Introduction to Software Engineering (ISAD1000)

Lecture 2: Functional Requirements

Updated: 15 th February, 2022

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

The Client

- The client is the person, company, department, etc. that you are building the software for.
- Communication between you and the client is vital. You must understand the client's *requirements*.
  - If you don't, you will build the wrong software! (And then you won't be very popular.)
- Software requirements can be complicated:
  - The client rarely knows precisely what they want.
  - Things often change mid-way through a project.
- (This subject is covered in depth in Requirements Engineering – CMPE2002.)

Types of Software Requirements

- *Functional* requirements deal with the software's functions – what it should *do*.
  - e.g. save file, print, launch missile, make a backup, etc.
- *Non-functional* requirements deal with the software's characteristics – what it should *be*.
  - e.g. fast, secure, easy to use, reliable, etc.
  - We won't discuss these in detail until lecture 7.
- We need <u>a lot more information</u> before these become proper requirements:
  - What are the *steps* to print something?
  - Exactly how fast and reliable should the software be, and how do we measure that?

Approaches to Requirements

- Some projects often use a Software Requirements Specification (SRS):
  - A document listing all the requirements (functional and non-functional).
  - Often forms a legal contract between the client and the developers.
  - Must have the right level of detail.
  - Errors in the SRS can lead to huge problems.
- Alternatively, some other projects use agile methods:
  - The client is closely involved throughout the project.
  - The software is developed incrementally.
  - Requirements are determined incrementally.

Requirements Principles

Whatever approach you take, there are some common principles:

"What", not "how"

- Software requirements only say *what* the software should do.
- No point figuring out "how" until you've nailed the "what".

Resolving uncertainty

- Projects always begin with great uncertainty.
- What on Earth is the client talking about?!
- We want to reduce that uncertainty as quickly as possible.

Divide and conquer (relevant to all SE activities really)

- Requirements must be broken down.
- Humans (you) understand things much better in small pieces.

Software Requirements Specification (SRS)

- With an SRS, all (or most) requirements are specified up-front.
- You negotiate with the client over the scope and the price.
  - "Scope" means a few *slightly* different things.
  - In general, it means "how much we're talking about" – what's relevant and what's irrelevant.
  - It *can* mean "how much you're going to deliver".
- *Then* you design and build the software.
- Intended to resemble a traditional engineering project.
- When writing an SRS, you must be unambiguous, consistent, complete and concise.
  - Otherwise, software developers will be confused.
  - They'll build the wrong system!
- You must also be very careful that you understand what the client needs.

SRS Layout

Adapted from the IEEE recommendations for SRSs [1]:

1. Introduction
2. Overall description
3. External inferface requirements
4. Functional requirements
5. Other Non-functional requirements

[1]IEEE Std 830-1998

SRS Layout

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, acronyms, and abbreviations
   - 1.4 References
   - 1.5 Overview
2. Overall description
3. External inferface requirements
4. Functional requirements
5. Other Non-functional requirements

SRS Layout

1. Introduction
2. Overall description
   - 2.1 Product perspectives
   - 2.2 Product functions
   - 2.3 User characteristics
   - 2.4 Constraints
   - 2.5 Assumptions and dependencies
3. External inferface requirements
4. Functional requirements
5. Other Non-functional requirements

SRS Layout

1. Introduction
2. Overall description
3. External inferface requirements
   (Particular kind of non-functional requirements and constraints.)
   - 3.1 User interfaces
   - 3.2 Hardware interfaces
   - 3.3 Software interfaces
   - 3.4 Communications interfaces
4. Functional requirements
5. Other Non-functional requirements

SRS Layout

1. Introduction
2. Overall description
3. External inferface requirements
4. Functional requirements
   (Can be organised in various ways; e.g. by mode, by user class, by feature, by stimulus, etc.)
5. Other Non-functional requirements

SRS Layout

1. Introduction
2. Overall description
3. External inferface requirements
4. Functional requirements
5. Other Non-functional requirements
   5.1 Performance requirements
   5.2 Safety requirements
   5.3 Security requirements
   5.4 etc.

Changing Requirements

- If the requirements *don't* change, life is easy.
  - You agree on the requirements.
  - You design, implement and test the product.
  - You deliver it.
  - Profit!
- But this never really happens – requirements always change.
- Why?
- Because it's near-impossible to get the details right, initially.
  - Tradeoffs must be made – e.g. cost vs. performance.
  - Software works in complex environments.
    - Different kinds of users, with different knowledge, preferences, working styles, etc.
    - Different kinds of hardware.
    - Interactions with many other different systems.
  - You write software because what the client needs *has never been done before* (at least, not exactly the right way).

Changing Requirements in Traditional SE

- Faulty requirements are very expensive to fix late in a project.
  - You need to redo a lot of work.
  - The later the requirements are left unfixed, the more work you will have to redo.
- If requirements are going to change, the sooner the better.
  - (To minimise the damage.)
- We use "inspection" to help make this happen.
  - Formal process for reviewing the SRS, and anything else we produce.
  - Acts as a rigorous approval process.
  - If inspection finds too many faults, then you're not finished.
  - Not perfect, but better than nothing.
  - Discussed further in Lecture 4.

Agile Development

- *Agile methods* take a different approach to requirements.
  - The whole project is a series of short bursts of development.
  - These are called iterations, or increments, or “sprints”, etc.
- The requirements, and the product, are built-up incrementally.
  - Rapid, incremental delivery.
  - Faulty requirements are very quick to find, because the client can see almost immediately.
- Not much rework is required when you get something wrong.
  - Damage is (basically) limited to one iteration’s worth of work.
- You and the client still talk about scope and price, but *for each iteration* (not the whole project).
- Several distinct agile methods: Scrum, eXtreme Programming, Crystal, etc.

Agile Iterations

Each of these iterations might be 1–4 weeks in length.

- First iteration:
  - Agree with client on initial (very, very basic) requirements.
  - Create initial (very, very basic) product.
- Second iteration:
  - Agree with client on additional requirements.
  - Update product.
- Third iteration (as above)
- . . .
- Final iteration (for now):
  - Agree with client on last requirements (for now).
  - Update product.

Prototyping

- Build a dummy version of the software very quickly.
  - Used to “try out” an idea, or set of ideas.
  - Often just the user interface (the visible part) of the software.
- Thrown away afterwards! (Because you cut corners in order to build it quickly.)
- Prototypes help resolve uncertainty.
  - Gives the client ideas.
  - Gives the client something to criticise, leading to better requirements.
  - Gives *you* ideas too.
  - Helps put you and the client “on the same page”.
- Perhaps a way for SRS-based projects to become more “agile”.

User Stories

- Agile methods often work with *user stories*.
- Tiny snippets of text with 3 parts:
  - User role – who are we talking about?
  - Ability – what does this user want to do?
  - Benefit – why is this important?
- Phrased like either of the following:

  As a [user role] I want to [ability] so that [benefit].

  OR

  In order to [benefit] as a [user role], I want to [ability].

  (Choose one form and use it consistently.)
- Created by you and the client, together.
- Forms the basis of the next iteration (i.e. what you're working on next).

User Story Examples

As a student, I want to view all prac classes so I can choose which one to attend.

As a student, I want to register for pracs so I don't miss out.

As an administrator, I want to add prac classes so there are classes available to students.

As an administrator, I want to remove prac classes so there aren't more classes than necessary.

User Stories: Why?

- Divide-and-conquer, and resolving uncertainty.
  - Breaks down the requirements.
  - Makes it obvious *what* features the software needs, and *who* needs them.
  - The *why* (the "benefit" part) ensures you and the client both understand the importance of each feature.
- Requirements $\rightarrow$ Planning! (Remember last week's lecture.)
  - Each user story is a *task* in your work breakdown structure.
  - Play planning poker to estimate how long it will take.
  - Some user stories will naturally depend on others.

User Story Common Sense

- User stories are about users – the people who will actually use the software that you're about to make.
- User stories are *not* about:
  - You. You're the developer, not the user (unless you're making the software for yourself).
  - Things that aren't people.
- In particular, the following are very silly:
  - As the software, I want to...
  - As the database, I want to...
  - As the payment system, I want to...

  A database can't call itself "I", nor can it "want" anything.
- Don't mix up "ability" and "benefit".
  - The **ability** is what the software should do.
  - The **benefit** is why we care about it.

Backlogs

- Agile methods use a *backlog* of user stories.
  - A queue of features waiting to be implemented.
  - Similar in principle to a work breakdown structure (WBS).
  - The difference: WBSs are fixed up-front; backlogs evolve.
- User stories are added to the backlog throughout the project.
  - A process of discovery (resolving uncertainty).
  - You and the client agree on what to add, based on what you've learnt so far.
  - This is done repeatedly and regularly.
- User stories are also removed from the backlog when they are finished.
  - User stories represent units of work.
  - You (as a software engineer) carry out the work, story-by-story.
  - Start with the highest priority user stories.

Stakeholders

- The question "who?" is very important.
- *Who* cares about the software, and who is affected by it?
- The obvious stakeholders:
  - The users, obviously;
  - The developers, since they have to build it;
  - The client, who (probably) has to pay for it.
- The not-quite-as-obvious stakeholders:
  - The passengers on an aircraft, if your software controls the aircraft;
  - Victims of crime, if your software facilitates police operations;
  - Website owners, if your software is an Internet search engine;
  - and many more.

Actors

- Some stakeholders are *actors*.
- An actor is an entity that:
  - interacts with the system being created, but which
  - is not part of the system.
- Some actors are human: administrators, managers, employees, students, lecturers, etc.
- Some actors may be non-human:
  - Databases – external systems for storing complex data;
  - Sensors – devices for observing the real world;
  - Robots;
  - etc.

Databases

- We'll talk about databases a lot when writing use cases.
- A database is (usually) an external piece of software for storing data.
  - Usually large amounts of complex, structured data.
  - Sometimes runs on a physically separate computer.
- You use a database to avoid worrying about issues like:
  - How to permanently store data yourself.
  - How to access it efficiently.
  - How to keep it safe, in case something fails.
- That means your system must communicate with the database to *store* and *retrieve* data when necessary.
- Databases are a classic non-human actor.
- *However*, not all systems have or need a database!
  - Don't automatically assume one is present.

Databases: What You Need to Know

- Databases are discusssed in depth in Database Systems – ISYS1001.
- You are not expected to know how they work in this unit, *except* that:
  - **Databases store, delete and retrieve data when asked.**
  - Databases don't make decisions or take actions.
  - Databases don't communicate directly with users, only with other software.
    - (Note: some people confusingly say "database" when they mean "a software system that *uses* a database". Don't do that here!)
  - Databases don't connect different systems together.
    - Technically they *could*, but it's better if different systems communicate directly.
    - A database should only be accessed from *one* software system.

Some External Systems are Irrelevant

- There are many external systems that we don't care about (when discussing functional requirements):
    - the operating system (Linux, Windows, OS X, etc.),
    - input/output devices (hard drives, keyboards, monitors, etc.),
    - networking infrastructure (network cards, routers, ISPs, etc.),
    - and possibly others.
- Don't state the obvious.
    - Of course there's a monitor. Of course the system will use it.
    - But talking about *that* means less focus on important details.
- Be concerned with "what", not "how".
    - "The system retrieves customer details from a database."
    - This might involve a hard drive, or a network card. But we don't care about those low-level details yet.
- Real actors just use these systems to interact with the system-under-construction.
    - Human actors use the keyboard, mouse, monitor and OS.
    - Non-human actors use the OS and networking.

Use Cases

- A use case ("case of use") is a description of how the software system will interact with the outside world.
- Many use cases may be needed for any given software system.
- Each use case describes a functional requirement.
  - "Functional requirement" is the concept.
  - "Use case" is the written description of it.

  (In fact, use cases can also describe *parts* of functional requirements, or several functional requirements together.)
- We'll use Alistair Cockburn's methodology [1] here (simplified).

[1] Cockburn (2000) Writing Effective Use Cases, ISBN 0-201-70225-8

Parts of a Use Case

Every use case is made up of the following:

Header
: Basic information about the use case: (1) what it's for, (2) who it involves, and (3) when it happens.

Flow of Events (FoE)
: A sequence of numbered steps:
  - Describes how the system interacts with its actors.
  - Describes what should "normally" happen.

Extensions
: A set of alternative scenarios:
  - Describes what happens if things go wrong, or differently.

Example System

- Say we're developing a video sharing website (like YouTube).
- We would write a use case for *each* of the following features:
  - Watching a video;
  - Uploading a video;
  - Reporting a video (for harassment, copyright infringement, etc.);
  - Posting a comment on a video;
  - etc. (You might be able to think of other features too.)

Complete Example (for Later Reference)

Use Case 7. Uploading a Video *[Note: this is only one use case. Others are needed for the other features.]*

Goal:	to allow a registered user to upload and share a video.
Primary actor:	uploader (human).
Secondary actors:	database.
Precondition:	uploader is logged in, and has at least 1MB of remaining space.
Trigger:	uploader selects the "upload video" option.

Flow of Events:

1. The uploader chooses a file to upload.
2. The system receives the uploaded file.
3. The system converts the video file to the site's standard format, while displaying the progress.
4. The uploader enters video details, including name, description, location & tags.
5. The uploader selects the publish option.
6. The system stores the finished video and its details in the database.

Extensions:

2A – The uploaded file is not a valid video file.

1. The system asks the uploader to choose another file.
2. The use case resumes at step 1.

3A – The system detects a match for an existing copyrighted video or soundtrack.

1. The system notifies the uploader.
2. The use case ends.

3B – The uploader has insufficient space to store the video.

1. The system calculates how much of the video can be stored.
2. The uploader selects a segment of that length.
3. The system replaces the whole video with the segment.
4. The use case resumes at step 4.

Example, Part 1 – Use Case Header

Use Case 7. Uploading a Video

Goal:	to allow a registered user to upload and share a video.
Primary actor:	uploader (human).
Secondary actors:	database.
Precondition:	uploader is logged in, and has at least 1MB of remaining space.
Trigger:	uploader selects the “upload video” option.

. . .

Actors and Goals

- Each use case has one *primary* actor:
  - Always a person!
  - The person who is most directly involved.
  - Usually the person who initiates the use case.
    - (On rare occasions, the system itself initiates the use case.)
- The *goal* of the use case is the goal of the primary actor.
  - What does the primary actor want to happen?
- A use case often also has *secondary* actor(s):
  - Any other people or external systems involved.
  - Can be human or non-human.
  - They have some responsibilies to the primary actor.
- Different use cases can have different actors.
  - The primary actor in one use case *could* be secondary in another.
- Note: the software that you're writing is NOT a primary or secondary actor.

Triggers and Preconditions

- A "trigger" is an event that starts the use case's flow of events (FoE).
  - May or may not be a user action.
  - Could (alternatively) be a time-of-day.
- A "precondition", if false, will *prevent* the use case from starting; e.g.
  - The user must be logged on.
  - The user must have permission to do X.
  - The system must be in mode M.
- A use case starts if
  1. The trigger occurs, AND
  2. At that moment in time, all preconditions are true.
- Don't confuse preconditions with:
  - Triggers; OR
  - Conditions that arise *after* the use case starts.

Scope

- Don’t describe things that are “out of scope”.
- The use case is only concerned with the operation of the software system.
  - In the real world, use cases can specify a wider or narrower scope.
  - For ISE we’ll focus on “system scope”.
- Valid, in-scope preconditions (which the system can verify):
  - “Uploader has logged in”.
  - “Uploader has not been banned from uploading”.
  - “Uploader has at least 1MB of remaining allocated space”.
- Invalid, out-of-scope preconditions:
  - “Uploader is awake.”
  - “System has been installed.”
  - “Universe exists.”

Example, Part 2 – Flow of Events

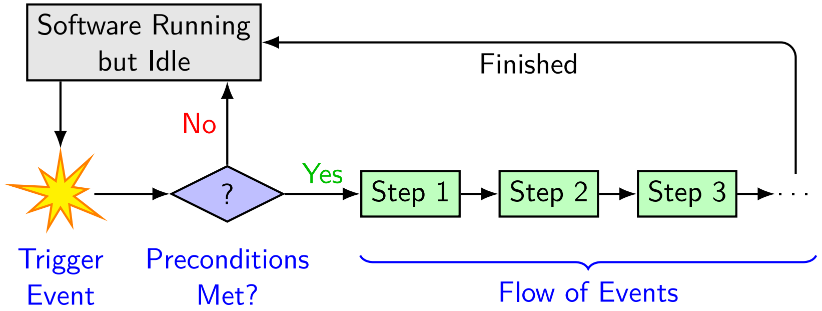
Here is an example Flow of Events for uploading a video:

1. The uploader chooses a file to upload.
2. The system receives the uploaded file.
3. The system converts the video file to the site's standard format, while displaying the progress.
4. The uploader enters video details, including name, description, location and tags.
5. The uploader selects the publish option.
6. The system stores the finished video and its details in the database.

A Closer Look at the Flow-Of-Events (FoE)

- As a rough guide, the FoE should have about 3–10 steps.
- Steps can be. . .
- Interactions initiated by an actor. e.g.
  - *"The President enters the nuclear launch code."*
  - *"The General selects the targets."*
- Interactions initiated by the system. e.g.
  - *"The system retrieves the actual launch codes from the database."*
  - *"The system displays the time remaining to the General."*
- Significant processing done by the system (possibly combine-able with other steps). e.g.
  - *"The system calculates the blast radii and estimates damage, as a percentage, to previously-specified infrastructure."*
- Some actors – especially databases – never initiate an interaction, but they can still be part of one.

Triggers, Preconditions and Flow-of-Events – A Visual Guide



- Shows the basic conceptual order of things in a use case.
  - Don't draw this yourself – it's just for your understanding.
- While the software is "running but idle", it's looking for trigger events for *any* use case.
  - Remember: there are generally *many different* use cases.

Flow-of-Events – Things to Avoid

- Don't show interactions between two actors.
  - ~~*"The President instructs the General to abort the launch."*~~
  - This is outside the scope of the system.
- Avoid technology-specific language.
  - ~~*"The General presses the abort button."*~~
  - Instead: *"The General **selects** the abort **option**."*
  - Use cases should capture the *essence* of the interactions, not the implementation details.
  - Don't limit your design options unnecessarily.
- Avoid the passive voice, and use the active voice.
  - ~~*"Missile launch is aborted."*~~
  - Instead: *"**The system** aborts the missile launch."*
  - Always state explicitly *who* or *what* carries out each action.
  - Reduces uncertainty and ambiguity.

Example, Part 3 – Extensions

2A – The uploaded file is not a valid video file.

1. The system asks the uploader to choose another file.
2. The use case resumes at step 1.

3A – The system detects a match for an existing copyrighted video or soundtrack.

1. The system notifies the uploader.
2. The use case ends.

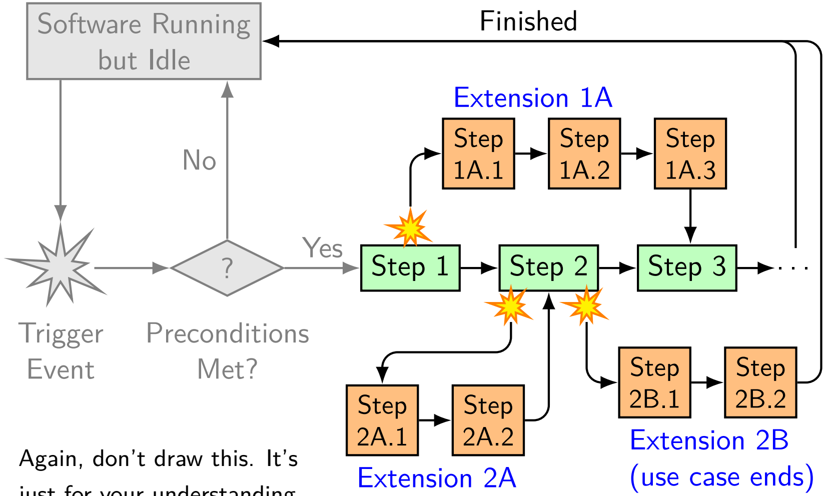
3B – The uploader has insufficient space to store the video.

1. The system calculates how much of the video can be stored.
2. The uploader selects a segment of that length.
3. The system replaces the whole video with the segment.
4. The use case resumes at step 4.

Extensions

- The main flow of events describes the "happy days" scenario:
  - everything goes right, and the goal is achieved.
- Variations to this main scenario are called *extensions*.
- There may be several extensions.
  - *Sometimes* there are zero extensions, but not often.
- Each extension "branches off" a particular point in the FoE.
  - The extension describes what happens if something specific goes wrong (or differently) at that point.
  - There may be several extensions at the same point.

Extensions – A Visual Guide



Again, don't draw this. It's just for your understanding.

Parts of an Extension

Each extension has:

An ID	– like “4B”. This identifies two things: ▶ The step in the FoE (e.g. 4) that something might go wrong (or differently). ▶ Distinguishes one extension from another, if they occur at the same point.
A condition	– states why the extension takes place.
A list of steps	– an alternate “mini” flow-of-events.
An exit point	– what happens afterwards: ▶ The use case might simply end, if the error is unrecoverable. ▶ The use case might resume at a particular step in the FoE.

Extensions – Things to Avoid

Avoid overly-generic problems. e.g.

- ~~Database network connection is lost.~~
- ~~Out of memory.~~
- ~~User closes program.~~

Because:

- These problems would apply to almost *all* use cases.
- We don't want to repeat the solution over and over and over – unnecessary work!

Avoid problems that the system can't address. e.g.

- ~~User is fired from their job.~~
- ~~Computer explodes.~~

Because:

- We write use cases to guide us in writing the software.
- Anything that the software can't help with is beyond the scope of the exercise.

What about “IF” statements?

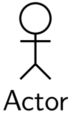
- In learning how to code, you will encounter “IF” statements, which allow your program to make a choice between two alternatives.
- Tempting to use these in the use case flow of events, instead of extensions.
- Actually not a good idea.
  - Extensions should be *separate* from the main FoE.
  - This makes it easier to see, overall, what’s going on.

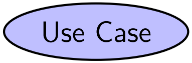
Design Scope and Goal Level (advanced)

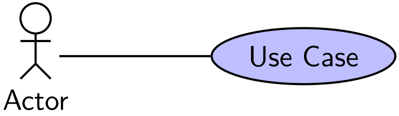
- Use cases also have a particular “design scope” and “goal level”.
- “Design scope”: what *thing* are you talking about?
  - Your software/system, OR
  - Just a part of your software/system (a “subsystem”), OR
  - The organisation that will use your software/system.

  (nb. Give a name to thing in question.)
- “Goal level”: how much *functionality* are you talking about?
  - A single functional requirement (“user goal”), OR
  - A single step within one functional requirement (“subfunction”), OR
  - A sequence of functional requirements (“strategic”).
- Scope and level help you deal with a manageable amount of information.

UML Use Case Diagrams

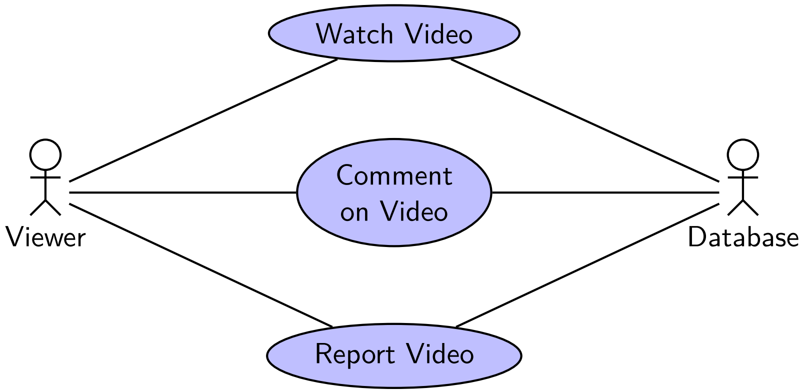
- A diagram that shows: (1) the names of all use cases, (2) all the actors, and (3) the relationships between them all.
- An actor is a stick figure: 

- A use case is an ellipse: 

- Lines indicate participation: 

- Use Case Diagrams *don't* show the preconditions, trigger, flow of events or extentions.
- They don't distinguish between primary and secondary actors.

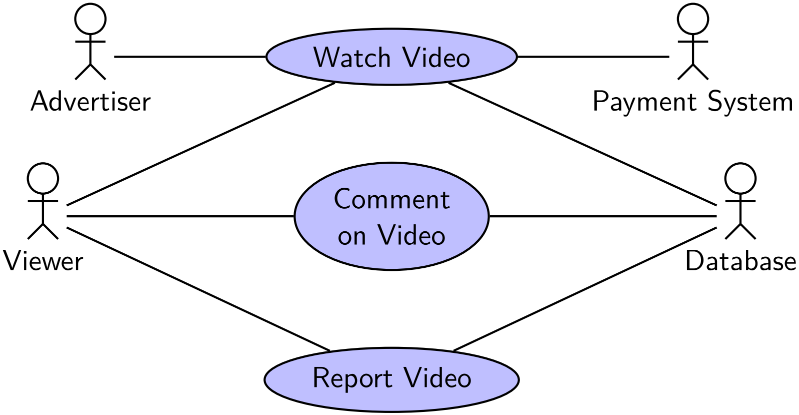
Rudimentary Use Case Diagram



- The viewer and database actors participate in three use cases here.

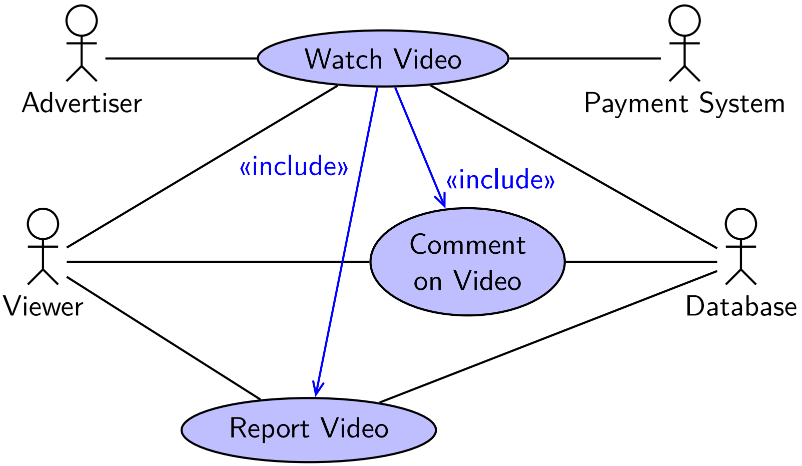
Who Does What?

- Use case diagrams clearly show *which* actors participate in *which* use cases.
- Some actors only participate in specific use cases.



The «include» Relationship

Some use cases may "include" others, as extensions, or steps in the flow of events.



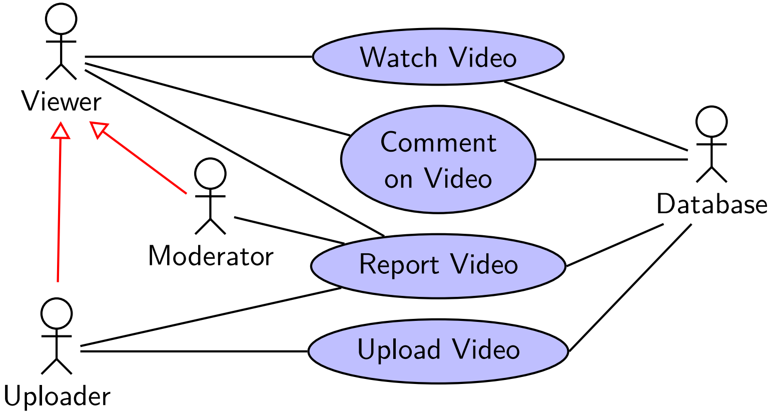
The «include» Relationship

- We haven't discussed use cases "including" each other before, but it can happen.
- You can "include" one use case inside another.
  - The included use case may be a step inside the larger use case.
  - However, the included use case is still a separate use case too.
- «include» represents an important SE principle: *reuse*.
  - "Comment on Video" and "Report Video" can be done:
    1. Separately, OR
    2. As part of "Watch Video".
  - To write all three use cases, we could have a lot of duplication.
  - Duplication wastes effort and leads to ambiguities.
  - Using «include» is clearer and more concise.
- (There's also an «extends» relationship, but Alistair Cockburn discourages this due to unnecessary confusion.)

Actor Generalisation

Some actors can do everything that others can do, plus more.

- Below, some users are “uploaders” or “moderators”, but are still *also* “viewers”.

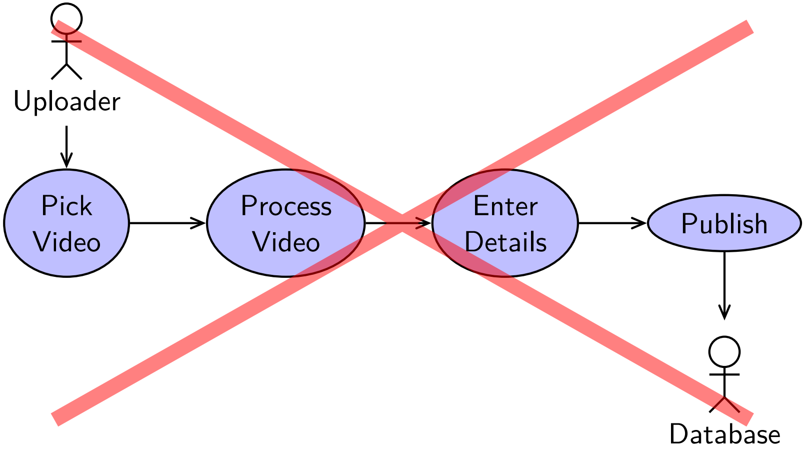


Use Cases vs. Use Case Diagrams

- Use case diagrams have *limited* usefulness.
    - They show an overview only.
- The fully-written use cases contain most of the information.
- In particular, use case diagrams *cannot* show the flow of events.
    - There's no way to show ordering/sequence of steps in the diagram.
    - «include» cannot be used this way, nor can any other relationship.

Use Case Diagrams Are Not Flow Charts!

This is *not* a use case diagram:



That’s all for now!