Design Patterns in C++

- Narayan Iyer

www.narayaniyer.com

- ni2@yahoo.com

Hands-on environment:

Visual Studio 2013 / 2010 express edition / Code::Blocks Latest 16.x ver NoSetup

I hear and I forget; I see and I remember; I do and I understand.

a Chinese proverb

Where ever possible we will understand the concepts and do hands on exercise

Introduction... name, total experience, experience in c++ & what are you working on, your expectation etc...

about me...

Bachelor in Computer Engineering, Bombay University, 1991

1991 - 1995

C/C++ - system level development;
Wrote debugger on x86 embedded platform
Worked at system level startup company
Further studies while at work:
CDAC – PGDST (1994-95)

1995 - 1999

Worked at Networking startup companies at Silicon Valley, CA - Development role (C/C++ /Java)

1999 - 2007

Intel San Jose:
Wrote Microcode dataplane libraries for IXP Network Processors; Developed UT infrastructure
Intel India:
Engineering Manager, PMP – Network domain;
Researcher – Storage Virtualization Domain
Intel Multicore University Program

2007 - date

Freelance Corporate Trainer:
Topics taken: Multicore, Unit Testing, C Prog & Optimization, Advanced C++, Design Patterns, Secure Coding, SDLC, SAS...
Consultant , Entrepreneur
Social Cause – Founded Science Society of India - Running science fairs, Hackathons

http://narayaniyer.com

Ground Rules – Let us agree

- Want to get your commitment on few ground rules... on controlling distractions
- Cell phones in silent mode.
- Email set to “out of office.. attending training”
- You do not need to answer a call – take it as a missed call and follow up during breaks
- Let us not take breaks at free-will
- (it is contagious if one person takes a bio break others tend to follow...)
- We will follow break timings: ~10.45 to 11.00 am and 3.30 to 3.45 pm
- Lunch: 12.45 to 1.45pm
- Let us check our emails & any other general internet access during the breaks...
- If you want any breaks in between tell me - and we will take a break.
- We may stop the session ~4.45 pm
- Any other expectation you have from my side??

Patterns that will be covered

Introduction
- Design Patterns makes use of OOPs concepts - Composition, Aggregation, Inheritance, Encapsulation
- Design Exercise
- Interface-vs-Implementation
- Design Exercise for “uses” scenario, Dependency Inversion

Mention MVC, Roles
Mention of The Factory Method Design Pattern
Mention of The Singleton Design Pattern
Mention of The Abstract Factory Design Pattern
The Proxy Design Pattern
The Composite Design Pattern
Design for Testability Pattern
The Visitor Design Pattern
The Command Design Pattern
The State Design Pattern
Chain of Responsibility Design Pattern
The Observer Design Pattern
The Adapter Design Pattern
The Facade Design Pattern
The Template Method Design Pattern
Mention of Decorator Design Pattern

Design Pattern

- What is it?

Design Pattern

- What is it?
  - Making use of a solution from the past experience and applying it immediately without rediscovering
  - *How many times have we said...i have solved this problem earlier…*
  - Inherently we are grounds-up people...
  - *“A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design”*

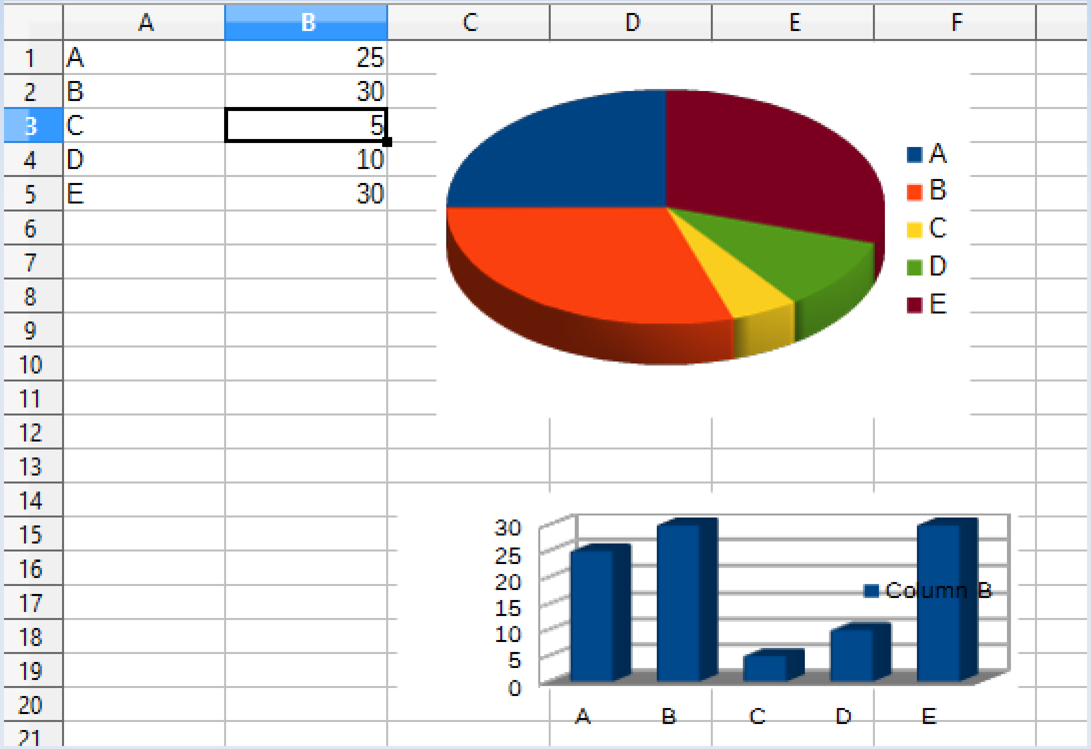
MVC

- Model / View / Controller
  - Model is the application / business logic
  - View is the presentation layer
  - Controller is the way UI reacts to input
  - Provides a framework of separation of functionality - instead of all together

MVC

- We could think as subscribe/notify protocol
- All the views register with model
- When model changes its value – it notifies all its subscribers
- Model notifies data change, view communicates to get the data

MVC demo



	A	B
1	A	25
2	B	30
3	C	5
4	D	10
5	E	30

There are others patterns as well..

Design Patterns Classification

- Creational
  - Deals with how objects are created
- Structural
  - Deals with composition of class/object
- Behavioural
  - Object interaction and responsibility division

Design Patterns Classification

- Creational
  - Deals with how objects are created
  - e.g. Factory Method (class)
  - Singleton (object)
- Structural
  - Deals with composition of class/object
  - e.g. Adapter (class)
  - Facade (object)
- Behavioural
  - Object interaction and responsibility division
  - Template method (class)
  - Observer (object)

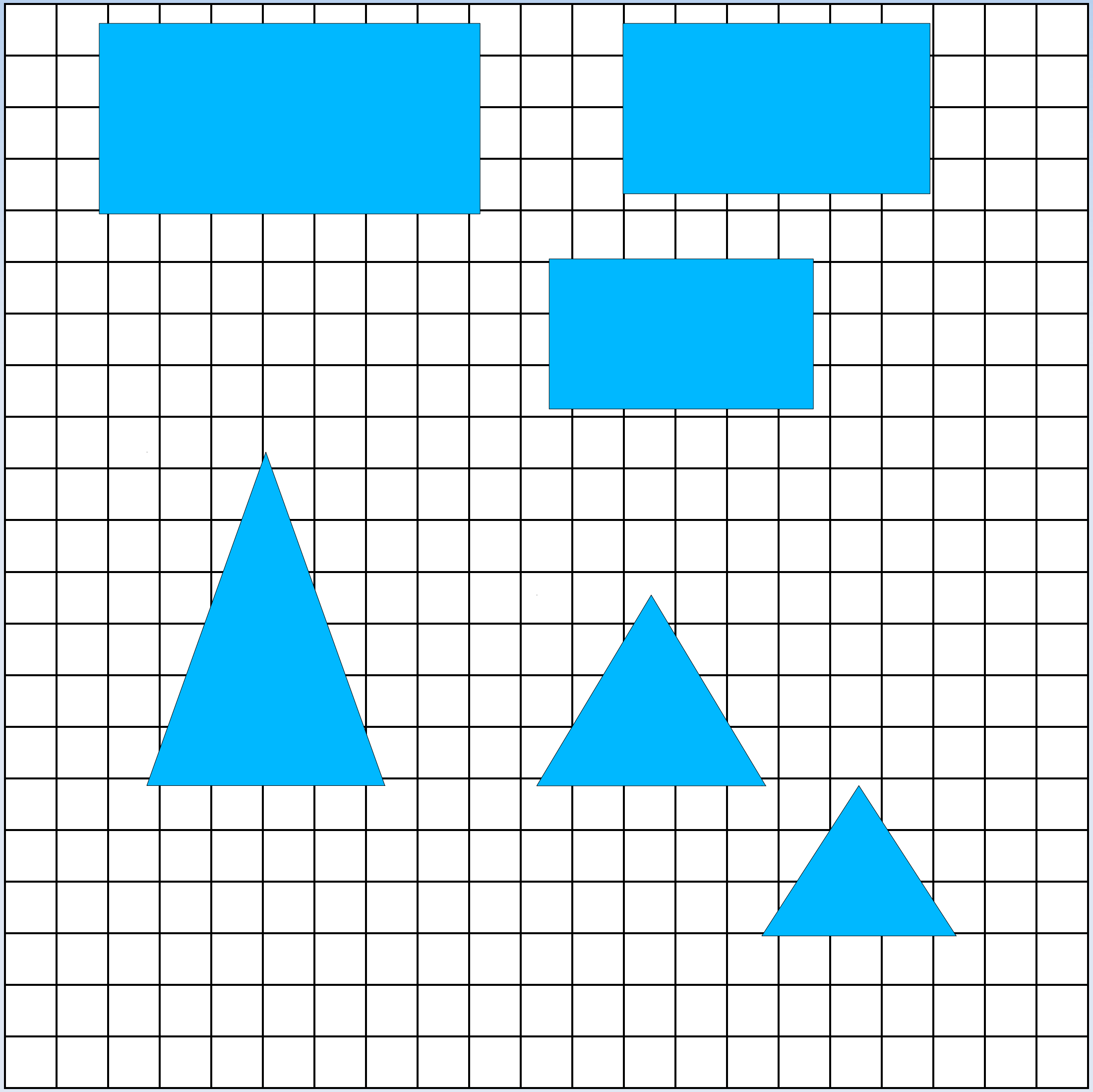
Class patterns deal with classes and derived classes, determined at compile time

Object patterns are more dynamic – deal with run-time creation of object

Both use inheritance

Most patterns are object scope

Design classes for this scenario



- This grid contains many Rectangles & Triangles
- Rectangle is a Polygon
- Triangle is a Polygon

Object Oriented Concepts

- Patterns make use of various OOP concepts
- Composition, Aggregation
- Inheritance, Encapsulation,
- Polymorphism

Object Oriented Concepts

- Let us identify the objects in this room?

Object Oriented Concepts

- Let us identify the objects in this room?
  - The table, the chair, the projector, the door, window, white-board, the computer, lights, attendees
  - Each has a specific functionality or single responsibility
  - And has a clearly defined independent interface(s) on how to use them

Object Composition

- Each object may be composed of other elements or “has” elements
- A mobile phone **has** a [key-pad], touch screen, speaker, mic, charging interface [USB interface], ...

Object Composition

- Each object may be composed of other elements or “has” elements
- A mobile phone **has** a [key-pad], touch screen, speaker, mic, charging interface [USB interface], ...
- A car **has** a steering wheel, engine, dashboard, wheels, radiator, …
- A chair **has** ?

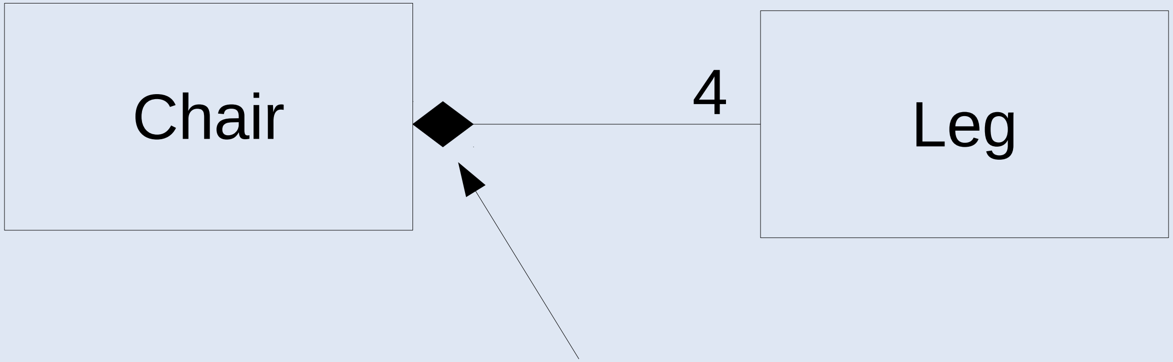
Object Composition

- Each object may be composed of other elements or “has” elements
- A mobile phone **has** a [key-pad], touch screen, speaker, mic, charging interface [USB interface], ...
- A car **has** a steering wheel, engine, dashboard, wheels, radiator, …
- A chair **has** legs, seat, arm-rest, back-rest

If we remove one leg from the chair – does it continue to be a chair?

Object Composition

- Removing one element from the object would make the object incomplete. e.g
  - Removing the leg from the chair may render it useless (it no longer continues to be a chair)
  - Any other examples?



Solid diamond shape

Ownership –
Company -->Employee Id
Bank ->Bank Account No...

Aggregation

- Specifying the containment relationships
- This room **consists of** attendees/participants
- Usage wise we may say
  - This room **has** many chairs & tables
  - This room **has** one projector
- But
  - The above is different from saying
  - A car **has** a steering wheel, engine, dashboard, wheels, radiator, …
- How?

Aggregation

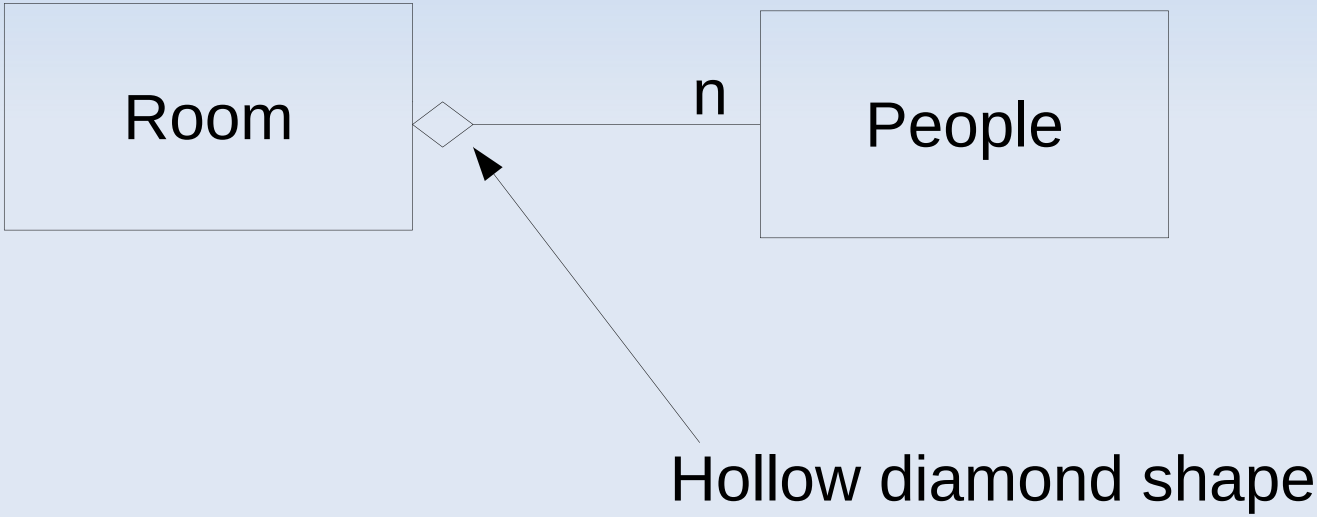
- Remove chairs from the room – still the room exists. Just that chairs might have been shifted to another room.
  - The above is a aggregate “containment” relationship
- Removing steering wheel from the car – makes the car incomplete.
  - The above is a composition relationship

Aggregation

- Say you had a mobile phone object (an instance)
  - While destroying a mobile phone object – one is likely to destroy its sub object such as [keypad], lcd etc because they are within a single composition
- Basket containing mobile phones – if the basket is being destroyed the cell phones can still exist and may be moved to another basket
  - Basket consists of mobile phones
- Similarly people in a room

Aggregation

- Room has many people



Composition & Aggregation

- Is it clear?

Association

- One can also think of association concept
  - e.g.
  - a teacher <u>teaches</u> students
  - an employee <u>uses</u> swipe badge
- It tells about two different entity having independent ownership but the associate together for some purpose
- We can think of this as a mapper for a task
- *From an implementation perspective - we should use aggregation concept and achieve the same*

Inheritance

- Now, let us see Inheritance

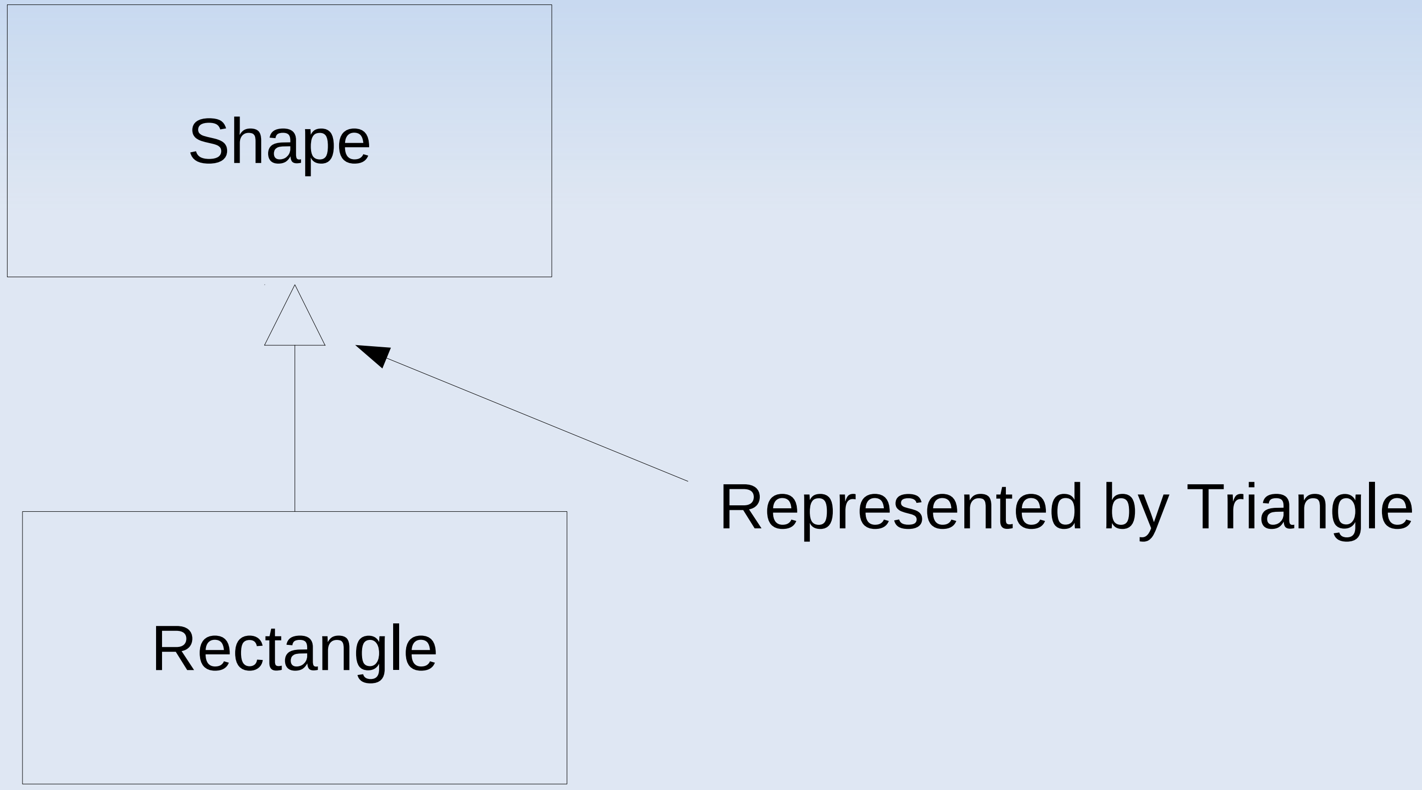
Inheritance

- This **is a** roll-able chair
  - This has all the properties of a chair with the addition that it is roll-able
  - We can also say – this has inherited all the properties of a chair with the addition of it rolling capability
  - Rollable-chair **is-a** chair
- This is a LED TV
  - This has inherited all the properties of TV
  - With the addition that it is based on LED technology
  - LED_TV **is-a** TV
- This **is a** USB Hard Disk
  - Has inherited all the properties of Hard Disk
  - It still maintains the original properties of a hard disk with additional capability that it is USB based
  - USB-HD **is-a** Hard Disk
  - *It has not changed its property or morphed to become a gaming console*

What about an org structure?

Inheritance

- Rectangle is-a Shape



Shape

Rectangle

Represented by Triangle

Encapsulation

- Hiding the internal complexity of the object
- For e.g. a driver does not need to know how the internal combustion engine works
    - You use the car by knowing the interface given - accelerator, break, clutch, start ignition
    - Whether the car is a petrol car or electric car or runs on CNG – the interface remains the same
- Generic interface but error-prone e.g. same diesel and petrol nozzle size
- Similarly objects you create must have interface that does not expose the internal workings (which may change)
- Users of your objects continue to call the interface exposed by you while you change/optimize the underlying implementation. e.g. audio player

Polymorphism

- Is an ability to create an object that has more than one form
- For e.g. if there is an object type Bird. Parrot and Crow are inherited from Bird
    - Parrot is a Bird; Crow is a Bird
    - At **run-time** if the Bird object is pointing to Parrot then calling a method fly() - will make the Parrot fly;
    - It is also called as late-binding

Design Exercise

- Do an OO software design for a two-player Chess game
- The application need not have to be played against computer

Player-2

Player-1

Write C++ classes

Chess game- what do you see?

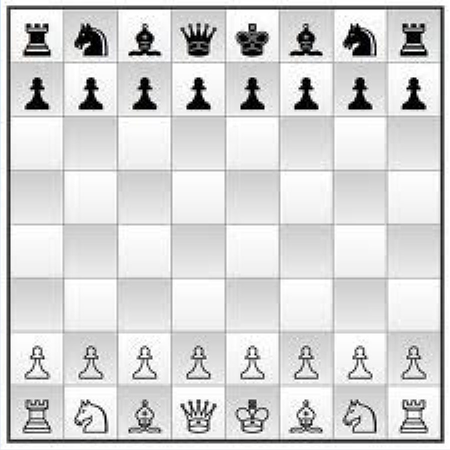
- What do you see

Player-2

Player-1

Chess game- what do you see?

Player-2

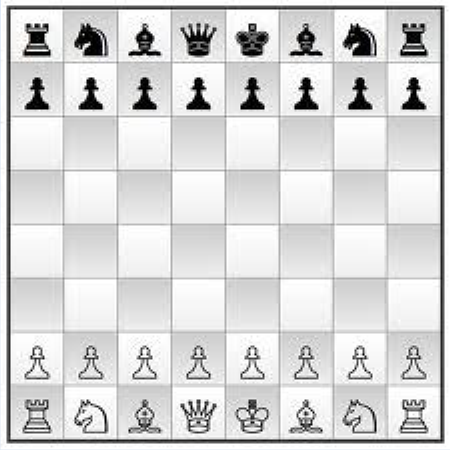


Player-1

- What do you see
- 1 ChessBoard
- 32 ChessPieces
  - 16 White colour - attribute
    - 8 Pawns, 1K, 1Q, 2B, 2Kn, 2R
  - 16 Black colour - attribute
    - 8 Pawns, 1K, 1Q, 2B, 2Kn, 2R
- 2 Players
  - Player 1, Player 2
- What are the controls – who decides how the logic will play

Chess game- what do you see?

Player-2



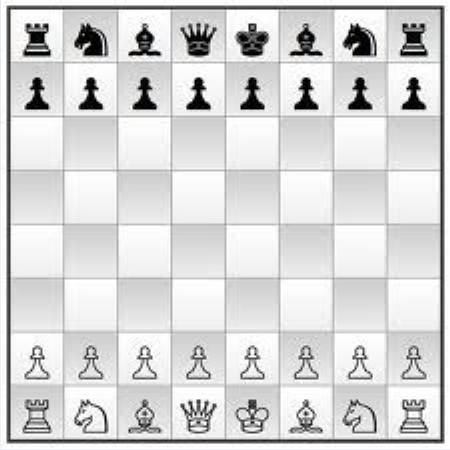
Player-1

- What do you see
- 1 ChessBoard
- 32 ChessPieces
  - 16 White colour - attribute
    - 8 Pawns, 1K, 1Q, 2B, 2Kn, 2R
  - 16 Black colour - attribute
    - 8 Pawns, 1K, 1Q, 2B, 2Kn, 2R
- 2 Players
  - Player 1, Player 2
- What are the controls – who decides how the logic will play
- Game Controller

For each of the above,
What are the possible operations?

Chess game - operations

Player-2



Player-1

what are the possible views?

- What do you see & what are the operations possible
- 1 ChessBoard
  - **Stores chesspieces, one can query pieces at location**
- 32 ChessPieces
  - **Checks the move, move itself**
  - 16 White colour - attribute
    - 8 Pawns, 1K, 1Q, 2B, 2Kn, 2R
  - 16 Black colour - attribute
    - 8 Pawns, 1K, 1Q, 2B, 2Kn, 2R
- 2 Players
  - **Makes the move, interacts with user, takes turn**
  - Player 1, Player 2
- What are the controls – who decides how the logic will play
- Game Controller
  - **Start Game**
  - **Control the game flow**
  - **Stop Game**
  - **Declare results**

Chess game

- Think about relationships (consists of)
    - Chessgame consists of players, board & chess pieces. They follow chess rules to play
        - *Consists* will generally go as?
            - a data member
        - class Chessgame

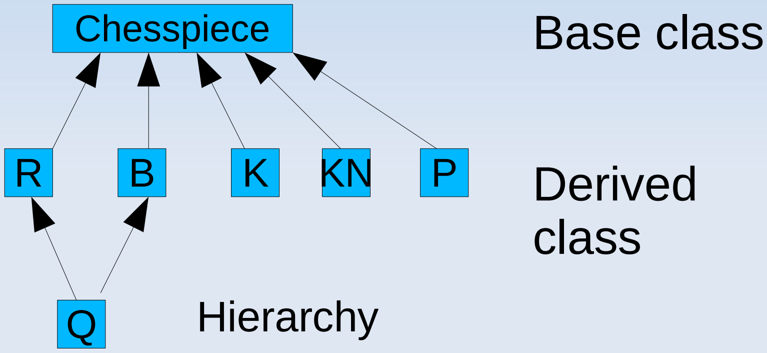
```
{
        ChessBoard cb;
        Player p1, p2;
        Chesspieces cp[32];
        ChessControl cc;
}
```

Use any drawing notation to capture this, with any constraint

Numbers generally will go as instances 32 chesspieces, 2 players etc.

Chess game

- Think about hierarchy (is-a)
  - Pawn *is a* Chesspiece
  - Knight *is a* Chesspiece
  - King *is a* Chesspiece
  - Rook *is a* Chesspiece
  - Bishop *is a* Chesspiece
  - Queen *is a* Chesspiece
    - It has all the characteristics of Rook & Bishop
- Operations on each classes – go as member function; associated storage will go as data member



Chessgame class

```
class Chessgame
{
protected:
      ChessBoard *cb;
      Chesspiece *cp[32];
      Player *p1;
      Player *p2;
      ChessControl *cc;
public:
      Chessgame(ChessBoard *p_cb,
        Chesspiece *p_cp[],
        Player *p_p1;
        Player *p_p2;
        ChessControl *p_cc)
      {
            cb = p_cb;
            cp = p_cp;
            p1 = p_p1;
            p2 = p_p2;
            cc = p_cc;
      }
...
};
```

Use pointers or references

When creating objects – one can think about
using factory design pattern

Exercise summary

- Object Model
  - Composition (ChessGame)
  - Inheritance (is a – Rook is-a chesspiece)
  - Encapsulation (hide implementation - clean interfaces; chessboard->getpiece(x,y); chesspiece->move())
  - Polymorphism (chesspieceView->draw())
- C++ implementation allows you to create such models

Object

- Object is an instance of class
  - Object supports interface defined by the class
- Is there a difference between object's type and it class?
  - Yes
  - Objects of different classes may have same type
  - e.g.
  - Shape& s1 = Rectangle();
  - Shape& s2 = Square();

Class inheritance & Interface inheritance

- Class inheritance involves representation and code sharing, you implement the additional functionality (or only the changes to the parent)
- In our chess example – For Queen we inherited from Rook & Knight
- Interface inheritance describes when an object can be used in place of another
  - If we have an abstract class message-interface and messageQ inherits from message-interface class
  - Later if we want to use Pipe, we just have to create a sub-class from message-interface
  - It would just need to be substituted with Pipe

Interface inheritance & Class inheritance

```
class mesg_if
{
public:
  virtual void init() = 0;
  virtual void recv() = 0;
  virtual void send() = 0;

};

class msgq : public mesg_if
{
public:
  void init()
  {
    // implement msgq-init
  }

  void recv()
  {
    // implement msgq - recv
  }

  void send()
  {
    // implement msgq - send
  }
};
```

```
class animal
{
};

class horse : public animal
{
public:
  virtual void mf1();
  virtual void mf2();
};

class flying_horse : public horse
{
public:
  void mf2(); // override
};
```

Class Inheritance:
Here we share/reuse the base implementation

Interface Inheritance:
Here we substitute with another implementation

Design patterns depend on this concept

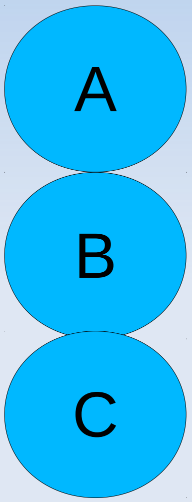
Benefits of interface inheritance

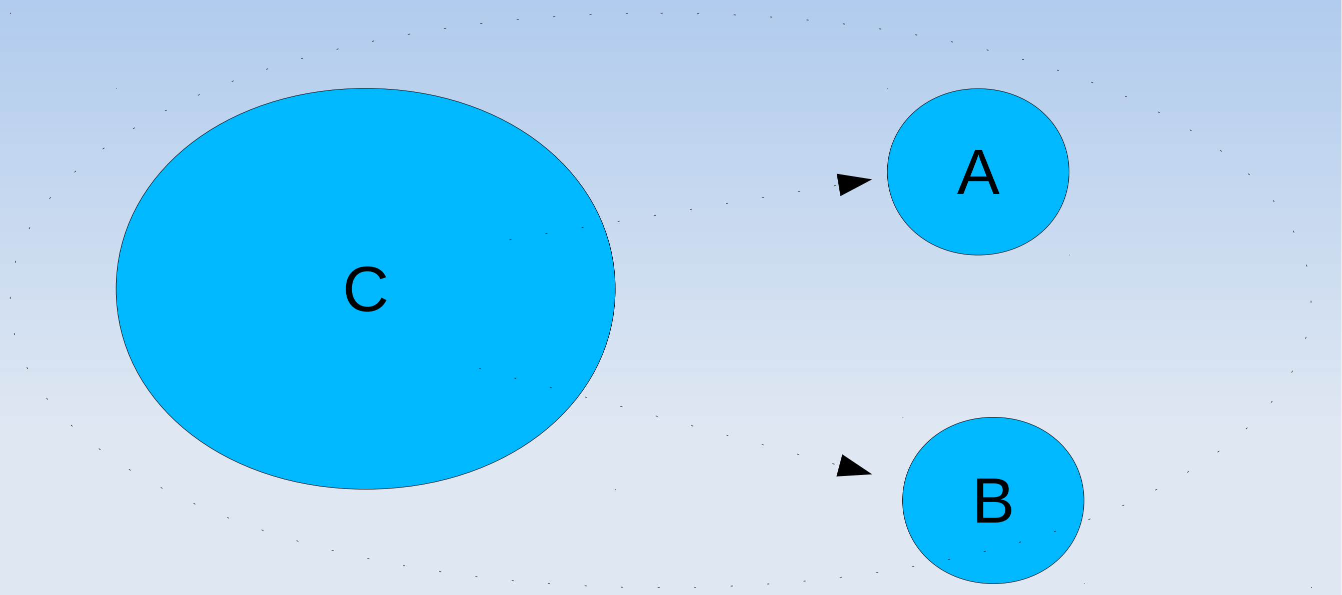
- *Benefits to manipulating objects solely in terms of the interface defined by abstract classes:*
  - *Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.*
  - *Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.*

Program to an interface, not an implementation.
Make code dependent on abstractions rather than concrete!
It also improves testability

When we need to create concrete things – use creational design patterns

Inheritance vs Composition



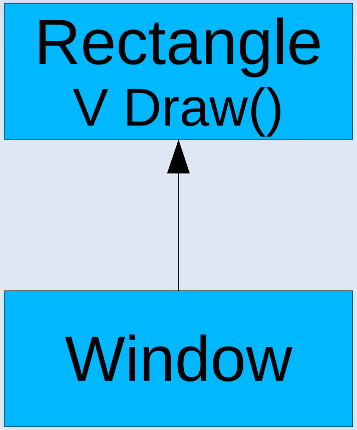


Given a choice,
use object composition instead of class inheritance

Delegation

- Inheritance vs Composition

A (Inheritance)



Rectangle
V Draw()

Window

B (Composition)

```
Window
Draw()
{
  pRectangle->Draw();
}
```

Delegation

- Inheritance vs Composition

A (Inheritance)

```
class Rectangle
{
        virtual void Draw();
        ...
};
class Window : public Rectangle
{

};
```

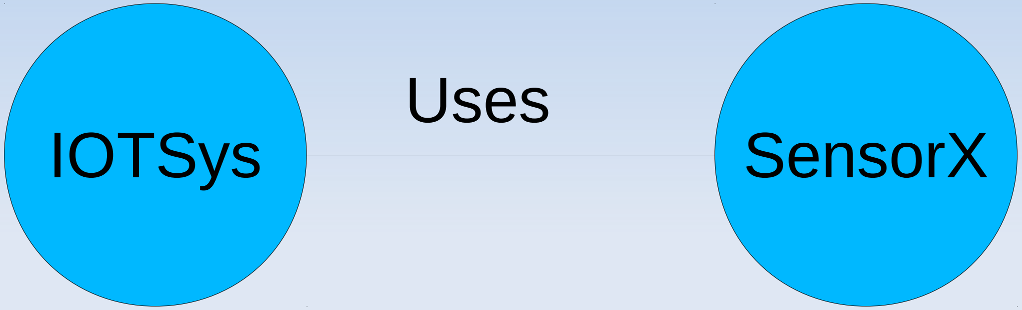
B (Composition)

```
class Window
{
        Rectangle *pRectangle;
        void setRect(Rectangle *pRect)
        {
                pRectangle = pRect;
        }
        void Draw()
        {
                pRectangle->Draw();
        }

};
```

Later, if we need a window with rounded rectangle
Which method will accommodate the change easily - A or B?

Exercise: IOT system uses SensorX

- IOT system class uses SensorX class to display value
- SensorX does open, getval, close – design this



IOTSys

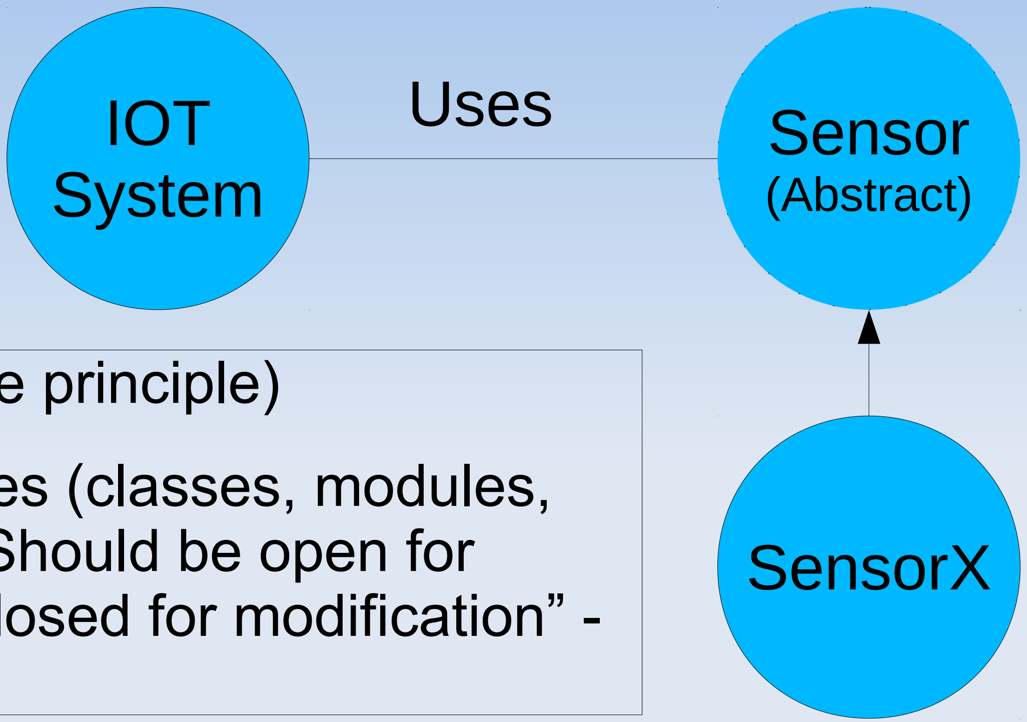
Uses

SensorX

```
class SensorX
{
public:
 void open() { // code specific to  SensorX }
 void getval() { // code specific to SensorX }
 void close(){ // code specific to SensorX }
};
class IOTSys
{
protected:
 SensorX sx;
public:
 void display() { sx.open(); cout << sx.getval(); sx.close() }
...
}
```

This is very rigid design (not good!)

Client uses SensorX



- OCP (open/close principle)

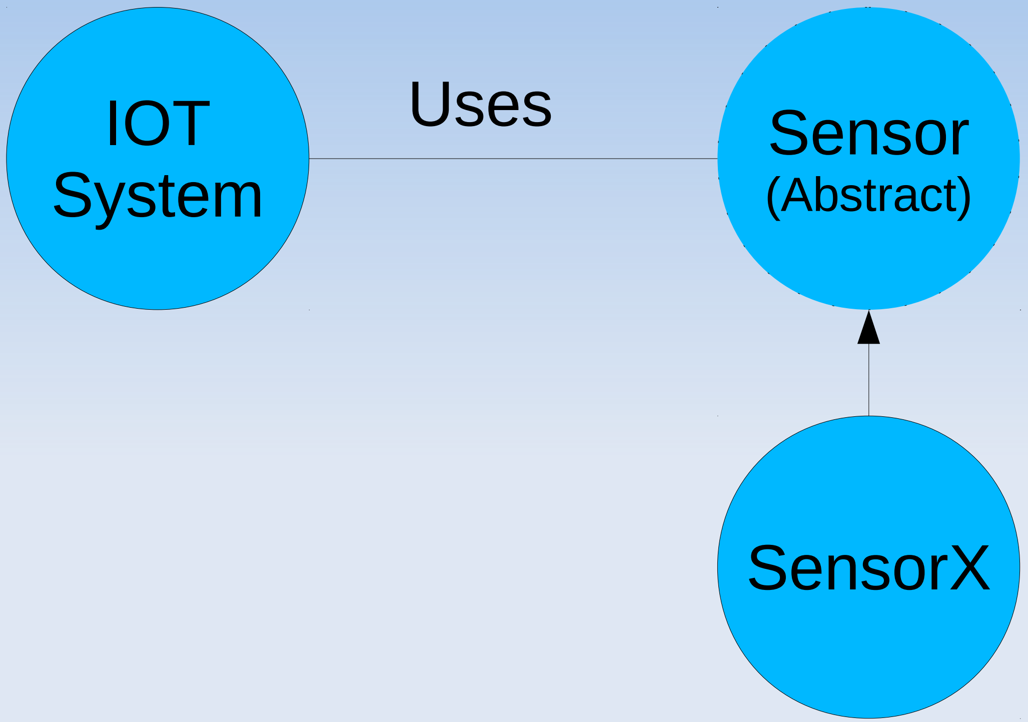
  “Software entities (classes, modules, functions, etc.) Should be open for extension, but closed for modification” - B. Meyer

- IOT System uses Sensor (Containment relationship)

Otherwise it is stuck with a specific SensorX behaviour;
Any new sensor SensorY comes, we will have to change the client;

Should not have If (x) then use sensorx else use sensory – sensorz comes we will have to modify the code for supporting zsensor feature

IOT System uses SensorX



```
class Sensor
{
public:
 virtual void open()=0;
 virtual void getval()=0;
 virtual void close()=0;
};
```

```
class SensorX : public Sensor
{
public:
 // code specific to SensorX }
 virtual void open () { // code }
 virtual void getval() { // code }
 virtual void close(){ // code }
};
```

```
class IOTSys
{
 Sensor *m_pSensor;
public:
 IOTSys(Sensor *pSensor)
 {
  m_pSensor = pSensor;
 }
  void display() { pSensor->open();pSensor->getval(); pSensor->close();}
};
```

What's a bad design?

- We never intend to design anything bad
- We understand what is a bad design

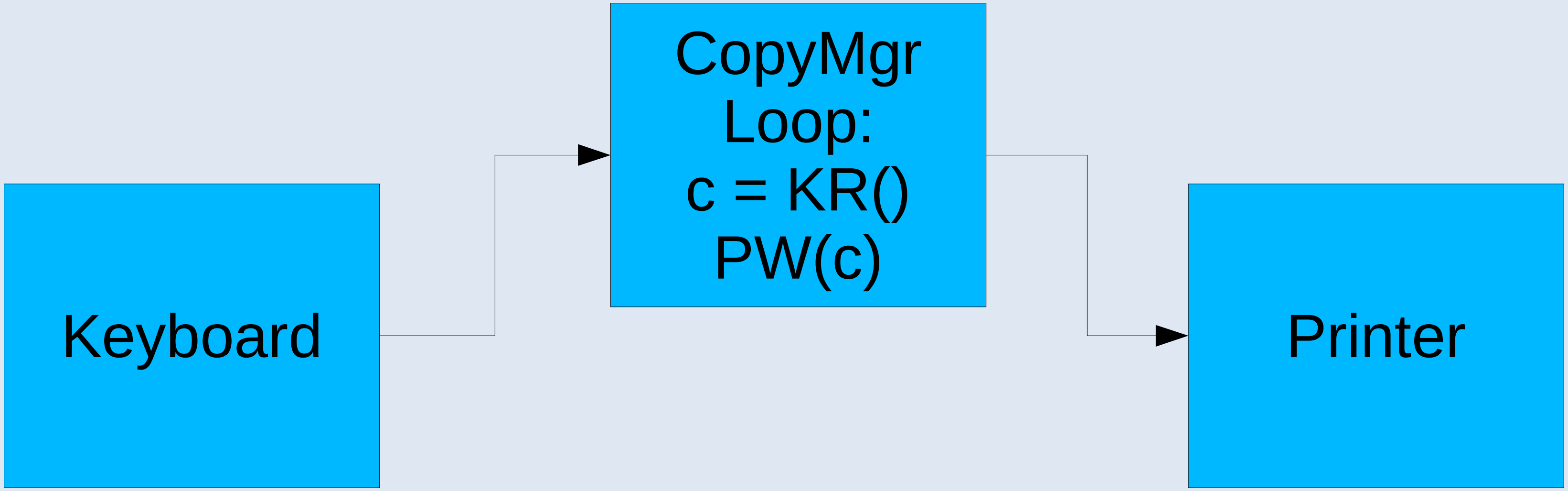
  a) One that is hard to change (Rigidity)

  b) When change was done then unexpected parts of system started breaking (Fragility)

  c) Hard to reuse in other application it is tightly entangled (Immobility)

Dependency Inversion

- E.g. If we have want to develop an application that reads from keyboard, copies in to memory and writes to printer
- How would you go about?



Exercise: Write classes for this.

Dependency Inversion

```
class Keyboard
{
 int opendev();
 unsigned char read();
 int closedev();
}

class Printer
{
 int opendev();
 int write(unsigned char);
 int closedev();
}
```

```
class CopyMgr
{
public:
 docopy()
 {
  Keyboard kb;
  Printer p;

  kb.opendev();
  p.opendev();
  bool end_flag = false;
  char ch;
  while (1)
  {
   ch = kb.read()
   if (ch == EOF)
     break;
   p.write(ch);
  }

  kb.closedev();
  p.closedev();
  }
};
```

Which class represents the core-logic?

CopyMgr

If we want to change it to read from disk – which component will have to be re-written?

CopyMgr

Is it a good design or bad design?

Bad design - does not allow reuse of the core component CopyMgr

Dependency Inversion

- Here high level component CopyMgr is dependent on low level component that it controls such as Keyboard & Printer
- If we make the higher level component independent of the low level component that it controls, then we can reuse it
- In general, dependency should be towards abstraction & not concrete classes
- Let us try to redesign it.

Dependency Inversion

```
class ReaderDevice
{
 int opendev()=0;
 unsigned char read()=0;
 int closedev()=0;
};

class Keyboard : public ReaderDevice
{
 int opendev();
 unsigned char read();
 int closedev();
};

class WriterDevice
{
 int opendev()=0;
 unsigned char write()=0;
 int closedev()=0;
};
class Printer : public WriterDevice
{
 int opendev();
 int write(unsigned char);
 int closedev();
};
```

```
class CopyMgr
{
public:
void docopy(ReaderDevice& rd, WriterDevice& wd)
{

 rd.opendev();
 wd.opendev();
 char ch;
 while (1)
 {
  ch = rd.read()
  if (ch == EOF)
   break;
  wd.write(ch);
 }
 rd.closedev();
 wd.closedev();
 }
};
```

-CopyMgr contains abstract reader and writer class

-CopyMgr depends on the abstractions - it does not depend on Keyboard or Printer

-Keyboard and Printer depends on abstract reader and writer class

-Dependencies “inverted”

Dependency Inversion Principle

1) High level modules should not depend upon low level modules. Both should depend upon abstractions.

2) Abstractions should not depend upon details. Details Should depend upon abstractions.

- The term “inverted” - traditional design thinking vouches for High Level Modules depend on low level module

The Liskov Substitution Principle (LSP, lsp) is a concept in Object Oriented Programming that states: Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

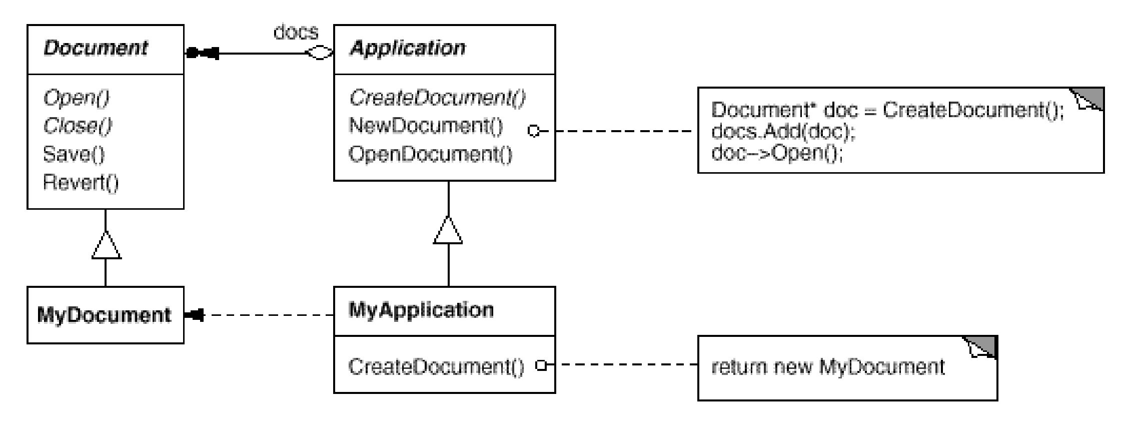
Cause for re-design

- To maximize reuse, the design should accept changes to existing requirements
- Causes for redesign
  - Creating object by directly specifying the class
    - Create it indirectly (Factory pattern)
  - Dependent on specific operation
    - Avoid hard-coding of requests (Chain of Responsibility pattern)
  - H/w, S/w dependence
    - Try making it independent of specific H/w or S/w
  - Dependence on object representation
    - Client that know how object is represented & use that fact. Client code will change when the object representation changes. Hide it. (Iterator pattern)
  - Tight Coupling
    - Tightly coupled classes makes is harder to re-use in isolation

Factory Method Design Pattern

- Problem
  - Consider a case of multiple document interface
  - When you do file→new, multiple types of documents can be created
  - If the framework has the “knowledge” of all the documents and application – it would not scale well
  - Extending it to support other document types will require modification to framework
  - Factory Method lets a class defer instantiation to subclasses
- Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses

Factory Method Design Pattern



•Application subclasses redefine an abstract CreateDocument operation on Application to return the appropriate Document subclass.

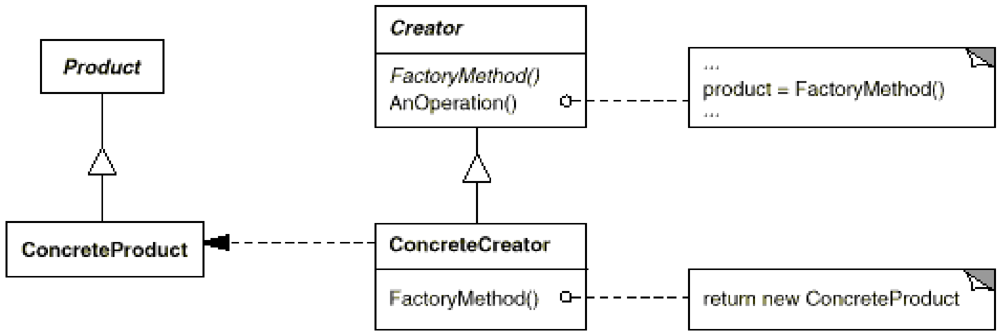
•Once an Application subclass is instantiated, it can then instantiate application-specific Documents without knowing their class.

•The **CreateDocument** is a factory method because it's responsible for "manufacturing" an object.

Factory Method Design Pattern

- Used When
  - a class can't anticipate the class of objects it must create.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

Hands-on



Let us do FactoryMethod exercise for the above design
Implement

1. an abstract Product class – method *GetName()=0*
2. two concrete class ProductA, ProductB
3. a class called Creator containing abstract method *Product * Create(int id)=0*
4. a class concrete subclass from Creator call it CreateSimple – implements Create and returns ProductA * or ProductB * based on id parameter
5. Let main() ask the user, a choice, ProductA or ProductB, based on selection create ProductA or ProductB using the CreatSimple class

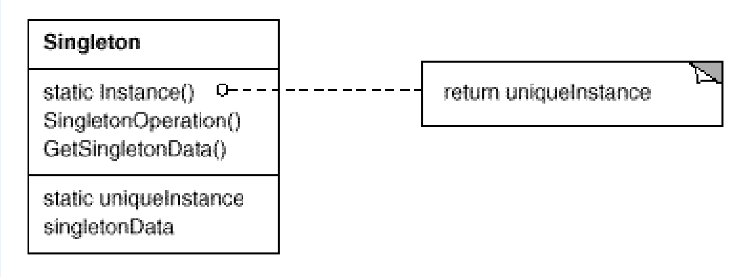
Singleton

- Problem:
  - There are many cases when only one instance is desired across the system. E.g one print spooler instance, one service configurator, one serial port handler
  - One can use global variables, but how do you ensure that there are no two global variable of the same type
- Intent:
  - Ensure a class only has one instance, and provide a global point of access to it

Singleton enforces only one instance of the class

Singleton

- Used When
  - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.



- Defines an Instance method that lets clients access its unique Instance

- Also may be responsible for creating its own unique instance

Singleton- Benefits

- Controlled access to sole instance
  - Can have strict control over how and when clients access it
- Reduced name space
  - Avoids polluting the global name space
- Flexible & more control
  - static member functions in C++ can never be virtual, so subclasses can't override them polymorphically

Singleton-hands-on

```
class Singleton
{
public:
        static Singleton* Instance();
        // constructor is protected,
// ensures only one instance cannot be
created outside the hierarchy methods
protected:
        Singleton() {}
private:
        static Singleton* _instance;

};

Singleton* Singleton::_instance = NULL;

Singleton* Singleton::Instance()
{
        // lazy initialization, only with
Instance() is requested it initializes
_instance
        if (_instance == NULL)
        {
                _instance = new
Singleton();
        }
        return _instance;
}
```

```
int main(int argc, char* argv[])
{
        Singleton* pSingleton =  Singleton::Instance();
        Singleton* pSingleton2 = Singleton::Instance();

        return 0;
}
```

Singleton – Hands-on

- Write a logger class using Singleton pattern
- Instance() method should return the existing instance of logger class
- Log (message) should log the message to a log file
- Open() - Should open the logger file
- Close() - Should close the logger file

Using singleton in logger context is useful – since we do not have to be worried about what is the current logger context and how to change it. Since there is only one instance

Singleton-Logger

```cpp
#include <iostream>
#include <fstream>

class Logger
{
public:
        static Logger* Instance();
        static void open();
        static void log(const char *message, int log_level);
        static void close();

        // constructor is protected, ensures only one instance will be created
protected:
        Logger() {}

        static int log_level;
        static bool inited;

        static const char * const LogFileName;
        static ofstream ofs;

private:
        static Logger* _instance;
};
```

```cpp
Logger* Logger::_instance = NULL;
const char* const Logger::LogFileName = "msg.log";
bool Logger::inited = false;
ofstream Logger::ofs;

Logger* Logger::Instance()
{
        // lazy initialization, only with Instance() is requested it initializes _instance
        if (_instance == NULL)
        {
                _instance = new Logger();
        }
        return _instance;
}

void Logger::open()
{
        if (!inited)
        {
                ofs.open(LogFileName);
                if (!ofs.good())
                {
                        throw runtime_error("Unable to initialize");
                }
                inited = true;
        }
}
```

Singleton- Logger

```
void Logger::log(const char *msg, int level)
{
        if (!inited) {
                open();
        }
        ofs << level << ":: " << msg<< endl;
}

void Logger::close()
{
        if (inited) {
                ofs.close();
                inited = false;
        }

}
```

```
int main(int argc, char* argv[])
{
        Logger* pLogger= Logger::Instance();
        pLogger->log("hello1", 1);
        pLogger->log("hello2", 1);
        pLogger->log("hello3", 1);
        pLogger->close();

        return 0;
}
```

Is this code multithreaded safe?
No. Thread-safety needs to be implemented for multi-threading use

Singleton- Logger

- If we change the existing logger class could be to move the open and close member functions to constructor and destructor respectively
  - By this change we would not be able to call close() to close the filestream; which we could have done prior to system shutdown
  - Note that we are creating an instance using new operator, so during system shutdown – the delete of _instance will not be called automatically
  - We would have to create a static cleanup class instance inside the Instance method & the destructor of the cleanup class will delete the instance pointer

Singleton - ServiceLocator

```
using namespace std;
#include <map>
class ComponentServiceMap
{
protected:
        static ComponentServiceMap CSMapInstance;
        std::map<string, void *> mComponentBroker;
private:
        ComponentServiceMap() { cout << "in constructor " << endl; }

        virtual ~ComponentServiceMap() { cout << "destructor" << endl; };
public:
        bool RegisterServiceForName(string service_name, void *service_ptr)
        {
                auto res = mComponentBroker.insert(make_pair(service_name, service_ptr));
                return res.second;
        }
        void *LookupServiceByName(string service_name)
        {
                auto it = mComponentBroker.find(service_name);
                if (it == mComponentBroker.end())
                {
                        string message = service_name + ": Not Found";
                        throw out_of_range(message);
                }
                return it->second;
        }
```

Singleton - ServiceLocator

```
        void DeleteService(string service_name)
        {
                mComponentBroker.erase(service_name);
        }
        static ComponentServiceMap& GetInstance() { return CSMapInstance; }
};
```

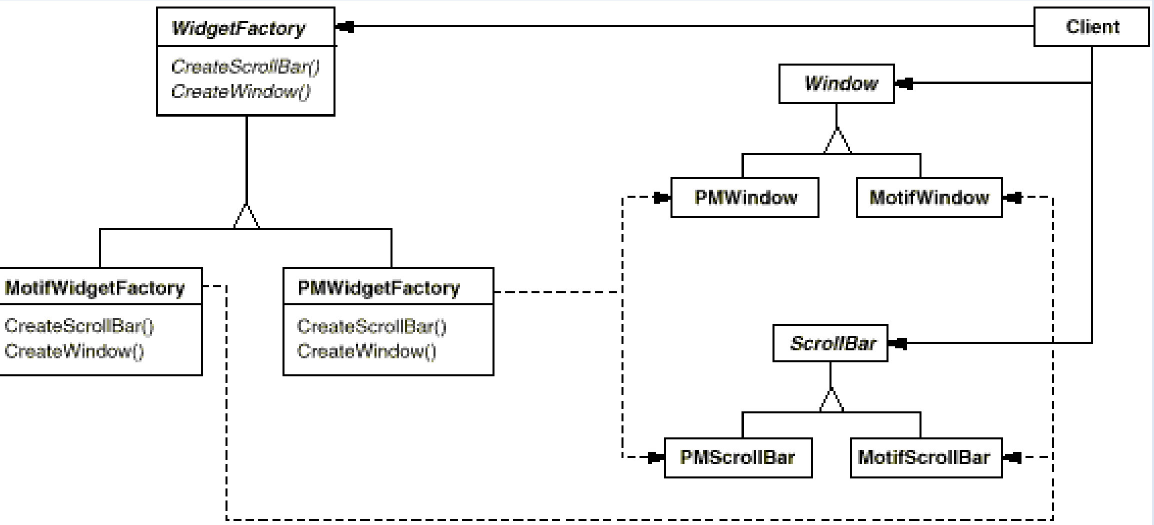
```
ComponentServiceMap ComponentServiceMap::CSMapInstance;

int main(int argc, char* argv[])
{
char *svc_ptr1 = "Time Service"; // this must point to Time Service
char *svc_ptr2 = "Web Service"; // this must point to Web Service
string service_name1 = "time";
string service_name2 = "web";

ComponentServiceMap::GetInstance().RegisterServiceForName(service_name1, svc_ptr1);
ComponentServiceMap::GetInstance().RegisterServiceForName(service_name2, svc_ptr2);

char *sptr1 = static_cast<char *> (ComponentServiceMap::GetInstance().LookupServiceByName("time"));
cout << "Looking up time service:" << sptr1 << endl;

char *sptr2 = static_cast<char *> (ComponentServiceMap::GetInstance().LookupServiceByName("web"));
cout << "Looking up web service:" << sptr2 << endl;
}
```

Abstract Factory

- Problem
  - Consider a case where you have to implement GUI. And there are different presentation manager libraries available. The gui widgets such as buttons, scroll bars may have a slightly different appearance in these sets.
  - By design, we need to have a way such that these can be changed as and when needed
  - If we code towards one widget manager then it may be hard to change the look & feel later.
  - Abstract Factory design helps here...

Abstract Factory

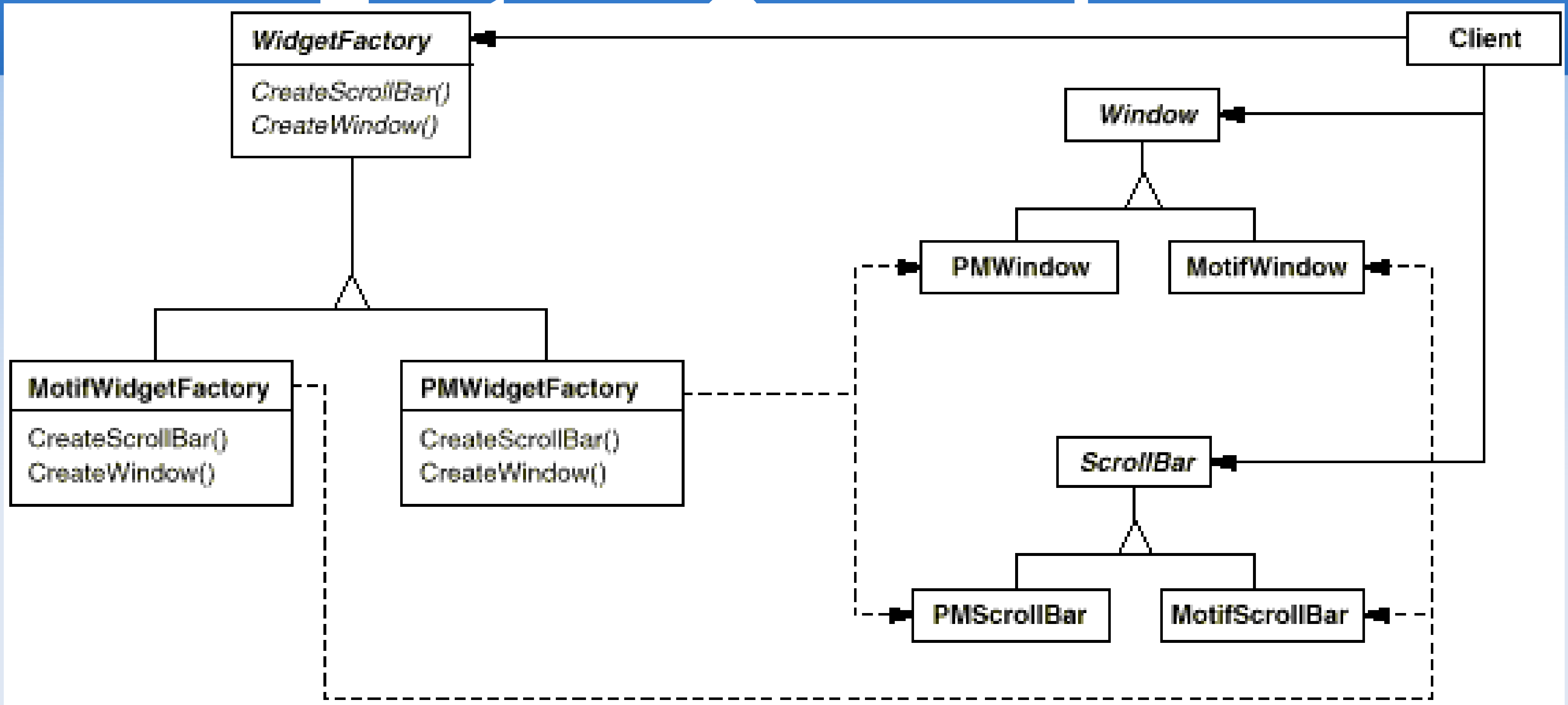
- Intent
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes

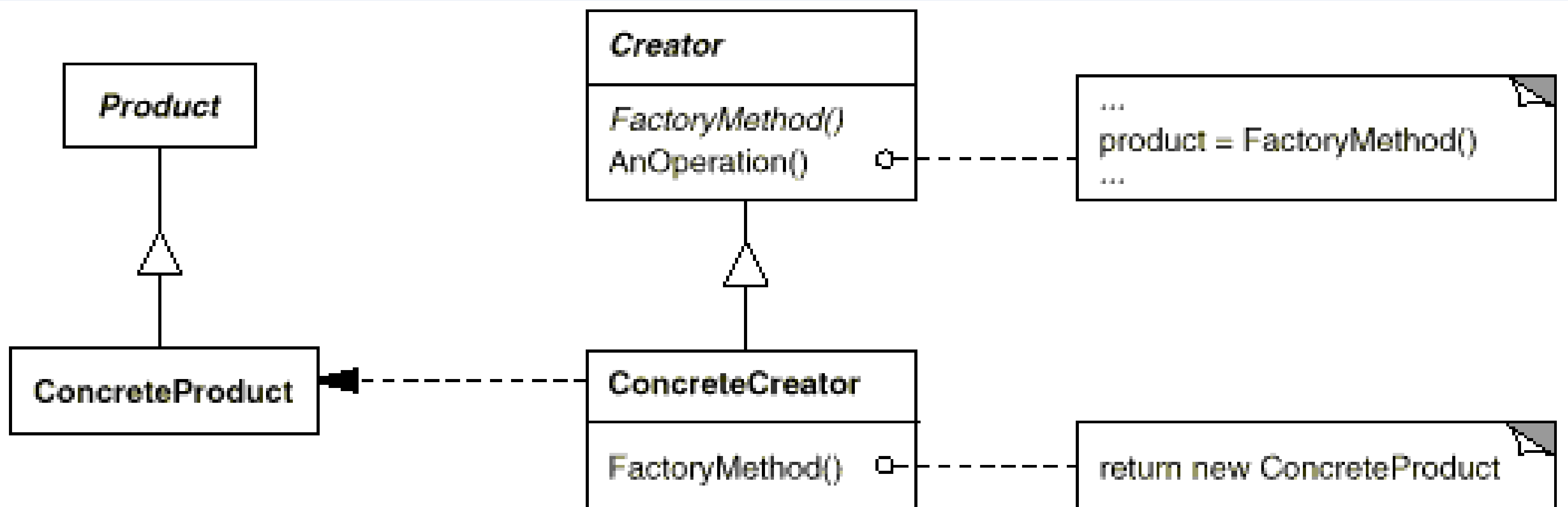


1. Define an abstract WidgetFactory class – it will have interface to create basic widgets
2. For each Widget there is also an abstract class
The concrete widget class will create the appropriate look and feel widget
3. Client will use Widget Factory's interface to get the appropriate concrete widget pointers
4. Client is not aware of exact concrete class it is using – thereby making it independent of the type of family widgets

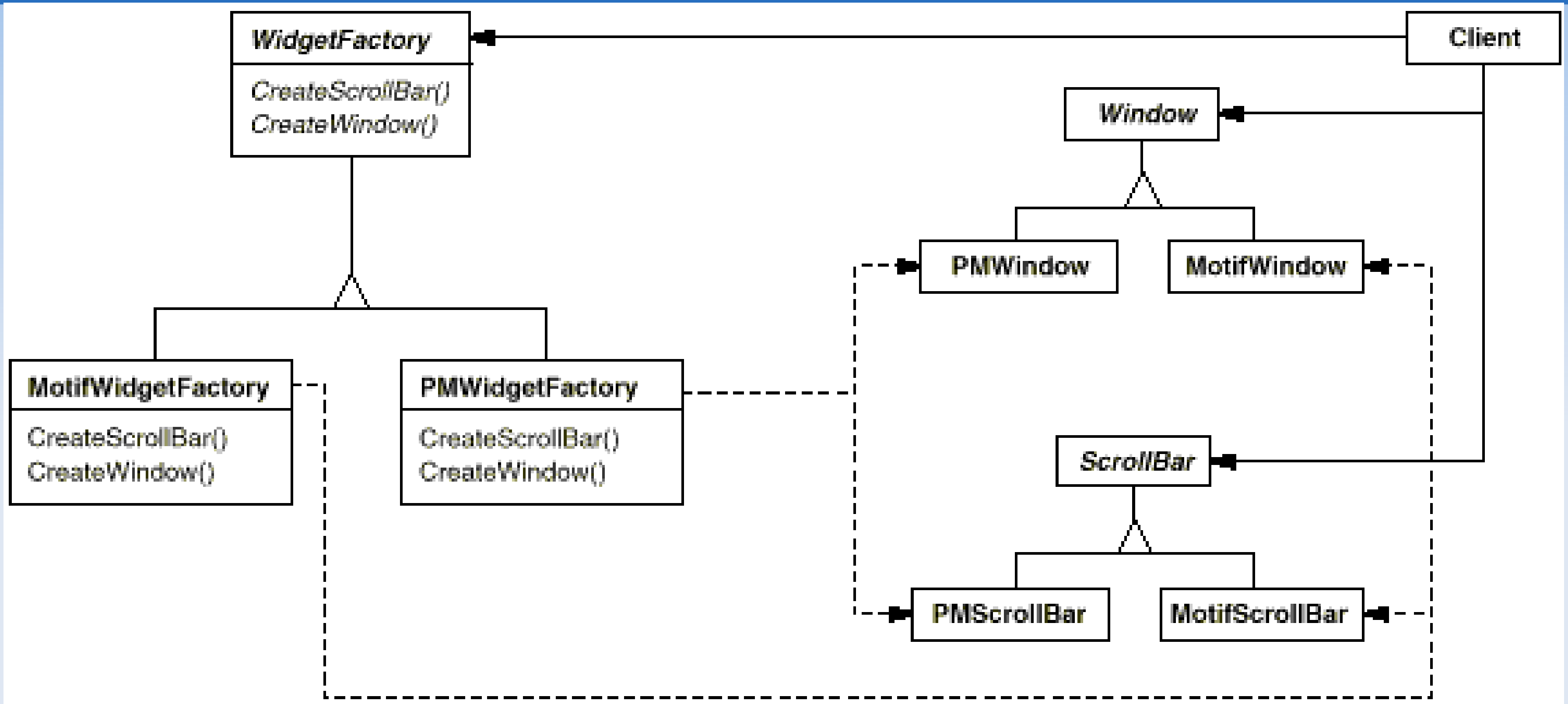
The concrete factory can be singleton

Factory Method & Abstract





Exercise



- Write AbsractFactory class for the above – you can use concrete AwidgetFactory, BWidgetFactory; AWindow, BWindow, AScrollBar, BScrollBar
- The client class will have CreateWindow, CreateScrollBar methods with a simple provision to use either AwidgetFactory or BWidgetFactory

Abstract Factory

```cpp
class Window
{
public:
        virtual const char *getName() = 0;
};

class ScrollBar
{
public:
        virtual const char *getName() = 0;

};
class AWindow : public Window
{
public:
        virtual const char *getName()
        {
                return "AWindow";
        }
};
class BWindow : public Window
{
public:
        virtual const char *getName()
        {
                return "BWindow";
        }
};
```

```cpp
class AScrollBar : public ScrollBar
{
public:
        virtual const char *getName()
        {
                return "AScrollBar";
        }
};
class BScrollBar : public ScrollBar
{
public:
        virtual const char *getName()
        {
                return "BScrollBar";
        }
};
class WidgetFactory
{
public:
        virtual Window *CreateWindow() = 0;
        virtual ScrollBar *CreateScrollBar() = 0;
};
```

Abstract Factory

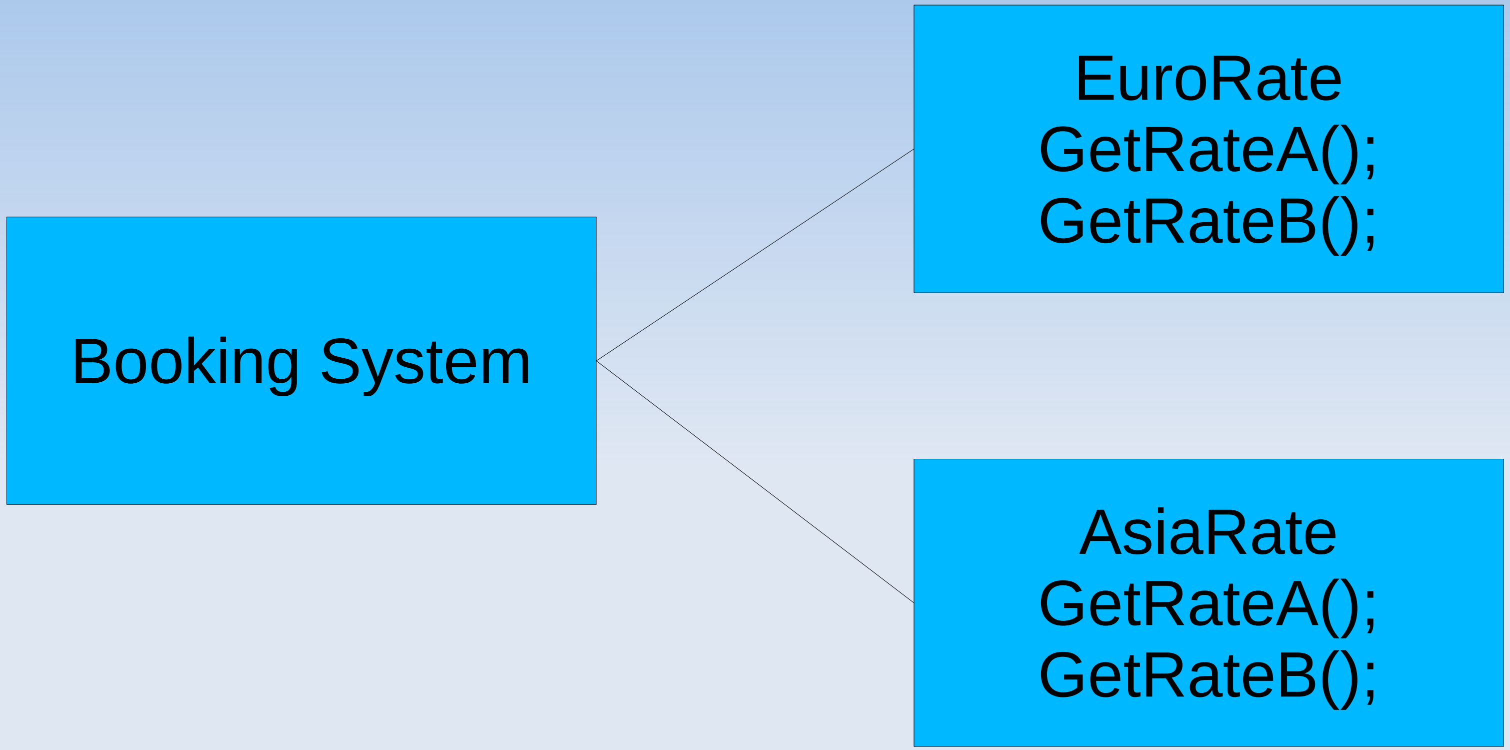
```cpp
class AWidgetFactory : public WidgetFactory
{
public:
        virtual Window *CreateWindow()
        {
                return new AWindow();
        }
        virtual ScrollBar *CreateScrollBar()
        {
                return new AScrollBar();
        }
};
class BWidgetFactory : public WidgetFactory
{
public:
        virtual Window *CreateWindow()
        {
                return new BWindow();
        }
        virtual ScrollBar *CreateScrollBar()
        {
                return new BScrollBar();
        }
};
```

```cpp
class Client
{
protected:
        WidgetFactory *pwFamily;

public:
        Client(WidgetFactory *_pwFamily)
        {
                pwFamily = _pwFamily;
        }

        Window *CreateWindow()
        {
                return pwFamily->CreateWindow();
        }

        ScrollBar *CreateScrollBar()
        {
                return pwFamily->CreateScrollBar();
        }
};
```

Abstract Factory

```
int main(int argc, char* argv[])
{
        AWidgetFactory awf;
        Client *pClient = new Client(&awf);
        ScrollBar *pScrollBar = pClient->CreateScrollBar();
        Window *pWindow = pClient->CreateWindow();
        std::cout << pScrollBar->getName() << endl;
        std::cout << pWindow->getName() << endl;

        return 0;
}
```

Abstract Factory

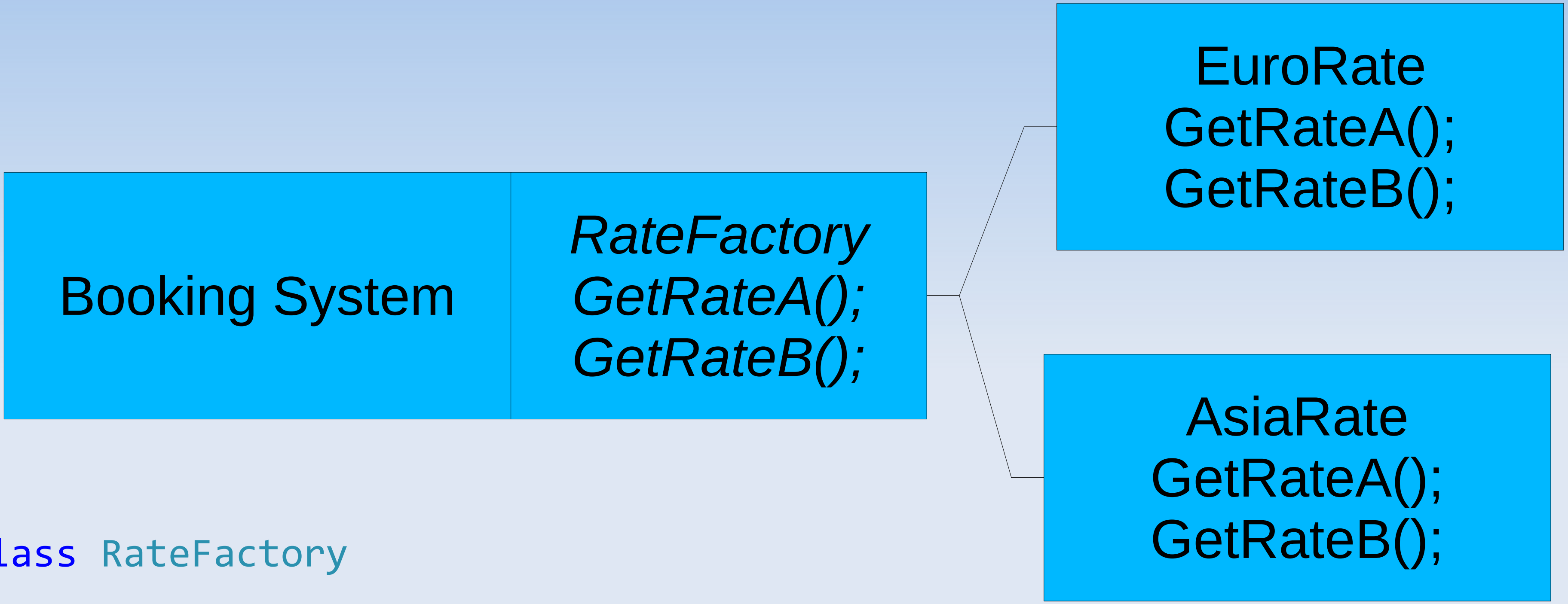


Booking System

EuroRate
GetRateA();
GetRateB();

AsiaRate
GetRateA();
GetRateB();

Abstract Factory



Booking System

RateFactory
GetRateA();
GetRateB();

EuroRate
GetRateA();
GetRateB();

AsiaRate
GetRateA();
GetRateB();

```
class RateFactory
{
public:
      virtual int GetRateA() = 0;
      virtual int GetRateB() = 0;
};

class EuroRate : public RateFactory
{
public:
      virtual int GetRateA();
      virtual int GetRateB();
};
```

```
class AsiaRate : public RateFactory
{
public:
      virtual int GetRateA();
      virtual int GetRateB();
};
```

Abstract Factory

```
class RateFactory
{
public:
      virtual int GetRateA() = 0;
      virtual int GetRateB() = 0;
};

class EuroRate : public RateFactory
{
public:
      virtual int GetRateA();
      virtual int GetRateB();
};

class AsiaRate : public RateFactory
{
public:
      virtual int GetRateA();
      virtual int GetRateB();
};
```

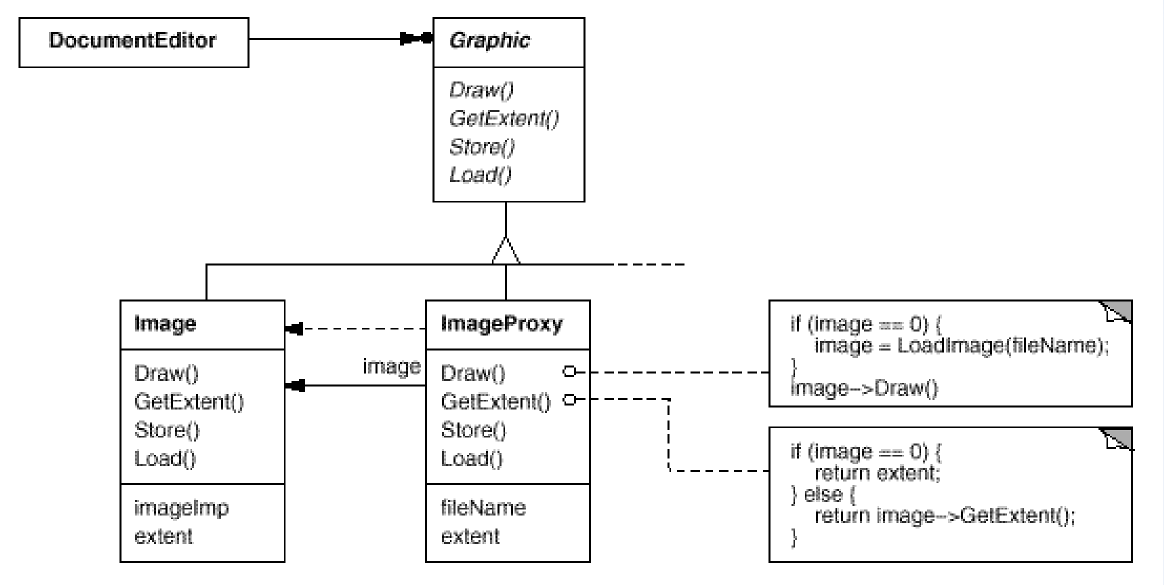
```
class BookingSystem
{
      RateFactory *pRate;
public:
      BookingSystem(RateFactory *_pRate)
      {
            pRate = _pRate;
      }
      virtual int GetRateA()
      {
            pRate->GetRateA();
      }
      virtual int GetRateB()
      {
            pRate->GetRateB();
      }
};
```

Proxy

- There are cases when we want to defer the cost of creation / initialization of the object until we really need to use it
- Example a document with text and images.
- We need the document to be opened quickly. Some images that are down below may not be visible & need not be created (as it would be expensive)
- How can we hide this fact from the editor that it is “on demand” created. We do not want to complicate/ change the editor code
- Proxy helps here
    - Use another image proxy that acts as a stands-in for real image

Proxy

- Intent
  - Provide a placeholder for another object to control access to it.



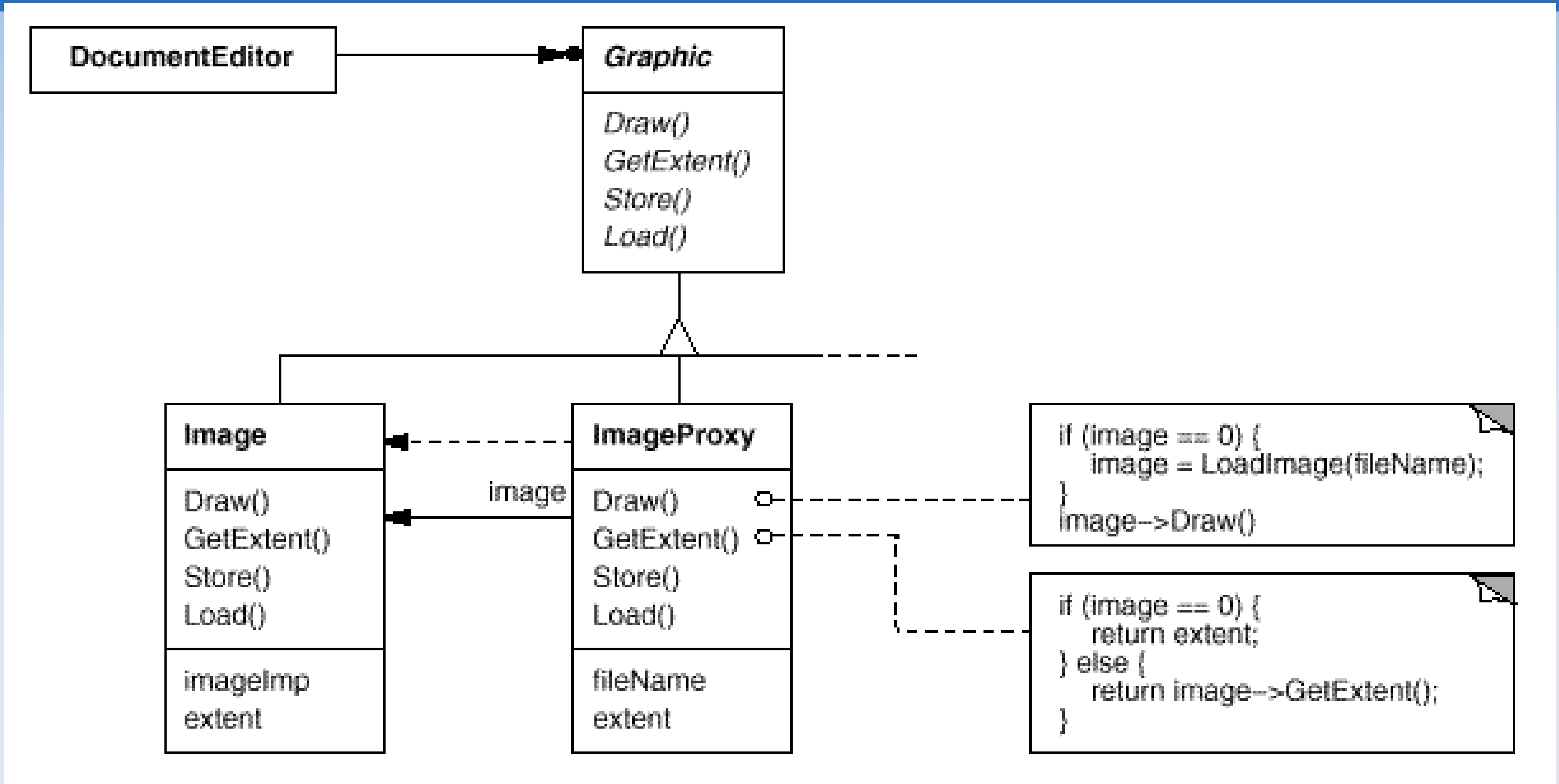
```
if (image == 0) {
    image = LoadImage(fileName);
}
image->Draw()
```

```
if (image == 0) {
    return extent;
} else {
    return image->GetExtent();
}
```

1. ImageProxy has the path to image file & has a reference to the image.
2. Here it just keeps the extent outline (bounding box)
3. When it is requested a Draw, then it creates the real image object by using LoadImage and calls its Draw() method
4. The Document Editor just deals with Graphic objects

Using has-a relationship

Exercise:Proxy



DocumentEditor

Graphic

Draw()
GetExtent()
Store()
Load()

Image

Draw()
GetExtent()
Store()
Load()

imageImp
extent

image

ImageProxy

Draw()
GetExtent()
Store()
Load()

fileName
extent

```
if (image == 0) {
    image = LoadImage(fileName);
}
image-->Draw()
```

```
if (image == 0) {
    return extent;
} else {
    return image-->GetExtent();
}
```

- Write classes for the above scenario
- Note this is using has-a relationship, imageProxy has image

Exercise: Proxy

- A gaming system has player class
- The player has sendmessage() method that returns a response string
- if network connectivity is not present, the player will not be able to send message
- The Player should not be concerned with network connectivity
- Write a PlayerProxy subclass that checks for network connectivity and forwards the sendmessage request to Player only when connectivity is there otherwise it returns a dummy message
- This is using is-a relationship

Proxy with is-a relationship

- Proxy in a way hides the latency / network issue

```
class Player
{
        string Name;
public:
        string getName() { return Name; }
        virtual string sendmessage(string message)
        {
                return networkSend(message);
        }
};


 class ProxyPlayer : public Player
 {
 public:
         virtual string sendmessage(string message)
         {
                 if (linkDown())
                 {
                         return "No connectivity";
                 }
                 else
                 {
                         return Player::sendmessage(message);
                 }
          }
 };
```

Game will be instantiated with ProxyPlayers

ProxyPlayer deals with all the latency related issues and will responding appropriately

We isolate the Player code with all environment related issues (here network connectivity) that may slow it down

Confirmation Notification

- In case of a ticket booking system – once the booking is confirmed then the system sends a text message notification with PNR number/eticket details

ProxyNotifier

```
SendSMS()
{
 Check for availability of service provider
 If available, pNotifier->SendSMS()
 Otherwise return error code
 OR
 Find the available telecom provider
 pNotifier->setProvider(provider);
 pNotifier->SendSMS()
}
```

Notifier

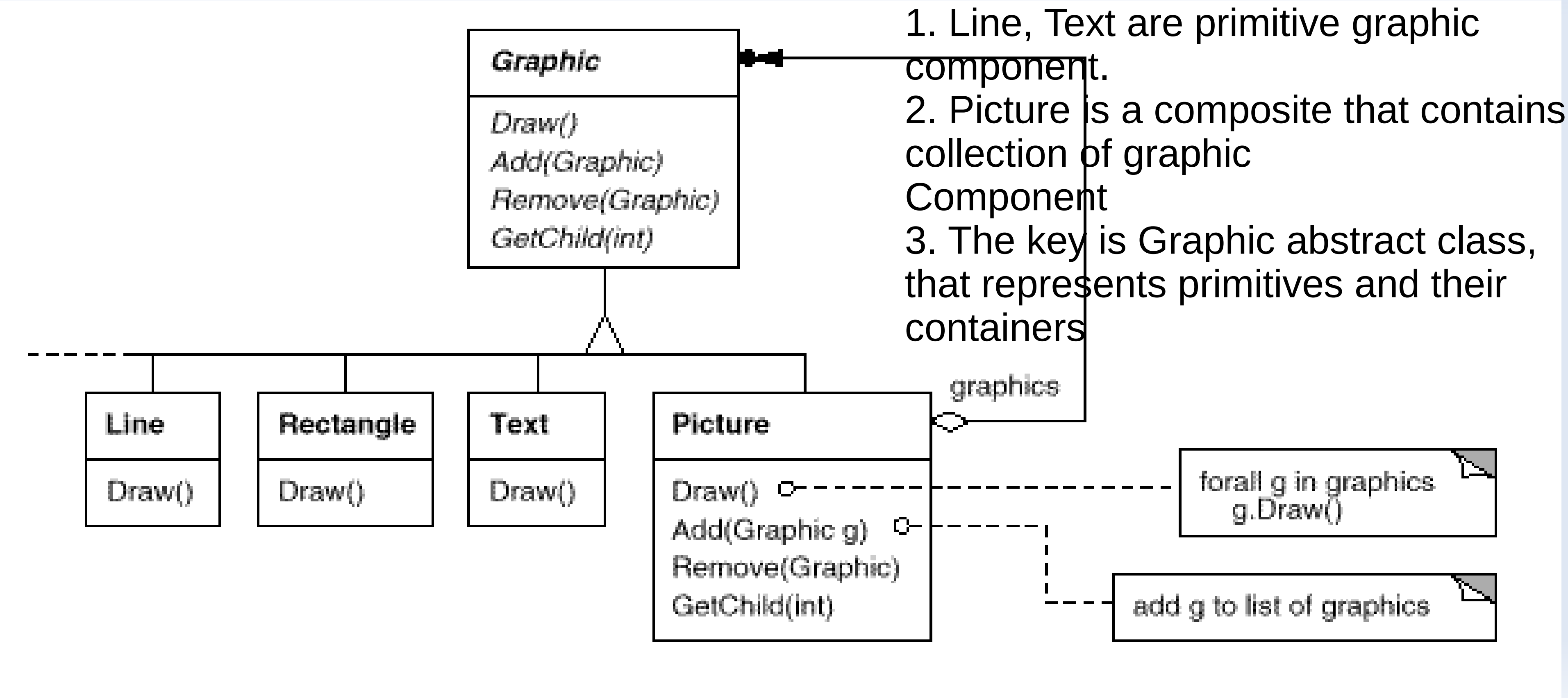
SendSMS()

Composite

- Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components.
- The user can group components to form larger components, which in turn can be grouped to form still larger components.
- A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.
- The key is to handle the primitive components and containers in the same way
- Composite design pattern helps here...

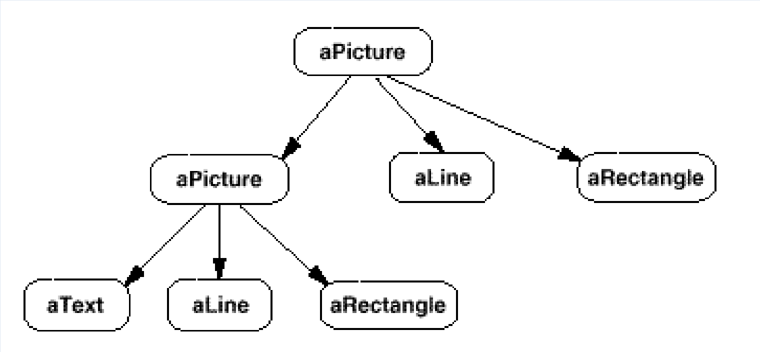
Composite

- Intent
  - Composite lets clients treat individual objects and compositions of objects uniformly

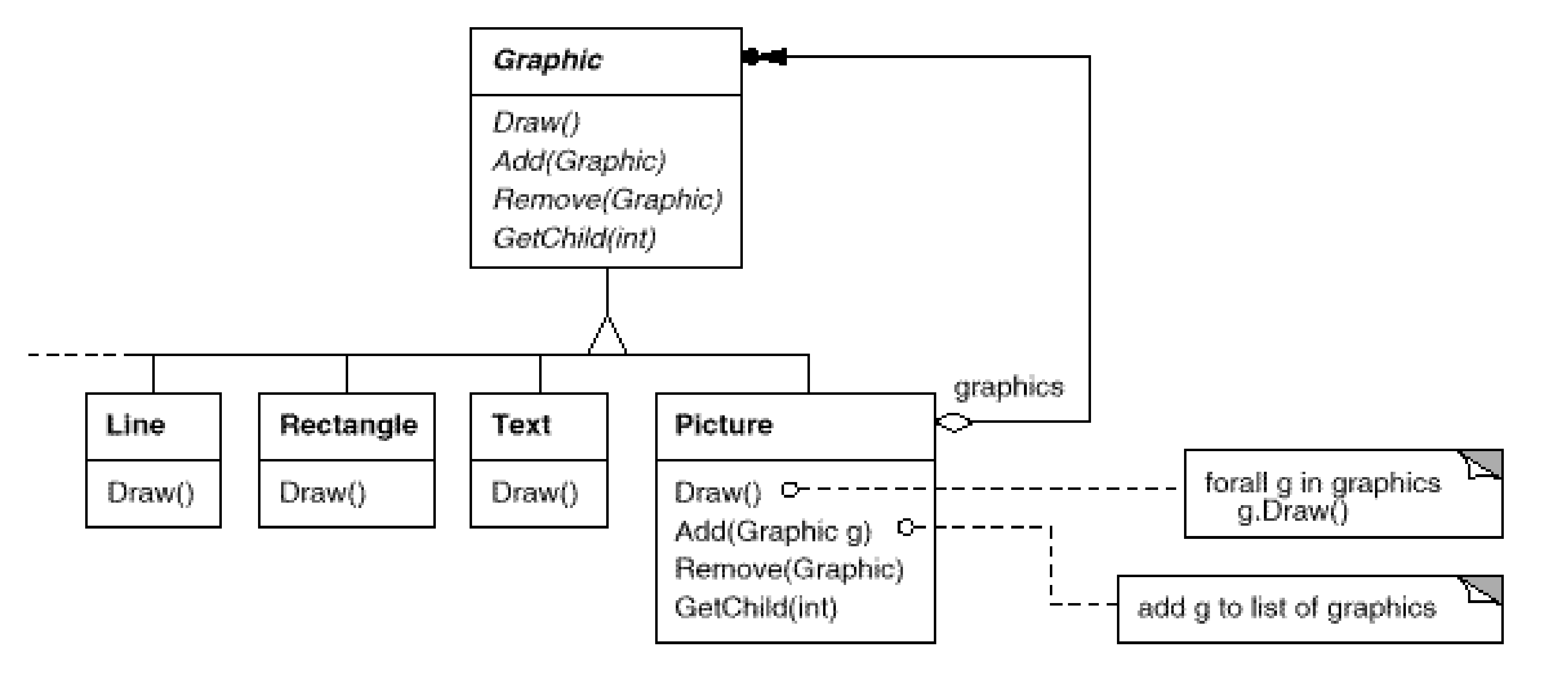


1. Line, Text are primitive graphic component.
2. Picture is a composite that contains collection of graphic Component
3. The key is Graphic abstract class, that represents primitives and their containers

- Used When
  - We want the client to ignore the difference between composition of objects and individual objects



Exercise: Composite



- Write class for the above scenario, Let Draw() print the name of the class
- Add() can return -1 if it is not able to handle the request; for e.g. Add makes sense only to composite class. For uniformity it is available to all
- Start with a composite object (Picture), keep adding other individual component to it. Then perform Draw() on Picture

Composite

```
#include <iostream>
#include <vector>
using namespace std;

class Graphic
{
public:
virtual void Draw() = 0;
virtual int Add(Graphic *pComponent)
{
 return -1;
}
};

class Line : public Graphic
{
public:
 virtual void Draw()
 {
 cout << "Line" << endl;
 }
};
```

```
class Rectangle : public Graphic
{
public:
virtual void Draw()
{
cout << "Rectangle" << endl;
}
};

class Text : public Graphic
{
public:
virtual void Draw()
{
cout << "Text" << endl;
}

};
}
```

```
class Picture : public Graphic
{
public:
 vector<Graphic*> graphics;

int Add(Graphic *pComponent)
{
 graphics.push_back(pComponent);
 return 0;
}
virtual void Draw()
{
for (auto component : graphics)
{
 component->Draw();
}
}
};
int main(int argc, char* argv[])
{
Graphic *gptr_base = new Picture();
gptr_base->Add(new Text());
gptr_base->Add(new Text());
gptr_base->Add(new Line());
Graphic *gptr = new Picture();
gptr_base->Add(gptr);
gptr->Add(new Text());
gptr->Add(new Rectangle());
gptr_base→Draw();
}
```

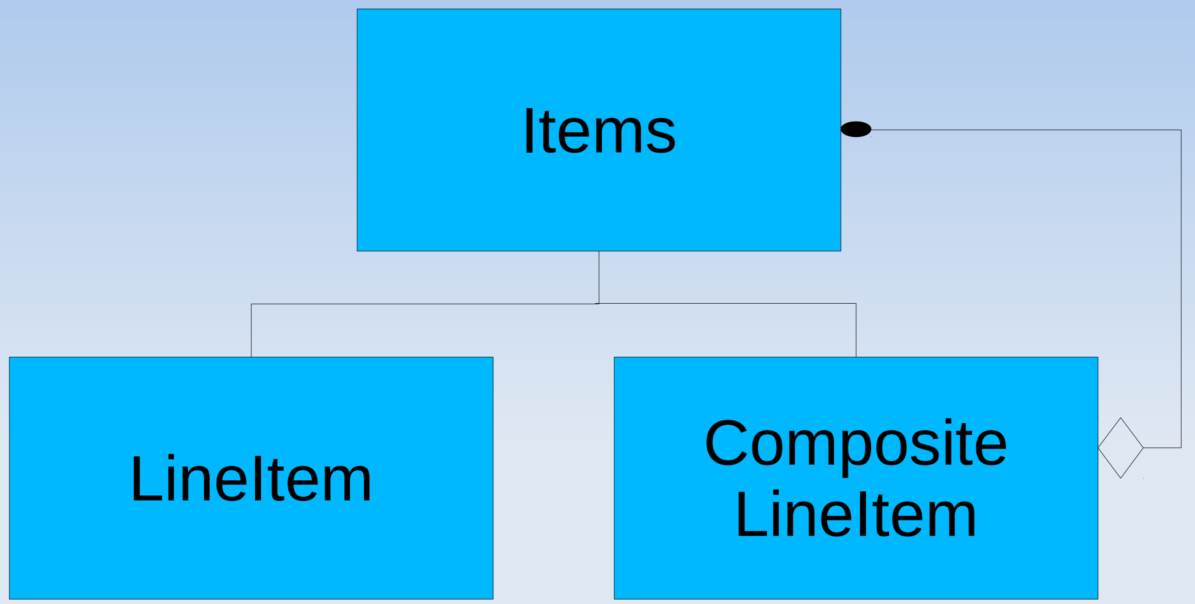
Any other examples ?

- Can you think about any other examples where composite design pattern can be applied?

Any other examples ?

- Can you think about any other examples where composite design pattern can be applied?
  - E.g. Any hierarchical structure having leaf and composite elements say - Files/Folders
  - Employees and subordinates (Add to employee will add within its collection of subordinates)

Example

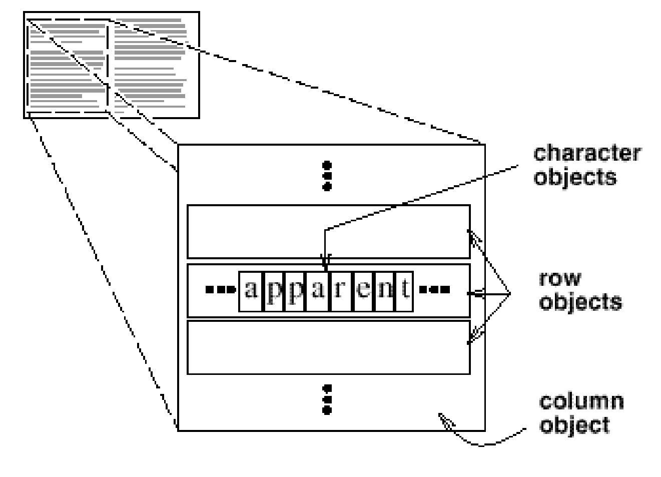


- An ordering system can have items
- Some items may be composite line items i.e. it is not one part but indicates a collection of sub items
- e.g. cricket bat (line item),
- Cricket-kit (containing cricket bat, ball, gloves, bag...)

Flyweight

- There are cases where lot of small similar objects being created, which, potentially could be shared
- Object-oriented document editors typically use objects to represent embedded elements like tables and figures.
- If they use object for each character. It would be increase flexibility. Characters and embedded elements could then be treated uniformly with respect to how they are drawn and formatted.
- The application could be extended to support new character sets without disturbing other functionality.
- The application's object structure could mimic the document's physical structure.
- But it suffers from a limitation

Flyweight



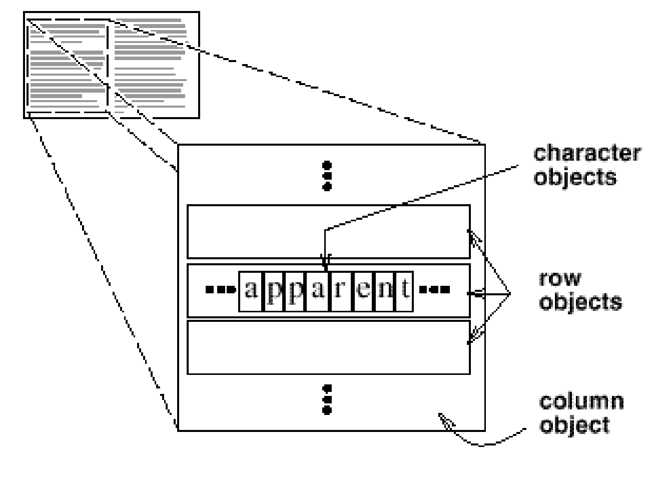
It will have lot more overhead in terms of number of objects, affecting runtime performance

For e.g. a small 2KB document will create over 2000 objects

Flyweight pattern helps here, it allows the objects to be shared & use it at the lowest level

A flyweight solves the issue of repeated representation of similar data – where it could have been shared instead.

Flyweight



A flyweight is a shared object that can be used in multiple context simultaneously

The flyweight acts as an independent object in each context

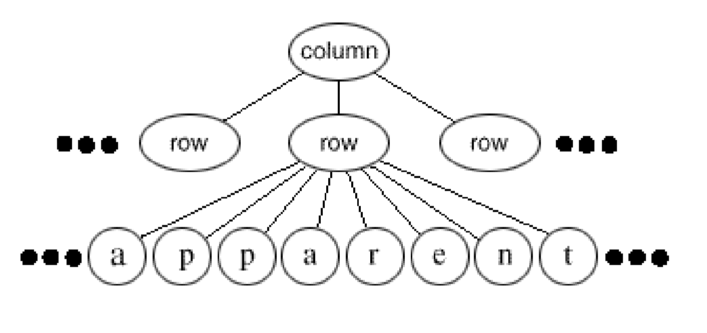
The ascii char set can be shared across the document – the information of where it appears in the document (Page/Row) should be stored independently

Flyweights cannot make assumptions about the context in which they operate

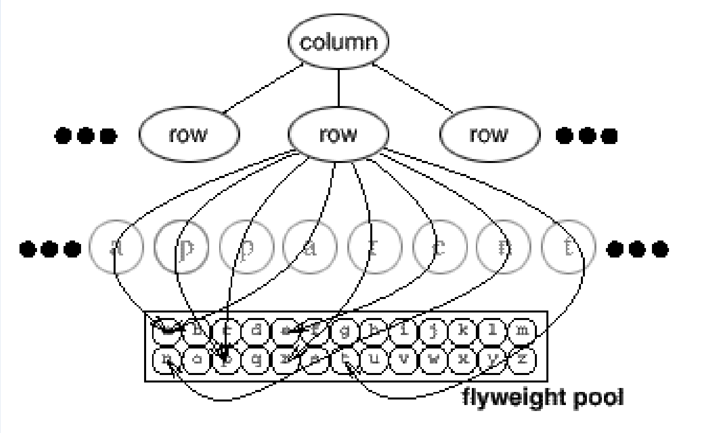
Two states exists – intrinsic and extrinsic
Intrinsic is store with the flyweight – it is independent of context e.g. char-set
Extrinsic – depends on and varies with flyweight's context – hence cannot be shared (e.g. position of character)
Client objects are responsible for passing extrinsic state to the flyweight when it needs it

Flyweight

- Logical Representation

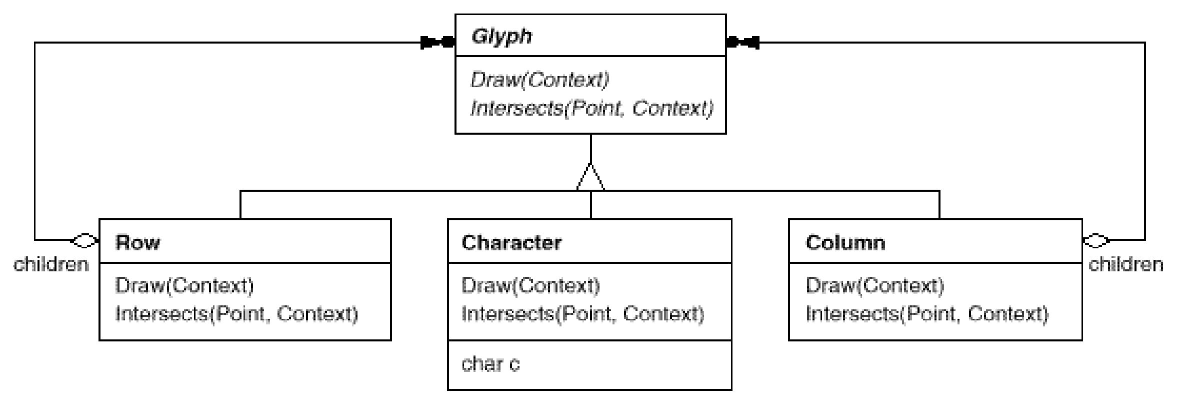


- Internal Representation



Here flyweight pool (say a vector) contains characters

Flyweight Representation


Glyph
Draw(Context)
Intersects(Point, Context)
Row
Draw(Context)
Intersects(Point, Context)
Character
Draw(Context)
Intersects(Point, Context)
char c
Column
Draw(Context)
Intersects(Point, Context)
children
children

- Character (our flyweight) that is a Glyph – it has only the character representation, no context such as font, position etc..
- Row and Columns have collection of Glyph
- With method Draw() a context is passed, which will tell position and style details

Exercise

- An expense needs to be processed,
- Engineer can approve upto Rs. 1000/-
- Manager level can approve > Rs 1000 and upto Rs. 5000/- and
- Director can approve greater than Rs 5000/- to 100000/-
- Beyond 100000/- no approval

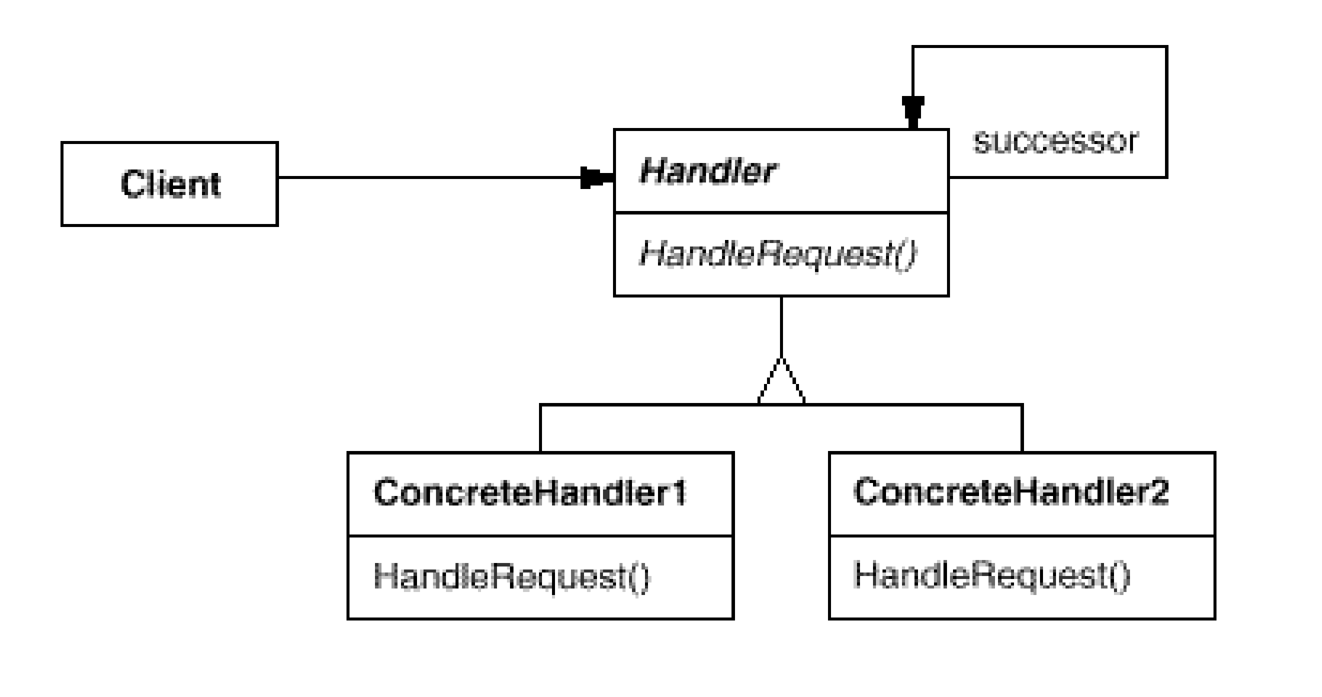
Chain of Responsibility

- Problem
  - There are cases when a request needs to be handled based on responsibility
  - E.g. expense approval – a manager can handle only x amount beyond which it has to be routed through next level and so on
  - The first object in the chain receives the request Based on who can handle the request, the COR will iterate through the successors
  - The object that made the request has no explicit knowledge of who will handle it

Chain of Responsibility

- An expense needs to be processed,
- Engineer can approve upto Rs. 1000/-
- Manager level can approve > Rs 1000 and upto Rs. 5000/- and
- Director can approve greater than Rs 5000/- to 100000/-
- Beyond 100000/- no approval
- Write a program using Chain of Responsibility

Chain of Responsibility



```
class ExpenseApprover          virtual bool ProcessRequest(int amount)
class EngineerEA : public ExpenseApprover
class ManagerEA : public ExpenseApprover
class DirectorEA : public ExpenseApprover
```

Chain of Responsibility

```
class ExpenseApprover
{
protected:
        ExpenseApprover *successor;
public:
        ExpenseApprover *getSuccessor();
        void setSuccessor(ExpenseApprover *ea);
        virtual const char *getName() = 0;
        virtual int getLimit() = 0;
        virtual bool ProcessRequest(int amount)
        {
                // implement
         }

};
```

```
class EngineerEA : public ExpenseApprover
class ManagerEA : public ExpenseApprover
class DirectorEA : public ExpenseApprover
```

Chain of Responsibility

```
class ExpenseApprover
{
protected:
        ExpenseApprover *successor;
public:
        ExpenseApprover *getSuccessor()
        {
                return successor;
        }
        void setSuccessor(ExpenseApprover *ea)
        {
                successor = ea;
        }
        virtual const char *getName() = 0;
        virtual int getLimit() = 0;
        virtual bool ProcessRequest(int amount)
        {
                if (amount <= getLimit())
                {
                        cout << "requested amount:" << amount << " approved by " << getName() << endl;
                        return true;
                }
                else
                {
                        if (getSuccessor() != NULL)
                        {
                                return getSuccessor()->ProcessRequest(amount);
                        }
                }
                return false;
        }

};
```

Chain of Responsibility

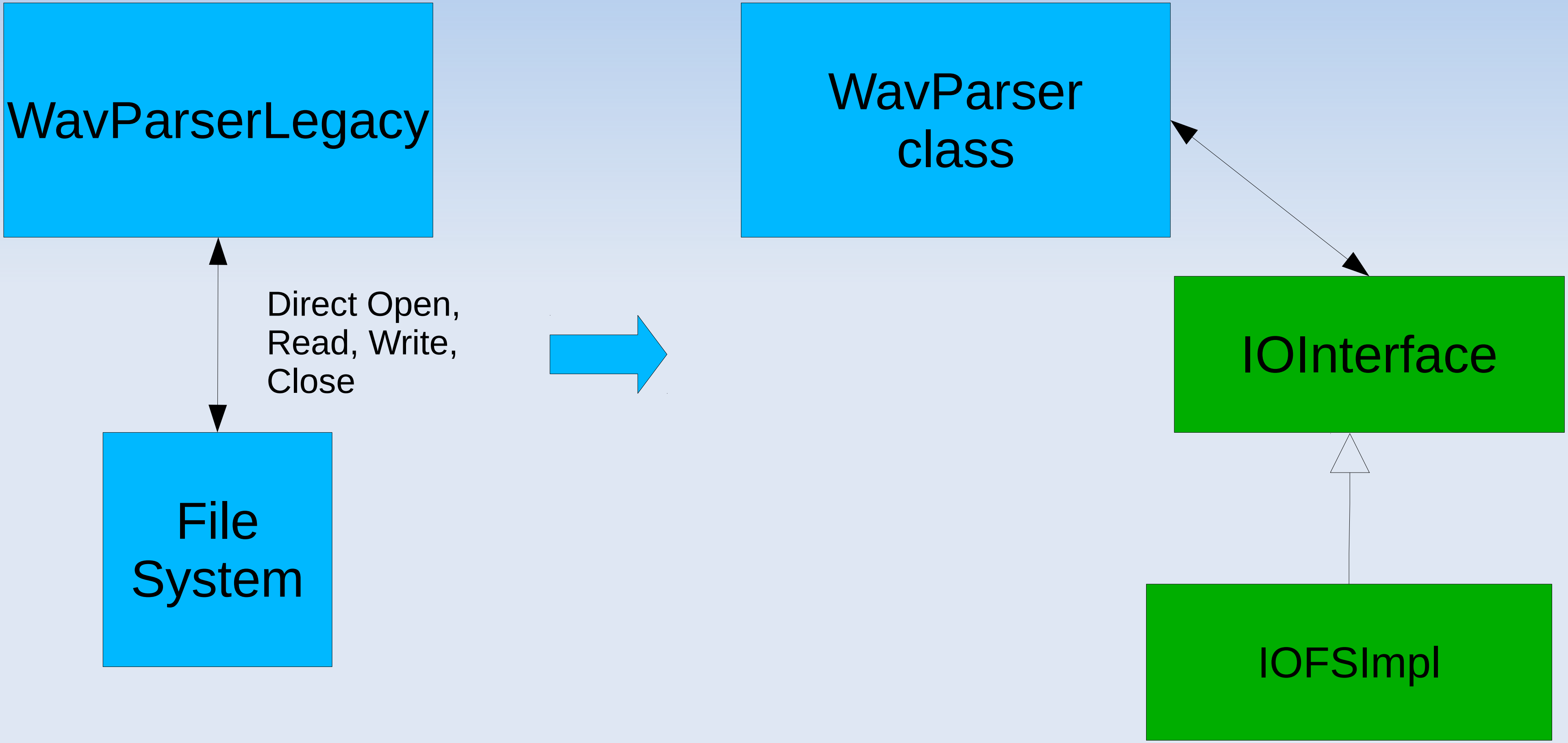
- Other Application
  - It could be a RBAC hospital system where based on their individual category responsibility, they are authorized to approve / handle certain tasks
  - It could be a message passing system, having low, medium and high priority messages – that can be handled by appropriate classes based on its severity
  - In GUI scenario, the mouse click, key-pressed event needs to propagate through different GUI elements
  - Error loggers also use this pattern
  - The key is the requester does not “know” who is going to finally handle the message

Pattern for Breaking Dependency

- Specially when we are refactoring the code, we will need to break the dependency
- One method is extracting interface

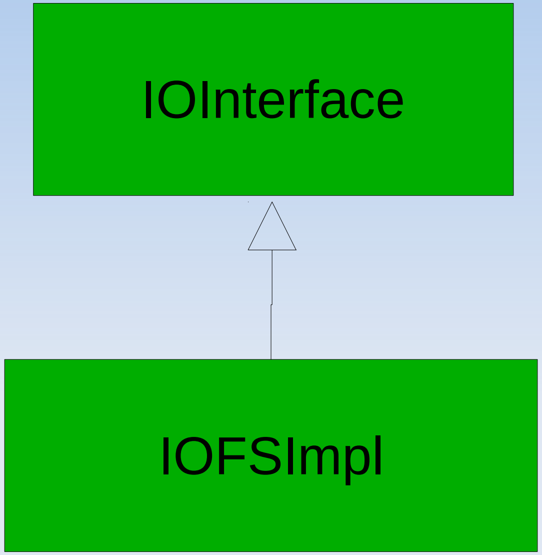
  (Design for Testability)

Breaking Dependency
Extracting Interface



- Direct system calls must go through *say IOInterface* to break the file system dependency

Abstracting it out...



We have to extract an interface and provide an implementation that uses the File System calls

- We create an abstract base class IOInterface
- Implement IOFSImpl

Exercise

- Create IOInterface.h file
- Create IOFSImpl.cpp
- Create WavParser class

IOInterface.h

```
class IOInterface
{
protected:
 string filename;
 string rwoptions;
public:
 IOInterface() { rwoptions = "r"; }
 void setfilename(string& fname) { filename = fname; }
 void setrwoptions(string& rwopt) { rwoptions = rwopt; }
 virtual int open()=0;
 virtual int close()=0;
 virtual int read(char *rd_data, int
          read_unit_size, int read_len)=0;
 virtual int write(const char *wr_data, int
          write_unit_size, int write_len)=0;
};
```

Download: http://narayaniyer.com/dp/WavParserLegacy.zip

Dependency Injection

- Constructor Injection: We tell the WavParser which concrete IOInterface to use at run time when we construct it.
- Setter Injection: We can provide a setter function that takes the dependency parameter (reference to concrete IOInterface) at run time

Exercise

- A ticketing system needs to have a save-as functionality in different format – say pdf, xml, txt...
- We know that, as time passes, new format keeps coming up.
- Design the class and implement the save-as functionality

Visitor

- Intent
    - Visitor lets you define a new operation without changing the classes of the elements on which it operates
- There is Element & Visitor class type
- Element is the business class, Visitor is the dispatcher mechanism
- Element has to accept visitor – i.e element has an accept() method.
- accepts method takes input as reference to Visitor type
- accept() method calls visit() of the Visitor and pass back its own reference (*this i.e element's reference)

Visitor

```
#include "stdafx.h"
#include <iostream>
using namespace std;

class Element;
class Element1;
class Element2;

class Visitor
{
public:
        virtual void visit(Element1& e) = 0;
        virtual void visit(Element2& e) = 0;
};
class Element
{
public:
        virtual void accept(Visitor& v) = 0;
};
```

```
class Element1 : public Element
{
public:
        int x;
        Element1() { x = 0; }

        virtual void accept(Visitor& v1)
        {
                v1.visit(*this );
        }
};

class Element2 : public Element
{
public:
        int y;
        Element2() { y = 1; }
        virtual void accept(Visitor& v)
        {
                v.visit(*this);
        }
};
```

Visitor

```
class Visitor1 : public Visitor
{

virtual void visit(Element1& e)
{
 cout << "visited for element1 " << e.x << endl;
}

virtual void visit(Element2& e)
{
 cout << "visited for element2 " << e.y << endl;
}

};
```

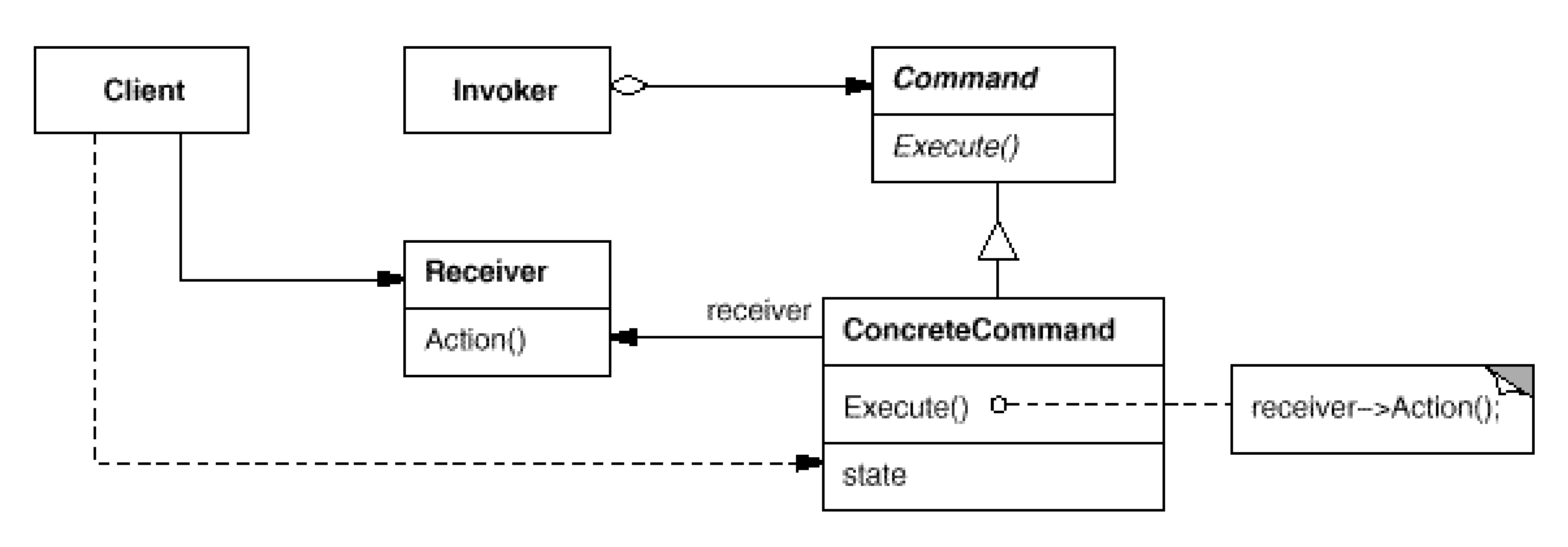
```
int main(int argc, char* argv[])
{
        Element *e1;
        e1 = new Element1();
        Visitor *v1;
        v1 = new Visitor1();

        e1->accept(*v1);

        Element *e2;
        e2 = new Element2();
        Visitor *v2;
        v2 = new Visitor1();

        e2->accept(*v2);
}
```

Exercise- now design this with visitor pattern

- A ticketing system needs to have a save-as functionality in different format – say pdf, xml, txt...
- We know that, as time passes, new format keeps coming up.
- Design the class and implement the save-as functionality

Command

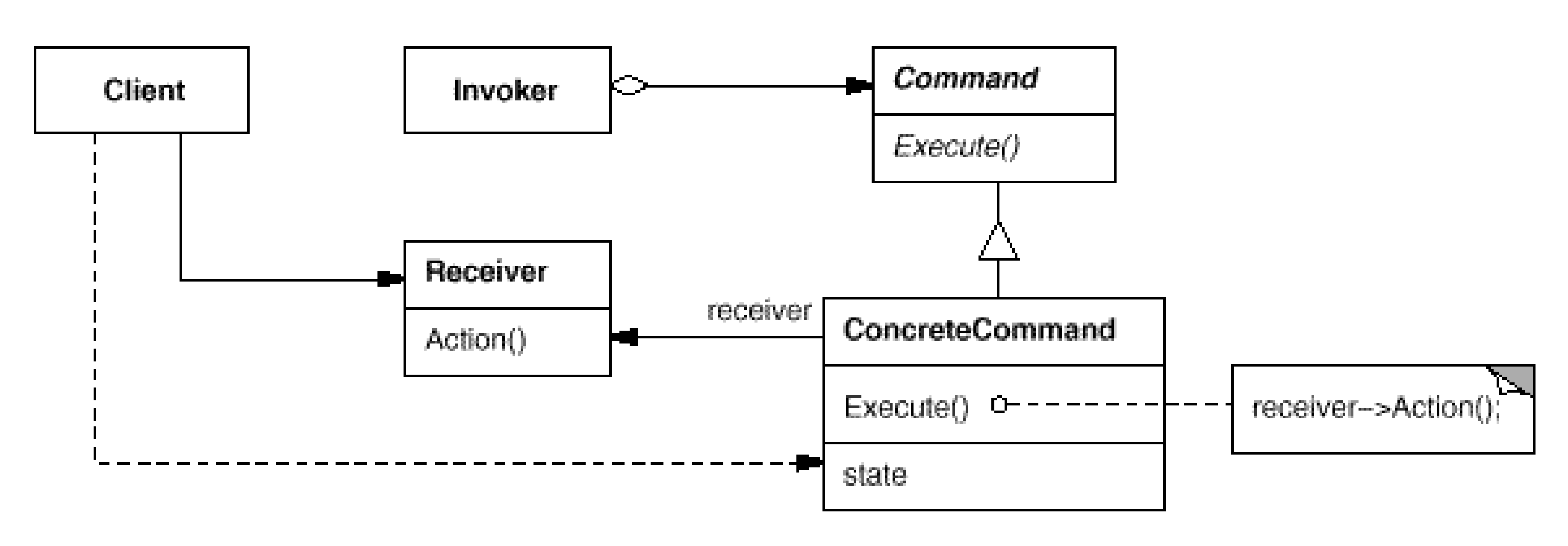
- Intent
  - To encapsulate all information needed to perform an action or trigger an event at a later time



Command decouples the object that invokes the operation from the one that knows how to perform it.

Command - Exercise

- Let Receiver be a Document, write a Paste command concrete class and let Menuitem be the invoker



Command decouples the object that invokes the operation from the one that knows how to perform it.

Command

Command
- declares an interface for executing an operation.

ConcreteCommand (PasteCommand, OpenCommand)
- defines a binding between a Receiver object and an action.
- implements Execute by invoking the corresponding operation (or multiple operation) on Receiver.

Client (Application)
- creates a ConcreteCommand object and sets its receiver.

Invoker (MenuItem)
asks the command to carry out the request.

Receiver (Document, Application)
-knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

Command

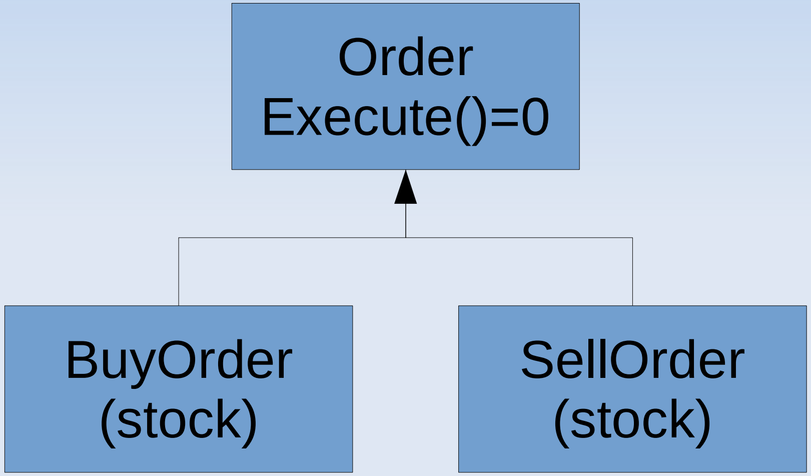
```
// The receiver
class Document
{
public:
 int Paste()
 {
   cout << "doing document paste " << endl;
   return 1;
}
};
// The Command
class Command
{
public:
       virtual int Execute() = 0;
protected:
       Command() {}
};

// The Concrete Command
class PasteCommand : public Command
{
public:
 PasteCommand(Document*_doc) { _document = _doc; }
       virtual int Execute() { return _document-
>Paste(); }
private:
       Document* _document;
};
```

```
// The invoker
class MenuItem
{
 Command *cptr;
 public:
 void StoreCommand(Command *_cptr)
 {
  cptr = _cptr;
 }
 void invoke()
 {
  cptr->Execute();
 }
};

MenuItem item;
// The Client
class Client
{
 public:

 void DoSetup(Document *_doc)
 {
 // Command cptr[0] = new OpenCommand(_doc);
 Command *cptr= new PasteCommand(_doc);
 item.StoreCommand(cptr);
 }
};
```

```
int main(int argc, char* argv[])
{
Document d;
Client c;
c.DoSetup(&d);
item.invoke();
return 0;
}
```

Stock example

command



```
void execute()
{
stock.buy();
}
```

```
void execute()
{
stock.sell();
}
```

```
void take_order(Order& order)
{
        orderList.add(order);
}
```

receiver

Stock
stock_name
Buy()
sell()

invoker

Broker
Orderlist

take_orders()
execute_orders()

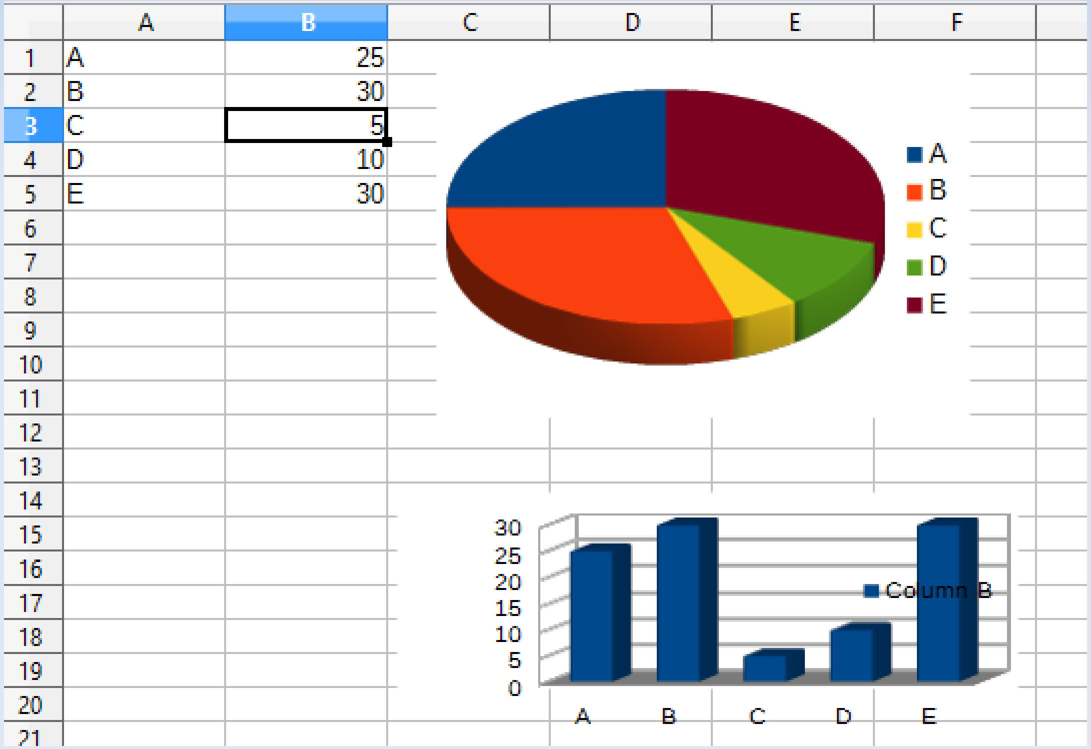
```
void executeOrders()
{
for (Order& order : orderList)
{
order.execute();
}
orderList.clear();
}
```

Orders are being taken as an object

Observer

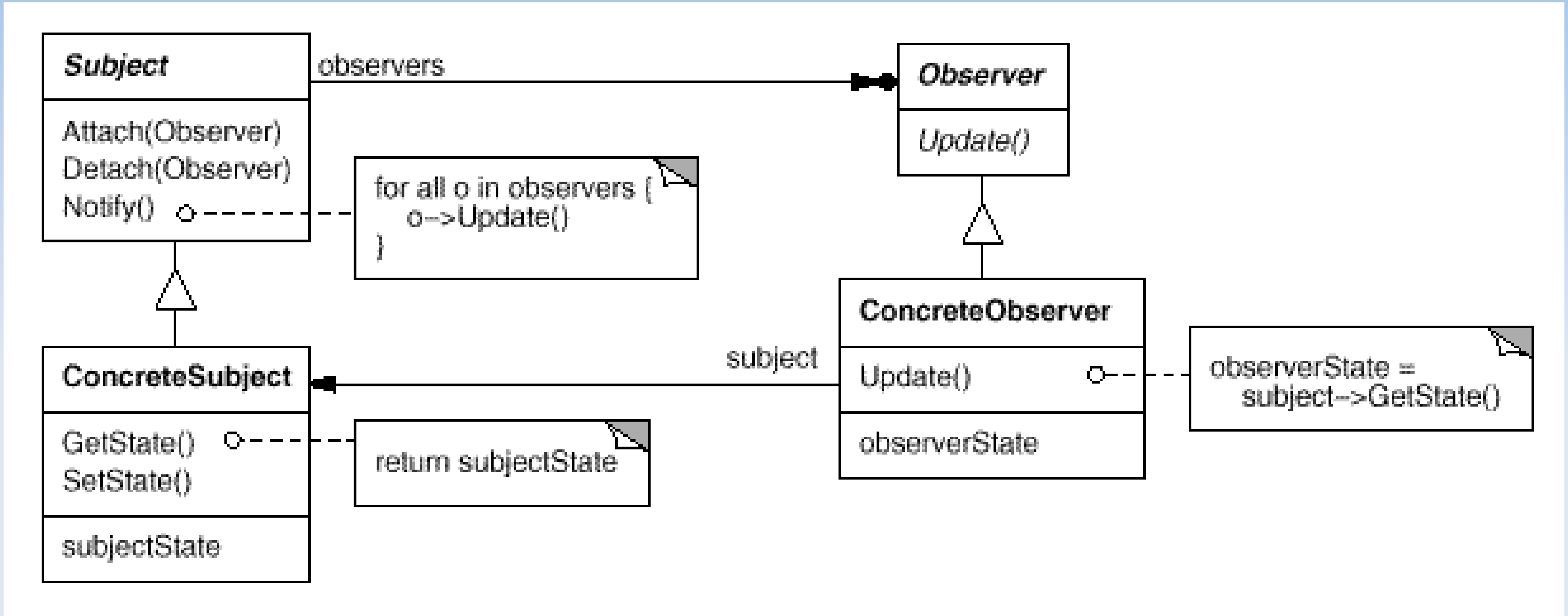
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Also know as publish-subscribe model relationships.
- The key objects in this pattern are subject and observer.
- A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state.
- In response, each observer will query the subject to synchronize its state with the subject's state.

Spreadsheet example



	A	B
1	A	25
2	B	30
3	C	5
4	D	10
5	E	30

Exercise - Observer



```
class Observer;

class Subject
```

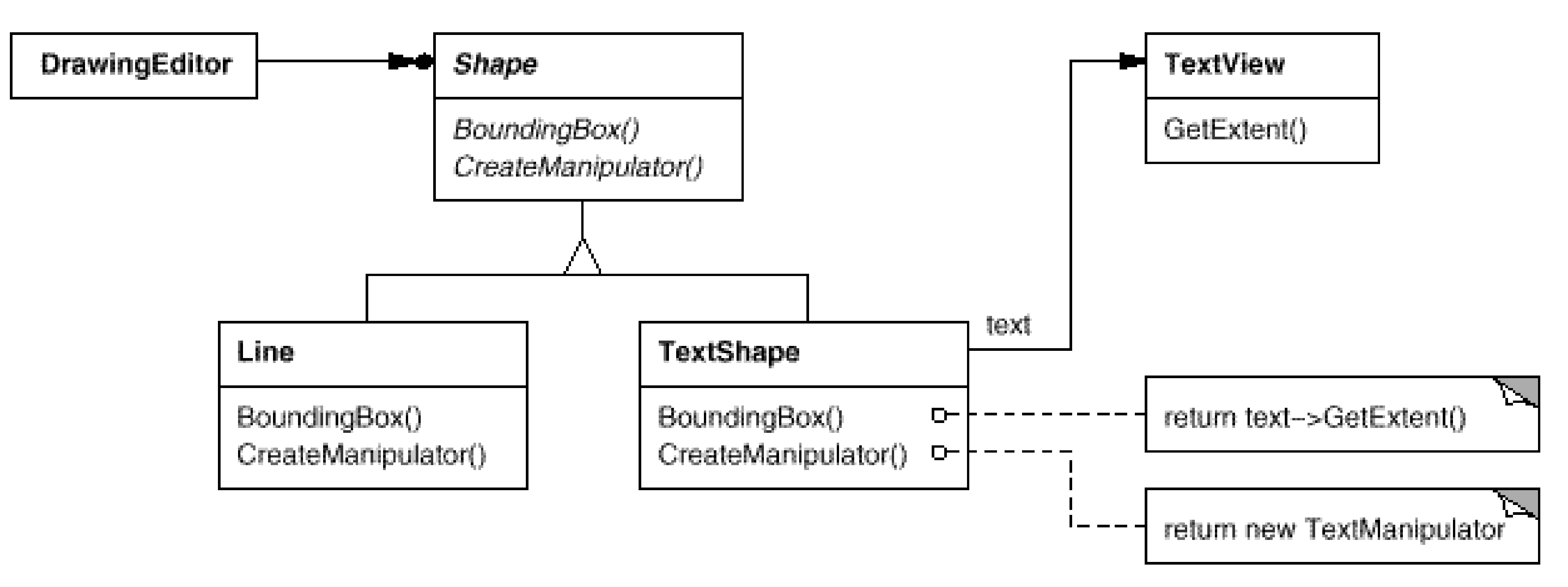
```
class SS_Modal : public Subject
{
private:
int value;
public:
int getValue() { return value; }
void setValue(int val) { value = val; }

};
```

```
class SS_View : public Observer
{
private:
SS_Modal *pSubject;
public:
SS_View(SS_Modal *p) { pSubject = p;  }
void Update()
{
int val = pSubject->getValue();

}
```

Adapter

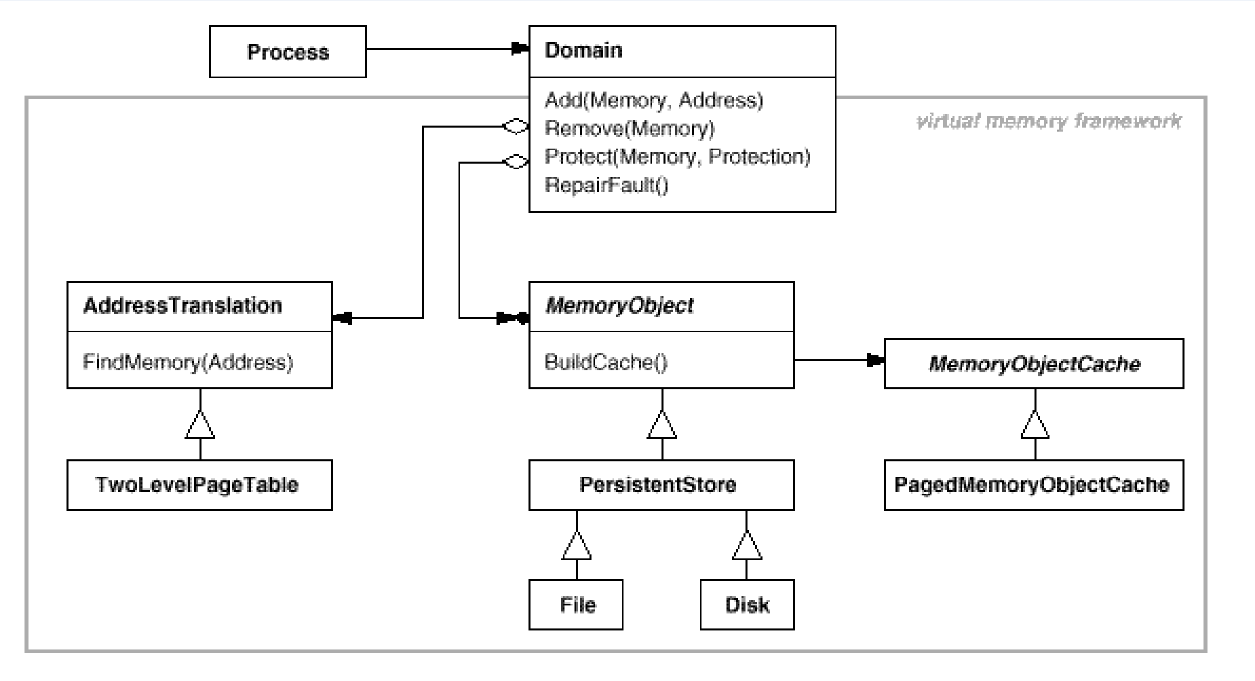
- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- If we have a case where TextView component which is not compatible with others but we need a similar capability
- We can do this in one of two ways:
- -by inheriting Shape's interface and TextView's implementation or
- by composing a TextView instance within a TextShape and implementing TextShape in terms of TextView's interface.
- TextShape is the adapter

Exercise- Adapter

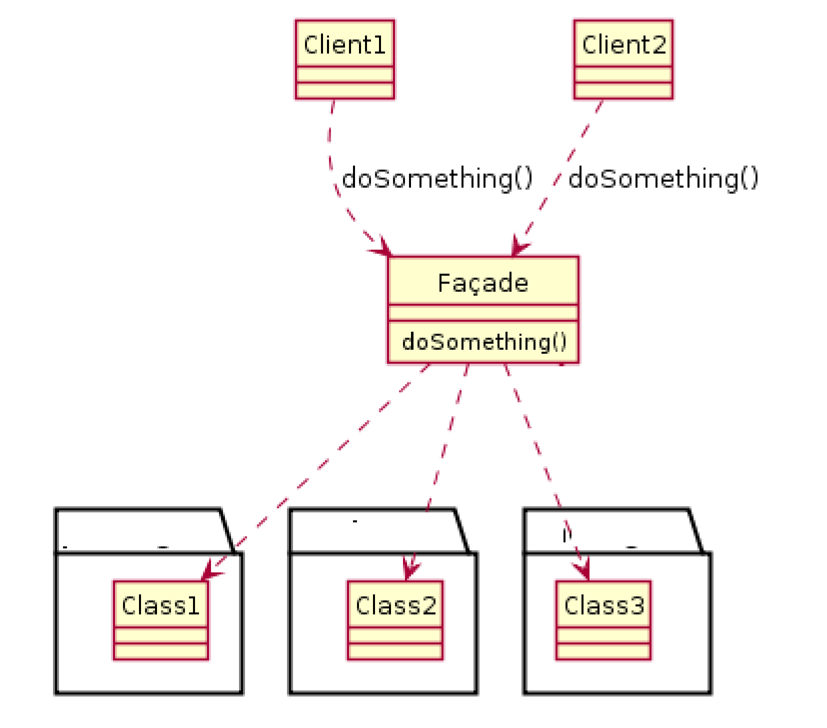


Facade

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



Facade



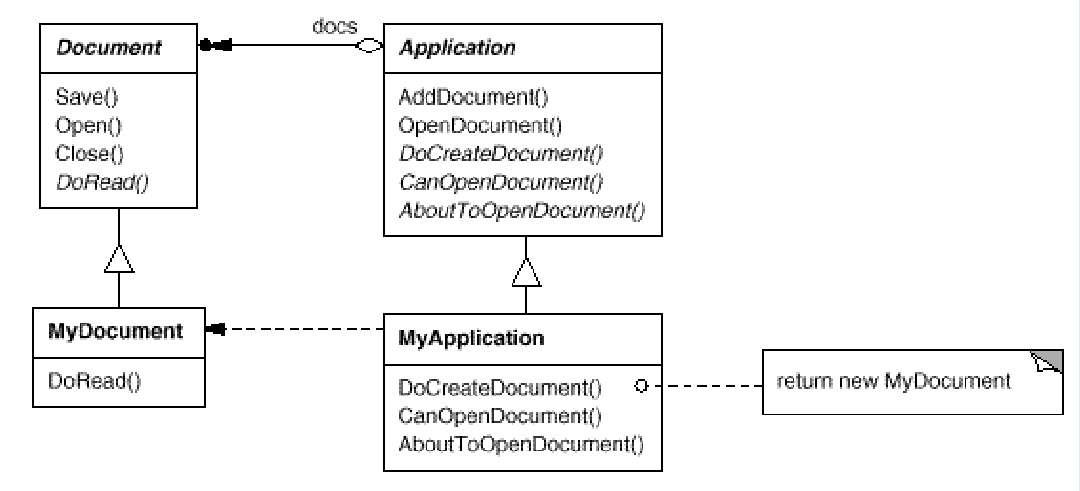
```
InitializeConfig1()
{
 cclass *pc1 = new Class1();
 class *pc2 = new Class2();
 class *pc3 = new Casse();
 pc1->doSuff(pc2);
 Pc2->doSuff(pc2);
return pc3->getY();
}
```

A Facade is used when an easier or simpler interface to an underlying object is desired

Image source adapted from: wikipedia/facade

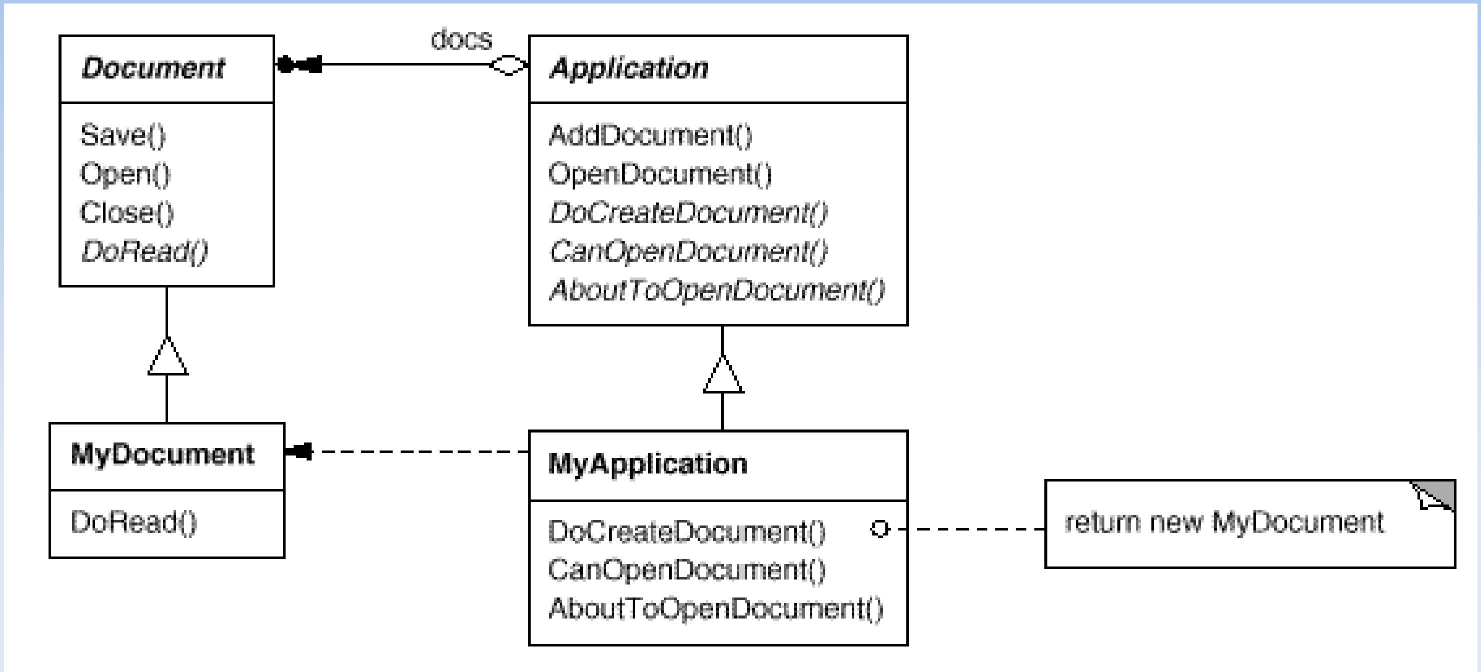
Template Method

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



In this case, Application class is abstract & expects subclass to do CreateDocument etc..
But OpenDocument is not virtual, it is a Template method that defines an algorithm using the abstract methods implemented by the subclass.

Template Method

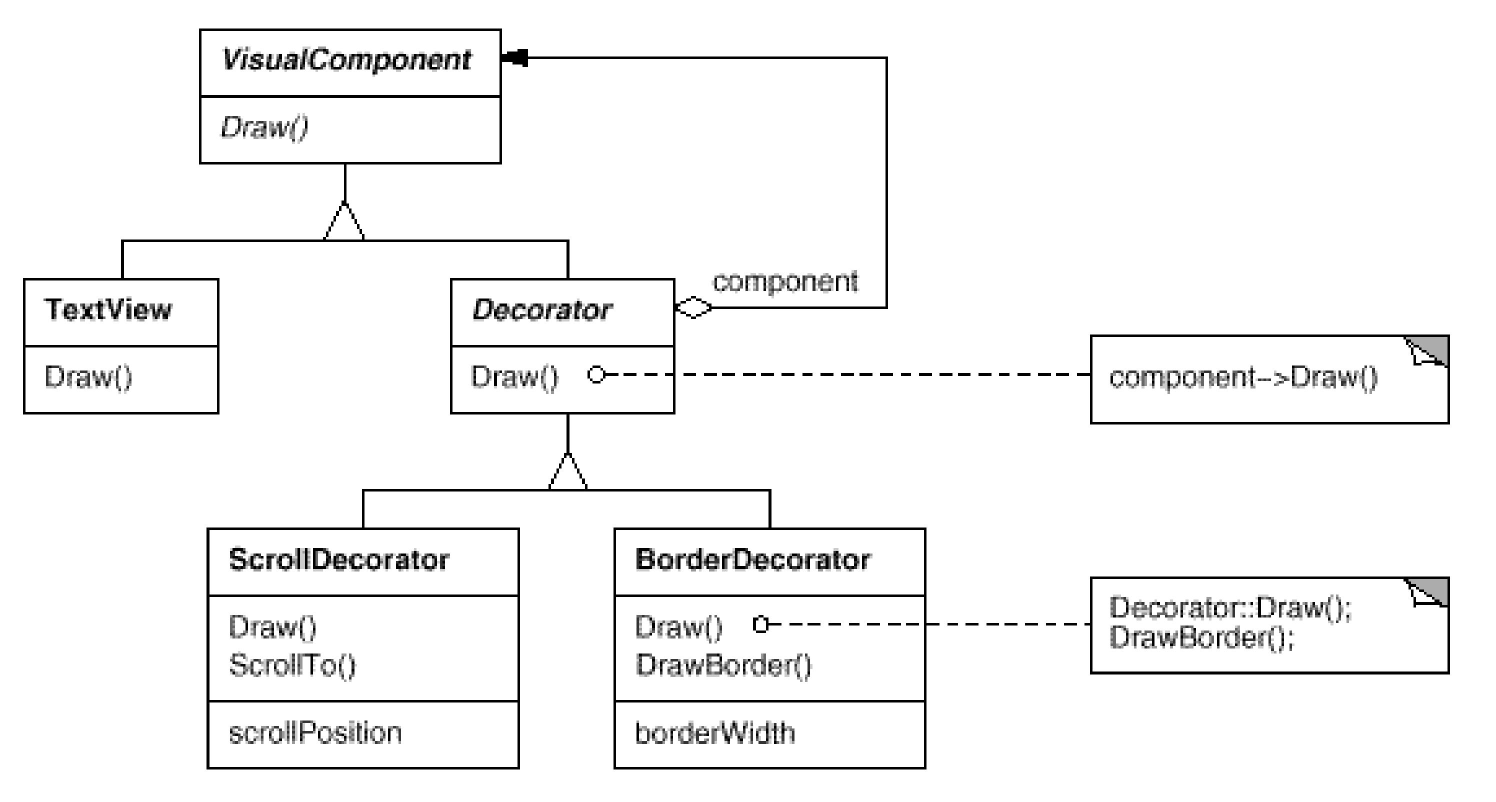


```
void Application::OpenDocument(const char* name)
{
 if (!CanOpenDocument(name)) {
 // cannot handle this document
 return;
 }
 Document* doc = DoCreateDocument();
 if (doc) {
 _docs->AddDocument(doc);
 AboutToOpenDocument(doc);
 doc->Open();
 doc->DoRead();
 }
}
```

Can be used
-to implement the invariant parts of an algorithm once and leave it upto subclasses to implement the behavior that can vary.
-avoid duplicationof code in subclass for common behaviour “refactoring to generalize”
- can have “hooks” to control subclass extensions at specific points

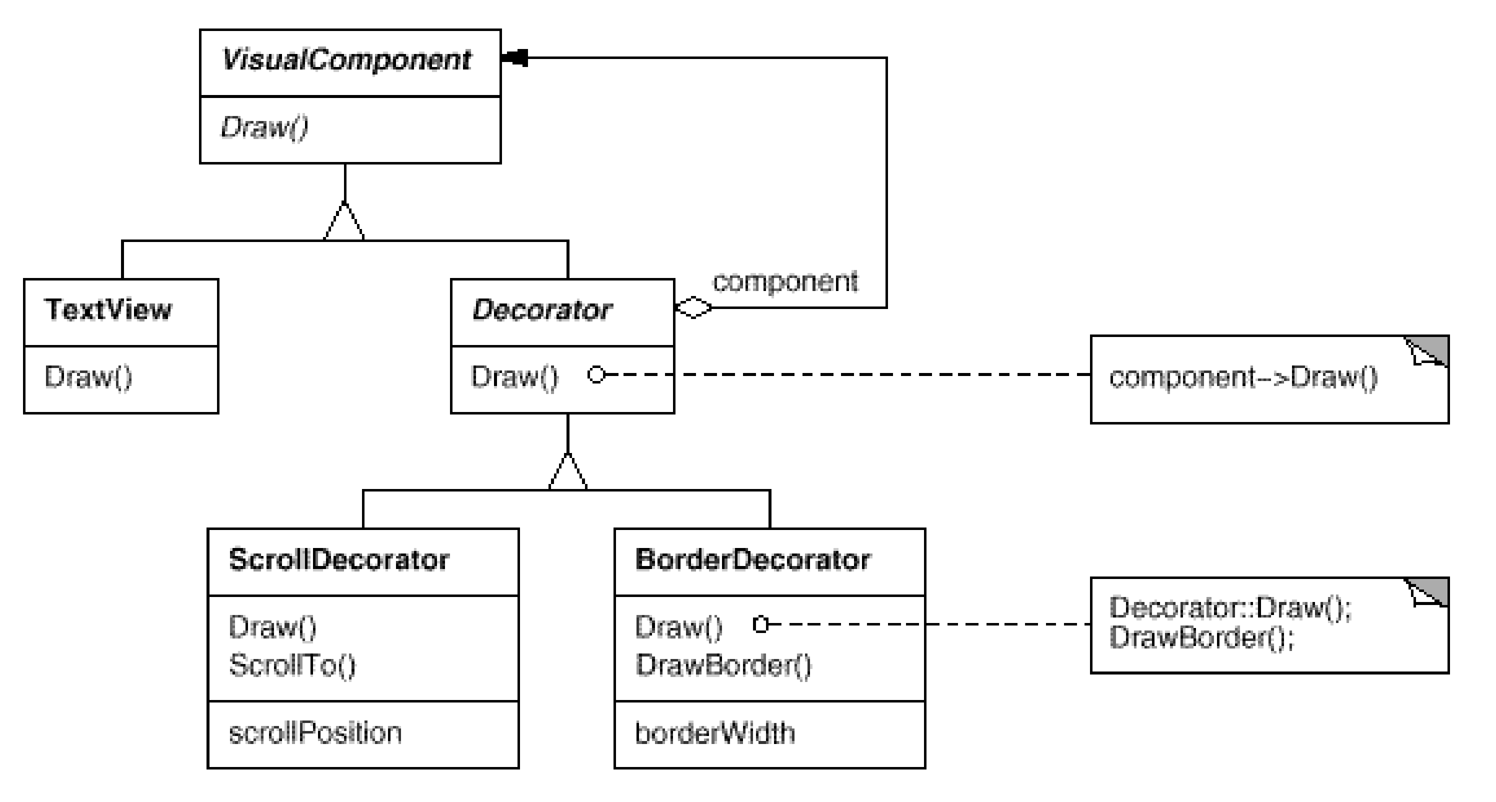
Decorator

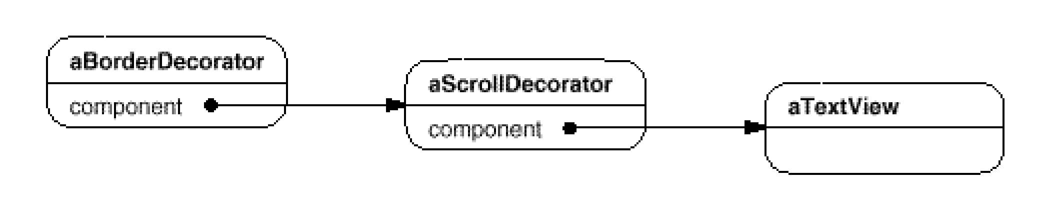
- Attach additional responsibilities to an object dynamically



Decorator - Exercise

- Add a decorator to text view such that it has border and scroll decorator





Decorator: Solution

```cpp
#include "stdafx.h"
#include <iostream>
using namespace std;

class VisualComponent
{
public:

        virtual void Draw() = 0;
};
class TextView : public VisualComponent
{
public:

        void Draw()
        {
                cout << "Draw of TextView"
<< endl;
        }
};
```

```cpp
class Decorator : public VisualComponent
{
        VisualComponent *component;
public:
        Decorator(VisualComponent *comp)
        {
                component = comp;
        }
        void Draw()
        {
                component->Draw();
        }
};

class BorderDecorator : public Decorator
{
public:

        BorderDecorator(VisualComponent
*comp) : Decorator(comp)
        {
        }
        void DrawBorder()
        {
                cout << "Draw of Border"
<< endl;
        }

};
```

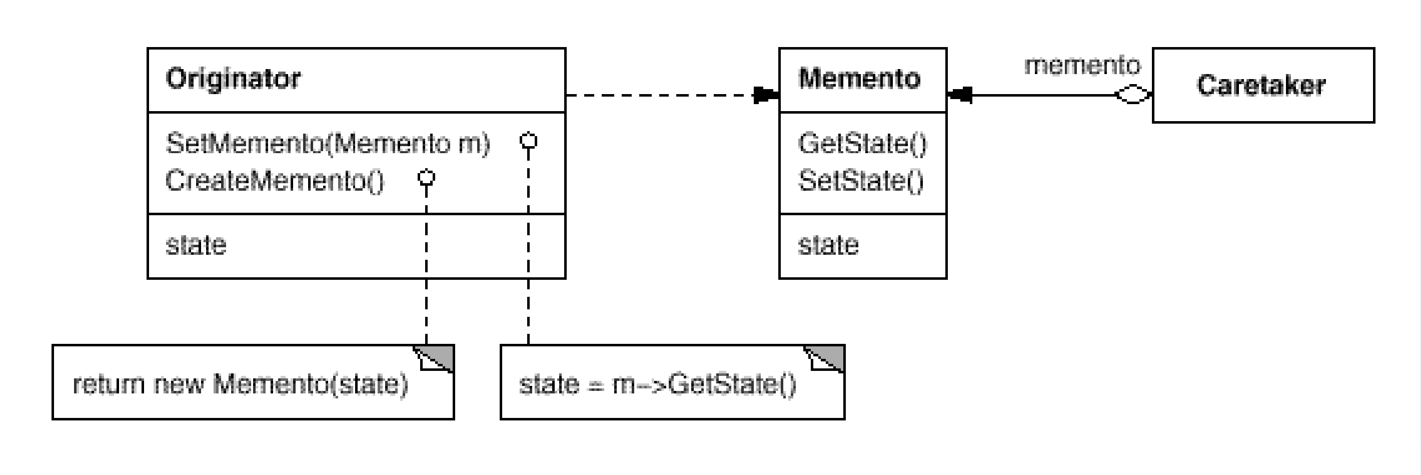
Decorator: Solution

```
        void Draw()
        {
                Decorator::Draw();
                DrawBorder();
        }

};
class ScrollDecorator : public Decorator
{
public:
        ScrollDecorator(VisualComponent *comp) : Decorator(comp)
                {
                }
        void DrawScrollBar()
        {
                cout << "Draw of Scroll" << endl;
        }
        void Draw()
        {
                Decorator::Draw();
                DrawScrollBar();
        }
};
```

```
int main(int argc, char* argv[])
{
        BorderDecorator *pBD = new BorderDecorator(new ScrollDecorator(new TextView));

        pBD->Draw();
        return 0;
}
```

Memento

- Intent
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- Sometimes is is necessary to save the internal states, for checkpointing, roll-back operations; but this requires the system to break the encapsulation (expose internal workings) – thereby compromising reliability and extensibility
- Memento is an object that saves snapshot of the internal state of another object – the memento;s originator



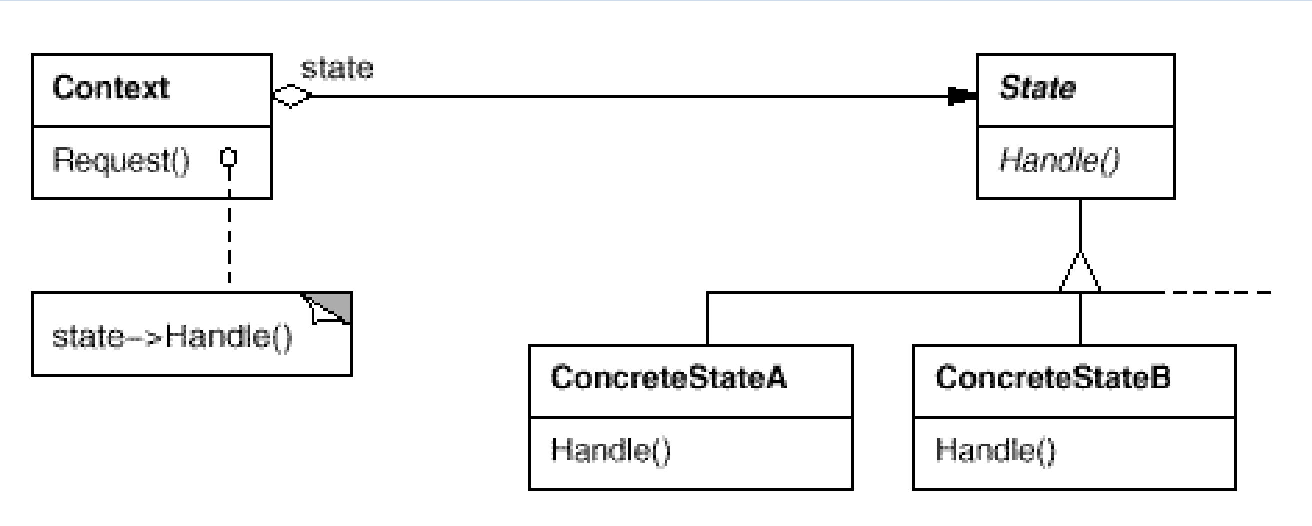
Memento

```
class State;
class Originator
{
public:
        Memento* CreateMemento();
        void SetMemento(const Memento*);
        // ...
private:
        State* _state;
        // internal data structures
        // ...
};
class Memento {
public:
        // narrow public interface
        virtual ~Memento();
private:
        // private members accessible only to Origina-
tor
        friend class Originator;
        Memento();
        void SetState(State*);
        State* GetState();
private:
        State* _state;
        // ...
};
```

State

- Implements the state machine

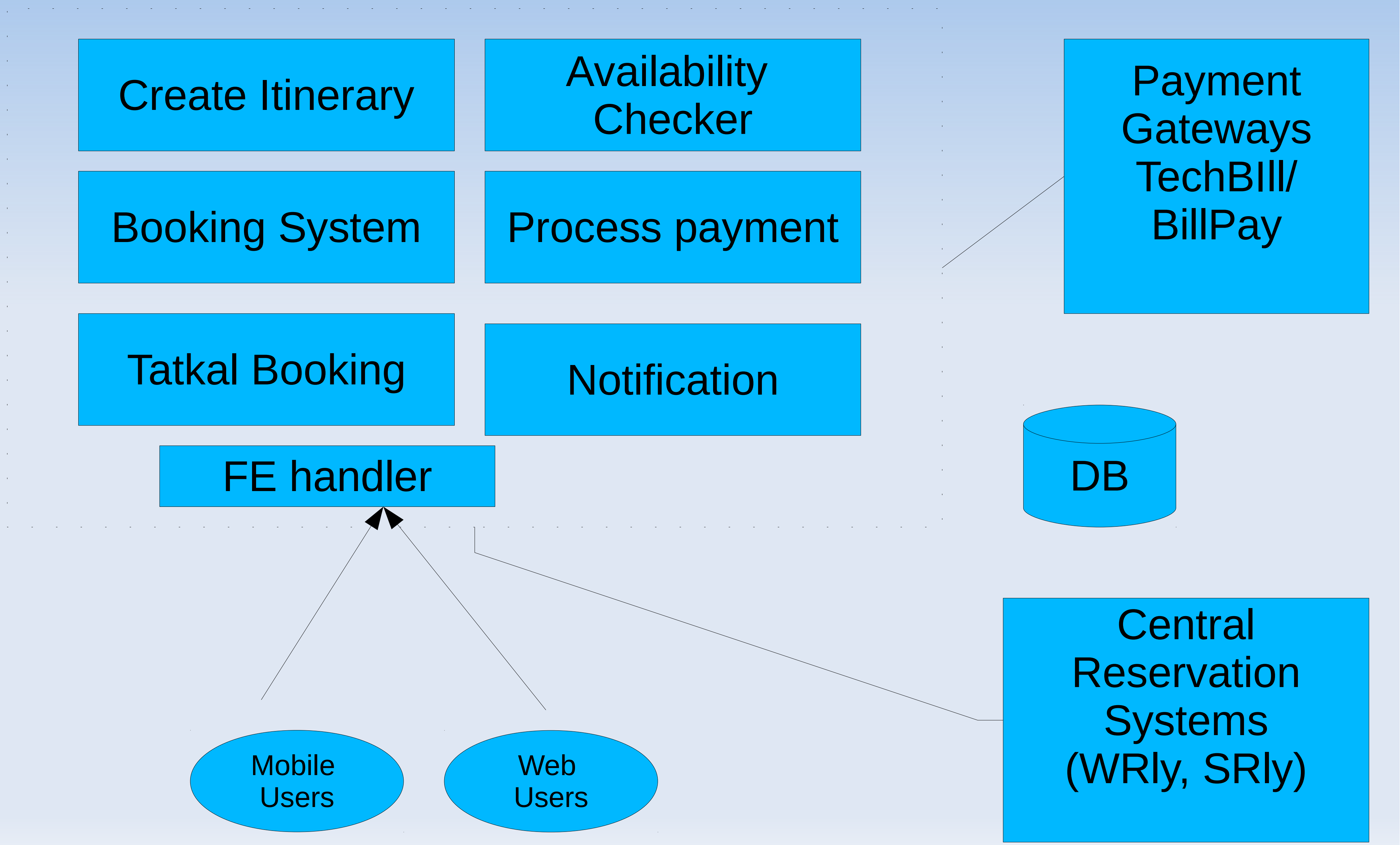
```
class State
{
public:
        virtual int Handle(int
e) = 0;
        virtual void PrintMe() =
0;

};
```



```
class StateA : public State
{
public:
 void PrintMe() { cout << "StateA" << endl; }
int Handle(int e)
 {
   if (e == 1)
   {
      Context::ChangeState(&Context::sb);
   }
   else ...

};
```

```
class Context
{
public:
static void ChangeState(State *to_state)
{
_currentstate = to_state;
_currentstate->PrintMe();
}
...
```

Case Study
MyRail Reservation System



General Flow

- User logs in
- Checks the availability for a route (with train)
- Selects the route, selects normal or tatkal
- Provides the passenger details
- Confirms the travel itinerary
- Makes Payment
- Gets Confirmation immediately
- Gets PNR/eticket asynchronously later