MSO
Principles of Object Orientation

Wouter Swierstra, Hans Philippi

October 10, 2018

This lecture

- How to apply design patterns?
- What are some of the more general principles of object-oriented design?
- Following lecture: how can a CVA help complement the existing analysis techniques?

Applying design patterns

- Should you work top-down?
- Or bottom-up?
- Or design a functionally correct solution, and then refactor by applying patterns?
- Or some other way entirely?

Christopher Alexander

… makes the following critical observation:

> *Design is often thought of as a process of synthesis, a process of putting things together, a process of combination. According to this view, a whole is created by putting together parts. The parts come first: and the form of the whole comes second.*

On the contrary, Alexander proposes an analogy with architecture:

> *Design with rooms in mind, not with several kinds of bricks*

Towards better design

1. Start out with a conceptual understanding of the whole: what is the big picture?
2. Identify patterns at this level
3. Start filling in these patterns, creating *context*
4. Work inward: repeatedly apply patterns, identify new patterns, and repeat
5. The final implementation is then guided by the sequence of design patterns you chose to apply, one at a time

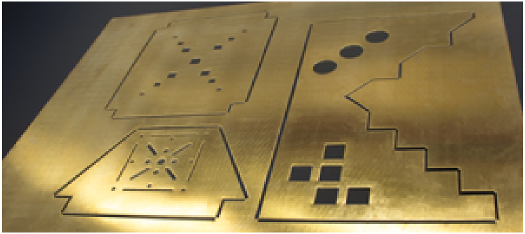
How is this different?

- Don't start think at the level of classes (bricks) – think more generally about problems and patterns that address them (rooms)!
- Start identifying high-level patterns that create *context* – this solves one problem, but may introduce new (smaller) subproblems
- Refine a design iteratively, adding more precision in every iteration

Case study: CAD/CAM software

We will look at a case study revolved around Computer-aided design (CAD) and Computer-aided manufacturing (CAM)

Problem description



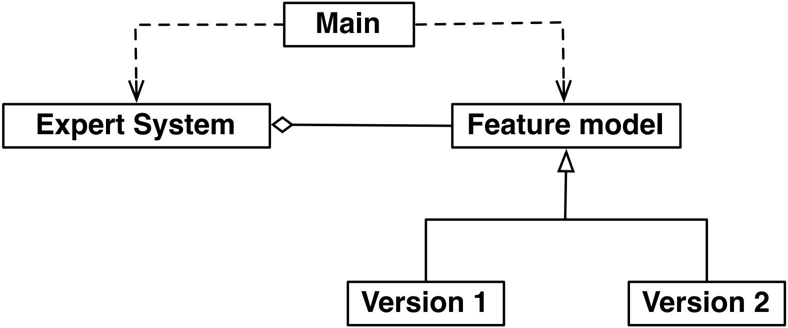
We want to design software that:

1. Analyzes a piece of sheet metal
2. Sees how it should be made, based on the features it contains
3. Generates a set of instructions for manufacturing equipment
4. Passes these instructions to the manufacturing equipment whenever you want to make such a part

The design challenge

- **Good news:** There is already an (expensive) expert system in place that determines how to make complex features . . .
- **Bad news:** . . . but the CAD/CAM software used to describe the features keeps changing (V1 and V2 running, may be V3 coming)
- **Challenge:** How can we handle this change?

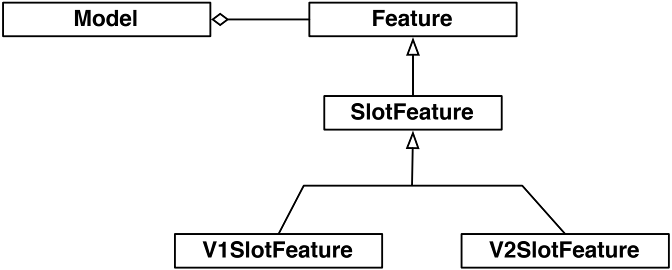
Global architecture



Architecture

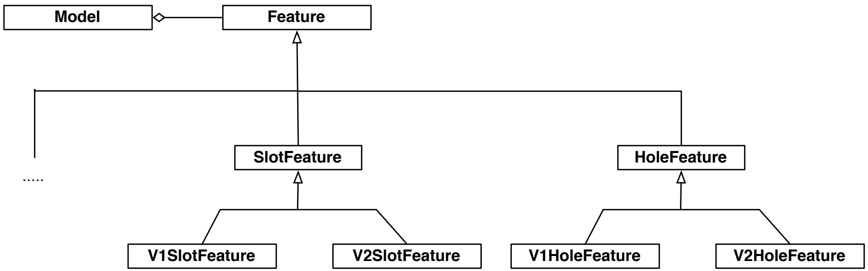
But how can we actually implement this idea?

Example: slot features



And similar subclasses for all other features

Adding new features



Criticism of this solution

- Redundancy among methods – `V1getX` for the Slot class and Hole class will probably be very similar
- Tight coupling – all the features of every version are related to one another through subtyping
- Weak cohesion – functionality scattered over different classes
- Room for error – there is no guarantee that you cannot mix V1 and V2 features in a single model
- But most importantly – what happens when a new version of the CAD/CAM software is released?

Criticism of this solution

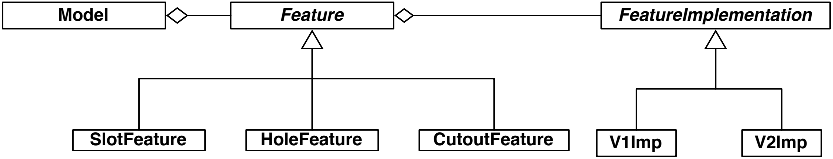
Let us revisit our solution, applying what we now know about design patterns

Situation

- We have different kinds of features: slots, cutouts, holes, etc
- These features are implemented differently in the different versions of the CAD/CAM software

Perhaps a Bridge might help?

Applying the Bridge



Refining the design

- What methods should the abstract Feature-Implementation class support?

A commonality-variability analysis can help:

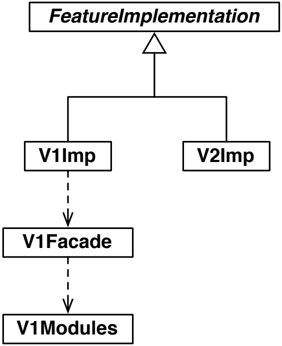
- getX method
- getY method
- getLength method
- . . .

Refining the design

- Applying the Bridge pattern determines an initial step of the design . . .
- But how do I hook the existing CAD/CAM systems to the V1Imp and V2Imp classes?
- The first version of the CAD/CAM software had a quite complicated interface, that did not fit well with this design
- So . . . ?

Version 1

- The first version of the CAD/CAM software had a quite complicated interface, that did not fit well with this design
- Applying a Facade pattern hides this complexity

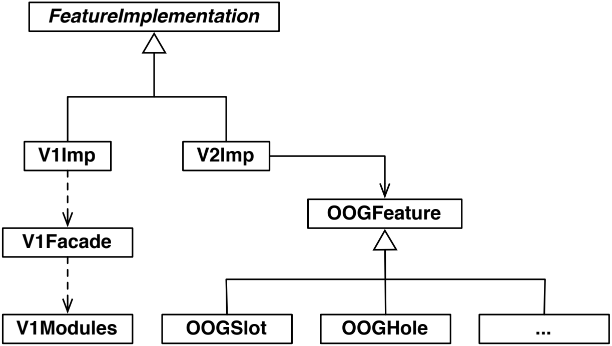


Version 2

- The second version of the CAD/CAM software was more object oriented, but did not share the same interface as the V1Imp
- So . . . ?

Version 2

- The second version of the CAD/CAM software was more object oriented, but did not share the same interface as the V1Imp
- Adding an Adapter fixes this



Reflection

- What are we doing?
- We start by identifying the patterns (Bridge) that solved some of the high level problems, and then apply more and more patterns, until we have a stable design
- By selecting high-level patterns that create *context*, we can iteratively refine our design
- *We are learning to think in patterns, rather than classes*
- *We are learning to think in rooms, rather than bricks*

Design patterns explained

- Shalloway and Trott argue the same point: design in terms of patterns
- Let us discuss their derivation:
    - Consider all patterns we've seen so far; which patterns are applicable?
    - Apply a pattern and repeat

Question: What do you think of this methodology?

Some considerations

- It seems that the solution depends heavily on the list of patterns that you have available
- What if you need to make important decisions for which there is no Gang-of-Four pattern that can help you in the design?
- For example, very few Gang-of-Four patterns describe the software *architecture*

Principles and Strategies of Design Patterns

Chapter 14 of Shalloway and Trott describes a lot of the theory underlying design patterns (and object oriented design in general):

- the open-closed principle
- the Liskov substitution principle
- encapsulating variation
- dependency inversion
- and much more

It is a good overview of the theory behind object oriented design . . .

. . . and sets up some of the design patterns we will see in the rest of the course

The Open-Closed Principle

Bertrand Meyer:

> *Software should be open for extension, but closed for modification*

The Open-Closed Principle

This is reflected in a lot of design patterns:

- the Strategy pattern can be extended with new strategies, but you cannot modify existing strategies without rewriting code
- . . .

The Open-Closed Principle

How to interpret this rule?

- It does *not* say that you should put an effort in making modification hard to realise
- It says that your design should be done in such a way that future adaptations of the software should (almost) always lead to extensions, not to modifications

The Open-Closed Principle

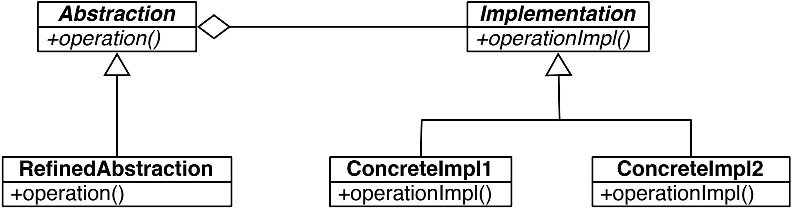
- It is not a golden rule . . .
- . . . but it is regarded to be a good guideline
- It may be hard to achieve in practice, but worthwile to strive for

Dependency inversion - I

When applying the Bridge pattern, where do you start?

- Study the different implementations, encapsulate variation between them by introducing an abstract superclass?
- Study the abstractions and define the interface they need?

Pictorally, do you start on the 'left' or the 'right' half of the bridge?



Dependency inversion principle - II

The dependency inversion principle suggests to *always* start on the left, the side of the abstractions

- High-level modules should not depend on low-level modules
- Abstractions should not depend on details
- Implementations are more likely to change than the concepts involved

Dependency inversion

- Big supermarkets probably have to handle a lot of different invoices from suppliers
- Who should fix the format of the invoices?

Dependency inversion

- Big supermarkets probably have to handle a lot of different invoices from suppliers
- Who should fix the format of the invoices?
- The supermarket should provide a single format for invoices: that makes their life so much easier
- In fact, *Dependency inversion* is variant of the the well known principle *Program to an interface, not to an implementation*

Liskov substitution principle

Barbara Liskov (Turing Award 2008):

> *A class deriving from a base class should support all the behaviour of the base class*

(The exact formulation states that any property that is provable of objects of some supertype S, should also be provable for objects of type T when T is a subtype of S)

The Liskov principle

- This seems a very basic property of inheritance, at first sight
- But note that the principle also applies to the contractual aspects: preconditions, postconditions, invariants

Breaking the Liskov principle?

- Suppose I have two classes, Square and Rectangle
- Intuitively: `Square : Rectangle`
- Now suppose a Rectangle has getters and setters for both `width` and `height`
- What should a Square do?
- Answer: every adaptation of `width` or `height` is followed by an adaptation of `height` or `width` . . .
- . . . enforcing invariant: `height == width` for Squares

Breaking the Liskov principle?

- Answer: every adaptation of `width` or `height` is followed by an identical adaptation of `height` or `width` …
- …enforcing invariant: `height == width` for Squares
- But now we write a unit test for Rectangles

-
  ```
  r.width = 4; r.height = 5;
  AssertEqual(r.Area(), 20);
  ```

Breaking the Liskov principle!

- …enforcing invariant: `height == width` for Squares
- But now we write a unit test for Rectangles:

-
  ```
  r.width = 4; r.height = 5;
  AssertEqual(r.Area(), 20);
  ```
- From an intuitive point of view, a Square is a special case of a Rectangle
- But we have to look at it from a behavioral point of view (methods, conditions, invariants, …)
- From an OO perspective, a Square is not a Rectangle!

There is no such thing as right or wrong

- Designing software is not a yes/no or right/wrong question
- You need to understand the context of the problem:
- What can vary?
- Which requirements are likely to change?
- Which solution is easier to maintain?
- Or has stronger cohesion? Weaker coupling?

Making design decisions

When comparing designs, ask yourself:

- When is this design better than that design?
- What kind of change is easy to accommodate in X, but harder to do in Y?

Characteristics of good design

The authors of Design Patterns explained have a recognizable style of writing designs:

- inheritance rarely goes more than two levels deep
- if it does, it is because there is a rich design pattern involved
- never capture more than one property in subtyping

Why?

SOLID Object oriented design

- **S**ingle responsibility – every class should be responsible for one thing only (strong cohesion)
- **O**pen/closed – open for extension, closed for modification
- **L**iskov substitution – a subclass should support all the behaviour of the superclass
- **I**nterface segregation – program to a (small, cohesive) interface, not an implementation
- **D**ependency inversion – depend on abstractions, not implementation

Epilog: healthy skepticisism

Applying patterns poorly can lead to problems:

- **Superficial understanding** – if you don't understand the problem completely, you may choose the wrong pattern for the job
- **Bias** – if you are keen on applying a particular pattern, you may never question your assumptions
- **Selection** – if you don't understand the context and conditions that define when a pattern is appropriate, you may apply it incorrectly
- **Fit** – ignore exceptions in the concrete instances, as they don't seem to fit the theory of the pattern

Always question your assumptions