MSO
Requirements engineering

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

September 5, 2018

Development Process: requirements engineering

Requirements Engineering aims to describe *what* the system should do, allowing customers and developers to agree on that description

Requirements – why?

- Software errors cost the American economy about $60 billion each year
- 84% of all software projects are unsuccessful (late, over budget, cancelled, etc.)
- Three of the top reasons for failure:
    - Poor user input 13%
    - Incomplete requirements 12%
    - Changing requirements 7%

Requirements are really important!

Stakeholders and concerns

Requirements should identify relevant *stakeholders* and *concerns*

A *stakeholder* is a person, group, or entity with an interest in or concerns about the realization of the architecture

A *concern* is a requirement, an objective, an intention, or an aspiration a stakeholder has for that system

Requirements – definition

A *requirement* is a feature that a system must have or a constraint it must satisfy to be accepted by the client

Requirements engineering is the process of identifying and defining the requirements of a software system

Requirements engineering is a cyclic process:

1. Elicitation
2. Specification (e.g. use cases, scenarios)
3. Validation and verification
4. Negotiation

Requirements examples

Requirements come in all kinds of shapes and sizes:

- Functional – features, capabilities
- Usability – help, documentation
- Reliability – frequency of failure, uptime, recoverability
- Performance – response time, accuracy, throughput
- Supportability – adaptibility, maintainability

But also sub-factors such as:

- Implementation – languages and tools, hardware
- Legal – licensing
- Interface – constraints arising from other systems

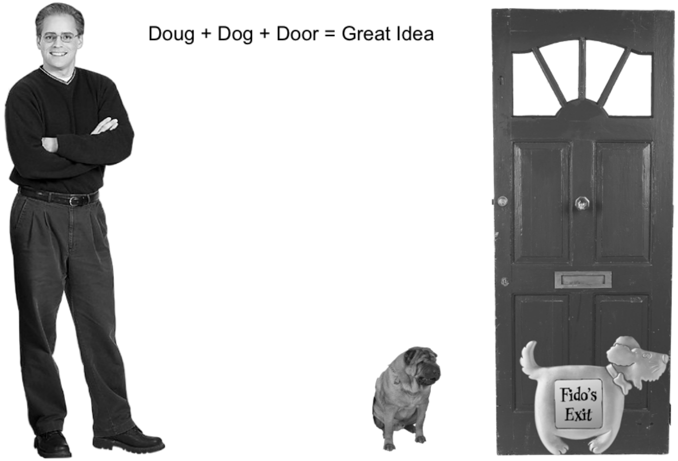
Requirements – how?

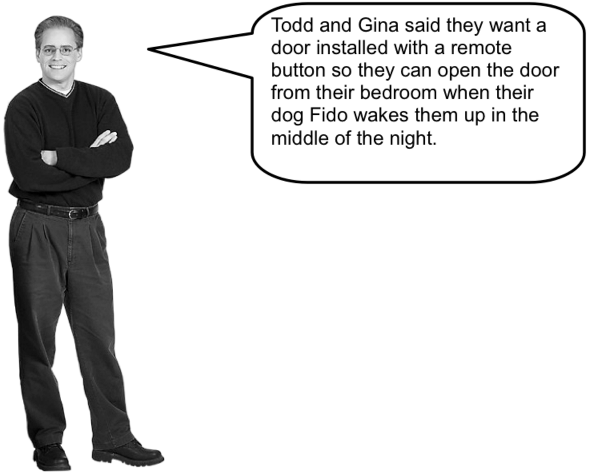
Many customers have little technical skills. Or they may not even know what they want themselves. How can you figure out what they want?

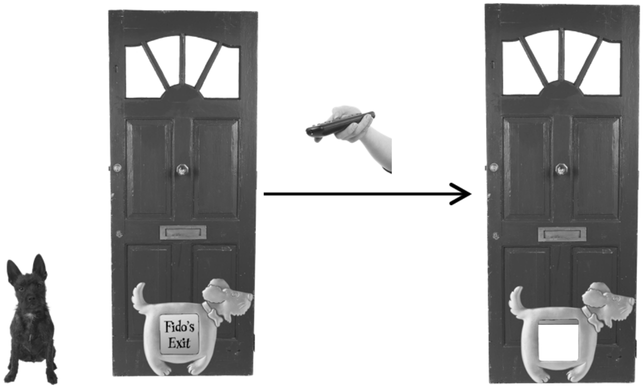
Eliciting clear requirements is not as easy as you may think. . .

(Example from O'Reilly's *Head First Object-Oriented Analysis and Design*)

Doug + Dog + Door = Great Idea




Todd and Gina said they want a door installed with a remote button so they can open the door from their bedroom when their dog Fido wakes them up in the middle of the night.



This is a piece of cake, right?
And you're getting paid for this!

Easy spec

- Push the button on the remote to open the door
- Push the button on the remote to close the door

Initial design

- `DogDoor` class
    - method `Open`
    - method `Close`
    - method `WaitForRemote`
- `Remote` class
    - tell the `DogDoor` to open and close
    - knows when the button is pressed.

```
public class DogDoor {

  private boolean open;

  public DogDoor() {
    this.open = false; }

  public void open() {
    Console.WriteLine("The dog door opens.");
    open = true; }

  public void close() {
    Console.WriteLine("The dog door closes.");
    open = false; }

  public boolean isOpen() {
    return open; }
}
```

```
public class Remote {

  private DogDoor door;

  public Remote(DogDoor d)
  { ... }

  public pressButton () {
    if door.isOpen()
    {
      door.close()
    }
    else
    {
      door.open()
    }
  }
}
```

What else?

- This defines the core class files
- (We will ignore the interface to the door's hardware)
- Perhaps we should write some tests?

```
public class DogDoorSimulator {

  public static void main(String[] args) {
    DogDoor door = new DogDoor();
    Remote remote = new Remote(door);

    Console.WriteLine("Fido barks to go out.");
    remote.pressButton();

    Console.WriteLine("\nFido has gone.");
    remote.pressButton();

    Console.WriteLine("\nFido’s all done.");
    remote.pressButton();

    Console.WriteLine("\nFido’s back inside.");
    remote.pressButton();
  }
```

So we're done!

- We have talked to our customer to establish some requirements
- We have implemented a system that satisfies these requirements
- We have tested that our system behaves as expected

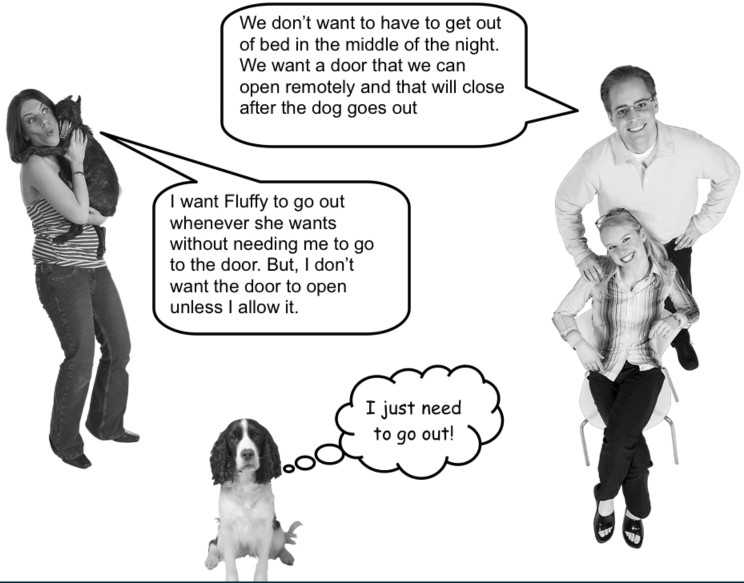
But . . .

What went wrong?

- We implemented what we were asked
- The hardware works
- The software works
- Our testsuite passes

Getting the requirements right is not just the customer's responsibility!

But . . .


We don't want to have to get out of bed in the middle of the night. We want a door that we can open remotely and that will close after the dog goes out
I want Fluffy to go out whenever she wants without needing me to go to the door. But, I don't want the door to open unless I allow it.
I just need to go out!

But what if…

Question: What else could possibly go wrong?

But what if. . .

- Fido does not bark to go outside?
- Todd and Gina don’t hear Fido barking?
- Fido is barking, but does not want to go outside?
- the door is closed before Fido returns?
- the door jams?
- the door closes as Fido is going outside?
- . . .

Why this example is important

- The implementation of the `DogDoor` is super simple – it is essentially a single boolean
- But figuring out what the customers want is really hard!
- How can we figure out what the customer wants?

Finding requirements

- Elicitation
    - talk to clients and end-users
    - gather existing documentation
    - observe how future users work now
- But be careful:
    - ‘Customers can check bank balances’ – but only their own!
    - ‘A grade above 6.0 is a pass’ – but so is a 6.0!
    - ‘Every telephone number starts with an area code’ – unless it’s for a cell phone

Elicitation techniques - I

- *Asking:*
    - What do you expect from the system?
    - Interview, brainstorm, questionnaire, ...
- *Task analysis:*
    - Which subtasks can you identify?
    - How can you organize tasks into a hierarchy?

Elicitation techniques - II

- *Scenario-based analysis*
    - How do you do Y?
    - But what happens if . . . ?
- *Ethnography*
    - Actively observing users do their work
    - May be more reliable than asking them

Elicitation techniques - III

- *Analyzing existing documents and systems*
    - What problems do the existing systems have?
- *Prototyping* – is this what you had in mind?
- *Domain analysis* - how are other systems in this domain built?

What makes a good set of requirements?

Correctness:	it accurately captures the client's views
Consistency:	it does not contradict other requirements
Completeness:	all possible scenarios are accounted for
Clarity:	unambiguously formulated
Realism:	it can be implemented and delivered
Verifiability:	it can be (automatically) tested

Examples of poor requirements

- *The system should be usable by elderly people*

Examples of poor requirements

- *The system should be usable by elderly people*
    - Not easily verifiable and unclear

Examples of poor requirements

- *The system should be usable by elderly people*
  - Not easily verifiable and unclear
  - Better: All text should be in at least 14pt font

Examples of poor requirements

- *The system should be usable by elderly people*
  - Not easily verifiable and unclear
  - Better: All text should be in at least 14pt font
- *Upon completion, the final product shall not have any errors*

Examples of poor requirements

- *The system should be usable by elderly people*
    - Not easily verifiable and unclear
    - Better: All text should be in at least 14pt font
- *Upon completion, the final product shall not have any errors*
    - Not verifiable and not realistic

Examples of poor requirements

- *The system should be usable by elderly people*
  - Not easily verifiable and unclear
  - Better: All text should be in at least 14pt font
- *Upon completion, the final product shall not have any errors*
  - Not verifiable and not realistic
  - Better: give specific test criteria

Examples of poor requirements

- *The system should be usable by elderly people*
  - Not easily verifiable and unclear
  - Better: All text should be in at least 14pt font
- *Upon completion, the final product shall not have any errors*
  - Not verifiable and not realistic
  - Better: give specific test criteria
- *The product shall respond rapidly*

Examples of poor requirements

- *The system should be usable by elderly people*
  - Not easily verifiable and unclear
  - Better: All text should be in at least 14pt font
- *Upon completion, the final product shall not have any errors*
  - Not verifiable and not realistic
  - Better: give specific test criteria
- *The product shall respond rapidly*
  - Not verifiable and unclear

Examples of poor requirements

- *The system should be usable by elderly people*
  - Not easily verifiable and unclear
  - Better: All text should be in at least 14pt font
- *Upon completion, the final product shall not have any errors*
  - Not verifiable and not realistic
  - Better: give specific test criteria
- *The product shall respond rapidly*
  - Not verifiable and unclear
  - Better: The system shall respond in less than 2s

Examples of poor requirements

- *The system should be usable by elderly people*
    - Not easily verifiable and unclear
    - Better: All text should be in at least 14pt font
- *Upon completion, the final product shall not have any errors*
    - Not verifiable and not realistic
    - Better: give specific test criteria
- *The product shall respond rapidly*
    - Not verifiable and unclear
    - Better: The system shall respond in less than 2s
    - Best: The system shall respond in less than 2s in 99% of the cases

Not all requirements are created equally

Everyone wants a system that is up 24/7, is easy to take down for maintainence, cheap to implement, immediately responsive, and that looks pretty

Trying to identify what is *really* important can save you a *lot* of development time down the road

Requirements are not a wish list

In the Software Project that CS students do in to complete their BSc, many students make the same mistake:

Requirements are not a wish list

Requirements are a contract for the minimal functionality that you can promise to deliver

Try different techniques

The more information you have ...

- the better you understand the domain
- the better you can write and prioritize your scenarios
- the greater the chance of success

Scenarios and use cases

- One of the most popular way to establish (functional) requirements is through writing *scenarios* and *use cases*
- These document the system's behaviour from the users point of view, in a way that customers can understand what the system does – how does the system *add value* to the user?
- Use cases and scenarios are crucial if you want to agree on what the system does
- More on use cases in the next lecture