Detailed Design

Overview

• What is detailed design?
• What is OO design?
• How should we do OO design?
Detailed Design

• To decompose subsystems into modules
• Two approaches of decomposition
  – Procedural
    • system is decomposed into functional modules
      which accept input data and transform it to
      output data
    • achieves mostly procedural abstractions
  – Object-oriented
    • system is decomposed into a set of
      communicating objects
    • achieves both procedural + data abstractions

Abstraction

• To focus on important, inherent properties
  while suppressing unnecessary details
  – Permits separation of concern
  – Allows postponement of design decision
• Two abstraction mechanisms
  – Procedural abstraction
    • Specification describes input/output
    • Implementation describes algorithm
  – Data abstraction
    • Specification describes attributes, values
    • Implementation describes representation and
      manipulation
OOD

• To identify responsibilities and assign them to classes and objects
  • Responsibilities for doing
    – E.g., create an object, perform calculations, invoke operations on other objects
  • Responsibilities for knowing
    – E.g., attributes, data involved in calculations, parameters when invoking operations

How Do Developers Design Objects?

• Code
  – Design-while-coding, ideally with power tools such as refactorings. From mental model to code
• Draw, then code
  – UML Diagrams
• Only draw
  – The tool generates everything from diagrams
**How Much Time Spent Drawing UML before Coding?**

- Spend a few hours or at most one day (with partners) near the start of the iteration
- Draw UML for the hard, creative parts of the detailed object design
- Stop and transition to coding
- UML drawings
  - inspiration as a starting point
  - the final design in code may diverge and improve

**Work Results**

- **Dynamic models**
  - help design the logic or behaviors of the code
  - UML interaction diagrams
    - (Detailed) sequence diagrams, or
    - Communication diagrams
- **Static models**
  - help design the definition of packages, class names, attributes, and method signatures
  - (Detailed) UML class diagrams
Guidelines

• Spend significant time doing interaction diagrams, not just class diagrams
• Apply responsibility-driven design and GRASP principles to dynamic modeling
• Do static modeling after dynamic modeling

UML Interaction Diagrams

• To illustrate how objects interact via messages
• Two types of interaction diagrams
  – Sequence diagrams
  – Communication diagrams
Sequence diagram

• Illustrate interactions in a kind of fence format, in which each new object is added to the right

What Is The Possible Representation in Code?

```java
public class A {
    private B myB = new B();
    public void doOne() {
        myB.doTwo();
        myB.doThree();
    }
}
```
Communication Diagram

• To illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram

```
    doOne → : A
    1: doTwo ↓
    2: doThree ↓
       myB: B
```

Sequence vs. Communication

• Sequence diagram
  – Tool support is better and more notation options are available
  – Easier to see the call flow sequence

• Communication diagram
  – More space-efficient
  – Modifying wall sketches is easier
How Should We Do OO Design?

• Responsibility-driven design (RDD)
  – Think about how to assign responsibilities to collaborating objects
  – Think about following questions
    • What are the responsibilities of an object?
    • Who does it collaborate with?
    • What design patterns should be applied?

Responsibilities

• Obligations or behaviors of an object in terms of its role
• Two types of responsibilities:
  – Doing responsibilities
  – Knowing responsibilities
Doing Responsibilities

• Doing something itself, such as creating an object or doing a calculation
  – “a Sale object is responsible for creating its SalesLineItem objects”
• Initiating action in other objects
• Controlling and coordinating activities in other objects

Self-behaviors and collaborations or interactions with others

Guideline

• The transition of responsibilities into classes and methods is influenced by the granularity of the responsibility
  – Big responsibilities take hundreds of classes and methods
    • “provide access to relational databases” may involve two hundred classes and thousands of methods
  – Little responsibilities take one method
    • “create a Sale” may involve only one method in one class
Knowing Responsibilities

• Knowing about private encapsulated data
• Knowing about related objects
• Knowing about things it can derive or calculate
  – “a Sale object is responsible for knowing its total”

Self-data and relevant objects/data

Guideline

• The attributes and associations illustrated by domain objects in a domain model often inspire the responsibilities
  – If the domain model Sale class has a time attribute, it’s natural that a software Sale class knows its time.
  – Design classes do not always have identical attributes as domain classes
**GRASP: A Methodical Approach to OOD**

- Principles (Patterns) to guide choices about assigning responsibilities
  - Creator
  - Information expert
  - Low coupling
  - Controller
  - High cohesion
- Applicable to design and implementation

---

**Principle 1: Creator (doing)**

- Problem: Who creates an A?
- Advice: Assign class B the **responsibility to create an instance** of class A if:
  - B “contains” or compositely aggregates A
    - Whole-part; Assembly-part (e.g., body-leg)
  - B records A
  - B closely uses A
  - B has the initializing data for A
Example

• Who should be responsible for creating a SalesLineItem?
• Sale aggregates SalesLineItem objects

:Register

makeLineItem(quantity)

:Sale

create(quantity)

:Sales
LineItem

Summary

• Usually, the container or recorder of objects are creators
• Contraindications: complex creation
  – E.g. using recycled objects for performance
    • Both trucks and buses aggregate tires, so apply a Factory pattern to get instead of creating tires
Principle 2: Information Expert (knowing)

- Problem: Who knows the information to fulfill a responsibility?
- Advice: Assign the responsibility to class A if the information:
  - is about A’s attributes
  - is derivable by A, sometimes may depend on some attributes of relevant classes

Example

- Who knows the information about a Sale’s total amount of money?

<table>
<thead>
<tr>
<th>Sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
</tr>
<tr>
<td>time</td>
</tr>
<tr>
<td>getTotal()</td>
</tr>
</tbody>
</table>
Example

• Who knows the information about a Sale line item's subtotal?

<table>
<thead>
<tr>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>LineItem</td>
</tr>
<tr>
<td>quantity</td>
</tr>
<tr>
<td>getSubtotal()</td>
</tr>
</tbody>
</table>

Example

• Who knows the information of an item's price?

<table>
<thead>
<tr>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
</tr>
<tr>
<td>description</td>
</tr>
<tr>
<td>price</td>
</tr>
<tr>
<td>itemID</td>
</tr>
<tr>
<td>getPrice()</td>
</tr>
</tbody>
</table>
Summary

• Objects fulfill tasks using their info or the info of objects they know of
• It is crucially important to separate concerns between collaborative objects
  – E.g., getTotal() & getSubTotal()
  – Related to low coupling and high cohesion (discuss later)

Principle 3: Low Coupling (relations)

• Problem: How to reduce the impact of change?
• Advice: put data and operations together
  – Goal: Avoid unnecessary coupling
Examples of Coupling

- Class A has an attribute (field) of class B
- An instance of A calls an instance of B
- A has a method that references B instances
  - local variable/parameter/return value is a reference (i.e., pointer) to a B object
- A is a direct or indirect subclass of B

Example: Two Alternatives
The second is better

• Sale needs to know payment. The coupling is always there.
• Register simply delegates Sale to create the payment, without creating the payment itself

Principle 4: Controller (doing)

• Problem: What first object beyond the UI layer receives and coordinates ("controls") a system operation?
• Advice: Assigns "control" to class A if it is:
  – Facade controller: a class representing the entire system or device
  – Use case controller: a class representing a use case within which the event occurs
    • E.g., XyzHandler, XyzCoordinator, XyzSession
      – Xyz=name of the use case
**Example**

- System events in POS system
  - `endSale()`, `enterItem()`, `makeNewSale()`, `makePayment()`, ...
  - Typically generated by the GUI

```plaintext
:SaleWindow

enterItem(itemID, qty)

:???
```

**Using Facade Controller**

- Facade controller: entire system/device
  - `POS_System`, `Register`
- Used when there are NOT too many system events
  - Avoid “bloated” controllers (e.g., too many responsibilities)
Using Use-case Controllers

- Use-case controller: handler for all system events in a use case
- Used when there are MANY system events
  - Several manageable controller classes
  - Track the state of the current use-case scenario

Principle 5: High Cohesion (relations)

- Problem: How to keep object focused, and manageable?
- Advice: DON’T put too much data and operations into the same class
  - Goal: avoid unnecessary responsibilities
Example

- **Who creates Payment objects?**

- If Register does the work for all system events, it will become bloated and not cohesive

A better solution: delegation

- Our better solution: delegate Payment creation to Sale
  - Higher cohesion for Register
  - Also reduces coupling
Rule of thumb

- Class with high cohesion has relatively small number of methods with highly related functionality, and does not do too much work (LAR, p 317)

Benefits

- Clear separation of concerns
  - Easy to comprehend, reuse, and maintain
- Often results in low coupling
- Contraindications:
  - Distributed server objects need to be larger, w/ coarse-grained operations
    - Reduces the number of remote calls
  - To simplify maintenance by an expert developer