Perspective in Object-Oriented Programming

How to be an Informed Consumer of Software Theory and Practice
Optimization

- A component of Programming
- Not just the *algorithms*, also
  - Readability
  - Maintainability
  - Testability
  - Etc.
“Why callest thou me good?”

What does “good” mean?

What does quality mean?

Long discussion: Zen, and the art of Motorcycle Maintenance

My view: Quality is the measure to which something fulfills its purpose

Purpose is individual, can be subjective
What makes a program “good”? Speed? Stability? Maintainability?

For any non-trivial criteria, maximizing “goodness” is hard.

Optimizing for one criteria may reduce performance of another.
On Being an Informed Consumer

- Computer Scientists are consumers
- At a minimum, they require a programming paradigm and compliant language
- More often, they also require
  - Methods, concepts, philosophies
  - Tools such as IDEs, debuggers
  - Tools such as UML, patterns
Motivations of the CS Producers

- Money (pure and simple)
- Ego
- Fame and/or Prestige
- Your best interest (rarely)

**NOTE:** The tools you use benefited one or more persons
How to Choose Wisely

- You do need to buy bullets
- How do you choose wisely?
- They’re obviously not silver...
- Do they have any value?
No-Free-Lunch Theorem

“[...] all algorithms that search for an extremum of a cost function perform exactly the same, when averaged over all possible cost functions” - (Wolpert & Macready, 1995)
NFLT Restated

Optimizations, when averaged over all possible problems, perform exactly the same.

Applied to CS: programming tools/methods/etc., when applied to all possible programming problems, perform exactly the same.
NFLT Restated Again

"NO SILVER BULLET"

--Brooks
Context: Key to Optimization

- Optimization algorithms
  - Perform well on function classes
  - Class size not impacted by NFLT
  - Identification of class is hard
Back to Programming

- Tools/Methods perform well on software problem *classes*
- Determining the class is hard
- Reverse engineer tool/method for approximation of the class
- Also requires understanding of solution space, and "goodness" function (don’t optimize wrong thing)
One View of Software

- Continuous range of abstraction
  - Requirements
  - Architectural/Conceptual
  - Design
  - Algorithmic
  - (Programming) Linguistic
  - Etc
Abstraction Interfaces

- As humans, we discretize
- Create abstraction layers
- Create interfaces between layers
Working in Layers

- Most abstraction layers can influence above or below
- However, flow is from abstract to concrete
- Most opposite flow is some form of feedback
Common Design Error

- Design with insufficient input from higher abstraction
- Feedback alters view of higher abstractions
- Adventure Example: lighted room
Optimization Problem

- The reverse abstraction flow problem is picking the wrong optimization.
- This error was independent of programming paradigm, language, and tool.
Personal Opinion:

- **Object-oriented programming** is generally less effective without object-oriented design
- **Object-oriented design** is generally less effective without object-oriented analysis
OOA Mini How-To

- (For more: read a book)
- Organize problem space into objects
- Objects may or may not have **ANY** connection with implementation objects later
- Objects have:
  - Structure
  - Behavior
  - State
OOA and UML

- You **CAN** use UML, **BUT**
  - DO NOT be tempted to translate to code
  - You don’t **have** to ascribe properties and methods to UML objects. For OOA, you generally don’t
OOA: Structural

- What is it?
- What is it composed of?
- What properties does it have?
- What is it related to?
- What does it interact with?
OOA: Behavioral

- Start with the answer to “What does it interact with” question
- What kind of messages does it send?
- What kind of messages does it receive?
- What kind of messages does it require?
OOA: State

- Describe *basic* state machine of major objects
- Tie in with messages from OOA: Behavioral
Why so much structure?

- OOA lends itself to planning more structure than behavior or state.
- Why?
  - OO[A/D/P] holds implicit belief that *form follows function*. 
Example: Adventure

- What is a Game?
- This is a significant question
  - Does a Game have rooms?
  - Does a Game have a World that has rooms?
  - Does a Game have input and output?
- What are the relationships?
Concrete Example: Rooms

- Does a room have a light property?
- If so, what properties does a flashlight have?
- Let’s do this on the board
One Concluding Note

Any work in OOA, OOD, or OOP should be driven by the requirements

This is the most broken commandment in computer science
On To Patterns

What are they?

According to Steve Bilow:

“... patterns have the potential to permanently alter the software engineering field, catapulting it into the realm of true elegant design.”
Yet More Praise for Patterns

The worlds of Tom Cargill

“After a modest investment of time with it, most C++ programmers will be able to start applying its ‘patterns’ to produce better software”
Hurray! The Silver Bullet!

- I guess my lecture is all wrong
- We’ve finally found the universal optimizer
- Also, it seems like it makes everything about software better
- Don’t you feel all warm and fuzzy?
But... wait...

- *Design Patterns*, the book by the *gang-of-four* came out in 1995
- I’ve worked for four companies since that time
- Many of us trained in patterns, still struggle to get designs right...
That’s odd...

Because the introduction to *Design Patterns* says this:

“Put simply, design patterns help a designer get a design ‘right’ faster” (page 2)
Where is the Disconnect?

- Personal opinion: forgetting that the bullet is not silver
- Design patterns are an amazingly effective tool
- In fact, they are an optimizer
- They optimize a class of problems
- So, let’s backtrack in time...
A History of Patterns

- The concept of patterns was first formalized for architecture by Christopher Alexander
- Three most cited books:
  - *The Timeless Way of Building*
  - *A Pattern Language*
  - *The Oregon Experiment*
History ... Again

- TWOB and a Pattern Language should be required reading
- In class:
  - What is a pattern? (PL, p. x).
  - What is a pattern language? (TWOB, p. 202)
- Advice: check you library
What is a Pattern, again?

- We’ve got three types of patterns going on here:
  - “pattern” - see dictionary.com
  - Pattern - as per Alexander
  - Pattern™ - as per GoF
- We need to talk about all three.
Dictionary.com:

“"A model or original used as an archetype”"

There are patterns in every aspect of software engineering at every level of abstraction

In reality, our world view is based on patterns
Pattern

- Describes an architectural problem that occurs over and over again
- Proposes a core solution that can be done a million ways without doing it the same way twice
- Part of a larger Pattern Language
Pattern™

- Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context
- Names, abstracts, and identifies the key aspects of a common design structure
Getting to the Core of Patterns and Patterns™

- The common idea is: *don’t re-invent the wheel.*
- What separates these from “patterns” is that they have a name, some analysis, and a written form.
Limitations on Applying Alexander

Alexander is discussing *intimate* architecture.

In other words, the producer is the consumer of the design.

*Slow* contextual changes.

Architecture has *thousands of years of experience*. 
Limitations on the Gang of Four

- Narrow view of a Pattern™
- Rigidity in pattern maintenance
- Worst of all:
  A significant dearth of work on when to use and when not to use a pattern
Limitations on Pattern™ Use in Software

- Superficial similarities between past and present problems
- Reverse abstraction flow
- Temptation to apply to wrong level of abstraction
- Fragility to context changes (capture only; no-meta)
- NO LANGUAGE
Using Patterns™
“Tastefully”

- Analysis of context
- Keep to the requirements
- Modify for contextual changes
- Use to solve problems that arise in design, not as “cookie cutters” in the design space
- **FINALLY:** it is about resolving forces!!!!
Alexander believed architecture was dead because of a dearth of communication.

Expand view beyond *Patterns*™ to something more like Alexander’s *Pattern Language*.

Don’t be too proud to learn from somebody else’s “patterns”.
Recognizing “patterns” and Pattern-like things

- Requirements patterns
- “Best Practices” is Pattern-like
- “Effective Java” is Pattern-like
Final Example

- Multi-paradigm Programming
- Significant context change
- Don’t wait for new patterns to be captured
- Explore the meta-space to translate to the new context
My Final Thought about the Patterns Community

One significant member said:

- Most people aren’t qualified to talk about what patterns are
- Only he, and a few others, with time and means should venture into that space
- Everyone else should just content themselves to write/mine patterns
Summary

- **YOU** are a consumer
- Be informed, understand motives
- Remember NFLT
- Understand the theory and background of what’s being sold to you
- Never trust marketing