UNIT-V

Behavioral Patterns

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

General Description

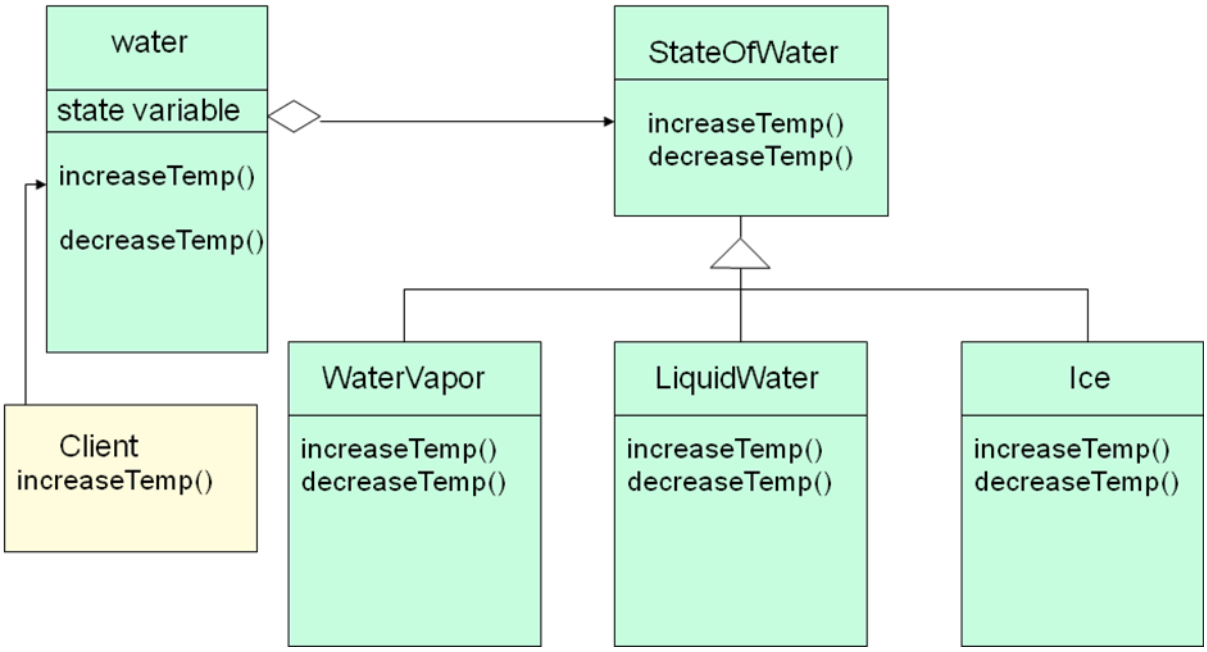
- A type of Behavioral pattern.
- Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Uses Polymorphism to define different behaviors for different states of an object.

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:

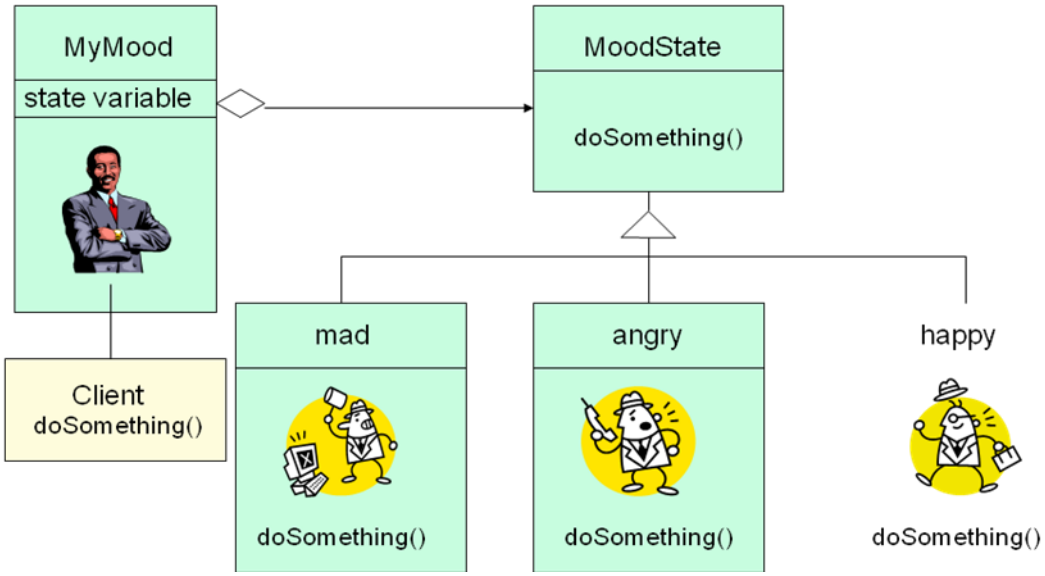
  Base class which defines the different states of the "state machine".

- "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.

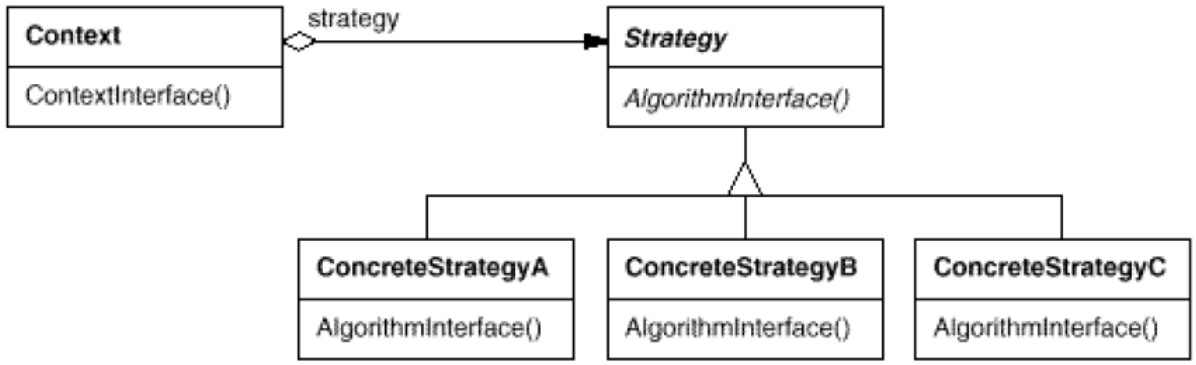
Food for thought…

- **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.

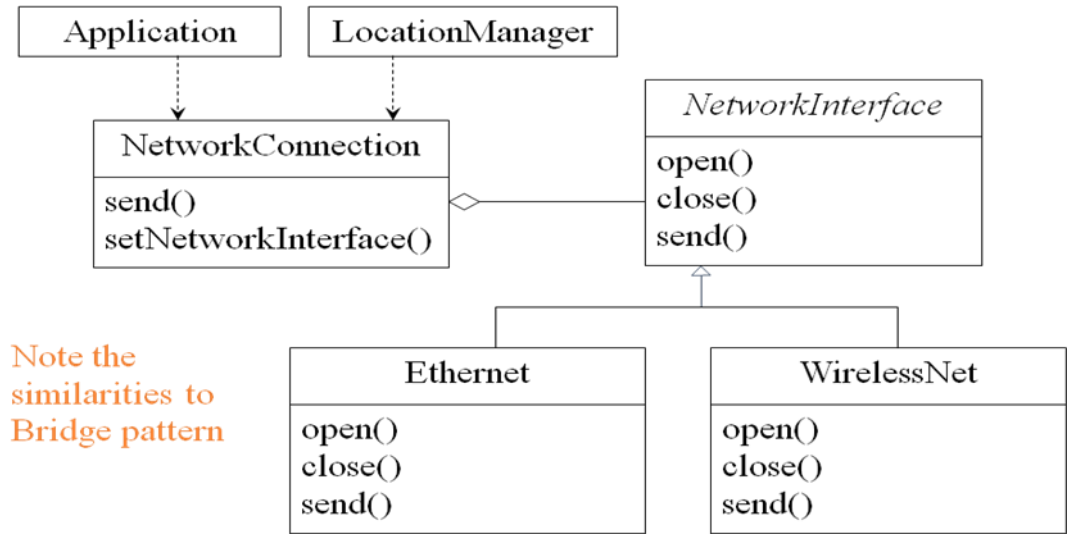
Solution:

A Client accesses services provided by a Context.

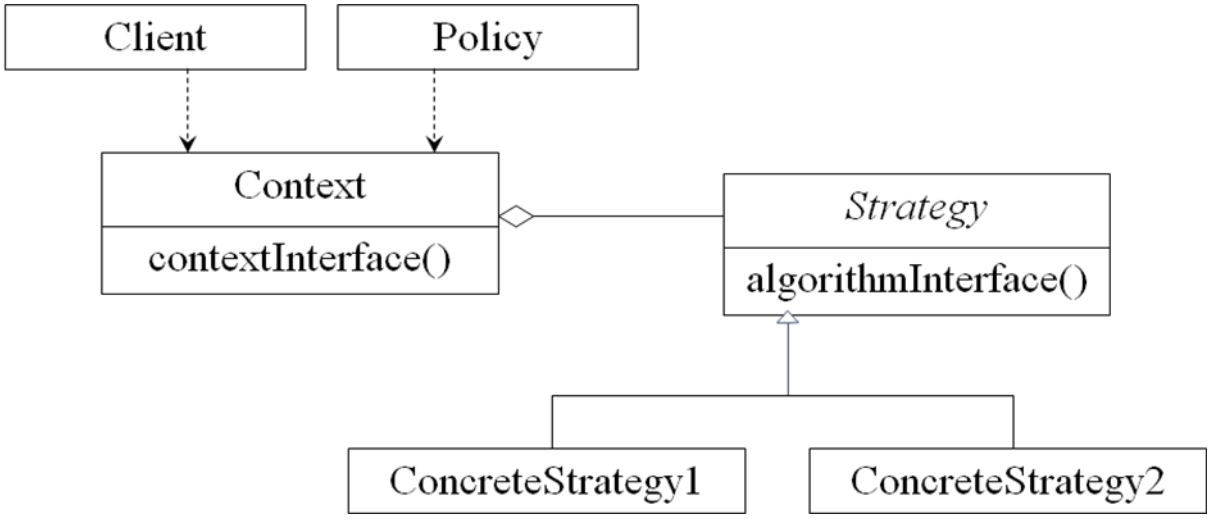
The Context services are realized using one of several mechanisms, as decided by a Policy object.

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:

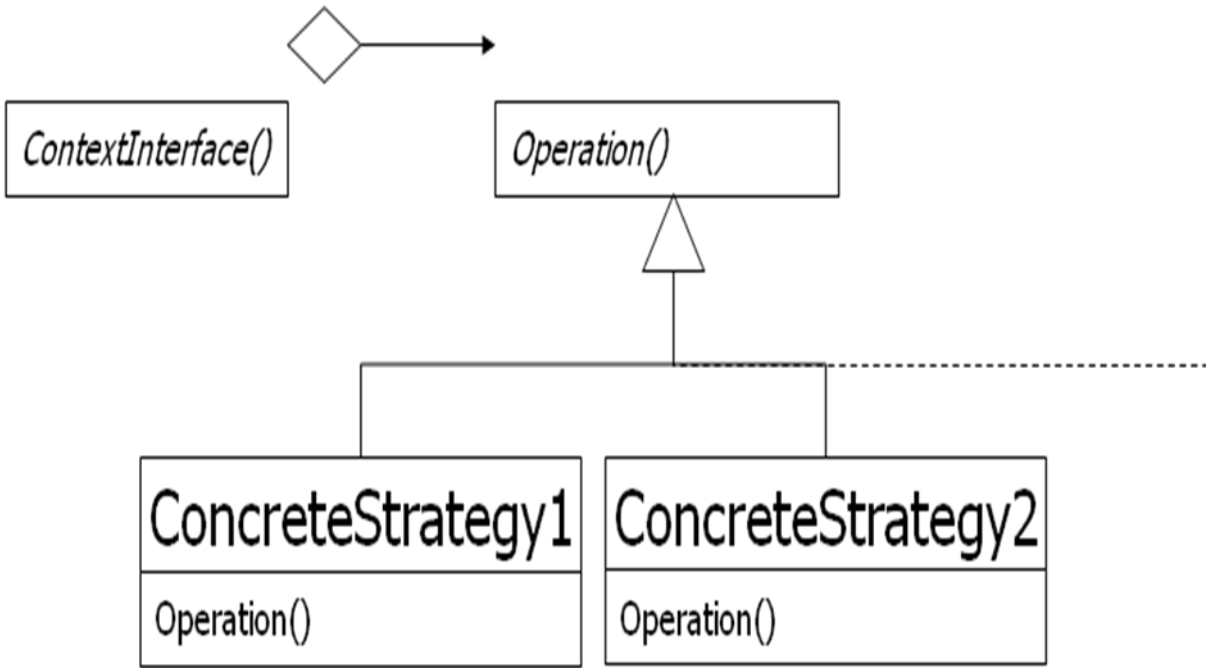
ConcreteStrategies can be substituted transparently from Context.

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**.

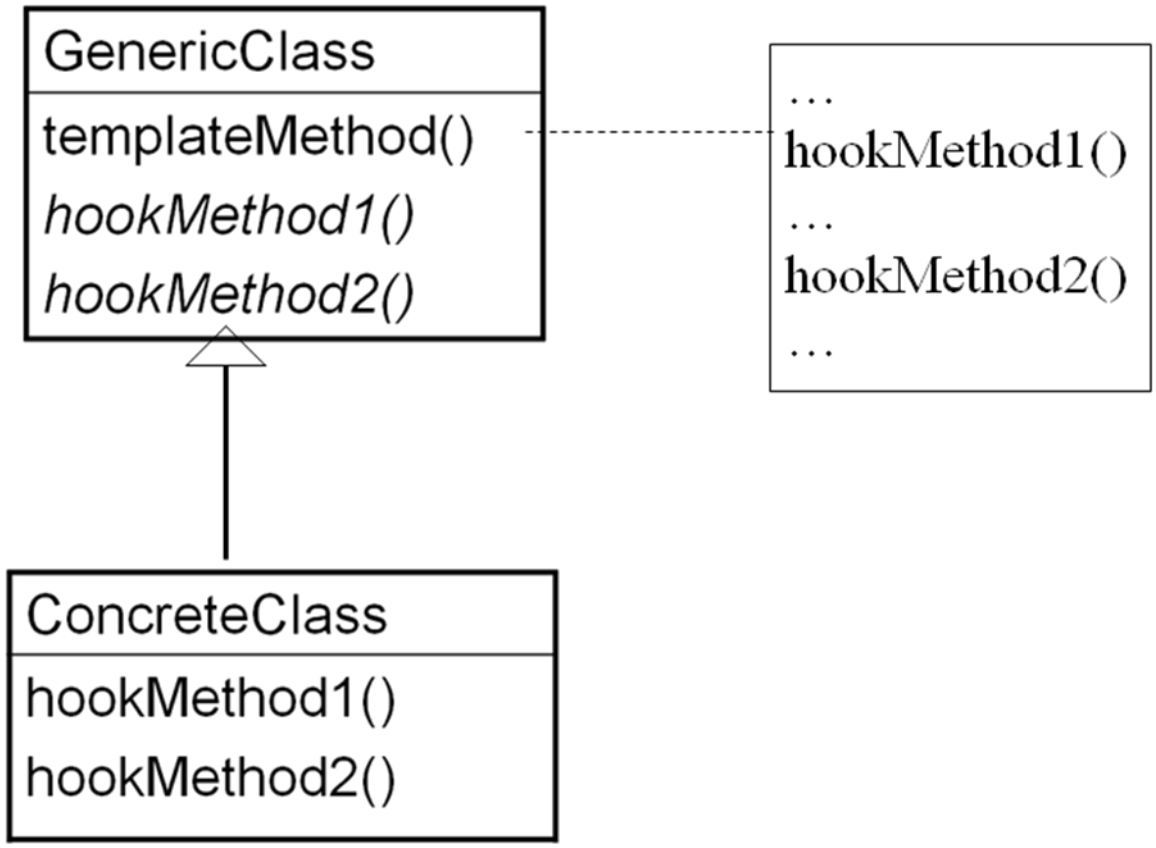
Hook methods indicate the changeable behaviors of a generic class, which are sometimes called **hot spots**.

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

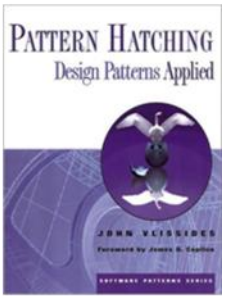
The generic plotter should factorize all the behavior related to drawing and leave only the definition of the function to be plotted to its subclasses.

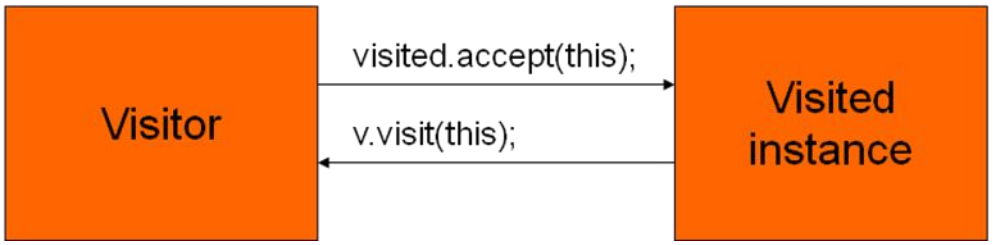
A concrete plotter PlotSine will be implemented to plot the function

$y = sin\ x$

Pattern Hatching

Visitor pattern

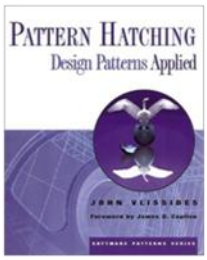




L8

Pattern Hatching

Visitor Pattern




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

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

```
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);}
```

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

```
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**.**

An adjacent to existing methods:

1. Object-oriented design methods are supposed to promote good design, to teach new designers how to design well, and standardize the way designs are developed.
2. A design method typically defines a set of notations (usually graphical) for modeling various aspects of design along with a set of rules that govern how and when to use each notation.
3. Design methods usually describe problems that occur in a design, how to resolve them and how to evaluate design. But then have not been able to capture the experience of expert designers.
4. A full fledged design method requires more kinds of patterns than just design patterns there can also be analysis patterns, user interface design patterns, or performance tuning patterns but the design patterns are an essential part, one that's been missing until now.

A target for refactoring:

1. One of the problems in developing reusable software is that it often has to be recognized or refactored [OJ90].
2. Design patterns help you determine how to recognize a design and they can reduce a amount of refactoring need to later.

The life cycle of object-oriented software has several faces. Brain Foote identifies these phases as the prototyping expansionary, and consolidating phases [Foo92**].**

Design Patterns Applied:

Example: An Hierarchical File System

Tree Structure → Composite

Patterns Overview

Symbolic Links → Proxy

Extending Functionality → Visitor

Single User Protection → Template Method

Multi User Protection → Singleton

User and Groups → Mediator

A Brief History of Design Patterns

- 1979--Christopher Alexander pens The Timeless Way of Building
  - Building Towns for Dummies
  - Had nothing to do with software
- 1994--Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the Gang of Four, or GoF) publish Design patterns: Elements of Reusable Object-Oriented Software
  - Capitalized on the work of Alexander
  - The seminal publication on software design patterns.

What's In a Design Pattern—1994

- The GOF book describes a pattern using the following four attributes:
  - The *name* to describes the pattern, its solutions and consequences in a word or two
  - The *problem* describes when to apply the pattern
  - The *solution* describes the elements that make up the design, their relationships, responsibilities, and collaborations
  - The *consequences* are the results and trade-offs in applying the pattern
- All examples in C++ and Smalltalk.

What's In a Design Pattern – 2002

- Grand's book is the latest offering in the field and is very Java centric. He develops the GOF attributes to a greater granularity and adds the Java specifics
  - Pattern name—same as GOF attribute
  - Synopsis—conveys the essence of the solution
  - Context—problem the pattern addresses
  - Forces—reasons to, or not to use a solution
  - Solution—general purpose solution to the problem
  - Implementation—important considerations when using a solution
  - Consequences—implications, good or bad, of using a solution
  - Java API usage—examples from the core Java API
  - Code example—self explanatory
  - Related patterns—self explanatory

Grand's Classifications of Design Pattern:

- Fundamental patterns
- Creational patterns
- Partitioning patterns
- Structural patterns
- Behavioral patterns
- Concurrency patterns

The Pattern Community An Invention

- ❑ Christopher Alexander is the architect who first studied

Patterns in buildings and communities and developed

A PATTERN LANGUAGE for generating them.

- ❑ His work has inspired time and again. So it's fitting worth

while To compare our work to his.

- ❑ Then we'll look at other's work in software-related patterns.

Alexander's Pattern Languages

There are many ways in which our work is like Alexander's

Both are based on observing existing systems and looking for patterns in them.

Both have templates for describing patterns although

our templates are quite different)..

But there are just as many ways in which our work different.

- ➢ People have been making buildings for thousands of years, and there are many classic examples to draw upon. We have been making Software systems for a

Relatively short time, and few are considered classics.

- ➢ Alexander gives an order in which his patterns should be used; we have not.
- ➢ Alexander's patterns emphasize the problems they adderss ,
- ➢ where as design patterns describes the solutions in more detail.
- ➢ Alexander claims his patterns will generate complete buildings.

We do not claim that our patterns will generate complete programs.

When Alexander claims you can design a house simply applying his patterns one after

Another ,he has goals similar to those of object-oriented design methodologies who

Gives step-by-step rules for design,

In fact ,we think it's unlikely that there will ever be a compete pattern language for soft

-ware.

But certainly possible to make one that is more complete.

A Parting Thought.

The best designs will use many design patterns that dovetail And intertwine to produce a greater whole.

As Alexander says:

It is possible to make buildings by stringing together pattern's,

In a rather loose way,

A building made like this , is an assembly of patterns. it is not Dense.

It is not profound. but it is also possible to put pattern's together

In such a way that many patterns overlap in the same physical

Space: the building is very dense; it has many meaning captured

In a small space; and through this density, it becomes profound.