```
class Subject {

public:
        virtual ~Subject;
        virtual   void Attach (observer*);
        virtual   void Detach (observer*) ;
        virtual   void Notify();
protected:
        Subject ();
private:
        List <Observer*> *_observers;
};


void Subject :: Attach (Observer* o){
      _observers -> Append(o);
}
void Subject :: Detach (Observer* o){
      _observers -> Remove(o);
}
void Subject :: Notify (){

                  iter.CurrentItem() -> Update(this);
      }
}
```

Abstract class defining the Subject interface.

Observer Pattern: A Concrete Subject [1]

```
class ClockTimer : public Subject {
public:
                ClockTimer();

                virtual int GetHour();

                virtual int GetMinutes();

                virtual int GetSecond();

                void Tick ();

}


ClockTimer :: Tick {

    // Update internal time keeping state.
    // gets called on regular intervals by an internal
       timer.

                Notify();

 }
```

Observer Pattern: A Concrete Observer [1]

```
class DigitalClock: public Widget, public
    Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();
    virtual void  Update(Subject*);
    virtual void  Draw();
private:
    ClockTimer* _subject;
    }
```

Override Observer operation.

Override Widget operation.

```
DigitalClock ::DigitalClock (ClockTimer* s) {
        _subject = s;
        _subject→Attach(this);
}

DigitalClock ::~DigitalClock() {
        _subject->Detach(this);

}
```

```
void DigitalClock ::Update (subject* theChangedSubject ) {
    If (theChangedSubject == _subject) {
        Draw();
    }
}

void DigitalClock ::Draw () {
  int hour = _subject->GetHour();
  int minute = _subject->GeMinute(); // etc.
  // Code for drawing the digital clock.
}
```

Observer Pattern: Main (skeleton)

```
ClockTimer* timer = new ClockTimer;

DigitalClock* digitalClock = new DigitalClock (timer);
```

Observer Pattern: Consequences

- Abstract coupling between subject and observer. Subject has no knowledge of concrete observer classes. (What design principle is used?)
- Support for broadcast communication. A subject need not specify the receivers; all interested objects receive the notification.
- Unexpected updates: Observers need not be concerned about when then updates are to occur. They are not concerned about each other's presence. In some cases this may lead to unwanted updates.

When to use the Observer Pattern?

- *When* an abstraction has two aspects: one dependent on the other. Encapsulating these aspects in separate objects allows one to vary and reuse them independently.
- *When* a change to one object requires changing others and the number of objects to be changed is not known.
- When an object should be able to notify others without knowing who they are. Avoid tight coupling between objects.

UNIT-V

Behavioral Patterns

Behavioural Patterns Part-II(cont'd) : State, Strategy, Template Method, Visitor, Discussion of Behavioural Patterns.

General Description

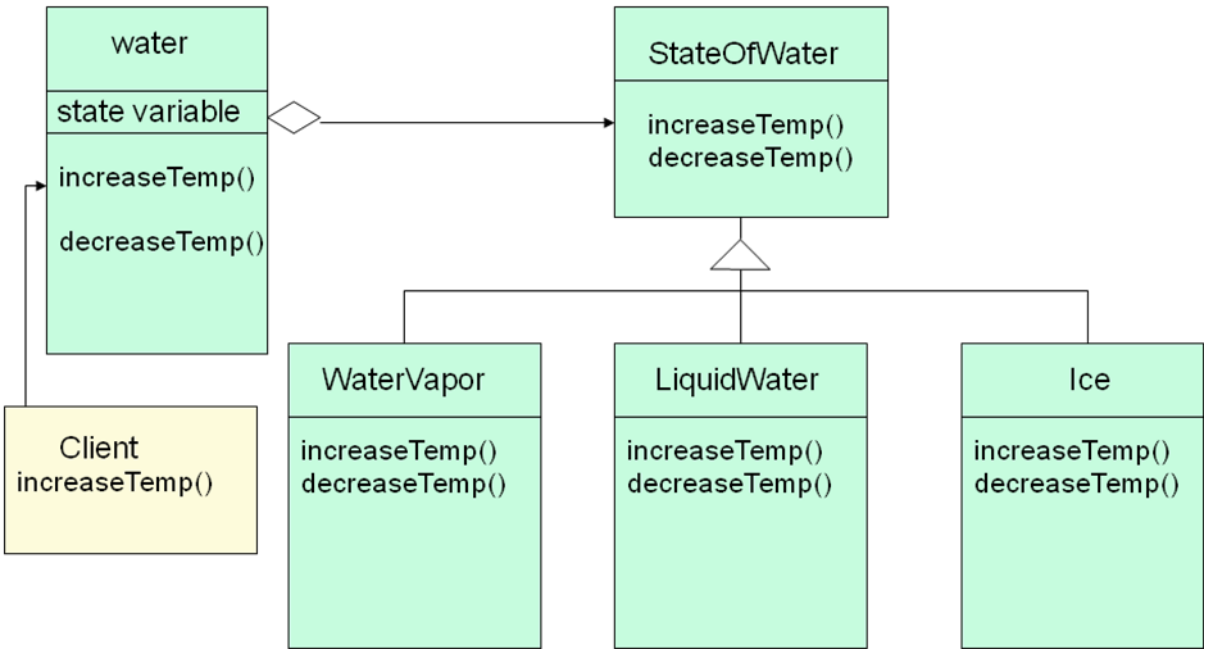
- A type of Behavioral pattern.
- Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Uses Polymorphism to define different behaviors for different states of an object.

When to use STATE pattern ?

- State pattern is useful when there is an object that can be in one of several states, with different behavior in each state.
- To simplify operations that have large conditional statements that depend on the object's state.

```
if (myself = happy)
then

{
                    eatIceCream();

                    ....

                    }

else if (myself = sad) then

{

                     goToPub();

                     ....

  }

else if (myself = ecstatic) then

{

                     ....
```

Example I



How is STATE pattern implemented ?

- "Context" class:

  Represents the interface to the outside world.

- "State" abstract class:

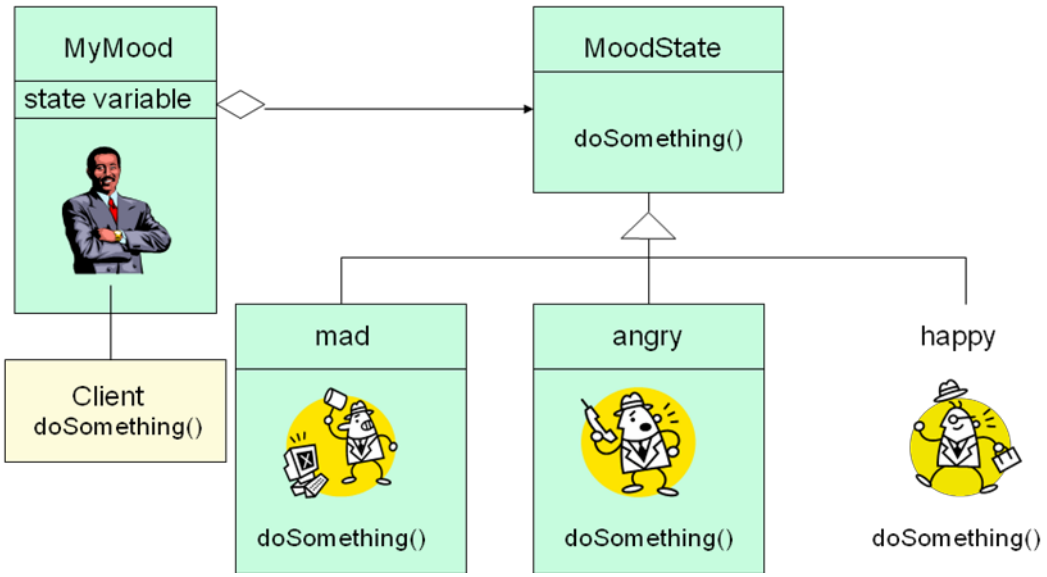
  Base class which defines the different states of the "state machine".

- "Derived" classes from the State class:

  Defines the true nature of the state that the state machine can be in.

Context class maintains a pointer to the current state. To change the state of the state machine, the pointer needs to be changed.

Example II



Benefits of using STATE pattern

- **Localizes all behavior associated with a particular state into one object.**
  - New state and transitions can be added easily by defining new subclasses.
  - Simplifies maintenance.
- **It makes state transitions explicit.**
  - Separate objects for separate states makes transition explicit rather than using internal data values to define transitions in one combined object.
  - **State objects can be shared.**
  - Context can share State objects if there are no instance variables.

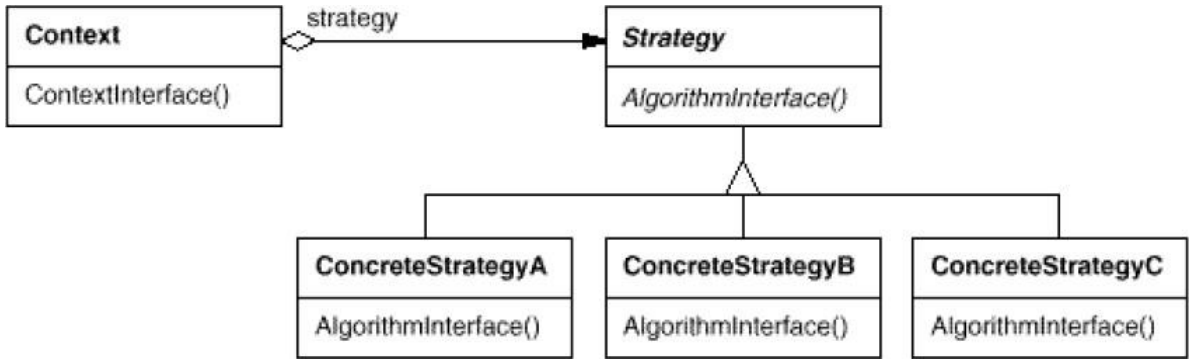
Food for thought…

- **To have a monolithic single class or many subclasses ?**
  - Increases the number of classes and is less compact.
  - Avoids large conditional statements.
  - **Where to define the state transitions ?**

- If criteria is fixed, transition can be defined in the context.
- More flexible if transition is specified in the State subclass.
- Introduces dependencies between subclasses.
- **Whether to create State objects as and when required or to create-them- once-and-use-many-times ?**
- First is desirable if the context changes state infrequently.
- Later is desirable if the context changes state frequently.

Pattern: Strategy

objects that hold alternate algorithms to solve a problem



Strategy pattern

- pulling an algorithm out from the object that contains it, and encapsulating the algorithm (the "strategy") as an object
- each strategy implements one behavior, one implementation of how to solve the same problem
  - how is this different from **Command** pattern?
- separates algorithm for behavior from object that wants to act
- allows changing an object's behavior dynamically without extending / changing the object itself
- **examples**:
  - file saving/compression
  - layout managers on GUI containers
  - AI algorithms for computer game players

Strategy example: Card player

```
// Strategy hierarchy parent
// (an interface or abstract
class) public interface
Strategy { public Card
getMove();
}
// setting a strategy
player1.setStrategy(new
SmartStrategy());
// using a strategy
Card p1move = player1.move(); // uses strategy
```

Strategy: Encapsulating Algorithms

Name: Strategy design pattern

Problem description:

Decouple a policy-deciding class from a set of mechanisms, so that different mechanisms can be changed transparently.

Example:

A mobile computer can be used with a wireless network, or connected to an Ethernet, with dynamic switching between networks based on location and network costs.

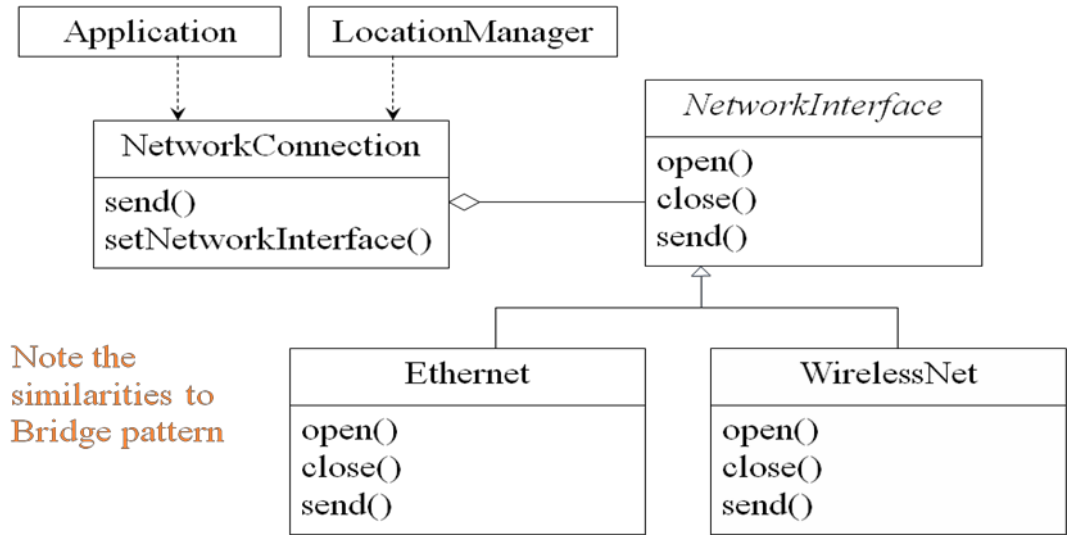
Solution:

A Client accesses services provided by a Context.

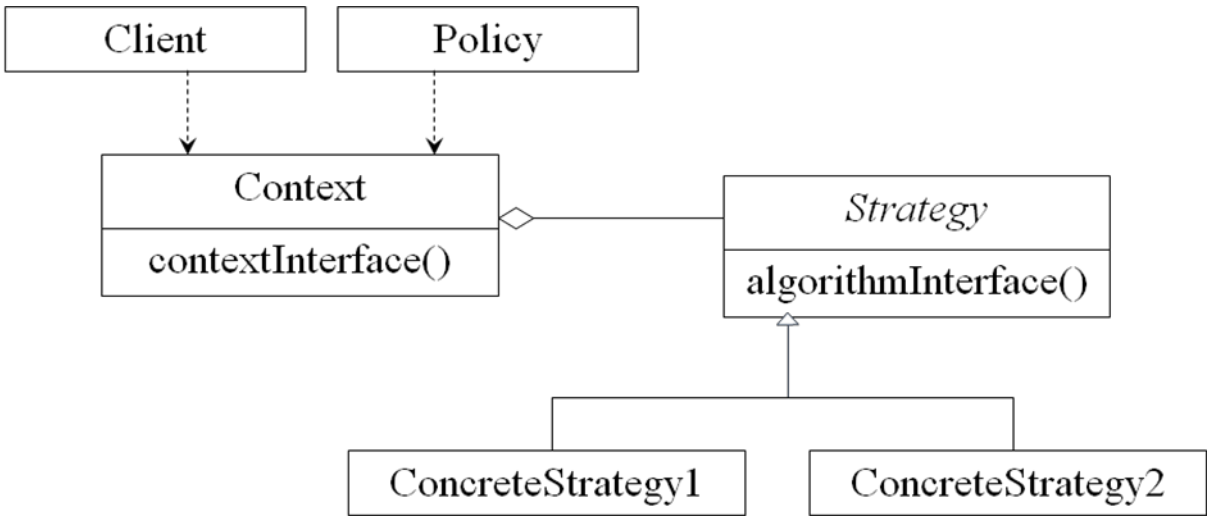
The Context services are realized using one of several mechanisms, as decided by a Policy object.

The abstract class Strategy describes the interface that is common to all mechanisms that Context can use. Policy class creates a ConcreteStrategy object and configures Context to use it.

Strategy Example: Class Diagram for Mobile Computer



Strategy: Class Diagram



Strategy: Consequences

Consequences:

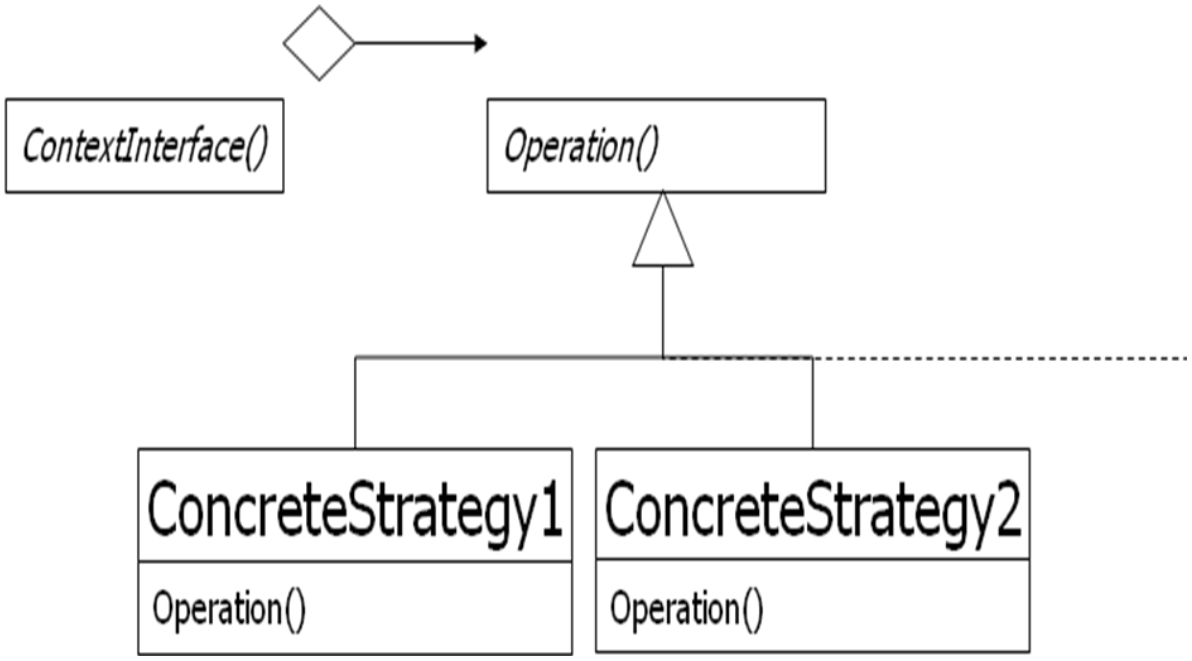
ConcreteStrategies can be substituted transparently from Context.

Policy decides which Strategy is best, given the current circumstances.

New policy algorithms can be added without modifying Context or Client.

Strategy

- **You want to**
  - use different algorithms depending upon the context
  - avoid having to change the context or client
- **Strategy**
  - decouples interface from implementation
  - shields client from implementations
  - Context is not aware which strategy is being used; Client configures the Context
  - strategies can be substituted at runtime
  - example: interface to wired and wireless networks
- Make algorithms interchangeable---"changing the guts"
- Alternative to subclassing
- Choice of implementation at run-time
- Increases run-time complexity



Template Method

Conducted By Raghavendar Japala

Topics – Template Method

- Introduction to Template Method Design Pattern
- Structure of Template Method
- Generic Class and Concrete Class
- Plotter class and Plotter Function Class

Introduction

The DBAnimationApplet illustrates the use of an **abstract class** that serves as a template for classes with shared functionality.

An abstract class contains behavior that is common to all its subclasses. This behavior is encapsulated in nonabstract methods, which may even be declared ***final*** to prevent any modification. This action ensures that all subclasses will inherit the same common behavior and its implementation.

The abstract methods in such templates ensure the interface of the subclasses and require that context specific behavior be implemented for each concrete subclass.

Hook Method and Template Method

The abstract method paintFrame() acts as a placeholder for the behavior that is implemented differently for each specific context.

We call such methods, *hook* methods, upon which context specific behavior may be hung, or implemented.

The paintFrame() hook is placed within the method update(), which is common to all concrete animation applets. Methods containing hooks are called *template* methods.

Hook Method and Template Method (Con't)

The abstract method paintFrame() represents the behavior that is changeable, and its implementation is deferred to the concrete animation applets.

We call paintFrame() a hook method. Using the hook method, we are able to define the update() method, which represents a behavior common to all the concrete animation applets.

Frozen Spots and Hot Spots

A template method uses hook methods to define a common behavior.

Template method describes the fixed behaviors of a generic class, which are sometimes called **frozen spots**.

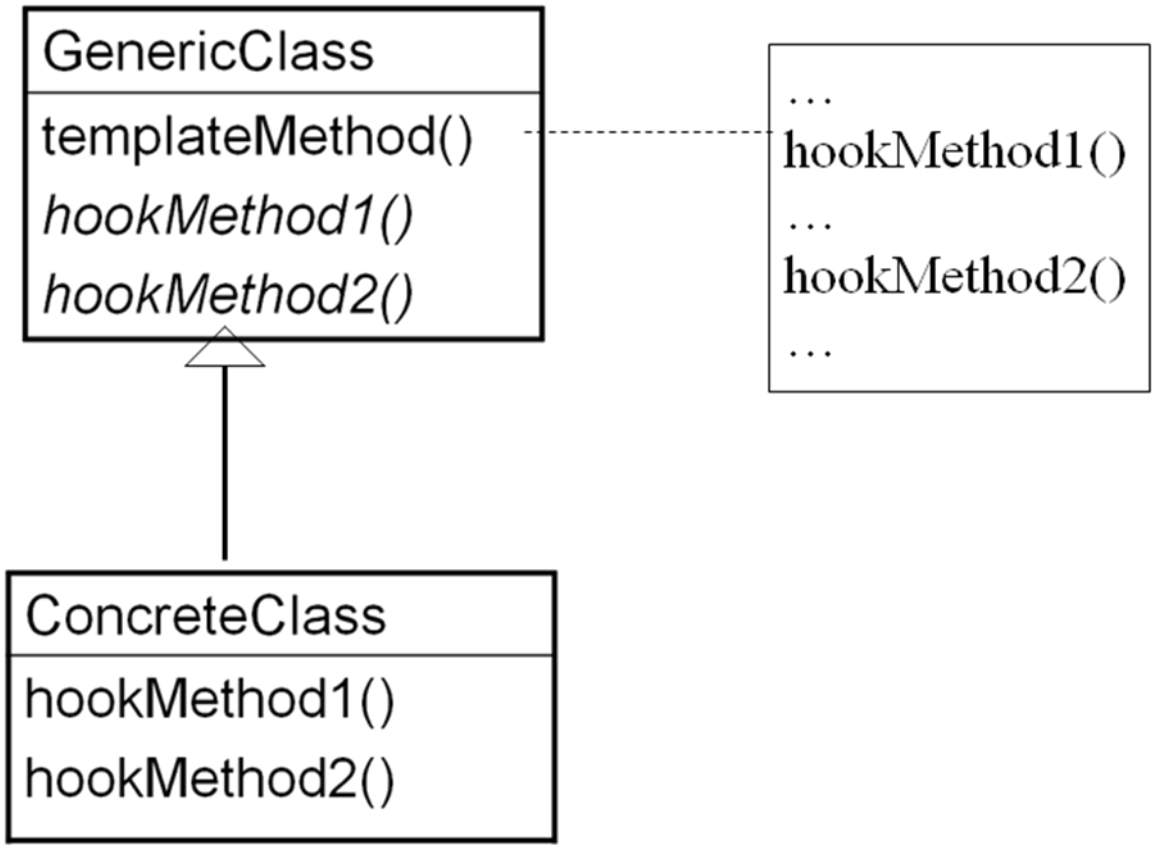
Hook methods indicate the changeable behaviors of a generic class, which are sometimes called **hot spots**.

Hook Method and Template Method (Con't)

The abstract method paintFrame() represents the behavior that is changeable, and its implementation is deferred to the concrete animation applets.

We call paintFrame() a hook method. Using the hook method, we are able to define the update() method, which represents a behavior common to all the concrete animation applets.

Structure of the Template Method Design Pattern



Structure of the Template Method Design Pattern (Con't)

GenericClass (e.g., DBAnimationApplet), which defines abstract hook methods (e.g., paintFrame()) that concrete subclasses (e.g., Bouncing-Ball2) override to implement steps of an algorithm and implements a template method (e.g., update()) that defines the skeleton of an algorithm by calling the hook methods;

ConcreteClass (e.g., Bouncing-Ball2) which implements the hook methods (e.g., paintFrame()) to carry out subclass specific steps of the algorithm defined in the template method.

Structure of the Template Method Design Pattern (Con't)

In the Template Method design pattern, *hook methods* **do not** have to be

abstract. The generic class may provide default implementations for the hook

methods.

Thus the subclasses have the option of overriding the hook methods or using the default implementation.

The initAnimator() method in DBAnimationApplet

is a nonabstract hook method with a default

implementation. The init() method is another

template method.

A Generic Function Plotter

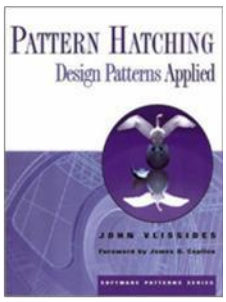
The generic plotter should factorize all the behavior related to drawing and leave only the definition of the function to be plotted to its subclasses.

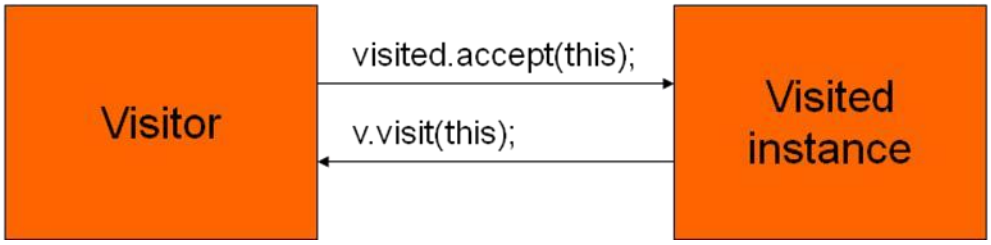
A concrete plotter PlotSine will be implemented to plot the function

$y = sin\ x$

Pattern Hatching

Visitor pattern

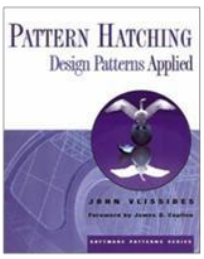




L8

Pattern Hatching

Visitor Pattern




```
Class Visitor {
public:
        Visitor();
        void visit(File*);
        void visit(Directory*);
        void visit (Link*);
};

        }
Visitor cat;
node->accept(cat);
```

```
class ClockTimer : public Subject {
public:
        ClockTimer();
        virtual int GetHour();
        virtual int GetMinute();
        virtual int GetSecond();
        void Tick ();
```

```
void Visitor::visit (File* f)
        {f->streamOut(cout);}

void Visitor::visit (Directory* d)
        {cerr << "no printout for a
        directory";}

void Visitor::visit (Link* l)
        {l->getSubject()->accept(*this);}
```

```
void File::accept (Visitor& v)
        {v.visit(this);}
void Directory::accept (Visitor& v)
        {v.visit(this);}
void Link::accept (Visitor& v)
        {v.visit(this);}
```

What to Expect from Design Patterns, A Brief History, The Pattern Community
An Invitation, A Parting Thought.

What to Expect from Design Patterns?

- A Common Design Vocabulary.
- A Documentation and Learning Aid.
- An Adjunct to Existing Methods.
- A Target for Refactoring.

A common design vocabulary

1. Studies of expert programmers for conventional languages have shown that knowledge and experience isn't organized simply around syntax but in larger conceptual structures such as algorithms, data structures and idioms [AS85, Cop92, Cur89, SS86], and plans for fulfilling a particular goal [SE84].

2. Designers probably don't think about the notation they are using for recording the designing as much as they try to match the current design situation against plans, data structures, and idioms they have learned in the past.

3. Computer scientists name and catalog algorithms and data structures, but we don't often name other kinds of patterns. Design patterns provide a common vocabulary for designers to use to communicate, document, and explore design alternatives.

A document and learning aid:

1. Knowing the design patterns makes it easier to understand existing systems.

2. Most large object-oriented systems use this design patterns people learning object- oriented programming often complain that the systems they are working with use inheritance in convoluted ways and that it is difficult to follow the flow of control.

3. In large part this is because they do not understand the design patterns in the system learning these design patterns will help you understand existing object-oriented system**.**