Object Design

Lian Loke
University of Technology, Sydney

Contributors
John Leaney, John Reekie, Lizveth Robles,
University of Technology, Sydney

Terms of Use: Creative Commons Attribution-ShareAlike 2.5
http://creativecommons.org/licenses/by-sa/2.5/

Where we've been

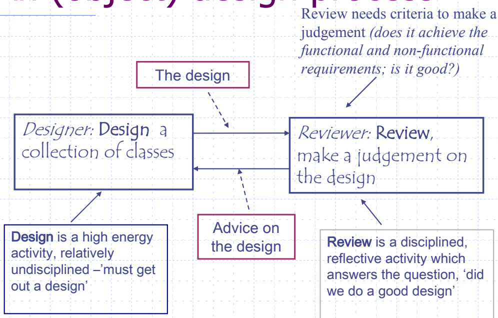
- An understanding of object-oriented design and the overall process
- Expressing the requirements of a software system
- Class diagram to capture object structures and relationships
- Sequence diagrams to capture interaction between objects

Object design is…

- Still part of object-oriented design, but focusing on the more detailed structure of classes (and hence, objects)
  - Classical concepts:
    - Cohesion
    - Coupling
  - Operation specification and implementation
    - Responsibility-driven design
    - Design by contract
    - Defensive programming
  - Inheritance design
    - Liskov substitution principle
    - Fragile baseclass problem

To be honest, I don't see "object design" as a separate topic at all. So don't get hung up about the terminology.

An (object) design process



Object Design

- How to design 'good' objects, and 'good' relationships between them?
  - *Aside*: Why is OO programming and design considered a good approach?
- Need **principles and criteria** for good design
- Clarity (a principle) obtained by (criteria):
  - Encapsulation
  - High Cohesion
  - Low Coupling
  - Keep messages and operations simple

Clarity

- Easy to understand
- Remember *someone else* in the future will be using, reading or modifying your design
- The next few slides look at more ways in which one can make clear designs
- Don't forget simplicity
  - Don't use a complex design when a simple one will do
  - Design for what you need now, don't design in complexity for future needs (which might change anyway)

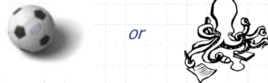
Encapsulation

- OO programming and design is firstly about encapsulation *(the ball)*
- Objects *hide* their internal structure,
  - presenting only their public interface to other objects
- To enforce encapsulation,
  - attributes are normally designated to have *private* visibility.

Not a nice interface

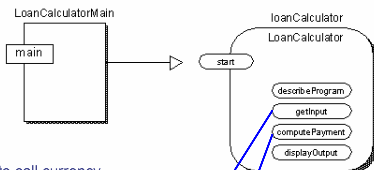
A nice interface

Cohesion



- Are the responsibilities of this module closely-related?
  - That is, do they all contribute to a single well-defined task?
  - Looser forms of cohesion exist:
    - Temporal: responsibilities are exercised close in time
    - Communicational: responsibilities all work on the same data
- In object-oriented design, we can also ask:
  - Do the data members of this class represent the complete and consistent state of a single thing?
  - Are all the operations of this class operating only on those data members?
  - Do the operations of this class make up a complete and useful set? The "round" object.

Loan calculator + currency converter ≠ international loan calculator



Need to call currency converter (blue lines), yet no ability without 'opening up', rewriting LoanCalculator

Ref: Wu, chapter 3

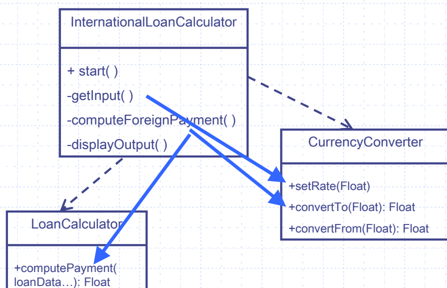
Coupling

- To what extent do the operations of this object depend on other objects?
  - Depending on fewer other objects is better than depending on more
  - Looser coupling means that the objects are more maintainable, as changes in one are less likely to require changes in others
  - Loose coupling is more likely to make code and designs more understandable. Although, taken too far, it can introduce unnecessary complexity *elsewhere* in the design…
- The goal of loose coupling often requires compromising on other goals of OO design.
  - For example, if you create a new object in order to "reuse" code that was previously duplicated in two other objects, you have introduced a dependency that wasn't there before. Sometimes, copying code actually **is** better!

Coupling is found…

- When one object interacts with or uses another object
  - A declares an object reference to B as attribute data
  - A sends a message to B
  - C is returned from A sending a message to B
  - A passes an object reference to C as input to a method in B
  - B declares an object reference to C as data local to a method
- With inheritance – superclasses and subclasses are tightly coupled

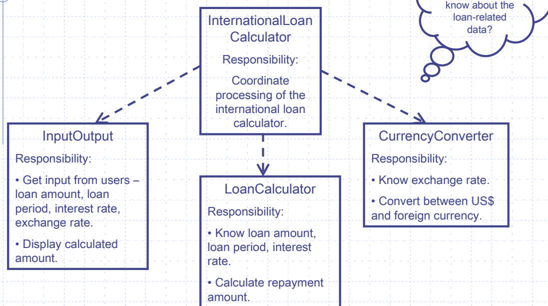
International Loan Calculator…



Responsibility-Driven Design

- Responsibility driven design is
  - another principle, like clarity, for obtaining good designs.
- Object design is done from the point of view of
  - objects have responsibilities
  - objects collaborate
- Ask questions:
  - What are the responsibilities of this object?
  - With which objects does it (need to) collaborate?

Back to the International Loan Calculator…



Object interfaces

- In object-oriented programming, *every* public method is part of an object's interface
  - In a collection of classes, every public method of every public object is…
  - Object-oriented APIs thus tend to be very "wide."
- In a typed programming language, the compiler checks (at compile time) that caller and callee are syntactically consistent
  - But what about at run-time? How do we know whether we are passing correct data to a method, and getting correct results back?
  - The answer to the above question revolves around the specification of the interface…

Design by contract

- Operations are defined by "contracts".
- A contract specifies *what* the operation will do, *not how* it will do it.
- Contracts go well with responsibility-driven design.
- More later…

Example of a contract

- A simple contract for a BankAccount class

  withdraw(amount: Real): Boolean

  - "Assume amount >= balance. The amount is deducted from the current balance of the account. True is returned. New balance >= 0."

  alternatively,

  - "If amount > balance, false is returned *and* no change to current balance. If amount <= balance, true is returned *and* amount is deducted from the current balance."

Reusability

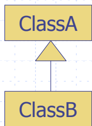
- We can promote reusability by designing reusable classes and reusing existing designs or classes.
- You can design for reuse by
  - Use of inheritance
  - Applying the Open/Closed Principle
    - Open for Extension, Closed for Modification
  - Designing classes with high cohesion
- You can reuse existing designs by
  - Reusing existing classes directly or by sub-classing them.
  - Using design patterns

Fragile base-class problem

- Object-oriented code is open to extension by sub-classing
  - You can extend a class even if you didn't write it and you don't have the source code
  - This is accepted practice, and one of the ways in which inheritance contributes to reusability and extensibility
- However, a change to the superclass made later can break the subclasses you wrote. The programmer of the superclass (base class) is restricted from making changes because he or she can't know in what ways you inherited from it.
- After a few years, these problems accumulate and a software system becomes *brittle*.

Liskov substitution principle

- An instance of a subclass can be used wherever an instance of the superclass can be used
- For example, a company has an employee induction process. While the induction documents refer to "employees," it applies just as well whether the employee is actually an engineer, a manager, or an administrative assistant.




```
ClassA thing1 = new ClassA();
thing1.doThis();
thing1.doThat();

ClassA thing2 = new ClassB();
thing2.doThis();
thing2.doThat();
```

In practice…

- In practice, you often do need to check the type of an object:

```
if (thing1 instanceof ClassB) {
    ((ClassB) thing1).doOneThing();
} else {
    thing1.doAnotherThing();
}
```

- It's not ideal to do this, but if it is in fact the simplest way to achieve your goal, then my advice would be to just do it. (The alternative may be more complex or more obscure.)