Foundations of Software Engineering

Creational Patterns

Fall 2016
Department of Computer Science Ben-Gurion university

Based on slides of:
Nurit Gal-oz, Department of Computer Science Ben-Gurion University
Mira Balaban Department of Computer Science Ben-Gurion university
F. Tip. IBM T J Watson Research Center.
Creational Patterns

- **Purpose**
  - deal with object creation mechanisms
  - abstract the process of creating objects
  - make a system unaware of how objects are created, composed, and represented

- **What they do**
  - encapsulate knowledge about which concrete classes a system uses (access created objects via interfaces)
  - hide how instances are created

- **Provide flexibility w.r.t.**
  - types of created objects
  - responsibility for creation
  - how and when objects are created
Creational Patterns

- Abstract Factory
- Factory Method
- Singleton
- Builder
- Prototype
Maze Game: Example to illustrate various creational patterns

- Simulation of “maze” computer game.
- Objectives:
  - find your way out of a maze
  - solve problems
  - create map
Running Example

• Building a maze for a computer game.

• A Maze is a set of Rooms.

• A Room knows its neighbors.
  – a wall
  – a door
Maze Game: Example to illustrate various creational patterns

- a Maze consists of a number of Rooms
  - each Room has 4 sides: North, South, East, West
  - on each side of a room is a Door or a Wall
- abstract superclass MapSite of Room, Door, Wall has method enter()
  - behavior depends on the kind of subclass
- class MazeGame has static method createMaze() for creating a Maze
An enumerated type: Direction

```java
final class Direction {
    private Direction(String n){ _name = n; }

    public final static Direction North =
        new Direction("North");

    public final static Direction South =
        new Direction("South");

    public final static Direction East =
        new Direction("East");

    public final static Direction West =
        new Direction("West");

    public String toString(){ return _name; }
    private String _name;
}
```
UML Diagram for Maze Game

**Maze**
- rooms
  + addRoom()

**MapSite**
+ enter()

**MazeGame**
+ createMaze(): Maze

**Room**
- roomNr: int
- roomCnt
- northSide: MapSite
- southSide: MapSite
- eastSide: MapSite
- westSide: MapSite
+ enter()
+ setSide(Direction, MapSite)
+ getSide(Direction): MapSite
+ Room()

**Wall**
- wallNr
- wallCnt
+ enter()
+ Wall()

**Door**
- isOpen
- room1: Room
- room2: Room
- doorNr
- doorCnt
+ enter()
+ otherSideFrom(Room): Room
+ Door(Room, Room)
class Maze {
    Maze() { System.out.println("creating a Maze"); }

    void addRoom(Room r) {
        if (!_rooms.contains(r)) {
            _rooms.add(r);
        }
    }

    private Set _rooms = new HashSet();
}

class MapSite {
...
}

Classes Maze and MapSite

Maze

Room
Class Room (1)

class Room extends MapSite {

    Room(){
        _roomNr = _roomCnt++;
        System.out.println("creating Room #" + _roomNr);
    }

    void setSide(Direction d, MapSite site){

        if (d == Direction.North){
            _northSide = site;
        } else if (d == Direction.South){
            _southSide = site;
        } else if (d == Direction.East){
            _eastSide = site;
        } else if (d == Direction.West){
            _westSide = site;
        }
        System.out.println("setting " + d.toString() + " side of " + this.toString() + " to " + site.toString());
    }
}
MapSite getSide(Direction d){
    MapSite result = null;
    if (d == Direction.North){
        result = _northSide;
    } else if (d == Direction.South){
        result = _southSide;
    } else if (d == Direction.East){
        result = _eastSide;
    } else if (d == Direction.West){
        result = _westSide;
    }
    return result;
}

public String toString(){
    return "Room #" + new Integer(_roomNr).toString();
}
Class Room (3)

```java
private int _roomNr;
private static int _roomCnt = 1;
private MapSite _northSide;
private MapSite _southSide;
private MapSite _eastSide;
private MapSite _westSide;
```
class Wall extends MapSite {
    Wall(){
        _wallNr = _wallCnt++;
        System.out.println("creating Wall "+
                new Integer(_wallNr).toString());
    }

    public String toString(){ return "Wall "+
                new Integer(_wallNr).toString(); }

    private int _wallNr;
    private static int _wallCnt = 1;
}
Class Door

class Door extends MapSite {
    Door(Room r1, Room r2){
        _doorNr = _doorCnt++;
        System.out.println("creating a Door "+_doorNr + " between " + r1 + " and " + r2);
        _room1 = r1;
        _room2 = r2;
    }

    public String toString(){
        return "Door "+new Integer(_doorNr).toString();
    }

    private static int _doorCnt = 1;
    private int _doorNr;
    private Room _room1;
    private Room _room2;
}

class MazeGame

```java
class MazeGame {
    public Maze createMaze() {
        Maze aMaze = new Maze();
        Room r1 = new Room();
        Room r2 = new Room();
        Door theDoor = new Door(r1, r2);
        aMaze.addRoom(r1); aMaze.addRoom(r2);
        r1.setSide(Direction.North, new Wall());
        r1.setSide(Direction.East, theDoor);
        r1.setSide(Direction.South, new Wall());
        r1.setSide(Direction.West, new Wall());
        r2.setSide(Direction.North, new Wall());
        r2.setSide(Direction.East, new Wall());
        r2.setSide(Direction.South, new Wall());
        r2.setSide(Direction.West, theDoor);
        return aMaze;
    }
}
```
Driver for creating a Maze

```java
public class Main {
    public static void main(String[] args) {
        MazeGame game = new MazeGame();
        game.createMaze();
    }
}
```
Output

creating a Maze
creating Room #1
creating Room #2
creating a Door #1 between Room #1 and Room #2
creating Wall #1
setting North side of Room #1 to Wall #1
setting East side of Room #1 to Door #1
creating Wall #2
setting South side of Room #1 to Wall #2
creating Wall #3
setting West side of Room #1 to Wall #3
creating Wall #4
setting North side of Room #2 to Wall #4
creating Wall #5
setting East side of Room #2 to Wall #5
creating Wall #6
setting South side of Room #2 to Wall #6
setting West side of Room #2 to Door #1
Observations

- The code in MazeGame.createMaze() is not very flexible:
  - the layout of the maze is hard-wired
  - the types of Rooms, Doors, Walls are hard-coded;
    - there is no mechanism for adding new components such as DoorNeedingSpell - a door that can be locked and opened subsequently only with a spell
    - EnchantedRoom - a room that can have unconventional items in it, like magic keys or spells
  - currently, any change to the structure or the components of the maze requires a complete rewrite of class MazeGame
Problem and Solution

- The problem is **inflexibility**.
- Pattern can make game creation more flexible... *not* smaller!
Making the design more flexible

• Replace explicit constructor calls with dynamic dispatch; use overriding to change kinds of Rooms. **Factory Method**

• Pass object to createMaze() that knows how to create Rooms; create different kinds of Rooms by passing another object. **Abstract Factory**

• Pass object that can create a complete new Maze using operation for adding Rooms; use inheritance to change the way the maze is built. **Builder**

• Parameterize createMaze() with prototypical Room object which it copies and adds to the maze; change the maze composition by passing different prototype. **Prototype**

• The **Singleton** pattern serves to ensure there is one maze per game, in a way that all objects have easy access to it.
Abstract Factory – Motivation
Write a Cross Platform Window Toolkit

• A *GUI toolkit* that supports multiple window management standards – *WM1*, *WM2*, ….

• A window manager defines a behavior for *Widgets* – *Scroll-bars, Windows, Buttons*, …

• The *GUI interface* should handle concrete widgets – *buttons, scroll-bars,* … of *WM1*, OR-*buttons, scroll-bars,* … of *WM2*,

• How to make the *GUI interface* portable/flexible?

  → *GUI interface* should not hard code *widgets*!
Abstract Factory -- Solution

- Insert a **Widget Factory** between the client – the **GUI toolkit** and the **concrete widgets** -- **concrete products**
- The client obtains a concrete widget by calling the **factory methods**.
- The client is not aware of the identity of the widgets it holds (**WM1**, **WM2**, …).
Abstract Factory -- Solution

Use factory

Need widgets
Abstract Factory: Participants

- **AbstractFactory**
  - declares interface for operations that create abstract products
- **ConcreteFactory**
  - implements operations to create concrete products
- **AbstractProduct**
  - declares an interface for a type of product object
- **ConcreteProduct**
  - defines the product object created by concrete factory
  - implements the AbstractProduct interface
- **Client**
  - uses only interfaces of *AbstractFactory/AbstractProduct*
public class Client{
    public static void main(String[] args){
        String input = getInputFromUser();
        AbstractFactory pf = getFactory("a"); //
        AbstractProductA product = pf.createProductA();
        //more function calls on product
    }
}

if(kit.equals("a")){
    pf = new ConcreteFactory1();
}
else if(kit.equals("b")){
    pf = new ConcreteFactory2();
}
return pf;
**GUIFactory**

- `createButton() : Button`

**WinFactory**

- `createButton() : Button`

**OSXFFactory**

- `createButton() : Button`

**WinButton**

- `paint() : void`

**OSXButton**

- `paint() : void`
Abstract Factory – intent and context

- provides an interface for creating families of related or dependent objects without specifying their concrete classes

- use `AbstractFactory` when
  - a system should be independent of how its products are created, composed, represented
  - a system should be configured with one or 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 (Creational)

- Applying context:
  A client or application may want to use different types of objects with common behavior and easily change between those types before or during execution.

- Problem:
  Changing types of objects through the application is difficult because instantiation requires to know the type of object to be declared. Also different types might have different methods to use their functionality.
Abstract Factory (Creational)

• Solution:
  Isolate the client code from object creation by having client just ask for (and handle) a factory object. This factory object is an abstract data type which can only be used through an interface.

• Discussion:
  By having the client dependant just on the interface of the Abstract Type Product provided by the Abstract Factory. Each Concrete Product must be able to work under the interface that the Abstract Product provides. The client is able to change the Concrete Type Product by changing the Concrete Factory that is being used.
```java
class MazeGame {
    public Maze createMaze() {
        Maze aMaze = new Maze();
        Room r1 = new Room();
        Room r2 = new Room();
        Door theDoor = new Door(r1, r2);
        aMaze.addRoom(r1); aMaze.addRoom(r2);
        r1.setSide(Direction.North, new Wall());
        r1.setSide(Direction.East, theDoor);
        r1.setSide(Direction.South, new Wall());
        r1.setSide(Direction.West, new Wall());
        r2.setSide(Direction.North, new Wall());
        r2.setSide(Direction.East, new Wall());
        r2.setSide(Direction.South, new Wall());
        r2.setSide(Direction.West, theDoor);
        return aMaze;
    }
}
```
Maze example revisited

- create a class MazeFactory that creates Mazes, Rooms, Walls, and Doors
- then change class MazeGame to use this factory

```java
class MazeFactory {
    public Maze makeMaze() { return new Maze(); }
    public Wall makeWall() { return new Wall(); }
    public Room makeRoom() { return new Room(); }
    public Door makeDoor(Room r1, Room r2) {
        return new Door(r1, r2);
    }
}
```
class MazeGame {
    public Maze createMaze(MazeFactory factory) {
        Maze aMaze = factory.makeMaze();
        Room r1 = factory.makeRoom();
        Room r2 = factory.makeRoom();
        Door theDoor = factory.makeDoor(r1, r2);
        aMaze.addRoom(r1); aMaze.addRoom(r2);
        r1.setSide(Direction.North, factory.makeWall());
        r1.setSide(Direction.East, theDoor);
        r1.setSide(Direction.South, factory.makeWall());
        r1.setSide(Direction.West, factory.makeWall());
        r2.setSide(Direction.North, factory.makeWall());
        r2.setSide(Direction.East, factory.makeWall());
        r2.setSide(Direction.South, factory.makeWall());
        r2.setSide(Direction.West, theDoor);
        return aMaze;
    }
}
public class Main {
    public static void main(String[] args) {
        MazeFactory factory = new MazeFactory();
        MazeGame game = new MazeGame();
        game.createMaze(factory);
    }
}

public class Main {
    public static void main(String[] args) {
        MazeGame game = new MazeGame();
        game.createMaze();
    }
}
Adding new Products is now easy after adapting MazeGame to use a factory, it is easy to create mazes with different components:

```java
class EnchantedRoom extends Room {
    EnchantedRoom(Spell s) { super(); ... }
    public String toString() {
        return "enchanted " + super.toString();
    }
}

class DoorNeedingSpell extends Door {
    DoorNeedingSpell(Room r1, Room r2) { super(r1,r2); .. }
    public String toString() {
        return super.toString() + " (needing spell)";
    }
}
```
New subclass of MazeFactory

```java
class EnchantedMazeFactory extends MazeFactory {

    public Room makeRoom() {
        return new EnchantedRoom(castSpell());
    }

    public Door makeDoor(Room r1, Room r2) {
        return new DoorNeedingSpell(r1, r2);
    }

    protected static Spell castSpell() {
        return new Spell();
    }
}
```
New Driver

- you can now build Enchanted Mazes by using an EnchantedFactory instead of the regular MazeFactory

```java
public class Maze {
    public static void main(String[] args){
        MazeFactory factory = new EnchantedMazeFactory();
        MazeGame game = new MazeGame();
        game.createMaze(factory);
    }
}
```
MazeGame example: observations

- the *MazeGame* example encodes a somewhat simplified form of the pattern:
  - *MazeFactory* is **not an abstract class**
  - *Room, Wall, Door* are **not abstract** either
  - *EnchantedMazeFactory* **only** overrides **some** of the methods in *MazeFactory*

- in general:
  - **downcasting** may be needed when you want to access methods/fields in *ConcreteProducts*
    - **If EnchantedRoom** had to access a subclass-specific member of *EnchantedWall* then it would have to cast a reference to its walls from *Wall* to *EnchantedWall* downcast is required.

- useful for situations where you create many instances of the same product, but where you want to be able to vary the product
- often used together with the **Singleton pattern (concrete factory)**
Abstract factory: observations

**Advantages:**
- Isolates concrete classes.
- Exchange in product families is easy:
  - A concrete factory is a singleton – created once in an application.
  - Changing a family of products = changing the factory instance.
- Promotes consistency among products.

**Disadvantages:**
- Supporting a new **kind** of products (e.g., new widget) is difficult – requires extending the interface.
- Client does not know the kind of product that is produced. Might require **downcasting**.
Abstract factory: Implementation

- Factories are *singletons*.

- How *concrete products* are created?
  - Most common – use a *factory method* for each product.
  - Or – use the *Prototype pattern*.

- Defining extensible factories:
  - Use parameterized create – product parameter.
  - This parameter specifies the kind of object to be created.
    - a class identifier, an integer, a string, or anything else that identifies the kind of product
  - A single “make” in a factory with a parameter indicating the kind of object to create
Factory Method
Factory Method -- Motivation

- A framework for applications that can present/maintain multiple documents to the user.

- Key abstractions:
  - Documents hierarchy:
    - A Drawing document is a Document.
  - Applications hierarchy:
    - An art application is an Application.

- An application is responsible for managing documents (create, open, hold, ...)

- The abstract application cannot predict the kind of document to create.
  - Knows when a new document should be created.
  - Knows what to do with a new document.
  - Does not know which kind of document to create.
Factory Method - Solution

**Dilemma:** Framework must instantiate subclasses that it does not recognize!

- **Encapsulate** knowledge of concrete *Document* subclasses.
- Move this knowledge out of the framework.
- Application should have concrete operations for using documents.
- Creation knowledge is **deferred** to subclasses of Application.
Factory Method -- Solution

public Document CreateDocument(String type) {
    if (type.isEqual("html")) return new HtmlDocument();
    if (type.isEqual("proprietary")) return new MyDocument();
    if (type.isEqual("pdf")) return new PdfDocument();
    return null;
}

An alternative Implementation using parameters

public class Client {
    public static void main(String arg[]) {
        Creator creator = new ConcreteCreator();
        creator.anOperation();
    }
}
Factory Method: Participants

- **Product**
  - defines the *interface* of objects created by the factory method

- **ConcreteProduct**
  - implements the Product interface

- **Creator**
  - declares the factory method, which returns a *Product*
  - may define default implementation that returns a default *ConcreteProduct* object
  - may call factory method to create a *Product*

- **ConcreteCreator**
  - overrides the *factory method* to return a *concreteProduct*
Factory Method – intent and context

- define an interface for creating an object, but let subclasses decide which class to instantiate
- Factory Method lets you create objects in a separate operation so that they can be overridden by subclasses
- use *Factory Method* 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.
Maze example revisited

- recall that existing Maze example hard-codes Maze, Room, Wall, Door classes

- alternative approach:
  - define factory methods in MazeGame for creating Maze/Room/Wall/Door objects
  - update MazeGame.createMaze() to use factory methods

- benefit:
  - allows one to create specialized versions of the game by creating subclasses of MazeGame
  - override some or all of MazeGame’s factory methods
MazeGame using factory methods

class MazeGame {
    // factory methods that create the products
    
    public Maze makeMaze(){ return new Maze(); }
    public Room makeRoom(){ return new Room(); }
    public Wall makeWall(){ return new Wall(); }
    public Door makeDoor(Room r1, Room r2){
        return new Door(r1, r2);
    }
    
    // create a maze by calling the factory methods
    public Maze createMaze(){
        Maze aMaze = makeMaze();
        Room r1 = makeRoom();
        Room r2 = makeRoom();
        Door theDoor = makeDoor(r1,r2);
    }
}
MazeGame using factory methods (2)

...
Creating specialized mazes

// classes EnchantedRoom and DoorNeedingSpell as before

class EnchantedMazeGame extends MazeGame {
    public Room makeRoom() {
        return new EnchantedRoom(castSpell());
    }
    public Door makeDoor(Room r1, Room r2) {
        return new DoorNeedingSpell(r1, r2);
    }
    private Spell castSpell() { return new Spell(); }
}
Updated driver

```java
public class Main {
    public static void main(String[] args) {
        MazeGame game = new EnchantedMazeGame();
        Maze maze = game.createMaze();
    }
}
```
Factory Method vs. Abstract Factory

- Abstract factories are often implemented using factory methods
  - class `AbstractFactory` contains the `FactoryMethods` that are overridden in class `ConcreteFactory`
  - factory is passed to Client as a parameter
  - Client invokes factory methods on this parameter

Note: `AbstractFactory` can also be implemented using `Prototype` (one of the 5 creational patterns)
Factory Method vs. Abstract Factory

- **AbstractFactory**
  - CreateProductA()
  - CreateProductB()

- **AbstractProductA**
  - ProductA1
  - ProductA2

- **AbstractProductB**
  - ProductB1
  - ProductB2

- **ConcreteFactory1**
  - CreateProductA()
  - CreateProductB()

- **ConcreteFactory2**
  - CreateProductA()
  - CreateProductB()

- **Client**

*This is abstract factory i.e. an interface to create family of related objects.*

*These two are factory methods*
Factory Method vs. Abstract Factory

- **Factory Method**
  - The client expects an implementation of an interface or abstract class, but doesn't know exactly what concrete class the factory will return.

- **Abstract Factory**
  - Here, there is one more level of abstraction.
    - The client does not even know what factory it's going to use.
    - First, it gets a Factory and then it calls a Factory method.
Factory Method
Factory Method: Observations

- **Advantages:**
  - Client code is free from application specific classes.
  - Provides hooks for further subclassing or versioning.

- **Disadvantage:**
  - Clients must subclass the creator just to create a concrete product object.
Factory Method:
-- for parallel class hierarchies

- Occurs when a class delegates some of its responsibilities to a separate class.
- Consider graphical figures that can be manipulated interactively; that is, they can be stretched, moved, or rotated using the mouse.
- Implementing such interactions isn't always easy.
- It often requires storing and updating information that records the state of the manipulation at a given time.
- This state is needed only during manipulation; therefore it needn't be kept in the figure object.
- Different figures behave differently when the user manipulates them.
  - For example, stretching a line figure might have the effect of moving an endpoint, whereas stretching a text figure may change its line spacing.
- In this case, it's better to use a separate Manipulator object that implements the interaction and keeps track of any manipulation-specific state that's needed.
Factory Method:
-- for parallel class hierarchies

- Different figures will use different Manipulator subclasses to handle particular interactions.
- If Figure class may implement CreateManipulator to return a default Manipulator instance, the Figure subclasses may simply inherit that default.
- The Figure classes that do so need no corresponding Manipulator subclass—hence the hierarchies are only partially parallel.

Provides a CreateManipulator factory method that lets clients create a Figure's corresponding Manipulator.

override this method to return an instance of the Manipulator subclass that's right for them.
Factory Method: Implementation

- The *Creator* class can be:
  - **fully abstract**.
  - **Concrete** – provide a *default implementation* for the factory method.

- **Parameterized factory methods**: the factory method can create *multiple kinds* of products.
It's important for some classes to have exactly one instance.

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.

An accounting system will be dedicated to serving one company.
## Singleton

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is it?</td>
<td>The singleton pattern ensures that only one object of a given type exists in the application.</td>
</tr>
<tr>
<td>What are the benefits?</td>
<td>The singleton pattern can be used to manage objects that represent real-world resources or to encapsulate a shared resource.</td>
</tr>
<tr>
<td>When should you use this pattern?</td>
<td>The singleton pattern should be used when creating further objects doesn’t increase the number of real-world resources available or when you want to consolidate an activity such as logging.</td>
</tr>
<tr>
<td>When should you avoid this pattern?</td>
<td>The singleton pattern isn’t useful if there are not multiple components that require access to a shared resource or if there are no objects that represent real-world resources in the application.</td>
</tr>
<tr>
<td>How do you know when you have implemented the pattern correctly?</td>
<td>The pattern has been correctly implemented when there is only one instance of a given type <em>and</em> when that instance cannot be copied and cloned <em>and</em> when further instances cannot be created.</td>
</tr>
</tbody>
</table>
Singleton

[Diagram showing a Singleton pattern with three calling components pointing to a Singleton object]
Singleton – motivation, intent, context

- **Singleton** ensures that:
  - a class has only **one instance**
  - this instance is **globally accessible**

- considerations:
  - use **Singleton** for classes that should have only one instance (e.g., Scheduler, Print Spooler, etc.)
  - lets you avoid parameter-passing of the singleton object
Singleton: Participants

- **Singleton**
  - defines an operation that lets clients access its unique instance.
  - This operation is static.
    - may be responsible for creating its own unique instance
Singleton: Class Diagram

```java
if (uniqueInstance == null){
    uniqueInstance = new Singleton();
}
return uniqueInstance;
```
Example: Apply Singleton to MazeFactory (AbstractFactory)

class MazeFactory {

    // constructor is PRIVATE so it cannot be called from //outside the class

private MazeFactory(){ }

    // method for returning the unique instance of MazeFactory
public static MazeFactory instance(){
    if (_theFactory == null){
        _theFactory = new MazeFactory();
    }
    return _theFactory;
}

    // private static field to store the unique instance
private static MazeFactory _theFactory = null;

Example: Apply Singleton to MazeFactory (AbstractFactory)

```java
public Maze makeMaze(){ return new Maze(); }
public Wall makeWall(){ return new Wall(); }
public Room makeRoom(){ return new Room(); }
public Door makeDoor(Room r1, Room r2){
    return new Door(r1,r2);
}
```
public Maze createMaze() {
    MazeFactory factory = MazeFactory.instance();
    Maze aMaze = factory.makeMaze();
    Room r1 = factory.makeRoom();
    Room r2 = factory.makeRoom();
    Door theDoor = factory.makeDoor(r1, r2);
    aMaze.addRoom(r1); aMaze.addRoom(r2);
    r1.setSide(Direction.North, factory.makeWall());
    r1.setSide(Direction.East, theDoor);
    r1.setSide(Direction.South, factory.makeWall());
    ...
    r2.setSide(Direction.South, factory.makeWall());
    r2.setSide(Direction.West, theDoor);
    System.out.println("Done.");
    return aMaze;
}

public Maze createMaze(MazeFactory factory) {...}
Singleton: Considerations

• There is no good solution for allowing Singletons to be subclassed
  • make the constructor protected instead of private
  • but you cannot override the static instance() method

• possible solution:
  • let instance() method read information from an environment variable, specifying what kind of MazeFactory it should build
  • requires rewriting the instance() method every time a subclass is added.

• in Java, an obvious solution would be to give instance() a String-typed parameter with the name of the factory, and to use reflection to create an object

  String className = // fully qualified class name
  Class c = Class.forName(className);
  Object theNewObject = c.newInstance();
Singleton: Discussion

  - Singletons frequently are used to provide a global access point for some service.
  - Create something as a global to avoid passing it around is a smell in your design
  - Singletons allow you to limit creation of your objects.
    - Mixing two different responsibilities into the same class.
    - A class should not care whether or not it is a singleton
  - “Almost every use of singleton I have encountered was best replaced by an attribute accessor in a higher level object ..”
    - that can be e.g., explicitly passed around via a parameter
Singleton: Discussion

- I see singletons as the cause of a dichotomy within an object model. All of a sudden there are two types of objects:
  - those that can be instantiated in a standard fashion and
  - those that cannot be created at all.

I would personally rather use a container which governs the number of a given object that can exist in a system and acquire the objects from the container. -- JohnHarby

- I've never used a singleton to make sure there was only one of something. Singletons usually are used to provide a single point of access to a global service.
  - I always make the singleton separate from the class itself so the class can be used any way you want.
  - The singleton can then use the class. The singleton also doesn't have to instantiate the object. It just has to provide access to the object.
  - The object returned can best be set by any means necessary. That's more like the different, but related, FactoryPattern.
When it really is a singleton

(J.B. Rainsberger jbr@diasparsoftware.com)

To decide whether a class is truly a singleton, you must ask yourself some questions.

Will every application use this class **exactly** the same way? (*exactly* is the key word)

Will every application **ever** need only **one** instance of this class? (*ever* and *one* are the key words)

Should the clients of this class be **unaware of the application** they are part of?

If you answer yes to all three questions, then you've found a singleton.

The key points here are that a class is only a singleton if all applications treat it exactly the same and if its clients can use the class **without an application context.**
A classic example of a true singleton is a logging service. Suppose we have an event-based logging service: Client objects request that text be logged by sending a message to the logging service. Other objects actually log the text somewhere (console, file, whatever) by listening to the logging service for these logging requests and handling them. First, notice that the logging service passes the classic test for being a singleton:

- The requesters need a well-known object to which to send requests to log. This means a global point of access.
- Since the logging service is a single event source to which multiple listeners can register, there only needs to be one instance.
- The classic singleton design pattern requirements are met.
The GoF says that the Builder design patterns let you “Separate the construction of a complex object from its representation so that the same construction processes can create different representations.”
Builder: Motivation

• A reader for the 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:
  The number of possible conversions is open-ended. It should be easy to add a new conversion without modifying the reader.
Builder: Solution

- Configure the *RTFReader* class with a *TextConverter* object that converts RTF to another textual representation.
  - The *RTFReader* parses the RTF document,
  - When it recognizes an RTF token \( t \) calls a *TextConverter* on \( t \).

- *TextConverter* responsibilities:
  - *perform* data conversion.
  - *represent* the token in a particular format.
    → *Create* and assemble a complex object.
    → *Hide* this process.

- Subclasses of *TextConverter* specialize in different conversions and formats.
Builder: Solution

while (t = get the next token) {
    switch t.Type {
    CHAR:
        builder->ConvertCharacter(t.Char)
    FONT:
        builder->ConvertFontChange(t.Font)
    PARA:
        builder->ConvertParagraph()
    }
}
Builder: Participants

- **Builder**
  - An 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.
Builder: Class Diagram

```
for all objects in structure {
    builder->BuildPart()
}
```
Builder: Sequence Diagram – interaction with a client:
Builder: intent and context

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

- Use Builder when:
  - 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 constructed object.
Maze example revisited

- define a variant of the `createMaze()` method that takes a `MazeBuilder` object as an argument
  - method for creating a Maze
  - method for creating a Room
  - method for creating a Door between two Rooms

```java
interface MazeBuilder {
    public void buildMaze();
    public Room buildRoom();
    public void buildDoor(Room from, Direction side1, Room to, Direction side2);
    public Maze getMaze();
}
```
Revised method createMaze()

class MazeGame {
    public Maze createMaze() {
        Maze aMaze = new Maze();
        Room r1 = new Room();
        Room r2 = new Room();
        Door theDoor = new Door(r1, r2);
        aMaze.addRoom(r1); aMaze.addRoom(r2);
        r1.setSide(Direction.North, new Wall());
        r1.setSide(Direction.East, theDoor);
        r1.setSide(Direction.South, new Wall());
        r1.setSide(Direction.West, new Wall());
        r2.setSide(Direction.North, new Wall());
        r2.setSide(Direction.East, new Wall());
        r2.setSide(Direction.South, new Wall());
        r2.setSide(Direction.West, theDoor);
        return aMaze;
    }
}

OLD VERSION
Revised method createMaze()

- Observe that:
  - all details about the representation of a Maze are now hidden
  - all details about how Mazes are assembled from Rooms, Doors, Walls are hidden as well

```java
class MazeGame {
    public static Maze createMaze(MazeBuilder builder) {
        builder.buildMaze();
        Room r1 = builder.buildRoom();
        Room r2 = builder.buildRoom();
        builder.buildDoor(r1, Direction.North, r2, Direction.South);
        return builder.getMaze();
    }
}
```
Revised method createMaze()

- Observe that:
  - all details about the representation of a Maze are now hidden
  - all details about how Mazes are assembled from Rooms, Doors, Walls are hidden as well

```java
class MazeGame {
    public static Maze createMaze(MazeBuilder builder) {
        builder.buildMaze();
        Room r1 = builder.buildRoom();
        Room r2 = builder.buildRoom();
        builder.buildDoor(r1, Direction.North, r2, Direction.South);
        return builder.getMaze();
    }
}
```
class StandardMazeBuilder implements MazeBuilder {
    public void buildMaze() {
        _currentMaze = new Maze();
    }

    public Room buildRoom() {
        Room r = new Room();
        _currentMaze.addRoom(r);
        r.setSide(Direction.North, new Wall());
        r.setSide(Direction.South, new Wall());
        r.setSide(Direction.East, new Wall());
        r.setSide(Direction.West, new Wall());
        return r;
    }
} ...
public void buildDoor(Room r1, Direction side1, Room r2, Direction side2) {
    Door d = new Door(r1, r2);
    r1.setSide(side1, d);
    r2.setSide(side2, d);
}

public Maze getMaze() {
    return _currentMaze;
}

private Maze _currentMaze;
public class Main {
    public static void main(String[] args) {
        MazeBuilder builder = new StandardMazeBuilder();
        MazeGame game = new MazeGame();
        Maze maze = game.createMaze(builder);
    }
}

class MazeGame {
    public Maze createMaze(MazeBuilder builder) {
        builder.buildMaze();
        Room r1 = builder.buildRoom();
        Room r2 = builder.buildRoom();
        builder.buildDoor(r1, Direction.North, r2, Direction.South);
        return builder.getMaze();
    }
}
Builder: Distribution of responsibility

- Client only knows the Director (the `createMaze()` method) and the ConcreteBuilder (StandardMazeBuilder) s/he wants to use
  - no details about how to construct Products
  - no details of Product representation
- The ConcreteBuilder (StandardMazeBuilder) creates the actual Products (Rooms, Doors, Walls) and determines their representation
- the Director method (`createMaze`) directs the ConcreteBuilder to build and assemble the Product parts (i.e., decides when and in which order to build these parts).
Builder: Observations

• **Advantages:**
  • Isolates construction from assembly.
  • Builders hide the assembly.
  • Addition of a new assembly method – simple: new builder.
  • Clients have no knowledge of parts and assemblies.
  • Director receives only the final product.
Builder: Implementation

- *Abstract builder* provides operations (possibly default) for parts *construction*.
- Only subclasses of *builder* *construct* (*assemble*).
- *Assemble operation* – may vary: “append” or “combine rooms by a door”.
- Why no abstract class for *products*? – because they are different (many combinations)
- Who knows about the products? *Builder* and *client*, which gives the *concrete builder* to the *director*.
Builder vs Abstract Factory

- **Similar!**
  - Both manipulate complex objects, different focus:
    - Builder: Construct, step by step.
    - Abstract factory: Families of products

- **Difference:**
  - Abstract Factory: the client uses the factories methods to create its own objects. No explicit construction (possibly by client).
  - Builder: the builder class is instructed on how to create the object and then it is asked for it, but the way that the class is put together is up to the Builder class.
Factory Method

https://www.youtube.com/watch?v=V_Oy7UphovM
Abstract Factory

https://www.youtube.com/watch?v=V_Oy7UphovM
Builder

https://www.youtube.com/watch?v=V_Oy7UphovM
Prototype
The Prototype design pattern is used for creating new objects (instances) by cloning (copying) other objects and this way we can improve the performance. This pattern uses Java cloning to copy the object.

This pattern is used when creation of objects is costly or complex. For example, if an object is to be created after a costly database operation. We can cache the object, return its clone on next request. Once we get the cloned object we can modify according to our needs.
Prototype – Motivation

- Build an editor for music scores.
- Approach: **Customize** a general framework for graphical editors and add new *graphical objects* that represent notes, rests, and staves.
- The editor framework may have a palette of *tools for manipulation* of music objects: selecting, moving, rotating, manipulating.
  - One of these tools specializes in adding music shapes to a score.
- The *tools* are **common** to any graphical editor.
- The *graphic shapes* are **specific** to the music scores editor.
Prototype – Solution

- **Abstract classes:**
  - *Tool* -- For graphic manipulation tools. It belongs to the framework.
  - *Graphics* – For graphic shapes like notes.

- **Concrete tool** (subclass) – *GraphicTool*: creates instances of graphical objects and adds them to the document.

- **Problem:** *GraphicTool* doesn't know how to create instances of music classes and to add to the score.

- **Solution** (bad):
  - Subclass *GraphicTool* for each kind of music object.
    - Produce lots of subclasses that differ only in the kind of music object they instantiate.
    - Produces many similar classes.
  - *GraphicTool* creates a new *Graphic* by copying or "cloning" an existing instance – the prototype – of a Graphic subclass.
Prototype – Solution

```
Prototype – Solution

Tool
Manipulate()

RotateTool
Manipulate()

GraphicTool
Manipulate()

Graphic
Draw(Position)
Clone()

Staff
Draw(Position)
Clone()

MusicalNote

WholeNote
Draw(Position)
Clone()

HalfNote
Draw(Position)
Clone()

p = prototype->Clone()
while (user drags mouse) {
    p->Draw(new position)
} insert p into drawing

return copy of self

return copy of self
```
Prototype: Participants

- **Prototype**
  - declares an interface for cloning itself
- **ConcretePrototype**
  - implements an interface for cloning itself
- **Client**
  - creates a new object by asking a prototype to clone itself
Prototype: Class Diagram

Client
+someOperation()

Prototype
+clone()

ConcretePrototype1
+clone()
+initialize()

ConcretePrototype2
+clone()

p = prototype.clone();

return copy of self;
Prototype – intent and context

- specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype
- use Prototype when
  - a system should be independent of how its products are created/composed/represented
  - one of the following conditions holds:
    - the classes to instantiate are specified at run-time
    - to avoid building a class hierarchy of factories that parallels the class hierarchy of products
    - instances of a class have only a few different combinations of state
Benefits of Prototype

• similar to *Abstract Factory* and *Builder*:
  • hide concrete product classes from the client
  • let client work with application-specific classes without modification

• additional benefits
  • allows for addition of products at run-time
    • especially important for applications that rely on dynamic loading to add classes after start of execution
  • reduced need for subclassing
Yet another version of “Maze”

- we will create a new subclass of class `MazeFactory` called `MazePrototypeFactory`
  - initialized by giving it a prototype `Wall, Door, Room, Maze`
  - `MazePrototypeFactory` stores these prototypes in private fields
  - whenever a new component is created, it calls `clone()` on the appropriate prototype
  - `initialize()` method need for class `Door`, to reset the `Rooms` connected by the prototype `Door`
class MazePrototypeFactory extends MazeFactory {

    MazePrototypeFactory(Maze m, Wall w, Room r, Door d) {
        _prototypeMaze = m;
        _prototypeWall = w;
        _prototypeRoom = r;
        _prototypeDoor = d;
    }

    public Maze makeMaze() {
        return (Maze)_prototypeMaze.clone();
    }

    public Room makeRoom() {
        return (Room)_prototypeRoom.clone();
    }
}
Class MazePrototypeFactory (2)

... 

public Wall makeWall()
{
    return (Wall)_prototypeWall.clone();
}

public Door makeDoor(Room r1, Room r2)
{
    Door door = (Door)_prototypeDoor.clone();
    door.initialize(r1,r2);
    return door;
}

private Maze _prototypeMaze;
private Wall _prototypeWall;
private Room _prototypeRoom;
private Door _prototypeDoor;
}
Maze with clone() method

class Maze {
    Maze(){ System.out.println("creating a Maze"); }  
    void addRoom(Room r){
        if (!_rooms.contains(r)){
            _rooms.add(r);
        }
    }
    protected Object clone() {
        if (!_rooms.isEmpty()){
            throw new Error("cloning of non-empty mazes not supported.");
        }
        Maze maze = new Maze();  
        maze._rooms = new HashSet();
        return maze;
    }
    private Set _rooms = new HashSet();
}
Door with clone() and initialize() methods

```java
class Door extends MapSite {
    Door(Room r1, Room r2) {
        _doorNr = _doorCnt++; _room1 = r1; _room2 = r2;
    }
    ...
    public Object clone() {
        Door door = new Door(_room1, _room2);
        return door;
    }
    public void initialize(Room r1, Room r2) {
        _room1 = r1; _room2 = r2;
        System.out.println("initializing Door \\
        \#" + _doorNr + " between " + r1 + " and " + r2);
    }
    ...
}
```
public class Main {
    public static void main(String[] args) {
        MazeGame game = new MazeGame();

        // create the prototypes
        Maze mazeProto = new Maze();
        Wall wallProto = new Wall();
        Room roomProto = new Room();
        Door doorProto = new Door(roomProto, roomProto);
        MazeFactory factory =
            new MazePrototypeFactory(mazeProto, wallProto,
                                       roomProto, doorProto);
        game.createMaze(factory);
    }
}
Creating specialized mazes

```java
public class Main {
    public static void main(String[] args) {
        MazeGame game = new MazeGame();

        // select different prototypes to change maze type
        Maze mazeProto = new Maze();
        Wall wallProto = new Wall();
        Room roomProto2 = new EnchantedRoom(new Spell());
        Door doorProto2 =
            new DoorNeedingSpell(roomProto2, roomProto2);
        MazeFactory factory =
            new MazePrototypeFactory(mazeProto, wallProto, roomProto2, doorProto2);
        game.createMaze(factory);
    }
}
```
Prototype: Implementation

- **Use prototype manager** – if number of prototypes is constantly changing.
  - A manager can store and retrieve prototypes using a key.
  - Clients contact the prototype manager.

- **Implementing clone():** Circular references within an object structure present a problem.
  - OO languages include a copy constructor – but with a shallow copying: The clone and the original share the reference variables.

- **Prototype operations** – clients might call operations right after the clone (like “initialize()” in Door).
Creational Patterns: Summary

- purpose: to make designs more flexible and extensible by instantiating classes in certain stylized ways
  - AbstractFactory
  - FactoryMethod
  - Singleton
  - Builder
  - Prototype
Creational Maze: Summary

- The creational patterns as implemented in the Maze example are illustrated in the following slides:
  - Maze with AbstractFactory
  - Maze with FactoryMethod
  - Singleton
  - Maze with Builder
  - Maze with Prototype
Maze Startup

1. create()
2. createMaze()
Maze with Abstract factory
Maze with Abstract Factory startup

1. create ()
   1.1 create ()
   1.2 createMaze (f)
Maze with Factory method

Maze
- addRoom(Room r)

Room
- roomCount : int
- roomNumber : int
- enter()

EnchantedRoom
- enchantedRoom()

0..* Room

Maze
- <<abstract>>
  - MapSite
    - enter()

The method createMaze() includes all assembly and therefore is complex

MazeGame
- makeMaze() : Maze
- makeRoom() : Room
- makeDoor(Room r1, Room r2) : Door
- makeWall() : Wall
- createMaze() : Maze

Separator
- +north
- +south
- +east
- +west

EnchantedMazeGame
- makeRoom() : Room
- makeDoor() : Door

Wall
- wallCount : int
- wallNumber : int
- enter()

Door
- doorCount : int
- doorNumber : int
- enter()

DoorNeedingSpell
Maze with Factory Method startup

```
1. create()
2. createMaze()
```
Maze with Builder

- Maze
  - addRoom(Room r)
- MazeGame
  - createMaze(MazeBuilder m) : Maze
- MazeBuilder
  - buildMaze() : Maze
  - buildRoom() : Room
  - buildDoor(Room r1, Direction d1, Room r2, Direction d2)
- StandardMazeBuilder
  - buildMaze() : Maze
  - buildRoom() : Room
  - buildDoor(Room r1, Direction d1, Room r2, Direction d2)
- Room
  - roomCount : int
  - roomNumber : int
  - enter()
- Separator
  - west
  - north
  - south
  - east
- Door
  - doorCount : int
  - doorNumber : int
  - enter()
- Wall
  - wallCount : int
  - wallNumber : int
  - enter()
Maze with Builder Startup

1. create()
2. create()
3. createMaze (b)
Maze with Prototype
Maze with Prototype Startup

1. create()
2. create()
3. create()
4. create(wp,wp)
5. create(mp,wp,rp,dp)
6. createMaze(f)