DESIGN PATTERNS

F. Tip and
M. Weintraub

Thanks go to Andreas Zeller for allowing incorporation of his materials

Northeastern University
College of Computer and Information Science
440 Huntington Avenue . 202 West Village H . Boston, MA 02115 . T 617.373.2462 . ccis.northeastern.edu

HISTORICAL PERSPECTIVE

- the term “design patterns” in Software Engineering was inspired by reusable elements of design (“patterns”) in the field of architecture

  - 1977 book “A Pattern Language: Towns, Buildings, Construction” by Christopher Alexander et al.

  - presents 253 patterns, covering advice on use of materials, physical arrangements of architectural elements, etc.

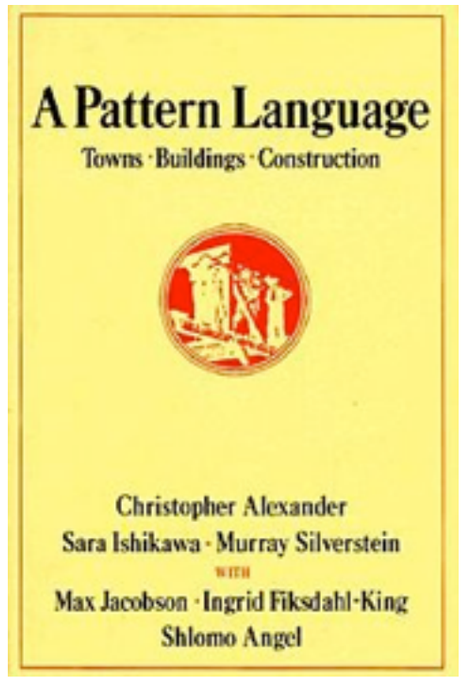
- Examples:

  173. GARDEN WALL

  174. TRELLISED WALK

  159. LIGHT ON TWO SIDES OF EVERY ROOM

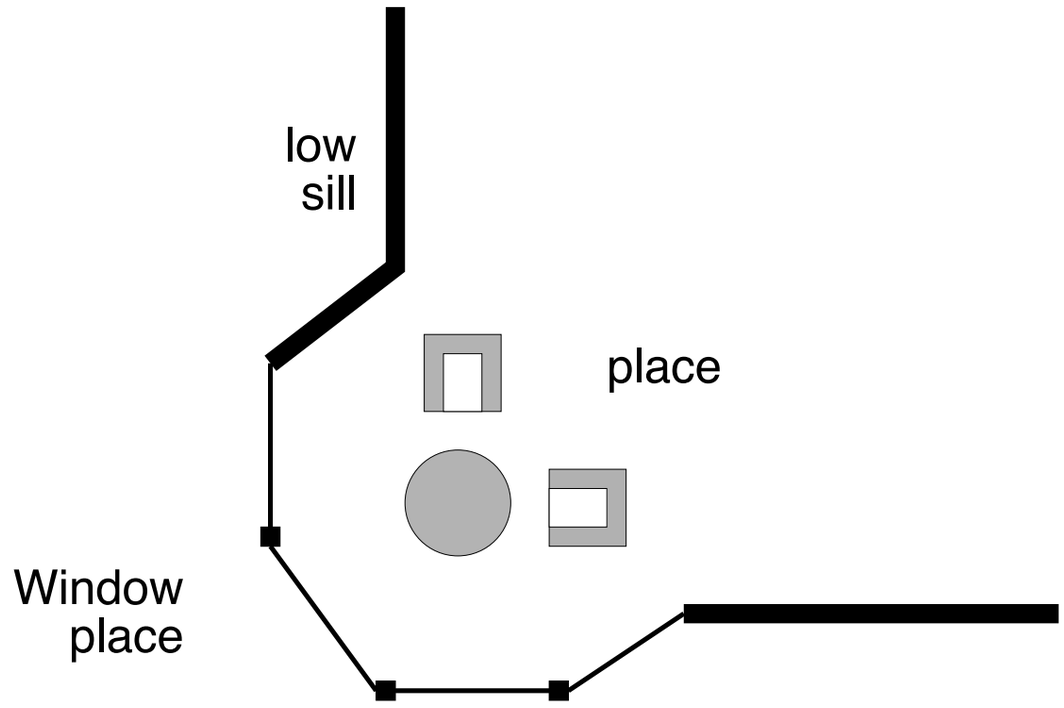
  180. WINDOW PLACE



180. WINDOW PLACE

Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them

In every room where you spend any length of time during the day, make at least one window into a “window place”



WHAT IS THE DIFFERENCE BETWEEN EXPERIENCED AND INEXPERIENCED SOFTWARE DESIGNERS?

- Experienced designers know from experience what works and what doesn't
- Often recognize "standard" design problems and apply "proven" solutions to them

PATTERNS IN SOFTWARE DESIGN

- The “Gang of Four” (Gamma, Helm, Johnson, Vlissides) catalogued a number of widely used patterns in software design

REUSING EXPERIENCE: DESIGN PATTERNS

design patterns = descriptions of communicating objects and classes, that have been adapted to solve a design problem in a specific context.

In other words: a design pattern is a generalized and reusable solution to a similar set of problems. Design patterns are abstract and *must be tailored or adapted to each situation*.

CLASS VS. INTERFACE INHERITANCE

- **class inheritance** defines an object's implementation in terms of another object's implementation
  - extend an application's functionality by reusing functionality in parent classes (code and representation sharing)
  - lets you get new implementations almost for free, inheriting most of what you need from existing classes

- **interface inheritance** describes when an object can be used in place of another
  - clients remain unaware of specific types of objects → greatly reduces dependencies between subsystems
  - reduces the impact of changes

MECHANISMS FOR REUSING FUNCTIONALITY

class inheritance: define implementation of one class in terms of another

- often referred to as white-box reuse: internals of parent class visible to extending class

"class inheritance breaks encapsulation"

object composition: compose objects to get new, more complex functionality

- implemented by giving objects references to other objects; access these objects via interfaces
- requires that objects have well-defined interfaces
- often called black-box reuse: no internal details of objects are visible to the class that uses them

"composition does not break encapsulation"

CLASS INHERITANCE: PROS & CONS

Advantages

1. directly supported by the programming language; easy to use
2. easy to modify the reused implementation (by overriding a few methods)

Disadvantages

1. cannot change inherited functionality at run-time (inheritance is fixed at compile-time)
2. parent classes define at least part of their subclasses' physical representation, and subclasses are exposed to details of their parent's implementation
3. implementation of subclass becomes very tightly coupled with implementation of parent
4. change in parent is likely to require changes in subclass

PRINCIPLES OF OBJECT-ORIENTED DESIGN

Program to an interface, not an implementation

Favor object composition over class inheritance

CLASS INHERITANCE

```
class C {
    void f(){
        /* do something */
    }
}
class D extends C {
    void f(){
        super.f();
        /* do other stuff */
    }
}
```

- **delegation** is an alternative to inheritance:
- two objects are involved:
  - a **receiving object** **delegates** an operation to its **delegate**
  - analogy: a **subclass** that defers a request to its **parent class**

DELEGATION

```
class C {
    C(){
        d = new D();
    }

    void f(){
        d.f();
    }

    private D d;
}
```

forwarding method

```
class D {
    public void f() {
      /* do the real work */
    }
}
```

- **delegation** is an alternative to inheritance:
- two objects are involved:
  - a **receiving object delegates** an operation to its **delegate**
  - analogy: a **subclass** that defers a request to its **parent class**

DELEGATION

```
class C {
    C(){
        d = new D();
    }

    void f(){
        d.f();
    }

    private D d;
}
```

forwarding method

```
class D {
    public void f() {
      /* do the real work */
    }
}
```

This design breaks encapsulation: C depends on D's implementation!

DELEGATION

```
class C {
    C(){
        i = new D();
    }

    void f(){
        i.f();
    }

    private I i;
}
```

```
interface I {
    public void f();
}

class D implements I {
    public void f() {
      /* do the real work */
    }
}
```

The use of an interface removes C's dependency on D's implementation details!

DELEGATION

```
class C {
    C(){
        i = new D();
    }

    void f(){
        i.f();
    }

    void update(I i){
        this.i = i;
    }

    private I i;
}
```

```
interface I {
    public void f();
}

class D implements I {
    public void f() {
      /* do the real work */
    }
}
```

DELEGATION EXAMPLE

```
import java.util.Vector;

public class Stack<T> {

    public Stack(){
        this.v = new Vector<T>();
    }

    public void push(T t){
        v.add(t);
    }

    public T pop(){
        // throw exception if empty
        return v.remove(v.size()-1);
    }

    private Vector<T> v;
}
```

ABUSING INHERITANCE FOR THE SAME PURPOSE

```
import java.util.Vector;

public class BadStack<T> extends Vector<T> {

    public BadStack(){
      // no need to create a Vector..
    }

    public void push(T t){
        add(t);
    }

    public T pop(){
        // throw exception if empty
        return remove(size()-1);
    }

}
```

ABUSING INHERITANCE FOR THE SAME PURPOSE

```
import java.util.Vector;

public class BadStack<T> extends Vector<T> {

    public BadStack(){
      // no need to create a Vector..
    }

    public void push(T t){
        add(t);
    }

    public T pop(){
        // throw exception if empty
        return remove(size()-1);
    }

}
```

Class Vector<E> offers two methods that come along via inheritance:

E remove(int index)
Removes the element at the specified position in this Vector

Void removeElementAt(int index)
Deletes the component at the specified index.

Now your stack offers one the chance to violate the semantics of the stack…

THE DEVELOPERS OF THE JAVA LIBRARIES ACTUALLY MADE THIS MISTAKE..

```
package java.util;

/**
 * The <code>Stack</code> class represents a last-in-first-out
 * (LIFO) stack of objects. It extends class <tt>Vector</tt> with five
 * operations that allow a vector to be treated as a stack. The usual
 * <tt>push</tt> and <tt>pop</tt> operations are provided, as well as a
 * method to <tt>peek</tt> at the top item on the stack, a method to test
 * for whether the stack is <tt>empty</tt>, and a method to <tt>search</tt>
 * the stack for an item and discover how far it is from the top.
 * <p>
 * When a stack is first created, it contains no items.
 *
 * <p>A more complete and consistent set of LIFO stack operations is
 * provided by the {@link Deque} interface and its implementations, which
 * should be used in preference to this class.  For example:
 * <pre>   {@code
 *   Deque<Integer> stack = new ArrayDeque<Integer>();}</pre>
 *
 * @author  Jonathan Payne
 * @since   JDK1.0
 */
public
class Stack<E> extends Vector<E> {
    /**
     * Creates an empty Stack.
     */
    public Stack() {
    }

    /**
     * Pushes an item onto the top of this stack. This has exactly
     * the same effect as:
     * <blockquote><pre>
     * addElement(item)</pre></blockquote>
     *
     * @param   item   the item to be pushed onto this stack.
     * @return  the <code>item</code> argument.
     * @see     java.util.Vector#addElement
     */
    public E push(E item) {
        addElement(item);

        return item;
    }
    ….
```

This could not be undone, because that would break backwards-compatibility..

WHY USE DELEGATION?

- inheritance can be **more convenient**:
  - only define method f() once
  - no need to forward calls
  - somewhat more efficient

- however, it is **less flexible**:
  - cannot change the implementation of f() after creating the object
  - in languages with single inheritance, you can only inherit methods from one superclass

TRUE DELEGATION

- with inheritance, the method in the superclass can use dynamic dispatch to invoke methods in a subclass

- with delegation, this requires some extra work:
  - pass receiving object's **`this`** pointer as argument to the delegate
  - delegate invokes methods on this reference when it needs to invoke methods on receiving object

- this form of delegation is called **true delegation**
- example of true delegation: "State" design pattern (p.305 in GoF book)

RULE OF THUMB FOR WHEN TO USE INHERITANCE VERSUS DELEGATION

use inheritance for

- **is-a** relationships that don't change over time
- situations where the class containing the actual operation is abstract

use delegation for

- **has-a**, **part-of** relationships
- **is-a-role-played-by** relationships
- relationships that change over time
- situations where multiple inheritance would be needed (in languages like Java that doe not allow MI)

DESIGNING FOR CHANGE

- many design patterns **introduce flexibility** to **avoid common causes of redesign** such as:
  - creating an object by specifying a class explicitly
  - dependence on specific operations
  - dependence on hardware/software platform
  - dependence on object representations or implementations
  - algorithmic dependencies
  - tight coupling
  - extending functionality by subclassing
  - inability to alter classes conveniently

THE ELEMENTS OF A PATTERN

A design pattern has 4 elements:

1. a **name**
    - *e.g, "Abstract Factory" or "Visitor"*
2. the **problem**
    - *that the pattern addresses*
3. the **solution**
    - *the program constructs that are part of the pattern*
4. the **consequences**
    - *the results and tradeoffs of applying the pattern*

Key considerations:

1. problem & solution have been **observed in practice**
2. choice of implementation language important

CLASSIFYING DESIGN PATTERNS

1. **purpose**: what a pattern does
   a) **creational**: concerned with creation of objects
   b) **structural**: related to composition of classes or objects
   c) **behavioral**: related to interaction and distribution of responsibility

2. scope
   a) **class-level**: concerned with relationship between classes and their subclasses
   b) **object-level**: concerned with object relationship (more dynamic, may be changed at run-time)

GOF DESIGN PATTERNS CLASSIFIED

	creational	structural	behavioral
class	Factory Method	Adapter (class)	Interpreter Template
object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Resp. Command Iterator Mediator Memento Observer State Strategy Visitor

CREATIONAL PATTERNS

- purpose
  - **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
- Builder
- Factory Method
- Prototype
- Singleton

EXAMPLE* TO ILLUSTRATE VARIOUS CREATIONAL PATTERNS

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

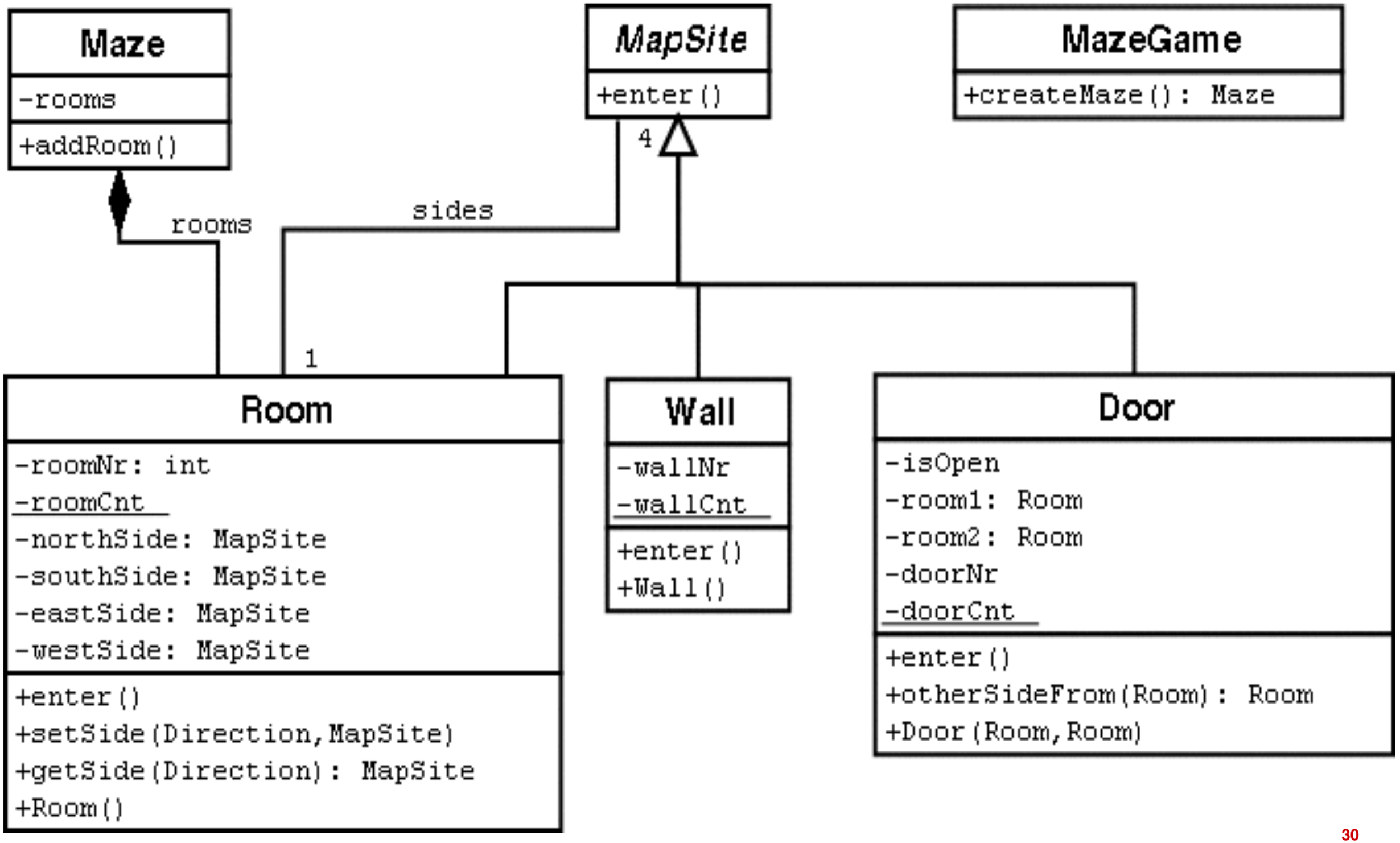
- a **Maze** consists of a number of **Rooms**
  - each Room has 4 sides: North, South, East, West
  - on each side of a room is a **Door** or a **Wall**

- 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

*adapted from Gamma, Helm, Johnson, Vlissides: “Design Patterns: Elements of Reusable Object-Oriented Software”

UML DIAGRAM FOR MAZE GAME



MAZE, MAPSITE, DIRECTION

```
class Maze {
    Maze() {
        System.out.println("creating a Maze");
    }

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

    private Set<Room> _rooms = new HashSet<Room>();
}

public abstract class MapSite {
    // enter() method omitted
}

public enum Direction {
    North, South, East, West
}
```

ROOM

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

    void setSide(Direction d, MapSite site) {
        switch(d){
        case North:
            _northSide = site;
        case South:
            _southSide = site;
        case East:
            _eastSide = site;
        case West:
            _westSide = site;
        }
        System.out.println("setting " +
              d.toString() + " side of “ +
              this.toString() + " to " +
              site.toString());
}
…
```

```
…
MapSite getSide(Direction d) {
    MapSite result = null;
    switch(d){
    case North:
        result = _northSide;
    case South:
        result = _southSide;
    case East:
        result = _eastSide;
    case West:
        result = _westSide;
    }
    return result;
 }
public String toString() {
    return "Room #" + new Integer(_roomNr).toString();
 }

 private int _roomNr;
 private static int _roomCnt = 1;
 private MapSite _northSide;
 private MapSite _southSide;
 private MapSite _eastSide;
 private MapSite _westSide;
}
```

WALL

```
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;
}
```

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;
}
```

MAZEGAME

```
public 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;
    }
}
```

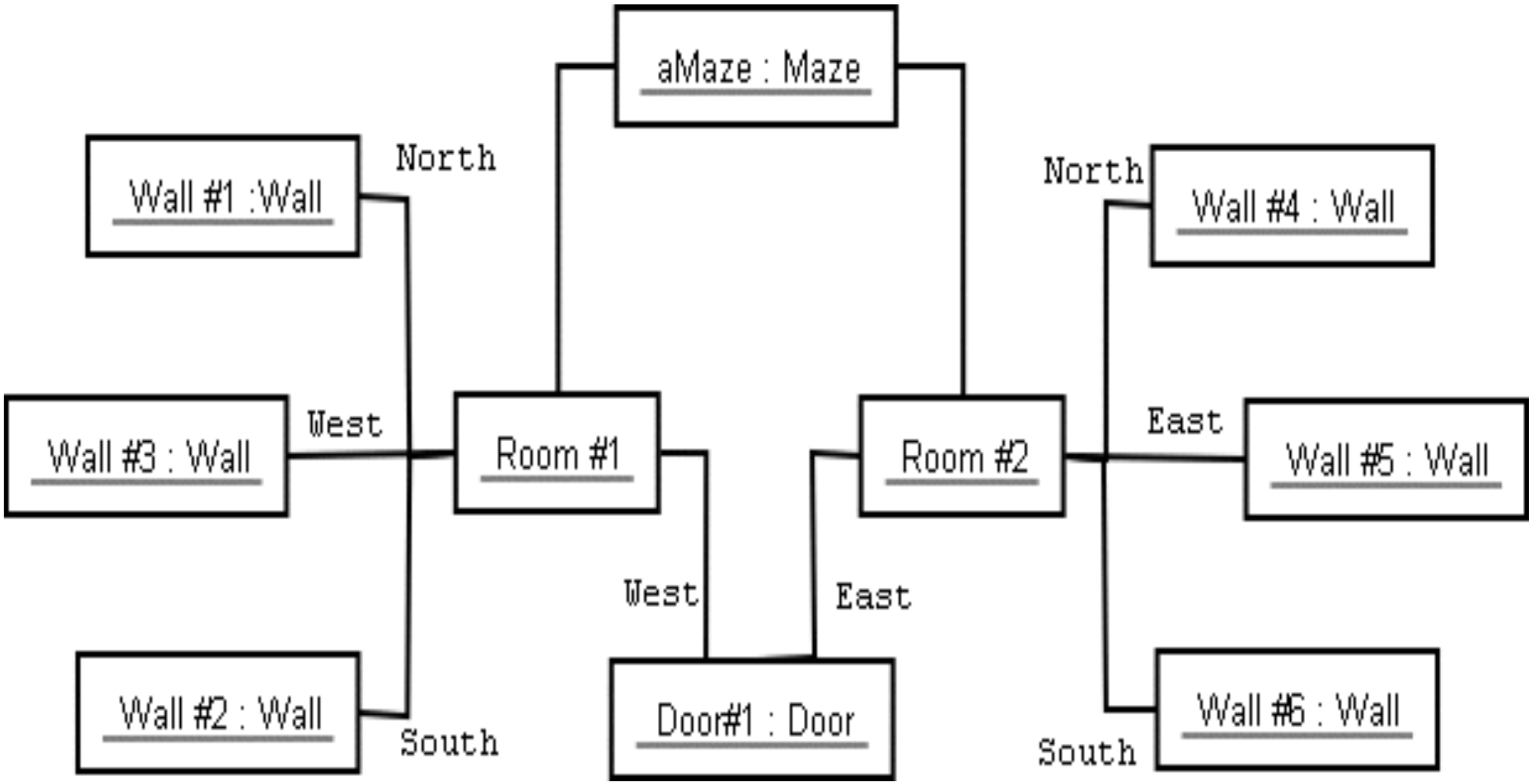
DRIVER FOR CREATING A MAZE

```
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
```

UML OBJECT DIAGRAM



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`, `EnchantedRoom`

Currently, any change to the structure or the components of the maze requires a complete rewrite of class MazeGame

WE CAN USE DESIGN PATTERNS TO MAKE 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**

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

- ensure that there is one maze per game, in a way that all objects have easy access to it.→ **Singleton**

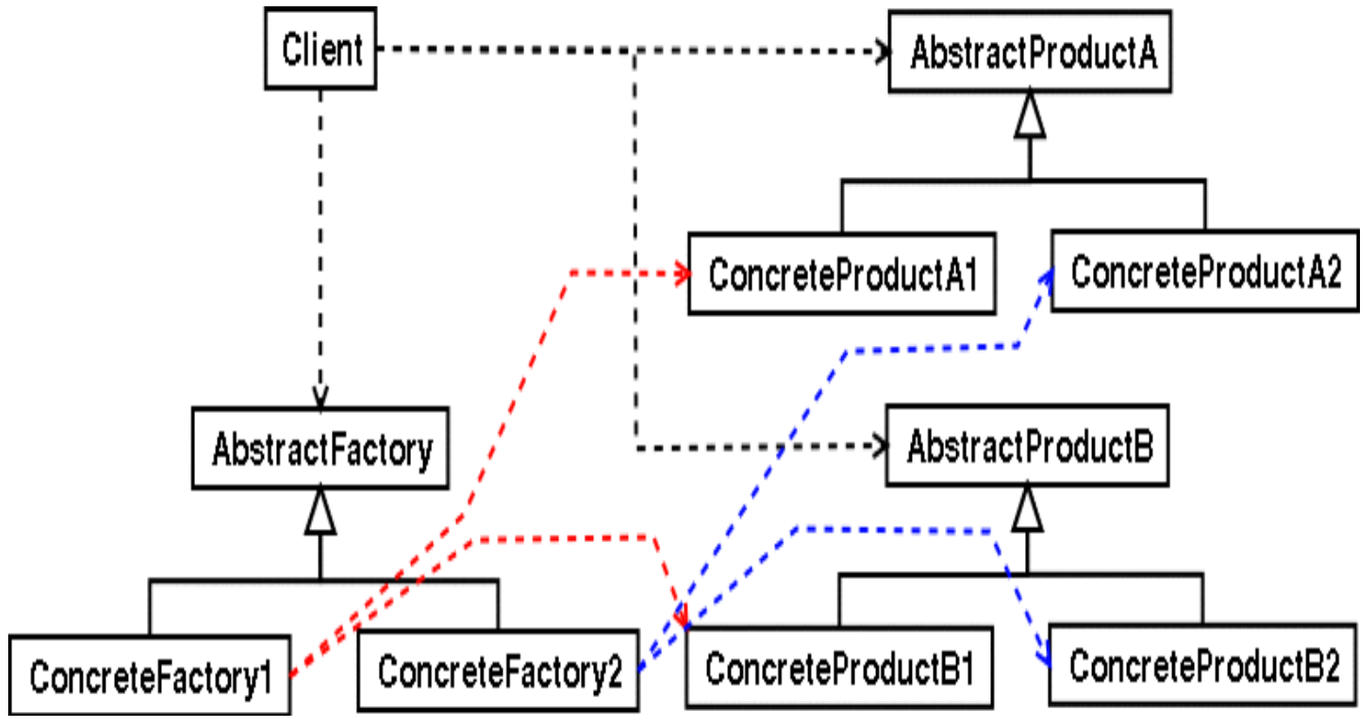
ABSTRACT FACTORY

- 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: 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 <u>only</u> interfaces of AbstractFactory/AbstractProduct



MAZE EXAMPLE REVISITED

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

```
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);
    }
}

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

```
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;
    }
}
```

ADDING NEW PRODUCTS IS NOW EASY

```
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)";
  }
}
```

```
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();
  }
}
```

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

SOME OBERVATIONS ABOUT THE EXAMPLE

- 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 to access methods/fields in ConcreteProducts
  - 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

FACTORY METHOD

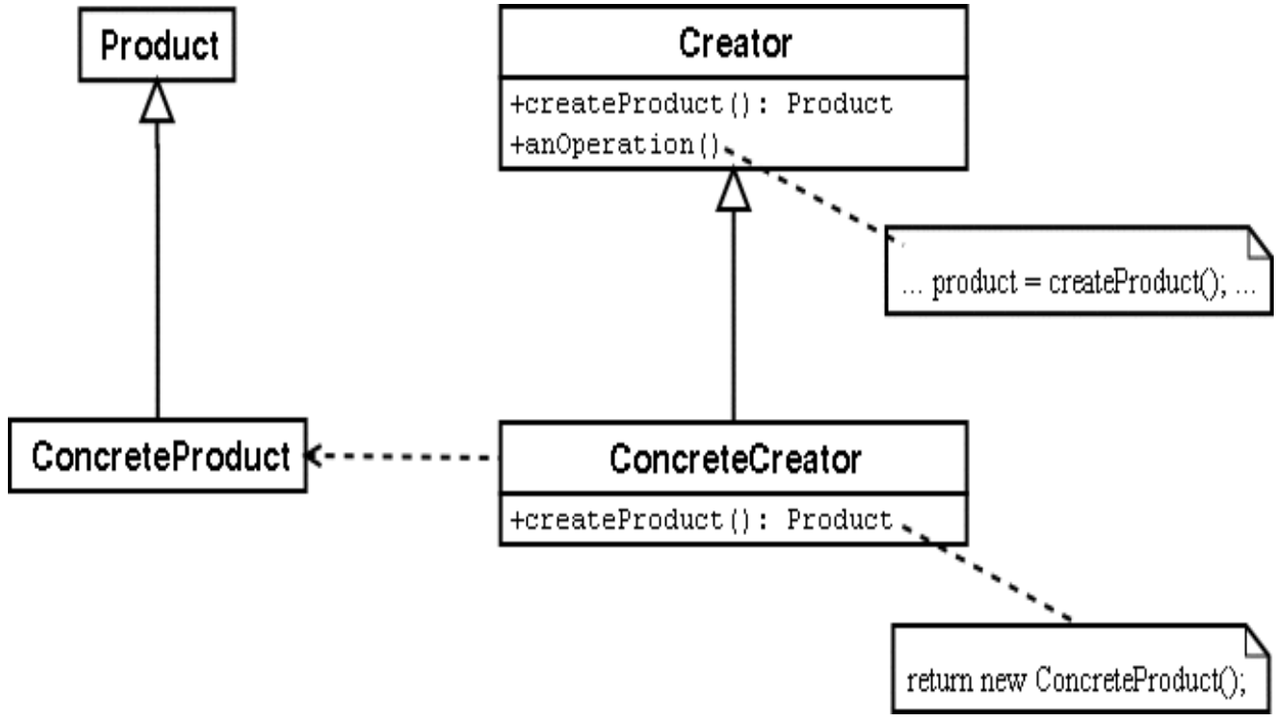
- define an interface for creating an object, but let subclasses decide which class to instantiate

- 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

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



MAZE EXAMPLE REVISITED

- 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 {

  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); }

  public Maze createMaze(){
    Maze aMaze = makeMaze();
    Room r1 = makeRoom();
    Room r2 = makeRoom();
    Door theDoor = makeDoor(r1,r2);
    aMaze.addRoom(r1); aMaze.addRoom(r2);
    r1.setSide(Direction.North, makeWall());
    r1.setSide(Direction.East, theDoor);
    r1.setSide(Direction.South, makeWall());
    r1.setSide(Direction.West, makeWall());
    r2.setSide(Direction.North, makeWall());
    r2.setSide(Direction.East, makeWall());
    r2.setSide(Direction.South, makeWall());
    r2.setSide(Direction.West, theDoor);
    return aMaze;
  }
}
```

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

public class MainEnchanted {
  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 (the next pattern we will study)

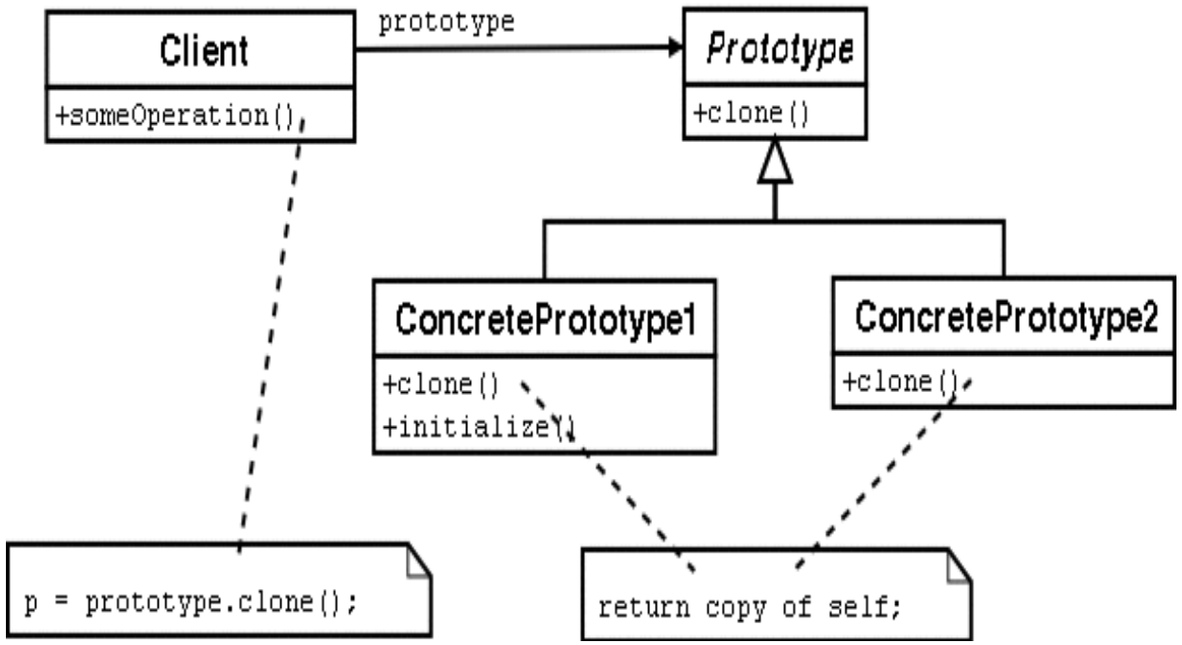
PROTOTYPE

- 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

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: BENEFITS

- 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”

- create a new subclass of class MazeFactory called MazePrototypeFactory
  - initialized by giving it a prototype Wall, Door, Room, Maze
  - these prototypes are stored in private fields
  - whenever a new component is needed, call clone() on the appropriate prototype

- initialize() method need for class Door, to reset the Rooms connected by the prototype Door

PROTOTYPE-BASED ABSTRACT FACTORY

```
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(); }
  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;
}
```

ADDING CLONE() METHODS

```
class Maze {
  …
  public Object clone(){
    // should restrict to empty mazes
    Maze maze = new Maze();
    maze._rooms = _rooms;
    return maze;
  }
  …
}

class Room extends MapSite {
  …
  public Object clone() {
    Room room = new Room();
    room._northSide = _northSide;
    room._southSide = _southSide;
    room._eastSide = _eastSide;
    room._westSide = _westSide;
    return room;
  }
  …
}
```

```
class Door extends MapSite {
  …
  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);
  }
  …
}

class Wall extends MapSite {
  …
  public Object clone() {
    Wall wall = new Wall();
    return wall;
  }
  …
}
```

UPDATED DRIVER

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

    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

```
class EnchantedRoom extends Room {
  …
}

class DoorNeedingSpell extends Door {
  …
}

class EnchantedMazeFactory extends MazePrototypeFactory {
  …
  public Room makeRoom(){ return new EnchantedRoom(new Spell()); }
  public Door makeDoor(Room r1, Room r2){ return new DoorNeedingSpell(r1,r2); }
}

public class Main {
  public static void main(String[] args){
    …
    MazeFactory factory =
      new EnchantedMazeFactory(mazeProto, wallProto,
                               roomProto, doorProto);

    game.createMaze(factory);
  }
}
```

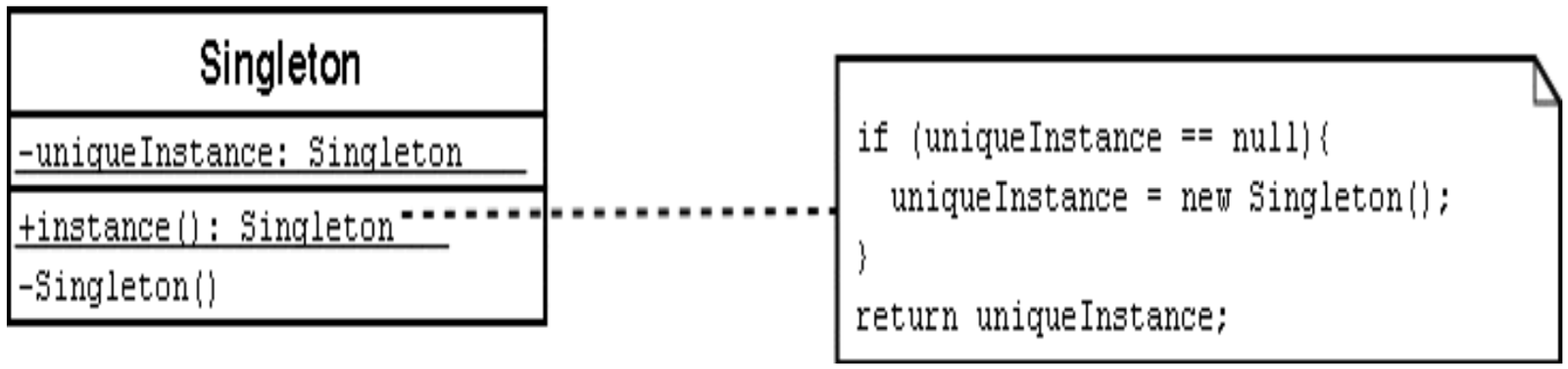
SINGLETON

- **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)
  - 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



EXAMPLE: APPLY SINGLETON TO MAZEFACTORY (ABSTRACTFACTORY)

```
class MazeFactory {

  private static MazeFactory _theFactory = null;

  private MazeFactory() { }

  public static MazeFactory instance() {
    if (_theFactory == null) {
      _theFactory = new MazeFactory();
    }
    return _theFactory;
  }
  …
}

public class Main {

  public static void main(String[] args) {
      MazeFactory factory = MazeFactory.instance();
      MazeGame game = new MazeGame();
      game.createMaze(factory);
  }

}
```

SINGLETON: CONSIDERATIONS

- unfortunately, there is no good solution that allows 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 what kind of MazeFactory it should build
  - requires rewriting the instance() method every time a subclass is added.
  - in Java, a solution would be to give instance() a String-typed parameter with the name of the factory, and to use reflection to create an object

- other languages have built-in support for singletons
  - e.g., Scala lets you declare an **`object`**

CREATIONAL PATTERNS: SUMMARY

- creational patterns make designs more flexible and extensible by instantiating classes in certain stylized ways
  - AbstractFactory
  - Builder
  - FactoryMethod
  - Prototype
  - Singleton
- next week:
  - structural design patterns
  - behavioral design patterns