

# **Requirements capture**

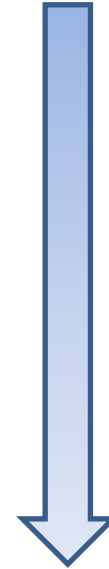
Computing & Information Sciences

W. H. Bell

# Software development lifecycle

## One iteration

- **Requirements definition.**
- Software and systems design.
- Implementation and unit testing.
- Integration and system testing.
- Operation and maintenance.



# Requirements

- Describe what users need from software.
  - Remove ambiguity concerning what should be built.
- Risk of conflict between developers and customers.
  - Requirements are used as part of a legal contract.
  - Minimise risk of conflict.

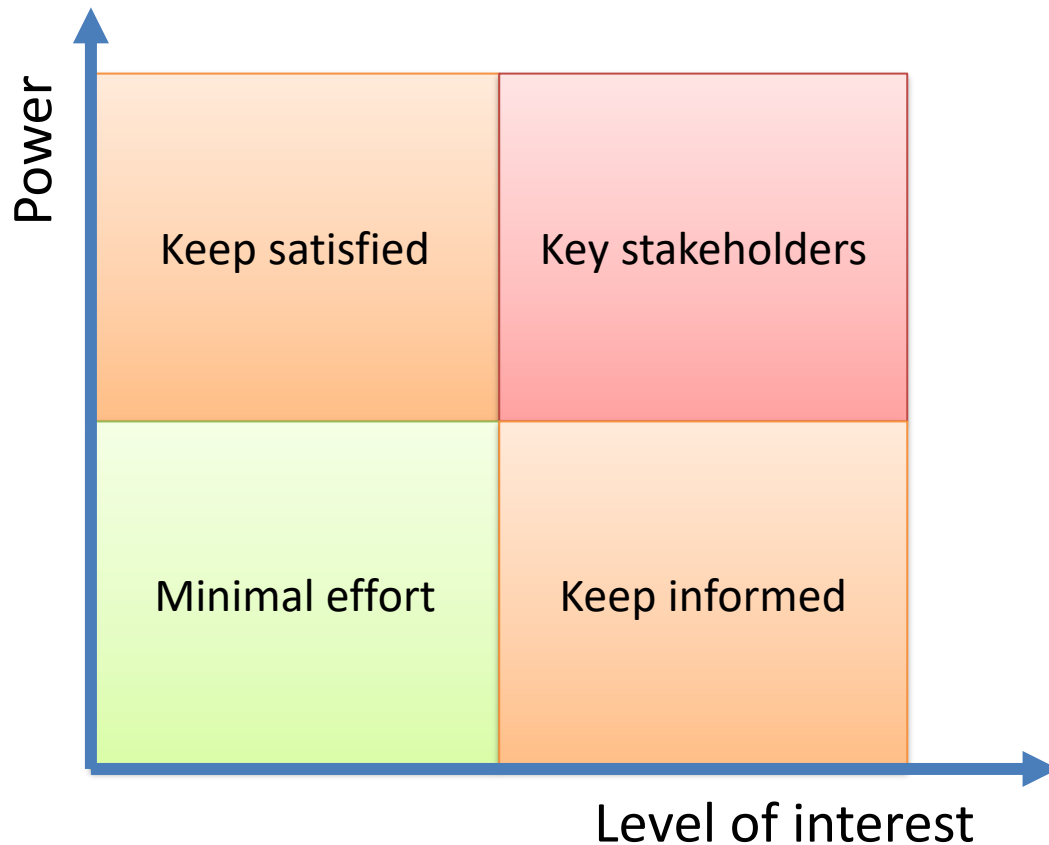
# Requirements capture

- Identify stakeholders.
- Requirements analysis.
  - Read technical documents.
  - Discussion with stakeholders.
  - Present requirements to stakeholders.
  - Update requirements.
- Allow updates.
  - During user interface design discussions.
  - During software implementation – Agile lifecycle.

# Identifying stakeholders

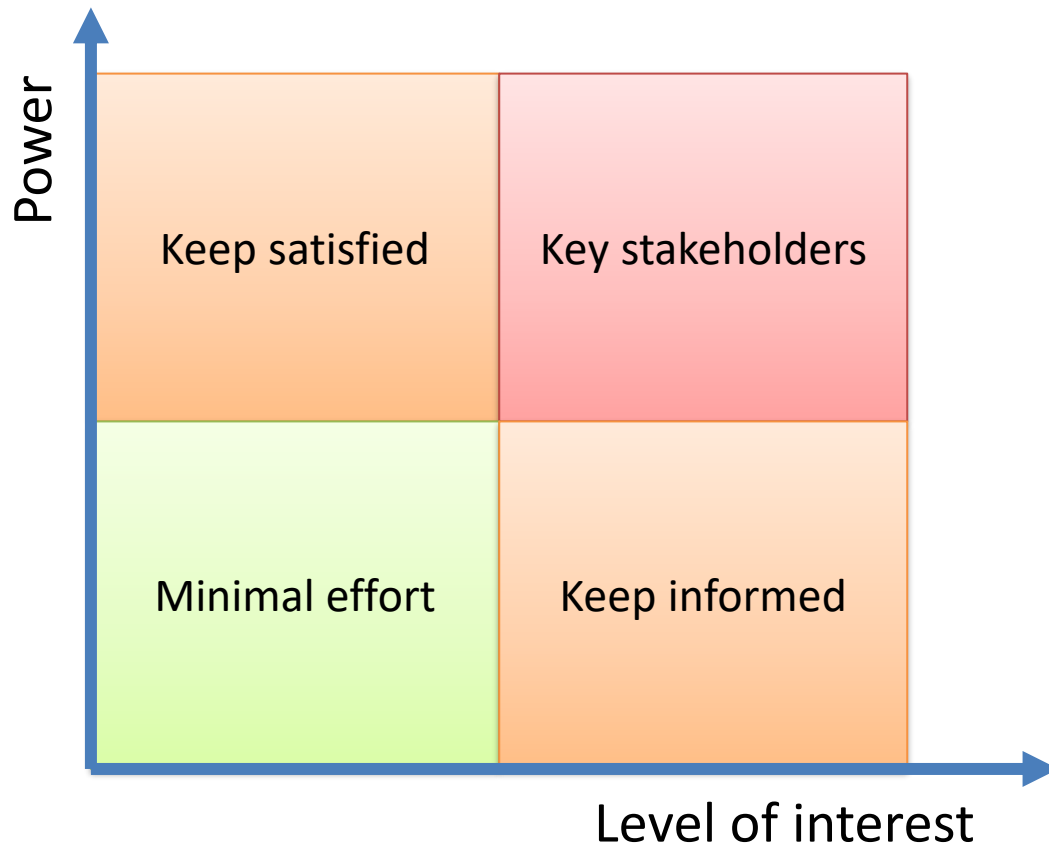
- Who are the users?
- Who are paying the bills?
  - The users.
  - Another company.
- Who is important within the project structure?
  - Project manager.
  - Team leader.
  - IT manager.
  - CEO.

# Stakeholder interest & power



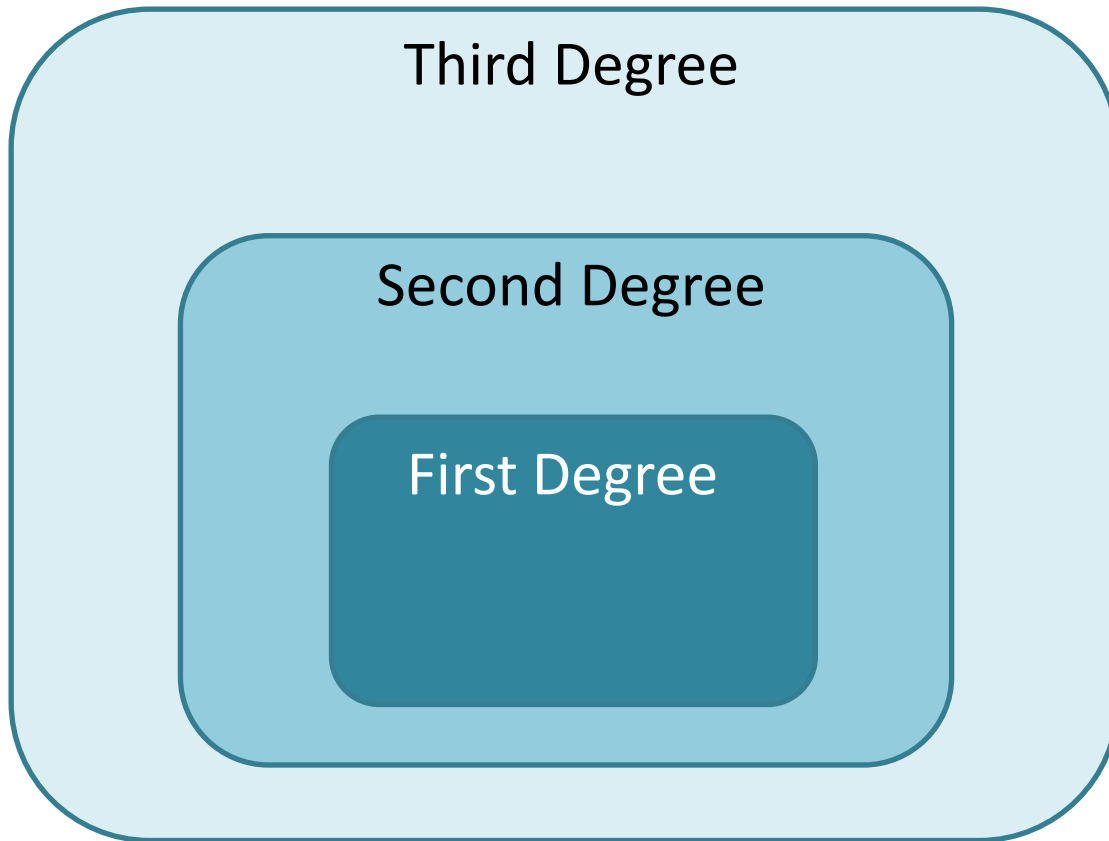
- Users have high interest.
  - Users may have less power.
  - Need to represent users.
  - Convince powerful stakeholders.

# Stakeholder interest & power



- Users may be outside.
  - Not within company organisation.
  - Represent users in discussion.
  - Survey or prototype testing.
- Powerful stakeholders can cause failures.
  - Avoid projects when they will not listen.

# Stakeholder needs



- 3<sup>rd</sup> – Install, deploy or support.
  - IT management.
- 2<sup>nd</sup> – Work with results.
  - Management, analysts.
- 1<sup>st</sup> – Direct users.
  - Operator, customer.
- What do they expect from software?
- What do they need to do?



# User requirements

- Need to understand what the stakeholders need.
- Formulate statements:  
“The <user type> requires the ability to <do something>.”
- The statements should be written as single sentences.
  - The statements relate to the user(s).
  - Without reference to a particular technical implementation.



# Associated information

- Capture the benefits to the stakeholders.
  - Why the requirement is needed.
  - Who benefits from the requirement being fulfilled.
- Add categories or tags.
  - Security – Security compliance.
  - User Interface – High-level user interface features.

# Requirements analysis

- Analyse existing information.
  - An existing version of the software.
  - Other products on the market.
  - Technical documents that describe processes.
    - How the software will be used.
- Customer might provide initial requirements.
  - Request for quote (RFQ) – invitation to bid for work.

# Requirements analysis

- Brainstorm ideas.
  - Initial user requirements.
  - High-level ideas that might be retained, refined or deleted.
- Consult users – more than one iteration.
  - Users in company – stakeholder workshop.
  - Users outside company – surveys.

# Stakeholder workshop

- Invite interested stakeholders.
  - Small team of three to five is better.
  - Include customer focus, technical, infrastructure and project manager.
- Distribute requirements before meeting.
  - Systematic discussion of requirements.
  - Create, update and delete requirements.

# Surveys

- Target main user group.
- Short (10-12 questions maximum) surveys are better.
  - Too many questions implies survey will not be completed.
  - Closed questions to challenge initial requirements.
  - Small number of open questions near end.
- Can use A/B testing for bulk testing some requirements.

<https://www.strath.ac.uk/is/software/qualtrics/>

# Other elicitation techniques

- Interface analysis
  - How interface is used with software.
- Focus group
  - Limited number of users and gather feedback.
- Interviews
  - Interview key stakeholders or experts.
  - Use interview to understand data model.
- Observation
  - Observe existing processes followed by users.
  - Observe users interacting with prototype.



# Tracking requirements

- Must have unique identifier.
  - Use identifier to refer to them – link from test or user story.
  - UR1 (2, 3, 4...) or similar to avoid confusion with tests.
- Never completely delete them.
  - Customer/user might change their mind.
  - Might allow deletion following first workshop/survey.
    - Customer might not care about requirements they did not request.

# Non-functional requirements

- Requirements that cannot be tested, but are constraints:
  - Type of database.
  - Security requirements.
  - Software framework.
  - Operating system.
  - Hardware.

# Example user requirements

Req. #	Description	Type
UR1	The user requires the ability to create their own account.	Functional
UR3	The user requires the ability to reset their password.	Functional
UR2	The user requires functionality on an Android installation.	Non-functional

Non-functional requirements should include supporting software constraints and versions.

# Traceability and testing

- Software should fulfil user requirements.
- Map user requirements to software features.
  - Reverse map for safety critical systems.
- Acceptance tests verify functionality.
  - Map tests to user requirements.

# Requirements in a lifecycle

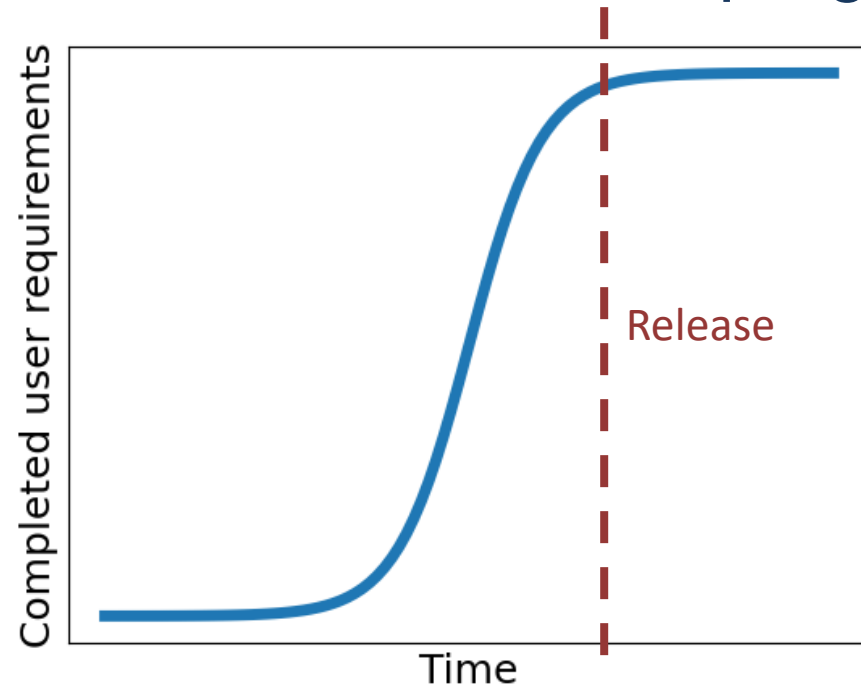
- High-level user requirement → Child user requirement.
- User requirement → system/software requirement.
- System/software requirement → functional requirement.
- Functional requirement → acceptance test.

# Requirements in a lifecycle

- Lack of agility – cannot easily change direction.
  - Hierarchy of requirements is difficult to change.
  - Might lose or omit benefit information.
- More suitable for waterfall or V-lifecycle.

# Requirements in a lifecycle

- S-curve towards delivery.
  - Fulfil requirements just before delivery.
  - Clients unable to see progress or evaluate software.



S-curve: no user requirements are completed in early stages.

# Prioritisation

- How long will it take to fulfil requirement?
  - Skill and size of the team.
- What is the benefit of fulfilling the requirement?
- Think about what could be in a minimum viable product.
- Balance cost vs benefit.
  - Make hard decisions concerning scope.
  - Expect future releases are possible.



# MoSCoW prioritisation

- **Must have.**
  - Must be part of the minimal version of software.
- **Should have.**
  - Very important functionality, which could be omitted.
- **Could have.**
  - Smaller impact on performance if left out.
- **Will not have (this time).**
  - Features that could be implemented in the future.

# **Agile requirements: user stories**

# User stories

- Encapsulate user type, user requirement and benefit.  
“As a <user>, I want <something>, such that <benefit>.”
- Something that the system should do.
  - Not necessarily as precise as a user requirement.
- Short easy to read statements.

# User stories

- Vertical slices of functionality.
  - Implementing a user story should add functionality.
  - Small enough to be developed in days to weeks.
  - Easy to estimate time to implement.
- Recorded in ordered lists.
  - Avoid complex tables and hierarchies of requirements.
  - Created before or during the development process.
    - Just-in-time definition is allowed.

# User stories: development gaps

- Need to avoid S-curves.
  - Add developer user stories to fill progress gaps.

“As a user, I want to be able to save data in the database, so that I can restore information.”
- Ideally should have a smooth burndown chart.
  - Cumulative total work completed in sprint.
- Think about the order of the user stories.
  - One story might be dependent on the implementation of another.

# User stories: INVEST

- Independent.
- **N**egotiable.
- **V**aluable.
- **E**stimable.
- **S**mall.
- **T**estable.

<https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/>

# Splitting user stories

- Fragment workflow user story into steps.
- Fragment business rule user story - e.g. user information.
- Simplest version that could work.
- Variations in data.
- Per user interface feature, rather than cluster them.
- Operations – create, read, update, delete (CRUD).
- Scenarios of a use case – how a user will interact.
- Split technical or functional spike – developer stories.

# Group user stories

- Tag user stories and group them.
  - Security.
  - User interface.
  - Data processing.



# Agile requirements stack

An Epic is addressed by one or more features.

Features are described by one or more stories.

- Investment theme → Epic.
  - Investment theme – High-level feature needed.
- Epic → User stories.
  - Epic – Development to realise investment theme.
- User stories → Tasks.
  - User story – user, feature, benefit.
- Tasks.
  - Developer tasks towards implementing user story.

# Agile requirements stack


An Epic is addressed by one or more features.

Features are described by one or more stories.

## Investment Theme: Security



**Epic:** User authentication



**User story:** “As a user, I want to be able to create an account, so that I can securely access the service.”

# Agile requirements stack

- May not need tasks, depending on complexity of stories.
- May not need Epics, depending on the project.
  - Can introduce Epics to cluster user stories afterwards.

# Tasks: SMART

- **S**pecific.
- **M**easurable.
- **A**chievable.
- **R**elevant.
- **T**ime-boxed.

<https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/>

# Prioritising user stories

- Apply MoSCoW prioritisation.
  - Need cost as well as benefit.

# Cost of user story

- Estimate cost of story.
  - Knowledge – Level of understanding of user story.
  - Complexity – How difficult it will be to implement.
  - Volume – How much there is to implement.
  - Uncertainty – what is unknown in general.
- Formulate estimate in story points.
  - Integer values.
  - Use Cohn's modified Fibonacci series 0, 1, 2, 3, 5, 8, 12, 20, 40, 100 as story values.

# Hybrid methods for new build

- Capture user requirements.
  - Use a stakeholder workshop to confirm these.
  - Estimate build time from requirements.
- Formulate user stories from user requirements.
  - Then continue with Agile sprints.

# Conclusions

- It is vital to:
  - Identify stakeholders.
  - Understand what the stakeholders want from the software.
  - Define the features carefully.
  - Deliver software that matches the deliverables.





# University of **Strathclyde** **Glasgow**