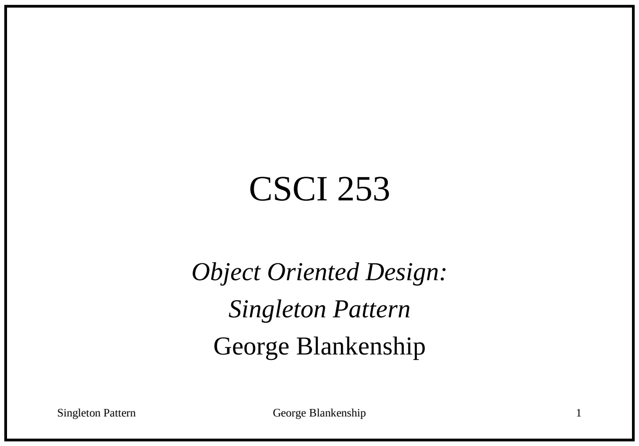
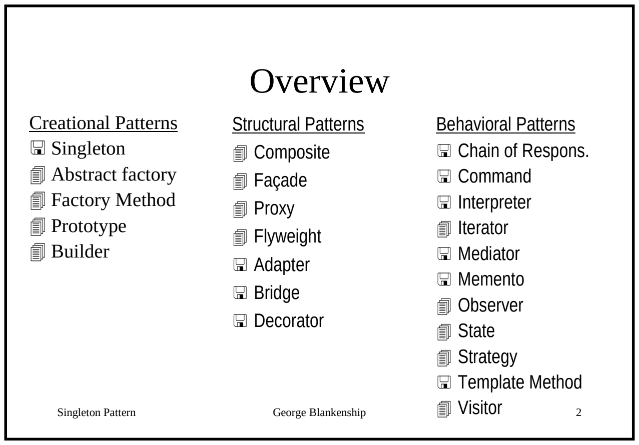


CSCI 253

Object Oriented Design:

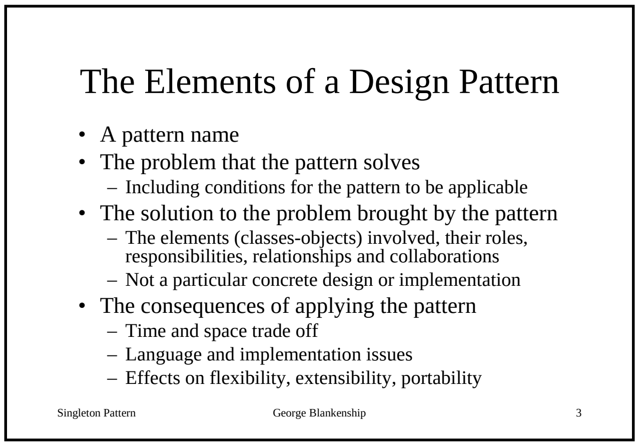
Singleton Pattern

George Blankenship



Overview

Creational Patterns	Structural Patterns	Behavioral Patterns
🖫 Singleton	🗐 Composite	🖫 Chain of Respons.
🗐 Abstract factory	🗐 Façade	🖫 Command
🗐 Factory Method	🗐 Proxy	🖫 Interpreter
🗐 Prototype	🗐 Flyweight	🗐 Iterator
🗐 Builder	🖫 Adapter	🖫 Mediator
	🖫 Bridge	🖫 Memento
	🖫 Decorator	🗐 Observer
		🗐 State
		🗐 Strategy
		🖫 Template Method
		🗐 Visitor

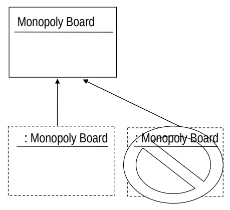


The Elements of a Design Pattern

- A pattern name
- The problem that the pattern solves
  - Including conditions for the pattern to be applicable
- The solution to the problem brought by the pattern
  - The elements (classes-objects) involved, their roles, responsibilities, relationships and collaborations
  - Not a particular concrete design or implementation
- The consequences of applying the pattern
  - Time and space trade off
  - Language and implementation issues
  - Effects on flexibility, extensibility, portability

The Singleton Pattern: The Problem

Ensure that a class has exactly one instance and provide a global point of access to it



- There can be only one print spooler, one file system, one window manager in a standard application
- There is only one game board in a monopoly game; one maze in a maze-game

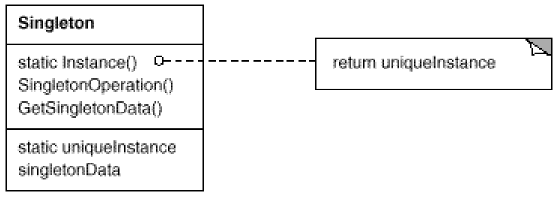
Singleton Pattern George Blankenship 4

The Singleton Pattern Participant & Collaboration

- Participant:
- Singleton:
  - is responsible for creating and storing its own unique instance
  - defines an Instance operation that lets clients access its unique instance
- Collaboration:
  - the “class level” Instance operation will either return or create and return the sole instance; a “class level” attribute will contain either a default indicating there is no instance yet or the sole instance

Singleton Pattern George Blankenship 5

Control Unique Existance



Singleton Pattern George Blankenship 6

Exception Definition

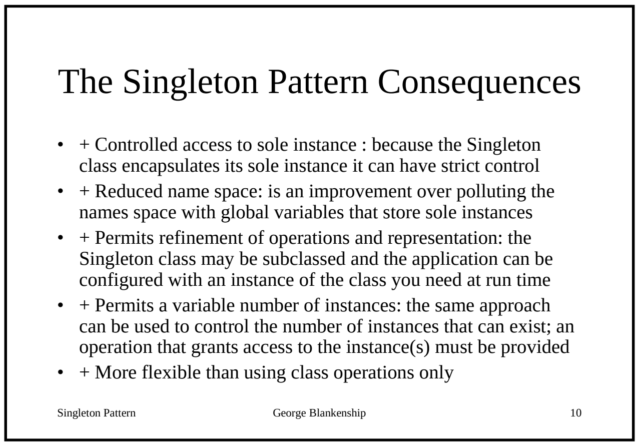
```
class SingletonException extends
   RuntimeException {
    // new exception type for singleton classes
   public SingletonException() {super();}
   // new exception type with description
   public SingletonException(String s) {super(s);}
   }
```

PrintSpooler Class

```
class PrintSpooler {
   //this is a prototype for a printer-spooler class
   //such that only one instance can ever exist
   static boolean instance_flag=false; //true if 1 instance
   public PrintSpooler() throws SingletonException {
         if (instance_flag)
                  throw new SingletonException("Only one spooler allowed");
         else
                  instance_flag = true; //set flag for 1 instance
         System.out.println("spooler opened");
   }
//-------------------------------------------
   public void finalize() {
         instance_flag = false; //clear if destroyed
   }
}
```

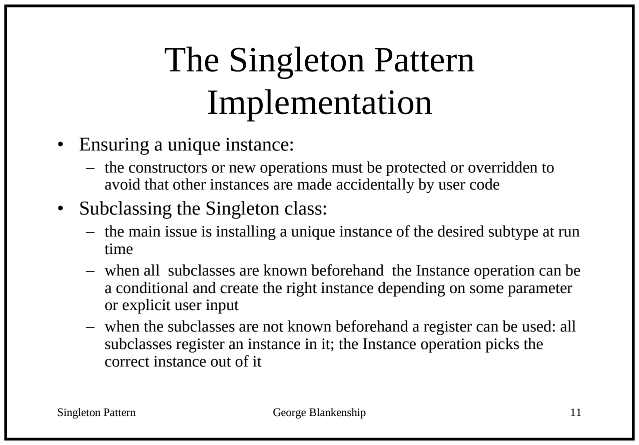
Print Spooler Creation

```
public class singleSpooler {
   static public void main(String argv[]) {
         PrintSpooler pr1, pr2;
         //open one spooler--this should always work
         System.out.println("Opening one spooler");
         try{pr1 = new PrintSpooler();}
         catch (SingletonException e) {System.out.println(e.getMessage());}
         //try to open another spooler --should fail
         System.out.println("Opening two spoolers");
         try {pr2 = new PrintSpooler();}
         catch (SingletonException e) {System.out.println(e.getMessage());}
   }
}
```



The Singleton Pattern Consequences

- + Controlled access to sole instance : because the Singleton class encapsulates its sole instance it can have strict control
- + Reduced name space: is an improvement over polluting the names space with global variables that store sole instances
- + Permits refinement of operations and representation: the Singleton class may be subclassed and the application can be configured with an instance of the class you need at run time
- + Permits a variable number of instances: the same approach can be used to control the number of instances that can exist; an operation that grants access to the instance(s) must be provided
- + More flexible than using class operations only



The Singleton Pattern Implementation

- Ensuring a unique instance:
  - the constructors or new operations must be protected or overridden to avoid that other instances are made accidentally by user code
- Subclassing the Singleton class:
  - the main issue is installing a unique instance of the desired subtype at run time
  - when all subclasses are known beforehand the Instance operation can be a conditional and create the right instance depending on some parameter or explicit user input
  - when the subclasses are not known beforehand a register can be used: all subclasses register an instance in it; the Instance operation picks the correct instance out of it