VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Software Architecture and Design Patterns

UNIT – III

Dr. T K Rao

VVIT

VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Contents

- **PATTERNS:**
  - **Pattern Description**
  - **Organizing catalogues**
  - **Role in solving design problems**
  - **Selection and usage**
- **CREATIONAL PATTERNS:**
  - **Abstract factory**
  - **Builder**
  - **Factory method**
  - **Prototype**
  - **Singleton**

Introduction

- Why do we need design patterns?
  - To create a class properly
  - To instantiate an object
  - To interact between objects
  - To write loosely coupled code
  - To write reusable code

What is a design pattern?

- A well defined solution to a common problem
- Industry standard approach
- It’s a template, not a solution
- Language independent

Benefits of Design Patterns

- Robust code
- Code reusability
- Highly maintainability
- Reduce Total Cost of Ownership (TCO)
- Loosely coupled application

Pattern Description

VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

- Each pattern describes a problem which occurs over and over again
- Then describes the core of the solution to that problem in such way that you can use this solution a million times over, without ever doing it the same way twice
- In general, a pattern has four essential elements:
  - The **pattern name**
  - The **problem**
  - The **solution**
  - The **consequences**

- **Pattern name** is a handle, used to describe a design problem, its solutions, and consequences in a word or two
- Naming a pattern immediately increases our design vocabulary
- Having a vocabulary for patterns, talk about them with colleagues, in documentation, and even to ourselves
- It makes it easier to think about designs and to communicate them and their trade-offs to others
- Finding good names has been one of the hardest parts of developing our catalogue

- **Problem** describes when to apply the pattern. It explains the problem and its context
- It might describe specific design problems such as how to represent algorithms as objects
- It might describe class or object structures that are symptomatic of an inflexible design
- Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern

- **Solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations.
- The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in different situations
- Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it

- **Consequences** are the results & trade-offs of applying the
- Though consequences are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern
- Consequences for s/w often concern space and time trade-offs
- They may address language and implementation issues as well
- Since reuse is a factor in O-O design, consequences of a pattern include its impact on a system's flexibility, extensibility, or portability

Organizing catalogues

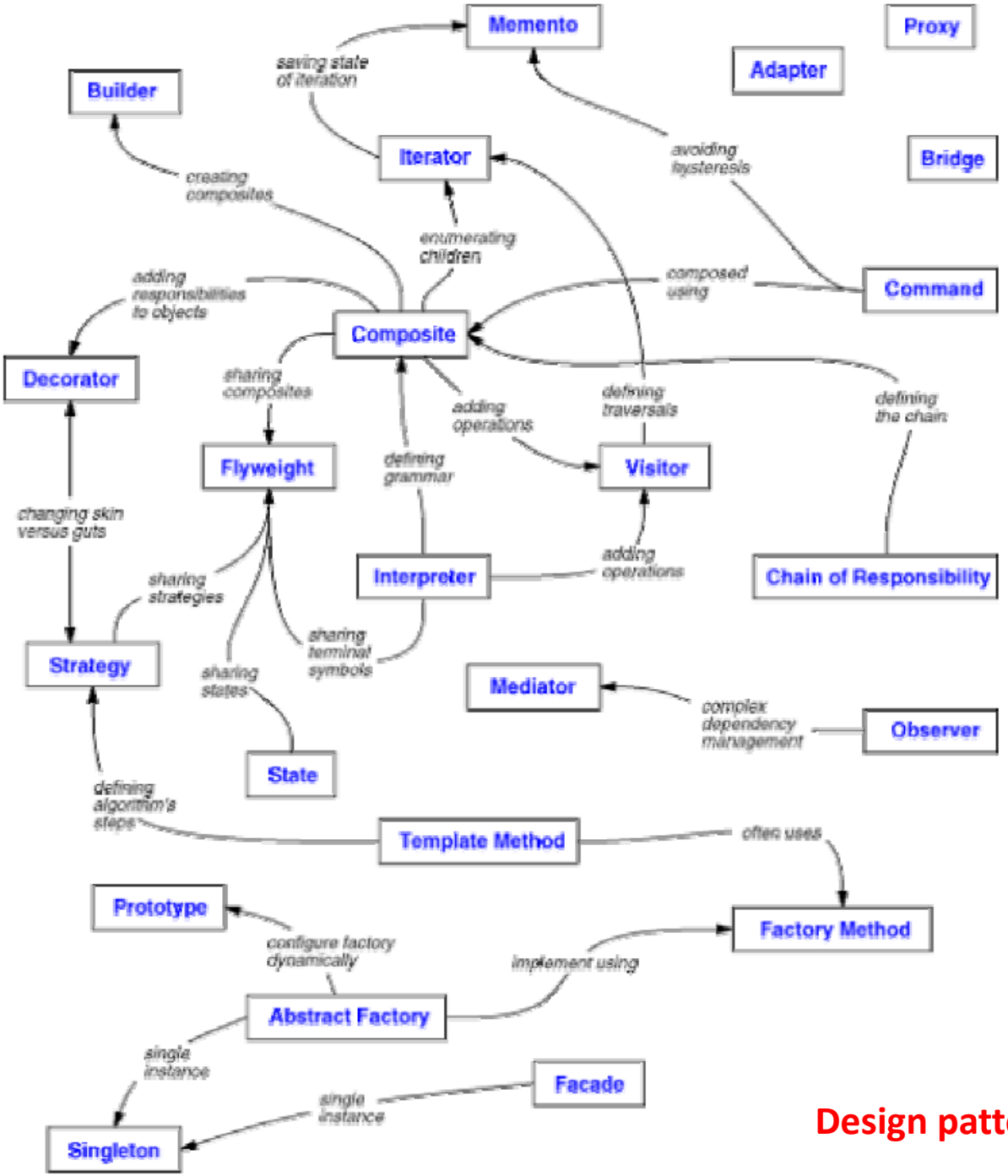
VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

- Design patterns vary in their granularity and level of abstraction
- Design patterns are classified by two criteria, viz., Purpose & Scope
  - **purpose**, reflects what a pattern does
  - **scope**, specifies whether the pattern applies to classes / objects
- Patterns can have **creational**, **structural**, or **behavioural** purpose
- Creational patterns concern the process of object creation
- Structural patterns deal with the composition of classes or objects
- Behavioural patterns characterize the ways in which classes or objects interact and distribute responsibility.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (121)	Adapter (157)	Interpreter (274) Template Method (360)
	Object	Abstract Factory (99) Builder (110) Prototype (133) Singleton (144)	Adapter (157) Bridge (171) Composite (183) Decorator (196) Facade (208) Flyweight (218) Proxy (233)	Chain of Responsibility (251) Command (263) Iterator (289) Mediator (305) Memento (316) Observer (326) State (338) Strategy (349) Visitor (366)

Table 1.1: Design pattern space

- Class patterns deal with relationships between classes and thei subclasses
- Relationships are established through inheritance, so they are static—fixed at compile-time
- Object patterns deal with object relationships, which can be changed at run-time (more dynamic)
- Some patterns are often used together, e.g., Composite is often used with Iterator or Visitor
- Some patterns are alternatives, e.g. Prototype is often an alternative to Abstract Factory
- Some patterns are similar designs (though patterns have different intents), e.g., structure diagrams of Composite & Decorator are similar



Design patterns relationships

How DP Solve Design Problems

- Here are several problems and how DP solve them
  - **Finding Appropriate Objects**
  - **Determining Object Granularity**
  - **Specifying Object Interfaces**
  - **Specifying Object Implementations**
  - **Class versus Interface Inheritance**
  - **Programming to an Interface, not an Implementation**
  - **Putting Reuse Mechanisms to Work**
  - **Inheritance versus Composition**
  - **Inheritance versus Parameterized Types**

Finding Appropriate Objects

- OOPs are made up of objects, which packages both data and the procedures that operate on that data
- The procedures are typically called **methods** or operations
- An object performs an operation when it receives a request (or **message**) from a **client**
- Operations are the *only* way to change an object's internal data
- Hence, the object's internal state is encapsulated
- It cannot be accessed directly, and its representation is invisible from outside the object.

- The hard part about OOD is decomposing a system into objects because of many factors:
  - encapsulation, granularity, dependency, flexibility, performance, evolution, reusability, etc.
- OOD methodologies favour many different approaches
- Write a problem statement, single out the nouns & verbs, & create corresponding classes & operations
- Or you can model the real world and translate the objects found during analysis into design
- There will always be disagreement on which approach is best

Abstract factory

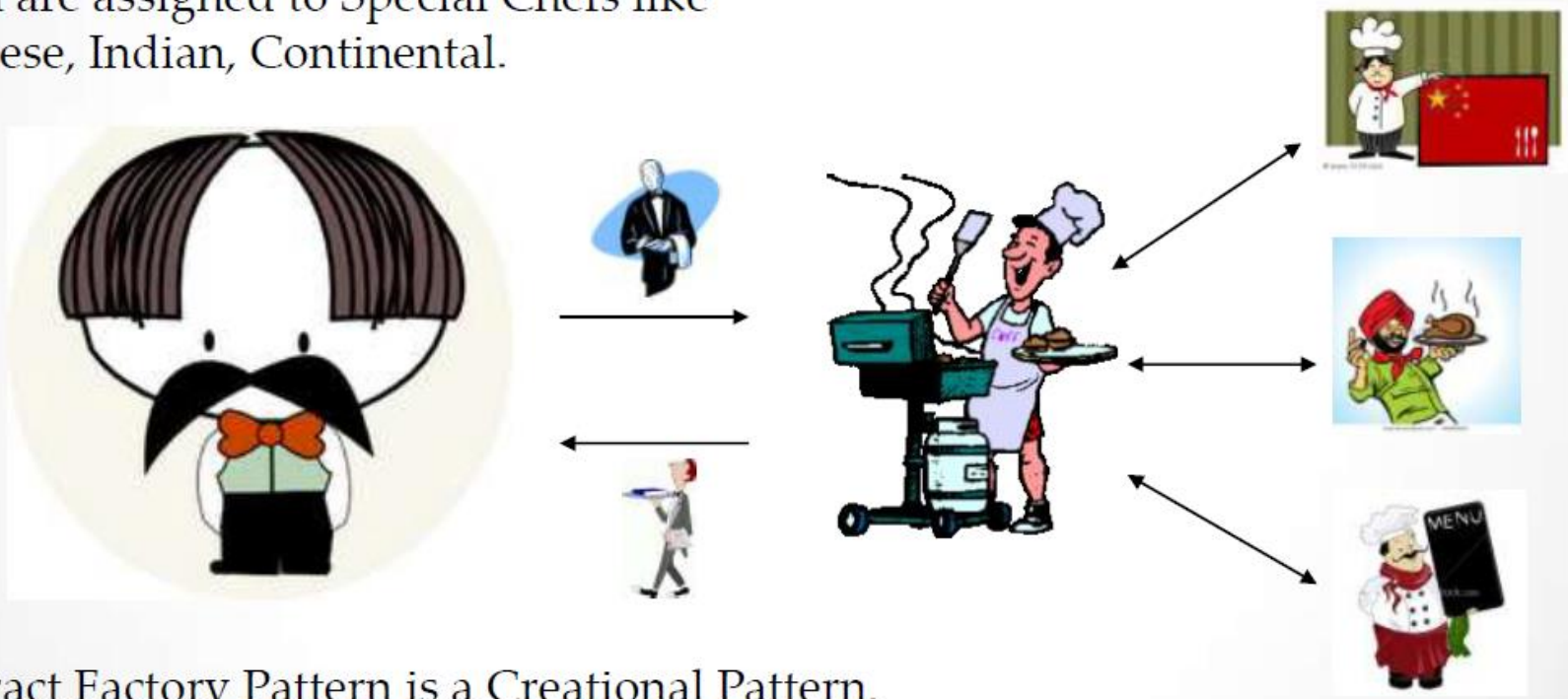
- Abstract Factory provides an interface for creating various kinds of objects without specifying concrete classes
- Abstract Factory can enforce dependencies between product classes
- Abstract Factory is also known as a “Kit”
- The standard form of the Abstract Factory is usually just a collection of Factory Methods
- A concrete factory will specify its products by overriding a factory method for each product

Abstract Factory Pattern in Real Life

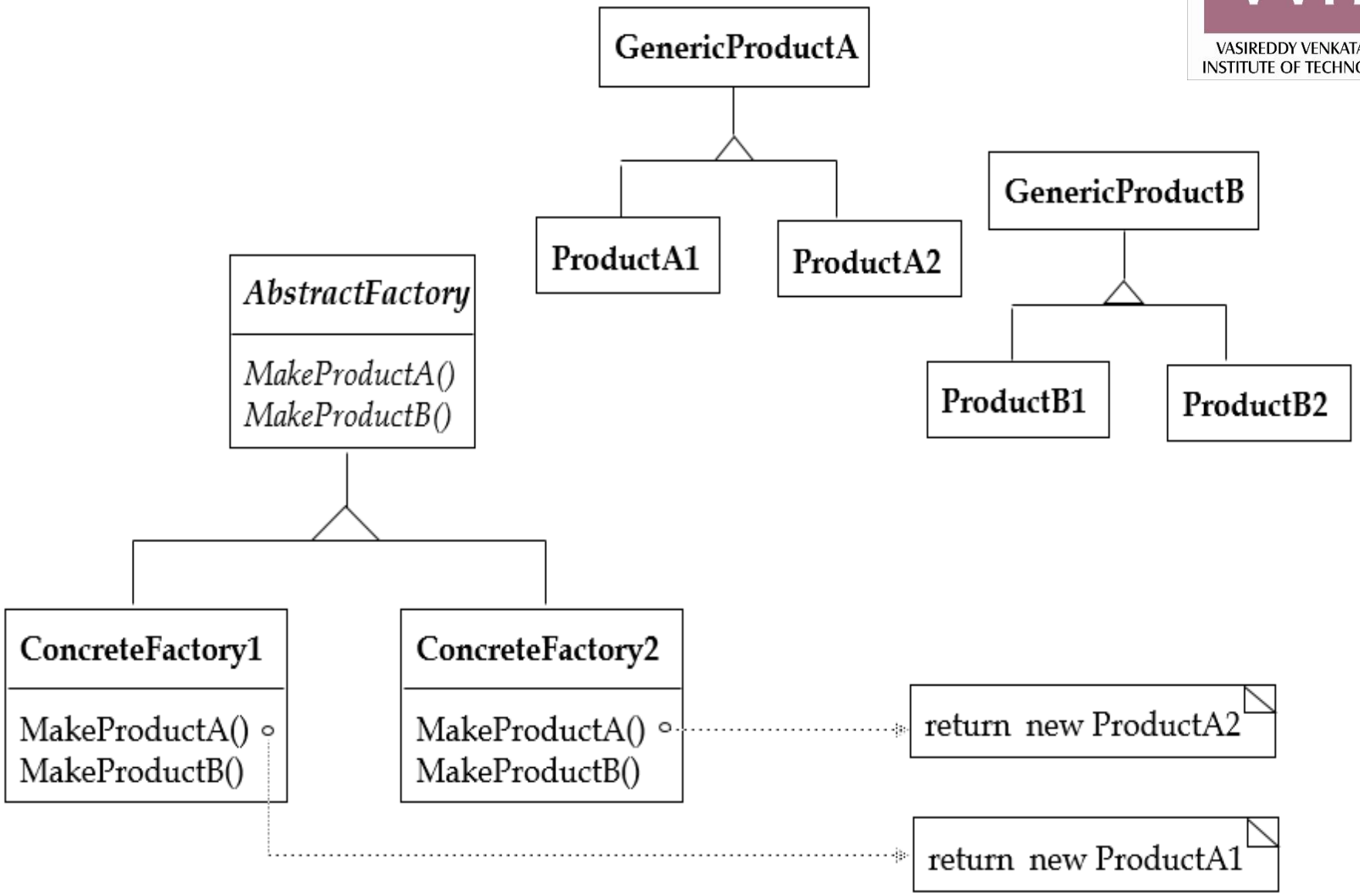
Abstract Factory Pattern is similar to Sub Contracting in real world.
Basically delegating the creation of Objects to expert Factories

---

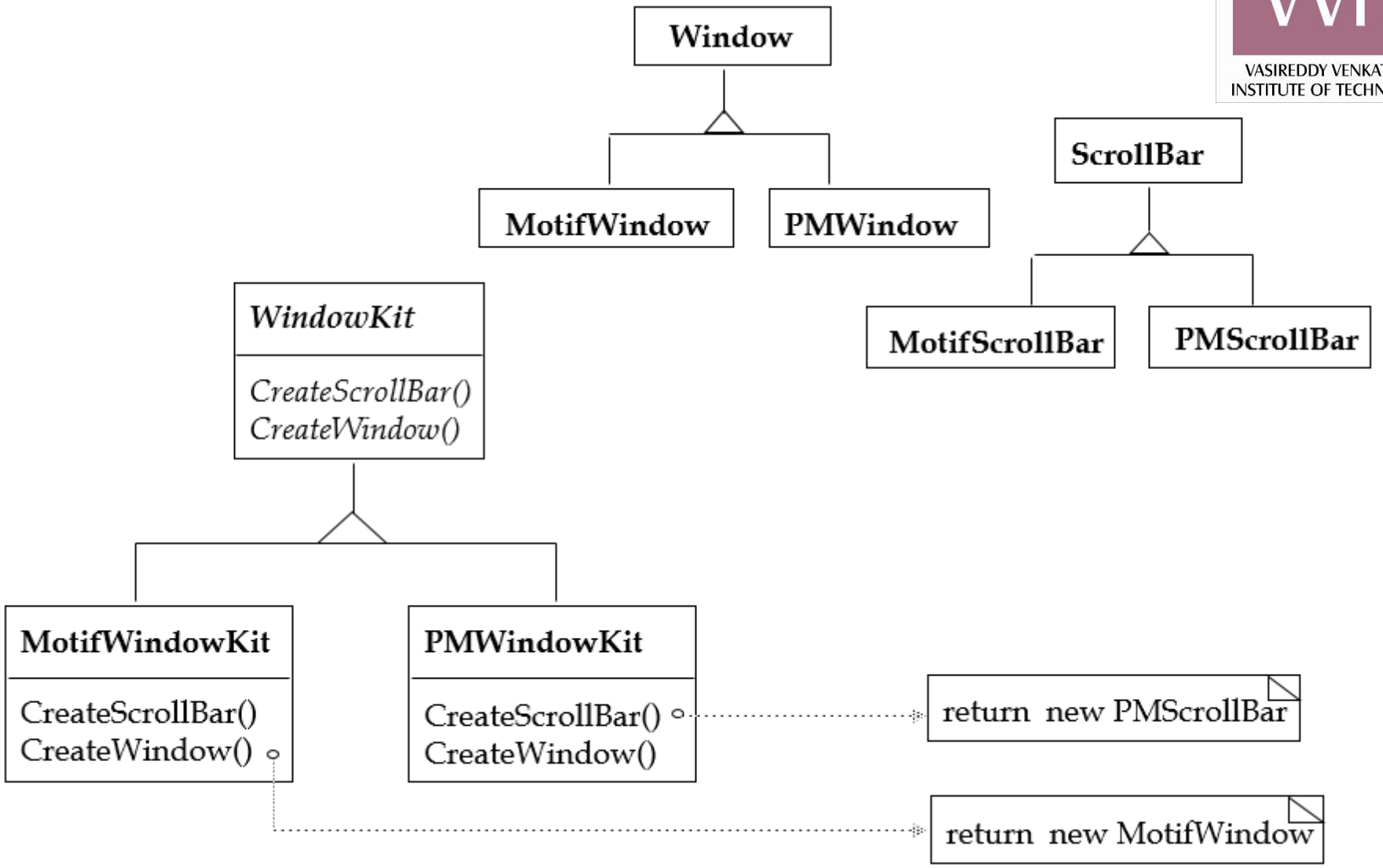
Orders in a restaurant are received by a Kitchen.
Then are assigned to Special Chefs like
Chinese, Indian, Continental.



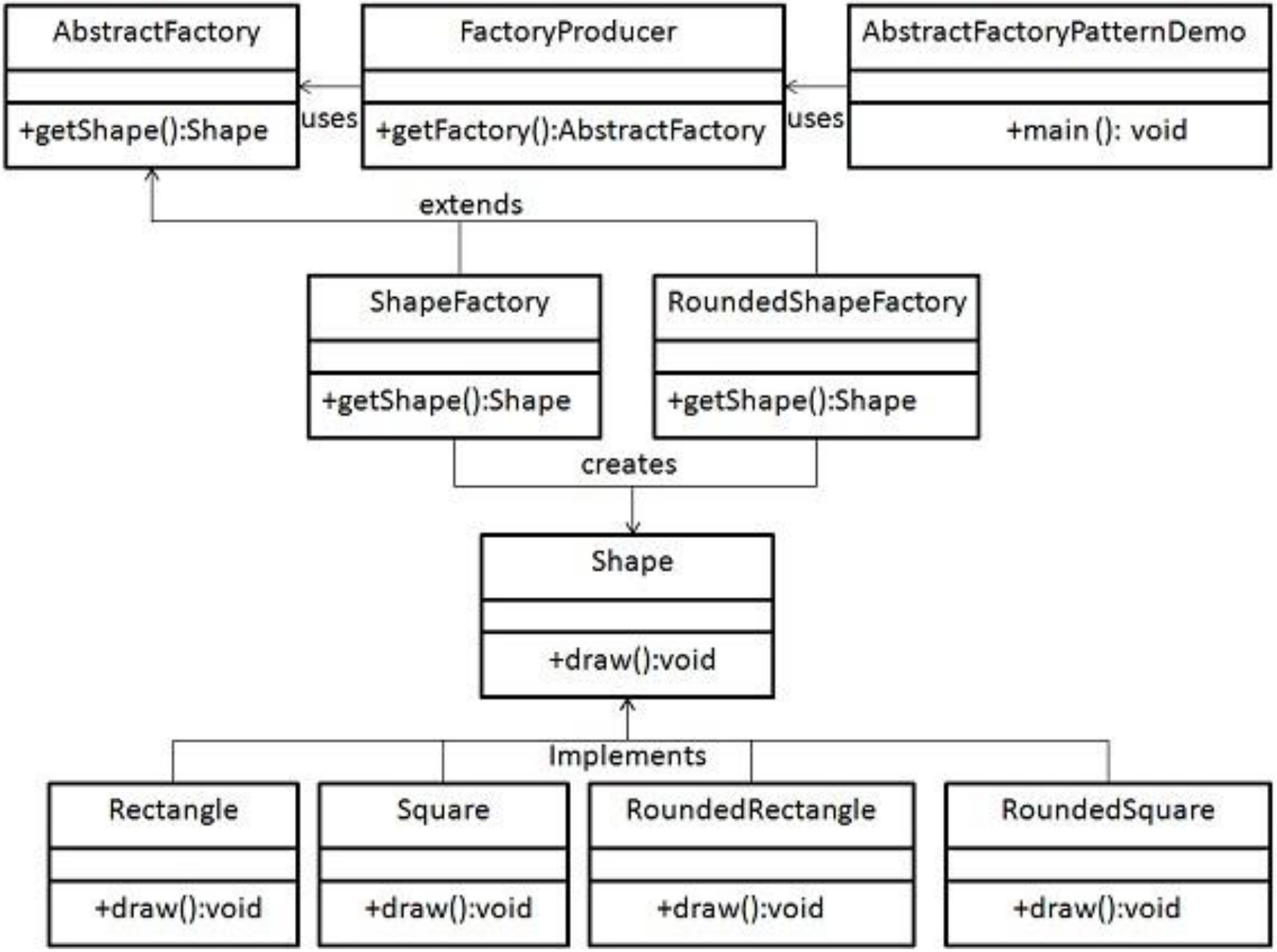
Abstract Factory Pattern is a Creational Pattern.
Similar to Factory Pattern it is Object Creation without exposing "HOW" ?



Abstract Factory: Structure



Abstract Factory: Simple Example


AbstractFactory
+getShape():Shape
uses
FactoryProducer
+getFactory():AbstractFactory
uses
AbstractFactoryPatternDemo
+main (): void
extends
ShapeFactory
+getShape():Shape
RoundedShapeFactory
+getShape():Shape
creates
Shape
+draw():void
Implements
Rectangle
+draw():void
Square
+draw():void
RoundedRectangle
+draw():void
RoundedSquare
+draw():void

Abstract Factory Implementation

Step 1

Create an interface for Shapes and Colors.

Shape.java

```
public interface Shape {
   void draw();
}
```

Step 2

Create concrete classes implementing the same interface.

RoundedRectangle.java

```
public class RoundedRectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside RoundedRectangle::draw()
method.");
   }
}
```

Step 2

Create concrete classes implementing the same interface.

RoundedRectangle.java

```
public class RoundedRectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside RoundedRectangle::draw()
method.");
    }
}
```

RoundedSquare.java

```
public class RoundedSquare implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside RoundedSquare::draw() method.");
    }
}
```

Rectangle.java

```
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}
```

Step 3

Create an Abstract class to get factories for Normal and Rounded Shape Objects.

AbstractFactory.java

```
public abstract class AbstractFactory {
   abstract Shape getShape(String shapeType) ;
}
```

Step 4

Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

RoundedShapeFactory.java

```
public class RoundedShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new RoundedRectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new RoundedSquare();
      }
      return null;
   }
}
```

Step 5

Create a Factory generator/producer class to get factories by passing an information such as Shape

FactoryProducer.java

```
public class FactoryProducer {
   public static AbstractFactory getFactory(boolean rounded){
      if(rounded){
         return new RoundedShapeFactory();
      }else{
         return new ShapeFactory();
      }
   }
}
```

Step 6

Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

AbstractFactoryPatternDemo.java

```
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {
      //get rounded shape factory
      AbstractFactory shapeFactory =
FactoryProducer.getFactory(false);
      //get an object of Shape Rounded Rectangle
      Shape shape1 = shapeFactory.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape1.draw();
      //get an object of Shape Rounded Square
      Shape shape2 = shapeFactory.getShape("SQUARE");
      //call draw method of Shape Square
      shape2.draw();
      //get rounded shape factory
      AbstractFactory shapeFactory1 =
FactoryProducer.getFactory(true);
      //get an object of Shape Rectangle
      Shape shape3 = shapeFactory1.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape3.draw();
      //get an object of Shape Square
      Shape shape4 = shapeFactory1.getShape("SQUARE");
      //call draw method of Shape Square
      shape4.draw();

   }
}
```

Step 7

Verify the output.

```
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside RoundedRectangle::draw() method.
Inside RoundedSquare::draw() method.
```

Abstract Factory: Advantages & Disadvantages

VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Advantages

- The Abstract Factory provides a focus during development for changing and controlling the type of objects created by clients. How and why?
- The family of product objects created by an AbstractFactory will often work together and provide behavior or functionality consistent with one another

Disadvantages

- **A typical AbstractFactory cannot be extended easily to produce new kinds of Products. What is the solution to this problem?**

Applicability

Use the Abstract Factory pattern when

- a system should be independent of how its products are created, composed, and represented.
- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to enforce this constraint.
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Participants

- **AbstractFactory** (WidgetFactory)
  - declares an interface for operations that create abstract product objects.
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory)
  - implements the operations to create concrete product objects.
- **AbstractProduct** (Window, ScrollBar)
  - declares an interface for a type of product object.
- **ConcreteProduct** (MotifWindow, MotifScrollBar)
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- **Client**
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Collaborations

- Normally a single instance of a ConcreteFactory class is created at run-time.
- This concrete factory creates product objects having a particular implementation.
- To create different product objects, clients should use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Consequences

The Abstract Factory pattern has the following benefits and liabilities:

1. *It isolates concrete classes.*

- The Abstract Factory pattern helps you control the classes of objects that an application creates.
- Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes.
- Clients manipulate instances through their abstract interfaces.
- Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.

2. *It makes exchanging product families easy.*

- The class of a concrete factory appears only once in an application—that is, where it's instantiated.
- This makes it easy to change the concrete factory an application uses.
- It can use different product configurations simply by changing the concrete factory.
- Because an abstract factory creates a complete family of products, the whole product family changes at once.
- In example, we can switch from Motif widgets to Presentation Manager widgets simply by switching the corresponding factory objects and recreating the interface.

3. *It promotes consistency among products.*

- When product objects in a family are designed to work together, it's important that an application use objects from only one family at a time.
- AbstractFactory makes this easy to enforce.

4. *Supporting new kinds of products is difficult.*

- Extending abstract factories to produce new kinds of Products isn't easy.
- That's because the AbstractFactory interface fixes the set of products that can be created.
- Supporting new kinds of products requires extending the factory interface, which involves changing the AbstractFactory class and all of its subclasses.
- We discuss one solution to this problem in the Implementation section.

Known Uses

- InterViews uses the "Kit" suffix [Lin92] to denote AbstractFactory classes.
- It defines WidgetKit and DialogKit abstract factories for generating look-and-feel-specific user interface objects.
- InterViews also includes a LayoutKit that generates different composition objects depending on the layout desired.
- For example, a layout that is conceptually horizontal may require different composition objects depending on the document's orientation (portrait or landscape).
- ET++ [WGM88] uses the Abstract Factory pattern to achieve portability across different window systems (X Windows and SunView, for example).
- The WindowSystem abstract base class defines the interface for creating objects that represent window system resources (MakeWindow, MakeFont, MakeColor, for example).
- Concrete subclasses implement the interfaces for a specific window system. At run-time, ET++ creates an instance of a concrete WindowSystem subclass that creates concrete system resource objects.

Related Patterns

- AbstractFactory classes are often implemented with factory methods (Factory Method), but they can also be implemented using Prototype
- A concrete factory is often a singleton (Singleton)

Builder

Intent:

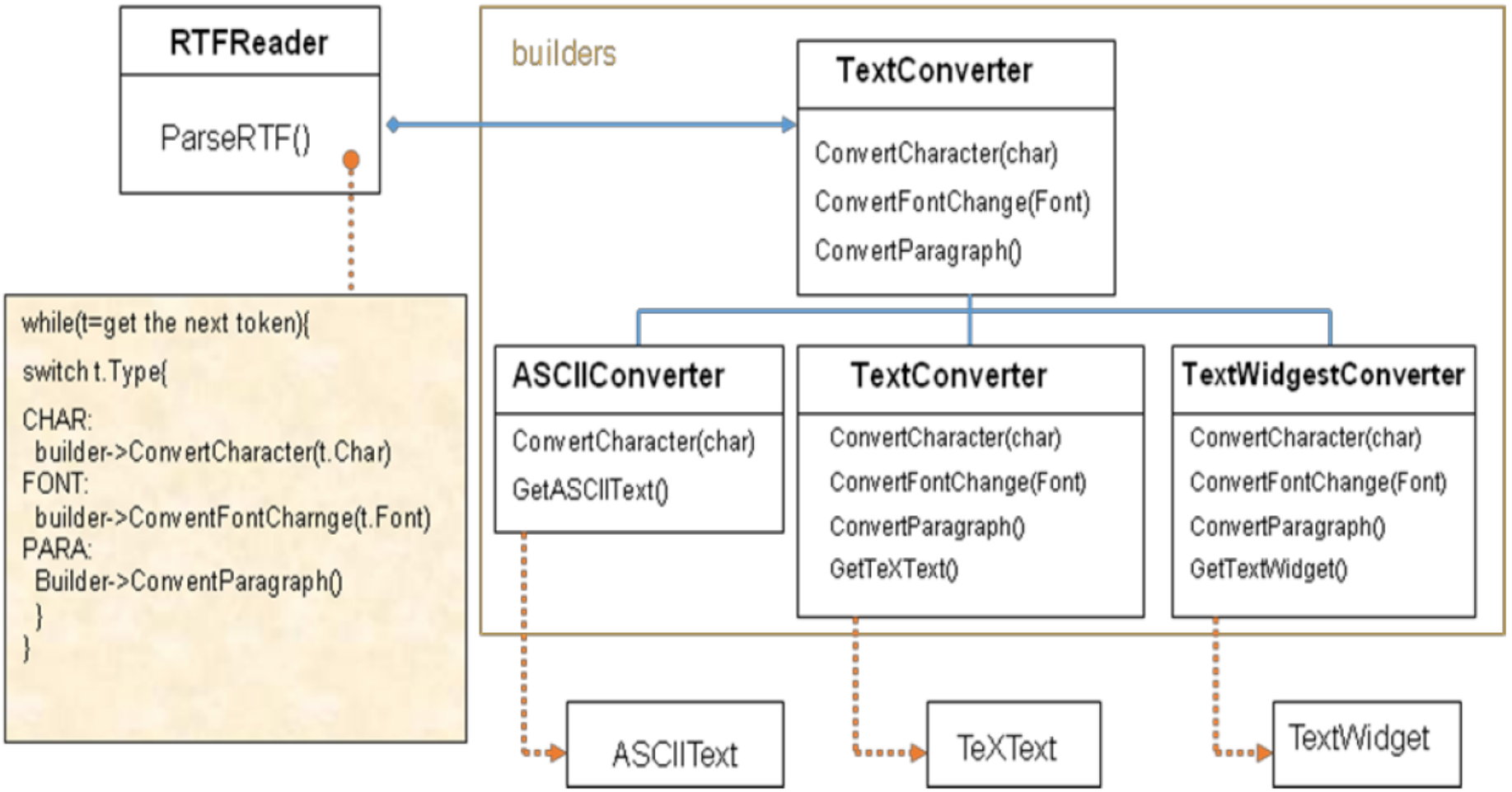
- Separate the construction of a complex object from its representation so that the same construction process can create different representations

Motivation:

- RTF reader should be able to convert RTF to many text format
- Adding new conversions without modifying the reader should be easy

- RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats
- The reader might convert RTF documents into plain ASCII text or into a text widget that can be edited interactively
- The problem, however, is that the number of possible conversions is open-ended
- So it should be easy to add a new conversion without modifying the reader.

BUILDER Motivation:-



A solution:

- Configure the RTFReader class with a TextConverter object that converts RTF to another textual representation
- As the RTFReader parses the RTF document, it uses the TextConverter to perform the conversion
- Whenever the RTFReader recognizes an RTF token it issues a request to the TextConverter to convert the token
- TextConverter objects are responsible both:
  - for performing the data conversion and
  - for representing the token in a particular format

- Subclasses of TextConverter specialize in different conversions and formats
- E.g., an ASCIIConverter ignores requests to convert anything except plain text
- A TeXConverter will implement operations for all requests in order to produce a TeX representation
- A TextWidgetConverter will produce a complex user interface object that lets the user see and edit the text
- Each kind of converter class takes the mechanism for creating and assembling a complex object and puts it behind an abstract interface
- The converter is separate from the reader, which is responsible for parsing an RTF document

- The Builder pattern captures all these relationships
- Each converter class is called a **builder** in the pattern, and the reader is called the **director**
- Builder pattern separates the algorithm for interpreting a textual format from how a converted format gets created and represented
- This reuses the RTFReader's parsing algorithm to create different text representations from RTF documents
- Just configure the RTFReader with different subclasses of TextConverter

Applicability

- Use the Builder pattern when
  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
  - the construction process must allow different representations for the object that's constructed.

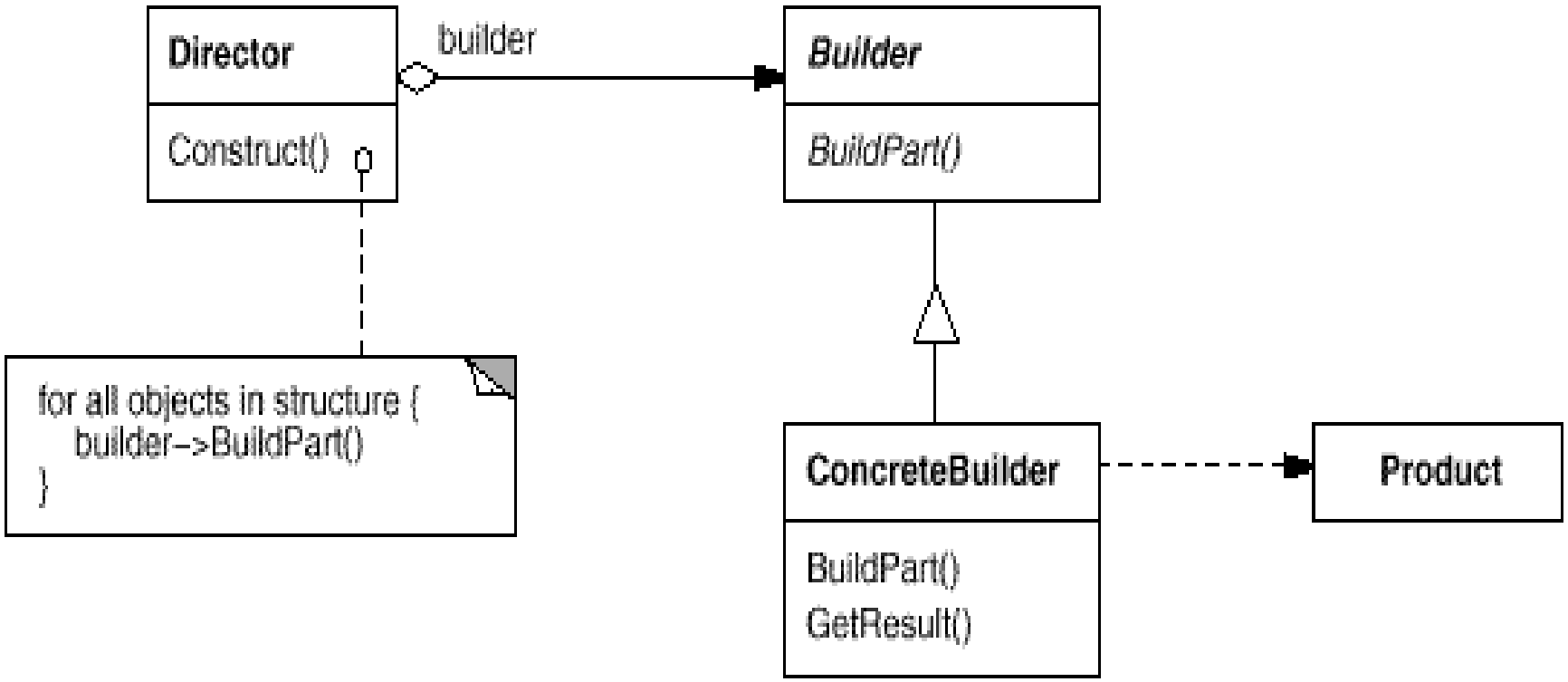
Participants

- **Builder** (TextConverter)
  - specifies an abstract interface for creating parts of a Product object.
- **ConcreteBuilder** (ASCIIConverter, TeXConverter, TextWidgetConverter)
  - constructs and assembles parts of the product by implementing the Builder interface.
  - defines and keeps track of the representation it creates.
  - provides an interface for retrieving the product (e.g., GetASCIIText, GetTextWidget).
- **Director** (RTFReader)
  - constructs an object using the Builder interface.
- **Product** (ASCIIText, TeXText, TextWidget)
  - represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

Builder – Collaborations:-

- Client creates Director object and configures it with the desired Builder object
- Director notifies Builder whenever a part of the product should be built
- Builder handles requests from the Director and adds parts to the product
- Client retrieves the product from the Builder

structure



Why do we use Builder?

- Common manner to Create an Instance
  - ***Constructor!***
  - Each Parts determined by Parameter of the Constructor

```
public class Room {
    private int area;
    private int windows;
    public String purpose;

    Room() {
    }

    Room(int newArea, int
    newWindows, String newPurpose){
            area = newArea;
            windows = newWindows;
            purpose = newPurpose;
    }
}
```

There are Only 2 different ways to Create an Instance part-by-part.

In the previous example,

- We can either determine all the arguments or determine nothing and just construct. We can‘t determine arguments partially.
- We can‘t control whole process to Create an instance.
- Restriction of ways to Create an Object
- Bad Abstraction & Flexibility

Discussion:-

- Uses Of Builder
  - Parsing Program(RTF converter)
  - GUI

Factory Method

Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate
- Factory Method lets a class defer instantiation to subclasses

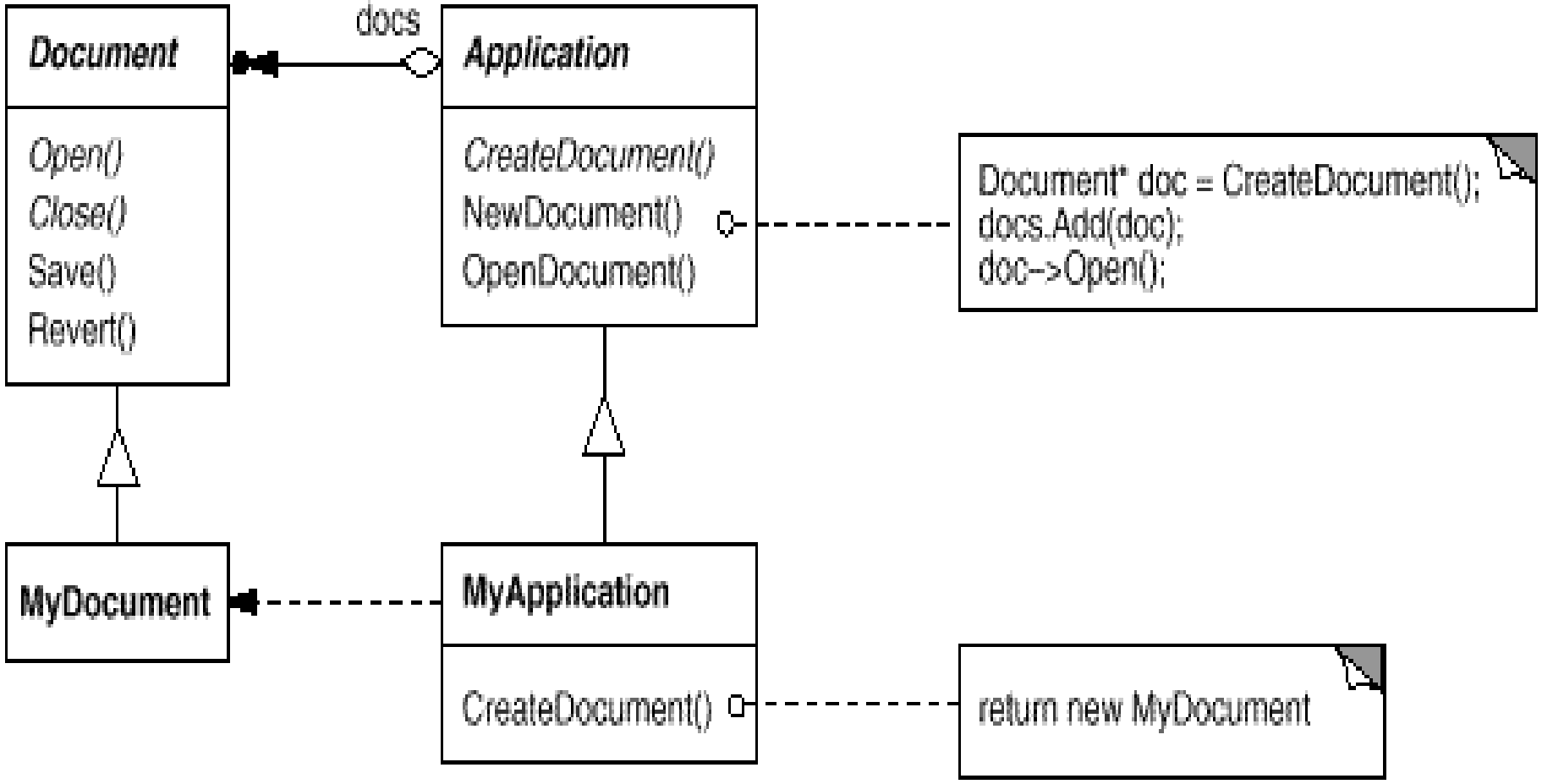
Also Known As

- Virtual Constructor

Motivation

- Consider a framework for applications that can present multiple documents to the user
- Two key abstractions in this framework are the 2 abstract classes; Application and Document
- Clients have to subclass them to realize their application-specific implementations
- E.g., To create a drawing application, we define 2 classes DrawingApplication and DrawingDocument
- Application class manages Documents and will create them as required—when the user selects Open or New from a menu
- E.g., Because the particular Document subclass to instantiate is application-specific, the Application class can't predict the subclass of Document to instantiate
- Application class only knows *when* a new document should be created, not *what kind* of Document to create

- This creates a dilemma: The framework must instantiate classes, but it only knows about abstract classes, which it cannot instantiate
- The Factory Method pattern offers a solution
- It encapsulates the knowledge of which Document subclass to create and moves this knowledge out of the framework
- Application subclasses redefine an abstract CreateDocument operation on Application to return the appropriate Document subclass
- Once an Application subclass is instantiated, it can then instantiate application-specific Documents without knowing their class
- We call CreateDocument a **factory method** because it's responsible for "manufacturing" an object.


Document
Open()
Close()
Save()
Revert()
docs
Application
CreateDocument()
NewDocument()
OpenDocument()
Document* doc = CreateDocument();
docs.Add(doc);
doc->Open();
MyDocument
MyApplication
CreateDocument()
return new MyDocument

Applicability

- Use the Factory Method pattern when
  - a class can't anticipate the class of objects it must create
  - a class wants its subclasses to specify the objects it creates
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.