Software Design

With UML, Design Patterns, Components and Software Frameworks

Gerson Sunyé
University of Nantes
gerson.sunye@univ-nantes.fr
http://sunye.free.fr

Agenda

- Introduction
- Design Process
- Key Principles
- Conclusion

Introduction

Software Design

"Art and science of conceiving the structure of software systems."

The Designer

"The designer is the one that simplifies, gives a strong and invisible personality to what he creates, prunes, purifies, clutters, and creates suitable products."

[Jasper Morrison]

Two Approaches

- There are two ways of designing software:
    - One way is to make it so simple that there are obviously no deficiencies
    - and the other way is to make it so complicated that there are no obvious deficiencies.
- The first way is far more difficult to achieve.
- [C. A. R. Hoare]

Software Design

Techniques

Objects, Components, Patterns, MDE

Plate-formes

Android, WebOS, iPhone, Symbian

Middleware

CORBA, SOAP, Peer-to-peer, Web services

Quality Constraints

Reusability, performance, etc.

Requirements

Specification

SOFTWARE DESIGNER 

Domain

Domain analysis

Methods

RUP, XP, TDD

Standards

ISO, OMG, W3C, IEEE

Frameworks

J2EE, .NET, XML, Zope, Rails

Software Design Model

Open source

Apache, Mozilla, Sourceforge

Different Steps

1. Architecture.
2. Component specification (preliminary design).
3. Detailed design.

Architecture

- Strategic choices for system implementation.
- More engineering than computer science.
- Architectural patterns application.

Components Specification

- High-level description of the collaboration between the system major components.
- System boundaries definition.
- Component desired behavior description.

Detailed Design

- Component internal description.
- Preparation of the target programming language projection.
- Design pattern application.

Other Design Aspects

- User interface design.
- Protocol design.
- Database design.
- Algorithm design.
- etc.

Design Process

Design Process Steps

1. Preliminary design
    1. Requirements/Domain decomposition into components.
2. Detailed design
    1. Component specification
    2. Additional component decomposition, if needed.

Low-Level Steps

1. List the hard decisions and decisions likely to change
2. Design a component specification to hide each such decision
   - Make decisions that apply to whole program family first
   - Modularize most likely changes first
   - Then modularize remaining difficult decisions and decisions likely to change
   - Design the uses hierarchy as you do this (include reuse decisions)
3. Treat each higher-level component as a specification and apply above process to each
4. Continue refining until all design decisions:
   - are hidden in a component
   - contain easily comprehensible components
   - provide individual, independent, low-level implementation assignments

Best Practices

- Separate interface from implementation.
- Separate orthogonal concerns: do not try to connect what is independent.
- The "what" before the "how".
  - First specify what a component should do, then how it does it.
- Work on different abstraction levels.
- Use rapid prototyping.

Bad Practices

- Depth-first design
- Requirement direct refinement.
- Potential changes misunderstanding.
- Design too detailed.
- Ambiguous design.
- Undocumented design decisions.
- Inconsistent design.

Key Principles

Design Principles

1. Decomposition
2. Abstraction
3. Information concealment
4. Cohesion
5. Decoupling
6. Reusability
7. Reuse.
8. Obsolescence anticipation
9. Portability
10. Testability
11. Simplicity
12. SOLID principles

Decomposition

- Complexity control by the decomposition of problems into sub-problems.

- Divide and conquer approach, common to all design techniques.

Abstraction

- Complexity control by the amplification of the essential features and the suppression of the less important details.
- Allows to postpone design decisions.
- Types of abstraction: functional, structural, control, etc.

Information Concealment

- Important means to achieve abstraction.
- Design decisions that are likely to change should be hidden behind interfaces.
- Components should communicate only through well-defined interfaces.
- Interface should be specified by as little information as possible.
- If component internal decisions change, clients should not be affected.

Typical Information Concealment

- Data representation
  - Data types, etc.
  - Algorithms
    - Data search and sorting
- Input and output formats
  - Machine dependencies, e.g., byte-ordering, character codes
- Lower-level interfaces
  - Ordering of low-level operations
  - Policy/Mechanism separation
    - Multiple policies for one mechanism.
    - Same policy for multiple mechanisms.

Cohesion

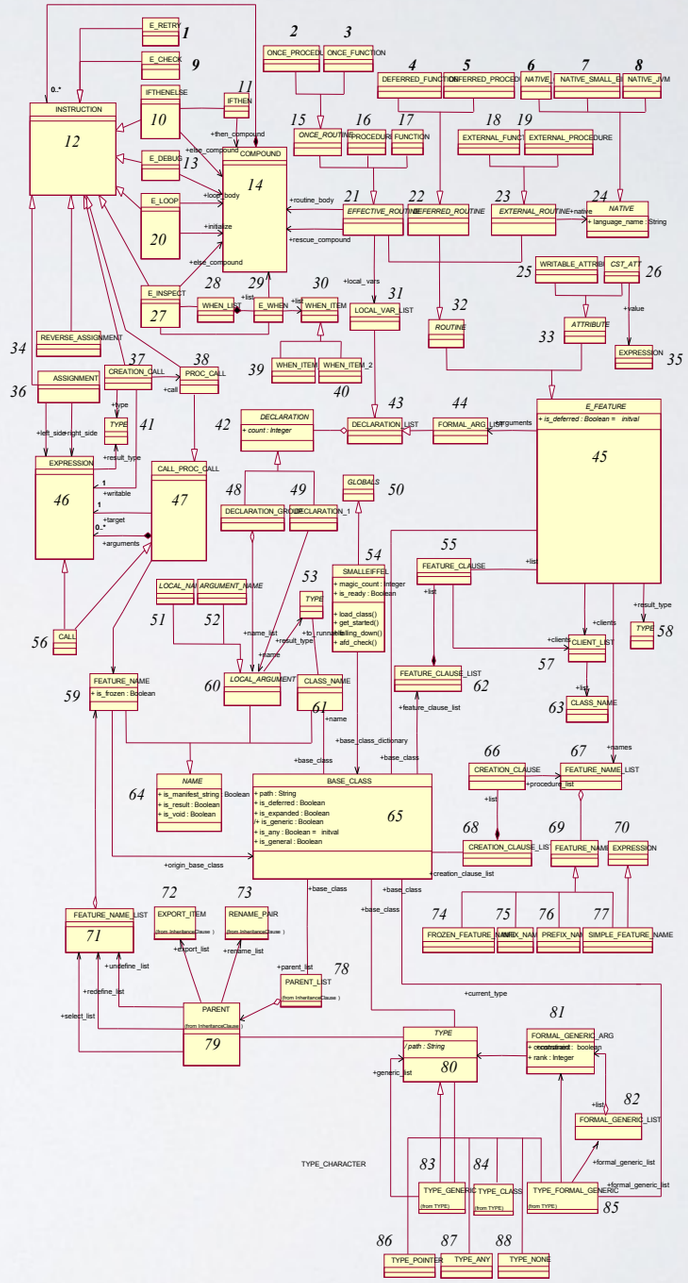
- Cohesion should be increased where possible.
- Types of cohesion: functional, layer, de communicational, sequential, procedural, temporal, utility.

Keeping Together

- All the code that performs a particular task (functional cohesion).
- All the code that provides or uses a set of related services (layer cohesion).
- All code that access or modify certain data (communicational cohesion).
- A sequence of operations, in which one operation provides input to the next (sequential cohesion).
- A set of operations that are used one after another (procedural cohesion).
- Operations that are performed at the same execution phase (temporal cohesion).
- Operations which cannot logically be placed in other cohesive units (utility cohesion).

Decoupling

- Coping should be reduced where possible.
- Interdependencies impacts maintainability, testability, and simplicity.



GNU Eiffel Compiler

Types of Coupling

- Content: when a class (module) modifies the content of another class. Violates encapsulation, Law of Demeter.
- Common: when classes share global variables.
- Control: when a parameter controls the behavior of an operation.
- Stamp: when a class of a component becomes the parameter type of another component.
- Data: when parameter types are primitive types.
- Operation call: when an operation calls another.
- Datatype use: when a component uses a datatype defined in another component.
- Merge or import: when a component includes another (merge) or when a class imports another.
- External: when a component depends on external artifacts: operating system, hardware, libraries, etc.

Reducing Coupling

- Dependencies among component should form a directed acyclic graph.
- Dependency between two components must follow the direction of stability (a component must always depend on a more stable one).

Reusability

- Reusability should be increased where possible.
- Components should be designed to work on different contexts.
- Generalize design as much as possible:
  - Use Frameworks, Patterns, and UML Collaborations.
- Design the system to contain hooks.
- Keep the design as simple as possible

Reuse Analysis, Design, and Code

- Reuse existing artifacts when possible, to take advantage of existing investment.

- Use Frameworks, Patterns, and UML Collaborations.

- Complementary to the principle of reusability.

Obsolescence Anticipation

- Avoid:
  - Immature technologies.
  - Undocumented features.
  - Software and Hardware without long-term support provision.
- Plan technology changes and adopt technologies that are supported by different vendors.

Portability

- Develop on and for different platforms.

- Avoid platform-specific frameworks or libraries.

Testability

- The internal state of a component should be accessible by external programs.

- Design components to be used directly by external clients, without user interfaces.

Simplicity

- The KISS principle [US Navy, 1960] :
  - Keep it simple, stupid!
- Most systems work best if they are kept simple rather than made complicated
- Simplicity should be a key goal in design and unnecessary complexity should be avoided.

SOLID

- Single responsibility [Robert Martin]:
  - a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class).
- Open/Closed principle [Bertrand Meyer]:
  - Software entities should be open for extensions but closed for modification.

SOLID

- Leskov substitution principle [Liskov 1994]:
  - Class instances can be used throughout the superclass interface, without notifying its clients.
- Interface-segregation [Robert Martin 2002]:
  - Many client-specific interfaces are better than one general-purpose interface.
- Dependency inversion [Robert Martin 2003]:
  - A client should depend upon abstractions and not upon concretions.

Conclusion

Conclusion

- Design is a creative activity that goes beyond diagrams drawing.

- A good design requires experience.

Références

- Real-Time UML: Developing Efficient Objects for Embedded Systems. Bruce Powel Douglass

- Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Bruce Powel Douglass

- Software Patterns. James Coplien. SIGS Books. New York, 1996.

- Smalltalk Patterns: Best Practices. Kent Beck. Prentice Hall, 1997, 256 pp., ISBN 0-13-476904-X

- Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm,Ralph Johnson, and John Vlissides. Addison Wesley. October 1994.

- Mohamed E. Fayad, Douglas C. Schmidt, Ralph Johnson (September 2009). Implementing Application Frameworks: Object-Oriented Frameworks at Work.