

Introduction To Design Principles and Patterns

June 24, 2011

Proprietary and Confidential

- 1 -



Copyright © 2011 IGATE Corporation. All rights reserved. No part of this publication shall be reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior written permission of IGATE Corporation.

IGATE Corporation considers information included in this document to be Confidential and Proprietary.

Introduction To Design Principles and Patterns

Document History

Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
15-Feb-2011	0.01D	NA	Veena Deshpande	Content Compilation
26 Jul 2011	1.0	NA	Veena Deshpande Kishori Khadilkar	Baselining after content and format Review

June 24, 2014

Proprietary and Confidential

- 2 -



Course Goals and Non Goals

➤ Course Goals

- At the end of this course, participants gain an understanding of
 - Key Design Principles
 - Overview of Design Patterns with some examples

➤ Course Non Goals

- Identification and application of design patterns for the purpose of designing



Pre-requisites

- **Fair knowledge of**
 - Object Oriented Concepts
 - Unified Modeling Language

Intended Audience

- Programmers working with Object Oriented Languages



Day Wise Schedule

➤ Day 1

- Lesson 1: Introducing Design Principles
- Lesson 2: Introducing Design Patterns
- Lesson 3: Examples of Design Patterns

Table of Contents

➤ Lesson 1: Introducing Design Principles

- 1.1: What goes into Good Design
- 1.2: Introducing Design Heuristics
- 1.3: Some Design Principles

➤ Lesson 2: Introducing Design Patterns

- 2.1: What is a Design Pattern
- 2.2: Why Design Patterns
- 2.3: Design Patterns Drawbacks
- 2.4: Design Pattern Categories

➤ Lesson 3: Examples of some Design Patterns

- 3.1: Fundamental Patterns: Delegation, Interface, Abstract Superclass
- 3.2: Creational Patterns: Factory Method, Singleton
- 3.3: Structural Patterns: Adapter, Façade
- 3.4: Behavioural Patterns: State, Strategy, Template Method

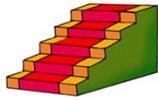
References

- **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process**
 - By Craig Larman
- **Object Oriented Design Heuristics**
 - By Arthur J. Riel
- **Head First Design Patterns**
 - By Elisabeth Freeman, Eric Freeman, Bert Bates, Kathy Sierra



Next Step Courses

- Object Oriented Analysis and Design
- GoF Design Patterns
- Technology Specific Designing and Technology Specific Design Patterns



Introduction to Design Principles and Patterns

Lesson 01 : Introducing Design Principles

Lesson Objectives

➤ At the end of this lesson, you will be able to :

- State and explain the principles guiding software designing.
- Introduce checks preventing a design from rotting.



Lesson Objectives:

We are now familiar with features of an object oriented language

We write our Object Oriented programs based on the designs provided for the application

For a given design, is there any method to gauge whether it is good, bad, or somewhere in between?

We may get the answer from an “OO Guru”. If the design “feels right”, the Guru certifies that the design is good.

How do we know if the design “feels right”?

We can get the answer by looking at the design heuristics and principles introduced in this lesson.

1.1: Design

Characteristics of a Good Design

- **A good software design:**
 - Is dynamic and resilient.
 - Is capable of adapting to frequent change requirements during:
 - Development phase
 - Maintenance phase
 - Changes minimally to accommodate extension of requirements.
 - Changes minimally to accommodate radical changes in the input and output methods of the program.
 - Has no redundancy.

Characteristics of a Good Design:

Today, we live in a world that is highly dynamic and diverse in nature. As a result, our requirements too are constantly changing. Therefore it is not surprising that “dynamic” and “resilient” software systems are the need of the day.

The challenge is in developing a software system capable of adapting to the ever changing requirements, complexities of the problem domain, and difficulty of managing software processes. A good design and code adapts easily to the frequent changes that are done during development and maintenance. Very often extensive changes are involved, when a new functionality is added to an application. The design should be able to incorporate the added functionality with minimum change to the existing codes.

Existing codes are already tested units, and changing them may result in unwanted changes in related functionalities. A software designer should make his or her design foolproof against such eventualities.

1.1: Design

Symptoms of a Rotting Design

➤ Here are some obvious symptoms of a rotting design:

- Rigidity:
 - Code does not adapt to changes.
 - Managers refuse to allow changes in the software.
- Fragility:
 - The smallest of changes results in a cascading effect.
 - Code breaks in many places, with every change.
- Immobility:
 - The design poses inability to reuse software from other projects.
- Viscosity:
 - It is easier to hack than retain the original design.
 - The development environment is slow and inefficient.

Symptoms of a Rotting Design:

There are four primary symptoms of a rotting design:

Rigidity: It is the tendency of a software to change even in the smallest of ways. Every change results in a cascading effect, bringing about subsequent changes in related modules. When software exhibits such characteristics, the managers avoid fixing even the simplest of problems.

Fragility: It is the tendency of the software to break in many places every time it changes. Often the break occurs at a point remotely connected with the point of change. In fact, the two points may not be related at all. Due to the break, the fragility increases, and soon the situation goes beyond control.

Immobility: It is the inability to reuse software from other projects or from other parts of the same project. This situation occurs when a module has too much related software that it depends on.

Viscosity: Viscosity is of two types: viscosity of design and viscosity of environment. When design preserving methods are more difficult to implement than the hacks, then we can say that the viscosity of design is very high. When the design environment is slow and inefficient we can say that the viscosity of environment is high.

1.2: Design Heuristics

Introduction to Design Heuristics

- Heuristics provide experience-based guidelines to help designers make the right design decisions.
- Heuristics are “rules of thumb”.
 - Not hard and fast rules, but can have ramifications if violated.
- All heuristics may not apply for a given scenario.
In fact, they can be contradictory, at times, for a design.
 - There are always trade offs, and a designer will have to choose the one that best satisfies the needs.

June 14, 2014

Proprietary and Confidential

- 5 -



Design Heuristics:

How do we know whether right decisions are taken with respect to designing? This is where guidelines captured over the years through experience help in taking the right decisions.

These guidelines, also referred as “rules of thumb”, are not hard and fast rules. However, these can be thought of as “warning bells” if violated. Using these guidelines, appropriate changes can be brought about for removing the heuristic violation, wherever necessary

1.2: Design Heuristics

Introduction to Design Heuristics

- There are several design heuristics, applicable to
 - Objects and Classes
 - God Classes, Proliferation of Classes
 - Inheritance between classes

June 14, 2014

Proprietary and Confidential

~ 6 ~



Design Heuristics:

Arthur J. Riel has put together 60 design heuristics, and these are applicable for various aspects like Objects and Classes, God Classes and Proliferation of classes, Inheritance between classes etc.

We will see some of these on the subsequent slides.

1.2: Design Heuristics

Design Heuristics: Objects and Classes

- All data should be hidden within its class.
- Do not put implementation details such as common-code private functions into the public interface of a class.
- A class should capture one and only one key abstraction.
- You should keep related data and behavior in one place.
- You should spin off non-related information into another class (that is, non-communicating behavior).

June 14, 2014

Proprietary and Confidential

- 7 -



Design Heuristics: Objects and Classes:

The above slide lists some of the design heuristics related to objects and classes.

Data is operated upon by operations, so there is a direct dependency between them. Data is bound to change, so it is useful to isolate consequences of that change to within the same class by enforcing encapsulation. For example, if data type changes, operations too will need modification. By keeping data, and operations that act on the data together, maintenance becomes easier.

This heuristic aims to reduce the complexity of class interface. Implementation details are meant to be “service operations”, which merely factors code within class considering reusability and modularity. They are not expected to be directly used by clients of class and hence must remain private.

Key abstraction is usually defined as an element of the problem domain. This heuristic implies the need for a class to be cohesive.

Violating this would mean that more than one class is affected in case of change in data because data and behavior actually belong to the same key abstraction and should have been captured in the same class.

Look out for classes where a subset of methods operate on a proper subset of data. In this case, one must spin off the other subset of related data and operations into another class.

1.2: Design Heuristics

Design Heuristics: God Classes, Proliferation of Classes

- Distribute system intelligence horizontally as uniformly as possible, that is, the top-level classes in a design should share the work uniformly.
- Eliminate irrelevant classes from your design.
- Do not turn an operation into a class.

Design Heuristics: God Classes, Proliferation of Classes:

The above slide lists some of the design heuristics related to God classes and proliferation of classes.

Object oriented paradigm can go haywire if the design moves in the direction stated above. Poorly distributed system intelligence and creation of too many classes vis-à-vis the problem at hand are respectively referred to as “God Class” and “Proliferation of Classes”.

Especially when one moves from procedural to object oriented platforms, the tendency is to create God object which will do most of the work and leave smaller details to rest of the classes. It is said that one cannot get spaghetti code with OO systems, but one can get ravioli code instead! If spaghetti is pasta in long thin strings which makes it easy for them to get entangled (“spaghetti code” refers to unstructured code where control can jump from one point to another), ravioli is small pasta pouches containing cheese / vegetables / meat (“ravioli code” is characterized by number of small and loosely coupled software components). If raviolis become too many (“Proliferation” of classes), then there is a different kind of maintenance problem. In which of these multiple classes should changes be incorporated?

1.2: Design Heuristics

Design Heuristics: Inheritance Relationship

- Inheritance should only be used to model a specialization hierarchy.
- All data in a base class should be private, that is, should not use protected data.
- All abstract classes must be base classes.
- All base classes should be abstract classes.
- Factor the commonality of data and behavior as high as possible in the inheritance hierarchy.

June 14, 2014

Proprietary and Confidential

- 9 -



Design Heuristics: Inheritance Relationship:

The above slide lists some of the design heuristics related to inheritance relationship.

The first point actually compares inheritance with containership. It emphasizes that inheritance has to be used only in the cases where there is specialization hierarchy coming into picture.

“Favor composition over inheritance” comes from here!

The 2nd point hints at the fact that inheritance potentially violates encapsulation! When something is protected, it becomes available in the derived classes, thereby weakening data hiding.

There are recommendations on how abstract classes and base classes must be considered in design. It is ideal to have an abstract class sitting at the base of the hierarchy.

By factoring commonality of data and behaviour as high up in the hierarchy as possible, multiple derived classes can leverage the commonality.

1.3: Design Principles

Introduction to Design Principles

➤ Let us go through some design principles:

- Open-Closed Principle (OCP):
 - Software entities should be open for extension, but closed for modification (B. Meyer, 1988).
- Single Responsibility Principle (SRP):
 - A class should have only one reason to change.
- Interface Segregation Principle (ISP):
 - Many client-specific interfaces are better than one general purpose interface.

Some Design Principles in more details:

We have looked at some design heuristics. In the above slide, we now look at some key Object-Oriented design principles.

1.3: Design Principles

The Open-Closed Principle (OCP)

- Software entities should be open for extension and closed for modification (B.Mayer, 1988; quoted by R. Martin 1996).
 - Open for extension: The behavior of the module can be extended.
 - Closed for modification: The source code of the module is not allowed to change!
- How is this possible?
 - Abstraction is the key!

June 14, 2014

Proprietary and Confidential

- 11 -



The Open-Closed Principle (OCP):

Software modules that conform to the open-closed principle, exhibit two primary attributes:

They are Open to Extension: This implies that the behavior of the module can be extended. The system can be made to behave in new and different ways as requirements change, or to meet new applications.

They are Closed to Modifications: This implies that the source code of such a module remains intact. New codes are added to implement new and changed behavior.

At first glance, these two attributes appear to be contradicting each other. The normal way to extend behavior of a module is by making changes to that module. A module that does not lend itself to change, is said to have fixed behavior. So how does one change existing modules without changing the source code? You can use the principle of abstraction to develop modules that are open to extensions and simultaneously closed to modifications. You can create abstractions that are fixed, yet represent an unbounded group of possible behaviors. The abstractions are abstract base classes, and all the possible derivative classes represent the unbounded group of possible behaviors.

1.3: Design Principles
OCP – An Example

➤ Take the example of a graphic editor:

```
classDiagram
    class GraphicsEditor {
        drawRectangle()
        drawCircle()
        drawShape()
    }
    class Rectangle
    class Shape
    class Circle
    GraphicsEditor --> Rectangle : 
    GraphicsEditor --> Shape : 
    GraphicsEditor --> Circle : 
    Rectangle --> Shape : 
    Circle --> Shape : 
```

In this design, what happens if a new shape is added?

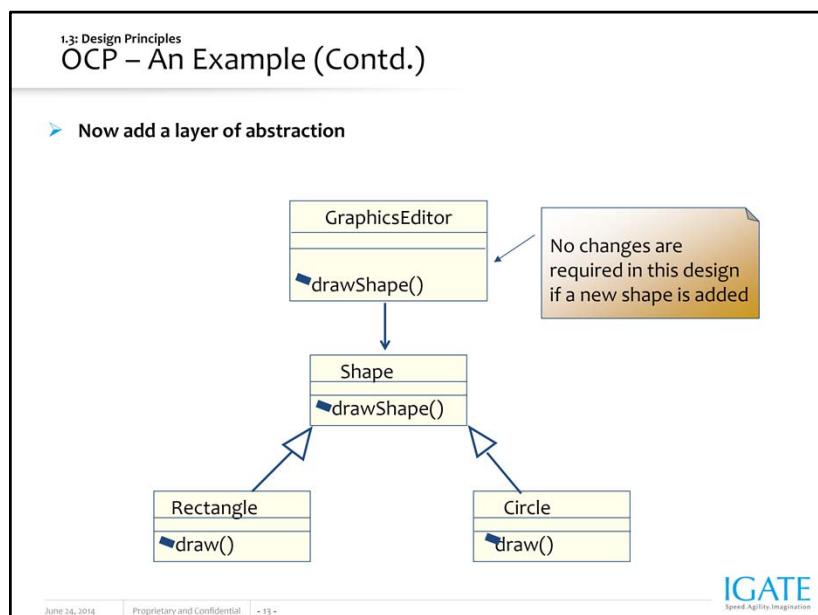
June 14, 2014 | Proprietary and Confidential | - 12 -

IGATE
Speed. Agility. Imagination.

OCP – An Example:

The above slide shows a graphic editor that can draw shapes, circles, and rectangles. However, when a new shape is added, the graphic editor has to be changed. This implies that the source code needs changing. Therefore this example does not conform to the OCP principle. The design cannot be closed against new kinds of shape.

The next slide shows a modified version of the example after incorporating OCP.



OCP – An Example (contd.):

Now, consider adding a layer of abstraction as shown in the above slide.

New shapes can extend the Shape class. So behavior can be extended without modification of GraphicsEditor class. The source remains intact.

This was a relatively simple example with a simple solution. In the real world, the Shape class would have many more methods. Still adding a new shape to the application is simple and requires the creation of the new derivative and the implementation of all its functions. The designs based on OCP incorporate changes by adding new codes rather than by changing existing codes. Hence one does not encounter the cascading effect seen otherwise.

It is important to note that no application can be 100% closed. The closure cannot be complete. Hence designers look for strategic closure. From a designers angle, this situation requires deciding on the kind of changes against which you want to close your design. This calls for a certain degree of intuition and experience. An experienced designer has his or her finger on the pulse of the industry and the user. He or she can normally foresee the probability of different kinds of changes and design accordingly.

1.3: Design Principles

Single Responsibility Principle

- The Single Responsibility Principle (SRP) states that a class should have only one reason to change.
- It is also known as “high cohesion”.

June 14, 2014

Proprietary and Confidential

- 14 -

IGATE
Speed. Agility. Imagination

Single Responsibility Principle (SRP):

In this context, a responsibility is considered to be one reason to change. This principle states that if we have two reasons for a class to change, then we have to split the functionality into two classes. Each class will handle only one responsibility and in future if we need to make one change, then we are going to make it in the class which handles it. When we need to make a change in a class having more responsibilities, the change might affect the other functionality of the classes.

Cohesion is sticking or working together, that is, the state or condition of joining or working together to form a united whole, or the tendency to do this. A class should be cohesive, that is, the class should have only a single purpose to live and all its methods should work together to help achieve this goal.

The Single Responsibility principle is a simple and intuitive. However, in practice it is sometimes hard to get it right.

1.3: Design Principles

SRP – An Example

Class A

compileReport()
printReport()

This class has two different reasons to change => Split into two classes

June 14, 2014 | Proprietary and Confidential | - 15 -

IGATE
Speed. Agility. Imagination.

SRP – An Example:

As an example, consider a class that compiles and prints a report. Such a class can be changed for two reasons.

First, the content of the report can change.

Second, the format of the report can change.

These two things change for very different causes – one substantive, and one cosmetic. The SRP says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes.

It is important to keep a class focused on a single concern because it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, then there is a greater danger that the printing code will break if it is part of the same class.

When a class has more than one responsibility (that is, reason to change), these responsibilities are coupled. This scenario makes the class more difficult to understand, more difficult to change, and more difficult to reuse. Cohesion should also be applied at the method level, and for the exact same reasons.

The challenge with SRP is getting the granularity of a responsibility right. Sometimes, it is easier to see the responsibilities are unrelated. But more often, it needs thorough thinking!

One last point regarding SRP is that if you cannot separate the responsibilities into separate classes, then at least consider separating them to different interfaces.

1.3: Design Principles

Interface Segregation Principle

- Many client-specific interfaces are better than one general-purpose interface.
- In other words,
 - Clients should not be forced to depend upon interfaces that they do not use.

June 14, 2014

Proprietary and Confidential

• 16 •

IGATE
Speed. Agility. Imagination

Interface Segregation Principle (ISP):

Interface Segregation Principle (ISP) deals with designing “cohesive” interfaces and avoiding “fat” interfaces. It focuses on the cohesiveness of interfaces with respect to the clients that use them. The idea here is that each client may use a particular object or subsystem in a different way.

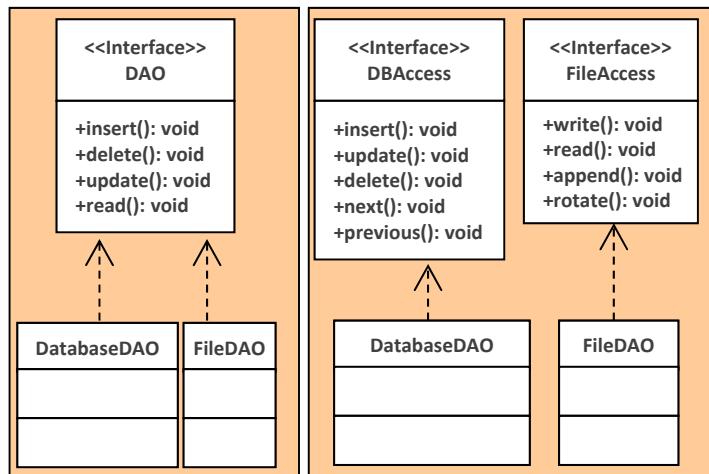
ISP states that clients should not be forced to implement interfaces they do not use. Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one sub-module.

Why should “fat” interfaces be avoided?

Each client depends on the single class interface. Hence there is an inadvertent coupling between the clients.

How is the client coupling harmful?

Suppose a client needs that additional functionality be added to the single class interface. When this functionality is added to the interface, every other client must change to support the functionality even though none of them need it. Thus one change of a client forces the change to propagate throughout the system. This situation, in turn, can result in time consuming code maintenance and hard to locate bugs.

ISP – An Example:

Which of these designs follow ISP?

- Imagine that in your application you are required to write some **Data Access Objects (DAOs)**. These data objects should support a variety of data sources. Let us consider that the two main data sources are file and database. You must be careful enough to come up with an **interface-based design**, where the implementation of data access can be varied without affecting the client code using your DAO object.
- What happens if the data source is read-only?
 - The methods for inserting and updating data are not needed. On the other hand, if the *DAO* object should implement the *DAO interface*, then it will have to provide a null implementation for those methods defined in the *interface*. This is still acceptable, but the design is gradually going wrong.
- What if there is a need to rotate the file data source to a different file once a certain amount of data has been written to the file?
 - That will require a separate method to add to the *DAO interface*.
- When a single interface is designed to support different groups of behaviors, then they are, by virtue, inherently poorly designed, and are called **Fat interfaces**. They are called **Fat** because they grow enormously with each additional function required by clients using that interface. Thus, for the problem with the Data Access Objects, follow the Interface Segregation Principle, and separate the interfaces based on the behaviors. The database access classes and file access classes should subscribe to two separate interfaces.

Summary

➤ **In this lesson, you have learnt:**

- OO Design Principles help us understand what constitutes a good design!



Review Question

- Question 1: Related data and behavior should be kept in same class.
(True/False)
- Question 2: A God class should centrally control the application.
(True/False)
- Question 3: Favor ___ over ___.
- Question 4: ___ principle says modules should be open for extension but closed for modification.
- Question 5: ___ principle discourages use of Fat interfaces.



Introduction To Design Principles and Patterns

Introducing Design Patterns

June 24, 2014

Proprietary and Confidential

- 1 -



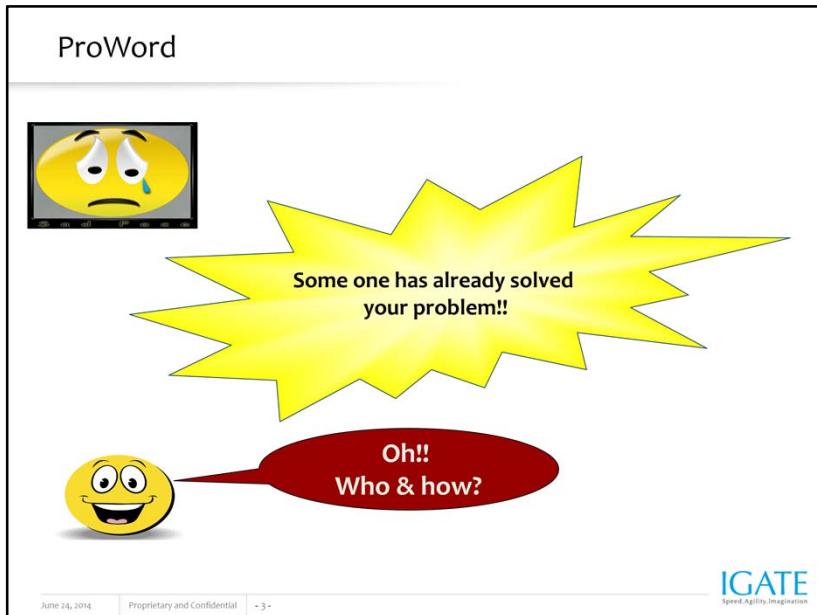
Lesson Objectives

- **In this lesson, you will learn:**
 - What is a design pattern?
 - Why Design Patterns
 - Design Pattern Drawbacks
 - Design Pattern Categories



Lesson Objectives:

In this lesson, we will understand the what and why of design patterns, see how design patterns are classified; and have a look at some of these design patterns.



Is it possible that solutions to some of our problems already exist??

ProWord



June 24, 2014

Proprietary and Confidential

- 4 -

IGATE
Speed. Agility. Imagination.

Yes... design patterns provide just that!

2.1: Design Pattern

Concept of Design Pattern

- Design Pattern is a solution to a problem in a context.
- Pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.”
- Design Patterns are “reusable solutions to recurring problems that we encounter during software development.”

June 24, 2014

Proprietary and Confidential

- 5 -



What is a Design Pattern?

“A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever using it the same way twice.” Patterns can be applied to many areas of human endeavor, including software development.

2.1: Design Pattern

Rationale behind using Design Patterns

- Patterns enable programmers to “... recognize a problem and immediately determine the solution without having to stop and analyze the problem first.”
- They provide reusable solutions.
- They enhance productivity.

June 24, 2014

Proprietary and Confidential

- 6 -



Why Design Patterns?

Designing object-oriented code is hard, and designing reusable object-oriented software is even harder.

Patterns enable programmers to “... recognize a problem and immediately determine the solution without having to stop and analyze the problem first.”

Well structured object-oriented systems have recurring patterns of classes and objects.

The patterns provide a framework for communicating complexities of OO design at a high level of abstraction. Bottom line is productivity.

Experienced designers reuse solutions that have worked in the past.

Knowledge of the patterns that have worked in the past allows a designer to be more productive and the resulting design to be more flexible and reusable.

2.1: Design Pattern

Drawbacks of Design Patterns

- Listed below are some of the drawbacks of design patterns:

- Patterns do not allow direct code reuse.
- Patterns are deceptively simple.
- Design might result into Pattern overload.
- Patterns are validated by experience and discussion rather than by automated testing.

June 24, 2014

Proprietary and Confidential

- 7 -



Note:

Design patterns have drawbacks too! Besides the drawbacks mentioned in the slide, Integrating patterns into a software development process is a human-intensive task.

2.1: Design Pattern

Broad level Categories of Design Patterns

- **Design Patterns can be broadly classified as:**

- Fundamental patterns
- Creational patterns
- Structural patterns
- Behavioral Patterns

June 24, 2014

Proprietary and Confidential

- 8 -



Classification of GOF Design Patterns:

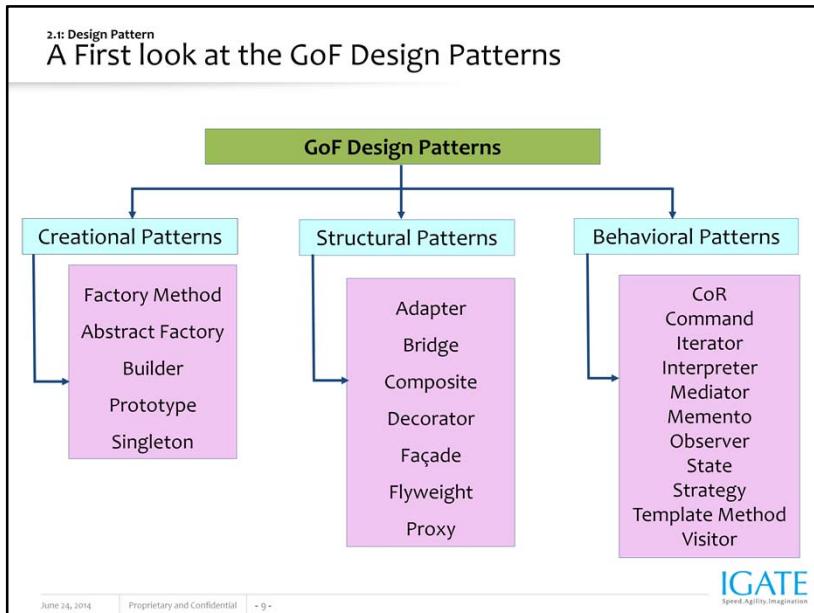
The Gang of Four (GoF) Design Patterns can be broadly classified as :

Fundamental Patterns: They are the building blocks for the other three categories of Design Patterns.

Creational Patterns: They deal with creation, initializing, and configuring classes and objects.

Structural Patterns: They facilitate decoupling interface and implementation of classes and objects.

Behavioral Patterns: They take care of dynamic interactions among societies of classes and objects. They also give guidelines on how to distribute responsibilities amongst the classes.



Note:

There are 23 GOF Design Patterns.

They have been classified as shown on the slide. We shall see some more details with examples for some of these design patterns in the next lesson.

Another classification for design patterns is class based or object based. Class based Design Patterns uses “inheritance” as the basic principle whereas the object based patterns use “composition”.

We have seen, “Favor Composition over Inheritance”.

Note that in design patterns, “composition” is being favored as most of the Design Patterns are “Object-based”. The class based design patterns are Factory Method, Adaptor, Interpreter and Template Method.

Summary

➤ **In this lesson, you have learnt:**

- Concept of Design Pattern
- Rationale behind using Design Patterns
- Drawbacks of Design Patterns
- Classification of Design Patterns



Review Question

- Question 1: Design pattern is a solution to ___ within a particular ___.
- Question 2: Name different types of GOF Design Patterns.



Introduction To Design Principles and Patterns

Examples of Design Patterns

Lesson Objectives

- In this lesson, you will learn some examples of design patterns:

- Fundamental Patterns: Delegation Pattern, Interface Pattern, Abstract Superclass Pattern
- Creational Patterns: Factory Method, Singleton
- Structural Patterns: Adapter, Façade
- Behavioural Patterns: State, Strategy, Template Method



Lesson Objectives:

This lesson provides some examples for design patterns.

3.1: Examples of Fundamental Patterns

What is a Fundamental Design Pattern

- These are widely used by other patterns or are frequently used in a large number of programs.
- Fundamental patterns are further classified as:
 - Delegation Pattern
 - Interface Pattern
 - Abstract Superclass
 - Interface and abstract class
 - Immutable Pattern
 - Marker Interface Pattern

June 24, 2014

Proprietary and Confidential

**Note:**

The fundamental design patterns are the building blocks of the other design patterns.

Fundamental Design Patterns:

Delegation Pattern: It is a way of extending and reusing a class using composition rather than inheritance.

Interface Pattern: This pattern facilitates the design principle – “Program to Interface rather than Implementation”.

Abstract Superclass: It ensures consistent behavior of conceptually related classes by giving them a common abstract superclass.

Interface and abstract class: It is a combination of Interface and Abstract superclass Patterns.

Immutable Pattern: This pattern prevents the object from changing its state information after it is constructed thereby eliminating the issues related to concurrent access of the object.

Marker Interface: It is used to mark an object to be part of a particular group thereby allowing other objects to treat them in a uniform way.

We will see Delegation, Interface, and Abstract Superclass in some more detail.

3.1: Examples of Fundamental Patterns

Delegation Pattern

- A class outwardly expresses certain behavior, but in reality delegates responsibility for implementing that behaviour to another class
- Used for extending a class behavior
 - As against inheritance where operations are inherited, delegation involves a class calling another class's operation
 - Suitable for “Is a role played by” relationship
- Helps in reducing coupling in the System

Delegation Pattern:

The delegation pattern is a technique where an object outwardly expresses certain behavior but in reality delegates responsibility for implementing that behavior to an associated object in an Inversion of Responsibility.

Inheritance is a common way to extend and reuse the functionality of a class. Delegation is a more general way for extending a class's behavior that involves a class calling another class's methods rather than inheriting them.

Inheritance is useful for capturing “is-a-kind-of” relationships because of their static nature. Delegation is suitable for “is-a-role-played by” relationship.

Suppose it is found that a class attempts to hide a method or variable inherited from a super class or from other classes. Then that class should not inherit from the super class. There is no effective way to hide methods or variables inherited from a super class. However, it is possible for an object to use another object's methods and variables while ensuring that it is the only object with access to the other object. This accomplishes the same thing as inheritance but uses dynamic relationships that can change over time.

3.1: Examples of Fundamental Patterns

What Delegation Pattern implies

```

classDiagram
    class Delegator {
        -theDelegate : Delegate
        +delegatedOperation()
        +normalOperation()
    }
    class Delegate {
        *delegatedOperation()
    }
    Delegator "2" --> "1" Delegate : theDelegate
  
```

```

Delegator.h
#include "Delegate.h"

class Delegator {
public:
    void delegatedOperation();
    void normalOperation();
private:
    Delegate theDelegate;
};

Delegator.cpp
#include "Delegator.h"

void Delegator::delegatedOperation()
{
    theDelegate.delegatedOperation();
}

void Delegator::normalOperation()
{
    /* Code as usual */
}
  
```

```

//Delegation is done to this class
Delegate.h
class Delegate {
public:
    void delegatedOperation();
};

Delegate.cpp
void Delegate::delegatedOperation()
{
/* Actual Implementation will be here
*/
}
  
```

June 24, 2014 | Proprietary and Confidential | - 5 -

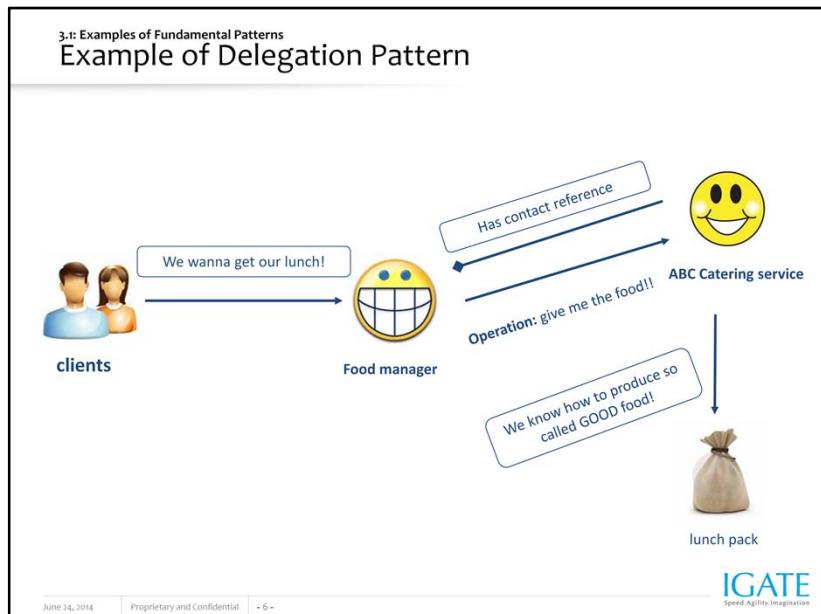
IGATE
Speed. Agility. Imagination.

Structure of Delegation Pattern:

You can use delegation to reuse and extend the behavior of a class. You do this by writing a new class (the delegator) that incorporates the functionality of the original class by using an instance of the original class (the delegatee) and calling its methods.

Delegation is more general purpose than inheritance. Any extension to a class that can be accomplished by “inheritance” can also be accomplished by “delegation”.

The implementation of delegation is very straightforward, for it simply involves the acquisition of a reference to an instance of the class to which you want to delegate and call its methods. The best way to ensure that a delegation is easy to maintain is to make its structure and purpose explicit.



Example of Delegation Pattern:

In real-life, the Delegation Pattern is used almost everywhere. The example on the slide shows how the food manager delegated the request for lunch to a catering service, who would actually prepare the lunch and deliver it to the client.

3.1: Examples of Fundamental Patterns

Interface Pattern

- An interface encapsulates a coherent set of services and attributes, without explicitly binding this functionality to that of any particular object or code.
- A class implementing the interface provides behaviour for the services
- Supports “Program to an interface, not to an implementation”

June 24, 2014

Proprietary and Confidential

- 7 -



Interface Pattern:

Interfaces are “more abstract” than classes since they do not say anything at all about representation or code. All they do is describe public operations.

You can keep a class that uses data and services provided by instances of other classes independent of those classes by having it access those instances through an interface.

3.1: Examples of Fundamental Patterns

What Interface Pattern implies

```
+ connect( )
+ read( )
+ display( )
```

//This is the interface
public interface
theInterface {
 public void connect();
 public void read();
 public void display();
}

```
//The implementer class provides the  
implementation  
public class theImplementer implements  
theInterface {  
    public void connect() {  
        //Code comes here  
    }  
    public void read() {  
        //Code comes here  
    }  
    public void display() {  
        //Code comes here  
    }  
}
```

Note: In C++, since there is no construct available for Interface, closest implementation will be to use Pure Virtual Functions

June 24, 2014 | Proprietary and Confidential | - 8 -

IGATE
Speed. Agility. Imagination.

The interface is only specifying the services. It is the implementer which needs to provide the actual behaviour through the implementation of these services.

3.1: Examples of Fundamental Patterns

Example of Interface Pattern

- A Uniform structure for “Street Address” to maintain address of employee/customer/vendor.

```

classDiagram
    class AddressPanel {
        <<AddressPanel>>
    }
    class DataClass {
        <<DataClass>>
    }
    interface AddressIF {
        <<AddressIF>>
        getAddress1()
        setAddress1()
        getAddress2()
        setAddress2()
        getCity()
        setCity()
        getState()
        setState()
        getPostalCode()
        setPostalCode()
    }
    AddressPanel "1" --> "1" AddressIF : uses
    DataClass "*" --> AddressIF : 
  
```

June 24, 2014 | Proprietary and Confidential | - 9 -

IGATE
Speed. Agility. Imagination.

Interface Pattern:

For example, suppose you are writing an application to purchase goods for a business. Your program will need to be informed of such entities as vendors, freight companies, receiving locations, and billing locations. One thing these have in common is that they all have a street address. These street addresses will be displayed in different parts of the user interface. You will want to have a class for displaying and editing street addresses so that you can reuse it wherever there is an address in the user interface. We will call that class AddressPanel.

You want AddressPanel objects to be able to get and set address information in a separate data object. This raises the question of what instances of the AddressPanel class can assume about the class of the data objects that they will use. Clearly, you will use different classes to represent vendors, freight companies, and the like. If you program in a language that supports multiple inheritance, like C++, you can arrange for the data objects that instances of AddressPanel use to inherit from an address class in addition to the other classes they inherit from. If you program in a language like Java that uses a single inheritance object model, then you must explore other solutions.

You can solve the problem by creating an address interface. Instances of the AddressPanel class would then simply require data objects that implement the address interface. They would then be able to call the accessor methods of that object to get and set its address information. Using the indirection that the interface provides, instances of the AddressPanel class are able to call the methods of the data object without having to be aware of what class it belongs to. Here is a class diagram showing these relationships.

Here in this example we have seen that the AddressPanel class is independent or does not have to specifically assume the data object it is going to deal with. In turn, it will work upon the data object with the help of the AddressIF interface which is actually implemented by the data classes.

3.1: Examples of Fundamental Patterns

Abstract Superclass

- **Common behaviour of related classes can be extracted and consolidated into an abstract Superclass**
 - Helps in ensuring that consistent behaviour is provided to the related set of classes
- **Concrete classes extend the abstract Superclass**
 - They use the “common” behaviour as described in the abstract superclass
 - They provide behaviour for “variant” operations

Abstract Superclass:

The abstract Superclass helps in ensuring that logic common to related classes is implemented consistently for each class. It helps avoid the maintenance overhead of redundant code and makes it easy to write related classes.

Concrete classes then extend the abstract superclass. The commonality of behaviour as defined in the abstract superclass is directly used by the concrete classes. For the operations that are defined as abstract in the superclass, the concrete class provides the behaviour.

3.1: Examples of Fundamental Patterns

What Abstract Superclass implies

```
classDiagram
    abstractClass {
        +abstractOperation()
        +concreteOperation()
    }
    concreteClass1
    concreteClass2
    abstractClass <|-- concreteClass1
    abstractClass <|-- concreteClass2
```

//This is the abstract class
abstractClass.h
class abstractClass {
public:
/* Implementation of abstractOperation is in
concrete classes */
void abstractOperation();
void concreteOperation();
};

abstractClass.cpp
void abstractClass::concreteOperation()
{ /* Implementation is here */
}

class concreteClass1 : public abstractClass {
/* Implement abstract operation */
/* Plus other code */
};

class concreteClass2 : public abstractClass {
/* Implement abstract operation */
/* Plus other code */
};

June 24, 2014 | Proprietary and Confidential | - 11 -

IGATE
Speed. Agility. Imagination.

The abstract class defines the abstract operations, which are implemented by the concrete classes that extend the abstract class.

3.1: Examples of Fundamental Patterns

Example of Abstract Superclass

- In Typical Hierarchies like Account, Customer etc., the base class would be the Abstract Superclass

Abstract Superclass Pattern:

We have seen examples like Savings and Current Account extending from a base “Account” superclass which is abstract. Similarly if there are different types of customers, an abstract “Customer” Superclass can be at the base of the Customer Class Hierarchy.

3.2: Examples of Creational Patterns

What is a Creational Design Pattern

- **Creational Design Patterns** are design patterns that deal with object creation mechanisms.
- They help to create objects in a manner suitable to the situation.
- They provide guidance on how to create objects when their creation requires making decisions.

June 24, 2014

Proprietary and Confidential

- 13 -



Introduction to Creational Patterns:

Creational Patterns provide guidelines on creation, configuration, and initialization for objects.

“Decisions typically involve dynamically deciding which class to instantiate or which objects an object will delegate responsibility to.”

3.2: Examples of Creational Patterns

Different Creational Design Patterns

- **Creational Design Patterns** are further classified as:

- Factory method
- Abstract Factory
- Prototype
- Builder
- Singleton

June 24, 2014

Proprietary and Confidential

- 14 -



Different Creational Patterns:

Creational Design Patterns are further classified as:

Factory Method: It creates objects without specifying the exact object to create.

Abstract Factory: It groups object factories that have a common theme.

Prototype: It creates objects by cloning an existing object.

Builder: It constructs complex objects by separating construction and representation.

Singleton: It restricts object creation for a class to only one instance.

We will see Factory and Singleton in some more detail.

3.2: Examples of Creational Patterns

Factory Method Pattern

- Defines an interface for creating an object, but lets subclasses decide which class to instantiate.
 - Factory Method lets a class defer instantiation to subclasses.
- Use of Factory pattern promotes loose coupling.

June 24, 2014

Proprietary and Confidential

- 15 -



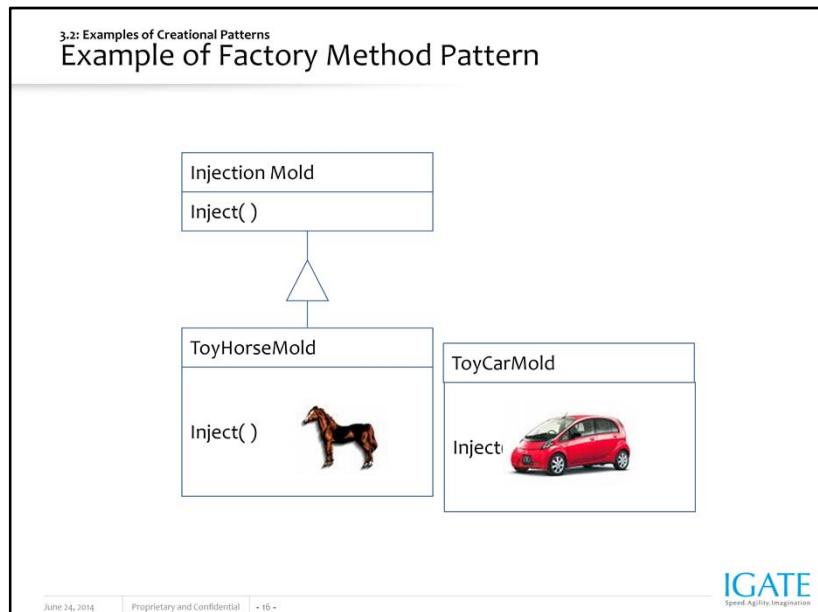
Factory Method:

You can define an interface for creating an object, however, let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

It is also called as a Factory Pattern since it is responsible for “manufacturing” an Object.

The Factory Method pattern is an object-oriented design pattern. Like other creational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. Factory Method handles this problem by defining a separate method for creating the objects, which can then be overridden by subclasses to specify the derived type of the product that will be created.

They promote loose coupling by eliminating the need to bind application specific classes into the code.



June 24, 2014

Proprietary and Confidential

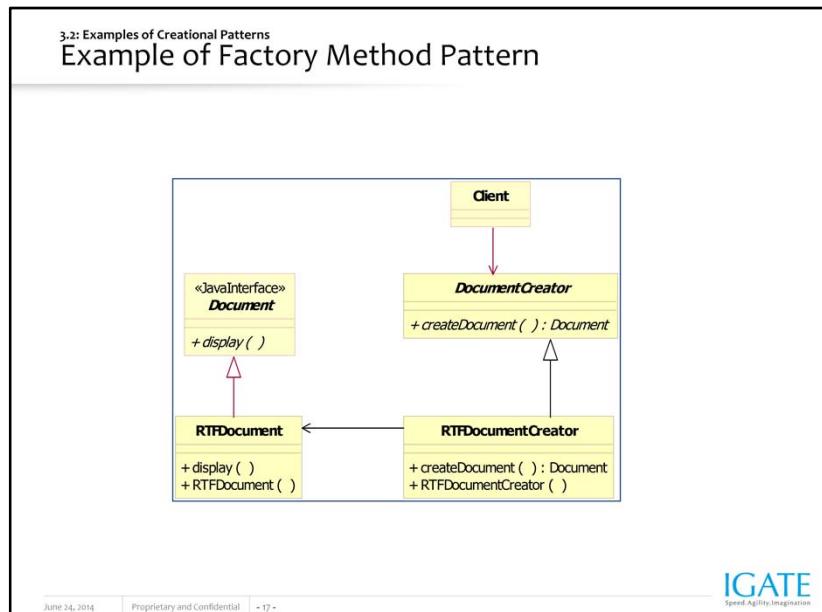
- 16 -

IGATE
Speed. Agility. Imagination.

Example of Factory Method Pattern:

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate.

Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.



June 24, 2014

Proprietary and Confidential

- 17 -

IGATE
Speed. Agility. Imagination.

In this example, the factory pattern is being used to create documents of different types – while RTF is an example shown, other types could be Plain text, HTML etc. Here, **Document** is the interface of the objects the factory method creates. **RTFDocument** implements this interface. The **DocumentCreator** declares the factory method which returns an object of type **Document**. The **RTFDocumentCreator** overrides the factory method to return an instance of **RTFDocument**. Similarly, we could have classes like **HTMLDocumentCreator** and **PlainTextDocumentCreator** overriding the factory methods to return specific concrete instances of the desired objects.

3.2: Examples of Creational Patterns
Singleton Pattern

- The Singleton pattern ensures that only one instance of a class is created.
- All objects that use an instance of that class use the same instance.

June 24, 2014

Proprietary and Confidential

- 18 -



Singleton Pattern:

There are cases in programming where you need to ensure that there can be one and only one instance of a class which is used by the application.

The Singleton pattern ensures that a class has only one instance, and provides a global point of access to it.

It is important for some classes to have exactly one instance.

Although there can be many printers in a system, there should be only one printer spooler.

There should be only one File system and one Window manager.

A digital filter will have one A/D converter.

An accounting system will be dedicated to serving one company.

How do we ensure that a class has only one instance and that the instance is easily accessible?

A global variable makes an object accessible. However, it does not keep you from instantiating multiple objects.

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

3.2: Examples of Creational Patterns

Example of Singleton Pattern

LogFile
- uniqueInstance : LogFile* = 0
- LogFile()
+ GetUniqueInstance()

```
LogFile.h
class LogFile {
public:
    /* STATIC operation implements the logic for returning the
     * unique instance of the Singleton. It creates the unique
     * instance if it does not already exist.*/
    static LogFile* GetUniqueInstance();

private:
    /* Notice that the default constructor of the Singleton
     * class is private, so that it CANNOT be accessed outside the
     * Singleton class.*/
    LogFile();

    /* This attribute stores the Singleton's unique instance.*/
    static LogFile* uniqueInstance;
};
```

```
LogFile.cpp
#include "LogFile.h"

LogFile* LogFile::GetUniqueInstance()
{
    if (uniqueInstance == 0)
    {
        uniqueInstance = new LogFile();
    }
    return uniqueInstance;
}

LogFile::LogFile()
{
}

LogFile* LogFile::uniqueInstance = 0;
```

IGATE
Speed. Agility. Imagination.

Note the following

A private constructor is used to restrict the application from creating multiple instances of the class

A static instance variable makes the single instance globally available

A getUniqueInstance method is used to ensure that an instance is created the first time round and the same instance is used each time

3-3: Examples of Structural Patterns

What are Structural Design Patterns

- **Structural patterns describe how objects and classes can be combined to form larger structures**
 - Structural class patterns use inheritance to compose interfaces or implementations
 - Structural object patterns describe how objects can be organized to work with each other to form a larger structure

June 24, 2014

Proprietary and Confidential

- 20 -



Structural Patterns:

Structural Patterns describe how objects and classes can be combined to form larger structures.

Class patterns describe abstraction with the help of “inheritance” and how it can be used to provide more useful program interface.

Object patterns, on other hand, describe how objects can be associated and composed to form larger, more complex structures.

Structural Patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together.

3.3: Examples of Structural Patterns

Different Structural Design Patterns

➤ **Structural Design patterns are further classified as:**

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

June 24, 2014

Proprietary and Confidential

- 21 -



Different Structural Patterns:

Adapter: It allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.

Bridge: It decouples an abstraction from its implementation so that the two can vary independently.

Composite: It composes one-or-more similar objects so that they can be manipulated as one object.

Decorator: It dynamically adds / overrides behavior in an existing method of an object.

Façade: It provides a simplified interface to a large body of code.

Flyweight: It reduces the cost of creating and manipulating a large number of similar objects.

Proxy: It provides a placeholder for another object to control access, reduce cost, and reduce complexity.

We will understand Adapter and Façade in some detail.

3-3: Examples of Structural Patterns

Adapter Pattern

- Adapter Pattern converts interface of one class into interface expected by the client.
- It allows two unrelated interfaces to work together.
- It allows reusability of older functionality.
- Adapter can be implemented in two ways :
 - By inheritance (Class Adapter pattern)
 - By object composition (Object Adapter pattern)

Adapter Design Pattern:

The Adapter Design Pattern is a type of design pattern that is used for converting the interface of a class into an interface that its clients expect to see. This pattern allows incompatible interfaces to work together.

The adapter design pattern (often referred to as the “wrapper pattern” or simply a “wrapper”) adapts one interface for a class into one that a client expects. An adapter allows those classes to work together that normally cannot due to their incompatible interfaces. This is done by wrapping its own interface around that of an already existing class.

The adapter is also responsible for handling any logic necessary to transform data into a form that is useful for the consumer.

For instance, if multiple Boolean values are stored as a single integer but your consumer requires a “true” / “false”, the adapter will be responsible for extracting the appropriate values from the integer value.

The adapter pattern is useful in situations where an already existing class provides some or all of the services you need but does not use the interface you need.

3.3: Examples of Structural Patterns

Example of Adapter Pattern

The diagram illustrates the Adapter pattern with three components:

- European Wall Outlet:** A grey wall outlet with two vertical slots and one central horizontal slot.
- AC Power Adapter:** A grey adapter with two pins inserted into the European outlet's slots.
- Standard AC Plug:** A grey plug with two pins, labeled "The US laptop expects another interface."

Annotations explain the roles:

- "The European wall outlet exposes one interface for getting power." (points to the outlet)
- "The adapter converts one interface into another." (points from the outlet to the plug)
- "The US laptop expects another interface." (points to the plug)

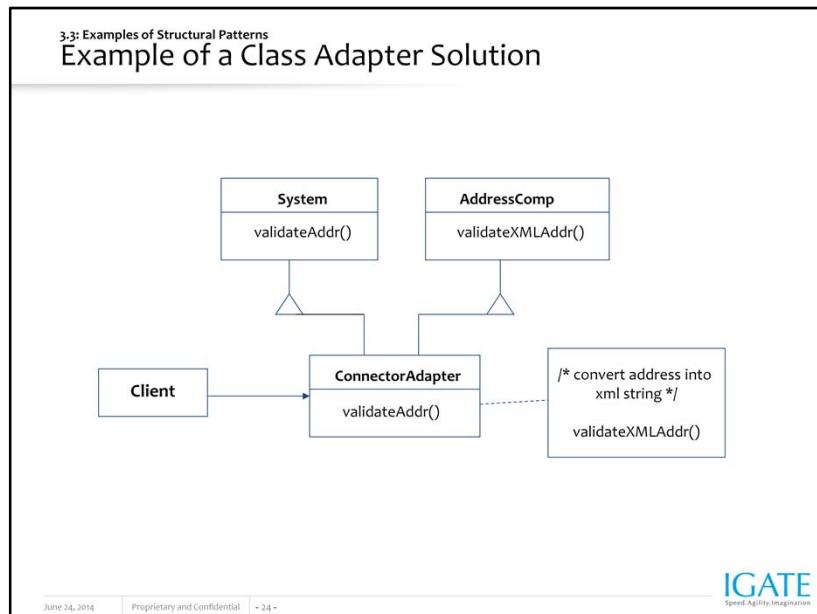
June 24, 2014 | Proprietary and Confidential | - 23 -

IGATE
Speed. Agility. Imagination.

Example of Adapter Pattern:

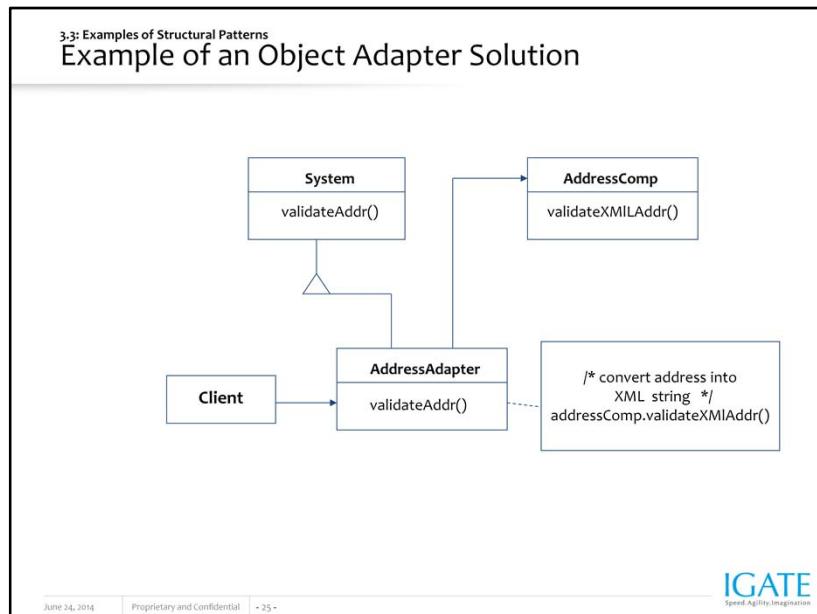
The Adapter pattern allows incompatible classes to work together by converting the interface of one class into an interface expected by the clients.

Socket wrenches provide an example of the Adapter. The adapter sits in between the plug of your laptop (US-made) and the European AC outlet. The adapter changes the interface of the outlet into one that your laptop expects.



Let us look at the requirement of a system validating Address using a third party address component. The client expects an interface for validating address in the form of a string. However, the address component needs the address in the form of XML string.

The AddressAdapter class will bridge this gap, and provide a interface to the system in form of “validateAddr()” method that will expect the address as a normal string. It will generate an XML out of it and send this XML string as input to the validateXMLAddr() method provided by the AddressComp. Thus you can say, though the interfaces of the system and address component were incompatible, the adapter class has made them work together. In case of Class adapter solution, the adapter class will inherit the implementations from both “System” and “AddressComp”. It will override the implementation of “System” (target) and call the implementation inherited from “AddressComp” (adaptee). Before calling this implementation, it will perform certain activities that are required for adapting (in our case it will convert the string address into xml format).



In this solution also, the AddressAdapter class has made the two incompatible interfaces work together.

In case of object adapter solution, the AddressAdapter inherits implementation from System (target) only. Subsequently, it will override this implementation to create an object of “AddressComp” (adaptee) and call the appropriate method on it. In this case also, before calling the method of the adaptee, it will need to perform the activities that are required for adapting.

3.3: Examples of Structural Patterns

Facade Pattern

- Facade Pattern provides a unified interface to a set of interfaces in a subsystem.
- Facade defines a higher-level interface that makes the subsystem easier to use.
- You can wrap a complicated subsystem with a simpler interface.
- They are normally Singleton as only one Facade object is required.

Facade Pattern:

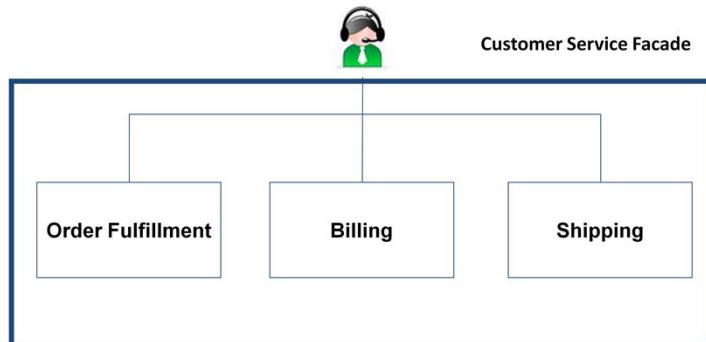
Facade encapsulates a complex subsystem within a single interface object. This reduces the learning curve that is necessary to successfully leverage the subsystem. It also promotes decoupling the subsystem from its potentially many clients. On the other hand, if the Facade is the only access point for the subsystem, it limits the features and flexibility that are needed by “power users”.

The Facade Design Pattern is used to provide a high-level interface that makes the subsystem easier to use. It helps create a unified interface to a set of interfaces in the subsystem.

Facade objects are often Singletons because only one Facade object is required.

3.3: Examples of Structural Patterns

Example of Facade Pattern



June 24, 2014

Proprietary and Confidential

- 27 -

IGATE
Speed. Agility. Imagination.

Example of Façade Pattern:

Consumers encounter a Facade while ordering from a catalog. The consumer calls one telephone number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department.

3-3: Examples of Structural Patterns

What are Structural Design Patterns

- Home Theater System is assembled with a DVD player, a projection video system, CD Player, surround sound, and a popcorn popper.



June 24, 2014

Proprietary and Confidential

- 28 -

IGATE
Speed. Agility. Imagination.

Here the Home Theatre System is the Façade which knows which subsystem the client request has to be passed on to.

3.4: Introduction to Behavioral Patterns

What is a Behavioral Design Pattern

- **Behavioral patterns are those design patterns which deal with interactions between the objects.**
 - Behavioral patterns are concerned with the assignment of responsibilities between objects, or encapsulating behavior in an object and delegating requests to it.

June 24, 2014

Proprietary and Confidential

- 29 -



Behavioral Patterns:

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not only patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow, which is difficult to follow at run-time. These patterns shift your focus away from flow of control, and let you concentrate just on the way objects are interconnected. Behavioral class patterns use inheritance to distribute behavior between classes. Behavioral object patterns use “object composition” rather than “inheritance”.

There are 11 different behavioral design patterns. They are Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor.

3.4: Introduction to Behavioral Patterns

Different Behavioral Design Patterns

➤ Here is a list of the different behavioral design patterns:

- Chain of Responsibility (COR)
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Different Behavioral Design patterns:

Chain of responsibility: It delegates commands to a chain of processing objects.

Command: It creates objects which encapsulate actions and parameters.

Interpreter: It implements a specialized language.

Iterator: It accesses the elements of an object sequentially without exposing its underlying representation.

Mediator: It allows loose coupling between classes by being the only class that has detailed knowledge of their methods.

Memento: It provides the ability to restore an object to its previous state (undo).

Observer: It is a publish/subscribe pattern which allows a number of observer objects to see an event.

State: It allows an object to alter its behavior when its internal state changes.

Strategy: It allows one of a family of algorithms to be selected on-the-fly at runtime.

Template method: It defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.

Visitor: It separates an algorithm from an object structure by moving the hierarchy of methods into one object.

3.4: Introduction to Behavioral Patterns

State Pattern

- State Pattern allows an object to alter its behavior when its internal state changes. The object appears to change its class.
- It uses polymorphism to define different behaviors for different states of an object.

June 24, 2014

Proprietary and Confidential

- 31 -



State Pattern:

State pattern is a behavioral type design pattern which is widely used in different applications. It is especially used in 3D-Graphics applications and applications for devices. In object-oriented design, object can change its behavior based on its current state. A state pattern design implements state and behavior of an object. By using state pattern, we can reduce the complexity in handling different states of an object. This helps to easily maintain code in future.

A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state.

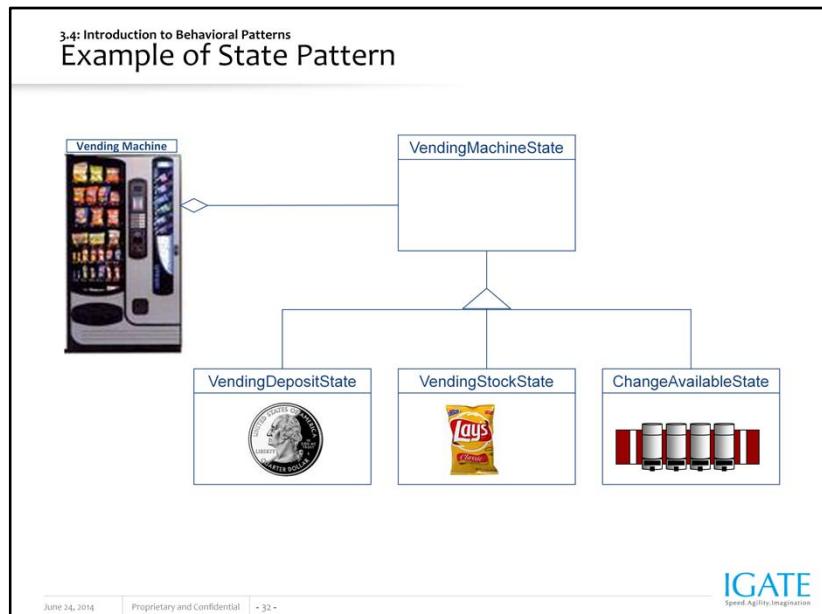
Alternatively, an application is characterized by large and numerous case statements that vector flow of control, based on the state of the application.

The State pattern does not specify the location where the state transitions will be defined. There are two choices:

The “context” object

Each individual State derived class

The advantage of the latter option is ease of adding new State derived classes. The disadvantage is that each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses.



Example of State Pattern:

The State pattern allows an object to change its behavior when its internal state changes.

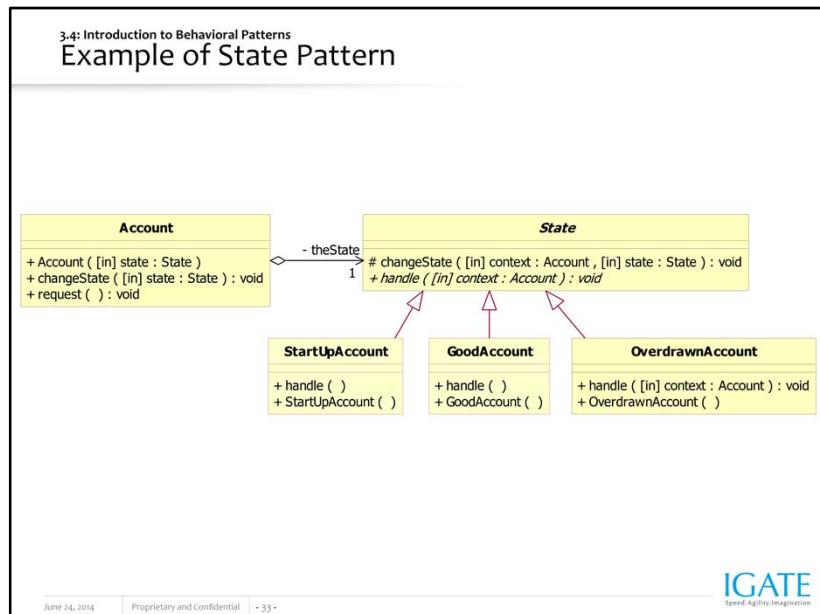
This pattern can be observed in a vending machine.

Vending machines have states based on:

- The inventory
- The amount of currency deposited
- The ability to make change
- The item selected, etc.

When currency is deposited and a selection is made, a vending machine will do either of the following:

- Deliver a product and no change
- Deliver a product and change
- Deliver no product due to insufficient currency on deposit
- Deliver no product due to inventory depletion



An account holder has an account with ICICI bank and the account behaves differently depending on its balance state.
The difference in behavior is delegated to State objects.

3.4: Introduction to Behavioral Patterns

Strategy Pattern

- Strategy Pattern defines a family of algorithms, encapsulate each one, and makes them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.
- It looks similar to state pattern but intent is to encapsulate interchangeable behavior which is not state-based.

June 24, 2014

Proprietary and Confidential | - 34 -



Strategy Pattern:

The Strategy pattern is useful for situations where it is necessary to dynamically swap the algorithms used in an application. The strategy pattern is intended to:

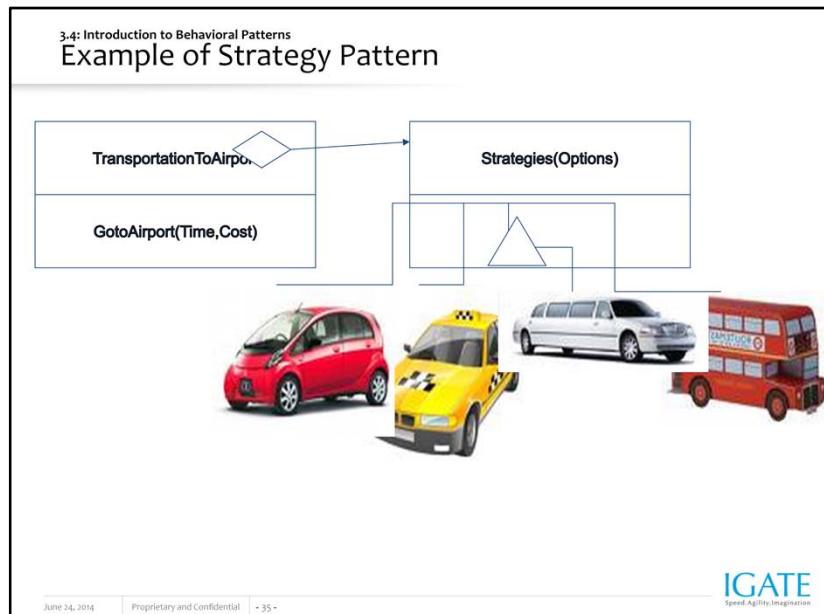
Provide a means to define a family of algorithms

Encapsulate each one as an object, and make them interchangeable

The strategy pattern lets the algorithms vary independently from clients that use them.

The Strategy pattern uses “composition” instead of “inheritance”. It is a good idea to use the Strategy Pattern when you have several objects that are basically the same, and differ only in their behavior. By using Strategies, you can reduce these “several objects” to “one class” that uses several Strategies. The use of strategies also provides a nice alternative to sub-classing an object to achieve different behaviors. When you subclass an object to change its behavior, the behavior that it executes is static.

Strategy is a bind-once pattern, whereas State is more dynamic.



Example of Strategy Pattern:

A Strategy defines a set of algorithms that can be used interchangeably.

Modes of transportation to an airport is an example of a Strategy.

Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service.

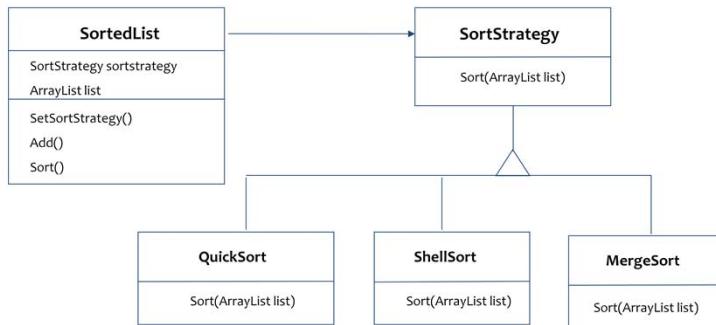
For some travelers airports, subways, and helicopters are also available as a mode of transportation to the airport.

Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on tradeoffs between cost, convenience, and time.

3.4: Introduction to Behavioral Patterns

Example of Strategy Pattern

- An application encapsulates sorting algorithms in the form of sorting objects.



June 24, 2014

Proprietary and Confidential

- 36 -

IGATE
Speed. Agility. Imagination.

This allows clients to dynamically change sorting strategies including Quicksort, Shellsort, and Mergesort.

3.4: Introduction to Behavioral Patterns

Template Method Pattern

- **Template Method Pattern** defines 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.
- **Template Method** uses inheritance to vary part of an algorithm.

June 24, 2014

Proprietary and Confidential

- 37 -



Template Method Pattern:

The Template Method pattern defines the program skeleton of an algorithm. The algorithm itself is made abstract, and the subclasses override the abstract methods to provide concrete behavior.

First a class is created that provides the basic steps of an algorithm design. These steps are implemented using abstract methods.

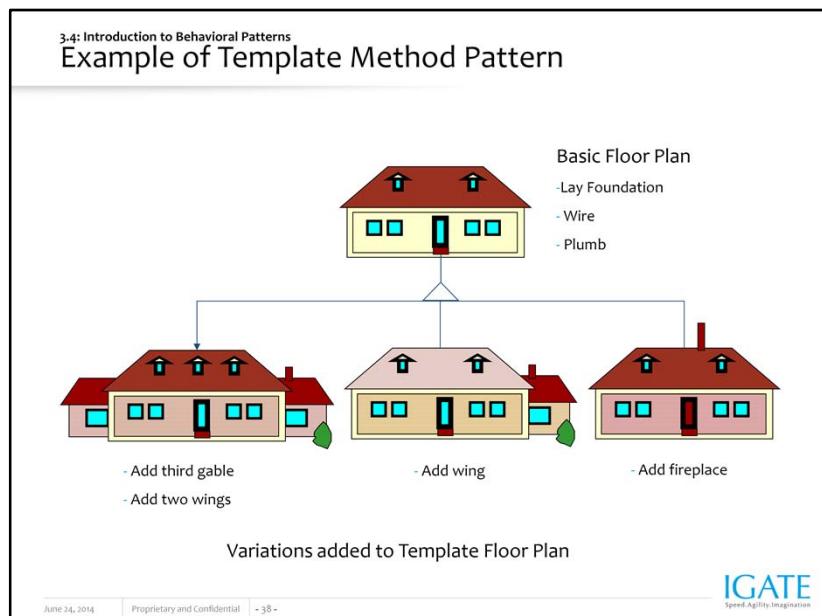
Later on, subclasses change the abstract methods to implement real actions. Thus the general algorithm is saved in one place, however, the concrete steps may be changed by the subclasses.

The Template Method defines a skeleton of an algorithm in an operation, and defers some steps to subclasses.

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

The invariant steps are implemented in an abstract base class.

The variant steps are either given a default implementation, or no implementation at all. The variant steps represent “hooks” or “placeholders”, which can or must be supplied by the component’s client in a concrete derived class.



Example of Template Method Pattern:

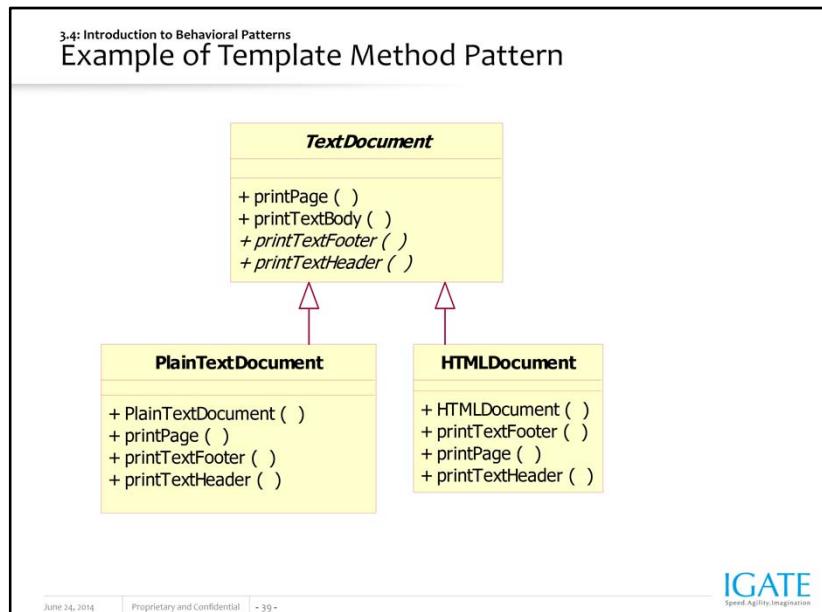
The Template Method defines a skeleton of an algorithm in an operation, and defers some steps to subclasses.

Home builders use the Template Method while developing a new subdivision.

A typical subdivision consists of a limited number of floor plans with different variations available for each.

Within a floor plan, the foundation, framing, plumbing, and wiring will be identical for each house.

Variation is introduced in the later stages of construction to produce a wider variety of models.



In this example, the `TextDocument` class is the place holder for `PlainTextDocument` and `HTMLDocument` classes. Note the abstract methods of the base class which will have an implementation defined in the derived classes.

Summary

➤ **In this lesson, you have learnt:**

- Concept of Fundamental Design Pattern with examples
- Concept of Creational, Structural & Behavioural Design Patterns with examples



Review Question

- Question 1: ___ pattern provides a level of Indirection for the client who wants to access a particular service.
- Question 2: ___ pattern supports the design principle – “Favor composition over Inheritance”.
- Question 3: ___ pattern allows the subclasses to decide which class to instantiate.
- Question 4: ___ pattern allows two unrelated interfaces to work together.



Review Question: Match the Following

1. State

a. Allows choice one of the family of algorithms at run-time

2. Strategy

b. Defines skeleton of an algorithm as an abstract class

3. Template

c. Used when change in state requires change in behavior

