IOE 321 Software Design Patterns

Chapter III

Creational Patterns

Presented by
Dr. Koppala Guravaiah
Assistant Professor
IIIT Kottayam

Syllabus

Indian Institute of Information Technology Kottayam

Creational Patterns

- Singleton
- Abstract Factory
- Builder
- Factory Method
- Prototype

Implementation in various languages like Python, Java

Reference:

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Design Patterns: Elements of Reusable Object oriented Software Addison-Wesley

Introduction

- Creational design patterns abstract the instantiation process, help make a system independent of how its objects are created, composed, and represented
  - A class creational pattern uses inheritance to vary the class that's instantiated,
  - whereas an object creational pattern will delegate instantiation to another object
- Creational patterns become important as systems evolve to depend more on object composition than class inheritance.

Introduction … Contd.

Indian Institute of Information Technology Kottayam

- Two recurring themes in these patterns.
  - First, they all encapsulate knowledge about which concrete classes the system uses.
  - Second, they hide how instances of these classes are created and put together.
- The creational patterns give you a lot of flexibility in
  - *what* gets created,
  - *who* creates it,
  - *how* it gets created,
  - and *when*.

Introduction … Contd.

Indian Institute of Information Technology Kottayam

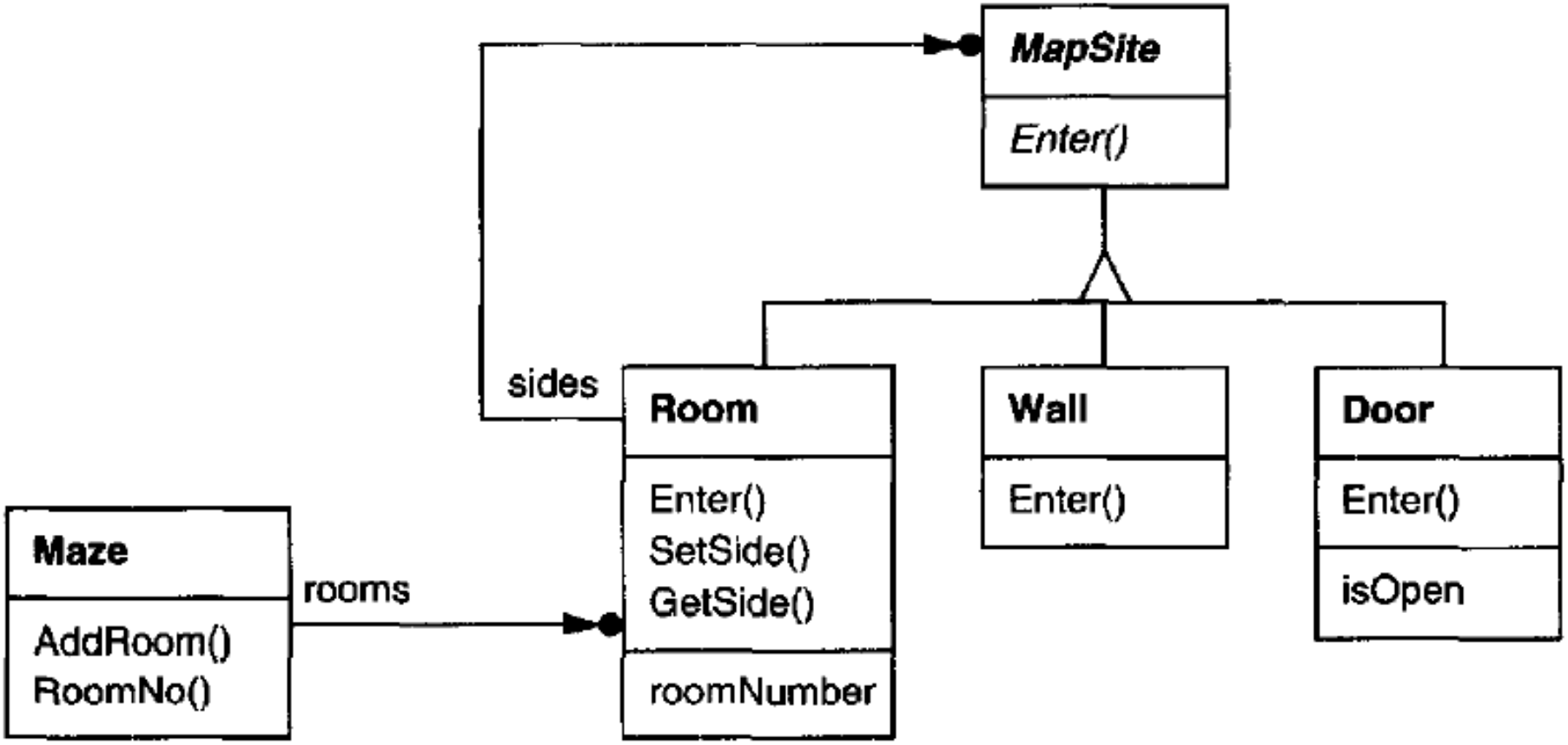
- Sometimes creational patterns are competitors.
  - For example, there are cases when either *Prototype* or *Abstract Factory* could be used profitably.
  - At other times they are complementary: *Builder* can use one of the other patterns to implement which components get built.
  - Prototype can use Singleton in its implementation.
- The creational patterns are closely related, study all five of them together to highlight their similarities and differences
- Use common example **building a maze for a computer game**

Introduction … Contd.

Indian Institute of Information Technology Kottayam

- The maze and the game will vary slightly from pattern to pattern.
- Sometimes the game will be simply to find your way out of a maze; in that case the player will probably only have a local view of the maze.
- Sometimes mazes contain problems to solve and dangers to overcome, and these games may provide a map of the part of the maze that has been explored.

Introduction … Contd.



Introduction … Contd.

- Each room has four sides.

enum Direction {North, South, East, West};

- The class MapSite is the common abstract class for all the components of a maze.
- If you try to enter a door, then one of two things happen: If the door is open, you go into the next room. If the door is closed, then you hurt your nose.

```
class MapSite
{
        public: virtual void Enter() = 0;
}
```

Introduction … Contd.

- Room is the concrete subclass of MapSite that defines the key relationships between components in the maze.
- It maintains references to other MapSite objects and stores a room number. The number will identify rooms in the maze.

```
class Room : public MapSite {
public:
Room(int roomNo);
MapSite* GetSide(Direction) const;
void SetSide(Direction, MapSite*);
virtual void Enter();
private:
MapSite* _sides[4];
int _roomNumber;
};
```

Introduction … Contd.

Indian Institute of Information Technology Kottayam

- The following classes represent the wall or door that occurs on each side of a room.

```
class Wall : public MapSite {
public:
Wall()
virtual void Enter();
};
```

```
class Door : public MapSite {
public:
Door(Room* = 0, Room* = 0);
virtual void Enter();
Room* OtherSideFrom(Room*);
private:
Room* _rooml;
Room* _room2;
bool _isOpen;
};
```

Introduction … Contd.

Indian Institute of Information Technology Kottayam

```
class Maze {
public:
Maze();
void AddRoom(Room* ) ;
Room* RoomNo(int) const;
private:
// . . .
};
```

Introduction … Contd.


```
Maze* MazeGame::CreateMaze ()
{
    Maze* aMaze = new Maze;
    Room* rl = new Room(l);
    Room* r2 = new Room (2);
    Door* theDoor = new Door(rl, r2);
    aMaze->AddRoom(rl);
    aMaze->AddRoom(r2);
    rl->SetSide(North, new Wall);
    rl->SetSide(East, theDoor);
    rl->SetSide(South, new Wall);
    rl->SetSide(West, new Wall);
    r2->SetSide(North, new Wall);
    r2->SetSide(East, n ew Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);
    return aMaze;
}
```

Introduction … Contd.

Indian Institute of Information Technology Kottayam

- If ***CreateMaze*** calls virtual functions instead of constructor calls to create the rooms, walls, and doors it requires, then you can change the classes that get instantiated by making a subclass of MazeGame and redefining those virtual functions. This approach is an example of the ***FactoryMethod*** pattern.
- If ***CreateMaze*** is passed an object as a parameter to use to create rooms, walls, and doors, then you can change the classes of rooms, walls, and doors by passing a different parameter. This is an example of the ***Abstract Factory*** pattern.
- If ***CreateMaze*** is passed an object that can create a new maze in its entirety using operations for adding rooms, doors, and walls to the maze it builds, then you can use inheritance to change parts of the maze or the way the maze is built. This is an example of the ***Builder*** pattern.

Introduction … Contd.

- If ***CreateMaze*** is parameterized by various prototypical room, door, and wall objects, which it then copies and adds to the maze, then you can change the maze's composition by replacing these prototypical objects with different ones. This is an example of the ***Prototype pattern***.
- The remaining creational pattern, ***Singleton***, can ensure there's only one maze per game and that all game objects have ready access to it—without resorting to global variables or functions. Singleton also makes it easy to extend or replace the maze without touching existing code.

Singleton pattern

Indian Institute of Information Technology Kottayam

- *The singleton pattern is a design pattern that restricts the instantiation of a class to one object.*
- **Intent**
  - Ensure a class only has one instance, and provide a global point of access to it.
- **Motivation**
  - It's important for some classes to have exactly one instance.
  - How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.
  - A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance.
  - This is the **Singleton pattern.**

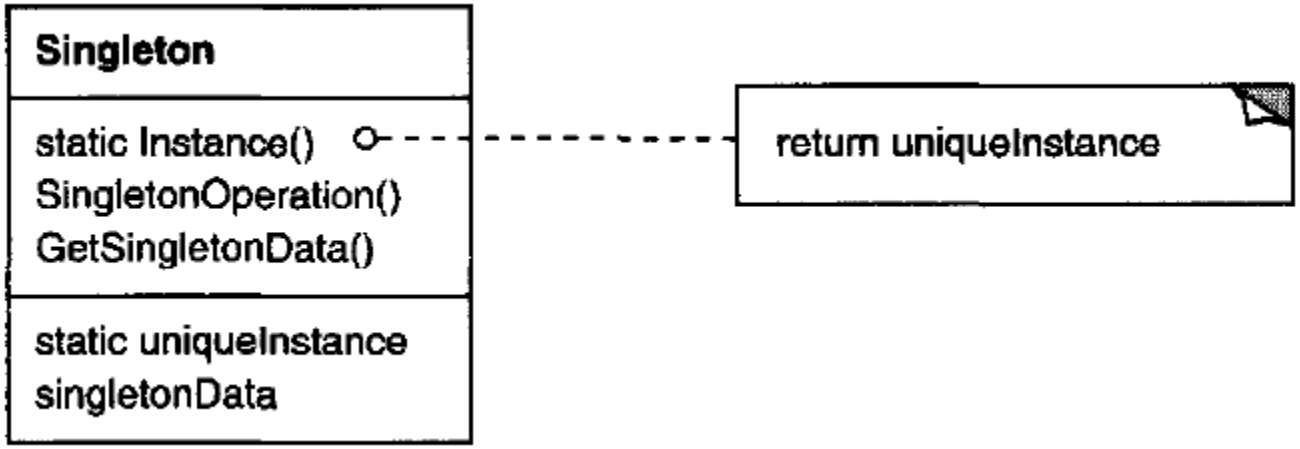
Singleton pattern … Contd.

- **Applicability**

  Use the Singleton pattern when

  - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
  - When the sole instance should be extensible by sub-classing, and client should be able to use an extended instance without modifying their code.

- **Structure**



Singleton pattern … Contd.

- **Participants**
  - **Singleton**
  - Defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that i s, a class method in Small talk and a static member function in C++).
  - May be responsible for creating its own unique instance.
- **Collaborations**
  - Clients access a Singleton instance solely through Singleton's Instance operation.

Singleton pattern … Contd.

- **Consequences**
- The Singleton pattern has several benefits:
  1. *Controlled access to sole instance*
  2. *Reduced name space*
  3. *Permits refinement of operations and representation*
  4. *Permits a variable number of instances*
  5. *More flexible than class operations.*

Singleton pattern … Contd.

- **Implementation**

The Singleton class is declared as

```
class Singleton
{
    public:
        static Singleton* Instance();
    protected:
        Singleton();
    private:
        static Singleton* _instance;
};
```

```
Singleton* Singleton::_instance = 0;
Singleton* Singleton::Instance ()
{
        if (_instance == 0)
        {
            _instance = new Singleton;
        }
        return _instance;
}
```

Singleton pattern … Contd.

- **Sample Code**

The Singleton class is declared as

```
class MazeFactory {
public:
        static MazeFactory* Instance();
protected:
        MazeFactory();
private:
        static MazeFactory* _instance;
};
```

Singleton pattern … Contd.

- **Sample Code**

```
MazeFactory* MazeFactory::_instance = 0;
MazeFactory* MazeFactory::Instance ( )
{
        if (_instance ==0 ) {
                _instance = new MazeFactory;
        }
        return _instance;
}
```

Factory Method

- Also Known as *Virtual Constructor*
- **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.
- **Motivation**
  - ***Frameworks*** use abstract classes to define and maintain relationships between objects
  - A framework is often responsible for creating these objects as well

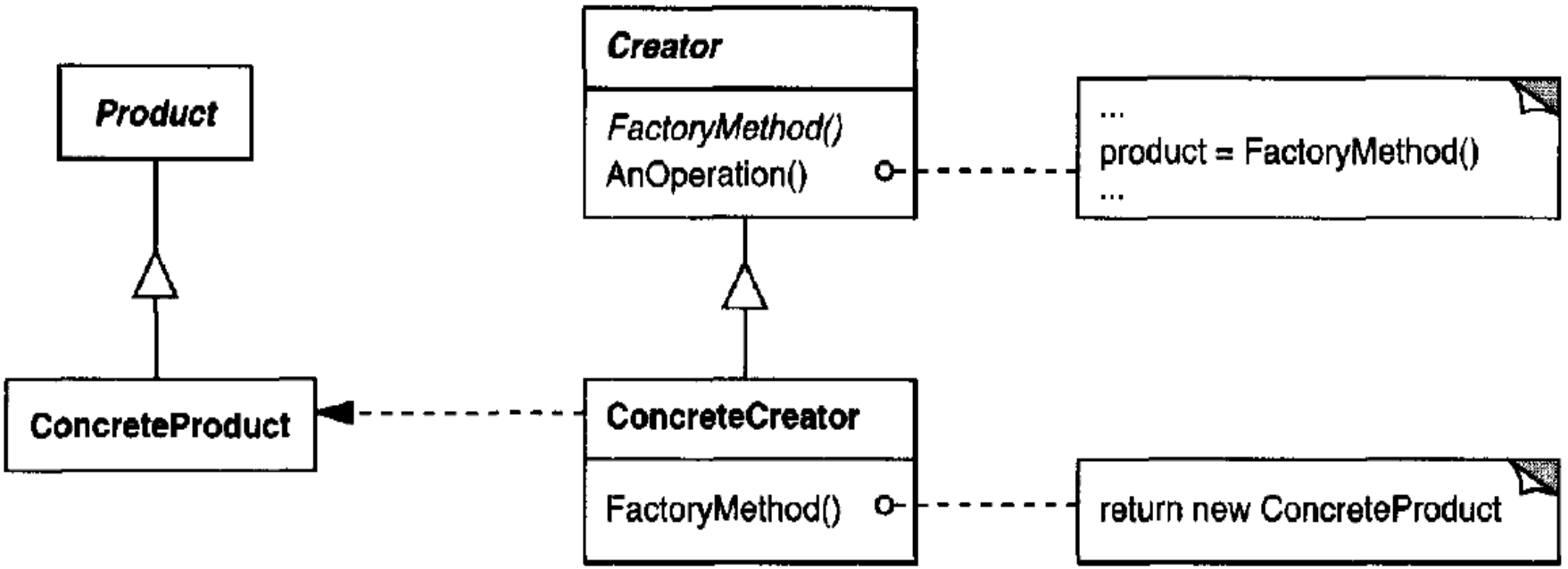
Factory Method … Contd.

- **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.

Factory Method … Contd.

Indian Institute of Information Technology Kottayam

- **Structure**



Factory Method … Contd.

- **Participants**
  - **Product (Document)** defines the interface of objects the factory method creates.
  - **ConcreteProduct (MyDocument)** implements the Product interface.
  - **Creator (Application)** declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - may call the factory method to create a Product object.
  - **ConcreteCreator (MyApplication)** overrides the factory method to return an instance of a ConcreteProduct.

Factory Method … Contd.

- **Collaborations**
- Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct

Factory Method … Contd.

- **Consequences**
- Factory methods eliminate the need to bind application-specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined ConcreteProduct classes.
- A potential **disadvantage** of factory methods is
  - that clients might have to subclass the Creator class just to create a particular ConcreteProduct object.
  - Subclassing is fine when the client has to subclass the Creator class anyway, but otherwise the client now must deal with another point of evolution.

Factory Method … Contd.

- **Consequences**
- Here are two additional consequences of the ***FactoryMethod*** pattern:
  1. *Provides hooks for subclasses.*
  2. *Connects parallel class hierarchies.*

Factory Method … Contd.

- **Implementation**
- Consider the following issues when applying the ***FactoryMethod*** pattern:
  - *Two major varieties.* The two main variation s of the Fact oryMethod pattern are
  - (1) the case when the Creator class is an abstract class and does not provide an implementation for the factory method it declares, and
  - (2) the case when the Creator is a concrete class and provides a default implementation for the factory method. It's also possible to have an abstract class that defines a default implementation, but this is less common

Factory Method … Contd.

- **Implementation … Contd.**
  - *Parameterized factory methods.*
  - *Language-specific variants and issues.*
  - *Naming convention*

Factory Method … Contd.

Indian Institute of Information Technology Kottayam

- **Sample Code**

```
class MazeGame {
        public:
        Maze* CreateMaze();
        // factory methods:
        virtual Maze* MakeMaze() const
        { return new Maze; }
        virtual Room* MakeRoom(int n) const
        { return new Room(n); }
        virtual Wall* MakeWall() const
        { return n ew Wall; }
        virtual Door* MakeDoor(Room* rl, Room* r2) const
        { return new Door(rl, r2); }
};
```

Factory Method … Contd.

- **Sample Code**

```
Maze* MazeGame::CreateMaze (){
Maze* aMaze = MakeMaze ();
Room* rl = MakeR oom(1);
Room* r2 = MakeRoom(2);
Door* theDoor = MakeDoor(rl, r2);
aMaze->AddRoom(rl);
aMaze->AddRoom(r2);
rl->SetSide(North, MakeWall());
rl->SetSide(East, theDoor);
rl->SetSide(South, MakeWall());
rl->SetSide(West, MakeWall());
r2->SetSide(North, MakeWall());
r2->SetSide(East, MakeWall());
r2->SetSide(South, MakeWall());
r2->SetSide(West, theDoor);
return aMaze;
}
```

Abstract Factory

- Also know as **Kit**
- **Intent**
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Motivation**
  - Consider user interface toolkit that supports multiple look-and-feel standards
  - To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel
  - To solve this problem by defining an abstract WidgetFactory class that declares an interface for creating each basic kind of widget
  - clients only have to commit to an interface defined by an abstract class, not a particular concrete class.

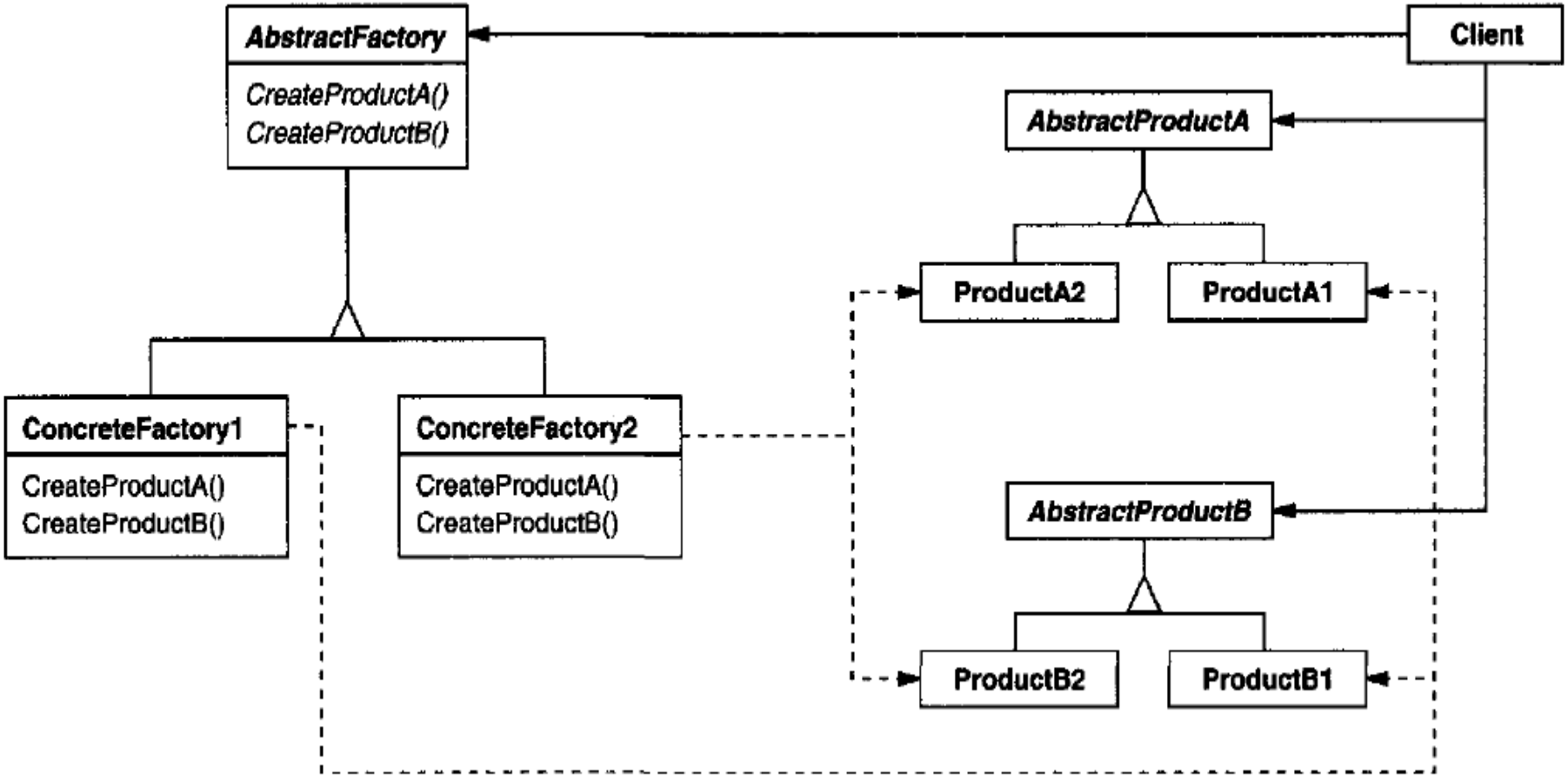
Abstract Factory … Contd.

- **Applicability**
- Use the Abstract Factorypattern 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.

Abstract Factory … Contd.

Indian Institute of Information Technology Kottayam

- **Structure**



Abstract Factory … Contd.

- **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 theAbstractProduct interface.
  - **Client:** uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Abstract Factory … Contd.

- **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.

Abstract Factory … Contd.

- **Consequences**
- The Abstract Factory pattern has the following benefits and liabilities:
  1. *It isolates concrete classes*
  2. *It makes exchanging product families easy*
  3. *It promotes consistency among products*
  4. *Supporting new kinds of products is difficult*

Abstract Factory … Contd.

- **Implementation**
- Here are some useful techniques for implementing the Abstract Factory pattern.
  - *Factories as singletons*
  - *Creating the products*
  - *Defining extensible factories.*

Abstract Factory … Contd.

- **Sample Code**

```
class MazeFactory {
    public:
    MazeFactory();
    virtual Maze* MakeMazeO const
    { return new Maze; }
    virtual Wall* MakeWall() const
    { return new Wall; }
    virtual Room* MakeRoom(int n) const
    { return new Room(n); }
    virtual Door* MakeDoor(Room* rl, Room* r2) const
    { return new Door(rl, r2); }
}
```

Abstract Factory … Contd.

- **Sample Code**

```
Maze* MazeGame::CreateMaze (MazeFactoryk factory) {
Maze* aMaze = factory.MakeMaze();
Room* rl = factory.MakeRoom(1);
Room* r2 = factory.MakeRoom(2);
Door* aDoor = factory.MakeDoor(rl, r2);
aMaze->AddRoom(rl);
aMaze->AddRoom(r2);
rl->SetSide(North, factory.MakeWall());
rl->SetSide(East, aDoor);
rl->SetSide(South, factory.MakeWall() ) ;
rl->SetSide(West, factory.MakeWall());
r2->SetSide(North, factory.MakeWall());
r2->SetSide(East, factory.MakeWall());
r2->SetSide(South, factory.MakeWall());
r2->SetSide(West, aDoor);
return aMaze;
}
```

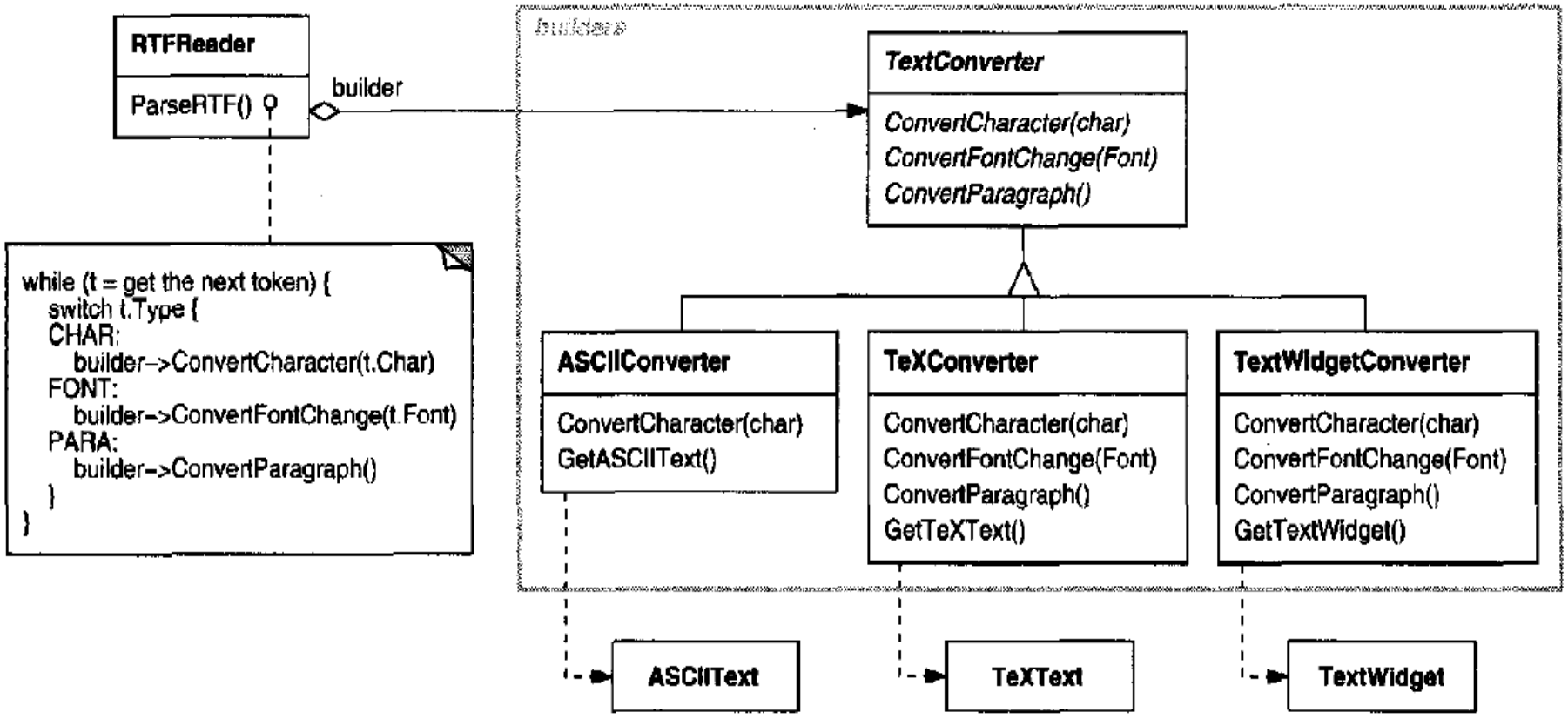
Builder Patterns

- Creational design pattern
- When we used too many arguments to send in constructor & it's hard to maintain the order
- When we don't want to send all the parameters in object initialization (generally need to send optional as a null parameters)
- **Intent**
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Builder Patterns … Contd.

Indian Institute of Information Technology Kottayam

Motivation



Builder Patterns … Contd.

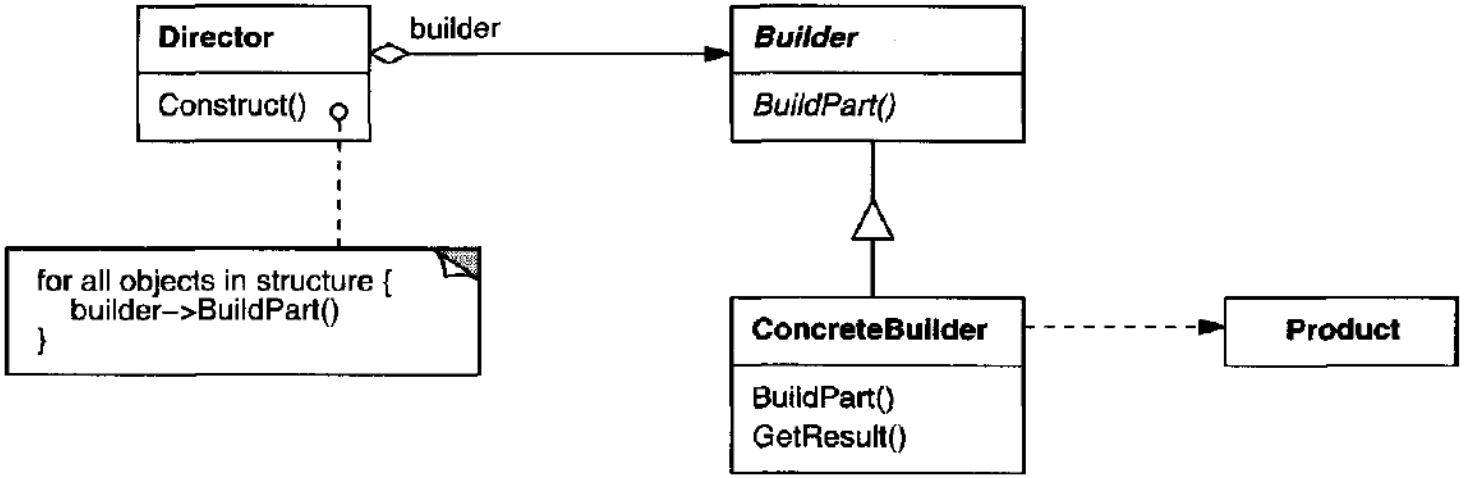
Indian Institute of Information Technology Kottayam

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.

Structure



Builder Patterns … Contd.

- **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, Get-TextWidget).
  - ***Director (RTFReader):***
    - constructs an object using the Builder interface.

Builder Patterns … Contd.

- **Participants … Contd.**
  - ***Product (ASCIIText, TeXText, TextWidget):***
    - represents the complex object under construction. Concrete Builder 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 Patterns … Contd.

- **Collaborations**
  - The client creates the Director object and configures it with the desired Builder object.
  - Director notifies the builder whenever a part of the product should be built.
  - Builder handles requests from the director and adds parts to the product.
  - The client retrieves the product from the builder.

Builder Patterns … Contd.

Indian Institute of Information Technology Kottayam

- **Consequences**
  - Here are key consequences of the Builder pattern:
  1. *It lets you vary a product's internal representation.*
  2. *It isolates code for construction and representation.*
  3. *It gives you finer control over the construction process.*
- **ImplementationsConsequences**
  - Here are other implementation issues to consider:
  1. *Assembly and construction interface.*
  2. *Why n o abstract class for products?*
  3. *Empty methods as default in Builder.*

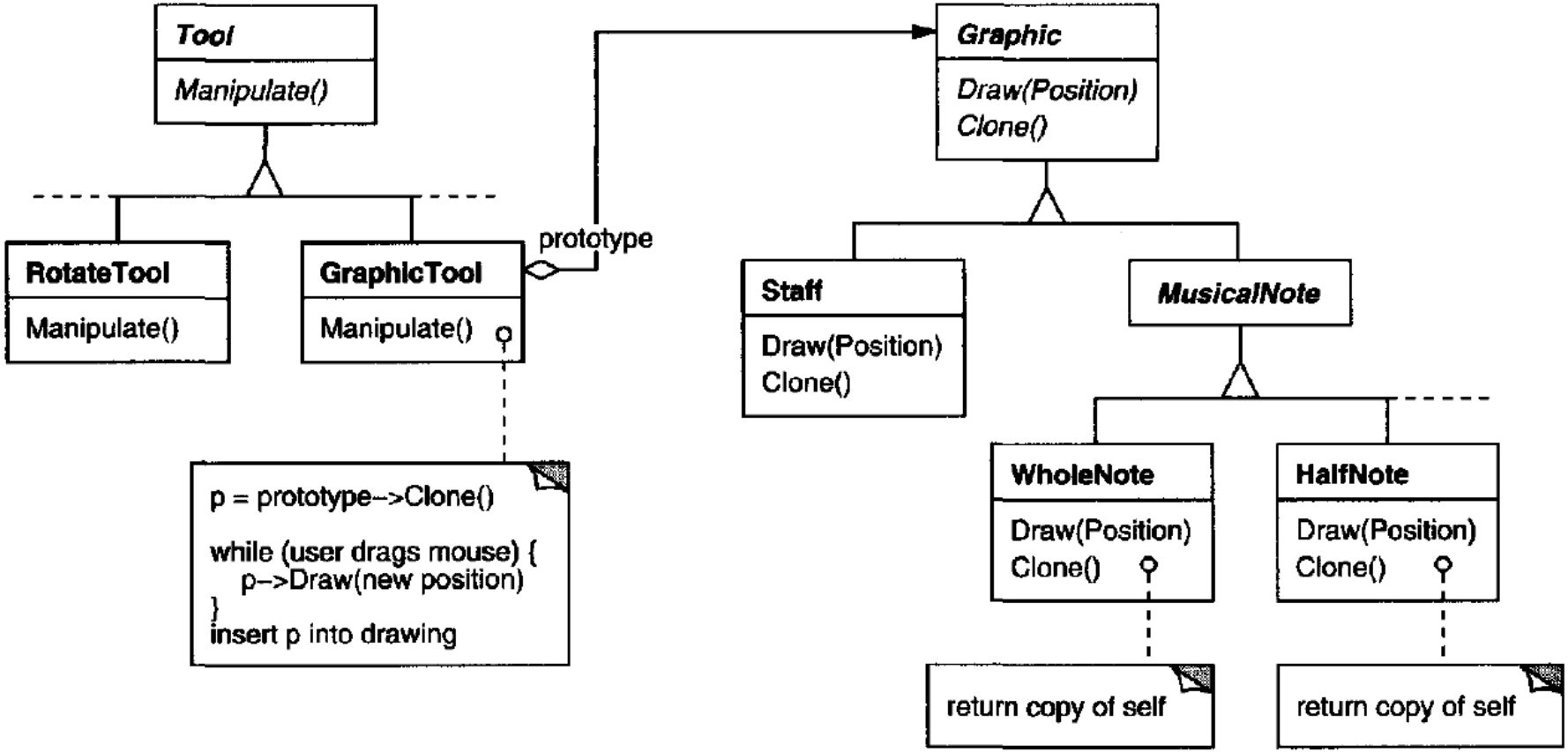
Prototype Patterns

- Creational Design pattern
- Used when you want to avoid multiple object creation of same instance; instead you copy the object to new object & then we can modify as per our need
- **Intent**
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Prototype Patterns … Contd.

Indian Institute of Information Technology Kottayam

- **Consequences**



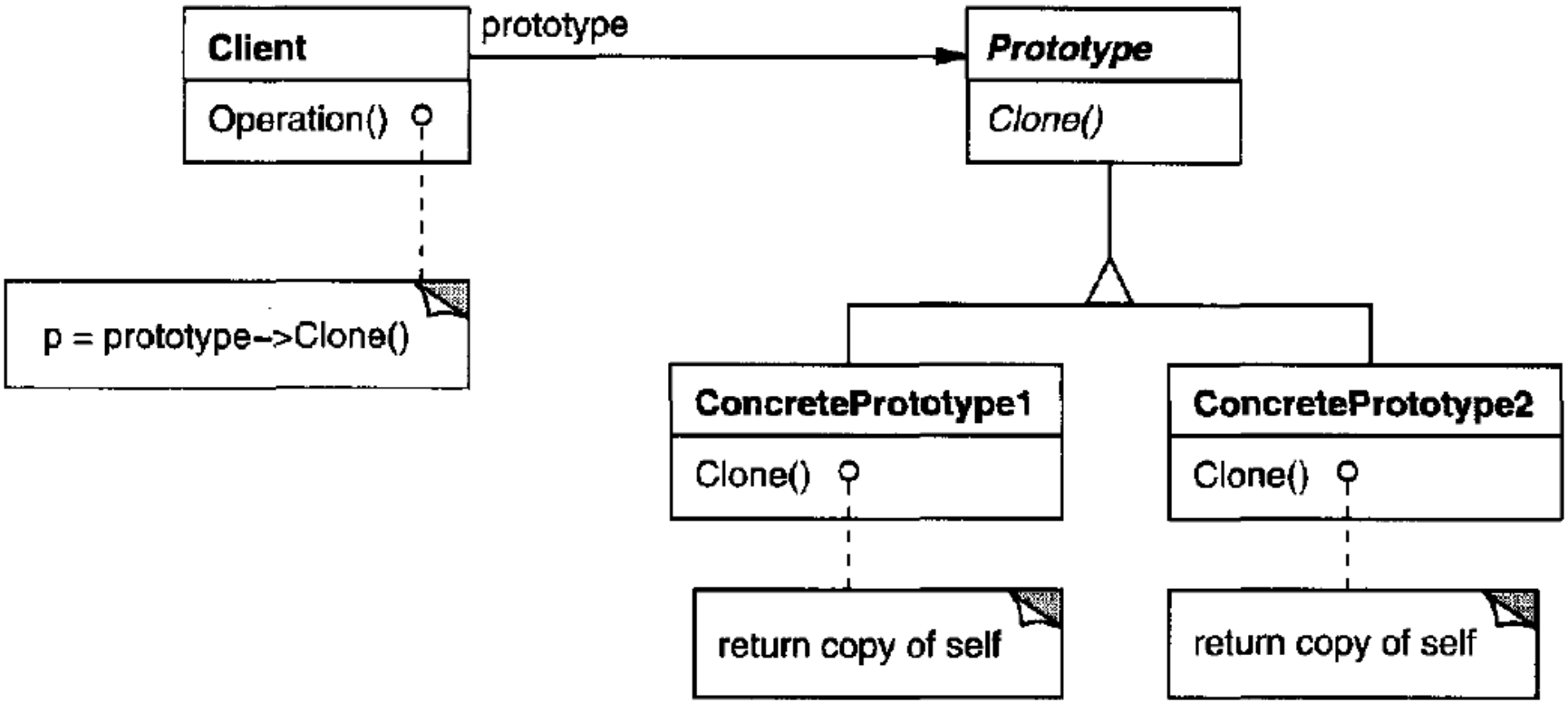
Prototype Patterns … Contd.

- **Applicability**
  - Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; *and*
  - When the classes to instantiate are specified at run-time, for example, by dynamic loading ; *or*
  - To avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*
  - • when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Prototype Patterns … Contd.

Indian Institute of Information Technology Kottayam

- **Structure**



Prototype Patterns … Contd.

- **Participants**
  - **Prototype (Graphic):** declares an interface for cloning itself.
  - **ConcretePrototype (Staff, WholeNote, HalfNote):** implements a noperation for cloning itself.
  - **Client (GraphicTool):** creates a new object by asking a prototype to clone itself.
- **Collaborations**
  - A client asks a prototype to clone itself.

Prototype Patterns … Contd.

Indian Institute of Information Technology Kottayam

- **Consequences**
  - Additional benefits of the Prototype pattern are listed below.
  1. *Adding and removing products at run-time.*
  2. *Specifying new objects by varying values*
  3. *Specifying new objects by varying structure*
  4. *Reduced subclassing*
  5. *Configuring an application with classes dynamically*
- **Implementation**
  - Consider the following issues when implementing prototypes
  1. *Using a prototype manager*
  2. *Implementing the Clone operation*
  3. *Initializing clones*

Conclusion

Indian Institute of Information Technology Kottayam

Creational Patterns

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Implementation in various languages like Python, Java

Thank you