Introduction to Software Engineering (ISAD1000)

Lecture 3: Non-Functional Requirements

Updated: 18th February, 2020
Minor revisions: 13th March, 2022

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

Outline

Requirements

- In Lecture 2, we discussed *functional* requirements:
  - FRs describes what the system should do.
  - A “function” transforms inputs into outputs.
  - Represented by user stories and use cases.
- We briefly mentioned *non-functional* requirements.
  - NFRs describes what the system should be; how the system works.
  - Consider about what qualities/characteristics/attributes the system must have?
  - These *mostly* end in “. . . ity”: usability, security, reliability, etc.
  - Can also be represented by user stories.
  - Can be attached to use cases.
    - (Though there’s no such thing as a purely non-functional use case.)
- Both are part of an SRS.

Non-Functional Requirements (NFRs)

- We don't just want "print", "save", "purchase", "find directions", etc.
  - Not everything the client needs is an actual function.
- We *also* want performance, reliability, usability, security etc.
  - Obviously, these are always desirable.
  - But the client usually *needs* specific *levels* of them.
  - When this happens, we have Non-Functional Requirements (NFRs).
- NFRs describe attributes or characteristics (not functions) the system must have.
  - Not generically-good qualities, but specifically-required ones.
  - They might apply to particular parts of the system, or to particular functions, or to the whole system.

NFR Examples

Some brief examples (more later):

- A performance requirement:

> The game must render the first level at 120 frames per second on a 2015-model Brand-XYZ laptop.

- A usability requirement:

> The game must allow the player to access special items with no more than 2 button clicks.

- A reliability requirement:

> The game must not exhibit unexpected restarts more than once every two months of playing time, on average.

Is There a Format?

- Sort of!
- NFRs typically start with “The system must. . . ” or similar.

> The system must format class lists in a table, with columns for name and ID, and one row for each student.

- They may relate only to a *particular feature* of the system:

> The system’s emailer component must format class lists...

- They can also be written as user stories:

> As a lecturer, I want my class lists shown in a table (with name & ID columns and a row for each student), so I can scan through them easily.

  - Don’t mix-and-match. Write them *all* as user stories, or not at all. Depends on how your organisation does it.
- The examples in this lecture are all **small,** due to space limits.

Where Do NFRs Fit In?

- Functional requirements are mostly separate.
  - You can work on one function, finish it, then work on another.
- NFRs are "*cross-cutting*":
  - They often govern the whole system, or large parts of it.
  - They make no sense on their own.
- NFRs are useful in both conventional and agile methods.
  - In non-Agile software projects, included in the Software Requirements Specification(SRS) under a separate topic. (see Lecture 2).
  - In agile projects, mostly included as constraints in backlogs.
- NFRs may be listed alongside particular use cases.
  - Some NFRs may only apply to specific use cases.
  - A use case as a whole represents a *functional* requirement.

Why Specify NFRs?

- Why do we need to consider NFRs?
- What if we didn't?
  - A word processor that does everything you need, but which randomly corrupts your work every few minutes?
  - An retail website that allows you to make a purchase, but it takes users hours to figure it out?
  - An mapping application that takes ten minutes to put a map on your screen?
- NFRs allow software engineers to make the right design choices.
  - There are many, many ways to design a system.
  - Some are better than others, *if you want certain qualities or characteristics*.
- But. . . couldn't we just maximise performance, security, etc. anyway?
  - Then we wouldn't need any NFRs!
  - We'd just make perfect systems all the time.
- Yes, if you're an SE superhero !!!.

Why Specify NFRs? (2)

- For non-superheros, these qualities cost money, and they are only important *to a certain extent*.
- Software engineers need to know when to start and stop caring about them (NFRs).
  - *Not* because they want to cut corners.
  - Because they want to provide the cheapest solution that meets the client's needs.
  - Once you've finished the software, the client will want proof that it meets its requirements. Specifying the NFRs helps to check this.
- We specify NFRs so we know **what level** of performance, security, etc. is needed.
  - We aim at providing that level.
  - We don't spend (too much) time providing anything more.
  - We can test and verify whether the intended level of requirement is met or not.

Performance Requirements

- Software performance relates to:
  - How much time it takes to respond to an event;
  - How much data it can process per second;
  - How efficient its use of resources is.
    - “Resources” can be many things.
    - But typically memory and network bandwidth.
- Poor performance can reduce the software’s usefulness.
  - If it takes too long or is too jarring. . .
  - Or if it uses too many resources. . .
  - Then perhaps it just won’t meet the needs of the client.
- However, performance requirements depend on the situation!

Hardware Factors

- *Overall* performance is as much due to hardware as software.
- Obviously, you can improve performance with a better CPU, more memory, etc.
- Therefore, to evaluate *software* performance, we need to take hardware out of the picture.
  - You'll be hired to write good software, not to tell the client they just need faster computers!
- To do this, our performance requirements need to specify a fixed set of hardware.
  - Specify a particular model of CPU, quantity of RAM and graphics card.
  - Specify a particular overall model of PC.

Response Time

- Most software *responds to something* at some point.
- Often, it's important that this happens within a certain amount of time; e.g.

> The navigation system must recalculate the route within 5 seconds of receiving updated user coordinates. (To be evaluated on a 2018-model MacBook.)

Data Processing Rate

- Many systems perform continuous/ongoing tasks.
- Often it makes more sense to ask *how much the system can do* over a period.
- e.g.

> The video encoder must be able to process 120 video frames per second. (To be evaluated on. . .)

> The system must perform facial recognition on up to 100 faces per second. (To be evaluated on. . .)

Resource Utilisation

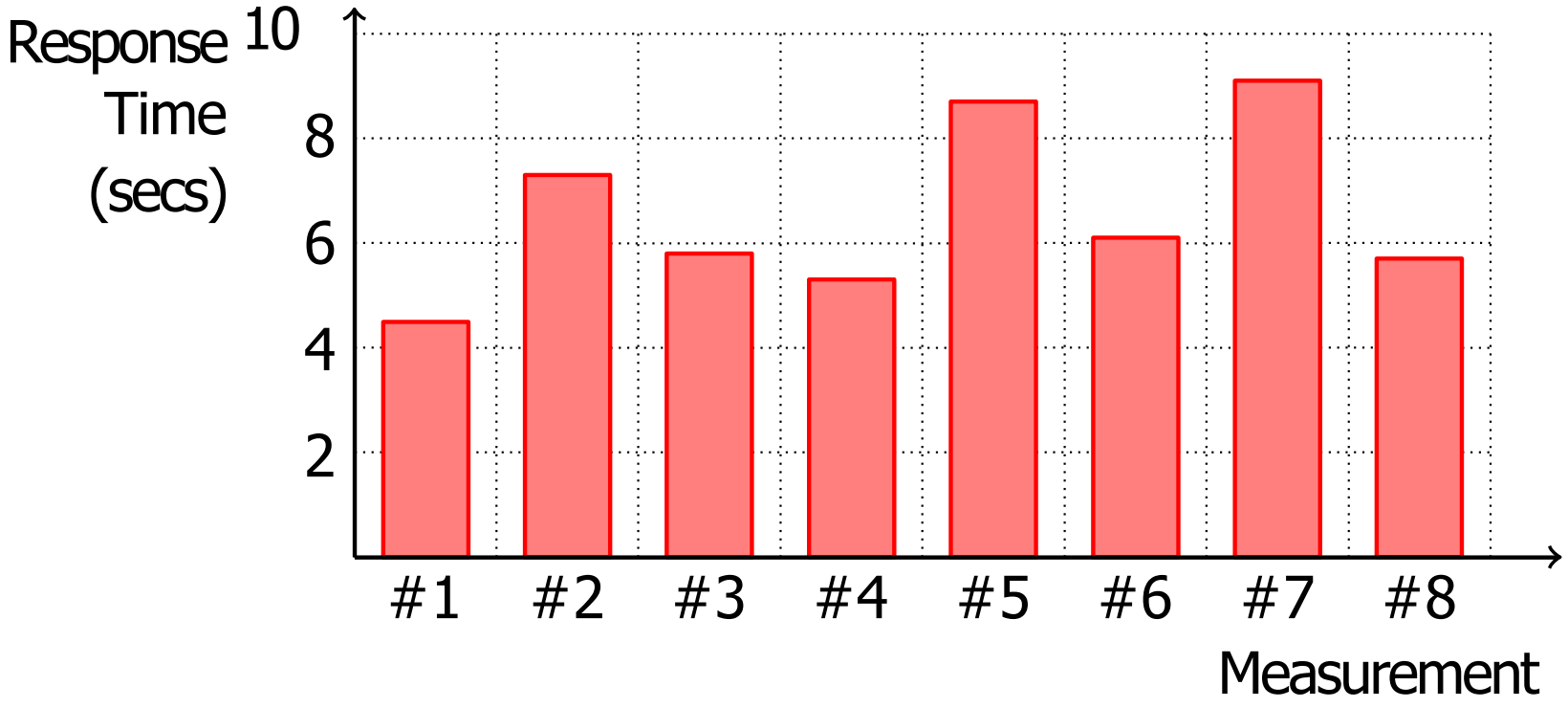
- There are lots of trade-offs in software development.
  - e.g. Through careful algorithm design, you can often speed up software at the cost of using more memory.
    - (See the unit “Data Structures and Algorithms”.)
- But resources (memory, bandwidth, etc.) are limited.
  - Using *some* is vital.
  - Use too much and the software won’t run on the user’s PC!
- If we require software to be fast, then either:
  - The developers could be lazy and assume there’s infinite memory to help make it faster, OR
  - We also require it to limit its memory usage; e.g.

> The music player must use no more than 500 MB of RAM in total, and no more than 0.5 MB for each track in the playlist.

Performance Testing

- For test performance requirements, you must measure the *actual* performance of the software you have developed.
- How long is the *actual* response time? What is the *actual* framerate / processing rate?
  - Run the software and measure it! There are tools to help with this.
- But performance is also “stochastic”.
  - Subject to certain unpredictable variation, or “noise” (random probability).
  - Measure the response time twice, and you’ll get two different times.
- So, make multiple measurements, and take the average.
  - Even better would be to determine the *range* of measurements.
  - If you know your stats, you can find a “confidence interval”.

Performance Testing: Stochastic Measurements



NFR: The system must respond to queries within 6 seconds (on a 2018-model MacBook).

Does the system meet this requirement, based on our measurements?

Reliability Requirements

- Reliability is basically the extent to which the system actually works.
- Reliability requirements may be needed because:
  - You can't have perfect reliability – there are always bugs.
  - You may still need a *specific level* of reliability.
- Different ways to measure this, depending on what the system is supposed to do.
  - Does the system perform a continuous task?
  - Does the system respond to particular requests?

Hardware/Software Reliability

- "Reliability", as a topic of discussion, is more often associated with hardware.
  - Parts of a computer – CPU, hard drives, etc.
  - Any other mechanical devices.
- Hardware failure is caused by physical deterioration.
  - It is inevitable, and the odds increase over time.
  - Predictable and testable.
- Software failure is caused purely by design defects (mistakes).
  - Software doesn't deteriorate; it's not a physical thing.
  - Used in a particular way, software *may never* fail.
  - *But* used a slightly different way, it may fail instantly, due to a defect that's always been there.
- In practice, we should assume software will be used in lots of different ways.
  - This allows us to make claims about its reliability.

Reliability Metrics

We have various ways to measure reliability:

MTTF/MTBF *(Mean Time To/Between Failure)* –
How long do you expect the system to run, on average, before it fails?

AVAIL *(Availability)* –
What percentage of the time is the system working?

POFOD *(Probability of Failure on Demand)* –
What percentage of attempts to use the system fail?

ROCOF *(Rate of Occurrence of Failure)* –
How many failures are expected in a given period?

Reliability Metrics: Mean Time To/Between Failure (MTTF/MTBF)

- Is the system doing something continuously, over weeks, months or years?
- The system must have a MTTF of at least three months.
- An *average* only, not a lower bound.
  - Over a long period (say 10 years), there will be several failures.
  - Some will happen sooner than 3 months, and some later.
  - Only the average must be at least 3 months.
- "Between" (MTBF) is slightly longer than "to" (MTTF).
  - The system will take some time to get working again.
  - MTBF incorporates this "repair" time; MTTF doesn't.

Reliability Metrics: Availability (AVAIL)

- Maybe the system will be running continuously, *but*. . .
  - You care more about cumulative “uptime” and “downtime”.
  - The length of any one failure-free period may be irrelevant if the system can just be restarted immediately.
- If so, you can write a reliability requirement like this:

> The system’s user registration component should be avail- able 99.99% of the time.

- Be careful with the amount of 9’s.
  - Each one makes an *order of magnitude* difference!
  - 99% → 3 days, 16 hours downtime per year (too sloppy?).
  - 99.9% → 8 hours, 46 minutes downtime per year.
  - 99.99% → 53 minutes downtime per year.
  - *99.999% → 5 minutes downtime per year.*
  - 99.9999% → 32 seconds downtime per year (unrealistic?).

Reliability Metrics: Probability of Failure on Demand (POFOD)

- Lots of systems basically just wait for you to use them.
  - We may not care about reliability *until* we try to use them.
  - At other times, they're not really doing anything.
- This is where we might use probability of failure on demand:

> The system's print feature should fail on no more than 0.1% of attempts.

Or equivalently:

> The system's print feature should have a POFOD of 0.1% or less.

Reliability Metrics: Rate of Occurrence of Failure (ROCOF)

- Perhaps you're concerned about the cost incurred for each failure.
  - The more failures, in a given period, the higher the cost.
- In this case, you might express a reliability requirement using ROCOF:

> The system must not make more than 1 incorrect payment in every 1000 hours.

Testing Reliability

- We assume reliability is *stochastic*, like performance.
  - i.e. subject to unpredictable variation.
  - (In fact, this is debatable for software. But for reliability requirements to make sense, we assume it's "true enough".)
- We need multiple measurements, so we can take the average to counteract the noise.
  - You can't just run the system until its first failure.
  - We must run it until we observe (say) 10 failures or more.
- Unfortunately, this may take a *very long time*.
  - Perhaps years! This would kill our project deadline!
- So, we could run the multiple copies of the software simultaneously.
  - If we run 100 copies on different PCs, then we (more or less) expect 100 times the failure rate.
  - i.e. we get our test results much sooner.

Testing Reliability: Measuring Failure

- To measure probability-of-failure-on-demand, access the software enough times to observe (say) 10 failures.
  - POFOD = number of failures (e.g. 10) ÷ number of attempts.
- To measure the others, run the software long enough to observe (say) 10 failures.
  - Record the “uptime” elapsed before each failure.
  - Record the “downtime” elapsed as a consequence of each failure.
  - MTTF = average of the uptimes.
  - MTBF = average of times between failures (each uptime plus the next downtime).
  - AVAIL = total uptime ÷ total overall time.
  - ROCOF = number of failures (e.g. 10) ÷ total overall time.
- Now you can prove that your system meets its reliability requirements! (Or that it doesn’t!)

Usability Requirements

- We obviously want software to be *usable*, or “easy to use”.
- We *really* want software to help us be efficient.
  - We want to locate and use the software’s features with minimal effort (Learnability)
  - We want to see all the stuff we need, and no more.
  - We want to avoid making mistakes.
- There are a couple of ways to create a usability NFR:
  1. State that the software must enable users to have a certain degree of efficiency.
  2. State that the software’s user interface must have certain characteristics.
- Usability is closely going with user interfaces(UI). Therefore, first we need to have an idea of a user interface (UI).

User Interfaces (UIs)

- A UI is the part of the software that interacts with the user(s).
- Most software has a user interface.
  - Any software with usability requirements certainly has a UI.
  - UIs are complex, and this is exactly *why* usability is an issue.
- Applications can have different kinds of UIs; e.g.
  - Text-based UIs or command-line interfaces (CLIs).
    - e.g. vim, javac, ls.
  - Graphical user interfaces (GUIs): desktop, web, mobile.
  - Speech: Siri, Alexa, Cortana, Google Assistant.
  - Camera-based interfaces: Microsoft Kinect (now discontinued)
  - Virtual-reality headsets.
- These bring different ways for users to interact with software.
- The requirements *usually* specify particular kinds of UIs.
  - At least some basic aspects of them, anyway.

Usability: Metrics and Measurements

Approach #1: Require the software to enable users to have a certain degree of efficiency.

- For instance:

> The software must allow customers to pay for their purchases within 30 seconds, on average.

> The software must facilitate customers choosing the correct payment method at least 99% of the time.

To be clear, we're specifying:

- How long it should take users to perform a given task.
- The maxmimum acceptable user error rate.

- These are *not* performance or reliability requirements.
  - We're not requiring the software to be fast & reliable here.
  - We're requiring the software to *help the user* be fast & reliable.

Usability: Measuring user activities

- To measure how fast and reliable the system makes its user. .
- . . . we must measure the activities of the users!
- This is stochastic (like for performance and reliability).
  - Different people have different knowledge, skills, working styles, preferences, etc.
  - So, we expect differences in how usable they find the software.
  - So, we to recruit a whole team of test users.
  - We take our measurements, and then take an average.
- Also, consider that person time is expensive!
  - You need to pay your testers.
    - Or, if they're already employed, you need to justify why you're spending their time.
  - Much more expensive than just running the software.
  - Be careful how you use your testers' time.

Usability: User Interface Characteristics

Approach #2: Say the UI must have certain characteristics.

- We can simply require the software to work in certain ways.
  - We may already know some tricks to assist usability.
  - If so, we may not need to measure user efficiency directly.
- For instance:

> The system must show all payment fields and options on one screen.

> The system must allow the user to query their purchase history with at most 2 mouse clicks.

- We can check whether the finished product meets these requirements *without* having any test users.
- This is *not* stochastic! No need to average multiple measurements to verify these requirements.

Other NFRs

- Performance, reliability and usability – important kinds of NFRs, but not the only ones.
- There are many other qualities that software tends to need:
  - Security and privacy.
  - Localisation: different languages, date formats, etc.
  - Interoperability with specific 3rd-party products, platforms, and file formats.
  - Developed using particular tools, processes, and timelines.
  - Accessibility ….
- Unlike most NFRs seen so far, these tend to be *qualitative* (non-numeric).
  - We don't need quantitative measurements.
  - But they're still *objective* – i.e. clearly testable, not just someone's opinion.
- We'll briefly look at some of the above. . .

Security

- Security is *everyone's* responsibility!
  - Programmers sometimes assume that "bad people" will be stopped before they can access the software.
  - This is a poor assumption, and leads to insecure software.
  - Security is achieved (as best it can be) by assuming the worst.
- Here are a couple of example security NFRs:

> The system must *not* store unencrypted passwords.

  - FYI, if your software does store unencrypted passwords, then you're a bad person, and you can't have nice things.

> The system must use TLS encryption for all network com- munication.

- It's not clear whether you could actually *measure* security.
  - Security incidents are unpredictable, but they're also generally *intentional*, not random/stochastic.

Localisation (“L10N”)

- Your software might need to be used by people around the world.
- Different cultures have different languages, so. . .

> The user interface must be displayable in English and Chinese.

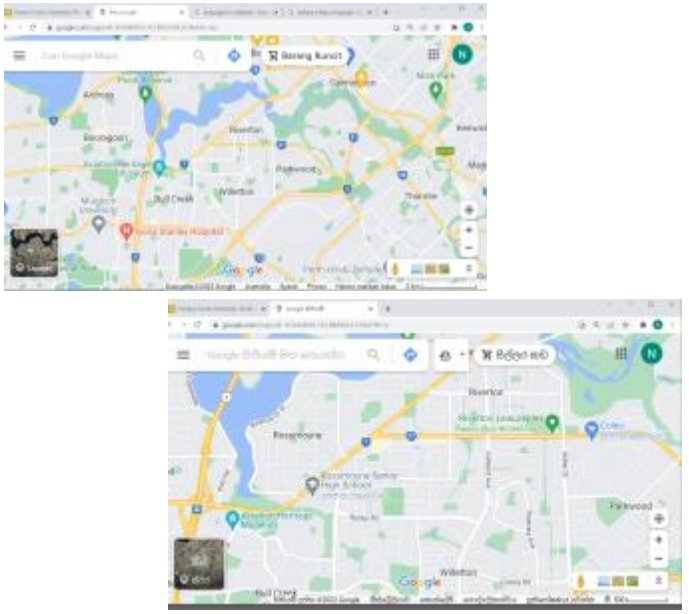
- Being able to change the language would be a *functional* requirement.
- But an NFR would define *which* languages must be available.
- There are also different number formats, currencies, date formats, date *systems*, etc. to consider.
- Additional note: Developing your software to manage localization need careful thoughts in the design stage.(even if it’s not going to be implemented now)

Localisation ("L10N"): Example work with localization

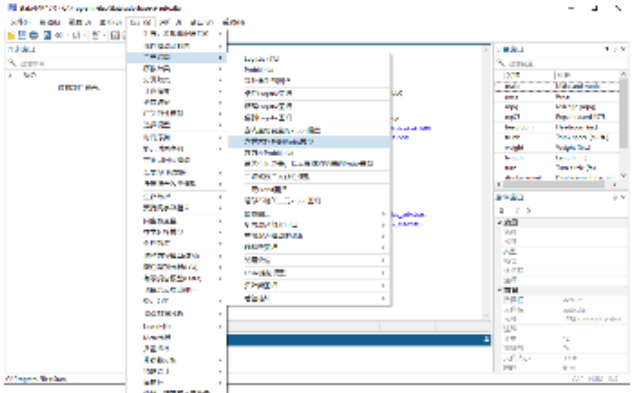
1. Example of good localization (maptiler):

2.Mobile interface with a Maori language

Source: https://www.waikato.ac.nz/__data/assets/pdf_file/0005/394907/chapter2.pdf



3. Map interface with a local languages Malay and Sinhala



Source: https://www.stata.com/features/overview/chinese-interface/

4. Software with Chinese interface

Exercise: Can you write the localization NFRs for some of the above applications?

Interoperability

- Software must typically interact with other software.
  - But we must be careful to say precisely *which* other software; e.g.:

> The system must interoperate with MySQL version 8.0, for the purposes of database storage.

  - FYI, MySQL is one of several database software packages.
- File formats are an indirect kind of interoperability:

> The system must generate all reports in the Open XML (.docx) format readable by Microsoft Word.

- And platforms tell us where the system can run:

> The system must run on Windows 10, macOS 10.12, and RHE Linux 7.

Brief look at Accessibility

- Software should cater for diverse abilities of the users.
- In order to get the required functionally, systems may need to adopt for the users with special needs or different abilities.
- Need to work closely with the end-users to identify the characteristics of the software they need.
  - Example: Vision impaired users will wish to use voice as the user interaction method

The currency reader must scan the amount irrespective of the placement of the note.

- Some people consider accessibility as a functional requirement, if the software cannot be utilized by the intendent audience without accessibility features(e.g., assistive technology tool)
- Some accessibility requirements can be business rules therefore, they not NFRs but FRs. (e.g., web accessibility rules)
- Sydney Olympics Website legal case :https://www.w3.org/WAI/business-case/archive/socog-case-study

What's a Good NFR?

Proper NFRs must have several properties:

Unambiguous. Must be clear what's being asked for.

Doable. Must be possible to implement the requirement.

Verifiable. Must be possible check afterwards that the product meets the requirement.

In-Scope. Must be a requirement *of the software itself*.

- Remember: you're only making the software, not everything and everyone around it.
- Often users must follow "business rules", which can look like NFRs, but out of scope.

Non-Functional. Must not be a functional requirement in disguise.

- Functional requirements shouldn't get mixed up with NFRs.

(Apart from the last, these also apply to functional requirements.)

Unambiguous NFRs

- An NFR must have the same meaning to everyone.
- It's not enough to say "the system must be really fast".
  - *How* fast?
  - Without a number, everyone will disagree!
- Performance and reliability requirements have minimum/maximum numbers.
- Qualitative (non-numerical) NFRs must have enough detail.

The system must store temperature data in a spreadsheet file.

The system must store temperature data in an Excel-compatible CSV (comma-separated-value) file, with columns for the date (in YYYY-MM-DD format) and degrees Celsius.

Doable (Implementable) NFRs

- Requirements are no good if you can't actually make a product that meets them!

The system must process customer payments within 0.000000001 microseconds.

The system must use at most 42 bytes of memory.

The system must be able to be used by a newborn baby

- These are all exaggerated to get the point across.
- In reality, it may not be obvious whether a requirement is doable.
- You may need to do some prototyping to find out!

Verifiable NFRs

- Every requirement must be *verified*, once the product is finished.
  - You must prove that the system meets its requirements.
  - Is the software as fast/usable/reliable as you said it was going to be?
- Requirements must be carefully chosen, in the first place, so they *can* be verified.
- i.e. Everyone must be able to confidently agree that either:
  - "The software definitely DOES meet requirement X," or
  - "The software definitely DOES NOT meet requirement X."
- Failing is bad, but "maybe" is worse!
- But how would we get a "maybe"?
  - If people don't agree on what the requirement means (ambiguous requirements, as before).
  - If it's impractical to obtain enough data to verify it.

Verifiable NFRs: Obtaining Data

- Verifying NFRs involves observing/measuring the software.
  Performance NFRs: measure the *actual* response time, data processing rate, etc.
  Usability NFRs: measure the actions of test users.
  Reliability NFRs: observe and time the system's failures.
- *Un*verifiable NFRs occur if you can't actually make the observations/measurements:

The system must work with all future web browsers.

The system must be rated 8/10 or higher by all users.

The system must prevent itself from being used to break the law

In-Scope NFRs vs Business Rules

- Some “requirements” govern *actors*, not the system itself:

> ✗ Users’ passwords must have at least 12 characters.

> ✗ Students cannot enrol in units after week 1 of semester.

> ✗ Drivers cannot run red lights.

- These are not requirements of the software, and so not NFRs.
- The system may help enforce a business rule, but that would be a *functional* requirement; e.g.:

> As an administrator, I want users to be forced to provide passwords with at least 12 characters, to reduce the risk of a security compromise. **[Functional Requirement!]**

Don't Confuse Functional and Non-Functional

Spot the difference:

The system must notify the uploader when one of their videos is reported for copyright infringement. **(Functional)**

- Something the system must do.
- Normally we'd write a use case.
- Or it may be *part* of a use case (e.g. a step, or an extension).

The system must deliver notifications to the uploader within 1 hour of their being generated. **(Non-functional)**

- A quality/characteristic.
- Cannot sensibly be expressed in use case form.

Summary

- FRs describes what the system should do while NFRs describe attributes or characteristics (not functions) the system must have.
- Performance, reliability, usability, security and privacy, localization, interoperability and accessibility are some important attributes of software.
- NFRs are specified using a definitive language.
- Specifying NFRs specifically helps software team to focus on only what sort of characteristics to be considered as well as to measure and verify them when testing the software.
- When specifying performance requirements, response time, data processing rate and the resource utilization become useful measurements.
- Reliability is measured using various metrices such as MTTD/MTBF, AVAIL, POFOD, ROCOF.
- Usability requirements mostly relate to user interfaces and can be specified as providing certain degree of efficiency or as having certain characteristics.
- Testing of most NFRs need running the software many times and obtaining an average measurement relating to the requirement.
- Good NFRs will have several properties such as unambiguity, doable, verifiable, in-scope, not mixing with functional requirement.

That's all for now!

We will start practical sign-off this week (Project planning and management).

Next week

-

Lecture : Agile Project Management

Practical : NFRs

Sign-off : FRs