Chapter 5: Design in Construction

Design Challenges

- Usually you will need to solve part of the problem before the whole problem can be clearly defined
- Design is a sloppy process (even if it produces a tidy result)
    - lots of mistakes
- Design is about tradeoffs and priorities
- Design involves restrictions
- Design is non-deterministic (many ways to skin a cat)
- Design is a heuristic process, rules of thumb that work get reused
- Design is emergent, things grow spontaneously after spending more and more time with the code seeing what's possible

Key Design Concepts

1) Software's Primary Technical Imperative: Managing Complexity
    a. Accidental and Essential Difficulties
        i. Essential ones are like difficulties interfacing with real world
        ii. Accidental is like implementation errors
    b. Managing Complexity
        i. When a project reaches the point where nobody completely understands the impact their code changes have, that is bad haha
        ii. There are two ways of constructing a software design: make it so simple that there are obviously no deficiencies, or to make it so complicated that there are no obvious deficiencies
        iii. Goal is to minimize the amount of a program you have to think about at any one time
        iv. Write code that compensates for inherent human limitations
    c. How to Attack Complexity
        i. Overly costly, ineffective designs arise from three sources:
            1. A complex solution to a simple problem
            2. A simple, incorrect solution to a complex problem
            3. An inappropriate, complex solution to a complex problem
        ii. Two-pronged approach to managing:
            1. Minimize amount of essential complexity that anyone's brain has to deal with at any one time
            2. Keep accidental complexity from needlessly proliferating
    d. Once you understand that all other technical goals in software are secondary to managing complexity, many design considerations become straightforward.
2) Desirable Characteristics of a Design
    a. Minimal Complexity
        i. Avoid "clever" designs

1. **clever designs are usually hard to understand**
2. **instead, make "simple" and "easy-to-understand" designs**

ii. easy maintenance

iii. loose coupling
1. **connections among different parts of a program to a minimum**
2. good abstractions in class interfaces
3. encapsulation

iv. extensibility
1. **enhance a system without changes causing violence to underlying or other pieces**

v. reusability

vi. high fan-in
1. **high number of classes that use a given class**
2. high fan-in means that a system has been designed to make good use of utility classes at the lower levels in the system

vii. low-medium fan-out
1. **given class uses low number of other classes (>7 is high)**

viii. portability
1. **can be easily moved to another environment**

ix. leanness
1. **a book is not finished when nothing more can be added, rather it is when nothing more can be taken away**

x. stratification
1. keep levels of decomposition stratified so you can view the system at any single level and get a consistent view
2. view at one level without dipping into other levels
3. compartmentalize code, create good interface

xi. standard technique
1. **the more a system relies on exotic practices, the more intimidating it will be for someone to try and understand**

3) Levels of Design

1) Software system
   a. First level is the entire system
   b. Just make sure you look at class combinations before you start coding
2) Division into subsystems/packages
   a. Main thing is need to identify all major subsystems
      i. Database, UI, business rules, interpreter, report engine
   b. Also need to figure out how these all communicate
      i. **MAKE EACH SUBSYSTEM MEANINGFUL BY RESTRICTING COMMUINICATIONS**
      ii. if all can communicate, there will be unintended consequences in every modification to code
      iii. **Allow communication on a NEED TO KNOW basis**

            iv. Easier to restrict communication early and then relax, than start relaxed and make strict
        c. Keep subsystems acyclic
    3) Division into classes within packages
        a. Define all interfaces and how each class interacts with rest of system
    4) Division into data and routines within classes
        a. Pretty free reign to decompose tasks as a programmer
    5) Internal routine design
        a. Lay out detailed functionality of individual routines

Design Building Blocks: Heuristics

- Chasing "Do A, B, and C and X, Y, Z will follow every time"
- Heuristics can be thought not of as the solution, but as the trials in a trial and error session to find the solution

Find Real-World Objects

- Like Employee, Timecard, Client, Bill
    - Identify objects and their attributes (methods and data)
    - Determine what can be done to each object
    - Determine what each object is allowed to do to other objects
    - Determine which parts of object will by public vs private
    - Define each objects public interface
- Also going to want to figure out inheritance from these

Form Consistent Abstractions

- Abstraction: ability to engage in a concept while ignoring some details
- Create abstractions at:
    - routine-interface level
    - class-interface level
    - package-interface level

Encapsulate Implementation Details

- Encapsulation picks up where abstraction leaves off
- **Abstraction says: "You're allowed to look at an object at a high level of detail"**
- **Encapsulation says: "Furthermore, you aren't allowed to look at an object at any other level of detail"**

Inherit

- When Inheritance Simplifies the Design
- When objects are a lot like other objects
- Most data overlaps, but some is different

Hide Secrets (Information Hiding)

- **Partially about private information, but mostly about hiding COMPLEXITY**
- In info hiding, each class (or package or routine) has secrets
- Idea is to keep changes from rippling outside of the implementation
    - **Class interfaces should be complete, but minimal (and reveal as little about the inner workings as possible)**
- Id example and why using id++ to assign new ids is bad
    - not thread safe
    - what if wanted non-sequential for security
    - what if want to reuse destroyed ids?
    - abstract to id = newID()
        - can play with implementation behind abstraction layer now
    - also, declare Id to be its own data type
        - that way can resolve to int
        - or change to resolve to string without changing anything code elsewhere in program

Two Categories of Secrets

- Hiding complexity so that your brain doesn't have to deal with it unless you're specifically concerned with it
- Hiding source changes so that when change occurs, the effects are localized

Barriers to Information Hiding

- In some cases, info hiding is truly impossible, but most of the time the barriers are just mental blocks built on old habits
- Don't use 100, use MAX_EMPLOYEES
- Don't interleave user interaction, really needs to be in single class, package or subsystem that can be changed without affecting the whole system
- Don't use circular dependencies
- Global data is cancer haha want class data
- Don't worry about perceived performance penalties
    - until you can measure systems performance don't sweat it
    - as long as your design his highly modular, even if changes arise it will be easy to modify

Value of Information Hiding

- **Found that programs that use information hiding are 4x easier to modify**
- **Get in the habit of asking "What should I hide?"**

Identify Areas that are Likely to Change

- A great study of designers found that one attribute they had in common was the ability to anticipate change
- The goal is to isolate unstable areas so that the effect of a change will be limited to one routine, class, or package

- Steps
    - Identify items that seem likely to change
        - if reqs done well, they include a list of potential changes and the likelihood of each change
    - Separate items that are likely to change
        - compartmentalize each volatile component into separate classes
    - Isolate items that seem likely to change
        - design interfaces to be insensitive to the potential changes
        - any other class using the changed class should be unaware that the change occurred

Typical areas:

- Business rules
    - tax laws, union renegotiates a contract, etc
- Hardware dependencies
    - screens, printers, keyboards, gpus, cpus
- Input and Output
    - application that creates its own data format will probably change
    - user level inputs and output formats will also change
    - positioning of fields on page, number of fields, sequence, etc
- Nonstandard language features
    - using extensions is a double edge sword b/c not be avail in all environments
- Difficult design and construction areas
    - first pass might be done poorly so want to have flexibility to make changes
    - compartmentalize and make change impacts minimal
- Status variables
    - Don't use booleans, use enumerated type so can add options
    - use access routines rather than checking variable directly
- Data size constraints
    - use named constants

Anticipating Different Degrees of Change

- When thinking about potential changes to a system, design system so that the effect or scope of the change is proportional to the change that will occur
- Start at core functionality that wont change, then incrementally identify potential changes
    - both functionally
    - and qualitatively (threading, localizable)

Keep Coupling Loose

- Coupling describes how tightly a class or routine is related to other classes or routines
- The goal is to create classes and routines with small, direct, visible and flexible relations to other classes and routines - "loose coupling"
    - sin() --> loosely coupled

    - InitVars(var1, var2, var3, var4) --> tightly b/c with all variables that need to be passed, the calling module practically knows whats happening inside
- Avoid classes that depend on each others use of the same global data
    - these are even more tightly coupled

Coupling Criteria

- Size, the number of connection b/w modules
    - small is beautiful b/c its less work to connect other modules to a module that has a smaller interface
- Visibility, the prominence of the connection b/w two modules
    - Want to make connections as obvious and well defined as possible
- Flexibility, how easily you can change the connection
    - want USB sticks not wire and solder
- In short, the more easily other modules can call a module, the more loosely coupled it is

Kinds of Coupling

1) Simple-object coupling (Instantiate)
    a. A module is simple-object coupled to an object if it instantiates that object
    b. This kind of coupling is fine
2) Object-parameter coupling
    a. Object1 requires Object2 to pass it an Object3
    b. Means that Object2 needs to know about Object3
3) Semantic coupling
    a. "The most insidious kind of coupling"
    b. Module makes use of another modules inner workings
    c. Arise from poor information hiding

Look for Common Design Patterns

- Abstract Factory
    - Supports creation of sets of related objects by specifying the kind of set but not the kinds of each specific object
- Adapter
    - Converts the interface of a class into a different interface
- Bridge
    - Builds an interface with an implementation in such a way that either can vary without the other varying
- Composite
    - Consists of an object that contains additional objects of its own type so that client code can interact with top level object and not concern itself with all the detailed objects
- Decorator
    - Attaches responsibilities to an object dynamically, without creating specific subclasses for each possible configuration of responsibilities

- Facade
    - Provides an interface to code that wouldn't otherwise offer a consistent interface
- Factory Method
    - Instantiates classes derived from a specific base class without needing to keep track of the individual derived classes anywhere but the Factory Method
- Iterator
    - A server object that provides access to each element in a set sequentially
- Observer
    - Keeps multiple objects in synch with one another by making object responsible for notifying the set of related objects about changes to any member of the set
- Singleton
    - Provides global access to a class that has one and only one instance
- Strategy
    - Defines a set of algorithms or behaviors that are dynamically interchangeable with each other
- Template Method
    - Defines the structure of an algorithm but leaves some of the detailed implementation to subclasses
- **Need to be conscious not to change or force code too much to fit a pattern**
- **Also, can't use a pattern because you want to try it out, only use pattern when it is the correct solution for the task**

Other Heuristics

- Aim for Strong Cohesion
    - Refers to how closely all the routines in a class or all the code in a routine support a central focus - how focused the class is
    - Typically looked at at the routine level, and rebranded as abstraction at the class level
- Build Hierarchies
    - Just like hierarchical understanding of the program to make it easier to grasp
- Formalize Class Contracts
    - Like, you need to give me x, y, z in these formats, and I will return a, b, c
- Assign Responsibilities
    - Asking what each object should be responsible for is important
- Design for Test
    - Good because can help formalize class interfaces to make easier to test
- Avoid Failure
    - Always look for ways the program could fail, instead of just looking at the successes
- Choose Binding Time Consciously
    - The time a specific value is bound to a variable
    - Code that binds early tends to be simpler, but less flexible
- Make Central Points of Control
- Consider Using Brute Force
    - A brute force solution that works is better than an elegant one that doesn't
- Draw a Diagram

Keep Your Design Modular

Black box everything

Simple interface, well defined functionality

Design Practices

- Previous section focused on heuristics related to design attributes
    - what you want the completed design to look like
- This section focuses on design practice heuristics, steps that you can take that often produce good results

Iterate

- Design is an iterative process.
- Most subsequent attempts are better than the previous
- "I have discovered a thousand things that don't work" - time not wasted

Divide and Conquer

- Divide the program into different areas of concern and tackle each one individually

Top-Down and Bottom-Up Design Approaches

- Top-Down begins at high level of abstraction, slowly increasing level of detail
- Bottom-up starts with specifics and works towards generalities

Argument for Top Down

- Guiding principal is the human brain can only concentrate on a certain amount of detail at a time
- Think decomposing

Argument for Bottom Up

- Sometimes top-down is so abstract its hard to get started, so you need to work with something more tangible
- Things to keep in mind w/ bottom up
    - Ask yourself what you know the system needs to do
    - Identify concrete objects and responsibilities from that question
    - Identify common objects, and group them using subsystem organization, packages, composition within objects, or inheritance, whichever is appropriate
    - Continue with the next level up, or go back to the top and try again to work down

No Argument, Really

- The key difference is one is a decomposition strategy and the other is a composition strategy

Experimental Prototyping

- Two heads are better than one
    - You informally walk over to co-workers desk and ask to bounce ideas around
    - You and your co-worker sit together in a conference room and draw alternatives on a whiteboard
    - You and your co-worker sit together at the keyboard and do detailed design in the programming language you're using - that is, you can use pair programming
    - You schedule a meeting to walk through your design ideas with one or more co-workers
    - You schedule a formal inspection with all required structure
    - You don't work with anyone who can review your work, so you do some initial work, put it into a drawer, and come back to it a week later
        - You will have forgotten enough that you should be able to give yourself a fairly good review
    - You ask someone outside your company for help: send questions to a specialized forum or newsgroup

How Much Design Is Enough

- Format: Pre-Construction detail, Documentation formality
- Design/construction team, has deep experience in applications area
    - Low, Low
- Design/construction team has deep experience but is inexperienced in the applications area
    - Medium, Medium
- Design/construction team is inexperienced
    - Medium to High, Low-Medium
- Design/construction team has moderate to high turnover
    - Medium, -
- Application is safety-critical
    - High, High
- Application is mission-critical
    - Medium, Medium-High
- Project is small
    - Low, Low
- Project is large
    - Medium, Medium
- Software is expected to have a short lifetime (weeks or months)
    - Low, Low
- Software is expected to have a long lifetime (months or years)
    - Medium, Medium
- **Most problems arise from a lack of design detail, not too much of it**
- **Exploring other design options is a better use of time than polishing design documentation**

Capturing Design Work

- Use Whatever
- Capturing Design Work
    - Comments
- Wiki discussions
- emails summaries
- take pictures if you have to
- CRC (Class, Responsibility, Collaborator) cards
- UML diagrams