended readings

# Application design



# Refactoring