GOF Creational Design Patterns

Prakash Badhe

prakash.badhe@vishwasoft.in

Vishwasoft Technologies

Objects Creation

In Java, the simplest way to create an instance of an object is by using the new operator.

Fred fr = new Fred(); //instance of Fred class

However, this really amounts to hard coding how you create the object within your program.

All of the creational patterns deal with the best way to create instances of objects. This is important because your program should not depend on how objects are created and arranged.

Creational Advantage

- In many cases, the exact nature of the object that is created could vary with the needs of the program from time to time and abstracting the object creation process into a special "creator" class can make your program more flexible and general.

Catalog - I

The Factory Method Pattern:provides a simple decision making class which returns one of several possible subclasses of an abstract base class depending on data it is provided.

The Abstract Factory Pattern: provides an interface to create and return one of several families of related objects.

Catalog - II

The Builder Pattern : separates the construction of a complex object from its representation, so that several different representations can be created depending on the needs of the program**.**

The Prototype Pattern : starts with an initialized and instantiated class and copies or clones it to make new instances rather than creating new instances.

The Singleton Pattern :provides a class of which there can be no more than one instance, and provides a single global point of access to that instance.

The Factory Method Pattern

To design system that opens image from file

- Several image formats exist (e.g., GIF, JPEG, etc.)
  - Each image format has different structure
- Method createImage of class Component creates Image objects
- Image is abstract class for images.
- Different types of Image objects (GIF image, JPEG image,, PNG image etc.) exist that derive from base Image

The Factory Method Pattern -II

Method createImage uses parameter to determine proper Image subclass from which to instantiate Image object

createImage("gif");

– Returns Image object with GIF data

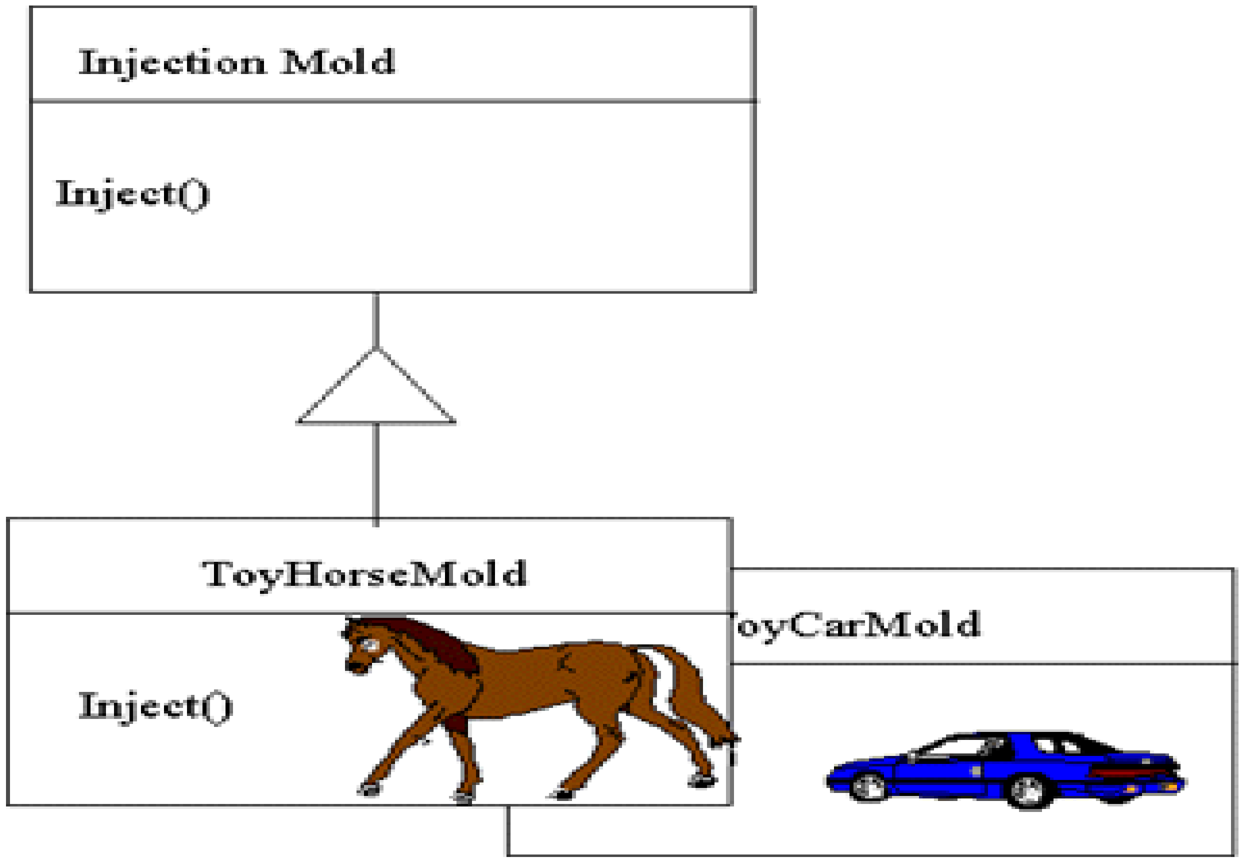
createImage("jpg");

– Returns Image object with JPEG data

Method createImage is called a factory method which

– Determines subclass to instantiate object at run time

Factory Method Analogy



Now Another Design Problem

1. Consider a user interface toolkit to support multiple look-and-feel standards.

2. For portability an application must not hard code its widget components for particular platformv(OS)look and feel.

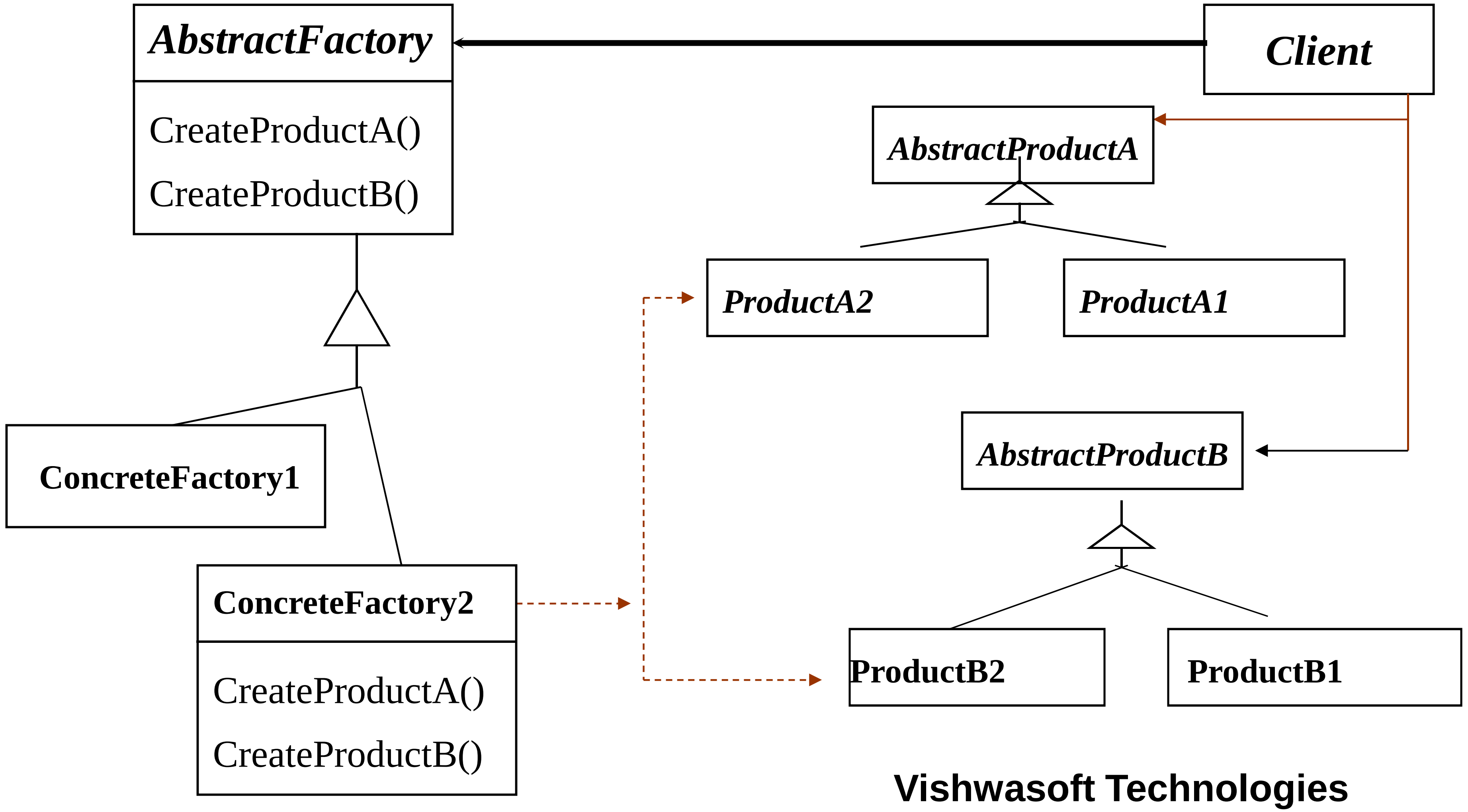
How to design the application so that incorporating new look and feel requirements will be easy?

The Solution is…

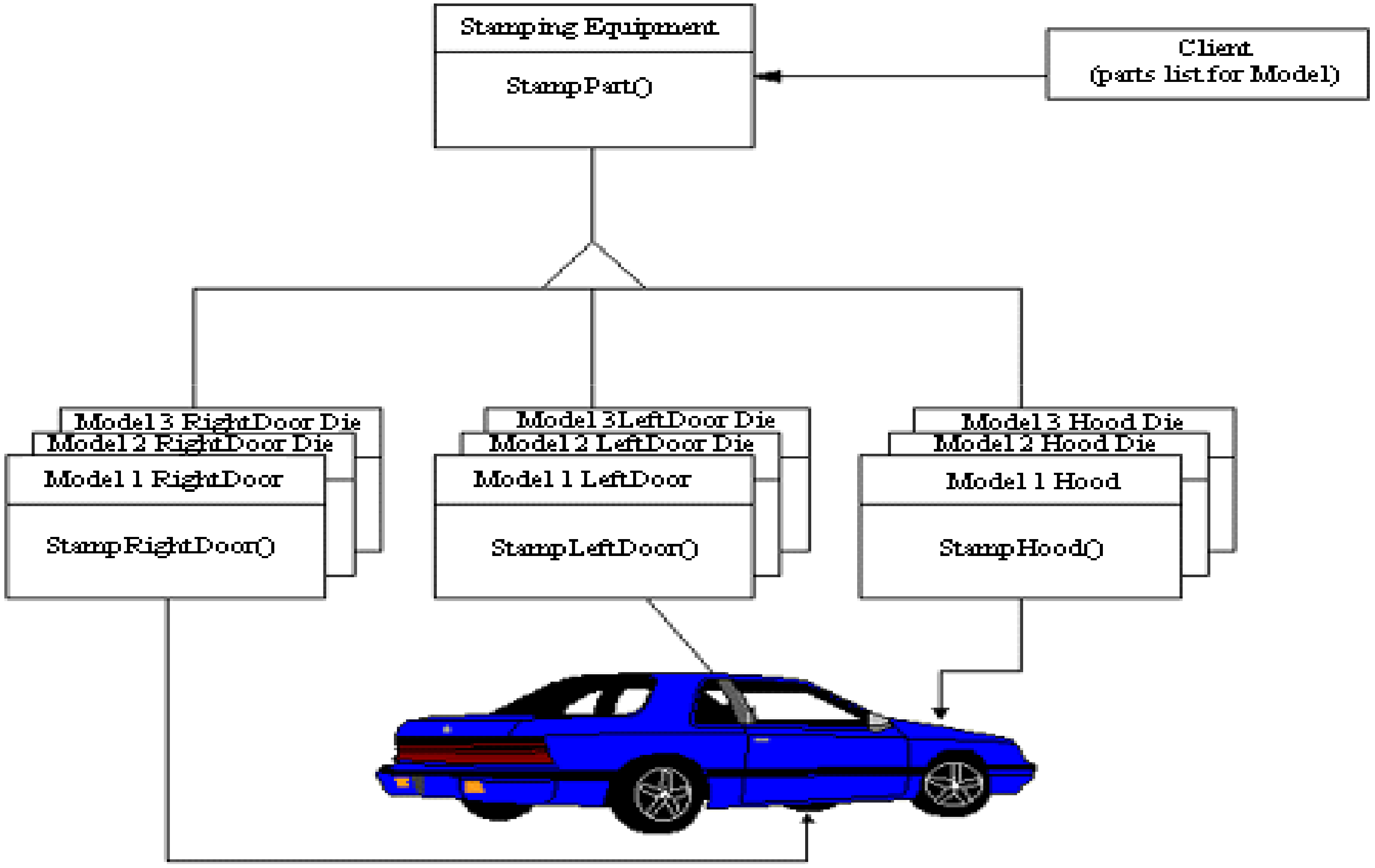
Use Abstract Factory Pattern

1. Define an abstract WidgetFactory class.
2. This class declares an interface to create different kinds of widgets.
3. There is one abstract class for each kind of widget and concrete subclasses implement widgets for different standards.
4. WidgetFactory offers an operation to return a new widget object for each abstract widget class. Clients call these operations to obtain instances of widgets without being aware of the concrete classes they use.

Abstract Factory: Structure



Abstract Factory : analogy



Abstract Factory : Participants and Communication

- AbstractFactory: Declares the interface for operations to create abstract product objects

- **ConcreteFactory**: Implements the operations to create concrete product objects.

- AbstractProduct: Declares an interface for a type of product object.

- ConcreteProduct: Defines a product object to be created by the corresponding factory.

- Client: Uses only the interface declared by the abstractFactory and AbstractProduct classes.

Abstract Factory Pattern – Consequences & Example

- Consequences:
  - Isolates concrete classes
  - Makes exchanging product families easy.
  - Promotes consistency among products
  - Supporting new kinds of products is not easy!
- Example : java.awt component drawing mechanism specific to the platform

Next Design Issue

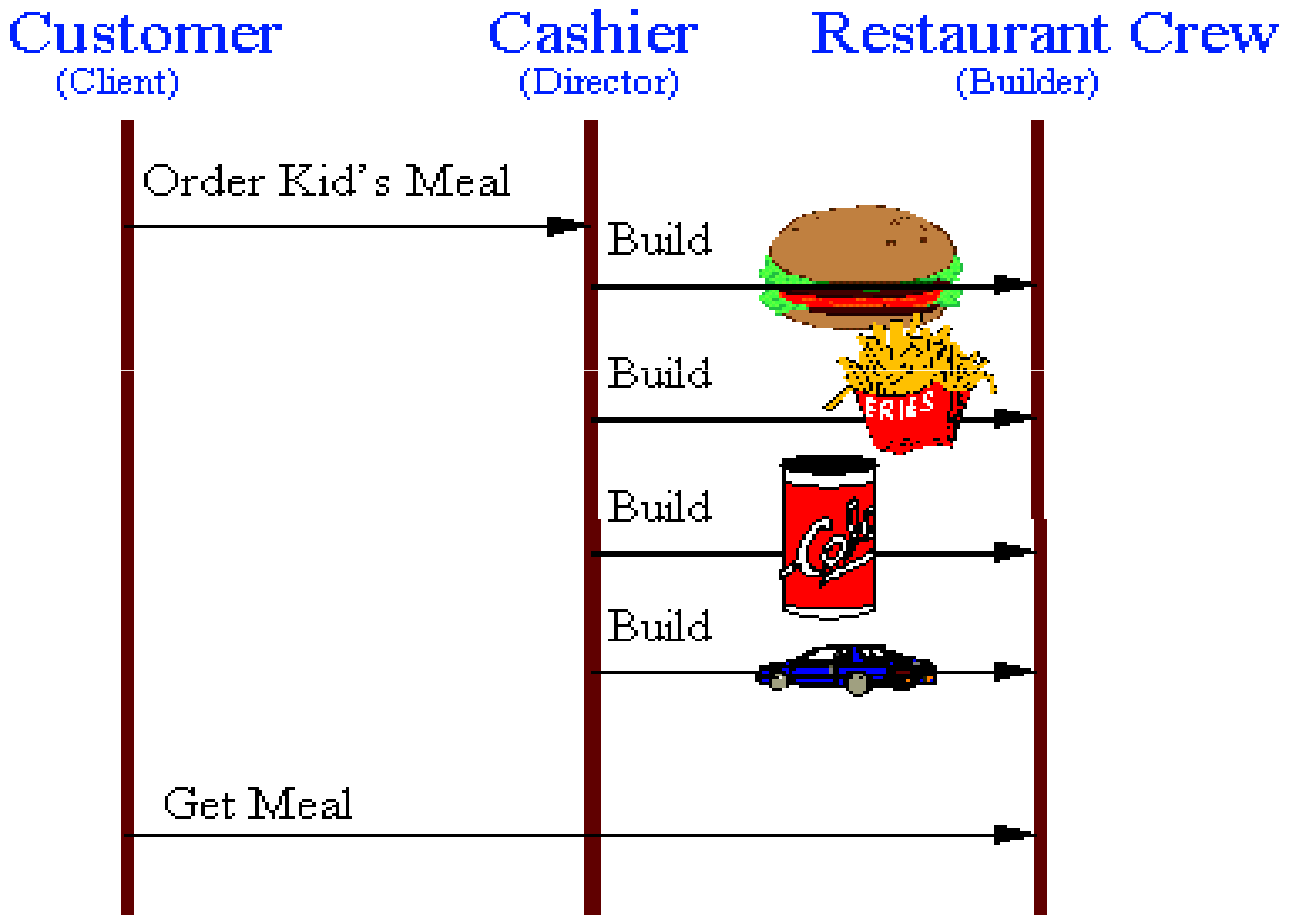
- We want the construction of objects separated from the representation so that same process can be used to create different types of objects and represented in different ways.
- WordPad opens the different types of files i.e. text,Word,RTF etc. and displays differently.

  -OpenFile( ) is the same process for reading from the file but to represent the text is in a different way.

Solution is : Builder Pattern

- The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.
- Builder focuses on constructing a complex object step by step

Builder Analogy



Builder consequences

- A Builder lets you vary the internal representation of the product it builds. It also hides the details of how the product is assembled.
- Each specific builder is independent of the others and of the rest of the program. This improves modularity and makes the addition of other builders relatively simple.
- Because each builder constructs the final product step-by-step, depending on the data, you have more control over each final product that a Builder constructs.

Object Creation Issues

- when creating an instance of a class is very

  time-consuming or complex in some way. then, rather than creating more instances, you make copies of the original instance, modifying them as appropriate.
- And the Creation process can be largely ignored by the creator.

Solution is Prototype Pattern

The Prototype design pattern creates new instances of classes by having initialized prototype instances copy or clone themselves.

Prototype: creation through delegation.

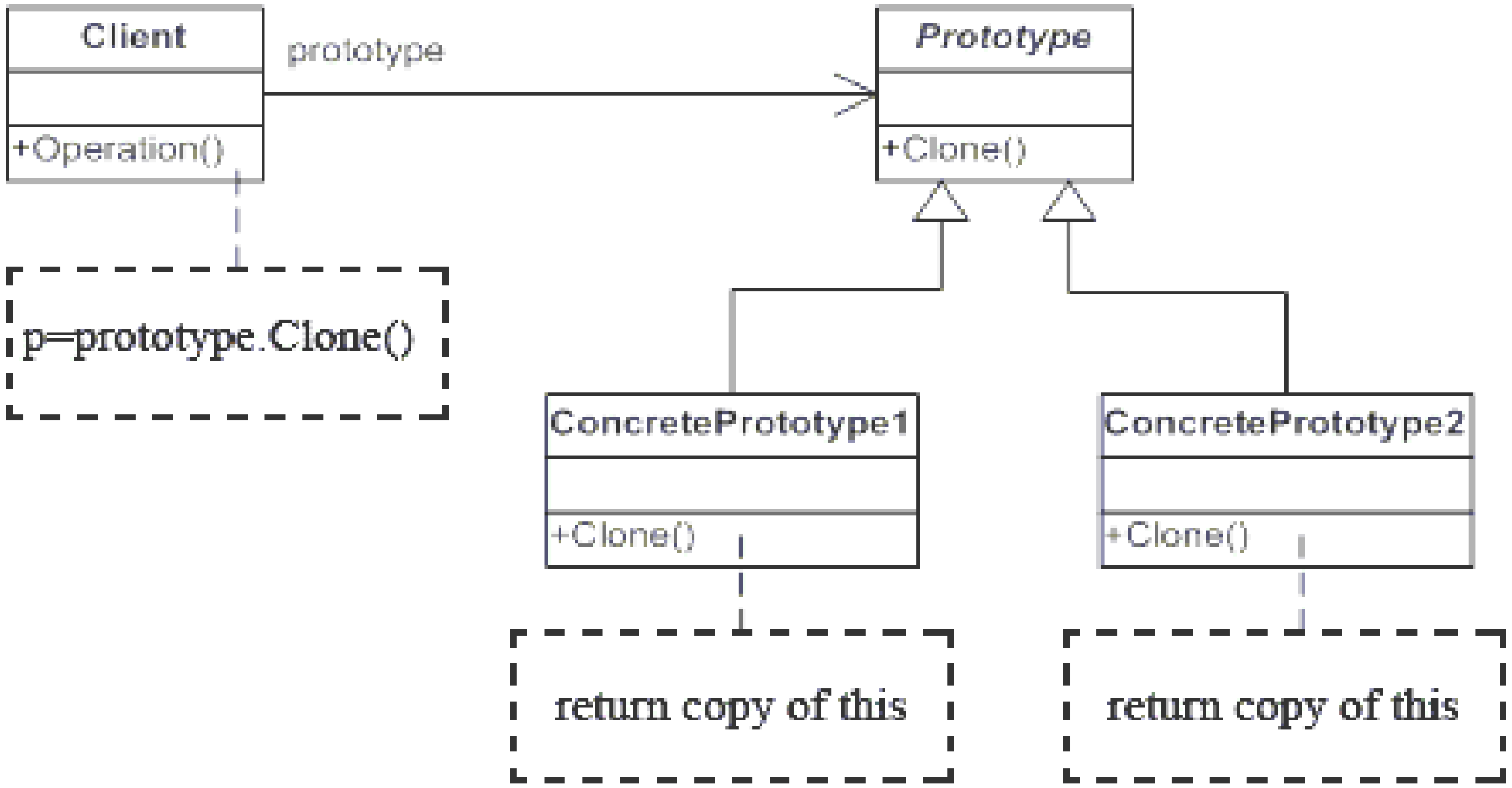
Prototype : Example

- An example of the Prototype design pattern
  is in a GUI building tool.
- For each graphic element which the user may customize and add to the application interface (buttons, text boxes, etc.), the user has many options to choose for style, color, font, size, etc.
- Keeping a single prototype of each GUI element on a palette for the user to choose from allows the user to customize each type once and drag them onto the interface project as desired.

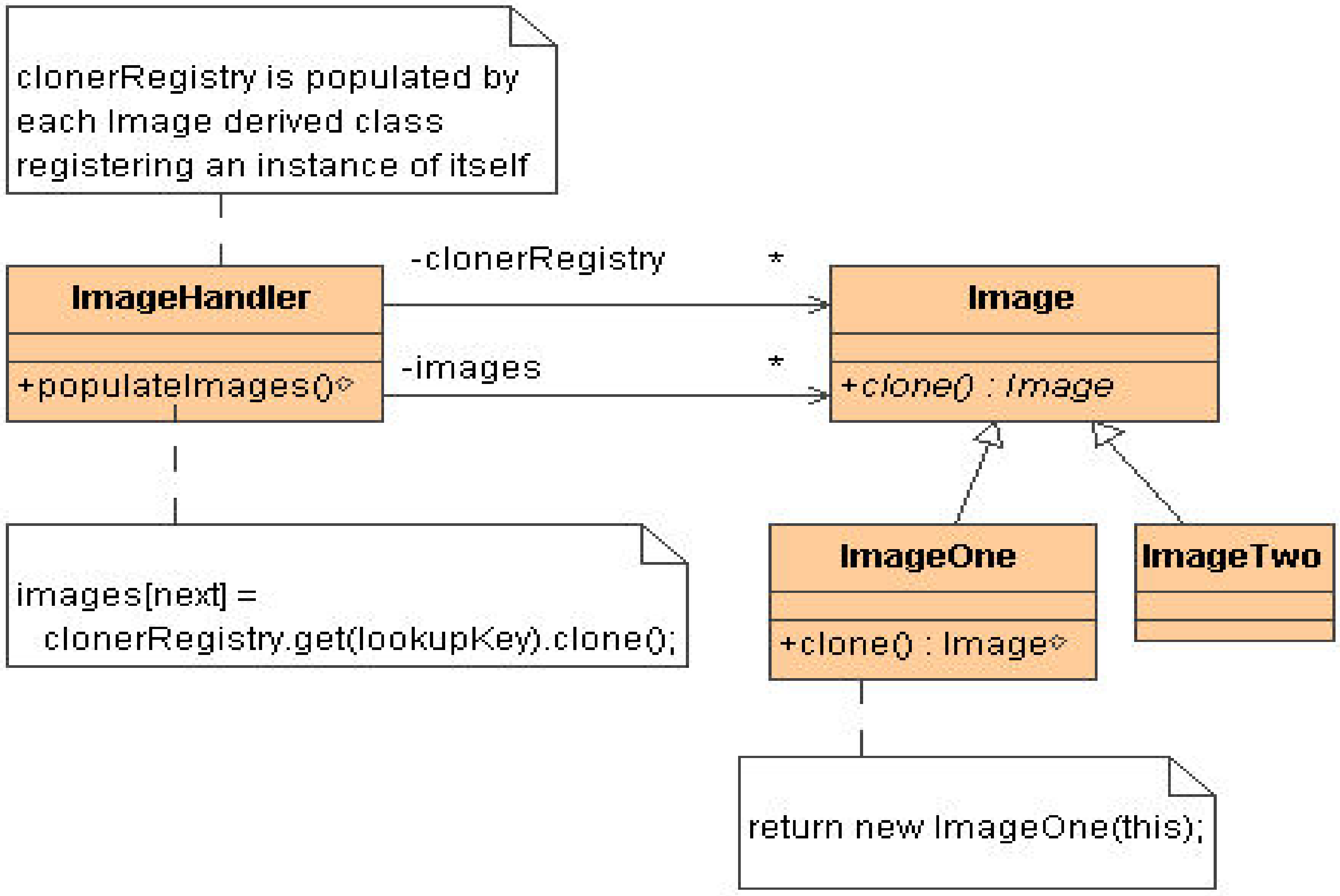
Prototype Example: GUI Building Tool

Not only does this produce a consistent look and feel across the application, but the GUI building tool need only call the instance's clone( ) method each time it is dragged from the palette to the GUI, regardless of the specific class selected by the user.

Prototype Std. Structure



Prototype Class diagram



clonerRegistry is populated by each Image derived class registering an instance of itself
ImageHandler
+populateImages()
-clonerRegistry
*
-images
*
Image
+clone() : Image
ImageOne
+clone() : Image
ImageTwo
images[next] = clonerRegistry.get(lookupKey).clone();
return new ImageOne(this);

Prototype : applicability

- Useful to design a system independent of how its objects are created, composed and represented
- Dynamic object specification
  - Specify the general class needed, but defer specifics until execution time
- No class hierarchies of "factory" classes are needed
  – Can avoid building a class hierarchy of factories paralleling the class hierarchy of products.

Prototype : Participants

- Client - The client object asks the prototype to clone itself
- Prototype - Data object defining an interface for creating clones of its self
  - A "clone" function, returning a copy of the original object
- ConcretePrototype - Implements the cloning operation defined in the Prototype class
  - Copies the data and state of the original object

Prototype : Collaborations

- Collaborations between participants are initiated by the Client
- Prototypical instance of any ConcretePrototypes to be used are created
  - **Initialized to the desired state of an instance typical for the Client's use**
- Prototype is cloned whenever a new instance of this type is desired

Prototype : consequences

- Alternative to Abstract Factory (and Builder, somewhat)
  - Instantiation code not in a separate concrete "factory" class
- Isolates product (data) class details from the client for instantiation
  - Streamlines client-side code for product instantiation process
  - Eliminates case-by-class handling of instantiation (same method)
  - Requires a clone (or getClone) method in each Prototype subclass

Prototype : consequences

- Difficult to reuse preexisting classes or those from external sources
- A deepClone() method virtually requires all the member values of the Prototype subclass be declared to implement Serializable
- Classes with circular references to other classes can’t really be cloned

Prototype : consequences

- Having prototype classes to copy implies that you have sufficient access to the data or methods in these classes to change them after cloning.
- This may require adding data access methods to these prototype classes so that you can modify the data once you have cloned the class.

Single Global Object

- In some cases you need to make sure that there can be one and only one instance of a class. For example, your system can have only one window manager or print spooler, or a single point of access to a database engine.
- ‘**ContolPanel’** window is always only one irrespective of number of times trying to create it.

Single Instance

The clients should be prevented from creating more than one instances of such classes and should be always directed towards the global single instance available.

The Solution is ‘Singleton’ Pattern

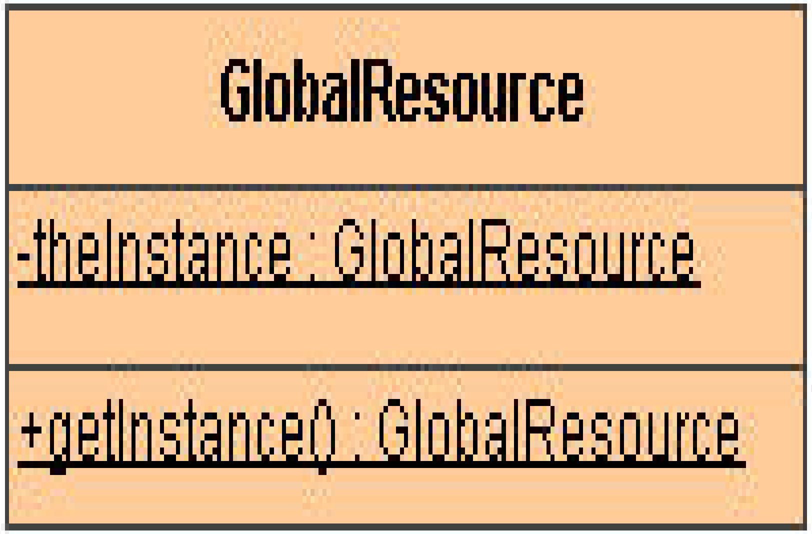
- **Intent**

  Ensure a class has only one instance, and provide a global point of access to it.

- **Problem**

  Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Singleton Class diagram


GlobalResource
-theInstance : GlobalResource
+getInstance() : GlobalResource

Singleton Implementation

- Make the class of the single instance object responsible for creation, initialization, access, and enforcement.
- Declare the instance as a private static data member. Provide a public static member function that encapsulates all initialization code, and provides access to the instance.
- The client calls the **accessor** function (using the class name and scope resolution operator) whenever a reference to the single instance is required.
- Can be extended to support access to an application-specific number of instances.

Singleton Consequences

- It can be difficult to subclass a Singleton, since this can only work if the base Singleton class has not yet been instantiated.
- You can easily change a Singleton to allow a small number of instances where this is allowable and meaningful.
- Singleton Application implementation can be system specific.