MSO
Design Patterns: upfront design?

Wouter Swierstra, Hans Philippi

September 14, 2018

This lecture

- Remember the cliffhanger ...
- Do Agile Development and Design Patterns conflict?

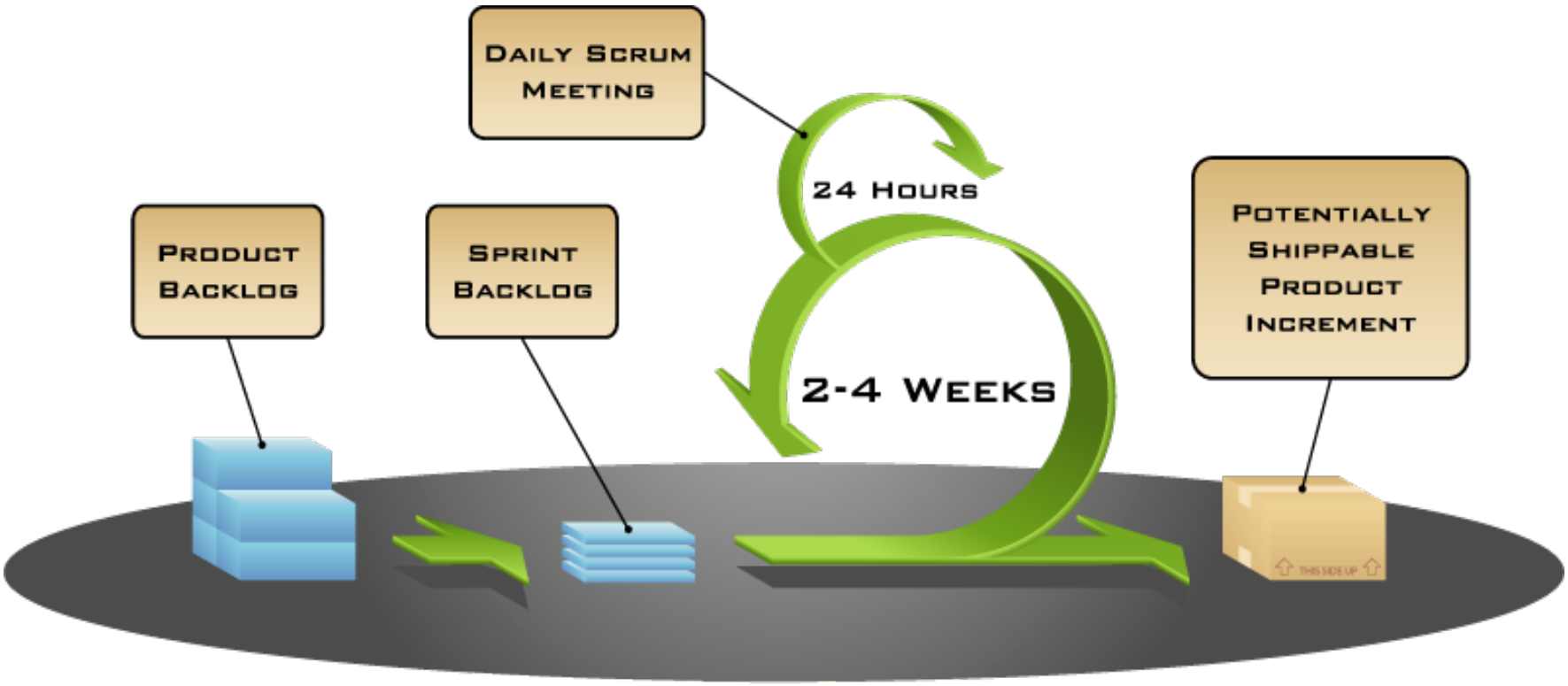
Recall: UP vs Agile

The Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Agile software development with Scrum



Design in an Agile setting

- Agile programming emphasizes not to invest too much time in Big Design Up Front

Design in an Agile setting

- Agile programming emphasizes not to invest too much time in Big Design Up Front
- Does that mean that design patterns are not relevant?

Design in an Agile setting

- Agile programming emphasizes not to invest too much time in Big Design Up Front
- Does that mean that design patterns are not relevant?
- Shalloway and Trott argue that by learning design patterns you will learn the qualities of good code and good design

Agile vs Design Patterns

Agile practices focus on:

- No redundancy
- Readability
- Testability

But Design Patterns value the same principles!

No redundancy

- Never, never duplicate code; copy-paste is a *sin*
- Why? This leads to strange coupling behaviour and unmaintainable code
- Instead: encapsulate change-sensitive code behind a well defined interface
- Applying design patterns can encourage code reuse

Readability

Choose the names of objects and methods to reveal their intent

> *Whenever you feel the need to comment something, write a method instead (Martin Fowler)*

Aim for short, cohesive methods – in line with design pattern philosophy

Testability

"Code that is tightly coupled or weakly cohesive is harder to test"

Incremental delivery requires extensibility!

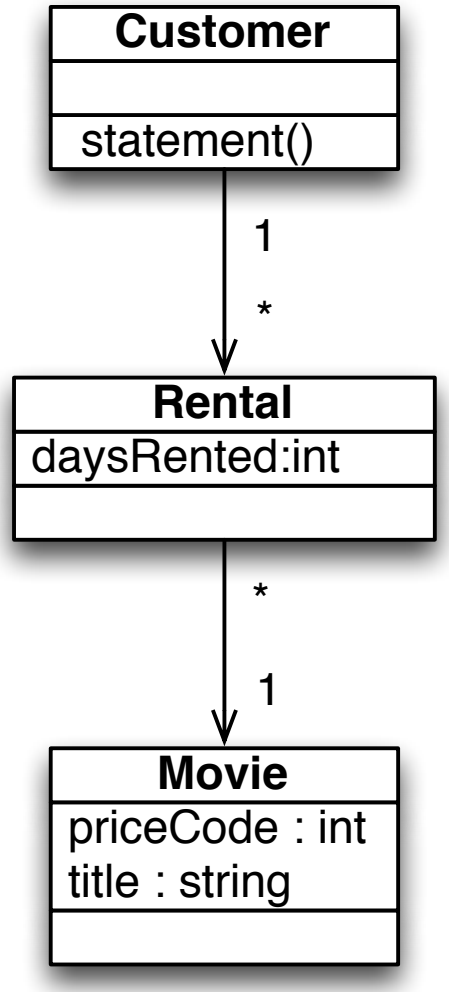
- Agile focuses on delivering new product increments in every iteration
- But some features are too big to implement in a single iteration
- Instead, you may want to lay the groundwork and implement a partial solution first and extend this later
- But how can you design a partial solution that is easy to extend?

In summary. . .

Although the design pattern philosophy may seem at odds with Agile/XP practices . . .

. . . they share the same ideas about what makes great code!

Motivating example: video rental store



Initial classes

- The Movie class defines three price codes as integer constants:
  CHILDREN $= 0$
  STANDARD $= 1$
  NEW $= 2$
- The Rental class tracks the number of days a particular movie has been rented
- Finally, the Customer class records a list Rentals

But ..., beware of the wicked switch!


```
switch (v.getMovie().getPriceCode()) {
  case Movie.STANDARD  :
    v_price = 3 euro  * v.daysRented; break
  case Movie.NEW  :
    v_price = 5 euro  * v.daysRented; break
  case Movie.CHILDREN  :
    v_price = 2 euro * v.daysRented
 }
```

Wicked switches!

- Switch commands are not bad in an intrinsic sense ...
- ... but they carry that smell
- Do not confront programmers with switches if making the choice with respect of specific details is not their *prime responsibility!*
- When programming the control flow of an order, you should not not be concerned with annoying details

But then...

- The video store owner calls

But then...

- The video store owner calls
- We will not only rent movies, ...

But then...

- The video store owner calls
- We will not only rent movies, ...
- ... but also provide streaming services, ...

But then...

- The video store owner calls
- We will not only rent movies, ...
- ... but also provide streaming services, ...
- ... which also involves a new quality level: UHD

Do they ... ?

Requirements
always
change

Why design?

- If requirements always change, what good are design patterns?
- Or any upfront design?
- Isn't quick-and-dirty programming more effective?

An analogy

- Every time I print a document, it is easier for me to put it on my desk, than file it away
- This works fine if I only ever have ten documents to manage
- But if I need to handle more documents, I need to invest in maintaining some kind of system
- *Lesson:* Cutting corners in the short term, may have a price to pay in the long term

Excuses, excuses. . .

- We don't know what requirements are going to change
- If we try to see how things will change, we'll stay in analysis . . .
- If we try to write our software that is extensible, we'll stay in design forever
- We don't have the money
- We don't have the time
- I'll get to it later . . .

Pick your poison

It would seem that there are only two choices:

- Overdesign
- Quick and dirty

A third way

- You don't need to fix the design upfront ...
- ... but you do need to think about where the changes are most likely to happen
- If you think about this beforehand, the investment to write adaptable code is much lower than having to refactor a large code base post hoc

Some principles of good design

- Program to an interface, not an implementation
- Favor aggregation over inheritance
- Encapsulate the concept that varies

Applying these can help write code that is robust to change

Soo much for philosophy