CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2005

Doc 14 Flyweight, Factory Method, Abstract Factory

Contents

References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 195-206, 107-116, 87-96

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, 1998, pp. 189-212, 63-76, 31-46

Sharable & Flyweight
Nation Example

Each country, like India or China, has a lot of information

A Nation object for one country may have a lot of data

A program may only have a few references to an India object

Having all references share the same object

- Saves space
- Saves time creating/copying the object
- Keeps all references consistent

Symbol & Interned Strings

Smalltalk

Only one instance a symbol with a give character sequence in an image

```
| a b |

a := #cat.
b := (‘ca’ , ‘t’) asSymbol.
a = b  “True”
a == b   “True – a & b point to same location”
```

Symbols

- Save space
- Make comparing symbols fast
- Make hashing symbols fast

Java

Compiler tries to use only one instance of a string with a given character sequence

String>>intern() returns a reference to a unique instance of a string

Text Example

Use objects to represent individual characters of the alphabet

Using objects allows the program to treat characters like any other part of the document - as an object

A character, like “G”, may require a fair amount of information:

- The character type - G not h or y
- Its font
- Its width, height, ascenders, descenders, etc.
- How to display/print the character
- Where it is in the document

Most of this information is the same for all instances of the same letter

So

- Have one G object and
- Have all places that need a G reference the same object

What if there is state information that is different between references?

Intrinsic State

- Information that is independent from the objects context
- The information that can be shared among many objects

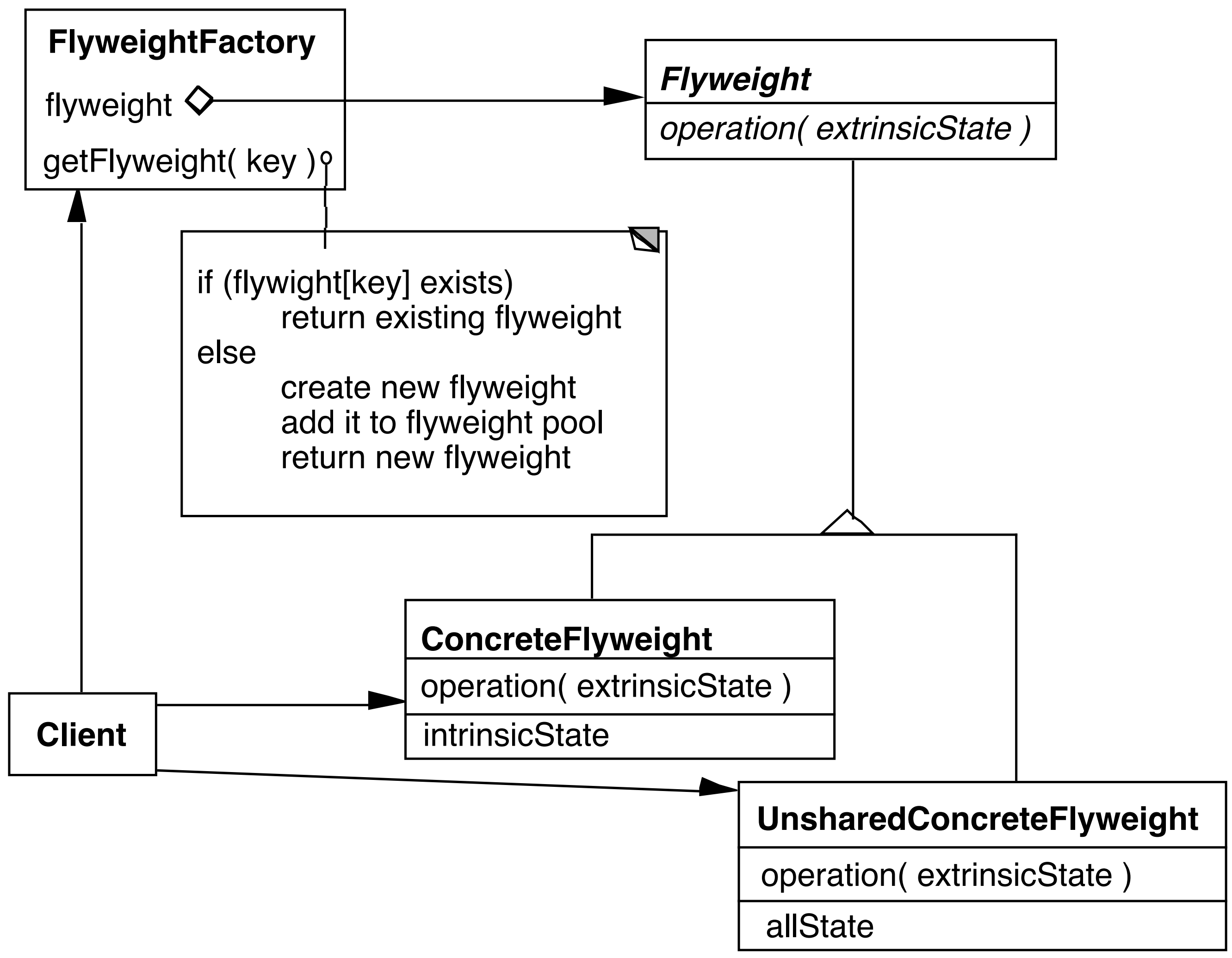
Extrinsic State

- Information that is dependent on the objects context
- The information that can not be shared among objects

So create one instance of the object and use the same object wherever you need an instance

The one object can store the intrinsic state, but someone else needs to store the extrinsic state for each context of the object

Structure


FlyweightFactory
flyweight
getFlyweight(key)
Flyweight
operation(extrinsicState)
if (flywight[key] exists)
return existing flyweight
else
create new flyweight
add it to flyweight pool
return new flyweight
ConcreteFlyweight
operation(extrinsicState)
intrinsicState
Client
UnsharedConcreteFlyweight
operation(extrinsicState)
allState

Applicability

The pattern can be used when all the following are true

- The program uses a large number of objects
- Storage cost are high due to the sheer quantity of objects
- The program does not use object identity

```
    MyClass* objectPtrA;
    MyClass* objectPtrB;

    if ( objectPtrA == objectPtrB ) //testing object identity
```

- Most object state can be made **extrinsic**

    - Extrinsic state is data that is stored outside the object
    - The extrinsic state is passed to the object as an argument in each method

- Many objects can be replaced by a relatively few shared objects, once the extrinsic state is removed

Implementation

Separating state from the flyweight

This is the hard part

Must remove the extrinsic state from the object

Store the extrinsic state elsewhere

- This needs to take up less space for the pattern to work

Each time you use the flyweight you must give it the proper extrinsic state

Managing Flyweights

Cannot use object identity on flyweights

Need factory to create flyweights, cannot create directly

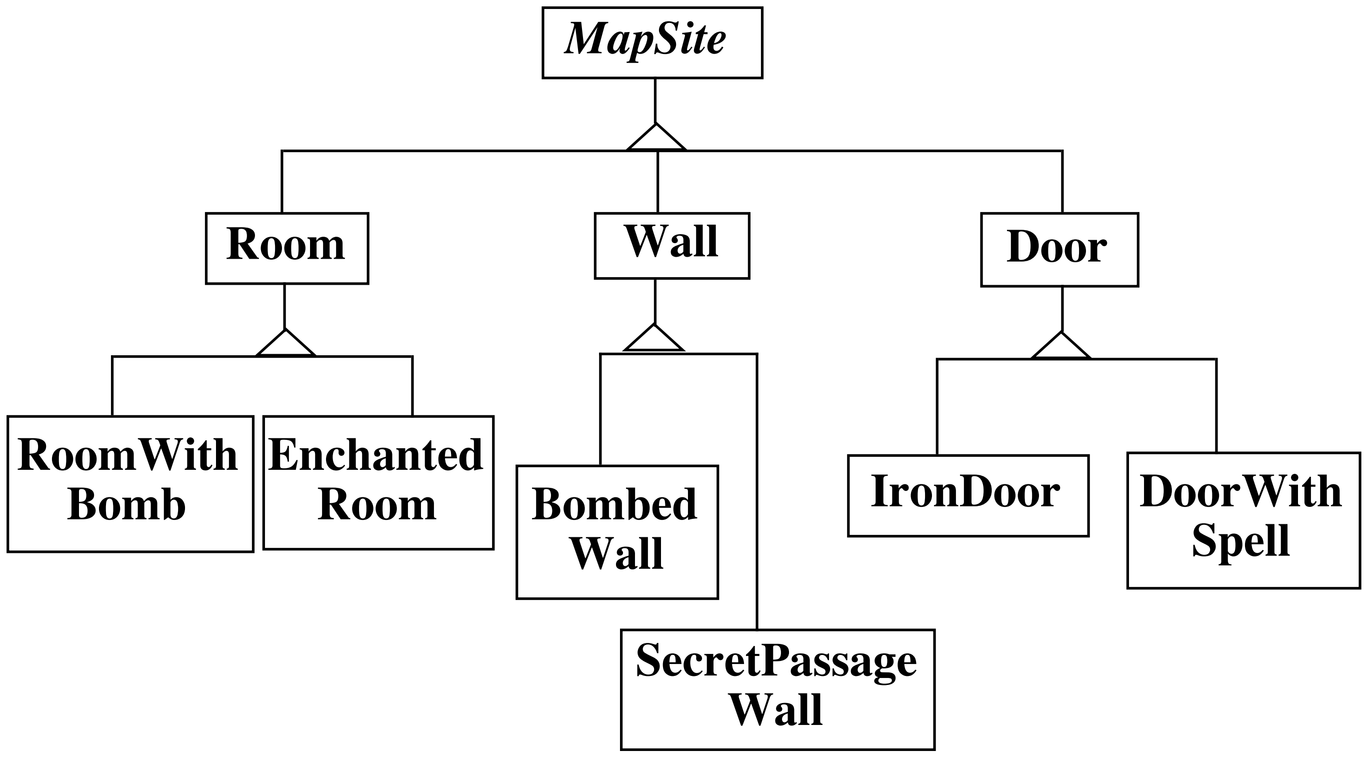
How do we know when we are done with a flyweight?

Factory Method

A template method for creating objects

Example - Maze Game

Classes for Mazes



Now a maze game has to make a maze

Maze Class Version 1

```
class MazeGame
   {

   public Maze createMaze()
      {
      Maze aMaze = new Maze();

      Room r1 = new Room( 1 );
      Room r2 = new Room( 2 );
      Door theDoor = new Door( r1, r2);

      aMaze.addRoom( r1 );
      aMaze.addRoom( r2 );

      etc.

      return aMaze;
      }
   }
```

How do we make other Mazes?

Subclass MazeGame, override createMaze

```
class BombedMazeGame extends MazeGame
   {

   public Maze createMaze()
      {
      Maze aMaze = new Maze();

      Room r1 = new RoomWithABomb( 1 );
      Room r2 = new RoomWithABomb( 2 );
      Door theDoor = new Door( r1, r2);

      aMaze.addRoom( r1 );
      aMaze.addRoom( r2 );

      etc.
```

Note the amount of cut and paste!

How do we make other Mazes?

Use Factory Method

```
class MazeGame
   {

   public Maze makeMaze() { return new Maze(); }
   public Room makeRoom(int n ) { return new Room( n ); }
   public Wall makeWall() { return new Wall(); }
   public Door makeDoor() { return new Door(); }

   public Maze CreateMaze()
      {
      Maze aMaze = makeMaze();

      Room r1 = makeRoom( 1 );
      Room r2 = makeRoom( 2 );
      Door theDoor = makeDoor( r1, r2);

      aMaze.addRoom( r1 );
      aMaze.addRoom( r2 );

      etc

      return aMaze;
      }
   }
```

Now subclass MazeGame override make methods

CreateMaze method stays the same

```
class BombedMazeGame extends MazeGame
   {

   public Room makeRoom(int n )
      {
      return new RoomWithABomb( n );
      }

   public Wall makeWall()
      {
      return new BombedWall();
      }
```

Applicability

Use when

- A class can't anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates
- You want to localize the knowledge of which help classes is used in a class

Consequences

- Eliminates need to hard code specific classes in code

- Requires subclassing to vary types used

- Provides hooks for subclasses

- Connects Parallel class hierarchies

Implementation
Two Major Varieties

- Top level Factory method is in an abstract class

```
abstract class MazeGame
   {
   public Maze makeMaze();
   public Room makeRoom(int n );
   public Wall makeWall();
   public Door makeDoor();
   etc.
   }

class MazeGame
   {
   public:
      virtual Maze* makeMaze() = 0;
      virtual Room* makeRoom(int n ) = 0;
      virtual Wall* makeWall() = 0;
      virtual Door* makeDoor() = 0;
```

- Top level Factory method is in a concrete class

  See examples on previous slides

Implementation - Continued
Parameterized Factory Methods

Let the factory method return multiple products

```
class Hershey
  {

  public Candy makeChocolateStuff( CandyType id )
    {
    if ( id == MarsBars ) return new MarsBars();
    if ( id == M&Ms ) return new M&Ms();
    if ( id == SpecialRich ) return new SpecialRich();

    return new PureChocolate();
    }
```

```
class GenericBrand extends Hershey
  {
  public Candy makeChocolateStuff( CandyType id )
    {
    if ( id == M&Ms ) return new Flupps();
    if ( id == Milk ) return new MilkChocolate();
    return super.makeChocolateStuff();
    }
  }
```

C++ Templates to Avoid Subclassing

```
template <class ChocolateType>
class Hershey
   {
   public:
      virtual Candy* makeChocolateStuff( );
   }

template <class ChocolateType>
Candy* Hershey<ChocolateType>::makeChocolateStuff( )
   {
   return new ChocolateType;
   }

Hershey<SpecialRich> theBest;
```

Java forName and Factory methods

With Java's reflection you can use a Class or a String to specify which type of object to create

Using a string replaces compile checks with runtime errors

```
class Hershey
   {
   private String chocolateType;

   public Hershey( String chocolate )
      {
      chocolateType = chocolate;
      }

   public Candy makeChocolateStuff( )
      {
      Class candyClass =  Class.forName( chocolateType );
      return (Candy) candyClass.newInstance();
      }


Hershey theBest = new Heshsey( "SpecialRich" );
```

Abstract Factory

Task - Write a cross platform window toolkit

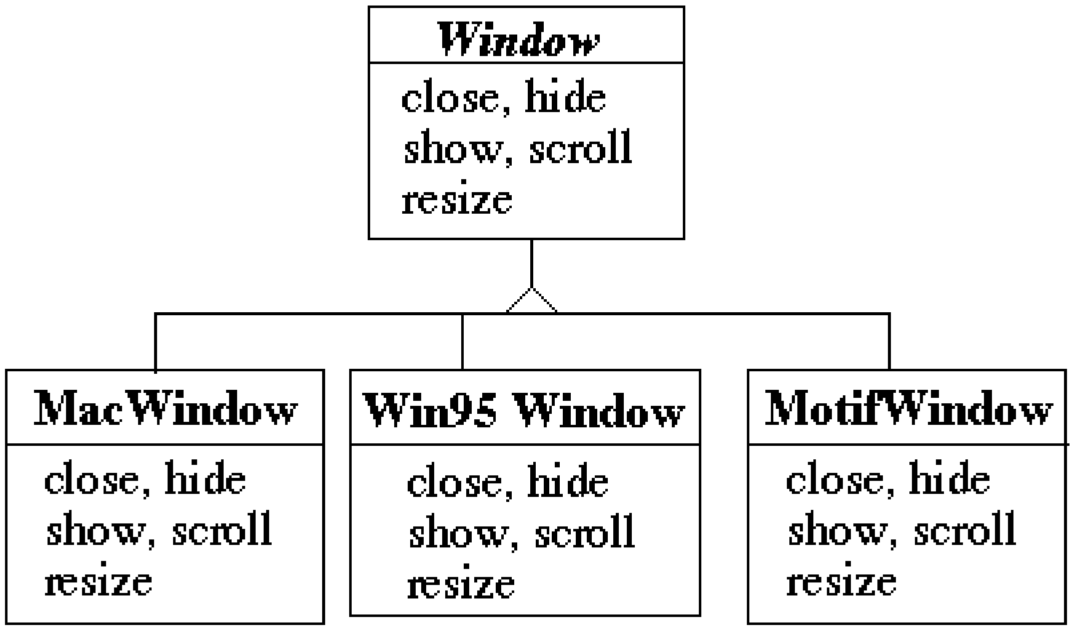
GUI interfaces to run on

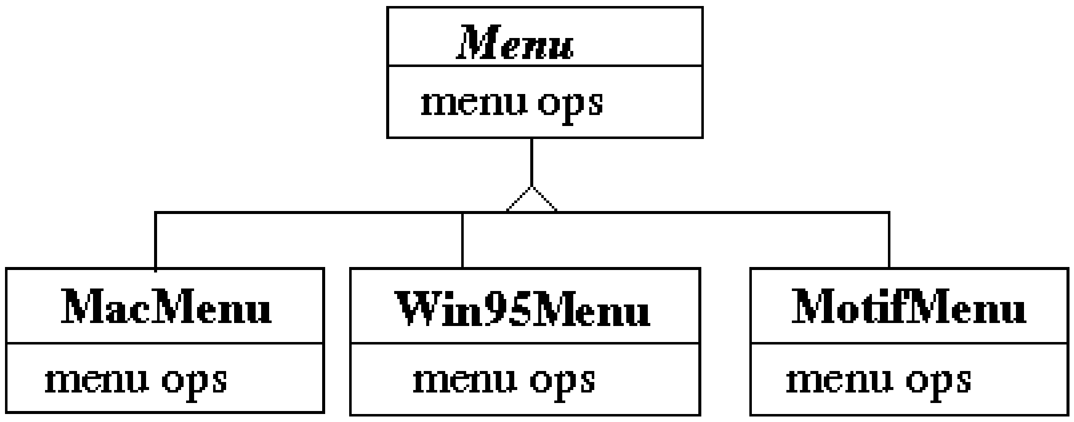
- Mac, PC & Unix
- Use the look and feel of each platform

We will look at widgets: Windows, Menu's and Buttons

Create

- An interface (or abstract class) for each widget
- A concrete class for each platform:





This allows the application to write to the widget interface

```
public void installDisneyMenu()
   {
   Menu disney = create a menu somehow
   disney.addItem( "Disney World" );
   disney.addItem( "Donald Duck" );
   disney.addItem( "Mickey Mouse" );
   disney.addGrayBar( );
   disney.addItem( "Minnie Mouse" );
   disney.addItem( "Pluto" );
   etc.
   }
```

How to create the widget so

- We get the correct interface widgets
- Minimize places in code that know the platform

Use Abstract Factory

```
abstract class WidgetFactory
   {
   public Window createWindow();
   public Menu createMenu();
   public Button createButton();
   }

class MacWidgetFactory extends WidgetFactory
   {
   public Window createWindow()
      { code to create a mac window }

   public Menu createMenu()
      { code to create a mac Menu }

   public Button createButton()
      { code to create a mac button }
   }

class Win95WidgetFactory extends WidgetFactory
   {
   public Window createWindow()
      { code to create a Win95 window }

   public Menu createMenu()
      { code to create a Win95 Menu }

   public Button createButton()
      { code to create a Win95 button }
   }
```

Now to get code that works for all platforms we get:

```
public void installDisneyMenu(WidgetFactory myFactory)
   {
   Menu disney = myFactory.createMenu();
   disney.addItem( "Disney World" );
   disney.addItem( "Donald Duck" );
   disney.addItem( "Mickey Mouse" );
   disney.addGrayBar( );
   disney.addItem( "Minnie Mouse" );
   disney.addItem( "Pluto" );
   etc.
   }
```

We just need to make sure that the application for each platform creates the proper factory

How Do Factories create Widgets?
Method 1) My Factory Method

```
abstract class WidgetFactory
   {
   public Window createWindow();
   public Menu createMenu();
   public Button createButton();
   }

class MacWidgetFactory extends WidgetFactory
   {
   public Window createWindow()
      { return new MacWidow() }

   public Menu createMenu()
      { return new MacMenu() }

   public Button createButton()
      { return new MacButton()  }
   }
```

How Do Factories create Widgets? Method 2) Their Factory Method

```
abstract class WidgetFactory {
   private Window windowFactory;
   private Menu menuFactory;
   private Button buttonFactory;

   public Window createWindow()
      { return windowFactory.createWindow() }

   public Menu createMenu();
      { return menuFactory.createMenu() }

   public Button createButton()
      { return buttonFactory.createMenu() }
}

class MacWidgetFactory extends WidgetFactory {
   public MacWidgetFactory() {
      windowFactory = new MacWindow();
      menuFactory = new MacMenu();
      buttonFactory = new MacButton();
   }
}

class MacWindow extends Window {
   public Window createWindow() { blah }
   etc.
```

Method 2) Their Factory Method
When does this make Sense?

There might be more than one way to create a widget

```
abstract class WidgetFactory {
   private Window windowFactory;
   private Menu menuFactory;
   private Button buttonFactory;

   public Window createWindow()
      { return windowFactory.createWindow() }

   public Window createWindow( Rectangle size)
      { return windowFactory.createWindow( size ) }

   public Window createWindow( Rectangle size, String title)
      { return windowFactory.createWindow( size, title) }

   public Window createFancyWindow()
      { return windowFactory.createFancyWindow() }

   public Window createPlainWindow()
      { return windowFactory.createPlainWindow() }
```

Using factory method allows abstract class to do all the different ways to create a window.

Subclasses just provide the objects windowFactory, menuFactory, buttonFactory, etc.

How Do Factories create Widgets?
Method 2.5) Subclass returns Class

```
abstract class WidgetFactory {

   public Window createWindow()
      { return windowClass().newInstance() }

   public Menu createMenu();
      { return menuClass().newInstance()  }

   public Button createButton()
      {  return buttoneClass().newInstance() }

   public Class windowClass();
   public Class menuClass();
   public Class buttonClass();
}

class MacWidgetFactory extends WidgetFactory {
   public Class windowClass()
      { return MacWindow.class; }

   public Class menuClass()
      { return MacMenu.class; }

   public Class buttonClass()
      { return MacButton.class; }
}
```

Smalltalk practice
   Parent class normally does more complex stuff

How Do Factories create Widgets?
Method 3) Prototype

```
class WidgetFactory
   {
   private Window windowPrototype;
   private Menu menuPrototype;
   private Button buttonPrototype;

   public WidgetFactory( Window windowPrototype,
                  Menu menuPrototype,
                  Button buttonPrototype)
      {
      this.windowPrototype = windowPrototype;
      this.menuPrototype = menuPrototype;
      this.buttonPrototype = buttonPrototype;
      }

   public Window createWindow()
      { return windowFactory.createWindow() }

   public Window createWindow( Rectangle size)
      { return windowFactory.createWindow( size ) }

   public Window createWindow( Rectangle size, String title)
      { return windowFactory.createWindow( size, title) }

   public Window createFancyWindow()
      { return windowFactory.createFancyWindow() }

   etc.
```

There is no need for subclasses of WidgetFactory.

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 implementation

Consequences

- It isolates concrete classes
- It makes exchanging product families easy
- It promotes consistency among products
- Supporting new kinds of products is difficult

Implementation

- Factories as singletons
- Defining extensible factories

Problem: Cheating Application Code

```
public void installDisneyMenu(WidgetFactory myFactory)
   {
   // We ship next week, I can't get the stupid generic Menu
   // to do the fancy Mac menu stuff
   // Windows version won't ship for 6 months
   // Will fix this later

   MacMenu disney = (MacMenu) myFactory.createMenu();
   disney.addItem( "Disney World" );
   disney.addItem( "Donald Duck" );
   disney.addItem( "Mickey Mouse" );
   disney.addMacGrayBar( );
   disney.addItem( "Minnie Mouse" );
   disney.addItem( "Pluto" );
   etc.
   }
```

How to avoid this problem?