


Design Pattern: Singleton

- Purpose:
 - to guarantee that a class has only one instance
 - make it possible to access that instance globally
- Example:
 - only one file system object
 - only one screen object
 - only one Application object

Singleton

Singleton

 getInstance() : Singleton {static; always returns same instance}

```
class Singleton {
    private static Singleton theInstance = new Singleton();

    private Singleton() {...}
        // constructor private so nobody can create more instances

    public static Singleton getInstance() {
        return theInstance;
    }
    // other stuff
}
```

Singleton

- Use when...
 - there's a good reason to have only one instance
 - example: two file system instances might overwrite each others changes
- Don't use to emulate procedural programming
- Don't forget to make the constructor private
 - otherwise everybody can create more instances

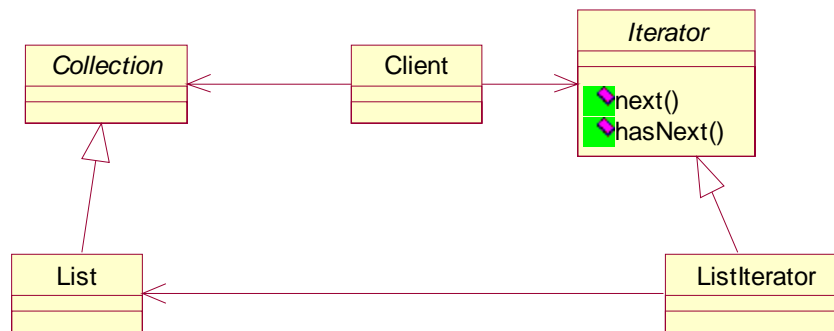
Pattern: Iterator

- Purpose:
 - allow sequential access to the elements of a collection
 - do not show internal implementation of the collection
- Example:
 - java.util.Iterator
 - provides access to hashtables, lists, sets...
 - all implement Collection interface
 - typically used in "for" loops

```
for( Iterator i = collection.iterator(); i.hasNext(); ) {
    System.out.println( i.next().toString() );
}
```
 - new Java 1.5 syntax:

```
for( Object o: collection ) {
    System.out.println( o.toString() );
}
```

Iterator: Sample Class Diagram



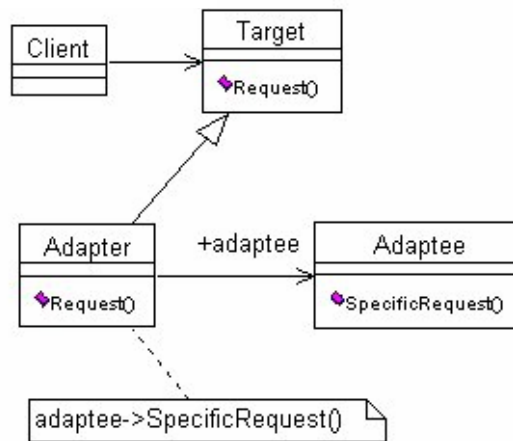
Iterator

- Use Iterator when...
 - you have collection classes
 - even though not all of them are sequential (e.g. Set), you want to be able to get the elements sequentially
 - typically, for loops
 - you don't care about the exact type of each collection
 - you don't need random access
- Disadvantages
 - no random access
 - overhead
- Advantages
 - changing the exact type of a collection does not break the code using it
 - writing loops becomes simpler: programmer does not need to know all the details about the collection
- Widely used in class libraries

Design Pattern: Adapter / Wrapper

- Convert the interface of one class into another one that is expected by a client
- Structural, object-based pattern
- Useful for adapting
 - legacy code
 - code from libraries
 - classes that are used in different contexts
- Advantage
 - allows reuse of existing class without changing it
- Disadvantage
 - one more class

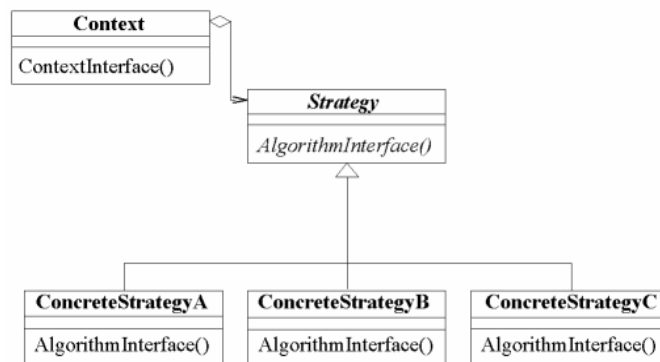
Adapter: Structure



Strategy Pattern

- Intent:
 - Define a family of algorithms
 - make them exchangeable
- It often depends on the situation, which algorithm is best
 - algorithm a: better for sorting numbers
 - algorithm b: better for sorting addresses
- Usable
 - when you need several algorithms
 - that all fulfill the same task

Strategy Pattern



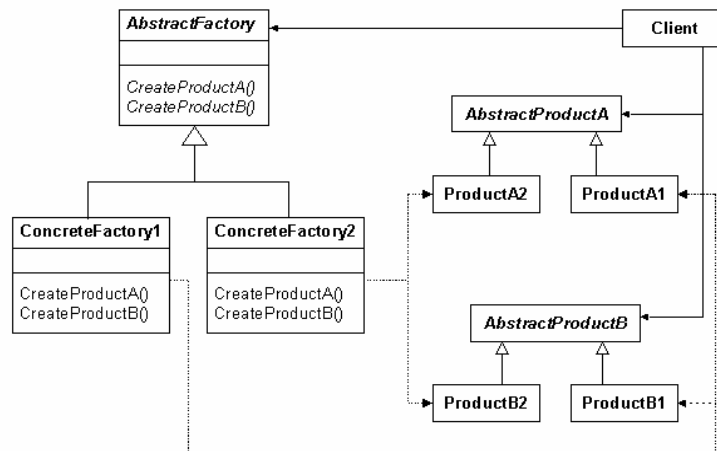
Strategy Pattern

- Advantages:
 - Better than subclassing the Context class itself
 - algorithms can be used by other Contexts too
 - Context may already have other subclasses
 - Reusability
 - Active choice between algorithms
 - clients can decide which they prefer
- Disadvantages:
 - clients need to understand tradeoffs between algorithms
 - increased number of objects
 - may be reduced by making Strategy objects stateless
 - can be shared

Abstract Factory Pattern

- Intent:
 - enables creation of a family of similar objects
 - without giving out their concrete classes
- Motivating example:
 - Gui library
 - runs on MS Windows, Motif, and Mac OS
 - how to create widgets?
 - bad solution:
 - have “new MacOSButton()”, “new MacOSScrollBar” all over the code
 - most of the code should be independent from OS!

Abstract Factory Pattern



Abstract Factory Pattern

- Advantages:
 - concrete classes are encapsulated
 - clients do not need to know them
 - object families can be easily exchanged
 - change from MS Windows 98 to MS Windows XP
 - objects are guaranteed to be consistent
 - can't happen that clients accidentally mix MS and Mac widgets
- Disadvantage:
 - if a new class is added to the family of objects, all **AbstractFactory** subclasses must be updated

Summary — Design Patterns

- Not the same as architecture
 - architecture deals with overall structure of an application
 - design patterns deal with a few classes / objects
- But there are overlaps
 - observer pattern
 - related to three-tier architecture
- Design patterns are more abstract than class libraries
 - do not define actual classes
 - just give some constraints for a set of classes
- Used very often in practice
- Hard to find and document
 - but easy to use once you understand a pattern
- Big theme: Anticipate Change!
 - find out what needs to be variable about your code
 - make it variable