GoF Design Patterns (cont'd)

The Facade Pattern (Structural)

Problems:

Case 1:

Our software system has to get services from an existing, complex system.

We need either to use just a subset of the system or use the system in a particular way.

In other words, we have a complicated system in which we need to use only a part.

We want to avoid dealing with the internal structure of this complex system.

Case 2:

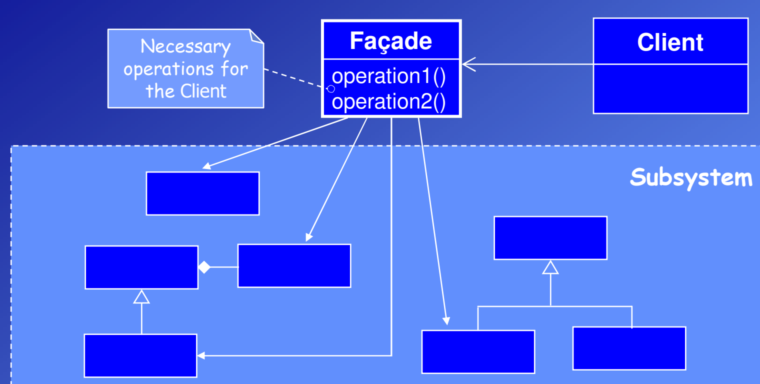
Our software system has to get services from a subsystem that has not been implemented yet.

We do not know the internal structure of this subsystem, which may also change.

Solution:

We create a new class (or classes) called **Facade** with the simple interface we require to get the (only) necessary services from the complex external system.

The Class diagram of the Facade Pattern:



Most of the work is done by the underlying subsystem.

The Facade provides a collection of easier-to-understand methods.

These methods use the underlying system to implement the newly defined functions.

The Facade also reduces the number of objects that a client object must deal with.

When to apply the Facade Pattern [1]:

- You do not need to use all of the functionality of a complex system and can create a new class that contains all of the rules for accessing that system.

  Usually, the API (Application Program Interface) you create in a new class should be much simpler than the original system's API.
- You want to encapsulate or hide the original system.
- You want to use the original system's functionality and add some new functionality as well.
- The cost of writing this new class is less than that of everybody learning how to use the original system or less than you would spend on maintenance in the future.

[1] Alan Shalloway, James R. Trott , *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley, 2002.

The Facade vs. The Adapter:

In both cases, there is a preexisting class (or a system with classes) that has the needed functionality.

We create a new object (or class) with the desired interface in both cases.

They are both wrappers, but there are also following differences between them:

- In the Facade, we do not have an interface to which we must design our system; we have a complex system.

  In the Adapter pattern, we need to convert an existing interface to make it compatible with the client object.
- In the Facade, we do not need polymorphism.

  In the Adapter pattern, we often need to convert interfaces of many existing classes to provide a stable interface (external tax calculators).
  In this case, we need the polymorphism (slide 8.16).

  Polymorphism may not be necessary when we design our system to a particular API (XXCircle in 8.12).
- In the case of the Facade pattern, the motivation is to simplify the interface. With the Adapter, we are trying to design a module to an existing interface.

A Facade simplifies an interface, while an Adapter converts the interface into a preexisting interface.

Example: *Pluggable business rules* in the NextGen POS system.

Problem:

Different companies (stores) wishing to purchase the NextGen POS would like to customize its behavior at some predictable points in the scenarios.

These rules may invalidate some actions.

```
if (rule_1) then …
…
if (rule_2) then …
…
```

These rules (conditions) may change.

For example:

- When a new sale is created, it is possible to identify that it will be paid using a gift certificate.

  A store may have a rule to allow only one item to be purchased if a gift certificate is used.

  In this case, subsequent `enterItem` operations after the first should be invalidated.
- If a gift certificate is used to pay the sale, invalidate all payment types of change due back to the customer except for another gift certificate.
- These rules can vary for different stores.

Solution:

The software architect (as always) wants a design that has a low impact on the existing software components (*open-closed principle*).

According to the *separation of concerns* principle, the software architect puts the rule handling into another subsystem (rule engine).

Furthermore, suppose that the architect is unsure of the best implementation for this pluggable rule handling and may want to experiment with different solutions.

To solve this design problem, **the Facade** pattern can be used.

We will define a "**rule engine**" subsystem whose specific implementation is not yet known. It will be responsible for evaluating a set of rules against an operation.

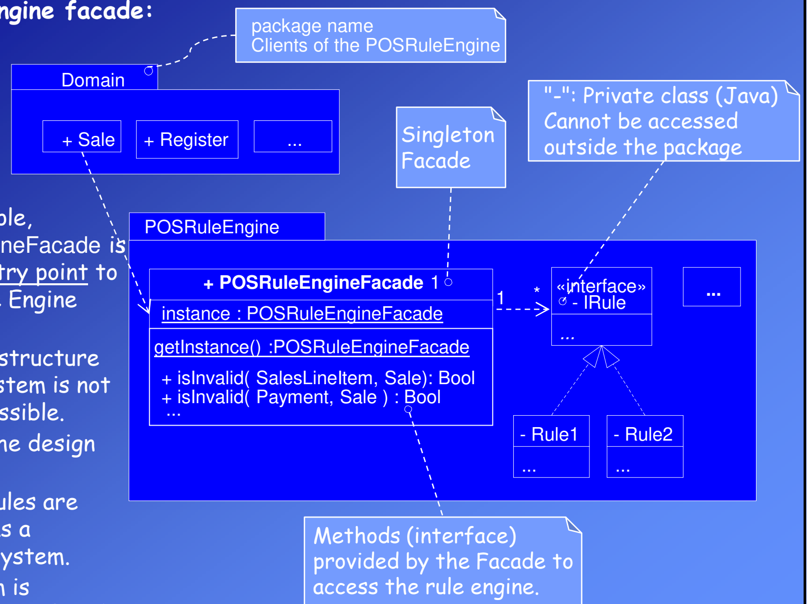
We will create a facade as a "front-end" object that is the single point of entry for the services of a subsystem.

The implementation and other components of the subsystem (rule engine) are private and can't be seen by external components.

Facade provides Protected Variations from changes in the implementation of a subsystem.

The facade object to this subsystem will be called POSRuleEngineFacade.

The Rule engine facade:



In this example, POSRuleEngineFacade is the single entry point to the POS Rule Engine subsystem.

The internal structure of the subsystem is not directly accessible.

To improve the design quality,

1. Variable rules are designed as a separate system.
2. Subsystem is accessed via a facade.

Accessing the Rule Engine using the POSRuleEngineFacade

When the customer of the POS system tries to buy an item, the Sale object asks the rule engine if this item is OK for itself.

The Sale object accesses the rule engine using the facade indirectly.

Example in Java:

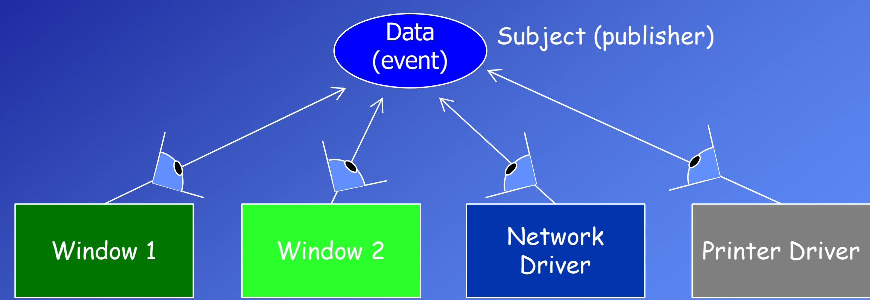
```
public class Sale
{
  public void makeLineItem (ProductSpecification spec, int quantity)
  {
      SalesLineItem sli = new SalesLineItem (spec, quantity);
      // call to the Facade
      if ( POSRuleEngineFacade.getInstance().isInvalid(sli, this) )
         return;             // not accepted
      lineItems.add( sli );    // OK, accepted
  }
// ...
}  // end of class
```

Calling the methods of singleton Facade

The Observer Pattern (Behavioral)

Sometimes objects are interested in changes in another object's states (attributes).

The number of interested objects (observers) can change in run-time.



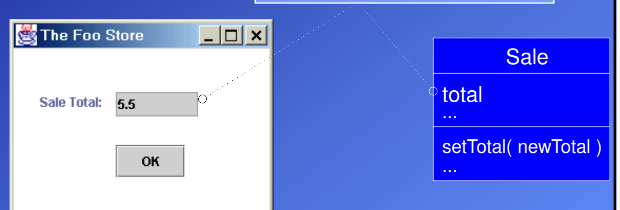
Observers (subscribers, listeners)

The object called the **subject** maintains a list of its dependents, called **observers** and notifies them automatically of any state changes, usually by calling one of their methods.

Example:

In the POS system, when the total of the sale changes, a GUI window has to refresh its display of the sale total.

When the total of the sale changes, refresh the display with the new value.



A possible solution (**inappropriate**):

When the Sale object changes its total, it sends a message to a window, asking it to refresh its display.

Problems in this design:

- The publishing object (Sale) is connected to (must be aware of) the subscriber (observer) object (window).
- It does not fit the Model-View Separation principle. The model object (Sale) is connected to a view (window) object.
  We do not want to impact the Sale object if a new one replaces the presentation layer (window).

Definition : The Observer pattern

Problem:

Different kinds of the observer (subscriber) objects are interested in the state changes or events of a subject (publisher) object and want to react in their own unique way when the publisher generates an event.

The publisher wants to maintain low coupling with the subscribers.

The number of subscribers may change in run-time.

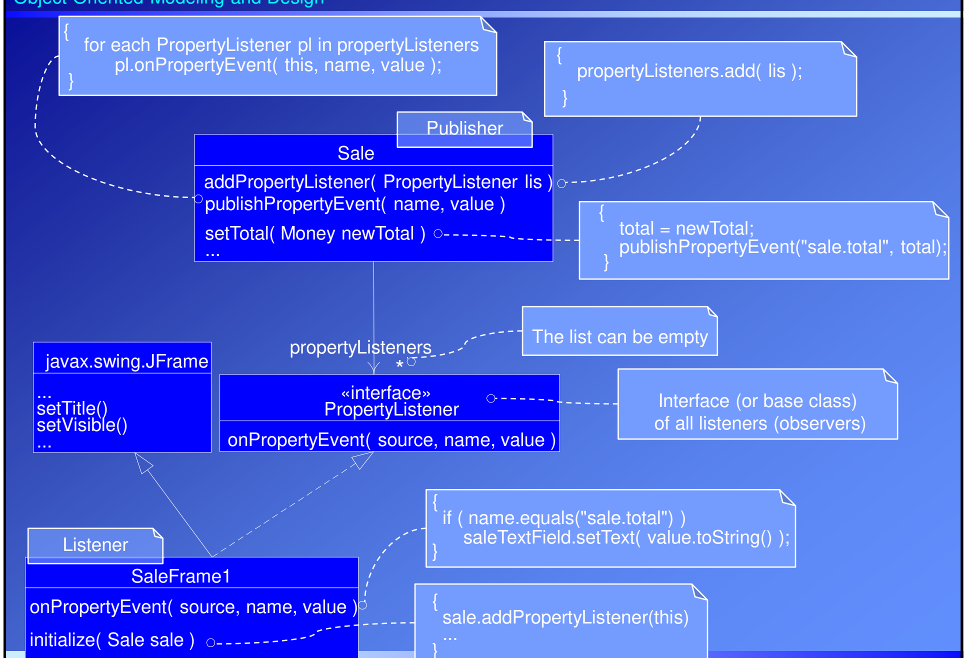
Solution (advice):

Derive all observers from a common base class (interface) "observer" or "listener".

The subject (publisher) has a dynamic list of observers (or listeners).

The subject can register observers (add to the list) interested in an event and notify them when it occurs.

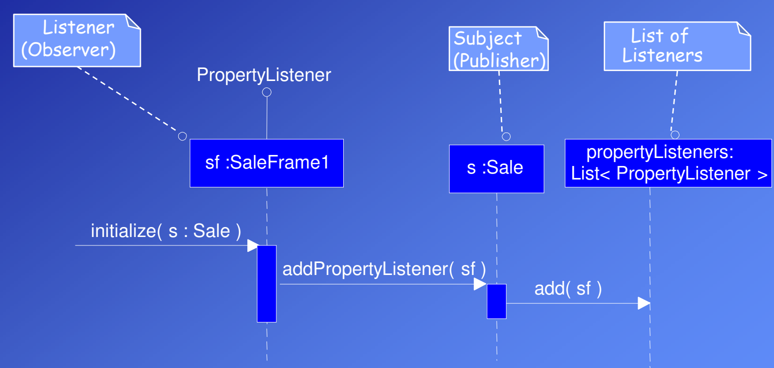
If needed, observers can also be deleted from the list.



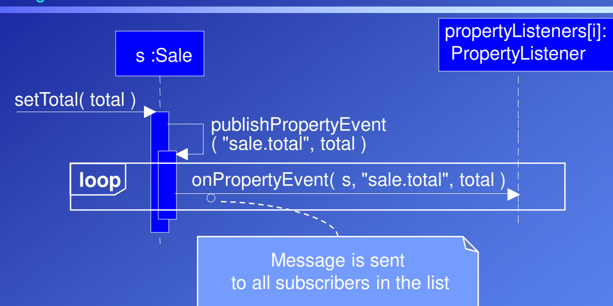
Registering the observer:

When the observer (listener, subscriber) SaleFrame1 is interested in property events of the Sale (publisher), it sends a subscription request to the publisher.

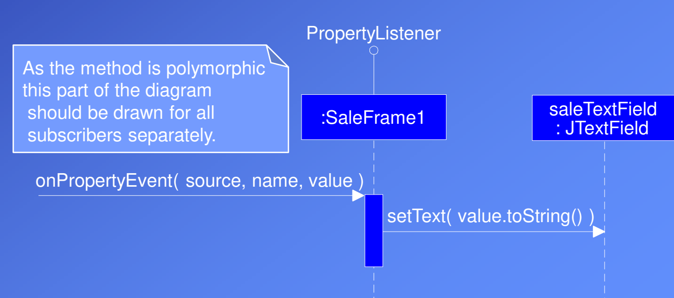
The Sale adds the SaleFrame object to its listeners (observers, subscribers) list.



Publishing the event:

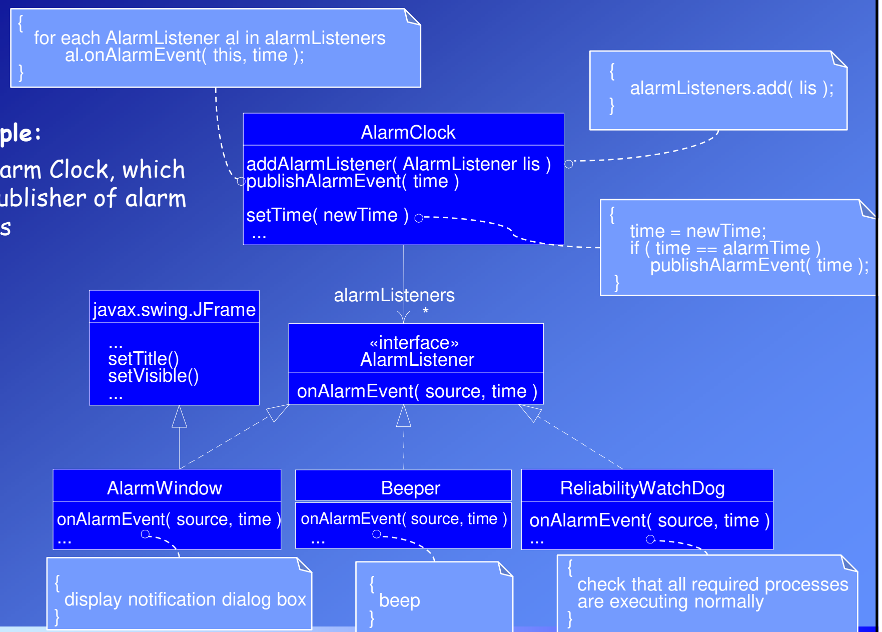


Receiving the event:



Example:

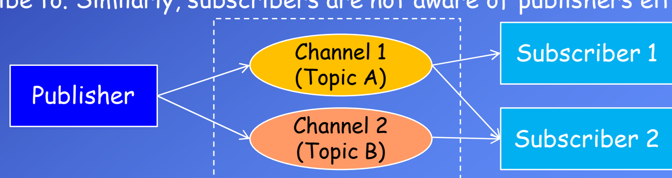
An Alarm Clock, which is a publisher of alarm events



The Publish-Subscribe (Pub/Sub) Design Pattern (Non GoF)

The Publish-Subscribe (Pub/Sub) design pattern differs from the Observer pattern and is not one of the GoF patterns.

- In the pub/sub pattern, publishers do not send messages directly to subscribers; instead, messages are sent via brokers.
- Publishers categorize messages into classes (topics).
- The Publisher sends messages (events) to related channels in the broker.
- Subscribers sign up for the topics they are interested in.
- The Publisher sends the message (event) to the message broker, and the message broker broadcasts the message to the right subscriber(s).
- Publishers do not know who the subscribers are or which (if any) topics they subscribe to. Similarly, subscribers are not aware of publishers either.



Communication Infrastructure

Source: MS Learn: https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber

The Decorator Pattern (Structural)

Remember: if a class's behavior changes according to some conditions in run time, we define different behaviors as different **strategies** separately from the context class (see the Strategy Pattern).

Sometimes it is not required to change the whole behavior of a class, but it is needed to add some new functionalities dynamically to the main behavior.

For example, the behavior (responsibility) B() of an object sometimes may consist of sub-behaviors (functions) b1,b2,b3,r, sometimes b3,r,b1; and sometimes r,b4,b1,b3, where r is the main responsibility in B().

The Decorator pattern intends to attach additional responsibilities to an object dynamically.

Definition of the pattern:

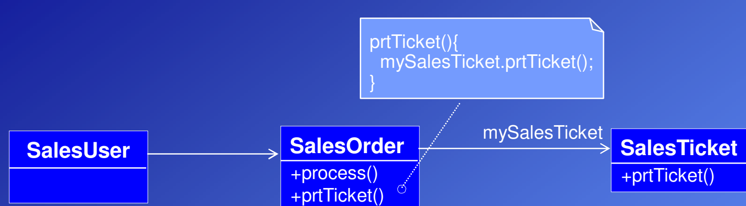
Problem:

The object that you want to use does the basic functions you require. However, you may need to add some additional functionality to the object, occurring before or after the object's base functionality.

Solution:

Create an abstract class that represents both the original class and the new functions to be added to the class.
In the decorators, place the new function calls before or after the trailing calls to get the correct order.

The Case Study : Printing tickets in e-sale system[1].



SalesUser is the client class of the **SalesOrder** class.

SalesOrder calls the **SalesTicket** object, requesting that it prints the ticket.

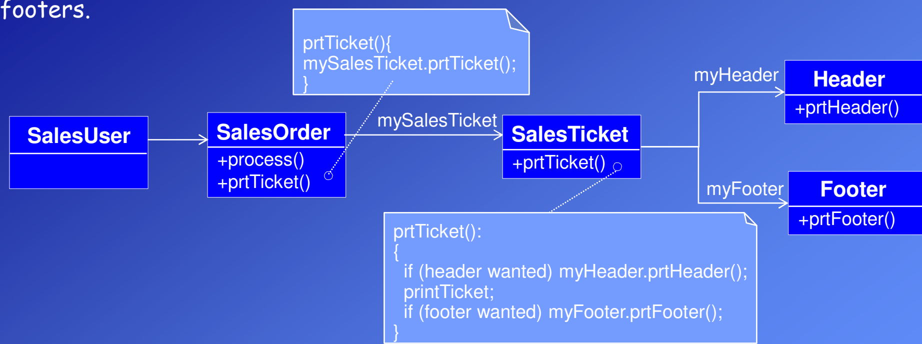
This modular design works.

Suppose that, during the design of the application, we get a new requirement to add header and footer information to the SalesTicket.

[1]Alan Shalloway, James R. Trott , *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley, 2002.

The Case Study (contd): Printing headers and footers in a ticket

If we are writing the system only just for one company, it may be easiest to add "if" statements in the SalesTicket class to control the printing of headers and footers.



This design works quite well if we do not have to deal with many options or if these headers/footers do not change.

What happens **if we have to deal with many different types of headers and footers?** For example, picture-header, text-header, QRCode-footer, etc. Different for concerts, for football matches, different for cinema, etc.

The Case Study (contd): Printing different headers and footers in a ticket

Can we apply here the **Strategy** pattern?

If we had to deal with many different types of headers and footers, printing **only one each time**, we might consider using one Strategy pattern for the header and another for the footer.

What happens if we have to print more than one header and/or footer at a time?

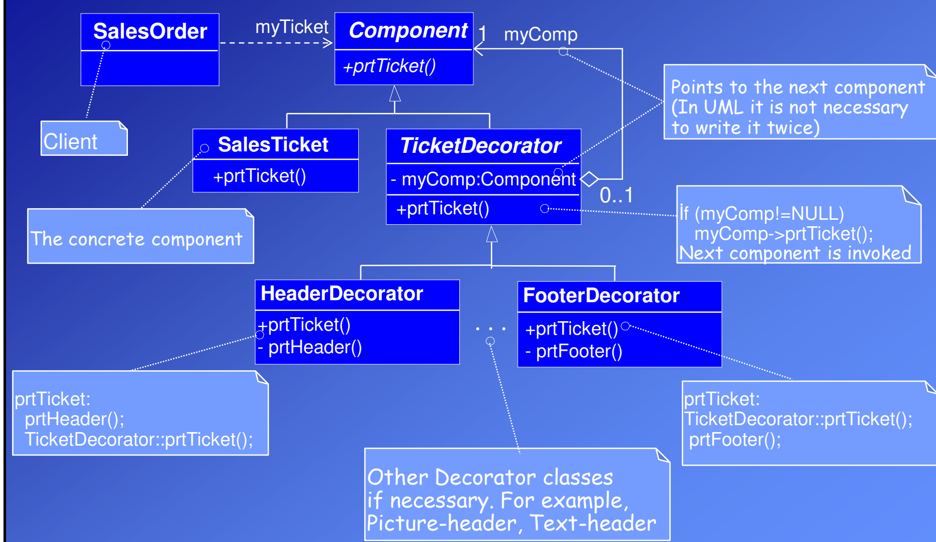
Or what if the order of the headers and/or footers must change?

We can solve this problem by using the Decorator pattern.

Solution with the Decorator Pattern:

- We will design all functionalities (headers, footers) as separate **Decorator** classes.
- The main (base) function will be designed as a **concrete component**.
- Concrete component and decorator classes are derived from the same base class named **Component** class.
- A list (chain) of decorator objects and a concrete component will be created in the desired order.
- The client object will call the first object in the chain. Then each object will invoke the next object in the list.

Solution with the Decorator Pattern:



Decorating the method (Creating the decorators)

In the desired order, a chain of objects (decorators and a concrete component) will be created in runtime.

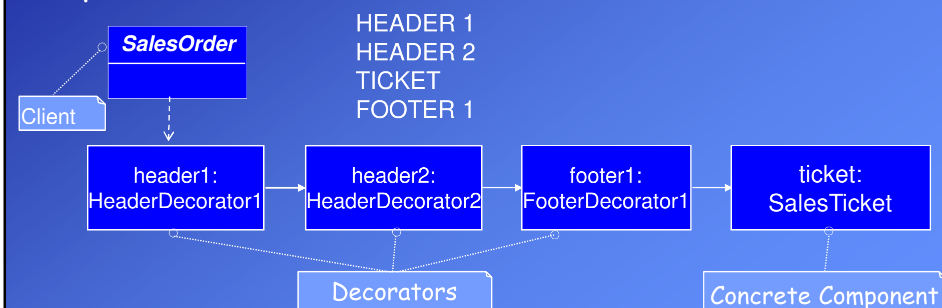
Each chain starts with a Component (a Concrete Component or a Decorator).

Each Decorator is followed either by another Decorator or by the original ConcreteComponent.

A Concrete Component always ends the chain.

The client object will get the address of the first object in the chain.

Example: If two headers and a footer are needed.



An Exemplary Program:

This program assumes that Header1 and Header2 (similarly Footer1 and Footer2) classes have different functionalities.

Therefore, they are implemented as separate classes.

If only their printing messages were different, we would implement only one Header class and one Footer class, with a text attribute, which can contain different messages.

Implementation in C++:

```
class Component {                    // Abstract component
  public:
      virtual void prtTicket()=0;
};

class SalesTicket : public Component{        // Concrete component
  public:
    void prtTicket(){                // Base function
        cout << "TICKET" << endl;
    }
};
```


Exemplary Program (contd): Headers and footers

```
class Decorator : public Component {        // Base of decorators
public:
  Decorator( Component *myC) : myComp {myC} {      //  Constructor
  }                      // Takes the address of the next component
  void prtTicket(){                        // Calls the next component
   if (myComp) myComp->prtTicket();
  }
private:
  Component *myComp;             // Pointer to the next component
};

class Header1 : public Decorator {       // Header1 decorator
public:
  Header1(Component *);
  void prtTicket();
};

Header1::Header1(Component *myC) : Decorator{myC} {}  // Constructor of Header1

void Header1::prtTicket(){
   cout << "HEADER 1" << endl;            // Header1's specific function
   Decorator::prtTicket();               // Calls the method of the base class.
}
```

```
class Header2 : public Decorator {                // Header2 decorator
  public:
     Header2(Component *);
     void prtTicket();
};
Header2::Header2(Component *myC):Decorator{myC}{}     // Constructor of Header2
void Header2::prtTicket(){
        cout << "HEADER 2" << endl;               // Header2's specific function
        Decorator::prtTicket();                   // Calls the method of the base
}

class Footer1 : public Decorator {                // Footer1 decorator
public:
  Footer1(Component *);
  void prtTicket();
};
Footer1::Footer1(Component *myC):Decorator{myC}{}     // Constructor of Footer1
void Footer1::prtTicket(){
        Decorator::prtTicket();            // Calls the method of the base class.
        cout << "FOOTER 1" << endl;               // Footer1's specific function

}
```

Class Footer2 is also written in a similar way.

Object-Oriented Modeling and Design

Exemplary Program (contd): Headers and footers

```
class SalesOrder {            // A client class to test the system
    Component *myTicket;            // Pointer to printer component
public:
  SalesOrder(Component *mT) : myTicket{mT} {}
  void prtTicket(){
                  //call the method of the first object in the chain
    myTicket->prtTicket();
  }
};

int main()          // The main function for testing
{
  SalesOrder sale{new Header1{new Header2{new Footer1
                                          {new SalesTicket{}}}}};
  sale.prtTicket();
  return 0;
}
```

In a real system this address can be received from a Factory object.

A list of components (decorators) is created.
In a real system, this chain can be created by a Factory

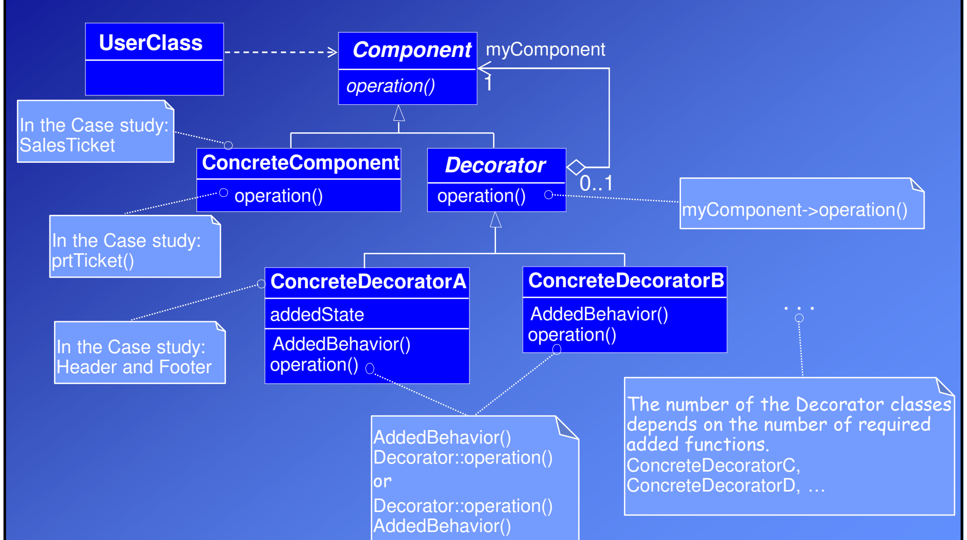
See Example Ticket_Decorator.cpp

Discussion and Summary:

- The Decorator pattern is a way to dynamically add additional function(s) in a desired order to an existing behavior (function).
- The Decorator pattern says, "to control the added functionality chain together the functions desired in the correct order needed".
- The instantiation of the chains of objects is completely decoupled from the Client objects that use it.

  This is most typically accomplished through the use of factory objects that create the chains based on some configuration information.
- Decorators provide a flexible alternative to subclassing (inheritance) for extending functionality (Favor composition over inheritance).

The UML Class Diagram of the Decorator Pattern:



The Template Method Pattern (Behavioral)

An exemplary problem:

A school has various types of students, e.g., undergraduate and master's.

Sometimes, a report that presents the status of a student must be generated.

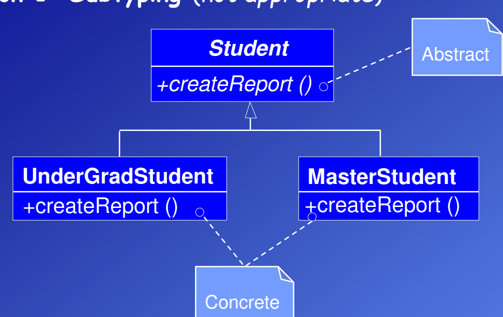
The creation of a report consists of some fixed steps.

However, the details of the steps may vary depending on the type of the student.

createReport():

1. Read the courses of the student.
   It is common (same algorithm) for all student types.
2. Calculate the average.
   It depends on the type of the student, i.e., different for different types.
   The average of the undergraduate and master's students is calculated differently (different algorithms).
3. Print report.
   It depends on the type of the student.
   Different information is printed for undergraduate and master students.

Solution 1: Subtyping (*not appropriate*)



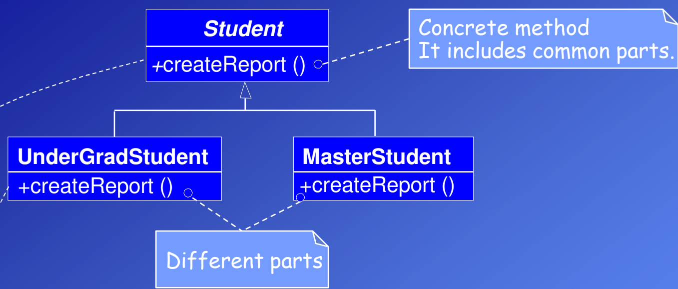
We write the createReport() method for each student type again.

If we want to add PhD students to this system, we must rewrite the createReport method.

Problems:

- The author of the subtype needs to know (remember) and repeat the steps of the algorithm (how to create a report).
- There may exist code duplications (reading courses of the student).

Solution 2: Subtyping and avoiding code duplications (*there are still problems*)



Student::createReport():
1. Read the courses of the student. *// Common (Same) for all students*

This function may include many statements

UnderGradStudent::createReport():
1. Student::createReport(); *// Calls the function of the base class (common part)*
2. Calculate the average. *// Specific to undergraduate students*
3. Print report. *// Specific to undergraduate students*

Solution 2: (cont'd)

MasterStudent::createReport():
1. Student::createReport(); *// Calls the function of the base class (common part)*
2. Calculate the average. *// Specific to master students*
3. Print report. *// Specific to master students*

Problems:

- The author of the subtype still needs to know (remember) and repeat the steps of the algorithm (how to create a report).
- The base class method is redefined (overridden), and it must be called in the derived class.

  However, the author of the subclass may forget to call it. The programming languages do not enforce the calls to redefined methods.

  If a subclass must call the base class method that it has been overridden, the "**call super anti-pattern**" occurs.

  "Whenever you must remember to do something every time, that's a sign of a bad API. Instead, the API should remember the housekeeping call for you."

  Martin Fowler: http://martinfowler.com/bliki/CallSuper.html

Solution with the Template Method

The main problem with the previous solutions is that the subclasses must control the process (creating a report).

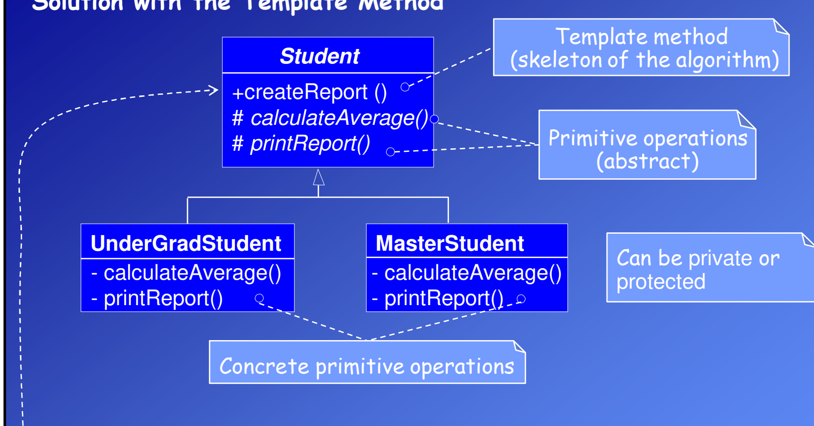
When a new type is added to the system, the programmer of the subclass must remember and repeat this process (how to create a report).

Template Method:

The control is inverted with the template method pattern, and the base class controls the overall process.

- The designer of the base class defines the skeleton (steps) of the algorithm in a template method.
- The designer decides which steps of the algorithm are invariant (common) and which are variant (different or customizable for different types).
- The invariant (common) steps are implemented in the abstract base class.
- For the variant steps, empty virtual methods (primitive operation) are written.
- The bodies of the primitive operations are implemented in subclasses.

Solution with the Template Method



Student::createReport(): // ***Template method*** *Skeleton (steps) of the algorithm*
Read thecourses of the student // *1. invariant code (common)*
calculateAverage(); // *2. calls primitive operation (implemented in a subclass)*
printReport(); // *3. calls primitive operation (implemented in a subclass)*

Solution with the Template Method

The authors of the subclasses need only to write the bodies of the primitive operations calculateAverage() and printReport().

They don't have control over the main algorithm (createReport()) and don't need to remember to call some base class methods.

The template method of the base class calls the methods (primitive operations) of the subclass.

This inverted control structure is called "**the Hollywood principle**" or "**don't call us, we'll call you**".

Suppose we need to add a new subtype (such as PhDStudent) to the system.

In that case, we only need to implement primitive operations (calculateAverage() and printReport()) that are specific to the new type.

Source code of the solution with the Template Method in C++

```
class Student{                            // Abstract base class
 public:
    void createReport ();                 // Template Method

 protected:
    virtual void calculateAverage() =0;    // Abstract primitive operation, pure virtual function
    virtual void printReport() =0;         // Abstract primitive operation, pure virtual function
};

void Student::createReport ()             // Template Method: Skeleton of the algorithm
{
   // Step 1,  common for all types
   cout << "Read Courses from a database (common for all students)" << endl;  // Step 1

   calculateAverage();                    // Step 2, specific to different types
   printReport();                         // Step 3, specific to different types
}
```

The primitive operations calculateAverage() and printReport() are abstract (virtual functions) in the base class.
They will be implemented in the subclasses according to the requirements of the subtypes.

```
//--------  Subtype: Undergraduate Student ----------
class UnderGradStudent : public Student{
 private:                       // It can also be protected
   void calculateAverage(){     // Concrete primitive function, specific to Undergraduate Students
       cout << "Average of the Undergraduate Student" << endl;
    }

    void printReport(){         // Concrete primitive function, specific to Undergraduate Students
        cout << "Report of the Undergraduate Student" << endl;
    }
};


//--------  Subtype: Master Student ----------
class MasterStudent : public Student{
 private:                          // It can also be protected
   void calculateAverage(){        // Concrete primitive function, specific to Master Students
       cout << "Average of the Master Student" << endl;
 }

   void printReport(){             // Concrete primitive function, specific to Master Students
       cout << "Report of the Master Student" << endl;
 }
};
```



```
// Testing the implentation
int main()
{
   UnderGradStudent  uStudent;
   uStudent.createReport();
   cout<< "------------" << endl;

   MasterStudent  mStudent;
   mStudent.createReport();

   return 0;
}
```

Output:

Read Courses from a database (common for all students)
Average of the Undergraduate Student
Report of the Undergraduate Student

Read Courses from a database (common for all students)
Average of the Master Student
Report of the Master Student

See Example
Student_Template.cpp

Hook operations

Note that primitive operations in the base class are abstract and <u>must</u> be implemented in the subclasses.

Bases classes can also include **hook operations**, which provide default behavior that subclasses can extend <u>if necessary</u>.

A hook operation often does nothing by default in the base class.

If necessary, the subclass's author can override the default hook operation.

Example:

```
Student::createReport():           // Skeleton (steps) of the algorithm
 // Read courses of the student  // 1. invariant code (common)
  calculateAverage();            // 2. calls primitive operation (implemented in a subclass)
  printReport();                 // 3. calls primitive operation (implemented in a subclass)
  printAdditionalInfo();          // 4. hook operation, default (prints nothing)
};

void Student::printAdditionalInfo() { } ;  // Default hook operation does (prints) nothing
```

Now only the subclasses that need to print additional information will override this method. **Other subclasses do not need to redefine this method.**

Summary: Template Method Design Pattern

Intent:

Define the skeleton of an algorithm, deferring some steps to subclasses. Subclasses can redefine specific steps of an algorithm without changing the algorithm's structure.

The base class declares the algorithm 'placeholders', and derived classes implement the placeholders.

Problem:

Two different components have significant similarities but demonstrate no reuse of a common interface or implementation.

If a change common to both components becomes necessary, duplicate effort must be expended.

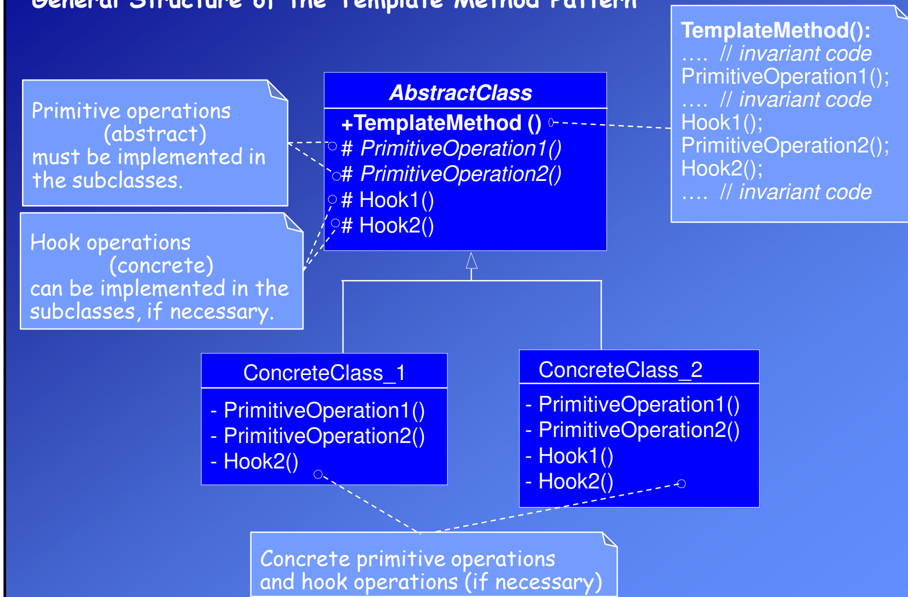
Discussion:

The component designer decides which steps of an algorithm are invariant (or standard) and which are variant (or customizable).

The invariant steps are implemented in an abstract base class, while the variant steps are either given a default or no implementation.

The variant steps (hook or primitive operations) can or must, be supplied by the component's client in a concrete derived class.

General Structure of the Template Method Pattern



The Bridge Pattern (Structural)

According to the Gang of Four:

"De-couple an **abstraction** from its **implementation** so that the two can vary independently."

Abstraction is how different things are related to each other conceptually.

For example, undergraduate student, graduate student, book, journal, line, rectangle, and circle are abstractions in different contexts.

The implementation here means the supporting algorithms and/or objects that the abstractions (business classes) use to implement themselves.

It is difficult to understand the Bridge pattern by only considering its intent.

However, it is powerful and applies to so many situations.

It is based on the following two important design principles:

- "Find what varies and encapsulate it"
- "Favor object composition over class inheritance"

I will explain the Bridge pattern using the following case study.

Case Study*:

Requirements:

Our customer needs a program that will draw rectangles with either of two drawing programs (drawing program 1 – DP1 or drawing program 2 – DP2).

Abstraction: Rectangles, Implementation: Drawing programs

Caution: The way of implementation varies.

The rectangles are defined as two pairs of corner points.

The contents of the drawing programs:

	DP1	DP2
Drawing a line :	draw_a_line(x1, y1, x2, y2)	drawline(x1, x2, y1, y2)
Drawing a circle:	draw_a_circle(x, y, r)	drawcircle(x, y, r)

The client of the rectangles (the user class) does not need to worry about what type of drawing program it should use.

During the instantiation of the rectangle, the drawing program is determined, and other classes can draw rectangles without knowing the type of the drawing program.

*Alan Shalloway, James R. Trott , *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley, 2002.

Solution 1: Using the Inheritance (*Not a proper solution!*)

In this solution, we will not apply the Bridge Pattern; instead, we will use inheritance to design different rectangles.

The way of thinking:

We have two different kinds of rectangle objects: one that uses DP1 and one that uses DP2.

Do we have really different rectangles?

Each would have a draw method but would implement it differently.

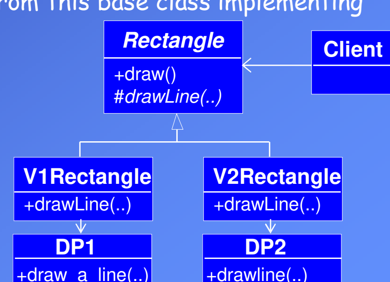
First, we write an abstract class Rectangle.

It has a template method (draw) that contains the skeleton to draw a rectangle.

Then we drive different types of rectangles from this base class implementing the drawLine methods (primitives) differently.

The drawLine methods in V1Rectangle calls the draw_a_line method of the DP1.

The drawLine methods in V2Rectangle calls the drawline method of DP2.



Discussion:

Is this solution object oriented? YES

Does it work? YES

But! Flexibility, extensibility, changes?

Program of the Solution 1 in Java:

```
abstract class Rectangle {
  private double _x1,_y1,_x2,_y2;
  public void draw () {              // Rectangle is responsible to draw itself  (Template)
     drawLine(_x1, _ y1, _x2 ,_y1);  // Primitive operations (see Template method pattern)
     drawLine(_x2, _y1, _x2, _y2);
     drawLine(_x2, _y2, _x1, _y2);
     drawLine(_x1, _y2, _x1, _y1);
  }
  abstract protected void  drawLine ( double x1, double y1, double x2, double y2);
}

class V1Rectangle extends Rectangle {
  drawLine( double x1, double y1, double x2, double y2) {     // Primitive operation
       DP1.draw_a_line( x1,y1,x2,y2);                         // It is connected to DP1
  }
}

class V2Rectangle extends Rectangle {
  drawLine( double x1, double y1, double x2, double y2) {       // Primitive operation
     // arguments are different in DP2 and must be rearranged
     DP2.drawline( x1,x2,y1,y2);                              // It is connected to DP2
  }
}
```

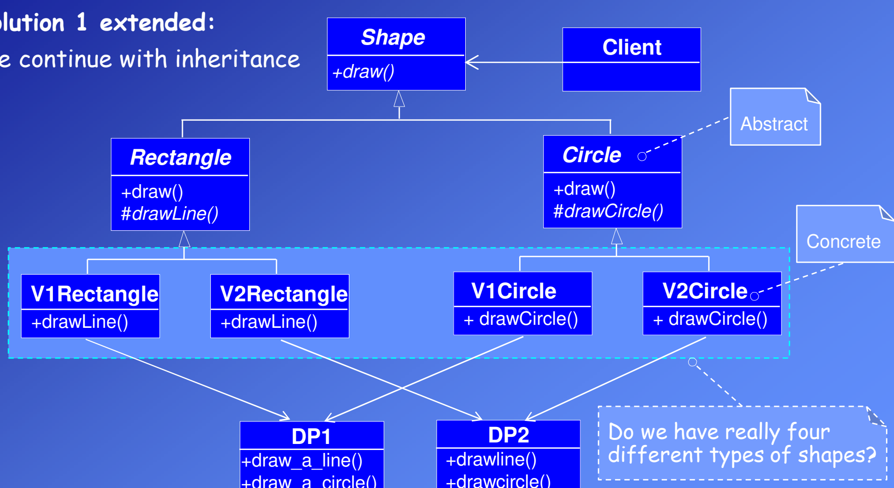

Requirements change!

The customer wants that we support another kind of shape, i.e., **a circle**.

It is also required that the client object does not know the difference between Rectangles and Circles.

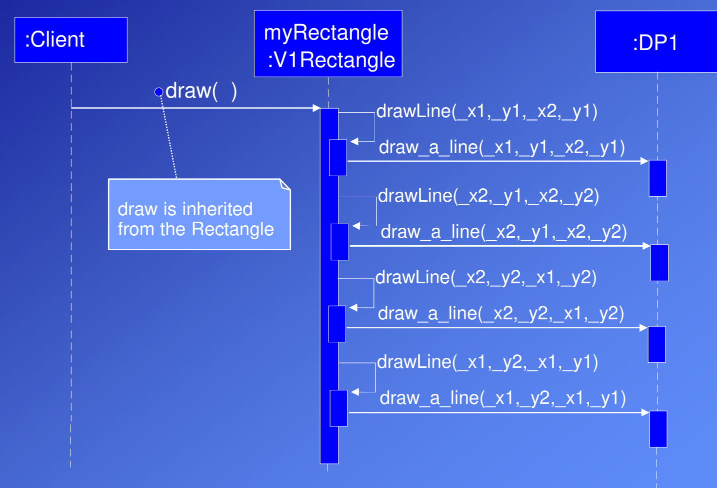
Solution 1 extended:

We continue with inheritance

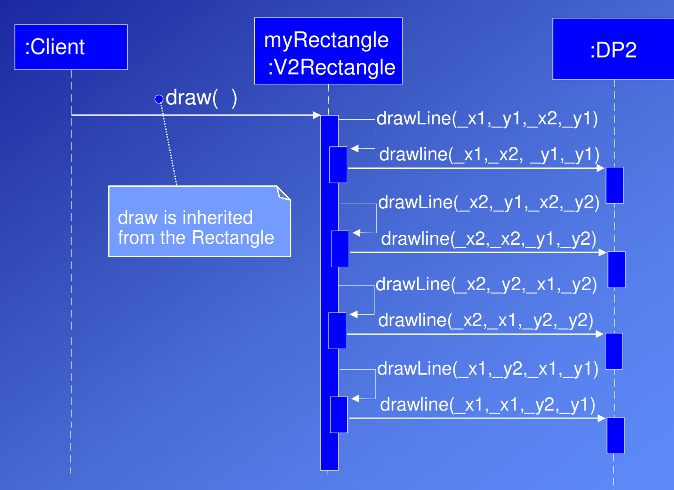


Operation of the system:

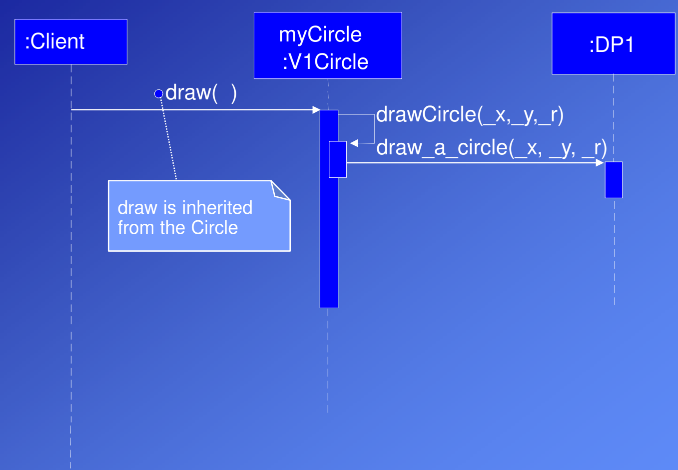
Assume that the client object is associated with a rectangle of type V1Rectangle.



When the client object is associated with a rectangle of type V2Rectangle:



When the client object is associated with a circle of type V1Circle:



Program of the extended Solution 1 in Java:

```
abstract class Shape {
  abstract public void draw ();
}
abstract class Rectangle extends Shape {
  public void draw () {    // Template method
      drawLine(_x1, _y1, _x2, _y1);
      drawLine(_x2, _y1, _x2, _y2);
      drawLine(_x2, _y2, _x1, _y2);
      drawLine(_x1, _y2, _x1, _y1);
  }
  abstract protected void
              drawLine( double x1, double y1,
                        double x2, double y2);
  private double _x1, _y1, _x2, _y2;
}

class V1Rectangle extends Rectangle {
  void drawLine ( double x1, double y1,
                        double x2, double y2) {
      DP1.draw_a_line( x1,y1,x2,y2);
  }
}
class V2Rectangle extends Rectangle {
  void drawLine (double x1, double x2,
                        double y1, double y2) {
      DP2.drawline( x1,x2,y1,y2);
  }
}
```

```
abstract class Circle extends Shape {
   public void draw () {
      drawCircle( cornerX, cornerY, radius);
   }
   abstract protected void
   drawCircle (double x, double y, double r);
   private double cornerX, cornerY, radius;

}
class V1Circle extends Circle {
  void drawCircle(x,y,r) {
     DP1.draw_a_circle( x,y,r);
  }
}
class V2Circle extends Circle {
  void drawCircle(x,y,r) {
      DP2.drawcircle( x,y,r);
  }
}
```

Discussion:

Is this solution object-oriented? YES

Does it work? YES

But!

a) What happens if we get another drawing program (DP3), another variation in implementation?

We will have *six* kinds of Shapes (two Shape concepts times three drawing programs).

To add just only one new drawing program (implementation), we have to add two shape classes.

b) What happens if we get another type of Shape, another variation in concept (abstraction)?

We will have *nine* types of Shapes (three Shape concepts times three drawing programs).

The class explosion problem!

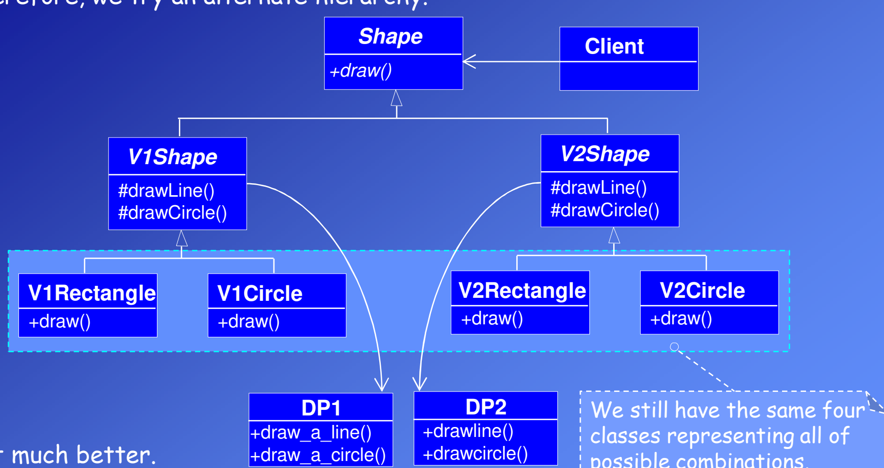
Reason for the problem: The abstraction (the kinds of Shapes) and the implementation (the drawing programs) are tightly coupled.

We used inheritance incorrectly and unnecessarily.

Remember: "Favor object composition over class inheritance"

Solution 2: We continue still with inheritance

We think that we were using the wrong kind of inheritance hierarchy.
Therefore, we try an alternate hierarchy.



Not much better.

The class explosion problem continues.

The abstraction (the kinds of Shapes) and the implementation (the drawing programs) are still tightly coupled.

The proper solution: The abstraction and the implementation are de-coupled

Instead of using the Bridge pattern directly, we will see that it is possible to find a proper solution by applying two principles.

These principles are:

- "*Find what varies and encapsulate it*".
- "*Favor composition over inheritance*".

What is varying in our system?

In our system, we have different types of Shapes and different types of drawing programs.

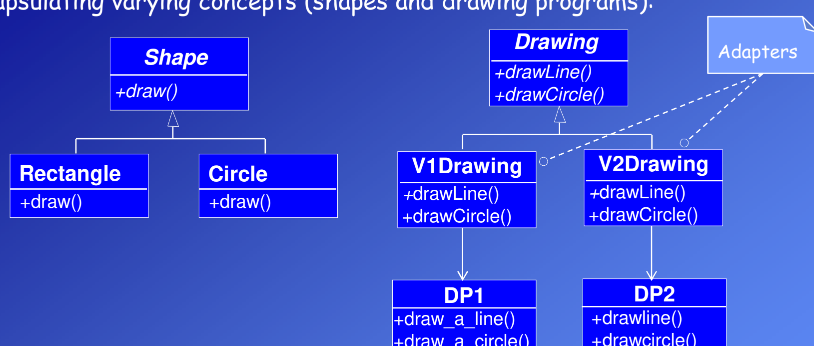
We will encapsulate varying concepts behind abstract classes.

Here, encapsulating means putting things in the same package and hiding the details (types) of these things from the users.

For example, we derive concrete classes Rectangle and Circle from the abstract base Shape, and the Client object is not aware of the particular kinds of shapes.

It is the base strategy of the "design to interface" principle.

Encapsulating varying concepts (shapes and drawing programs):



In the case study, the drawing programs DP1 and DP2 are external systems with different interfaces.

Therefore we apply the Adapter pattern and encapsulate the different adapters.

Actually, the Adapter pattern is not part of the Bridge pattern.

The **Shape** class encapsulates the concept of the types of shapes.

Shapes are responsible for knowing how to draw (+draw()).

Drawing objects are responsible for drawing lines and circles (drawLine(), drawCircle()).

Connecting two groups:

Now, we have two groups of classes.

How will they relate to each other?

Principle: "**Favor object composition over class inheritance**"

Can classes of one group use (have) classes of the other group?

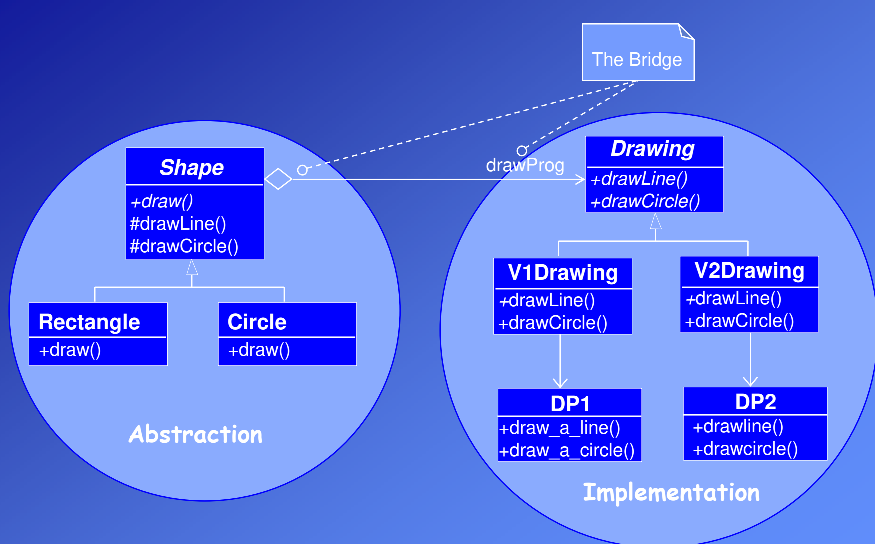
There are two possibilities:
1. Shape uses (has) the Drawing programs or
2. The Drawing programs use (have) Shape.

Consider the second case:

If drawing programs draw shapes directly, it violates encapsulation (Separation of concerns).
Drawing objects have to know specific information about the **Shape**s (the kind of shapes, how to draw them).
In this case, the objects are not responsible for their behaviors.

Consider the first case:
If Shapes use Drawing objects to draw themselves, they don't need to know what type of Drawing object is used.
We can connect Shapes to the Drawing class over a reference (or pointer) to the base class (interface).

Solution: with the Bridge pattern



Program of the solution in C++:

```
//DP1 and DP2 are external drawing programs
class DP1 {    // First drawing program
public:
     void static draw_a_line (double x1, double y1, double x2, double y2);
     void static draw_a_circle (double x, double y, double r);
};
class DP2 {   // Second drawing program
public:
    void static drawline (double x1, double x2, double y1, double y2);
    void static drawcircle (double x, double y, double r);
};

// Adapters to access the external drawing programs (implementation)
class Drawing {              // Abstract base class of adapters
public:
   virtual void drawLine (double, double, double, double)=0;
   virtual void drawCircle (double, double, double)=0;
};
```



```
class V1Drawing : public Drawing {    // Adapter of DP1)
public:
  void drawLine (double x1, double y1, double x2, double y2);
  void drawCircle( double x, double y, double r);
};

void V1Drawing::drawLine ( double x1, double y1, double x2, double y2) {
       DP1::draw_a_line(x1,y1,x2,y2);  // Access to DP1
}
void V1Drawing::drawCircle (double x, double y, double r) {
      DP1::draw_a_circle (x,y,r);
}

class V2Drawing : public Drawing {       // Adapter of DP2
public:
      void drawLine (double x1, double y1, double x2, double y2);
      void drawCircle(double x, double y, double r);
};
void V2Drawing::drawLine (double x1, double y1, double x2, double y2) {
      DP2::drawline(x1,x2,y1,y2); // Access to DP2
}
void V2Drawing::drawCircle (double x, double y, double r) {
     DP2::drawcircle(x, y, r);
}
```

```
// Shapes (Abstraction)
class Shape {                    // Abstract base class of Shapes
public:
    Shape (Drawing *);           // Constructor: Parameter is pointer to a drawing program
    virtual void draw()=0;
protected:
    void drawLine( double, double, double , double);
    void drawCircle( double, double, double);
private:
   Drawing *drawProg;               // Pointer to the related drawing program (bridge)
};

Shape::Shape (Drawing *dp) : drawProg{ dp }
{}                               // Constructor: Connection to the related implementation

void Shape::drawLine( double x1, double y1, double x2, double y2){
    drawProg->drawLine(x1, y1, x2, y2);    // Currently connected drawing program is used
}

void Shape::drawCircle(double x, double y, double r){
    drawProg->drawCircle(x, y, r);
}
```

The bridge
It connects shape to the drawing program.


```
//Concrete shape classes
class Rectangle : public Shape{
public:
    Rectangle (Drawing *, double, double, double, double);
    void draw();
private:
    double m_x1, m_y1, m_x2, m_y2;
};

Rectangle::Rectangle (Drawing *dp, double x1, double y1, double x2, double y2)
                    : Shape{ dp }, m_x1{x1}, m_x2{ x2 }, m_y1{ y1 }, m_y2{ y2 }
{}

void Rectangle::draw () {
    drawLine(m_x1, m_y1, m_x2, m_y1);     // drawLine is inherited from the Shape
    drawLine(m_x2, m_y1, m_x2, m_y2);
    drawLine(m_x2, m_y2, m_x1, m_y2);
    drawLine(m_x1, m_y2, m_x1, m_y1);
}
```

```
class Circle : public Shape{
public:
   Circle (Drawing *, double, double, double);
   void draw();
private:
   double _x, _y, _r;
};

Circle::Circle (Drawing *dp, double x, double y, double r)
                                          : Shape(dp), m_x{ x }, m_y{ y }, m_r{r}
{}

void Circle::draw () {
   drawCircle( m_x, m_y, m_r);           // drawCircle is inherited from the Shape class
}
```



```
// The Client  (user) class that uses the Shapes library written for testing purposes
class Client{
public:
   Client (Shape * inputShape) : shapePtr {inputShape}   //Initial shape to be used
    {}
   void setShape(Shape * inputShape) {           //change current shape
       shapePtr = inputShape;
   }
   void operate();                          // Responsibility of the Client
private:
   Shape *shapePtr;                         // It can point to any type of Shape
};

void Client::operate () {
   shapePtr->draw();                       //The client does not know the type of the shape
}
```

```
int main () {                          // The main function for testing purposes
  // Drawing objects
  Drawing *dp1, *dp2;
  dp1= new V1Drawing;
  dp2= new V2Drawing;

  // Shape objects
  Rectangle *rectangle1 = new Rectangle{ dp1, 1, 1, 2, 2 };        // Rectangle1 uses dp1
  Rectangle *rectangle2 = new Rectangle{ dp2, 10, 15, 20, 30 };    // Rectangle2 uses dp2
  Circle *circle = new Circle{dp2 , 2, 2, 4 };                     // Circle uses dp2

  Client user{ rectangle1 };             //The client (user) will use the rectangle1
  user.operate();
  user.setShape(circle);                 //The client (user) will use the circle
  user.operate();
  user.setShape(rectangle2);             //The client (user) will use the rectangle2
  user.operate();

  delete rectangle1; delete rectangle2; delete circle;     // Housekeeping
  delete dp1; delete dp2;
  return 0;
}
```

The adapters (or implementation objects) can be created by a factory

The shapes do not know the type of the drawing programs.

See Example Shapes_Bridge.cpp

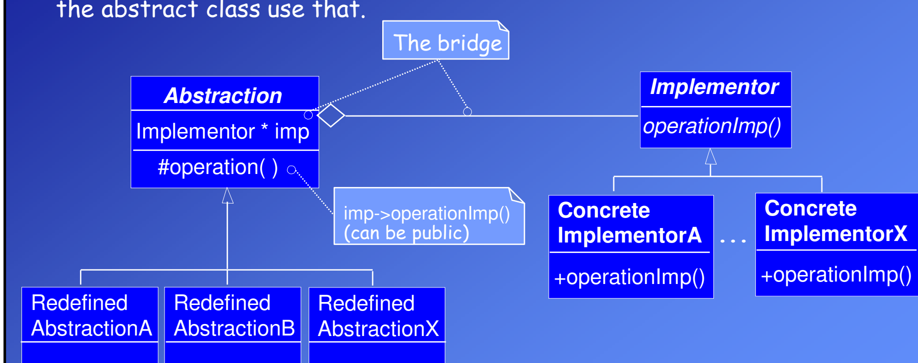
Definition of the Bridge Pattern:

Problem:

The derivations of an abstract class must use multiple implementations without causing an explosion in the number of classes.

Solution:

Define an interface for all implementations to use and have the derivations of the abstract class use that.



A concrete object (of redefined abstraction) can get the address of the proper Implementor object from a factory.

Summary: Important design principles

To develop flexible and reusable software, you have to consider the following design principles:

- Separation of concerns
  Each class focuses on its own responsibilities.
- Find what varies and encapsulate it
  Separate varying parts from stable parts.
- Favor object composition over class inheritance
  Do not use inheritance to add dynamic behavior to objects.
- Design to interface, not to implementation
  Client (user) classes should only consider (be aware of) common properties (interface) of varying objects.