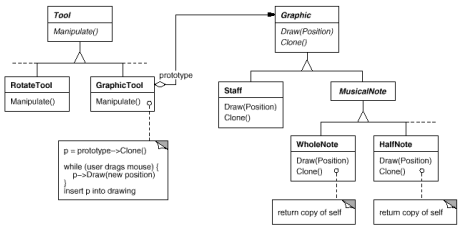
Prototype

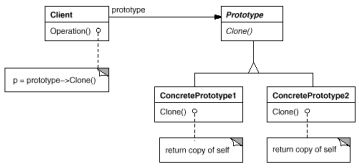
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  - e.g., reduce # of classes (# of tools) by initializing a generic tool with a prototype



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 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

```
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

```
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

```
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

```
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

```
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

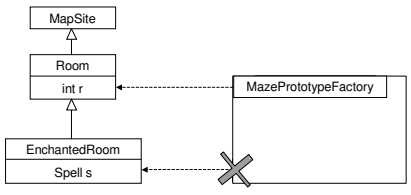
```
public static void main(String args[]) {
    MazeFactory mf = new MazePrototypeFactory(
                         new Maze(), new Wall(),
                         new Room(0), new Door(null,null));
    Maze m = new MazeGame().createMaze(mf);
}


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


Consequences

- Many of the same as AbstractFactory
- 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 + deep copy
- reducing subclassing
  - no need to have a factory or creator hierarchy
- dynamic load
  - cannot reference a new class's constructor statically
  - must register a prototype
- Disadvantage
  - implement clone() all over the place (can be tough).

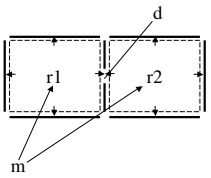
Consequences

- No parallel class hierarchy
  - awkward initialization



Implementation

- Use a prototype manager
  - store and retrieve in a registry (like Abstract Factory e.g.).
- Shallow versus 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()
        - !!!

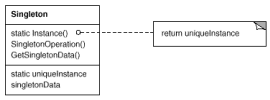
Singleton

- Ensure a class only has one instance, and provide a global point of access to it.
  - Many times need only one instance of an object
    - one file system
    - one print spooler
    - …
  - How do we ensure there is exactly one instance, and that the instance is easily accessible?
    - Global variable is accessible, but can still instantiate multiple instances.
    - make the class itself responsible

Applicability

- Use when:
  - there must be exactly one instance accessible from a well-known access point
  - the sole instance should be extensible via subclassing
    - clients should be able to use the extended instance without modifying their code

Structure



- Singleton
  - defines a class-scoped instance() operation that lets clients access its unique instance
  - may be responsible for creating its own unique instance

Sample Code

```
package penny.maze.factory;
public class MazeFactory {
    MazeFactory() { }

    private static MazeFactory theInstance = null;
    public static MazeFactory instance() {
        if( theInstance == null ) {
            String mazeKind =
                AppConfig.getProperties().getProperty("maze.kind");
            if( mazeKind.equals("bombed") ) {
                theInstance = new BombedMazeFactory();
            } else if( mazeKind.equals("enchanted") ) {
                theInstance = new EnchantedMazeFactory();
            } else {
                theInstance = new MazeFactory();
            }
        }
        return theInstance;
    }
    …
}
```


Sample Code

file .mazerc:

```
# Maze application parameters
maze.kind=enchanted
```

Sample Code

```
import java.io.*;
import java.util.Properties;

public class AppConfig {
    public static Properties getProperties() {
        if( props == null ) {
            props = new Properties(defaults());
            try {
                props.load(new FileInputStream(".mazerc"));
            } catch(IOException e) {
                System.err.println("Cannot read .mazerc, using defaults");
            }
        }
        return props;
    }

    private static Properties defaults() {
        Properties p = new Properties();
        p.put("maze.kind", "bombed");
        return p;
    }

    private static Properties props = null;
}
```


Consequences

- Controlled access to sole instance.
  - Because singleton encapsulates the sole instance, it has strict control.
- Reduced name space
  - one access method only
- Variable # of instances
  - can change your mind to have e.g., 5 instances
- Easy to derive and select new classes
  - access controlled through a single point of entry

Implementation

- Ensuring a unique instance
  - can't define singleton as a global object!
    - no guarantee only one will get created
      - must prohibit instantiation
    - may not know the state settings at init time (prior to main())
    - must create whether used or not
  - can't define as a static object
    - all of the above +
      - C++ doesn't define the order of construction across translation units.
- Subclassing the singleton class
  - as shown
  - C++: implement Singleton::instance() in each sub-class, only link one in at link time.
  - registry of singletons: `instance("bombed")`
    - subclasses register in static initializers (or "init()" methods).

Registry Technique