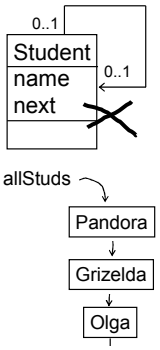
Advanced OO

- Collections
- Java generics (etc)
- OO wisdom
- Design patterns

Java Collections

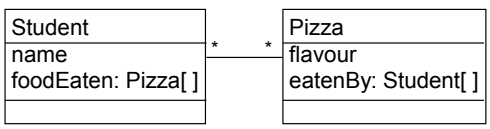
- Collections:
  - Contain a bunch of objects.
  - Implement relationships.
  - Support iteration.
  - Are an essential part of OO libraries.
  - Supercede "data structures" in pre-OO design.
  - Got fixed up in Java 2.

Immoral relationships



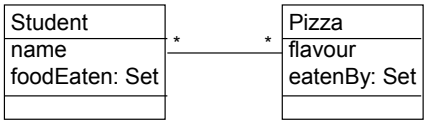

```
public Student getAllKlingons()
{
   Student klingons;
   for (Student s = allStuds; s != null; s = s.next)
   {
      if (!s.isKlingon())
      {
         s.next = klingons; // Eeeek!
         klingons = s;
      }
   }
   return klingons;
}
```

Using arrays for relationships



- Array limitations:
  - Fixed size (wastes space, complicates code…)
  - Allows duplicate entries.
  - Enforces ordering.
  - Rigid indexing (0..n).
  - Awkward to remove items.
  - Linear searching.

Better

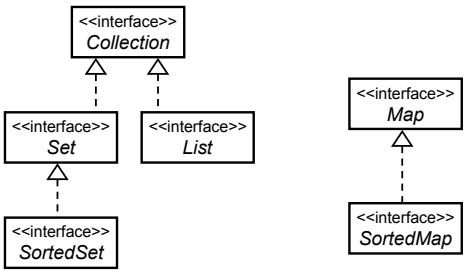



```
public void eat(Pizza: food)
{
   foodEaten.add(food);
   food.addEatenBy(this);
}
```

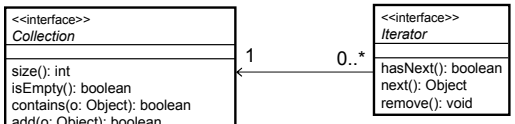
```
Set allStudents;
Set klingons;
```

Java collection Interfaces

import java.util.*;



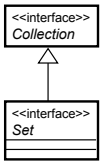
Collection & Iterator




```
Iterator foolter = foos.iterator();
while (foolter.hasNext()) {
   Foo foo = (Foo) foolter.next();
   foo.twiddle();
   ...
}
```

```
static void filter(Collection c) {
    Iterator i = c.iterator()
    while (i.hasNext()) {
        Wibble w = (Wibble) i.next();
        if (!cond(w))
            i.remove();
    }
```

Set: No duplicates




```
import java.util.*;

public class FindDups {
   public static void main(String args[]) {
      Set s = new HashSet();
      for (int i=0; i<args.length; i++)
         if (!s.add(args[i]))
            System.out.println("Duplicate detected: "+ args[i]);

         System.out.println(s.size()+" distinct words detected: "+s);
      }
}
```

Set semantics

- s1.containsAll(s2) -- s2 is a subset
- s1.addAll(s2) -- union
- s1.retainAll(s2) -- intersection
- s1.removeAll(s2) -- difference

Idiom: Given any collection c, make a copy with no duplicates:

```
Collection noDups = new HashSet(c);
```

When are 2 objects the same?

```
public void foo(Set mySet) {
   mySet.add("blarg"); // Works
   mySet.add("blarg"); // Doesn't work.
}
```

How?

```
public boolean add(Object o) {
   …
   if (oldObject.equals(o))
      return false;
   …
}
```

equals() & hashCode()

```
public class Person {
   private String name;

   public boolean equals(Object o) {
      if (o instanceof Person)  {
         Person p = (Person) o;
         return name.equals(p.name);
      }
      return false;
   }
}
```

```
public int hashCode() {
   return name.hashCode();
}
```

If you override equals(), you must also override hashCode().

Equals & inheritance

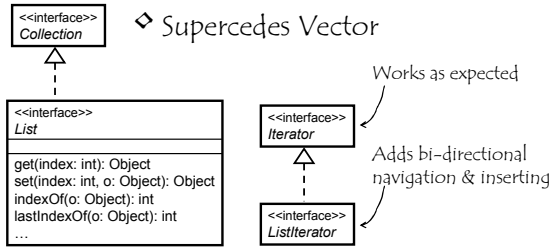
```
public class Ghost extends Person {
   private int transparency;

   public boolean equals(Object o) {
      if (o instanceof Ghost )  {
         Ghost g = (Ghost ) o;
         return name.equals(g.getName()) &&
            transparency == g. transparency;
      }
      return false;
   }
}
```

```
Person nick = new Person("Nick");
Person nhNick = new Ghost("Nick", 50);
System.out.println(nick.equals(nhNick));
System.out.println(nhNick.equals(nick));
```

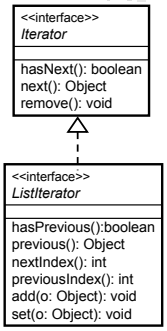
```
public boolean equals(Object o) {
   if (o == this) return true;
   if (o == null) return false;
   if (getClass() != o.getClass()) return false;

   Foo that = (Foo) o;
   return <Do the real comparisons here.>
}
```

List: ordered, duplicates allowed

- add() & addAll() *append*
- remove(o) takes out first match
- Supercedes Vector



ListIterator




```
public boolean swapLast(List l, Object oldO, Object newO) {

   ListIterator i = l.listIterator(l.size());
   while (i.hasPrevious()) {
      if (i.previous() == oldO) {
         i.set(newO);
         return true;
      }
   }

   return false;
}
```

```
int index = l.lastIndexOf(oldO);
if (index == -1)
   return false;
l.set(index, newO);
return true;
```

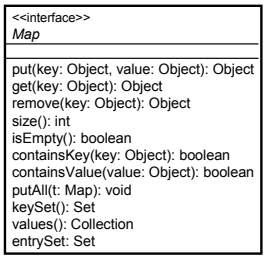
ConcurrentModificationException

```
Iterator foolter = foos.iterator();
while (foolter.hasNext()) {
   Foo foo = (Foo) foolter.next();
   foo.twiddle();
   ...
}
```

```
public void twiddle() {
   foos.add(new Foo());
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException
   at java.util.AbstractList$Itr.checkForComodification(Unknown Source
   at java.util.AbstractList$Itr.next(Unknown Source)
   at Foo.main(Foo.java:20)
```

Map: indexed (uniquely)



- Supercedes Hashtable

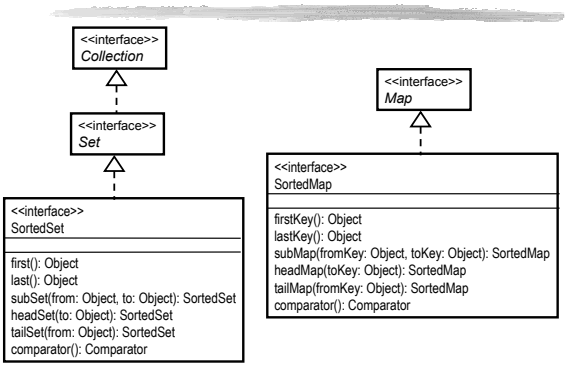
```
public class Frequency {
   private static final Integer ONE = new Integer(1);
   public static void main(String args[]) {
      Map m = new HashMap();
      for (int i=0; i<args.length; i++) {
         Integer freq = (Integer) m.get(args[i]);
         m.put(args[i], (freq==null ? ONE :
               new Integer(freq.intValue() + 1)));
      }
      System.out.println(m.size()+" distinct words:");
      System.out.println(m);
   }
}
```

Iterating over a Map

```
public void grokNicknames(Map nicknames) {

   Iterator keyIter = nicknames.keySet().iterator();
   while (keyIter.hasNext()) {
      String nickname = (String) keyIter.next();
      ...
   }

   Iterator valueIter = nicknames.values().iterator();
   while (valueIter.hasNext()) {
      Dude dude = (Dude) valueIter.next()
      ...
   }
```

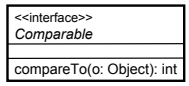
```
Iterator entryIter = nicknames.entrySet().iterator()
   while (entryIter.hasNext()) {
      Map.Entry entry = (Map.Entry) entryIter.next();
      String nickname = (String) entry.getKey();
      Dude dude = (Dude) entry.getValue();
      ...
   }
}
```

Sorted collections



Natural order

```
class Snork implements Comparable {
   String name;
   int IQ;

   Snork(String theName, int theIQ) {
      name = theName;
      IQ = theIQ;
   }
}
```

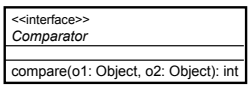



```
public int compareTo(Object o) {
   assert o instanceof Snork;
   Snork that = (Snork) o;
   int res = name.compareTo(that.getName());
   if (res != 0)
      return res;

   if (IQ < that.getIQ())
      return -1;

   if (IQ > that.getIQ())
      return 1;

   return 0;
}
```

```
Set snorks = new TreeSet();
snorks.add(new Snork("Grizelda", 6));
```

```
Exception in thread "main" java.lang.ClassCastException
   at java.util.TreeMap.compare(Unknown Source)
   at java.util.TreeMap.put(Unknown Source)
   at java.util.TreeSet.add(Unknown Source)
   at Snork.main(Snork.java:14)
```

Alternate orders

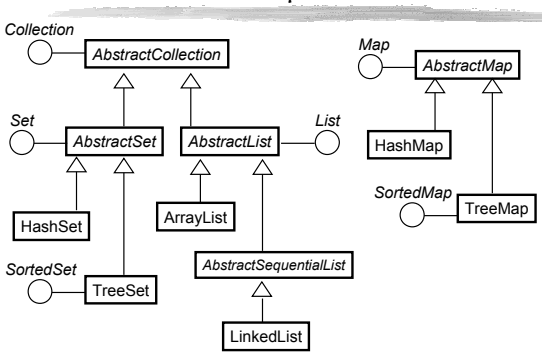



```
SortedSet snorksByIQ = new TreeSet(
   new Comparator() {
      public int compare(Object o1, Object o2) {
         assert o1 instanceof Snork;
         assert o2 instanceof Snork;
         Snork snork1 = (Snork) o1;
         Snork snork2 = (Snork) o2;
         if (snork1.getIQ() < snork2.getIQ())
            return -1;
         if (snork1.getIQ() > snork2.getIQ())
            return 1;
         return 0;
      }
   }
);
```

Implementation matrix

	Set	*List*	*Map*
Hash table	HashSet		HashMap
Array		ArrayList	
Tree	TreeSet		TreeMap
Linked list		LinkedList	

Java collection Implementations



Algorithms

Collections

$ binarySearch(list: List, key: Object): int
$ binarySearch(list: List, key: Object, c: Comparator): int
$ indexOfSubList(source: List, target: List): int
$ max(coll: Collection): Object
$ max(coll: Collection, c: Comparator): Object
$ min(coll: Collection): Object
$ min(coll: Collection , c: Comparator): Object
$ replaceAll(list: List, oldVal: Object, newVal: Object): boolean
$ reverse(list: List): void
$ sort(list: List): void
$ sort(list: List , c: Comparator): void
$ unmodifiableSet(s: Set): Set
$ unmodifiableList(list: List): List
$ unmodifiableMap(m: Map): Map
…

To me, collection classes are one of the most powerful tools for raw programming. You might have gathered that I'm somewhat disappointed in the collections provided in Java through version 1.1. As a result, it's a tremendous pleasure to see that collections were given proper attention in Java 2, and thoroughly redesigned (by Joshua Bloch at Sun). I consider the collections library to be one of the two major features in Java 2 (the other is the Swing library) because they significantly increase your programming muscle and help bring Java in line with more mature programming systems. -- Bruce Eckel.

http://www.cosc.canterbury.ac.nz/web_software/docs/java_docs/guide/collections/overview.html

Collections design goals

Design Goals

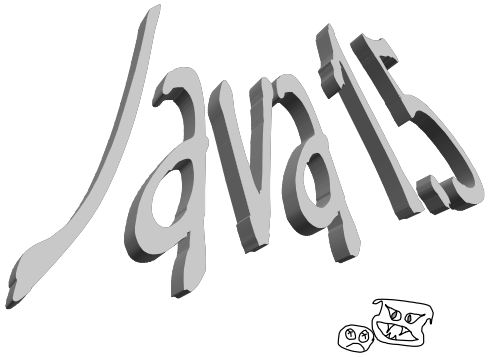
The main design goal was to produce an API that was reasonably small, both in size, and, more importantly, in "conceptual weight." It was critical that the new functionality not seem alien to current Java programmers; it had to augment current facilities, rather than replacing them. At the same time, the new API had to be powerful enough to provide all the advantages described above.

To keep the number of core interfaces small, the interfaces do not attempt to capture such subtle distinctions as mutability, modifiability, resizability. Instead, certain calls in the core interfaces are *optional*, allowing implementations to throw an UnsupportedOperationException to indicate that they do not support a specified optional operation. Of course, collection implementers must clearly document which optional operations are supported by an implementation.

To keep the number of methods in each core interface small, an interface contains a method only if either:

1. It is a truly *fundamental operation*: a basic operations in terms of which others could be reasonably defined,
2. There is a compelling performance reason why an important implementation would want to override it.

It was critical that all reasonable representations of collections interoperate well. This included arrays, which cannot be made to implement the Collection interface directly without changing the language. Thus, the framework includes methods to allow collections to be dumped into arrays, arrays to be viewed as collections, and maps to be viewed as collections.




```
for (Monster monster: monsters)
    monster.bite(victim);
```

Java 1.5

- Biggest ever change to Java:
  - Generics.
  - Enhanced for loop.
  - Autoboxing.
  - Varargs.
  - Printf.
  - Typesafe enums.
  - Annotations.
  - Static imports.

Generics

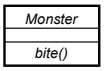
```
List monsters = new ArrayList();
monsters.add(new Vampire("Dracula"));
monsters.add(new Gorgon("Medusa"));
monsters.add(new Lecturer("Neville"));
...
...
...
Monster myMonster = (Monster) monsters .get(1);
```

Parameterized type

Compile-time type safety

```
List<Monster> monsters = new ArrayList<Monster>();
monsters.add(new Vampire("Dracula"));
monsters.add(new Gorgon("Medusa"));
monsters.add(new Lecturer("Neville"));
...
...
...
Monster myMonster = monsters .get(1);
```

Generic iterators




```
public void attack(Set<Monster> monsters, Person victim) {
   Iterator monsterIter = monsters.iterator();
   while (monsterIter.hasNext()) {
      Monster monster = (Monster) monsterIter.next();
      monster.bite(victim);
   }
}
```

```
public void attack(Set<monster> monsters, Person victim) {
   Iterator<Monster> monsterIter = monsters.iterator();
   while (monsterIter.hasNext()) {
      Monster monster = monsterIter.next();
      monster.bite(victim);
   }
}
```

```
public void attack(Set<monster> monsters, Person victim) {
   Iterator<Monster> monsterIter = monsters.iterator();
   while (monsterIter.hasNext())
      monsterIter.next().bite(victim);
}
```

Enhanced for

```
public void attack(Set<Monster> monsters, Person victim) {
   Iterator<Monster> monsterIter = monsters.iterator();
   while (monsterIter.hasNext())
      monsterIter.next().bite(victim);
}
```

```
public void attack(Set<Monster> monsters, Person victim) {
   for (Monster monster: monsters)
      monster.bite(victim);
}
```

```
int[ ] a = { 1, 2, 3 };
for (int i: a) {
   System.out.println(i);
}
```

Autoboxing

box

```
int i = 42;
Integer io = new Integer(256);
i = io.intValue();
io = new Integer(i);
```

unbox

autobox

```
int i = 42;
Integer io = 256;
i = io;
io = i;
```

autounbox

box

```
Set numbers = new HashSet();
numbers.add(new Integer(3));
...
Iterator numIter = numbers.iterator();
while (numIter.hasNext()) {
   Integer num = (Integer) numIter.next();
   int i = num.intValue();
   System.out.println(i);
}
```

unbox

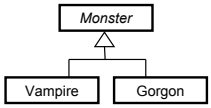
```
Set<Integer> numbers = new HashSet<Integer> ();
numbers.add(3);
...
for (int i: numbers) {
   System.out.println(i);
}
```

Map example revisited

```
public class Frequency {
   private static final Integer ONE = new Integer(1);
   public static void main(String args[]) {
      Map m = new HashMap();
      for (int i=0; i<args.length; i++) {
         Integer freq = (Integer) m.get(args[i]);
         m.put(args[i], (freq==null ? ONE :
               new Integer(freq.intValue() + 1)));
      }
   }
}
```

```
public class Frequency {
      public static void main(String args[]) {
      Map<String, Integer> m = new HashMap<String, Integer>();
      for (String word : args) {
         Integer freq = m.get(word);
         m.put(word, (freq == null ? 1 : freq + 1));
      }
   }
}
```

Generics & subtyping




```
List<Vampire> vampires = new ArrayList<Vampire>();
List<Monster> monsters = vampires;
...
monsters.add(new Gorgon("Medusa"));
...
Vampire myVampire = vampires.get(1);
```

Generics & subtyping

```
public void attack(Set<Monster> monsters, Person victim) {
   for (Monster monster: monsters)
      monster.bite(victim);
}
```

bounded wildcard

```
public void attack(Set<? extends Monster> monsters, Person victim) {
   for (Monster monster: monsters)
      monster.bite(victim);
}
```

```
Set<Vampire> vampires = new ArrayList<Vampire>();
...
attack(vampires, roger);
```

wildcard

```
public void print(Collection<?> stuff) {
   for (Object o: stuff)
      System.out.println(o);
}
```

Declaring generics

```
public interface List {
   void add(Object x);
   Iterator iterator();
}

public interface Iterator {
   Object next();
   boolean hasNext();
}
```

```
public interface List<E> {
   void add(E x);
   Iterator<E> iterator();
}

public interface Iterator<E> {
   E next();
   boolean hasNext();
}
```

Generic methods

```
public void fromArrayToCollection(Object[ ] a, Collection<?> c) {
   for (Object o : a) {
      c.add(o); // compile time error
   }
}
```

```
public <T> void fromArrayToCollection(T[ ] a, Collection<T> c) {
   for (T o : a) {
      c.add(o); // correct
   }
}
```

Mixing old & new

```
public void attack(Set<? extends Monster> monsters, Person victim) {
   for (Monster monster: monsters)
      monster.bite(victim);
}
```

```
Set oldSet = new HashSet();
oldSet.add(new Vampire("Tinky Winky"));
attack(oldSet, Po);
```

> Note: Monster.java uses unchecked or unsafe operations

javac –source 1.4 Monster.java

Varargs

```
public void printGroup(int id, String[ ] members) {
   System.out.println("Group " + id + " contains: ");
   for(String s: members)
      System.out.println(s);
}
```

```
String[ ] group1 = { "Pooh", "Tigger", "Eeyore" };
printGroup(1, group1);
```

```
public void printGroup(int id, String… members) {
   System.out.println("Group " + id + " contains: ");
   for(String s: members)
      System.out.println(s);
}
```

Both work!

```
printGroup(1, "Pooh", "Tigger", "Eeyore" );
```

The march of progress

- 1980: C
  ```
  printf("%10.2f", x);
  ```
- 1988: C++
  ```
  cout << setw(10) << setprecision(2) << showpoint << x;
  ```
- 1996: Java
  ```
  java.text.NumberFormat formatter = java.text.NumberFormat.getNumberInstance(
  formatter.setMinimumFractionDigits(2);
  formatter.setMaximumFractionDigits(2);
  String s = formatter.format(x);
  for (int i = s.length(); i < 10; i++)
    System.out.print(' ');
  System.out.print(s);
  ```
- 2004: Java
  ```
  System.out.printf("%10.2f", x);
  ```

Printf

```
int x = 5;
int y = 6;
int sum = x + y;

System.out.println(x + " + " + y + " = " + sum);
System.out.printf("%d + %d = %d\n", x, y, sum);
```

```
public PrintStream printf(String format, Object… args)
```

Enums (old)

```
public class Size {
   public static final int SMALL = 1;
   public static final int MEDIUM = 2;
   public static final int LARGE = 3;
}
```

```
int howBig = Size.MEDIUM;
```

```
public class Size {
   private String name;
   public static final Size SMALL = new Size("SMALL");
   public static final Size MEDIUM = new Size("MEDIUM");
   public static final Size LARGE = new Size("LARGE");

   private Size(String name) {
      this.name = name;
   }
   public String toString() {
      return name;
   }
}
```

```
Size howBig = Size.MEDIUM;
```

Enums (new)

```
public enum Size { SMALL, MEDIUM, LARGE }
```

```
Size howBig = Size.MEDIUM;

for (Size s: Size.values())
   System.out.println(s);
```

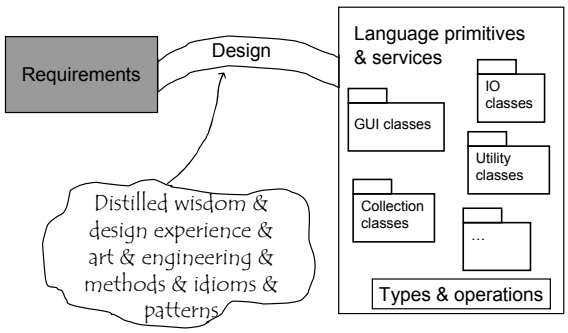
```
public enum Size {
   SMALL(10),
   MEDIUM(14),
   LARGE(18);

   private int diameter;

   public Size(int theDiameter) {
      diameter = theDiameter;
   }

   public void printDiameter() {
      System.out.println(diameter);
   }
}
```

```
for (Size s: Size.values())
   s.printDiameter();
```

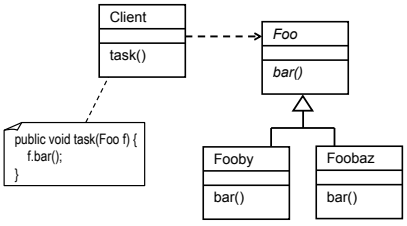
OO wisdom

- Fundamentals & principles.
- Data & behaviour – yin & yang.
- Inheritance.
- Design by contract.

Bridging the gap



Behold!

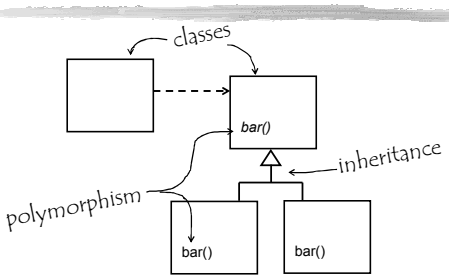


Fundamentals

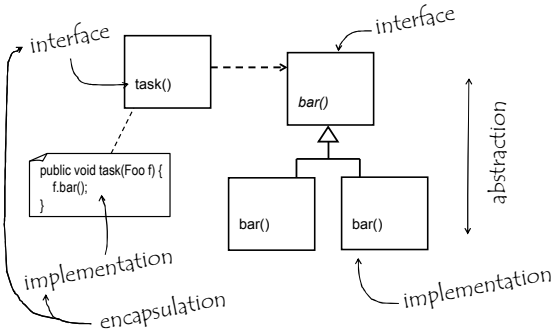
- We have *only one* problem: **complexity**
- We have *only one* solution: **decomposition**

Complexity	Decomposition
size	break up
number of parts	hide parts
connectedness	decouple
change	abstract

OO mechanisms



OO semantics



Decomposing complexity

Principle: separation of concerns

Principle: keep related data and behaviour together

Hiding

Principle: information hiding

Student
dob: Date
getAge():int getAgeOn(Date):int

Encapsulation means drawing a boundary around something. It means being able to talk about the inside and the outside of it.

Information hiding is the idea that a design decision should be hidden from the rest of the system to prevent unintended coupling.

Encapsulation is a programming language feature. Information hiding is a design principle. Information hiding should inform the way you encapsulate things, but of course it doesn't have to. They aren't the same thing. – Ward's wiki

Encapsulation leak

```
public class Student {
    private Set enrolments;

    public Set getEnrolments() {
        return enrolments;
    }
}
```

```
public class Course {
    private List enrolments;

    public void add(Student s) {
        Enrolment e = new Enrolment( s, this);
        enrolments.add(e);
        s.getEnrolments().add(e);
    }
}
```

```
public Set getEnrolments() {
    return Collections.unmodifiableSet(enrolments);
}
```

```
public Student getNthStudent(int n) {
    List enrolments = course.getEnrolments();
    Enrolment enrolment = (Enrolment) enrolments.get(n);
    return enrolment.getStudent();
}
```

Decoupling

Principle: cohesion good, coupling bad

data and behaviour together

separation of concerns
information hiding

Principle: program to the interface, not the implementation

Coping with change

- Find the solid bits.
  - Make them the framework of your program.
- Find the wobbly bits.
  - Hide them away.

Principle: encapsulate that which varies

Solid: the problem domain.
Wobbly: your brain.

Principle: hide your decisions

Hide design decisions

Information hiding

- If you *chose* it, you should hide it.
  - Data representations
  - Algorithms
  - IO formats
  - Mechanisms (garbage, scheduling, persistence…)
  - Lower-level interfaces
  - …

StudentDB

```
public static void setFinal(String ID) {
   Row[ ] rows = query("SELECT FROM Students WHERE id = " + id);
   for (int i; i < rows.length; i++)
   ..
}
```

Student

```
public void setFinal() {
   final = true;
}
```

Coping with change

- Find the solid bits.
  - Abstract, high-level concepts.
- Find the wobbly bits.
  - Concrete, low-level details.

Principle: make stable abstractions

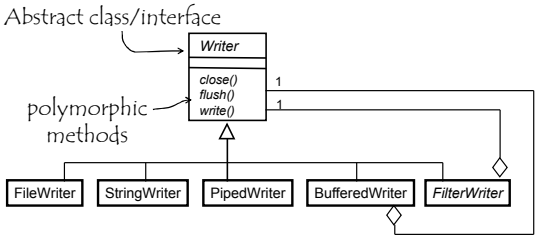
The open/closed principle

Principle: make your system open for extension but closed for modification

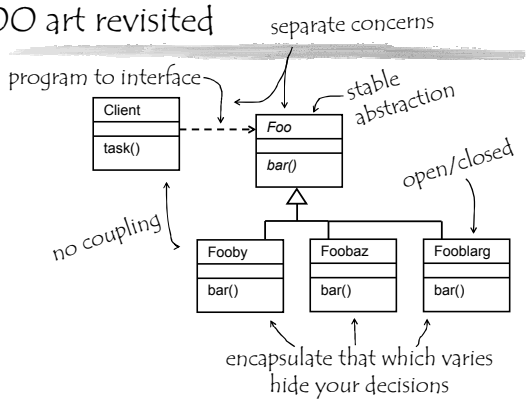
The open-closed principle

"programming by difference"

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."
-- Bertrand Meyer, 1988.

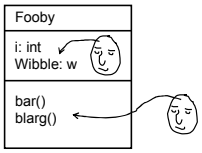


OO art revisited

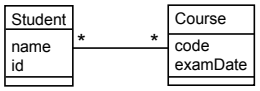


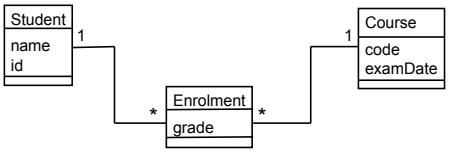
The yin and yang of OO design

- *OO is* data modelling
  - Focus on data internal to object.
- *OO is* behaviour modelling
  - Focus on services to external world.

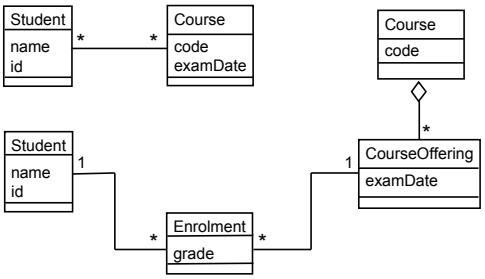


OO is Data Modelling

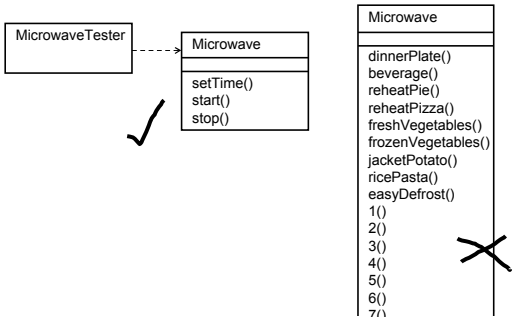




OO is Data Modelling



OO is behaviour modelling



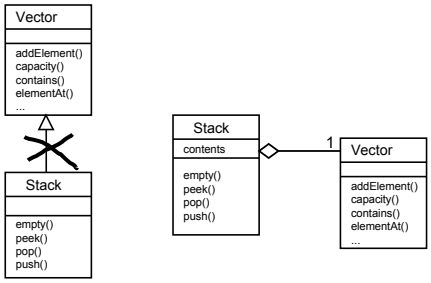
Inheritance: The Dark Side

- Inheritance mistakes:
  - Inheritance for implementation.
  - "Is-a-role-of".
  - "Becomes".
  - Over-specialization.
  - Violating the LSP.
  - Changing the superclass contract.

If it can change, it ain't inheritance.

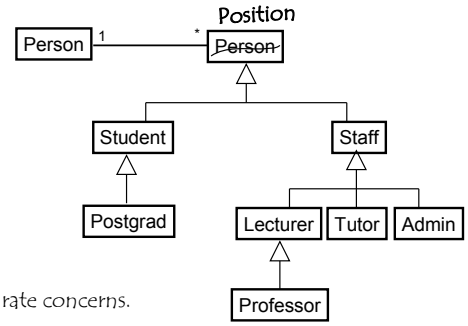
Principle: favour composition over inheritance

Inheritance for Implementation



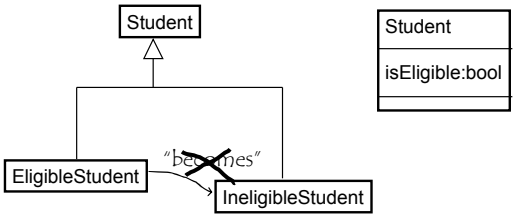
Hide your decisions.

"is-a-role-of"



Separate concerns.

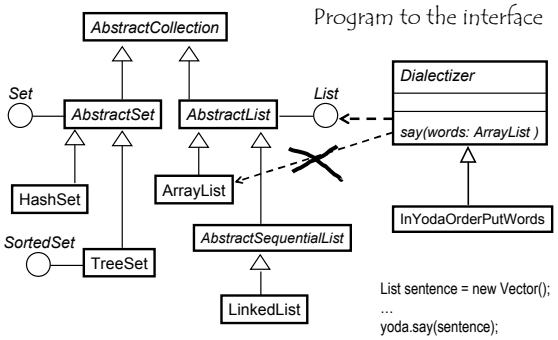
"Becomes"



Inheritance isn't dynamic

Over-specialization

Program to the interface

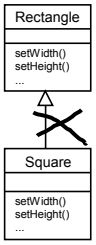



```
List sentence = new Vector();
…
yoda.say(sentence);
```

Violating the Liskov Substitution Principle

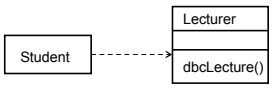
"If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T."
-- Barbara Liskov, 1988

```
private void foo(Rectangle r)
{
   r.setWidth(5);
   r.setHeight(4);
   if (r.getWidth() * r.getHeight() != 20)
      System.out.print("huh?");
}
```



Design by contract (DBC)

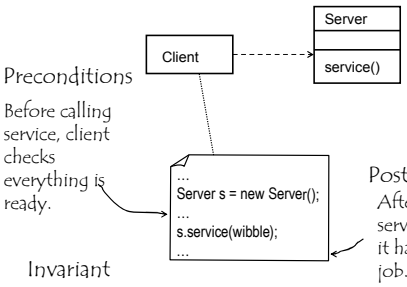
(TM) Bertrand Meyer
Invented by Tony Hoare



- Contract for dbcLecture():
  - When the lecture begins, the *student* will-
    - Be present
    - Know OO
    - Be conscious
    - Not smell too bad
    - …
  - When the lecture ends, the *lecturer* will -
    - Have explained DBC

Design by contract

"Mutual trust"



Preconditions

Before calling service, client checks everything is ready.

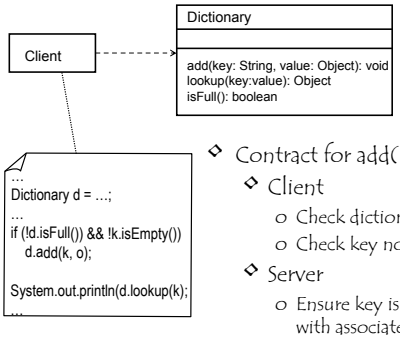
Postconditions

After service, server promises it has done its job.

Invariant

Something the method/class promises will always be true

Design by contract

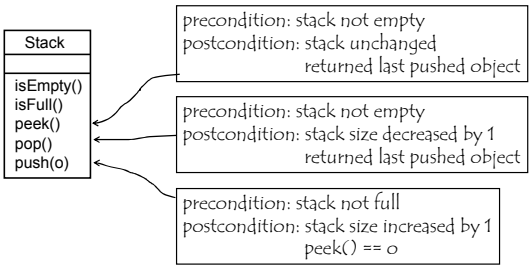


- Contract for add(k, o)
  - Client
    - Check dictionary not full
    - Check key not empty
  - Server
    - Ensure key is in dictionary with associated value.

Preconditions & Postconditions

```
public class Dictionary {
  …

  /** Inserts value into the dictionary at the given key.
    * A later call to lookup(key) will return the value.
    *
    * Preconditions: The dictionary is not full.
    *                The key is not null or an empty string.
    *                (It is OK if the dictionary already has an entry
    *                at this key; the new value replaces the old one.)
    * Postconditions: The dictionary contains the value indexed by the key.
    */
  public void add(String key, Object value) {
```

Stack contract

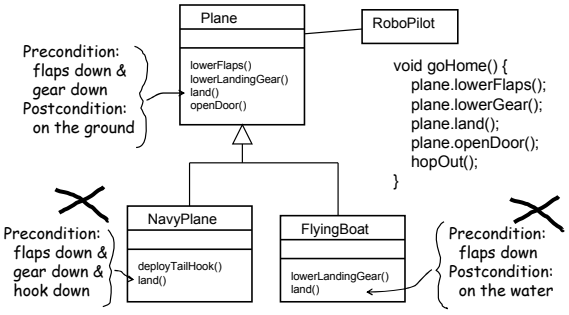


Contract guidelines

Derived from work of Jim Weirich & Todd Plessel (websites)

- No precondition on queries.
  - It should be safe to ask a question.
- No fine print.
  - Don't require something the client can't determine I.e. preconditions should be supported by public methods.
  - (It is OK to have postconditions a client can't verify.)
- Use real code where possible.
  - Better to say "!isEmpty()" rather than "the stack must not be empty". (It's what the client must do.)
- Can't show all semantics in code. (So use english)
  - E.g. pop() returns last pushed object.
- No hidden clauses.
  - The preconditions are sufficient & complete.
- No redundant checking
  - Don't check that preconditions are satisfied inside server!

Contracts & inheritance



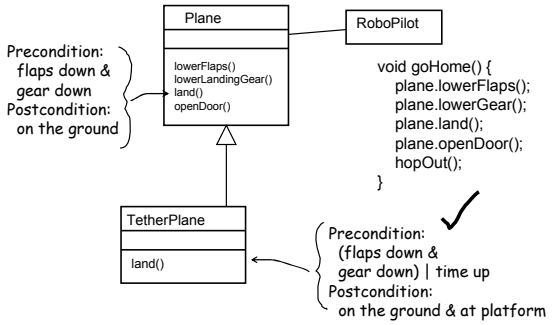
Inheriting a contract

> "The contracts of the ancestors shall be honoured by the descendants, yea even unto the Nth generation."
> -- Paul Johnson

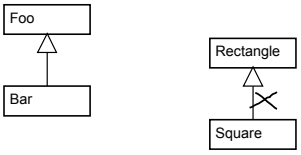
- Contracts are inherited.
  - Preconditions can be loosened
  - Postconditions can be tightened.
  - Invariants can be tightened too.

Principle:require no more, promise no less.

Contracts & inheritance



What is inheritance, really?



- Previously, we said "a Bar **is-a** Foo".
- More precisely, "a Bar **conforms to the contract of** Foo".

Formal support for contracts

- DBC developed by Bertrand Meyer in Eiffel
  - Keywords in method declaration: **require** (precondition) & **ensure** (postcondition).
- Added to UML as part of Object Constraint Language (OCL)
- Strong following in formal methods community.
- Some efforts to support it in Java
  - http://www.cs.iastate.edu/~leavens/JML/
  - http://www.mmsindia.com/DBCForJava.html
  - http://www.javaworld.com/javaworld/jw-02-2002/jw-0215-dbcproxy.html
  - All use tags in comments, e.g. @pre i < 42, or @requires i < 42

Informal support for contracts

- New keyword in Java 1.4:
  - **assert** *expression*;
  - http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html
  - Conditional compilation:
    - javac -source 1.4 Thingy.java
  - Conditional execution
    - java –ea myprog

```
public void push(Object o) {
   assert !isFull(); // throws AssertionError if false.
   …
   assert size() == oldSize + 1;
}
```

From the FAQ

Why not provide a full-fledged design-by-contract facility with preconditions, postconditions and class invariants, like the one in the Eiffel programming language?

We considered providing such a facility, but were unable to [do it] without massive changes to the Java platform libraries… Further, we were not convinced that such a facility would preserve the simplicity that is Java's hallmark. On balance, we came to the conclusion that a simple boolean assertion facility was a fairly straight-forward solution and far less risky. It's worth noting that adding a boolean assertion facility to the language doesn't preclude adding a full-fledged design-by-contract facility at some time in the future.

The simple assertion facility does enable a limited form of design-by-contract style programming. The assert statement is appropriate for postcondition and class invariant checking. Precondition checking should still be performed by checks inside methods that result in particular, documented exceptions, such as IllegalArgumentException and IllegalStateException. [*Debatable!*]

A philosophy for using exceptions

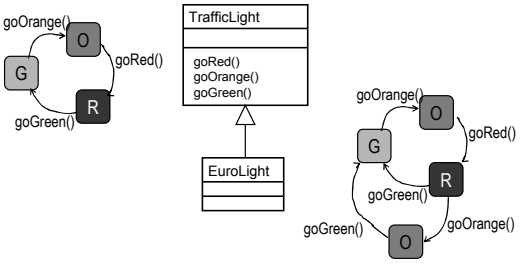
- Use java exceptions iff a contract violation occurs.
- Handling violations
  - If possible, fix the problem, otherwise
  - if possible, try an alternative approach, otherwise
  - clean-up & throw an exception.
- Clean-up: release resources, locks, rollback transactions, set consistent state…
- When an exception is thrown, catch it anywhere clean-up is needed, then try the 3 alternatives above.
- If not handled earlier, catch the exception at the root level where inputs may be changed.
  - In interactive systems, this is usually the *event* level.

A philosophy for using Interfaces

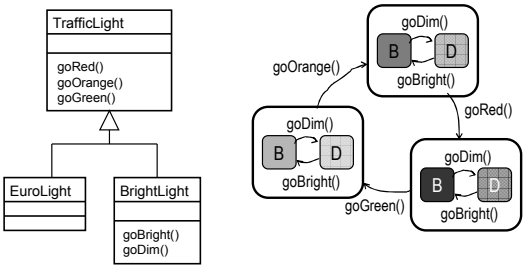
- Interfaces **are** contracts.
  - Whenever a contract can be recognised independent from a particular implementation, an interface should be considered.
  - E.g. Stack
  - Interfaces can be composed; one class may implement many interfaces.
  - Interfaces can be extended to specialise contracts.

Contracts & state machines

- State behaviour is part of the contract.

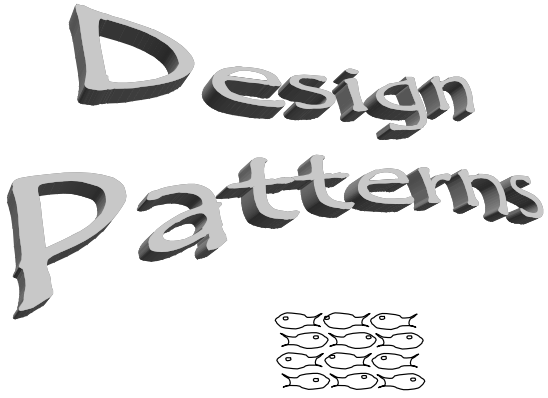


Contracts & state machines



Inheritance: The Dark Side revisited

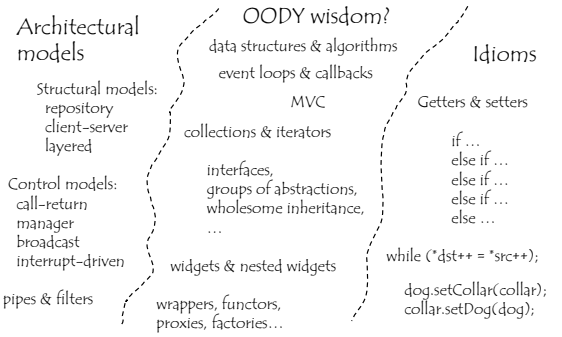
- Inheritance for implementation.
  - No intention to honour the inherited contract
- "Is-a-role-of".
  - Merging of 2 contracts
- "Becomes".
  - Switching contracts.
- Over-specialization.
  - Contract more specific than necessary
- Violating the LSP.
  - Breaking the contract.



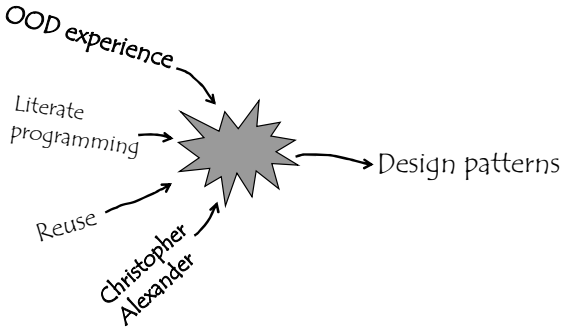
Design Patterns

- Grassroots movement in the OO community, from early 90's.
- Major emerging trend in industry.
- Massively hyped:
  - The biggest advance since OO itself?
  - A paradigm shift?
- Essential vocabulary for software engineers.

OOD experience



Design Patterns Roots

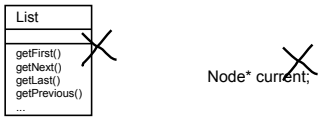


What is a design pattern?

- Distilled wisdom about a specific problem that occurs frequently in OO design.
- A reusable *design* micro-architecture.
- The core of a design pattern is a simple class diagram with extensive explanation.
  - It documents an elegant, widely-accepted way of solving a common OO design problem.
- Patterns are *discovered*, as opposed to written.

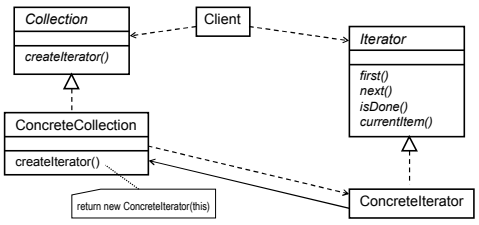
The Iterator pattern

- Name:
  - Iterator (a.k.a. Cursor)
- Problem:
  - Sequentially access the elements of a collection without exposing implementation.
  - Allow for different types of traversals (e.g. different order, filtering).
  - Allow multiple traversals at same time.



Encapsulate that which varies

- Solution:
  - Move responsibility for traversal from the collection into an Iterator object. It knows current position and traversal mechanism.
  - The collection creates an appropriate Iterator.



A definition for "Design Pattern"

"A *solution* to a *problem* in a *context*."

A canonical design form that someone can apply to *resolve* these forces.

A set of *forces* -- goals and constraints -- that occur in this context.

A recurring set of situations in which the pattern applies

-- Patterns-Discussion FAQ

Forces

"Criteria that software engineers use to justify designs and implementations." -- FAQ

- Correctness
  - Completeness, type safety, fault tolerance, security, transactionality, thread safety, robustness…
- Resources
  - Efficiency, space, "on-demand-ness", fairness, equilibrium, stability…
- Structure
  - Modularity, encapsulation, coupling, independence, extensability, reusability, context dependence, interoperability…
- Construction
  - Understandability, minimality, simplicity, elegance, vrror-proneness, co-existence with other software, maintainability, impact on processes, teams, users…
- Usage
  - Ethics, adaptability, human factors, aesthetics, economics…

Resolution of Forces

"A pattern should represent a kind of *equilibrium* of forces."

- Impossible to *prove* a solution is optimal; arguments must be backed up with:
  - Empirical evidence for goodness
    - The rule of 3: don't claim something is a pattern unless you can point to three independent usages.
  - Comparisions
    - With other solutions, (possibly including failed ones).
  - Independent authorship
    - Not written solely by their inventors.
  - Reviews
    - By independent domain and pattern experts.

The GanG of Four (GoF)

- ***The*** design patterns book:
  - *"Design Patterns; Elements of Reusable Object-Oriented Software"*, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1995
  - On 3-hour loan in PSL.
  - Catalog of 23 design patterns, classified as:
    - Creational patterns
    - Structural patterns
    - Behavioral patterns
  - Uses OMT, examples in C++ & Smalltalk

Creational Patterns

Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Builder	Separate the construction of a complex object from its representationso that the same construction process can create different representations
Factory Method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Singleton	Ensure a class only has one instance, and provide a global point of access to it.

Structural Patterns

Adapter	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
Bridge	Decouple an abstraction from its implementation so that the two can vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
Façade	Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.
Proxy	Provide a surrogate or placeholder for another object to control access to it.

Behavioral Patterns…

Chain of Responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulate the request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in that language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

… More Behavioral Patterns

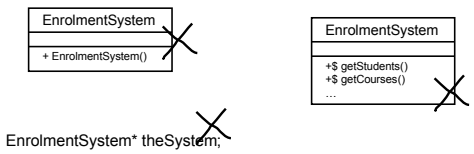
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
State	Allow an object to alter its behavior when internal state changes. The object will appear to change its class.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable, Strategy lets the algorithm vary independently from clients that use it.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Documenting Patterns (GoF style)

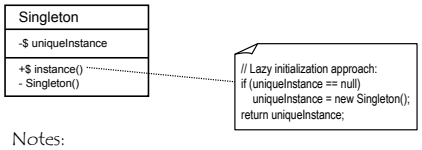
- Name
- Intent
  - Brief synopsis (as on previous slides)
- Motivation
  - The context of the problem
- Applicability
  - Circumstances under which the pattern applies
- Structure
  - Class diagram of solution
- Participants
  - Explanation of the classes/objects and their roles
- Collaborations
  - Explanation of how the classes/objects cooperate
- Consequences
  - Discussion of impact, benefits, & liabilities
- Implementation
  - Discussion of techniques, traps, language dependent issues…
- Sample code
- Known uses
  - Well-known systems already using the pattern
- Related patterns

Singleton

- Problem:
  - Some classes should have only one instance.
    - Eg. EnrolmentSystem, PrintSpooler, FileSystem…
  - How can we ensure someone doesn't construct another one?
  - How should other code find the one instance?



- Solution:
  - Make the constructor private.
  - Use a static attribute in the class to hold the one instance
  - Add a static getter for the instance

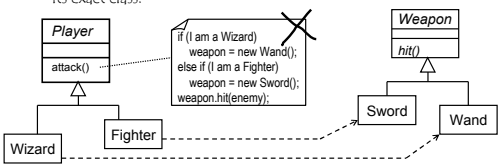


- Notes:
  - Subclassing Singleton is possible (unlike all-static approach), but does require more elaborate initialization of uniqueInstance.
  - java.lang.Runtime is a singleton class.

Factory Method

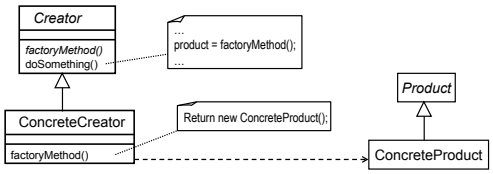
a.k.a. Virtual Constructor

- Problem:
  - Normally, code that expects an object of a particular class does not need to know which subclass the object belongs to.
    - E.g. a Player in an adventure game uses a Weapon, and does not need to know exactly what kind of Weapon it is.
  - Exception: when you create an object, you need to know its exact class. The "new" operator increases coupling!
  - Need a way to create the right kind of object, without knowing its exact class.



loose coupling
Program to the interface

- Solution:
  - Move the "new" into an abstract method. (Can be parameterized.)
  - Override that method to create the right subclass object.



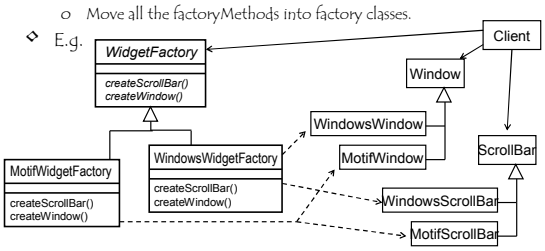
- Notes:
  - It is common to have more than one factory method.
    E.g. weaponFactory(), treasureFactory(), potionFactory()
  - Swing UIManager.getUI(Jcomponent) is a fancy factory method.

Abstract Factory

keep related behaviour together

a.k.a. Kit

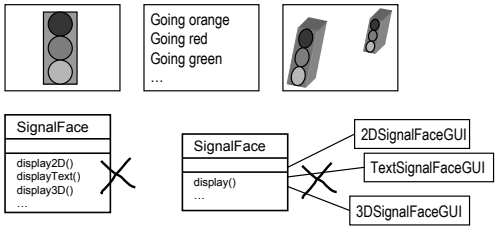
- Problem:
  - Same as factory method, but want to create whole families of related objects.
- Solution
  - Move all the factoryMethods into factory classes.
- E.g.



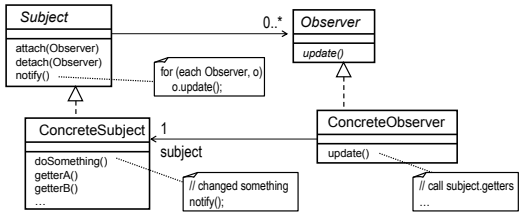
Observer

a.k.a. Dependents
a.k.a. Publish-Subscribe

- Problem:
  - Separate concerns into different classes, but keep them in synch.
    - E.g. separate GUI code from model.
  - Avoid tight coupling.



- Solution:
  - Separate into Subject and Observers.
    - Can have many observers for one subject.
  - The Subject knows which objects are observing it, but it doesn't know anything else about them.
  - When the Subject changes, all Observers are notified.



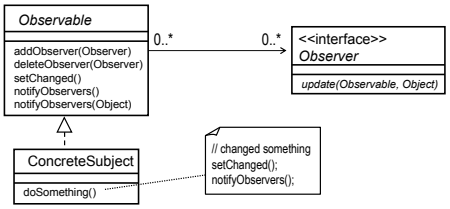
Observer Notes

loose coupling

- Changes are broadcast to all Observers.
  - It is up to each Observer to decide if it cares about a particular change.
- Observers don't know about each other.
  - They are unaware of the true cost of changes to the Subject.
  - Complex dependencies and cycles are possible (& should be avoided).
- Observers aren't told what changed.
  - They usually just get all relevant attributes again.
  - Figuring out what changed can be a lot of work, and might require the Observer to retain a lot of the Subject's state.
  - A variant of the pattern allows the update method to contain details of what changed. (More efficient, but tighter coupling.)
- The Subject should call notify() only when it is in a consistent state (at end of transaction).
  - Beware of subclasses that call base class methods, and the base class does the notify().

Java support for Observer

- Subject is a class called Observable; Observer is an interface
- Allows many Observers to many Subjects
- Adds a "dirty" flag to help avoid notifications at wrong time.

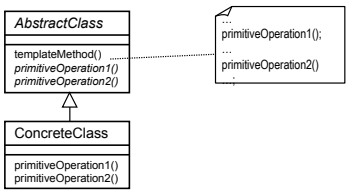


- Swing EventListeners are another variant of Observer

Template Method

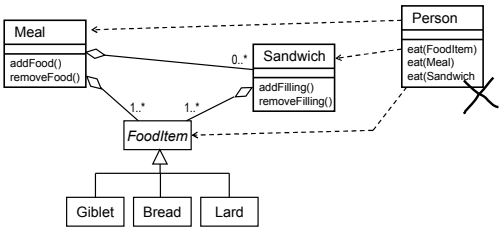
open/closed

- Problem:
  - Implement the skeleton of an algorithm, but not the details.
- Solution
  - Put the skeleton in an abstract superclass and use subclass operations to provide the details.



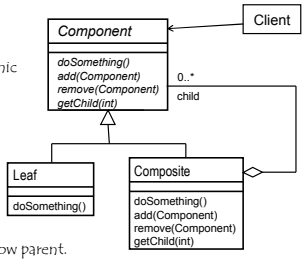
Composite

- Problem:
  - When objects contain other objects to form a tree (i.e. a containment hierarchy), how can client code treat the composite objects and the atomic objects uniformly?



open/closed
stable abstractions?

- Solution:
  - Create an abstract superclass that represents both composite and atomic objects.

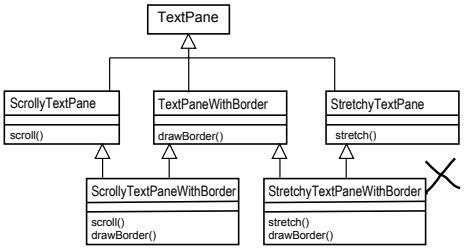


- Notes:
  - Common for child to know parent.
  - Easy to add new components.
  - Can make containment too general.
  - Swing JComponent uses the Composite pattern.

Decorator

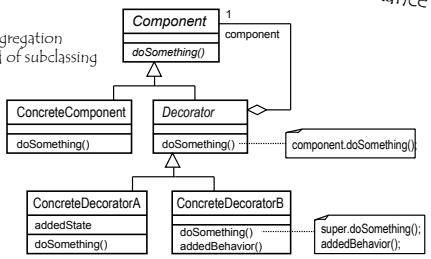
a.k.a. Wrapper

- Problem:
  - Add additional responsibilities to an object dynamically, rather than through inheritance.



open/closed
encapsulate that which varies
favour composition over inheritance

- Solution:
  - Use aggregation instead of subclassing

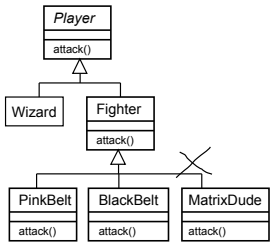


- Notes:
  - Can omit abstract Decorator class.
  - Only works if Component class is lightweight.
  - Swing's JScrollPane is a Decorator.

Strategy

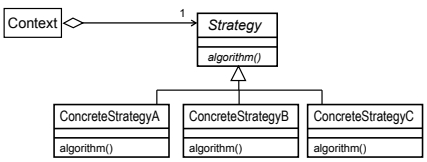
a.k.a. Policy

- Problem:
  - Change an object's algorithm dynamically, rather than through inheritance.



open/closed
encapsulate that which varies
favour composition over inheritance

- Solution:
  - Move the algorithms into their own class hierarchy.



- Notes:
  - Contexts know different strategies exist (because have to choose one).
  - Strategy needs access to relevant context data. (Parameter/reference?)
  - AWT (& Swing) LayoutManager is a Strategy.

Alexandrian patterns

"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."
-- Christopher Alexander, *A Pattern Language*, 77

- Over 250 patterns for buildings, e.g.:
  - Alcoves
  - Intimacy Gradient
  - Pools of Light
  - Common Areas at the Heart
  - …

"It is quite possible that all the patterns for a house, in some form, be present and overlapping in a simple one-room cabin. The patterns do not need to be strung out and kept separate. Every building, every room, every garden is better when all the patterns are compressed as far as it is possible for them to be. The building will be cheaper, and the meanings in it will be denser."
-- Alexander, '77

"Pattern Language"

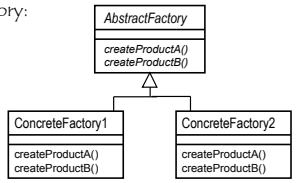
A pattern language is a set of interrelated patterns, all sharing some of the same context, and perhaps classified into categories.
-- Patterns-discussion FAQ

"When related patterns are woven together they form a "language" that provides a process for the orderly resolution of software development problems. Pattern languages are not formal languages, but rather a collection of interrelated patterns, though they do provide a vocabulary for talking about a particular problem."
-- CACM *Special Issue on Patterns and Pattern Languages*, Vol. 39, No. 10, '96

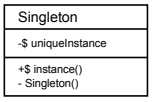
"The means of designing a building using a pattern language is to determine the most general problem to be solved and to select patterns that solve that problem. Each pattern defines subproblems that are similarly solved by other, smaller patterns. Thus we see that the solution to a large problem is a nested set of patterns."
-- Richard Gabriel, *Patterns of Software*, 1995

Abstract Factory meets Singleton…

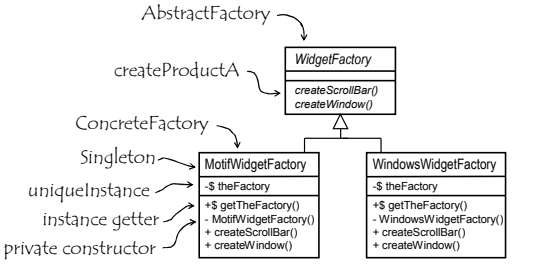
- Abstract Factory:



- Singleton:

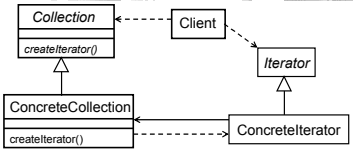


e.g.

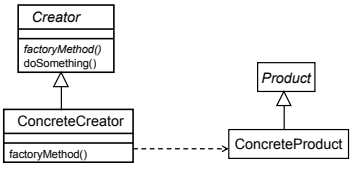


Iterator meets Factory Method…

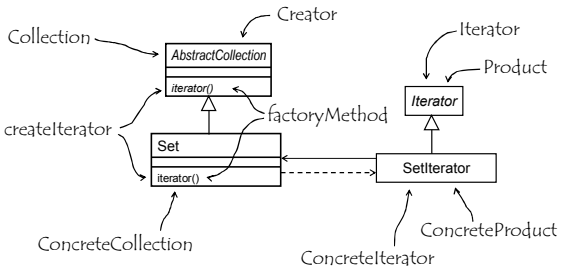
- Iterator:



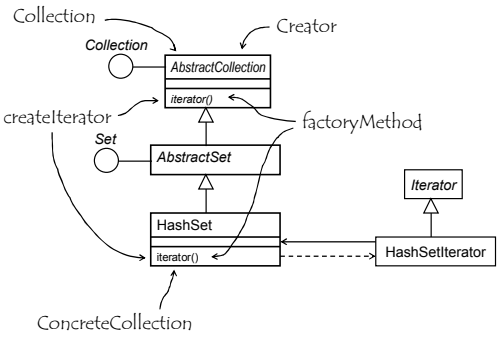
- Factory Method:



e.g. Simplified Java Collections

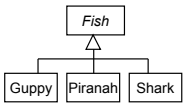


e.g. Real Java Collections

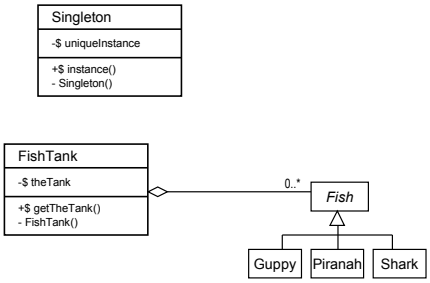


Case study

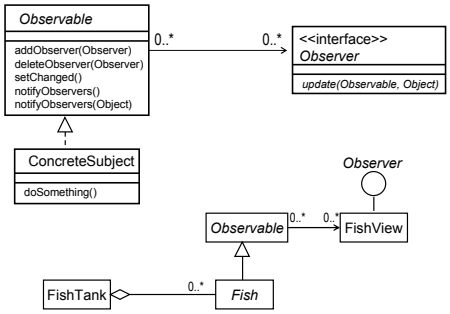
- Simulated fish tank
  - Lots of different kinds of fish
  - Only ever one tank
  - Multiple views possible
  - Schools of fish
  - Remoras and barnacles
  - Fish spawn new fish



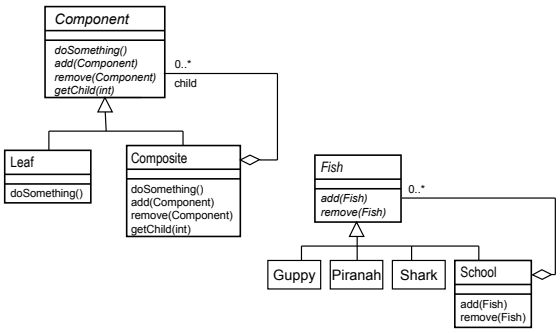
Singleton — Only ever one tank



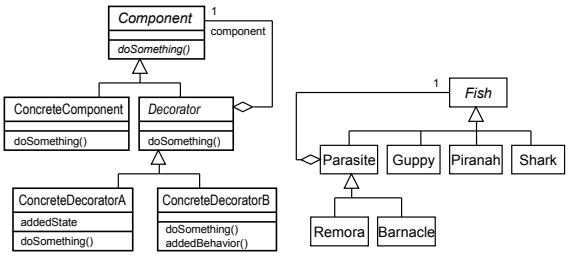
Observer — Multiple views possible



Composite — Schools of fish

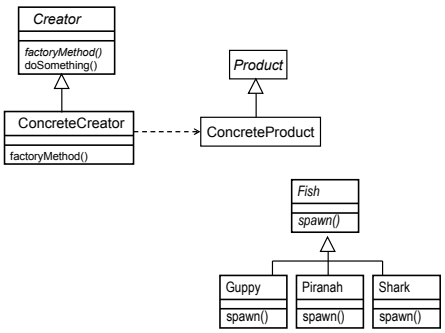


Decorator — Remoras and barnacles

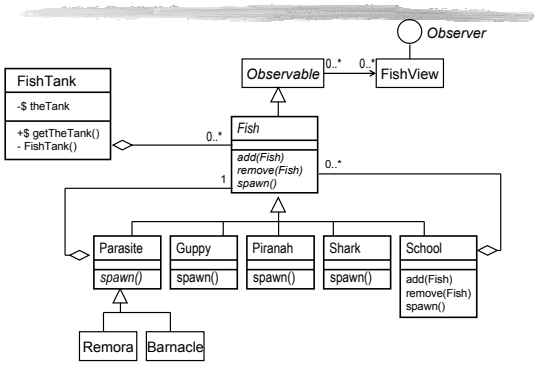


Factory Method

Fish spawn new fish



The result



Patterns: the dark side!

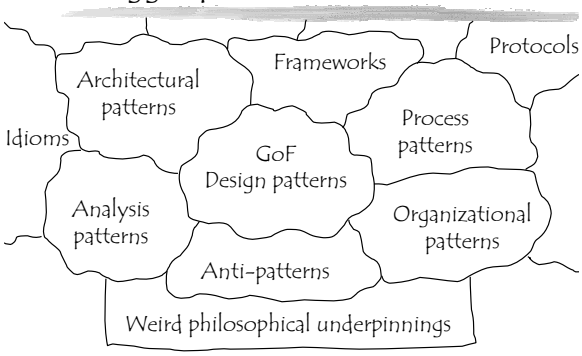
"No discussion of how to use design patterns would be complete without a few words on how *not* to use them. Design patterns should not be applied indiscriminately. Often they achieve flexibility and variability by introducing additional levels of indirection, and that can complicate a design and/or cost you some performance. A design pattern should only be applied when the flexibility it affords is actually needed."
-- GoF.

Is the GoF God?

"Those who ascribe extraordinary powers to the Gang of Four will be appalled by our generally chaotic process of pattern development."
--John Vlissides, *Pattern Hatching*, '98.

"Ordre ex chaos is a theme in the natural sciences, and we shouldn't expect the science of design to be any different. Patterns are about people working together to discover and document constructs that contribute to the quality of life of humanity as a whole. It is a necessarily organic process.*"*
--James Coplien, Foreward to *Pattern Hatching*, '98.

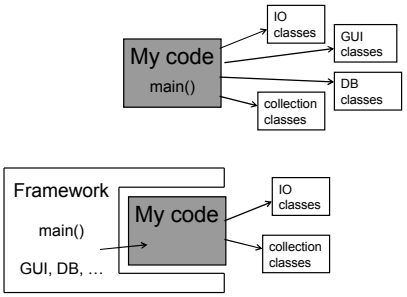
The bigger picture



Patterns, libraries, & frameworks

- NOT implementations:
  - Design patterns
    - Reusable abstract solutions.
- Implementations:
  - Class libraries
    - Reusable software components.
  - Frameworks
    - Reusable, but incomplete, applications

The Hollywood principle



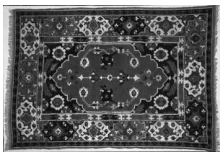
Christopher Alexander

- *Notes on the Synthesis of Form, '64*
- *A City is Not a Tree, '65*
- *A Pattern Language, '77*
- *The Timeless Way of Building, '79*
- *A Foreshadowing of 21st Century Art '93*
- *The Nature of Order, 2002-ish*

"A year or two ago, I was astonished to get several letters from different people in the computer science field, telling me that my name was a household word in the software engineering community: specifically in the field of object-oriented technology. I had never even heard of object-oriented programming, and I had absolutely no idea that computer scientists knew my work, or found it useful or interesting; all this was a revelation to me."
-- Alexander, Foreward to *Patterns of Software*, '96.

"Objective beauty"

Beauty is in the eye of the beholder.



There are few fields that blend art and science:
Architecture is one, and computer science is another.
-- Richard Gabriel.

Alexander would take different carpet designs, or different configurations of colored beads, and then ask observers to tell him which of two designs (or configurations) was more beautiful or pleasing to them and why. Confoundingly enough, it appeared that what we all might think of as being purely "in the eye of the beholder" was in fact less subjective than we believed. Apparently the overwhelming majority of individuals from all various walks of life seemed to converge on what they felt was the most pleasing or beautiful.

Alexander tried to carefully study the different characteristics or "properties" which, when present, seemed to trigger a preference in observers for designs which possessed that property. Certain things like centers, symmetry, and "effective" use of positive and negative space, kept recurring as aspects to which all people seemed aesthetically attracted. Alexander was trying to use the results of these experiments to verify/validate to his belief that beauty, is in fact *objective* (at least at its deepest and most fundamental levels of recognition).

-- James Coplien

QWAN & other "weird" stuff

The Quality Without a Name

Timelessness
Aliveness
Wholeness
Piecemeal growth
Organic order
…

Users should be involved in design.
Designers should be involved in construction.
Constructions should evolve.

The existence of a master plan alienates the users… After all, the very existence of a master plan means, by definition, that the members of the community can have little impact on the future shape of their community, because most of the important decisions have already been made. In a sense, under a master plan people are living with a frozen future, able to affect only relatively trivial details. … people lose the sense of responsibility for the environment they live in, and realize that they are merely cogs in someone else's machine…

Second, neither the users nor the key decision makers can visualize the actual implications of the master plan.
-- Alexander, '75

Excellence

In my life as an architect, I find that the single thing which inhibits young professionals, new students most severely, *is their acceptance of standards that are too low*. If I ask a student whether her design is as good as Chartres, she often smiles tolerantly at me as if to say, "Of course not, that isn't what I am trying to do… I could never do that."

There are programs we can look at and about which we say, "no way I'm maintaining that kluge". And there are other programs about which we can say, "Wow, who wrote this!"…

> Computer scientists who try to write patterns without understanding [the quality without a name] are quite likely not following Alexander's program, and perhaps they are not helping themselves and others as much as they believe. Or perhaps they are doing harm.
>
> -- Richard Gabriel

http://www.natureoforder.com/overview.htm

The Nature of Order

We've all been captivated by this "patterns" stuff in the software community as of late, but for Alexander, this was 20 years ago and his ideas have evolved into bigger and better things." -- Brad Appleton

- Alexander's *magnum opus*
  - 4 volumes:
    - Book 1: The Phenomenon of Life
    - Book2: The Process of Creating Life
    - Book3: A Vision of a Living World
    - Book4: The Luminous Ground

"One of the twentieth century's most important documents."
-- Nikos Salingaros

"This will change the world as effectively as the advent of printing changed the world."
-- Doug Carlston (former pres. of Broderbund)

Here is acclaimed architect Christopher Alexander's four volume masterwork: the result of 27 years of research and a lifetime of profoundly original thinking.

Consider three vital perspectives on our world:
- A scientific perspective
- A perspective based on beauty and grace
- A commonsense perspective based on our intuitions about everyday life

This groundbreaking work allows us to form one picture of the world in which all three perspectives are interlaced. It opens the door to 21st-century science and cosmology.
-- Inside the jacket of *The Nature of Order*

With the publication of *The Nature of Order* and with the mature development of my work in construction and design, the problems that I began to pose 35 years ago are finally being solved. There are immense difficulties, naturally, in implementing this program… But the feasibility of the whole matter and the extent to which it is well-defined can, I think, no longer be in doubt. What is most important is that all this can actually be *done*.
…
I get the impression that the road seems harder to software people than maybe it did to me, that the quality software engineers might want to strive for is more elusive because the artifacts -- the programs, the code -- are more abstract, more intellectual, more souless than the places we live in every day.
-- Alexander

The redemption of hacking?

As it turns out, the type of process described in **The Nature of Order** is very much like what members of the so called "gentlemen hacker" culture (which sprang out of MIT in the 60s) used to build their systems. Examples of such "gentlemen hackers" are/were: Doug Lea, Richard Gabriel, Don Knuth, ... Sadly, most people failed to realize that, although these processes appeared to lack rigorous formality, they did in fact entail a great deal of discipline, integrity, and intellectual rigor that stemmed from pride in artisanship, and in one's craft in general. These profoundly insightful "hackers" looked at the system as "a whole" rather than as the mere sum of its parts. It is unfortunate that this lack of formality was misconstrued as a lack of discipline, transforming the word "hacker" from something to be emulated, into something which is now loathed within software engineering circles that place great emphasis upon process definition and improvement.

-- Jim Coplien

More pattern resources

- PoSA (a.k.a GoV) *Pattern-oriented software architecture*, Frank Buschmann et.al. '96
  - Primarily addresses architectural patterns.
- *Patterns of Software*, Richard Gabriel, 1996
  - Philosophical discussion of Alexandrian and software patterns.
  - May cause brain damage.
- PLoP 1, 2, 3, & 4. *Pattern Languages of Program Design*
  - Shows background working behind patterns.
- *Patterns in Java*, Volume 1 & Volume 2, Mark Grand, 1999
  - Helpful translations of GoF patterns.
  - Beware: extremely general definition of pattern.
- *Head First Design Patterns*, Eric & Elisabeth Freeman
  - GoF patterns, with attitude.
- The Hillside Group (a.k.a. the patterns homepage)
  - www.hillside.net/patterns
- The Portland Pattern Repository
  - http://c2.com/ppr/
- The Patterns-Discussion FAQ
  - http://gee.cs.oswego.edu/dl/pd-FAQ/pd-FAQ.html

A parting thought…

Technology, science, engineering, and company organization are all secondary to the people and human concerns in the endeavor. Companies, ideas, processes, and approaches ultimately fail when humanity is forgotten, ignored, or placed second. Alexander knew this, but his followers in the software pattern language community do not. Computer scientists and developers don't seem to know it either.

-- Richard Gabriel.