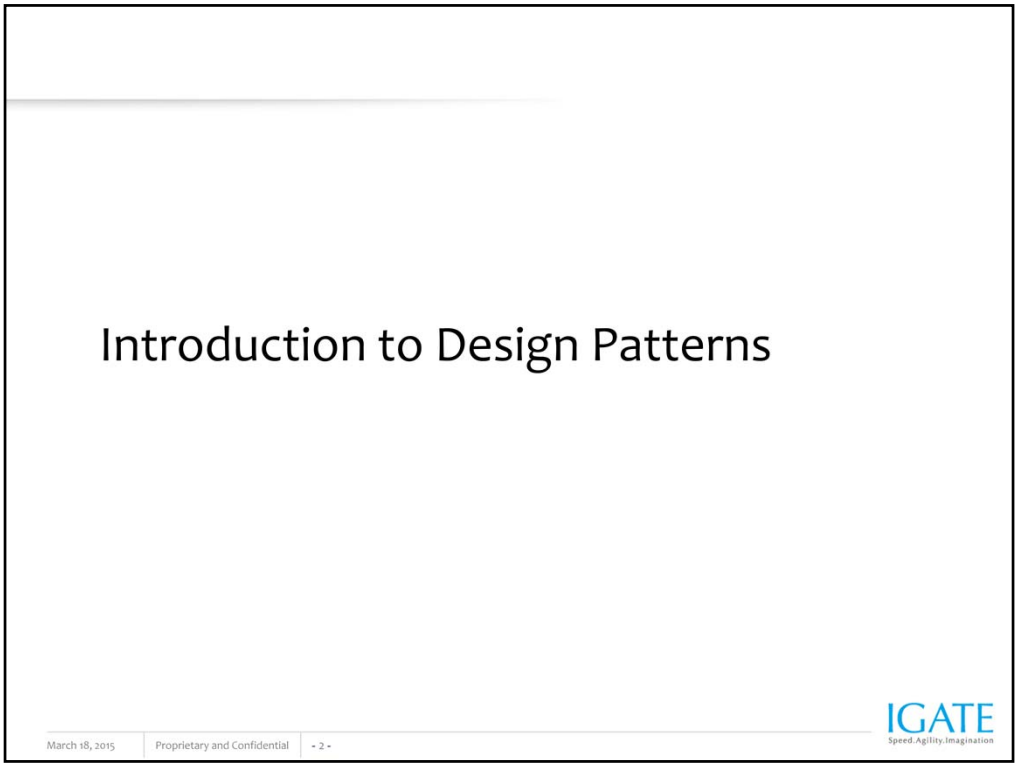
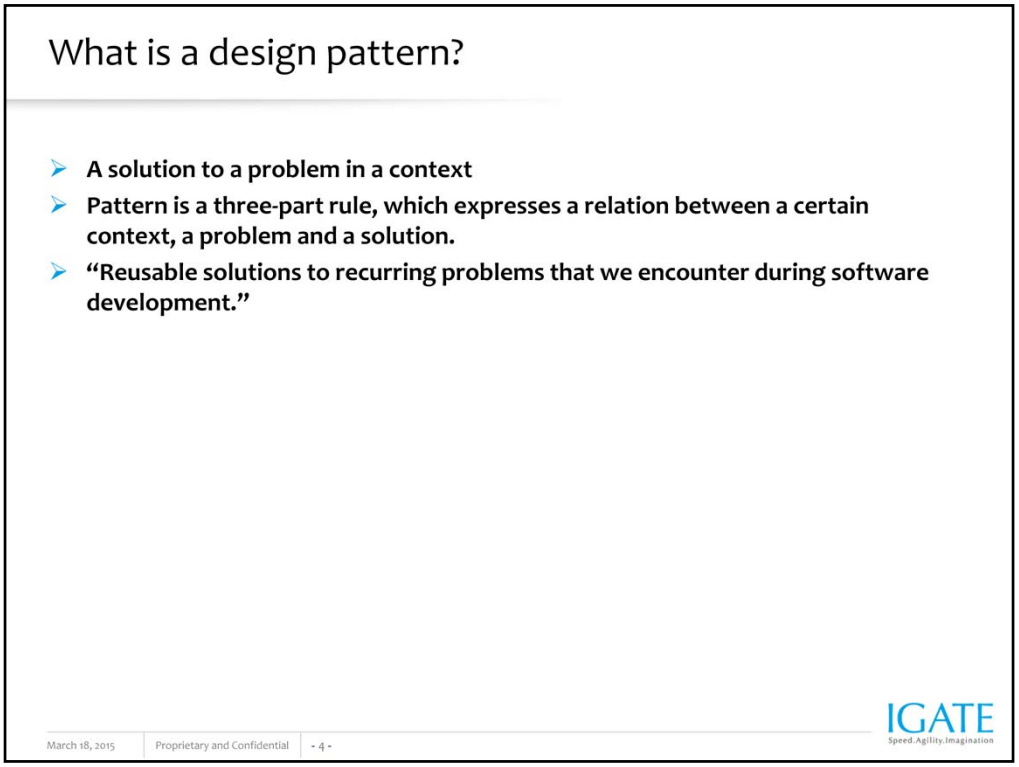
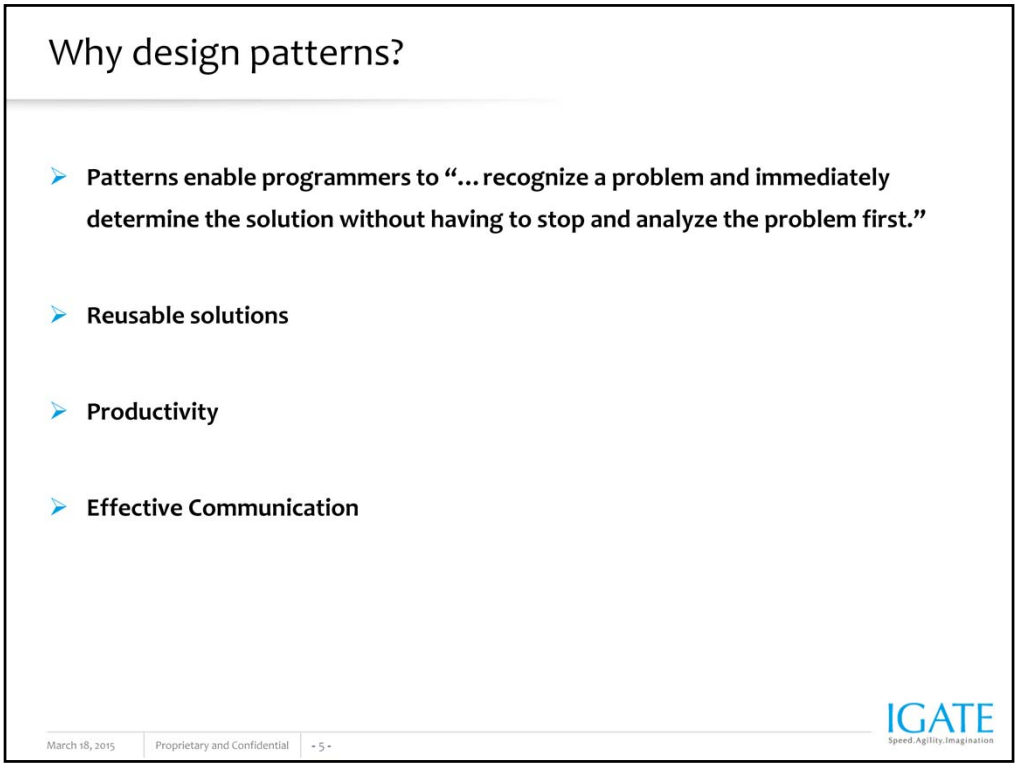

IGATE
Speed.Agility.Imagination


Introduction to Design Patterns
March 18, 2015
Proprietary and Confidential
- 2 -
IGATE
Speed.Agility.Imagination


Objectives
After completing this session, you will be able to get an introduction to
What design patterns are
Gang Of Four Patterns
Way ahead for Platform Specific Patterns
March 18, 2015
Proprietary and Confidential
- 3 -
IGATE
Speed.Agility.Imagination



"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.



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."

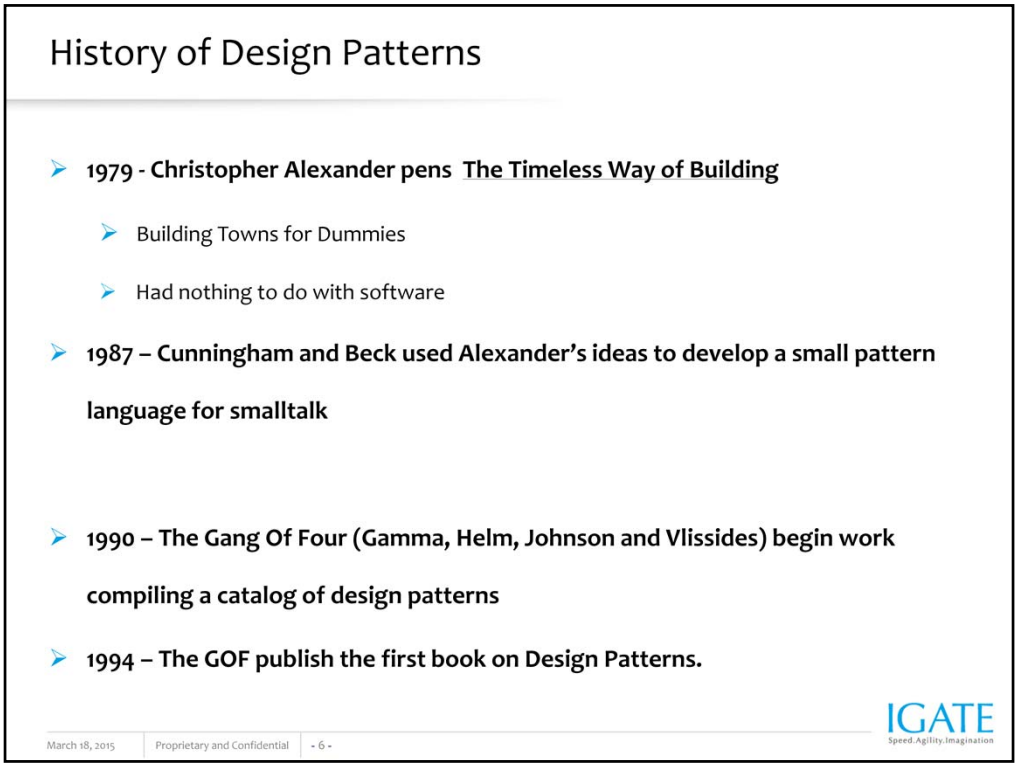
Well structured object-oriented systems have recurring patterns of classes and objects.

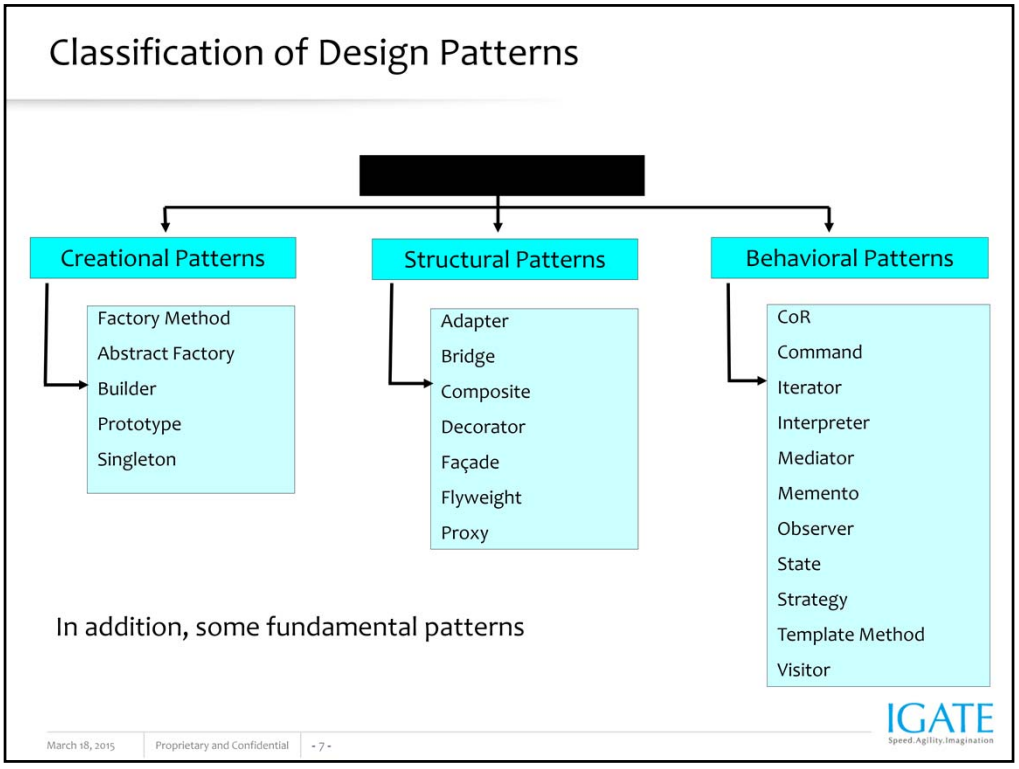
The patterns provides a framework for communicating complexities of OO design at a high level of abstraction. Bottom line: productivity.

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.

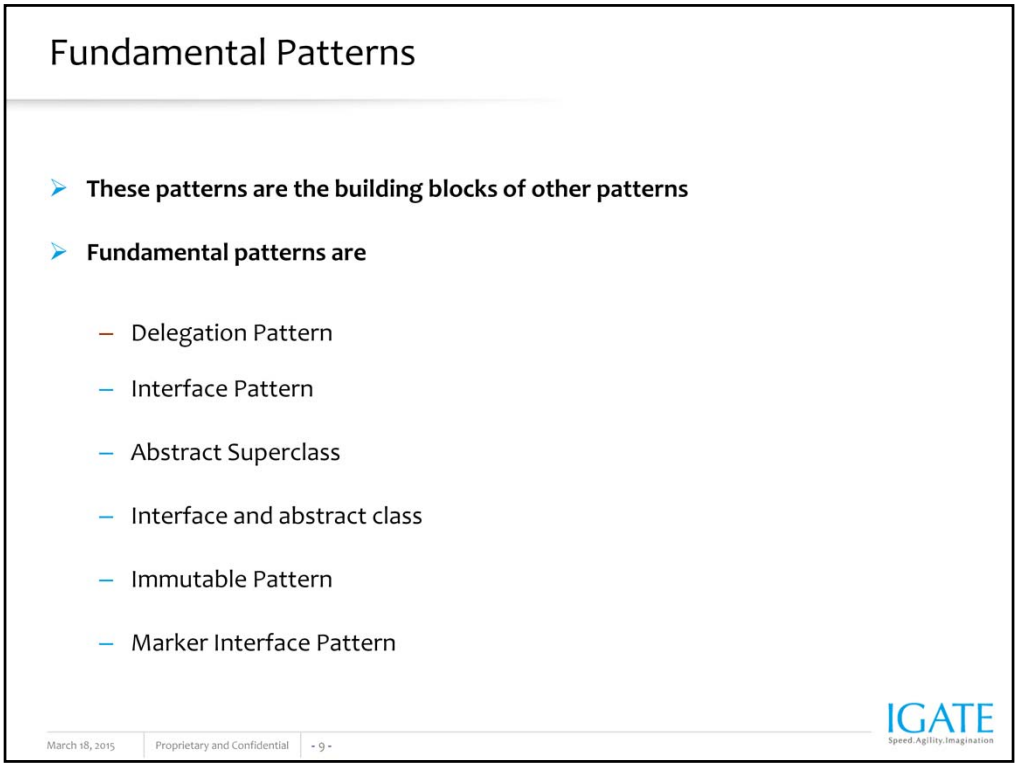
Moreover, effective communication of ideas between designers as well as between a designer and a programmer is facilitated since all speak in the common language of "patterns"


History of Design Patterns
1979 - Christopher Alexander pens The Timeless Way of Building
Building Towns for Dummies
Had nothing to do with software
1987 – Cunningham and Beck used Alexander's ideas to develop a small pattern language for smalltalk
1990 – The Gang Of Four (Gamma, Helm, Johnson and Vlissides) begin work compiling a catalog of design patterns
1994 – The GOF publish the first book on Design Patterns.
March 18, 2015
Proprietary and Confidential
- 6 -
IGATE
Speed.Agility.Imagination

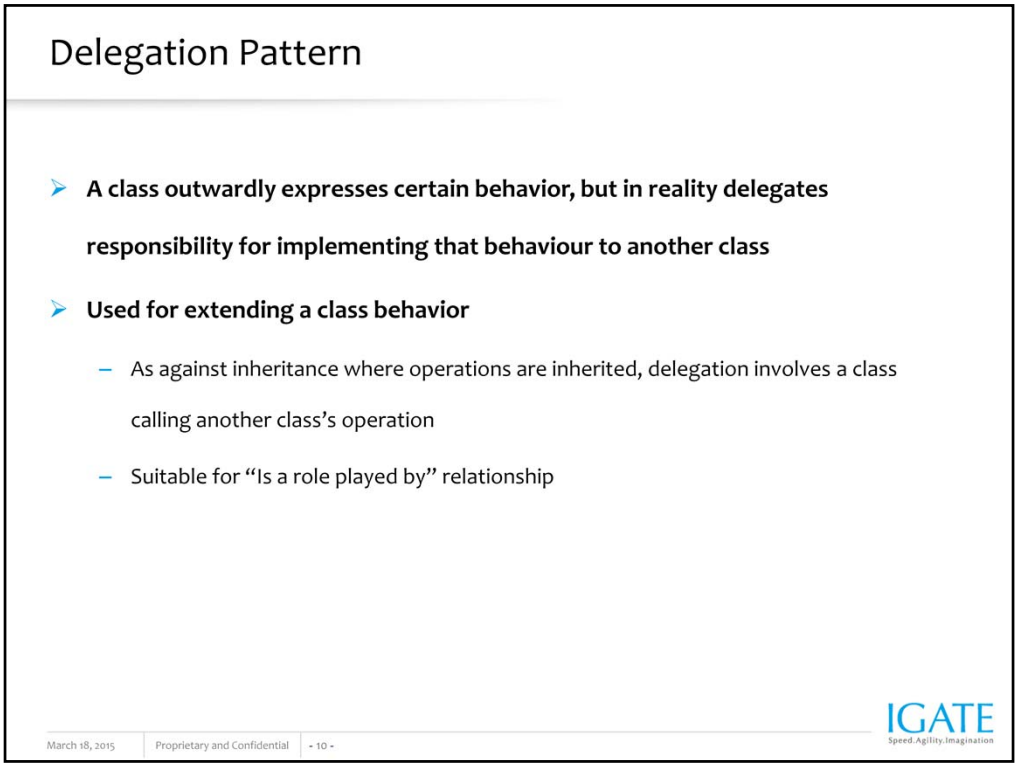

Classification of Design Patterns
[redacted]
Creational Patterns
Factory Method
Abstract Factory
Builder
Prototype
Singleton
Structural Patterns
Adapter
Bridge
Composite
Decorator
Façade
Flyweight
Proxy
Behavioral Patterns
CoR
Command
Iterator
Interpreter
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor
In addition, some fundamental patterns
March 18, 2015
Proprietary and Confidential
- 7 -
IGATE
Speed.Agility.Imagination

Categories of Design Patterns

Scope/Purpose	Creational	Structural	Behavioral
Class	Factory Method	Adapter	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



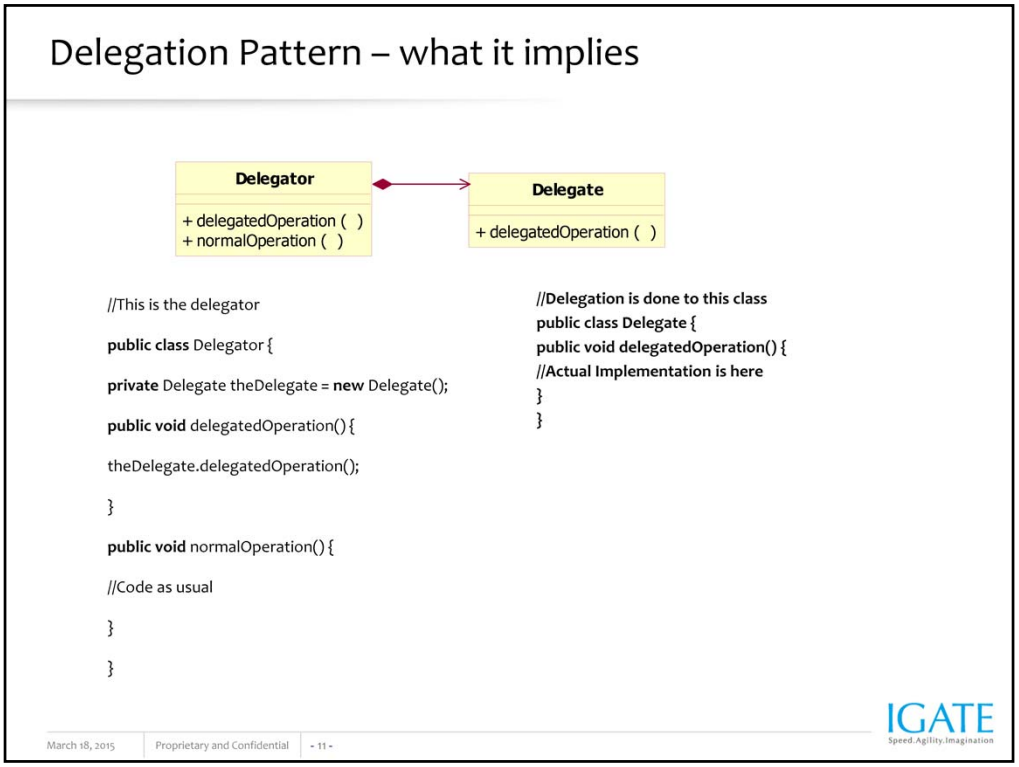
Fundamental Patterns are fundamental in the sense that they are widely used by other patterns or are frequently used in a large number of programs.



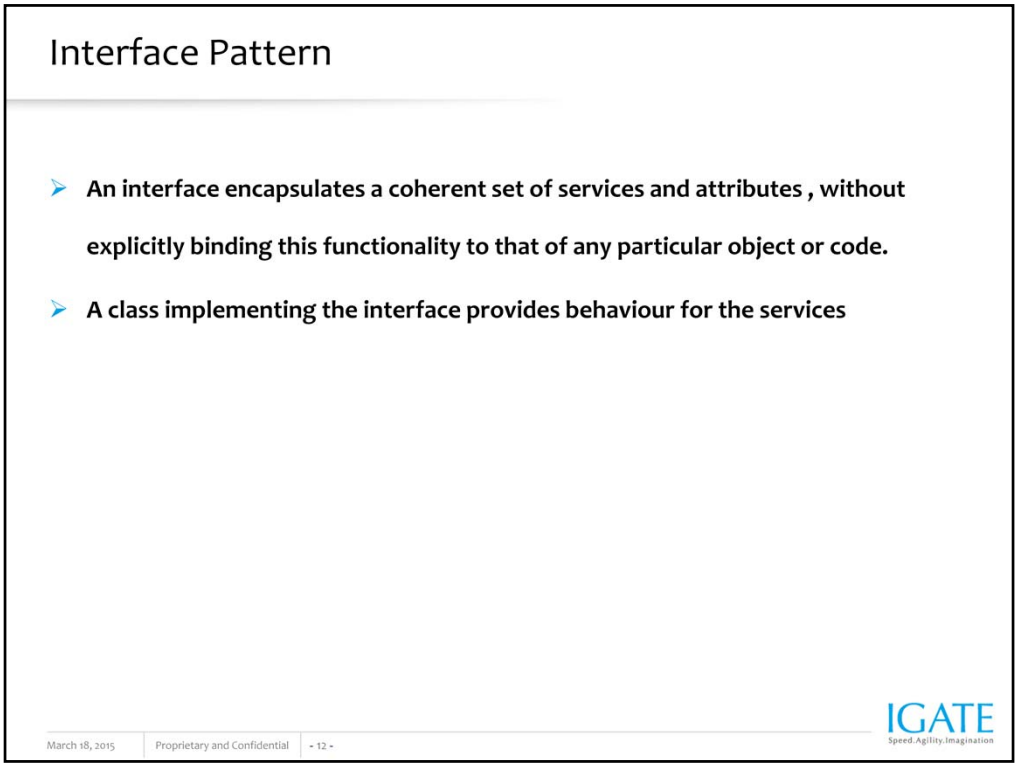
The delegation pattern is a technique where an object outwardly expresses certain behavior but in reality delegates responsibility for implementing that behavior to an associated object in an Inversion of Responsibility.

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.

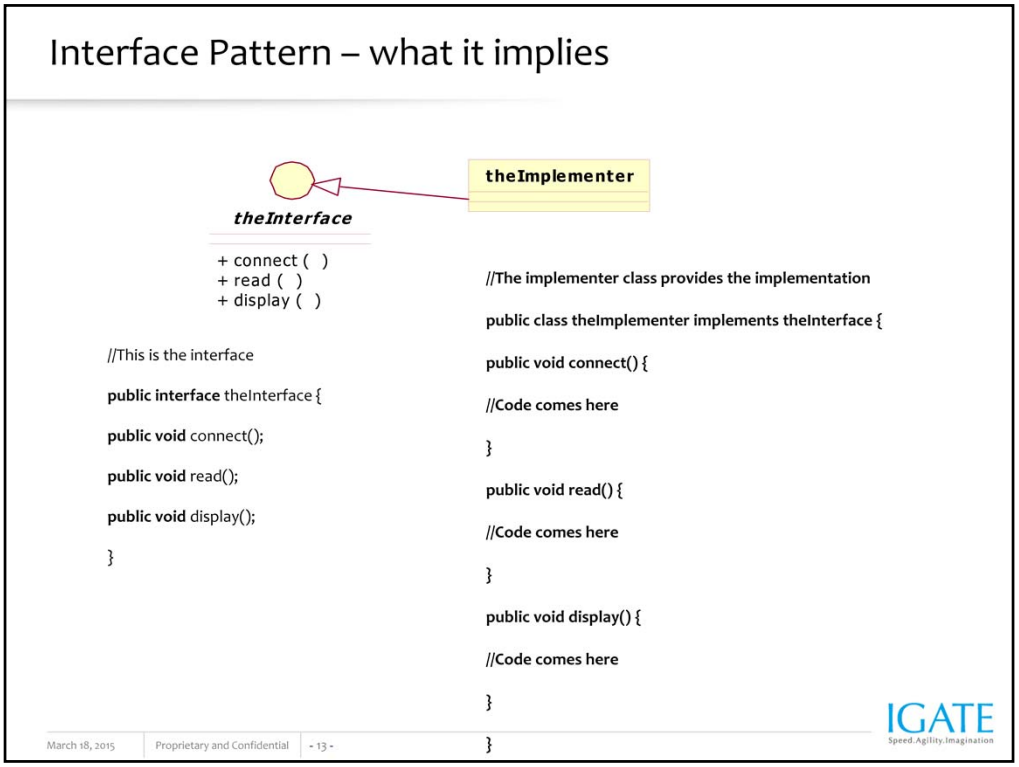
Inheritance is useful for capturing "is-a-kind-of" relationships because of their static nature. Delegation is suitable for "is-a-role-played by" relationship.



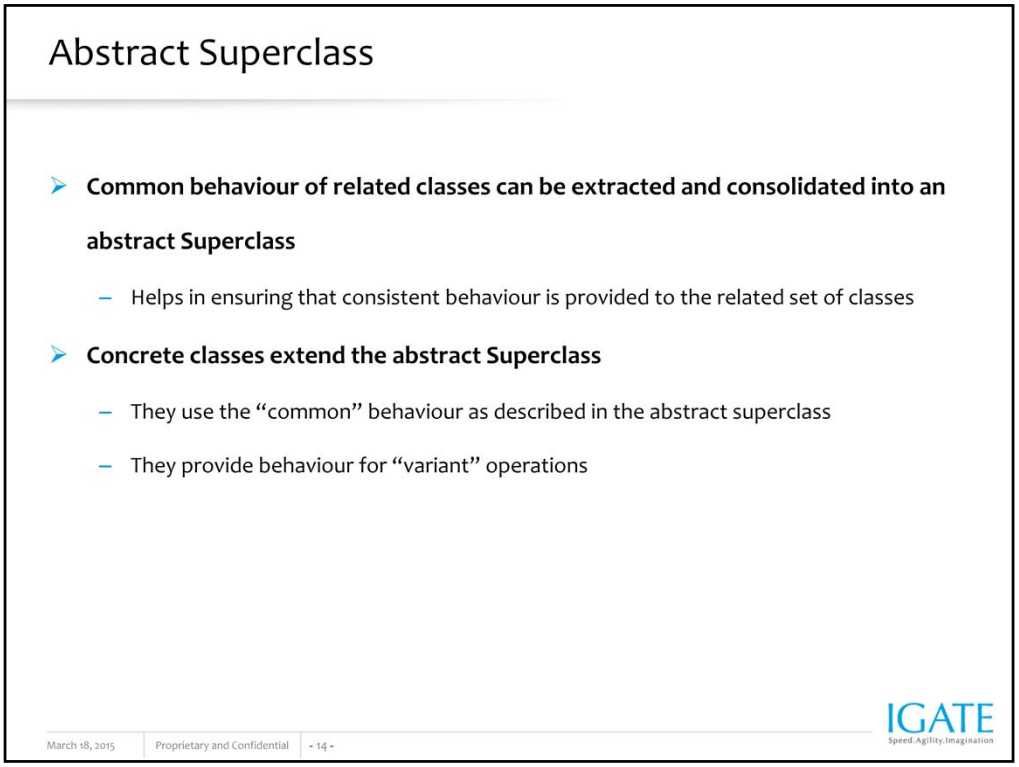
The Delegator exposes the operation "delegatedOperation()" – when this operation is invoked by the outside world, the Delegator calls this operation on the Delegate class.



An interface only describes public operations but does not say anything about code or representation. A class realizing the interface provides behaviour for the services described in the interface.

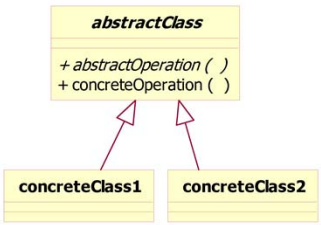


The interface is only specifying the services. It is the implementer which needs to provide the actual behaviour through the implementation of these services.



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.

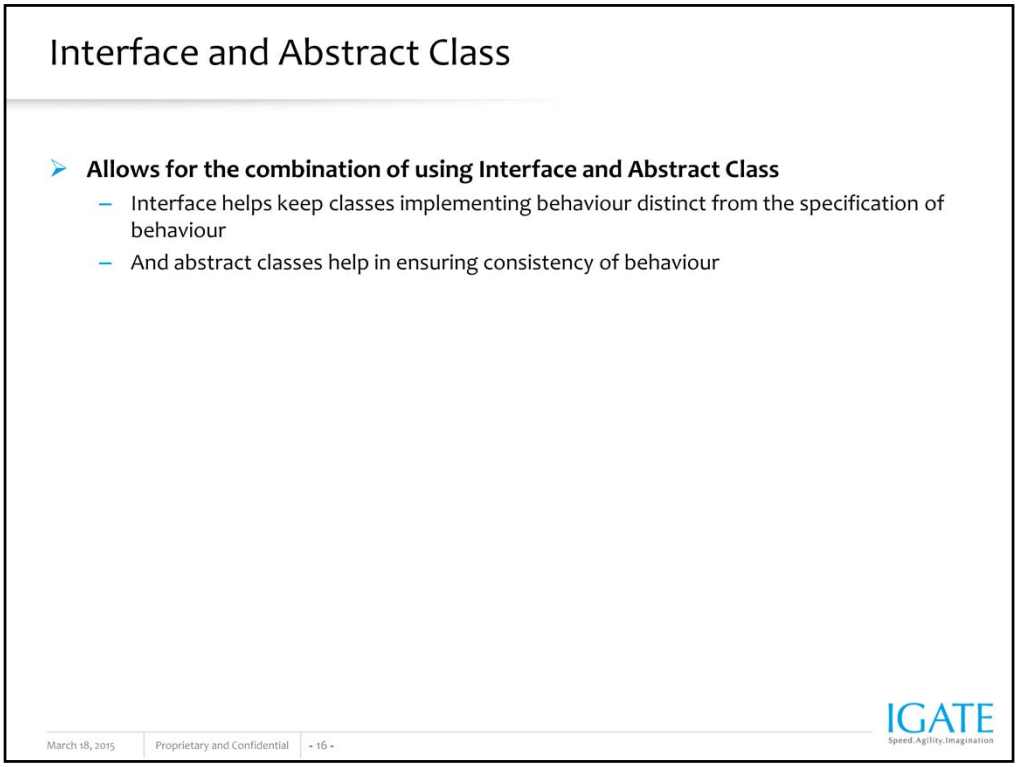
Abstract Superclass – what it implies



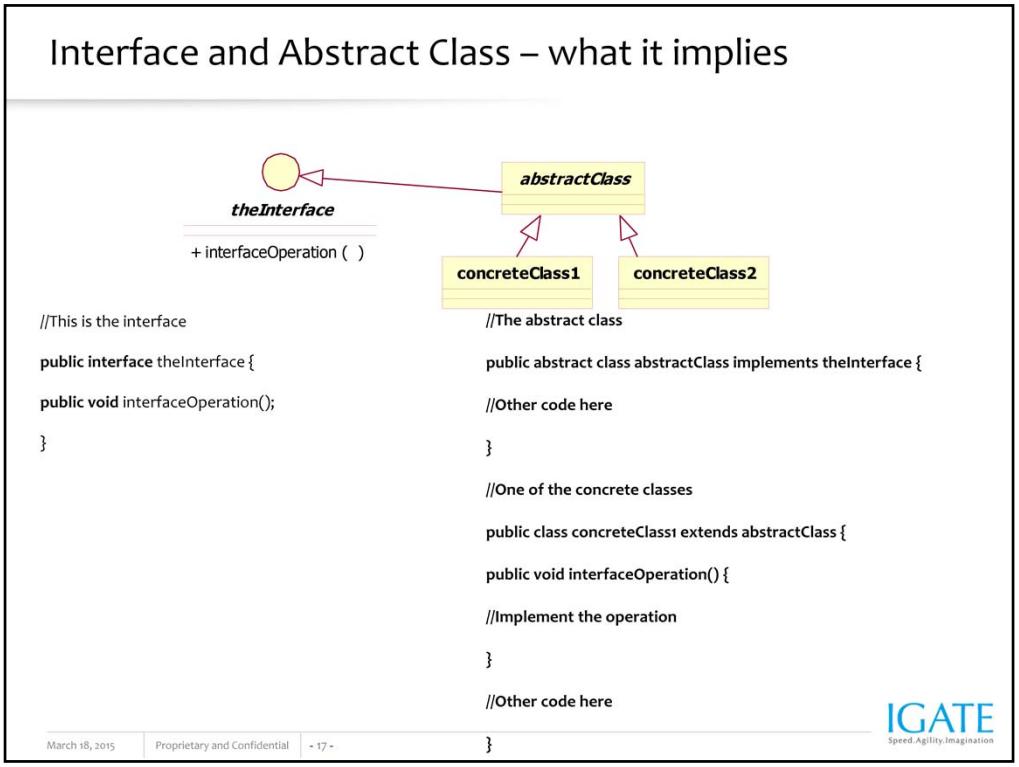

```
//This is the abstract class
public abstract class abstractClass {
//Implementation is to be provided by concrete class
public abstract void abstractOperation();
public void concreteOperation() {
//Code goes here
}}
```

```
//One of the concrete classes
public class concreteClass1 extends abstractClass {
public abstract void abstractOperation() {
//Implement this operation
}
//Other code for this class
}
```

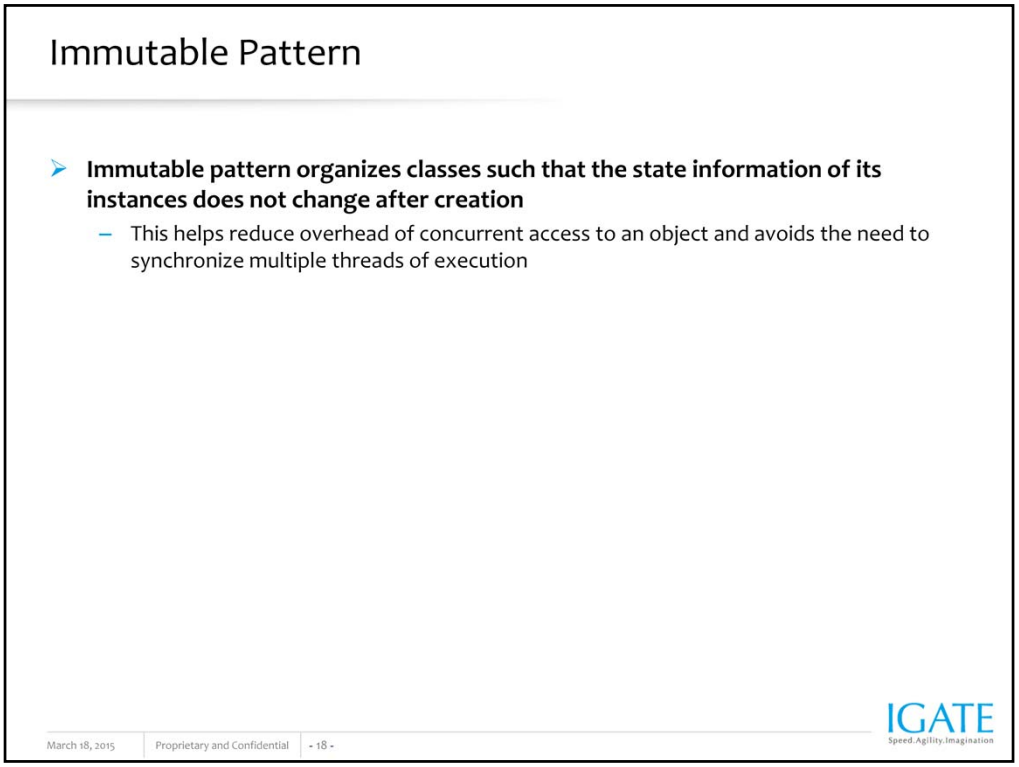

The abstract class defines the abstract operations, which are implemented by the concrete classes that extend the abstract class.



If you need to keep client classes independent of classes that implement a behavior and ensure consistency of behavior between the behaviorimplementing classes, go in for this approach. Instead of choosing between using an interface and an abstract class; we can have the classes implement an interface *and* extend an abstract class.



The interface defines the services. The abstract class realizes this interface. Actual implementation of the services is in the concrete classes.



The Immutable pattern increases the robustness of objects that share references to the same object and reduces the overhead of concurrent access to an object. It accomplishes this by not allowing an object's state information to change after it is constructed. It avoids the need to synchronize multiple threads of execution that share an object. Example:- String class in java is an immutable class

Immutable Pattern – what it implies

```
// The immutable class

public class ImmutablePosition {

private int x;

private int y;

public ImmutablePosition (int x, int y) {

   this.x = x;

   this.y = y;

}

public int getX() {

return this.x;

}

public int getY() {

return this.y;

}
```

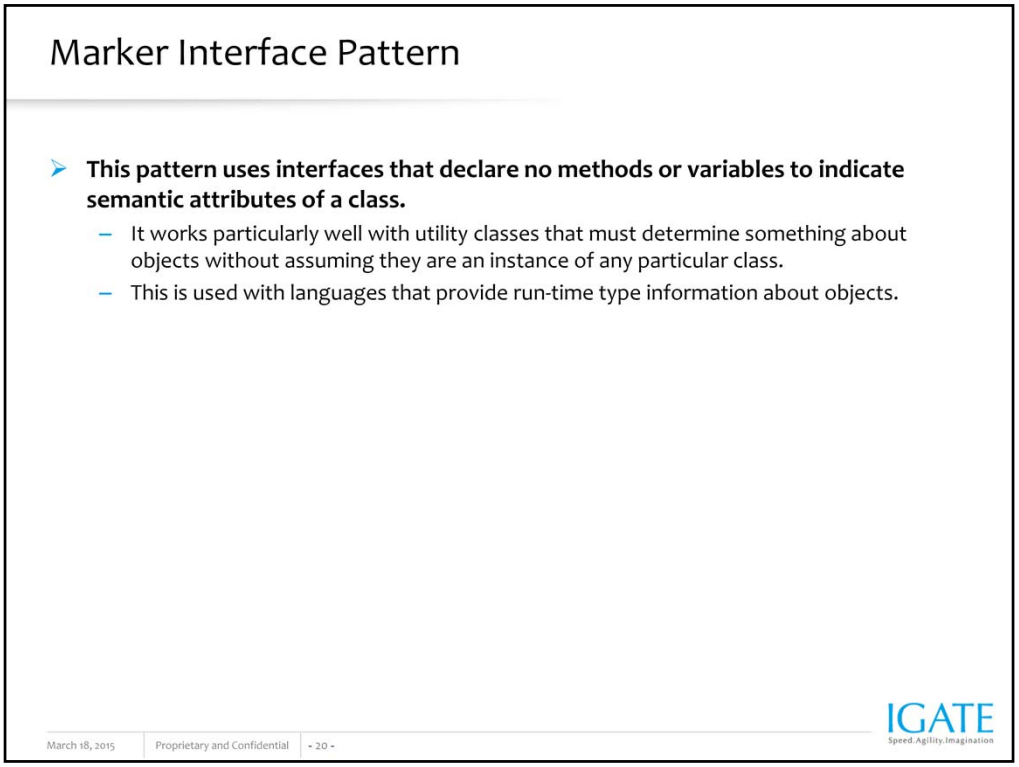
ImmutablePosition
- x : int - y : int
+ getX ( ) + getY ( ) + offset ( ) + ImmutablePosition ( )

```
//Code contd.
public ImmutablePosition offset(int xOffset, int yOffset) {
return new ImmutablePosition(x+xOffset, y+yOffset);
}
}
```

March 18, 2015 | Proprietary and Confidential | - 19 -

IGATE Speed.Agility.Imagination

Note that this immutable class does not have any operations which allow the attribute values to change.



The Marker Interface pattern uses interfaces that declare no methods or variables to indicate semantic attributes of a class.
It works particularly well with utility classes that must determine something about objects without assuming they are an instance of any particular class.
This is used with languages that provide run-time type information about objects. It provides a means to associate metadata with a class where the language does not have explicit support for such metadata.

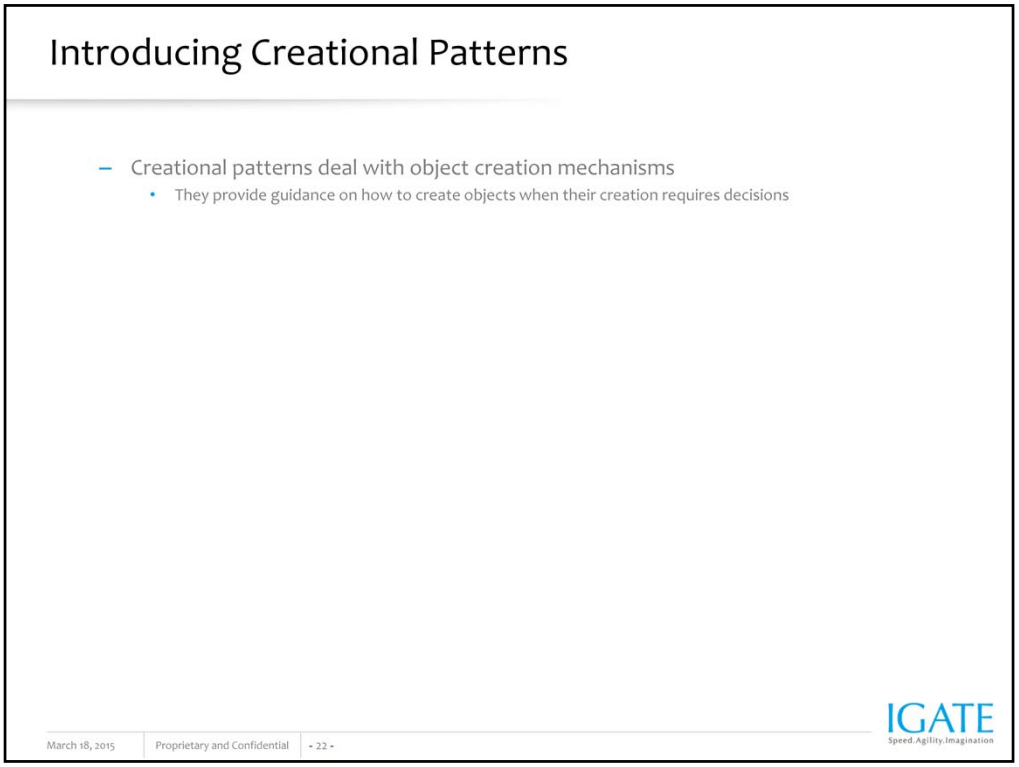
This pattern is used when classes do not have anything common but we still want them to be a part of same group
Example:- Serializable interface in Java

Marker Interface Pattern – what it implies


```
//This is the Marker Interface
public interface MarkerInterface {
}
```

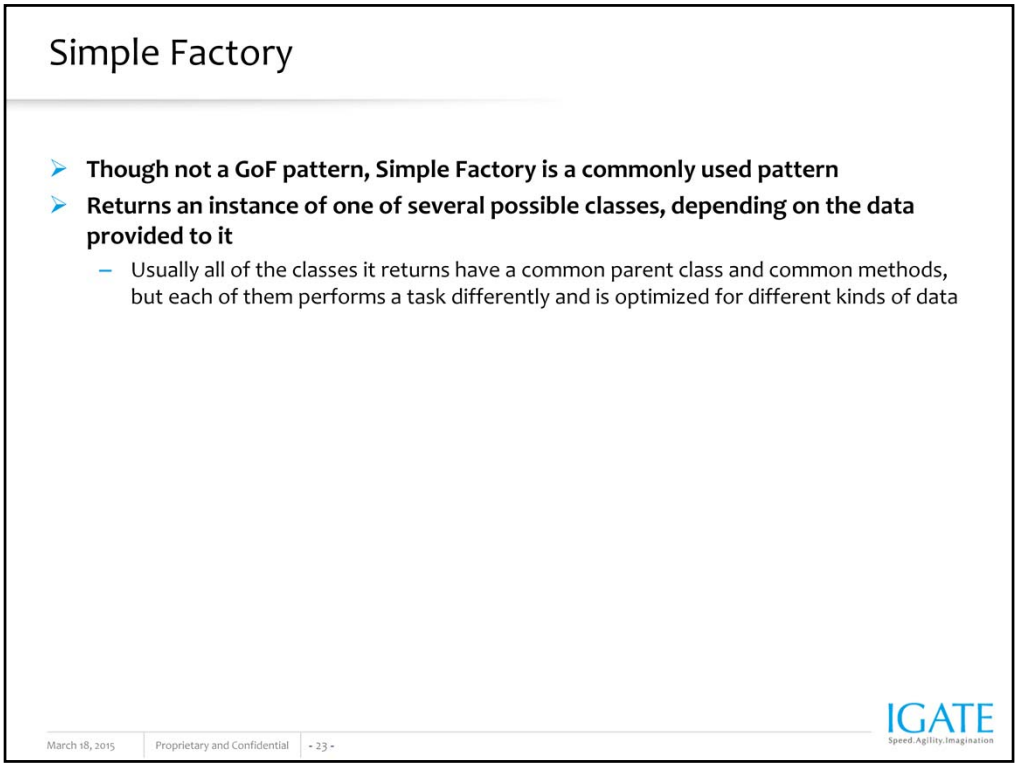
```
//The Marked class – has its own structure and behaviour
public class MarkedClass implements MarkerInterface {
}

//Client of the Marker Interface.
public class UtilityClass {
private MarkerInterface _MarkerInterface;
//Code goes here
}
```


Note that the marker interface does not define any services.



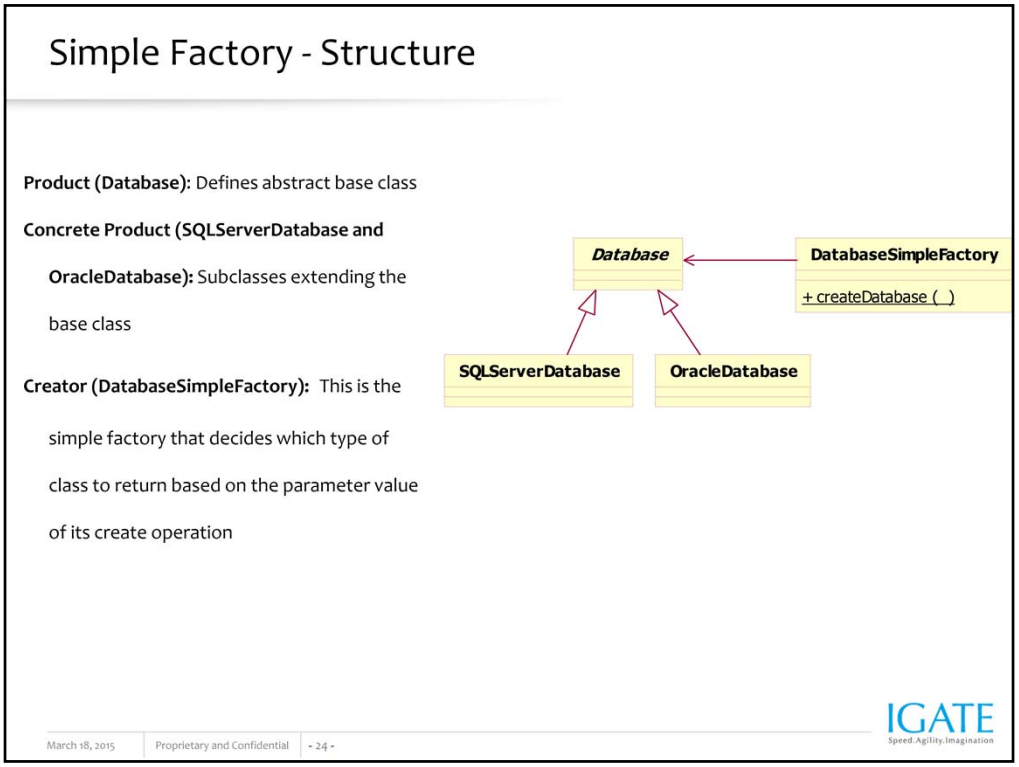
Creational patterns help to create objects in a manner suitable to the situation.

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



A Simple Factory pattern is one that returns an instance of one of several possible classes, depending on the data provided to it. Usually all of the classes it returns have a common parent class and common methods, but each of them performs a task differently and is optimized for different kinds of data.

The Simple Factory is not amongst the GoF patterns but is a fairly commonly used pattern.



Elements of this Patterns are Product, Concrete Product & Creator.
In the above example, the simple factory is being used to instantiate one of the database types depending on the parameter value passed to the createDatabase method. Here, roles of different elements are:

1. Product: Database
2. Concrete Product: SQLServerDatabase and OracleDatabase
3. Creator: DatabaseSimpleFactory

Simple Factory - Code

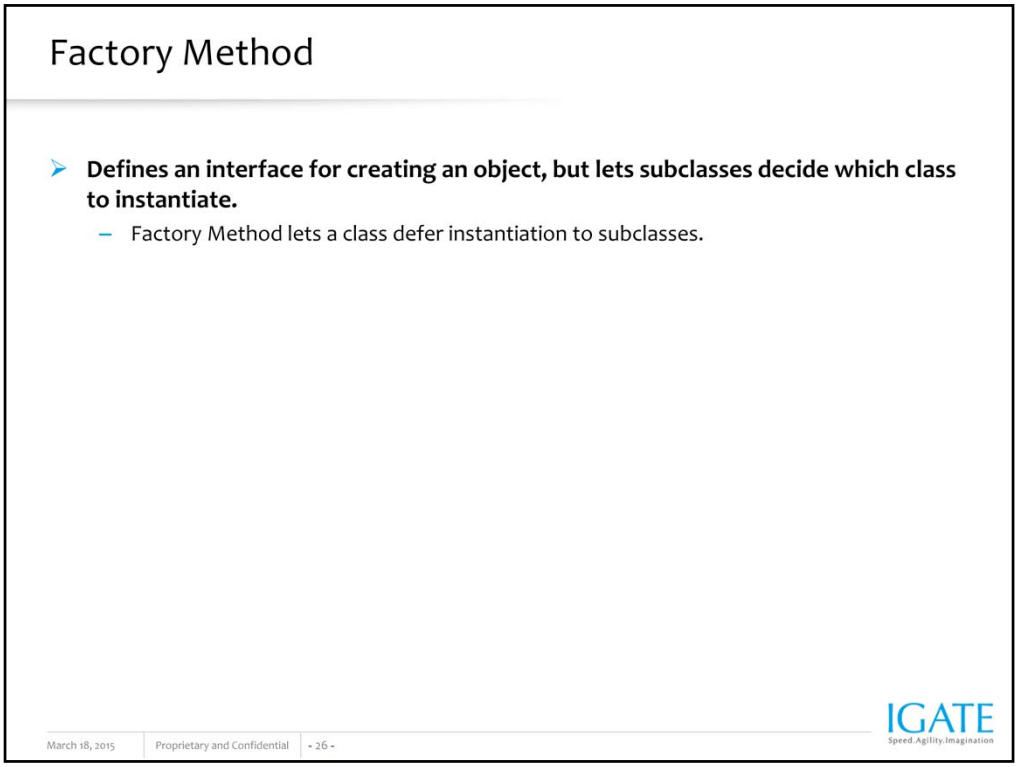
```
//Product
public abstract class Database {
}

//Concrete Product
public class OracleDatabase extends Database {
}
public class SQLServerDatabase extends Database
    {
}
```

```
//Creator
public class DatabaseSimpleFactory {
public static Database createDatabase(String spec) {
/* Depending on value of spec, appropriate database
    type is returned */
}
}
```


IGATE Speed.Agility.Imagination

Here is the code. Note the following:

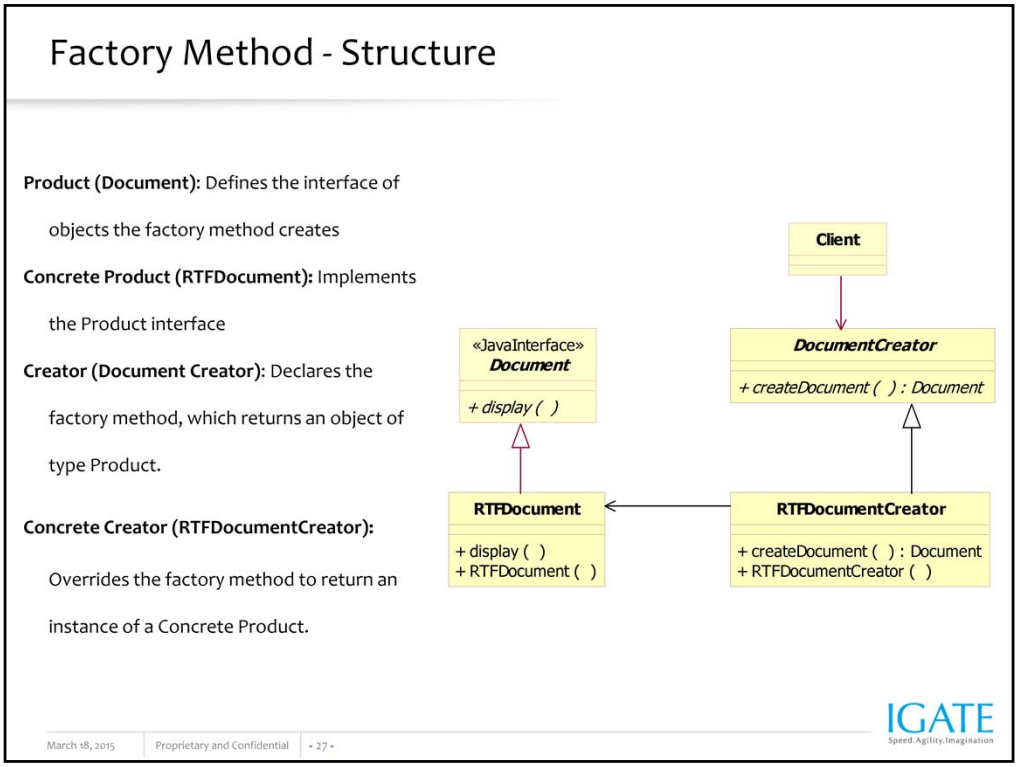
1. The Document (Product) interface is implemented by the RTFDocument (Concrete Product).
2. The abstract class DocumentCreator (Creator) defines the factory method.
3. The RTFDocumentCreator (Concrete Creator) extends the creator and overrides the factory method to return an instance of RTFDocument.



It is also called as a Factory Pattern since it is responsible for "Manufacturing" an Object.
Like other creational patterns, Factory method deals with the problem of creating objects (products) without specifying the exact class of object that will be created. It handles this problem by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of the product that will be created.

An entity bean is an object representation of persistent data that are maintained in a permanent data store, such as a database. A primary key identifies each instance of an entity bean. Entity beans can be created by creating an object using an object factory Create method. Similarly, Session beans can be created by creating an object using an object factory Create method.

The Factory Method pattern is probably the most commonly found design pattern in the .NET framework. For example, the .NET Framework contains a rich and extensible collection of cryptography classes that you can use within your own programs. Cryptography classes are part of the System.Security.Cryptography namespace. These classes use the Factory Method pattern to decouple the choice of algorithms used from their specific implementations.



Elements of this Patterns are Product, Concrete Product, Creator & Concrete Creator.
In the above 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 the classes that are playing the roles of different elements are:

1. Product: Document
2. Concrete Product: RTFDocument (others like PlainText and HTML documents can also be shown as implementing the interface)
3. Creator: DocumentCreator.
4. Concrete Creator: RTFDocumentCreator. (if we have other concrete products like PlainText and HTML, we will include corresponding concrete creators for each one. They will all extend the DocumentCreator class).

Factory Method - Code

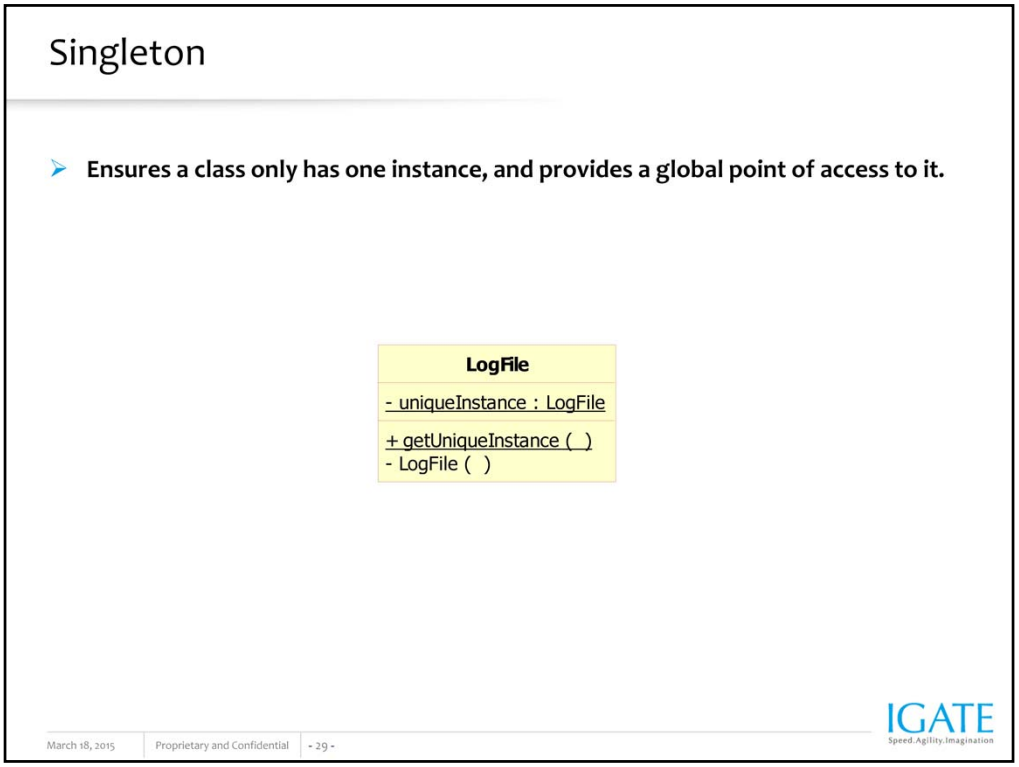
```
//Product
public interface Document {
public void display();
}
//Concrete Product
public class RTFDocument implements Document {
public RTFDocument() {
}
public void display() {
}
}
```

```
//Creator
public abstract class DocumentCreator {
private Document theDocument;
public abstract Document createDocument();
}
public class RTFDocumentCreator extends DocumentCreator {
public RTFDocumentCreator() {
}
public Document createDocument() {
/* The operation overrides the factory method to return an
    instance of a ConcreteProduct. You can customize the
    operation based on your application needs. */
 return new RTFDocument(); }
}
```


IGATE
Speed.Agility.Imagination

Here is the code. Note the following:

1. The Document (Product) interface is implemented by the RTFDocument (Concrete Product).
2. The abstract class DocumentCreator (Creator) defines the factory method.
3. The RTFDocumentCreator (Concrete Creator) extends the creator and overrides the factory method to return an instance of RTFDocument.



There are cases in programming where you need to make sure that there can be one and only one instance of a class which is used by the application. Eg. 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. Only one log file for an application. And so on.
The Singleton pattern ensure a class only has one instance, and provide a global point of access to it. The above example depicts a singleton for Logfile for an application.

Many Java setups have one virtual machine and multiple classloaders; typical singleton implementations based on class variables and methods ensure uniqueness only within each classloader, not across the whole application.

In .NET scenarios, all dialog boxes are singletons.

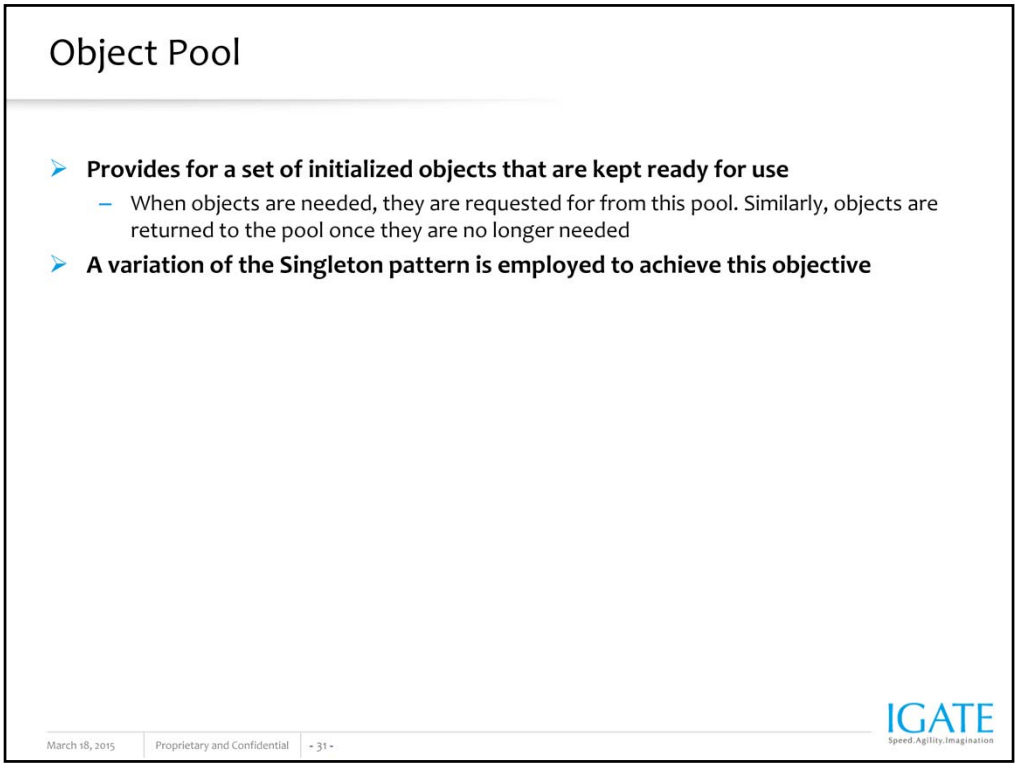
Singleton - Code

```
public class LogFile {
// This attribute stores the instance of the Singleton class.
private static LogFile uniqueInstance;
// We could also initialize static member immediately as: private static LogFile uniqueInstance = new LogFile()
//This operation implements the logic for returning the same instance of the  Singleton pattern.
public static synchronized LogFile getUniqueInstance() {
// You can customize the operation based on your application needs.
if (uniqueInstance == null) {
uniqueInstance = new LogFile();
}
return uniqueInstance;
}
private LogFile() {
}}
```


Here is the code for singleton: Note the following

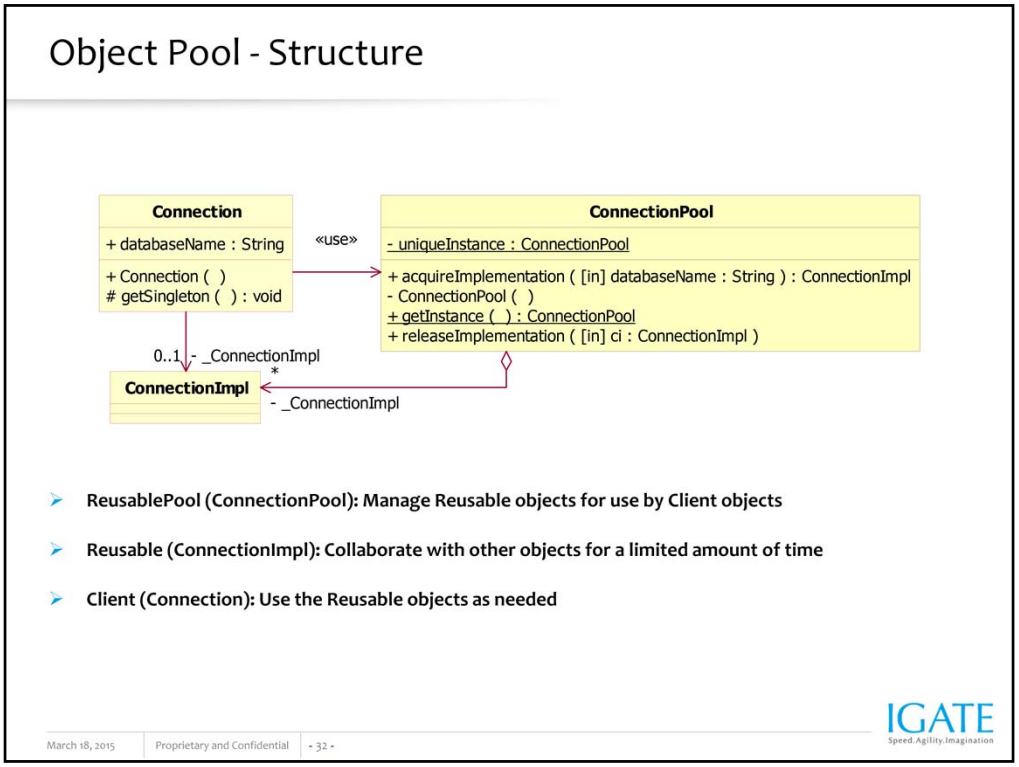
1. A private constructor to restrict the application from creating multiple instances of the class
2. A static instance variable to make the single instance globally available
3. A getUniqueInstance method to ensure that an instance is created the first time round and the same instance is used each time

Note that the static member variable can also be initialized immediately rather than having a lazy initialization. We can do this with the following statement as given in comments in the above code: private static LogFile uniqueInstance = new LogFile();



Object Pools provide a set of ready to use initialized objects. A client with access to an Object pool can avoid creating a new Objects by simply asking the pool for one that has already been instantiated instead. Generally the pool will be a growing pool, i.e. the pool itself will create new objects if the pool is empty, or we can have a pool, which restricts the number of objects created. Object Pools help in boosting performance.
The object pool is a variation of a singleton – there are certain number of objects that are created and managed by the same pool.

In both Java & .NET environments, aspects like multithreading and database connections are maintained using object pools.



Elements of this Patterns are Reusable Pool and Reusable. Client uses the reusable objects. In the above example, for connecting to the database, certain number of objects are needed. The Connection Implementation objects will be created and managed by the Connection Pool. The role played by the various classes are:

1. ReusablePool:ConnectionPool
2. Reusable: ConnectionImpl
3. Client:Connection

Object Pool - Code

```
//Reusable
class ConnectionImpl {
  private String dbName;
  private ConnectionImpl(String dbName) {
    this.dbName = dbName;
  }

  //Send a request to the database and return the result.
  Object sendRequest(Request request) {
    Object result = null;
    ...
    return result;
  }
}
```

```
//Reusable Pool
static class ConnectionPool {
 //One instance of this class
 private static ConnectionPool thePool = new ConnectionPool();
 /* Hash table for database names with the corresponding Vector
  that contains a pool of connections for that database */
 private Hashtable poolDictionary = new Hashtable();
 private ConnectionPool() {}
 public static ConnectionPool getInstance() {
   return thePool;
 }
 // Code Continued...
```


Here is the code. Note the following:

1. The private constructor of the ConnectionImpl class sets the database name and sends request to database.
2. ConnectionPool is the Singleton class.

Object Pool - Code

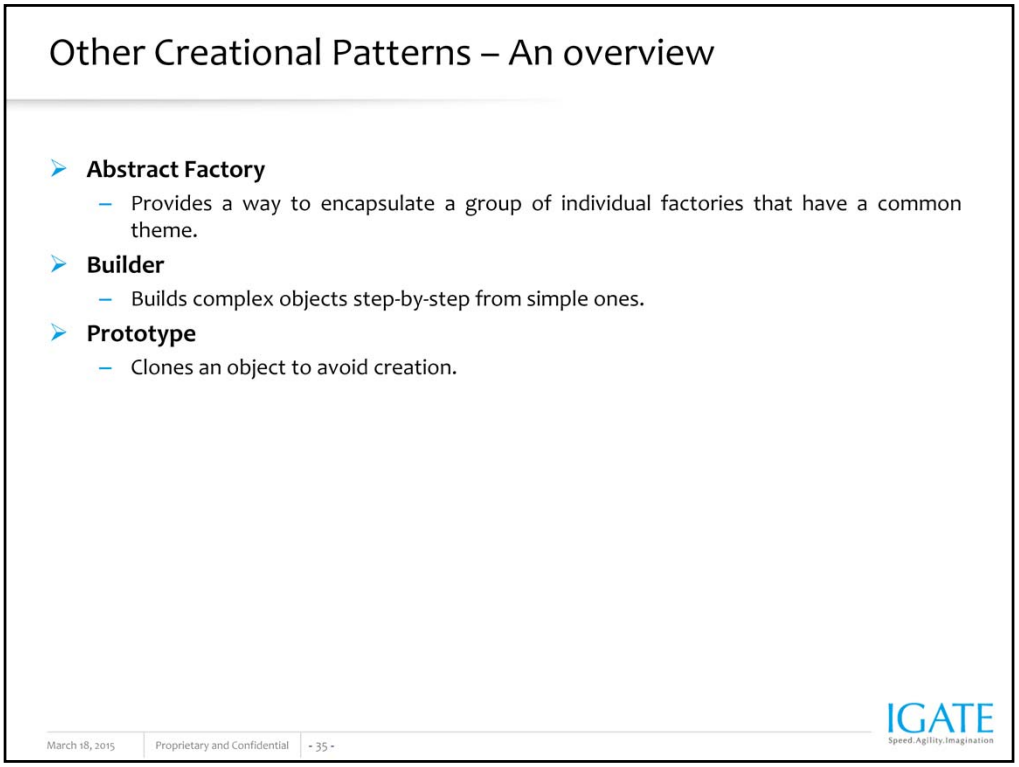
```
/*Return a ConnectionImpl from the apropriate pool or create one  if
    the pool is empty.*/
public synchronized ConnectionImpl acquireImpl(String dbName) {
   Vector pool = (Vector)poolDictionary.get(dbName);
   if (pool != null) {
     int size = pool.size();
     if (size > 0)
       return (ConnectionImpl)pool.remove(size-1);
   }
   return new ConnectionImpl(dbName);
 }
//Code Continued..
```

```
//Add a ConnectionImpl to the appropriate pool.
 public synchronized void releaseImpl(ConnectionImpl impl) {
   String databaseName = impl.getDatabaseName();
   Vector pool = (Vector)poolDictionary.get(databaseName);
   if (pool == null) {
     pool = new Vector();
     poolDictionary.put(databaseName, pool);
   }
   pool.addElement(impl);
 }
}
```


IGATE
Speed.Agility.Imagination

Here is the code. Note the following:

3. The class here returns a ConnectionImpl from the apropriate pool or creates one if the pool is empty. The object is released to Pool once job is done.

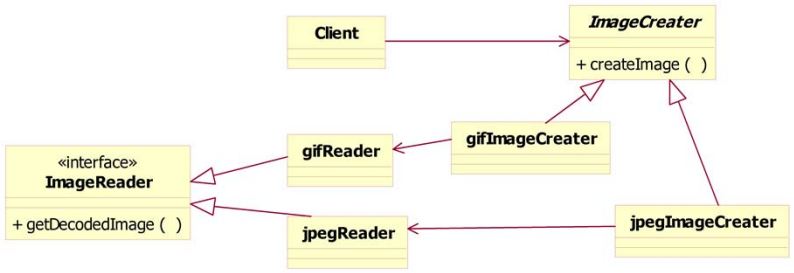


The Abstract Factory Pattern provide an interface for creating families of related or dependent objects without specifying their concrete classes.
In normal usage, the client software would create a concrete implementation of the abstract factory and then use the generic interfaces to create the concrete objects that are part of the theme. The client does not know (nor care) about which concrete objects it gets from each of these internal factories since it uses only the generic interfaces of their products.
This pattern separates the details of implementation of a set of objects from its general usage.

In using the Builder pattern, the intention is to separate the construction of a complex object from its representation so that the same construction process can create different representations. Often, Builder Pattern builds Composite pattern, a structure pattern.

Prototype is used when it is more convenient to clone classes rather than instantiating the class manually.

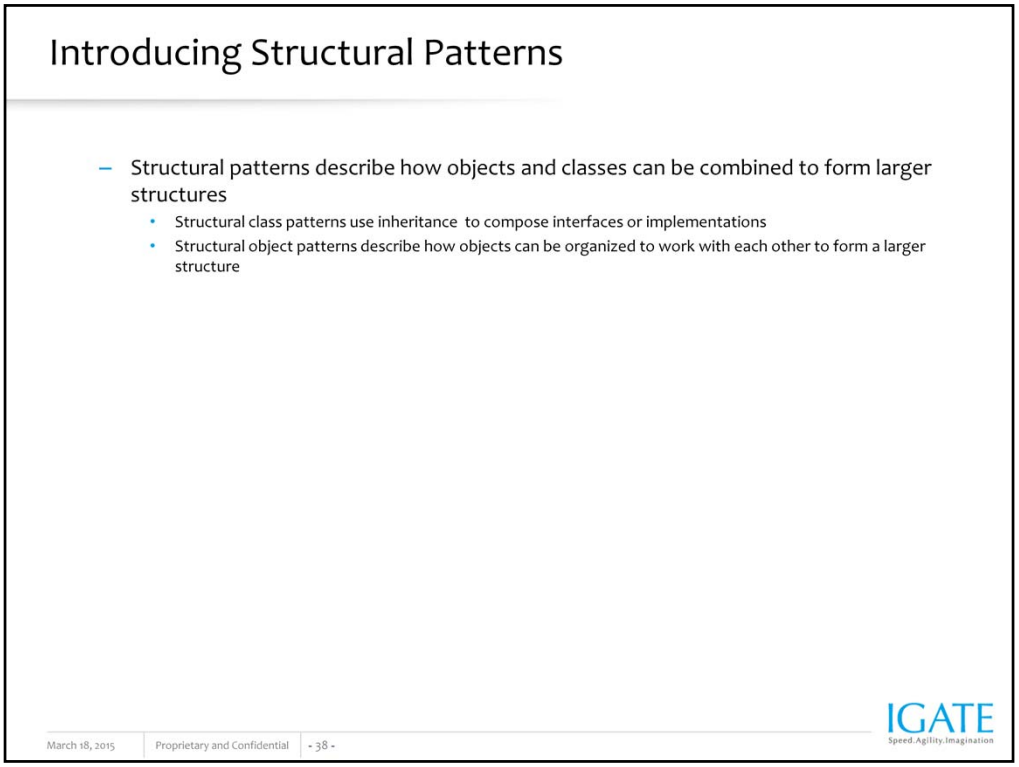
Creational Patterns – Exercises



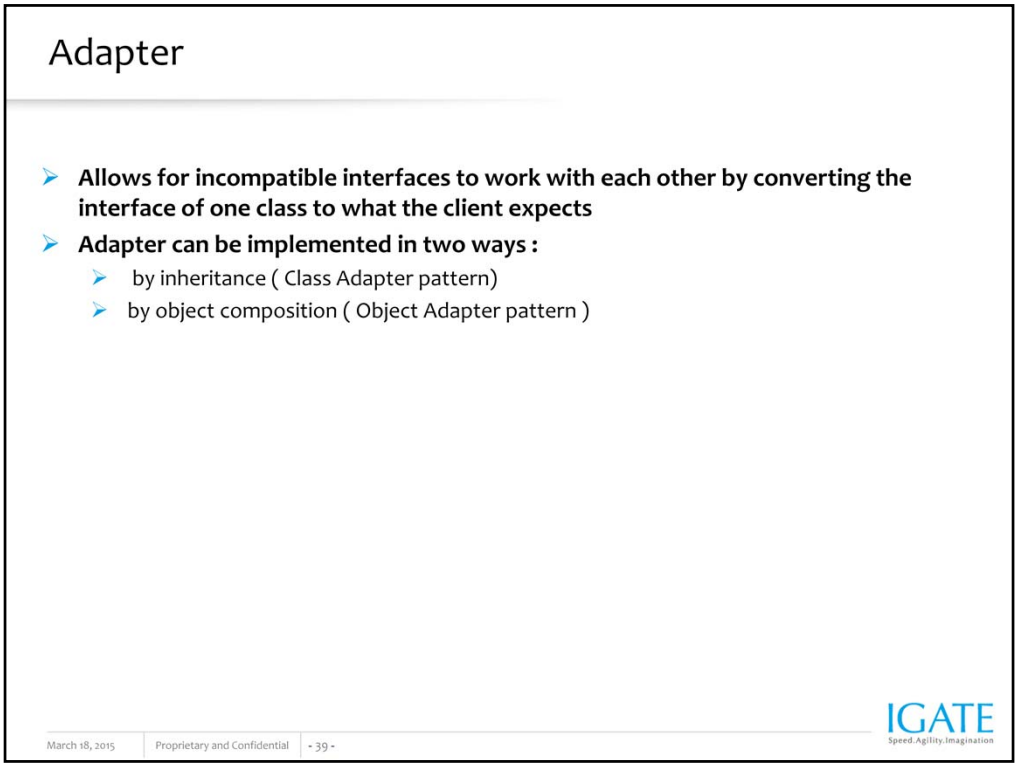
- **Write skeletal code/pseudo code for the above pattern (factory method)**

Creational Patterns – Exercises

- **Write skeletal code/pseudo code for having a class RandomNumberGenerator as a Singleton class. The RandomNumberGenerator class will be used to generate random numbers.**



Structural Patterns describe how objects and classes can be combined to form larger structures. The difference between class patterns and object patterns is that 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.

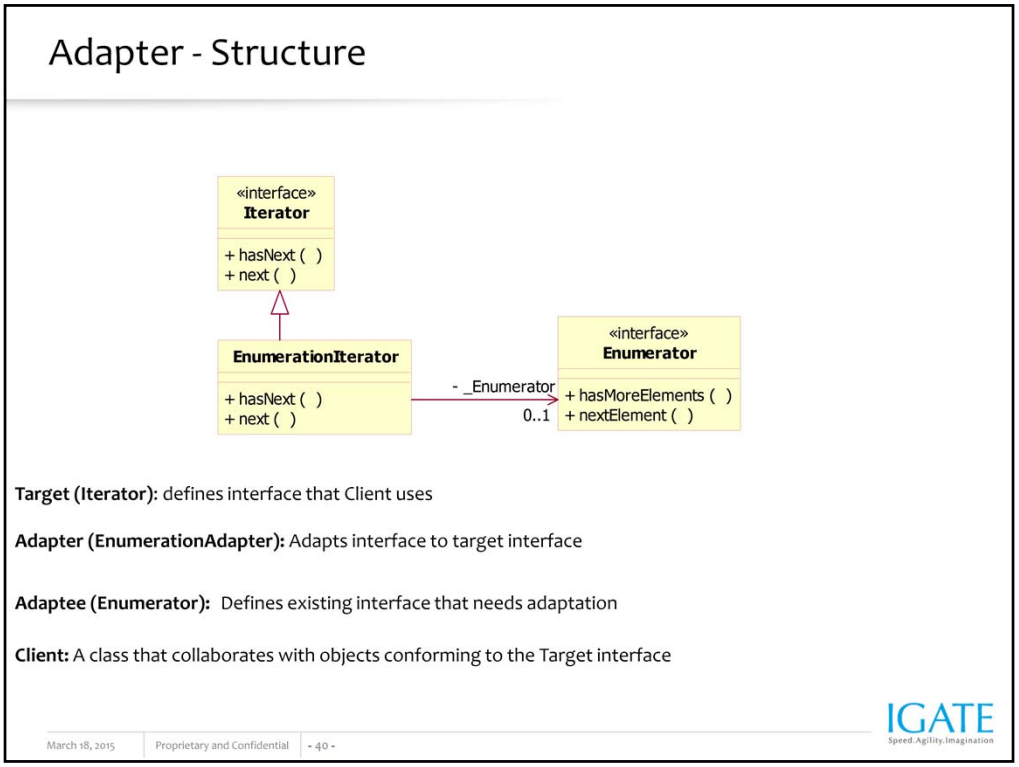


The adapter design pattern (often referred to as the wrapper pattern or simply a wrapper) 'adapts' one interface for a class into one that a client expects. An adapter allows classes to work together that normally could not because of incompatible interfaces by wrapping its own interface around that 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 would be responsible for extracting the appropriate values from the integer value.

JavaBeans GUI Builder tools make use of Adapter classes to bind bean properties together using events.

An example in .NET is the several types of data adapters (like SqlDataAdapter, OdbcDataAdapter & OracleDataAdapter) that are available in the framework to take care of database connections and commands.



Elements of this Patterns are Target, Adapter, Adaptee and Client.
In the above example, we want to perform selective copy of a vector of Document objects without modifying the Document class. Here, roles of different elements are:

1. Target: Iterator interface
2. Adapter: DocumentAdapter
3. Adaptee: Document
4. Client: Any class that collaborates with the target interface

Adapter - Code

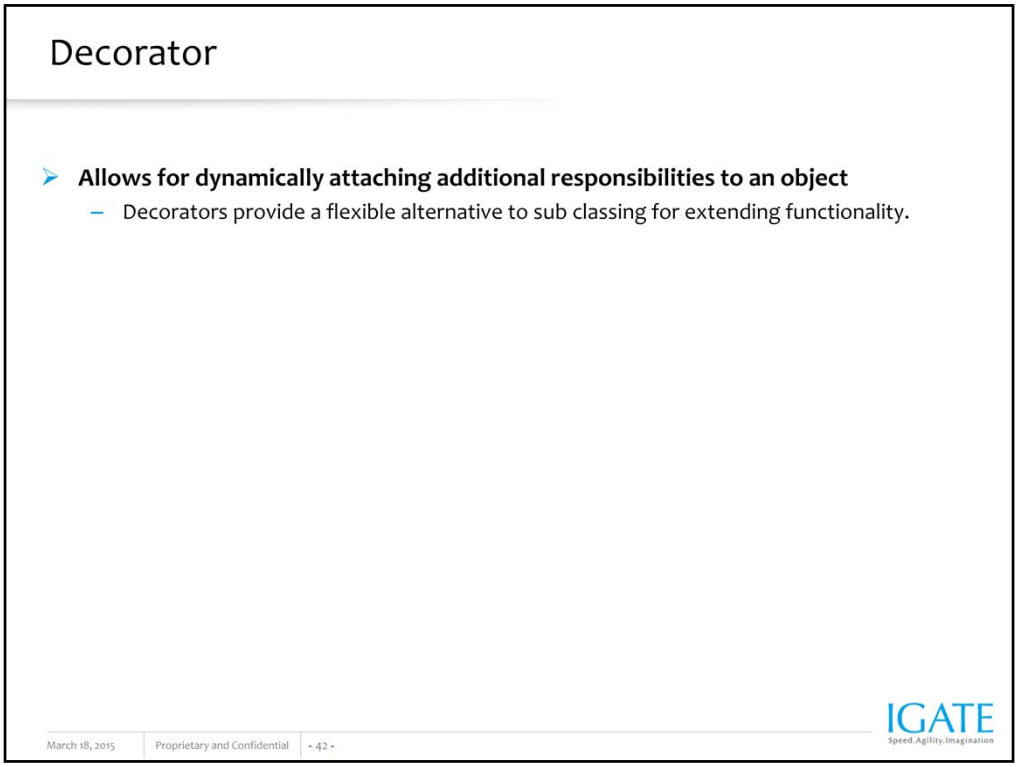
```
//Target
public interface Iterator {
public boolean hasNext();
public object next();
}

//Adaptee
public interface Enumeration {
// Operations to be adapted by the Adapter.
public boolean hasMoreElements();
public object nextElement();
}
```

```
//Adapter
public class EnumerationIterator implements Iterator {
Enumeration enum;
public EnumerationIterator (Enumeration enum) {
  this.enum = enum;
}
public boolean hasNext() {
  return enum.hasMoreElements();
}
public Object next() {
  return enum.nextElement();
}
}
```


IGATE
Speed.Agility.Imagination

Here is the code. Note the following:

1. The target, Iteratorinterface has operations for checking whether there is a next element, and if so, returning the next element
2. Adaptee, Enumeration, needs to be adapted by the Adapter
3. EnumerationIterator is the Adapter, which provides the required adaptation.



The Decorator Pattern is used for adding additional functionality to a particular object as opposed to a class of objects. It is easy to add functionality to an entire class of objects by subclassing an object, but it is impossible to extend a single object this way. With the Decorator Pattern, you can add functionality to a single object and leave others like it unmodified.

An example of usage of decorator is java.io package for InputStream/OutputStream.

In .NET too, the decorator pattern is in use in the overall design of input and output streams.

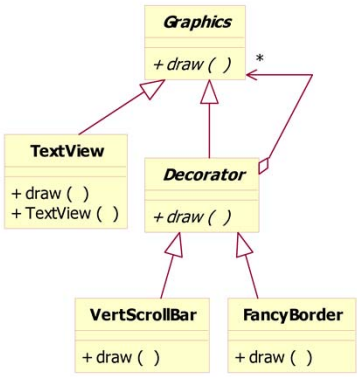
Decorator - Structure

Component (Graphics): Defines the interface for objects that can have responsibilities added to them dynamically

Concrete Component (TextView): Defines an object to which additional responsibilities can be attached

Decorator: Defines an interface that conforms to Component's interface

Concrete Decorator (VertScrollBar & FancyBorder): Adds responsibilities to component



March 18, 2015 | Proprietary and Confidential | - 43 -

IGATE Speed.Agility.Imagination

Elements of this Patterns are Component, Concrete Component, Decorator and Concrete Decorator.

In the above example, we start with a basic object textview and decorate it at runtime with VertScrollBar or FancyBorder as needed. Here, roles of different elements are:

1. Component: Graphics
2. ConcreteComponent: TextView
3. Decorator: Remains Decorator
4. ConcreteDecorator: VertScrollBar and FancyBar

Update Class Diagram as per code

Decorator - Code

```
//Component.

public abstract class Graphics {

public abstract void draw();

}


//Decorator

public abstract class Decorator extends Graphics {

public abstract void draw();

}
```

```
//Concrete Component
public class TextView implements Graphics {
public TextView() {
}
public void draw() {
// Base behaviour of TextView draw operation is here
}
}

//You could have more concrete components of Graphics defined
//For eg. ImageView having its own base behavior for draw
```


IGATE
Speed.Agility.Imagination

Here is the code. Note the following:

1. The component Graphics, an abstract class, has an abstract draw() method. Component Graphics could be defined as an interface as well.
2. The concrete decorators need to re-implement the draw, so the draw() method here is abstract in the Decorator.
3. The concrete component TextView defines an object to which additional behaviours can be attached. Similarly you could have additional concrete objects with base behaviors defined.

Decorator – Code Contd.

```
//Concrete Decorator
public class FancyBorder extends Decorator {
Decorator decorator;
public void FancyBorder (Decorator d) {
this.decorator = d;
}
public void draw() {
// invoke the Decorator's  operation for core behavior.
super.operation();
// Now this is where additional behaviour comes
//Implement draw for Fancy border for graphics component.
}
}
```

```
//Similarly Concrete Decorator VertScrollBar
public class VertScrollBar extends Decorator {
Decorator decorator;
public void VertScrollBar (Decorator d) {
this.decorator = d;
}
public void draw() {
// invoke the Decorator's  operation for core behavior.
super.operation();
// Now this is where additional behaviour comes
//Implement draw for Vertical ScrollBar for the graphics component.
}
}
```


Here is the code. Note the following:

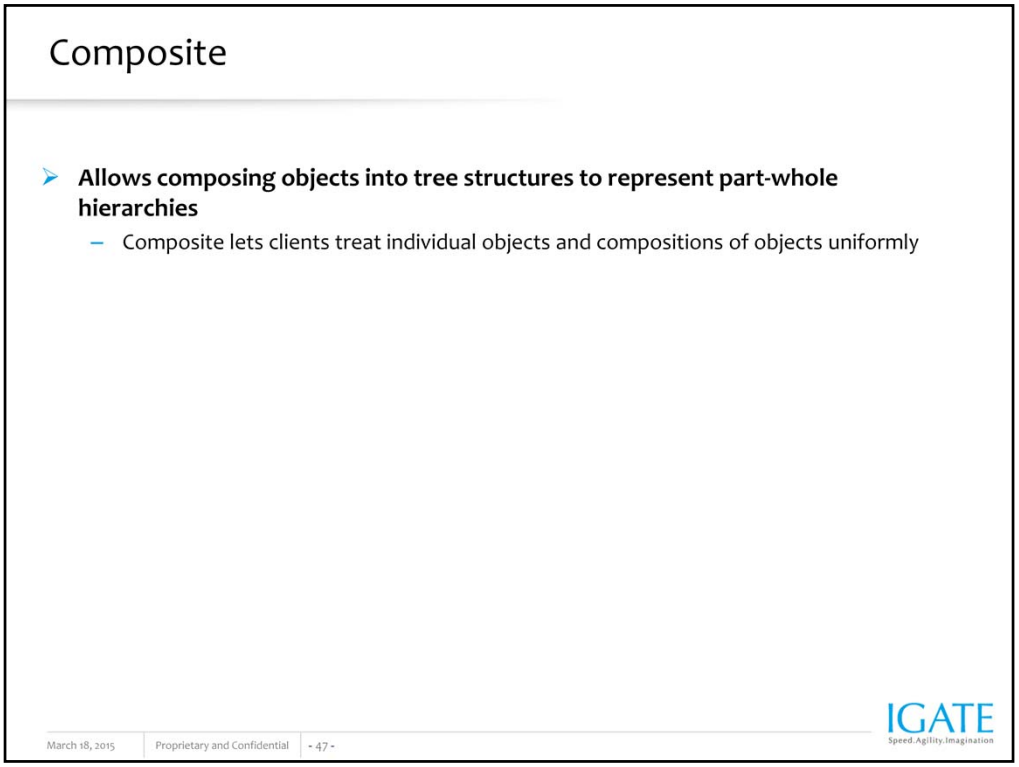
4. The ConcreteDecorators, FancyBorder and VertScrollBar add behaviours to the core draw behaviour that is defined the TextView draw. It invokes the TextView's draw operation for the core behaviour, and in the same operation, the decorator's operation for drawing the Fancy Borders or VertScrollBar is implemented.

Decorator – Code Contd.

```
//Client
public class Client {
public Client() {
}
//Implementing an example of using a Decorator.
public void buildDecorator() {
Graphics tview = new TextView();
tview.draw();    //This takes care of core behaviour of draw for TextView
Graphics tview1 = new TextView();
tview1 = new FancyBorder(tview1);  //TextView object decorated with Fancy Border
tview1 = new VertScrollBar (tview1); //TextView object decorated with Vertical Scroll Bar
tview1.draw();  //draw behavior of TextView, FancyBorder & VertScrollBar – All obtained
}
```


Here is the code. Note the following:

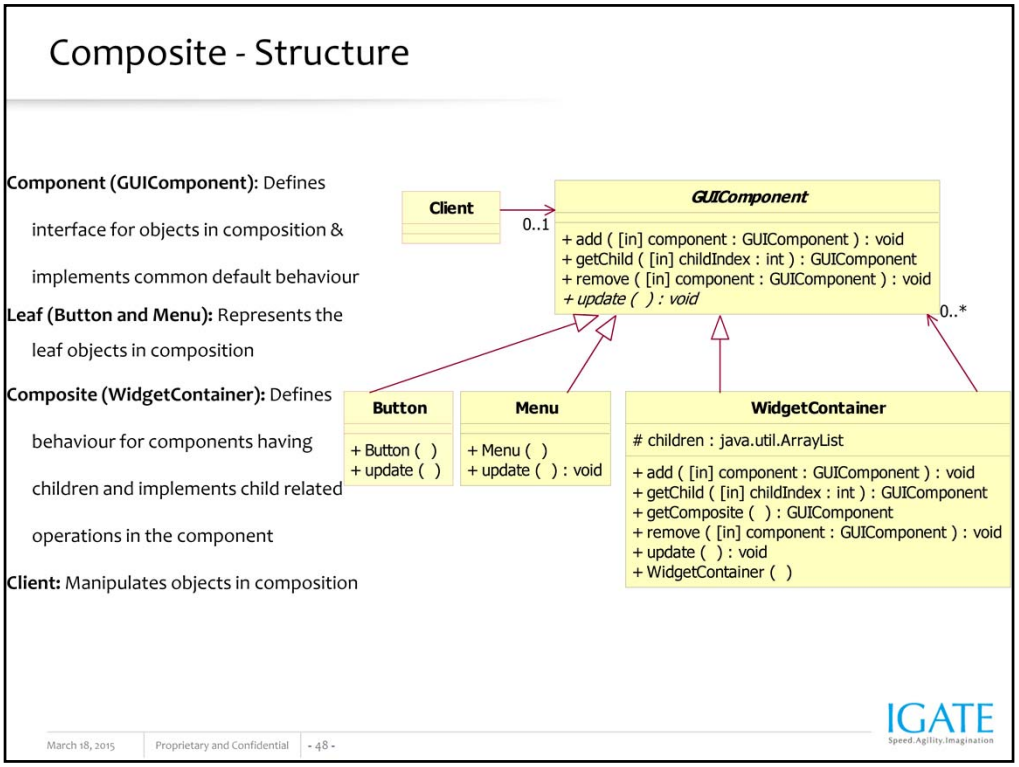
5. The Client illustrates using the Decorator pattern. In the buildDecorator operation, tview is the object which can exhibit core behaviour of draw. The object tview1 is TextView decorated with Fancy Border and Vertical Scroll Bar, hence additional draw behaviours of Fancy Border and Vertical Scroll Bar is also obtained.



Composite is an object designed as a composition of one-or-more similar objects, all exhibiting similar functionality. This is known as a "has-a" relationship between objects. The key concept is that you can manipulate a single instance of the object just as you would a group of them.

Java AWT's container and component classes, and MenuContainer and MenuComponent classes are examples of a hierarchical Composite relationship.

In the .NET world, some examples are the *TreeNodeCollection* class that represents a collection of *TreeNode* objects in the *TreeView* control; or the *ToolStrip* class in .NET Framework 2.0 provides container for Windows toolbar objects.



Elements of this pattern are component, leaf and composite. The example above models a GUI system. WidgetContainer is again composed of GUI elements. Here the classes that are playing the roles of different elements are:

1. Component: GUIComponent
2. Leaf: Button and Menu
3. Composite: WidgetContainer

Composite - Code

```
//Component
public abstract class GUIComponent {
public abstract void update() ;
public void add(GUIComponent component) {
/* Update the return type based on your custom operation
      signature */
}
public void remove(GUIComponent component) {
/* Update the return type based on your custom operation
      signatur e*/
}
public GUIComponent getChild(int childIndex) {
return null;
//Update - returns child of type GUIComponent
}
```

```
//Leaf - Button
public class Button extends GUIComponent {
public Button() {
}
public void update() {
// Update based on your custom operation signature
}}
//Leaf - Menu
public class Menu extends GUIComponent {
public Menu() {
}
public void update() {
// Update based on your custom operation signature
}}
```

March 18, 2015 | Proprietary and Confidential | - 49 -

IGATE Speed.Agility.Imagination

Here is the code. Note the following:

1. The Component, GUIComponent declares the interface for objects in the composition, and implements the default behavior for the interface common to all classes, when appropriate. The Composite declares an interface for accessing and managing its child Component objects. It defines an interface for accessing a Component object's parent in the recursive structure, and implements it if that's appropriate.
2. The Leaf – Buttona and Menu - represents leaf objects in the GUIComponent. A leaf has no children. It defines behavior for primitive objects in the composition.

Composite – Code Contd.

```
//Composite
public class WidgetContainer extends GUIComponent {
protected java.util.ArrayList children;
private GUIComponent aGUIComponent;
public GUIComponent getChild(int childIndex) {
// Customize the operation based on application needs.
 if (childIndex < children.size())
return (GUIComponent) children.get(childIndex);
else
return null;
return (GUIComponent) children.get(childIndex);
}
public WidgetContainer() {
children = new java.util.ArrayList();
}
```

```
//Composite – Contd.
public void remove(GUIComponent component) {
children.remove(component);
}
public void add(GUIComponent component) {
children.add(component);
}
public void update() {
// Customize the operation based on your application needs.
 for (int i = 0; i < children.size(); i++) {
// Update your code to match the Operation signature.
((GUIComponent)children.get(i)).operation();
}
}
```


IGATE Speed.Agility.Imagination

Here is the code. Note the following:

3. The Composite, WidgetContainer defines behavior for Components with children, stores child Component objects, and implements child-related operations in the GUIComponent interface.

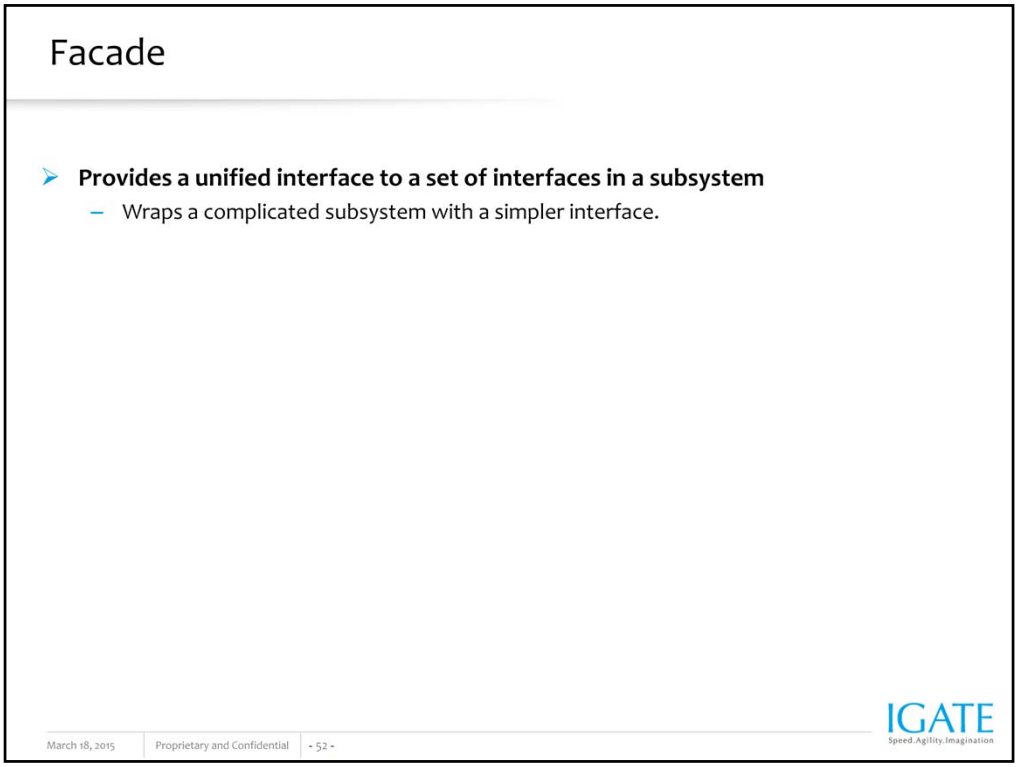
Composite – Code Contd.

```
//Client
public class Client {
        //Code to create leaf nodes, create composition
        GUIComponent button1 = new Button("Button one");
        GUIComponent menu1 = new Menu("Menu one");
        GUIComponent theGUIComponent1 = new WidgetContainer();
        GUIComponent theGUIComponent2 = new WidgetContainer();
        // add leaves to composition
        theGUIComponent1.add(button1);
        theGUIComponent1.add(menu1);
        // add composition to composition
        theGUIComponent2.add(theGUIComponent1);
        theGUIComponent2.update();
        theGUIComponent2.remove(button1);
         theGUIComponent2.update();
}
```

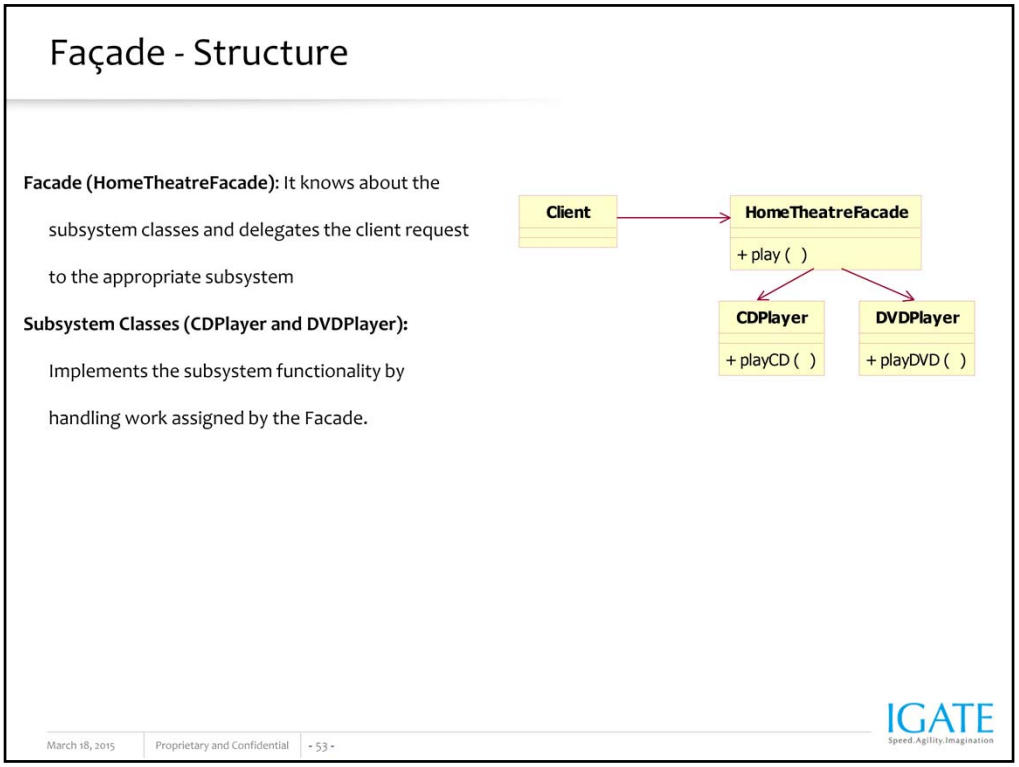

IGATE Speed.Agility.Imagination

Here is the code. Note the following:

4. The client is creating new leaf nodes and composition. The leaves are added to the composition, so is the other composition. Operations defined are invoked using the compositions.



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. It hides the complexities of the system and provides an interface to the client from where the client can access the system.
In the context of Java Persistence clients and the entities and operations of the Java Persistence API, the Façade pattern provides a single interface for operations on a set of entities. Using the Facade pattern can make the code in your Java Persistence clients cleaner. A web component that uses a Facade to access the model tier does not have to be aware of all the details of the APIs for each entity. Neither does the web component need to be aware of the persistence mechanisms being used when accessing the entity managers or transaction managers.
In .NET, Façade is used for calls from Presentation Layer to Business Layer; similarly from Business to Data layer.



Elements of this pattern are Façade & subsystem classes. In the example here, roles played by the classes are:

1. **Façade:** HomeTheatreFacade
2. **Subsystem classes:** CDPlayer and DVDPlayer

Facade - Code

```
//Facade
public class HomeTheatreFacade {
private CDPlayer aCDPlayer;
private DVDPlayer aDVDPlayer;
public void play() {
/* Route the operation to CDPlayer or DVDPlayer depending
      on the requirement i.e. One of the two statements below
  aCDPlayer.playCD();
  aDVDPlayer.playDVD();
*/
}
}
```

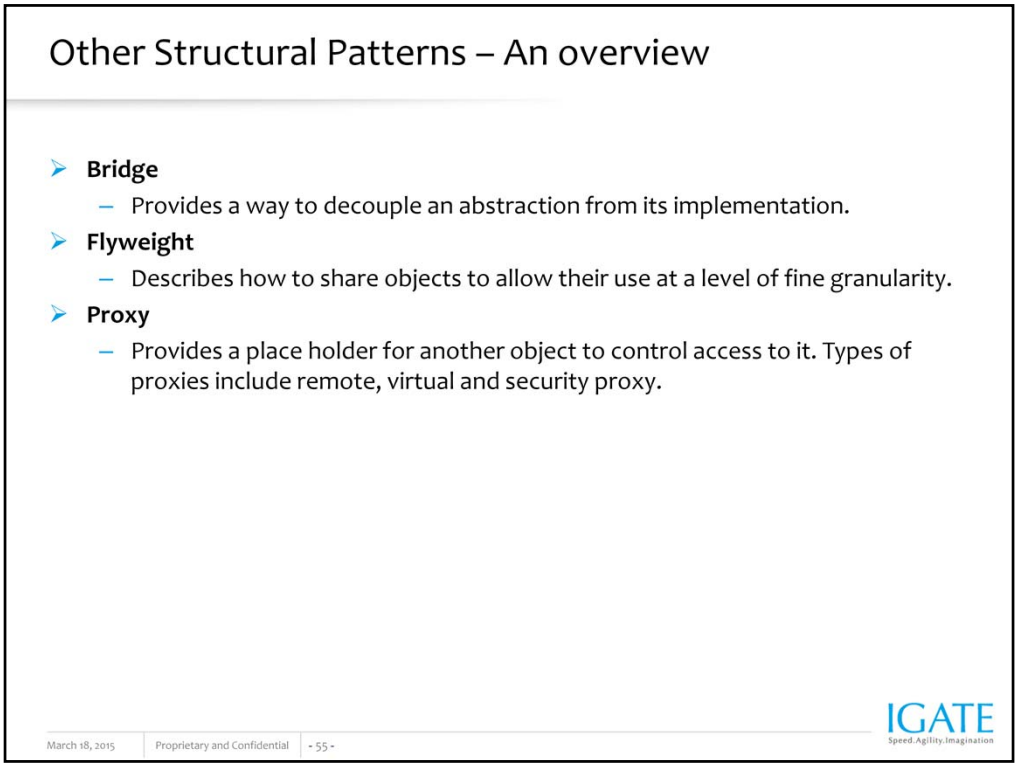
```
//Subsystem classes
public class CDPlayer {
public void playCD() {
//Specific Code
}
}
//Subsystem classes
public class DVDPlayer {
public void playDVD() {
//Specific Code
}
}
```

}

IGATE Speed.Agility.Imagination

Here is the code. Note the following:

1. The Façade, HomeTheatreFacade routes the operation to appropriate subsystem classes
2. The subsystem classes, CDPlayer and DVDPlayer, implement the operation assigned by the Facade



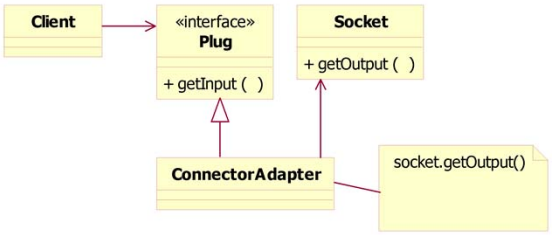
The bridge pattern is a design pattern used in software engineering which is meant to "decouple an abstraction from its implementation so that the two can vary independently" (Gamma et al.). The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost. Each "flyweight" object is divided into two pieces: the state-dependent (extrinsic) part, and the state-independent (intrinsic) part. Intrinsic state is stored (shared) in the Flyweight object. Extrinsic state is stored or computed by client objects, and passed to the Flyweight when its operations are invoked.

The proxy pattern is used when you need to represent a complex with a simpler one. If creation of object is expensive, its creation can be postponed till the very need arises and till then, a simple object can represent it. This simple object is called the "Proxy" for the complex object

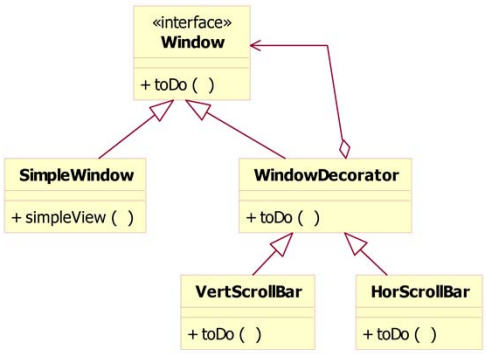
Remote Proxy provides a reference to an object located in a different address space on the same or different machine. Virtual Proxy allows the creation of a memory intensive object on demand. The object will not be created until it is really needed. <Security Proxy>

Structural Patterns - Exercises



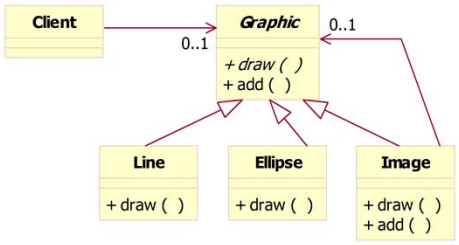
- **Write skeletal code/pseudo code for the above pattern (Adapter)**

Structural Patterns - Exercises



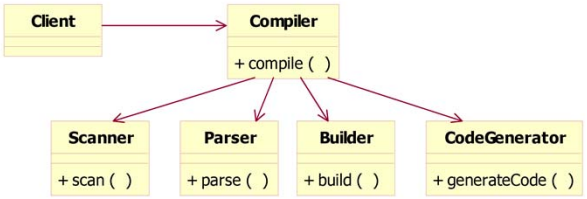
- **Write skeletal code/pseudo code for the above pattern (Decorator)**

Structural Patterns - Exercises

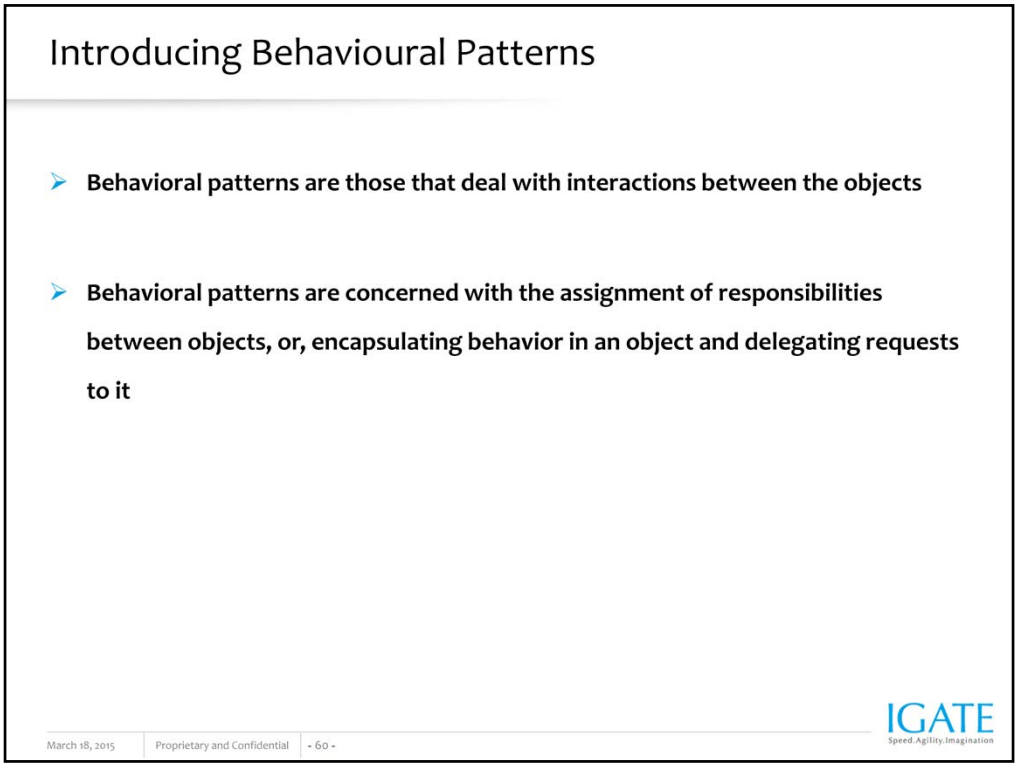


- **Write skeletal code/pseudo code for the above pattern (Composite)**

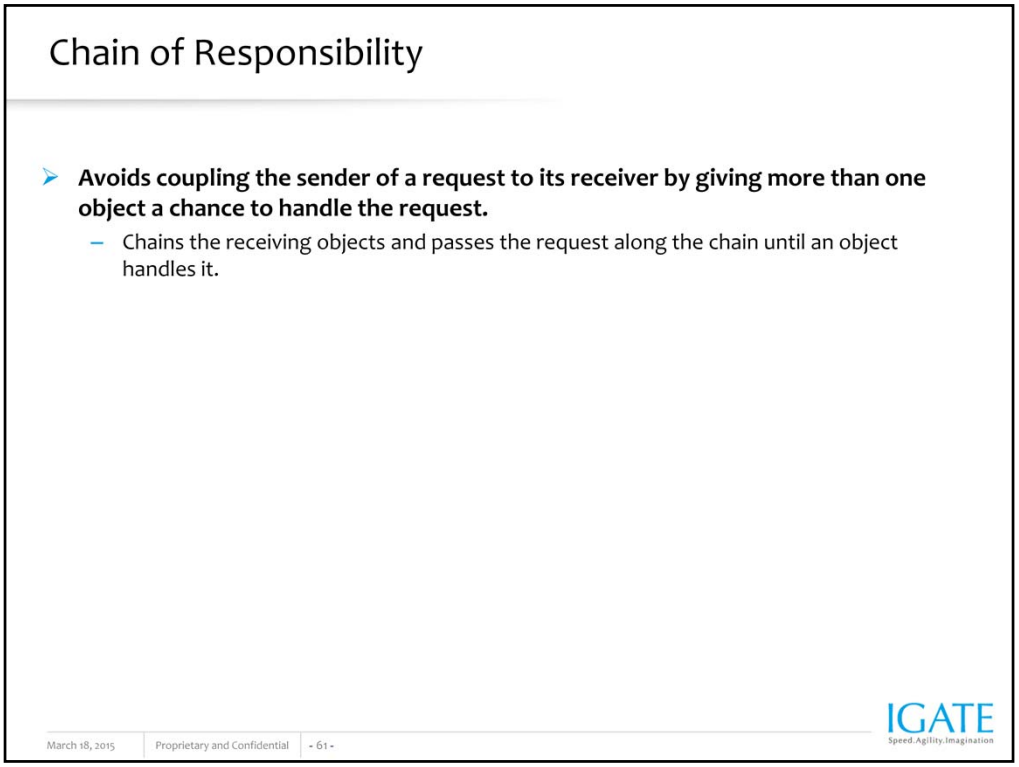
Structural Patterns - Exercises



- **Write skeletal code/pseudo code for the above pattern (Facade)**



Structural Patterns describe how objects and classes can be combined to form larger structures. The difference between class patterns and object patterns is that 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.



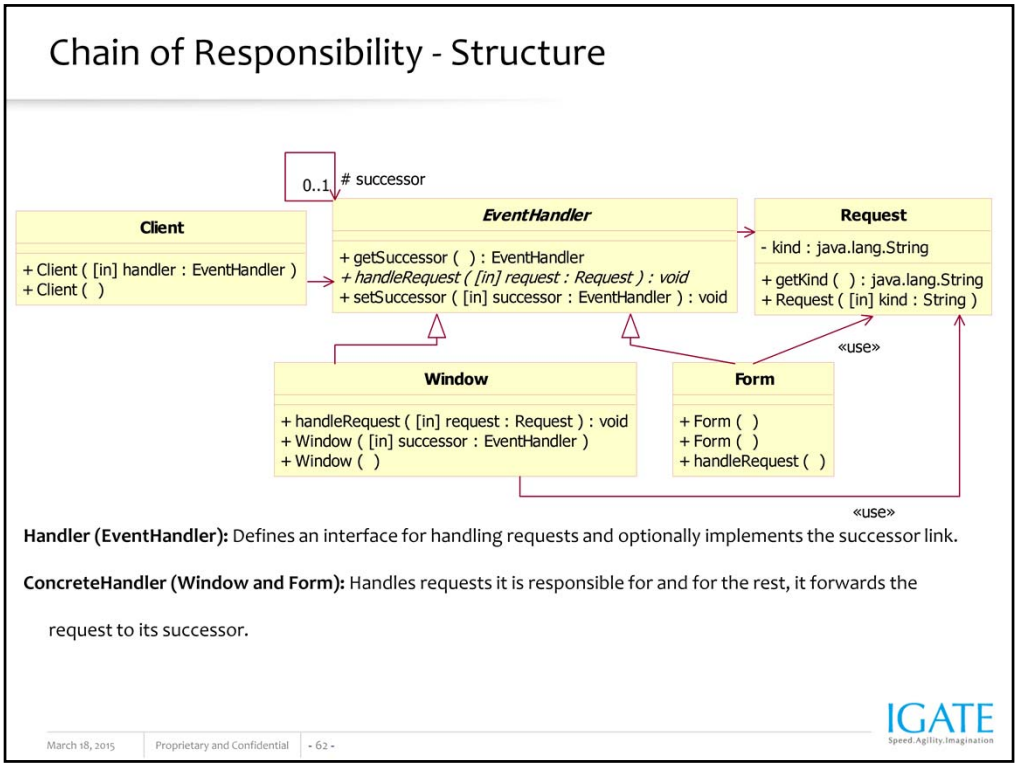
The chain of responsibility pattern decouples the sender of the request from the receiver. The only link between sender and the receiver is the request which is sent. Based on the request data sent, the receiver is picked. This is called "data-driven". The responsibility of handling the request data is given to any of the members of the "chain". If the first link of the chain cannot handle the responsibility, it passes the request data to the next level in the chain, i.e. to the next link.

Java AWT's Container and Component classes participate in a Chain of Responsibility

relationship.

Net framework implements Chain Of Responsibility design pattern for many its internal

mechanisms – for eg. HttpModule.



Elements of this pattern are Handler and ConcreteHandler. Consider event handling as an example. The mouse click event can be chained through the window, form and form_element objects for the event to be handled. In this example, roles played by different classes would be:

1. Handler: EventHandler class
2. ConcreteHandler: Each of the Window, Form and Form_element classes for the event to be handled appropriately

Chain of Responsibility - Code

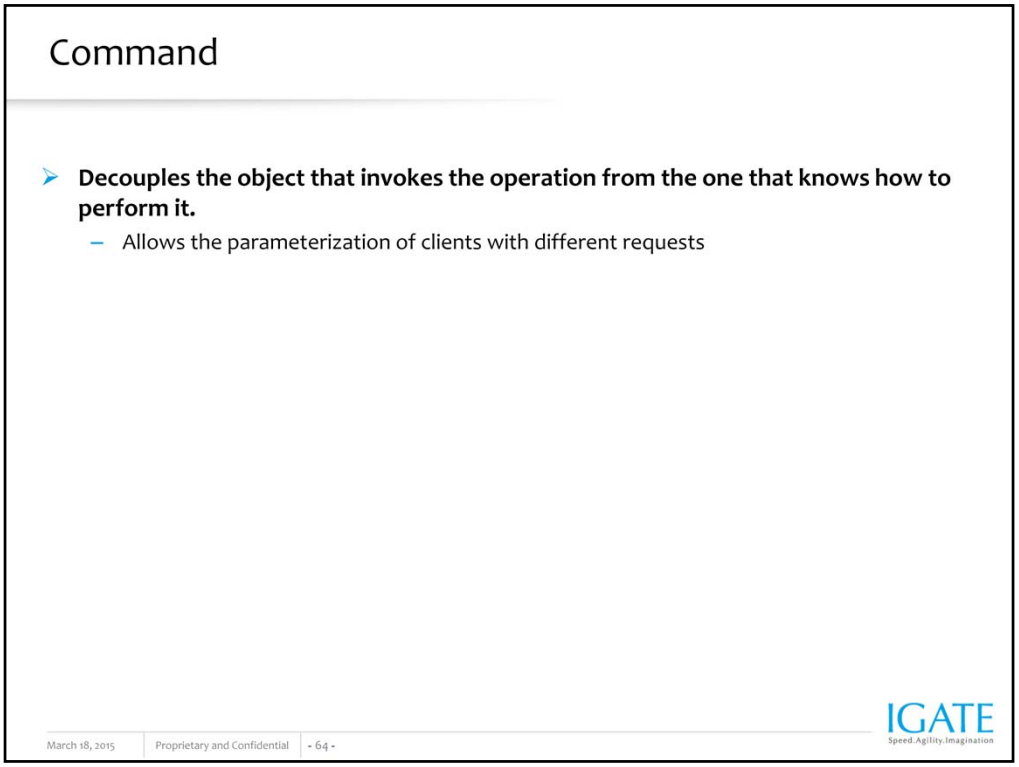
```
//Handler
public abstract class EventHandler {
protected EventHandler successor;
public EventHandler getSuccessor() {
// Return the successor based on application need
 return this.successor;
}
// Operation to be implemented by the ConcreteHandler.
public abstract void handleRequest(Request request);
public void setSuccessor(Handler successor) {
//stores successor
this.successor = successor;
}
}
```

```
//Concrete Handler. Similar code for Form class.
public class Window extends EventHandler {
public Window(Handler successor) {
//Setting reference to successor
this.successor = successor; }
public Window() { }
public void handleRequest(Request request) {
// Handle request else forward the request to successor.
if ( request.getKind().equals("test")) {
// add your codes here to handle the request.}
else { // the successor will handle the request.
this.successor.handleRequest(request);
} }
}
```


IGATE Speed.Agility.Imagination

Here is the code. Note the following:

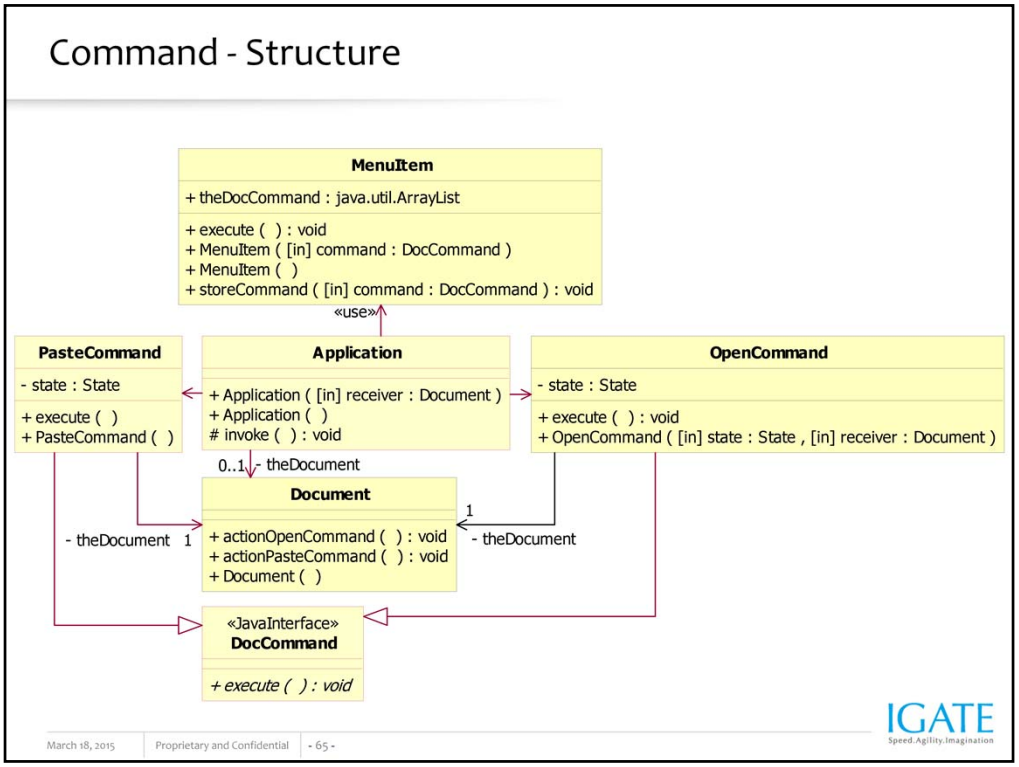
1. The EventHandler defines the interface for handling requests.
2. The Window (ConcreteHandler) has access to the successor. If it cannot handle the request, the same is passed on to the successor.



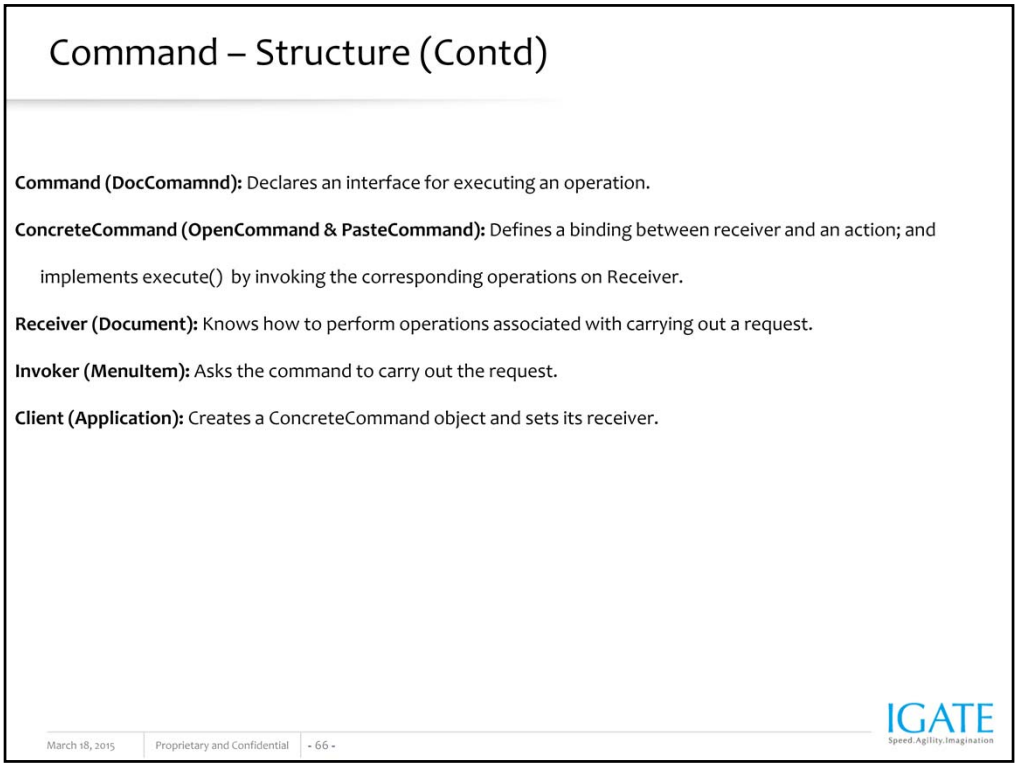
Command decouples the object that invokes the operation from the one that knows how to perform it. The client invokes a particular module using a command. This request gets propagated as a command. The command request maps to particular modules. According to the command, a module is invoked.

In Swing, an Action is a command object. In addition to the ability to perform the desired command, an Action may have an associated icon, keyboard shortcut, tooltip text, and so on. A toolbar button or menu item component may be completely initialized using only the Action object.

There is extensive support for the Command pattern in Windows Presentation Foundation, making it easy to separate the implementation of a command from all the UI elements that provide access to it.



Here is the structure of the Command pattern. Details are on the next slide.



Elements of this pattern are Command, Concrete Command, Client, Invoker and Receiver. Take an example of word processor application. In this example, roles played by different classes would be:

1. Command: DocCommand interface for executing operation.
2. ConcreteCommand: These are commands like OpenCommand and PasteCommand.
3. Receiver: This could be the document or the application.
4. Invoker: menuItem
5. Client: Application

Command - Code

```
//Command
public interface DocCommand {
/* Operation must be implemented by the
      ConcreteCommand.*/
public void execute();
}

//Concrete Command – Similar code for PasteCommand
public class OpenCommand implements DocCommand {
private State state;
private Document theDocument;
//Continued on other side.
}
```

```
//Concrete Command Contd.
public void execute() {
/* Implement "execute" by invoking the corresponding
    operations(s) on Receiver. */
this.state.set();
this.theDocument.actionOpenCommand();
}
public OpenCommand(State state, Document receiver) {
//Can customise this constructor
this.state = state;
this.theDocument = receiver;
}
public OpenCommand() {
}}
```


IGATE Speed.Agility.Imagination

Here is the code. Note the following:

1. The DocCommand declares an interface for executing operations.
2. The execute() operation of the OpenCommand needs to invoke corresponding operations on receiver i.e. document.

Command – Code Contd.

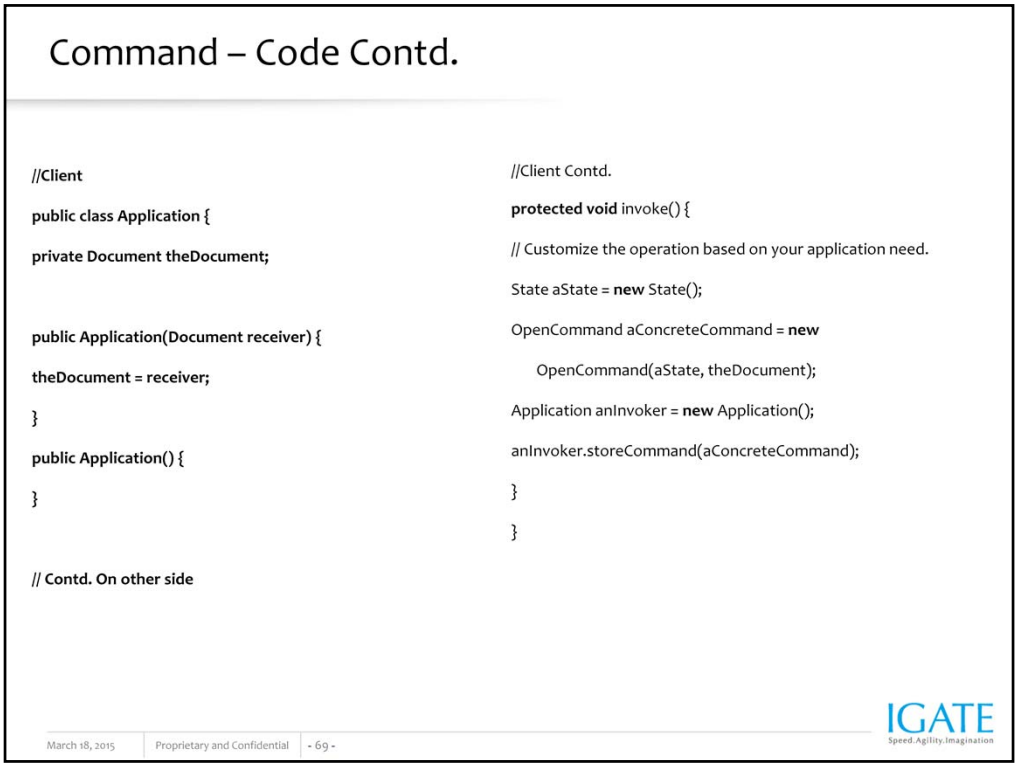
```
//Receiver
public class Document {
public void actionOpenCommand() {
//This performs the action }
public void actionPasteCommand() {
//This performs the action }
public Document () {
}}

//Invoker
public class MenuItem {
public java.util.ArrayList theDocCommand;
public MenuItem() {
theDocCommand = new java.util.ArrayList(); }
//Continued on other side
```

```
//Invoker Contd.
public MenuItem(DocCommand command) {
// Relace based on your applicaton needs.
 this.theDocCommand = new java.util.ArrayList();
this.theDocCommand.add( command);
}
public void storeCommand(DocCommand command) {
if(this.theDocCommand != null) {
this.theDocCommand.add(command); }}
public void execute() {
if ( theDocCommand.size() > 0) {
for ( int i=0;i<theDocCommand.size();i++) {
((DocCommand)theDocCommand.get(i)).execute(); }}
}}
```


IGATE Speed.Agility.Imagination

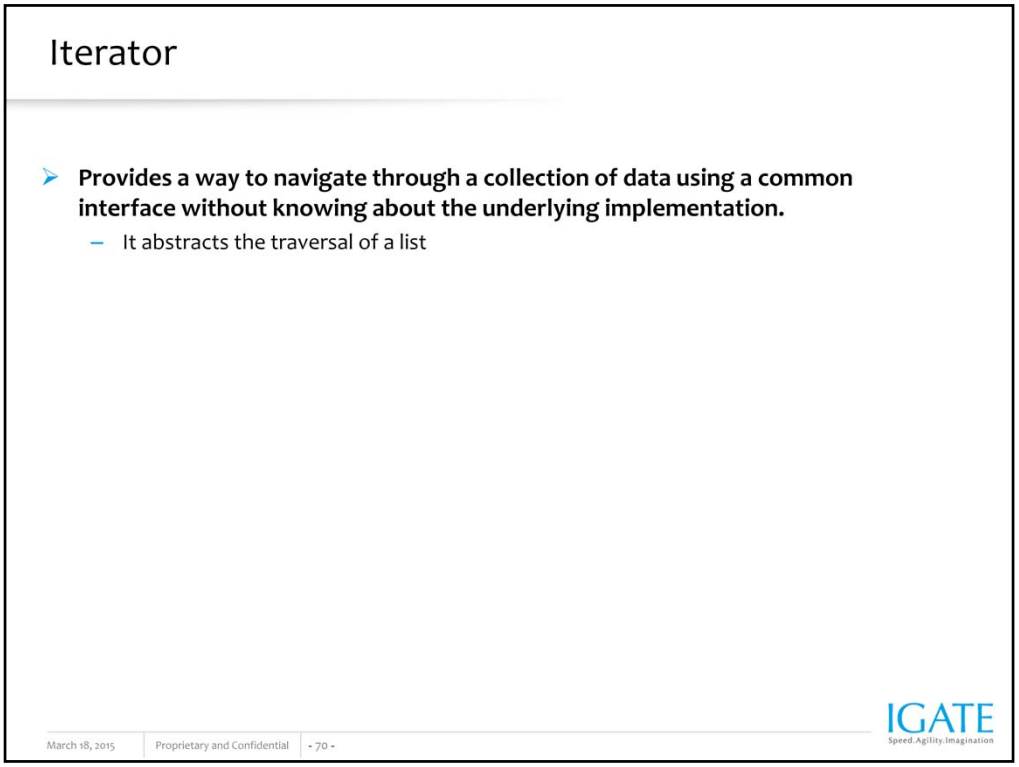
Here is the code. Note the following:

1. The Document class would have operations that actually perform the commands open and paste.
2. The constructor within MenuItem stores the ConcreteCommand object reference into an ArrayList. The storeCommand operation stores a ConcreteCommand object. This can be used for undo/redo (command history) mechanism. The execute() operation performs the actions associated with the ConcreteCommand. This is the place where Invoker asks the command to carry out the requests - this operation can be customised based on your application needs.



Here is the code. Note the following:

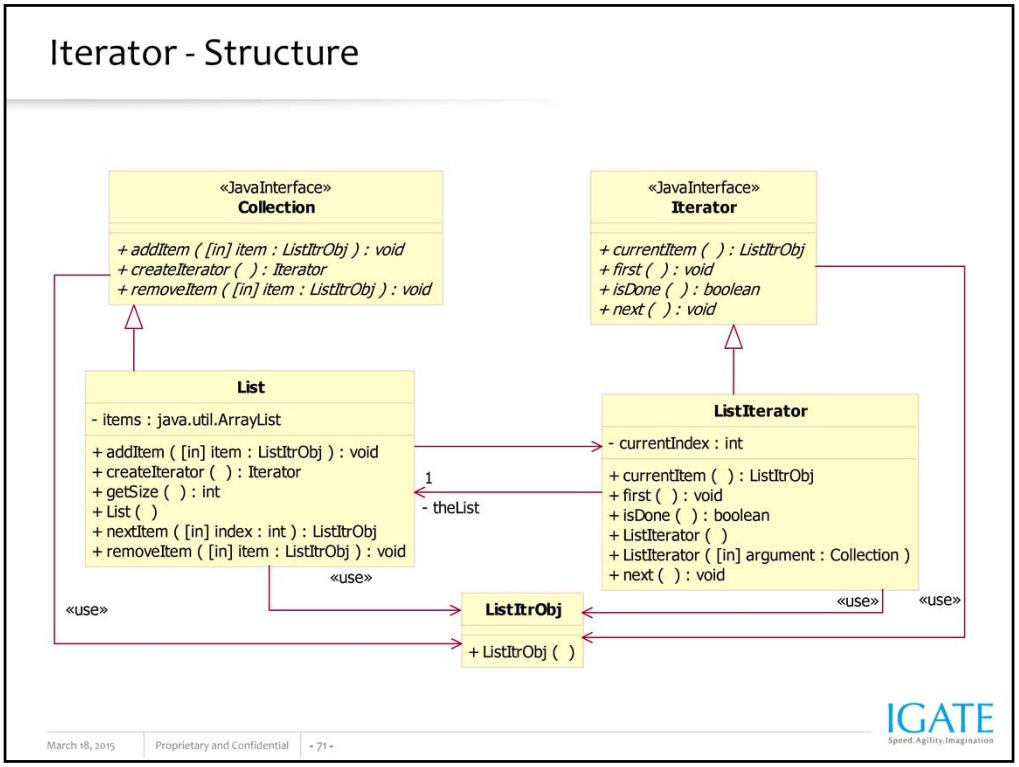
1. The Application i.e. client's constructor sets the Document object reference. The invoke() operation creates a ConcreteCommand object, specifying a Receiver. The invoke() operation shows an example of using the command pattern.



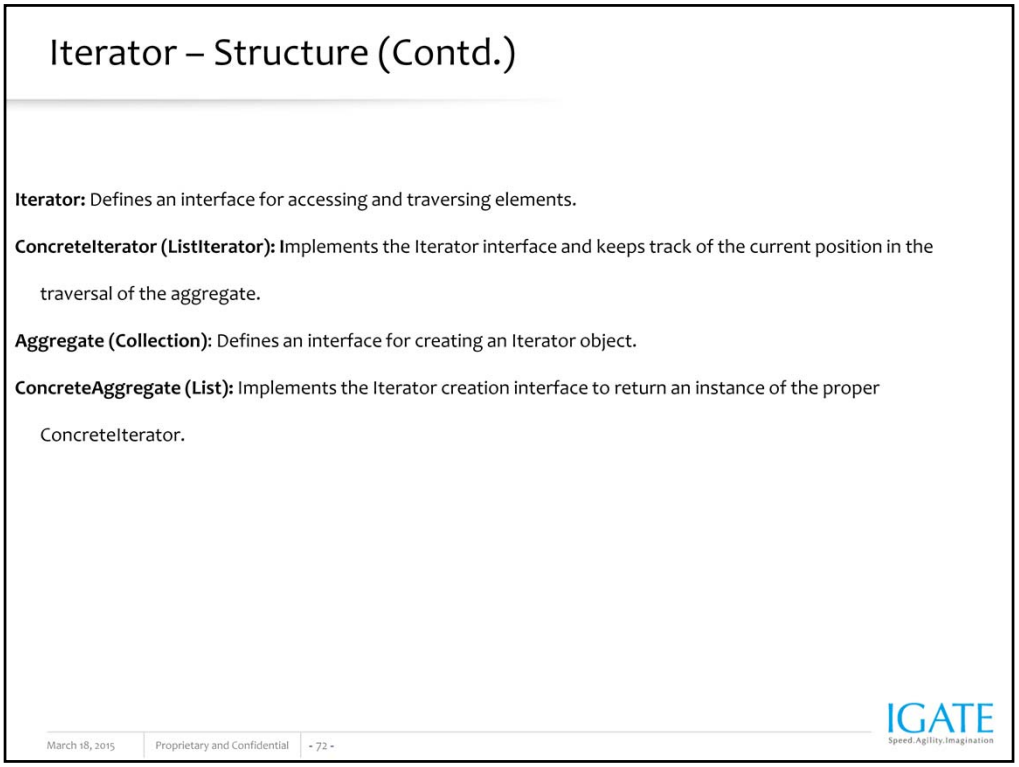
The Iterator pattern allows you to navigate through a collection of data using a common interface without knowing about the underlying implementation. The key idea in this pattern is to take the responsibility for access and traversal out of the collection object and put it into an **iterator** object. The Iterator class defines an interface for accessing the elements. It is responsible for keeping track of the current element and it knows which elements have been traversed already.

The class java.util.Enumeration is an example of the Iterator pattern.

Similarly in .NET, IEnumerable and IEnumerator interfaces allow implementation of the Iterator pattern.



Elements of this pattern are Iterator, ConcreteIterator, Aggregate and ConcreteAggregate. They are briefly described on the next slide.



Consider any example of traversing over a collection. An iterator pattern can be used here.
1. Iterator: An interface by the same name, defining all the operations needed for accessing and traversing.
2. ConcreteIterator: ListIterator class that provides implementation to the interface operations.
3. Aggregate: Collection - This maps to the Items in the collection.
4. ConcreteAggregate: List - This maps to the actual collection object.

Iterator - Code

```
//Iterator
public interface Iterator {
public ListItrObj currentItem();
public void first();
public boolean isDone();
public void next();
}

//Aggregate
public interface Collection {
public void addItem(ListItrObj item);
public Iterator createIterator();
public void removeItem(ListItrObj item);
}
```

```
//Concrete Iterator
public class ListIterator implements Iterator {
private int currentIndex;
private List theList;
public void first() {
this.currentIndex  = 0; }
public void next() {
this.currentIndex++; }
public boolean isDone() {
boolean bIsDone = false;
if ( currentIndex == theList.getSize() ) {
bIsDone = true; }
return bIsDone; }
```


Here is the code. Note the following:
The Iterator is an interface which defines the operations needed for accessing/traversing.
The Aggregate i.e. Collection defines an interface for creating an Iterator object.
The Concrete Iterator provides an implementation for the operations defined in the Iterator interface.

Iterator – Code Contd.

```
//Concrete Iterator Contd.
public ListItrObj currentItem() {
if( currentIndex < theList.getSize())
 return theList.nextItem(currentIndex);
else
 return null;
}
public ListIterator() {
currentIndex = 0;
 }
public ListIterator(Collection argument) {
currentIndex = 0;
}
}
```

```
//Concrete Aggregate
public class List implements Collection {
private java.util.ArrayList items;
public Iterator createIterator() {
// Example: return new ConcreteIterator(this);
return null;
}
public List () {
items = new java.util.ArrayList();
/* Regarding the number of ConcreteIterators, set up the
    elements of the (collection of)  items with the instance(s) of
    ConcreteIterator(s). */
}
```


IGATE Speed.Agility.Imagination

Here is the code. Note the following:
The createIterator operation of ConcreteAggregate creates a new Iterator. The code needs to be customized based on application needs.

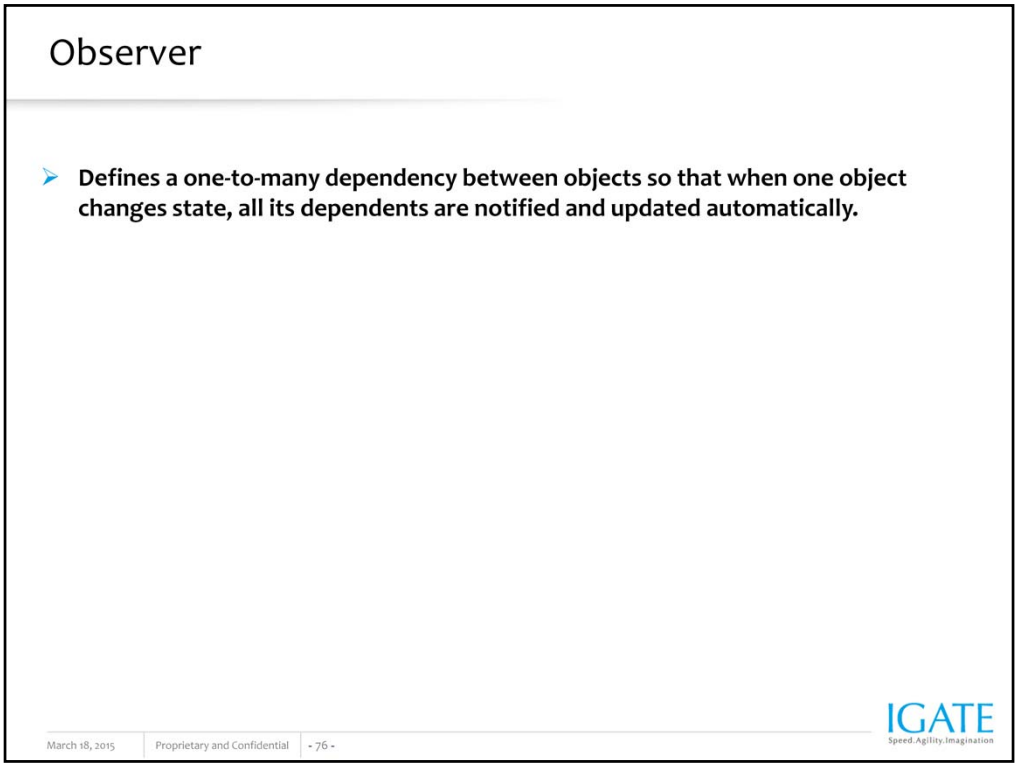
Iterator – Code Contd.

```
//Concrete Aggregate Contd.
public void addItem(ListItrObj item){
items.add(item);
}
public void removeItem(ListItrObj item) {
items.remove(item);
}
public int getSize() {
return items.size();
}
public ListItrObj nextItem(int index) {
return (ListItrObj) items.get(index);
}
}
```

```
//Iterator Object
public class ListItrObj {
public ListItrObj() {
}
}
```


IGATE Speed.Agility.Imagination

Here is the code. Note the following:
The IteratorObject is used by the Iterator pattern to define the collection of items of the ConcreteAggregate class. The representation of the object is custom for your application.

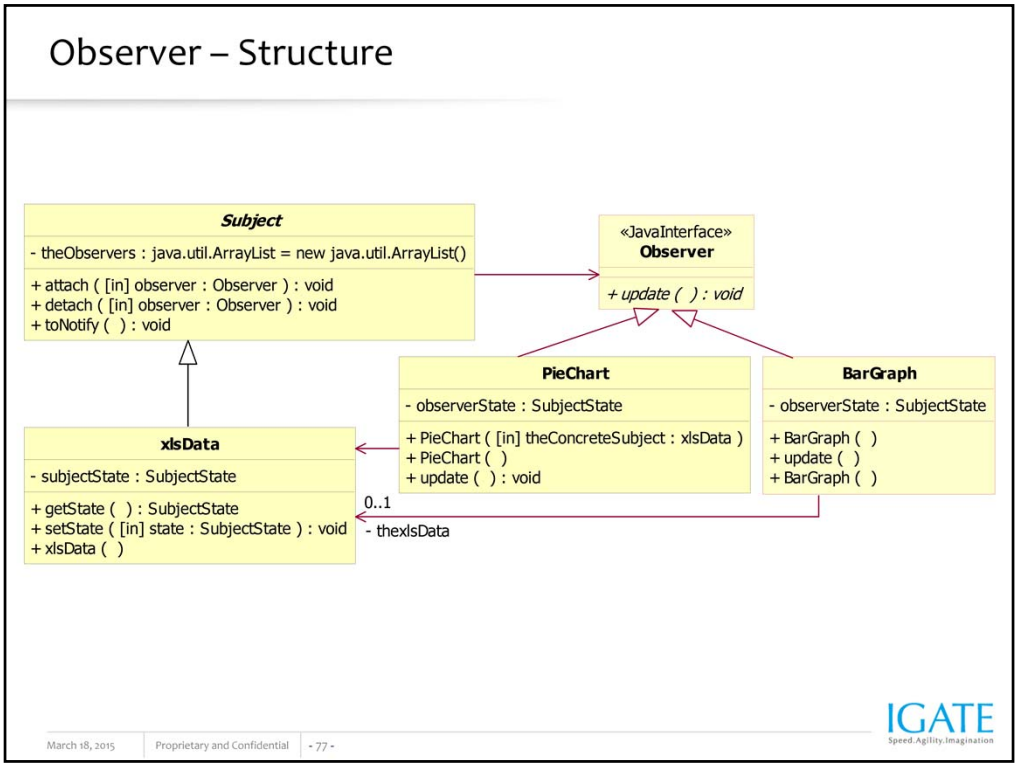


The observer pattern (sometimes known as publish/subscribe) is a design pattern used in computer programming to observe the state of an object in a program.

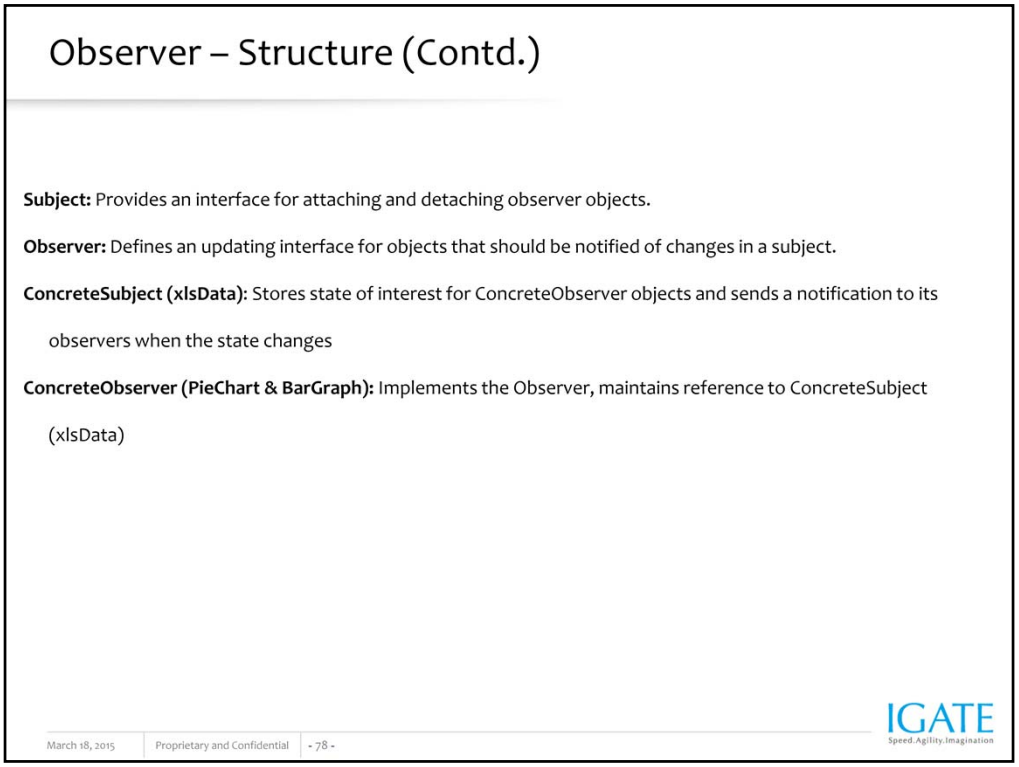
The essence of this pattern is that one or more objects (called observers or listeners) are registered (or register themselves) to observe an event which may be raised by the observed object (the subject). (The object which may raise an event generally maintains a collection of the observers.)

Java provides direct support for implementing the Observer pattern via the java.util.Observer interface and the java.util.Observable class in the Java API. Java AWT ImageObserver class uses subjects and observers to implement a variant of the Observer pattern.

In .NET, ICanonicalObserver interface and CanonicalSubjectBase class can be used to implement Observer pattern.



Elements of this pattern are Subject, Observer, ConcreteSubject and ConcreteObserver. Details are on the next slide.



Consider an example where updates in the data from an xls has to be reflected in the graphs. Roles played by different classes would be:
Subject: An abstract class by the same name Subject that provides the interface
Observer: An interface by the same name Observer defining operations to take care of the updates
ConcreteSubject: xlsData - The document containing the data in our example
4. ConcreteObserver: PieChart & BarGraph - The different graphs which need to get updated

Observer - Code

```
//Subject
public abstract class Subject {
private java.util.ArrayList theObservers = new
      java.util.ArrayList();
public void attach(Observer observer) {
theObservers.add(observer); }
public void detach(Observer observer) {
theObservers.remove(observer); }
public void toNotify() {
for (int i=0; i < theObservers.size(); i++) {
((Observer) theObservers.get(i)).update();
} } }
```

```
//Concrete Subject
public class xlsData extends Subject {
private SubjectState subjectState;
public xlsData() {
}
public SubjectState getState() {
return this.subjectState;
}
public void setState(SubjectState state) {
subjectState = state;  // set Subject's State
// notify all observers to update the Subject's state.
toNotify();
}}
```


IGATE Speed.Agility.Imagination

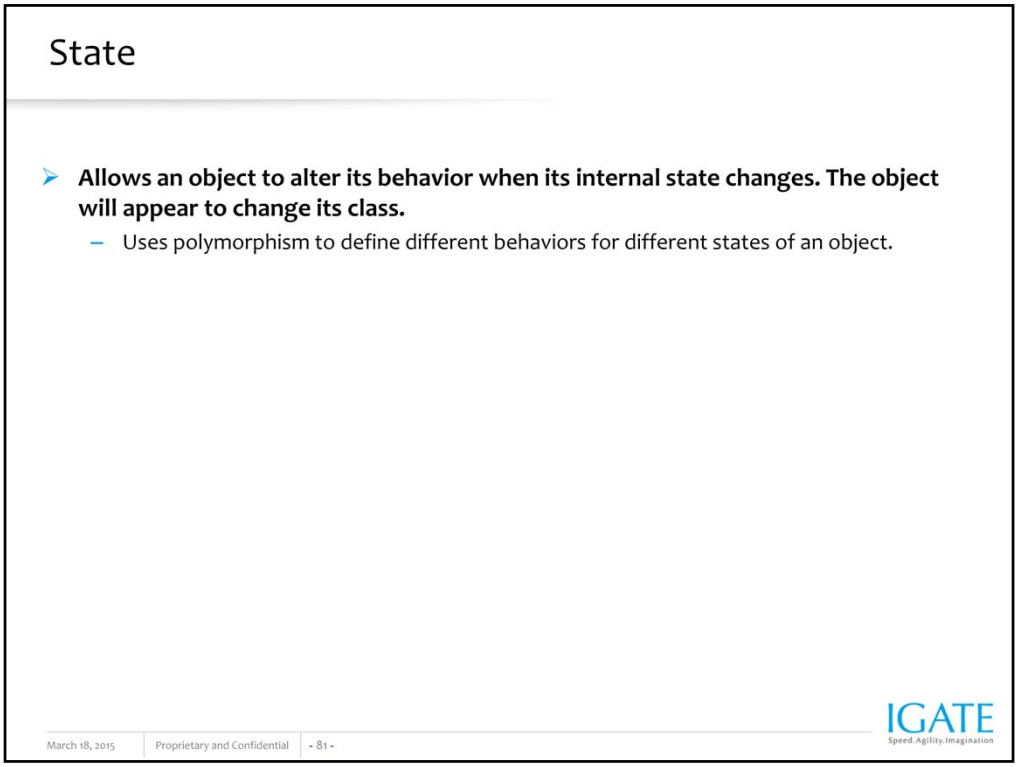
Here is the code. Note the following:
The Subject, an abstract class, stores a reference to its Observer objects, and provides an interface for attaching and detaching Observer objects. The toNotify() operation is to notify all observers whenever the subject undergoes a change in its state. These operations (attach, detach and toNotify) can be customized based on application needs.
The ConcreteSubject, xlsData in this case, manages the attaching and detaching of ConcreteObserver objects. It stores states that are of interest to the ConcreteObserver objects, and notifies the ConcreteObserver objects when its state changes.

Observer – Code Contd.

```
//Observer
public interface Observer {
public void update();
}
```

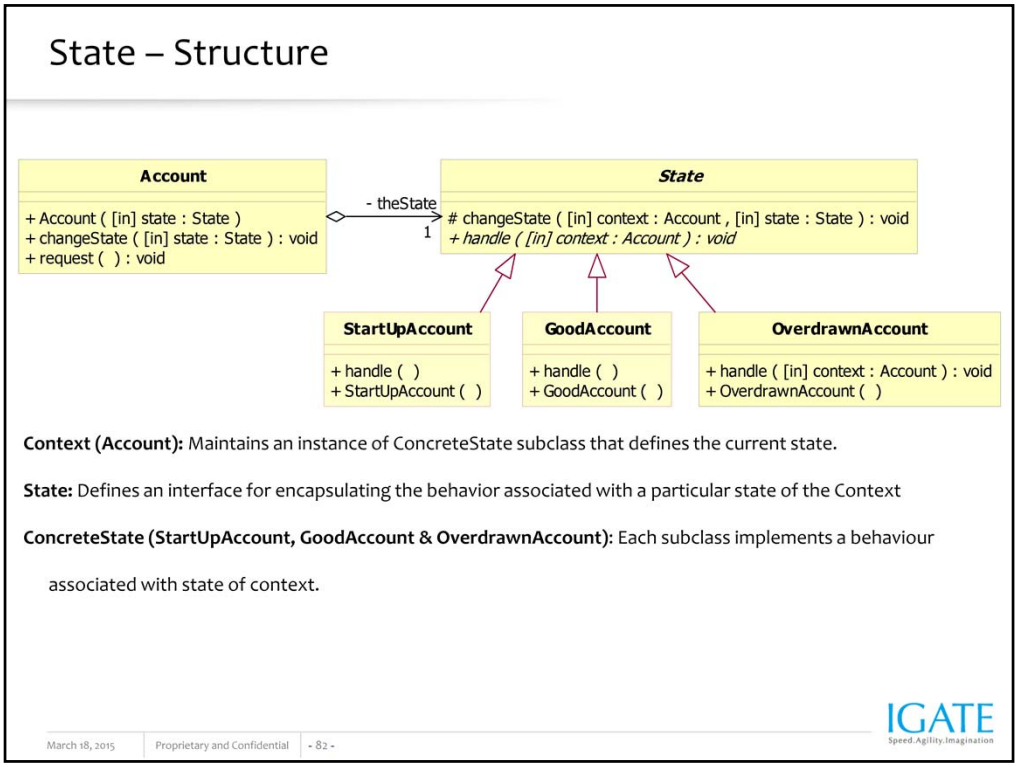
```
//Concrete Observer – Similar Code for BarGraph
public class PieChart implements Observer {
private SubjectState observerState;
private xlsData thexlsData;
public PieChart(xlsData theConcreteSubject) {
this.thexlsData = theConcreteSubject;
}
public void update() {
/* Each observer will query the subject to synchronize its state with the
    subject's state.*/
 observerState = thexlsData.getState();
}
public PieChart() {
}
```


Here is the code. Note the following:
3. The update operation of the Observer interface requests the Observer object to update itself to reconcile its state with that of the Subject object.
4. This update operation of the ConcreteObserver (BarGraph and PieChart) updates the ConcreteObserver's state to be consistent with the ConcreteSubject's state.



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.

In both Java & .NET environments state management is an area where the State pattern comes into picture.



Elements of this pattern are Context, State and ConcreteState subclasses. Consider a bank which has different rules and facilities applicable for accounts that are new, accounts that are overdrawn and accounts that are in good standing. Roles played by different classes would be:

Context: Account Class

2. State: An abstract superclass State to encapsulate behaviour associated with each state of the context

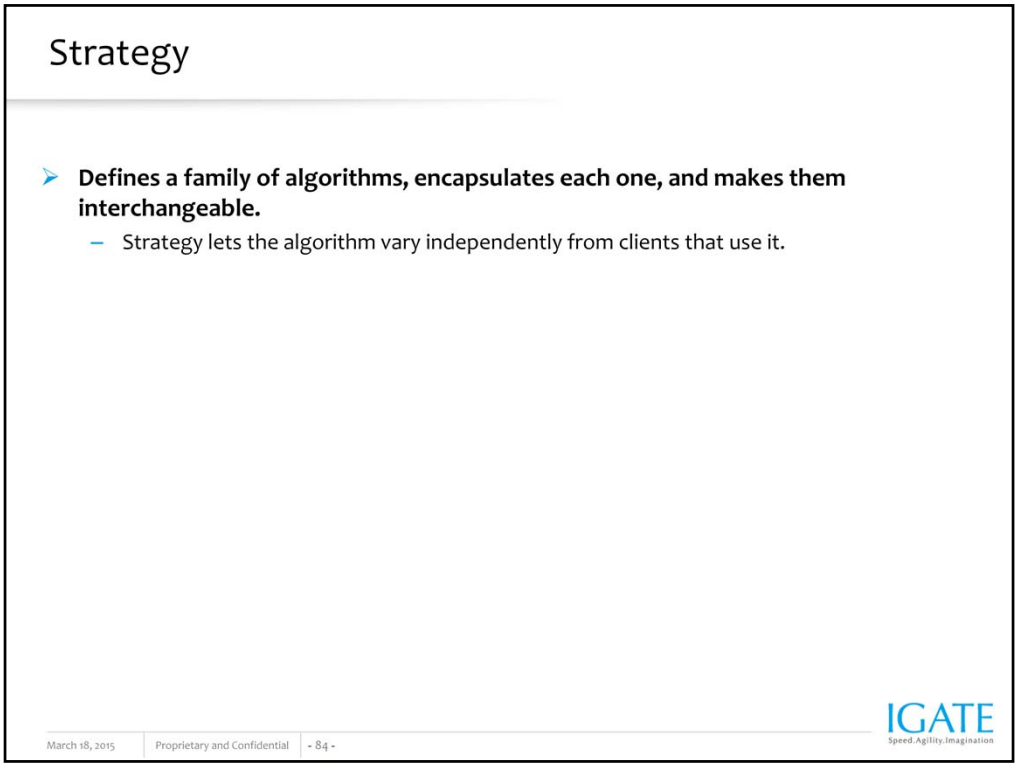
3. ConcreteState subclasses:Each of the subclasses StartUpAccount, OverdrawnAccount and GoodAccount

State - Code

```
//Context
public class Account {
private State theState;
public void request() {
// The request defines the interface of interest to clients.
this.theState.handle(this);
}
public Account(State state) {
this.theState = state; }
public void changeState(State state) {
// The state is any object that can describe your system.
 this.theState = state;
} }
```

```
//State
public abstract class State {
public abstract void handle(Account context);
protected void changeState(Account context, State state) {
// Customize the operation based on your application needs.
 context.changeState(state);
}}
//Concrete State
public class GoodAccount extends State {
public void handle(Account context) {
// Be sure you provide the specific state class in this call.
super.changeState(context, this); }
public GoodAccount() {
} }
```


Here is the code. Note the following:
The class Context defines the interface of interest to clients. As an example here, the operation request defines the interface of interest to clients. The changeState operation modifies the state of the Context object. These operations can be customized based on application need.
The abstract operation handle must be implemented by the ConcreteState. The changeState operation changes the state of a State object.
The ConcreteState implements the behaviour associated with a state of the Context.

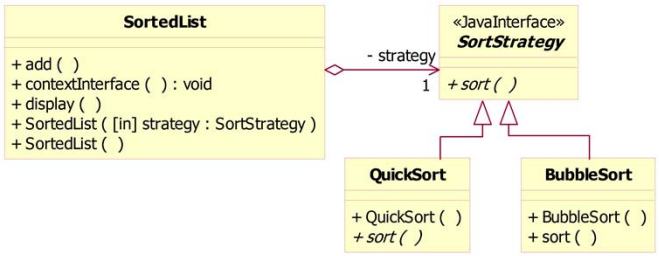


The strategy pattern is intended to provide a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable. The strategy pattern lets the algorithms vary independently from clients that use them.
The strategy pattern uses composition instead of inheritance.

Java AWT LayoutManagers work on the lines of Strategy pattern. There is an abstract LayoutManager class/interface that specifies the policy with which each layout management strategy must comply; and then there are multiple concrete layout manager classes for things like GridLayout, FlowLayout, etc. which provide different behavioral mechanisms for implementing a layout strategy.

The collection classes like ArrayList make use of Strategy for the sorting algorithms. Also List<T> (.NET 2.0) makes heavy use of the Strategy pattern.

Strategy – Structure



Context (SortedList): Maintains a reference to Strategy object. Defines interface that lets Strategy access its data.

Strategy (SortStrategy): Defines an interface common to supported algorithms. Context uses this interface to call algorithm defined by ConcreteStrategy.

ConcreteStrategy (QuickSort & BubbleSort): Implements the algorithm specified in the interface

Elements of this pattern are Context, Strategy (SortStrategy) and ConcreteStrategy (QuickSort and BubbleSort). In this example, different sorting strategies are to be applied to the List. Roles played by different classes would be:

1. Context: SortedList
2. Strategy: SortStrategy
3. ConcreteStrategy:QuickSort and BubbleSort

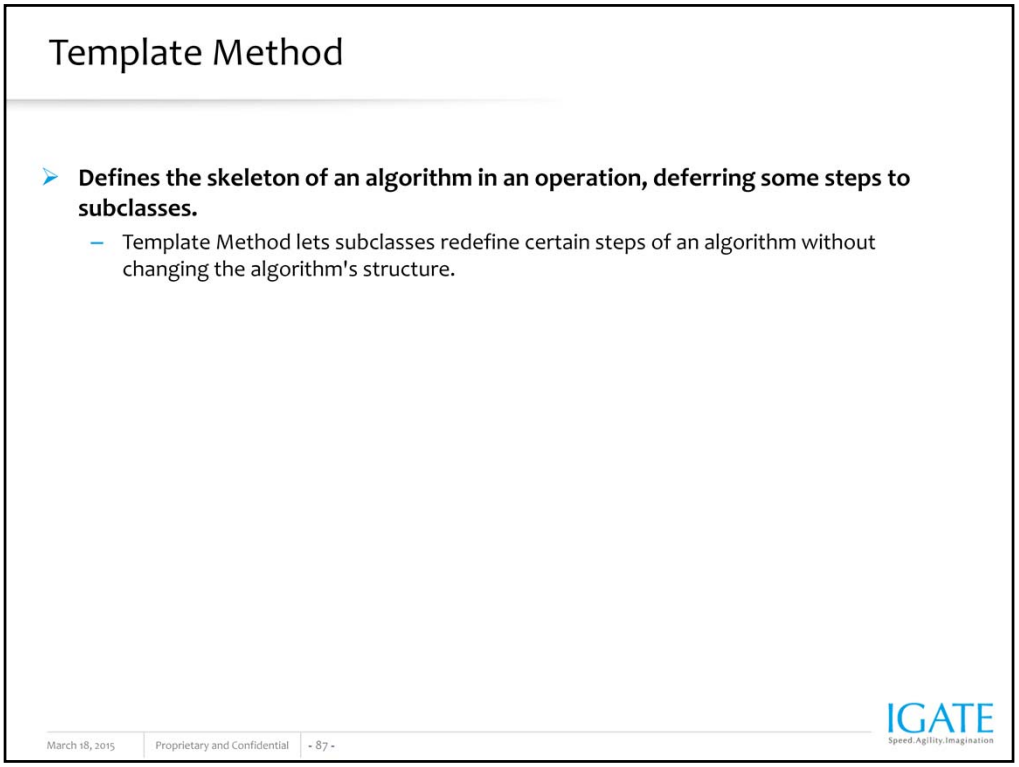
Strategy - Code

```
//Context
public class SortedList {
private SortStrategy strategy;
public void sort() { }
public void add() { }
public void display() { }
public void contextInterface() {
// Add code for the AlgorithmInterface signature.
// this.strategy.sort();
}
public SortedList(SortStrategy strategy) {
this.strategy = strategy; }
public SortedList() { }
}
```

```
//Strategy
public interface SortStrategy {
public void sort();
}
//Concrete Strategy
public class QuickSort implements SortStrategy {
public void sort() {
// Implements the Strategy algorithm
}
public QuickSort() {
}
}
```


IGATE
Speed.Agility.Imagination

Here is the code. Note the following:
The contextInterface operation provides access to the Context object attributes. The code you add to accomodate the AlgorithmInterface signatures can be any operation(s) you define or map to/from your model. The algorithm uses data that clients shouldn't know about . The overloaded SortedList operation creates a Context object, storing the strategy & returning the created Context object.
The Strategy interface defines operations to be supported by the concrete strategy.
The ConcreteStrategy implements the strategy algorithm.



A template method 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 but 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.

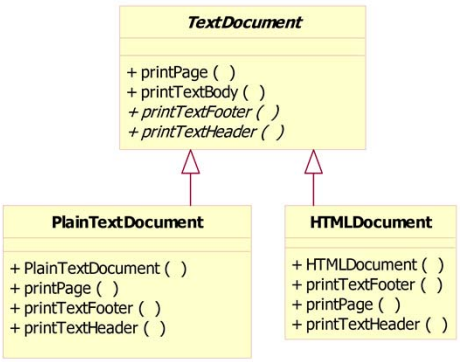
Java.io has a read() method in InputStream that subclasses must implement and is used by the template method read (byte b[], int off, int len).

Custom controls in .NET make good use of Template method - The main algorithm for how a control should be loaded, rendered, and unloaded is contained in the control base class.

Template – Structure

Abstract Class (TextDocument): Defines abstract primitive operations and implements a template method defining skeleton of algorithm.

Concrete Class (Plain Text Document and HTMLDocument): Implements the primitive operations to carry out subclass specific steps of algorithm



Elements of this pattern are Abstract Class & Concrete Classes. In this example, printing of Plain Text Documents and HTML Documents need to be handled separately. Roles played by different classes would be:

1. AbstractClass: TextDocument
2. ConcreteClass: PlainTextDocument and HTMLDocument

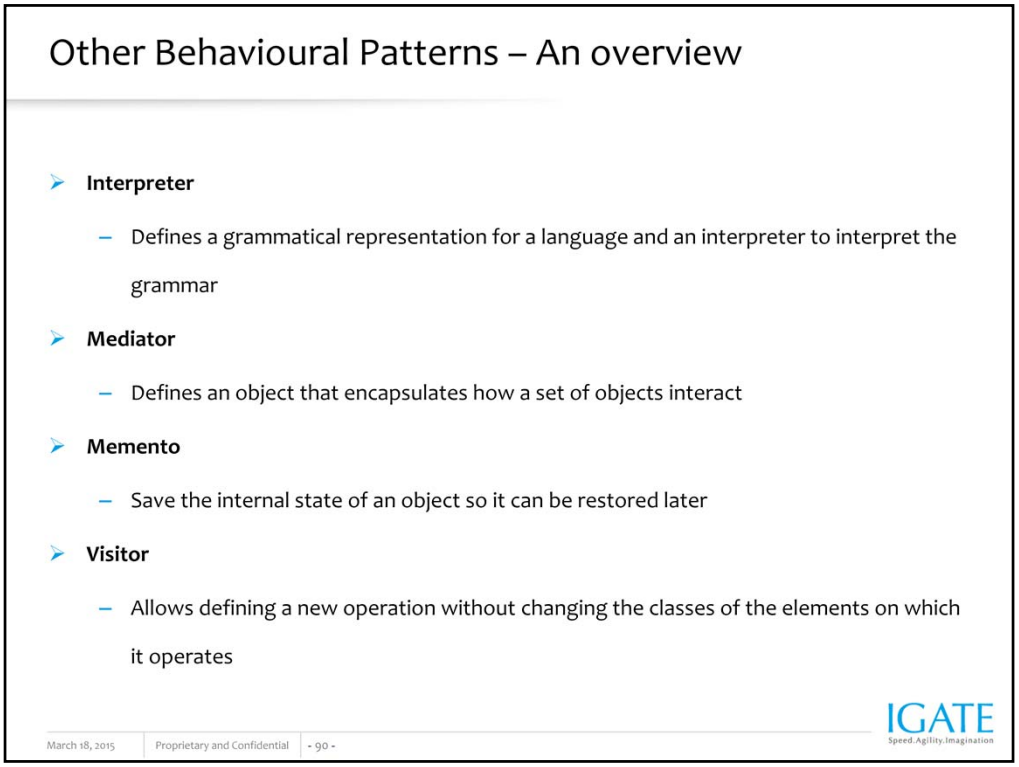
Template - Code

```
//Abstract Class
public abstract class TextDocument {
public abstract void printTextHeader();
public abstract void printTextFooter();
public void printTextBody() {
}
public void printPage() {
/* Add your code and call the primitive operations.  The
      primitive operation is your custom operation
      defined/selected  in/from  the model. */
// this.primitiveOperation();
}
}
```

```
//Concrete Class
public class HTMLDocument extends TextDocument {
public HTMLDocument() {
}
public void printTextFooter() { //Provide implementation
}
public void printTextHeader() {//Provide implementation
}
public void printPage() {
/* If multiple primitiveOperations are applied, then copy/paste the
    code line below to complete the task.*/
// this.primitiveOperation();
}
}
```


IGATE Speed.Agility.Imagination

Here is the code. Note the following:
The abstract class has abstract operations printTextHeader and printTextFooter that needs to be implemented by the concrete classes. The printPage is the Template Method that provides the skeletal algorithm using the abstract primitive operations. The concrete class provides implementation to a primitive operation to perform a step in the algorithm defined in the AbstractClass.



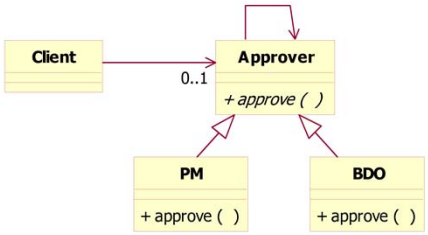
The Interpreter pattern discusses: defining a domain language (i.e. problem characterization) as a simple language grammar, representing domain rules as language sentences, and interpreting these sentences to solve the problem. The pattern uses a class to represent each grammar rule.

The mediator pattern promotes loose coupling between interacting classes by being the only class that has detailed knowledge of the operations of other classes. Classes send messages to the mediator when needed and the mediator passes them on to any other classes that need to be informed.

The memento pattern is a design pattern that provides the ability to restore an object to its previous state.

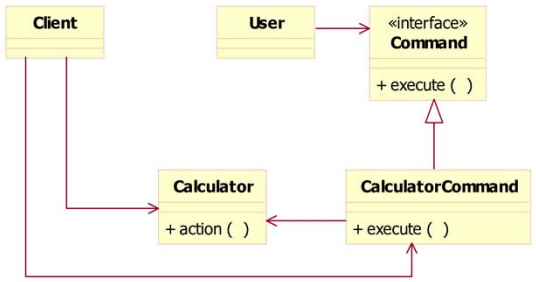
Using a Visitor pattern allows you to decouple the classes for the data structure and the algorithms used upon them. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures.

Behavioural Patterns - Exercises



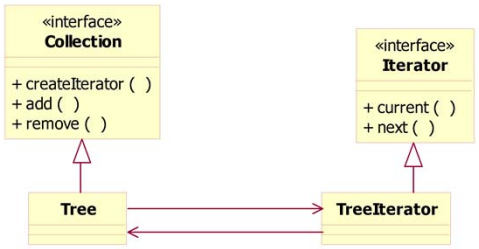
- **Write skeletal code/pseudo code for the above pattern (Chain of Responsibility)**

Behavioural Patterns - Exercises



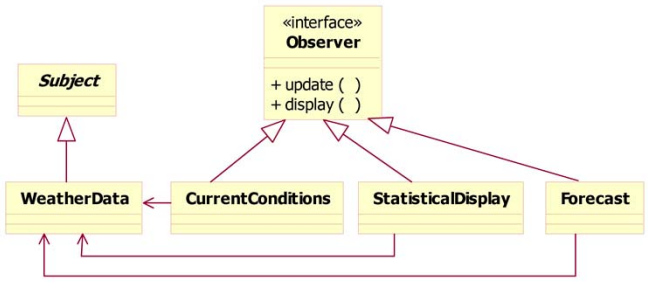
- Write skeletal code/pseudo code for the above pattern (Command)

Behavioural Patterns - Exercises



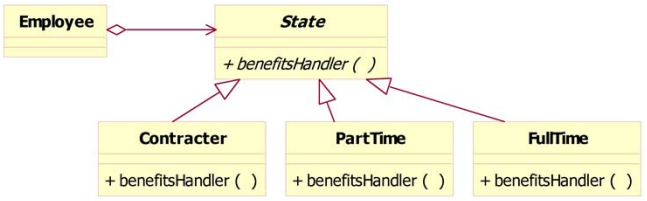
- **Write skeletal code/pseudo code for the above pattern (Iterator)**

Behavioural Patterns - Exercises



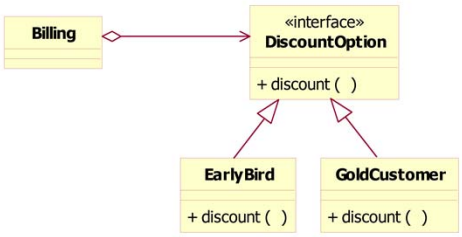
- Write skeletal code/pseudo code for the above pattern (Observer)

Behavioural Patterns - Exercises



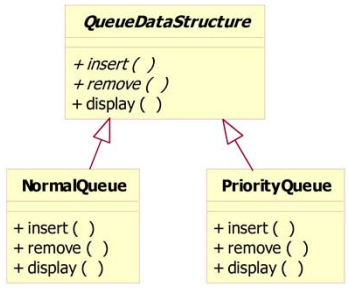
- **Write skeletal code/pseudo code for the above pattern (State)**

Behavioural Patterns - Exercises

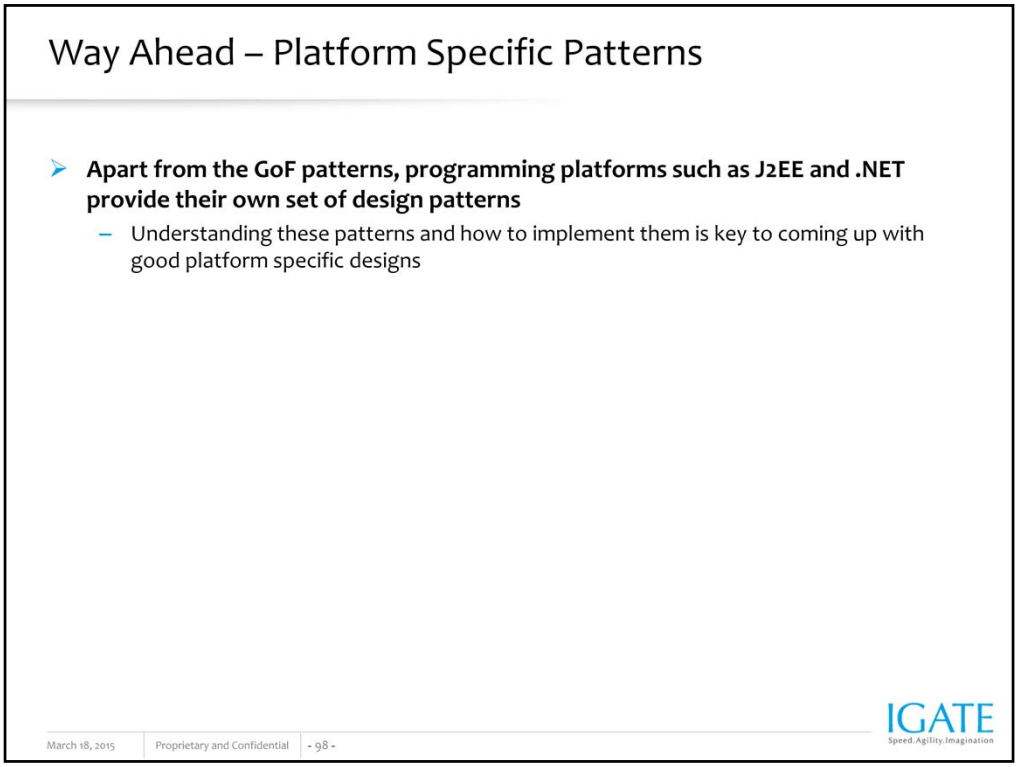


- **Write skeletal code/pseudo code for the above pattern (Strategy)**

Behavioural Patterns - Exercises



- **Write skeletal code/pseudo code for the above pattern (Template Method)**



Platform specific platforms for J2EE and .NET are dealt with separately.

Summary

- **You now have an understanding of**
  - What design patterns are
  - Gang Of Four Patterns

IGATE
Speed.Agility.Imagination