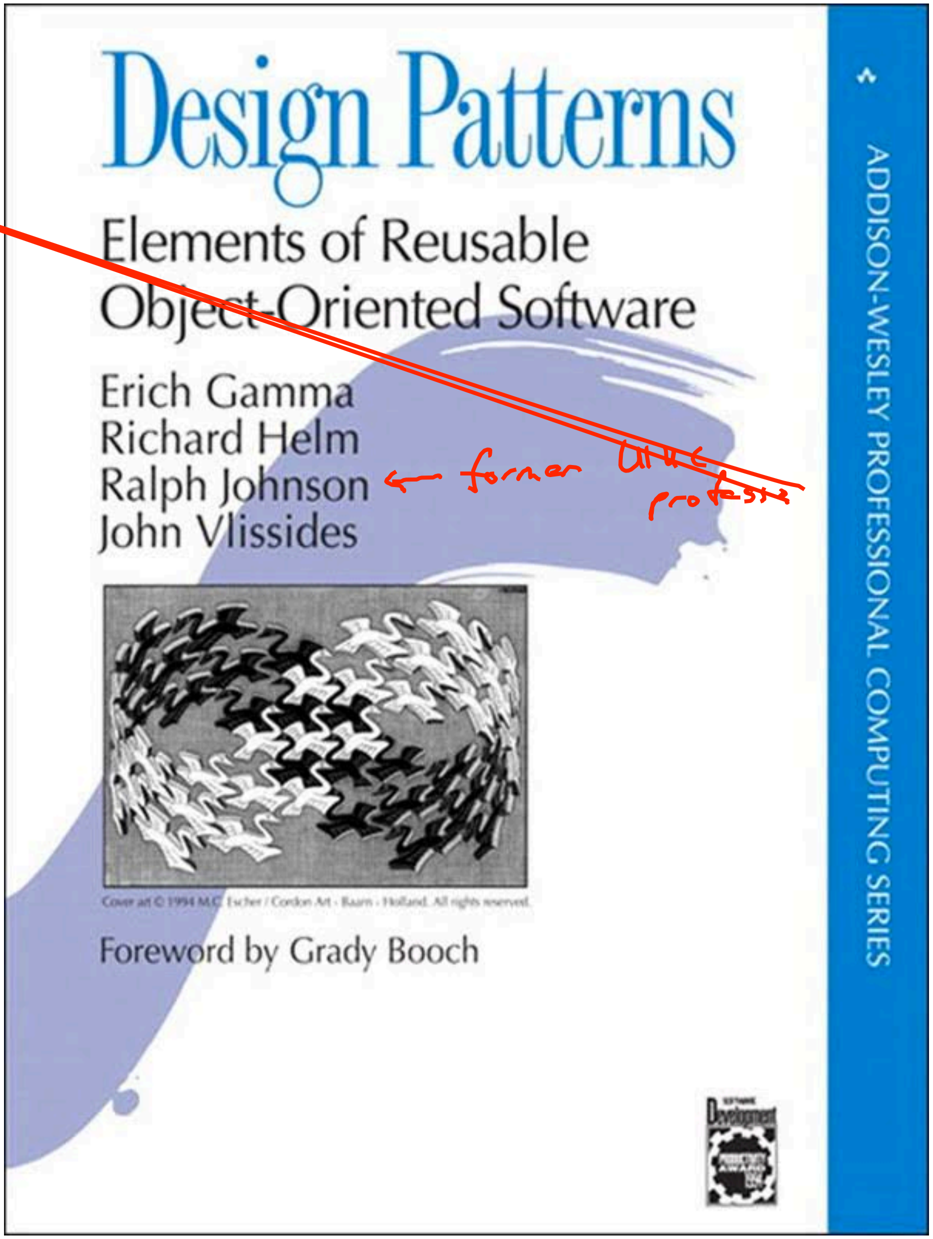
Introduction to Design Patterns



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson ← former UIUC professor
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

How hard was week 6 code review assignment?

A) Easy

B) Moderate

C) Challenging

D) Unreasonable

How long did week 6 assignment take?

A) Less than 2 hours

B) 2 to 4 hours

C) 4 to 6 hours

D) 6 to 8 hours

E) More than 8 hours

Design Pattern

- **“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**

- **Each pattern has 4 essential elements:**
  - A name
  - The problem it solves
  - The solution
  - *The consequences*

Let's start with some "Micro-Patterns" (1)

- Name: **Most-wanted holder**
- Problem: **Want to find the "most wanted" element of a collection.**
- Solution: **Initialize most-wanted holder to first element. Compare every other element to value in most-wanted holder, replace if the new value is better.**

```
Thing mostWanted = things[0];
for (int i = 1 ; i < things.length ; i ++) {
   if (thing[i].isBetterThan(mostWanted)) {
      mostWanted = thing[i];
   }
}
```

Let’s start with some “Micro-Patterns” (2)

- Name: One-way flag
- Problem: **Want to know if a property is true/false for every element of a collection.**
- Solution: **Initialize a boolean to one value. Traverse the whole collection, setting the boolean to the other value if an element violates the property.**

```
boolean allValid = true;
for (Thing thing : things) {
   if (!thing.isValid()) {
      allValid = false;      break; ...
   }
}
```

Let’s start with some “Micro-Patterns” (3)

- Name: **Follower**
- Problem: **Want to compare adjacent elements of collection.**
- Solution: **As you iterate through a collection, set the value of the follower variable to the current element as the last step.**

```
boolean collectionInOrder = true;
Thing follower = null;
for (Thing thing : things) {
    if (follower != null &&
            thing.isBiggerThan(follower)) {
        collectionInOrder = false;
    }
    follower = thing;
}
```

← one-way flag

“Design Patterns” focus on object-level

- **Relate to relationships between classes & objects**
  - IsA (inheritance) and HasA (containment) relationships

- **Many of these seem obvious (in hind sight)**
  - The power is giving these names, codifying a best practice solution, and understanding their strengths/limitations.

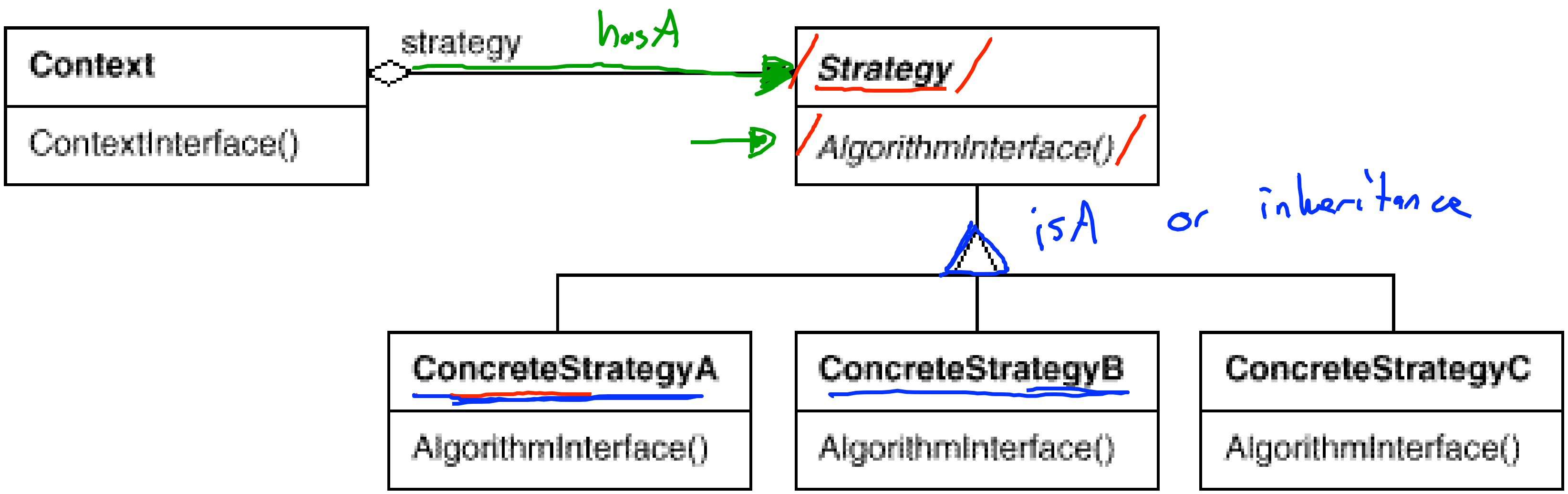
Strategy

- **Intent: define a family of algorithms, encapsulate each one, and make them interchangable. Strategy lets the algorithm vary independently from clients that use it.**

- **Use the strategy pattern when:**
  - Many related classes differ only in their behavior.
  - You need different variants of an algorithm (e.g., trade-offs)
  - An algorithm uses data that clients shouldn't know about
    - E.g., encapsulate the algorithm data from client
  - A class defines multiple behaviors and these are implemented using conditionals.

Strategy Pattern

- **Solution**
  - Strategy abstract base class exposes algorithm interface.
  - Context object *HasA* Concrete Strategy object.
  - Context object invokes algorithm interface from strategy.



Predator / Prey simulation discrete time

- **The simulation progresses in units of time, called 'epochs'.**

- **A Cell has a given amount of vegetation, which is a non-negative value.**
  - Each epoch, the vegetation grows to: previousValue * proportionalGrowthRate + linearGrowthRate
  - Each animal can eat up to vegetation / (2 * number of animals) S
  - There is a maximum amount of vegetation that a cell can hold

- **Rabbits eat vegetation.**
  - A rabbit will eat up to its share of the vegetation or (half its weight + 1), whichever is less s
  - If it doesn't eat enough, it loses weight, and accumulates a 'hungerDeficit'
  - The larger the hunger deficit, the more likely that the rabbit dies of hunger
  - If the rabbit has plenty of food it gains weight
  - If the rabbit is large enough, it reproduces

Wolves eat (mostly) bunnies

Epoch : 172 Wolves : 17 Rabbits : 230

Tracking number of objects of given kind

1 copy

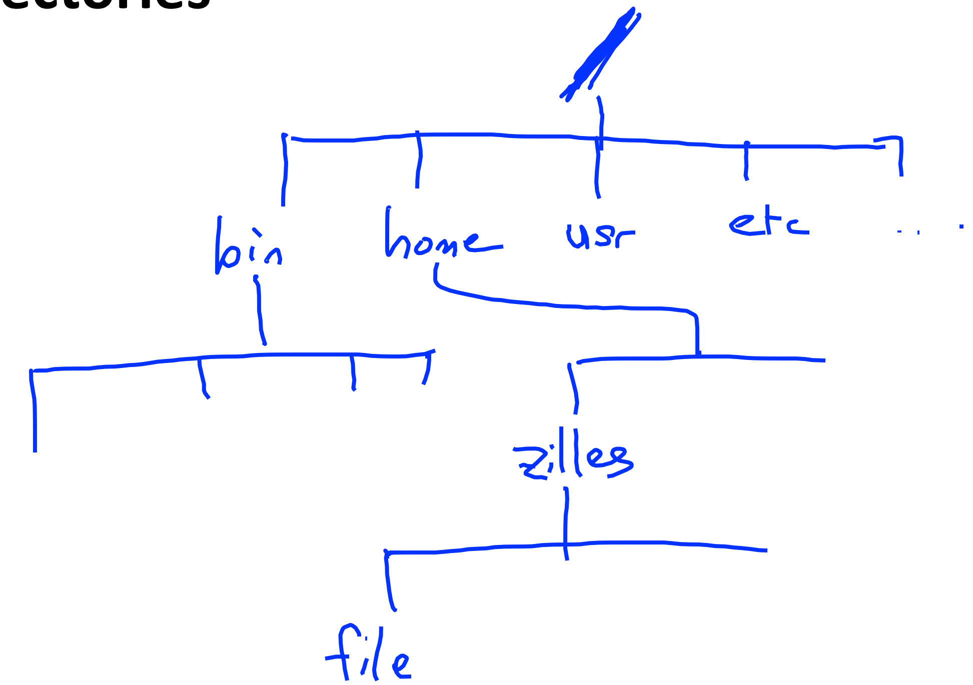
- Allocate a static integer variable
- Increment this variable in the constructor
- Decrement this variable when you are done with the object

count

static

Unix Filesystems

- Are generally tree-like
- The root is called: /
- Leaf nodes are files
- Non-leaf nodes are directories



Working with Files

- **touch – create an empty file with a given name**
  - E.g., touch blah
- **rm – remove a file of a given name**
  - E.g., rm blah
- **mv – rename a file from one name to another**
  - E.g., mv old_filename new_filename

Paths: two kinds

- **Absolute paths start from:**
  - Filesystem root: **/**usr/bin/tail
  - Home directories: **~**/temp/file, **~username**/foo/bar
- **Relative paths start from the current working directory:**
  - filename — in cwd
  - dirname1/dirname2/filename
  - (pwd – print current working directory)

- **Special path elements:**
  - . – current working directory
  - .. – up one directory

Navigating the filesystem

- cd – change directory
- mkdir – make (e.g., create) directory
- rmdir – remove directory