Aspect-Oriented Software Development

Introduction and Motivation

Jianjun Zhao (赵建军)
Department of Computer Science and Engineering
Shanghai Jiao Tong University
zhao-jj@cs.sjtu.edu.cn
http://cse.sjtu.edu.cn/~zhao/
Table of Contents

- Elements of software engineering
- Key principle of software engineering
- Key concepts of AOSD
Short Survey

- Survey – show of hands
  - Who has ever heard of Aspect-Oriented Software development?
  - Java language
  - UML
  - AspectJ
  - Design Pattern
Impact of AOSD on Society...

- MIT Technology Review lists AOP as one of the top 10 emerging technologies that will change the world
  - (MIT Technology Review, January 2001)
Software Engineering Elements

- Concepts of software engineering
- Principle of software engineering
Software Engineering Definition

- (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
- (2) The study of approaches as in (1).
  - IEEE Standard 610.12:
  - Another definition:
    - Creating cost-effective solutions to practical problems by applying scientific knowledge building things in the service of mankind
Software Engineering

Student Registration system

Method

application layer

application\textsubscript{n}

method layer

method\textsubscript{m}

model layer

model\textsubscript{0}

support layer

tools\textsubscript{1}

CASE, OO compiler

objects, classes, inheritance
What is a Method?
Software Engineering = Problem Solving

Phases

Problem

Solution

Requirements Analysis

Analysis

Design

Implementation

Testing

What? (client)

What? (domain)

How? (Detailed)

Do

Test
Different Models at Different Phases

**Phases**
- Requirements Analysis
- Analysis
- Design
- Implementation
- Testing

**Models**
- Use case diagrams
- Class diagrams
- Java
Use Case Modeling

- An actor represents any external role (human, hardware device, another system etc.) that interacts with the system.

- Use case is a sequence of actions that a system performs which represents a functional requirement.
Use Case Diagram

- Diagram that shows the system’s actors and use cases together with their relationships.
- Models the context of a system.
  - Actors outside of the system
- Models the (functional) requirements of the system.
  - Use cases
Example

- Use case diagram for library system
Example of Model Layer: UML

UML Structural Models

- Class models
- Package

<table>
<thead>
<tr>
<th>Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>origin</td>
</tr>
<tr>
<td>size</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>open()</td>
</tr>
<tr>
<td>close()</td>
</tr>
<tr>
<td>move()</td>
</tr>
</tbody>
</table>

| Business Rules |
Example of Model Layer: UML Relations

- **Aggregation**
  - Company
  - Department

- **Composition**
  - Person
  - Head

- **Association**
  - Person
  - Company
  - Works for

- **Dependency**
  - Record
  - Display

- **Generalization**
  - Vehicle
  - Car
Example of Method/Process Layer – Unified Process
Example Tool – Rational Rose
Example: Student Registration System

- Application: Student Registration System
  - University consists of several departments. If students fulfill the requirements, students can enroll in the university. Every student can register/withdraw for/from a course.
Use Case Diagram

Student

- register for course
- withdraw from course
- enroll in university
Student Registration Class Diagram
States of a Student

1. Apply [Must be accepted first]
2. Enrolled
3. EnrollInClass (Add a Transcript)
4. Withdraw
5. Registered
6. AddCourse
7. Graduate [All courses must be completed]
Sequence Diagram: Registering for Course
public class Student {

    private String name;
    private int id;

    public String getName () {
        return name;
    }

    public void setName (String str) {
        name = str;
    }

    public int getId() {
        return id;
    }

    public void setId(int i) {
        id = i;
    }

}
Example of Application Layer

- Security System
- Transaction System
- Insurance System
- Newsgroup application
- Car dealer system
- Book reservation system
- University personnel registration system
- Aircraft control system
- Chemical process control
- Games
- On-line tutoring system
- Etc.
Software Engineering Principles

- Modularity
- Abstraction
- Encapsulation
- Information Hiding
- Separation of concerns
- Cohesion & Coupling
Software Engineering Goal

- Develop software systems that are
  - On time
  - Within budget
  - With the required quality
  - Possible large and complex
  - Extendible
Functional vs. Quality Requirements

- Requirements
  - Functional Requirements
  - Quality Requirements
    - External
      - Understandability
      - Robustness
      - Correctness
      - Usability
      - Reliability
    - Internal
      - Reusability
      - Maintainability
      - Portability
      - Testability
      - Adaptability
Quality Issues

- Quality depends on many factors:
  - Adopted Models
  - Adopted Methods
  - Adopted Tools
  - Personnel
  - Organization
  - Management
  - etc.
Principle of Software Engineering
Key Principles

- Abstraction
- Modularity
- Encapsulation
- Information Hiding
- Separation of Concerns

- Apply these principles and provide the right models!
Model of Software Engineering
What is a Model?

- Abstract
  - Which part model?
- Representation
  - How to model?
- Of Reality
  - What to model?
Abstraction

- Focus only on relevant properties of the problem
- Ignore details.
Abstraction

- Helps to cope with complexity of problems and allows to
  - Control the level of detail (abstract-specific)
  - Control the amount of detail
  - Supports communication among stakeholders
Abstraction

- **Process**
  - Denotes the extracting of the essential details about an item, or a group of items, while ignoring the inessential details

- **Entity**
  - Denotes a model, a view or some other focused representation for an actual item
Representation

- **Textual**
  - In natural or formal language

- **Graphical**
  - Using diagrams

- **Mathematical**
  - Formulas
Model = Abstract Representation of Reality

Reality → Abstraction

Representation

Lungs consists of...

Bloodflow = f (....)
Abstraction in Software

- **Procedural abstractions**
  - Procedure: named sequence of instructions

- **Data abstraction**
  - Organize software around data objects
Modularity - Monolithic Software

- Large program
- Consisting of one module
- Difficult to understand
- Difficult to reuse
- Difficult to adapt
Modularity - Decomposition

- ‘Divide and conquer’
- Divide software into separately named and addressable components or modules.
- Problem decomposition $\rightarrow$ Solution Decomposition

$P \rightarrow P_1..P_n \rightarrow S_1..S_n \rightarrow S$
Decomposability
Example: Design of Atomic Transaction System

Problem: Atomic Transaction Design
P1. Transaction Management
P2. Data Management
P3. Adapting Transaction Protocols
P4. Concurrency Control
P5. Failure Management

Solution: Atomic Transaction Design
S1. Transaction Manager
S2. Data Manager
S3. PolicyManager
S4. Scheduler
S5. RecoveryManager
Assume two problems P1 and P2
- \(C(P1)\) and \(C(P2)\) represent complexity of problem P1 and P2 respectively.
- \(E(P1)\) and \(E(P2)\) represent effort to solve problem P1 and P2 respectively.

If \(C(P1) > C(P2)\), then \(E(P1) > E(P2)\) \((1)\)

and from experimentation in human problem solving it follows: \(C(P1+P2) > C(P1)+C(P2)\) \((2)\)

from \((1)\) and \((2)\) it follows that \(E(P1+P2) > E(P1) + E(P2)\)
Composability
Composability Problems

Unable to compose appropriately?
Composability - Example
Cost of Modules/Integration
But What About Time Performance Critical Systems

- Example: Real-time systems, embedded systems
- Design should always be modular to achieve the related benefits
- Code may be developed “in-line”
Encapsulation & Information Hiding

- Module implementation details is inaccessible from outside
Encapsulation - Example

```java
public abstract class Student {
    private Name name;
    private Address address;
    private int id;
    private String gender;
    private Schedule sch;
}
```
Separation of Concerns

- Identification of the various distinct concerns
- Assignment of single concerns to single modules
- At various levels: design and implementation
Dijkstra: Separate Program in Layers...

- **E. W. Dijkstra (1968-2002):**
  - “...Correct arrangement of the **structure** of software systems before simple programming...”
- **Layered Structure**
  - Programs are grouped into layers
  - Programs in one layer can only communicate with programs in adjoining layers
- **Conceptual integrity**
  - Each layer has its own goal
  - With easier development and maintenance

Separation of Concerns

- Identification of the various distinct concerns
- Assignment of single concerns to single modules
- At various levels: design and implementation
Parnas - Design Principles for Decomposition

- Information hiding modules (1972)
  - Identify design decisions that are likely to change
  - Isolate these in separate modules (separation of concerns)

- Different design decisions might require different decompositions.
What is a Concern?

- Are properties or areas of interest
- Can be functional or nonfunctional (quality, systemic)
- At different abstraction levels:
  - Problem domain concerns vs. solution domain concerns
  - Requirements vs. design
  - Design vs. implementation
Separation of Concerns applied

- Separate software development into phases each dealing with specific activities (e.g. requirements, analysis, design, implementation)
- Separation of different artifacts: class, subsystems, attributes.
- Separation of different design views (static, dynamic, implementation, ...)
- Separation of different roles
- ...

...
Benefits of Separation of Concerns

- Supports high cohesion among components
- Supports low coupling among components
- Increases modularity
Cohesion

- Cohesive component performs only **one concern/task**
- Maximize cohesion within a component
  - Required changes can be easily localized and will not propagate

![Diagram showing cohesion]

- Many concerns in one module
  - Low Cohesion

- Each module addresses one concern
  - High Cohesion

- Synchronization
- Recovery
- Authentication
Coupling

- Two components are independent if they do not have interactions
- Highly coupled components have many dependencies/interactions
- Minimize coupling between components
  - Reduces complexity of interactions
  - Reduces ‘ripple’ effect
Coupling and Dependency

Uncoupled

Loosely Couple: Some Dependencies

Highly Couple: Many Dependencies

'ripple' effect
Advantages of Separation of Concerns

- Understandability
- Maintainability
- Extensibility
- Reusability
- Adaptability

- Separation of Concerns directly supports quality factors.
- Lack of separation of concerns directly negatively impact quality factors.
Principles of Modeling - Summary

- Select the right model to the problem
  - Abstraction from domain
- Decompose the problem into logical pieces
  - Decomposition
- Modules should hide unnecessary detail
  - Encapsulation (packaging)
  - Information hiding
- Separation of Concerns
  - Do the modules represent single concerns?
  - Low coupling, high cohesion?
Key Concepts of AOSD

- Crosscutting
- Scattering and Tangling
- Aspect
Exercise-Figure Editor Example

- A figure consists of several figure elements. A figure element is either a point or a line. Figures are drawn on Display. A point includes X and Y coordinates. A line is defined as two points.

- Exercise: Provide object-oriented design in UML. Provide also the operations.

- Apply principles of abstraction, separation of concerns, modularity, encapsulation. Ensure that the components are cohesive and loosely coupled.
Crosscutting Concern - Example

Components are
- Cohesive
- Loosely Coupled
- Have well-defined interfaces (abstraction, encapsulation)

Nice Modular Design!
Well done!

I would like one extension. Notify Screenmanager if a figure element moves...
Crosscutting Concern - Example
**Crosscutting Concerns**

- Concerns that naturally tend to be scattered over multiple components
- Which cannot be localized into single units (components, objects, procedures, functions)...
- If not appropriately coped with:
  - Scattered over multiple components
  - Tangled code per component
Crosscutting, Scattering and Tangling

- **Crosscutting**
  - Concern that *inherently* relates to multiple components
  - Results in scattered concern and tangled code

- **Scattering**
  - Single concern affects multiple modules

- **Tangling**
  - Multiple concerns are interleaved in a single module
Aspects Visualized - Tracing

Tracing concern is scattered over many modules.
In every module tangled code.
The Cost of Crosscutting Concerns

- Reduced understandability
  - Redundant code in many places
  - Non-explicit structure
- Decreased adaptability
  - Have to find all the code involved
  - Have to be sure to change it consistently
  - Have to be sure not to break it by accident
  - New concerns cannot be easily added
- Decreased reusability
  - Component code is tangled with specific tangling code
- Decreased maintainability
  - ‘Ripple effect’
Example of Crosscutting Concerns

- Synchronization
- Real-time constraints
- Error-checking
- Object interaction constraints
- Memory management
- Persistency
- Security
- Caching
- Logging
- Monitoring
- Testing
- Domain specific optimization
- ...

Many crosscutting concerns may appear in one system.

Example: Distributed System Design

- Component interaction
- Synchronization
- Remote invocation
- Load balancing
- Replication
- Failure handling
- Quality of service
- Distributed transactions
What to Do...?

Ne yapsam acaba...?!
Historical Context

- Crosscutting concerns are new type of concerns that have not been (appropriately) detected/handled before.
- No explicit management until recently at programming level
- No explicit consideration in design methods
- No explicit consideration in process
- No explicit consideration in case tools

BUT:
- Aspects are present, and severely reduce the quality of software if not appropriately managed.
Aspect-Oriented Software Development

- Provides better separation of concerns by explicitly considering crosscutting concerns (as well)
- Does this by providing explicit abstractions for representing crosscutting concerns, i.e. aspects
- And composing these into programs, i.e. aspect weaving or aspect composing.
- As such AOSD improves modularity
- And supports quality factors such as
  - Maintainability
  - Adaptability
  - Reusability
  - Understandability
- ...

Basic AOSD Technologies

- Composition Filters (since 1991)
  - University of Twente, The Netherlands
- AspectJ (since 1997)
  - XEROX PARC, US
- DemeterJ/DJ (1993)
  - Northeastern University, US
- Multi-dimensional separation of Concerns/HyperJ (1999)
Conclusion

- Crosscutting concerns are typically scattered over several modules and result in tangled code.
- This reduces the modularity and as such the quality of the software system.

- AOSD provides explicit abstractions mechanisms to represent these so-called aspects and compose these into programs
- This increases the modularity of systems.