V22.0474-001 Software Engineering Spring 2008

Lecture 6: Design

Clark Barrett, New York University

Review

We have been talking about *requirements*: *the descriptions of the services provided by the system and its operational constraints.*

Requirements focus on *what* the program should do, not on *how* this functionality is implemented.

Today, we will talk about *design*, the next step in the software process.

Outline

Design

- Design Challenges
- Key Design Concepts
- Design Heuristics
- Design Practices

Sources for today's lecture:

Eric Van Wyk. *CSci 3081: Program Design and Development*. Lecture notes, Spring 2005.

McConnell, Steve. *Code Complete*, Second Edition, ch. 5.

Software Design

What is software design?

Software Design is the process by which a specification for computer software is elaborated and refined into operational software.

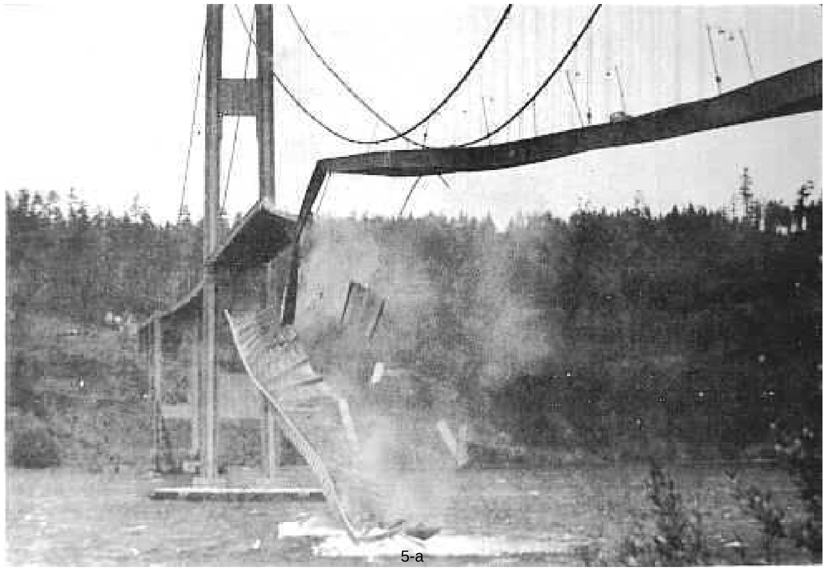
Design

- links requirements with coding and debugging
- is useful even on small projects
- is indispensable on large projects

Design Challenges

Why is software design difficult?

Design is a “wicked” problem, one that can only be understood by solving it.



Design Challenges

Why is software design difficult?

Design is a sloppy process

- Involves many false starts and mistakes
- This is the whole point of design
- Hard to know how much detail is enough
- Common answer: when you’re out of time

Design Challenges

Why is software design difficult?

- *Design is about trade-offs and priorities* Often, desirable system characteristics conflict with each other. Part of design is deciding which characteristics to emphasize.
- *Design involves restrictions* Designers are reluctant to impose restrictions on their design. But constraints force simplifications that ultimately improve the design.
- *Design is nondeterministic* There are many ways to design a program, even many acceptable ways.
- *Design is a heuristic process* No one tool or technique is right for every project. Design involves trial and error.

Key Design Concepts

Managing complexity: Software’s Primary Technical Imperative

- Tony Hoare:

  There are two ways of constructing a software design: one way is to make it so simple that there are obviously *no deficiencies, and the other is to make it so complicated that there are no* obvious *deficiencies*

- Fred Brooks describes two types of complexity: *essential* and *accidental*.
- Essential complexity cannot be removed by better programming languages or development environments, it is fundamental
- By choosing a poor design, we may introduce accidental complexity
- Strategy for dealing with complexity
  - *Problem Decomposition:* Minimize the amount of essential complexity that has to be dealt with at any one time. In most cases, this is the *top priority*.
  - *Avoid accidental complexity:* Understand desirable characteristics of a good design and how to achieve them.

The importance of minimizing and managing complexity cannot be overemphasized.

Key Design Concepts

Problem decomposition: levels of design

- *Software system* The system as a whole
- *Division into subsystems/packages* Identify major subsystems like: database, user interface, business rules, command interpreter, OS-specific routines, etc. Communication between subsystems only on a “need to know” basis. Try to keep dependency graph acyclic.
- *Division into classes within packages* Divide packages into reasonable implementation chunks. Make sure every service described at the package level is implemented by some class.
- *Division into data and routines within classes* Make sure each class has a clear mission. Decide what supporting data and routines are needed to implement the mission.
- *Internal routine design* For routines that are not straightforward, this may require writing and analyzing pseudocode, looking up algorithms, and considering trade-offs.

We will have more to say about problem decomposition and hierarchical design.

Key Design Concepts

Accidental complexity can be avoided by aiming for a high-quality design. High-quality designs share a number of characteristics.

Desirable design characteristics

- *Minimal complexity* The first priority is to minimize complexity. Be wary of “clever” designs. Instead, aim for simple designs. If you can’t ignore most of the system when focusing on one specific part, the design isn’t doing its job.
- *Ease of maintenance* Try to design the system to be self-explanatory. Ask yourself how hard it would be for another programmer to understand and modify your code.
- *Loose coupling* Use the principles of abstraction, encapsulation, and information hiding to design classes with as few interconnections as possible.
- *Extensibility* Design in such a way that the most likely changes can be done easily.
- *Reusability* When possible, design components for reuse, both within the program and in other programs.

Key Design Concepts

Desirable design characteristics

- *High fan-in, Low-to-medium fan-out* Try to maximize the number of classes that use a given class, while minimizing the number of classes used by a given class. Aim for an inverted-tree structure.
- *Portability* Design so that it’s easy to move the program to another platform
- *Leanness* Voltaire said that a book is finished not when nothing more can be added but when nothing more can be taken away. Take the same approach to software development.
- *Stratification* Design levels of decomposition so you can view the system at any single level and get a consistent view: i.e. details that are hidden from one class should be hidden from other classes in the same level.
- *Standard techniques* Give the system a familiar feeling by using standardized, common approaches.

Design Heuristics

Software developers like processes that are deterministic and work every time. Unfortunately, because design is nondeterministic, no single process is always the best. There are, however, a set of heuristics that often work. Think of these as a guide to possible trials in a “trial and error” process.

Design Heuristics

Find real-world objects

- *Identify objects and their attributes* Programs are usually based on real-world entities. These are a good place to start for objects. Object attributes are characteristics of the real-world object that are relevant to the program.
- *Determine what can be done to each object* Based on the objects and their attributes, decide what operations are needed to modify the object.
- *Determine what each object is allowed to do to other objects* Define possible object interactions including *containment* and *inheritance*.
- *Determine the parts of each object that will be visible to other objects* Try to keep the public part of each object as simple as possible to minimize complexity.
- *Define each object's interfaces* Define and write code for both the public and protected interface.

These steps are not necessarily done in order and may involve several iterations.

Design Heuristics

Form consistent abstractions

- *Abstraction* is the ability to consider something while ignoring some of its details.
- Abstraction should present a consistent and understandable view of the object
  - Abstract concept *house* could be described as a set of *rooms*
  - This is more intuitive than say, *left half* and *right half*
  - *Kitchen* is an appropriate sub-component.
  - *Fireplace brick* is probably not appropriate: too low-level.
- Abstraction should be used at every level of the design.

Encapsulate implementation details

- Abstraction says “You're allowed to ignore details”
- *Encapsulation* says “You're not allowed to not ignore details”
- Encapsulation enforces the principle of managing complexity by decomposition.

Design Heuristics

Information hiding

- Principle for using encapsulation
- Identify and hide secrets at all design levels
- Two kinds of secrets:
  - Complexity that can be safely ignored
  - Implementation details that are likely to change
- Information hiding has proven its value in practice
- Get into the habit of asking “What should I hide?” when designing a class

Use inheritance when it simplifies the design

- Many objects are similar to other objects
- Similarities can be organized into inheritance hierarchies
- Simplifies program by avoiding repetition of common routines
- Works synergistically with abstraction to reduce complexity
- Make sure inheritance captures an *ISA* relationship

Design Heuristics

Identify areas likely to change

- *Prepare for change*
  - *Identify items that seem likely to change* Use requirements or your own experience (or the list below) to identify areas likely to change
  - *Separate items that are likely to change* Put volatile components in their own class
  - *Isolate items likely to change* Design the interface of the class to hide the details that are likely to change
- *Some areas likely to change*
  - Business rules, Hardware dependencies, Input and Output
  - Nonstandard language features, Difficult design and construction areas
  - Flags or status variables, Data-size constants
- *Avoid “designing ahead”*
  - Preparing for change does not mean guessing what the change will be
  - Design-ahead code rarely anticipates changes correctly
  - Design-ahead code is often untested or broken

Design Heuristics

Keep coupling loose

- *Coupling criteria*
  - *Size* Keep number of connections between modules minimal
  - *Visibility* Prefer a direct and visible connection to an indirect connection: i.e. passing a parameter is better than communicating through shared or global data.
  - *Flexibility* Make it as easy as possible to change connections: i.e. prefer passing primitive data types to object types
- *Kinds of coupling*
  - *Simple-object coupling* A module is simple-object coupled to an object if it instantiates that object. This is usually fine.
  - *Object-parameter coupling* Modules are object-parameter coupled if one must pass the other an object rather than a primitive data type. This is less flexible than passing primitive data types.
  - *Semantic coupling* A module relies on some knowledge of the internal implementation of another module. This is dangerous and should be avoided.

Design Heuristics

Other heuristics

- *Design patterns* Be aware of commonly occurring design patterns
- *Cohesion* Routines in a class should share a common purpose
- *Build hierarchies* Fundamental for managing complexity
- *Formalize class contracts* Think about preconditions and postconditions
- *Assign responsibilities* Ask what each object is responsible for
- *Design for test* Think about how easy the design will be to test
- *Avoid failure* Consider possible failure modes
- *Choose binding time consciously* Choose when variables get assigned
- *Make central points of control* Makes it easier to make changes
- *Consider using brute force* Premature optimization creates bugs

Design Heuristics

Other heuristics

- *Design patterns* Be aware of commonly occurring design patterns
- *Cohesion* Routines in a class should share a common purpose
- *Build hierarchies* Fundamental for managing complexity
- *Formalize class contracts* Think about preconditions and postconditions
- *Assign responsibilities* Ask what each object is responsible for
- *Design for test* Think about how easy the design will be to test
- *Avoid failure* Consider possible failure modes
- *Choose binding time consciously* Choose when variables get assigned
- *Make central points of control* Makes it easier to make changes
- *Consider using brute force* Premature optimization creates bugs
- *Draw a diagram* Diagrams can help organize your thoughts
- *Keep your design modular* Think of system as composed of black boxes

Design Practices

Design practices are additional heuristics that describe general strategies that are independent of the specific heuristics just described.

Iterate

- Design is an iterative process
- After one iteration, you have a better grasp of the big picture
- The second attempt is almost always better than the first attempt

Divide and conquer

- Key idea for managing complexity
- Break problem into sub-problems and tackle them one at a time
- If you run into a dead-end, iterate: try again with a different design

Design Practices

Top-down vs Bottom-up design

- *Top-down*
  - Start with generalities and refine with more and more details
  - Decompose until the design seems easy and obvious
  - Starts simple and slowly becomes more complex
- *Bottom-up*
  - Start with known components of the system
  - Build by composition instead of decomposition
  - Useful when system is still too unclear for top-down
- *Top-down vs Bottom-up*
  - Both are useful
  - Often, alternating between them is the best approach

Design Practices

Experimental prototyping

- Understand design questions by experimenting
- Write minimum amount of code necessary to answer questions
- Make sure design question is specific
- Make sure programmers understand this is throwaway code

Collaborative design

Often, two heads are better than one. Lots of ways to use collaboration:

- Informal discussion
- Design reviews
- Self review
- Pair programming

Design Practices

How much design is enough?

- Depends on experience of team, lifetime of system, desired level of reliability, size of project and team
- Individually, stop designing when the path to implementation is clear
- Both over-designing and under-designing can waste a lot of time

Capturing your design work

- Insert design documentation into the code itself
- Capture design discussions and decisions on a Wiki
- Write email summaries
- Use a digital camera or whiteboard printouts
- Save design flip charts
- Use CRC (Class, Responsibility, Collaborator) index cards
- Create UML diagrams

Project

Requirements

- Requirements documents are due on Tuesday before midnight
- Requirements presentations will take place next week, on Feb. 14.
- Design assignment will be posted on Tuesday and is due Feb. 21.