Design Principles and Principles of Design

Dr. Stuart Faulk
Computer and Information Science
University of Oregon

Overview

- Review of why we design
- The role of design principles
- General Design Principles
  - Separation of concerns
  - Abstraction
  - Rigor and Formality*
  - Simplicity*
- More Specific Software Principles
  - Modularity
    - Information hiding
    - Abstraction (again)
  - Most solid first

System “Design” Implies…

- A Design (n) implies we have:
  - Goals – What are we trying to achieve?
  - Choices – What design choices can we make?
  - Assumptions or Constraints – How are the design choices limited?
- To Design (v) implies:
  - Processes – What sequence of activities and work products are needed to achieve the goals?
  - Methods – What technologies, techniques and notations will we use?
- Design principles embody goals/assumptions and help guide choices

What are Principles?

- Principle (n): a comprehensive and fundamental rule, doctrine, or assumption
- Design Principles – rules that guide developers in making design decisions consistent with overall design goals and constraints
  - Guide processes
  - Embodied in methods and techniques (e.g., for decompositions)
  - Disc: Which principles drive OOD?

Key Points

- Principles are associated with particular design goals or kinds of goals
  - Must understand the relationship between the principle and the goal to apply properly
  - Principles can overlap or conflict just as goals can
- Principles may exist and/or be applied at different levels of specificity depending on the problem context
  - Separation of concerns – general problem solving strategy applicable everywhere
  - Transparency – deals with a specific problem of abstract machine layers

Principles vs. Heuristics

- Heuristic: involving or serving as an aid to problem-solving by experimental and especially trial-and-error methods
  - AKA: rules of thumb
  - Often method or problems specific version of a principle
- Heuristics arise from practice
- Principles arise from the essence of the problem
  - Can be extrapolated from nature of the problem
  - Universally applicable where assumptions hold

Principles vs. Heuristics (2)

- Approaches to system decomposition
- Heuristics for choosing objects
  - Find the nouns in the problem description
  - Identify the “things” that act
- Principles
  - Information hiding – encapsulate likely changes
  - Abstraction – encapsulate details not directly relevant to the problem
- Heuristics are generally method specific and weaker (less reliable) but easier to apply

Example Heuristics

- Underline the nouns
- Identify causal agents
- Identify coherent services
- Identify real-world items
- Identify physical devices
- Identify essential abstractions
- Identify transactions
- Identify persistent information
- Identify visual elements
- Identify control elements
- Execute scenarios

Separation of Concerns

Separation of Concerns

- Principle: divide a problem into parts such that each part addresses a distinct concern
- Goal: divide a problem parts that can be addressed separately
- Assumptions
  - A “divide and conquer” strategy
  - Parts will be simpler than the whole
  - Concerns are relatively independent

Typical SE Separations

- Separation in time
  - Basis for life cycle models
- Separation of qualities
  - Security, performance, reliability
- Separation of problem views
  - Data flow vs. calls vs. threading
- Separation of purpose
  - Identify what SW must do (requirements) separately from how it does it (design)

Application

- Very broad applicability if concerns can be cleanly separated
- Clean separation typically requires removing some conflicting concerns
  - Must make certain overall decisions constraining the design space
  - Efficiency vs. correctness – establish correctness then permit only semantic preserving transformations to achieve efficiency
- Doing this right requires skill and experience
  - The single most misapplied principle is SE

Design Applications

- Creating distinct views of a system based on distinct concerns
  - Esp. for software architecture where each view comprises particular set of components, relations, and interfaces
  - “Task” (process, thread) view
    - Shows potentially concurrent tasks, task dependencies (e.g. precedence, exclusion), and synchronization mechanisms
    - Used for scheduling, deadlock detection
  - “Calls” view
    - Shows procedures, call protocol and parameters, and which calls which
    - Used for performance (e.g., bottlenecks)

Caveats

- Must be able to identify and characterize distinct concerns
- Must adequately understand dependencies between different concerns
- If done too soon, may miss issues and opportunities that span concerns (e.g., efficiency, simplicity)
- What makes SoC difficult to apply in software design?

Difficulties

- Complexity and tightly related concerns make SoC difficult to apply
- The most misapplied principle in software engineering
  - People (and some methods) act as if strongly connected issues were distinct
  - Implication: connections that are really there cannot be seen
    - Cannot see the side-effects of design decisions
    - E.g., Improve performance but compromise security
  - Results in loss of control (usefulness of the SE definition)

Abstraction

Abstraction

- General: disassociating from specific instances to represent what the instances have in common.
  - Abstraction defines a one-to-many relationship
  - E.g., one type, many possible implementations
- Modular decomposition: Interface design principle of providing only essential information and suppressing unnecessary details by exploiting commonality.

Examples

- Maps, finite state machines, circuit diagrams
- Abstract data types like stacks (many possible implementations)
- Virtual machine interfaces like the Mac or Windows desktop (common interface to different types of entities including different file types and executable programs)

Design Applications

- Two primary uses

1) Reduce Complexity
   - Goal: manage complexity by reducing the amount of information that must be considered at one time
   - Approach: Separate information important to the problem at hand from that which is not
   - Abstraction suppresses or hides “irrelevant detail” (problem dependent)
   - A form of separation of concerns (different abstractions provide different views)
- Examples: stacks, queues, objects

Design Applications (2)

2) Model the problem domain
   - Goal: leverage domain knowledge to simplify understanding, creating, checking designs
   - Approach: Provide components that make it easier to model a class of problems
     - May be quite broad (e.g., type real, type float)
     - May be very problem specific (e.g., class automobile, book object)
- Usually reduces complexity but not always (different purpose)

Common Confusions

- Euphemism for “vague”
  - Abstractions are precise about what they model
  - Should always be definitive whether an instance is in the set covered in the abstraction
- Euphemism for “high level”
  - Meaningless without describing the relation that defines the partial order
  - “More abstract” usually ill defined
- Euphemism for “untrue” or “not really”
  - An abstraction is not a lie
  - Anything true of an abstraction must be true of its instances

Other General Principles

- Rigor and Formality
  - Should be as rigorous and formal as possible
  - See handout
- Simplicity
  - “Things should be made as simple as possible – but no simpler” – A. Einstein
  - Simplicity is the hallmark of great design

More Specific Software Principles

Modularity
Information hiding
Most solid first

Modularity

- System design implies that we decompose a problem into parts such that it is easier to understand, develop, or check the parts, then their relations, than the problem as a whole
  - Application of separation of concerns
- A system is “well-structured” when
  - Components can be easily mastered individually
  - Connections between components contain little information (I.e., connections are few and assumptions are simple)
- E.g., functions, modules, objects

Design Applications

- Reduce complexity: Components are simpler hence easier to understand, write, show correct
- Ease of change: Can limit changes to small numbers of components
- Ease of construction: Can compose complex system from components rather than underlying language
- Reuse: Can reuse larger units

Decomposition Approaches

- Methods vary most obviously in their method of decomposition
  - Vary in assumptions of what's important hence, goals for the decomposition
  - Vary principles or heuristics guiding decomposition
  - Vary in methods of representation
  - Vary in notion of "good design" and how to check it
  - Vary in binding times

Common Confusions

- One size fits all
  - The same decomposition is unlikely to address all design goals
  - Need to be clear which goals have priority
  - Issue with heuristics
    - Heuristics: Model nouns as objects
    - Which goal(s) is this intended to support?
- Binding time
  - The same decomposition may not be appropriate to distinct concerns over time
  - Need to be clear what is being controlled
  - E.g., design time decomposition vs. requirements or code

Information Hiding

Definitions: Information Hiding

- Information hiding (or encapsulation): Design principle of limiting dependencies between components by hiding information other components should not depend on.
- An *information hiding decomposition* is one following the design principles that:
  - System details that are likely to change independently are encapsulated in different modules.
  - The interface of a module reveals only those aspects considered unlikely to change.
- The details hidden by the module interface are called secrets of the module.

Design Applications

- Maintainability, Ease of Change
  - Limits dependencies between modules to things on the interface
  - Confines aspects of the system that are likely to change to a small set of modules (hopefully, one)
- Manage complexity
  - Reveals only aspects of the system common across implementations

Common Confusions

- Confused with “having something to hide”
  - There are ulterior motives to “hiding” information
- Confused with “unclear,” “imprecise,” or other forms of not providing the information needed by other designers
  - Perceived to make other designer’s job harder because of what they cannot use
  - Harder to understand because one cannot look at the implementation

Abstraction

- Applied at the module level to choose what to put on the module interface
  - Model the problem
  - Manage complexity
- Can be viewed as the obverse of information hiding
  - Both may apply and be used on the same module
  - Both govern what is on the interface and what is not
  - But, they have different goals

Similarities

- An information hiding module necessarily abstracts from its hidden details (e.g., there are typically many possible implementations).
- Similarly, an abstract interface hides those details it abstracts from (i.e., the qualities of instances that are not in common)
- Two sides of the same coin but differing in focus and intent

Differences

- Goals of abstraction focus on simplification and conceptual integrity (e.g., modeling the problem space)
  - Abstractions focus on providing appropriate virtual machines for the problem to be solved.
  - An abstraction is characterized by its interface (visible operations and state)
  - The conceptual integrity of a design depends on choosing a set of consistent abstractions.

Differences

- Goals of information hiding focus on localization (limiting dependencies).
  - Information hiding decisions focus on deciding what information should not be used by other parts of the system (particularly, which aspects are likely to change independently)
  - An information hiding decision is characterized by describing the module's secret.
  - The maintainability of a design depends on limiting dependencies such that changes are localized.

Examples

- In general, one will not suffice to do the job of the other. Both must be considered
- Windows 95/98
  - Provides common abstract interface to data and programs
  - Fails to provide information hiding among programs
    - Common data and programs in registry
    - Installing or uninstalling one program can disable an unrelated program

Examples (2)

- Conversely, a program may hide details but provide poor abstract interfaces
- Windows 3.1
  - Hides differences among files and programs (i.e., user cannot access the information through the windows interface)
  - But, same operation (drag and drop) may or may not work on different types of files
  - Result - still had to know DOS to really use 3.1

Summary

- Use abstraction when the issue is what should be on the interface (form and content)
- Use information hiding when the issue is what information should not be on the interface (visible or accessible)

But...

- Don’t accept *unrealistic* simplifications
- Again, there is a difference between an abstraction and a lie
  - e.g. Stack with infinite capacity
  - Desktop “folder” that’s actually on the web

Most Solid First

Definition

- Most solid first: in a sequence of decisions, the decisions that are least likely to change should be made first
- Goal: reduce rework by limiting the impact of changes
- Application: used to order a sequence of design decisions

Principles in Context

- Modularization divides tasks into
  - Work assignments
  - Units of change
  - But not abstract machine layers
- Abstraction
  - Applied to design of module interfaces
  - E.g., to produce problem-specific types
  - Addresses: What belongs on the interface
- Information Hiding
  - Applied to determine what belongs inside of a module (or different modules)
  - Used to categorize modules
  - Addresses: What should not be depended on
- Most solid first – guides the order of decomposition decisions

Summary

- Principles provide guidance in making “good” design decisions
- Must understand the underlying goal to apply properly
- Typically require experience to apply well (in contrast to heuristics)
- Are method-independent and more durable