Software Design

Robert B. France
Colorado State University
Fort Collins
Colorado

Overview

• Tackling complexity
  – Software design principles
• Software design process phases
• From architecture to detailed design

Design Principles
What is Software Design?

• A software design expresses a solution to a problem in programming language independent terms.
  – This permits a design to be implemented in any programming language.
• Software designs must satisfy requirements
  – Implement the system’s functional requirements and satisfy the constraints imposed by the non-functional requirements; including the budget and schedule
  – Implies that requirements must be expressed in a manner that allows one to check that they have been satisfied.
• Designs should adhere to principles that foster good quality

Design as a series of decisions

• A designer is faced with a series of design issues
  – These are sub-problems of the overall design problem.
  – Each issue normally has several alternative solutions:
    • design options.
  – The designer makes a design decision to resolve each issue.
    • This process involves choosing the best option from among the alternatives.
• Satisfying multiple, possibly conflicting design goals and requirements, may require making tradeoffs

Making decisions

• To make each design decision, the software engineer uses:
  – Knowledge of
    • the requirements
    • the design as created so far
    • the technology available
    • software design principles and ‘best practices’
    • what has worked well in the past
Design space

- The space of possible designs that could be achieved by choosing different sets of alternatives is often called the design space
  - For example:

```
client-server

monolithic

separate
user interface
layer for
client

no separate
user interface
layer for client

fat-client

thin-client
```

Controlling Complexity

- Some factors contributing to complexity
  - Inherent complexity of novel software systems
  - Need to address concerns related to safety, security, reliability, flexibility, performance, etc., during design
  - Changing requirements and implementation platforms
  - Need to develop software that will be deployed on widely distributed platforms
  - Accidental complexity: Lack of adequate software tools, languages and development environments to support programming-in-the-large
    - Similar to ancient Egyptian building the great pyramids, or the ancient Chinese building the Great Wall — it can be done, but requires significant effort and resources.
  - Challenge: How do we manage complexity during design?

Design Principles

- Design principles capture experience related to tackling complexity
- Principles are the basis for software design methods and techniques
  - Methods and techniques are often packaged in a methodology (e.g., Fusion OO methodology developed by HP)
  - Methodologies can be enforced by tools
- Major design principles
  - Separation of concerns, Design for change, Reusability, Generality, Incrementality
Separation of Concerns Principle

- Allows one to solve problems by solving sub-problems.
- Can result in separation of responsibilities
  - Basis for dividing work into work assignments that can be carried out in a relatively independent manner.

Types of Concerns

- **Separation in Time**: eliminates overhead arising from activity switching
- **Separation by Qualities**: focus on a particular quality at any given time, e.g., focus on getting system correct before improving performance
- **Separation by Views**: allows different views to be analyzed separately, e.g., class model and use case views.

Modularity

- Separation of parts: 2 aspects
  - Decomposition: decomposing a complex system/problem
  - Composition/Synthesis: synthesizing a system from existing modules
- Proper modularity helps:
  - Confine search for errors
  - Limit the impact of change
- A design is described in terms of modular units and the relationships between the units
  - Modular units are referred to as modules
Types of modules

- Examples of modular units (modules):
  - Design-level modular units
    - Subsystems
    - Components
    - Design classes
  - Code-level modular units
    - PL classes (e.g., Java classes, C++ classes)
    - Functional modules (e.g., C functions, Pascal procedures, Modula 2 modules)
- A design is a graph of modular units.
  - Modules: functional abstraction, object, data pool.
  - Relationships: uses, is a part of, message passing, procedure call

Types of Module Relationships

- USES relationship
  - M1 USES M2 if correct execution of a service provided by M2 is necessary for M1 to complete a specified task.
  - M1 is the client; M2 is the server
- IS_ASSOCIATED_WITH relationship: class A is associated with class B
- IS_A_COMPONENT_OF relationship
  - M1 IS_A_COMPONENT_OF M2 means that M2 is implemented by aggregating several modules, one of them being M1.
- IS_A_SPECIALIZATION_OF relationship
- Use case relationships: include, extend

Abstraction

- Process of focusing on important aspects while ignoring irrelevant detail.
  - In SE design models are abstractions of code
- Functional abstraction: Implementation of functional concept is hidden from user
  - Given a sine function we know the number and types of inputs to provide and the type of output produced
- Data abstraction: Details of data implementation hidden from user
  - Abstract Data Type: type defined solely in terms of operations that manipulate instances of type
Design for Change Principle

• Anticipate how systems will evolve and design to accommodate change.
  – Lack of attention to this principle can result in changes that make system unstructured and difficult to understand and maintain.
• Plan for changes in the technology or environment so the software will continue to run or can be easily changed
  – Avoid using early releases of technology, software libraries that are specific to particular environments, undocumented features or little-used features of software libraries, software or special hardware from companies that are less likely to provide long-term support
  – Use standard languages and technologies that are supported by multiple vendors

Design For Change - 2

• What can change?
  – algorithms, data representations, hardware platform, social environment
• Techniques
  – Information hiding: a module reveals only what is needed from it by other modules.
  – Encapsulation: information about a single concept is localized in one module.

Reuse Principle

• Design the various aspects of your system so that they can be used again in other contexts
  – Generalize your design as much as possible
  – Follow the preceding three design principles
  – Design your system to contain hooks
  – Simplify your design as much as possible
Generality Principle

- Concerns identifying a general problem from a specific problem.
- General problem may be less complex.
- Solving general problem increases potential for reuse.
  - Success of tools such as word-processors, spreadsheets attributed to their generality
  - Increased likelihood that problem may be solved by an existing package

Incrementality Principle

- Concerned with incremental development of software.
- Desired goal is reached by successively closer approximations.
- Each approximation (version) is reached by an increment of the previous approximation.

More on modularity - Building designs with good modular structure

- A good design should have
  - modules with high cohesion, and
  - low coupling between modules
- Cohesion: The degree of interaction within a module
- Coupling: The degree of interaction between two modules
  - Reflects the degree to which a module’s behavior is dependent on another module’s behavior.
Module Cohesion

- **Degree of cohesion** can range from coincidental cohesion (worst case) to Functional or Informational cohesion (best case).
- **Coincidental cohesion**: Module performs multiple, unrelated actions.
  - Example module: Read in average rainfall, calculate average, read in temperatures, output temperatures, …
  - Problems: Difficult to maintain, cannot reuse, difficult to understand
- **Logical cohesion**: Module performs a set of related actions, one of which is selected by the client.
  - Example: Module that performs all I/O; code for all I/O; code for input only; code for output only; code for disk I/O; code for tape I/O; code for keyboard input; …
  - Problems: Difficult to understand interface, difficult to maintain, difficult to reuse

Cohesion - 2

- **Temporal cohesion**: Module performs a series of actions that are related in time
  - Example module: Open master file, open transaction file, print file, initialize sales table, read transaction record, …
  - Problems: Impact of change not likely to be localized, difficult to reuse
- **Procedural cohesion**: Module performs a series of actions related by the sequence of steps to be carried out.
  - Example module: Read part number from database, update repair record, write updated record to database, …
  - Problems: Difficult to reuse, parts are weakly related
- **Communicational cohesion**: Module performs a series of actions related by the sequence of steps to be carried out and all actions are performed on the same data.
  - Example module: read record, update record, write record
  - Problem: Difficult to reuse

Cohesion - 3

- **Informational cohesion**: Module performs a set of actions related to a data structure
  - Example module: A Stack ADT
  - Benefits: Localizes information around data structure; Facilitates reuse of data structures
- **Functional cohesion**: Module performs only one action or has a single functional task.
Cohesion Examples

Examples of modules with their degree of cohesion indicated

Coupling

- Range from content coupling (worst case) to data coupling (best case)
- **Content coupling**: When a module M1 directly references the contents of module M2, content coupling exists.
  - Examples: M1 branches to a section of code in M2; M3 modifies a statement in M4
  - Problem: Changes to M2 requires that M1 be examined for possible changes.
- **Common coupling**: When modules M1 and M2 have access to the same global data then common coupling occurs.
  - Example: functions/procedures accessing global variables
  - Problems: Difficult to reuse modules (global variables will have to be supplied when reused); Side effects occur; Change to global data requires change to all modules that access the data; A module may be exposed to more data than it needs

Common Coupling (Problems)

- Code not easy to read

```c
while (global_data == 0) {
    if (x > 25)
        module3();
    else
        module4();
}
```

What is global_data???
**Control Coupling**

- **Control coupling**: When a module M1 passes a control element to module M2 control coupling occurs.
  - Control element tells M2 what action to take; M1 controls behavior of M2
  - Module M1 sends a control message to M2 that means “I am unable to complete task b; generate the error code p”
  - Note that control passing is different than data passing; if the message means “I am unable to complete task b” then this is data passing because the signal does not mandate the action M2 must take when it receives the message.
- Problems: Modules are not independent; a module controls the behavior of another module; if M1 sends a control message to M2 then module M1 must be aware of M2’s internal structure and logic.

**Stamp & Data Coupling**

- **Stamp coupling**: When a module M1 passes a data structure to M2, but M2 uses only part of the data in the structure, stamp coupling occurs.
  - Example: Passing a pointer to a record structure that contains more data than is required
  - Problem: Module M2 is exposed to more data than it needs.
- **Data coupling**: When a module M1 sends only the data needed by module M2, data coupling occurs.
  - Desired form of coupling

**Design Process Phases**
Design Overview

• Two major phases
  – Architectural design
    • Concerned with identifying subsystems and their relationships.
  – Detailed design
    • Concerned with refining architecture
      – define algorithms, threads and data structures
• Static and Behavioral design views
  – Static view: Concerned with identifying solution concepts and their relationships.
  – Behavioral view: Concerned with defining interactions between objects and behavior of individual objects

Moving from Problem to Solution

Requirements
- System attributes (non-functional requirements, anticipated changes, design constraints)
- Use Cases
- Requirements class diagram

Initial Design
- Design goals
- Architecture
  • subsystem structure
  • subsystem interfaces

Design Strategies

• Functional Decomposition
  – Develop solution in a top-down manner
  – Focus is on defining system services
• Object Composition
  – Develop solution by composing objects
  – Focus is on defining objects and their operations
**Functional Decomposition Process**

- Architectural Design: Decompose solution into subsystems that are functional abstractions and define their interfaces.
  - Abstract Specification: Specify the services provided in each interface.
    - Specified in terms of pre- and post-conditions
- Detailed Design: Decompose subsystems into functional modules
  - Specify algorithms, data structures for leaf modules

**Object-Oriented Design Process**

- Architectural Design: Group objects into subsystems and define interfaces between subsystems.
  - Static view: Define relationships between subsystems
  - Behavioral view: Define services in interfaces and interactions between subsystems

**Object-Oriented Design Process - 2**

- Detailed Design: Define how services defined in subsystem interfaces are carried out by objects in subsystems
  - Develop class diagrams
  - Develop behavioral models
  - Define methods
From architecture to detailed design: An Overview

A Design Process

• Identify design goals
• Create an initial architecture
• Elaborate subsystems
• For distributed systems: Map subsystems to processors and components
• Identify persistent elements
• Specify control structure
• Specify exception handling features

Architectural Design Products

Solution structure specification consists of:
• Design goals: prioritized system attributes that should be optimized
• Software architecture: model of software structure in terms of subsystems and their relationships.
Identifying Subsystems

- **Use Case based identification**
  - Assign objects associated with a use case into the same subsystem
  - Objects that are used in more than one use case can be put into a separate subsystem
  - Reduce subsystem coupling and improve subsystem cohesion

- **Function-based identification**
  - Group use cases along functional lines and put objects involved in a group into the same subsystem

Subsystem Decomposition

- Subsystems have interfaces that describe provided and required services
  - Provided service: a service offered by the subsystem
  - Required service: a service needed by the subsystem that must be provided by another subsystem

- Objects in a subsystem collaborate to carry out the provided services

Architectures across different levels of abstraction

- **Logical Architecture**
  - Organization of a system into logical units (e.g., layers, subsystems)
  - Logical units do not necessarily result in units of implementation (e.g., Java packages, components)

- **Process Architecture**
  - Grouping of tasks into computational units (e.g., threads, processes)

- **Implementation (Deployment) Architecture**
  - Organization of a system into physical units (e.g., source files, Java packages, components)
Logical Architectures

• Layered
  – System organized into layers of functionality
    • Presentation/Application/Persistence (Data)
  – Each layer uses the services of a lower layer
  – Supports separation of concerns
• Subsystems
  – System decomposed into peer units
  – Layers can be decomposed into layers

Multi-Tier Layered Architectures

• Separate presentation and application logic, and other areas of concern.

A Simple Logical Architecture

• In the UML, the logical partitioning is illustrated with package diagrams.
Mapping Subsystems to Processors and Components

- Select hardware configuration
  - includes selection of OS, middleware, DBMS, other system software
- Allocate subsystems and objects to nodes
  - can result in identification of other objects and subsystems used for communicating among nodes

Using Existing Components

- Design patterns provide solutions for incorporating existing components into an architecture
  - Adapter pattern: make interface of component compatible with subsystem interface expectations.
  - Bridge pattern: allows one to vary implementation independently of interface

Persistency

- Identification of persistence dependent on application needs.
- Database concerns can be encapsulated in a subsystem
  - ODBC, JDBC can provide abstractions (subsystem interfaces) for relational DBs
Specifying Global Control Flow

- Control flow: organization of actions in system
  - Procedure-driven control: operations wait for input when needed
  - Event-driven control: main loop waits for events, which are then dispatched to appropriate objects.
  - Threads: system can handle events concurrently

What’s next?

- Design class diagrams
- Describing behavior using the UML
- Design patterns