Project Management

Lecture 5

Ruzanna Chitchyan, Jon Bird, Pete Bennett
TAs: Alex Elwood, Alex Cockrean, Casper Wang

(Using materials created by N. Walkinshaw and R. Craggs)

Overview

- About Measurement
- Measurement under White Box:
  - Lines of code
  - Cyclomatic Complexity
- Measurement under Black Box:
  - Planning Poker
- Software Laws:
  - Patents, Copyright, Contract, Privacy

Measurement is Central to Quality

- How to plan for the project time and effort?
  - For the team?
  - For the customer?
- Which software/part of it needs more time for testing?
- Which developer should get a bonus payment for productivity?....

"You cannot **control** what you cannot measure."

Tom DeMarco, 1982

What is “Measurement”?

- Attributing values to objects.
  - The fuel efficiency of a car (gallons per mile)
  - The number of goals scored by a footballer
  - The cost of a house
- Can use these values as basis for comparison
  - What is the cheapest house?
  - Who is the best goal scorer?
- Can use these measurements and comparisons to **make better decisions**.
  - Which car should I buy (e.g., given five candidate cars)
  - Which striker should I put in my team?

Measurement is Difficult in Software Engineering

- Most entities are difficult to measure reliably
- Difficult or impossible to “pin down” a single value

E.g., Software Quality (ISO/IEC 25010):

- Functional Suitability
  - Functional Completeness
  - Functional Correctness
  - Functional Appropriateness
- Performance Efficiency
  - Time Behaviour
  - Resource Utilisation
  - Capacity
- Compatibility
  - Co-existence
  - Interoperability
- Usability
  - Appropriateness
  - Realisability
  - Learnability
  - Operability
  - User Error Protection
  - User Interface Aesthetics
  - Accessibility
- Reliability
  - Maturity
  - Availability
  - Fault Tolerance
  - Recoverability
- Security
  - Confidentiality
  - Integrity
  - Non-repudiation
  - Authenticity
  - Accountability
- Maintainability
  - Modularity
  - Reusability
  - Analysability
  - Modifiability
  - Testability
- Portability
  - Adaptability
  - Installability
  - Replaceability

Usual Metrics: Size and Complexity

- After development ...
  - How much effort will it require for maintenance?
  - Where should we direct testing effort?
  - How much effort was required for development?
  - Metrics are based upon source code (“white box”)

- Before development has started ...
  - How much programming effort will module X require?
  - What will be the estimated cost of the final product?
  - Metrics are based upon requirements / specification (“black box”)

White Box Complexity Metrics

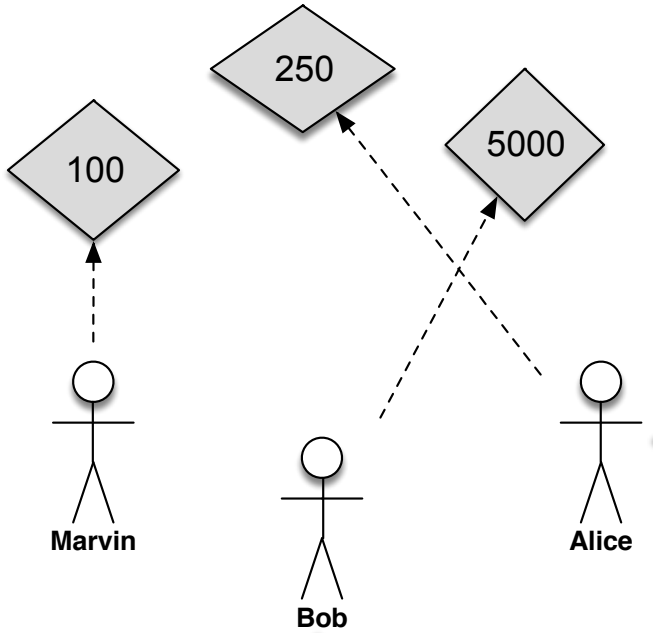
Number of lines in a file (or a group of files)

- Easy to compute
- Easy to understand and interpret
- Often sufficient for an approximate measure of size
- Widely used (perhaps the most widely used) metric

- Comments
- What is a line?
- Blank lines
- Not all “lines” are equal
- Ignores logical/ architectural complexity
- Highly language-specific

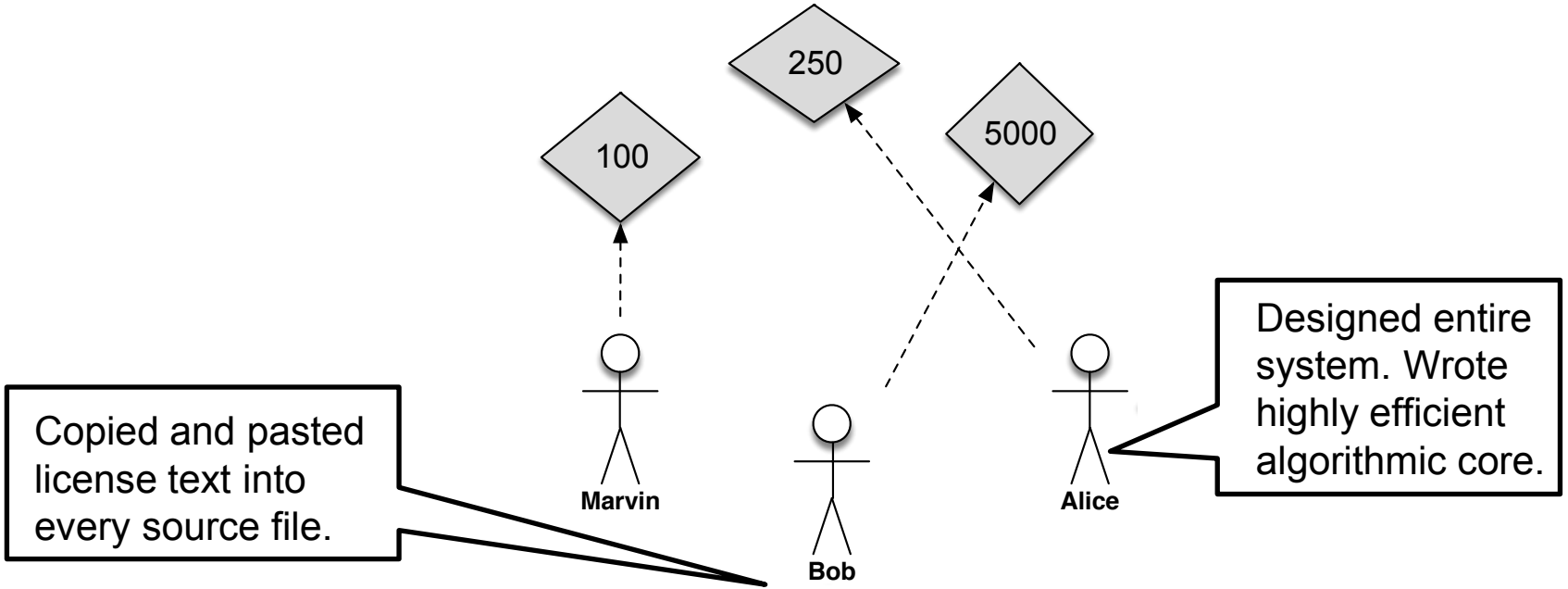
Example: Who is the most productive programmer?

Measured in lines of code



Example: Who is the most productive programmer?

Measured in lines of code



Cyclomatic Complexity

- Calculated from the control flow graph:

  V(G) = E – N + 2P

  E – number of edges;

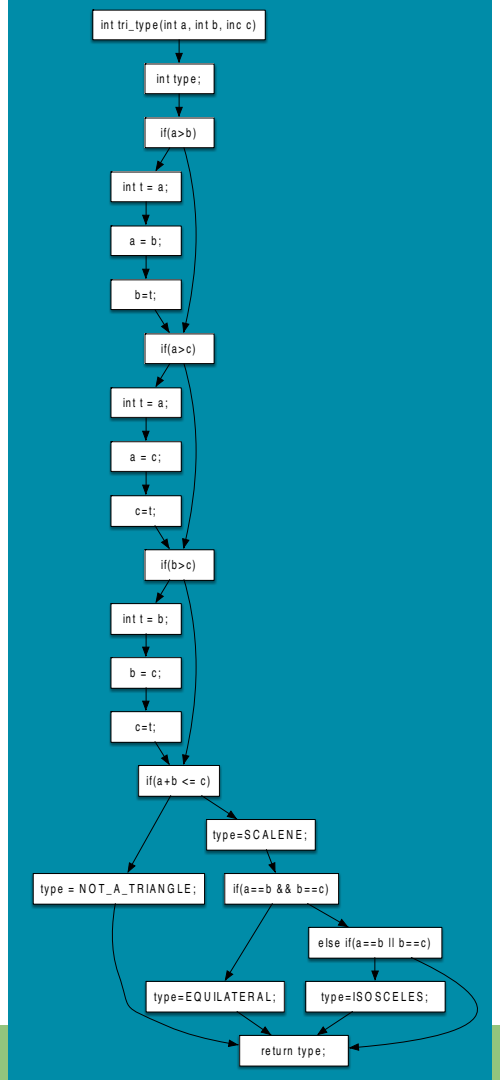
  N – number of nodes;

  P – number of procedures (usually 1)

- Number of independent paths through the code
- Independent path – any path that introduces at least one new statement/condition

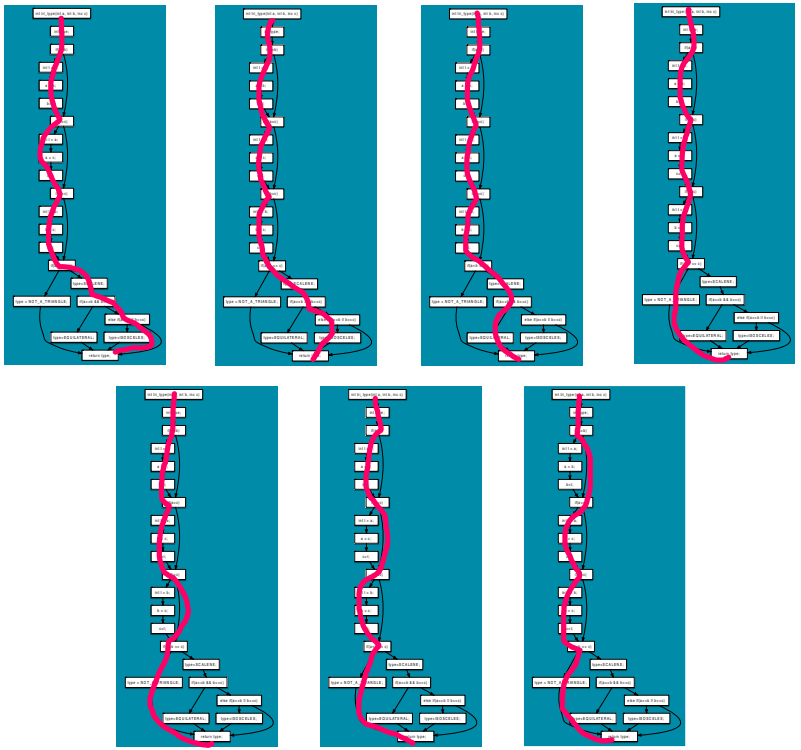
Triangle Example

```
1   int tri_type(int a, int b, int c) {
2     int type;
3     if (a > b)
4       { int t = a; a = b; b = t; }
5     if (a > c)
6       { int t = a; a = c; c = t; }
7     if (b > c)
8       { int t = b; b = c; c = t; }
9     if (a + b <= c)
10      type = NOT_A_TRIANGLE;
11    else {
12      type = SCALENE;
13      if (a == b && b == c)
14        type = EQUILATERAL;
15      else if (a == b || b == c)
16        type = ISOSCELES;
17    }
18  return type;
19  }
```



Number of Edges = 27
Number of Nodes = 22

V = 27 – 22 + 2 = 7



Black Box Complexity Merics

Estimating Agile Projects

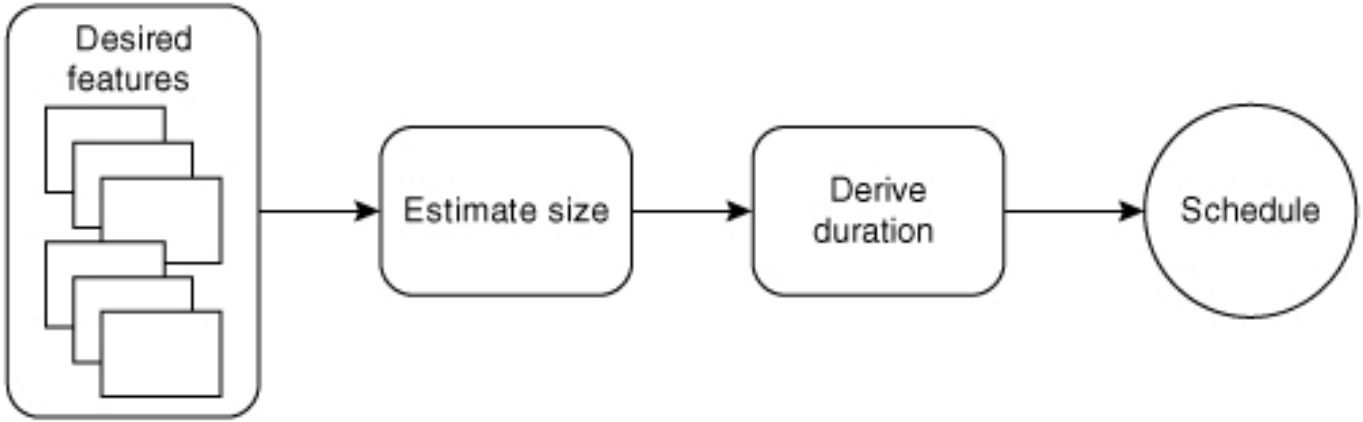


Figure from: Agile Estimating and Planning by Mike Cohn

Storey Points (Size Estimation)

- An informal, agile unit of “size measurement”
    - Usually an estimate from 1-10
- Derive an estimate from the whole team at sprint planning meetings
- Based on the idea of the “Wisdom of the Crowds”
    - The collective estimate of groups (i.e., of effort required for a story) is better than the estimate of an individual

Accuracy vs Effort in Project Estimation

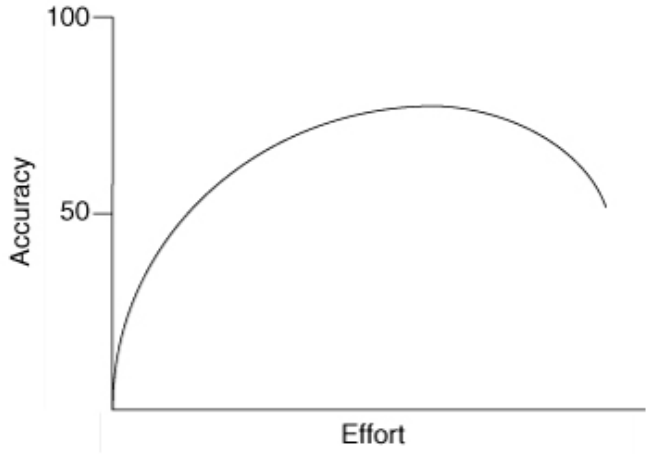
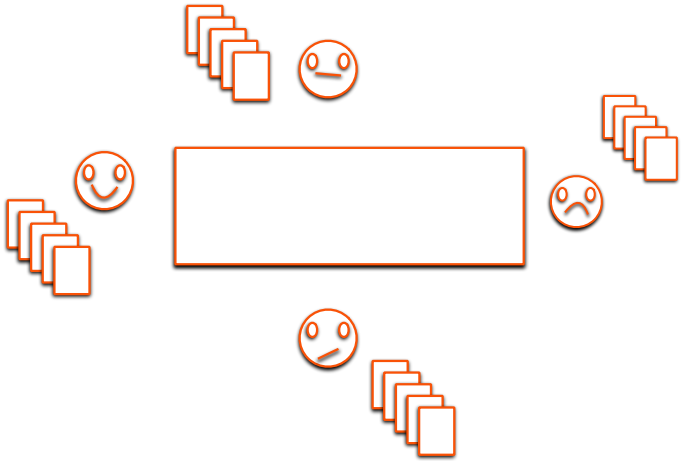


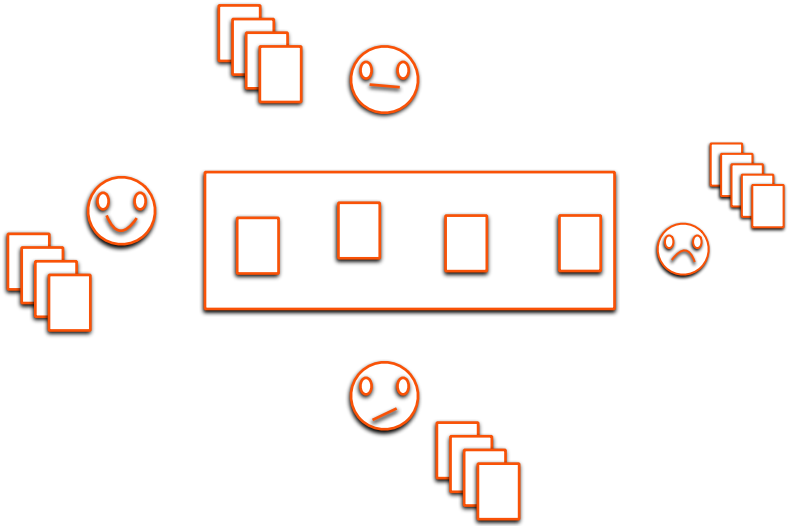
Figure from: Agile Estimating and Planning by Mike Cohn

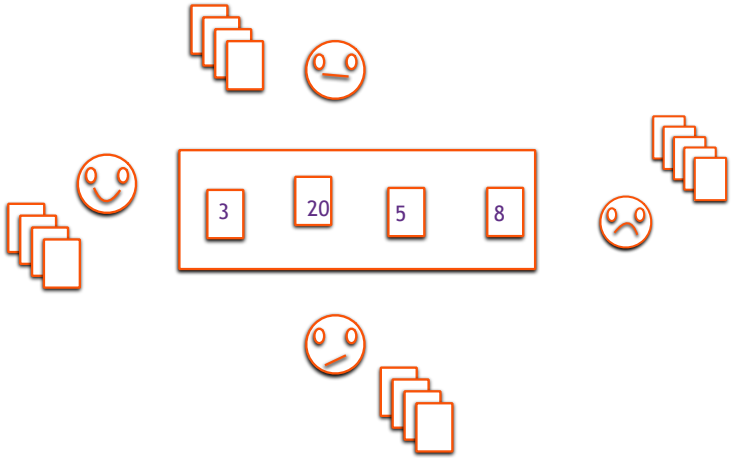
Planning Poker

- The whole team is involved
- Each member is given a set of numbered cards
- Numbers follow the Fibonacci sequence
- 1,3,5,8,13,20,...
  - Larger tasks become harder to estimate in exact terms
  - Low values - trivial to implement
  - High values - difficult to implement
- Each member is also given a “?” card

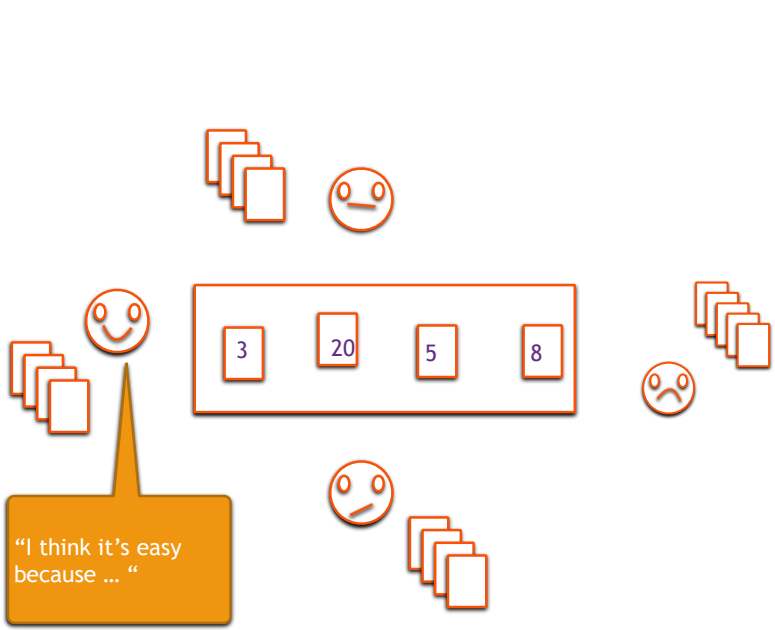


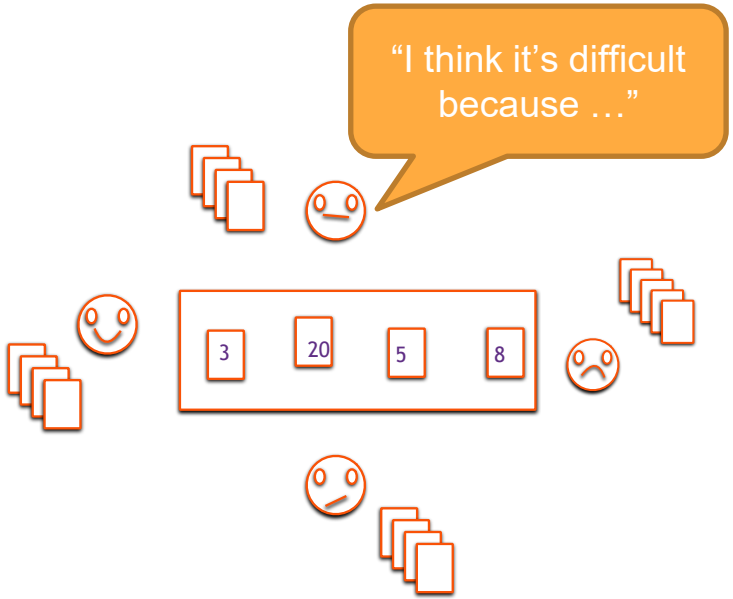
Planning Poker: Process





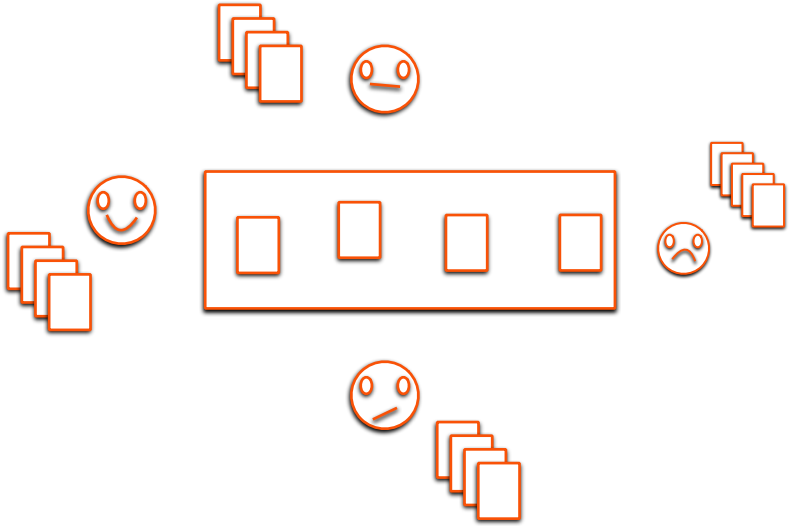
Planning Poker: Process

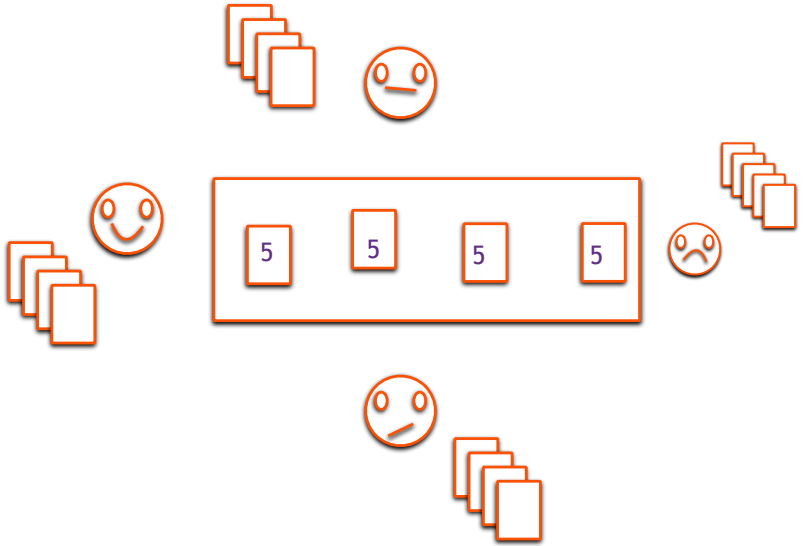




Planning Poker: Process

Cycle repeats for a maximum of 3 iterations (to avoid infinite loops!)



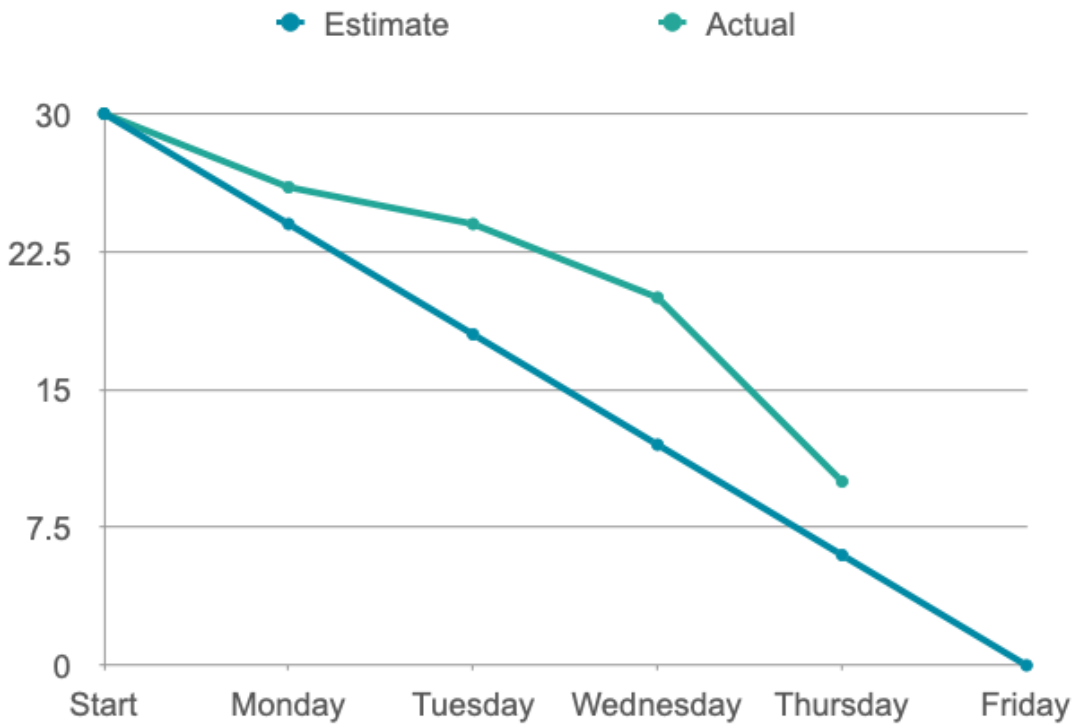


Team Velocity

- Number of (estimated) story points implemented per sprint.
- Can be derived from previous sprints.
    - e.g., Average points implemented from previous *x* sprints.

- Can be used to estimate:
    - Time required to complete project.
    - Target number of stories that can be completed in a sprint.

Burn Charts


Estimate
Actual
30
22.5
15
7.5
0
Start
Monday
Tuesday
Wednesday
Thursday
Friday

Software Laws: Patents, Copyright, Contract, Privacy

Patent Law

A government license giving a right for a set period, especially to exclude others from making, using, or selling an invention

- Granted by the government
- to stop others exploiting **your** invention
- Lasts 20 Year

Inventions Must

- be **new**
- be an **inventive** step (not an obvious improvement)
- capable of industrial application

The “Social Network”

Did Mark Zuckerberg infringe a patent?

- No patent was granted
- The idea was not new, social networks existed before this

Copyright

- Creator has **exclusive rights** to perform, copy, adapt their work.
- Everyone else must get **Permission** (and possibly pay)
- "literary, dramatic, musical and artistic works" **includes software**
- Automatically owned (not granted)
- Lasts **70 years** after authors death (lots of exceptions)

This affects software in 2 different ways:

- Illegal Copies of Applications (Piracy) !
- Using someone else's code/UI design/etc. in your application

(Not the "idea" but the actual "stuff" (code, design, documents) created by someone else)

Copyright Theft?

NO:

- Get permission (obtain a licence)
- Be within "fair use" (e.g. for study or review)
- Use "open source" software
- Create something similar yourself, independently
- "Obvious" code can't be copywrited

YES:

- Displaying an image from another page
- Using code found on the internet
- Copying Windows 95 for your friends

The “Social Network”

Did Mark Zuckerberg infringe copyright?

Maybe

- but there is no evidence he copied
- it it's not fair use
- it wasn't OSS
- he saw the code so didn't invent it himself

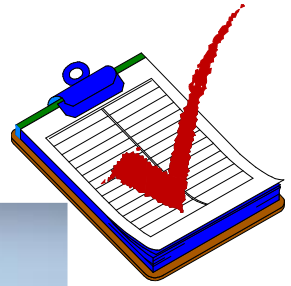
Contract Law

Employer contracts usually force an employee to:

- Not work for anyone else
- Hand over any ideas (Intellectual Property)
- Not disclose company secrets (Non-disclosure-agreements)

(even after you stop working for them)

The “Social Network”



Did Mark Zuckerberg break contract?

Probably Not

- there was no written contract
- he did not disclose any secrets about the other project

Data Protection

- **UK** : Data Protection Act
- **EU** : Data Protection Directive
- **US** : a "patchwork" of state and national laws

8 Principles of Data Protection:

Any company storing "personal data" must make sure it is:

- fairly and lawfully processed (consent, contractual and legal obligations, public interest, ...)
- processed for **limited purposes**;
- adequate, relevant and **not excessive**;
- accurate and, where necessary, kept up to date;
- not kept longer than necessary;
- processed in accordance with the data subject's rights;
- secure;

not transferred to countries without adequate protection

Review

- How can we measure complexity?
- Why do we use black box options?
- What is a patent
- What is the difference between patent and copyright?
- What do we learn about contract from Social Network?

What is Measurement?

GPT:

In Software engineering, Measurement refers to the process of quantifying various aspects of software products, processes, & projects. It involves collecting data & using metrics to evaluate & understand software attributes, performance, quality & progress.

Lecture Notes:

For measurement we use attributing values to objects. In software engineering, attributing values are often numerical or qualitative representations used to quantify or describe specific aspects of software.

Generic example of attributing value:
Fuel efficiency of a car
Cost of a house

SE related examples:
Number of lines in a codebase
Complexity of algorithms.

These attributing values create measurements which is the basis for comparisons, allowing developers to make better, more informed decisions.

Unfortunately, there is no one metric that can be applied universally. In fact most entities are difficult to measure reliably.

A list of just a few potential measurements that could be used.

- Functional Suitability
  - Functional Completeness
  - Functional Correctness
  - Functional Appropriateness
- Performance Efficiency
  - Time Behaviour
  - Resource Utilisation
  - Capacity
- Compatibility
  - Co-existence
  - Interoperability
- Usability
  - Appropriateness
  - Realisability
  - Learnability
  - Operability
  - User Error Protection
  - User Interface Aesthetics
  - Accessibility
- Reliability
  - Maturity
  - Availability
  - Fault Tolerance
  - Recoverability
- Security
  - Confidentiality
  - Integrity
  - Non-repudiation
  - Authenticity
  - Accountability
- Maintainability
  - Modularity
  - Reusability
  - Analysability
  - Modifiability
  - Testability
- Portability
  - Adaptability
  - Installability
  - Replaceability

The most common metrics are some form of size and/or complexity. These will be used throughout the development process. For example:

Before Development:

- How much programming effort does Module X require?
- What is the estimated cost of final product?
- Additional metrics based on requirements/specification. (Black box)

After Development:

- How much effort does maintenance require?
- Where should we direct testing effort?
- How much effort was required for development
- Metrics that are based on the source code (White box)

White box Complexity Metrics

Whitebox complexity metrics are measures used to quantify the structural complexity of software from an internal perspective, focusing on the codes structure, logic & design. These metrics provide insight into how complex & maintainable the codebase is & can help identify areas of improvement.

Commonly used metrics include:

- Number of Lines of Code (LOC)

This is a simple metric, providing a basic measurement of size. However, it does not capture the complexity or quality of the code

- Cyclomatic Complexity

This measures the number of linearly independent paths through a program's source code. A higher cyclomatic complexity indicates greater program complexity & ∴ higher likelihood of defects & increased testing effort. It effectively counts the number of decision points or branches in the code such as if-else statements, loops & case switches.

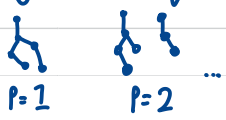
This measurement is based on graph theory. A program's control flow can be represented as a graph, where nodes represent the program's basic blocks (statements, functions, methods etc...) & edges represent the control flow between these blocks.

The formula to calculate cyclomatic complexity is:

$$V(G) = E - N + 2P$$

Cyclomatic Complexity = (No of edges) − (No of nodes) + (2 × No of procedures)

The number of procedures means the number of seperate regions or clusters in the control flow graph. This value is usually 1



There are more white box complexity metrics:

1. **Lines of Code (LOC)**: Measures the total number of lines in the source code. While this metric provides a basic measure of size, it does not capture the complexity or quality of the code.
2. **Cyclomatic Complexity**: Measures the number of linearly independent paths through a program's source code. A higher cyclomatic complexity indicates greater program complexity and may suggest a higher likelihood of defects and increased testing effort.
3. **Depth of Inheritance Tree (DIT)**: Measures the number of class hierarchies that a class is away from the root class in an inheritance hierarchy. A higher DIT value can indicate increased complexity due to multiple inheritance levels.
4. **Number of Methods per Class (NOM)**: Counts the number of methods defined in a class. A high number of methods in a class can indicate potential design issues or complexity.
5. **Coupling Between Objects (CBO)**: Measures the number of classes coupled to a particular class. High coupling can indicate a higher degree of interdependence between classes, making the codebase more complex and harder to maintain.
6. **Depth of Code (DOC)**: Measures the nesting level of control structures like loops and conditionals within methods. Higher levels of nesting can increase complexity and reduce code readability.
7. **Maintainability Index**: A composite metric that combines various code attributes, including cyclomatic complexity, lines of code, and Halstead volume, to provide an overall measure of code maintainability. Higher values indicate better maintainability.

These white box complexity metrics help developers and project managers assess the quality, maintainability, and potential risks associated with the software codebase. By monitoring these metrics, teams can identify complex areas that may require refactoring, additional testing, or architectural improvements to enhance the software's overall quality and maintainability.

But we're not going into that much detail here.

Black box Complexity Metrics

These focus on measuring the complexity of software from an external perspective, without considering the internal structure or implementation details. These metrics are based on the software's requirements, specifications & functionalities rather than the code itself.

Black box complexity metrics evaluate the complexity of software based on its behaviour, interactions & interfaces with external components, users or systems. They are especially useful for assessing the complexity of software systems at a higher level, such as system architecture, design & user interactions

Storey Points

These are a black box metric used in agile development to estimate size or effort required to implement user stories or features

Each story is given a size estimation (these are the storey points). This is an informal agile unit from a scale of 1-10. This allows teams to quickly gauge the relative complexity of different user stories or features during sprint planning.

During the sprint planning meeting, the whole team estimates Storey Points for each story/feature. This provides a more diverse range of perspectives & ∴ arrives at a more accurate & balanced estimation.

"Wisdom of crowds"

- **Based on the idea of the "Wisdom of the Crowds"**

  - The concept of "Wisdom of the Crowds" suggests that the collective judgment of a group is often more accurate than the judgment of an individual. In the context of agile estimation, the collective estimate of the team for the effort required to complete a user story is considered to be more reliable and realistic than an estimate made by an individual team member.

  - **The collective estimate of groups (i.e., of effort required for a story) is better than the estimate of an individual**

  - By pooling together the insights, knowledge, and perspectives of all team members, the team can make a more informed and balanced estimation of the effort required for each user story or feature. This collaborative approach helps in reducing biases, improving accuracy, and fostering team collaboration and communication.

Other Black box methods + more info:

Some commonly used black box complexity metrics include:

1. **Functional Size:** Measures the size of the software based on its functional requirements, such as the number of use cases, user stories, or functional points. Common methods for calculating functional size include Function Point Analysis (FPA) and Use Case Points (UCP).
2. **User Interface Complexity:** Evaluates the complexity of the software's user interface based on factors such as the number of screens, controls, interactions, and navigation paths.
3. **Integration Complexity:** Assesses the complexity of integrating the software with external systems, services, or components. It considers factors such as the number of interfaces, data exchange formats, protocols, and dependencies.
4. **Data Complexity:** Measures the complexity of data structures, data entities, and data flows within the software. It includes factors like the number of data elements, relationships, and data validation rules.
5. **Interoperability Complexity:** Evaluates the complexity of ensuring interoperability between different software components, platforms, or systems.
6. **Functional Dependency Complexity:** Assesses the complexity arising from the dependencies between different functional modules or components within the software.

Benefits of Using Black Box Complexity Metrics:

1. **Focus on User Perspective:** Helps in understanding and evaluating the software complexity from the user's perspective, focusing on functionalities, interactions, and interfaces.
2. **Early Risk Identification:** Enables early identification of potential complexity-related risks, helping in better planning, design, and mitigation strategies.
3. **Improved Design and Architecture:** Provides insights into the design and architecture aspects that may require attention to manage complexity effectively.
4. **Enhanced Communication:** Facilitates better communication among stakeholders by providing a common language and understanding of software complexity.
5. **Supports Decision Making:** Assists in making informed decisions related to resource allocation, prioritization, and trade-offs based on the complexity and criticality of functionalities.

Planning Poker:

This is so stupid I'm just using GPT screenshots:

How Planning Poker Works:

1. **Preparation:** Before starting the Planning Poker session, the Product Owner prepares a list of user stories or tasks to be estimated and ensures that they are well-defined and understood by the team.
2. **Card Deck:** Each team member is given a set of Planning Poker cards, typically featuring values like 0, 1, 2, 3, 5, 8, 13, 20, 40, and 100 (representing story points or effort units).
3. **Estimation Process:**
   - The Product Owner presents a user story or task to the team.
   - Team members independently select a Planning Poker card that represents their estimate of the effort required to complete the user story or task.
   - The cards are then revealed simultaneously to the entire team.
4. **Discussion:**
   - If there is a wide variation in the estimates, team members discuss their rationale behind their estimates.
   - The discussion helps in clarifying assumptions, understanding different perspectives, and reaching a consensus on the estimated effort.
5. **Re-estimation:** After the discussion, the team members re-estimate the user story or task using Planning Poker cards.
6. **Consensus:** The process is repeated until the team reaches a consensus on the estimation. Consensus doesn't necessarily mean everyone agrees, but rather that everyone can live with the estimate.

+ look at diagrams in lecture notes

Benefits of Planning Poker:

- **Collaborative Estimation:** Planning Poker encourages active participation and collaboration among team members, leveraging the collective wisdom and expertise of the team to arrive at more accurate and balanced estimations.
- **Reduced Bias:** By allowing team members to independently estimate and then discuss their estimates, Planning Poker helps in reducing individual biases and anchoring effects, leading to more objective and unbiased estimations.
- **Improved Understanding:** The discussion during the Planning Poker session fosters better understanding and clarity of the user stories or tasks among team members, reducing ambiguities and enhancing shared understanding.
- **Efficiency:** Planning Poker is a relatively quick and efficient estimation technique, making it well-suited for agile teams that value iterative and collaborative approaches to estimation.
- **Engagement and Morale:** The gamified nature of Planning Poker makes the estimation process more engaging and enjoyable for team members, potentially boosting morale and team cohesion.

In summary, Planning Poker is a popular and effective agile estimation technique that promotes collaboration, reduces biases, and enhances the accuracy and understanding of effort estimates among agile teams.

Team Velocity:

Same again This sucks...

Team velocity is a metric used in agile and Scrum methodologies to measure the amount of work a team can complete in a given iteration or sprint. It represents the average number of story points or backlog items the team has successfully delivered in previous sprints. Velocity helps agile teams forecast future work and plan their capacity for upcoming sprints.

Key Points about Team Velocity:

1. **Measurement:** Velocity is typically measured in story points, which are units of estimation representing the effort required to complete a user story or task. However, some teams might use other units like backlog items, hours, or ideal days.
2. **Calculation:** To calculate velocity, you sum up the total number of story points completed by the team in the last sprint(s) and then divide by the number of sprints. For example, if a team completed 30 story points in the last three sprints, their average velocity would be 10 story points per sprint.
   $$\text{Velocity} = \frac{\text{Total Story Points Completed}}{\text{Number of Sprints}}$$
3. **Forecasting:** Velocity is used for sprint planning to forecast how much work the team can commit to in the upcoming sprint. By understanding their average velocity, teams can set realistic goals and avoid overcommitting or undercommitting.
4. **Trend Analysis:** Monitoring velocity over time helps teams identify trends, improvements, or potential issues in their performance. A consistent or increasing velocity indicates stable or improving team productivity, while a decreasing velocity might signal challenges or impediments that need addressing.
5. **Factors Influencing Velocity:** Various factors can influence a team's velocity, including team composition, experience, technical challenges, changes in team dynamics, and external dependencies. It's essential to consider these factors when interpreting velocity metrics.
6. **Relative Metric:** Velocity is a relative metric and varies from team to team based on their context, expertise, and environment. Comparing velocity between teams is generally not recommended due to these variations.

Benefits of Using Team Velocity:

- **Improved Planning:** Velocity helps teams make informed decisions during sprint planning, ensuring that they commit to a realistic amount of work based on their historical performance.
- **Transparency:** Velocity provides transparency into the team's performance, allowing stakeholders to understand the team's capacity, progress, and potential challenges.
- **Continuous Improvement:** By regularly tracking velocity and analyzing trends, teams can identify areas for improvement, implement changes, and continuously enhance their productivity and efficiency.

In conclusion, team velocity is a valuable metric in agile and Scrum methodologies that helps teams plan, forecast, and improve their performance over time. It serves as a tool for understanding capacity, setting realistic expectations, and promoting transparency and continuous improvement within agile teams.

Burn Charts:

Burn charts are visual tools used in agile and Scrum methodologies to track and visualize the progress of a project or sprint over time. They provide a clear and concise representation of the work completed versus the work remaining, helping teams and stakeholders monitor progress, identify trends, and make informed decisions.

Key Components of Burn Charts:

1. **X-axis (Horizontal Axis):** Represents time, typically measured in iterations or sprints. Each point on the x-axis corresponds to a specific time period, such as a day, week, or sprint.
2. **Y-axis (Vertical Axis):** Represents the amount of work remaining or effort required, usually measured in story points, backlog items, or hours. The scale on the y-axis indicates the total amount of work that needs to be completed.
3. **Ideal Burndown Line:** This line connects the total amount of work (starting point) with the point where all work should ideally be completed by the end of the iteration or project (zero work remaining). It serves as a guide to visualize the ideal progress of completing the work.
4. **Actual Burndown Line:** This line represents the actual progress of the team over time, showing how much work has been completed and how much remains. It is based on real data gathered during the project or sprint.

Types of Burn Charts:

1. **Sprint Burndown Chart:** Tracks the progress of a specific sprint by plotting the total effort remaining against the time elapsed in the sprint. It helps the team monitor their progress towards completing the sprint backlog and achieving their sprint goal.
2. **Release Burndown Chart:** Tracks the progress of a release or project over multiple sprints or iterations. It provides an overview of the cumulative work completed and remaining across all sprints, helping stakeholders monitor the project's overall progress and predict the release date.
3. **Burnup Chart:** Similar to a burndown chart but also includes the total scope of work (or the total work added to the backlog) over time. It helps in visualizing both the completed work and the total scope, providing a more comprehensive view of progress.

Benefits of Using Burn Charts:

- **Visibility:** Burn charts offer a visual representation of progress, making it easy for team members and stakeholders to understand the current status, trends, and potential risks.
- **Transparency:** By displaying both the ideal and actual progress lines, burn charts promote transparency by highlighting any deviations and enabling early identification of issues or delays.
- **Decision Making:** Burn charts provide valuable insights that can guide decision-making processes related to scope management, resource allocation, and prioritization.
- **Motivation:** Seeing the progress made can boost team morale and motivation, encouraging team members to maintain their pace and strive towards achieving their goals.

In summary, burn charts are powerful visual tools that facilitate tracking and monitoring of project progress in agile and Scrum environments. They promote transparency, enable informed decision-making, and support continuous improvement by providing a clear and real-time view of the work completed versus the work remaining.

Software Laws: Patents, Copyright, Contract, & Privacy

Covered well in the lectures.

HCI Evaluation Part One

Dr Jon Bird
jon.bird@bristol.ac.uk

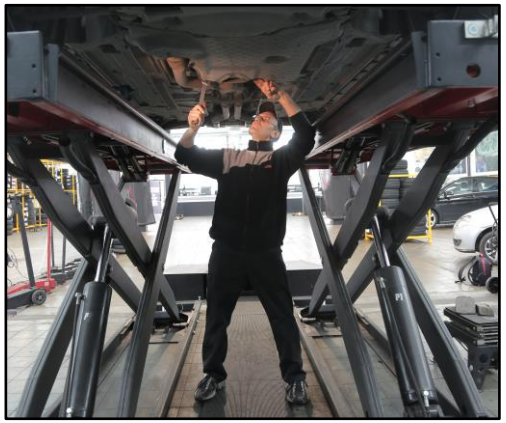
Thanks to Stuart Gray, Pete Bennett, Simon Lock, Thomas Bale, Harry Field who developed some of these slides

Today’s Lecture

- What is HCI evaluation?
- Why is it important
- The Think Aloud evaluation technique
- Heuristic evaluation

HCI Evaluation

- Evaluation is a crucial part of the user-centred development process – we want to ensure our software meets our users' requirements
- The focus of this lecture is on Think Aloud technique and Heuristic Evaluation, which are two of the most widely used evaluation methods in industry
- They are methods that we recommend you carry out on your game as part of your group project – you can write up the results in your report



Why is evaluation important?

- *“Iterative design, with its repeating cycle of design and testing, is the only validated methodology in existence that will consistently produce successful results. If you don’t have user-testing as an integral part of your design process you are going to throw buckets of money down the drain.”*

Bruce Tognazzini (we’ll meet him later in the lecture)

The Think Aloud evaluation technique

- Users are asked to verbalise what they are thinking and doing as they perform a task using your software
- The Think Aloud technique provides insights into the user experience of using your software
- It can identify issues with the software e.g. navigation problems or content that can be improved
- It can be used as part of the software development process to iteratively improve software or used with a finished product

Benefits of Think Aloud

- Cheap
- Relatively easy
- It provides insight into people's experiences as they interact with your product
- It can be carried out with low numbers of participants
- Fits in with most software development processes

Drawbacks of Think Aloud

- it relies on people verbalising thoughts and impressions, rather than objective measures
- Participants may say what they believe to be the right answer rather than what they really think (social desirability). This can distort your results and conclusions

Planning a Think Aloud evaluation

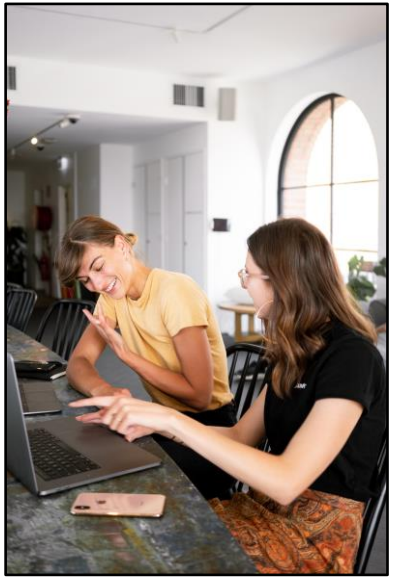
- Decide what questions you want your study to answer. For example, whether users can find particular content or what their understanding is of the information presented.
- Write down the tasks you want the user to complete while using your software
- Decide how many participants you want to recruit and how long you want the sessions to last (45 to 90 minutes works well)

Carrying out a Think Aloud evaluation 1

- Have a **facilitator** to run the evaluation and one or two observers to take notes on what the user says
- Explain to the participants how a think aloud works: they should tell you their thoughts, reactions and emotions as they occur while they are performing the task
- Explain that there is no right answer and it's fine to be critical

Carrying out a Think Aloud evaluation 2

- Ask the participants to complete the tasks you have planned. This should be **uninterrupted** as far as possible, although the facilitator will probably need to give some prompts.
- If the user goes silent then prompt them to verbalise their thoughts by saying “what are you thinking”



Analysing a Think Aloud evaluation

- Put the written notes together from both observes in to one document
- Organise the notes into meaningful categories e.g. what features helped users; what features led to problems; any additional features that users wanted.
- You can make your own meangingul categories
- Count the number of times users comment about different categories to identify the biggest issues

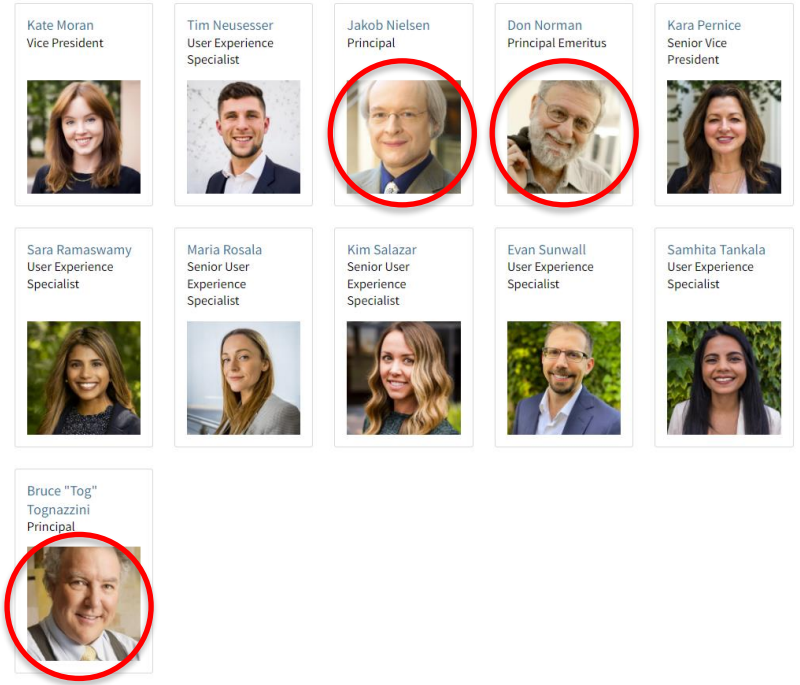
Jakob Nielsen – heuristic evaluation

Nielsen, J., and Molich, R. (1990). Heuristic evaluation of user interfaces, *CHI'90*, 249-256.

https://www.nngroup.com/articles/how-to-conduct-a-heuristic-evaluation/

Nielsen Norman group http://www.nngroup.com/

- The Nielsen Norman group is a UX research and consulting firm
- It was founded by two big figures in the HCI world:
  - Don Norman coined the term “user experience” and developed a set of design heuristics
  - Jakob Nielsen also developed a set of usability heuristics and was a pioneer of heuristic evaluation



What is a heuristic?

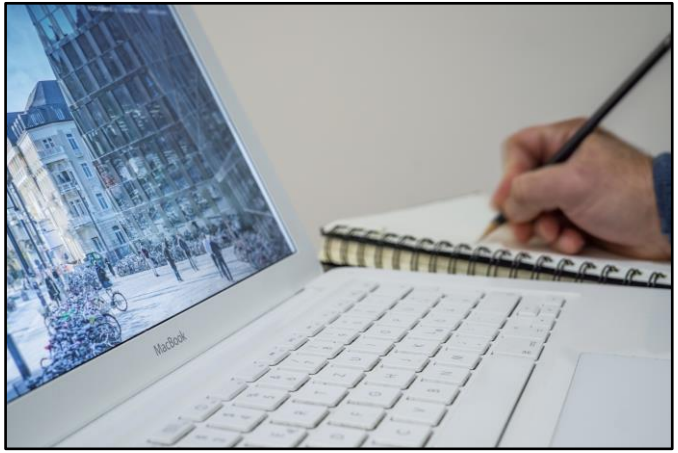
- A rule of thumb
- Experienced-based strategies
- E.g. if you’re doing some DIY then ‘measure twice, cut once’ is a useful heuristic

Heuristic evaluation 1

- An evaluation technique conducted **without** users
- Also known as **expert** evaluation as it's sometimes carried out by external experts (sometimes by the development team) aka evaluators
- It's a type of **analytical** evaluation, that is, based on a set of principles or a model…
- …rather than by observing users (which is known as **empirical** evaluation)

Heuristic evaluation 2

- It's an **inspection** method – it involves inspecting a design to find usability problems
- This involves asking whether the design complies with **usability principles** (a set of heuristics)



Heuristic evaluation is widely used because…

- It's **cheap** (only needs a small number of evaluators and no specialist equipment or labs)
- Relatively **easy** to carry out (can do it after a few hours of training)
- **Instant gratification** – lists of problems are **available immediately** after the inspection
- It **fits in** with most software development processes used in industry
- It's a very **cost effective**: benefit-cost ratio of 48: cost of $10,500; expected benefits $500,000 (Nielsen 1994).

Where are the users?

- Heuristic evaluation is based on HCI researchers' extensive experience of designing and evaluating interfaces
- By focusing on users, HCI researchers learned what works and what doesn't
- Their experience is distilled into **usability principles** (a set of heuristics)
- The principles represent the findings from thousands of user studies
- They have been used for over 30 years

What are Nielsen's 10 principles of heuristic evaluation?

- visibility of system status
- match between system and real world
- user control and freedom
- consistency and standards
- error prevention
- recognition rather than recall
- flexibility and efficiency of use
- aesthetic and minimalist design
- help users recognise, diagnose and recover from errors
- help and documentation

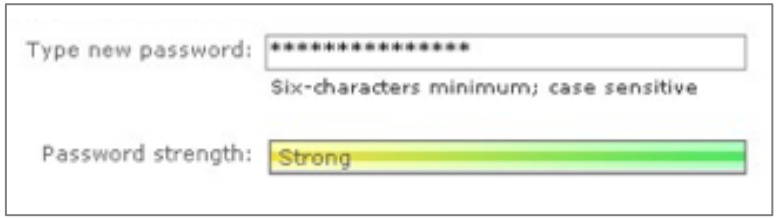
Nielsen’s 10 principles of heuristic evaluation (minimal information)

- feedback
- metaphor
- user control and freedom
- consistency
- error prevention
- recognition not recall
- flexible use
- minimal information
- error recognition and recovery
- help

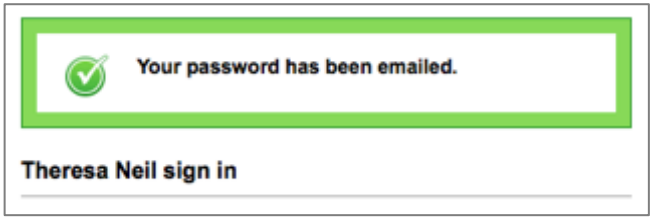
Visibility of system status - feedback

- Inform the user about what's going on:
  - show appropriate feedback and progress
  - do not show blank screens
  - do not show static "load" or progress messages

Visibility of system status: examples



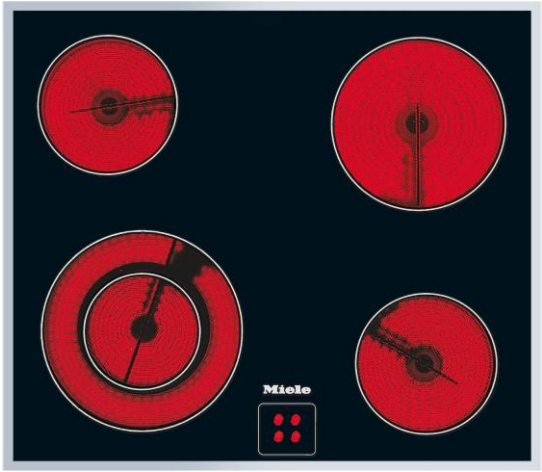
Microsoft Live
Password strength is shown as the password is entered. Colors are used to augment the message.



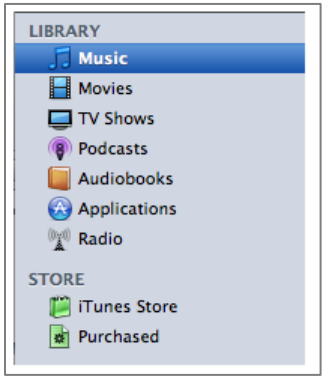
Tick
A feedback message is displayed when an action is performed

Match between system and real world - metaphor

- There must be a match between the system's interface controls and the real world
- The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms
- Follow real-world conventions, making information appear in a natural and logical order



Match between system and real world - examples

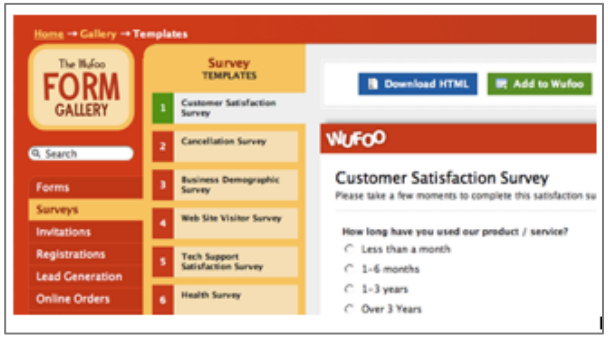


iTunes
Organized as a library that contains your media library: music, movies, TV shows, audiobooks. Beneath the Library is the Store where you can buy more media to put in your Library.

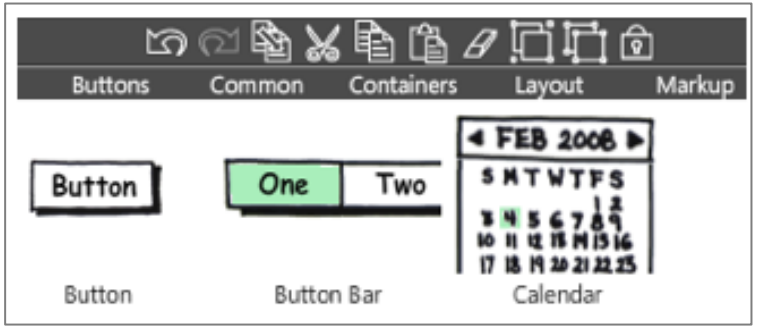
User control and freedom - navigation

- Users often choose system functions by mistake and will need a clearly marked “emergency exit” to leave the unwanted state without having to go through an extended dialog.
- Support undo and redo and a clear way to navigate.
- Provide bread crumbs to clearly show where the user is.

User control and freedom - examples



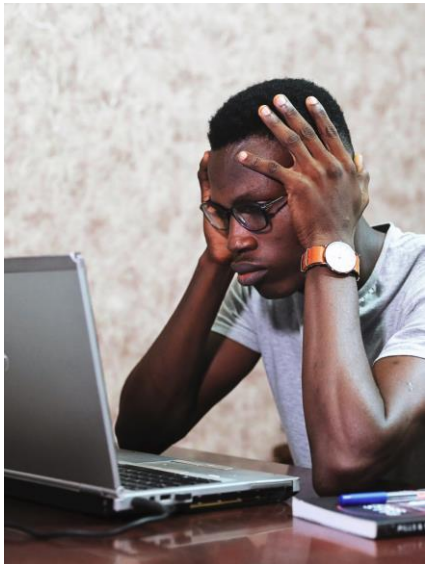
Wufoo
Clearly marks where the person is and where they can go by showing the selection in each menu



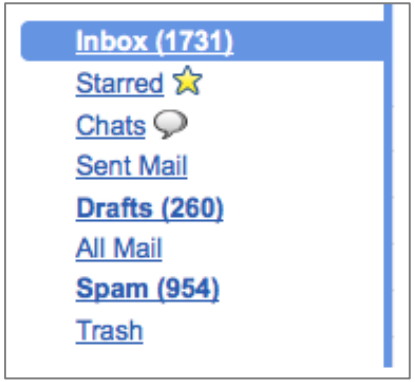
Balsamiq
Undo and Redo buttons are available in the toolbar, and can also be accessed with the standard keyboard shortcuts

Consistency and standards

- Users should not have to wonder whether different words, situations, or actions mean the same thing.
- Follow platform conventions.



Consistency: examples



Gmail
When Gmail was designed, they based the organizational folders on the same ones used in other client email applications: Inbox, Drafts, Sent Mail.

Microsoft Office
Word, Excel, and PowerPoint all use the same style toolbar with the same primary menu options: Home, Insert, Page Layout.

Error prevention

- Even better than good error messages is a careful design which prevents a problem from occurring in the first place.

- Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

Error prevention: examples

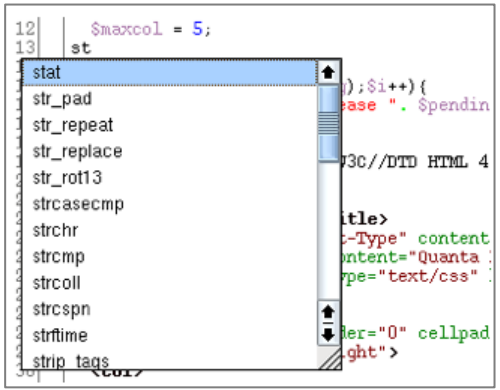
Yammer
Disables the update button after it is clicked, so the person cannot update the post twice by accident

Example from "Web form Design:Filling in the Blanks" by Luke W.
Make the primary action prominent with a larger click area. Cancel and other secondary actions are just shown as links

Recognition rather than recall

- Minimize the user's memory load.
- Make objects, actions, and options visible.
- The user should not have to remember information from one part of the dialogue to another.
- Instructions for use of the system should be visible or easily retrievable whenever appropriate.

Recognition: examples

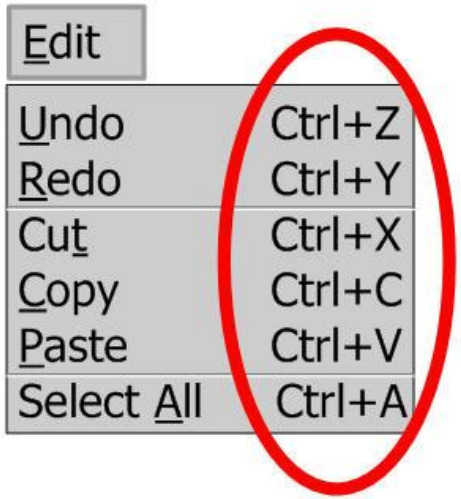


Quanta IDE
Auto completion for coding in a development environment

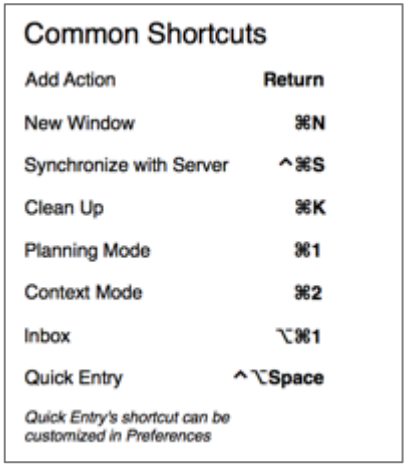
Keynote
Previews the fonts you can pick from, instead of just the font name

Flexibility and efficiency of use

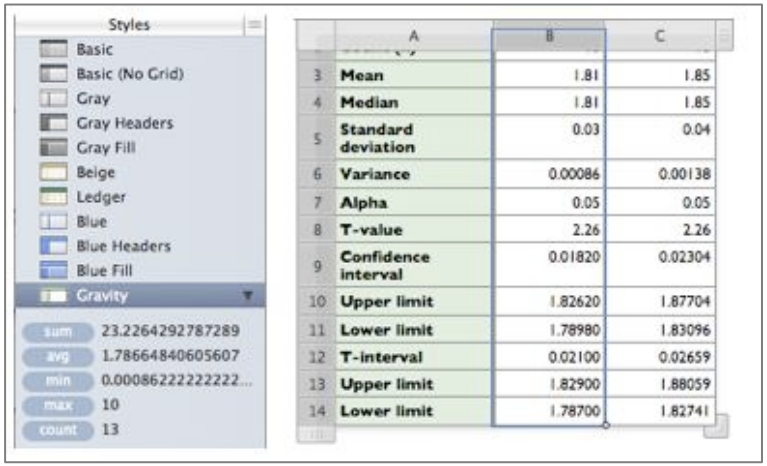
- **Accelerators** — unseen by the novice user — may often speed up the interaction for the expert user so that the system can cater to both inexperienced and experienced users
- Allow users to tailor frequent actions



Flexibility and efficiency: examples



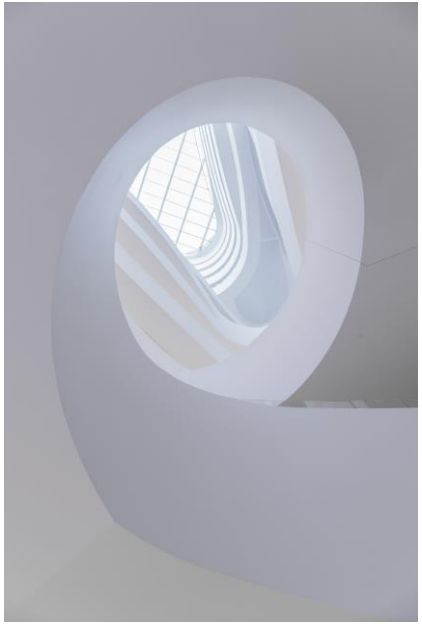
OmniFocus
List of keyboard shortcuts and accelerators



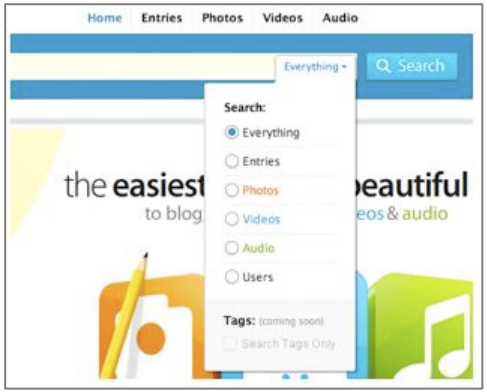
Numbers by Apple
Previews common function results on the left when a column is selected, more efficient than clicking on an action in the toolbar

Aesthetic and minimalist design

- Dialogues should not contain information which is irrelevant or rarely needed
- Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility
- Visual layout should respect the principles of contrast, repetition, alignment, and proximity.



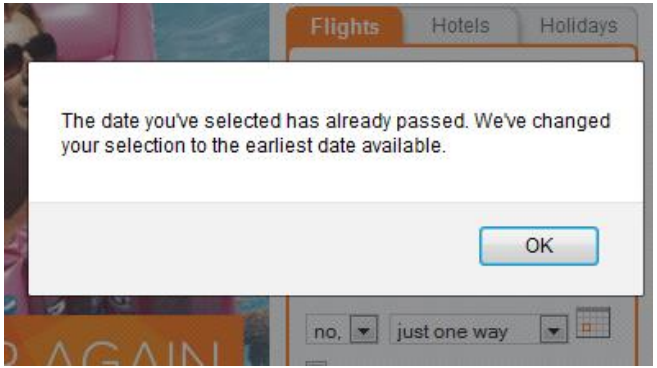
Aesthetics: example



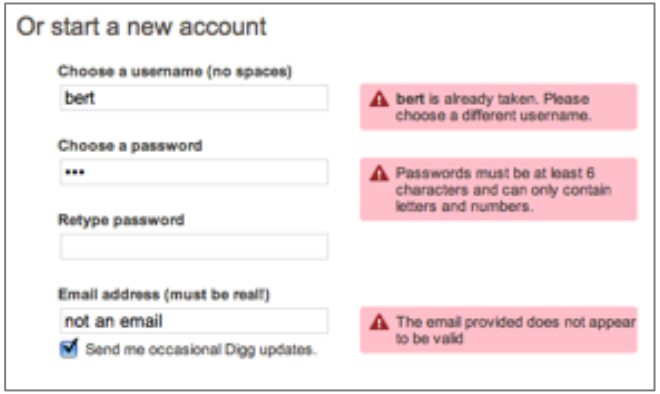
Kontain's search menu exemplifies the four principles of visual design:
1.Contrast: bold text is used for the two labels in the search
2.Repetition: the orange, blue, and green text match the media types
3.Alignment : strong left alignment of text, right aligned drop down
4.Proximity: a light rule is used to separate tags from the other options

Help users recognise, diagnose and recover from errors

- Help users recognize, diagnose, and recover from errors.
- Error messages should be expressed in plain language (no jargon), precisely indicate the problem, and constructively suggest a solution.



Error recognition and recovery: examples

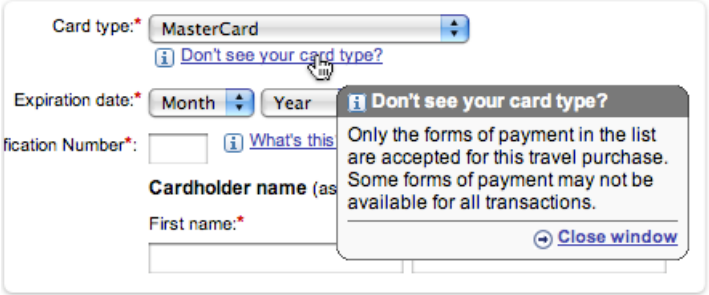


Digg
Provides immediate feedback with specific instructions

Humorous ‘ Page Not Found’ Error
Uses a funny image and text, but provides viable alternatives (article listings and blog link) and a course of action (report it)

Help and documentation

- Even though it is better if software can be used without documentation, it may be necessary to provide help and documentation.
- Any such information should be contextual, easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

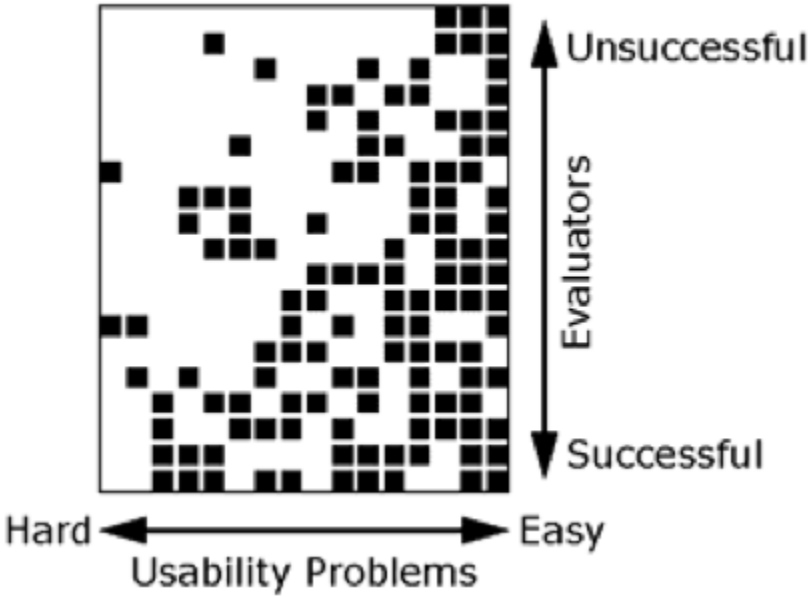


Help and documentation: examples

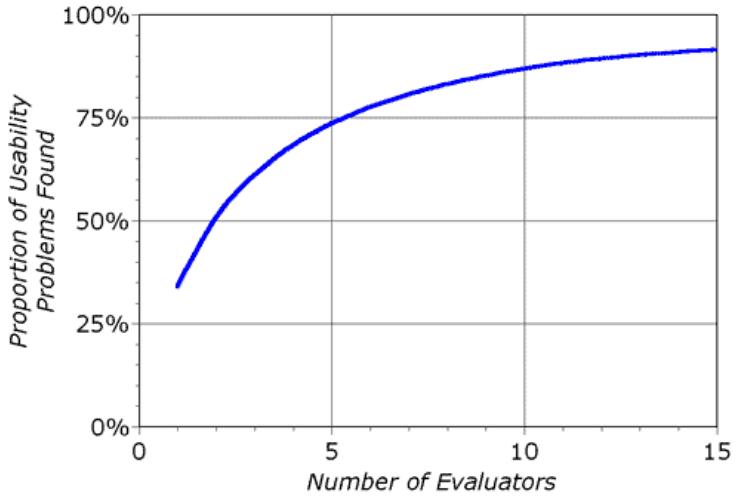
Picnik
Contextual tips in Picnik are clear and easy to navigate

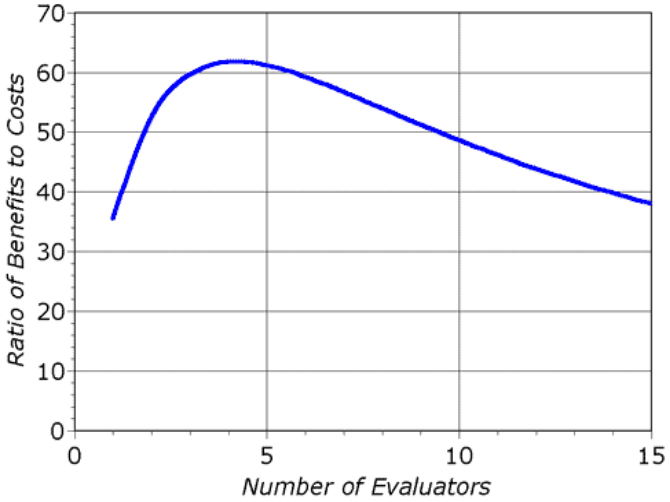
GoodBarry
Embedded videos can be used to showcase features as well as get people started using the product

How many evaluators are needed for heuristic evaluation?



Practical considerations





How to run a heuristic evaluation 1

- Each of the 3 – 5 evaluators does a heuristic evaluation of an interface alone
- Sometimes a facilitator can record the evaluator's comments, sometimes the evaluator does it
- A facilitator **can** answer evaluators' questions, in contrast to traditional user testing, particularly if it's not a walk up and use system
- Heuristic evaluation can be done on paper prototypes

How to run a heuristic evaluation 2

- Heuristic evaluations typically last 1 – 2 hours, but it does depend on the complexity of the software
- The expert goes through the interface several times – first time to get a feel for the system, second time to focus on specific elements
- Evaluators can be given scenarios that describe typical usage scenarios (built from a task analysis of users)
- Evaluators produce a list of usability problems: the usability principle and the design feature that violated it

Benefits of heuristic evaluation

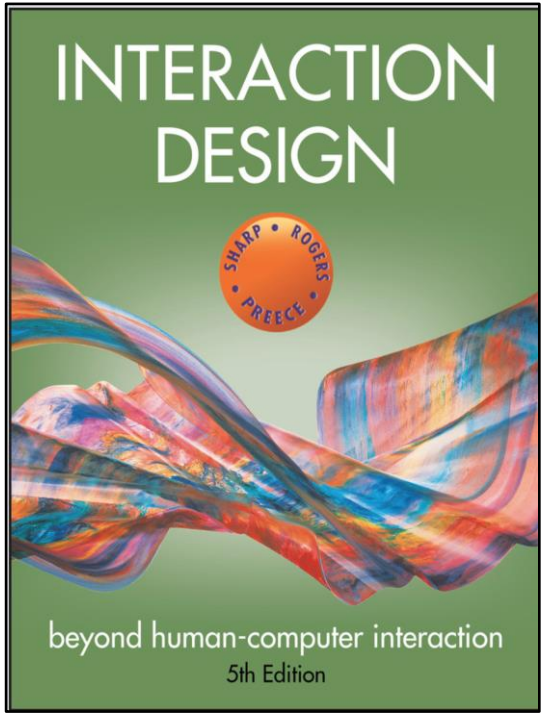
- Cheap
- Relatively easy
- Instant gratification lists of problems are available immediately after the inspection
- It can be carried out with low numbers of participants
- Fits in with most software development processes
- Cost effective

Drawbacks of heuristic evaluation

- Important issues may get missed
- Might identify false issues
- Many trivial issues are often identified, making it seem overly critical
- Experts have biases

Reading

- *Interaction Design: Beyond Human-Computer Interaction* covers all HCI evaluation techniques. It's available through the university library as an eBook. Read about the evaluation techniques covered in this lecture to deepen your understanding
- Read the original Nielsen paper on heuristic evaluation: https://www.nngroup.com/articles/how-to-conduct-a-heuristic-evaluation/



Reading 2

- Explore the materials (papers, articles and videos) on heuristic evaluation on the Nielsen Norman group website:

https://www.nngroup.com/articles/ten-usability-heuristics/

Before the workshop today

- Please review the lecture materials on the Think Aloud and Heuristic Evaluation techniques
- Your workshop activities will involve evaluating your games using these two techniques

HCI Evaluation

HCI = Human Computer Interaction

Regular, iterative evaluation is important as we want to asure our software meets the users' requirements.

GPT:
It ensures quality, effectiveness & success of software products. Other key beefits of regular evaluation include:

① Quality Assurance
- Identifying Defects
- Ensuring Compliance with Specified requirements

② Performance Optimisation
- Improving efficiency by identifying bottlenecks & inefficiencies
- Enhancing user experiance through feed back

③ Decision Making
- Insight to make informed decisions
- Validating Assumptions

④ Continuous Improvment
- Iterative Refinment
- Promoting learning

⑤ Stakeholder Engagment & Transparency
- Building developer-client trust
- Managing Expectations

⑥ Risk Mitigation
- Identifying potential risks, challenges & vulnerabilities
- Ensuring Resiliance

This lecture covers 2 techniques

- Think Aloud Evaluation
- Heuristic Evaluation

Think Aloud Evaluation

Lecture:

Essentially, users are asked to verbalise what they are thinking & doing as they perform a task using your software.

This provides insights into the user experience of using your software

It can identify issues with the software or reveal content that can be improved

It can be used either during the development process to iteratively improve software or applied to a finished problem

Benefits:
- Cheap
- Relatively Easy
- Insight into people's experiences as they interact with your product
- Can be carried out with a low number of participants
- Fits in with most software development processes

Drawbacks:
- Relies on users verbalising thoughts & feelings (subjective) rather than using objective measurements.
- Participants may say what they think is the right answer (socially desirable) rather than what they actually think. This can distort the results/conclusions.

GPT:

Think Aloud Evaluation is a user-centred evaluation method used to gather insights into users' thought processes, actions, & experiences while interacting with a product or system.

It involves asking users to verbalize their thoughts, feelings, & reactions out loud as they navigate through a website, application, or prototype.

Key Features:

① Verbalisation
Participants are encouraged to express their thoughts, opinions, & feedback in real-time as they interact with the product. This provides researches/developers with rich qualitive data about users' decision making processes, challenges & preferences

② Observation
Researches/developers observe & listen to users' verbalisations, actions & interactions to identify patterns, usability issues, pain points & areas of improvment in the user interface & user experience.

③ Contextual Insights
TAE captures users' context specific reactions, emotions & perceptions, providing deeper insights into their needs, motivations & behaviours in real-world scenarios.

Still GPT:

Benefits:

- (Potentially) Insightful Feedback.
  - Valuable insights into users' experience, expectations, & perceptions.
  - Helps identify usability issues, design flaws & areas of improvement.

- User-Centred Design
  - Prioritising user's needs / feedback leads to more intuitive, user-friendly solutions.

- Validation
  - Validates decisions, assumptions & hypotheses by testing them in real world contexts
  - Ensures product meets users' expectations & requirements.

- Iterative Improvement
  - Allows for iterative refinement & optimisation early in the development process.
  - This enhances usability, functionality & overall user experience.

Planning a think aloud evaluation

1. Decide what questions you want your study to answer (e.g can they find particular content? do they understand certain info?)
2. Write down the tasks you want the user to complete while using your software
3. Decide how many participants you want to recruit & how long you want each session to last

Carrying out a think aloud evaluation

- Have a facilitator to run the evaluation & one or two observers to take notes on what the user says
- Explain how a TAE works. Users should tell you their thoughts, reactions & emotions as they occur while they are performing the task
- Explain there is no right answer - its ok to be critical
- Participants should complete the tasks you have planned & be uninterrupted as far as possible
- If they go silent, prompt them to verbalise their thoughts by asking "what are you thinking?"

Analysing a TAE

- Put written notes from both observers into one document
- Organise notes into meaningful categories e.g what helped users? What hindered users? What lead to issues? Any additional features they wanted.
- Count the number of times users comment about different categories to identify the biggest issues.

Heuristic Evaluation.

Heuristic Evaluation is a usability inspection method used in UI design & HCI to identify usability problems in a user interface design.

It involves having a small group of evaluators examine a (UI) design & identify potential usability issues based on a set of predefined heuristics or usability principles.

- What is a Heuristic?
- Basically a rule-of-thumb / an experience based strategy.

(analytical evaluation)

This is a some what more objective evaluation. There are no users. Sometimes it is carried out by external experts. The UI is evaluated on a set of principles (heuristics) or a model, rather than observing users.

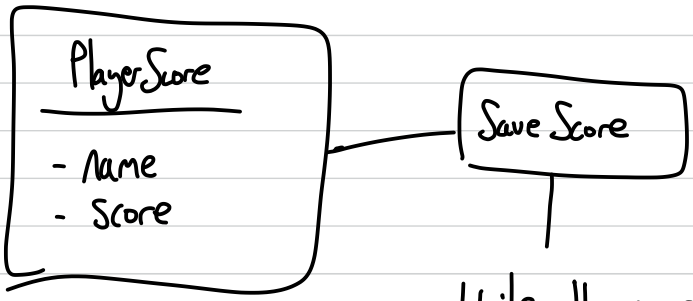
(emperical evaluation)

> Heuristic evaluation is a usability inspection method used in user interface (UI) design and human-computer interaction (HCI) to identify usability problems in a user interface design. It involves having a small group of evaluators examine a user interface design and identify potential usability issues based on a set of predefined heuristics or usability principles.

1. **Selection of Heuristics**: Before the evaluation begins, a set of heuristics or usability principles is selected. These heuristics are guidelines or best practices that are known to impact the usability of a user interface. Commonly used heuristics include visibility of system status, match between system and the real world, user control and freedom, and error prevention, among others.
2. **Evaluation Process**: Each evaluator independently examines the user interface design and interacts with it as a typical user would. They identify potential usability issues by comparing the design against the selected heuristics.
3. **Issue Identification**: Evaluators make notes of any usability problems or violations of the selected heuristics they identify during their examination. These issues are typically described with a brief description of the problem, its severity, and a recommendation for improvement.
4. **Consolidation of Findings**: After all evaluators have completed their evaluations, their findings are consolidated. This involves reviewing and categorizing the identified usability issues, prioritizing them based on their severity and impact on user experience.
5. **Recommendations**: Based on the consolidated findings, recommendations for design improvements are made. These recommendations aim to address the identified usability issues and enhance the overall user experience of the interface.

> Heuristic evaluation is a valuable method for identifying usability problems early in the design process, allowing designers and developers to make informed decisions and improvements before the design is implemented and tested with actual users. While heuristic evaluation can uncover many usability issues, it's important to note that it does not replace user testing with real users, as actual user feedback is essential for understanding and addressing the full range of usability issues in a design.

Tom's notes cover the rest of this very well



Write this.name
& this.score

to a saved scores output file

... For display of leaderboard, read in output file & display top 10 scores

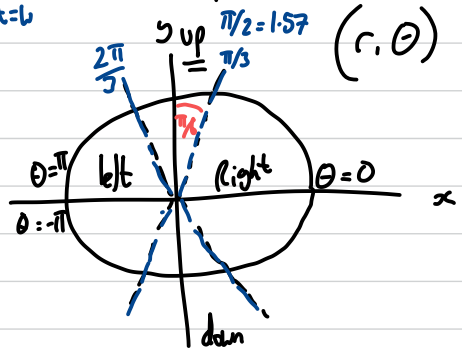
WASD

WA left — right if [1]==D & [0] !=A
WS still
WD right — left if
AS left —
AD still
SD right —

WAS - left
WAD - up
WSD - right
ASD - down

1 0
1 1 1
1 2 1 2
1 3 3 1 3
1 4 6 4 1 4

left=6



$\frac{\pi}{2} - \frac{\pi}{3} = \frac{3\pi}{6} - \frac{2\pi}{6} = \pi/6$

$\frac{\pi}{2} + \frac{\pi}{6} = \frac{4\pi}{6} = \frac{2\pi}{3}$

$\frac{64}{48} = \frac{4}{3}$

32 x 48
⇓
43 x 64
⇓
64 x 96

HCI Evaluation 1 - Qualitative Techniques

- overview of useful techniques that come from Human Computer Interaction
- individual experiences

Assessing your Software met the Requirements

- use evaluation techniques
- evaluation is a crucial part of the **user-centred development process**
  - we must ensure that the software meets the users' requirements
- Focus on Think Aloud and Heuristic Evaluation in this lecture

Why is evaluation important?

“Iterative design, with its repeating cycle of design and testing, is the only validated methodology in existence that will consistently produce successful results. If you don’t have user-testing as an integral part of your design process you are going to throw buckets of money down the drain.”

Bruce Tognazzini (we’ll meet him later in the lecture)

The Think Aloud Evaluation Technique

- users are asked to verbalise what they are thinking and doing as they perform a task using the software
- The TA technique provides insights into the user experience of using your software
- it can help identify issues such as navigation problems (e.g. I cant find the settings menu) or content which can be imporved
- It can be used as part of the software development process to *iteratively* improve software or used with a finished product

Benefits of Think Aloud

- cheap
- easy to set up and no trainig required to understand how it works

    - easy to gather the data
- low number of participants needed
- fits in with most software development processes due to its cost effectiveness and easy

Drawbacks of Think Aloud

- it is based on subjective experiences
    - therefore could get very different feedback from different people
- Various psychological biases may have effects (Social Desirability bias for example) - participants may give what they *think* the right answer is instead of what they really think
    - this can distort results and conclusions
- people may also need to be remined to keep verbalising their thoughts

How to plan a Think Aloud Evaluation

- decide what questions you want your study to answer. For example whether users can find particular content
- Write down some tasks you want the user to complete while using your software
- Recruit a suitable pool of participants and decide on how long you want the sessions to last (45 - 90 mins works well) each participant should ideally not spend longer than 10 minutes doing a think aloud evaluation

Carrying out a Think Aloud Evaluation 1

After the above planning steps are finished, you need to carry out the think aloud

- Include a **facilitator** who should explain tot he participant what they should be doing and then **two observers** who will take notes on what the participant says

The facilitator shoud:

- explain to the participant how a think aloud wokrs: they should be telling you their thoughts, reactuions, emotions as they occur whilst they are using the software
- explain there is no right answer and it is okay to be critical

Carrying out a Think Aloud Evaluation 2

When they are performing the task the facilitator should:

- asl the participant to complete the task you have planned and this should be as **uninterrupted** as possible, although the fascilitator will likely need to give some prompts
- remind them to keep thinking aloud if they stop or go silent by saying “what are you thinking?”

The observers should:

- compile the two sets of notes into one document
- organise the notes into meaningful categories e.g. what features helped the users, what features gave the users problems, any additional features the users wanted
- count the number of times users comment about different categories to identiofy the biggest issues - helps if one view is incredibly niche them maybe shouldnt give it much weight

Neuristic Evaluation

What is a heuristic?

- **A mental shortcut that helps us make decisions fast and effectively**
- Rules of thumb based on experience
- for example a DIY heuristic is “measure twice, cut once”
    - Skiing - “never eat yellow snow”
- these things are not guaranteed but they are quick rules to help us make good decisions

The **Neilsen and Norman Group** is a big UX research and consulting firm

- founded by big figures in the Human Computer Interaction world:
    - Don Norman couined the term *“user experience”* and developed a set of design heuristics
    - Jakob Neilsen also developed a set of heuristics and was a pionerr of heuristic evaluation

Running a Heuristic Evaluation

Heuristic Evaluation 1

- Aka **Expert Evaluation**
- heuristic evaluation is done **without users** instead you would use a group of external **experts**
- the experts being experts in UX

- its a type of **analytical** evaluation that is, it is based on a set of principles or a model, compared to user observation such as think aloud which is known as **empirical** evaluation

Heuristic Evaluation 2

- it is an **inspection method** as it involes inspecting a design to find usability problems
    - or asking a set of experts to inspect the software
- this involves askign whether the design complies with **usability principes** (a set of heuristics)

Why conduct Heuristic Evaluation?

- it is cheap, only requiring a small number of evaluators and no specialist equipment or labs
- relatively easy to carry out - can do it after a few hours of training to become familiar with the heuristics
- instand gratification - a list of results is available imediately after inspection
    - think aloud for example takes some time to produce results after analysis etc
- Similar to Think Aloud, it also fits in with most software development processes used in industry
- It is **very cost-effective**: benefit-cost ratio of 48: cost of $10,500; expected benefits of $500,000 (Neilsen, 1994)
    - this is based on data neilsen collected but take with pinch of salt

Who are the users/experts?

- Heuristic evaluation is based on HCI researchers extensive experiences of designing and evaluation interfaces

- By focussing on users, HCI researchers learned what worksd and what doesnt
- theyr experience is distilled into **usability principles**
- the principles represent the findings from thousands of studies
- they have been used for 30 years

Neilsen’s 10 Principles of Heuristic Evaluation

- visibility of system status
- match between system and real world
- user control and freedom
- consistency and standards
- help and documentation
- error prevention
- recognition rather than recall
- flexibility and efficiency of use
- aesthetic and minimalist design
- help users recognise, diagnose, and recover from errors

Neilsen’s 10 Principles of Heuristic Evaluation (simplified)

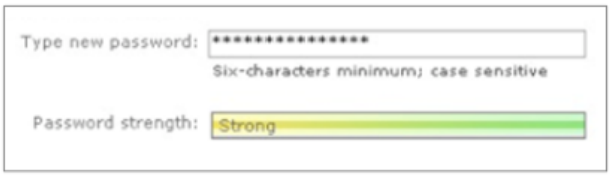
- feedback
- metaphor
- user control and freedom
- consistency
- error prevention
- recignition not recall
- flexible use
- minimal information

- error recognition and recvery
- help

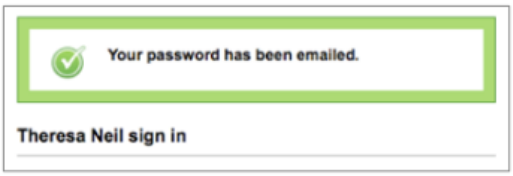
1. Visibility of System Status (feedback)

- inform the user about what is going on:
  - show appropriate feedback progress
  - do not show blank screens
  - do not show static “load” or progress messages
- every interaction the user makes the system needs to register it and let the user know things are happening

Examples:



Microsoft Live
Password strength is shown as the password is entered. Colors are used to augment the message.



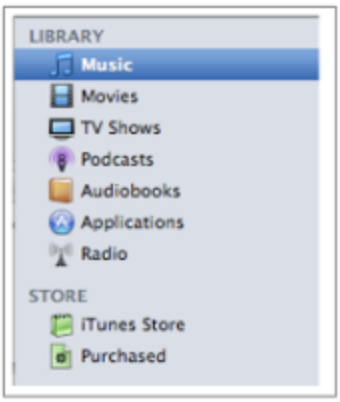
Tick
A feedback message is displayed when an action is performed

2. Match between the system and real world (metaphor)

- there must be a match between the system’s interface controls and the real world

- if there are existing conventions in the world then you should use them, else you risk confusing the user
- the system should speak to the users' language, with words, phrases and concpepts which are familiar to the user rather than system-oriented terms
    - think about throwing exceptions in Java with helpful user friendly error messages, instead of priniting the stack trace which is only helpful to the developer
- follow real-world conventions making info appear in a natural and logical order

Examples:



iTunes
Organized as a library that contains your media library: music, movies, TV shows, audiobooks. Beneath the Library is the Store where you can buy more media to put in your Library.

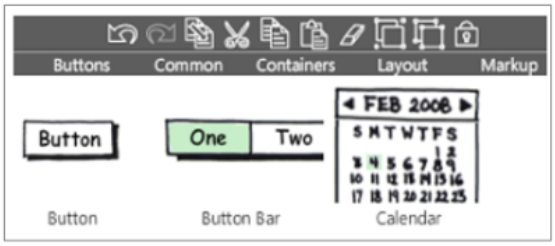
- the library metaphor to real like libraries

3. User Control and Freedom (navigation)

- users often choose system functions by mistake and will need a **clearly marked** “emergency exit” to leave the unwated state without having to go through extensive dialogue
- suport for undo and redo and a clear way to navigate
- provide breadcrumbs to clearly show where the user is
    - i.e. letting them know what level on a website the user is in the navigation bar

Examples:

Wufoo
Clearly marks where the person is and where they can go by showing the selection in each menu



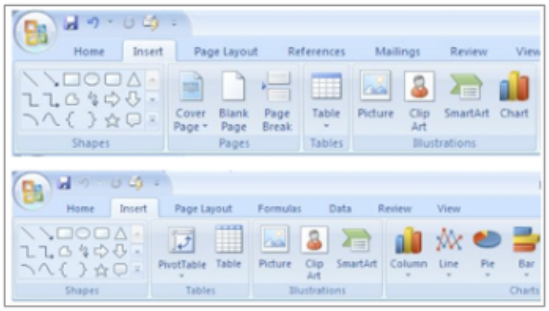
Balsamiq
Undo and Redo buttons are available in the toolbar, and can also be accessed with the standard keyboard shortcuts

4. Consistency and Standards

- users shouldnt have to wonder whether different words, situations, or actions mean the same think
- follow platform conventions

Examples:

Gmail
When Gmail was designed, they based the organizational folders on the same ones used in other client email applications: Inbox, Drafts, Sent Mail.



Microsoft Office
Word, Excel, and PowerPoint all use the same style toolbar with the same primary menu options: Home, Insert, Page Layout.

5. Error Prevention

- what is better than good error messages is careful desgin which prevents a problem from occuring in the first place
- Either eliminate error-prone conditions or check for them and preent users with a confirmation option before they commit the action
    - e.g. “are you sure you want to do this [ACTION]?”

Examples:

Yammer
Disables the update button after it is clicked, so the person cannot update the post twice by accident

Example from “Web form Design:Filling in the Blanks” by Luke W.
Make the primary action prominent with a larger click area. Cancel and other secondary actions are just shown as links

6. Recogntion rather than Recall

- minimise the memory load of the user
- make objects, acrtions, options visible
- user shouldnt have to remember information from one part of the dialogue to another
- Instructions for use of the system should be visible or easily retrievable when appropriate

Examples:

Quanta IDE
Auto completion for coding in a development environment

Keynote
Previews the fonts you can pick from, instead of just the font name

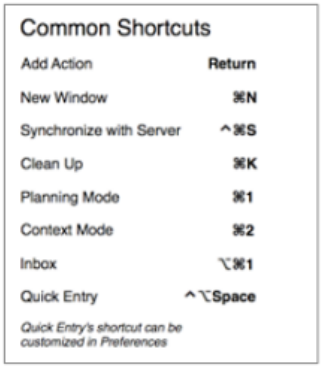
E.g. IDEs such as Intellij offer auto complete options when writing code

7. Flexibility and Efficiency of Use

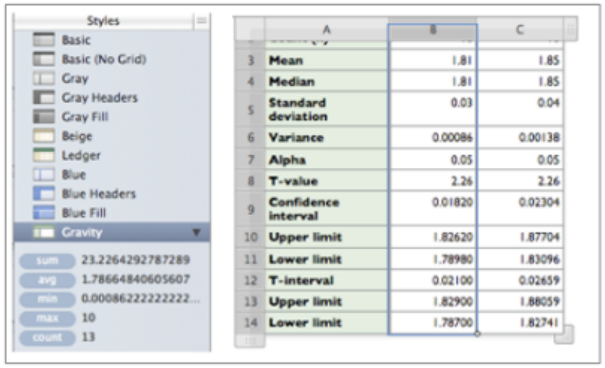
- **Accelerators** - shortcuts for users to interact with the system in a faster way when they are more familiar with the system

- allows frequent users to interact more efficiently

Examples:



OmniFocus
List of keyboard shortcuts and accelerators



Numbers by Apple
Previews common function results on the left when a column is selected, more efficient than clicking on an action in the toolbar

8. Aesthetic and Minimalist Design

- dialogues should not contain irrelevant information or info which is rarely needed
- every extra unit of info in a dialogue competes with the relevant bits of info will distract and overload user attention
- Visual layout should respect the principles of contrast, repetiion, alignment, and proximity

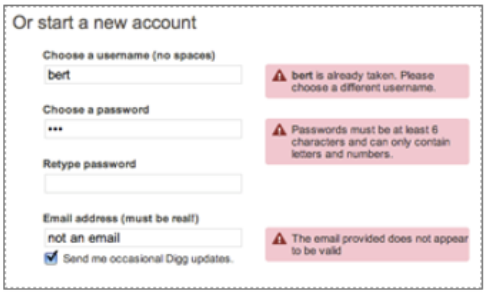
Example:

Kontain's search menu exemplifies the four principles of visual design:
1.Contrast: bold text is used for the two labels in the search
2.Repetition: the orange, blue, and green text match the media types
3.Alignment : strong left alignment of text, right aligned drop down
4.Proximity: a light rule is used to separate tags from the other options

9. Help users recognise, diagnose, and recover from errors

- error messages should be expressed in plain langauage, no technical jargon
- precisely indicate the problem and constructively suggest a solution

Examples:

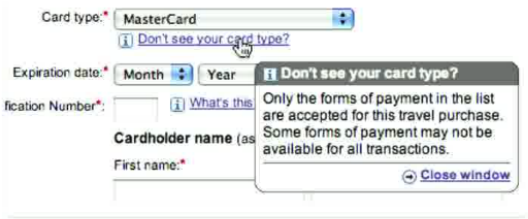


Digg
Provides immediate feedback with specific instructions

Humorous ' Page Not Found' Error
Uses a funny image and text, but provides viable alternatives (article listings and blog link) and a course of action (report it)

10. Help and Documentation

- even though it is better if software can be used without documentation, it may sometimes be necessary to provide it
- any such info should be contextual, easy to search, focussed on the users task, list concrete steps to be carried out, and not be too large

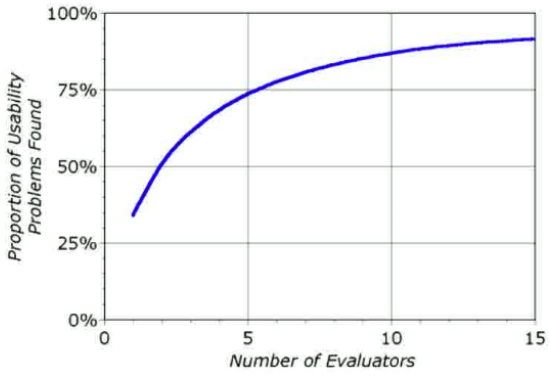


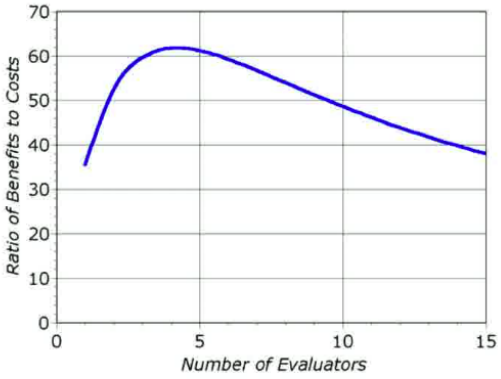
Examples:

Picnik
Contextual tips in Picnik are clear and easy to navigate

GoodBarry
Embedded videos can be used to showcase features as well as get people started using the product

How many evaluators needed for heuristic evaluation?





- after a point, the number of evaluators doesnt have much more of an impacrt on identificatiopn of problems
- the best is around 3 - 5 evaluators

How to run Heuristic Evaluation 1

- each evaluator does a heuristic evaluation of an interface **alone**
- sometimes a facilitator will record the evaluators comments, or sometime the evaluator themselves does this
- a facilitator can answer the evaluator's questions, this is in contrast to traditional use testing. This isnt an empirical study therefore if they have a questions it is likely they have already identified an issue.

How to run Heuristic Evaluation 2

- Heuristic evaluations can often last 1 - 2 hours but this depends on the complexity of the software
- the expert goes though the system first and become familiar with it and then a second time to focus on specific elements
- evaluators can be given scenarios that describe typical use scenarios
- evaluators produce a list of **usability problems,** these specify the usability principle and the design feature that violated it

Benefits of Heuristic Evaluation

- cheap
- relatively easy
- instant list of problems is provided
- low number of participants required
- fits in with most software development processes

- cost effective

Downsides of Heuristic Evaluation

- important issues may get missed
- might identify false issues
- many trivial issues are often identified making it seem overly critical
- experts are also prone to biases

HCI Evaluation Part Two

Dr Jon Bird
jon.bird@bristol.ac.uk

Thanks to Stuart Gray, Pete Bennett, Simon Lock, Thomas Bale, Harry Field who developed some of these slides

Today’s Lecture

- Questionnaires
- NASA Task Load Index (NASA TLX)
- System Usability Scale (SUS)
- Statistical tests to determine if the perceived workload or system usabiity score has changed significantly

Questionnaires - defined

- Questionnaires involve asking people to answer questions either on paper or digitally e.g. on a webpage or app
- They can be used at scale with low resource requirements
- They generate a collection of demographic data and user opinions
- They can be used to evaluate designs and for understanding user requirements

Questionnaires - tips

- Ensure that you are asking a feasible number of questions (question fatigue is a thing)
- Watch out for leading questions e.g. “Why did you have difficulty with the navigation?”
- It is difficult to produce your own questionnaires
- It is best to use existing questionnaires that have been validated i.e. they measure what they claim to be measuring
- I’ll now introduce you to two widely used questionnaires

NASA TLX

- The NASA Task Load Index (TLX) is a questionnaire that estimates a user's perceived workload when using a system.
- Workload is a complex construct but essentially means the amount of effort people have to exert, both mentally and physically, to use a system.
- It was developed by Sandra Hart of NASA's human performance group and Lowell Staveland of San Jose University.
- The focus is on measuring the "immediate often unverbalized impressions that occur spontaneously" (Hart and Staveland, 1988). These are difficult or impossible to observe objectively.

NASA TLX 2

- Originally the NASA TLX questionnaire was developed for use in aviation but it's since been used in many different domains, including air traffic control, robotics, the automative industry, healthcare, website design and other technology fields.
- Since it was introduced in 1988, it has had over 8000 citations.
- It is viewed as the gold standard for measuring subjective workload.

NASA TLX 3

- Originally it was developed as a paper and pencil questionnaire but there are also free apps for iOS and Android
- The official website is here: https://humansystems.arc.nasa.gov/groups/TLX/index.php

NASA TLX 4

- The NASA TLX uses a multi-dimensional rating procedure that derives an overall workload score based on a weighted average of ratings on six subscales:

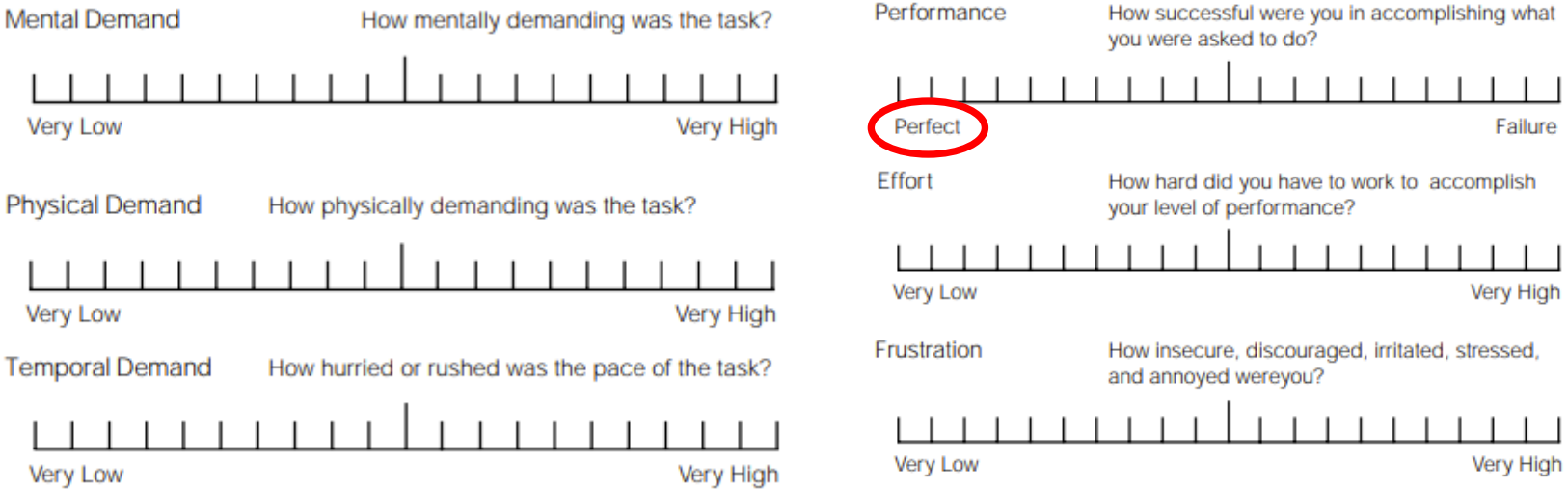
  - Mental Demand
  - Physical Demand
  - Temporal Demand
  - Performance
  - Effort
  - Frustration

NASA TLX 5

- Mental demand – how much mental and perceptual activity was required?
- Physical demand – how much physical activity was required?
- Temporal demand – how much time pressure did the user feel due to the rate at which tasks occurred?
- Frustration – how insecure, discouraged or irritated did the user feel in the task?
- Effort – how hard did the user have to work (mentally and physically) to accomplish their level of performance?
- Performance – how successfully did the user think they accomplished the task?

NASA TLX 6

20 point rating scale
range of 0-100



NASA TLX Scoring 1

- Users answer the NASA TLX after they have completed a task. This is necessary as asking them to complete it during task is typically not possible. However, it may mean that users forget details of the perceived workload.
- The questionnaire is scored in a two step process:

1. Identifying the relative importance of the 6 dimensions on a user's perceived workload
2. Rating each of the 6 dimensions on a scale

NASA TLX Relative weighting of dimensions 1

- A user reflects on the task they've been asked to perform and is shown each paired combination of the six dimensions to decide which is more related to their personal definition of workload as related to the task.
- This means a user considers 15 paired comparisons. For example, they need to decide whether Performance or Frustration "represents the more important contributor to the workload for the specific task you recently performed."
- Each time a dimension is selected as more important it receives a score of 1. The total score is the weight of the dimension and ranges from 0 to 5.
- The sum of the weights should be 15.

	Mental demand	physical demand	temporal demand	performance	effort	frustration
Mental demand	X					
physical demand	~~X~~	X				
temporal demand	~~X~~	~~X~~	X			
performance	~~X~~	~~X~~	~~X~~	X		
effort	~~X~~	~~X~~	~~X~~	~~X~~	X	
frustration	~~X~~	~~X~~	~~X~~	~~X~~	~~X~~	X

X self comparison

~~X~~ duplicate comparison

15 comparisons. Each time a dimension is selected as more important it gets a +1 score. This means each dimension has a score of 0-5 - the total sum also = 15.

End up with something like:

Mental demand	physical demand	temporal demand	performance	effort	frustration
4	2	2	0	5	2

Say a user gives the scores

Mental demand	physical demand	temporal demand	performance	effort	frustration
10	60	75	25	40	90

$$\text{Weighted Score} = \frac{(10 \times 4) + (60 \times 2) + (75 \times 2) + (25 \times 0) + (40 \times 5) + (90 \times 2)}{15}$$

NASA TLX Relative weighting of dimensions 2

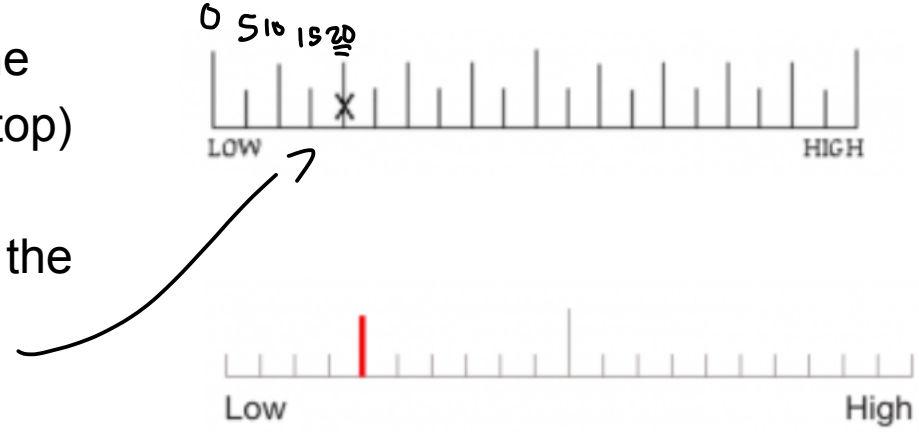
- The relative weighting of the six dimensions is often **not** measured or used.
- Not measuring the relative weighting makes the NASA TLX simpler to administer.
- Several studies have compared raw TLX scores to weighted TLX scores and have found mixed results (some showing better sensitivity when removing weights, others showing no difference, and others showing less sensitivity).
- When the dimensions are not rated the method is called the ‘raw TLX score’

NASA TLX Rating the dimensions 1

- Users mark their score on each of the six dimensions.
- Each dimension consists of a line with 21 equally spaced tick marks, which divide the line from 0 to 100 in increments of 5. If a user marks between two ticks then the value of the right tick is used.
- The score on a dimension is calculated as the tick number (1, 21) – 1 multiplied by 5.

NASA TLX Rating the dimensions 2

- For example, the images show the rating on a paper questionnaire (top) and on a mobile app (bottom)
- The fifth tick mark is selected, so the rating score is: (5 – 1) * 5 = 20



This is basically just converting it from

0 - 20 → 0 - 100

NASA TLX What do the scores tell us?

- If the weights are used then the individual ratings on each of the dimensions are multiplied by their respective weights, summed and divided by 15, resulting in an aggregate perceived workload score for a task ranging from 0 – 100.
- If the weights are not used then the individual ratings on each of the dimensions can be summed and divided by 6, resulting in an aggregate perceived workload score ranging from 0 – 100.
- The individual ratings on the 6 dimensions also give some insight in to where the workload is coming from. This can be helpful for developers hoping to improve their design.

NASA TLX Validity

- Hart and Staveland validated that the sub-scales measure different sources of workload.
- Subsequent independent studies have also found that the NASA TLX is a valid measure of subjective workload (Rubio et al, 2004; Xiao et al, 2005).

System Usability Survey (SUS)

- The System Usability Scale (SUS) provides a “quick and dirty”, reliable tool for measuring usability.
- It was created by John Brooke in 1986.
- It consists of a 10 item questionnaire with five response options for each item ranging from Strongly agree to Strongly disagree.
- It enables the evaluation of a wide variety of products and services, including hardware, software, mobile devices, websites and applications.

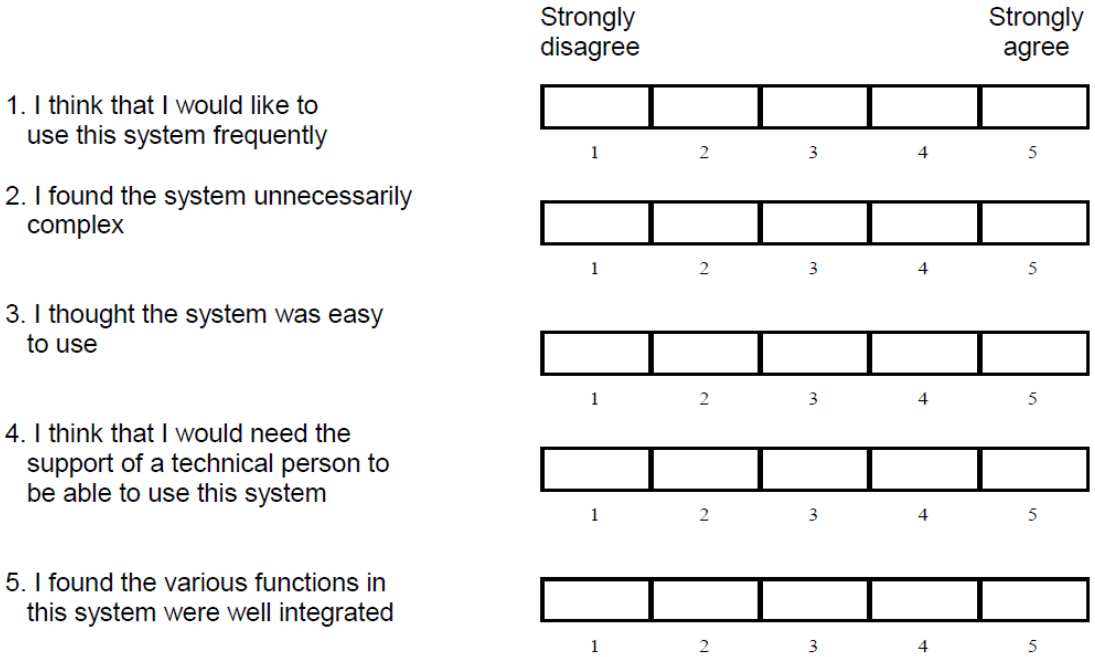
System Usability Survey (SUS) - benefits

- SUS has become an industry standard, with references in over 1300 articles and publications.
- The noted benefits of using SUS include:
- It is a very easy scale to administer to participants
- It can be used on small sample sizes with reliable results
- The SUS has been validated and shown to effectively differentiate between usable and unusable systems

System Usability Survey (SUS) - scale

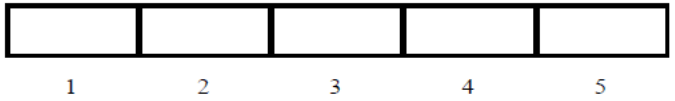
- When an SUS is used, participants are asked to score the 10 items with one of five responses that range from Strongly Agree to Strongly disagree i.e. using a five point Likert scale

System Usability Survey (SUS) – scale 2

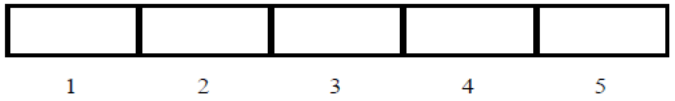


System Usability Survey (SUS) – scale 3

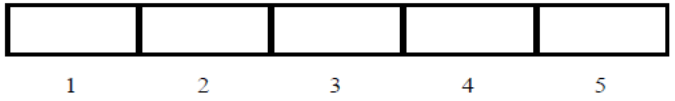
6. I thought there was too much inconsistency in this system



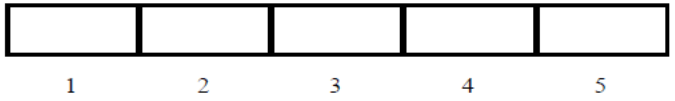
7. I would imagine that most people would learn to use this system very quickly



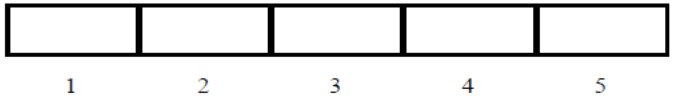
8. I found the system very cumbersome to use



9. I felt very confident using the system



10. I needed to learn a lot of things before I could get going with this system



System Usability Survey (SUS) – scoring 1

- The SUS is given to users when they have completed using the system which is being evaluated
- They score each of the 10 items by marking one of the five boxes
- The SUS yields a single number representing a composite measure of the overall usability of the system being studied. Note that scores for individual items are **not** meaningful on their own.

System Usability Survey (SUS) – scoring 2

- To calculate the SUS score, first sum the score contributions from each item. Each item's score contribution will range from 0 to 4.
- For items 1,3,5,7,and 9 (the odd numbered items) the score contribution is the scale position minus 1. For items 2,4,6,8 and 10 (the even numbered items) the contribution is 5 minus the scale position.
- Multiply the sum of the scores by 2.5 to obtain the overall score.
- SUS scores have a range of 0 to 100.
- Based on research, a SUS score above **68** would be considered above average and anything below 68 is below average.

Important!!

This is just to make it from 0-40 ↓ 0-100

Statistical testing

- You might get a user to rate the SUS of two different designs and want to know if one design is significantly better than the other.
- Similarly, you might want to know if two levels of difficulty in your game are significantly different so you get a user to rate the workload of both levels.
- To determine whether the differences in scores are significantly different we can use a statistical test

Statistical testing 2

- There are many statistical tests but I am going to show you two that will be useful for your project.
- The first is the Wilcoxon Signed Rank Test and it is ideal for analysing data from Likert and other scales e.g. the NASA TLX and SUS.
- It is used when **one user carries out two evaluations** e.g. rates the workload of your game at two different difficulty levels.
- It is a good test when you have small numbers of users – the minimum is 5; however, it's better at identifying significant differences when you have larger numbers of users.

Statistical testing 3

- Make a table where each row represents a user's scores and each column a separate evaluation score.
- I've shown the results of three users evaluating the workload of a game at two difficulty levels using the NASA TLX.
- You need a minimum of 5 and ideally more

User ID	Workload level 1	Workload level 2
U1	25	67
U2	32	56
U3	18	43

Statistical testing 4

- Enter the data into the online calculator: https://www.statology.org/wilcoxon-signed-rank-test-calculator/
- Look up the calculated **W test statistic** in the table of critical values
- To do this you need to know N, which is the number of users, and the significance level, which we will set at 0.05
- This means that if a significant difference is found then it is 95% certain that this is a real difference rather than due to randomness

Statistical testing 5

- We use an alpha value aka significance level of 0.05
- We find the row that corresponds to our number of users aka n.
- If we have 10 users then the W test statistic generated by the online calculator **needs to be less than** 8 otherwise there is no significant difference.

	Alpha value				
n	**0.005**	**0.01**	**0.025**	**0.05**	**0.10**
5	-	-	-	-	0
6	-	-	-	0	2
7	-	-	0	2	3
8	-	0	2	3	5
9	0	1	3	5	8
10	1	3	5	8	10
11	3	5	8	10	13
12	5	7	10	13	17
13	7	9	13	17	21
14	9	12	17	21	25
15	12	15	20	25	30
16	15	19	25	29	35
17	19	23	29	34	41
18	23	27	34	40	47
19	27	32	39	46	53
20	32	37	45	52	60

Statistical testing 6

- If we are comparing two sets of values generated by two different groups e.g. experienced gamers and novice gamers then we use a different test to see if they are significantly different
- This is known as the Mann-Whitney U test. There is also an online calculator and you can read about the test here:

https://www.statology.org/mann-whitney-u-test/

Reading

- Read the original paper on the NASA TLX:

  Hart, S. G., & Staveland, L. E. (1988). Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology* (Vol. 52, pp. 139-183). North-Holland.
- Read the original SUS paper
- Read more about the Wilcoxon signed rank test

Before the workshop next week

- Please review the lecture materials on the NASA TLX and SUS
- Your workshop activities will involve evaluating your games using these two techniques

HCI Evaluation 2 - Quantitative Techniques

- relating to bigger numerical data sets
- not subjective experiences

Questionnaires

- a good way to gather large amounts of data
- involve asking people questions either digitally or in person using paper
  - e.g. on a webpage or app
- can be used at scale with low resource requirments
- they generate a collection of demographic data and user opinions
- they can be used to evaluation designs and for understanding user requirements

Tips for a good questionnaire

- dont ask too many questions (helps avoid question fatigue)
  - question fatigue may lead to extreme response bias and pick one end of the scale (if using a scale for responses)
  - can help mitigate it with attention checks (invalidates that specific data point)
- avoid leading questions (”why did you have a difficult time with navigation?”)
- producing your own questionnaire is **difficult**

- advised to use existing ones which have already been validated (they actually successfulyl measure what they claim to measure)

We will be using twio questionnaires:

---

1. NASA TLX

- The **NASA Task Load Index** aims to estimate a users' percieved workload when using a system
- workload is a can refer to both cognitive or physical effort that needs to be exerted when completing a task or using a system
- The TLX was developed by Sandra Hart of NASA's human performance group and Lowell Staveland of San Jose University
- The focus is on measuring the "immediate often unvberbalised impressions that occur spontaneously" (Hart and Staveland, 1988). These are difficult or impossible to observe objectively.
- originally the NASA TLX was developed for use in aviation but since it has been used in many different domains including air traffic control, robotics, the automotive industry, healthcare, websiter design, and other tech fields
- it has had over **8000 citations** since it was introduced in 1988
- it is viewed as the **gold standard** for measuring subjective workload
- originally it was paper and pencil to gather data for it, now there are various apps or websites for it

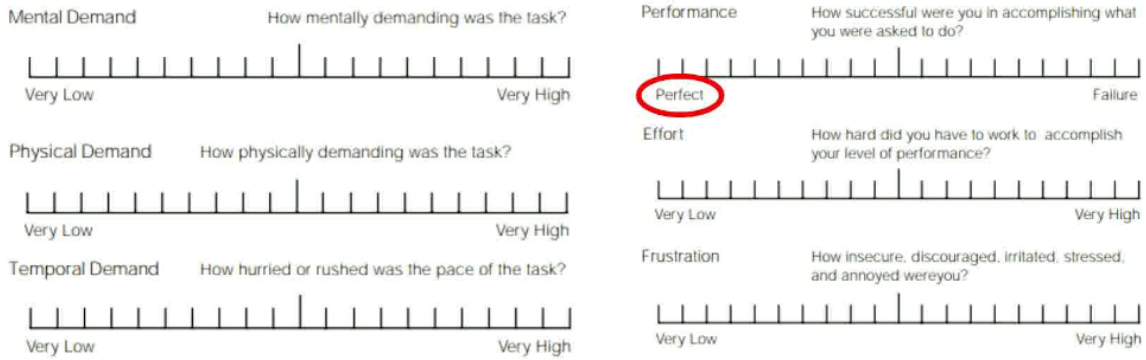
How does it measure workload?

It uses **multi-dimensional rating** procedures that derive an overall workload scored based on a *weighted average* of ratings on **six subscales:**

1. **Mental demand**

a. the cognitive and perceptual demand required to

2. **Physical demand**
    a. how much physical activity was required
3. **Temporal demand**
    a. how much time pressure did the user feel due to the rate in which the tasks occured
4. **Performance**
    a. How successfully did the user think they acomplished the task
5. E**ffort**
    a. how hard did the user have to work(mentally and physically) to accomplish their level of performance
6. **Frustration**
    a. how insecure, discoruaged, or irritated did the user feel in the task



- Measured using a *visual analogue scale*
- the default is to mark the tick to the **right**

Scoring the NASA TLX

- users answer the TLX immediately after they have completed a task, this is necessary as asking them to complete it during the task is not possible. However, one drawback is that users may forget or misrepresent detaisl of their percieved workload
- The questionnaire is scored in a two-step process:
    1. Identifying the relative importance of the 6 dimensions on a users' percieved workload
    2. Rating each of the 6 dimensions on a scale

(nowadays most people dont bother with the weighting step, only doing step 2 is more common)

Finding the Relative Weightings of the Dimensions

- a user reflects on the taks theyve been asked to perform and is shown each paried combination of the six dimensions to decide which is more related to their personal definition of workload relating to the task
- This means a user considers **15 paired comparisons** for example: they need to decide whether Performance **or** Frustration "represent the more important contributor to the workload for the specific task you recently performed"
- each time a dimension is selected as more important it **recieves a score of 1**
- the total score is the weight of the dimension and **ranges from 0 - 5**
- the sum of the weights should be **15**

T**he relative weighting of the six dimensions is often not measured or used**

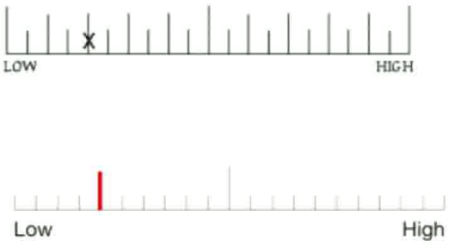
- not measuring the relative weighting makes the NASA TLX **simpler to administrator**
- several studies have compared the raw TLX scores to weighted TLX scores and have found mixed results (some showing better sensitivity when removing weights, others less, others no difference)

- when the dimensions are not weighted the method is know as the **Raw TLX Score**

Rating the Dimensions

- users mark their score on each of the 6 dimensions (subscales)
- each dimension consists of a line with 21 equally spaced tick marks which divide the line from **0 — 100 in increments of 5**
    - if a user marks *between the ticks* then the value of the **right side of the tick is used**
- the score on a dimension is calculated as the tick number, subtract 1, multiplied by 5 which gives the final score out of 100
    - e.g. if they marked the 16th tick if would be (16 - 1) x 5 = 75

- For example, the images show the rating on a paper questionnaire (top) and on a mobile app (bottom)
- The fifth tick mark is selected, so the rating score is: (5 – 1) * 5 = 20



What do the scores tell us?

If the weights are used:

- individual ratings on each of the dimensions are multiplied by their respective weights, summed and divided by 15, resulting in an aggregate percieved workload score for a task ranging from 0 — 100

If the weights are not used:

- individual ratings on each dimension can be **summed and divided by 6,** resulting in aggregate percieved workload score ranging from 0 — 100

The individual ratings on the 6 dimensions also give some insight into where the workload is coming from, this can be helpful for developmers hoping to *improve their design*

NASA TLX Validity

- Hart and Staveland validated that the sub-scales measured different sources of workload
- Subsequent independant studies have also found that the NASA TLX is a valid measure of subjective workload *(Rubio et al, 2004; Xiao et al, 2005)*.

---

2. System Usability Scale (SUS)

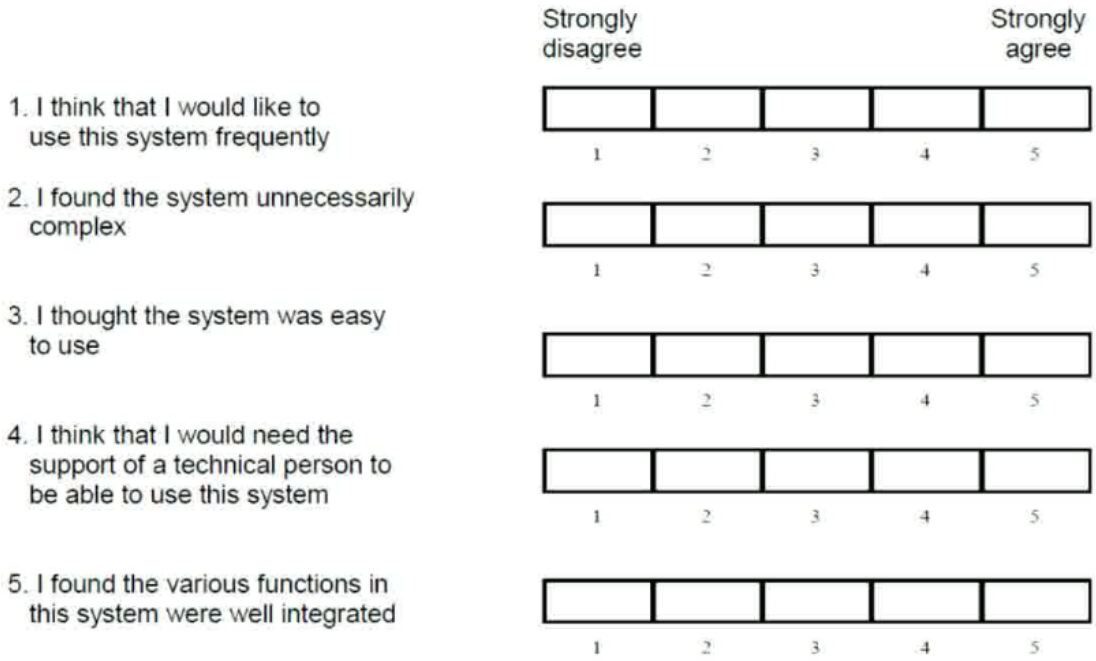
This provides a “quick and dirty” reliable tool for measuring usability

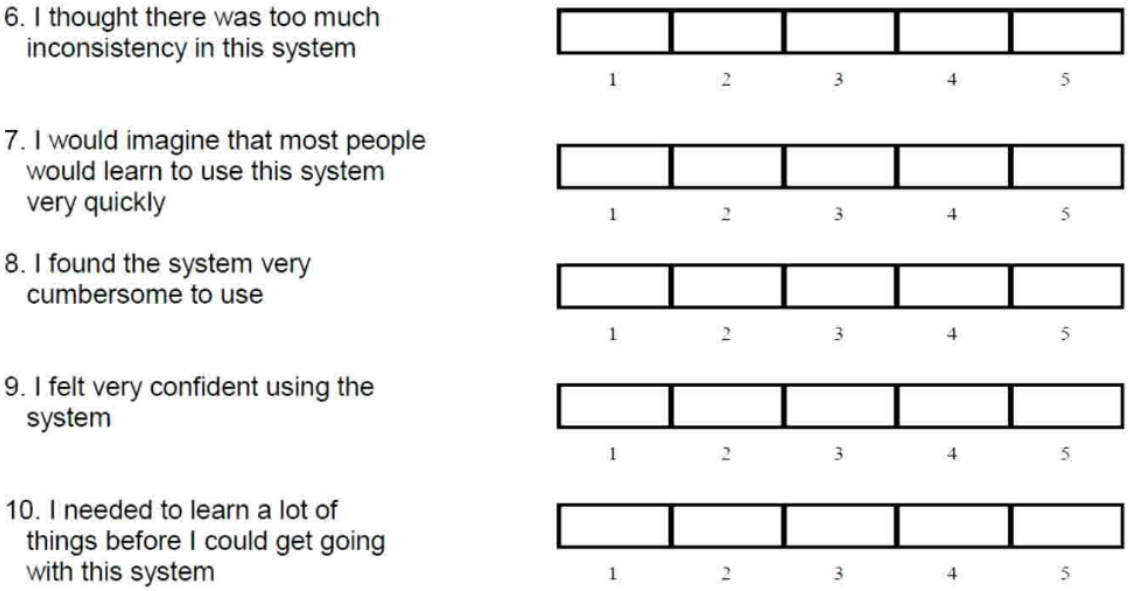
- Created by John Brooke in 1986
- Consists of **10 questions** with 5 response options for each question (likert scale ranging from strongly disagree to strongly agree)
- It enables the evaluation of a wide variety of products and services includiojng hardware, software, mobile devices, websites and applications
- use the SUS to geta quick usability assessment of your system

Benefits of the SUS

- it has become the industry standard, cited over 1300 times
- it is very easy to administer to participants
- can be used on small sample sices with reliabel results
- has been validated and been shown to effectively differentiate between useable and unuseable systems

The Scale





- **odd** numbered questions designed so that if you **agree** with them then the system is more useabel
- the **even** numbered questions are designed so that if you **disagree** with them then the system is more useable

Once responses have been gather, you must come up with an **overall useability score**

- scores for individual items are **not meaningful on their own** unlike in the NASA TLX
- it is meant for an overall score

So how do you come up with the final overall score?

1. sum the score contributions from each item. Each item's score contribution will range from

   0 — 4

2. then do the score for the odd numbered items (1, 3, 5, 7, 9) the score contrubution is the scale position minus 1. For even numbers the contribution

is 5 minus the scale position

3. Multiply the sum of the scores by **2.5** to obtain the overall score

- SUS scores have a range of **0 — 100**
- based on research, a SUS score above **68** would be considered above average and anything below 68 is *below average*

Statistical Testing

- you may ask a user to rate the TLX on different difficulty levels
- and you want to find out a significnt difference in the percieved workload between the levels
- or could give the same user two different systems and asking them to rate them on the SUS, and find out if one system has significantly better usability

Wilconxon Signed Rank Test

- ideal for analysing Likert data and other scales e.g. TLX and SUS
- it is used when **one user carries out two evaluations**
- good test when you have low number of users (minimum is 5)
    - however it is better at identifying significant differences when you have largwer numbers of users

- Make a table where each row represents a user's scores and each column a separate evaluation score.
- I've shown the results of three users evaluating the workload of a game at two difficulty levels using the NASA TLX.
- You need a minimum of 5 and ideally more

User ID	Workload level 1	Workload level 2
U1	25	67
U2	32	56
U3	18	43

- workload is the score on the TLX

To determine if there is a significant difference we need to know *N* and and alpha level will be set to 0.05

- if significance is acheived, there is a 95% chance that the result is due to actual differences and not random chance

- We use an alpha value aka significance level of 0.05
- We find the row that corresponds to our number of users aka n.
- If we have 10 users then the W test statistic generated by the online calculator **needs to be less than** 8 otherwise there is no significant difference.

	Alpha value				
n	0.005	0.01	0.025	0.05	0.10
5	-	-	-	-	0
6	-	-	-	0	2
7	-	-	0	2	3
8	-	0	2	3	5
9	0	1	3	5	8
10	1	3	5	8	10
11	3	5	8	10	13
12	5	7	10	13	17
13	7	9	13	17	21
14	9	12	17	21	25
15	12	15	20	25	30
16	15	19	25	29	35
17	19	23	29	34	41
18	23	27	34	40	47
19	27	32	39	46	53
20	32	37	45	52	60

Comparing two different groups

- e.g. a group of experience vs non experienced gamers

Use a **Mann-Whitney U Test**

- e.g. get both types of gamers to play the harder level, and see if there is a statistically significant different between percieved workload

Software Quality

Lecture 8

Ruzanna Chitchyan, Jon Bird, Pete Bennett
TAs: Alex Elwood, Alex Cockrean, Casper Wang

Overview

- Software quality and how to get to it
- Test-driven development
  - White box testing
  - Black box testing

Software Quality

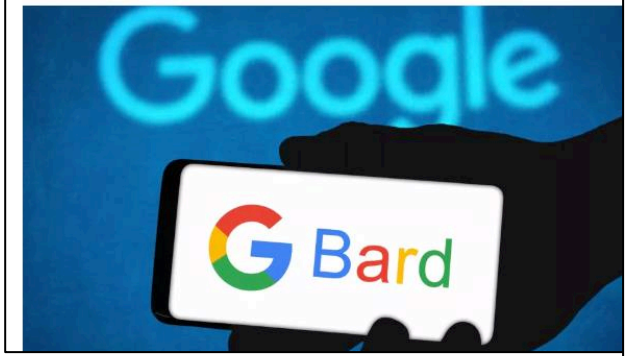
Why is Software Quality relevant: Case of Bard (Gemini)

Google Bard AI mistake just cost Google over $100 billion

By Philip Michaels last updated 8 days ago

AI-powered chatbot makes costly error in early demo

Comments (0)



https://www.tomsguide.com/news/google-bard-ai-is-off-to-an-embarrassing-start

Isabel Angelo
@IsabelNAngelo · **Follow**

Unfortunately a simple google search would tell us that JWST actually did not "take the very first picture of a planet outside of our own solar system" and this is literally in the ad for Bard so I wouldn't trust it yet

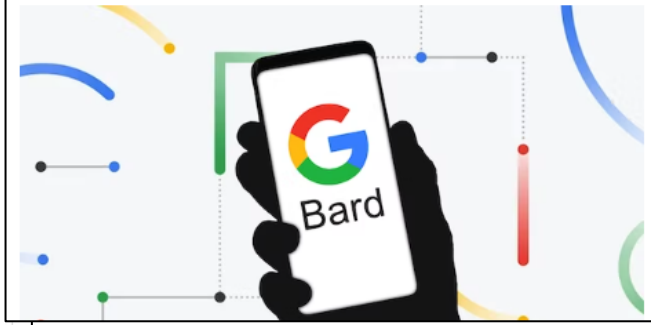
Google @Google

Bard is an experimental conversational AI service, powered by LaMDA. Built using our large language models and drawing on information from the web, it's a launchpad for curiosity and can help simplify complex topics → goo.gle/3HBZQtu

Introducing Bard,
an experimental conversational AI service
powered by LaMDA

You can use Bard to —
Plan a friend's baby shower
Compare two Oscar nominated
Get lunch ideas based on what's in your fridge

The chatbot generated an incorrect fact about the James Webb Space Telescope in its very first public demo. The incident has dramatically highlighted one of the most pertinent dangers for marketers using AI: it doesn't always tell the truth.



https://www.thedrum.com/news/2023/02/09/attention-marketers-google-s-100bn-bard-blunder-underscores-current-dangers-using-ai

Why is Software Quality relevant?

- Reputation
- Cost of Product and Maintenance
- Software Certification
- Organizational Certification
- Legality
- Moral/ethical codes of practice

Software Quality is Multi-dimensional

- Subjective or “fitness for use”: as perceived by an individual user (e.g., aesthetics of GUI, missing functionality…)
- Objective or “conformance to requirements”: can be measured as a property of the product (e.g., detailed documentation, number of bugs, compliance with regulations ….)
- Practical: what does it mean to your team and your clients?

Quality Models: ISO/IES25010

- **Functional Suitability**
  - Functional Completeness
  - Functional Correctness
  - Functional Appropriateness
- **Performance Efficiency**
  - Time Behaviour
  - Resource Utilisation
  - Capacity
- **Compatibility**
  - Co-existence
  - Interoperability
- **Usability**
  - Appropriateness
  - Realisability
  - Learnability
  - Operability
  - User Error Protection
  - User Interface Aesthetics
  - Accessibility
- **Reliability**
  - Maturity
  - Availability
  - Fault Tolerance
  - Recoverability
- **Security**
  - Confidentiality
  - Integrity
  - Non-repudiation
  - Authenticity
  - Accountability
- **Maintainability**
  - Modularity
  - Reusability
  - Analysability
  - Modifiability
  - Testability
- **Portability**
  - Adaptability
  - Installability
  - Replaceability

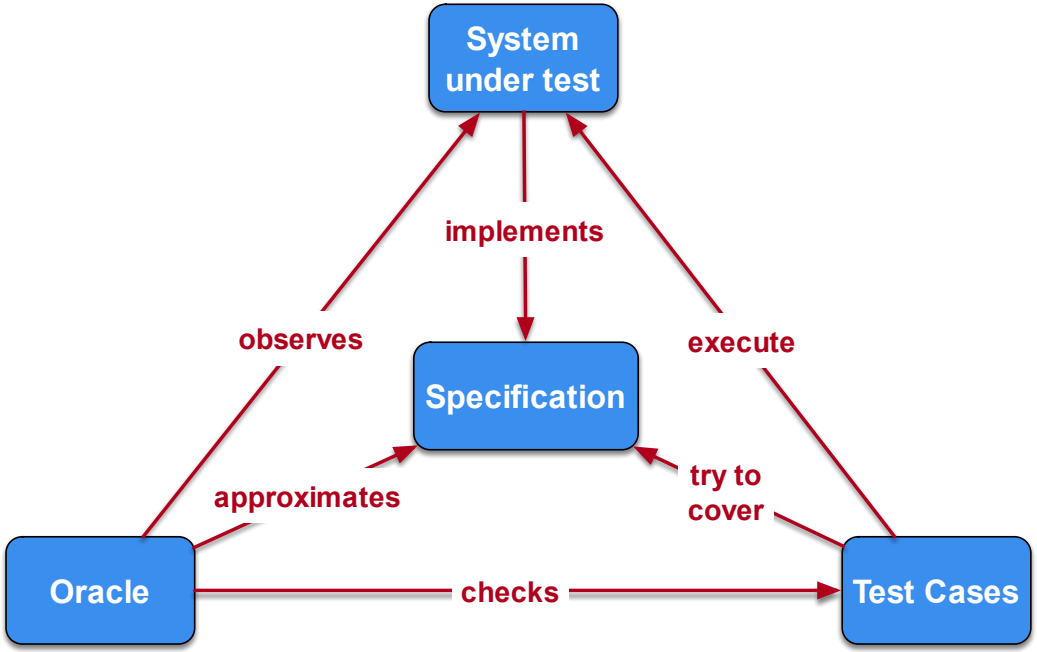
Steps Towards Software Quality:

- Use a standard development process
- Use a coding standard
    - Compliance with industry standards (e.g., ISO, Safety, etc.)
    - Consistent code quality
    - Secure from start
    - Reduce development costs and accelerate time to market
- Define and monitor metrics (defect metrics and complexity metrics)
    - High complexity leads to higher number of defects
- Identify and remove defects
    - Conduct manual reviews
    - Use Testing

Testing

Mauro Pezze and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

Testing process: key elements and relationships



From: M. Staats, M. W. Whalen, and M. P. E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 391–400. IEEE, 2011.

Testing: White Box

Mauro Pezze and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

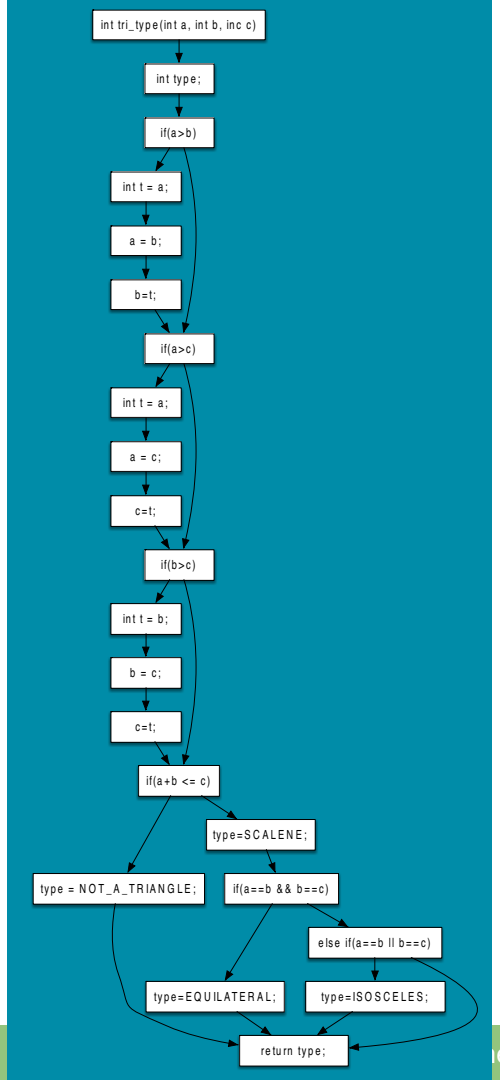
White Box Testing

- Access to software ``internals'':
  - Source code
  - Runtime state
  - Can keep track of executions.
- White box testing exploits this to
  - Use code to measure coverage
    - Many different ways
  - Drive generation of tests that maximise coverage

```
1   int tri_type(int a, int b, int c) {
2     int type;
3     if (a > b)
4       { int t = a; a = b; b = t; }
5     if (a > c)
6       { int t = a; a = c; c = t; }
7     if (b > c)
8       { int t = b; b = c; c = t; }
9     if (a + b <= c)
10      type = NOT_A_TRIANGLE;
11    else {
12      type = SCALENE;
13      if (a == b && b == c)
14        type = EQUILATERAL;
15      else if (a == b || b == c)
16        type = ISOSCELES;
17    }
18  return type;
19  }
```

White Box Testing

- Access to software ``internals'':
  - Source code
  - Runtime state
  - Can keep track of executions.
- White box testing exploits this to
  - Use code to measure coverage
    - Many different ways
  - Drive generation of tests that maximise coverage.



White-Box Testing

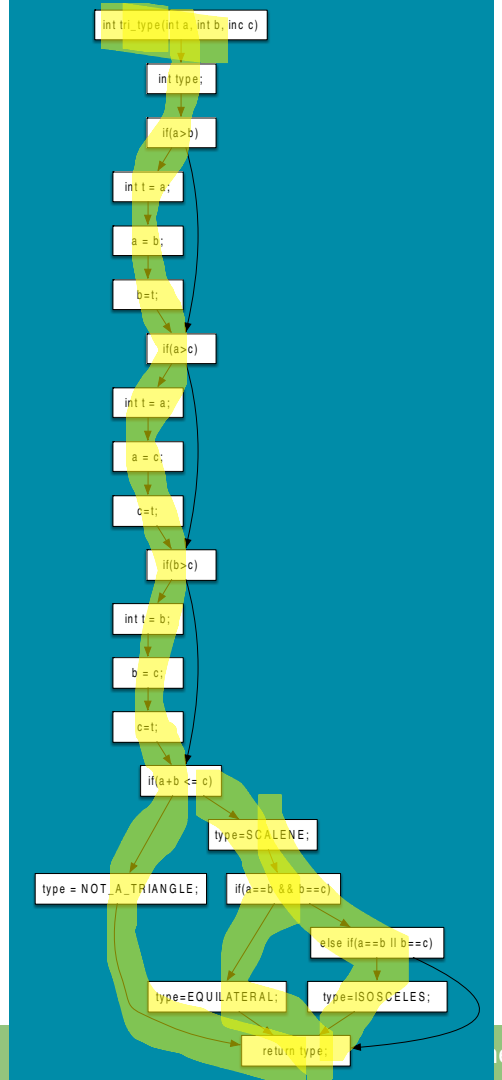
- Coverage Metrics:
  - Statement coverage
  - Branch coverage
  - Def-Use or Dataflow coverage
  - MC/DC (Modified Condition / Decision Coverage)
  - Mutation coverage…

- Prescribed metrics, e.g., DO178-B/C standard for civilian aircraft software
  - non-critical - statement coverage
  - safety-critical - MC/DC coverage

Statement Coverage

- Test inputs should collectively have executed each statement
- If a statement always exhibits a fault when executed, it will be detected
- Computed as:

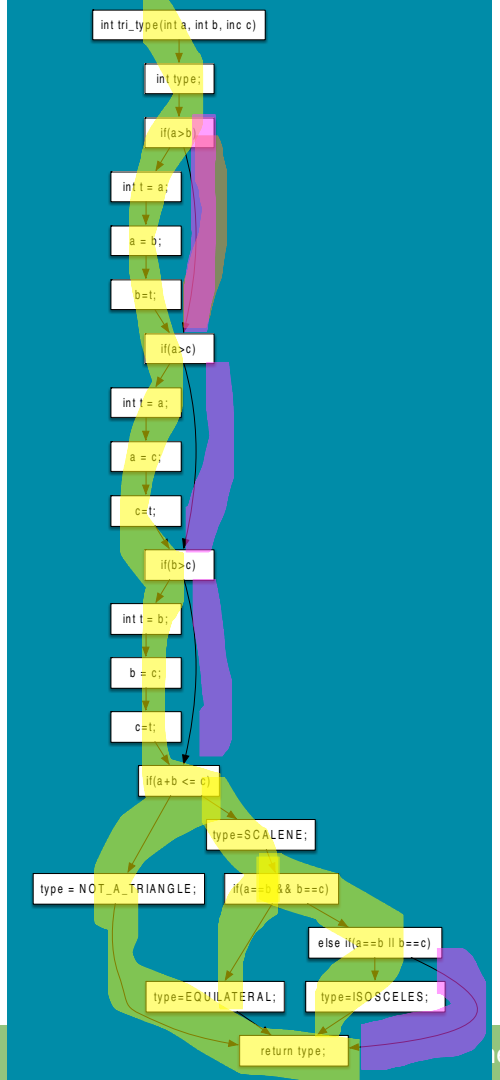
$$Coverage = \frac{|Statements\ executed|}{|Total\ statements|}$$



Branch Coverage

- Test inputs should collectively have executed each branch
- Subsumes statement coverage
- Computed as:

$$Coverage = \frac{|Branches\ executed|}{|Total\ branches|}$$

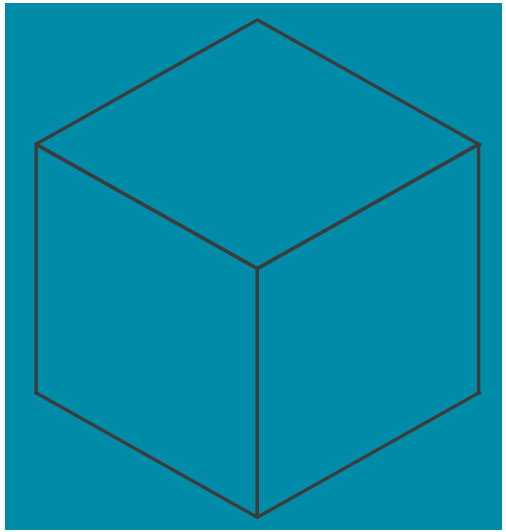


Testing: Black Box

Mauro Pezze and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

Black Box Testing

- No access to “internals”
  - May have access, but don’t want to
- We know the interface
  - Parameters
  - Possible functions / methods
- We may have some form of specification document



Testing Challenges

- Many different types of input
- Lots of different ways in which input choices can affect output
- An almost infinite number of possible inputs & combinations

Equivalence Partitioning (EP) Method

Identify tests by analysing the program interface

1. Decompose program into “functional units”
2. Identify inputs / parameters for these units
3. For each input

    a) Identify its limits and characteristics

    b) Define “partitions” - value categories

    c) Identify constraints between categories

    d) Write test specification

Example – Generate Grading Component

The component is passed an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it generates a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark which is calculated as the sum of the exam and c/w marks, as follows:
greater than or equal to 70 - 'A'
greater than or equal to 50, but less than 70 - 'B'
greater than or equal to 30, but less than 50 - 'C'
less than 30 - 'D'
Where a mark is outside its expected range then a fault message ('FM') is generated. All inputs are passed as integers.

EP – 1. Decompose into Functional Units

- Dividing into smaller units is good practice
  - Possible to generate more rigorous test cases.
  - Easier to debug if faults are found.
- E.g.: dividing a large Java application into its core modules / packages
- Already a functional unit for the Grading Component example

EP – 2. Identify Inputs and Outputs

- For some systems this is straightforward
  - E.g., the Triangle program:
    - Input: 3 numbers,
    - Output: 1 String
  - E.g., Grading Component
    - Input: 2 integers: exam mark and coursework mark
    - Output: 1 String for grade
- For others less so. Consider the following:
  - A phone app.
  - A web-page with a flash component.

EP – 3.a Identify Categories

Category	Description
Valid	valid exam mark
	valid coursework mark
	valid total mark
Invalid	invalid exam mark
	invalid coursework mark
	Invalid total mark

EP: 3.b Define “Partitions” - value categories

- Significant value ranges / value-characteristics of an input

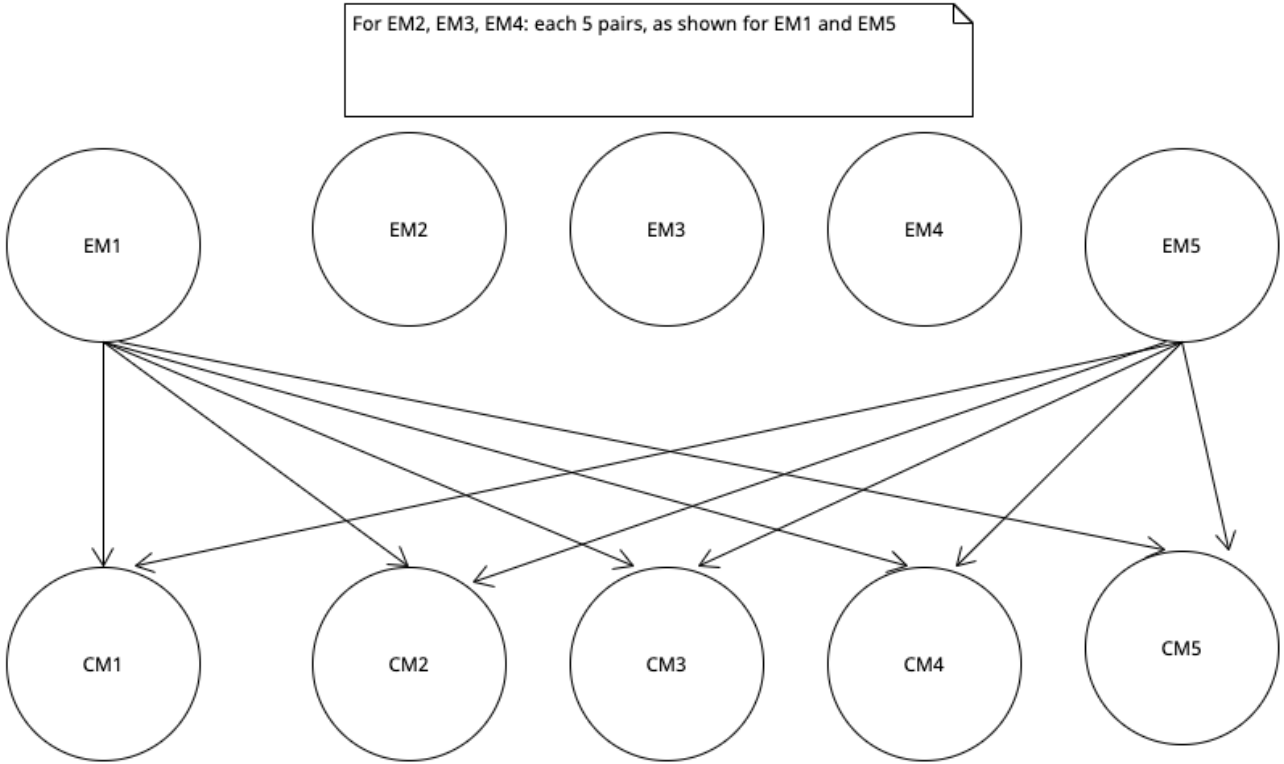
Category	Description	Partition
Valid	EM_1 valid exam mark	0 ≤ Exam mark ≤ 75
	CM_1 valid coursework mark	0 ≤ Coursework mark ≤ 25
Invalid	EM_2 invalid exam mark	Exam mark > 75
	EM_3 invalid exam mark	Exam mark < 0
	EM_4 invalid exam mark	alphabetic
	EM_5 invalid exam mark	Other real number (outside of EM_1)
	CM_2 invalid coursework mark	Coursework mark > 25
	CM_3 invalid coursework mark	Coursework mark < 0
	CM_4invalid coursework mark	alphabetic
	CM_5 invalid coursework mark	Other real number (outside of CM_1)

EP – 3. c Identify Constraints between Categories

- Not all categories can combine with each other

Category		Condition
valid exam mark	EM_1	0 ≤ Exam mark ≤ 75
invalid exam mark	EM_2	Exam mark > 75
invalid exam mark	EM_3	Exam mark < 0
invalid exam mark	EM_4	alphabetic
invalid exam mark	EM_5	Other real number
valid coursework mark	CM_1	0 ≤ Coursework mark ≤ 25
invalid coursework mark	CM_2	Coursework mark > 25
invalid coursework mark	CM_3	Coursework mark < 0
invalid coursework mark	CM_4	alphabetic
invalid coursework mark	CM_5	Other real number

EP – 3. d Write Test Specifications



Example: Inputs and Expected Outputs

The test cases corresponding to partitions derived from the input exam mark are:

Test Case	1	2	3
Input (exam mark)	44	-10	93
Input (c/w mark)	15	15	15
total mark (as calculated)	59	5	108
Partition tested (of exam mark)	$0 \leq e \leq 75$	$e < 0$	$e > 75$
Exp. Output	'B'	'FM'	'FM'

Boundary Values

- Most frequently errors occur in "edge" cases
  - Test just under boundary value
  - Test just above the boundary value
  - Test the boundary value

How do we go about using this?

- Testing applied in Java: Use JUnit
  - uses “Assertions” to test the code
  - Allow us to state what *should* be the case
  - If assertions do not hold, JUnit’s logging mechanisms reports failures
  - Various types of assertion are available, e.g., assertEquals(expected, actual); assertTrue(condition); assertFalse(condition); assertThat (value, matchingFunction)

Review

- What is Software Quality?
- What are key elements and relationships for test specifications?
- How do we carry out white-box testing?
- How do we carry out black-box testing?

Equivalence Partitioning Method is also known as Equivalence class partitioning (ECP). It is a software testing technique or black-box testing that divides input domain into classes of data, and with the help of these classes of data, test cases can be derived. An ideal test case identifies class of error that might require many arbitrary test cases to be executed before general error is observed.
In equivalence partitioning, equivalence classes are evaluated for given input conditions. Whenever any input is given, then type of input condition is checked, then for this input conditions, Equivalence class represents or describes set of valid or invalid states.

Guidelines for Equivalence Partitioning :

- If the range condition is given as an input, then one valid and two invalid equivalence classes are defined.
- If a specific value is given as input, then one valid and two invalid equivalence classes are defined.
- If a member of set is given as an input, then one valid and one invalid equivalence class is defined.
- If Boolean no. is given as an input condition, then one valid and one invalid equivalence class is defined.

1. Identify the input field: Determine the input field to be tested, such as the username field in our example.
2. Define equivalence classes: Categorize the possible input values into distinct equivalence classes. Each class represents a set of inputs with similar behavior or characteristics.
3. Determine representative values: Select representative values from each equivalence class. These values should cover the boundary conditions and critical scenarios within each class.
4. Create test cases: Generate test cases that cover each equivalence class. For example, one test case would include a valid username, another would include an invalid username, and a third test case would cover an empty username.
5. Execute the test cases: Run the test cases using the defined equivalence classes to validate the behaviour of the system. Observe and record the results for each test case.

Software Quality

- quality of software is very important
    - many examples of software lacking quality checks costing companies lots of money (Bard's mistake cost Google $100 billion stock value)

Why is software quality relevant?

- **reputation of the software and the company/individual producing it** - no one will want to use it or buy from you
- **cost of the product and maintenance** - Y2K example, misuse of date (see slides)
- **software certification** - People want to
- **Organisational certification** - Assuring an organisation adheres to set standards and practices, if people are qualified to a certain standard then people are more likely to trust you, you are more able to get government contracts
- **Legality** - gambiling games need to adhere to gampling laws for example
- **Moral/ethical codes of practise -**

Software Quality is Multi-dimensional

Software quality can be measured:

- **Subjective/fitness for use** - user perception or opinioons on aesthetics, missing functionality etc, does it do what the users want it to do?
- **Objective/conformity to requirements** - measured as property of the product i.e. doesit have detailed certification? how many bugs does it contain? does it comple with regulations?

- **Practically** - what does it mean to your team and your clients?

Models of Software Quality: ISO/IES25010

- these relate to many different ways to assess quality

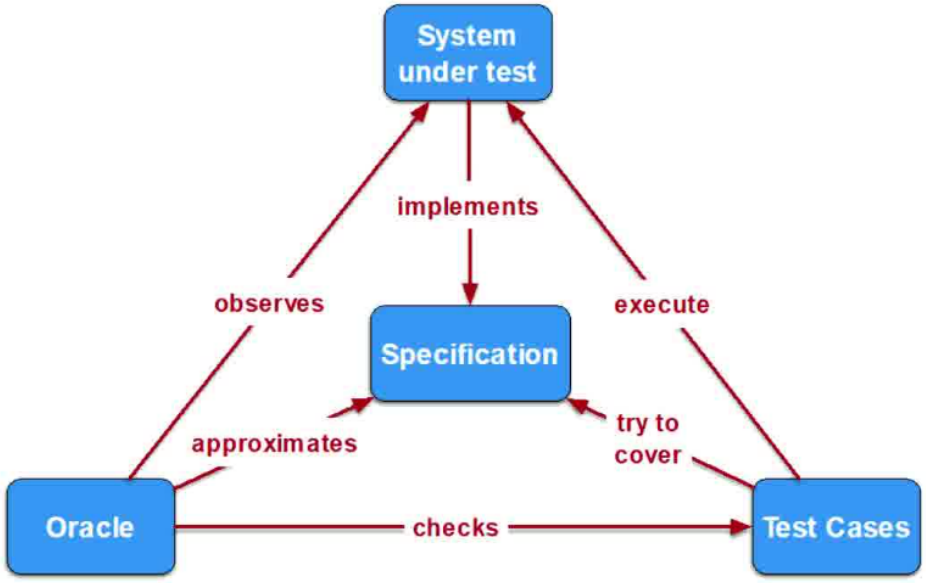
Steps Towards Software Quality:

- use a standard development process - e.g. Agile Development
- use a **coding standard**
  - compliance with industry standards (e.g. ISO, Safety, etc)
    - sensible variable names, good commenting etc etc
  - Consistent code quality

    - Secure from the start -
    - Reduce development costs and accelerate time to market
- Define and monitor metrics - decide on metrics to measure defects and metrics to define complexity
    - high complexity leads to higher number of defects
- Identify and remove defects
    - conduct manual reviews
    - Test thouroughly

Testing

Testing Process: Key Elements and Relationships



- Oracle is the standards in the testing process which define what a test needs to do to pass or fail

White Box Testing

- the type of testing you can perform when you have fill access to the source code
  - Runtime state
  - Can keep track of execution of this source code to determine if it is behaving as expected

White box testing exploites this to:

- use code to measure coverage - to tests cover all source code?
- you can then make tests to try and fully cover all the source code

Coverage Metrics

Coverage means number of tests covering lines of code, e.g. if every line of code has been tested then there is full coverage

- statement coverage - have you executed and tested all the statements
- branch coverage - have you executed and tested all the branches

Statement Coverage

- you should test each statement execution
- if a statement always exhibits a fault upon execution, it will be detected
- it is calculated it by:

$$Coverage = \frac{StatementsExecuted}{TotalStatements}$$

Branch Coverage

- tests should have collectively executed each branch/condition
- ensures all conditionals are checked at least once

- Calculated by

$$Coverage = \frac{BranchesExecuted}{TotalBranches}$$

Black Box Testing

When you cant see the source code

- e.g. third party libraries

Black box testing tests the interfaces with no access to internals

- we know the parameters and possible functions
- might have access to a specification document

Testing Challenges

- many different types of input
- lots of different ways in which input choices can affect output
- an almost infinite number of possible inputs and combinations of inputs

Equivalence Partitioning (EP) Method

Equivalence Partitioning is a testing technique used in the field of software development, particularly within software testing, to reduce the number of test cases to a manageable level while still covering the maximum amount of scenarios. This technique involves dividing input data of a software unit into partitions of equivalent data from which test cases can be derived. The fundamental principle behind equivalence partitioning is that it assumes how the

program will handle a certain input value is representative of how it will handle all values in that input class. This approach can significantly reduce the total number of test cases that need to be developed.

Key Concepts

- **Equivalence Class**: This refers to a set of valid or invalid states for input conditions. An equivalence class represents a set of valid or invalid inputs that are expected to be treated similarly by the system. If one condition in an equivalence class passes the test, the entire class is considered to have passed. Conversely, if one condition in the class fails, then the whole class is considered to have failed.
- **Partitioning**: The process of dividing or grouping the input data and output results into classes. One can partition the data into 'valid' and 'invalid' classes.

Steps in Equivalence Partitioning

1. **Identify Test Data**: Look at the specifications and requirements to identify the test data.
2. **Divide the Data**: Based on the test data, divide the input and output data into equivalence data classes. One should create classes for input data that is expected to give similar results. These are often based on the requirements and specifications of the system under test.
3. **Select Test Cases**: From each partition, select test cases. Ideally, you want to select a test case that represents the entire partition, meaning if it passes, the whole partition would theoretically pass.
4. **Execute Test Cases**: Run your test cases to validate that the software behaves as expected for the representative set of test inputs.

Advantages

- **Reduces Testing Time**: By minimizing the number of test cases, equivalence partitioning can significantly reduce testing time.
- **Efficient Test Coverage**: It ensures coverage of all possible input scenarios by testing only representative values from each partition.

- **Simplifies Testing Process**: Helps in simplifying the testing process by categorizing data and focusing on key input scenarios.

Applications

Equivalence partitioning is broadly applicable in the testing phases of software development, especially useful in functional testing, system testing, and acceptance testing. It's particularly beneficial when the range of input values for the software is vast and testing each one individually would be impractical.

By identifying equivalence classes, testers can create a more efficient and effective testing strategy, ensuring that the application is thoroughly tested for a wide range of inputs with a reduced set of test cases.

Steps in Equivalence Partitioning: (NEED MORE NOTES ON THIS BEFORE EXAM)

We will use an example of software such as Blackboard keeping track of valid exam scores and then then using them as inputs to calculate overall coursework mark

1. Decompose into functional units

- dividing into smaller units is good practise
    - allows for more rigorours test cases
    - easier to debug if faults are found
        - for example, dividing a large Java application inot its core modules/packades

2. Identify Inputs and Outputs

- for some systems this is straightforward
- some systems very complicated

- e.g. phone applications
- web page with flash component

3a. Identify Categories

Category	Description
Valid	valid exam mark
	valid coursework mark
	valid total mark
Invalid	invalid exam mark
	invalid coursework mark
	Invalid total mark

- e.g. valid and invalid inputs
- valid and invalid outputs

3b. Define Partitions - value categories

- Significant value ranges / value-characteristics of an input

Category	Description	Partition
Valid	EM_1 valid exam mark	0 ≤ Exam mark ≤ 75
	CM_1 valid coursework mark	0 ≤ Coursework mark ≤ 25
Invalid	EM_2 invalid exam mark	Exam mark > 75
	EM_3 invalid exam mark	Exam mark < 0
	EM_4 invalid exam mark	alphabetic
	EM_5 invalid exam mark	Other real number (outside of EM_1)
	CM_2 invalid coursework mark	Coursework mark > 25
	CM_3 invalid coursework mark	Coursework mark < 0
	CM_4invalid coursework mark	alphabetic
	CM_5 invalid coursework mark	Other real number (outside of CM_1)

Overview of Software Engineering COMSM

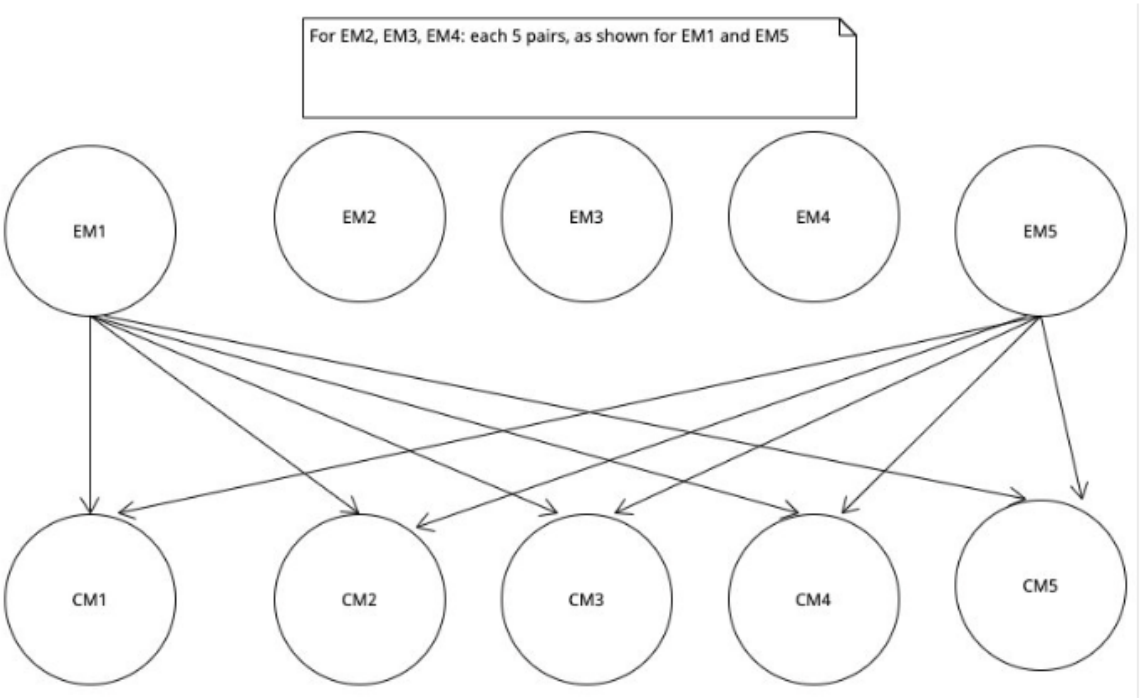
3c. Identify Constraints between Categories

- Not all categories can combine with each other

Category		Condition
valid exam mark	EM_1	0 ≤ Exam mark ≤ 75
invalid exam mark	EM_2	Exam mark > 75
invalid exam mark	EM_3	Exam mark < 0
invalid exam mark	EM_4	alphabetic
invalid exam mark	EM_5	Other real number
valid coursework mark	CM_1	0 ≤ Coursework mark ≤ 25
invalid coursework mark	CM_2	Coursework mark > 25
invalid coursework mark	CM_3	Coursework mark < 0
invalid coursework mark	CM_4	alphabetic
invalid coursework mark	CM_5	Other real number

- you ask yourself which of these valid an dinvalid inputs can do together - for test case design
  - e.g. you cant have ivali

3d. Write Test Specifications



Example: Inputs and Expected Outputs

The test cases corresponding to partitions derived from the input exam mark are:

Test Case	1	2	3
Input (exam mark)	44	-10	93
Input (c/w mark)	15	15	15
total mark (as calculated)	59	5	108
Partition tested (of exam mark)	$0 \leq e \leq 75$	$e < 0$	$e > 75$
Exp. Output	'B'	'FM'	'FM'

Boundary Values

- the most frequent errors are found in “edge cases”
  - test just under and just above the boundary values
    - e.g. under 0 or over 100 for the min and max values
  - test the boundary itself

How do we go about doing this?

- Testing applied in Java: use **JUnit**
  - uses assertions
  - allows us to state what should be the case
  - if assertions do not hold, JUnit will log failures
  - Various types of assertions are available