Introduction to Software Engineering (ISAD1000)

Lecture 1: Planning

Updated: 15 th February, 2022

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

Outline

Definition of Software Engineering

How do you:

- build something with no physical size, that you can't see or feel?
- know what to build, when even your customer doesn't?
- know when it goes wrong, and how to fix it?
- improve it without breaking it?
- co-ordinate a team of people to do this, on a large scale?
- know how long it will take, and get it done on time?

Software engineering is the art and science of writing software effectively.

Is SE important?

- Software Engineering centres around coding.
- Coding can be done well, and it can be done very, very badly.
- The consequences of poor coding can be disasterous:
  - Lives can be put in danger.
    - In 1985–1987, the THERAC 25 medical radiation machine gave fatal doses to several patients.
    - In 2000–2010, dozens of deaths were recorded due to "sudden unintended acceleration" in Toyota and Lexus vehicles.
    - Lots of very dangerous things are controlled by software!
  - Billions of dollars can be wasted.
    - Stops people getting work done – productivity losses.
    - Drives customers away.
    - Can cause organisations to get sued/prosecuted.
    - Can destroy valuable information.
- SE is there to ensure that coding is done well.

What happens in SE?

- Code is written! Programming is crucial to SE.
- But SE is more than programming.
- We also need to:
  - Co-ordinate a team, and plan a project.
  - Understand user requirements.
  - Test the software, piece by piece.
  - Design the components/modules of the software.
  - Verify that the software meets its goals.
- Whole books – and lots of them – have been written about each of these points.
- However, every SE company is different.
- Every project is different.
- There is no "one-size-fits-all" approach.

Very short history of SE/programming

- The world's first computer programmer was Ada Lovelace (1815–1852).
  - Wrote programs for a computer that was never made – Charles Babbage's Analytical Engine.
- "Software Engineering" was coined by a NASA engineer called Margaret Hamilton.
  - Also popularised by a NATO conference.
  - Attendees reported many problems in writing software: the "software crisis".
- Object Oriented (OO) programming slowly gained popularity; now the dominant programming paradigm.
- "Agile" software engineering is gaining popularity over previous SE practices.
- As software engineering gets older, we are learning what works and what doesn't.

Limitations of this unit

- Software engineering is needed most of all for *very large* programs:
  - Operating Systems (Windows, Linux, Android, Mac OS, iOS, etc.);
  - Web browsers (Chrome, Firefox, Safari, IE, etc.);
  - Office suites (Microsoft Office, OpenOffice, etc.);
  - Certain popular games;
  - Certain popular websites;
  - Nuclear power plant control software.
- Most of you aren't ready to launch into this (yet). We need to start small.

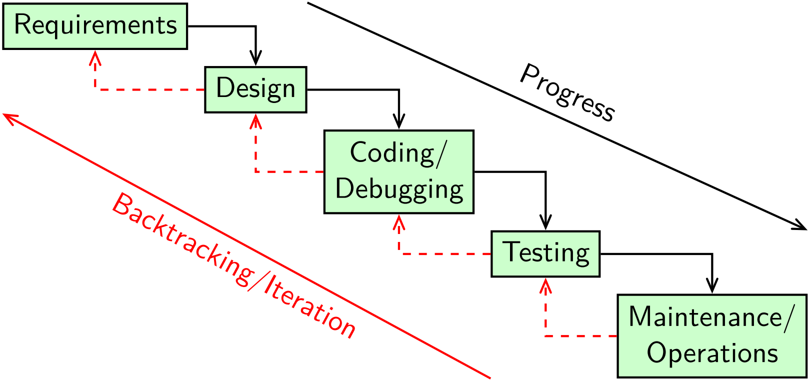
Software Development Life Cycle (SDLC)

- The "Software Development Life Cycle" is a very broad idea.
- It's the idea that software projects are divided into different kinds of activities:
  - Planning your project.
  - Gathering requirements.
  - Designing the system.
  - Implementing the system.
  - Testing and inspecting it.
  - Maintaining (fixing and improving) it.
- Sometimes called "phases". All software projects include them.
- *But* they can fit together in many different ways.

Life Cycle Models

- The SE industry has many different approaches to getting the job done.
- Academics love to analyse and categorise things.
  - We look at the real world and try to create ways of understanding it.
- For SE, academics long ago proposed various “lifecycle models”.
  - Each model is a way of understanding how the software development life cycle (SDLC) works.
  - Each model is also a somewhat idealistic simplification.
- Two of the best known ones:
  - Iterative waterfall.
  - Spiral.

The Iterative Waterfall Model (Graphical Depiction)



The typical depiction of Iterative Waterfall[1].

[1] Adapted from Bell and Thayer (1976), *Software Requirements: Are They Really a Problem?*

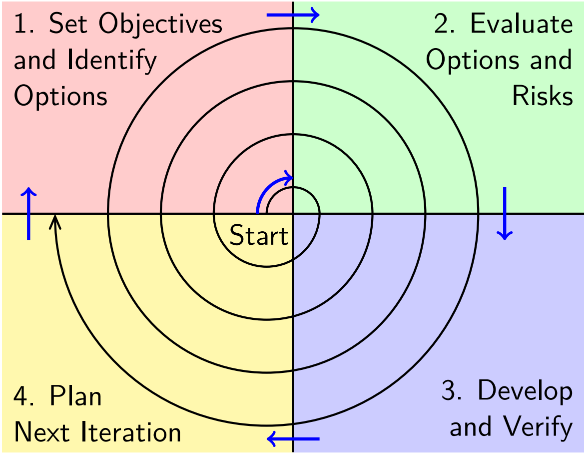
The Iterative Waterfall Model (Discussion)

- The software life cycle phases are done one after another, in order.
- It is expected that some “iteration” will be needed.
  - It will take several attempts to get each phase “right”.
- You will encounter iteration when you write code.
  - It’s also called “looping”.
  - But here it relates to the humans, rather than the code.

Non-Iterative Waterfall?

- "The waterfall model" is often the subject of ridicule.
  - Some people will say it's outdated and unworkable, and an example of what *not* to do.
- They are *sort-of* right, if you squint.
- There are several things to unpack here:
  1. Often people consider the "waterfall model" to be non-iterative (by default).
  2. A non-iterative waterfall is indeed a *terrible* idea – you can't correct your mistakes!
  3. *But*, nobody ever actually *had* the idea in the first place! Some people just got confused.
  4. *BUT*, it is still a useful discussion point on the need for iteration. (Hence this slide!)

The Spiral Model (Graphical Depiction)



Adapted from Boehm (1986), *A Spiral Model of Software Development and Enhancement*.

The Spiral Model (Discussion)

- The Spiral Model makes iteration both *more detailed* and *more flexible*.
  - By contrast, Iterative Waterfall *has* iteration, but doesn't really say how it works.
- A project's timeline is divided into cycles (iterations).
  - Each cycle accomplishes something.
  - *What* it accomplishes is flexible, and not precisely mapped out in advance.
  - Not certain how many cycles you will need.
  - *But basically*, as you get further "outwards", you get further through the life cycle phases: requirements, design, coding, testing and maintenance.

Parts of a Spiral Cycle

- Each cycle consists of four parts:
  1. Determine what the cycle is going to accomplish, and possible options for how.
  2. Determine the *risks* of each option (what could go wrong), and resolve them, possibly using *prototyping*.
  3. Develop the material (whether requirements, design, code or testing) that needs to be done, and verify that it's acceptable.
  4. Review the work done, and ensure that everyone is ready to proceed to the next cycle.
- Throughout the software project, we have requirements, design, coding, testing and maintenance.
- The four parts of the cycle address *all* of the phases.
  - The Spiral Model is a systematic way to develop *anything* in a team.

"Risk Mitigation" (Playing it Safe)

- Use the Spiral Model when your project is risky.
  - Don't worry, you're not going to be in mortal danger. . . [2]
  - SE risks are all to do with money and reputation.
- Every project always has risks:
  - It could go over-budget.
  - It could go over-time.
  - It could fail altogether.
- Some projects have *more* risks than others, mainly when you don't have enough information:
  - About what the client wants.
  - About the best way to do something.
- *Prototyping* is one way to get more information, to help address risks.

[2] Although. . . xkcd.com/292

Prototyping

- A *prototype* is a quick throw-away piece of work.
- It helps you get early insight into the best ways to design your software.
  - The earlier you know something, the fewer mistakes you'll make as a result!
- User interface (UI) prototypes:
  - A poor UI can make software virtually unusable.
  - The look and layout of the software (on the screen) can be a real artform.
  - So, design several mockups of the screen layout, and show them to the client.
  - The client may not like them, but you'll *learn* useful info.

Work Breakdown

- The Waterfall/Spiral models are very abstract, lofty ideas.
  - Not the whole story.
- To actually plan and manage a project, we also need something more detailed and down-to-Earth.
- A Work Breakdown Structure (WBS):
  - You break up a project into a series of tasks.
  - You break up tasks into smaller sub-tasks.
  - Eventually your tasks are "bite-sized" enough that you can easily see how to do each one, and how long it will take.
  - Helps the *team* understand everything that needs to be done.
  - Different tasks get assigned to different team members.

Work Breakdown Structure – Example

1. Market research
2. Prototyping
3. Game design
   - 3.1 Identify core game concepts
   - 3.2 Develop basic storyline
   - 3.3 Develop game world and mechanics
   - 3.4 Specify game characters and abilities
4. Game implementation
   - 4.1 Artwork
   - 4.2 Coding
5. Testing
   - 5.1 Functionality
   - 5.2 Playability
   - 5.3 Rework
6. Release

Work Breakdown Structure – Discussion

- A case of "divide-and-conquer".
  - We're *much* better at solving a series of small problems than one big one.
  - You'll see this idea coming up again and again!
- Not specific to software projects.
  - This is a generic planning tool – you can use it for planning any project.
- Not a schedule.
  - We have more work to do before we get to that.
  - We haven't even assigned an *order* to the tasks yet.
- Not broken down "too far".
  - Nobody wants to deal with a thousand 1-minute tasks.

Dependencies and Milestones

- Some tasks must be completed before others can begin. e.g.
  - We can't test a game until after it's been coded.
  - To code a game, all four design tasks to be complete.
- We need to know the dependencies.
  - Ideally we'd like to do things in parallel – much faster!
  - Dependencies tell us what *can't* be done in parallel.
- *Milestones* are a way of measuring progress.
  - A milestone is "reached" upon completion of a specific activity (or activities), so that others can begin.
  - Milestones are defined in advance, at the start of a project.

Graphs (Computer Science Terminology)

- A *graph* is a network of *nodes* (also called *vertices*) and *edges*.
- A node/vertex is drawn as a point, circle, rectangle, etc.
- An edge is a connection between two nodes.
- Sometimes edges have arrows. These edges are called *arcs*, and appear on *directed* graphs.
- Nodes and edges often have labels: names, numbers, or other information.
- Can be several different types of nodes and edges, depending on the situation.
- Graphs appear in many different situations, for different purposes.

Activity Graphs

- Shows a graphical breakdown of a project.
- Shows which activities depend on which other activities.
- Two representations:

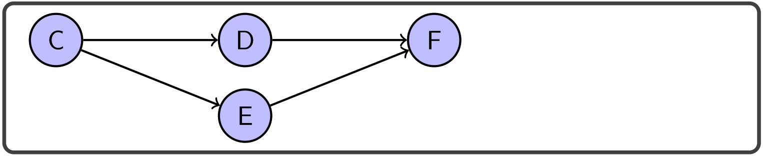
  Activity-on-Node (AON) – nodes are activities, arcs indicate dependencies.

  Activity-on-Arc (AOA) – nodes are *milestones*, arcs are activities.
- These represent the exact same information, but in different ways. (One will probably feel more "natural" to you.)
- In both AON and AOA graphs, "start" and "stop" nodes represent the beginning and end of a project.

Activity-on-Node (AON) Graphs

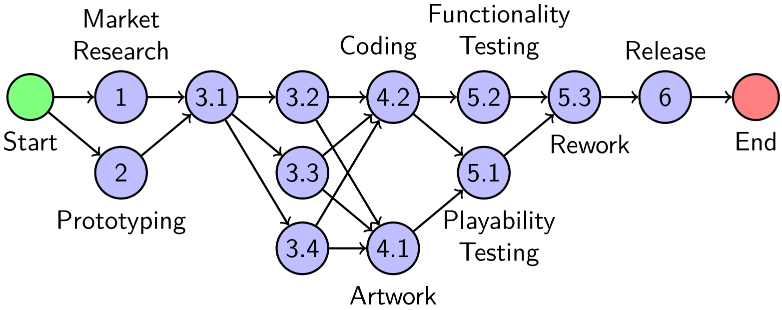
- Say activity B depends on activity A:

- F depends on both D and E, which both in turn depend on C.



- Both I and J depend on both G and H:

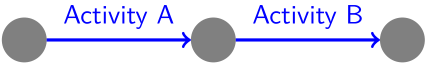
Activity-on-Node (AON) Example



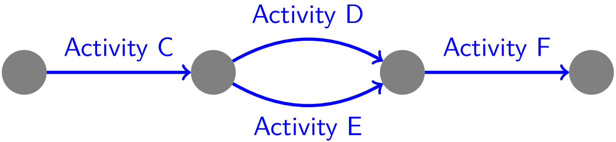
- Nodes (circles) are tasks. Arrows are dependencies.
- Note the special dummy “Start” and ”End” tasks.
  - Not real tasks, but help to show what happens first and last.
- (I haven’t labelled every task here, but you get the idea.)

Activity-on-Arc (AOA) Graphs

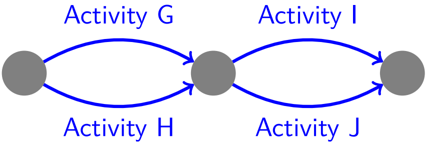
- Activity B depends on activity A:



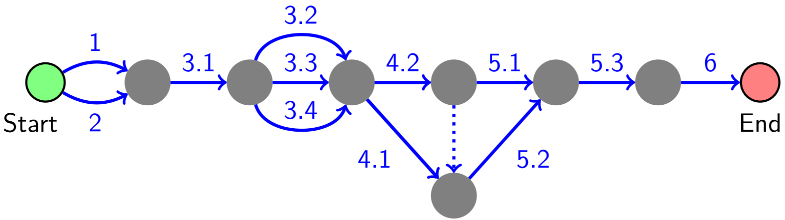
- F depends on both D and E, which both in turn depend on C:



- Both I and J depend on both G and H:

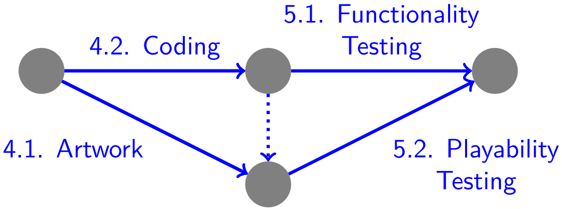


Activity-on-Arc (AOA) Example



- Arrows are tasks. Nodes (circles) are “milestones”.
- Same information as before; different way of thinking.
- The dotted line is a special dummy task.
  - Sometimes needed to show dependencies properly.

Dummy Tasks



- The dummy task (dotted line):
  - Not real, but needed to make the AOA diagram work.
  - Conceptually, when we finish Coding (4.2), the dummy task is instantaneously finished too.
- Playability Testing (5.2) depends on Artwork (4.1) and the dummy task, and the dummy task depends on Coding (4.2).
- Playability Testing *really* just depends on Artwork and Coding.
- (And Functionality Testing (5.1) *only* depends on Coding.)

Time (or Effort) is Money

- In SE, you spend money on two main things:
  - Staff salaries (per person per year).
  - Office space (rent per m^2 per year).

  (There's also hardware, various services, etc., but those costs are usually relatively small.)
- And your office space is based on how many staff you have.
- *So*, the total cost of an SE project mainly just depends on:
  - How many people it takes.
  - How much time it takes.
- We often express this in "effort":

$$\boxed{\text{Effort} = \text{Time} \times \text{Number of People}}$$

  (Effort is measured in "person-hours", "person-months", etc.)

Estimation

- Vitally important to estimate time, effort and (hence) cost.
  - You're spending someone else's money!
  - They want to know how much you need, and when you'll be done.
- Unfortunately, SE estimation is extremely difficult!
  - SE is the ultimate "self-automation" profession.
  - You never really need to do the same thing twice.
    - Because you can just copy-and-paste what you did before.
    - (Or, in practice, use more sophisticated "reuse" techniques.)
  - *Which means:* everything you do is a "first", more or less.
  - So how can you tell how long it will take?

Estimating Time, or Scope?

- Sometimes the question is "How *long* will it take to make X?"
  - Fixed scope, flexible timeline/budget.
- You could also be asked "How *much* can you make in T months?"
  - Fixed timeline/budget, flexible scope.
  - Before you begin a project, you'll have many choices of *what* exactly you should build.
  - You need to decide what's possible, and most important, in the time given.
- "Project scope" basically means "everything the project is supposed to do".
  - "Scope" generally means "everything relevant" to a particular situation.
  - ("Scope" has another meaning in programming languages too, but that's a completely separate thing – see PDI/OOPD/FoP.)

Estimation Models: COCOMO

- COCOMO (the COnstructive COst MOdel) [3] lets you estimate time and effort of a SE project.
- It actually doesn't really work! But *why* is interesting.
- Based on statistical "regression":
  - Boehm collected data on (initially) 61 software projects.
  - Total time, effort and total size (lines of code, or LOC).
  - Other characteristics: team experience and interdepenencies on other systems.
  - Created equations to calculate time & effort from size.
- To *use* COCOMO, you:
  - Estimate the eventual *size* (LOC) of your upcoming project.
  - Categorise: experienced/inexperienced team, separate from or dependent on other systems, etc.
  - Finally, evaluate the equations.

[3] Boehm (1981) Software Engineering Economics.

The Problem with COCOMO

- "Lies, damned lies and statistics":
  - COCOMO will give predictions for time and effort.
  - Wrong predictions. Likely *very* wrong.
- Statistics don't lie *when used properly*.
  - Statistical regression is well used in science (and in data analysis generally).
  - But you need a lot of data, even to predict something simple.
  - You need *huge* amounts of data to predict something complex.
- SE projects are very complex (for estimation purposes).
  - "Time" and "effort" sound simple, but you're really trying to measure aspects of human creativity – *not* simple!
  - Someone had to *try*, and Boehm remains a significant figure in pioneering SE research.
  - But, for COCOMO, he just didn't have enough data.
  - Practically speaking, you *can't* collect enough data.

Real-World Estimation

- In reality, cost/time estimation is done on a task-by-task basis, using a WBS.
- Several experienced engineers independently take an educated guess, for each task, based on experience and intuition.
- Estimates are compared, argued over and adjusted.
- Roughly speaking, this is the *Wideband Delphi* process[4].
- It works because:
  - Small tasks are much less complex than a whole project.
  - Experience and intuition take into account factors that are difficult to model mathematically.

[4] Boehm (1981) Software Engineering Economics.

Planning Poker (yes, this is a real thing!)

- A card game for estimating things (say, activity duration).
- Based on Wideband Delphi.
- Each team member holds a set of cards labelled: 0, $\frac{1}{2}$, 1, 2, 3, 5, 8, 13, 20, 40, 100, "?" and "∞".
  - Partly based on the Fibonnaci sequence.
  - There are variations, but they all look roughly like this.
- Before you start, pick your units:
  - Does a "3" mean 3 weeks, 3 person hours, etc?
  - Are you estimating time, or effort?
  - Are you working in hours, days, weeks or months?
  - The whole team must use the same units consistently!
- Play the game for each task in the Work Breakdown Structure.
  1. All team members play a card, face down, at the same time.
  2. All team members turn over their card, at the same time.
  3. Discuss why your estimates are all different!

Planning Poker – Psychology and Conversation

- Playing cards *face down* is important psychologically.
  - Forces participants to think independently first.
  - If you hear someone else's estimate before making your own, *you won't be able* to make your own.
    - The human brain is very susceptible to suggestion.
    - Don't think about ELEPHANTS. Now, what are you thinking about?
- The conversation afterwards is also important.
  - Different team members will have different ideas about what the task may involve.
  - They must share their understanding, to help the team agree on a single number.
  - If there is still disagreement, you can take an average of the estimates, but don't do that unless you have to.

Planning Poker – The Cards

- Those cards again were 0, $\frac{1}{2}$, 1, 2, 3, 5, 8, 13, 20, 40, 100, "?" and "∞". But why?
- Too small, too big, or too mysterious?
  - "0" means you think a task is trivial (too small).
  - "∞" means you think a task is impossible, or too big to estimate properly.
  - "?" means you have no idea – you need more information.
  - The task should be combined with others, split up, or re-thought as needed.
- Otherwise, choose one of the remaining number cards.
  - Your choices are very limited. You can't just pick *any* number.
  - This actually forces you to be more realistic.
  - Estimation is not very precise, and so arguing about 15 vs 20 is just wasting time.
  - "Round up" to the next available number, where needed.

Planning Poker – Example 1

- Say:
  - You have a team of 3.
  - You're working in person-days.
  - You want to estimate the task "3.2. Specify game characters and abilities".
- Everyone plays a card face down. . . and then turns them over:

2	5	13	Significant difference of opinion!

- So, the team discusses what "specifying game characters and abilities" is really all about.
  - Is the person who said "13" considering things that the others have missed?
  - Or are they making incorrect or pessimistic assumptions?

Planning Poker – Example 2

- Let's move onto "4.2. Coding". Now the team gives us this:

∞ | 100 | 100 — The task is probably too big.

- Two estimates are right at the edge, and one is "impossible".
- A good course of action here is to break up this task; e.g.:

> 4.2. Coding
> 4.2.1 Core game logic
> 4.2.2 Networking
> 4.2.3 Player interface

- Each of these sub-sub-tasks is then estimated separately.
- We have to adjust our AOA or AON graph accordingly.
- This is what "iteration" is about!

Program Evaluation and Review Technique (PERT)

- PERT[5] is a project management technique, based on what we've been discussing:
  1. Identifying activities and milestones;
  2. Determining their dependencies;
  3. Drawing an AOA or AON graph;
  4. Estimating activity times;
  5. Determining the *critical path* (see later);
  6. Updating the graph as needed.
- A "PERT chart" is just an AOA or AON graph that incorporates various timing information:
  - Earliest possible start & finish times for each task;
  - Latest possible start & finish times.

[5] www.netmba.com/operations/project/pert/

PERT Estimation

- Instead of just one estimate for each task, PERT asks you to make *three*:
  - The *optimistic* – the shortest plausible time (so that there is a 1% chance of it being correct);
  - The *most likely* – the time with the highest probability;
  - The *pessimistic* – the longest plausible time.
- This helps deal with uncertainty.
- PERT then asks you to calculate the *expected* time:

$$\boxed{\text{Expected} = \frac{\text{Optimistic} + (4 \times \text{Most Likely}) + \text{Pessimistic}}{6}}$$

- Expected times are used to construct PERT charts.
  - (Don't confuse the *most likely* and *expected* times. They will be similar, not the same.)

Task Table

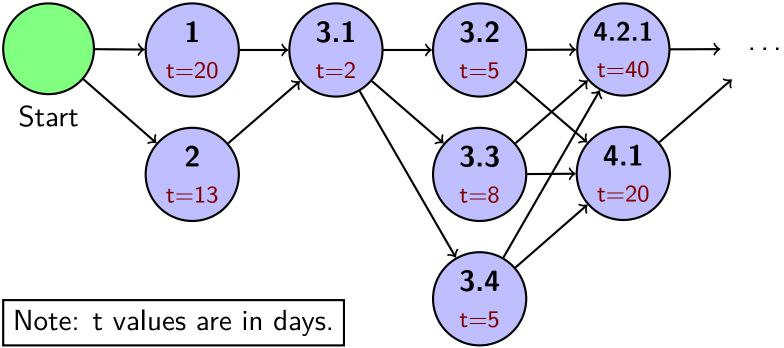
- Many representations for timing and dependency information.
- A tabular format is one of the simplest:

	Activity	Time (days)	Dependencies
1.	Market research	20	–
2.	Prototyping	13	–
3.1.	Identify core game concepts	2	1, 2
3.2.	Develop basic storyline	5	3.1
3.3.	Develop game world and mechanics	8	3.1
3.4.	Specify game characters and abilities	5	3.1
4.1.	Implement artwork	20	3.2, 3.3, 3.4
4.2.1.	Code core game logic	40	3.2, 3.3, 3.4
	etc. . . .		

Graphs with Timing Information

- Tables are easy to construct, but graphs are easier to read.
- We can add timing information to both AOA and AON graphs.

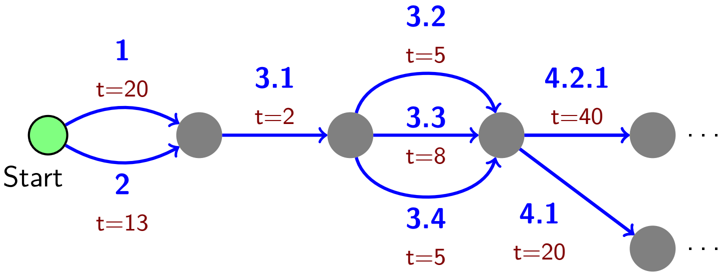
Activity-on-Node (AON) Graph With Timing



Note: t values are in days.

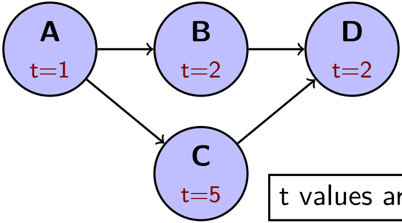
- This is only part of the graph from before, of course.
- We could also include the earliest/latest start & finish times.
  - This makes for rather bulky graphs.
  - But it's nice to have all the information.

Activity-on-Arc (AOA) Graph With Timing



Slack Time (Float Time)

- We like to finish projects in the shortest possible time (without taking shortcuts).
- So, activities are done in parallel, whenever possible.
- But they often take different times to complete:



t values are in days.

  - B and C can be done in parallel, but B can be finished 3 days before C (5 – 2).
  - B could also be delayed for 3 days *without* delaying the project.
  - We say that *B* has 3 days "slack time" or "float time".

Earliest and Latest Times

- Assume the project starts on "day 0".
- We give each task four other numbers:
  - Earliest Start (ES) and Earliest Finish (EF): given the task's dependencies, when can it first conceivably be started and finished?
  - Latest Start (LS) and Latest Finish (LF): when must the task be started/finished, so that the project remains on-time?
- For tasks with no dependencies, ES $= 0$.
  - You can start them right away.
- Other tasks can start once their dependencies are complete.
  - ES for task X $=$ highest EF among task X's dependencies.
- For all tasks, EF $=$ ES $+$ Duration.
- This is enough information to find ES and EF for every task.

Project Duration and Latest Start/Finish Times

- To find the latest start/finish times, we work *backwards* from the end of the project.
- First, we need to know how long the project should take!
  - Find the task with the highest earliest-finish (EF) time.
  - By definition, this is when you expect the project to end.
- This becomes the latest-finish (LF) time for that task, and all tasks "at the end" (that no other task depends on).
- For other tasks, there's a set of tasks that depend on it.
  - Say tasks Y and Z depend on X.
  - LF for task X = lowest latest-start (LS) among tasks Y and Z.
  - If task X were completed *after* this point, it would break the scheduling for either Y or Z.
- For all tasks, LS = LF − Duration.
- We can now find LF and LS for every task.

Slack Time (Again)

- Slack time is the difference between earliest and latest:

$$\boxed{\text{Task Slack} = \text{LS} - \text{ES} = \text{LF} - \text{EF}}$$

- This mainly helps in pinning down the exact definition.
- For most tasks, you can usually determine the slack more easily.
  - The formula looks simple, but getting ES and LS in the first place involves some effort.
  - Instead, you can often just look at the duration of any parallel tasks, and take *that* difference.
  - Unless the graph is particularly complex.

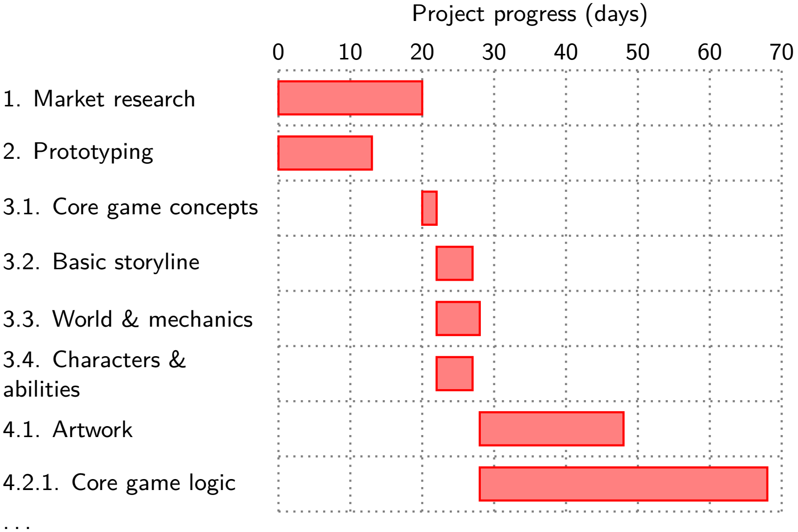
The Critical Path

- There is always a sequence of activities that each have zero slack time.
  - Equivalently, their earliest and latest times are the same.
- If you delay any of them, you delay the whole project.
- This is the *critical path*; each activity on it is a critical activity.
  - "Critical" in terms of time, not the final result.
  - "Non-critical" activites are still essential.
- Monitoring the progress of critical activities will tell you whether the whole project is on track or not.
- Must always span the whole project timeline from start to end.

Gantt Chart

- Shows the actual project plan – when each activity will be done.
- One last planning decision: *when* do you *actually* plan to start each activity?
  - Ideally at the "earliest start time", but not always possible.
  - How many software engineers do you have?
  - You can't do 5 activities at the same time with only 3 people.
  - Some activities may have to be delayed.
- Drawing the Gantt chart itself:
  - Like a weird bar graph. Each activity is a horizontal bar.
    - The length indicates the activity's duration.
    - The horizontal position indicates the planned start time.
  - Time units are written along the top, and activity names down the left.
  - Can be tricky to draw. Use a tool to do it.

Gantt Charts – Basic Example



Gantt Charts – Extra

- Multi-colour Gantt charts are common. Colours could be used to show:
  - Groups of related tasks.
  - *Who* is doing which activity.
- Dependency information can be added, using arrows.
  - Show arrows from the end of one activity...
  - To the start of another.
- You can add milestones too, if desired.

That's all for now!