CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2009
Doc 8 Pattern Intro
Feb 19, 2009

References

A Pattern Language, Christopher Alexander, 1977

Patterns for Classroom Education, Dana Anthony, pp. 391-406, Pattern Languages of Program Design 2, Addison Wesley, 1996

Smalltalk Best Practice Patterns, Kent Beck, 1997

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution"

A Pattern Language, Christopher Alexander, 1977

The process of waiting has inherent conflicts in it.

Waiting for doctor, airplane etc. requires spending time hanging around doing nothing

Cannot enjoy the time since you do not know when you must leave

Classic "waiting room"	**Fundamental problem**
Dreary little room	How to spend time "wholeheartedly" and
People staring at each other	Still be on hand when doctor, airplane etc arrive
Reading a few old magazines	
Offers no solution	

Fuse the waiting with other activity that keeps them in earshot

- Playground beside Pediatrics Clinic
- Horseshoe pit next to terrace where people waited

Allow the person to become still meditative

- A window seat that looks down on a street
- A protected seat in a garden
- A dark place and a glass of beer
- A private seat by a fish tank

Therefore:

"In places where people end up waiting create a situation which makes the waiting positive. Fuse the waiting with some other activity - newspaper, coffee, pool tables, horseshoes; something which draws people in who are not simple waiting. And also the opposite: make a place which can draw a person waiting into a reverie; quiet; a positive silence"

Problem

Two concepts are each a prerequisite of the other
To understand A one must understand B
To understand B one must understand A
A "chicken and egg" situation

Constraints and Forces

First explain A then B
    Everyone would be confused by the end

Simplify each concept to the point of incorrectness to explain the other one
    People don't like being lied to

Solution

Explain A & B correctly by superficially

Iterate your explanations with more detail in each iteration

Patterns for Classroom Education, Dana Anthony, pp. 391-406, Pattern Languages of Program Design 2, Addison Wesley, 1996

Program to an interface, not an implementation

Use abstract classes (and/or interfaces in Java) to define common interfaces for a set of classes

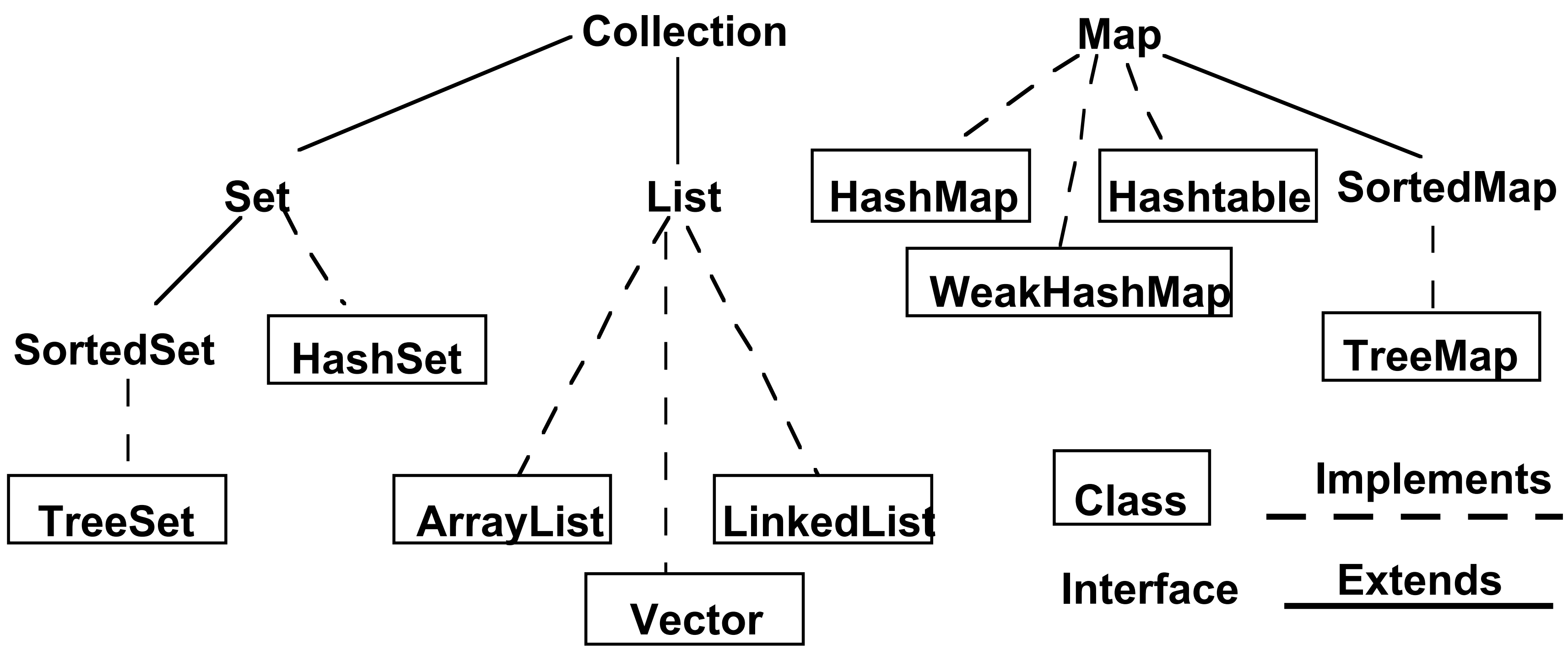
Declare variables to be instances of the abstract class not instances of particular classes

Benefits of programming to an interface

Client classes/objects remain unaware of the classes of objects they use, as long as the objects adhere to the interface the client expects

Client classes/objects remain unaware of the classes that implement these objects.

Clients only know about the abstract classes (or interfaces) that define the interface.



Collection students = new XXX;
students.add(aStudent);

students can be any collection type

We can change our mind on what type to use

In dynamically typed languages programming to an interface is the norm

Dynamically typed languages tend to lack a way to declare an interface

Favor object composition over class inheritance

Composition

- Allows behavior changes at run time
- Helps keep classes encapsulated and focused on one task
- Reduce implementation dependencies

Inheritance

```
class A {
	Foo x
	public int complexOperation() { blah }
}

class B extends A {
	public void bar() { blah}
}
```

Composition

```
class B {
	A myA;
	public int complexOperation() {
		return myA.complexOperation()
	}

	public void bar() { blah}
}
```

Creating an object by specifying a class explicitly

- Abstract factory, Factory Method, Prototype

Dependence on hardware and software platforms

- Abstract factory, Bridge

Dependence on object representations or implementations

- Abstract factory, Bridge, Memento, Proxy

Algorithmic dependencies

- Builder, Iterator, Strategy, Template Method, Visitor

Tight Coupling

- Abstract factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer

Extending functionality by subclassing

- Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy

Dependence on specific operations

- Chain of Responsibility, Command

Inability to alter classes conveniently

- Adapter, Decorator, Visitor

One and only once

In a program written in good style, everything is said once and only once

Methods with the same logic
Objects with same methods
Systems with similar objects

rule is not satisfied

"Good code invariably has small methods and small objects"

Small pieces are needed to satisfy "once and only once"

Make sure you communicate the big picture or you get a mess

Don't put two rates of change together

An object should not have a field that changes every second & a field that change once a month

A collection should not have some elements that are added/removed every second and some that are add/removed once a month

An object should not have code that has to change for each piece of hardware and code that has to change for each operating system

Good style leads to easily replaceable objects

"When you can extend a system solely by adding new objects without modifying any existing objects, then you have a system that is flexible and cheap to maintain"

"Another property of systems with good style is that their objects can be easily moved to new contexts"