Requirements Engineering: Specification & Validation

Software Requirements and Design
CITS4401

Week4

Key ideas (this week)

Testing requirements

Convince me!

Requirements Specification Document

Tell me!

Prototyping requirements

Show me!

Why do requirements need tests?

- An essential property of a software requirement is that it should be possible to **show** (validate) that the finished product satisfies it.
- Requirements that cannot be validated are really just “wishes.”
- An important task is therefore planning how to verify each requirement.
- In most cases, acceptance tests are prescribed based on how end-users typically conduct business using the system.

Source: SWEBoK Guide 6.4 Acceptance Tests

Requirement Test

- A test for a requirement is a way to demonstrate whether a system satisfies the requirement.
- Best to think about and write the tests at the same time as writing the requirement.
- The test is a type of contract for that requirement
- See also “Fit Criterion” – how will we know if this requirement has been satisfied?

Testing Requirements

- Engineers aren't typically great at assessing the testability of requirements.
- Agile deals with this by bringing **testers** in right from the start
- (vs Waterfall when testers see everything much later in the process when it is often too late to change/fix the requirements as they are documented and agreed).

Verifiable Requirements Specs

- Definition. A requirements spec is **verifiable**

  if (if and only if) every requirement statement is verifiable

  If there is some finite cost-effective way in which a person or machine can check to see if the SW product meets the requirement

- We can use test cases, analysis or inspection to decide

Some Objective Metrics for Non-functional Requirements (1)

- Performance Speed
  - Number of processed transactions per second
  - User/event response time
  - Screen refresh time

Example 1: The system shall respond to user requests in < 1 second when the system is running at normal user load of <100 concurrent users.

Example: The traffic gate shall close in a most 3 seconds.

- Size
  - Kilobytes
  - Number of RAM chips

Example 2: The installed app requires less than 200 MB of memory on an Android phone.

Some Objective Metrics for Non-functional Requirements (2)

- Reliability
  - Mean time to failure
  - Probability of unavailability
  - Rate of failure occurrence

Example 3: The system shall be available to users for at least 1400 minutes in any 24 hour period.

- Robustness
  - Time to re-start after system failure
  - Percentage of events causing failure
  - Probability of data corruption on failure
- Integrity
  - Maximum permitted data loss after system failure

Example 4: No more than 1 minute of entered data can be lost if the system crashes.

Some Objective Metrics for Non-functional Requirements (3)

- Ease of Use
  - Training time taken to learn 75% of user facilities
  - Average number of errors made by users in a given time period
  - Number of help frames

Example 5: Users who have completed the system tutorial should be able to complete the PlanTrip use case within 2 minutes.

- Portability
  - Percentage of target-dependent statements
  - Number of target systems

Example 6: The app shall run on all Android phones models since 2015 for all operating system versions from 7 onwards.

Some Objective Metrics for Non-functional Requirements (1)

Example 1: The system shall respond to user requests in < 1 second when the system is running at normal user load of <100 concurrent users.

Test: Write a script to run 100 common user requests and launch this script 99 times (worst case). Measure the total response time for each request (or all) and test 100 requests can be served in <100 seconds.

Example 2: The installed app requires less than 200 MB of memory on an Android phone.

Test: Generate an executable file for the app on (different versions) of Android phone and check that the app size is <200 MB for all versions.

Some Objective Metrics for Non-functional Requirements (2)

Example 3: The system shall be available to users for at least 1400 minutes in any 24 hour period.

Test: Write a script to run continuous “normal” interactions and run it for a long term. Record any time the system is not available. Note these types of requirements are hard to test effectively.

Example 4: No more than 1 minute of entered data can be lost if the system crashes.

Test: Write a script that continuously enters data. Write another script that crashes the system at random times. For each crash, measure the amount of data that was lost ie not saved before the crash.

Hard-to-test requirements (1)

System requirements

Requirements which apply to the system as a whole.

In general, these are the most difficult requirements to validate irrespective of the method used as they may be influenced by any of the functional requirements.

Hard to test for non-functional system-wide characteristics such as usability.

Hard-to-test requirements (2)

Exclusive requirements

Requirements which exclude specific behaviour.

For example, a requirement may state that system failures must never corrupt the system database.

It is *not possible* to test such a requirement exhaustively.

Hard-to-test requirements (3)

Some non-functional requirements

Some non-functional requirements, such as reliability requirements, can only be tested with a large test set.

Designing this test set does not help with requirements validation.

But the tests are still necessary for system delivery and for test during development.

Make a test?

- We can use test cases,
- analysis or
- inspection to decide
- If a requirement is has been satisfied

Dilbert Testing

Summary for testing requirements

- Each requirement should have a test to determine whether it has been satisfied (or not)
- Non-functional requirements can be hard to test objectively
- Requirements tests can be a form of contract for delivery
- Thinking of ways to test your requirements and test them is hard. You need to practice that skill
- Requirements can also be tested by analysis or inspection

Testing requirements

Convince me!

Requirements Specification Document

Tell me!

Prototyping requirements

Show me!

Domain Modelling

- Definition: A domain is a representation of real-word conceptual classes, not of software components. It is not a set of a set of diagrams describing software classes, or software objects with responsibilities. - Larman

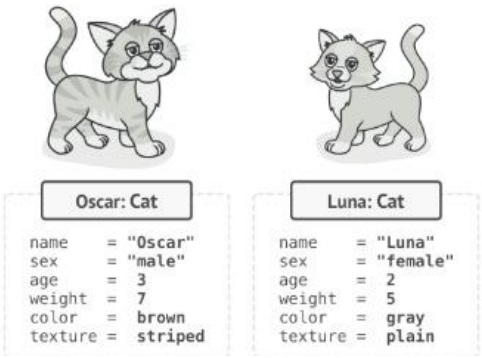
- So, what is conceptual class?
- conceptual class is an idea, thing, or object
- it represents things in the real world, not classes in code

- How to capture them?
- UML class diagram

Domain Modelling

- **Classes**: a class is a template of objects. Class is an abstraction of all existing and new objects with similar characteristic.

- **Objects**: objects are instances of a class. Data stored inside the object fields is often references as state and all object methods define it's behaviour.



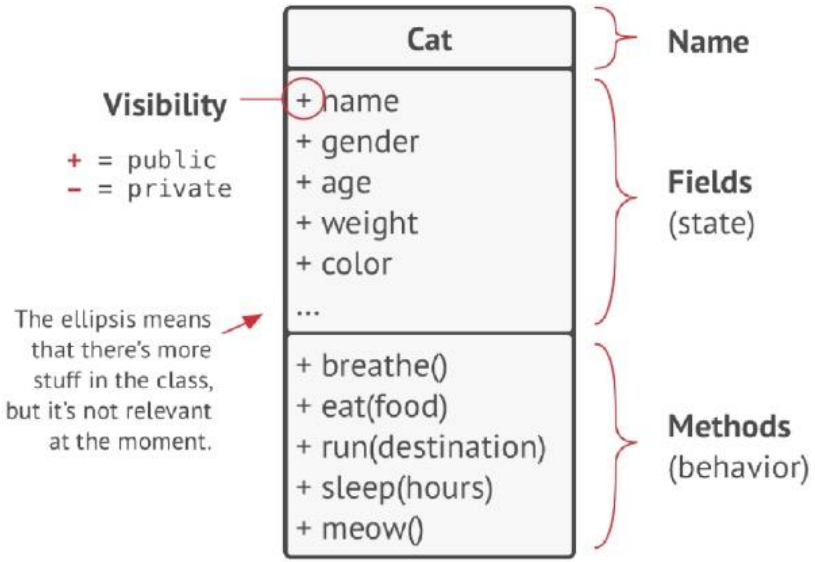
Domain Modelling

- **Components**:

Classes: rectangle containing the class name, and optionally fields/states/attributes and methods/operations

- **Classes have relationships that are**:

  Associations
  Dependency
  Aggregation
  Generalization



Domain Modelling

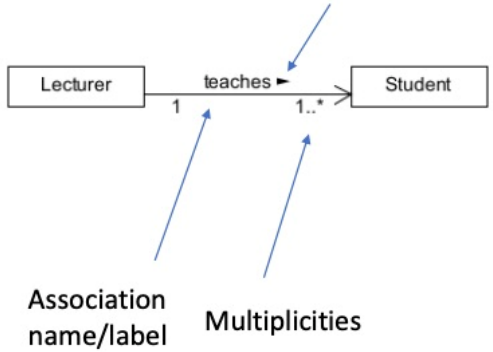
- **Associations**: a type of relationship in which one object uses or interacts with another

UML Association. Professor communicates with students.

```
public class Lecturer {
    private ArrayList<Student> students;
}

public class Student {
...
}
```

- Reading direction arrow
- It has no meanings except reading the direction of association label.
- It is different from association direction.



Domain Modelling

- **Dependency**: a weaker variant of association that usually implies that there is no permanent link between objects.

```
public class Professor {
        private void mySalary(Salary salary) // I am dependent on Salary
}
public class Salary {
            ...
}
```

UML Dependency. Professor depends on salary.

Domain Modelling

- **Aggregation**: appears between classes when one object merely contains a reference to another.

Aggregation – Code Example

```
public class Department {
    Professor professor;
    public void setProfessor(Professor professor) {
      this.professor = professor;
}

public class Professor{
…
}
```

UML Aggregation. Department contains professors.

Even if you delete class Department, Professor will exist

Domain Modelling

- **Composition**: a type of aggregation shows a "whole-part" relationship between two objects, one of which composed of one or more instances of the other.

Composition – Code Example

```
public class University{
    Department department;
    public void setDepartment(){
        this.department = new Department();
}
public class Department{
…
}
```

UML Composition. University consists of departments.

If we delete class University, Department won't exist

Testing requirements

Convince me!

Requirements Specification Document

Tell me!

Prototyping requirements

Show me!

Why Prototype?

- Prototyping is commonly a means for validating the software engineer's interpretation of the software requirements, as well as for eliciting new requirements.

- The advantage of prototypes is that they can make it easier to interpret the software engineer's assumptions and, where needed, give useful feedback on why they are wrong.

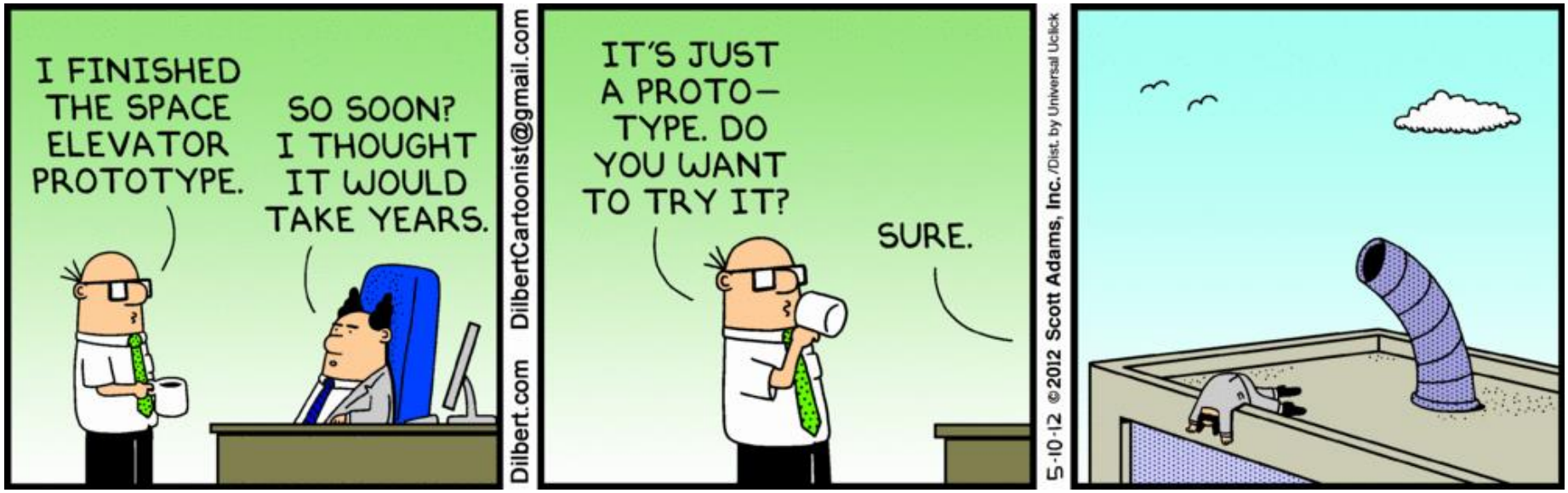
- For example, the dynamic behaviour of a user interface can be better understood through an animated prototype than through textual description or graphical models.

Source: SWEBoK Guide 6.2 Prototyping

Prototyping

- Prototypes for requirements validation demonstrate the requirements and help stakeholders discover problems
- **Analysis prototypes** can be lightweight and need not contain SW at all; they are for discovering what the users want
- **Validation prototypes** should be complete, reasonably efficient and robust. It should be possible to use them in the same way as the required system
- User documentation and training should be provided

Dilbert prototyping



Types of Prototype

- Software: create a new executable prototype
  - Use rapid prototyping tools (eg UI as html forms)
- Write a first draft of the user interface
  - Incremental development
- Modify existing SW to create a prototype for the new system
- Generate screen mock-ups with drawing software (eg powerpoint is fine)
- Draw mock-ups on white-boards or paper

Provide three examples of software that are amenable to the prototyping model. Be specific.

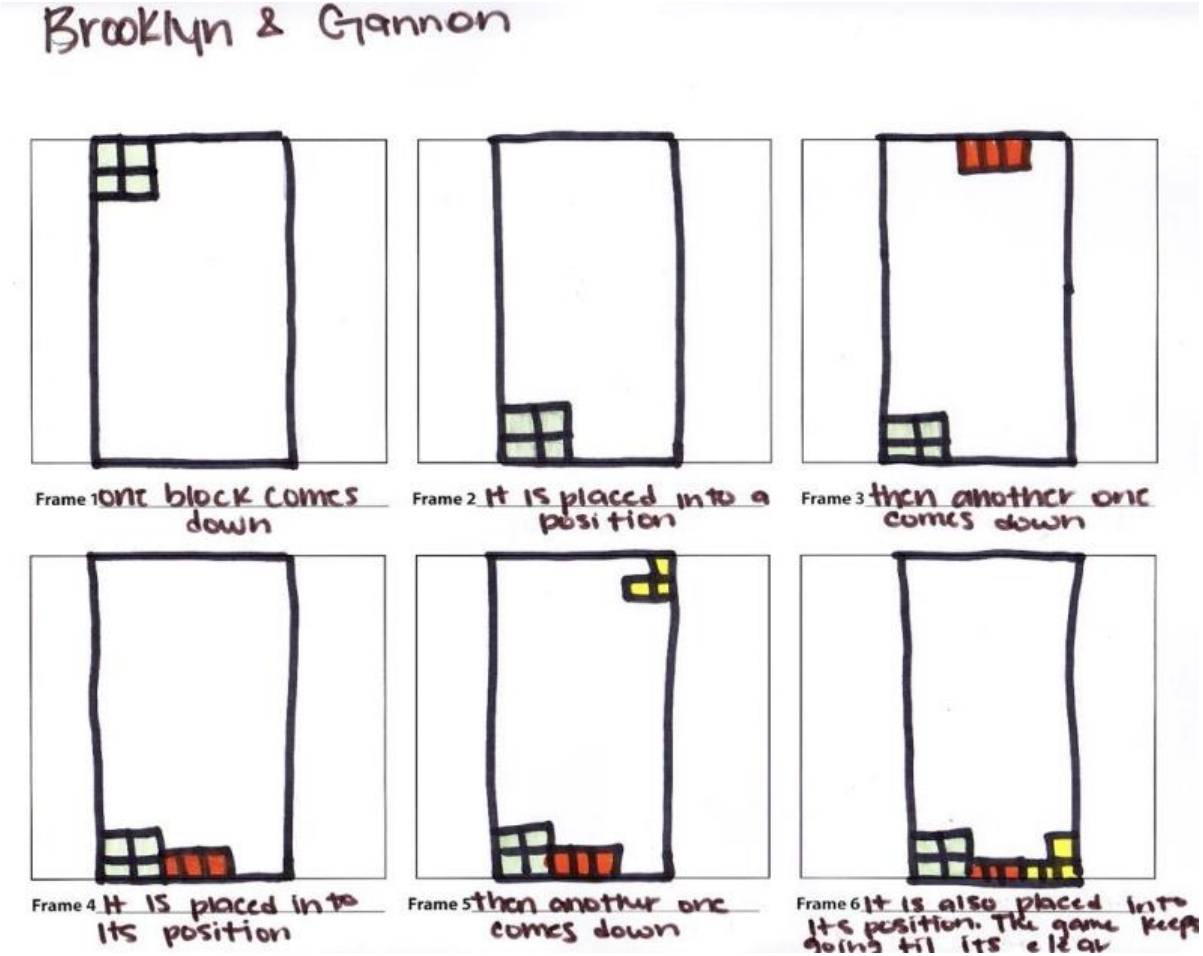
- Software applications that are relatively easy to prototype almost always involve **human-machine interaction** and/or heavy computer graphics.

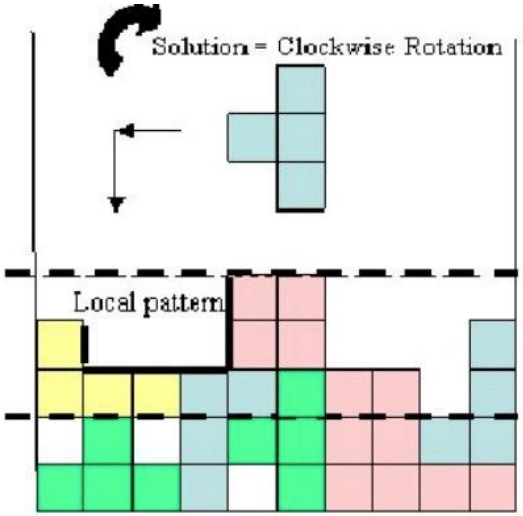
- Other applications that are sometimes amenable to prototyping are certain classes of **mathematical algorithms**, subset of **command driven systems** and other applications where results can be easily examined without real-time interaction.

- Applications that are difficult to prototype include control and process control functions, many classes of real-time applications and embedded software.

Source Pressman 2.6

Prototyping

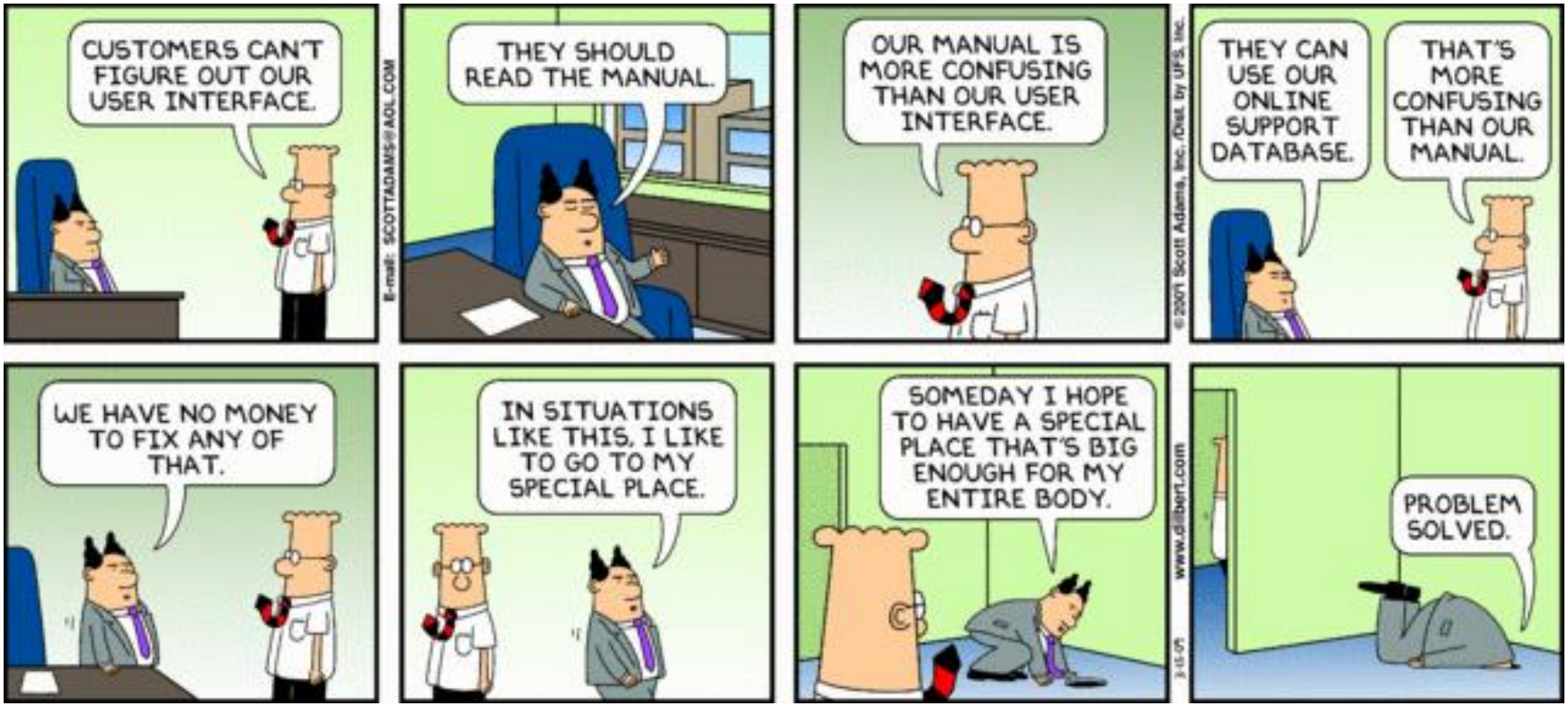




User manual development

- Writing a user manual from the requirements forces a detailed requirements analysis and thus can reveal problems with the document
- Information in the user manual
  - Description of the functionality and how it is implemented
  - Which parts of the system have not been implemented
  - How to get out of trouble
  - How to install and get started with the system

Dilbert user manual



Summary

- **Prototyping** is effective for requirements validation if it is in the initial stages of software engineering.
- Prototypes can be physical (eg on paper) or written as software
- Writing a user manual early can help to clarify requirements