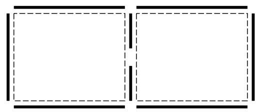
ECE450 – Software Engineering II

Today: **Design Patterns II**

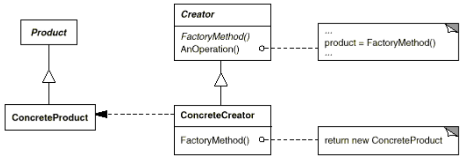
Reminder: Maze example

- Building a maze for a computer game
- A maze is a set of rooms
- A room knows its neighbours
  - Another room
  - A wall
  - A door



Reminder: Factory method

- Product
  - Defines the interface of objects the factory method creates
- ConcreteProduct
  - Implements the Product interface
- Creator
  - Declares the factory method which returns a Product type
  - Defines a default implementation
  - Calls the factory method itself
- ConcreteCreator
  - Overrides factory method: returns instance of ConcreteProduct



Sample code

```
public class MazeGame {
  public static void main(String[] args) {
    Maze m = new MazeGame().createMaze();
  }

  private Maze makeMaze() { return new Maze(); }
  private Wall makeWall() { return new Wall(); }
  private Room makeRoom(int r) { return new Room(r); }
  private Door makeDoor(Room r1, Room r2) {
    return new Door(r1, r2);
  }

  public Maze createMaze() {
    // do what's needed
  }
}
```

Sample code (cont)

```
public Maze createMaze() {
   Room r1 = makeRoom(1);
   Room r2 = makeRoom(2);
   Door d  = makeDoor(r1, r2);

   r1.setSide(Direction.North, makeWall());
   r1.setSide(Direction.East, d);
   r1.setSide(Direction.West, makeWall());
   r1.setSide(Direction.South, makeWall());

   r2.setSide(Direction.North, makeWall());
   r2.setSide(Direction.East•, makeWall());
   r2.setSide(Direction.West, d);
   r2.setSide(Direction.South, makeWall());

   Maze m = makeMaze();
   m.addRoom(r1);
   m.addRoom(r2);
   return m;
 }
```

Sample code (cont 2)

```
public class BombedMazeGame extends MazeGame {
  private Wall makeWall() {
    return new BombedWall();
  }
  private Room makeRoom(int r) {
    return new RoomWithABomb(r);
  }
}

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

Sample code (cont 3)

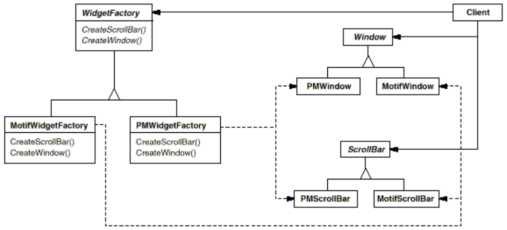
```
public static void main(String[] args) {
  Maze m = new EnchantedMazeGame().createMaze();
}

public static void main(String[] args) {
  Maze m = new BombedMazeGame().createMaze();
}
```

Several factory methods

- In our previous example, we had several *factory methods* helping us with object construction
- Sometimes it is useful to lump them together
  - Treat all features of enchanted mazes as one group, all of bombed mazes as another group
  - We'll call them "families"
  - User would only need one or the other

Enter the *Abstract Factory* pattern

- Abstract Factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes
  - e.g. look and feel portability
    - Independence
    - Enforced consistency

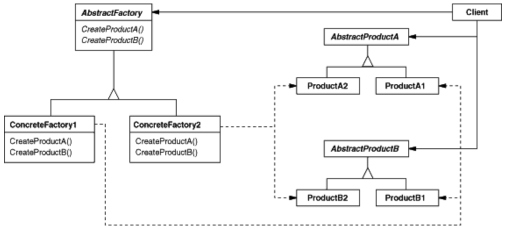


Applicability

- Use when:
  - A system should be independent of how its products are created, composed, and represented
  - A system should be configured with one of multiple families of products
  - A family of related product objects is designed to be used together, and you need to enforce this constraint
  - You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations
  - You want to hide and reuse awkward or complex details of construction
- Usually one starts by using Factory Methods and then moves on to Abstract Factories (or Prototypes, or Builders) when the methods are not flexible enough

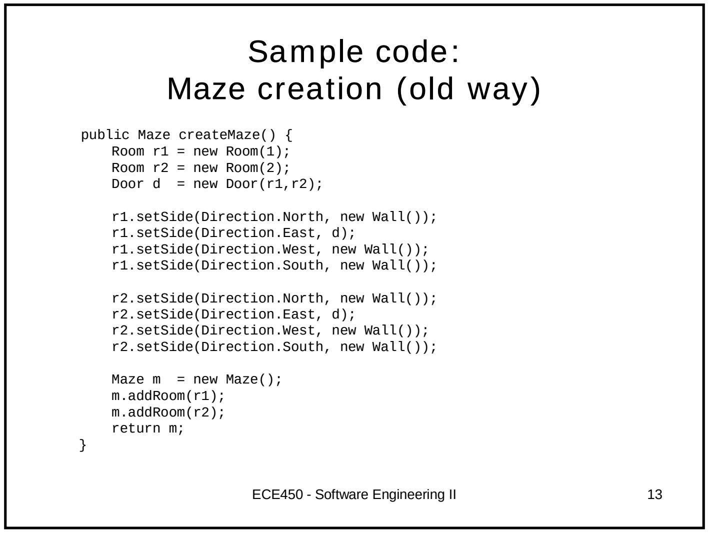
Structure

- AbstractFactory
  - Declares an interface for operations that create product objects
- ConcreteFactory
  - Implements the operations to create concrete product objects
- AbstractProduct
  - Declares an interface for a type of product object
- Product
  - Defines a product to be created by the corresponding concrete factory
  - Implements the AbstractProduct interface
- Client
  - Uses only interfaces declared by AbstractFactory and AbstractProduct classes



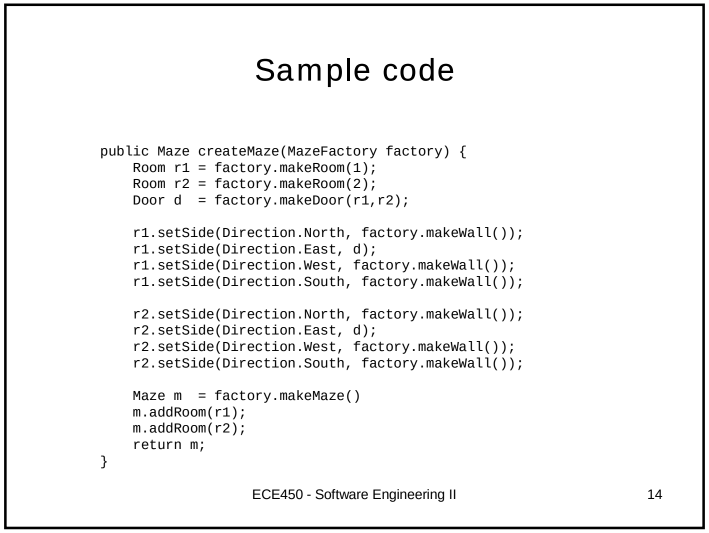
Sample code

```
public class MazeFactory {
    Maze makeMaze() { return new Maze(); }
    Wall makeWall() { return new Wall(); }
    Room makeRoom(int r) { return new Room(r); }
    Door makeDoor(Room r1, Room r2) { return new Door(r1,r2);}
}
```



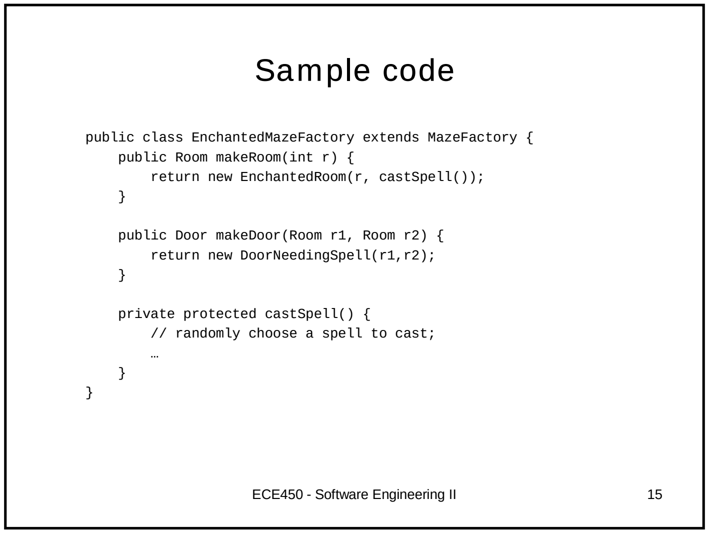
Sample code: Maze creation (old way)

```
public Maze createMaze() {
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door d  = new Door(r1,r2);

    r1.setSide(Direction.North, new Wall());
    r1.setSide(Direction.East, d);
    r1.setSide(Direction.West, new Wall());
    r1.setSide(Direction.South, new Wall());

    r2.setSide(Direction.North, new Wall());
    r2.setSide(Direction.East, d);
    r2.setSide(Direction.West, new Wall());
    r2.setSide(Direction.South, new Wall());

    Maze m  = new Maze();
    m.addRoom(r1);
    m.addRoom(r2);
    return m;
}
```



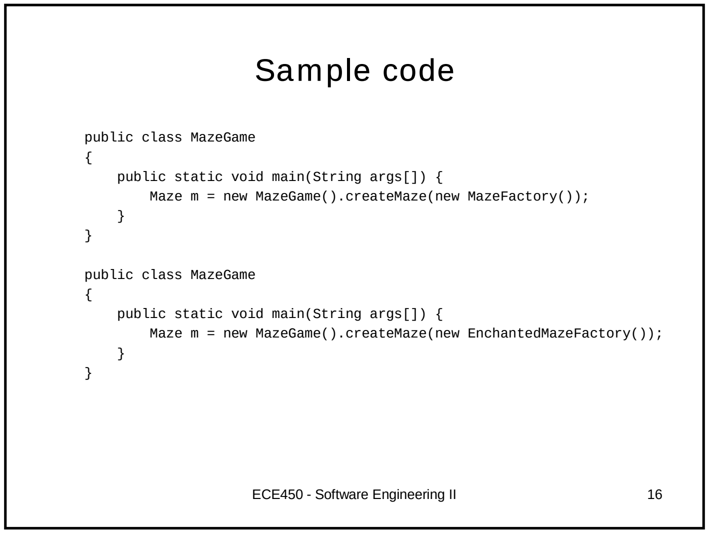
Sample code

```
public Maze createMaze(MazeFactory factory) {
    Room r1 = factory.makeRoom(1);
    Room r2 = factory.makeRoom(2);
    Door d  = factory.makeDoor(r1,r2);

    r1.setSide(Direction.North, factory.makeWall());
    r1.setSide(Direction.East, d);
    r1.setSide(Direction.West, factory.makeWall());
    r1.setSide(Direction.South, factory.makeWall());

    r2.setSide(Direction.North, factory.makeWall());
    r2.setSide(Direction.East, d);
    r2.setSide(Direction.West, factory.makeWall());
    r2.setSide(Direction.South, factory.makeWall());

    Maze m  = factory.makeMaze()
    m.addRoom(r1);
    m.addRoom(r2);
    return m;
}
```



Sample code

```
public class EnchantedMazeFactory extends MazeFactory {
    public Room makeRoom(int r) {
        return new EnchantedRoom(r, castSpell());
    }

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

    private protected castSpell() {
        // randomly choose a spell to cast;
        …
    }
}
```



Sample code

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

public class MazeGame
{
    public static void main(String args[]) {
        Maze m = new MazeGame().createMaze(new EnchantedMazeFactory());
    }
}
```

Consequences

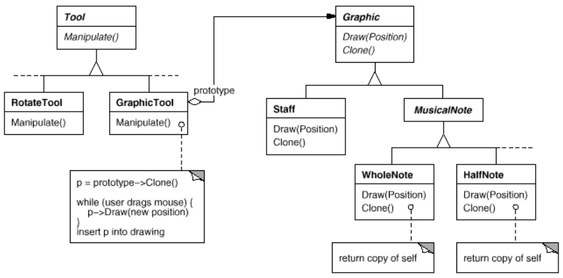
- It isolates concrete classes
  - Helps control the classes of objects that an application creates
  - Isolates clients from implementation classes
  - Clients manipulate instances through abstract interfaces
  - Product class names are isolated in the implementation of the concrete factory
    - They do not appear in the client code
  - It makes exchanging product families easy
    - The class of a concrete factory appears only once in the application (when it is instantiated)
    - Easy to change the concrete factory an application uses
    - The whole product family changes at once
  - It promotes consistency among products
    - When products are designed to work together, it's important that an application use objects only from one family at a time
    - Abstract Factory makes this easy to enforce
  - Supporting new kinds of products is difficult
    - Extending Abstract Factory to produce new product types isn't easy (need to extend factory interface and all concrete factories, add a new abstract product, plus implementing a new class in each family)

Implementation

- Factories as Singletons
  - An application typically needs only one instance of a ConcreteFactory per product family
  - Best implemented as a Singleton
    - More on that later
- Defining extensible factories
  - Hard to extend to new product types
  - Add parameter to operations that create products
    - Need only make()
    - Less safe, more flexible
    - Easier in languages that have common subclass (e.g. Java's Object)
    - Easier in more dynamically-typed languages (e.g. Smalltalk)
    - All products have same abstract interface
- Can also create the products through Prototypes instead of Factory Methods
  - Creates new products by cloning a prototype
  - *Prototype* is our next topic...

Prototype

- Specify the kinds of objects to create using a prototypical instance, and create new objects by cloning this prototype



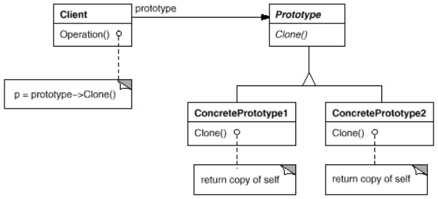
```
p = prototype->Clone()
while (user drags mouse) {
    p->Draw(new position)
}
insert p into drawing
```

Applicability

- Use...
  - When the classes to be instantiated are specified at run-time
    - E.g. for dynamic loading
  - To avoid building a class hierarchy of factories to parallel the hierarchy of products
  - When instances can have only one of a few states
    - May be better to initialize once, and then clone the prototypes

Structure

- 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



Sample code

```
public class MazePrototypeFactory extends MazeFactory
{
    private Maze prototypeMaze;
    private Wall prototypeWall;
    private Room prototypeRoom;
    private Door prototypeDoor;

    public MazePrototypeFactory(Maze pm, Wall pw, Room pr, Door pd) {
        prototypeMaze = pm;
        prototypeWall = pw;
        prototypeRoom = pr;
        prototypeDoor = pd;
    }

    …
}
```

Sample code (cont)

```
public class MazePrototypeFactory extends MazeFactory
{
    Wall makeWall() {
        Wall wall = null;
        try {
            wall = (Wall)prototypeWall.clone();
        } catch(CloneNotSupportedException e) { throw new Error(); }
        return wall;
    }
    Room makeRoom(int r) {
        Room room = null;
        try {
            room = (Room)prototypeRoom.clone();
        } catch(CloneNotSupportedException e) { throw new Error(); }
        room.initialize(r);
        return room;
    }
    …
}
```

Sample code (cont)

```
public abstract class MapSite implements Cloneable
{
    public abstract void enter();

    public String toString() {
        return getClass().getName();
    }

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

Sample code (cont)

```
public class Door extends MapSite {
  public Door(Room s1, Room s2) {
    initialize(s1, s2);
  }

  public void initialize(Room s1, Room s2) {
    side1 = s1;
    side2 = s2;
    open = true;
  }

  private Room side1;
  private Room side2;
  boolean open;
 }
```

Sample code (cont)

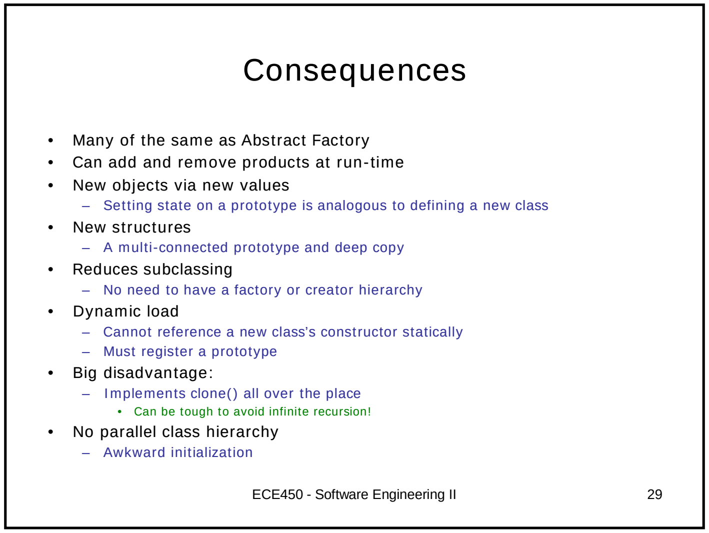
```
public class Room extends MapSite
{
    public Room(int r) {
        initialize(r);
    }

    public void initialize(int r) {
        room_no = r;
    }

    public Object clone() throws CloneNotSupportedException {
        Room r = (Room)super.clone();
        r.side = new MapSite[Direction.Num];
        return r;
    }
    …
    private int room_no;
    private MapSite[] side = new MapSite[Direction.Num];
}
```

Sample code (cont)

```
public class EnchantedRoom extends Room
{
   public EnchantedRoom(int r, Spell s) {
       super(r);
       spell = s;
   }

   public Object clone() throws CloneNotSupportedException {
       EnchantedRoom r = (EnchantedRoom)super.clone();
       r.spell = new Spell();
       return r;
   }

   private Spell spell;
}
```

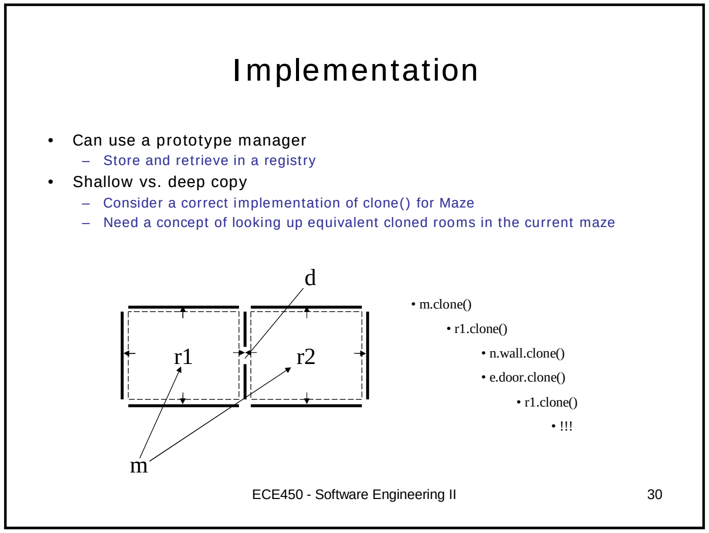
Sample code (cont)

```
public static void main(String args[]) {
    MazeFactory mf = new MazePrototypeFactory(
                             new Maze(), new Wall(),
                             new Room(0), new Door(null,null));
    Maze m = new MazeGame().createMaze(mf);
}


public static void main(String args[]) {
    MazeFactory mf = new MazePrototypeFactory(
        new Maze(), new Wall(),
        (Room)Class.forName("EnchantedRoom").newInstance(),
        (Door)Class.forName("DoorNeedingSpell").newInstance());
    Maze m = new MazeGame().createMaze(mf);
}
```



Consequences

- Many of the same as Abstract Factory
- Can add and remove products at run-time
- New objects via new values
  - Setting state on a prototype is analogous to defining a new class
- New structures
  - A multi-connected prototype and deep copy
- Reduces subclassing
  - No need to have a factory or creator hierarchy
- Dynamic load
  - Cannot reference a new class's constructor statically
  - Must register a prototype
- Big disadvantage:
  - Implements clone() all over the place
    - Can be tough to avoid infinite recursion!
- No parallel class hierarchy
  - Awkward initialization



Implementation

- Can use a prototype manager
  - Store and retrieve in a registry
- Shallow vs. deep copy
  - Consider a correct implementation of clone() for Maze
  - Need a concept of looking up equivalent cloned rooms in the current maze

- m.clone()
  - r1.clone()
    - n.wall.clone()
    - e.door.clone()
      - r1.clone()
        - !!!