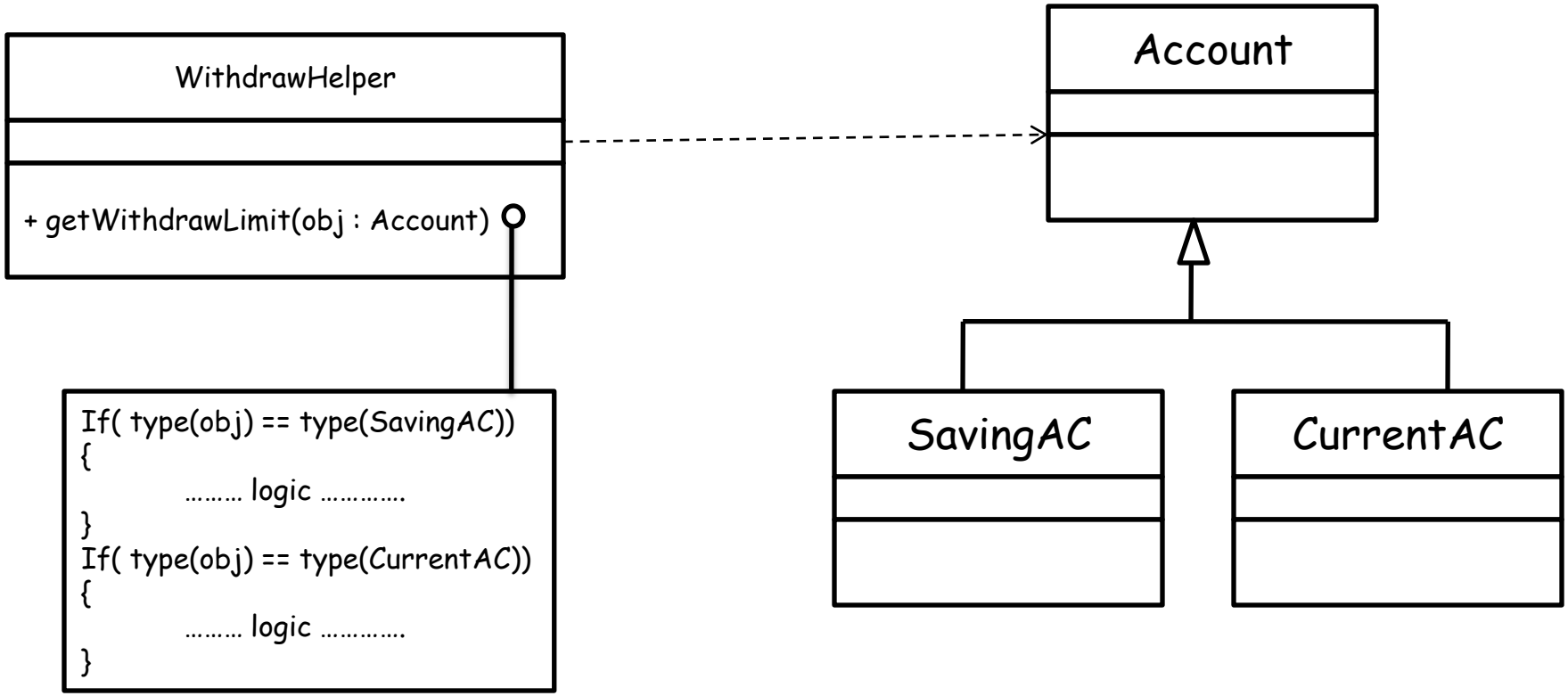
Problem


WithdrawHelper
+ getWithdrawLimit(obj : Account)
If(type(obj) == type(SavingAC))
{
…… logic ……
}
If(type(obj) == type(CurrentAC))
{
…… logic ……
}
Account
SavingAC
CurrentAC

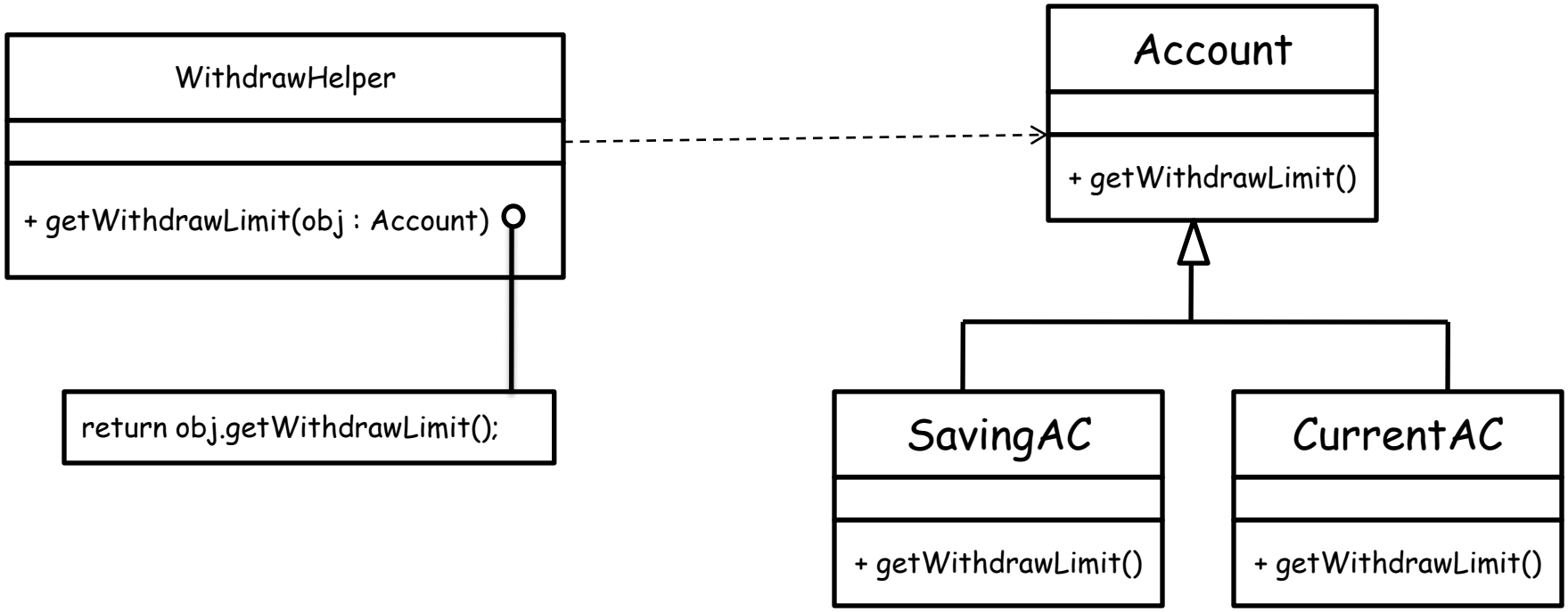
Open/Closed Principle (OCP)

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

- Open for extension.
  - As the requirements of the application change, we can extend the module with new behaviors that satisfy those changes.

- Closed for modification.
  - Extending the behavior of a module does not result in changes to the source or binary code of the module.

Applying Polymorphism

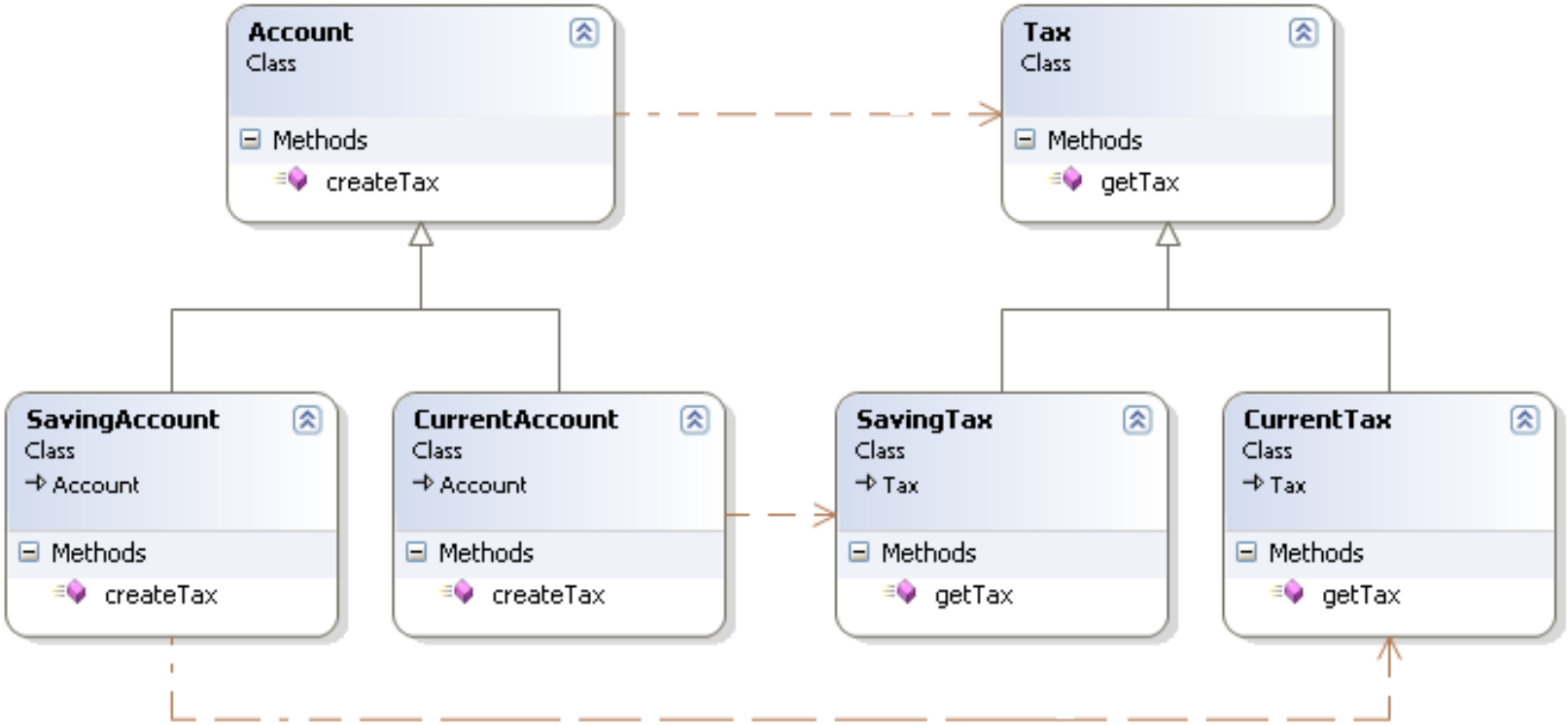


Liskov Substitution Principle (LSP)

- Subtypes must be substitutable for their base types.

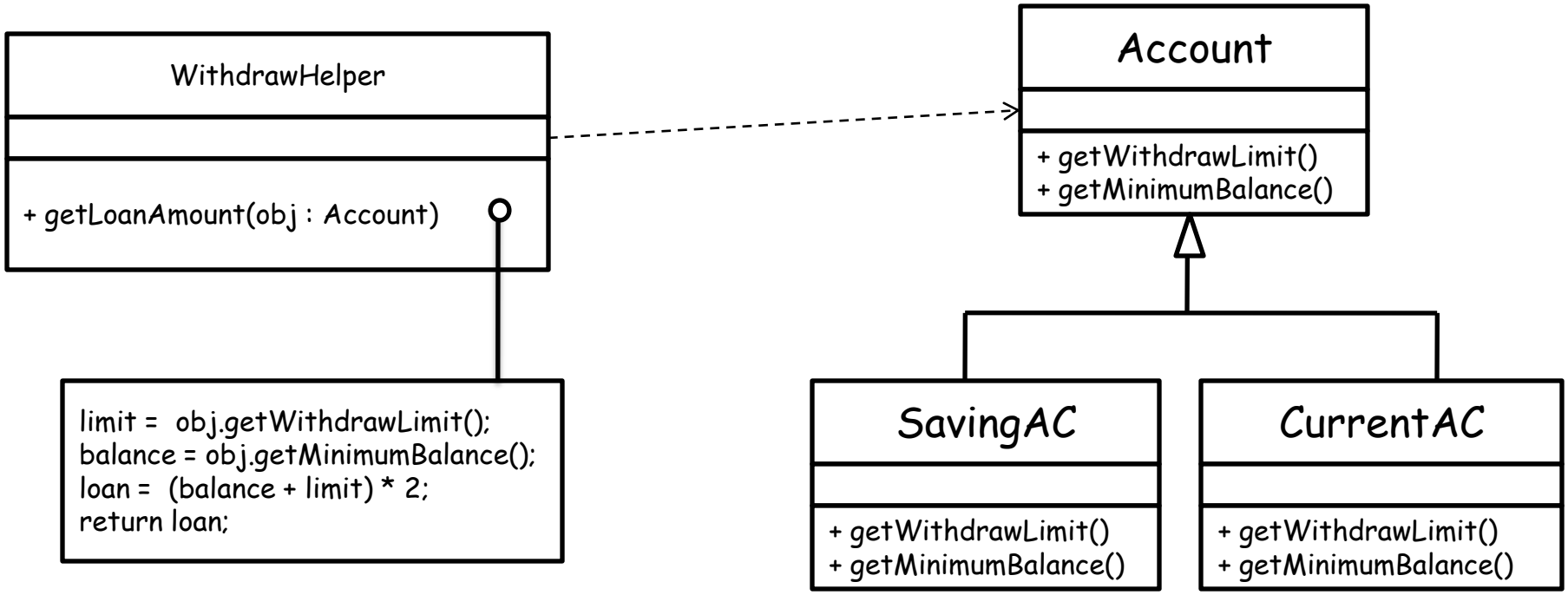
```
class T{}
class S : T{}

S o1;
T o2;

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

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




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

Problem

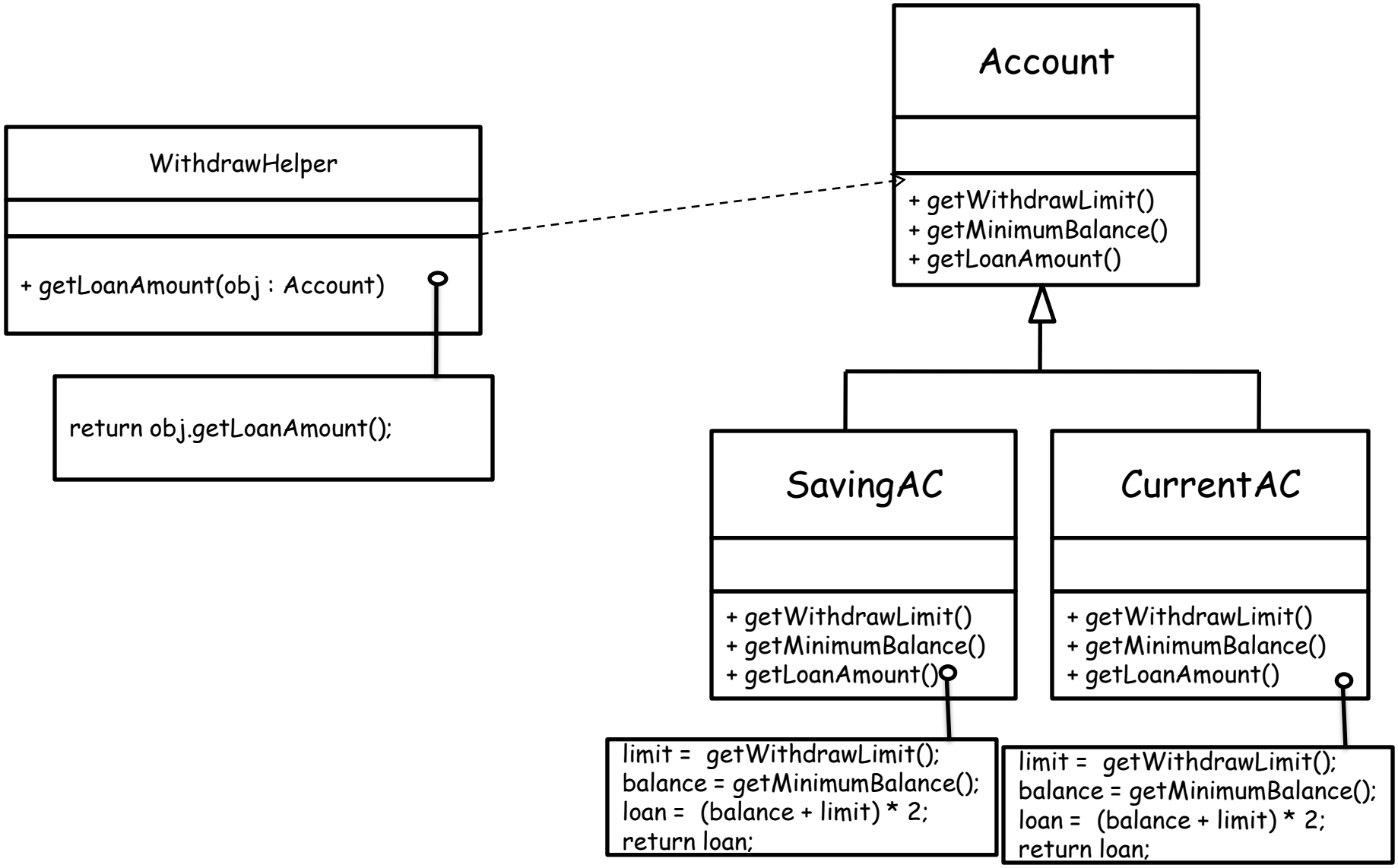

WithdrawHelper
+ getLoanAmount(obj : Account)
limit = obj.getWithdrawLimit();
balance = obj.getMinimumBalance();
loan = (balance + limit) * 2;
return loan;
Account
+ getWithdrawLimit()
+ getMinimumBalance()
SavingAC
+ getWithdrawLimit()
+ getMinimumBalance()
CurrentAC
+ getWithdrawLimit()
+ getMinimumBalance()

Information Expert

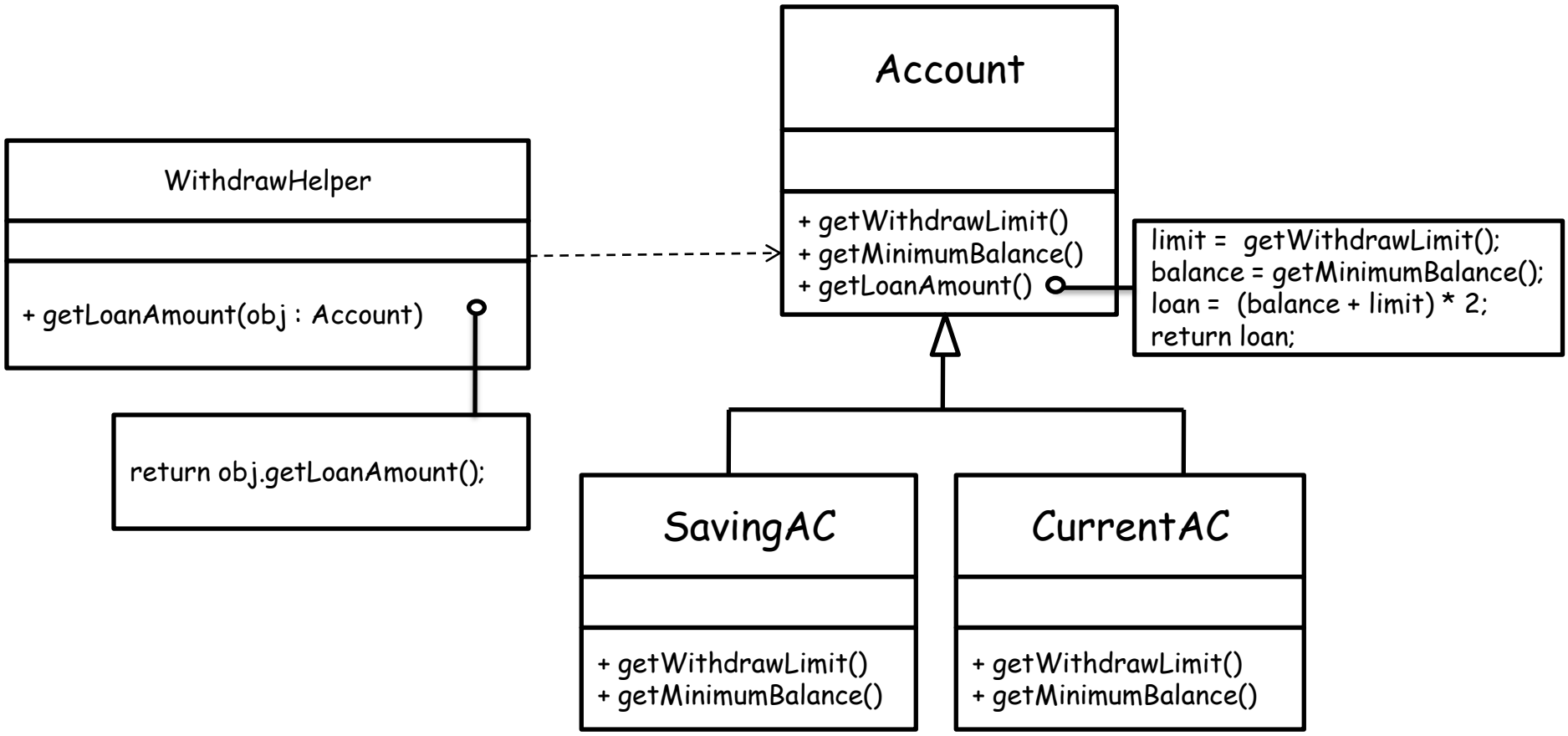
A system will have hundreds of classes. How do I begin to assign responsibilities to them?

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

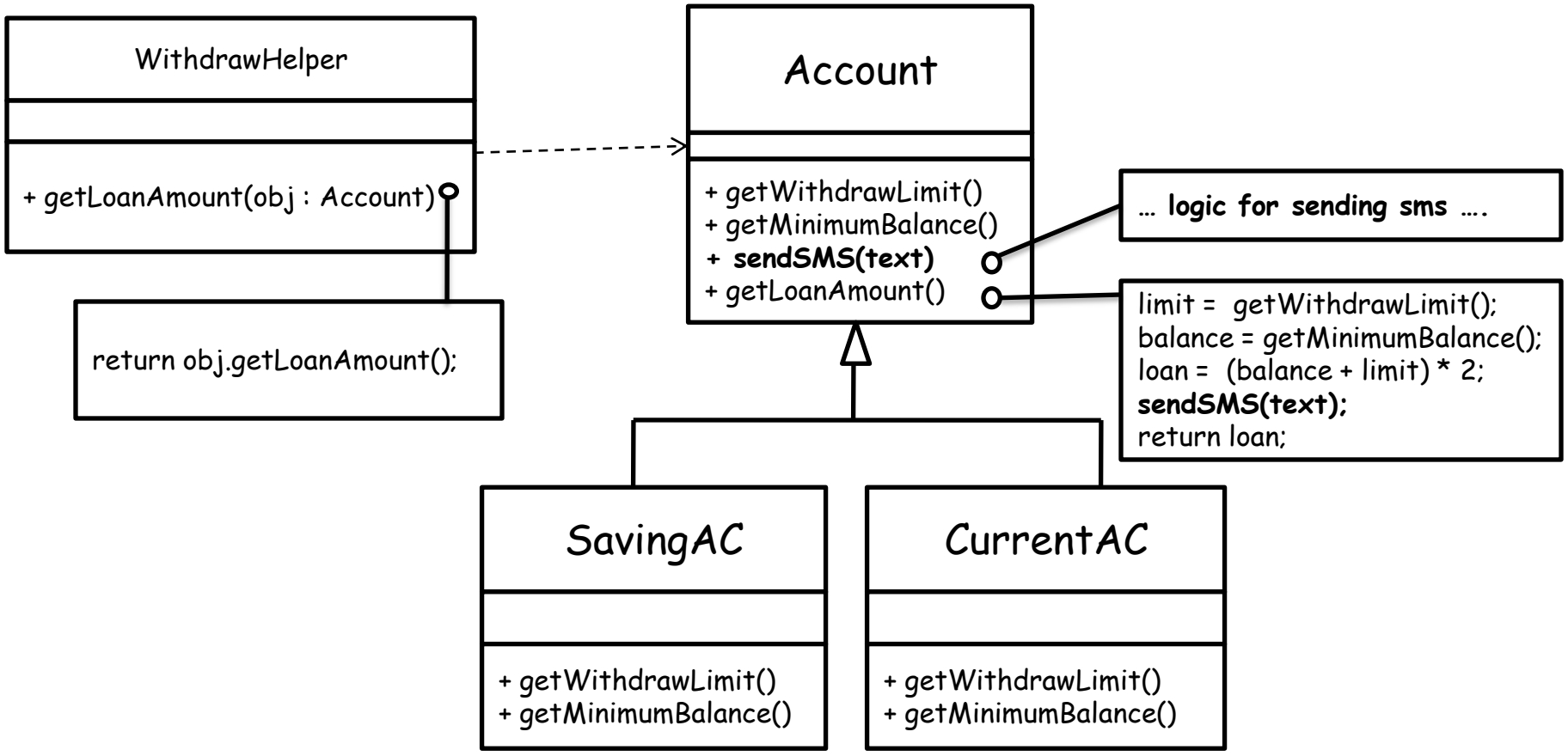
Applying Information Expert



Applying Template Method



Problem


WithdrawHelper
+ getLoanAmount(obj : Account)
return obj.getLoanAmount();
Account
+ getWithdrawLimit()
+ getMinimumBalance()
+ sendSMS(text)
+ getLoanAmount()
… logic for sending sms ….
limit = getWithdrawLimit();
balance = getMinimumBalance();
loan = (balance + limit) * 2;
sendSMS(text);
return loan;
SavingAC
+ getWithdrawLimit()
+ getMinimumBalance()
CurrentAC
+ getWithdrawLimit()
+ getMinimumBalance()

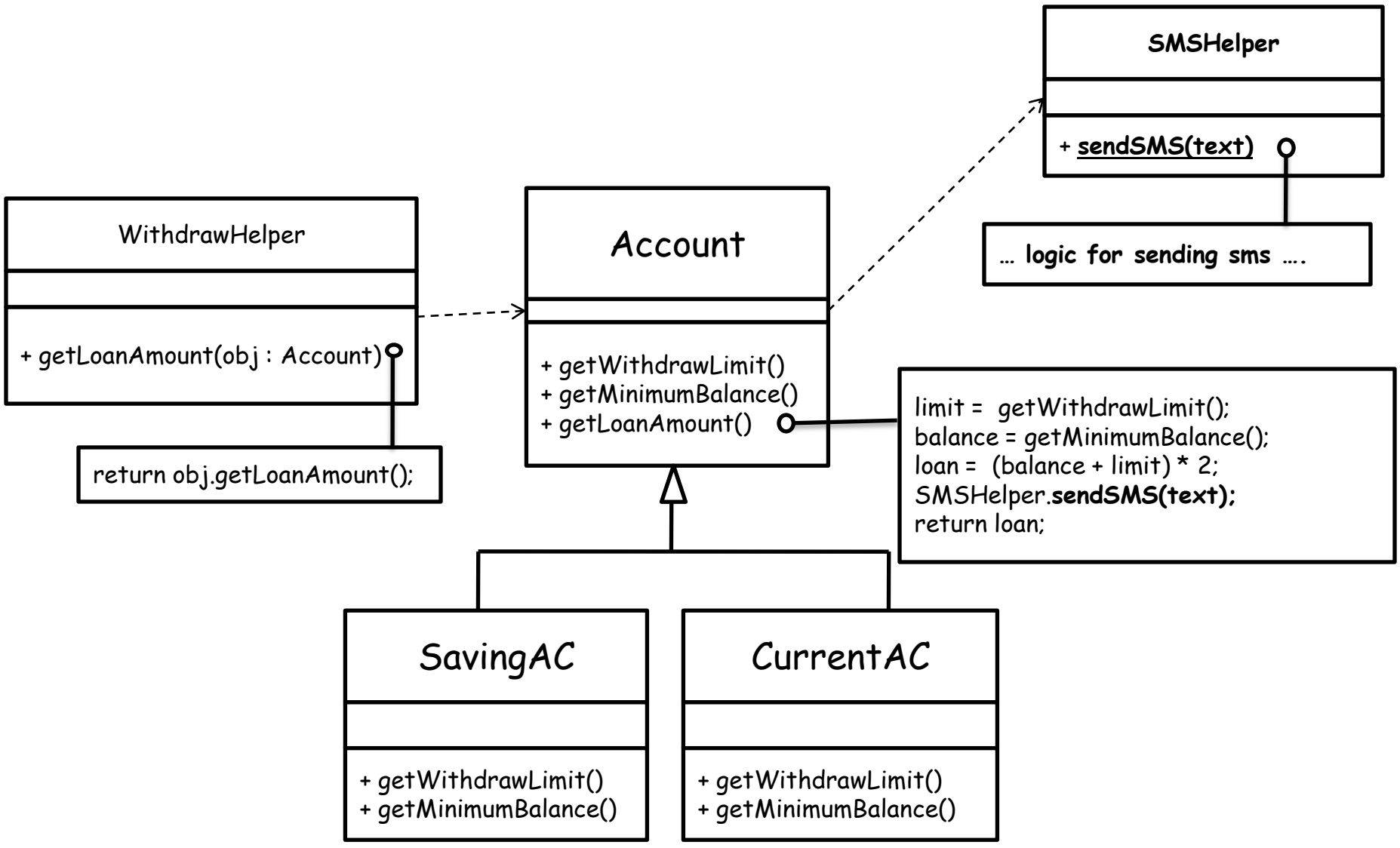
High Cohesion

- Cohesion is a measure of how strongly related and focused the responsibilities of a class are in brief class should do only highly related responsibilities.

- **Problem:** How to keep Cohesion High?
  - **Solution :** Single-Responsibility Principle

- 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.

Applying SRP

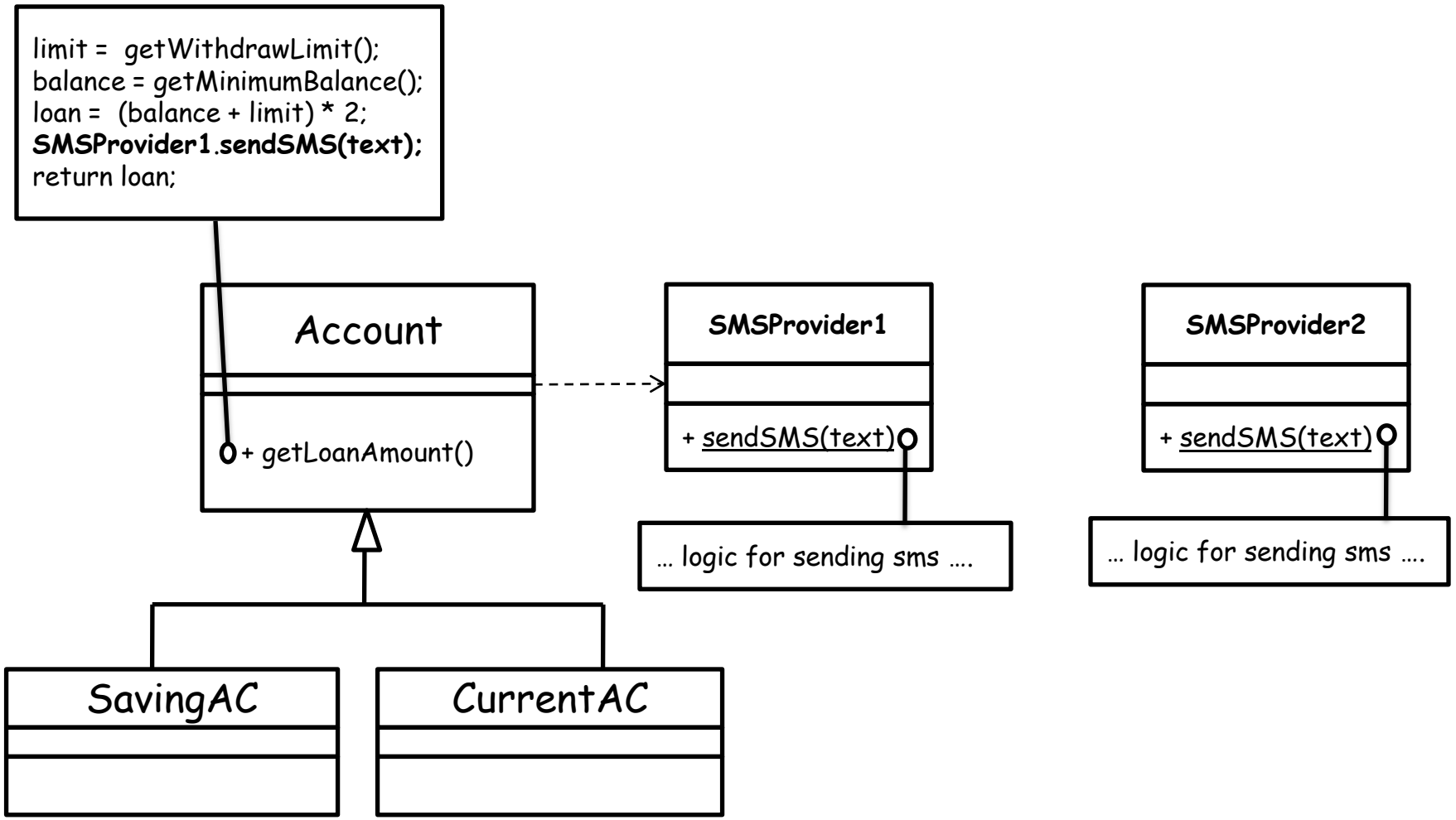


Pure Fabrication

- 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.

Problem


limit = getWithdrawLimit();
balance = getMinimumBalance();
loan = (balance + limit) * 2;
SMSProvider1.sendSMS(text);
return loan;
Account
+ getLoanAmount()
SMSProvider1
+ sendSMS(text)
… logic for sending sms ….
SMSProvider2
+ sendSMS(text)
… logic for sending sms ….
SavingAC
CurrentAC

Low Coupling

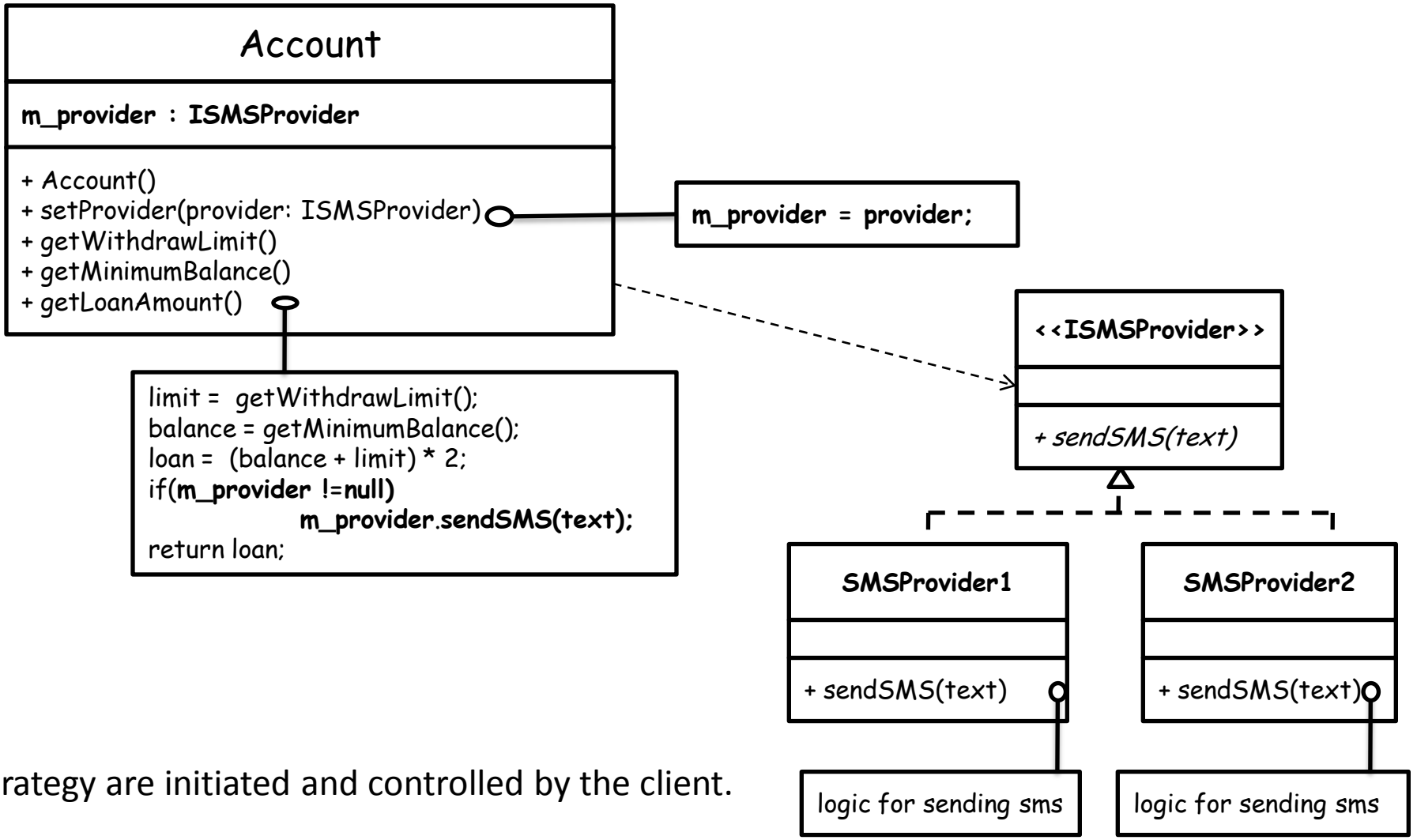
A class with high (or strong) coupling relies upon many other classes. Such classes are undesirable.

Law of Demeter

Don’t Talk to Stranger

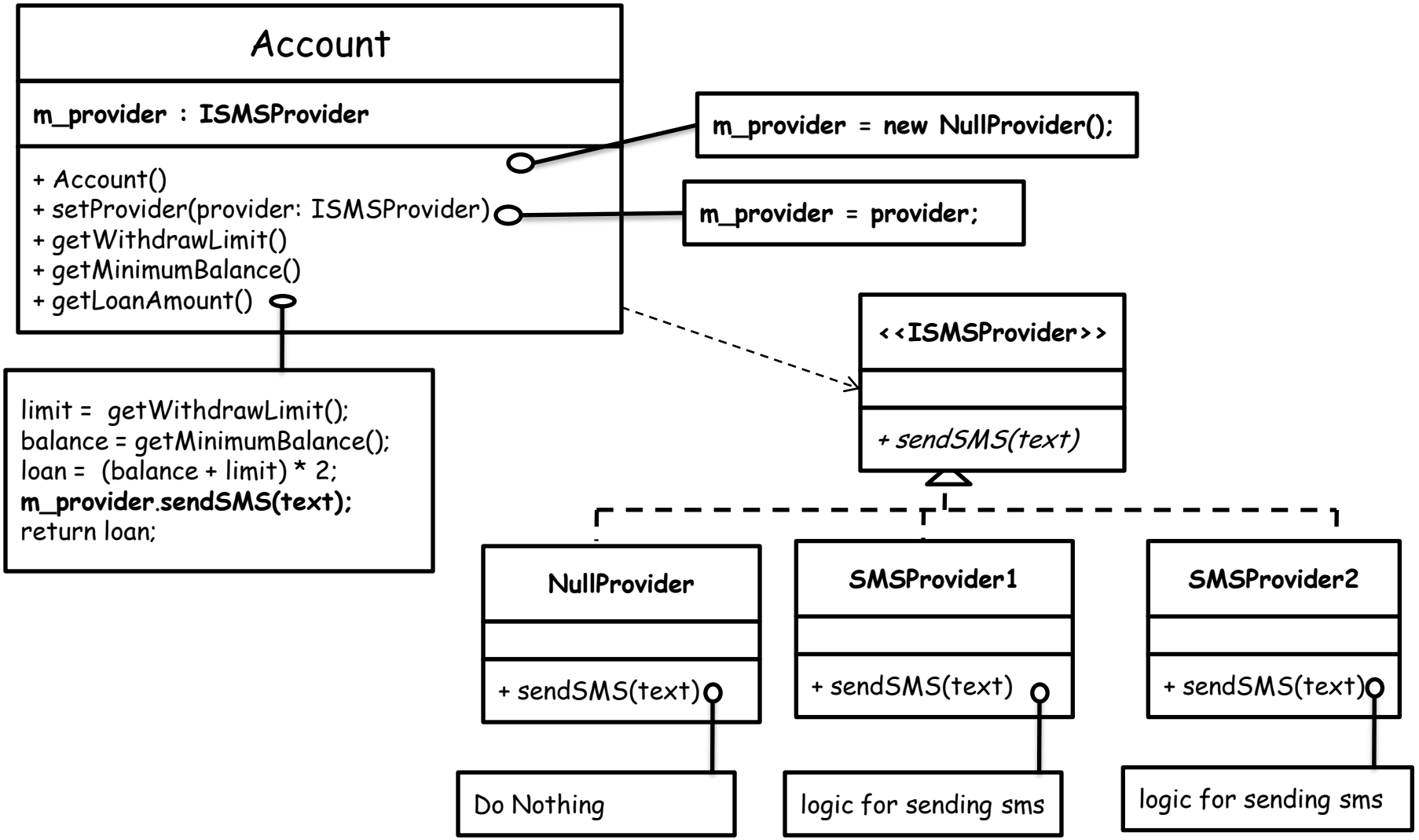
1. Your method can call other methods in its class directly
2. Your method can call methods on its own fields directly (but not on the fields' fields)
3. When your method takes parameters, your method can call methods on those parameters directly.
4. When your method creates local objects, that method 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.

Applying Strategy

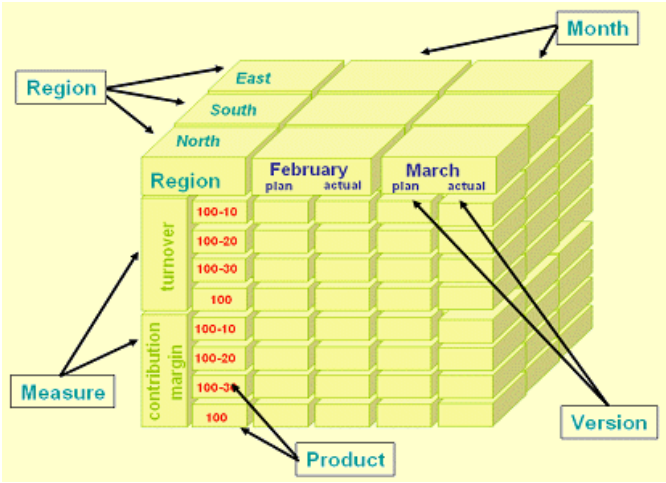


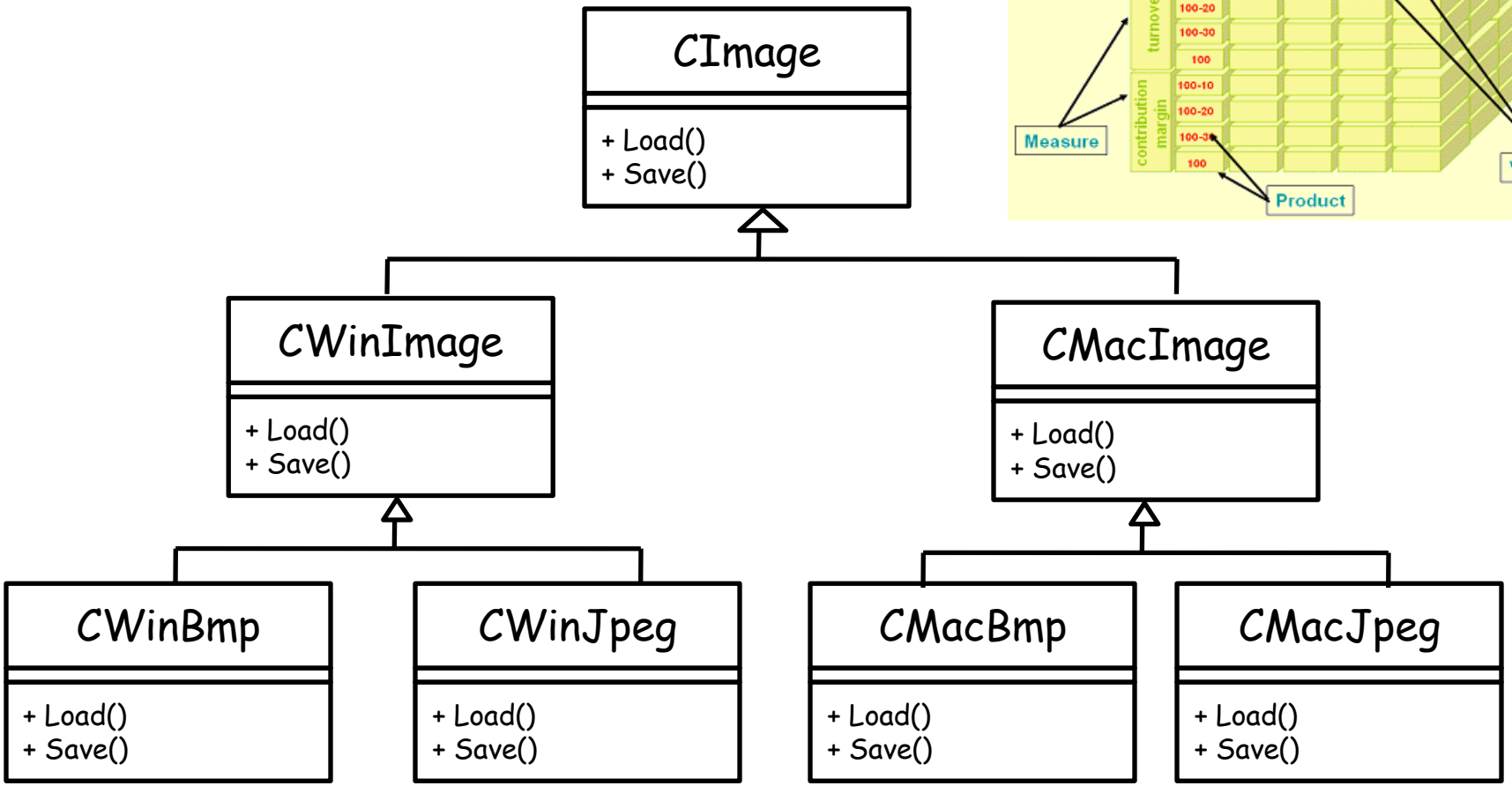
Strategy are initiated and controlled by the client.

Applying Null Object

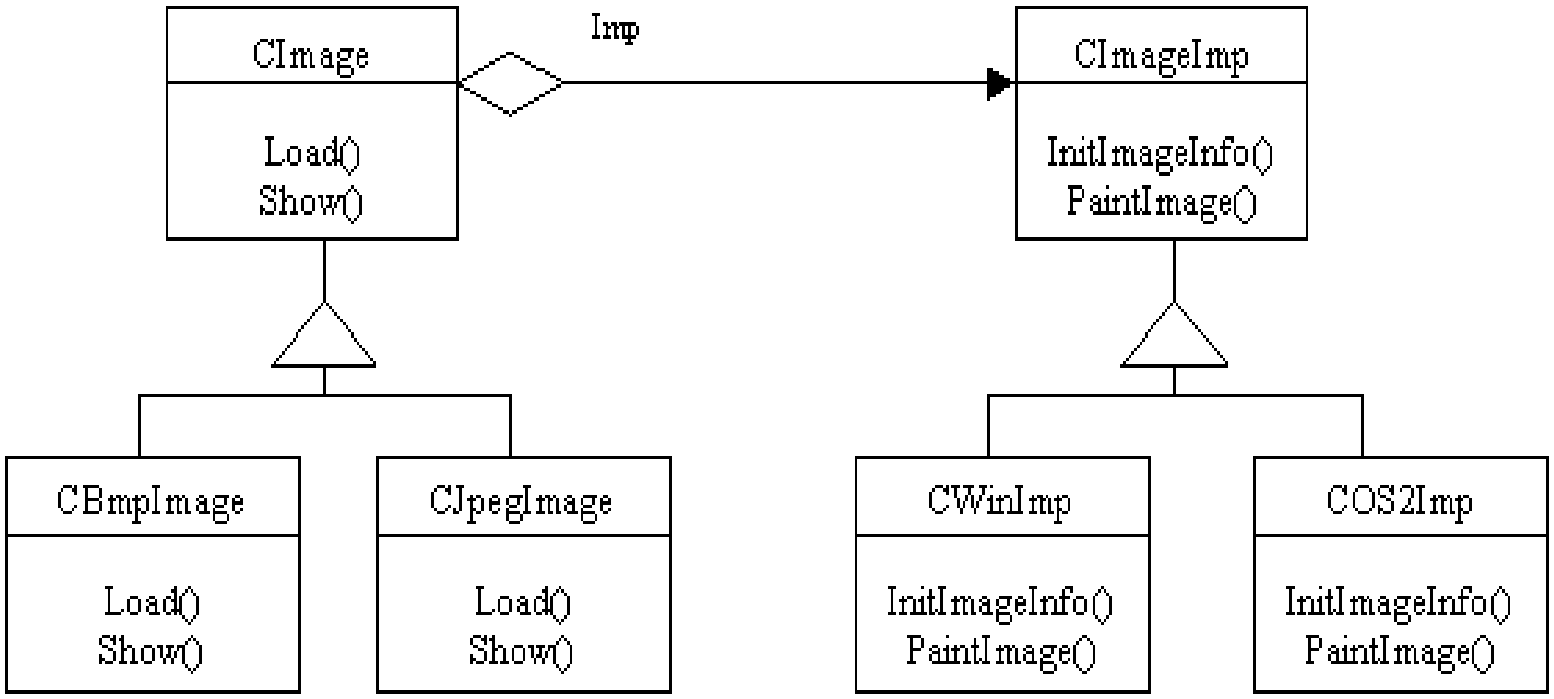


MultiDimensional Inheritance problem



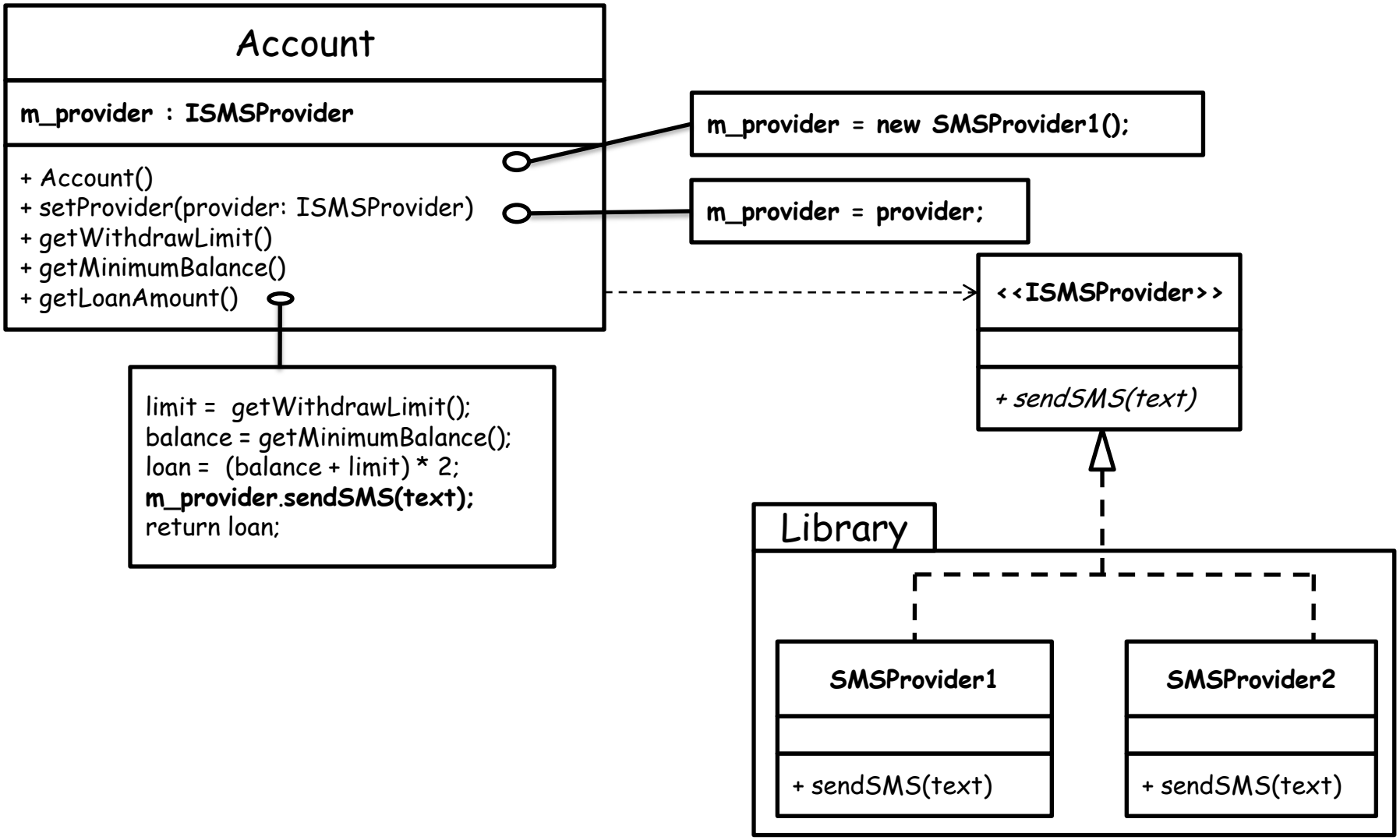


Applying Bridge

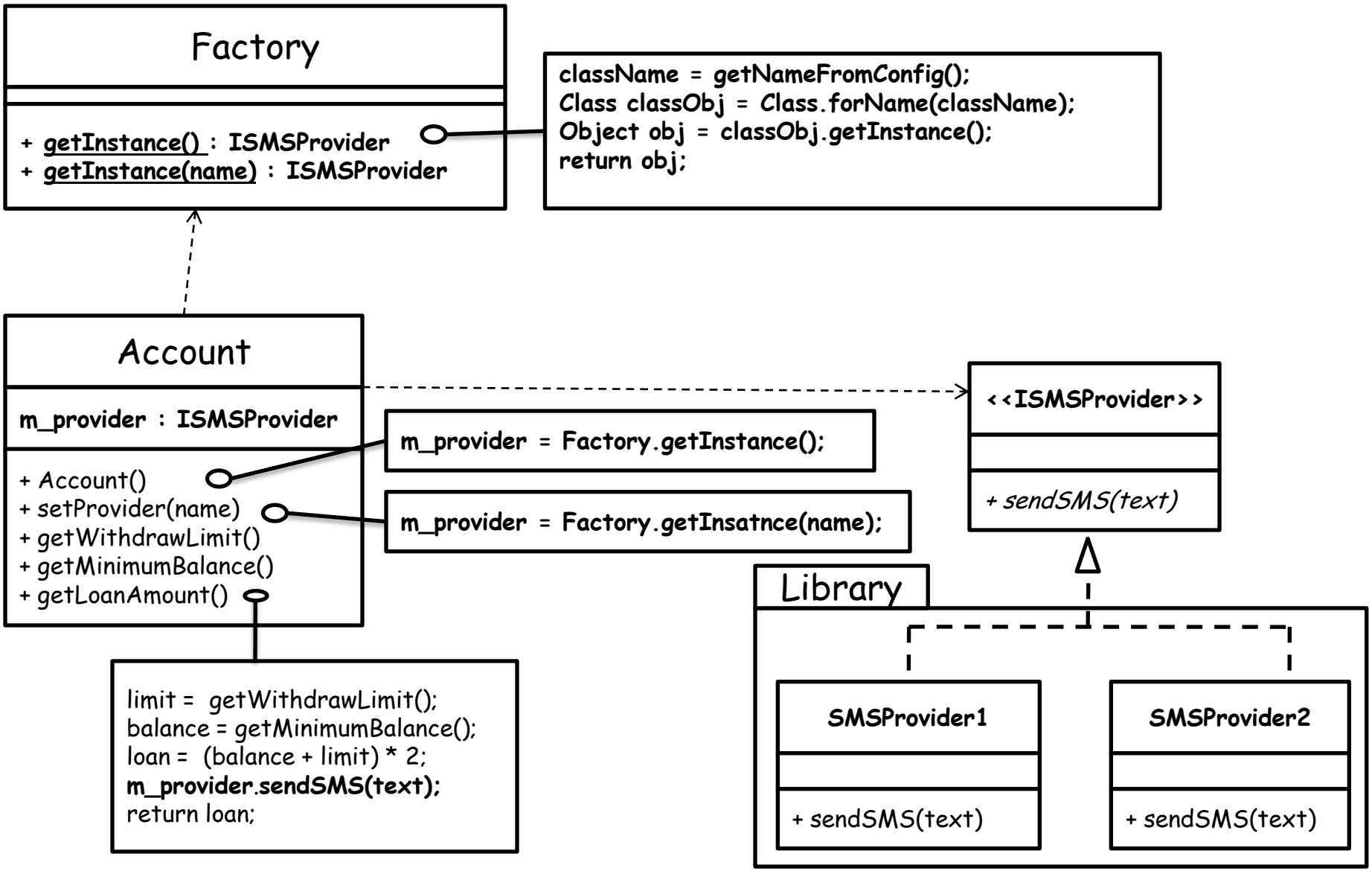


Strategy and Bridge are almost identical, differing mostly in intent only.

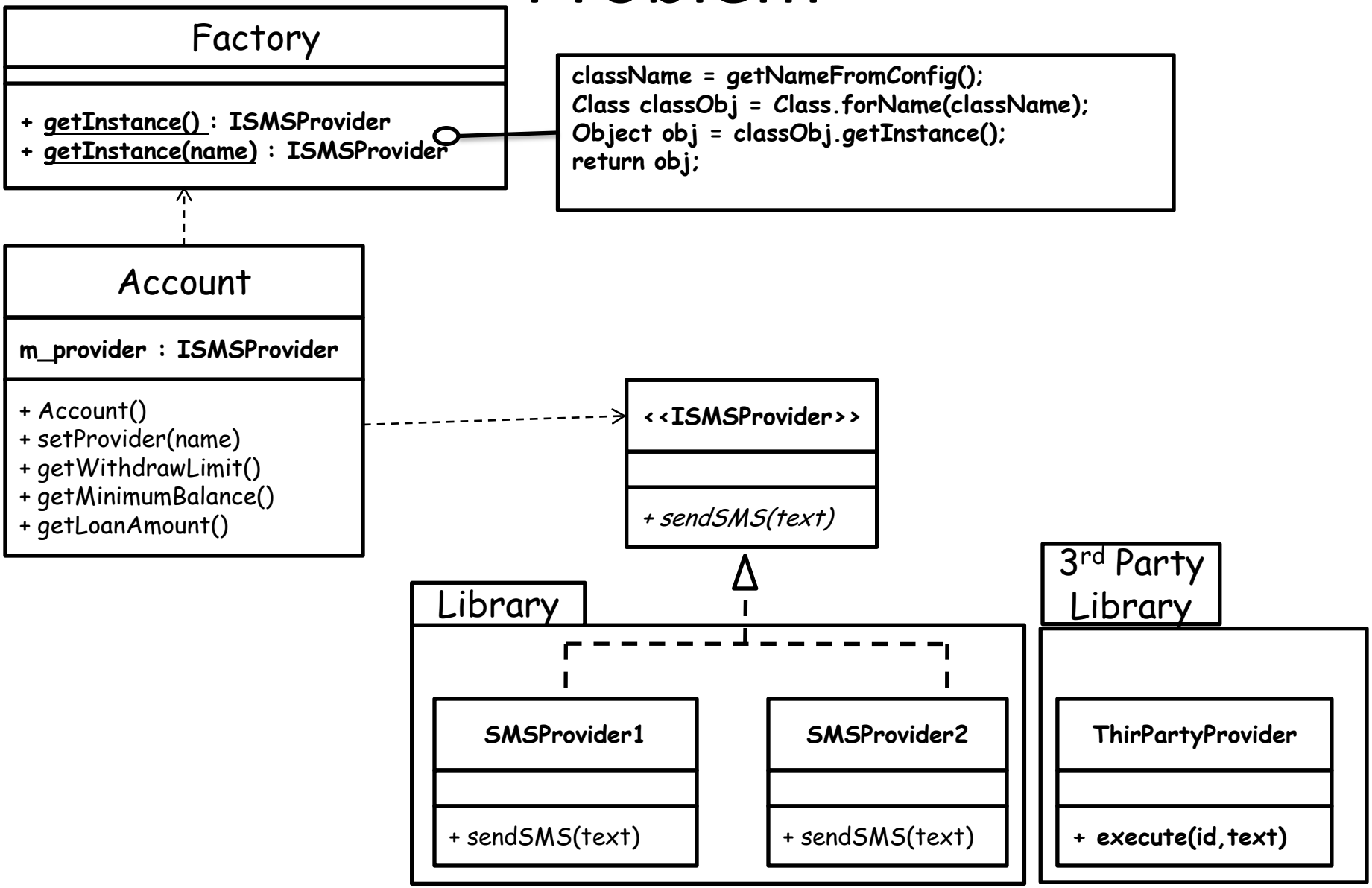
Problem


Account
m_provider : ISMSProvider
+ Account()
+ setProvider(provider: ISMSProvider)
+ getWithdrawLimit()
+ getMinimumBalance()
+ getLoanAmount()
m_provider = new SMSProvider1();
m_provider = provider;
limit = getWithdrawLimit();
balance = getMinimumBalance();
loan = (balance + limit) * 2;
m_provider.sendSMS(text);
return loan;
<<ISMSProvider>>
+ sendSMS(text)
Library
SMSProvider1
+ sendSMS(text)
SMSProvider2
+ sendSMS(text)

Applying ClassFactory



Problem

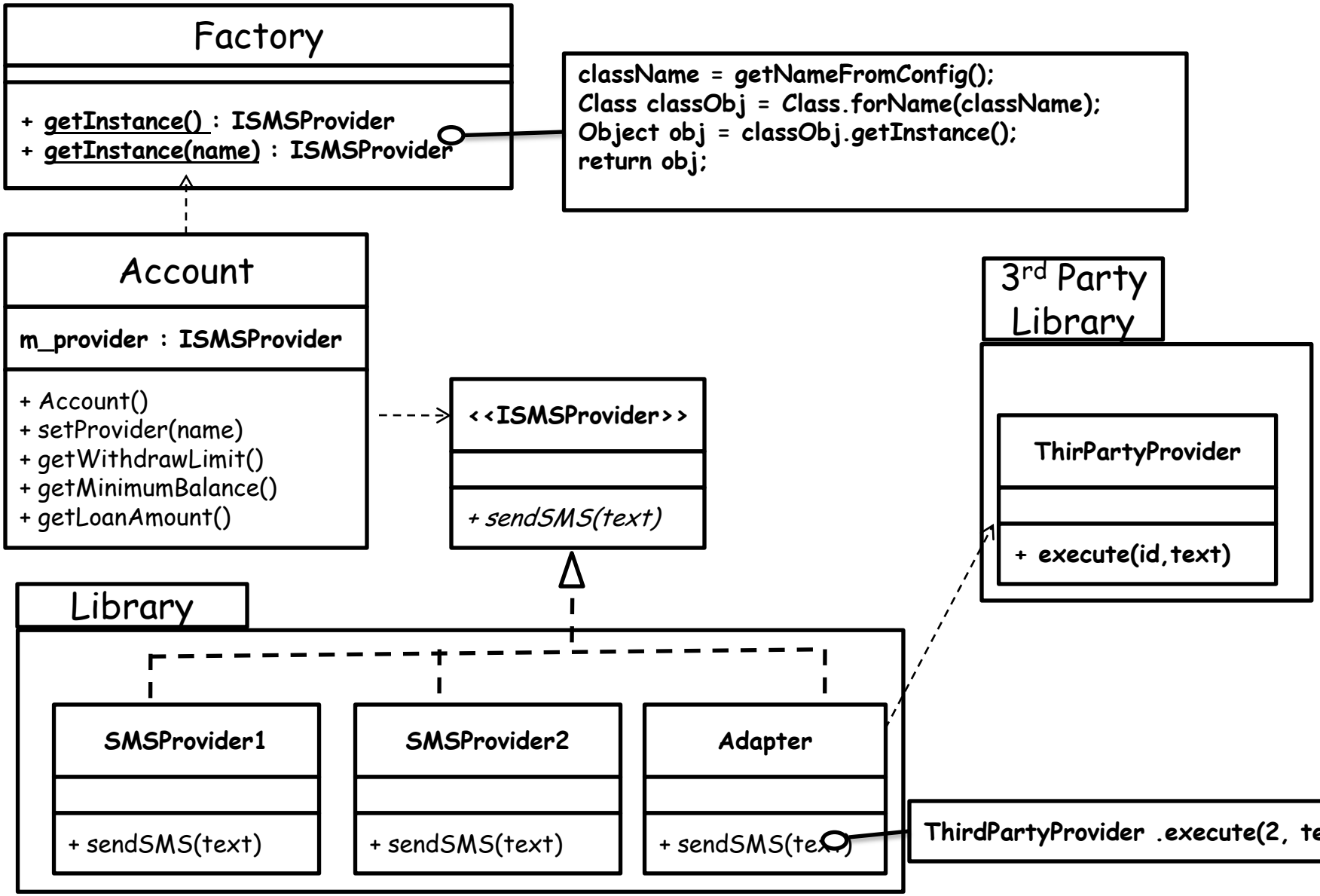

Factory
+ getInstance() : ISMSProvider
+ getInstance(name) : ISMSProvider
className = getNameFromConfig();
Class classObj = Class.forName(className);
Object obj = classObj.getInstance();
return obj;
Account
m_provider : ISMSProvider
+ Account()
+ setProvider(name)
+ getWithdrawLimit()
+ getMinimumBalance()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

Indirection

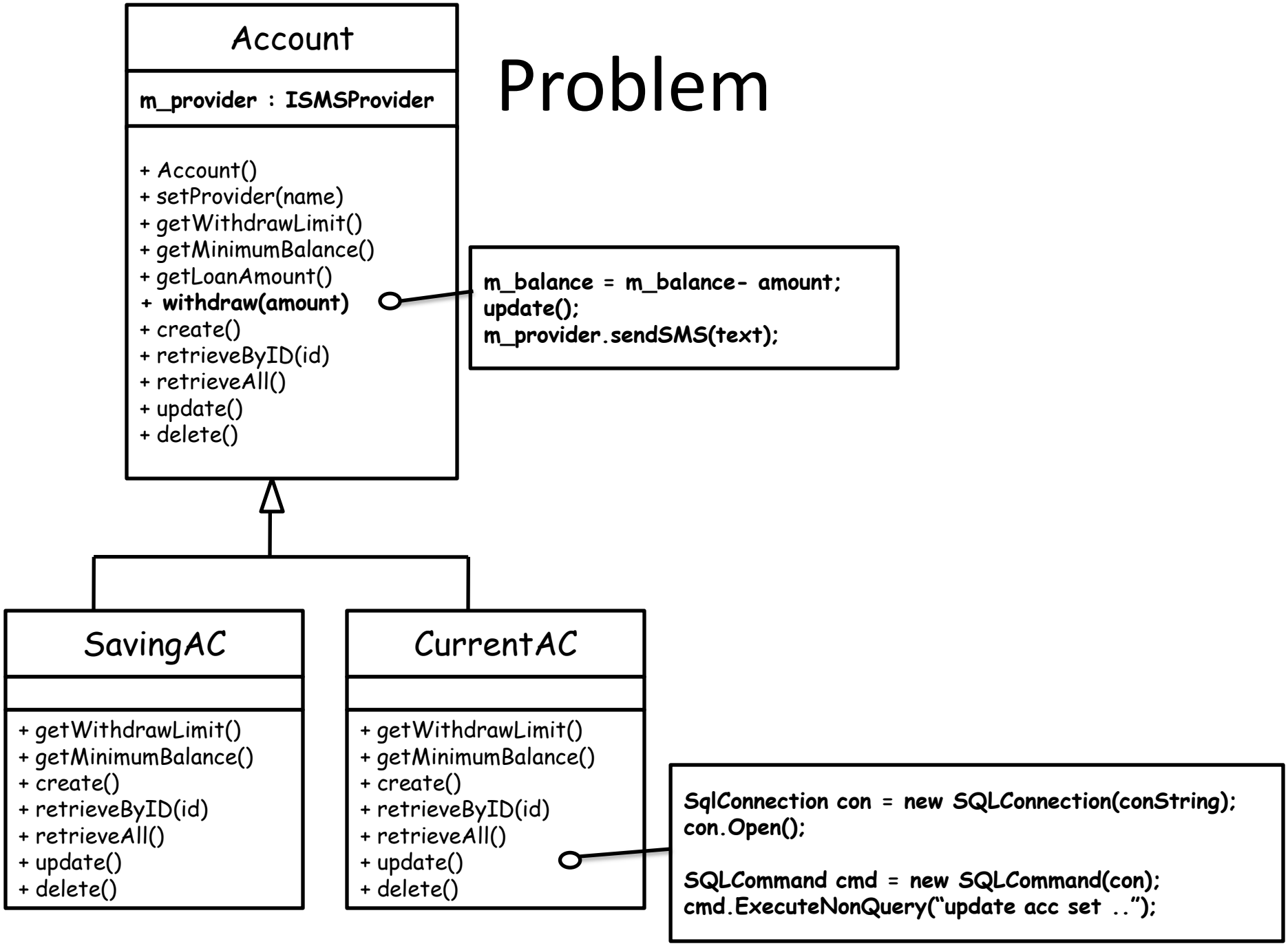
- Problem: How do we assign responsibility to avoid direct coupling between two or more classes?

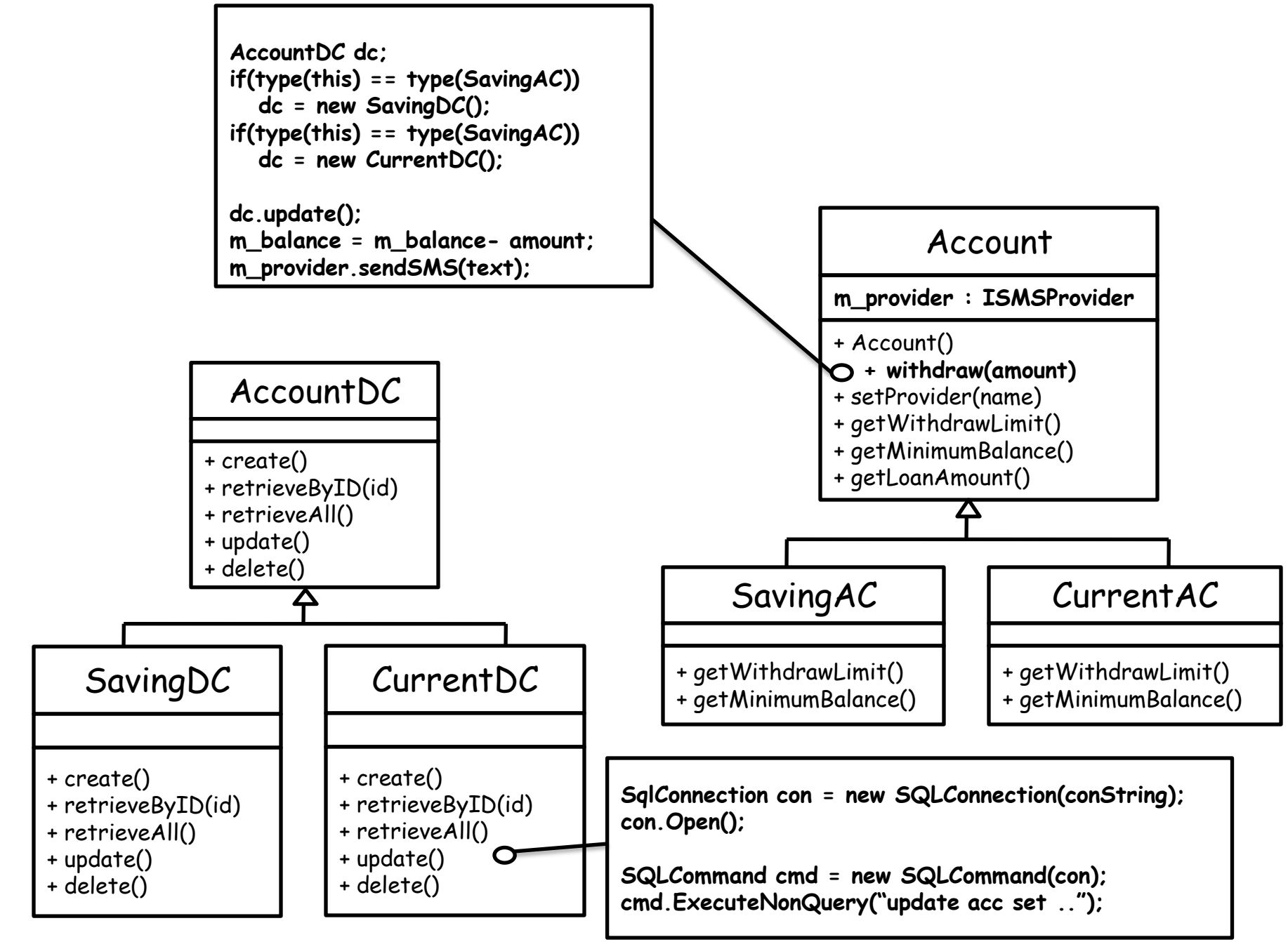
- Solution: Assign responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.

Applying Adapter



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 ..");

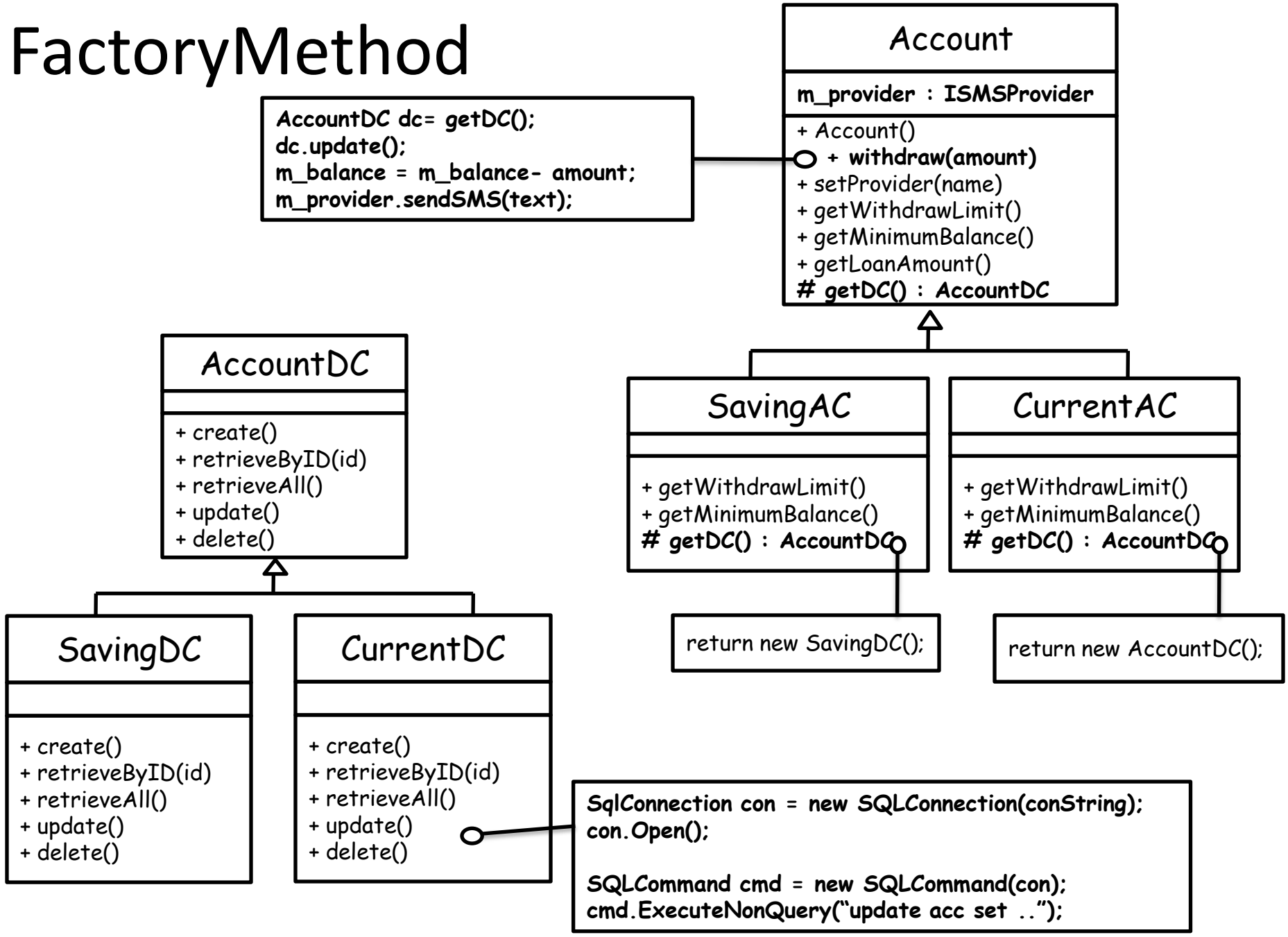
Dependency Inversion principle

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

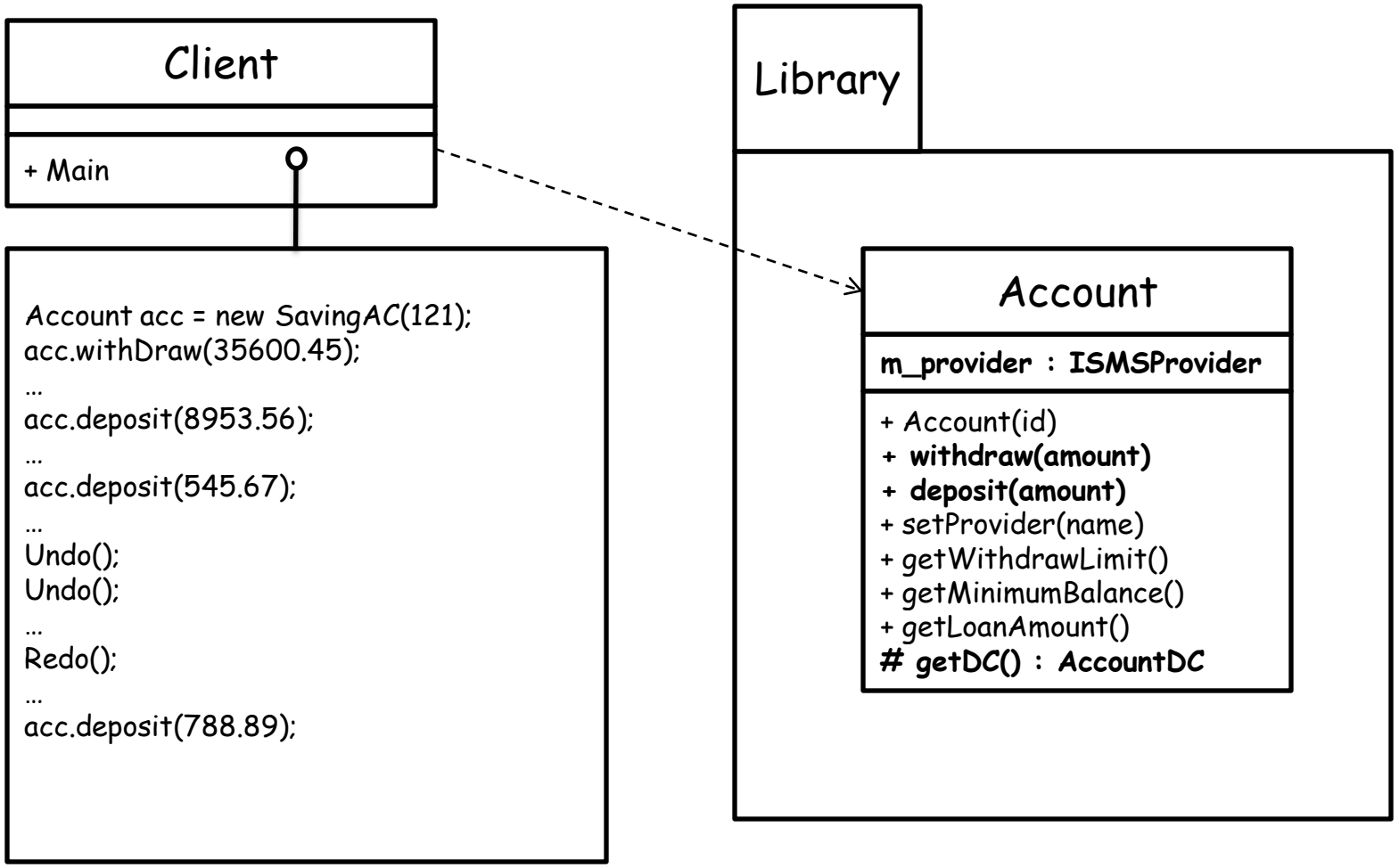
"Structured" methods of the 1970's tended towards a "top-down decomposition", which encouraged high-level modules to depend on modules written at a lower level of abstraction.

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

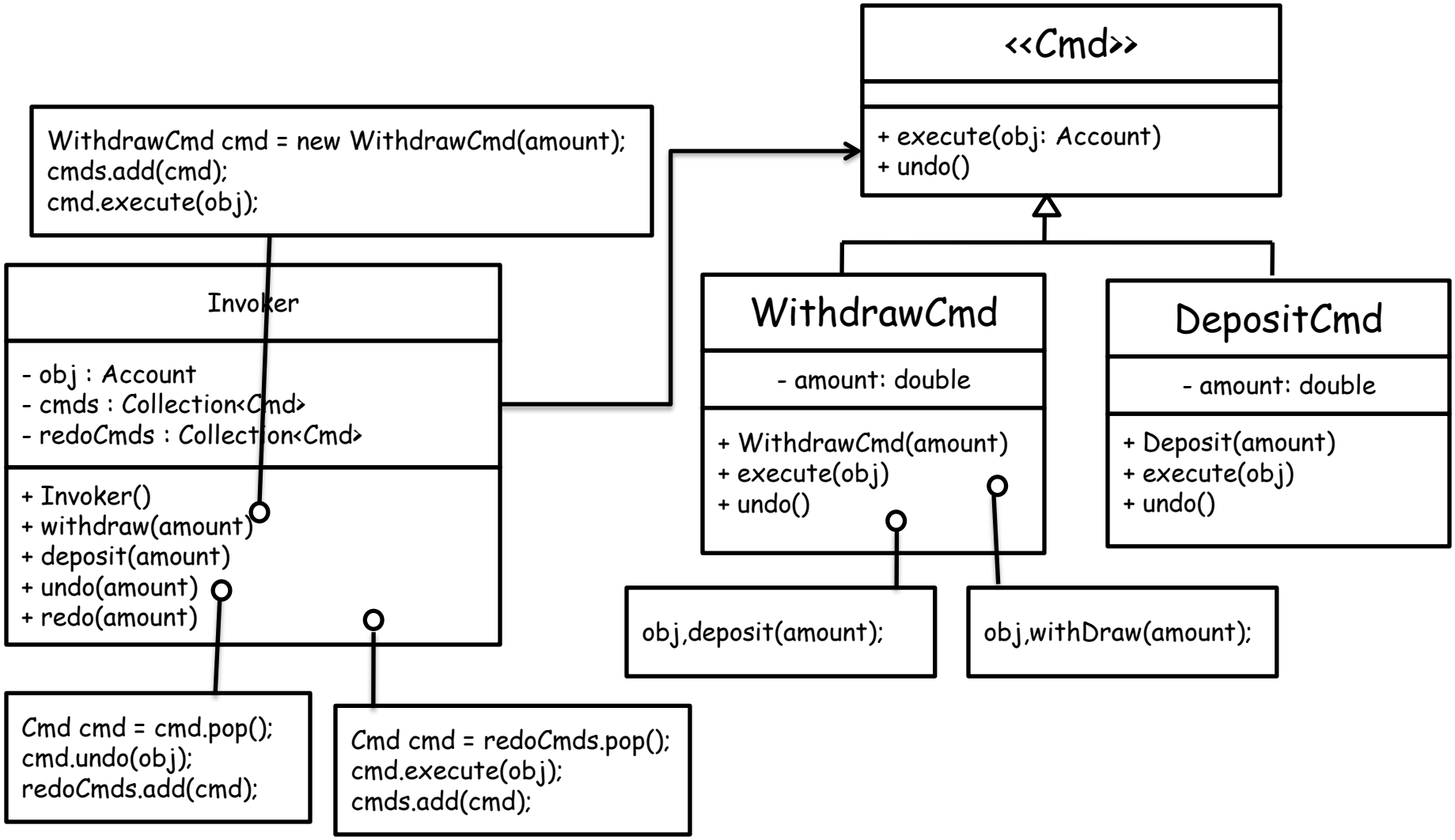
Applying FactoryMethod

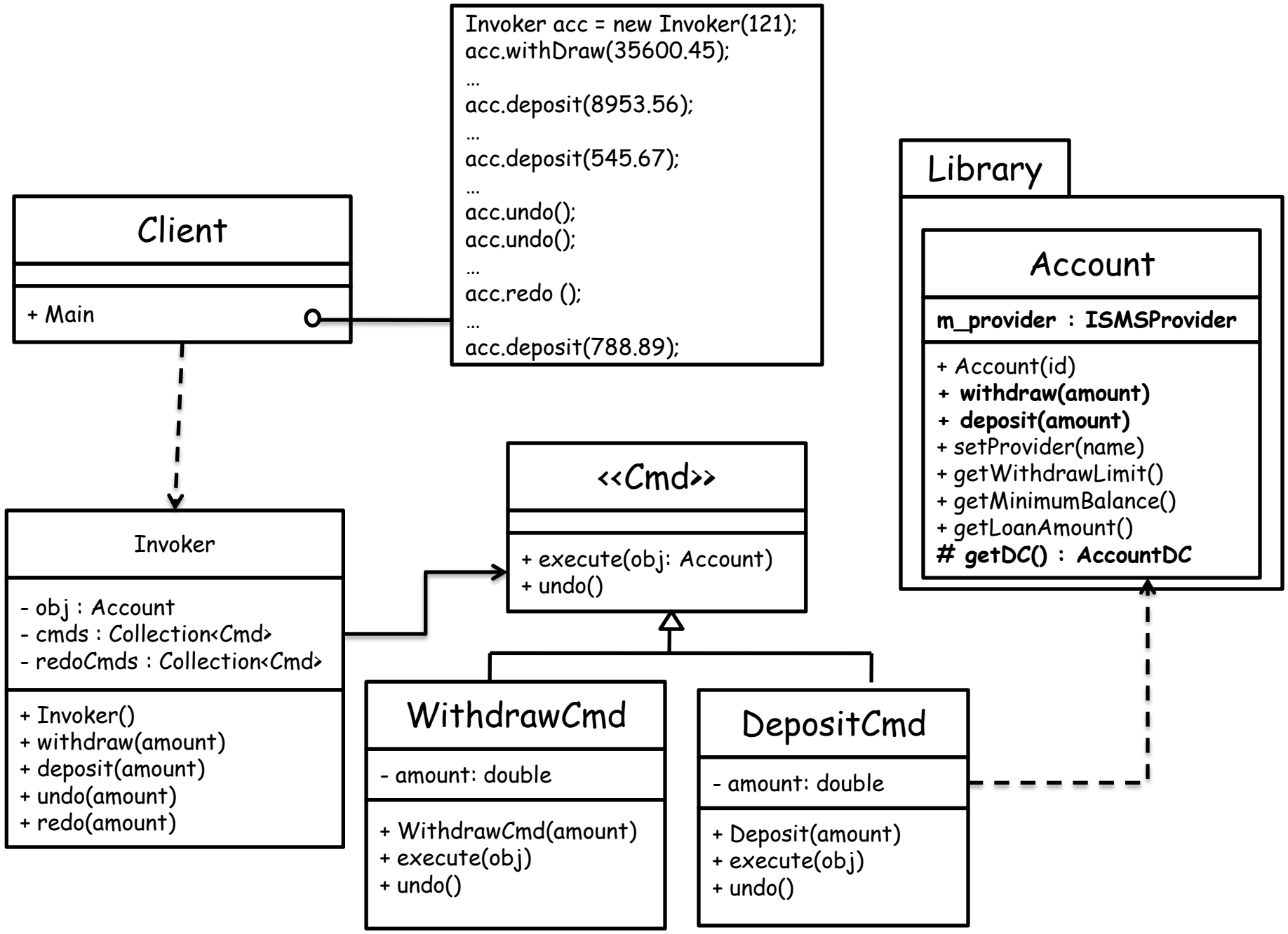


Long running transaction Problem

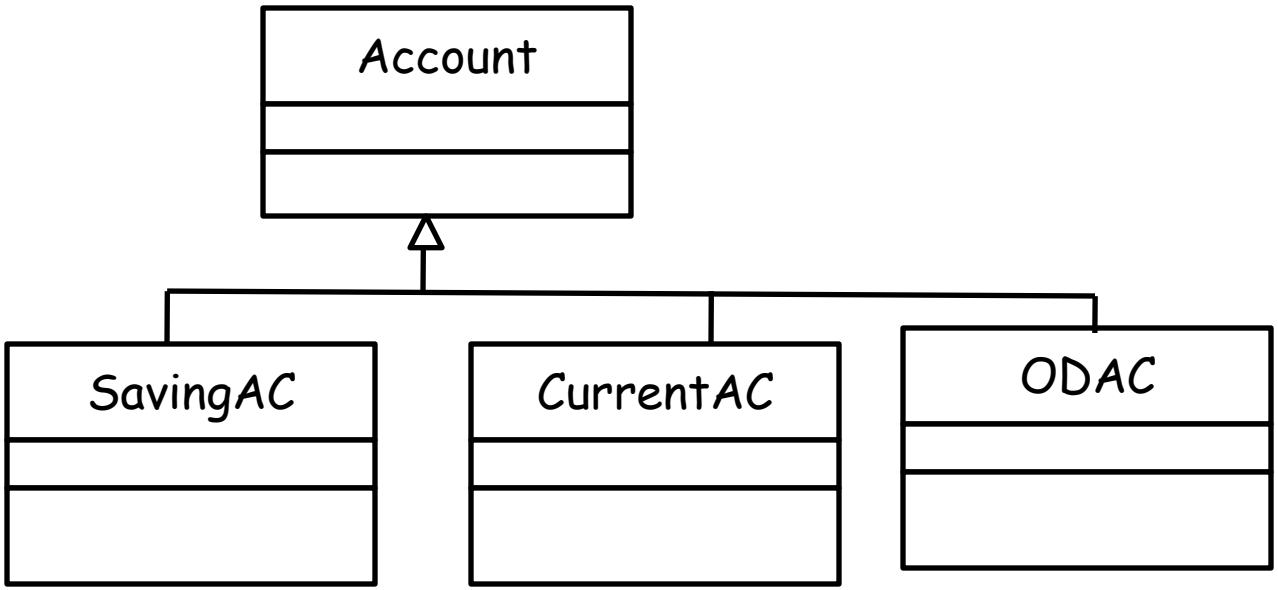


Applying Command



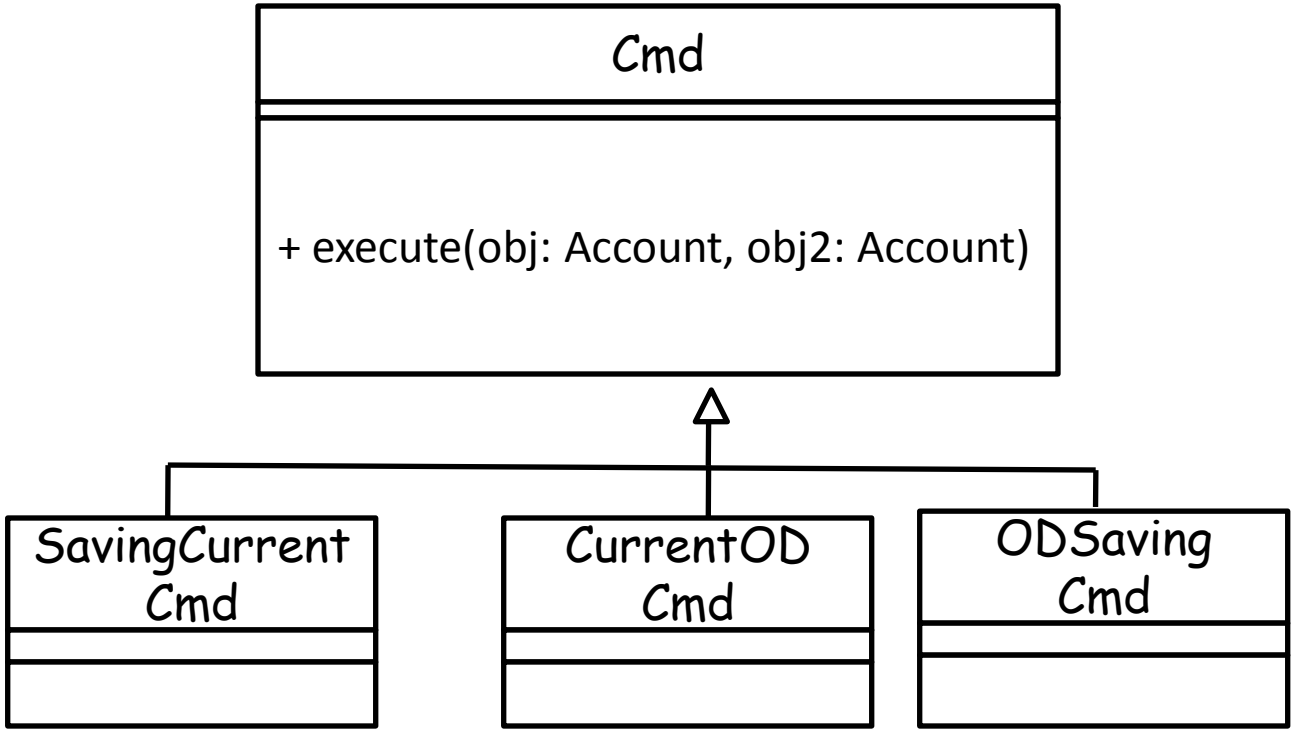

Invoker acc = new Invoker(121);
acc.withDraw(35600.45);
...
acc.deposit(8953.56);
...
acc.deposit(545.67);
...
acc.undo();
acc.undo();
...
acc.redo ();
...
acc.deposit(788.89);
Client
+ Main
Library
Account
m_provider : ISMSProvider
+ Account(id)
+ withdraw(amount)
+ deposit(amount)
+ setProvider(name)
+ 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



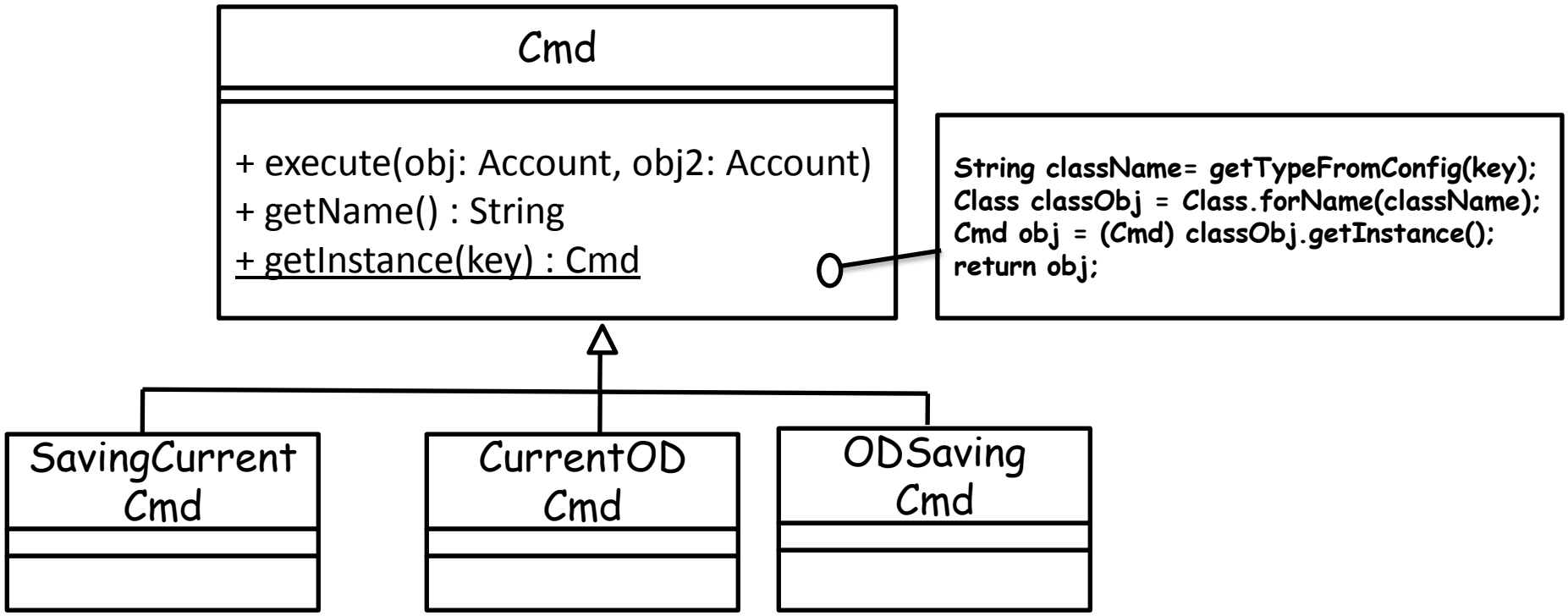

```
GetOffer(Account obj, Account obj2)
{
        If(typeof(obj)== typeof(SavingAC) && typeof(obj2) == typeof(CurrentAC))
                //…..
        If(typeof(obj)== typeof(ODAC) && typeof(obj2) == typeof(CurrentAC))
                //…..
        If(typeof(obj)== typeof(SavingAC) && typeof(obj2) ==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

Applying Command



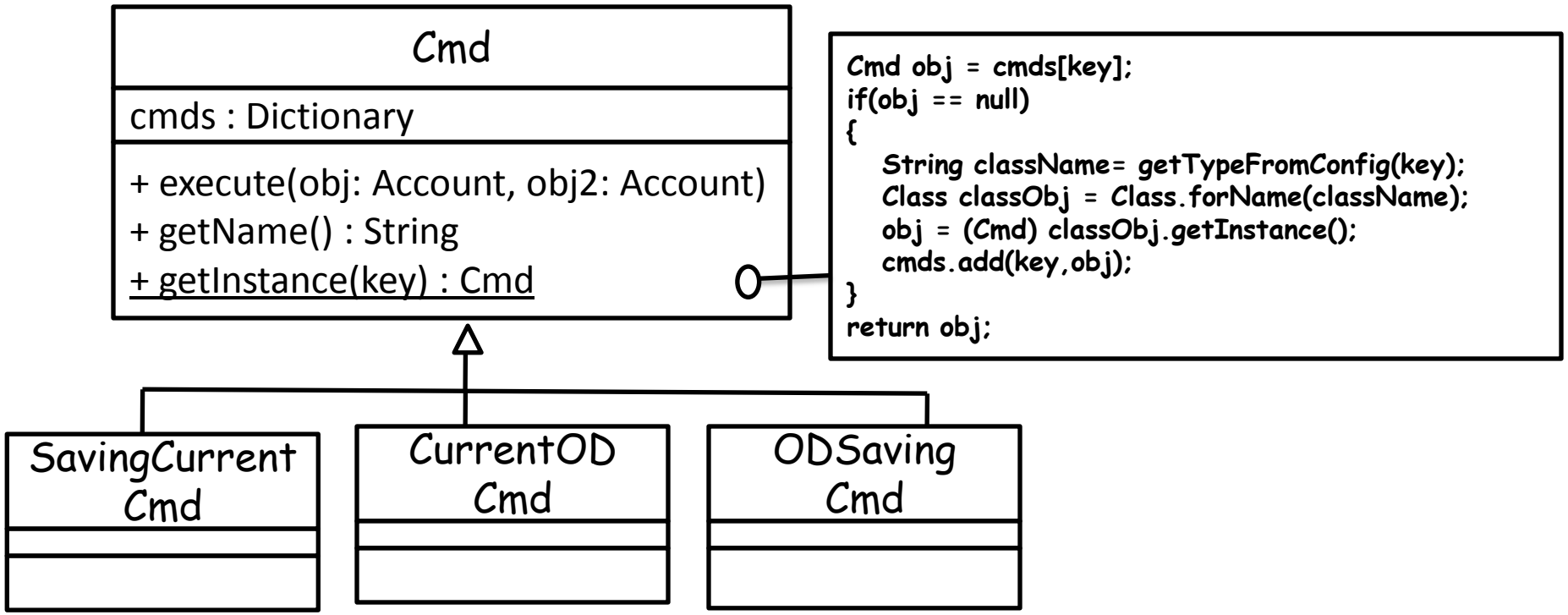

```
GetOffer(Account obj, Account obj2)
{
        Cmd  cmd = new ?;
        cmd. execute(obj,obj2);
}
```

Applying Creator Method



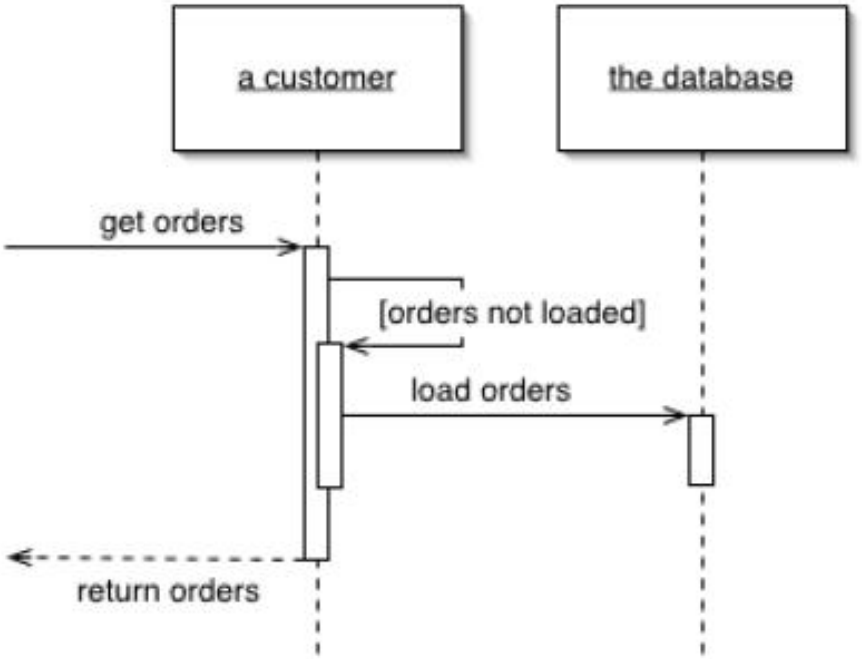

```
GetOffer(Account obj, Account obj2)
{
        String  key = obj.getName() + obj.getName();
        Cmd cmd= Cmd.getInstance(key);
        cmd. execute(obj,obj2);
}
```

Applying FlyWeight



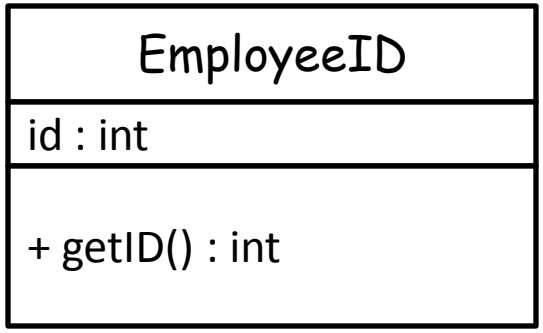

```
GetOffer(Account obj, Account obj2)
{
        String  key = obj.getName() + obj.getName();
        Cmd cmd= Cmd.getInstance(key);
        cmd. execute(obj,obj2);
}
```

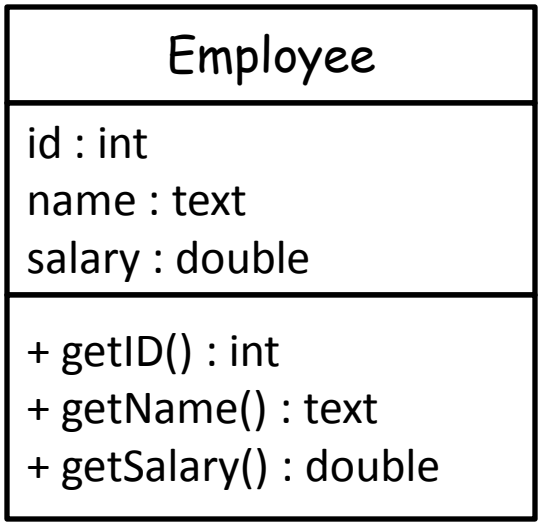
Identity Map



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

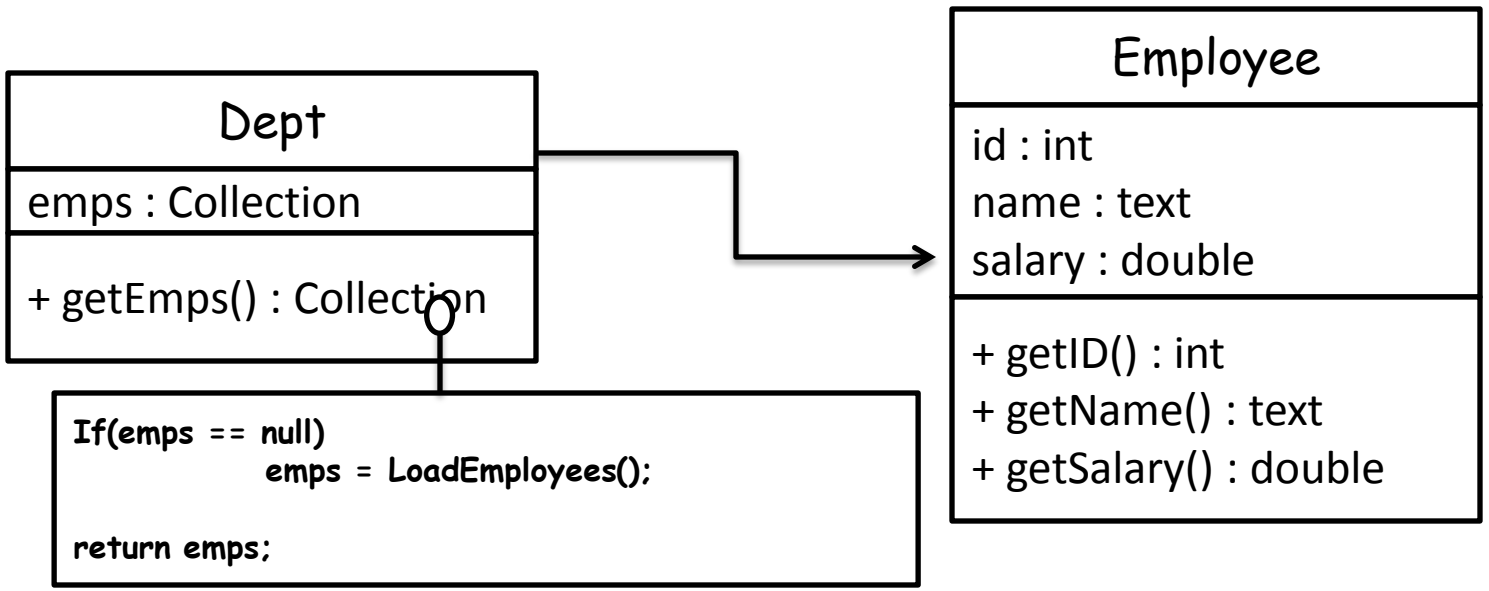
Identity Field





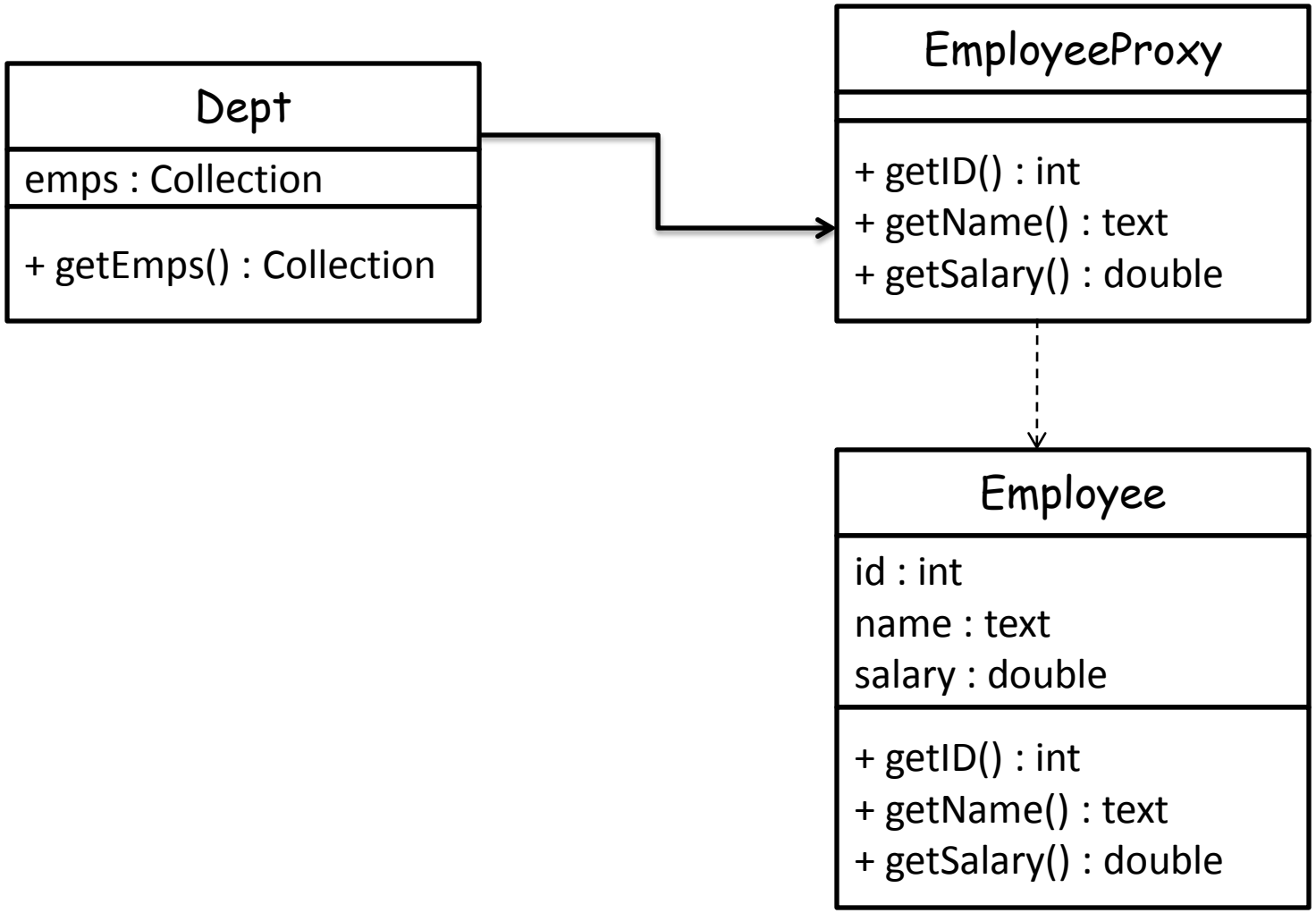
Save a database id field in an object to maintain identity between an in-memory object and a database row.

Lazy Loading

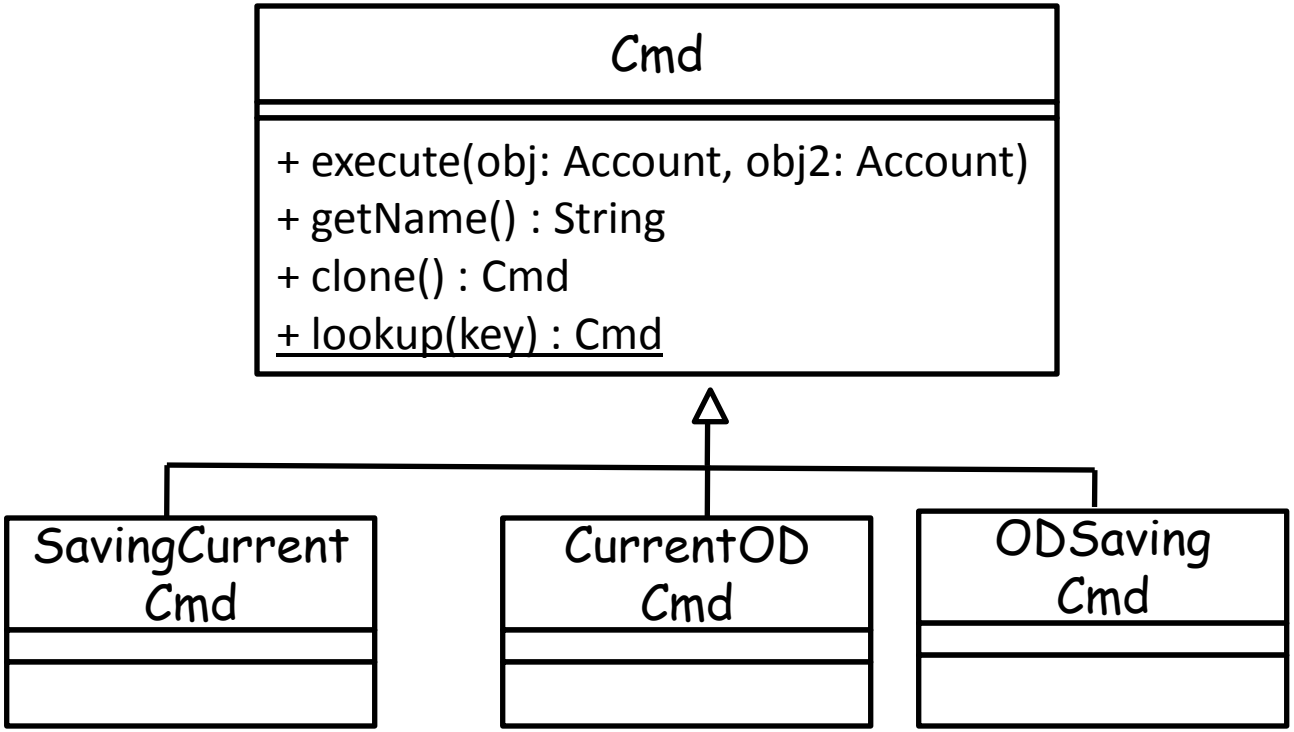


An object that doesn't contain all of the data you need, but knows how to get it.

Lazy Loading with Proxy

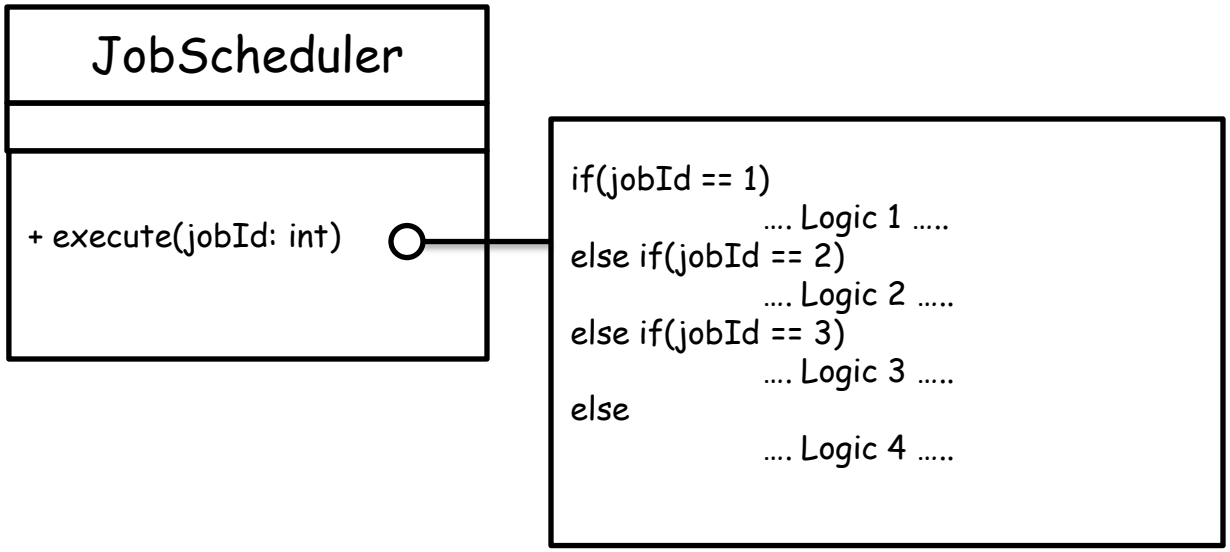


Applying Prototype

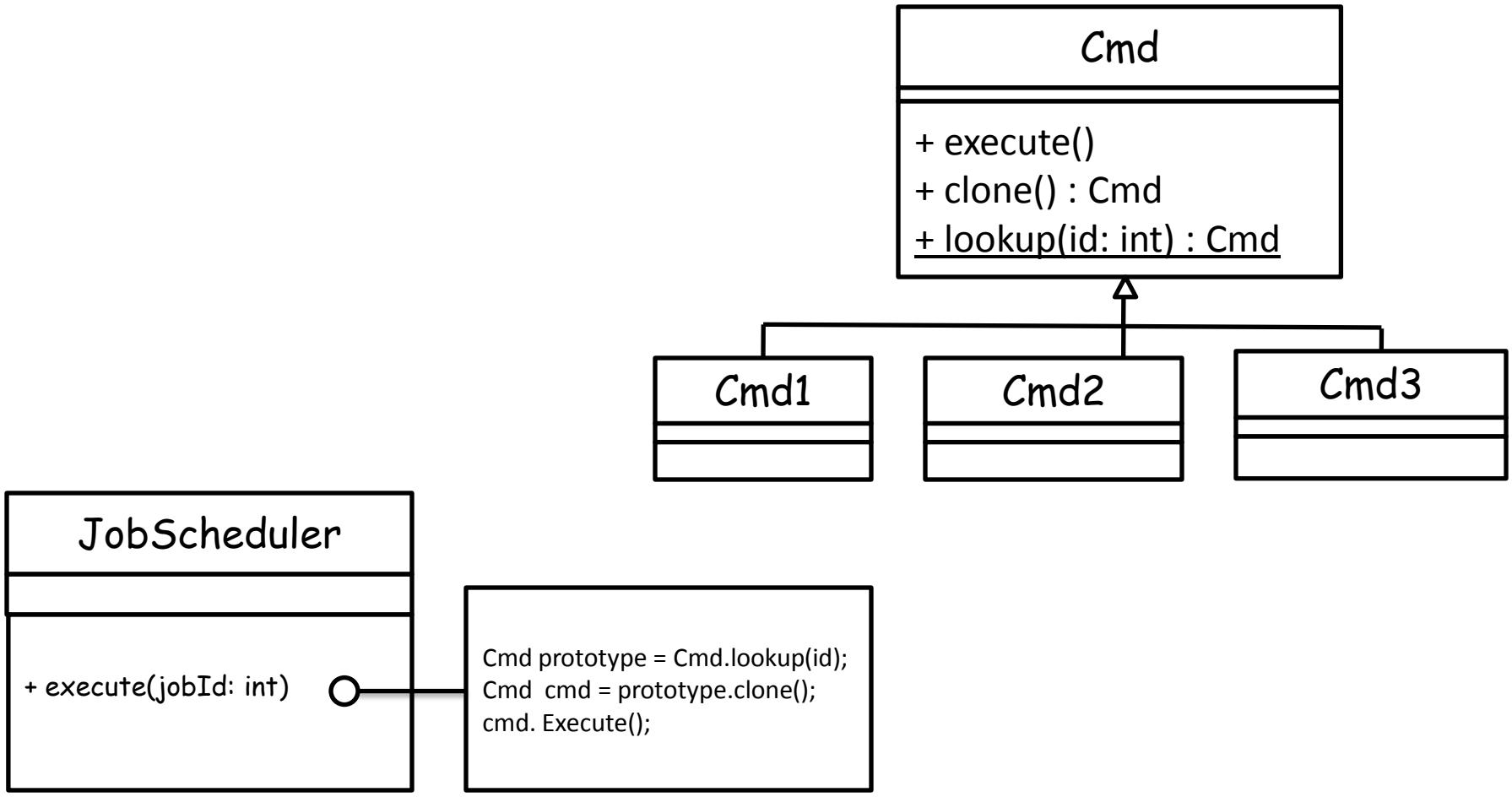



```
GetOffer(Account obj, Account obj2)
{
        String  key = obj.getName() + obj.getName();
        Cmd prototype = Cmd.lookup(key);
        Cmd  cmd = prototype.clone();
        cmd. execute(obj,obj2);
}
```

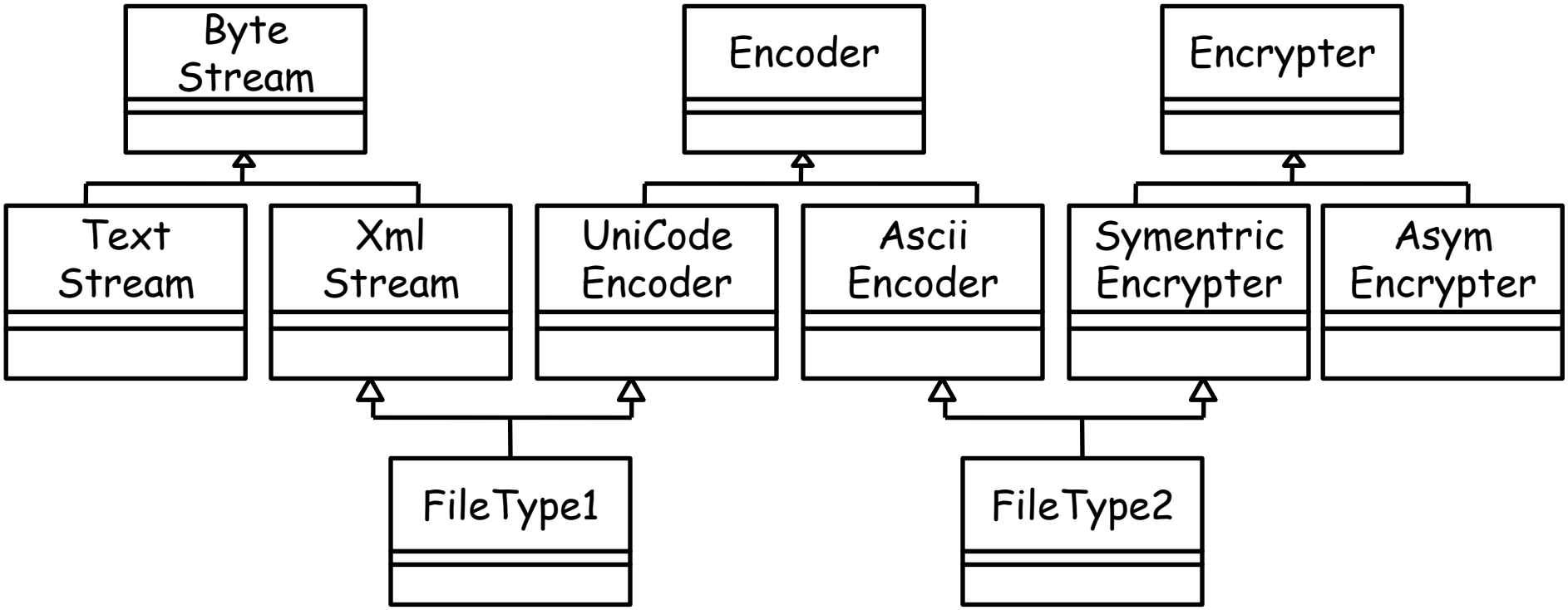
Problem


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

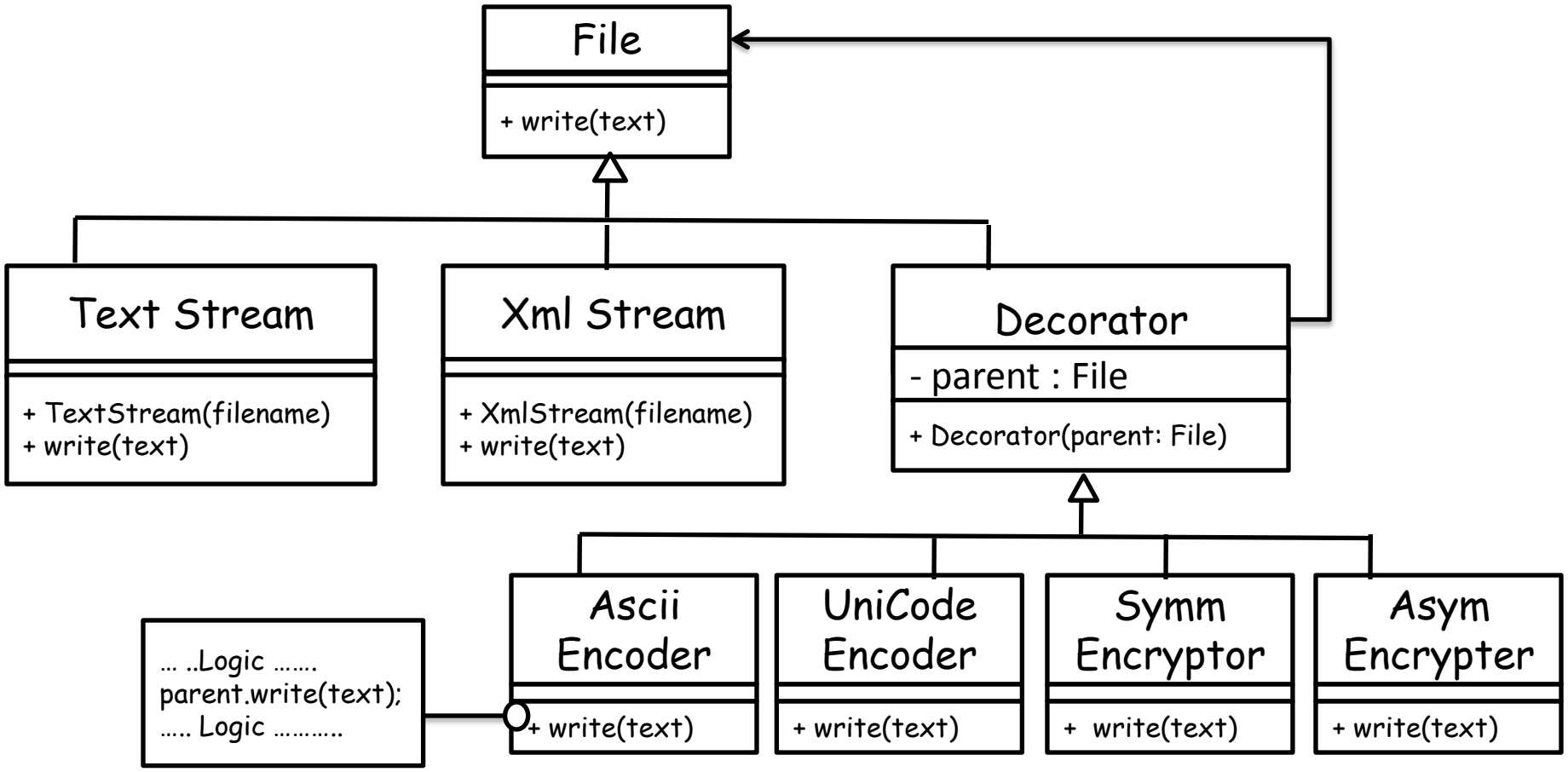
Applying Command



Multiple Inheritance problem



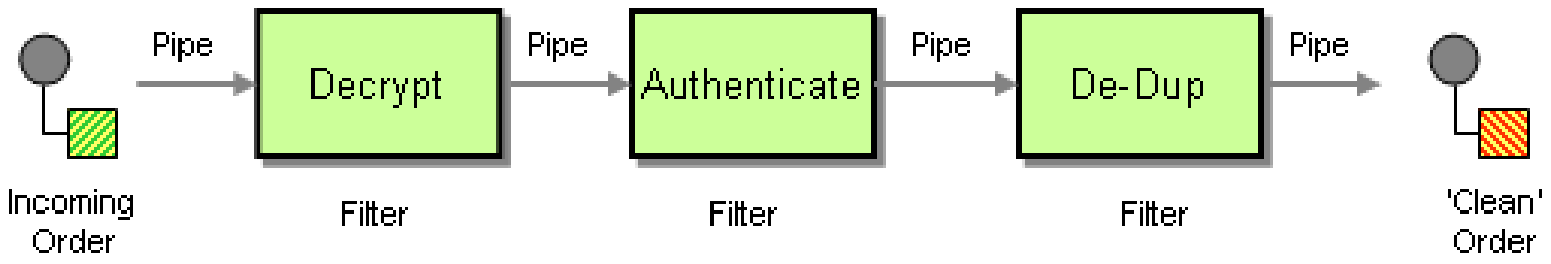
Applying Decorator



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

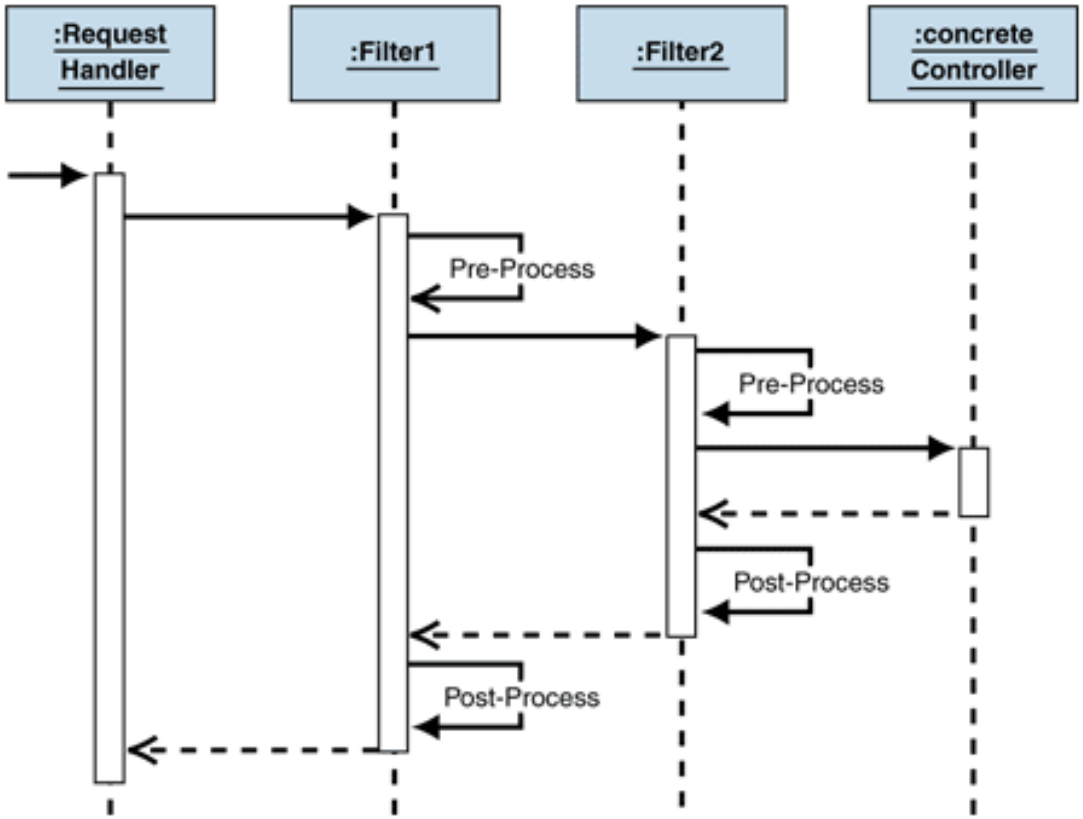
Pipeline Pattern

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](#).

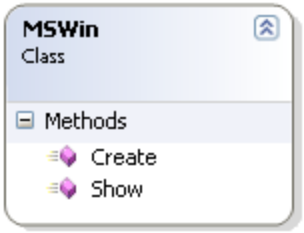
Intercepting Filter

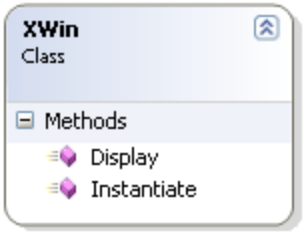


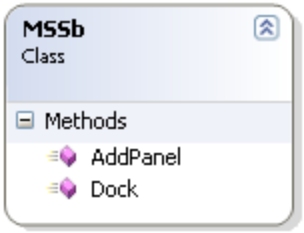
How do you implement common pre- and post-processing steps around Web page requests?

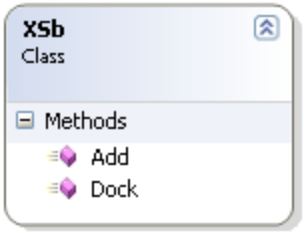
Toggle between Implementation

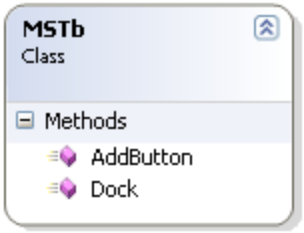
```
MSWin w = new MSWin();
w.Create();
w.Show();

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

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

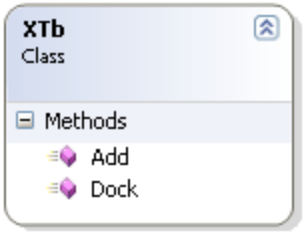






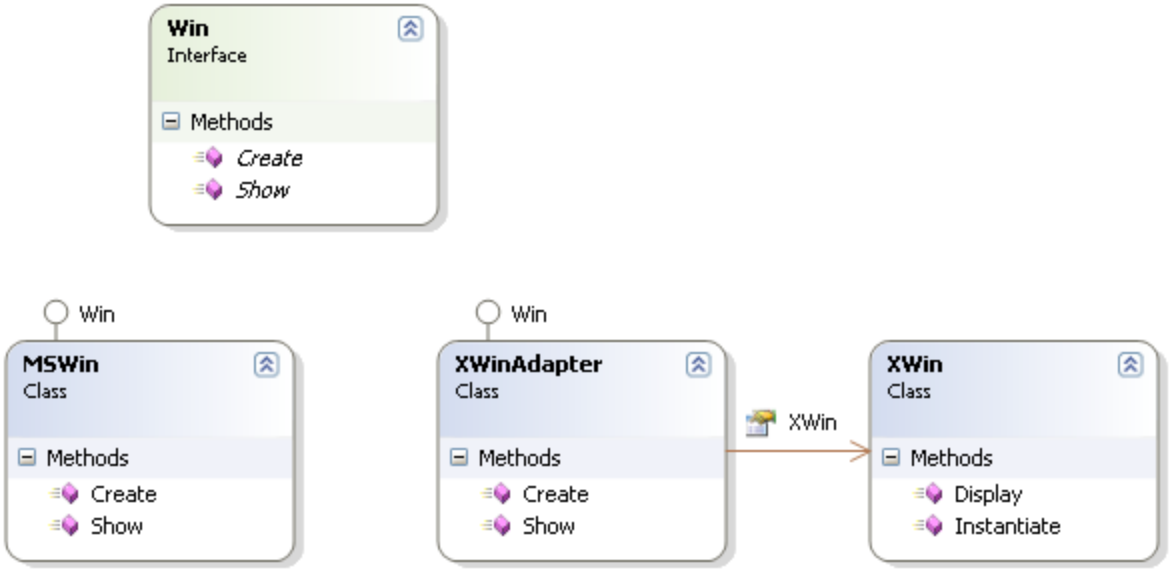






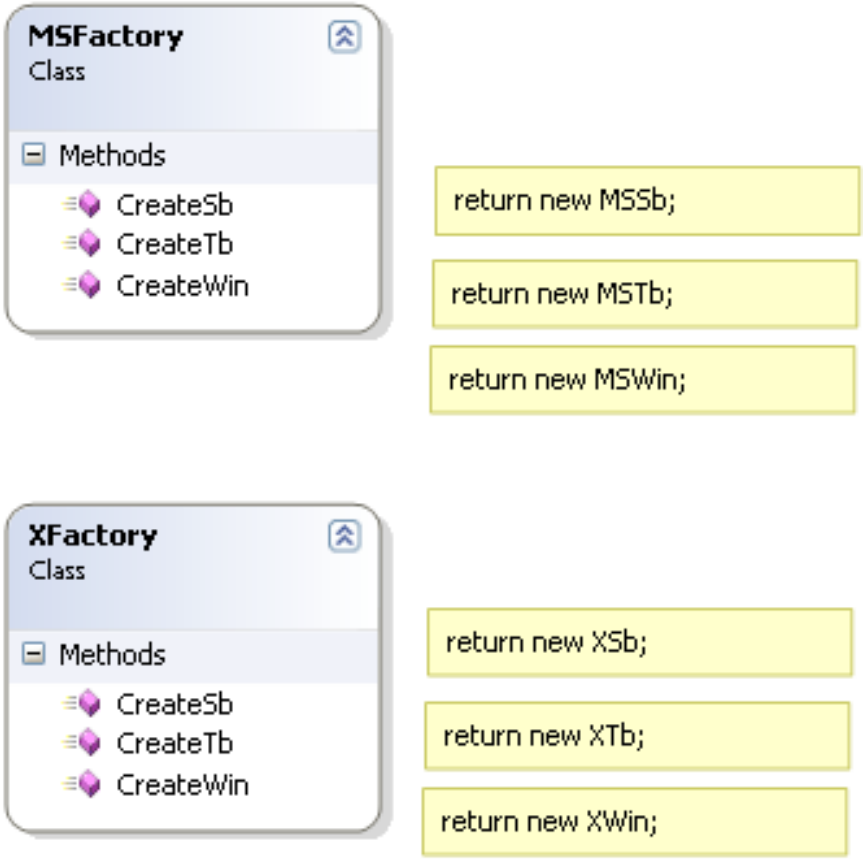
Applying Adapter

```
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();
```



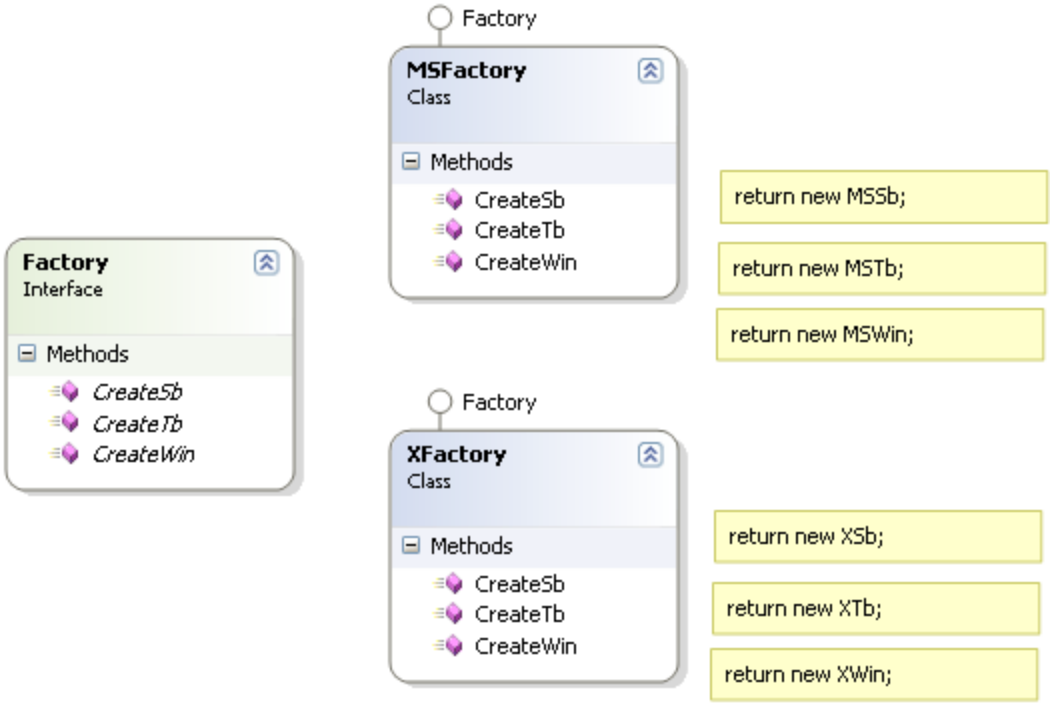
Applying ClassFactory

```
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();
```



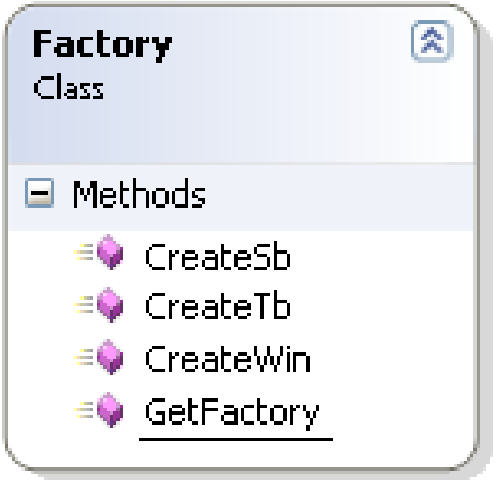
Applying Abstract Factory

```
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;
```

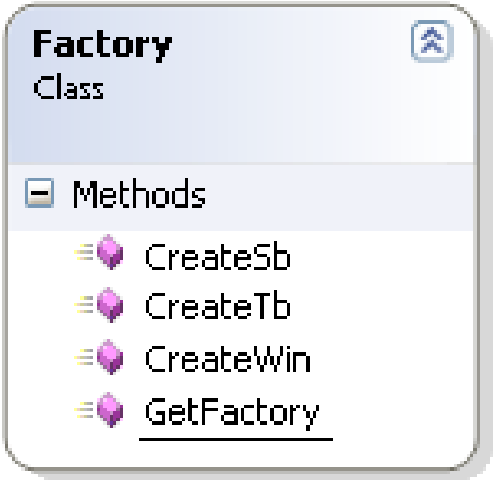
Protected Variations

- 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();
```



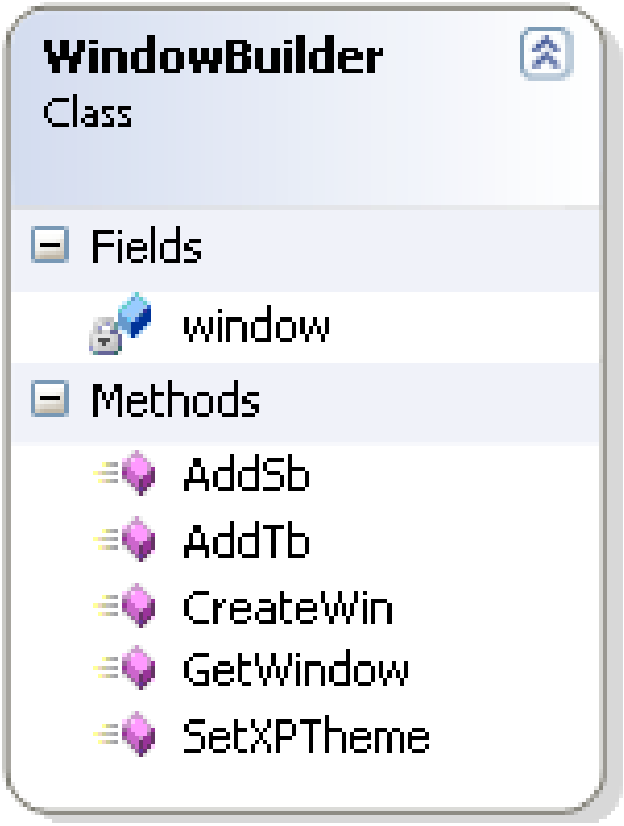

```
Factory* f= lookup(type);
return f;
```

Singleton


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

 return m_instance;
```

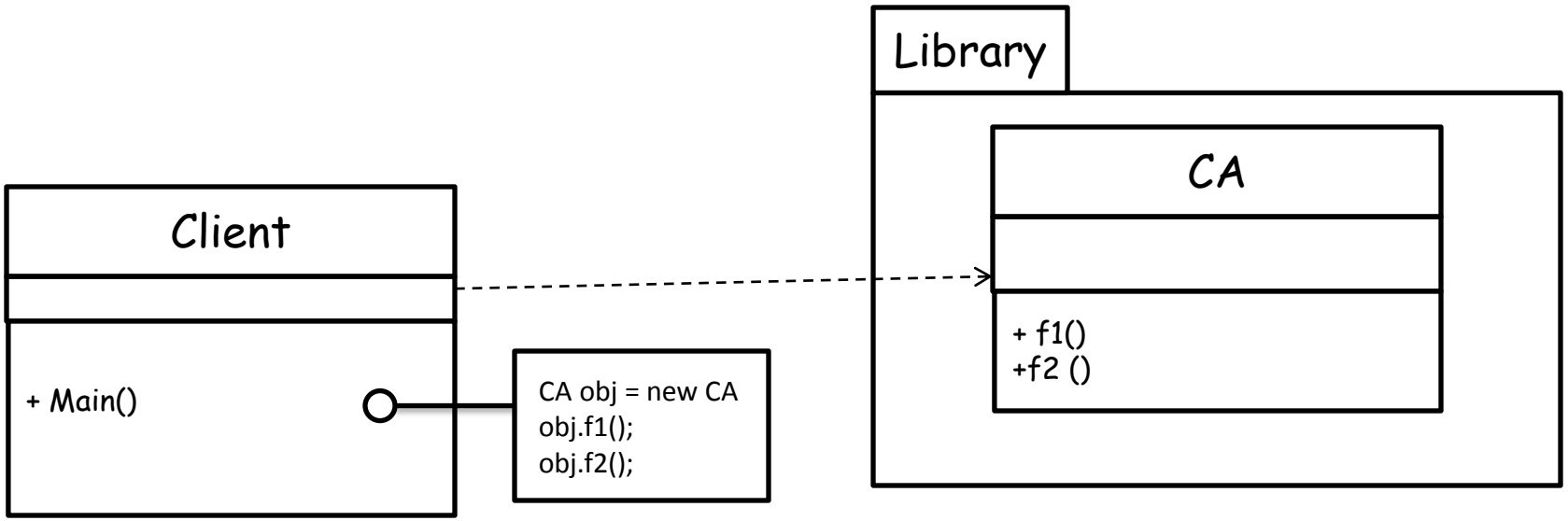
Builder

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

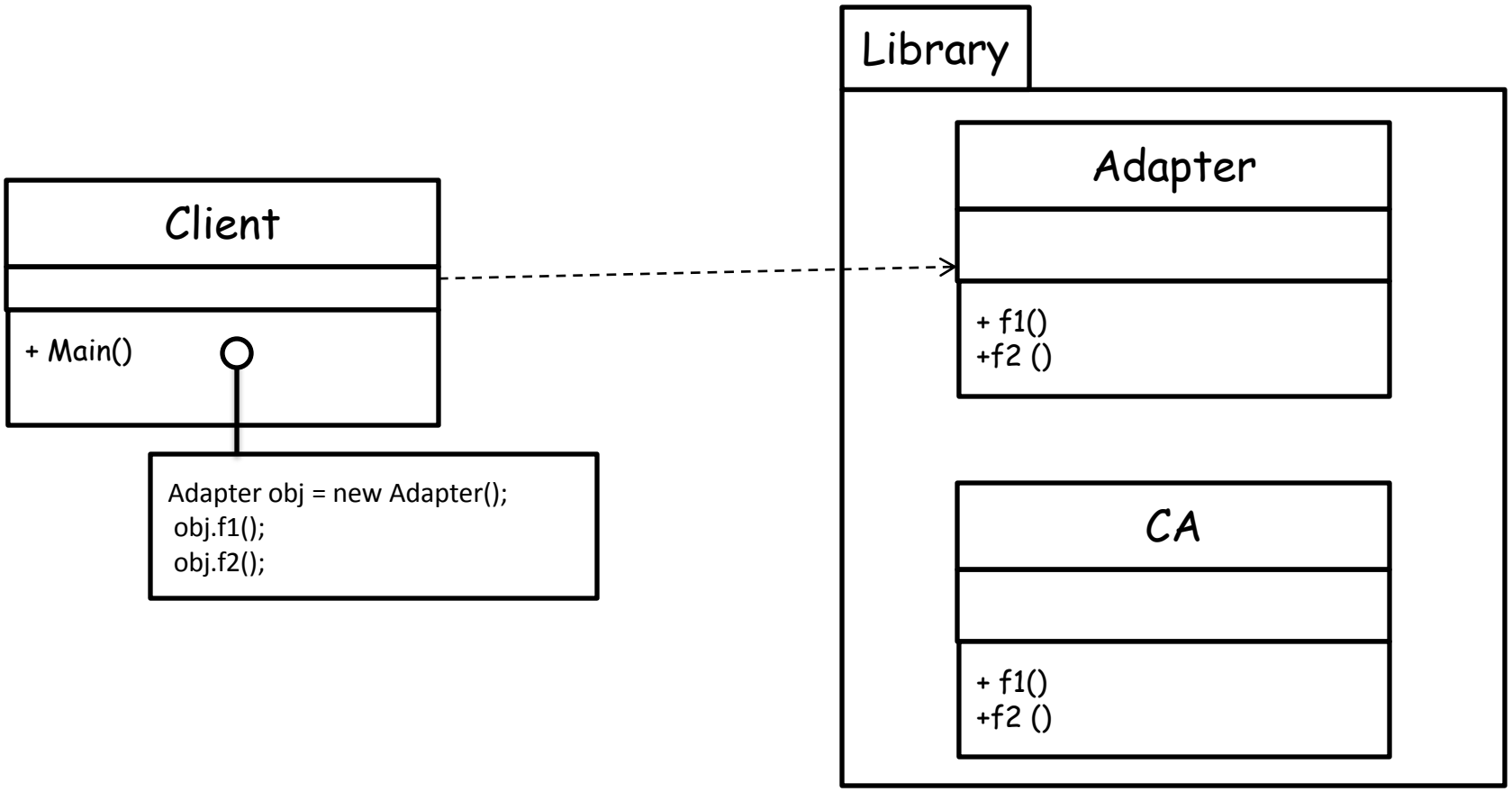


Abstract steps of construction of objects

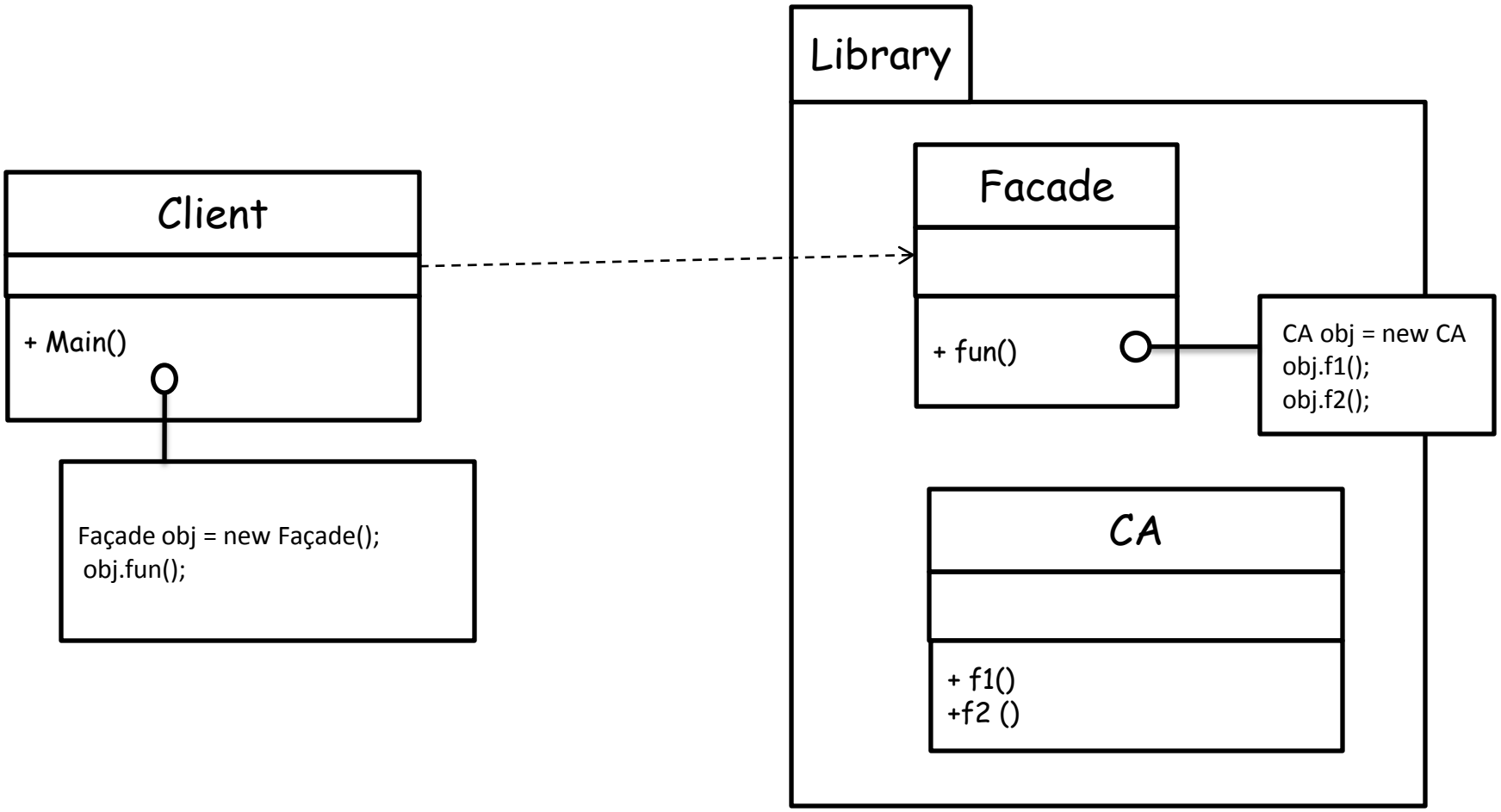
Low Coupling problem



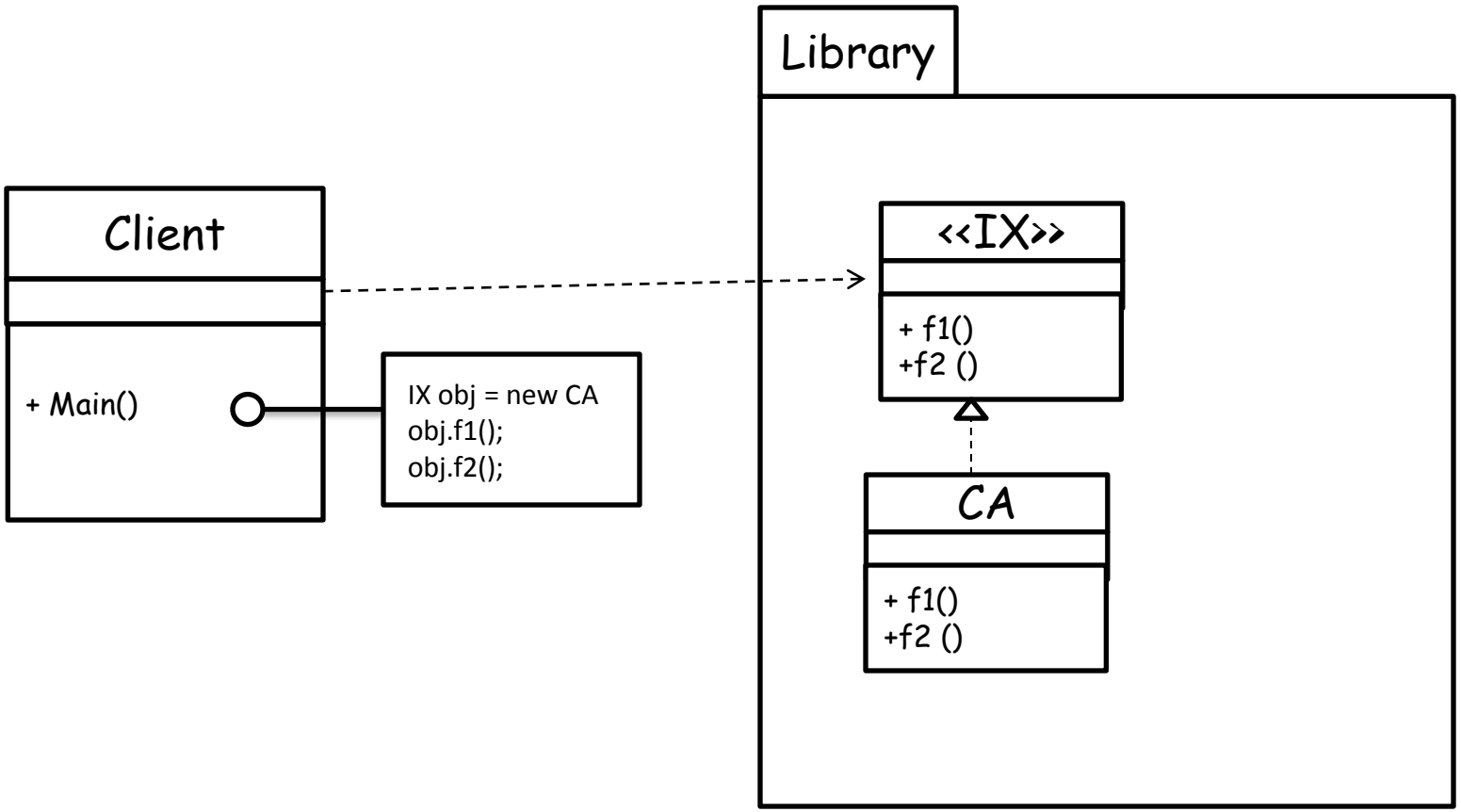
Applying Adapter



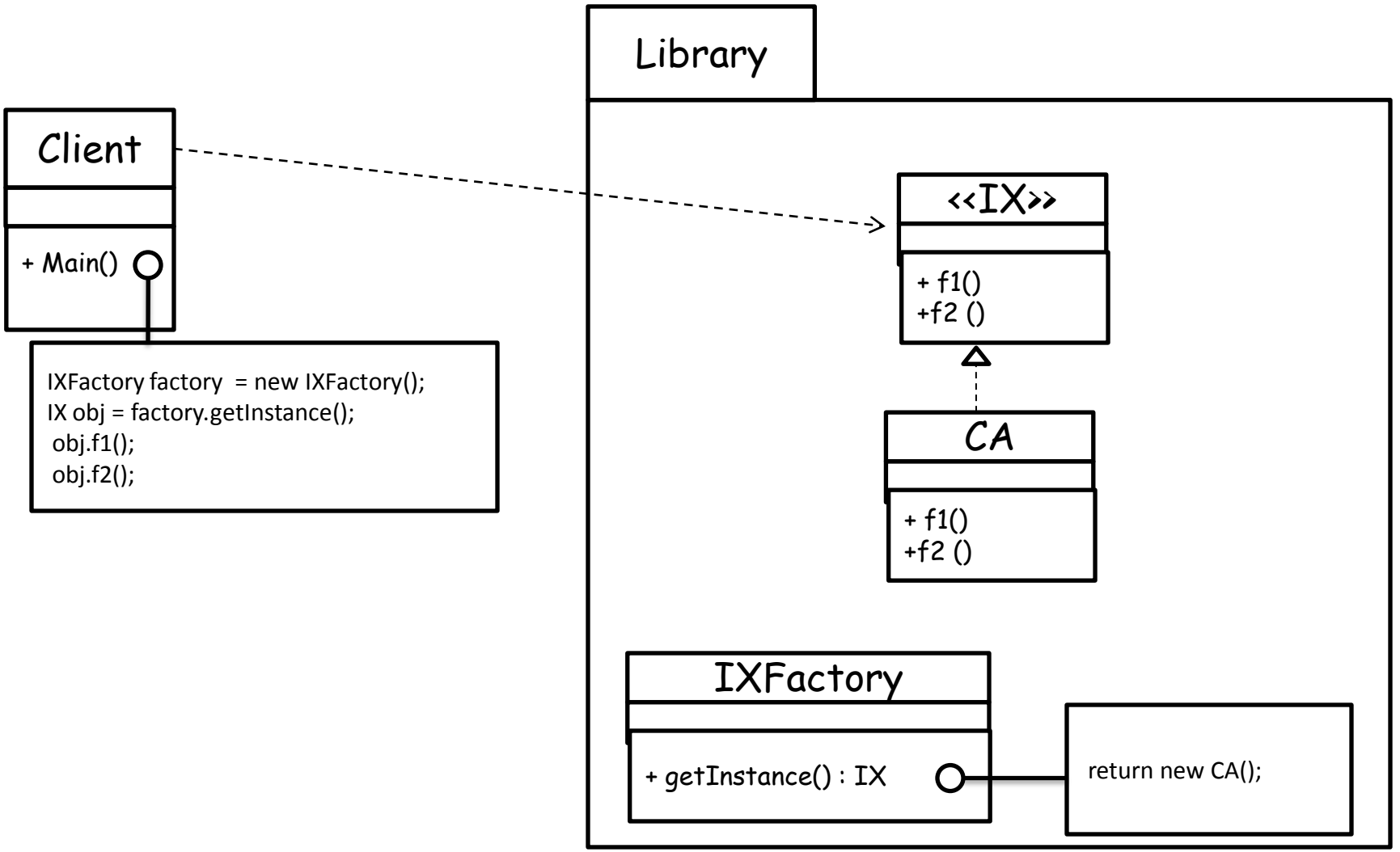
Applying Facade



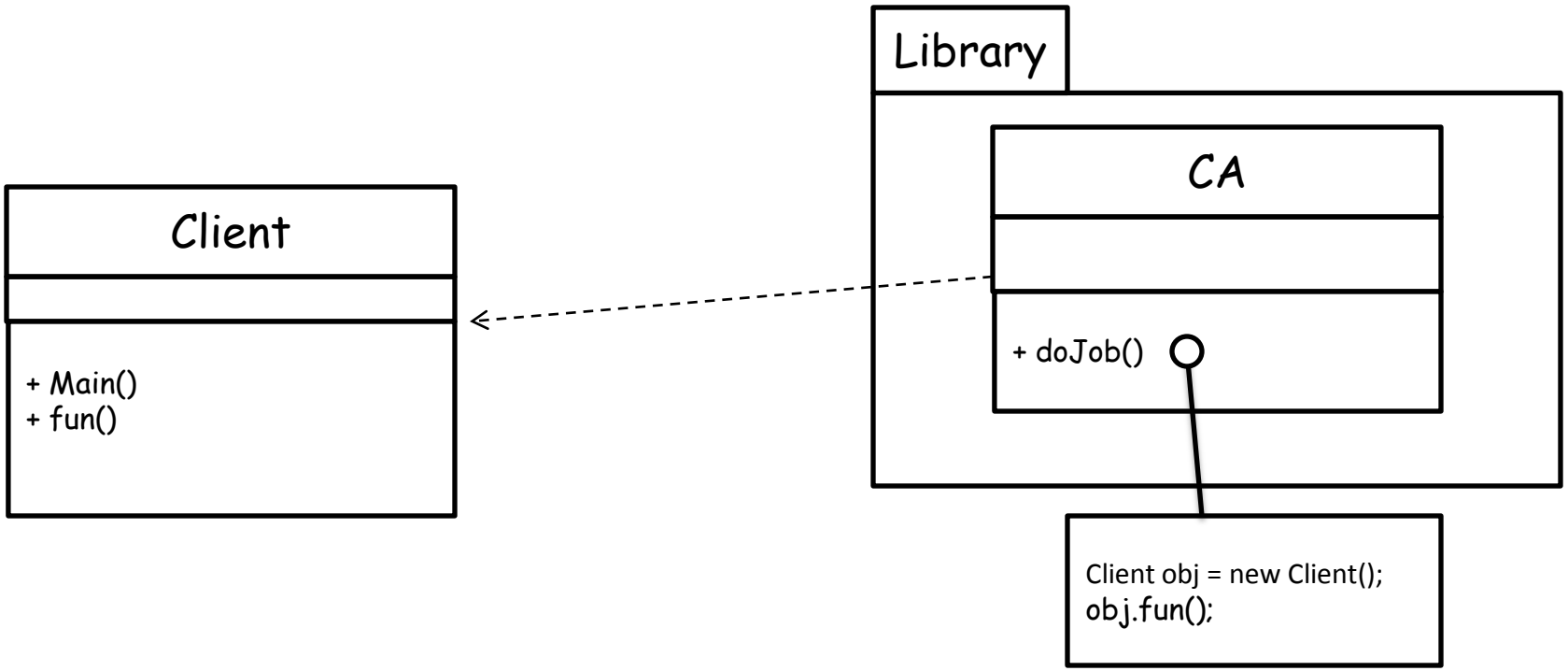
Applying interface pattern



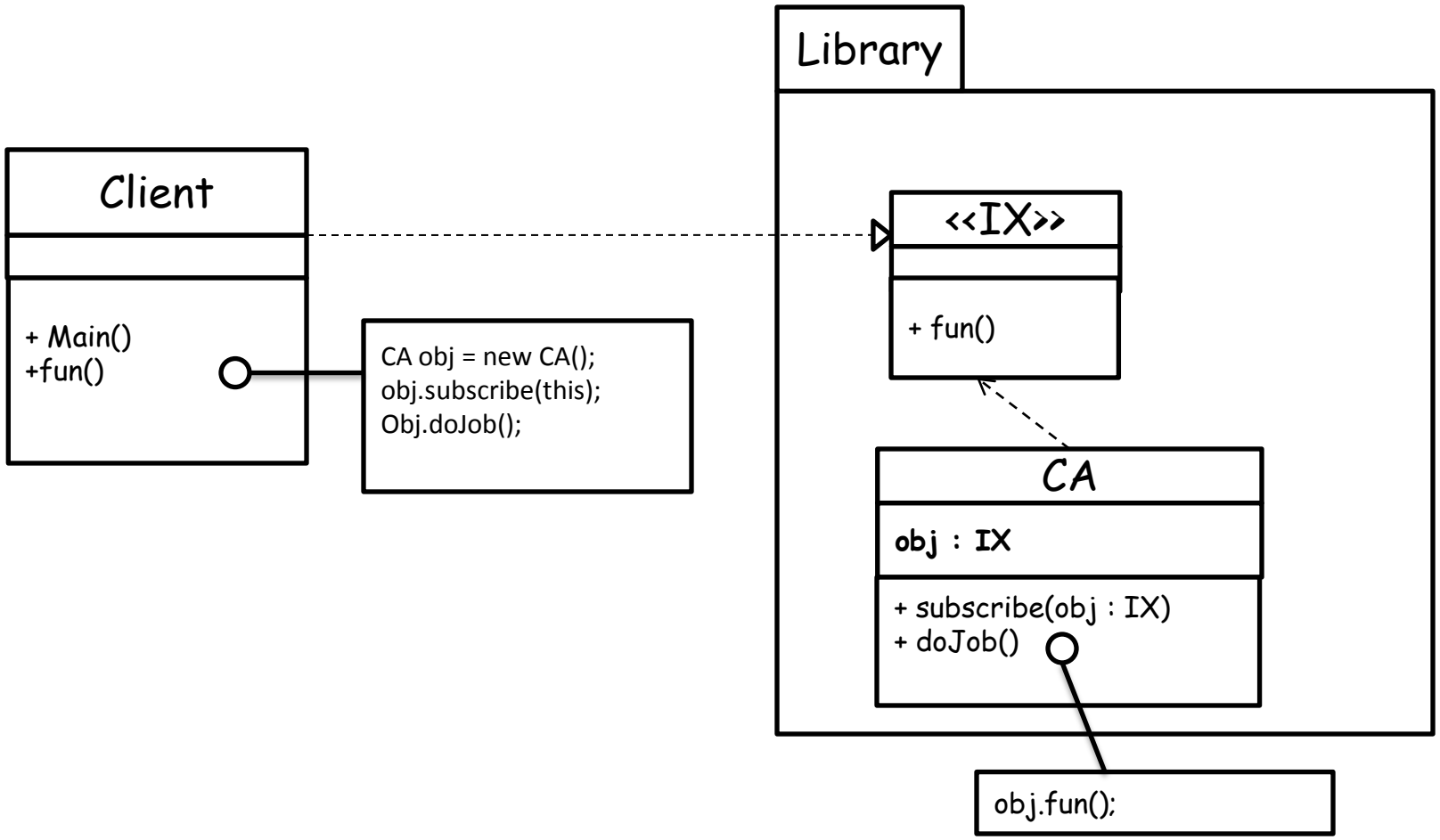
Applying Class Factory



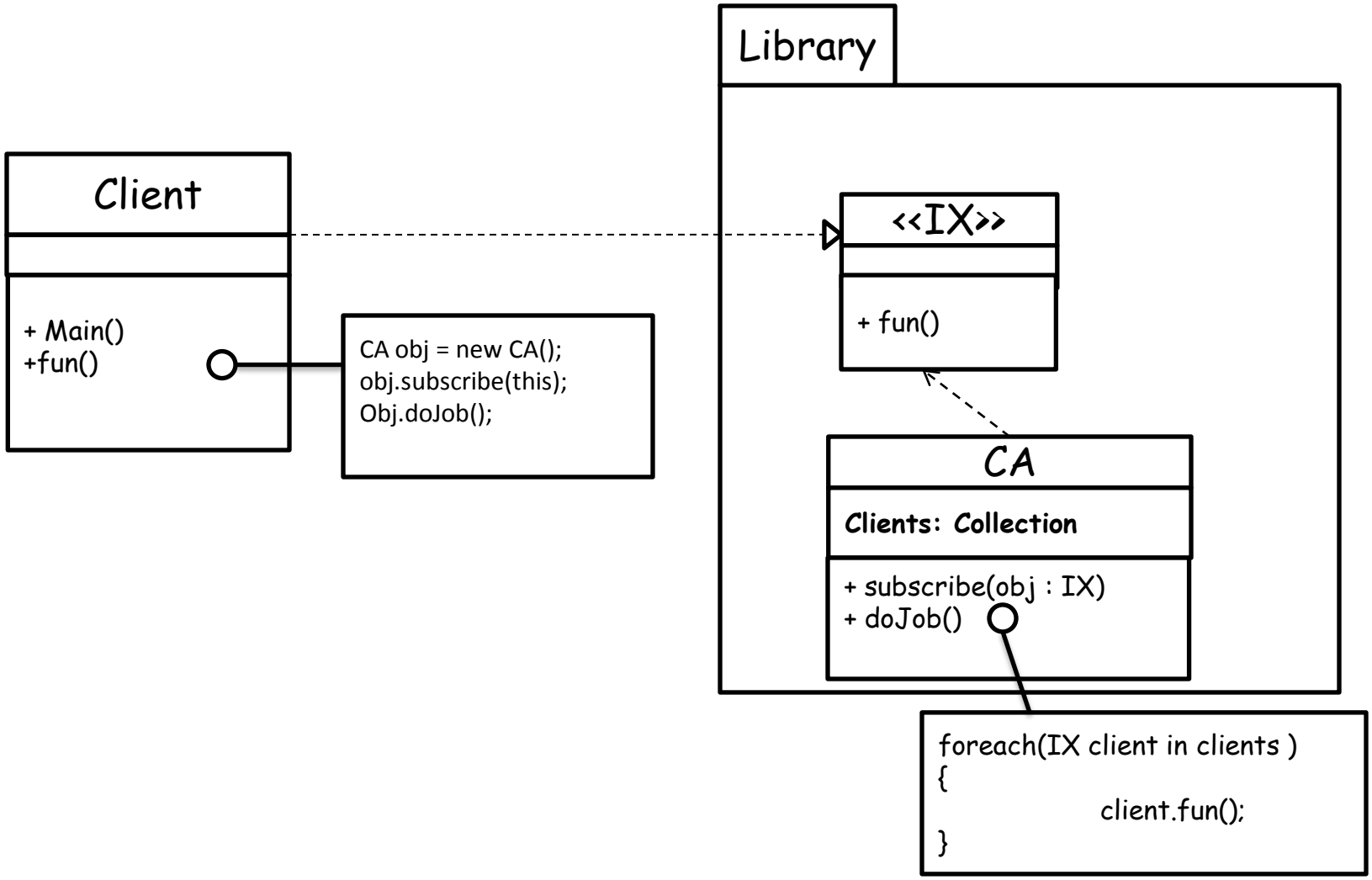
Callback Coupling Problem



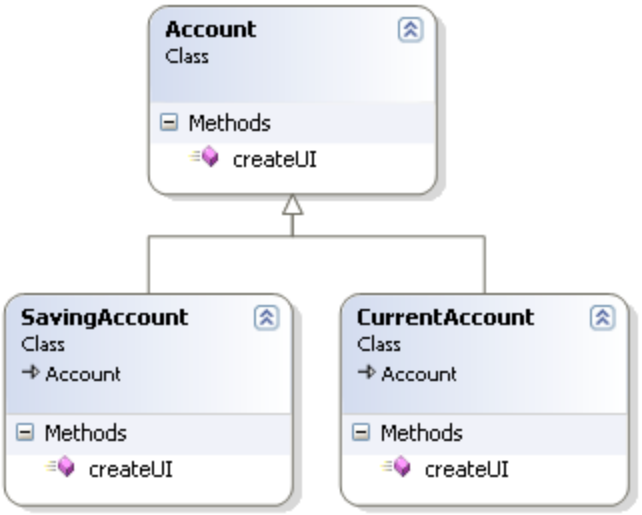
Applying interface pattern

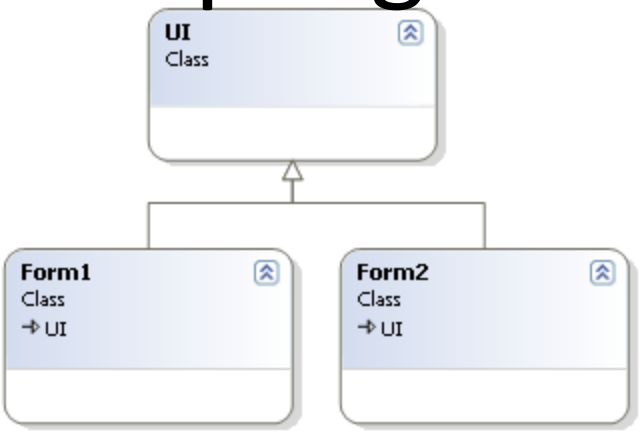


Applying Observer



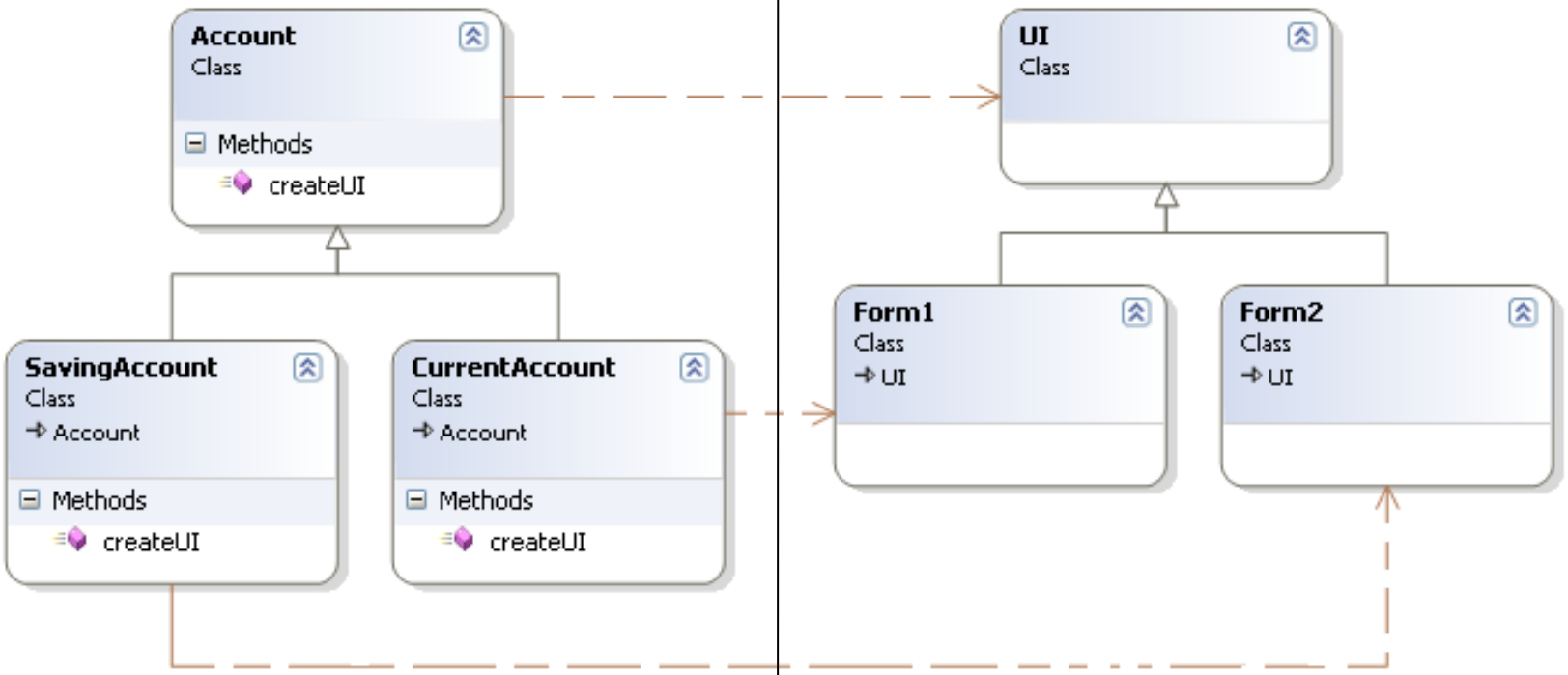
Low Coupling






```
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();
}
```

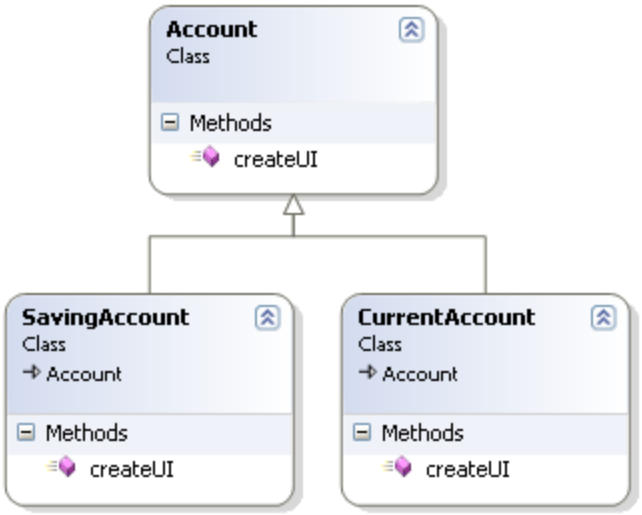
Applying FactoryMethod

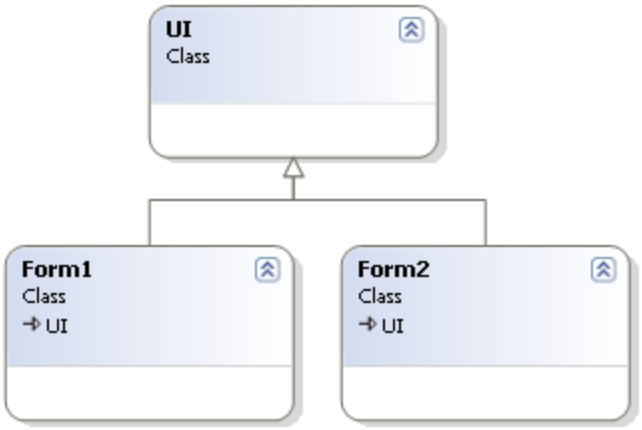



```
void createUI()
{
   return new Form1;
}
```

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

Applying Static Polymorphism

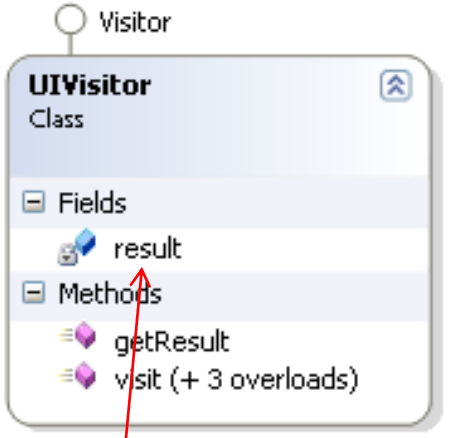





```
void ShowUI(Account obj)
{
        ....
}
void ShowUI (SavingAccount obj)
{
        ...
}
void ShowUI (CurrentAccount obj)
{
        ....
}
```

Applying Visitor

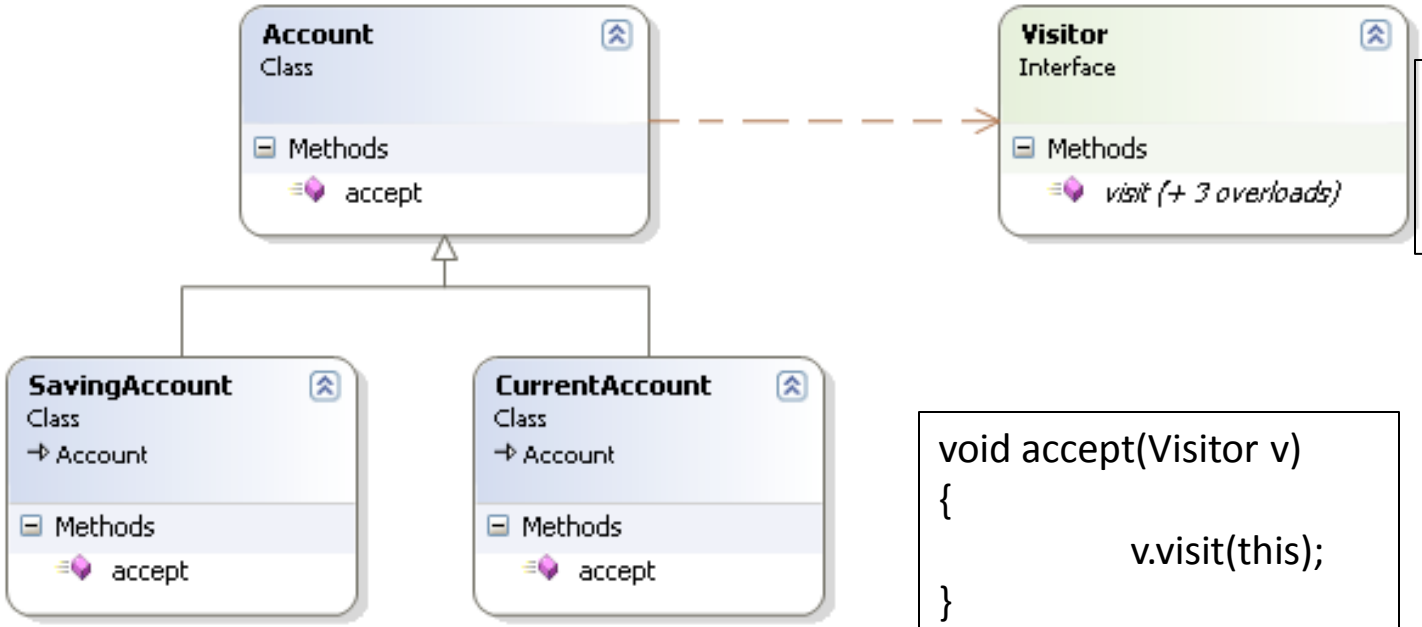

```
void visit(Account obj)
void visit(SavingAccount obj)
void visit(CurrentAccount obj)
```




```
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();
}
```

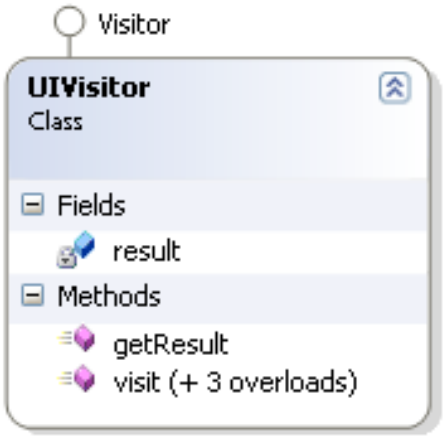
Applying Visitor



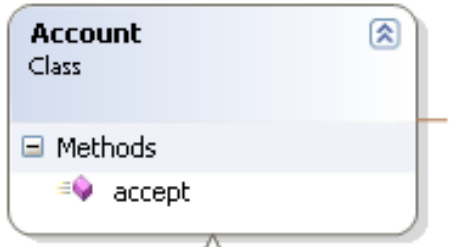
visit(Account)
visit(SavingAccount)
visit(CurrentAccount)

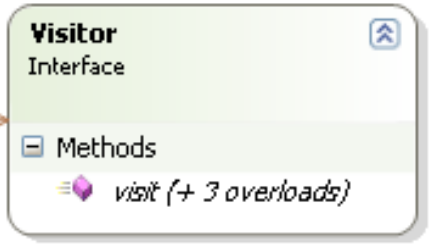
```
void accept(Visitor v)
{
        v.visit(this);
}
```

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

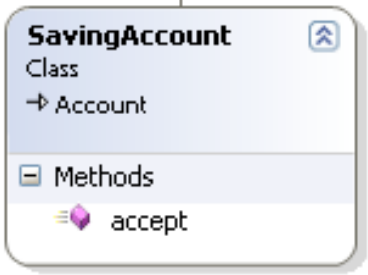


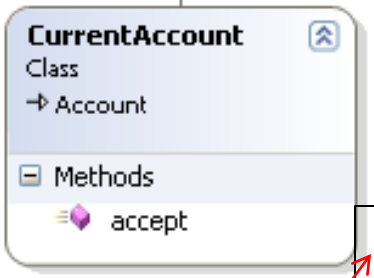

```
visit(Account){
        //.. code;
}
visit(SavingAccount){
        //.. code;
}
visit(CurrentAccount){
        //.. code;
}
```



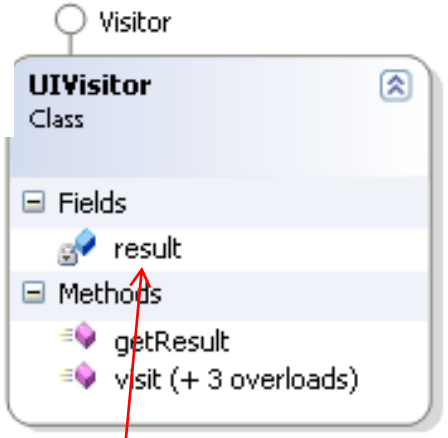



```
void visit(Account obj)
void visit(SavingAccount obj)
void visit(CurrentAccount obj)
```





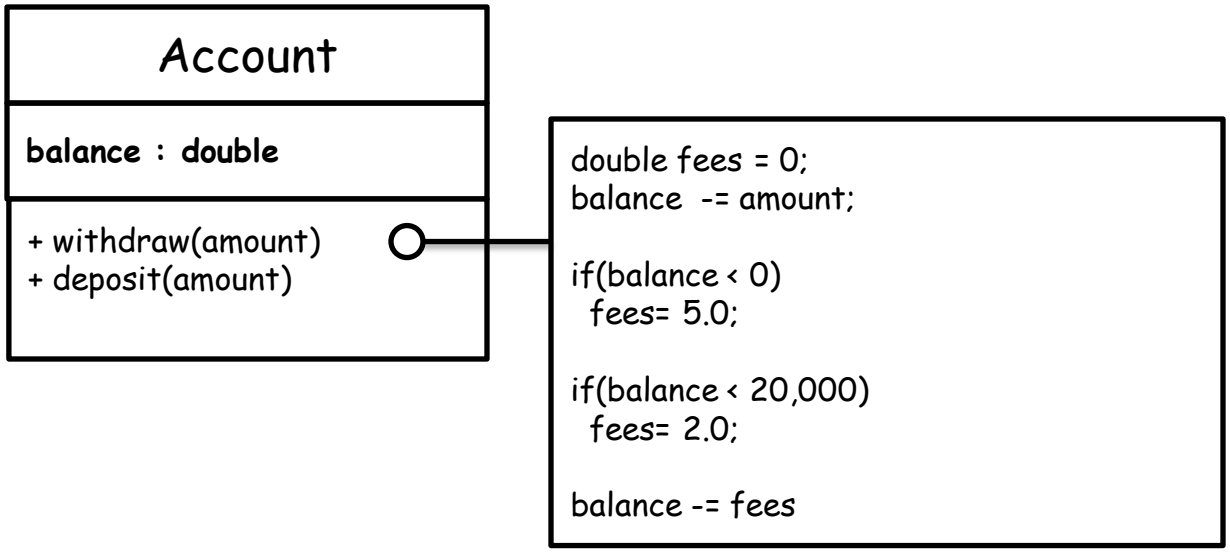

```
void accept(Visitor v)
{
        v.visit(this);
}
```



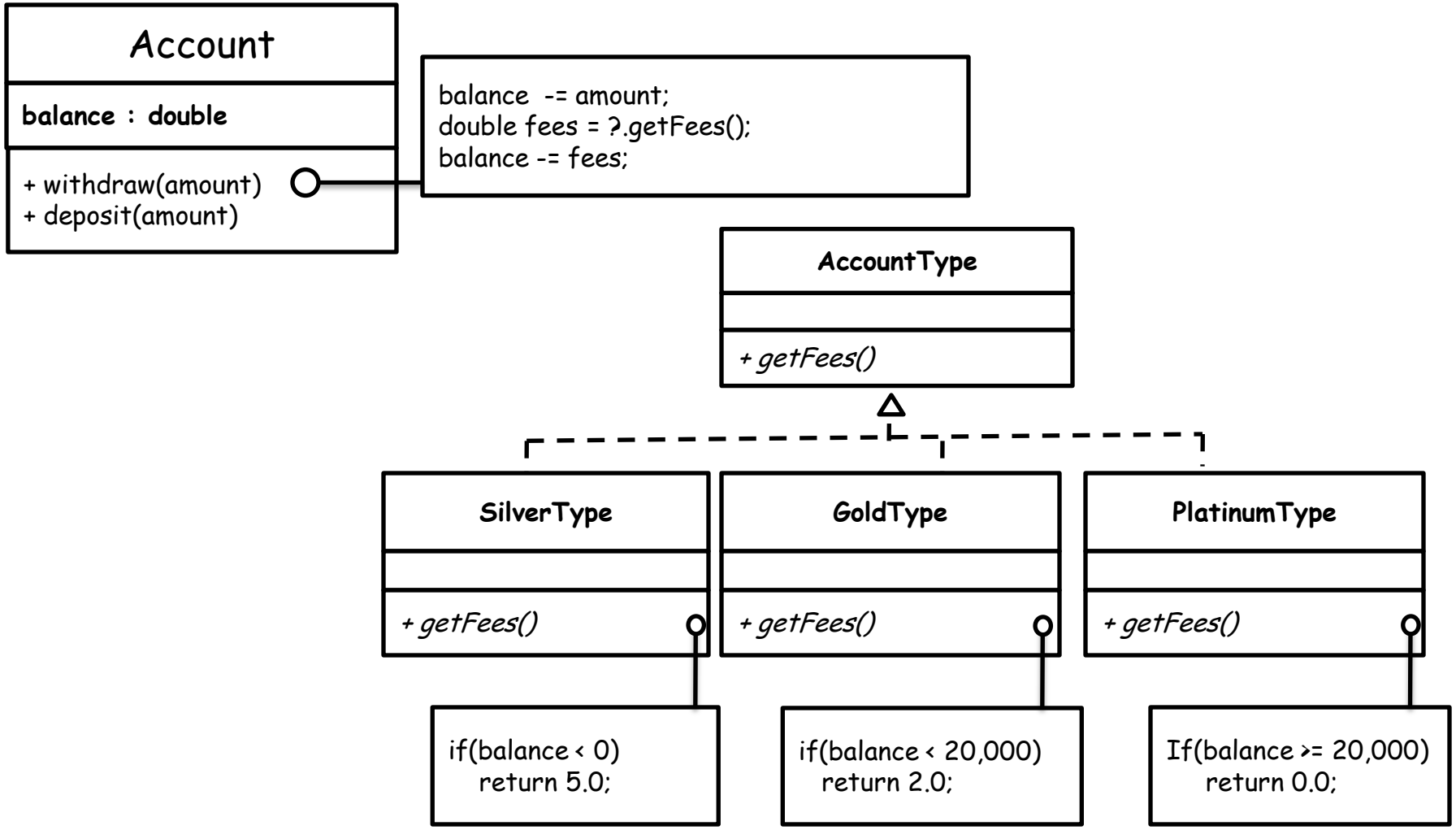

```
showUI(Account obj)
{
        UIVisitor v=  new UIVisitor();
        obj.accept(v);

        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();
}
```

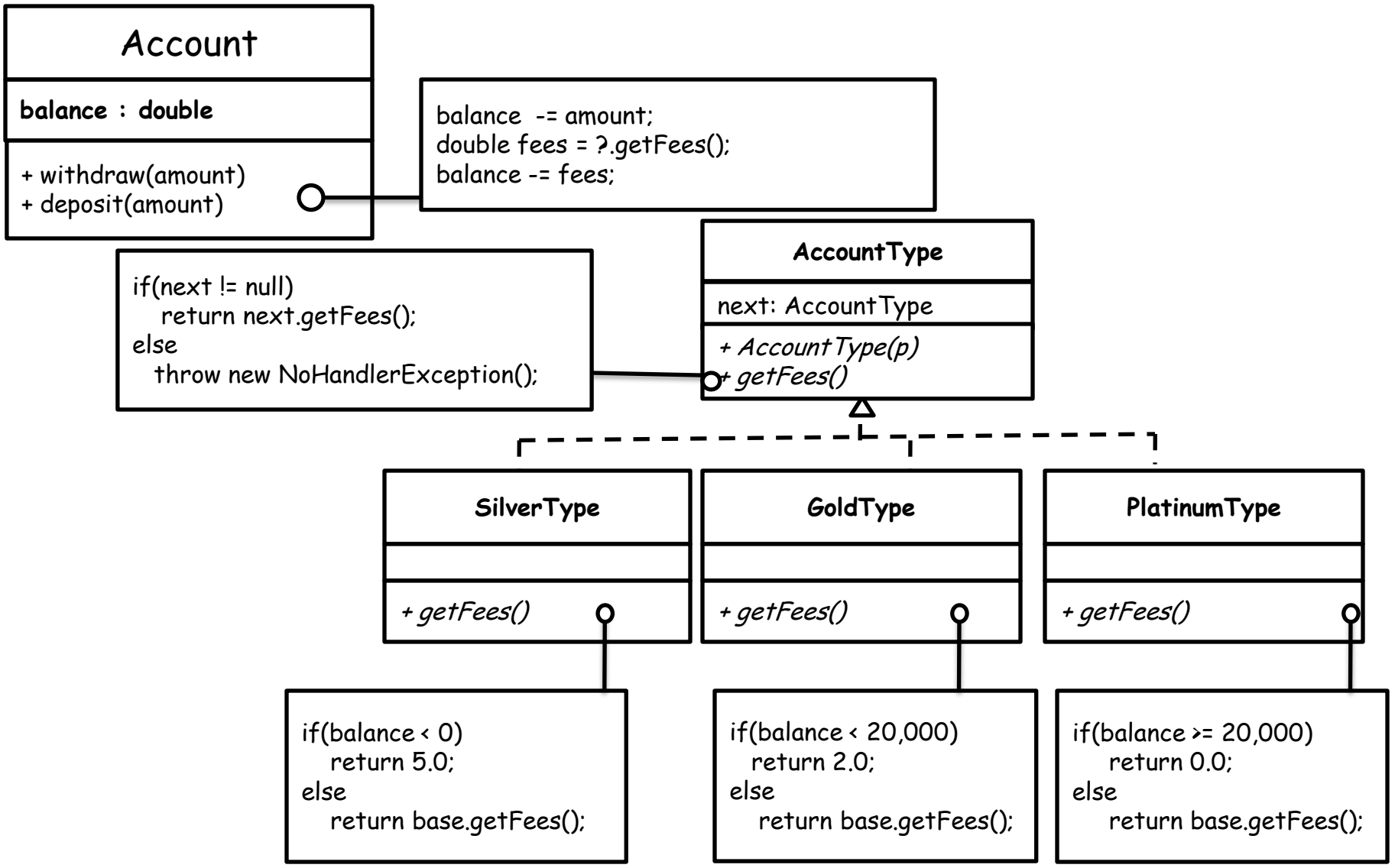
Problem


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

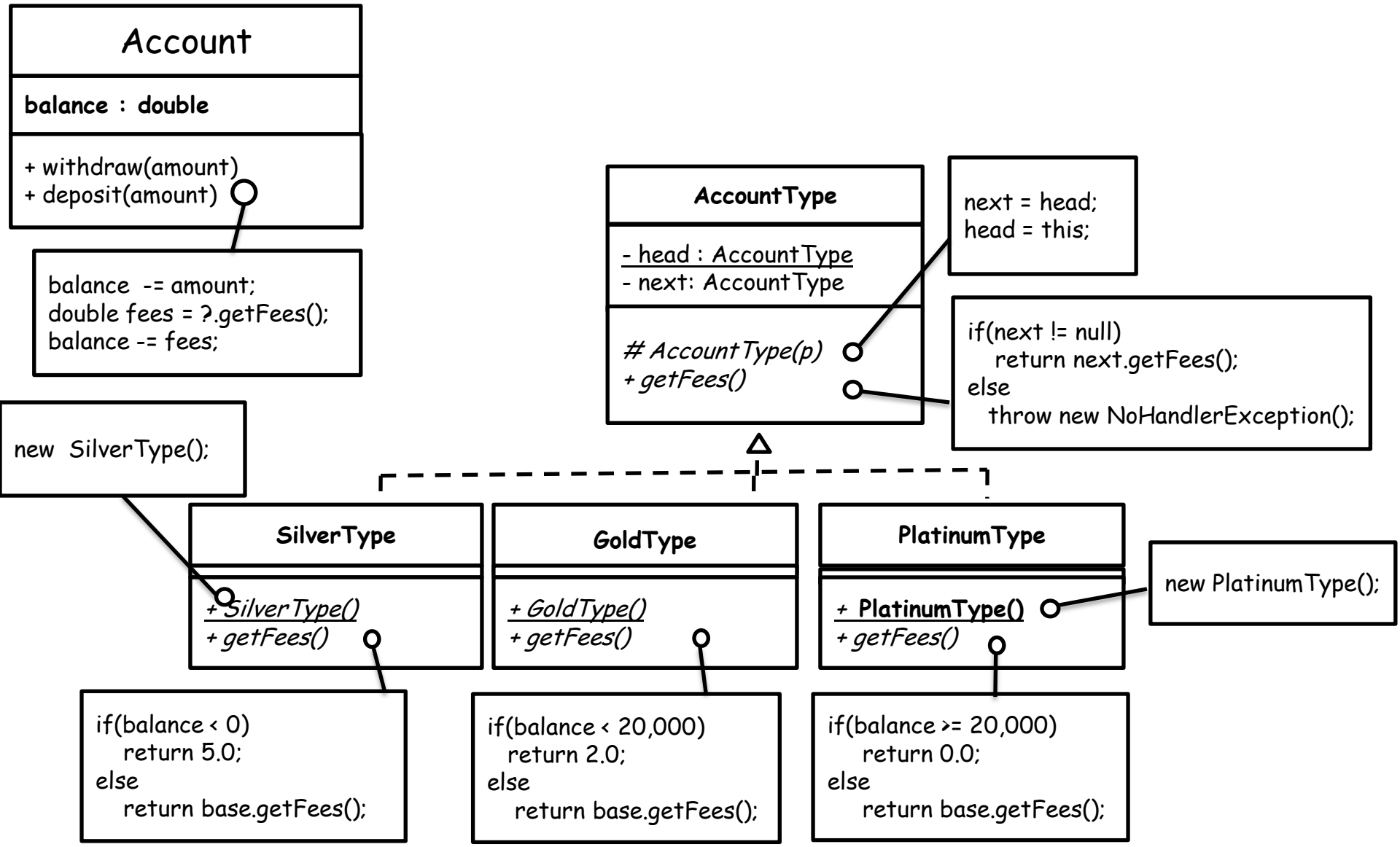
Moving Conditions Horizontally



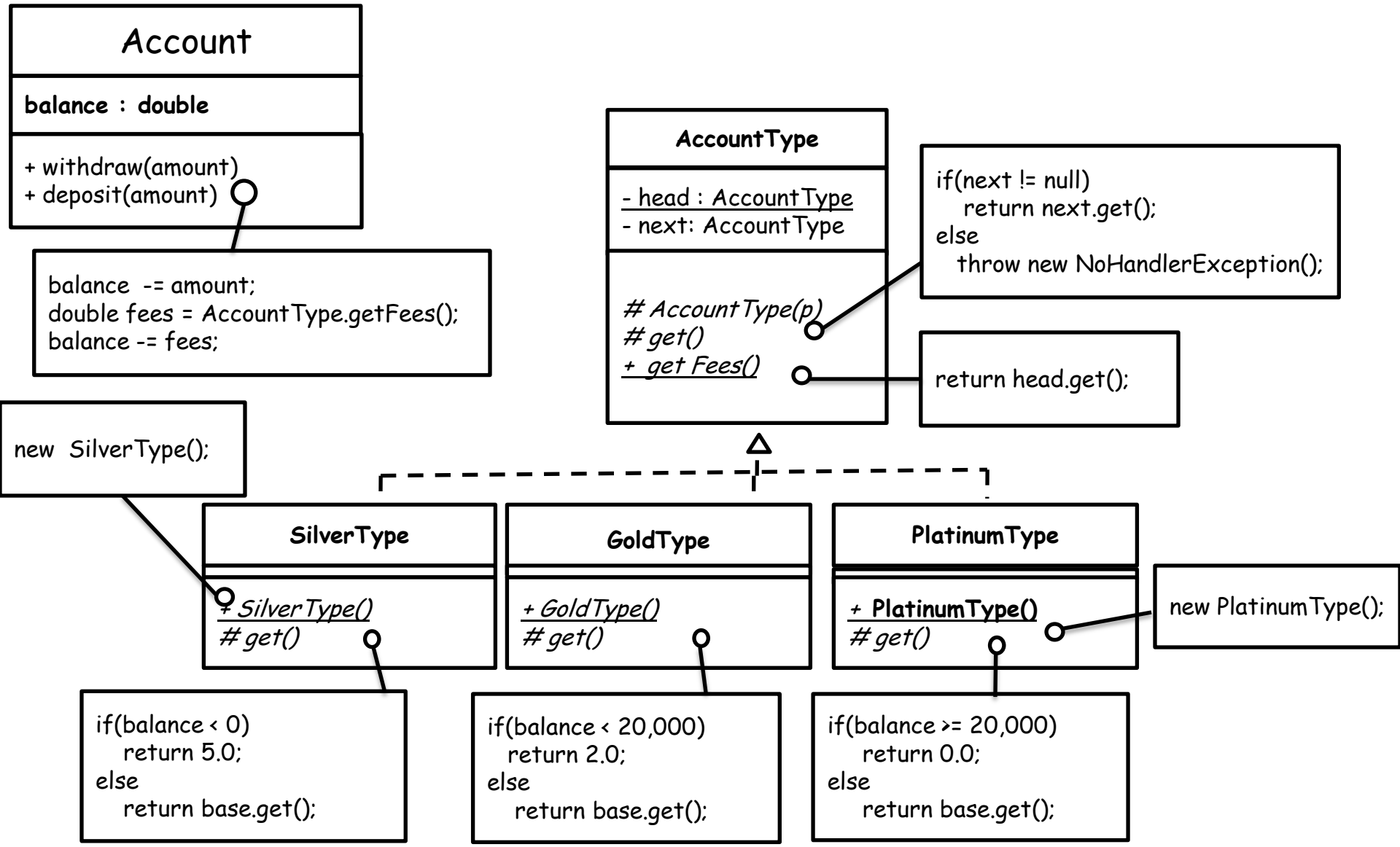
Applying Chain of Responsibility



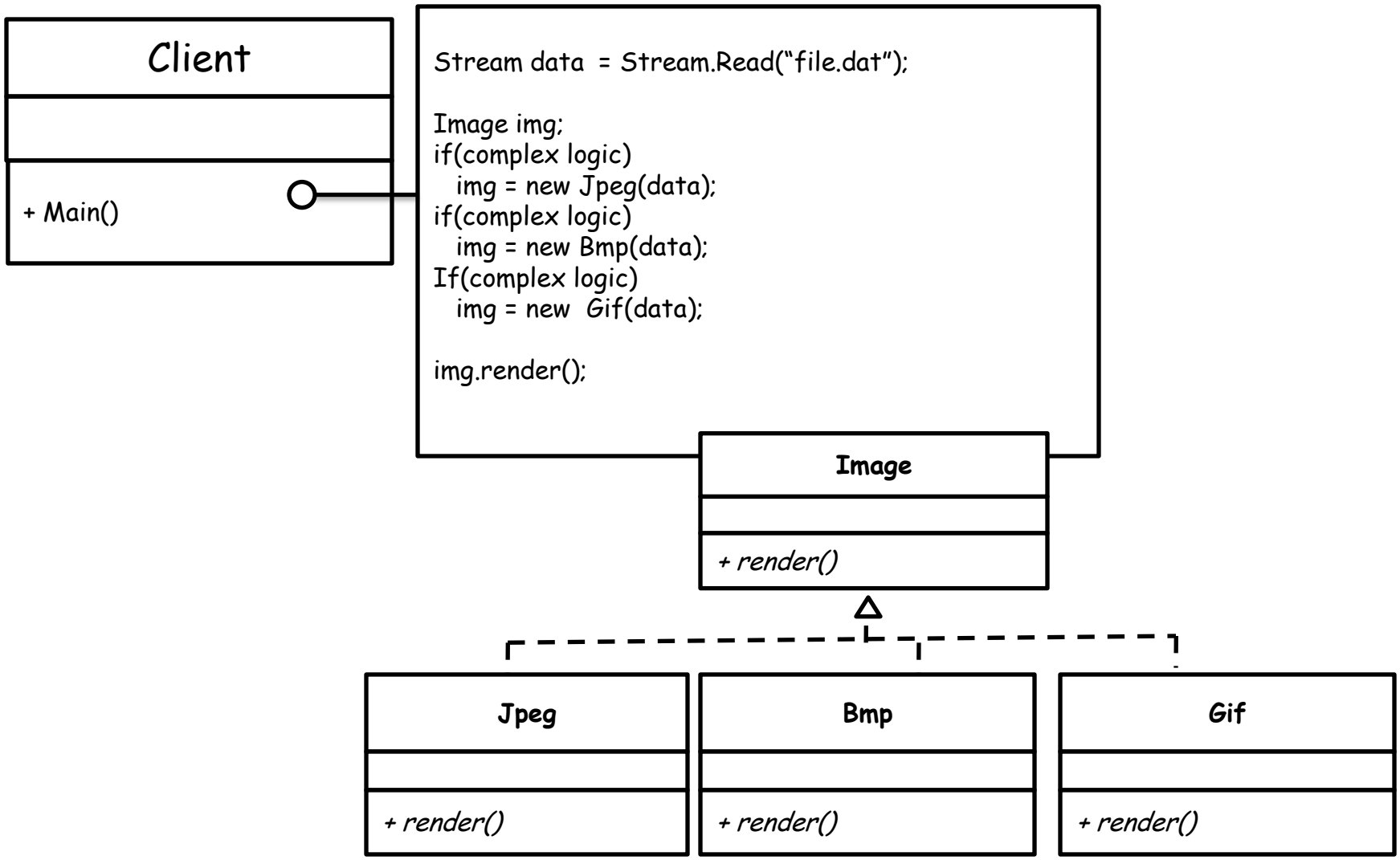
Applying Static Constructor



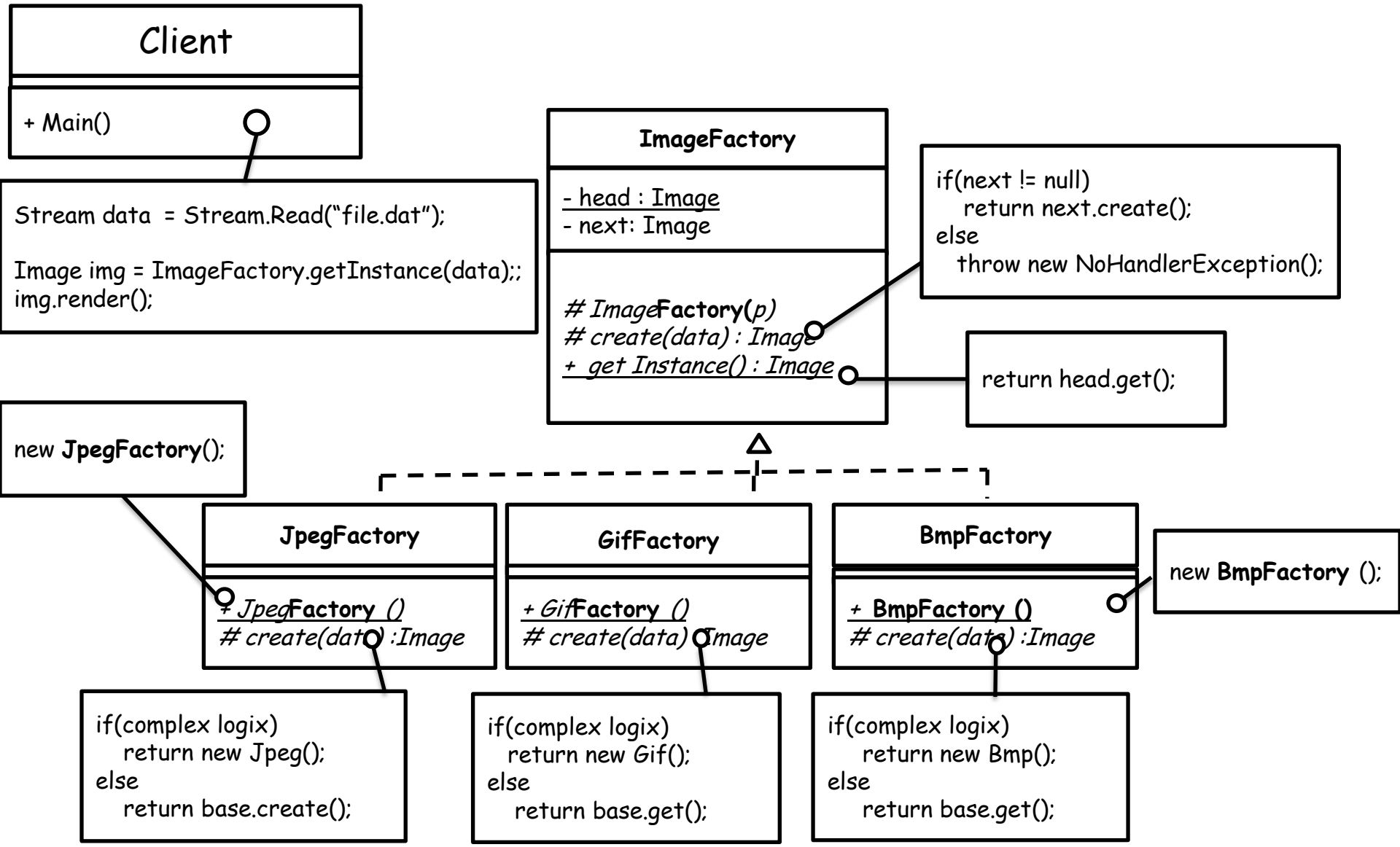
Applying Creator Method



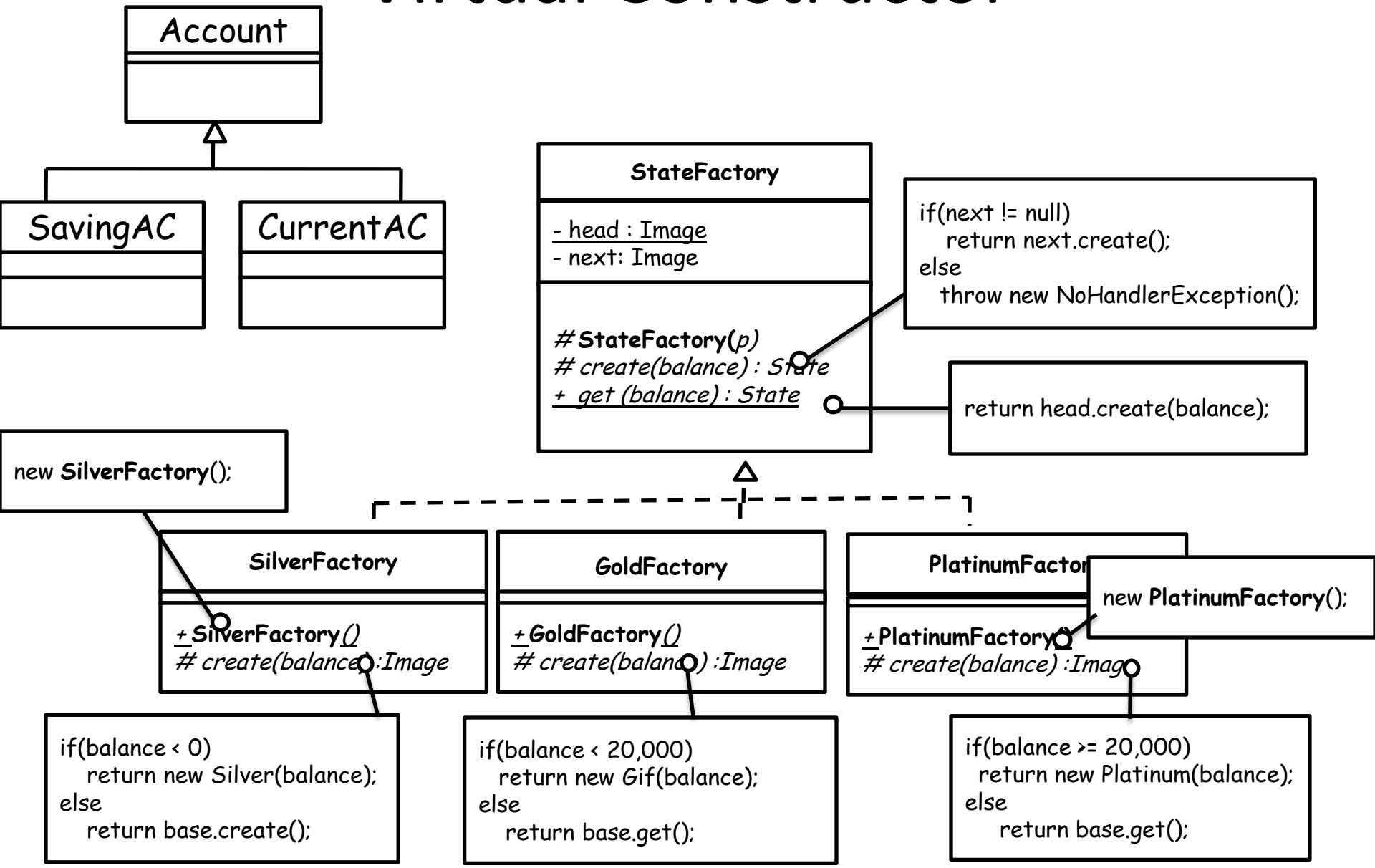
Virtual Constructor



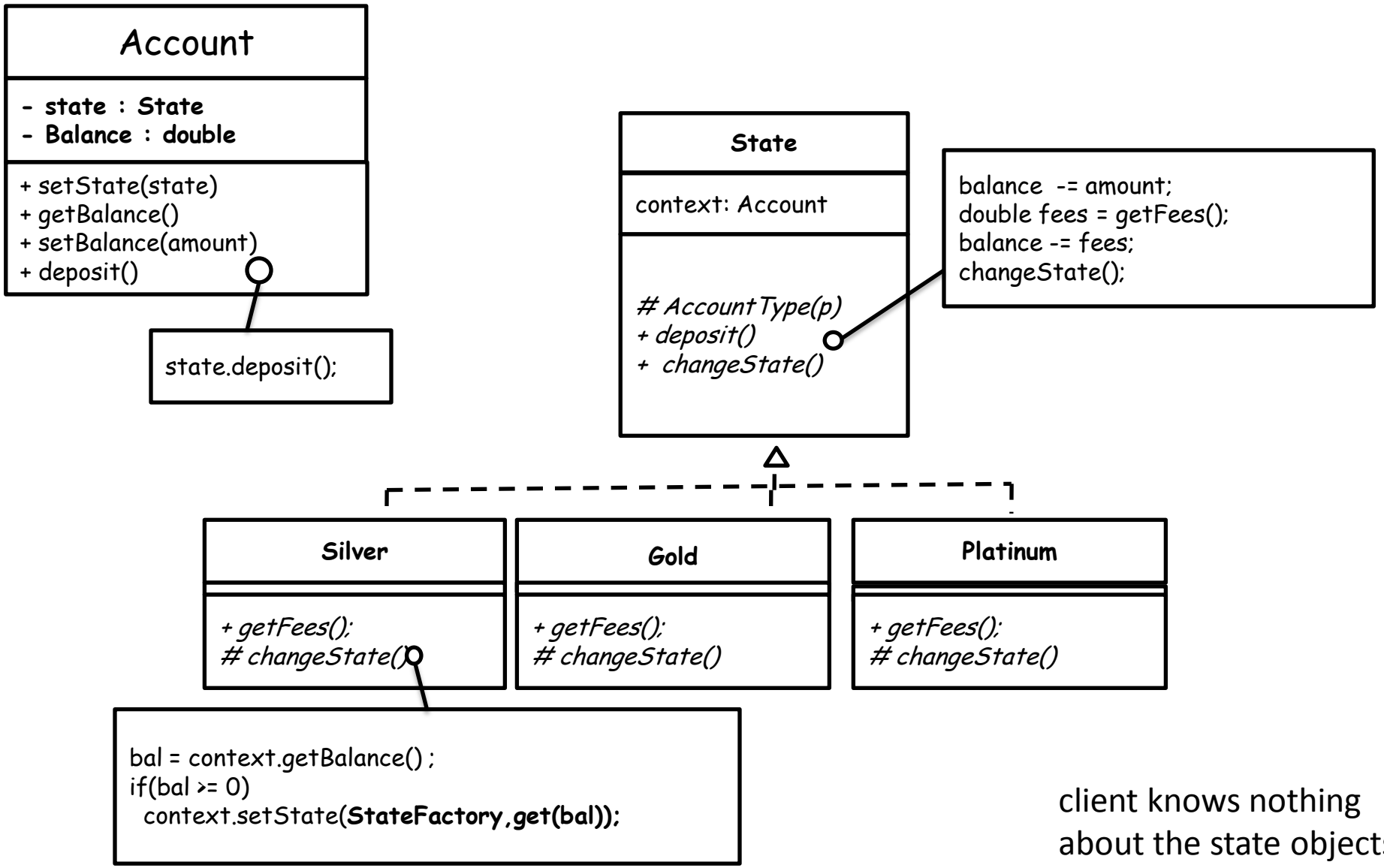
Virtual Constructor



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.

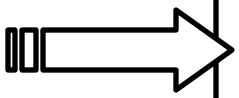
Dependency Lookup: With lookup a component must acquire a reference to a dependency. Dependency Lookup comes in two types:

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

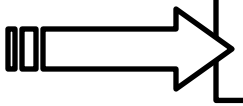


Dependency Pull
Contextualized Dependency Lookup (CDL).

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).

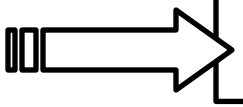
- Lookup is performed against the container that is managing the resource, not from some central registry.

- 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



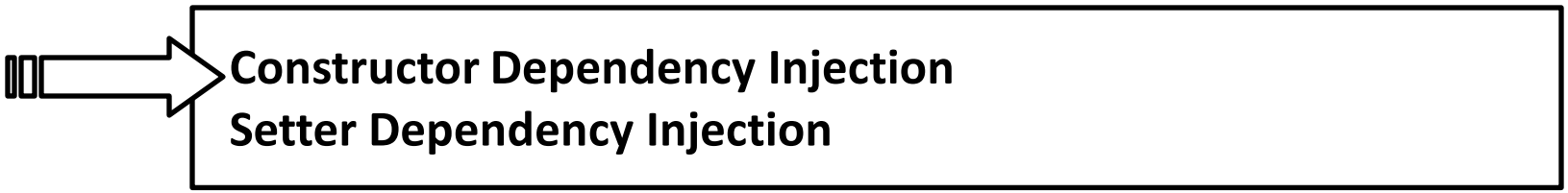
Dependency Pull
Contextualized Dependency Lookup (CDL).

- 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

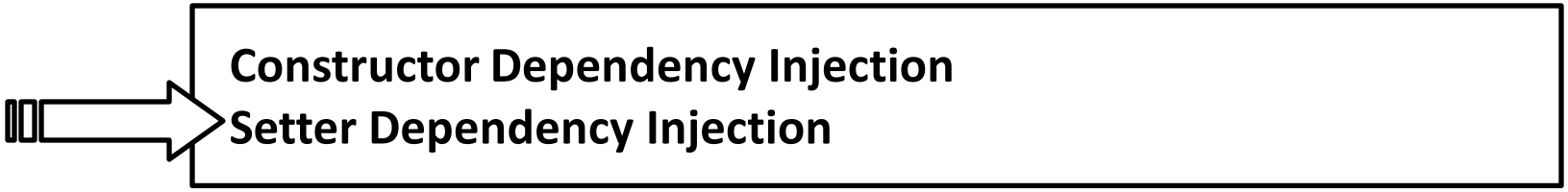


The component declares a constructor or a set of constructors taking as arguments its dependencies, and the IoC container passes the dependencies to the component when it instantiates it.

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

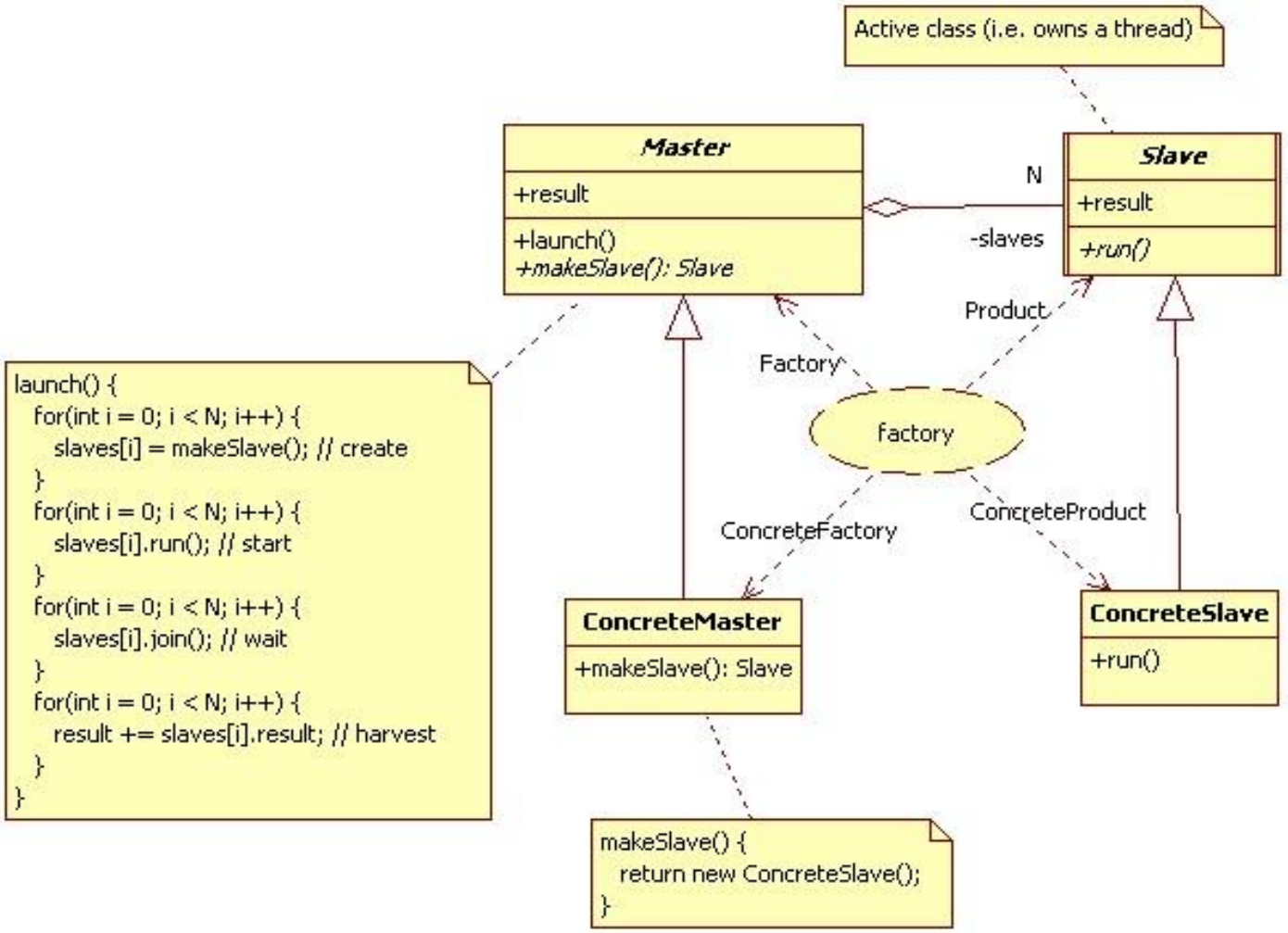


The IoC container injects a component's dependencies into the component via setter methods.

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

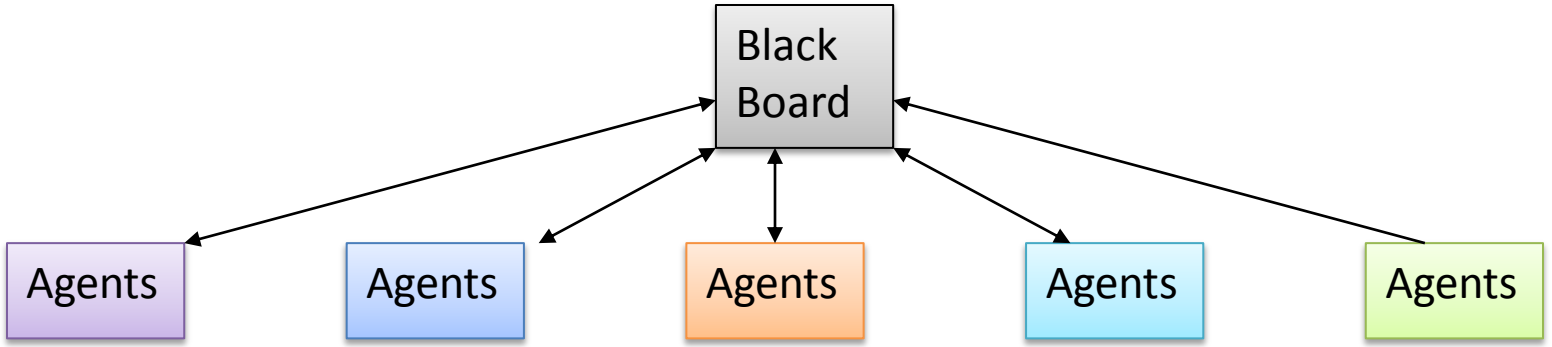
```
public class MyBean
{
	private Dependency dep;

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

Master Slave Pattern



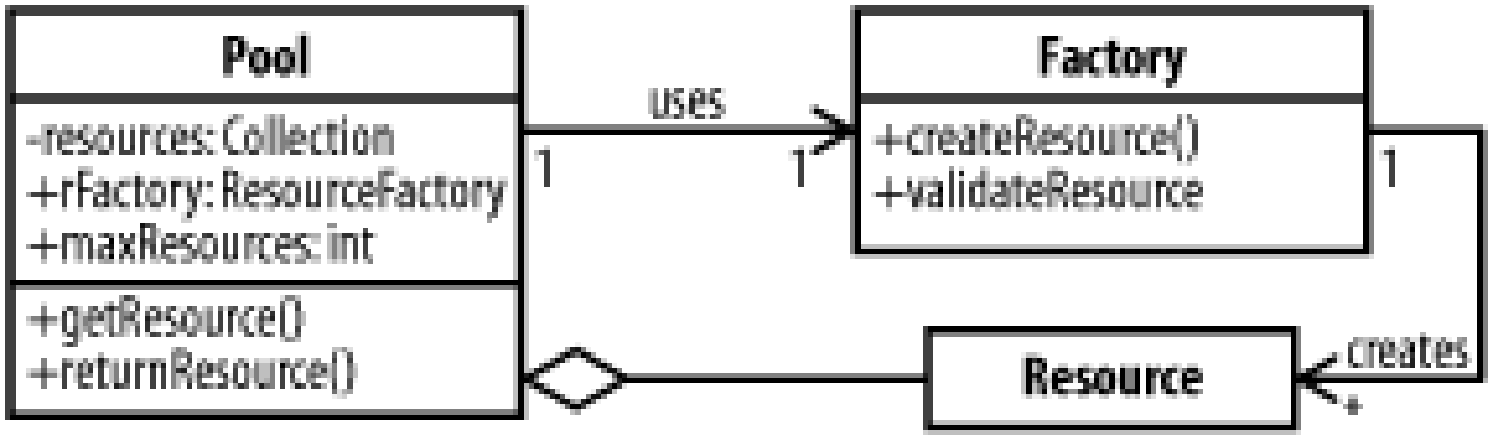
Blackboard Pattern

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



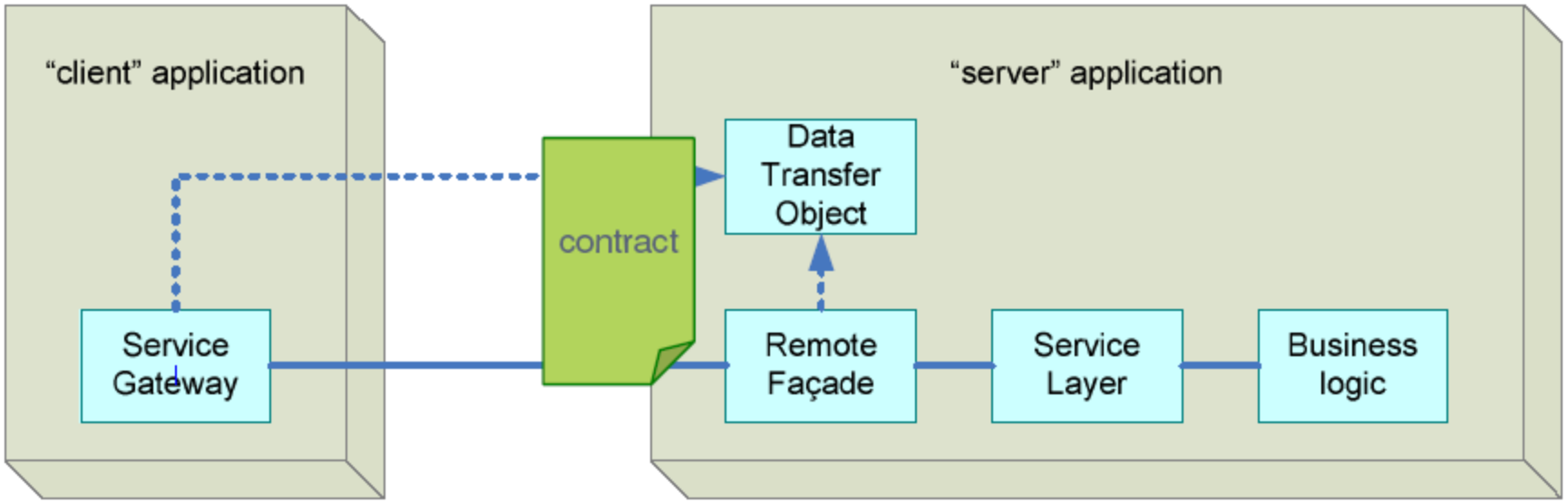
A process which posts an announcement to the blackboard has no idea whether zero, one, or many other processes are paying attention to its announcements.

Resource Pool

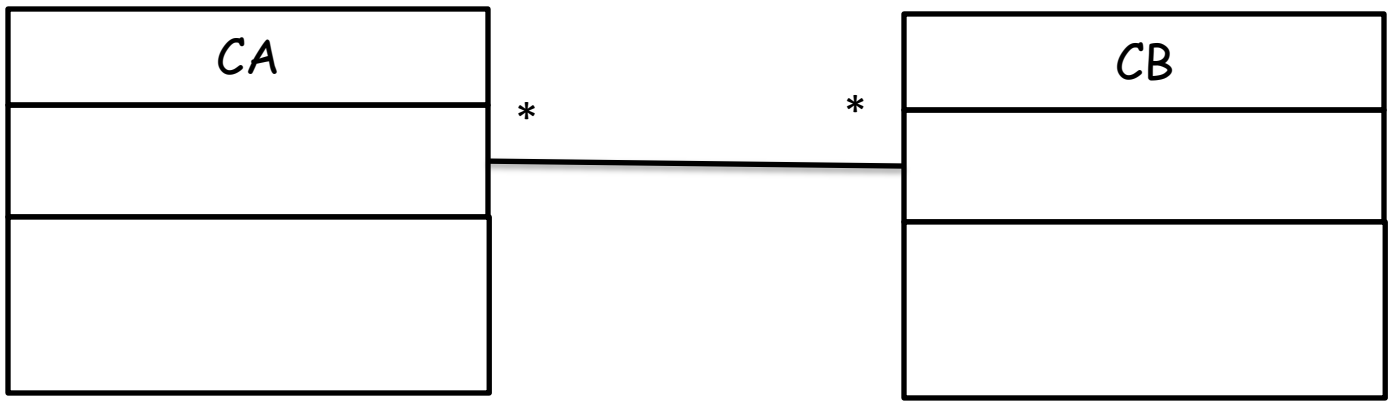


Pools show the most benefits for objects like database connections and threads that have high startup costs.

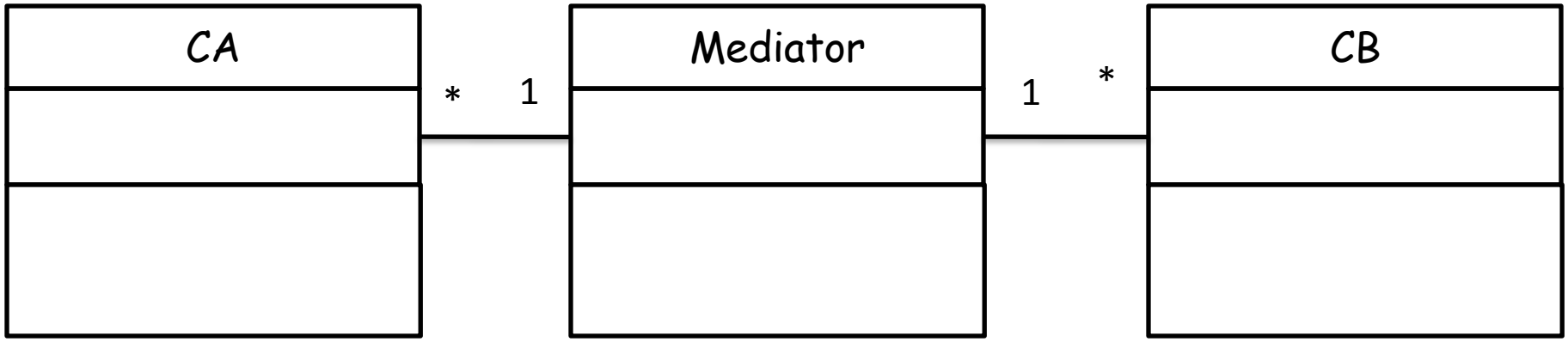
Patterns for Distributed applications



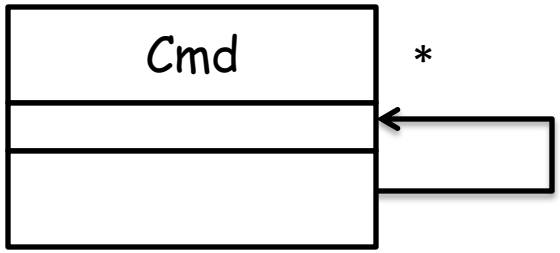
Problem

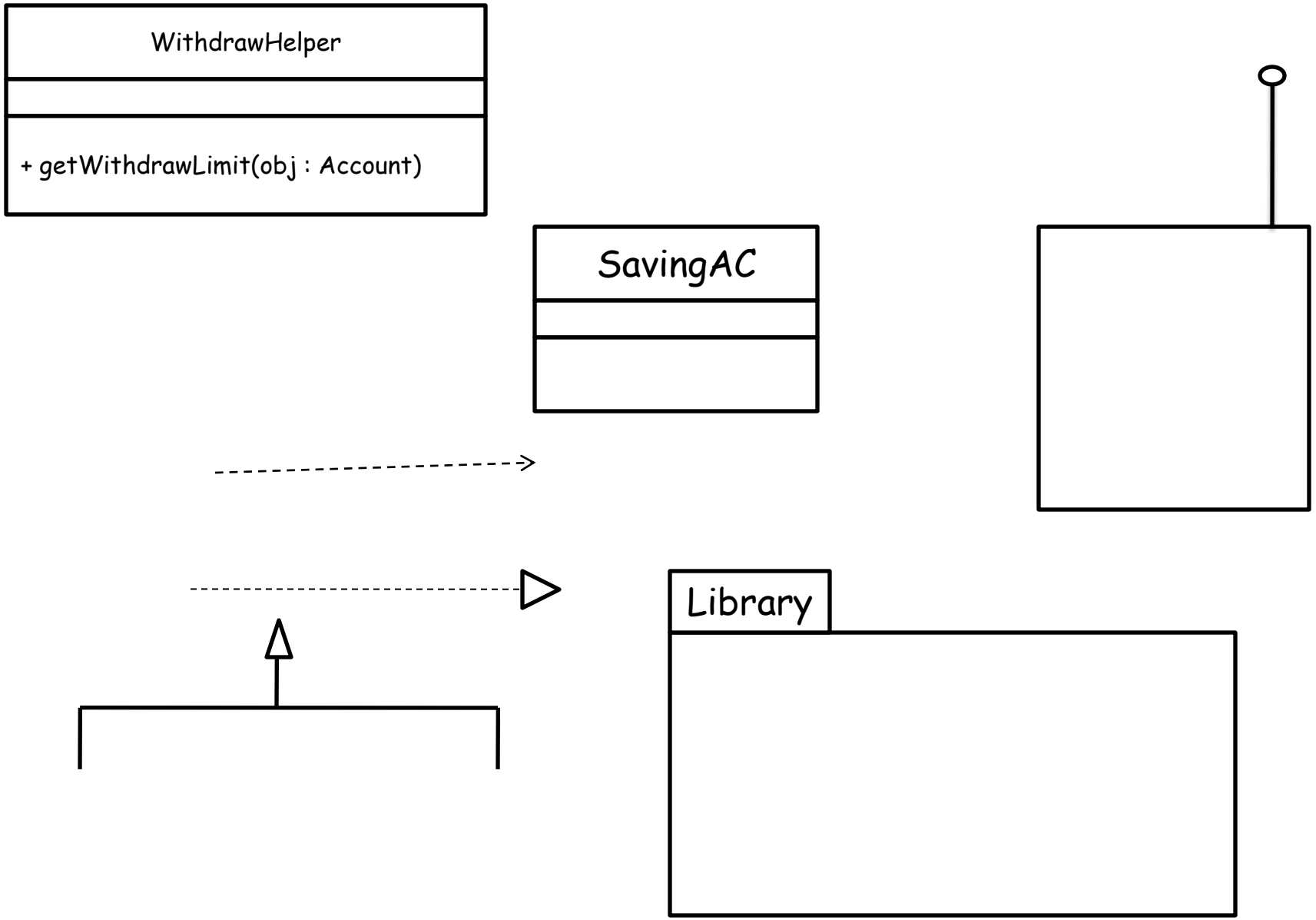


Applying Mediator



Composite Pattern




WithdrawHelper
+ getWithdrawLimit(obj : Account)
SavingAC
Library

Business Rule Engine