CSC323 2010S *Software Design*

Class 23: Design Patterns (1): Philosophy and Some Basic Patterns

Held: Tuesday, April 27, 2010

Summary: We consider *design patterns*, a technique for encapsulating common designs that solve common problems.

Related Pages:

- EBoard.

Notes:

- Reading for Thursday: Creational Patterns at `http://c2.com/cgi/wiki?CategoryCreationalPatterns`.
- EC for Thursday's convocation.
- EC for Friday's lunchtime talk.

Overview:

- Aggregation, Again.
- Multiple perspectives on software reuse.
- The Design Patterns philosophy.
- Some Patterns from Scheme.
- An example: Sorting.
- The Patterns Format(s).
- The Design Patterns catalog, abbreviated.
- Project Stuff.

Aggregation, Again

- "Aggregation vs. Acquaintance" == "Aggregation vs. Association" == "Aggregation vs. Composition"
- Note: In some situations, it is useful to distinguish the "ownership" of an object
  - Is it only used by exactly one other object? (Aggregation)
  - Is it potentially used by many objects? (Any of the other terms)
- For example,
  - Each hand is typically used by only one human.
  - In contrast, a pen in that hand may be used by multiple humans.
- We might also categorize classes in this way (e.g., whenever I create an object in this class, it's aggregated).

Aggregation and Storage

- Aggregation has an effect on how we do the underlying implementation.
- We'll use C structs to explain.
- Consider a utility struct, U

```
typedef struct U
  {
    ...
  } U;
```

- A non-aggregated object must be stored by reference.

```
typedef struct Composed
  {
    U *u;
    ...
  } Composed;
```

- In contrast, an aggregated object can be stored directly within the class.

```
typdef struct Aggregated
  {
    U u;
    ...
  } Aggregated;
```

Why Distinguish Aggregation?

- We've just seen one reason: It can affect how we lay out values.
- It tells you something about construction and destruction of objects in the enclosing class (and of objects in the aggregated class, particularly if the class is always aggregated).
- It can help with analysis: If we know an object is aggregated, then we can be sure that it maintains the values we set.
- If your aggregation ensures that "an aggregate object and its owner have identical lifetimes" (p. 22) then you also know that you'll never again assign to the "field" that contains the aggregate object.
- In some patterns (e.g., Composite), we know that we use aggregation to indicate that we have a tree structure rather than a graph structure.
- ...

Multiple perspectives on software reuse

- Good software design often involves *reuse*
  - We try to use code that's been written previously
  - We try to design code that can be used again
- Why reuse?
  - It speeds up programming (or at least it can)
  - It can help improve the quality of programs (if the reused parts are well tested)
  - It can improve the maintainability of programs (changes propagate easily)

- There are a number of ways we can reuse code
  - We can copy and paste code from one program to another
  - We can factor common code into a procedure
  - We can put related frequently-used procedures into a library
  - We can put related frequently-used ADTs and Data Structures into a library
  - We can put related classes into a library
  - We can build the skeleton of a particular kind of program and leave stubs to be filled in for a particular version (the DP book calls these frameworks)
  - We can identify heuristics that guide us in the development of code (reusing design knowledge, rather than code)
  - ...

The Design Patterns philosophy

- The Design Patterns philosophy suggests that we can reuse fairly high-level strategies for design without reusing particular code.
- A strategy identifies a programming "problem" and a typical way that the problem is solved well.
- You may find that multiple patterns apply to the same problem; in this case it's your job to decide which is best for your particular situation.
- Like heuristics, patterns are guides, not rules.
- Warnings!
  - The DP book is much like any philosophy text: It is quite self-referential, and it is unlikely you will understand everything on the first (or second or ...) reading.
  - I will admit that I am more casual in my use of the DP book (I know a lot of patterns, but don't always refer directly to the "official" patterns)
  - There are multiple perspectives on what a "pattern" is.

Some Patterns from Scheme

- When programming in Scheme, we consciously or subconsciously identified some important patterns.
- "Build a new list by doing something to every element of a list"
- "Compute a compound value based on the elements of a list"
- One of the nicer aspects of Scheme is that you can encapsulate these patterns in code (e.g., as a higher-order procedure)

An example: Sorting

- To see that many patterns can apply to the same problem, let us consider the problem of generalized sorting (writing a sorting algorithm that applies to different types and permits different orderings of the same type).
- We can accept the comparator as a parameter to the sorting routine (I'd call this a "higher-order" pattern).
- We can leave a stub for the comparison and require that programmers subclass the sorting class to

include a new comparison function.
- We can require the data to provide its own comparison function. When we want data to compare itself differently, we must subclass the data.

The Design Patterns catalog, abbreviated

- See pp. 8-9 of the book.

The Patterns Format(s)

- A high-level overview:
  - Name
  - The problem
    - How do I do X?
    - I've done Y, how can I do it better?
    - My program requires Z1, Z2, and Z3. What pattern applies?
  - The solution
  - The consequences
- The low-level details (pp. 6-7)
  - Name
  - Classification
  - Intent
  - AKA
  - Motivation (example)
  - Applicability
  - Structure
  - Participants
  - Collaborations
  - Consequences
  - Implementation problems
  - Sample code
  - Real-world examples
  - Related patterns

Project

- How to convey information to other teams?
- Shall we do the Networked game walkthrough?

Copyright © 2010 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc/2.5/` or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.