

# Introduction to Software Engineering (ISAD1000/5004)

## Lecture 4: Agile Project Management

---

Updated: 30<sup>th</sup> March, 2020

Minor revisions on 20<sup>th</sup> March 2022

Discipline of Computing

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

Copyright © 2020, Curtin University

CRICOS Provide Code: 00301J

***ELECTRONIC WARNING NOTICE FOR  
COPYRIGHT STATUTORY LICENCES***

**WARNING**

This material has been reproduced and communicated to you by or on behalf of **Curtin University** in accordance with section 113P of the *Copyright Act 1968* (**the Act**)

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

# Outline

[From Planning to Doing](#)

[Burn-up charts](#)

[Kanban](#)

[Scrum](#)

[Code Reviews](#)

# Project Management

- ▶ In lecture 1, we discussed basic project management concepts and software project planning.
- ▶ Now we must follow that up. How do we keep control of a project while we're *doing* it?
- ▶ How do we know if we're on track?
  - ▶ Because if we don't know, we're heading for disaster!
  - ▶ *Monitoring* is an important aspect of project management
- ▶ Tools for monitoring
  - ▶ Conventionally, Gantt chart is updated while the project is progressing.
  - ▶ **Burn-up charts**/ burn-down charts
- ▶ Project Management applies to software and non-software projects, *but*. . .
  - ▶ There are a few software-specific practices.
- ▶ Project process models help to manage projects.

# Project Processes

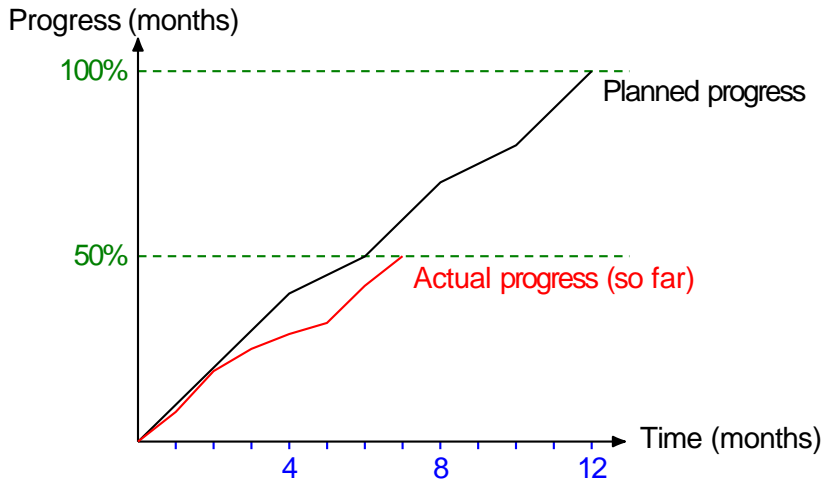
- ▶ “Process” is a loaded word.
  - ▶ In science: a transformation that happens over time.
  - ▶ In engineering: a procedure, or a broad set of instructions, for how to conduct a task.
- ▶ If you work for a software company, they will have a “process” (of the engineering form).
  - ▶ A large-scale process for undertaking the whole project.
  - ▶ Various smaller-scale processes for different parts of it.
- ▶ Processes vs lifecycle models (waterfall, spiral, etc.)?
  - ▶ Lifecycle models are broad approaches to organising software projects.
    - ▶ They let us understand how SE works in general.
  - ▶ Processes are specific and instructive.
    - ▶ They actually tell you what to do!

# Types of Processes

- ▶ There are many well-known processes; e.g.:
  - ▶ **Scrum**, Lean Development, Extreme Programming, the Unified Process, the Team Software Process, etc.
  - ▶ **Kanban** may be considered a process, or at least *part* of a process.
- ▶ Many (most?) organisations will use a hybrid/customised process.
  - ▶ Borrowing ideas from several processes and models.
  - ▶ What happens in practice is never *exactly* what you read about in books.
- ▶ Most organisations will try to be “agile”.
  - ▶ “Agility” means that you can deal well with changing situations.
  - ▶ Scrum is one of the best-known agile processes.
  - ▶ These often look a lot like a case of the spiral model.
  - ▶ Lots of very quick cycles/iterations.

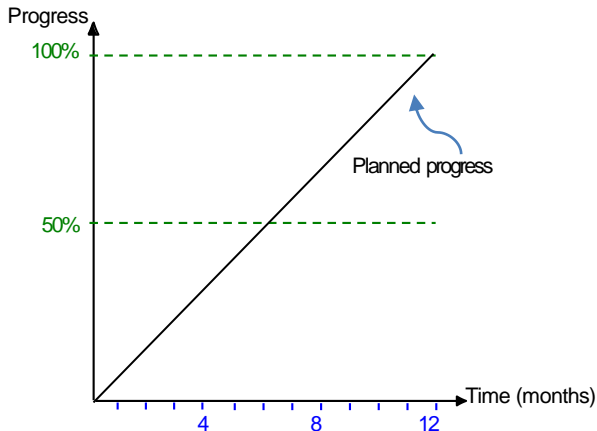
## Burn-Up (Earned Value) Chart

- ▶ “Burn-up charts” show us the big picture.
- ▶ They show “planned” vs “actual” progress



# Burn-Up (Earned Value) Chart

## ► “planned” progress:



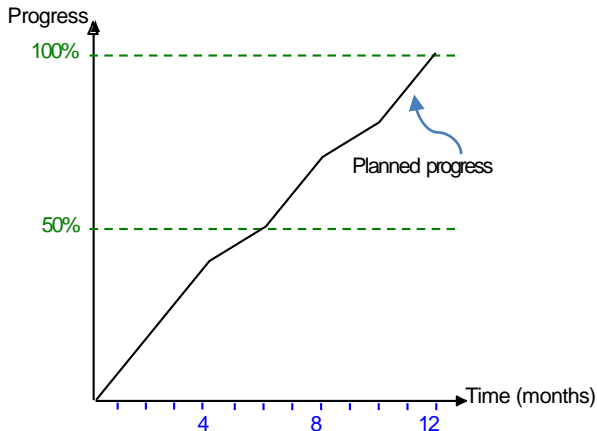
Planned progress:

- How far through the project did you *expect* to be at time  $t$ ?
- Probably a straight line in many cases.
- May have non-straight bits if you plan for certain interruptions to occur, or for people to join/leave your team.



# Burn-Up (Earned Value) Chart

## ► “planned” progress:

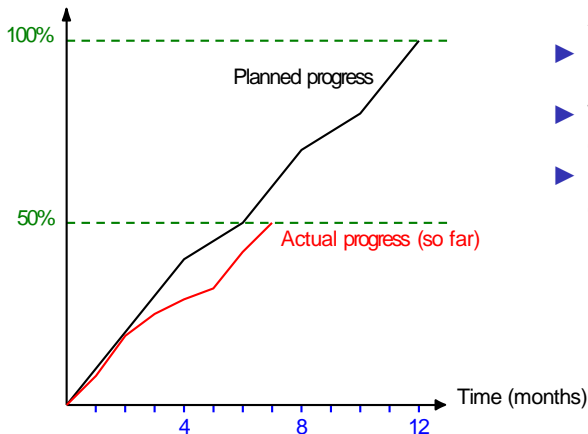


Planned progress:

- How far through the project did you *expect* to be at time  $t$ ?
- Probably a straight line in many cases.
- May have non-straight bits if you plan for certain interruptions to occur, or for people to join/leave your team.

# Burn-Up (Earned Value) Chart

## ► “Actual Progress”



Actual progress:

- How far through are you *really* at time  $t$ ?
- What tasks have you completed?
- How much “value” is each completed task worth?
  - The “value” of a task is the original estimated duration (remember planning poker?).
  - Some tasks are “worth more” than other tasks, because they were expected to take longer.

## Burn-Up Charts: Explanation

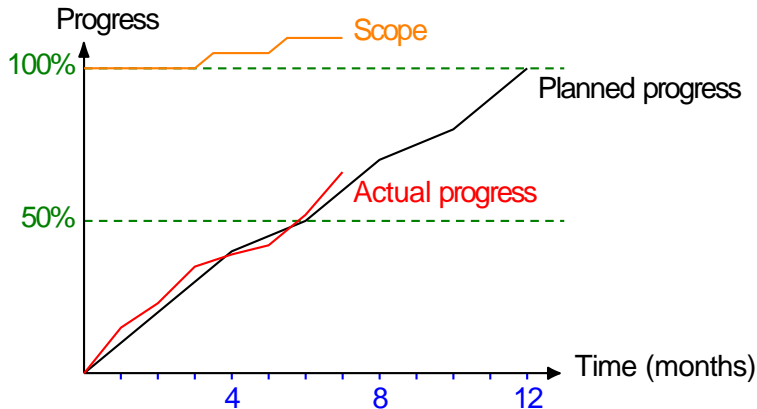
- ▶ Burn-Up charts show “planned” vs “actual” progress.
- ▶ Planned progress:
  - ▶ How far through the project did you *expect* to be at time  $t$ ?
  - ▶ Probably a straight line in many cases.
  - ▶ May have non-straight bits if you plan for certain interruptions to occur, or for people to join/leave your team.
- ▶ Actual progress:
  - ▶ How far through are you *really* at time  $t$ ?
  - ▶ What tasks have you completed?
  - ▶ How much “value” is each completed task worth?
    - ▶ The “value” of a task is the original estimated duration (remember planning poker?).
    - ▶ Some tasks are “worth more” than other tasks, because they were expected to take longer.

## Burn-Up Charts: What's the Situation?

- ▶ Compare the planned and actual progress.
- ▶ Is actual progress higher than planned progress?
  - ▶ Good times! Your project is ahead of schedule.
- ▶ What if actual is lower than planned?
  - ▶ Bad news. You're behind. Time for some tough choices.
  - ▶ Put in more work. Sounds virtuous, but overwork can cause disasters too.
  - ▶ And/or. . . negotiate with the client for more time.
  - ▶ And/or. . . negotiate with the client for less scope; i.e., will they accept a product with less functionality or lower quality?
  - ▶ And/or. . . accept a lower payment.
- ▶ Knowing things are going badly is better than *not knowing until it's too late*.
  - ▶ If you know, you can make the right decision!

## Burn-Up Charts: Scope

- ▶ Burn-up charts often have a “scope” line too.
- ▶ In agile project management, we often get extra work to do mid-way through a project.



# Burn-Up Charts: How to find progress?

Filling the activity times in WBS and find the planned progress per day

1

3

Activity	Estimated Duration (days)	Actual Finish
1. Market research	20	17
2. Prototyping	13	15
3.1. Identify core game conce	2	20
3.2. Develop basic storyline	5	30
3.3. Develop game world and	8	
3.4. Specify game characters a	5	32
4.1. Implement artwork	20	
4.2. Code core game logic	40	
5.1. Testing Functionality	20	
5.2. Testing playability	8	
5.3. Rework	13	
6. Release	5	
Sum of Task Durations	159	Sum of estimated durations
Planned Project Duration	<b>108</b>	Activity graph's project duration
Planned Progress per Day	<b>1.5</b>	Calculation: 159/108

2

1. Using the WBS/ activity lists and the time estimates from planning poker find the estimated duration for each task.

2. Find sum of task duration by adding the duration of all tasks. (a)  
Find planned project duration using the activity graph. (b)  
Find the planned progress per day ( a/b)

3. While the project is progressing, fill the actual finish time.

We are planning to finish a **project worth 159 days** in **108 days**.

# Burn-Up Charts: Planned progress

Table 1 : WBS with activity time

Activity	Estimated Duration (days)	Actual Finish
1. Market research	20	17
2. Prototyping	13	15
3.1. Identify core game concept	2	20
3.2. Develop basic storyline	5	30
3.3. Develop game world and	8	
3.4. Specify game characters	5	32
4.1. Implement artwork	20	
4.2. Code core game logic	40	
5.1. Testing Functionality	20	
5.2. Testing playability	8	
5.3. Rework	13	
6. Release	5	
Sum of Task Durations	159	Sum of estimated durations
Planned Project Duration	108	Activity graph's project duration
Planned Progress per Day	1.5	Calculation: 159/108

1.Fill the actual time(any interval) to monitor the progress. In this example, from time 0 to planned project duration, in 5 day interval

2.Calculate the planned progress for each time interval :  
**time in days X planned progress per day**

Table 2: Activity Progress

Time (actual day)	Planned Progress ( days)	Actual Progress ( days)
0	0.0	
5	7.5	
10	15.0	
15	22.5	
20	30.0	
25	37.5	
30	45.0	
35	52.5	
40	60.0	
45	67.5	
50	75.0	
55	82.5	
60	90.0	
65	97.5	
70	105.0	
75	112.5	
80	120.0	
85	127.5	
90	135.0	
95	142.5	
100	150.0	
105	157.5	
110	165.0	

$$= 20 \times 1.5$$

$$= 55 \times 1.5$$

# Burn-Up Charts: Actual progress

Table 1 : WBS with activity time

Activity	Estimated Duration (days)	Actual Finish
1. Market research	20	17
2. Prototyping	13	15
3.1. Identify core game concept	2	20
3.2. Develop basic storyline	5	30
3.3. Develop game world and	8	
3.4. Specify game characters and	5	32
4.1. Implement artwork	20	
4.2. Code core game logic	40	
5.1. Testing Functionality	20	
5.2. Testing playability	8	

Calculate the actual progress for each time interval :

- Step 1: Identify the time at which the actual progress needs to be calculated. (e.g., 20)
- Step 2: Look at the actual finish time column in the WBS and identify the activities which have finished before the selected time in step 1.
- Step 3: Find the **estimated duration** of the activities identified in the step 2.
- Step 4: **Sum up the estimated duration** of the activities identified in the step 3 and fill the actual progress corresponding to the selected time. (e.g.,  $20+13=33$ )

Table 2: Activity Progress

Time (actual day)	Planned Progress ( days)	Actual Progress ( days)
0	0.0	0
5	7.5	0
10	15.0	0
15	22.5	0
20	30.0	33
25	37.5	35
30	45.0	35
35	52.5	45
40	60.0	45
45	67.5	
50	75.0	

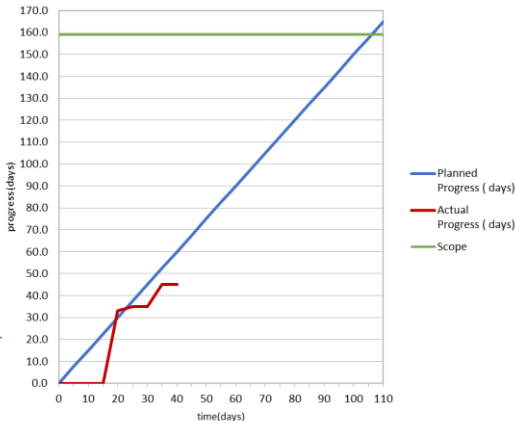


# Burn-Up Charts:

Time (actual day)	Planned Progress (days)	Actual Progress (days)
0	0.0	0
5	7.5	0
10	15.0	0
15	22.5	0
20	30.0	33
25	37.5	35
30	45.0	35
35	52.5	45
40	60.0	45
45	67.5	
50	75.0	
55	82.5	
60	90.0	
65	97.5	
70	105.0	
75	112.5	
80	120.0	
85	127.5	
90	135.0	
95	142.5	
100	150.0	
105	157.5	
110	165.0	



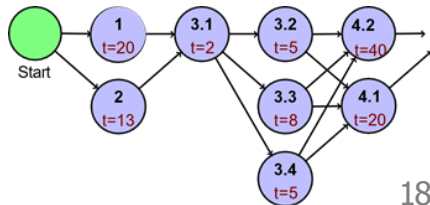
Burn-Up / Earned-Value Chart



- ▶ We can use a spreadsheet to calculate the actual progress and draw the burn-up chart.
- ▶ Useful function for calculate actual progress:
  - ▶ `SUMIF(range, criteria, [sum_range])`
  - ▶ Example : `SUMIF($C$2:$C$13,"<" & E2, $B$2:$B$13)`
- ▶ You will create a burn-up chart in Practical 4

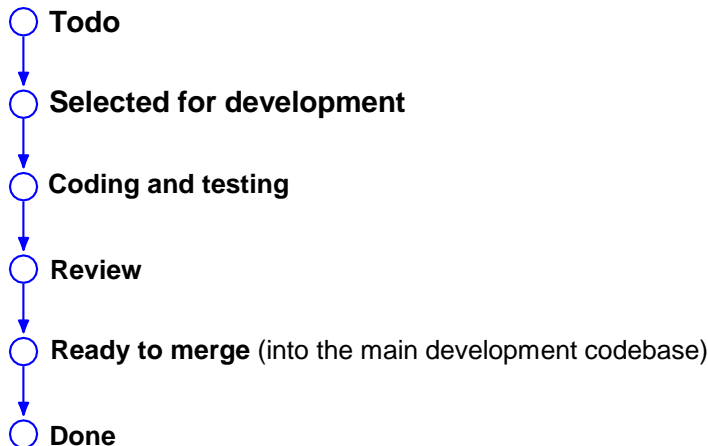
## Completing Tasks

- ▶ How do you actually know if you've "completed" a task?
- ▶ Sounds obvious, but it's not!
  - ▶ Tasks are all connected. They build on each other.
  - ▶ You can't just say you've finished one. If you haven't finished it *properly*, it will wreck other tasks that follow on.
- ▶ Part of SE project management is about the "definition of done".
  - ▶ How does your team *verify* a task is done?
  - ▶ Different organisations might do this differently.
  - ▶ But whatever the criteria, apply it consistently.



# Workflows

Coding tasks typically go through several steps, like this:



# Kanban

- ▶ Kanban boards are another way to visualise your project.
  - ▶ Burn-up charts show the big picture.
  - ▶ Kanban shows the status of individual tasks, related to the workflow.
- ▶ Kanban boards contain columns.
  - ▶ One column for each workflow step, arranged left-to-right; e.g. “Todo”, “Selected”, “Coding/testing”, “Review”, “Ready to merge” and “Done”.
- ▶ Columns contain “cards”.
  - ▶ Each card represents a task.
- ▶ Each card (task) is moved from left-to-right (from column to column) as it progresses through its steps.

# Example Kanban Board

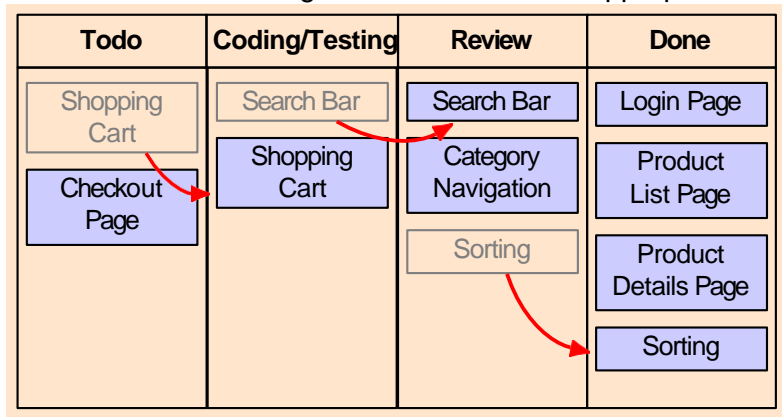
All cards start on the left (“todo”), and move rightwards.

Todo	Coding/Testing	Review	Done
<div>Shopping Cart</div> <div>Checkout Page</div>	<div>Search Bar</div>	<div>Category Navigation</div> <div>Sorting</div>	<div>Login Page</div> <div>Product List Page</div> <div>Product Details Page</div>

(We're only showing four columns for simplicity.)

## Moving Kanban Cards

You move each card right one column at the appropriate time.



- (Don't draw the red lines – those are just to illustrate how the chart gets changed.)

# Work in Progress (WIP)

- ▶ “Work in progress” is everything between “todo” and “done”.
- ▶ We want to limit WIP – only work on a few tasks at a time.
- ▶ Some important principles<sup>1</sup>:

## 1. A “culture of done”.

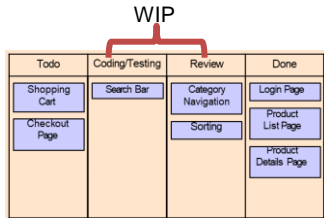
- ▶ Tasks are either finished completely, or still WIP.
- There’s no such thing as “almost” done.

## 2. Using our brains efficiently.

- ▶ “Multitasking” is inefficient. Your brain needs time to switch from doing one thing to doing another.

## 3. Identify (and solve) problems ASAP.

- ▶ Some tasks will turn out to be much harder than we thought.
- ▶ Some parts of our *process* may not work well.
- ▶ Easier to see these problems if you have only a few tasks.



<sup>1</sup><https://www.atlassian.com/agile/kanban/wip-limits>

## Work In Progress: Enforcing Limits

- ▶ We might put a limit of (say) 3 tasks in the coding/testing, and 3 in review.
- ▶ So, if there are already 3 tasks under review. . .
  - ▶ We *can't* move another task from coding/testing to review.
  - ▶ One of the existing reviews has to complete first.
- ▶ This may cause a “pile-up” in an earlier column.
  - ▶ But, in a sense, this is actually a good thing, because it lets us see an underlying problem.
  - ▶ We can identify parts of our process that are “bottlenecks” (slow points), and re-assign team members to deal with them.
  - ▶ The review stage is holding everything up? Then we need more reviewers and fewer coders.



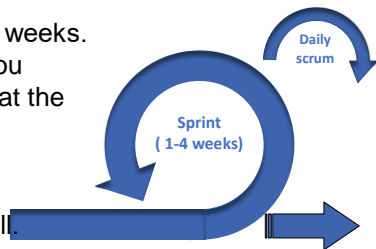
## “Bugfixes” as Tasks

- ▶ You can add more cards to the “todo” column, mid-project.
- ▶ The client may (all of a sudden) want more functionality.
- ▶ Or you might discover problems (bugs) that need to be

Todo	Coding/Testing	Review	Done
Shopping Cart	Search Bar	Bugfix: page layout issue	Login Page
Bugfix: login not working	Bugfix: incorrect pricing	Category Navigation	Product List Page
Checkout Page			Product Details Page
			Sorting

# Scrum

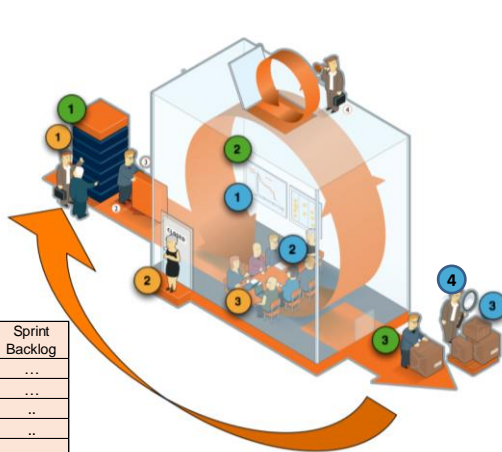
- ▶ Scrum is one of the most popular SE processes.
  - ▶ Think of the spiral model, but with more details nailed down.
  - ▶ We focus on Scrum in ISE because we can't cover everything!
- ▶ Scrum revolves around “sprints” (in other contexts called “iterations”).
  - ▶ Each sprint is a mini-project
  - ▶ Each has a fixed length; say 4 weeks.
  - ▶ Sprints *do not* go over time. You simply deliver what you have at the end.
  - ▶ Each sprint creates something concrete/useful for the customer, even if it's only small.



<sup>1</sup><https://www.scrumguides.org/scrum-guide.html>

<sup>1</sup><https://www.atlassian.com/agile/scrum>

# Scrum process



## Roles

- 1 Product Owner
- 2 Scrum Master
- 3 Team



## Artefacts

- 1 Product Backlog
- 2 Sprint Backlog  
Burndown chart  
Impediment list
- 3 Product Increment

## Ceremonies

- 1 Sprint Planning
- 2 Daily Scrum
- 3 Sprint Review
- 4 Sprint Retrospective

Original Image Source: <https://documentation.cochrane.org/wwirpt/scrum-process>,  
Copyright <http://cochrane.org>,

## The Two Backlogs

- ▶ A backlog is a list of user stories not yet implemented.
  - ▶ Remember user stories? Each user story represents a piece of software functionality for a particular kind of user.
- ▶ In Scrum, you keep track of two backlogs.
- ▶ The “**Product Backlog**” – a todo list for the overall product.
  - ▶ Everything the customer wants the software to do that has not yet been implemented.
- ▶ The “**Sprint Backlog**” – a todo list for the current sprint.
  - ▶ Select a *few* user stores from the product backlog, at the start of the sprint.
  - ▶ Each user story takes time to implement, and you only have 4 weeks.
  - ▶ Go for the “highest priority” user stories.
  - ▶ You may not necessarily finish them all in this sprint.
  - ▶ Each user story *may* need to be broken down into sub-tasks!

## Scrum and Kanban

- ▶ Scrum and Kanban are not the same thing.
  - ▶ e.g. Kanban has no fixed-length sprints, and instead delivers each user story by itself incrementally.
- ▶ *But* they can be joined into a single process – “Scrumban”.
- ▶ The sprint backlog could be the left-most column on your Kanban board.
- ▶ What ends up in the “done” column is what you deliver, at the end of the sprint.

## Sprint Meetings (“Ceremonies”)

Each sprint has several key meetings between the people involved:

- ▶ **Sprint Planning** is a day-long meeting at the start of a sprint.
  - ▶ **The developers talk to the customer and figure out what to do for the next 4 weeks.**
- ▶ **Sprint Review** is a half a day meeting at the end of a sprint.
  - ▶ The developers show the customer what has been done.
- ▶ **Sprint Retrospective** is another one day meeting right after the sprint review.
  - ▶ The developers discuss how well things went, and what might be improved.
  - ▶ This is about the team itself, not the product.
- ▶ **Daily Scrum (or “stand-up”)** is a 15-minute meeting at the start of each day.
  - ▶ The developers discuss their planned work for the day.

# Scrum Roles

- ▶ Scrum assigns special roles to people.
- ▶ The **Product Owner** helps the team make the right product.
  - ▶ They make sure user stories are correct, and decide which have the highest priority.
  - ▶ (The PO is not the customer though.)
- ▶ The **Scrum Master** helps the team follow Scrum.
  - ▶ They must be on top of all the Scrum practices.
  - ▶ They must understand what should happen when, and make sure everyone else is aware of this.
- ▶ The **Development Team** writes the software.
  - ▶ 3–9 software developers.
  - ▶ They take user stories and implement them.

## Other Processes

- ▶ There are other approaches to this too.
- ▶ Other processes require different working arrangements.
  - ▶ In “Extreme Programming”, all development work is done in pairs – **pair programming**.
- ▶ Often there are other roles; e.g.:
  - ▶ A “team lead” – a senior developer with with overall responsibility.
  - ▶ A “configuration manager” may have responsibility for the tools and libraries used by the team.
    - ▶ (Not an IT support issue. This is part of the software development process.)
  - ▶ Security specialists.
  - ▶ Artists (e.g., for games).
  - ▶ Particular “domain” specialists who understand the requirements more clearly than the software engineers.



# Code Review

- ▶ A collection of techniques for achieving two ends:
  1. for one person (or the rest of the team) to understand another's work, and
  2. to find faults in that work.
- ▶ Different techniques:
  - ▶ Inspection – formal group-based fault-detection process,
  - ▶ Walkthrough – informal meeting to understand the code,
  - ▶ Review – fault-detection process with varying formality,
  - ▶ Pair Programming – programmers write all code in pairs, with one reviewing the other's work in real-time. (And roles swapping as needed.)
- ▶ In all cases, you requires someone *other* than the author of the work.
  - ▶ . . . *but* someone who is familiar with the project.
  - ▶ A peer.

## Pull Requests and Code Reviews

- ▶ Typically, a developer makes a “pull request” when they think they’ve finished a task.
  - ▶ Your code will be discussed, and your request accepted or rejected.
- ▶ Pull requests typically trigger a “code review”.
  - ▶ Another team member will look through your code in detail.
  - ▶ They’ll check:
    - ▶ Whether your changes do what is needed.
    - ▶ Whether they break other things.
    - ▶ Whether they’re otherwise up-to-standard.
  - ▶ They often find defects, and will ask you to fix them.

## Code Review vs. Testing

- ▶ Testing and peer review are *both* essential – they are not alternatives.
- ▶ Very different fault-detection processes.
- ▶ Testing:
  - ▶ Automated (once the test cases are written).
  - ▶ Only applicable to *executable* things (i.e. code).
  - ▶ Finds failures – extra effort required to isolate faults.
- ▶ Peer Review:
  - ▶ Manual – human effort required all the way through.
  - ▶ Applicable to anything – code, documentation, use cases, design diagrams, etc.
  - ▶ Finds faults directly, not failures.
- ▶ Testing and peer review find *different kinds* of faults.

## Communication via Peer Review

- ▶ Peer review is a two-way communication.
- ▶ The reviewer gains an understanding of your work and your ideas.
- ▶ The reviewer offers advice, on:
  - ▶ Faults found.
  - ▶ Efficiency issues – could your code be faster, or use less memory?
  - ▶ Maintenance issues – could your code be simpler?
  - ▶ Style issues – could your code be more readable?
  - ▶ Test case issues – do your test cases cover enough of the system's behaviour?

## Fault Reporting

- ▶ The reviewer *does not* fix faults, but just reports them.
- ▶ Every fault found must be carefully recorded.
- ▶ There are fault-reporting forms (paperwork) for this purpose.
- ▶ Each fault is recorded with a variety of information:
  - ▶ The reviewer.
  - ▶ The type of fault.
    - ▶ Faults can be categorised in various ways:
    - ▶ By criticality; e.g. minor, major, critical.
    - ▶ By checklist question (see later).
  - ▶ The location of the fault.
    - ▶ Which source code file? Which method? Which line number?
    - ▶ Some faults concern *relationships* between multiple parts of a system.
  - ▶ A description of the fault.
    - ▶ Might include its effects, or how to fix it, if these are simple to describe.

# Reading Techniques

- ▶ There are a few techniques for reviewers or inspectors to find faults.
- ▶ They try to support or manipulate the way that you take in the information.
- ▶ Checklists are a common technique:
  - ▶ Reviewers work through a list of potential faults.
  - ▶ Check off each one as they go.
  - ▶ Mainly useful for relatively inexperienced reviewers.
  - ▶ For experienced reviewers, checklists just get in the way.

## Guidelines for Checklist Questions

- ▶ Each checklist item should be a yes/no question:
  - ▶ A “yes” answer means everything is okay.
  - ▶ A “no” answer means there is a problem.
  - ▶ Phrase each question so that “yes” and “no” have these meanings. This helps avoids confusion.
- ▶ Number the questions sequentially (1, 2, 3, . . . ), so reviewers can refer to them easily.
- ▶ Each question should represent a potential *type* of fault.
  - ▶ Reviewers may find many such faults, or none at all.
  - ▶ Questions should be generic. They *should not* refer to a single, specific fault in a single part of the system.
- ▶ *Don't* list syntax errors.
  - ▶ These will be found anyway by compiling / running the code.
  - ▶ Don't waste human effort checking things that can be checked automatically.

## Checklists – How Long?

- ▶ Checklists should be no more than one page in length.
  - ▶ Otherwise, the process is too cumbersome.
- ▶ However, there are an infinite variety of possible faults.
- ▶ Obviously, you cannot list them all.
- ▶ Make a judgement call:
  - ▶ Which types of faults are more likely to occur?
  - ▶ Which types of faults would cause more damage if they occurred?
- ▶ Put the most important and relevant fault types on the checklist.
- ▶ Any *previously-reported* fault types are a good candidate for the checklist.
  - ▶ Faults can often re-occur, after being fixed.



## Checklist Questions – Examples

1. Is each “/” operator working with the correct datatype?

 **Java-specific**

**Yes?** Then everything is good.

**No?** Then we may need to convert the datatypes from integers to reals (or the other way around).


2. Are the correct datatypes passed to functions/methods?

 **Python-specific**

**Yes?** Then everything is good.

**No?** Then we need to re-consider what the function/method's purpose is, and supply it the kind of data it needs.

## Checklist Questions – More Examples

3. Do all string comparisons use the “equals” method (and not the “==” operator)? **[ Java-specific]**

- ▶ If “no”, then we’re really comparing object references, not string values (see PDI/OOPD).

4. Do calls to a function/method supply parameters in the *right order*?

5. For functions/methods that return a value, is the returned value being used (and not just discarded)?

6. Where real numbers are compared for equality, is a “tolerance” value used (to deal with rounding errors)?

## Formal Software Inspection

- ▶ Invented by Michael Fagan (1976)<sup>1</sup>.
- ▶ Variations were developed with time.
- ▶ Requires a team of about 4 people, headed by a moderator.
- ▶ Several rigidly-defined steps.
- ▶ Often a criterion for transitioning from one *phase* to another.
  - ▶ Large-scale projects are often (and traditionally) separated into separate phases, starting with requirements, then design, then implementation.
  - ▶ Move from design to implementation once the design passes inspection (and not before).
  - ▶ This doesn't really happen with agile methods.

---

<sup>1</sup>[https://researcher.watson.ibm.com/researcher/view\\_page.php?id=6981](https://researcher.watson.ibm.com/researcher/view_page.php?id=6981)

# Inspection Steps

**Planning:** the moderator schedules a time/venue, and ensures the code/design is ready.

**Overview:** the team shares existing insights into the system.

**Preparation:** team members *individually* read the code/design.

**Inspection:** the team *collectively* searches for defects.

**Rework:** those responsible fix the defects.

**Follow-up:** the team verifies the fixes.

**Re-inspect:** if more than 5% of the code/design has been changed.

## Summary

- ▶ Monitoring the project progress is important to keep the project on track.
- ▶ Burn-up charts are used to show how the planned progress and actual progress vary with time.
- ▶ Software processes provide specific and instructive guidance to actually conduct a project. Scrum, Kanban, extreme programming are some process models.
- ▶ Kanban shows the status of individual activities with time based on the workflow of the software project.
- ▶ Scrum is a widely used agile process. Scrum is based on “sprints”, small mini-projects which are to be completed in few weeks time. Backlogs and different types of meetings happening frequently are key features of scrum process.
- ▶ Once the software code is developed and tested, they are reviewed by another person to identify the defects which would be difficult to see by the developer.
- ▶ Checklists are a common techniques useful in code reviews.

That's all for now!

We will have Functional Requirements  
worksheet sign-off this week

Next week

-

Lecture : Version control  
Practical : Agile project Management  
Sign-off : NFRs