DESIGN PATTERNS

Course 3

CONTENT

- Creational patterns
  - Singleton patterns
  - Builder pattern
  - Prototype pattern
  - Factory method pattern
  - Abstract factory pattern

CREATIONAL PATTERNS

- Design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation

- Make a system independent of the way in which objects are created, composed and represented

- Easily Change
  - What gets created?
  - Who creates it?
  - When is it created?

CREATIONAL PATTERNS

Patterns used to abstract the process of instantiating objects.

- class-scoped patterns
  - uses inheritance to choose the class to be instantiated
    - Factory Method

- object-scoped patterns
  - uses delegation
    - Abstract Factory
    - Builder
    - Prototype
    - Singleton

CREATIONAL PATTERNS

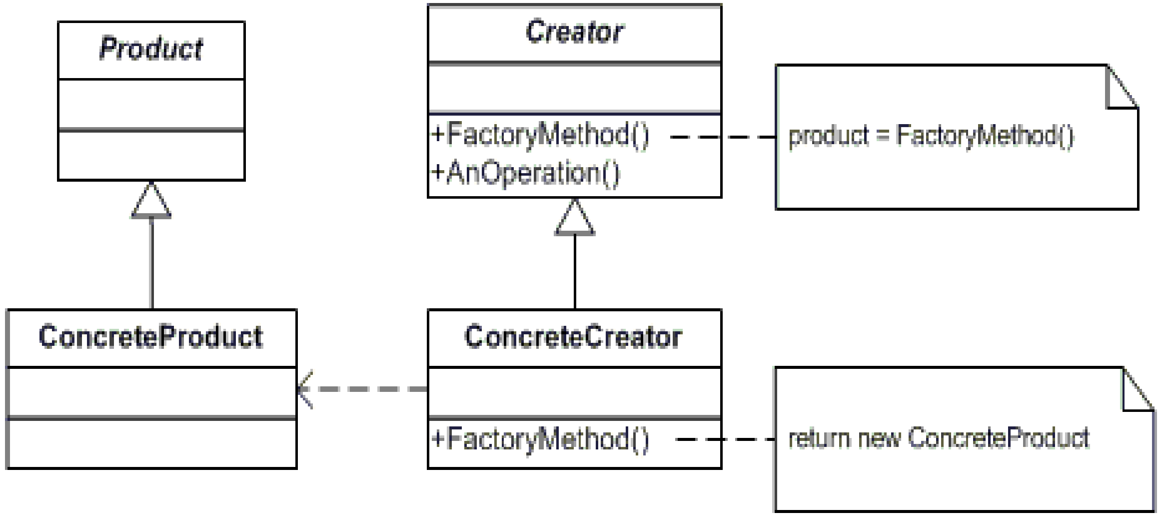
- **Abstract factory** provides an interface for creating families of related objects, without specifying concrete classes
- **Factory method** defines an interface for creating objects, but lets subclasses decide which classes to instantiate
- **Builder** separates the construction of a complex object from its representation, so that the same construction process can create different representation
- **Prototype** specifies the kind of objects to create using a prototypical instances
- **Singleton** ensures that a class has only one instance, and provides a global point of access to that instance

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.
  - Defining a "virtual" constructor.
  - The new operator considered harmful.

- Problem
  - A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

FACTORY METHOD



Product

- Defines the interface for objects the factory method creates.

ConcreteProduct

- Implements the Product interface.

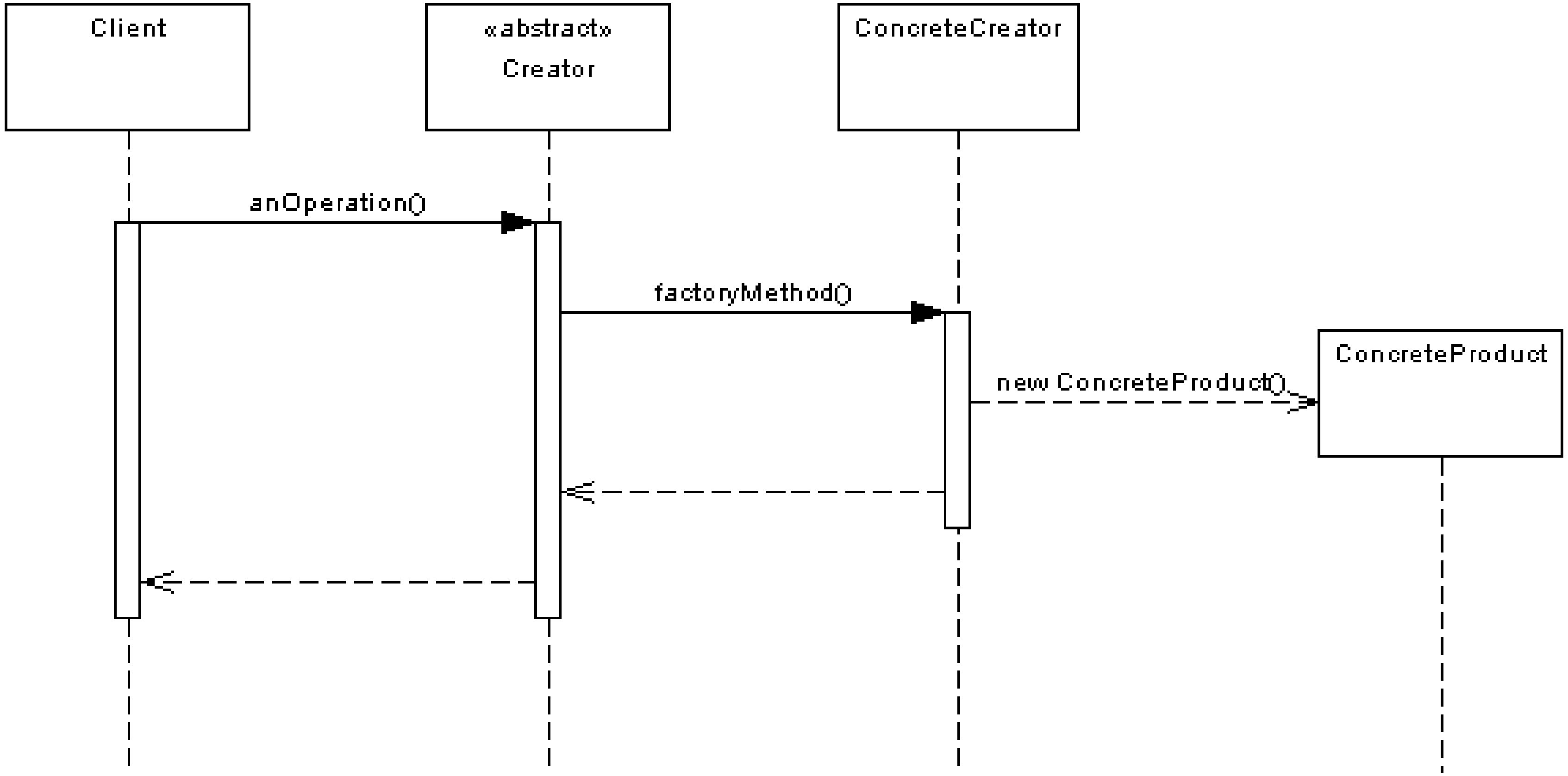
Creator(also referred as **Factory** because it creates the Product objects)

- Declares the method *FactoryMethod*, which returns a Product object. May call the generating method for creating Product objects

ConcreteCreator

- Overrides the generating method for creating ConcreteProduct objects

FACTORY METHOD. PATTERN INTERACTION



FACTORY METHOD. EXAMPLE

```
interface Currency {
    String getSymbol();
}

// Concrete Rupee Class code
class Rupee implements
Currency {
    @Override
    public String getSymbol() {
        return "Rs";
    }
}
```

```
// Concrete SGD class Code
class SGDDollar implements Currency {
    public String getSymbol() {
        return "SGD";
    }
}
// Concrete US Dollar code
class USDollar implements Currency {
        public String getSymbol() {
        return "USD";
    }
}
```

```
class CurrencyFactory {// Factroy Class code
    public static Currency createCurrency (String country) {
     if (country. equalsIgnoreCase ("India")){
            return new Rupee();
     }else if(country. equalsIgnoreCase ("Singapore")){
            return new SGDDollar();
     }else if(country. equalsIgnoreCase ("US")){
            return new USDollar();
     }
     throw new IllegalArgumentException("No such currency");
     } }
public class Factory {// Factory client code
     public static void main(String args[]) {
            String country = args[0];
            Currency rupee = CurrencyFactory.createCurrency(country);
            System.out.println(rupee.getSymbol());
     } }
```

FACTORY METHOD

Advantage

- eliminates the need to bind application specific classes into your code; your code deals with ***Product*** interface implemented by ***ConcreteProduct*** subclasses

Potential disadvantage

- clients might have to subclass the ***Creator*** class just to create a particular ***ConcreteProduct*** object

Provides hooks for subclasses

- Factory Method gives subclasses a hook for providing an extended version of an object

Connects parallel class hierarchies

ABSTRACT FACTORY

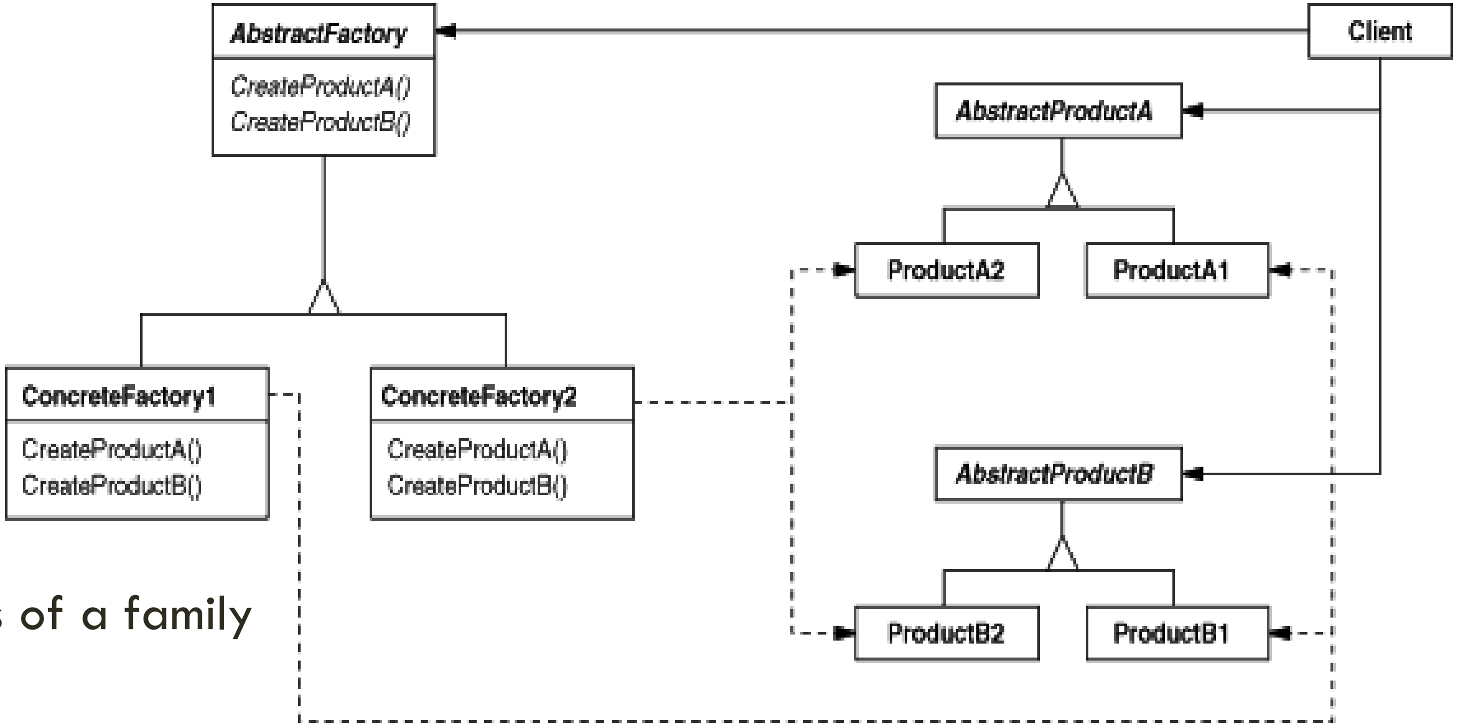
Intent

- Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes.

Problem

- If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc.

ABSTRACT FACTORY



AbstractFactory

- provides an interface for creating products of a family

ConcreteFactory

- implements the operations to create concrete products

AbstractProduct

- declares the interface for concrete products

ConcreteProduct

- provides an implementation for the product created by the corresponding ConcreteFactory

Client

- creates products by calling the ConcreteFactory uses the AbstractProduct interface

ABSTRACT FACTORY. EXAMPLE

When to use Abstract Factory design pattern?

- the system needs to be independent from the way the products it works with are created.
- the system is or should be configured to work with multiple families of products.
- a family of products is designed to work only all together.
- the creation of a library of products is needed, for which is relevant only the interface, not the implementation, too.

Example

- Look and Feel

ABSTARCT FACTORY. EXAMPLE

```
//Abstract Product
interface Button { void paint(); }
//Abstract Product
interface Label { void paint(); }
//Abstract Factory
interface GUIFactory {
        Button createButton();
        Label createLabel();
}
//Concrete Factory
class WinFactory implements GUIFactory {
        public Button createButton() { return new WinButton(); }
        public Label createLabel() { return new WinLabel(); }
}
```

```
//Concrete Factory
class OSXFactory implements GUIFactory {
    public Button createButton() { return new OSXButton(); }
    public Label createLabel() { return new OSXLabel(); }
}
//Concrete Product
class OSXButton implements Button {
    public void paint() { System.out.println("I'm an OSXButton"); }
}
//Concrete Product
class WinButton implements Button {
    public void paint() { System.out.println("I'm a WinButton"); }
}
```

ABSTARCT FACTORY. EXAMPLE

```
//Concrete Product
class OSXLabel implements Label {
    public void paint() { System.out.println("I'm an OSXLabel"); }
}

//Concrete Product
class WinLabel implements Label {
    public void paint() { System.out.println("I'm a WinLabel"); }
}
```

```
//Client application is not aware about the how the product is created. Its only responsible to give a name of
//concrete factory
class Application {
  public Application(GUIFactory factory) {
      Button button = factory.createButton();
      Label label = factory.createLabel();
      button.paint();
      label.paint();
  }
}
```

ABSTARCT FACTORY. EXAMPLE

```
public class ApplicationRunner {
    public static void main(String[] args) {
         new Application(createOsSpecificFactory());
    }
    public static GUIFactory createOsSpecificFactory(){
        String osname = System.getProperty("os.name").toLowerCase();
        if(osname != null && osname.contains("windows"))
            return new WinFactory();
        else
            return new OSXFactory();
   }
}
```

ABSTRACT FACTORY VS. FACTORY METHOD

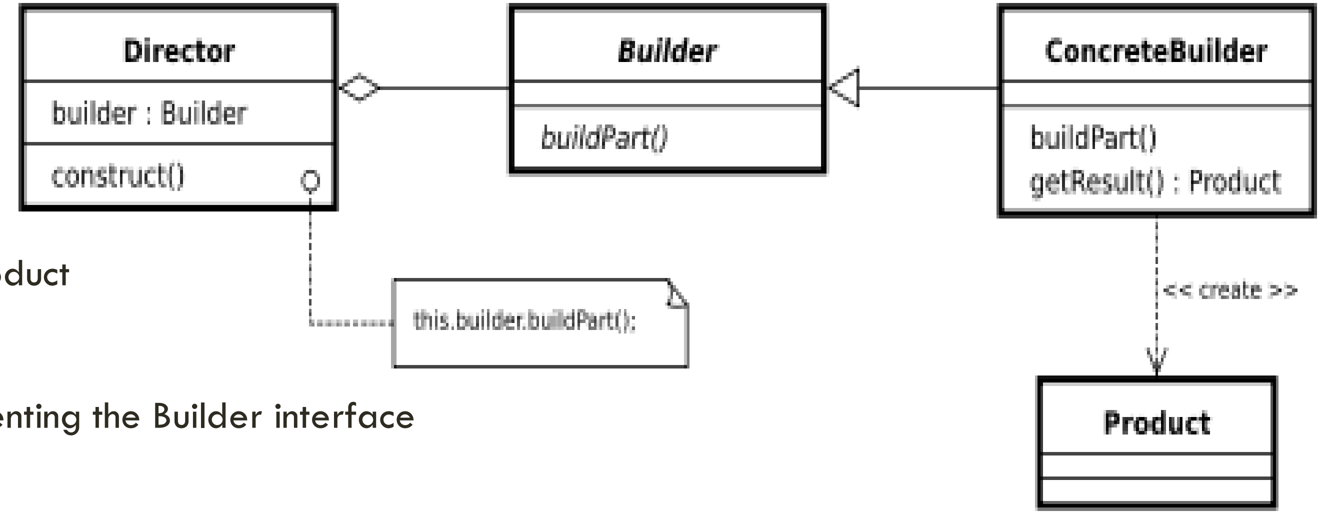
- Both patterns are good at decoupling applications from specific implementations
- Both patterns create objects – that's their job
- Factory Method uses inheritance to decouple applications form specific implementations
- Abstract Factory uses object composition to decouple applications form specific implementations

BUILDER

- Intent
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - Parse a complex representation, create one of several targets

- Problem
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
  - the construction process must allow different representations for the object that is constructed

BUILDER



Builder

- specifies an abstract interface for creating parts of a Product

ConcreteBuilder

- 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

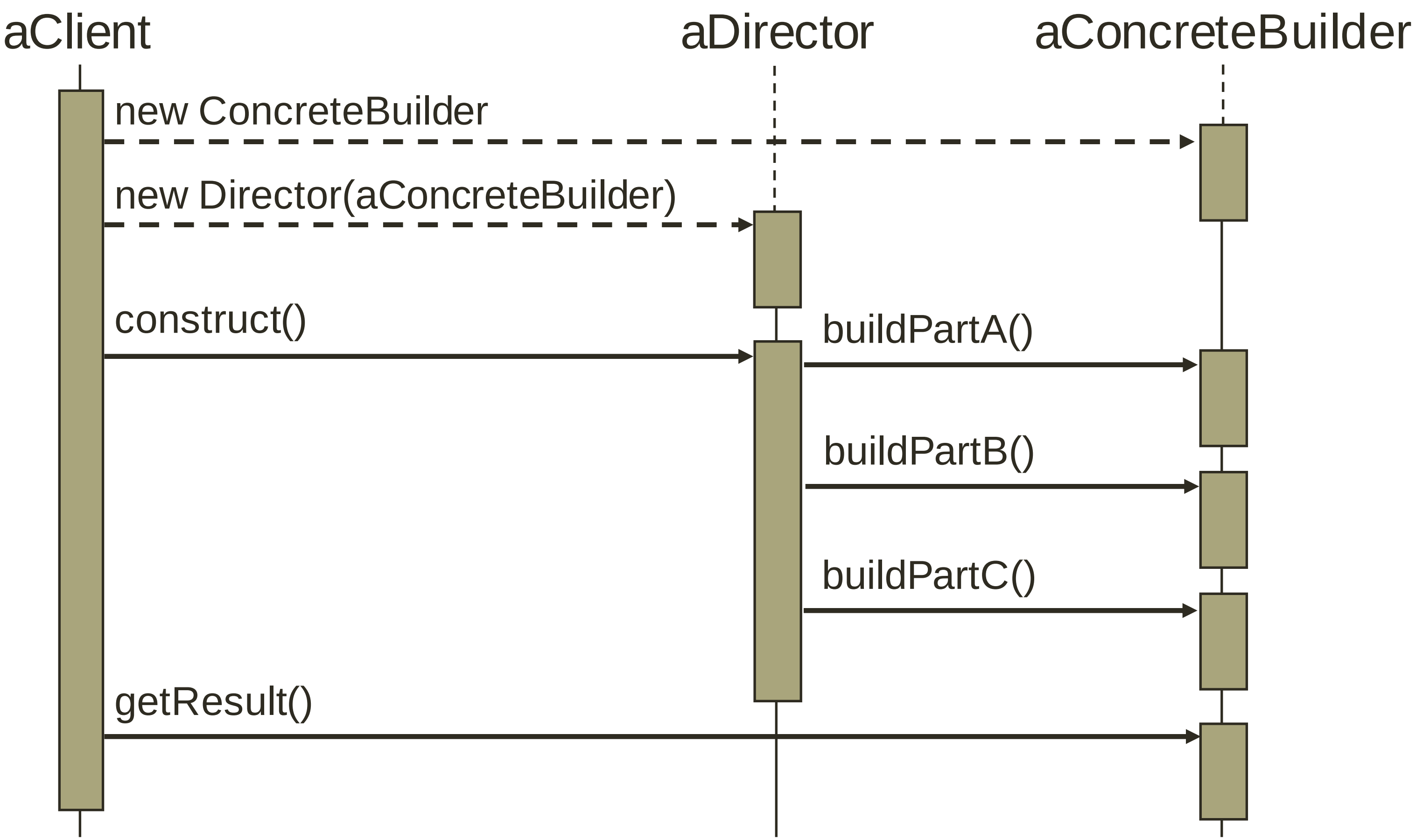
Director

- constructs an object using the Builder interface

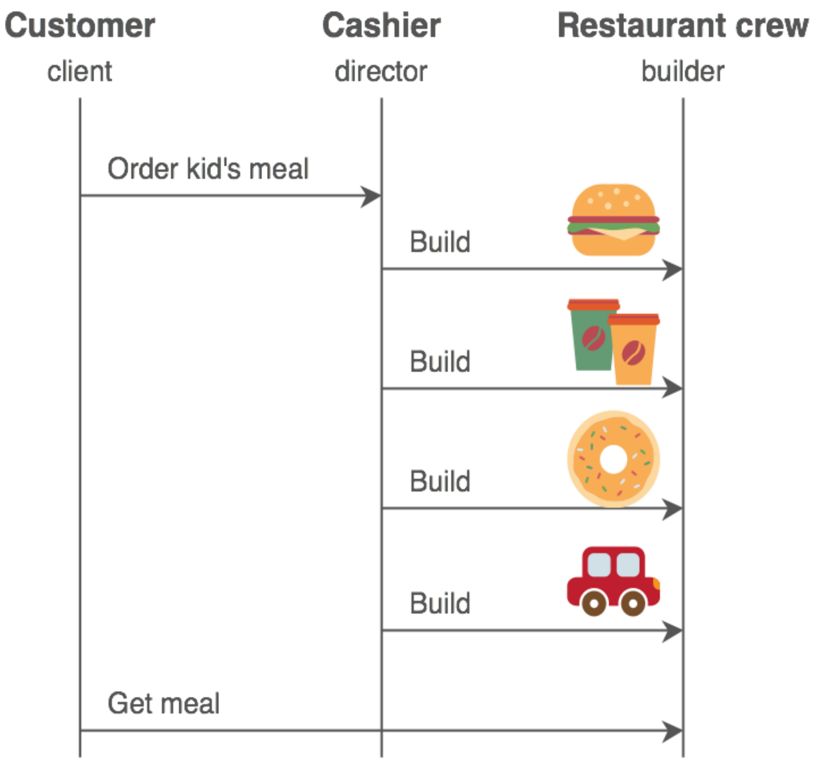
Product

- Represents the complex object under construction
- Includes classes that define the constituent parts including the interfaces for assembling the parts into the final result

BUILDER. PATTERN INTERACTION



EXAMPLE



BUILDER. MOTIVATIONAL EXAMPLE

User class

```
public class User {
    //required
    private final String firstName;
    private final String lastName;
    //optional
    private final int age;
    private final String phone;
    private final String address;
...
}
```

Creation modes

```
public User(String firstName, String lastName) {  this(firstName, lastName, 0);   }

 public User(String firstName, String lastName, int age) {  this(firstName, lastName, age, "");

 public User(String firstName, String lastName, int age, String phone) {  this(firstName, lastName, age,
phone, ""); }

 public User(String firstName, String lastName, int age, String phone, String address) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.phone = phone;
    this.address = address;
}
```

BUILDER. MOTIVATIONAL EXAMPLE

```
public class User {
    private String firstName; // required
    private String lastName; // required
    private int age; // optional
    private String phone; // optional
    private String address;  //optional

    public String getFirstName() {
            return firstName;
    }
    public void setFirstName(String firstName) {
            this.firstName = firstName;
     }
    public String getLastName() {  return lastName;}
     public void setLastName(String lastName) {
       this.lastName = lastName;
    }
    public int getAge() {  return age;  }
    public void setAge(int age) {  this.age = age;    }
    public String getPhone() {  return phone;    }
    public void setPhone(String phone) {
      this.phone = phone;
     }
    public String getAddress() { return address; }
    public void setAddress(String address) {
        this.address = address;
    }
}
```

BUILDER. MOTIVATIONAL EXAMPLE

```java
public class User {
    private final String firstName; // required
    private final String lastName; // required
    private final int age; // optional
    private final String phone; // optional
    private final String address; // optional

    private User(UserBuilder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.age = builder.age;
        this.phone = builder.phone;
        this.address = builder.address;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public int getAge() {
        return age;
    }
    public String getPhone() {
        return phone;
    }
    public String getAddress() {
        return address;
    }
    public static class UserBuilder {
        private final String firstName;
        private final String lastName;
        private int age;
        private String phone;
        private String address;

        public UserBuilder(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }

        public UserBuilder age(int age) {
            this.age = age;
            return this;
        }
        public UserBuilder age(int age) {
            this.age = age;
            return this;
        }
        public UserBuilder phone(String phone) {
            this.phone = phone;
            return this;
        }
        public UserBuilder address(String address) {
            this.address = address;
            return this;
        }
        public User build() {
            return new User(this);
        }
    }
}
```

BUILDER. MOTIVATIONAL EXAMPLE

Observations

- The User constructor is private, which means that this class can not be directly instantiated from the client code.
- The class is immutable. All attributes are final and they're set on the constructor. Only provide getters for them.
- The builder constructor only receives the required attributes and this attributes are the only ones that are defined "final" on the builder to ensure that their values are set on the constructor.

Instantiation

```
public User getUser() {
    return new
            User.UserBuilder("Jhon", "Doe")
            .age(30)
            .phone("1234567")
            .address("Fake address 1234")
            .build();
}
```

BUILDER. MOTIVATIONAL EXAMPLE

Where are the attributes validated?

```
public User build() {
    if (age 120) {
        throw new IllegalStateException(“Age out of range”);
    }
    return new User(this);
}
```

BUILDER

Advantages

- Allows you to vary a product's internal representation
- Encapsulates code for construction and representation
- Provides control over steps of construction process

Disadvantages

- Requires creating a separate ConcreteBuilder for each different type of Product

PROTOTYPE

- Intent
  - Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype
  - The **new** operator considered harmful.

- Problem
  - When an application needs the flexibility to be able to specify the classes to instantiate at run time
  - Avoiding the creation of a factory hierarchy is needed
  - When instance of a class have only very few different combinations of state, it is more convenient to copy an existing instance than to create a new ne

PROTOTYPE. EXAMPLES

1. In Java: usage of the clone() method or de-serialization when deep copies are needed

2. The mitotic division of a cell - resulting in two identical cells

3. Building stages for a game that uses a maze and different visual objects that the character encounters it is needed a quick method of generating the haze map using the same objects: wall, door, passage, room...

PROTOTYPE

Prototype

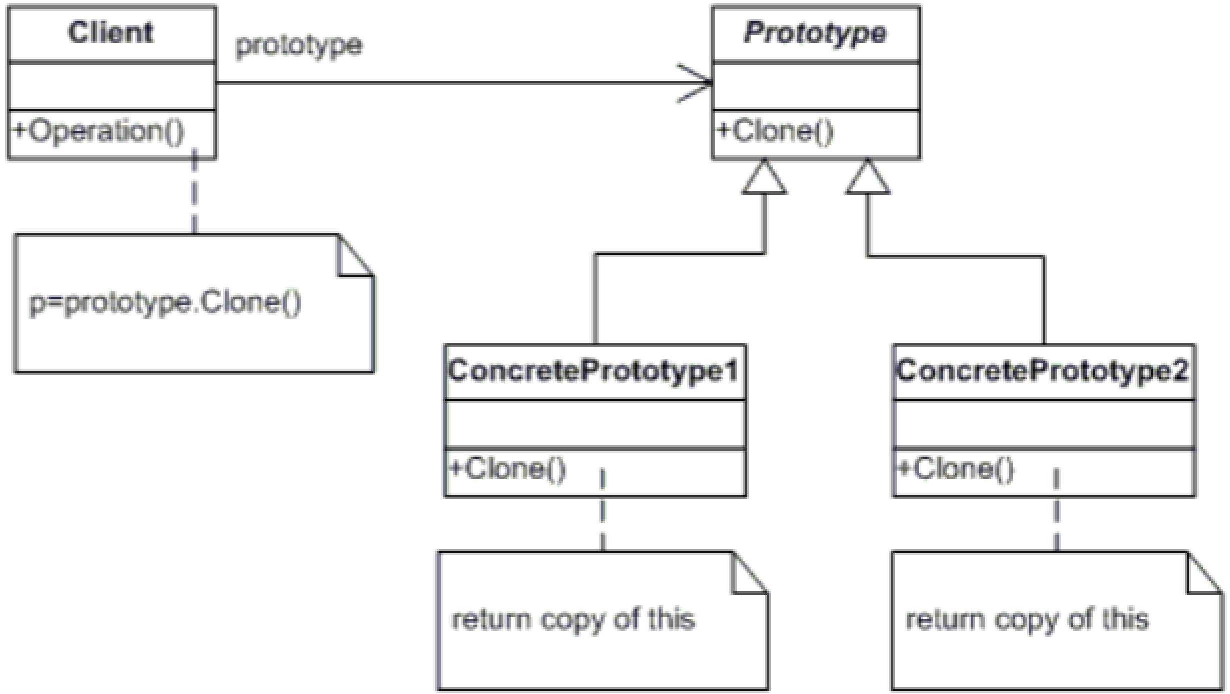
- ❑Declares an interface for cloning itself

ConcretePrototype

- ❑Implements an operation for cloning itself

Client

- ❑Creates a new object by asking a prototype to clone itself and then making required modifications



PROTOTYPE. IMPLEMENTATION

Implementation

- Declare an interface that contains a *clone()* method
- A concrete class that implements the interface

Clone can be implemented either as a deep copy or a shallow copy:

• In a deep copy, all objects are duplicated,

• In a shallow copy, only the top-level objects are duplicated and the lower levels contain references.

PROTOTYPE

```java
public interface Prototype {
    public abstract Object clone ( );
}
public class ConcretePrototype implements Prototype {
    public Object clone() {
        return super.clone();
    }
}
public class Client {
    public static void main( String arg[] ) {
        ConcretePrototype obj1 = new ConcretePrototype ();
        ConcretePrototype obj2 = (ConcretePrototype)obj1.clone();
    }
}
```

EXERCICE:

Propose a cache management system for a list of figures

PROTOTYPE

Benefits

- Hides the complexities of making new instances from the client,
- Provides the option for the client to generate objects whose type is not known,
- In some circumstances, copying an object can be more efficient than creating a new object.

Uses

- Prototype should be considered when a system must create new objects of many types in a complex class hierarchy.

Drawbacks

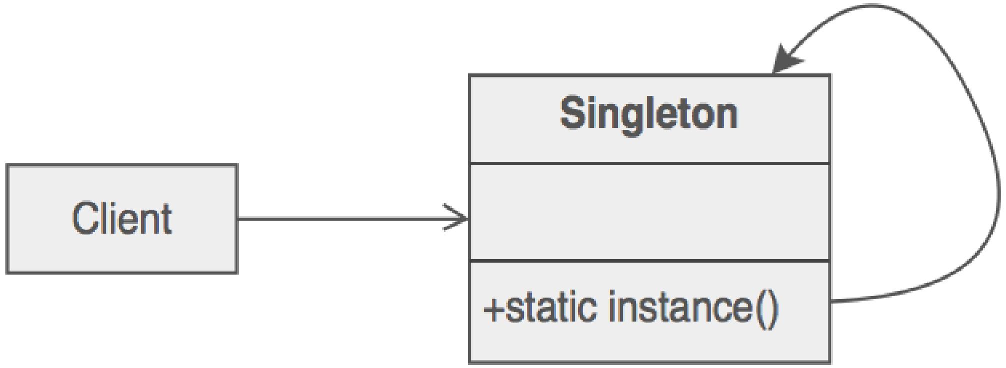
- A drawback to using the Prototype is that making a copy of an object can sometimes be complicated.

Abstract Factory ans Protype Patterns may work together

SINGLETON

- Intent
  - Ensure a class has only one instance, and provide a global point of access to it.
  - Encapsulated "just-in-time initialization" or "initialization on first use“

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



SINGLETON. EXAMPLES

1. Incremental counter, the simple counter class needs to keep track of an integer value that is being used in multiple areas of an application

2. Logging

3. Reading configuration files that should only be read at startup time and encapsulating them in a Singleton.

SINGLETON

How to implement?

- Define a private **static** attribute in the "single instance" class.
- Define a public **static** accessor function in the class.
- Do "lazy initialization" (creation on first use) in the accessor function.
- Define all constructors to be **protected** or **private.**
- Clients may only use the accessor function to manipulate the Singleton

SINGLETON – IMPLEMENTATION EXAMPLE

```
public class Singleton {
   private static Singleton instance = null;
   protected Singleton() {
         // Exists only to defeat instantiation.
   }
   public static Singleton getInstance() {
       if(instance == null) {
              instance = new Singleton();
       }
       return instance;
   }
}
```

What problem can occur in case of Java language?

SINGLETON – IMPLEMENTATION EXAMPLE. POSSIBLE SOLUTION

```
public class Singleton {
        private static volatile Singleton instance = null;
        // private constructor
        private Singleton() { }
        public static Singleton getInstance() {
                if (instance == null) {
                    synchronized (Singleton.class) {
                      // Double check
                      if (instance == null) {
                          instance = new Singleton();
                      }
                    }
                }
        return instance;
}
```

PROBLEM:
Multithreading

SINGLETON

- Singleton vs static variables
  - The advantage of Singleton over global variables is that you are absolutely sure of the number of instances when you use Singleton, and, you can change your mind and manage any number of instances
- When is Singleton unnecessary?
  - most of the time – visibility of objects
  - when it's simpler to pass an object resource as a reference to the objects that need it, rather than letting objects access the resource globally
- Global data
  - Transforming global data into singletons

SINGLETON. PRO AND CONS

Positive

- Lazy instantiation
  - the singleton variable will not get memory until the property or function designated to return the reference is first called
- Static Initialization
  - memory is allocated to the variable at the time it is declared. The instance creation takes place behind the scenes when any of the member singleton classes is accessed for the first time
  - private static Singleton instance = new Singleton()

Negative

- A singleton class has the responsibility to create an instance of itself along with other business responsibilities.
- Singleton classes cannot be sub classed.
- Singletons can hide dependencies.

QUESTION

- Which creational pattern can be used in the bank application?
  - Explain the choice

  - Creational patterns
    - Singleton patterns
    - Builder pattern
    - Prototype pattern
    - Factory method pattern
    - Abstract factory pattern