

Design Heuristics: class/program/function

(see text by Arthur Riel)

- **Coupling**
 - classes/modules are independent of each other
 - goal: minimal, loose coupling
 - do classes collaborate and/or communicate?
- **Cohesion**
 - classes/modules capture one abstraction/model
 - keep things as simple as possible, but no simpler
 - goal: strong cohesion (avoid kitchen sink)
- **The open/closed principle**
 - classes/programs: open to extensibility, closed to modification

Programming Heuristics

- **Identify the aspects of your application that vary and separate them from what stays the same**
 - Take what varies and encapsulate it
- **Program to an interface, not an implementation**
 - Specify behavior by name, not by working code
- **Favor Composition over Inheritance**
 - Use "has-a" rather than "is-a"
- **Classes and code should be open for extension, but closed to modification**
 - The Open-Closed Principle

Tell, Don't Ask

- **Tell objects what you want them to do, do not ask questions about state, make a decision, then tell them what to do** (*Pragmatic Programmers, LLC*)
 - **Think declaratively, not procedurally**
 - **Don't ask for a map, then walk through the map**
 - **Instead of iteration, apply to all**
 - **Breaks when we don't want to apply to all**
- **Rules are made to be broken**
 - **Reduce coupling, better code**

Law of Demeter

- Don't talk to objects, don't call methods. The more you talk, the more you rely on something that will break later
 - Call your own methods
 - Call methods of parameter objects
 - Call methods if you create the object
- Do *NOT* call methods on objects returned by calls

```
List all = obj.getList();
all.addSpecial(key,getValue());
obj.addToList(key,getValue()); // ok here
```

Design patterns

“... 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, quoted in GOF

- **Name**
 - good name is a handle for the pattern, builds vocabulary
- **Problem**
 - when applicable, context, criteria to be met, design goals
- **Solution**
 - design, collaborations, responsibilities, and relationships
- **Forces and Consequences**
 - trade-offs, problems, results from applying pattern: help in evaluating applicability

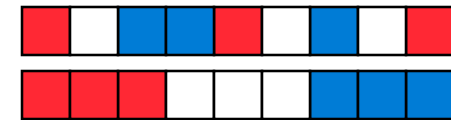
Patterns are discovered, not invented

- You encounter the same “pattern” in developing solutions to programming or design problems
 - develop the pattern into an appropriate form that makes it accessible to others
 - fit the pattern into a language of other, related patterns
- Patterns transcend programming languages, but not (always) programming paradigms
 - OO folk started the patterns movement
 - language idioms, programming templates, programming patterns, case studies
- *Patterns capture important practice in a form that makes the practice accessible*

Pattern/Programming Interlude

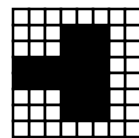
- Microsoft interview question (1998)
- Dutch National Flag problem (1976)
- Remove Zeros (AP 1987)
- Quicksort partition (1961, 1986)
- Run-length encoding (SIGCSE 1998)

3	3	5	5	7	8	8	8
3	5	7	8				



2	1	0	5	0	0	8	4
2	1	5	8	4			

4	3	8	9	1	6	0	5
3	1	0	4	8	9	6	5



11 3 5 3 2 6 2 6 5 3 5 3 5 3 10

One loop for linear structures

- Algorithmically, a problem may seem to call for multiple loops to match intuition on how control structures are used to program a solution to the problem, but data is stored sequentially, e.g., in an array or file. Programming based on control leads to more problems than programming based on structure.

Therefore, use the structure of the data to guide the programmed solution: one loop for sequential data with appropriately guarded conditionals to implement the control

Consequences: one loop really means loop according to structure, do not add loops for control: what does the code look like for run-length encoding example?

Coding Pattern

- **Name:**
 - one loop for linear structures
- **Problem:**
 - Sequential data, e.g., in an array or a file, must be processed to perform some algorithmic task. At first it may seem that multiple (nested) loops are needed, but developing such loops correctly is often hard in practice.
- **Solution:**
 - Let the structure of the data guide the coding solution. Use one loop with guarded/if statements when processing one-dimensional, linear/sequential data
- **Consequences:**
 - Code is simpler to reason about, facilitates develop of loop invariants, possibly leads to (slightly?) less efficient code

Design patterns you shouldn't miss

- **Iterator**
 - useful in many contexts, see previous examples, integral to both C++ and Java
- **Factory**
 - essential for developing OO programs/classes, e.g., create iterator from a Java List? `list.iterator()`
- **Strategy**
 - encapsulate an algorithm as an object, supports swapping algorithms during execution
- **Command**
 - encapsulate a request as an object, supports undo, reusable commands
- **Observer/Observable, Publish/Subscribe, MVC**
 - separate the model from the view, smart updates

Essential Features of Design Patterns

- **Delegation**

- If many ways to do something, delegate actual details to a separate module (i.e., packages, classes, methods)
- Add an extra level of indirection to support flexibility

- **Substitution**

- If many ways to do something, try to give it the same interface so that one can be substituted for another (i.e., member functions, method names, parameters)
- Create correct one and use it at correct time