MSO
Object-oriented design

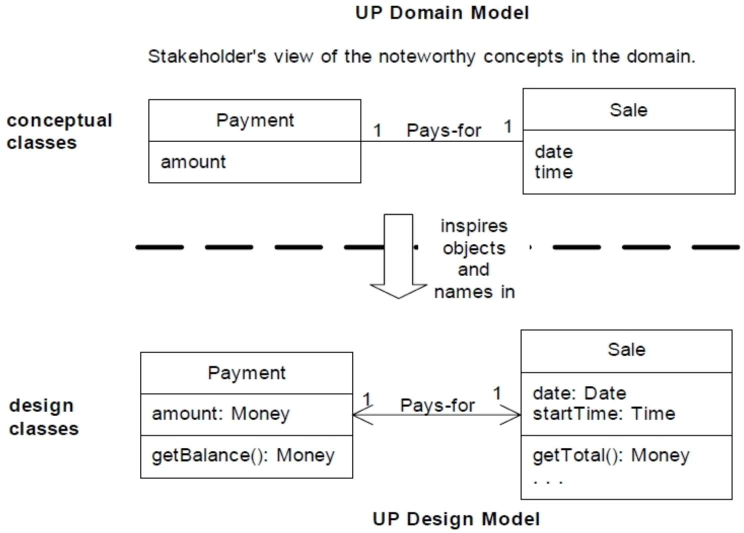
Hans Philippi
(based on the course slides of Wouter Swierstra)

August 30, 2018

Recap & topics

- Last lecture: we have met with UML diagrams
- Today: we will meet some design principles
- To be more concrete: we will discuss guidelines to extract class diagrams from domain models

This lecture



This lecture

From analysis to design:

- How can I use a domain model to write class diagrams?
- Can I make quality requirements more explicit?
- What *principles* guide this process?

Domain models vs classes

- A domain model should not attempt to organize a system in classes
- So how do you design a software system?
- There is a lot of freedom when designing a new class:
  - What methods should this class define?
  - Which methods should be public? Which methods should be private?
  - How much code should go into a method?
  - Should these methods all be defined by the same class?
- What constitutes good design?
- How can I decompose a big problem into manageable pieces?

Case study

Let us start by explaining *functional* design...

Case study

- Suppose you need to access a list of shapes stored in a database and then display them
- How would you go about planning to implement this?

Plan of attack

1. Locate the list of shapes in the database
2. Open up the list of shapes
3. Sort this list of shapes in some manner
4. Display every shape in the list individually

Functional decomposition

- A functional decomposition breaks a problem into small subproblems (repeatedly)
- This is fine for some problems, but . . .

Requirements change

$\boxed{\text{Requirements always change}}$

Requirements change

- We need a new database schema
- Can you extend the code to handle a new shape?
- We need to generate different visualization methods
- Can you sort the shapes in a different way?
- . . .

A critique of functional design

- A functional decomposition is what you would typically do to split code into methods
- On a bigger scale, it might mean that a single change to the requirements may have consequences for a lot of different pieces of code

For example, if a new kind of shape is introduced, I may need to adapt all my code

Requirements change

$\boxed{\text{Requirements always change}}$

Changing requirements

Requirements change for all kinds of reasons:

- You may have missed something when talking to a client
- You made a mistake in the requirements document
- You learn something during the initial implementation, uncovering technical problems or hidden assumption
- Client changes his mind
- Politics change
- The market changes
- . . .

The functional solution

You might say that the functional design solution sketched above is *weakely cohesive* and *tightly coupled*

What do these words mean?

Cohesion

Cohesion refers to how closely the operations in a unit of code (typically a method or object) are related

Cohesion: example

- *Low cohesion* – The convenience store. You go there for everything from gas to milk to ATM banking. Products and services have little in common, and the convenience of having them all in one place may not be enough to offset the resulting increase in cost and decrease in quality.
- *High cohesion* – The cheese store. They sell cheese. Nothing else. Can't beat them when it comes to cheese though.

(*Source*: Shog9's response to a question on stackoverflow)

Strong or Weak Cohesion?

```
public void HandleStuff
  (Data d, int quantity,
  int screenX, int screenY,
  int status, Color c, ...) {
    UpdateCorporateDatabase (d,q,status);
    for (int i = 0; i < quantity; i++)
      profit[i] = revenue[i] -
        expense[i] * status;
    newColor = c;
    status = SUCCESS;
    DisplayProfits(screenX,screenY,status,d,c);
    ...
```

Cohesion

- A function like `sin(x)` or `tan(x)` has very strong cohesion: it does exactly one thing
- You should try to have every method do *one* thing and one thing *only*

Why is strong cohesion important?

- Code is easier to maintain – if the implementation of `sin` changes, no other method needs to be updated
- Code is easier to understand – strong cohesion can help to break a large code base into easily digestable parts
- Code is easier to reuse – code that performs one clearly defined task is more generally applicable than one giant method

Cohesion

- People distinguish many different forms of cohesion
- Steve McConnell's *Code Complete* (Microsoft Press, 1993) has a good chapter discussing different forms of cohesion – wikipedia's entry is more terse
- If you cannot come up with a clear name for a method, it's usually a sign of weak cohesion:
  - `DoIt`, `HandleStuff`, `Step7`
  - `GetIconLocation`, `EraseFile`, `CalculateInterestRate`
- *Exercise:* Look at some old code you wrote, for example the labs from IMP. How cohesive are your methods?

Coupling

- *Cohesion* refers to how closely the operations *within* a program module (method, object, or whatever) are related
- *Coupling* refers to how closely different program modules (or more specifically, objects or methods) are related

Coupling: examples

- *Loose*: You and the guy at the convenience store. You communicate through a well-defined protocol to achieve your respective goals - you pay money, he lets you walk out with the bag of Cheetos. Either one of you can be replaced without disrupting the system.
- *Tight coupling*: Me and my partner
  (*Source*: Shog9's response to a question on stackoverflow)

Types of coupling

- *Data coupling* – one method relies on the data produced by another method
  - Example: calling the `tan` method
- *Control coupling* – one method or object tells the other what to do
  - Example: setting control flags to print in colour or not
- *Global data coupling* – methods sharing the same global variable.
  - Example: storing configuration data in a global variable
- *Pathological coupling* – one method relies on an exact implementation of another method.
  - Example: relying that `sin(0)` returns 0 within 0.001ms
  - Example: relying that the third field of every table in the database is a key

Why is loose coupling important ?

- If two modules are tightly coupled, changing one is more likely to need additional changes
  - Example: changing the database schema needs a complete code overhaul
- If modules are tightly coupled, they are harder to assemble or tear apart
  - Example: two modules sharing the same global variable
- If two modules are tightly coupled, they are harder to reuse or test
  - Example: it takes a lot of set-up time to test a series of modules sharing global variables, compared to a function like `sort`

Coupling: functional programming

Haskell enthusiasts always claim that Haskell programs are:

- easy to test
- easy to reason about
- easy to reuse

Pure functions are always (fairly) loosely coupled

Coupling: subtle business. . .

- A floating point number is passed to the `tan` method
- A name, address, date of birth, and customer ID is passed to a method that uses all these parameters
- A `Person` object is passed to a method, that uses all this information
- A `Person` object is passed to a method that sends a birthday card, if it is the customer's birthday
- A `X` object with 27 fields is passed to a method, that uses 19 of them
- A method edits the customer information in a database. It calls another method, passing the customer ID. This second method consults the database to see the changed values.

Coupling

- *Exercise:* Look at some old code you wrote, for example the labs from IMP. How are your methods and objects coupled?

Back to functional design

If a design is tightly coupled or weakly cohesive, this is regarded to be a *Very Bad Thing:* a minor change can effect a lot of other parts of the system!

Functional design (repeatedly) decomposes a problem into smaller subproblems – and solves these subproblems in separate methods

But if the input data changes, all these functions could be affected!

Responsibilities

- That's it for this moment
- Next lecture: more about *assigning responsibilities*