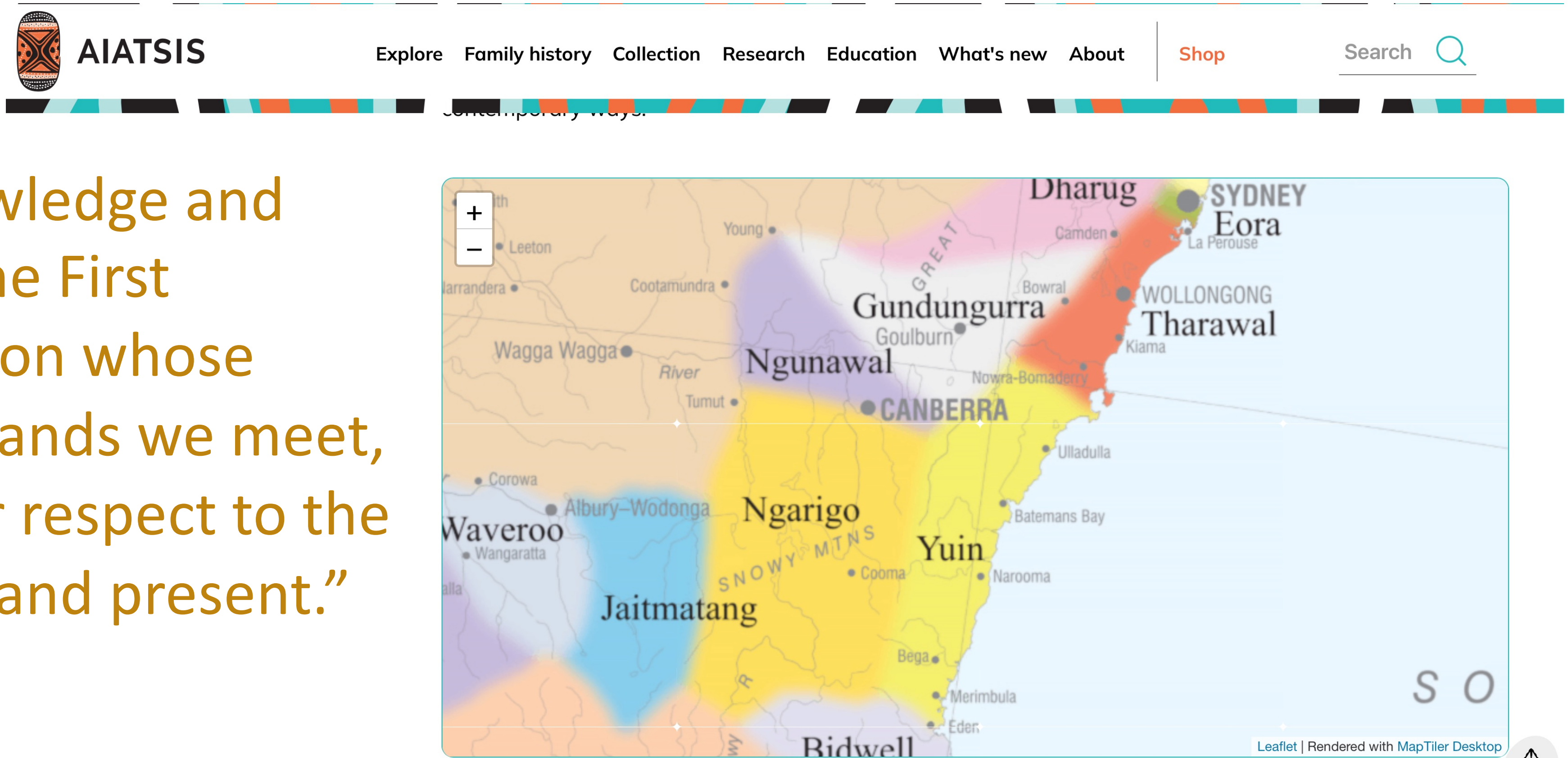
Week:
4 of 12

COMP 2120 / COMP 6120

INSPECTION

A/Prof Alex Potanin and Dr Melina Vidoni

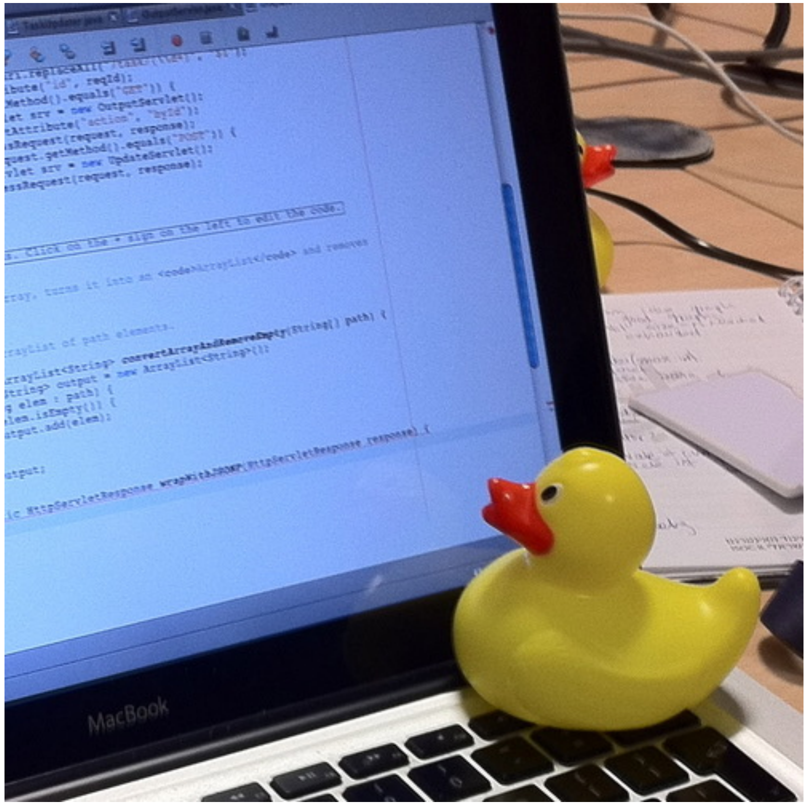
ANU Acknowledgment of Country



“We acknowledge and celebrate the First Australians on whose traditional lands we meet, and pay our respect to the elders past and present.”

https://aiatsis.gov.au/explore/map-indigenous-australia

Rubber Duck Debugging



REVIEWS AND INSPECTIONS

Reviews and inspections

- A group examines part or all of a process or system and its documentation to find potential problems.
- Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.
- There are different types of review with different objectives
  - Inspections for defect removal (product);
  - Reviews for progress assessment (product and process);
  - Quality reviews (product and standards).

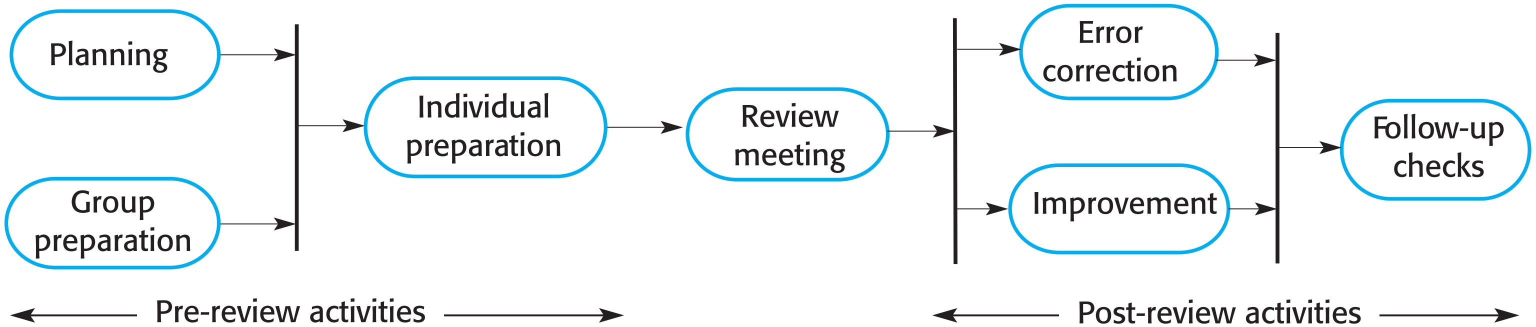
Quality reviews

- A group of people carefully examine part or all of a software system and its associated documentation.
- Code, designs, specifications, test plans, standards, etc. can all be reviewed.
- Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.

Phases in the review process

- Pre-review activities
  - Pre-review activities are concerned with review planning and review preparation
- The review meeting
  - During the review meeting, an author of the document or program being reviewed should ‘walk through’ the document with the review team.
- Post-review activities
  - These address the problems and issues that have been raised during the review meeting.

The software review process



Distributed reviews

- The processes suggested for reviews assume that the review team has a face-to-face meeting to discuss the software or documents that they are reviewing.
- However, project teams are now often distributed, sometimes across countries or continents, so it is impractical for team members to meet face to face.
- Remote reviewing can be supported using shared documents where each review team member can annotate the document with their comments.

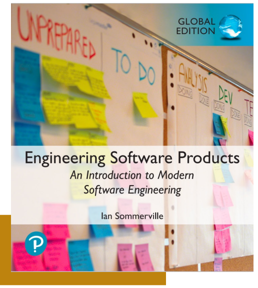
Program inspections

- These are peer reviews where engineers examine the source of a system with the aim of discovering anomalies and defects.
- Inspections do not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

Inspection checklists

- Checklist of common errors should be used to drive the inspection.
- Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- In general, the 'weaker' the type checking, the larger the checklist.
- Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

An inspection checklist (a)



Fault class	Inspection check
Data faults	• Are all program variables initialized before their values are used? • Have all constants been named? • Should the upper bound of arrays be equal to the size of the array or Size -1? • If character strings are used, is a delimiter explicitly assigned? • Is there any possibility of buffer overflow?
Control faults	• For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for? • If a break is required after each case in case statements, has it been included?
Input/output faults	• Are all input variables used? • Are all output variables assigned a value before they are output? • Can unexpected inputs cause corruption?

An inspection checklist (b)

Fault class	Inspection check
Interface faults	• Do all function and method calls have the correct number of parameters? • Do formal and actual parameter types match? • Are the parameters in the right order? • If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	• If a linked structure is modified, have all links been correctly reassigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly deallocated after it is no longer required?
Exception management faults	• Have all possible error conditions been taken into account?

QUALITY MANAGEMENT AND AGILE DEVELOPMENT

Quality management and agile development

- Quality management in agile development is informal rather than document-based.
- It relies on establishing a quality culture, where all team members feel responsible for software quality and take actions to ensure that quality is maintained.
- The agile community is fundamentally opposed to what it sees as the bureaucratic overheads of standards-based approaches and quality processes as embodied in ISO 9001.

Shared good practice

- *Check before check-in*
  - Programmers are responsible for organizing their own code reviews with other team members before the code is checked in to the build system.
- *Never break the build*
  - Team members should not check in code that causes the system to fail. Developers have to test their code changes against the whole system and be confident that these work as expected.
- *Fix problems when you see them*
  - If a programmer discovers problems or obscurities in code developed by someone else, they can fix these directly rather than referring them back to the original developer.

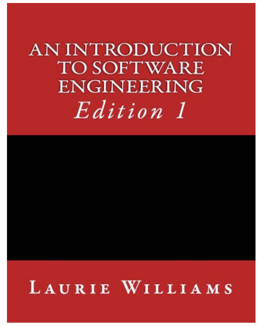
Reviews and agile methods

- The review process in agile software development is usually informal.
- In Scrum, there is a review meeting after each iteration of the software has been completed (a sprint review), where quality issues and problems may be discussed.
- In Extreme Programming, pair programming ensures that code is constantly being examined and reviewed by another team member.

Pair programming

- This is an approach where 2 people are responsible for code development and work together to achieve this.
- Code developed by an individual is therefore constantly being examined and reviewed by another team member.
- Pair programming leads to a deep knowledge of a program, as both programmers have to understand the program in detail to continue development.
- This depth of knowledge is difficult to achieve in inspection processes and pair programming can find bugs that would not be discovered in formal inspections.

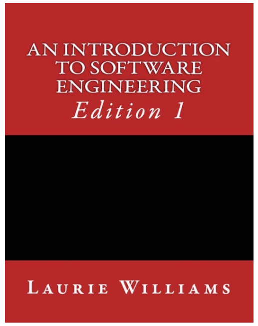
Software Reviews



… three experienced engineers worked for three months to find a subtle system defect that was causing persistent customer problems. At the time they found this defect, the same code was being inspected by a different team of five engineers. As an experiment, this team was not told about the defect. Within two hours, this team found not only this defect, but also 71 others! Once found, the original defect was trivial to fix.

W. S. Humphrey, A Discipline for Software Engineering . Reading, Mass.: Addison Wesley Longman, 1995.

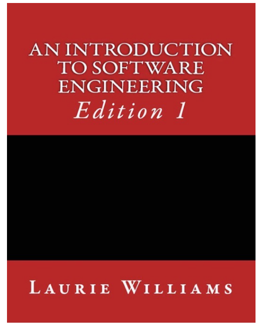
Software Reviews



Software reviews are *a process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users, customers, or other interested parties for comment or approval. Types include code review, design review, formal qualification review, requirements review, test readiness review.*

IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.

Objectives



- To detect errors in program logic/structure or inconsistencies from one artifact to the next.
  - Programming should be a public process – exposing programs to others helps quality, both through the pressure by peers to do things well and because peers spot flaws and bugs that an individual might not. (F. P. Brooks, The Mythical Man-Month, Anniversary Edition : Addison-Wesley Publishing Company, 1995.)
- To make sure the intention of the artifact is clear (the more clear the better)
- To verify that the design and/or software meets its requirements
- To ensure software has been developed in a uniform manner, using agreed-upon standards

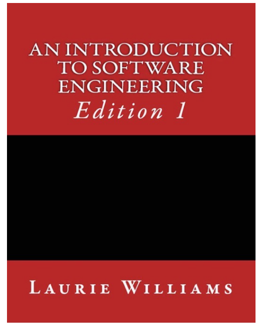
“Many eyes make all bugs shallow”

Standard Refrain in Open Source

“Have peers, rather than customers, find defects”

Karl Wiegers

Walkthroughs



- A *walkthrough* is a *static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a segment of documentation or code, and the participants ask questions and make comments about possible errors, violations of development standards, and other problems.*
- Three roles:
  - **Author:** The author of the material presents their work
  - **Moderator:** The moderator handles the administrative aspects of the walkthrough, such as determining the schedule and distributing materials, and ensures it is conducted in an orderly manner.
  - **Recorder:** The recorder writes down the comments made during the walkthrough. The comments pertain to errors found, questions of style, omission, contradictions, and suggestions for improvement and alternative approaches.

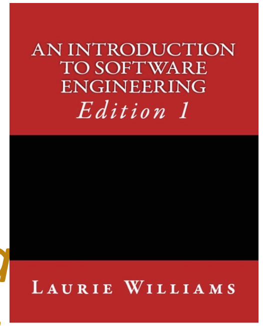
Formal Inspections

- Idea popularized in 70s at IBM
- Broadly adopted in 80s, much research
  - Sometimes replacing component testing
- Group of developers meets to formally review code or other artifacts
- Most effective approach to find bugs
  - Typically 60-90% of bugs found with inspections
- Expensive and labor-intensive

Inspection Team and Roles

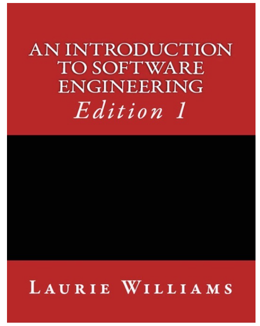
- Typically 4-5 people (min 3)
- Author
- Inspector(s)
  - Find faults and broader issues
- Reader
  - Presents the code or document at inspection meeting
- Scribe
  - Records results
- Moderator
  - Manages process, facilitates, reports

Inspections



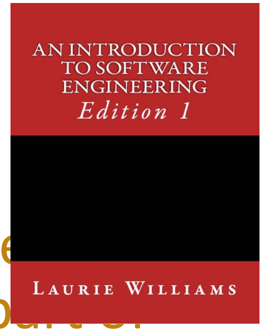
- An *inspection* is a *static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems.*
- Michael Fagan originated this technique and required several participants with particular roles:
  - **Author:** The person who created the document being inspected. As opposed to the walkthrough, they are present at the inspection to answer questions to help others understand the work but does not step through the work; the reader does that. The authors listen to the input of the inspection team but should not “defend” their work. The author does not take on any of the four roles defined on the next slide.

Inspections



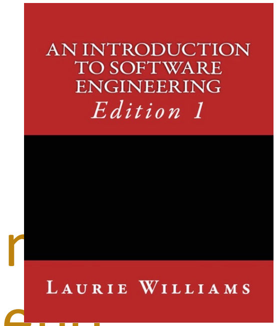
- **Moderator:** The moderator chooses the inspection team, schedules the inspection meeting, ensures the artifact to be reviewed is complete, and distributes the materials. In the inspection meeting, the moderator runs the inspection and enforces the protocols of the meeting. The moderator's job is mainly one of controlling interactions and keeping the group focused on the purpose of the meeting – to discover (but not fix) deficiencies in the document. The moderator also ensures that the group does not drift off onto a tangent and that everyone sticks to a schedule.

Inspections



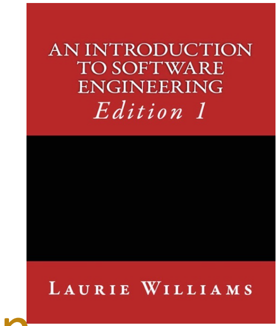
- **Reader:** The reader leads the inspection team through the software eleme in a logical and comprehensive fashion. He or she calls attention to each p the document in turn – paraphrasing or reading line-by lines as appropriate. The reader paces the inspection.
- **Recorder:** Whenever any problem is uncovered in the document being inspected, the recorder describes the defect in writing. After the inspection, the recorder and moderator prepare an inspection report.
- **Inspectors:** The inspectors raise questions and suggest problems with the document. Inspectors are not supposed to "attack" the author or the document but instead they should strive to be objective and constructive. Everyone except the author can act as an inspector. Often inspectors are chosen to represent different viewpoints, for example requirements, design, code, test, project management, quality management.

Why Inspections not as Common?



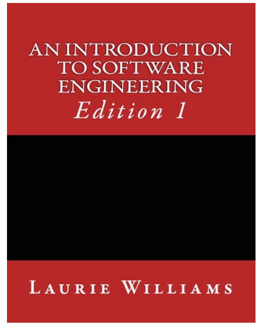
- Developers simply don't believe that the reviews are worth their time they've got a deadline to meet. Instead, these same developers spend endless hours in long, error-prone debugging sessions, finding errors that could have been efficiently found in a review.
- Developers might have ego problems in reviews. They might have trouble admitting their own mistakes and don't want a room full of people seeing their defects. We need to develop an egoless programming culture where we each learn from each other and benefit from each others' input so we can grow as software engineers and so we can produce higher quality products.
- Some software engineers avoid inspections because they find inspections boring.

Pair Programming



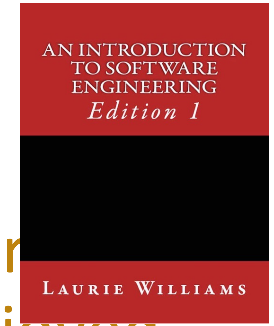
- Pair programming is a technique that can be used to complement or as an alternative to software reviews.
- One of the pair, called the *driver*, types at the computer or writes down a design.
- The other partner, called the *navigator* does many jobs:
  - Observe the work of the driver – looking for tactical (e.g. syntax errors, typos, calling the wrong method) and strategic (e.g. heading down the wrong path) defects in the driver's work.
  - The navigator is the strategic, longer-range thinker of the programming pair.
  - The navigator can have a more objective point of view and can better think strategically about the direction of the work
- Both can brainstorm at any time the situation calls for it! Need to periodically *swap roles*.

Why Pair Program?



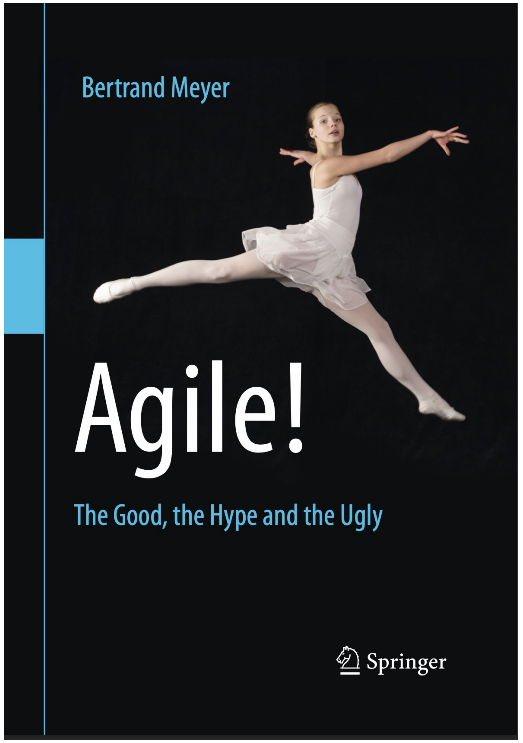
- Why pay two programmers to do the work one could do?
- Research shows that student pairs develop *higher-quality code* faster with only a minimal increase in the total time spent in coding:
  - L. Williams, R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair-Programming," IEEE Software, vol. 17, no. 4, pp. 19-25, July/August 2000 2000.
  - L. A. Williams, "The Collaborative Software Process," in Department of Computer Science Salt Lake City, UT: University of Utah, 2000.
  - N. Nagappan, L. Williams, M. Ferzli, K. Yang, E. Wiebe, C. Miller, and S. Balik, "Improving the CS1 Experience with Pair Programming," in ACM Special Interest Group Computer Science Education (SIGCSE) 2003 , Reno, 2003, pp. 359 362.
  - L. Williams, E. Wiebe, K. Yang, M. Ferzli, and C. Miller, "In Support of Pair Programming in the Introductory Computer Science Course," Computer Science Education, vol. 12, no. 3, pp. 197-212, 2002.
  - L. Williams, K. Yang, E. Wiebe, M. Ferzli, and C. Miller, "Pair Programming in an Introductory Computer Science Course: Initial Results and Recommendations," in OOPSLA Educator's Symposium , Seattle, WA, 2002, pp. 20-26.

Why Pair Program?



- *Increased Morale.* Pair programmers are happier programmer the surveys on the previous slide, 92% indicated that they enjoyed programming more and 96% indicated they felt more confident in their product.
- *Increased Teamwork.* Pair programmers get to know their classmates much better because they work so closely together. Classmates then seem more "approachable" when you have a question about the class.
- *Enhanced Learning.* Pairs continuously learn by watching how their partners approach a task, how they use their language capabilities, and how they use the development.

Pair Programming Criticisms



To study processes, by the way, things are not so easy even with students. If I want to study experimentally whether technique X, say pair programming, is better than technique Y, say code inspections, I cannot just tell all the students
with r
proje
bette

InfoQ: Do you also have examples of agile hypes, things that you think are unlikely to improve software development performance?

and
be
xample.
out all
do not

InfoQ: Finally, which agile practices do you consider to be good or even brilliant? Why?

> **Meyer**: There is a whole chapter in the book on this topic but let me mention just one example: the closed-window rule. It's not emphasized very much in the agile literature, although it's there, but deserves to be better known and applied. It's the rule that during an iteration no one regardless of status can add anything to the task list. The only alternative is to break the iteration. Very carefully considered, practical, and truly brilliant.

https://www.infoq.com/articles/agile-good-hype-ugly/

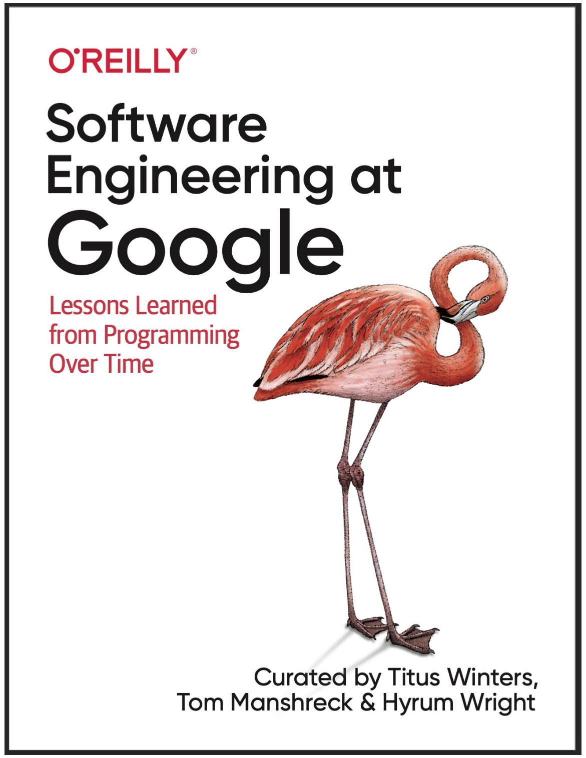
Pair programming weaknesses

- *Mutual misunderstandings*
  - Both members of a pair may make the same mistake in understanding the system requirements. Discussions may reinforce these errors.
- *Pair reputation*
  - Pairs may be reluctant to look for errors because they do not want to slow down the progress of the project.
- *Working relationships*
  - The pair’s ability to discover defects is likely to be compromised by their close working relationship that often leads to reluctance to criticize work partners.

Agile QM and large systems

- When a large system is being developed for an external customer, agile approaches to quality management with minimal documentation may be impractical.
  - If the customer is a large company, it may have its own quality management processes and may expect the software development company to report on progress in a way that is compatible with them.
  - Where there are several geographically distributed teams involved in development, perhaps from different companies, then informal communications may be impractical.
  - For long-lifetime systems, the team involved in development will changeWithout documentation, new team members may find it impossible to understand development.

Code Inspection @ Google



Style Guides and Rules @ Google

- Rules are laws (not just suggestions or recommendations, but strict, mandatory laws).
  - The goal is to encourage “good” and discourage “bad” behaviour (subjective to each Organisation)
- Separate style guides for each of the programming languages
  - Either overarching principles like naming and formatting (Dart, R, Shell)
  - Or delving into specific features and far lengthier (C++, Python, Java)
    - E.g. Google disallows the use of exceptions in C++ - a feature used widely outside of Google code
- Key question: “What goal are we trying to advance?”

Overarching Principles @ Google

- Pull their weight
  - Note modern automation for formatting!
- Optimize for the reader
  - E.g. https://google.github.io/styleguide/pyguide.html#211-conditional-expressions
- Be consistent
- Avoid error-prone and surprising constructs
- Concede to practicalities when necessary

The Style Guide @ Google

- Rules to avoid dangers
- Rules to enforce best practices
- Rules to ensure consistency

https://google.github.io/styleguide/

- When adding a rule, pros, cons, and consequences are analysed to verify that change is appropriate for the scale of Google – these are weighted and documented and have to follow a *process* – decisions are made by *consensus*, not voting by the committees of around 4 language experts.

Code Review @ Google

- See Chapters 9 and 19 (about the Critique Tool @ Google)
- Best Practices
  - Be Polite and Professional
  - Write Small Changes
  - Write Good Change Descriptions
  - Keep Reviewers to a Minimum
  - Automate Where Possible

EXPECTATIONS AND OUTCOMES OF MODERN CODE REVIEWS

Reasons for Code Reviews

- Finding defects
  - both low-level and high-level issues
  - requirements/design/code issues
  - security/performance/… issues
- Code improvement
  - readability, formatting, commenting, consistency, dead code removal, naming
  - enforce to coding standards
- Identifying alternative solutions
- Knowledge transfer
  - learn about API usage, available libraries, best practices, team conventions, system design, "tricks", …
  - “developer education”, especially for junior developers

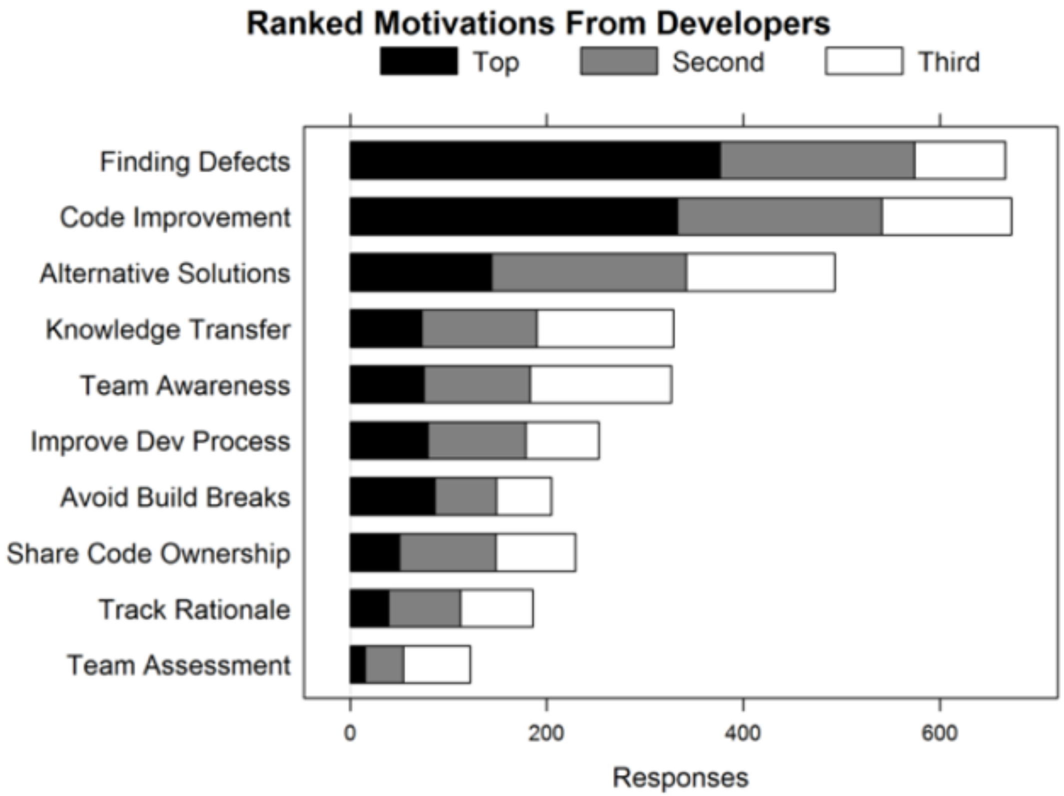
Bacchelli, Alberto, and Christian Bird. "Expectations, outcomes, and challenges of modern code review." *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013.

Reasons for Code Reviews (continued)

- Team awareness and transparency
  - let others "double check" changes
  - announce changes to specific developers or entire team ("FYI")
  - general awareness of ongoing changes and new functionality
- Shared code ownership
  - shared understanding of larger part of the code base
  - openness toward critique and changes
  - makes developers "less protective" of their code

Bacchelli, Alberto, and Christian Bird. "Expectations, outcomes, and challenges of modern code review." *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013.

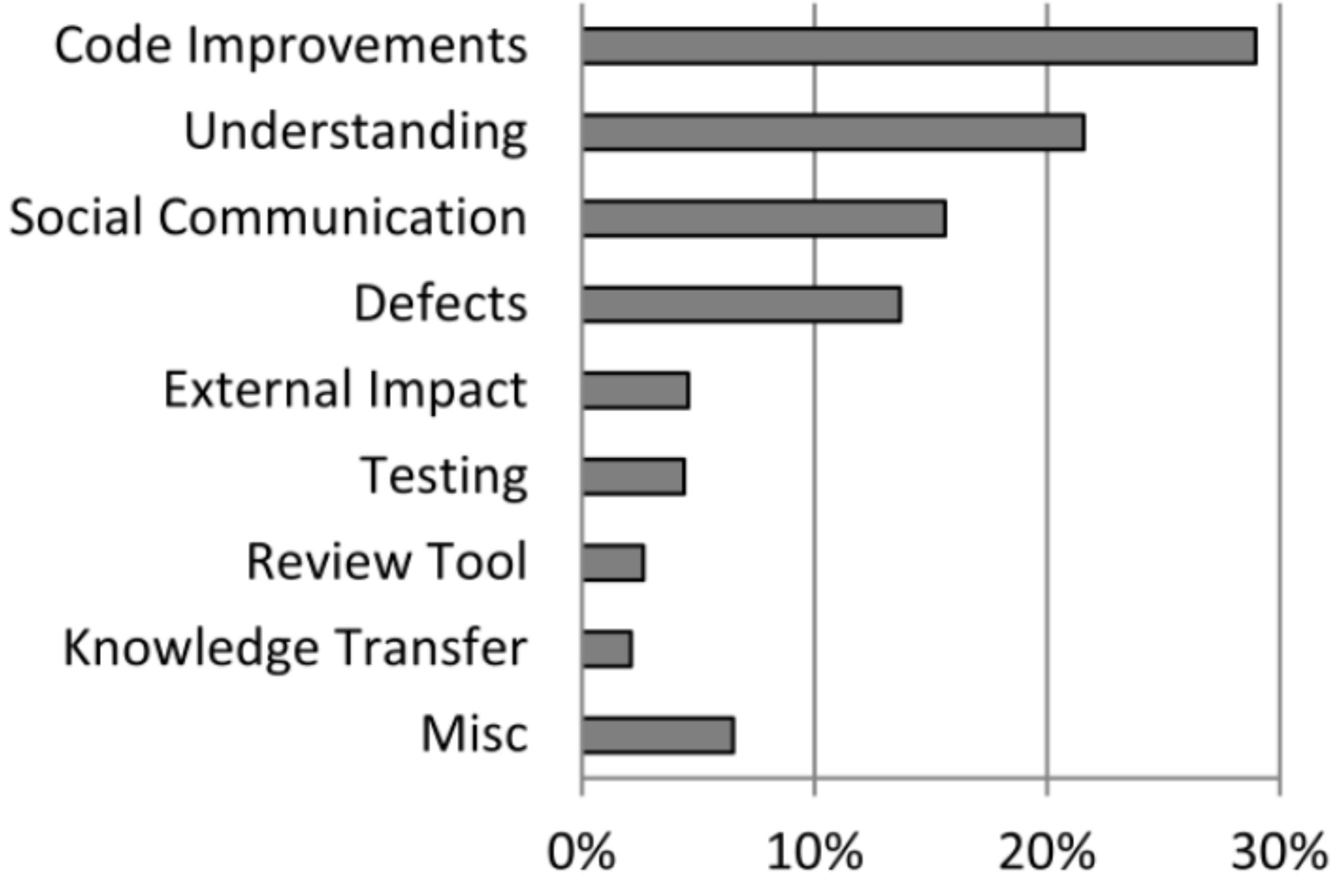
Code Review at Microsoft



Outcomes (at Microsoft analyzing 200 reviews with 570 comments)

- Most frequently code improvements (29%)
  - 58 better coding practices
  - 55 removing unused/dead code
  - 52 improving readability
- Defect finding (14%)
  - 65 logical issues (“uncomplicated logical errors, eg., corner cases, common configuration values, operator precedence)
  - 6 high-level issues
  - 5 security issues
  - 3 wrong exception handling
- Knowledge transfer
  - 12 pointers to internal/external documentation etc

Outcomes (Analyzing Reviews)

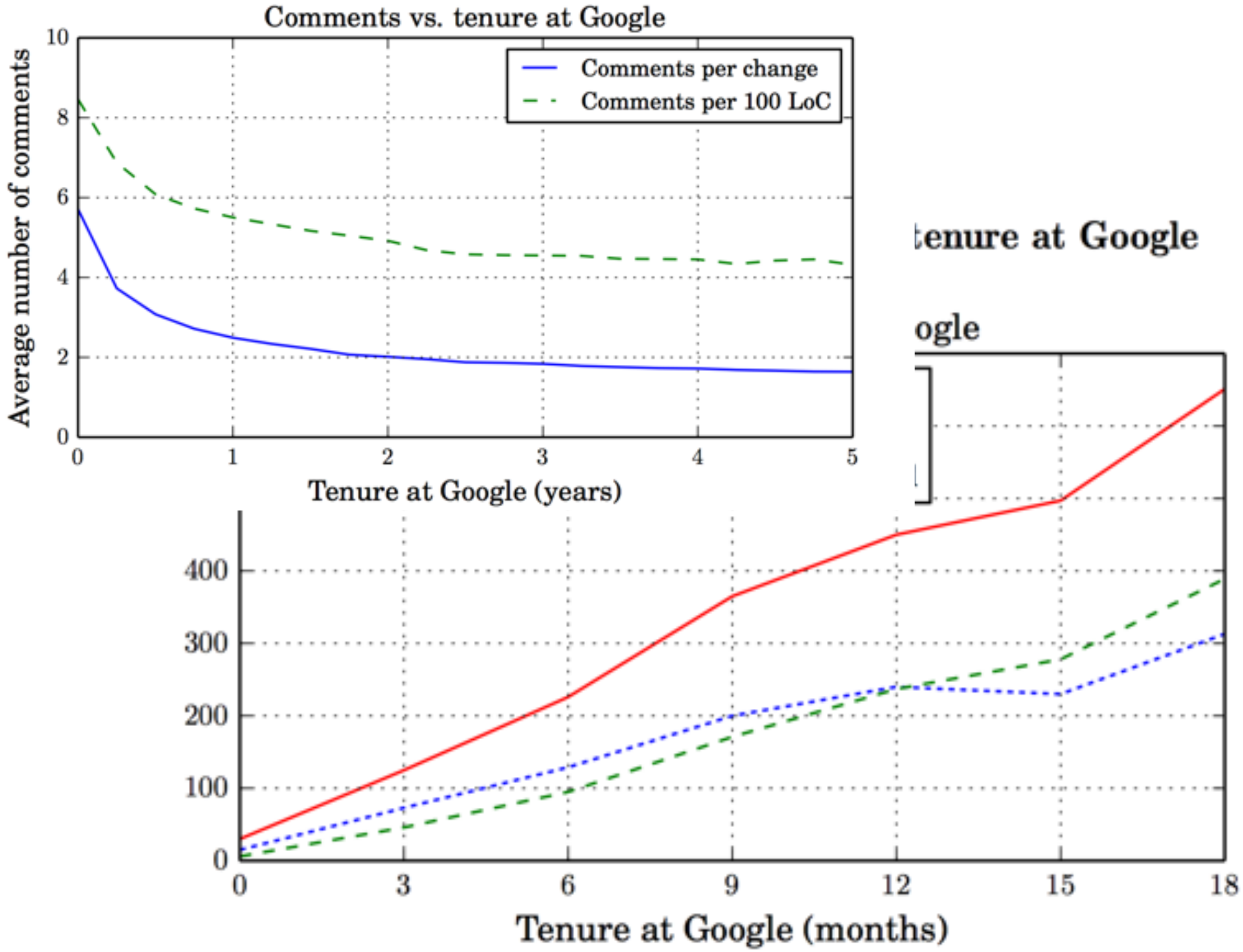


Mismatch of Expectations and Outcomes

- Low quality of code reviews
  - Reviewers look for easy errors, as formatting issues
  - Miss serious errors
- Understanding is the main challenge
  - Understanding the reason for a change
  - Understanding the code and its context
  - Feedback channels to ask questions often needed
- No quality assurance on the outcome

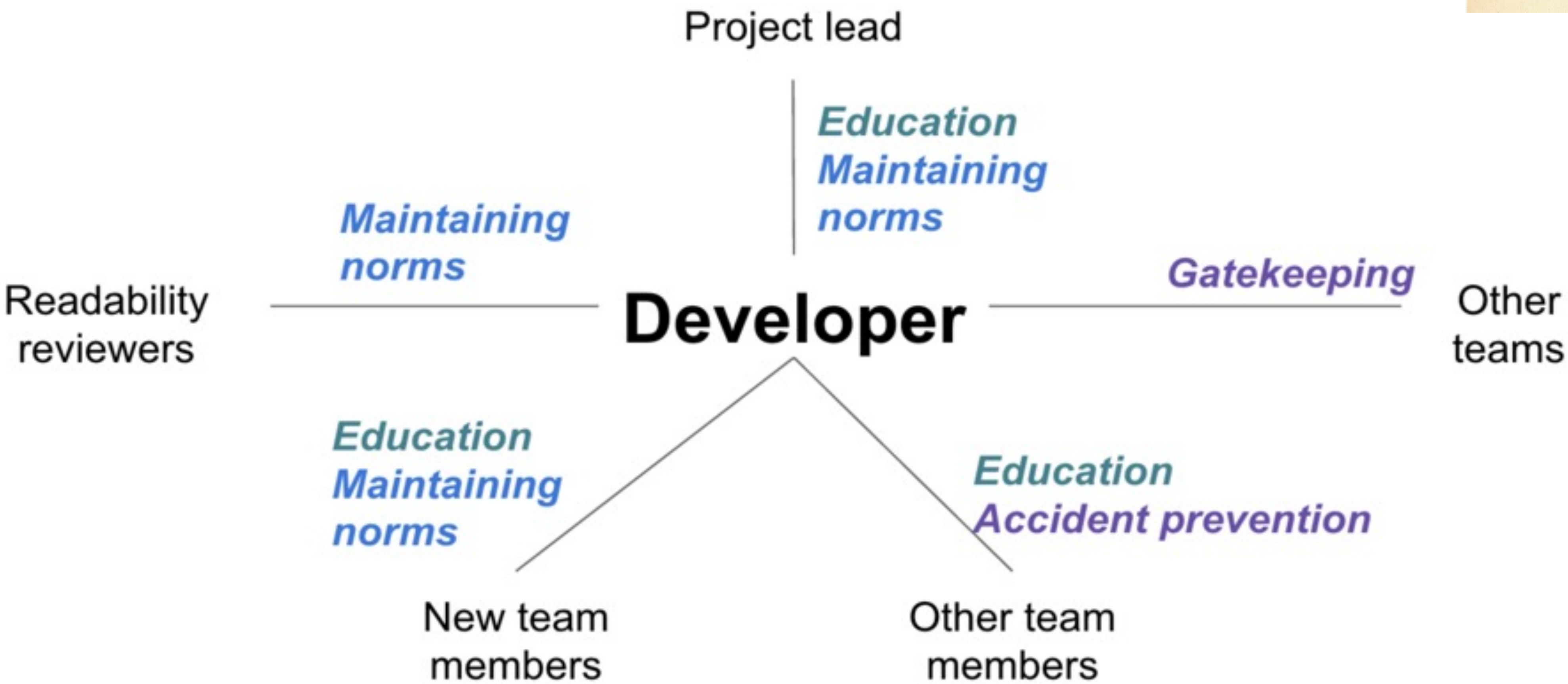
Code Review at Google

- Introduced to *“force developers to write code that other developers could understand”*
- 3 Found benefits:
  - checking the consistency of style and design
  - ensuring adequate tests
  - improving security by making sure no single developer can commit arbitrary code without oversight


Comments vs. tenure at Google
Comments per change
Comments per 100 LoC
Average number of comments
10
8
6
4
2
0
0
1
2
3
4
5
Tenure at Google (years)
tenure at Google
ogle
400
300
200
100
0
0
3
6
9
12
15
18
Tenure at Google (months)


17-313
Foundations of
SOFTWARE
ENGINEERING

Reviewing relationships



Observations?

GOOGLE VS. MICROSOFT?

Don’t forget Devs are Humans too

- Author’s self-worth in artifacts
- CI can avoid embarrassment
- Identify defects, not alternatives; do not criticize authors
  - “you didn’t initialize variable a” -> “I don’t see where variable a is initialized”
- Avoid defending code; avoid discussions of solutions/alternatives
- Reviewers should not “show off” that they are better/smarter
- Avoid style discussions if there are no guidelines
- Author decides how to resolve fault

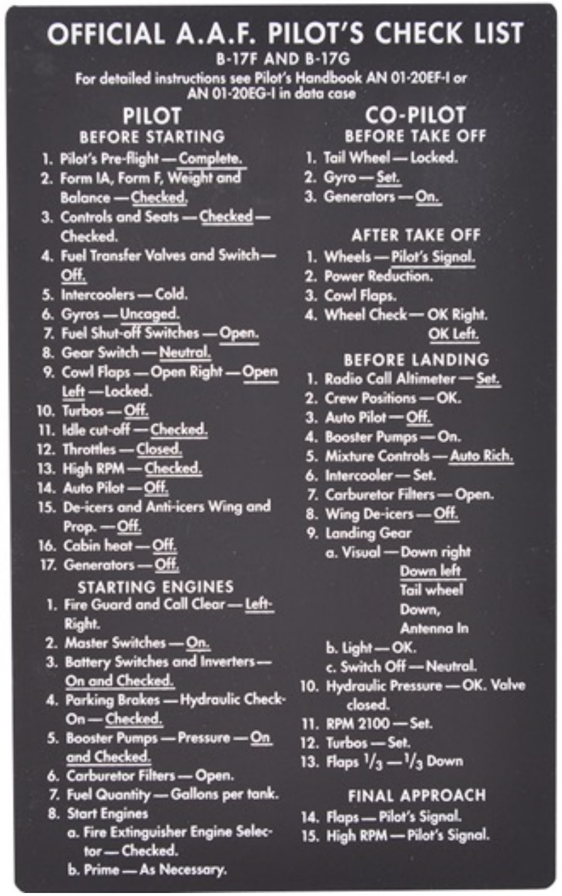
Code Review

Let computers do the parts they are good at,

Let the humans focus on the parts they are good at.

Process: Checklists!

https://en.wikipedia.org/wiki/File:B17_-_Chino_Airshow_2014_(framed).jpg



OFFICIAL A.A.F. PILOT'S CHECK LIST
B-17F AND B-17G
For detailed instructions see Pilot's Handbook AN 01-20EF-I or AN 01-20EG-I in data case

PILOT

BEFORE STARTING
1. Pilot's Pre-flight — Complete.
2. Form IA, Form F, Weight and Balance — Checked.
3. Controls and Seats — Checked — Checked.
4. Fuel Transfer Valves and Switch — Off.
5. Intercoolers — Cold.
6. Gyros — Uncaged.
7. Fuel Shut-off Switches — Open.
8. Gear Switch — Neutral.
9. Cowl Flaps — Open Right — Open Left — Locked.
10. Turbos — Off.
11. Idle cut-off — Checked.
12. Throttles — Closed.
13. High RPM — Checked.
14. Auto Pilot — Off.
15. De-icers and Anti-icers Wing and Prop. — Off.
16. Cabin heat — Off.
17. Generators — Off.

STARTING ENGINES
1. Fire Guard and Call Clear — Left-Right.
2. Master Switches — On.
3. Battery Switches and Inverters — On and Checked.
4. Parking Brakes — Hydraulic Check-On — Checked.
5. Booster Pumps — Pressure — On and Checked.
6. Carburetor Filters — Open.
7. Fuel Quantity — Gallons per tank.
8. Start Engines
a. Fire Extinguisher Engine Selector — Checked.
b. Prime — As Necessary.

CO-PILOT

BEFORE TAKE OFF
1. Tail Wheel — Locked.
2. Gyro — Set.
3. Generators — On.

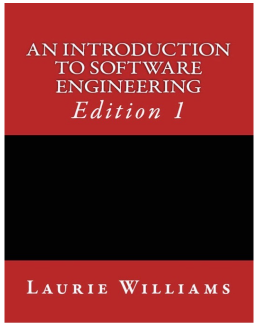
AFTER TAKE OFF
1. Wheels — Pilot's Signal.
2. Power Reduction.
3. Cowl Flaps.
4. Wheel Check — OK Right. OK Left.

BEFORE LANDING
1. Radio Call Altimeter — Set.
2. Crew Positions — OK.
3. Auto Pilot — Off.
4. Booster Pumps — On.
5. Mixture Controls — Auto Rich.
6. Intercooler — Set.
7. Carburetor Filters — Open.
8. Wing De-icers — Off.
9. Landing Gear
a. Visual — Down right Down left Tail wheel Down, Antenna In
b. Light — OK.
c. Switch Off — Neutral.
10. Hydraulic Pressure — OK. Valve closed.
11. RPM 2100 — Set.
12. Turbos — Set.
13. Flaps $1/3$ — $1/3$ Down

FINAL APPROACH
14. Flaps — Pilot's Signal.
15. High RPM — Pilot's Signal.

The Checklist: https://www.newyorker.com/magazine/2007/12/10/the-checklist

Personal Review Checklist

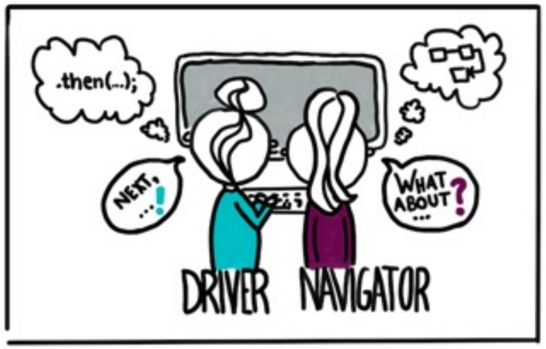


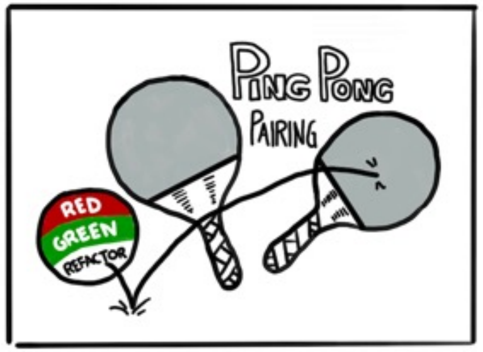
- Are all requirements traceable back to a specific user need?
- Are any requirements included that are impossible to implement?
- Could the requirements be understood and implemented by an independent group?
- Are security requirements specified for each function?
- Is there a glossary in which each term is defined?

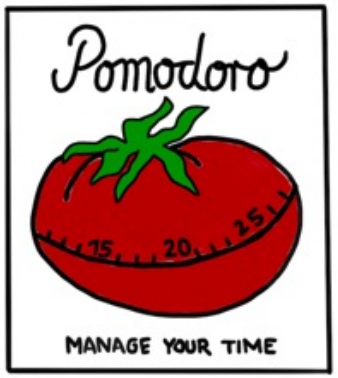
Mini Break in Monday Lecture

PAIR/MOB PROGRAMMING

Pair Programming

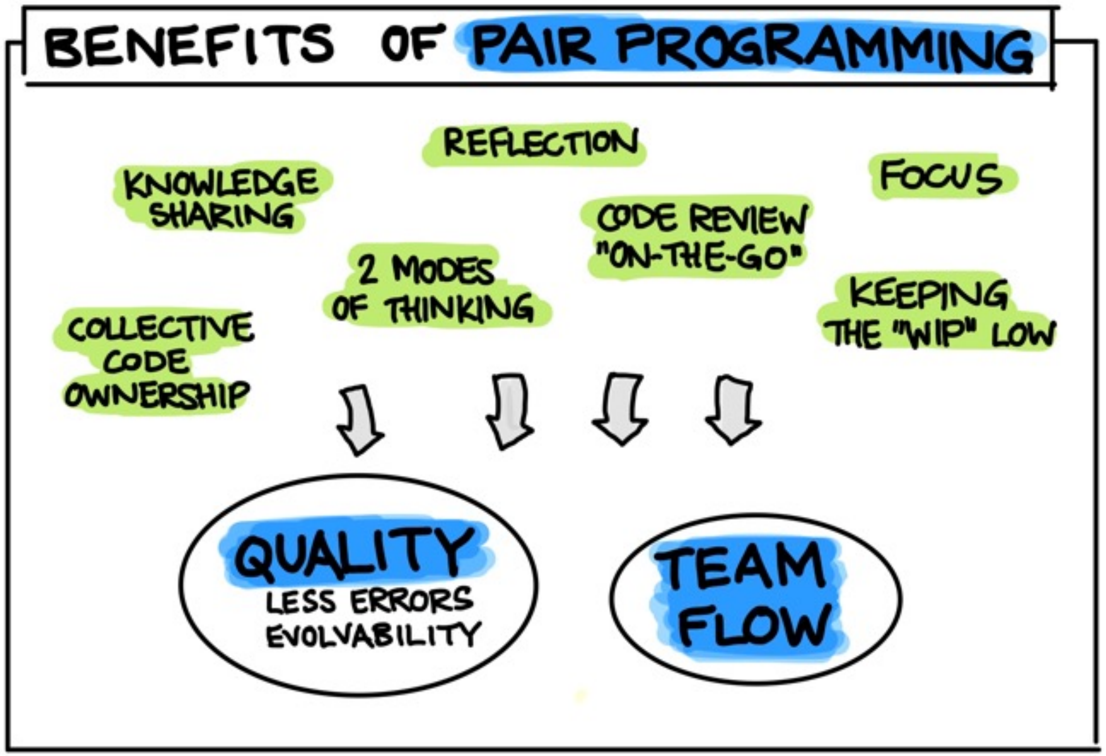






https://martinfowler.com/articles/on-pair-programming.html

Benefits



BENEFITS OF PAIR PROGRAMMING
REFLECTION
KNOWLEDGE SHARING
FOCUS
CODE REVIEW "ON-THE-GO"
2 MODES OF THINKING
COLLECTIVE CODE OWNERSHIP
KEEPING THE "WIP" LOW
QUALITY
LESS ERRORS
EVOLVABILITY
TEAM FLOW

Mob Programming

All the brilliant people working on the same thing, at the same time, in the same space, on the same computer.

– Woody Zuill (the discoverer of Mob Programming)

https://dev.to/albertowar/mob-programming-revisited-2fo4

Its not about getting the MOST out of your Team, Its about getting the Best out of your team

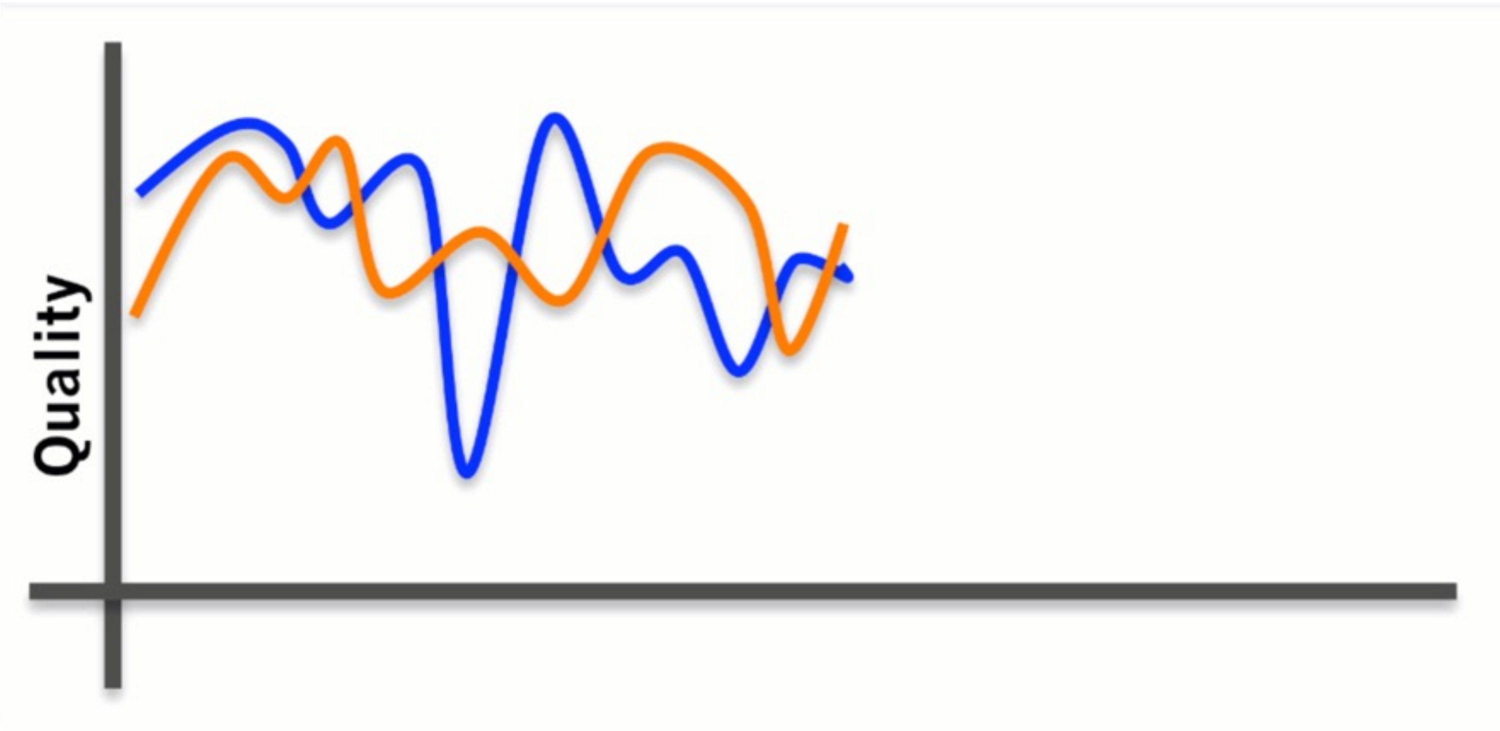
Solo Programming

source: http://i.imgur.com/fGlgTyg.gif

Separate Programming

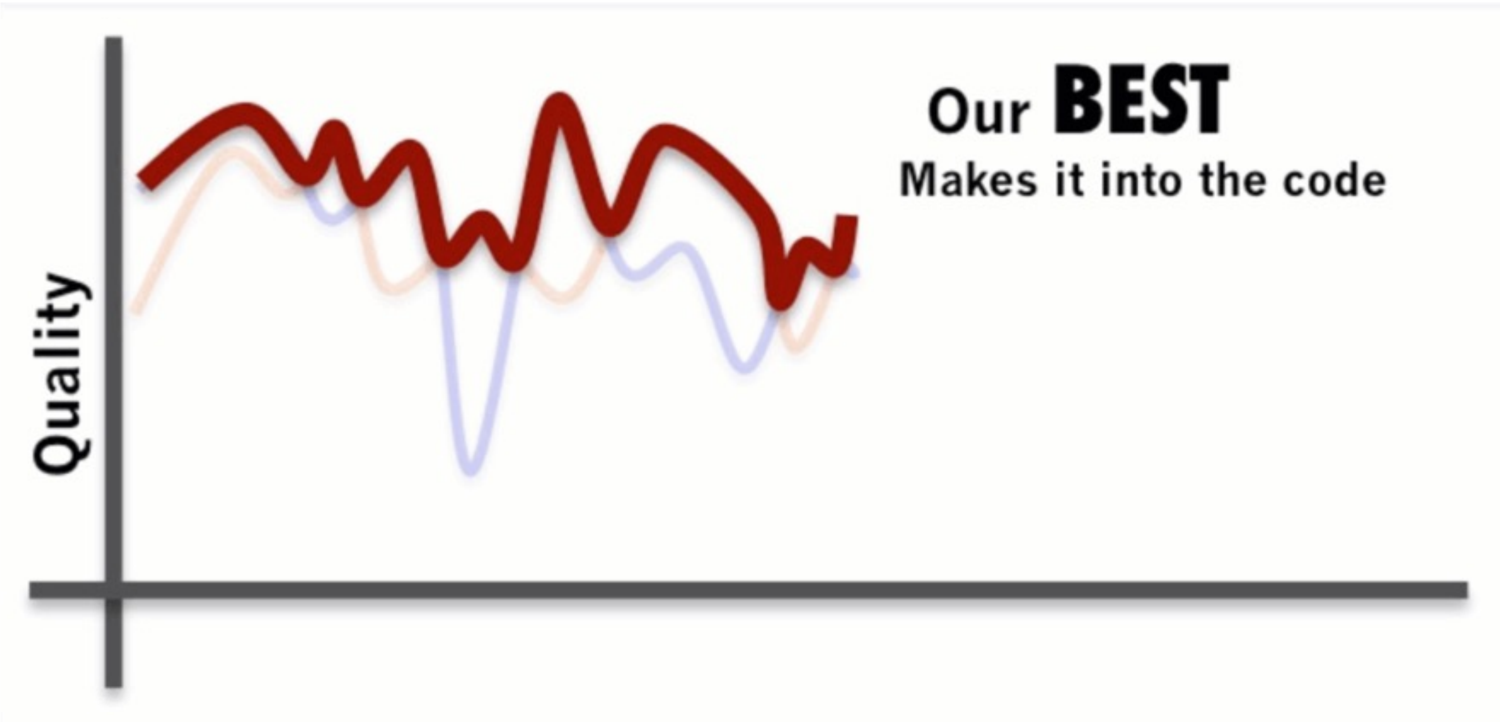
source: http://i.imgur.com/fGlgTyg.gif

Pair Programming



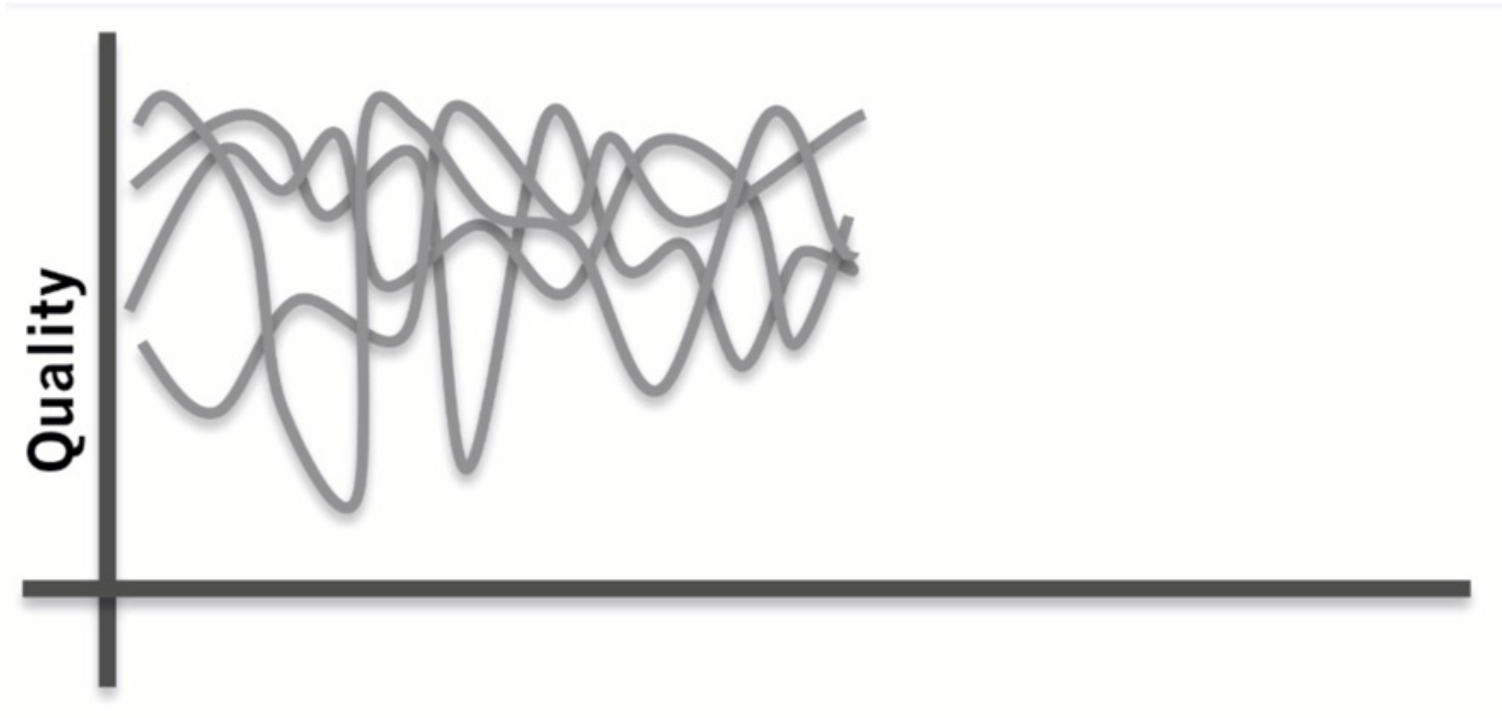
source: http://i.imgur.com/fGlgTyg.gif

Pair Programming



source: http://i.imgur.com/fGlgTyg.gif

Mob Programming

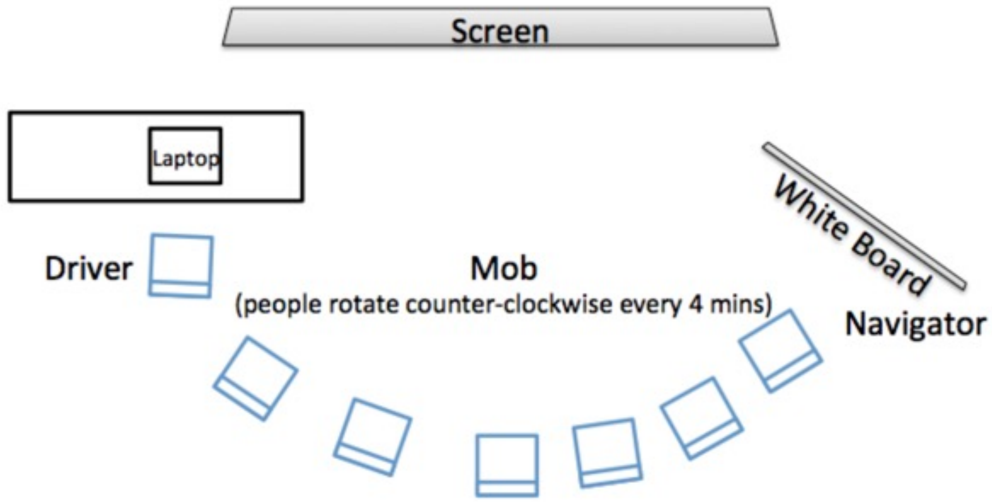


source: http://i.imgur.com/fGlgTyg.gif

Mob Programming

source: http://i.imgur.com/fGlgTyg.gif

Mob Programming Setup



Mob Programming Roles

The Driver: “no thinking, just typing”

The Navigator: the main person programming

The Mob: Checking the navigator, Contributing insights, Getting ready to rotate

The Facilitator: Help guide the mob (Instructor)

RUNNING A MEETING

How to get good answers

- Ask good questions:
- "I am trying to ___, so that I can ___.
  I'm running into ___.
  I've looked at ___ and tried ___."

http://kwugirl.blogspot.com/2014/04/how-to-be-better-junior-developer_25.html

Good Questions

- Keep a log of questions and answers (Make new mistakes! Ask new questions!)
- Try to find answers first (timebox search)
- Keep mental model of who knows what
- Help others learn how to ask good questions too

How to run a meeting

- **The Three Rules of Running a Meeting**
  - Set the Agenda
  - Start on Time. End on Time.
  - End with an Action Plan
- **Give Everyone a Role**
  - **Establish Ground Rules**
  - **Decision, or Consensus?**

https://www.nytimes.com/guides/business/how-to-run-an-effective-meeting

How to run a meeting

Control the Meeting, Not the Conversation

- Let Them Speak

Make Everyone Contribute

- Manage Personalities
- Be Vulnerable
- Make Everyone a Judge

Make Meetings Essential

- Do a Meeting Audit

https://www.nytimes.com/guides/business/how-to-run-an-effective-meeting

https://www.atlassian.com/blog/teamwork/how-to-run-effective-meetings

Random Advice

- Note takers have a lot of power to steer the meeting
  - Collaborative notes are even better!
- Different meeting types have different best practices
  - **Regular team meeting**
  - **Decision-making meeting**
  - **Brainstorming meeting**
  - **Retrospective meeting**
  - **One-on-one meeting**

RELIABILITY

Software quality

- Creating a successful software product does not simply mean providing useful features for users.
- You need to create a high-quality product that people want to use.
- Customers have to be confident that your product will not crash or lose information, and users have to be able to learn to use the software quickly and without mistakes.

Software product quality attributes

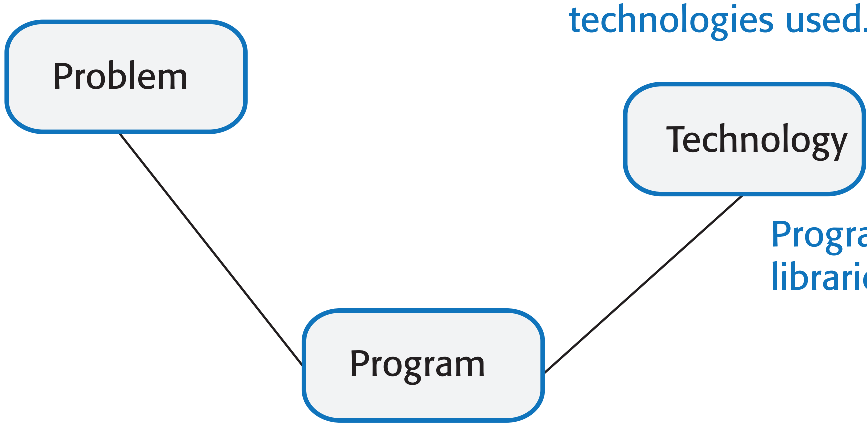
Programming for reliability

- There are three simple techniques for reliability improvement that can be applied in any software company.
  - *Fault avoidance* You should program in such a way that you avoid introducing faults into your program.
  - *Input validation* You should define the expected format for user inputs and validate that all inputs conform to that format.
  - *Failure management* You should implement your software so that program failures have minimal impact on product users.

Underlying causes of program errors

Programmers make mistakes because they don't properly understand the problem or the application domain.

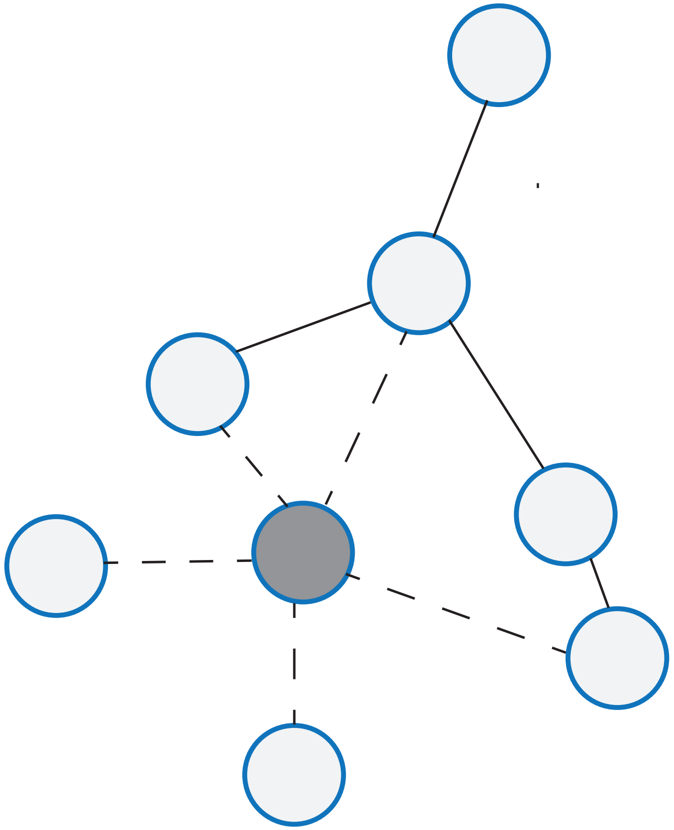
Programmers make mistakes because they use unsuitable technology or they don't properly understand the technologies used.



Programming language, libraries, database, IDE, etc.

Programmers make mistakes because they make simple slips or they do not completely understand how multiple program components work together and change the program's state.

Software complexity



The shaded node interacts, in some ways, with
the linked nodes shown by the dotted line

Program Complexity

- Complexity is related to the number of relationships between elements in a program and the type and nature of these relationships
- The number of relationships between entities is called the coupling. The higher the coupling, the more complex the system.
  - The shaded node on the previous slide has a relatively high coupling because it has relationships with six other nodes.
- A static relationship is one that is stable and does not depend on program execution.
  - Whether or not one component is part of another component is a static relationship.
- Dynamic relationships, which change over time, are more complex than static relationships.
  - An example of a dynamic relationship is the ‘calls’ relationship between functions.

Types of complexity

- **Reading complexity**
  This reflects how hard it is to read and understand the program.
- **Structural complexity**
  This reflects the number and types of relationship between the structures (classes, objects, methods or functions) in your program.
- **Data complexity**
  This reflects the representations of data used and relationships between the data elements in your program.
- **Decision complexity**
  This reflects the complexity of the decisions in your program

Complexity reduction guidelines

- ***Structural complexity***
  - Functions should do one thing and one thing only
  - Functions should never have side-effects
  - Every class should have a single responsibility
  - Minimize the depth of inheritance hierarchies
  - Avoid multiple inheritance
  - Avoid threads (parallelism) unless absolutely necessary
- ***Data complexity***
  - Define interfaces for all abstractions
  - Define abstract data types
  - Avoid using floating-point numbers
  - Never use data aliases
- ***Conditional complexity***
  - Avoid deeply nested conditional statements
  - Avoid complex conditional expressions

Ensure that every class has a single responsibility

- You should design classes so that there is only a single reason to change a class.
  - If you adopt this approach, your classes will be smaller and more cohesive.
  - They will therefore be less complex and easier to understand and change.
- The notion of 'a single reason to change' is, I think, quite hard to understand. However, in a blog post, Bob Martin explains the single responsibility principle in a much better way:
  - Gather together the things that change for the same reasons.
  - Separate those things that change for different reasons.

The DeviceInventory class

DeviceInventory

laptops
tablets
phones
device_assignment

addDevice
removeDevice
assignDevice
unassignDevice
getDeviceAssignment

(a)

DeviceInventory

laptops
tablets
phones
device_assignment

addDevice
removeDevice
assignDevice
unassignDevice
getDeviceAssignment
printInventory

(b)

Adding a printInventory method

- One way of making this change is to add a printInventory method, as shown in the previous slide.

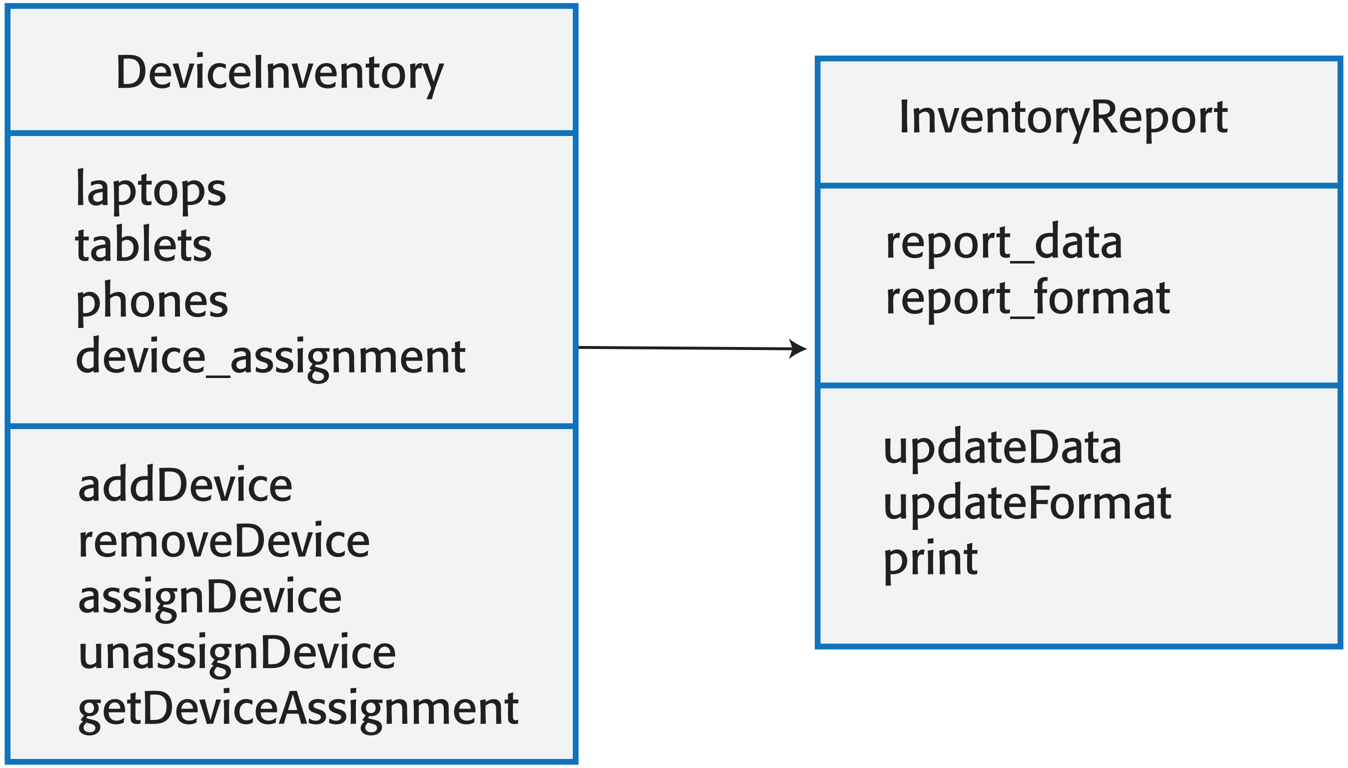
- This change breaks the single responsibility principle as it then adds an additional ‘reason to change’ the class.

  - Without the printInventory method, the reason to change the class is that there has been some fundamental change in the inventory, such as recording who is using their personal phone for business purposes.

  - However, if you add a print method, you are associating another data type (a report) with the class. Another reason for changing this class might then be to change the format of the printed report.

- Instead of adding a printInventory method to DeviceInventory, it is better to add a new class to represent the printed report as shown on the next slide.

The DeviceInventory and InventoryReport classes



Avoid deeply nested conditional statements

- Deeply nested conditional (if) statements are used when you need to identify which of a possible set of choices is to be made.
- For example, the function ‘agecheck’ in Program 8.1 is a short Python function that is used to calculate an age multiplier for insurance premiums.
  - The insurance company’s data suggests that the age and experience of drivers affects the chances of them having an accident, so premiums are adjusted to take this into account.
  - It is good practice to name constants rather than using absolute numbers, so Program 8.1 names all constants that are used.

Deeply nested if-then-else statements


```
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80

YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1

YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5

def agecheck (age, experience):

   # Assigns a premium multiplier depending on the age and experience of the driver

   multiplier = NO_MULTIPLIER

   if age <= YOUNG_DRIVER_AGE_LIMIT:

      if experience <= YOUNG_DRIVER_EXPERIENCE:

         multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER * YOUNG_DRIVER_EXPERIENCE_MULTIPLIER

      else:

         multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER

   else:

      if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:

         if experience <= OLDER_DRIVER_EXPERIENCE:

            multiplier = OLDER_DRIVER_PREMIUM_MULTIPLIER

         else:

            multiplier = NO_MULTIPLIER

      else:

         if age > ELDERLY_DRIVER_AGE:

            multiplier = ELDERLY_DRIVER_PREMIUM_MULTIPLIER

   return multiplier
```

Using guards to make a selection


```
def agecheck_with_guards (age, experience):

    if age <= YOUNG_DRIVER_AGE_LIMIT and experience <= YOUNG_DRIVER_EXPERIENCE:
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER * YOUNG_DRIVER_EXPERIENCE_MULTIPLIER
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER
    if (age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE) and experience <= OLDER_DRIVER_EXPERIENCE:
        return OLDER_DRIVER_PREMIUM_MULTIPLIER
    if age > ELDERLY_DRIVER_AGE:
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER
    return NO_MULTIPLIER
```

Avoid deep inheritance hierarchies

- Inheritance allows the attributes and methods of a class, such as RoadVehicle, can be inherited by sub-classes, such as Truck, Car and MotorBike.

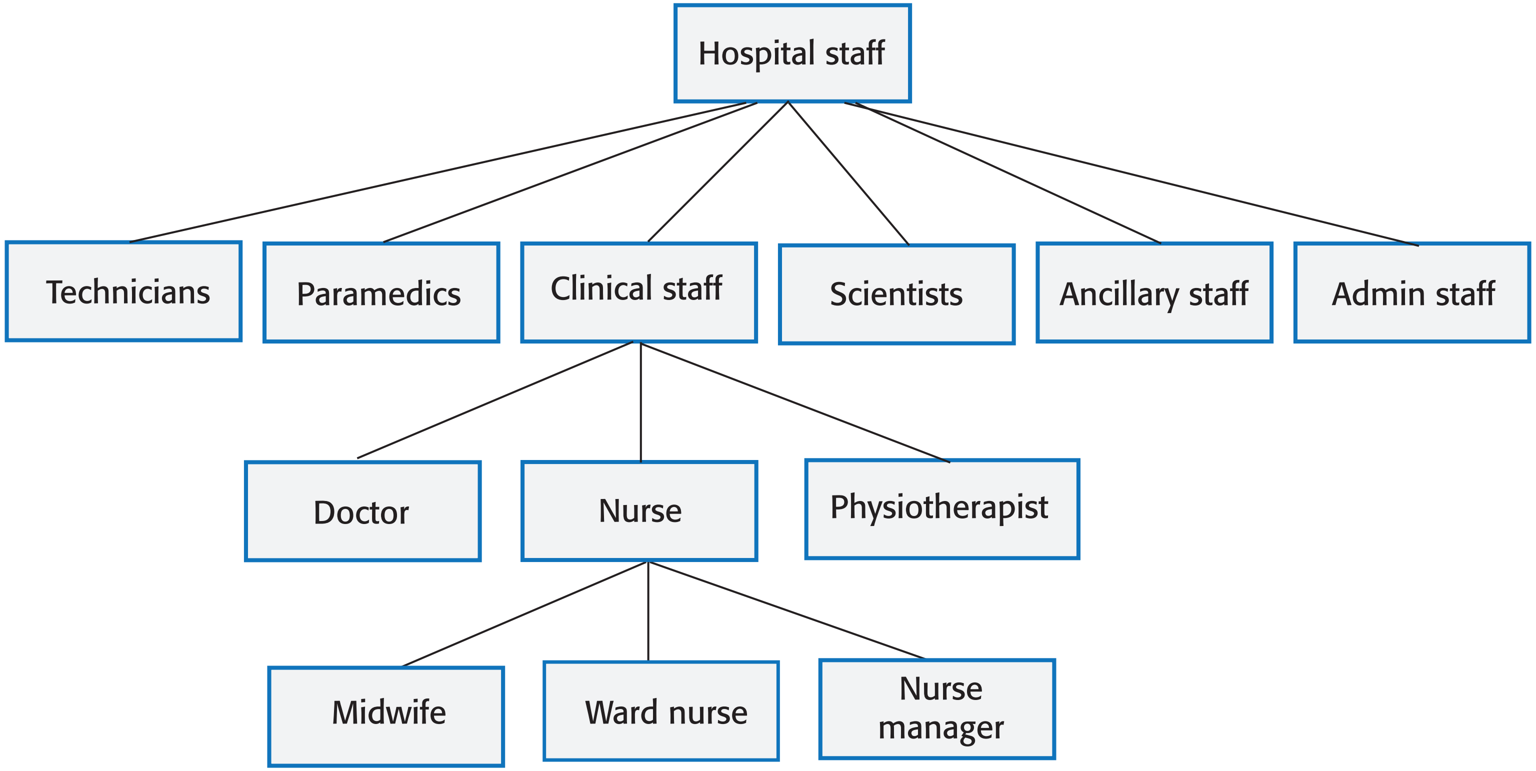
- Inheritance appears to be an effective and efficient way of reusing code and of making changes that affect all subclasses.

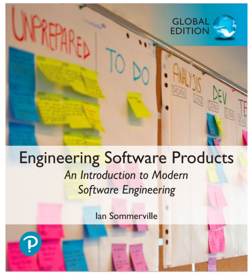
- However, inheritance increases the structural complexity of code as it increases the coupling of subclasses. For example, next slide shows part of a 4-level inheritance hierarchy that could be defined for staff in a hospital.

- The problem with deep inheritance is that if you want to make changes to a class, you have to look at all of its superclasses to see where it is best to make the change.

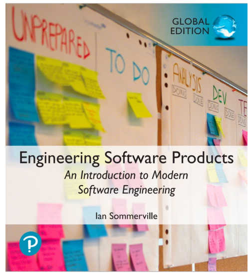
- You also have to look at all of the related subclasses to check that the change does not have unwanted consequences. It’s easy to make mistakes when you are doing this analysis and introduce faults into your program.

Part of the inheritance hierarchy for hospital staff





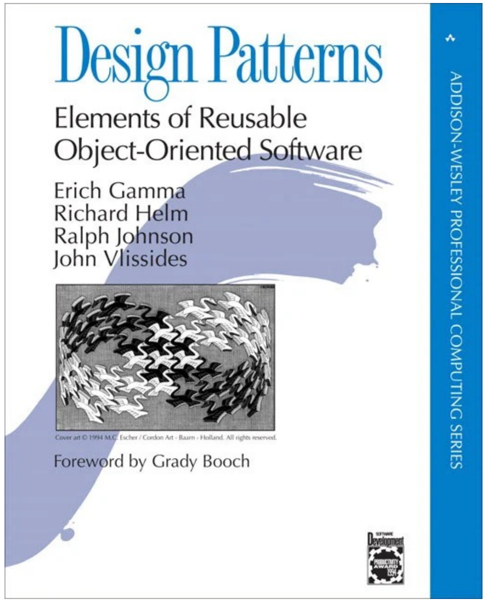
Design Patterns



- Definition
  - **A general reusable solution to a commonly-occurring problem within a given context in software design.**
- Design patterns are object-oriented and describe solutions in terms of objects and classes. They are not off-the-shelf solutions that can be directly expressed as code in an object-oriented language.
- They describe the structure of a problem solution but have to be adapted to suit your application and the programming language that you are using.

Design Patterns

Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.



Programming principles

- Separation of concerns
  - This means that each abstraction in the program (class, method, etc.) should address a separate concern and that all aspects of that concern should be covered there. For example, if authentication is a concern in your program, then everything to do with authentication should be in one place, rather than distributed throughout your code.
- Separate the 'what' from the 'how
  - If a program component provides a particular service, you should make available only the information that is required to use that service (the 'what'). The implementation of the service ('the how') should be of no interest to service users.

Common types of design patterns

- Creational patterns
  - These are concerned with class and object creation. They define ways of instantiating and initializing objects and classes that are more abstract than the basic class and object creation mechanisms defined in a programming language.
- Structural patterns
  - These are concerned with class and object composition. Structural design patterns are a description of how classes and objects may be combined to create larger structures.
- Behavioural patterns
  - These are concerned with class and object communication. They show how objects interact by exchanging messages, the activities in a process and how these are distributed amongst the participating objects.

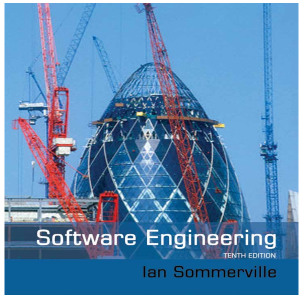
Examples of Design Patterns

Pattern name	***Type***	***Description***
Factory	Creational	Used to create objects when slightly different variants of the object may be created.
Prototype	Creational	Used to create an object clone i.e. a new object with exactly the same attribute values as the object being cloned.
Adapter	Structural	Used to match semantically-compatible interfaces of different classes.
Facade	Structural	Used to provide a single interface to a group of classes in which each class implements some functionality accessed through the interface.
Mediator	Behavioural	Used to reduce the number of direct interactions between objects. All object communications are handled through the mediator.
State	Behavioural	Used to implement a state machine where the behaviour of an object when its internal state changes.

Pattern Description / “Pattern Language”

- Design patterns are usually documented in the stylized way. This includes:
  - a meaningful name for the pattern and a brief description of what it does;
  - a description of the problem it solves;
  - a description of the solution and its implementation;
  - the consequences and trade-offs of using the pattern and other issues that you should consider.

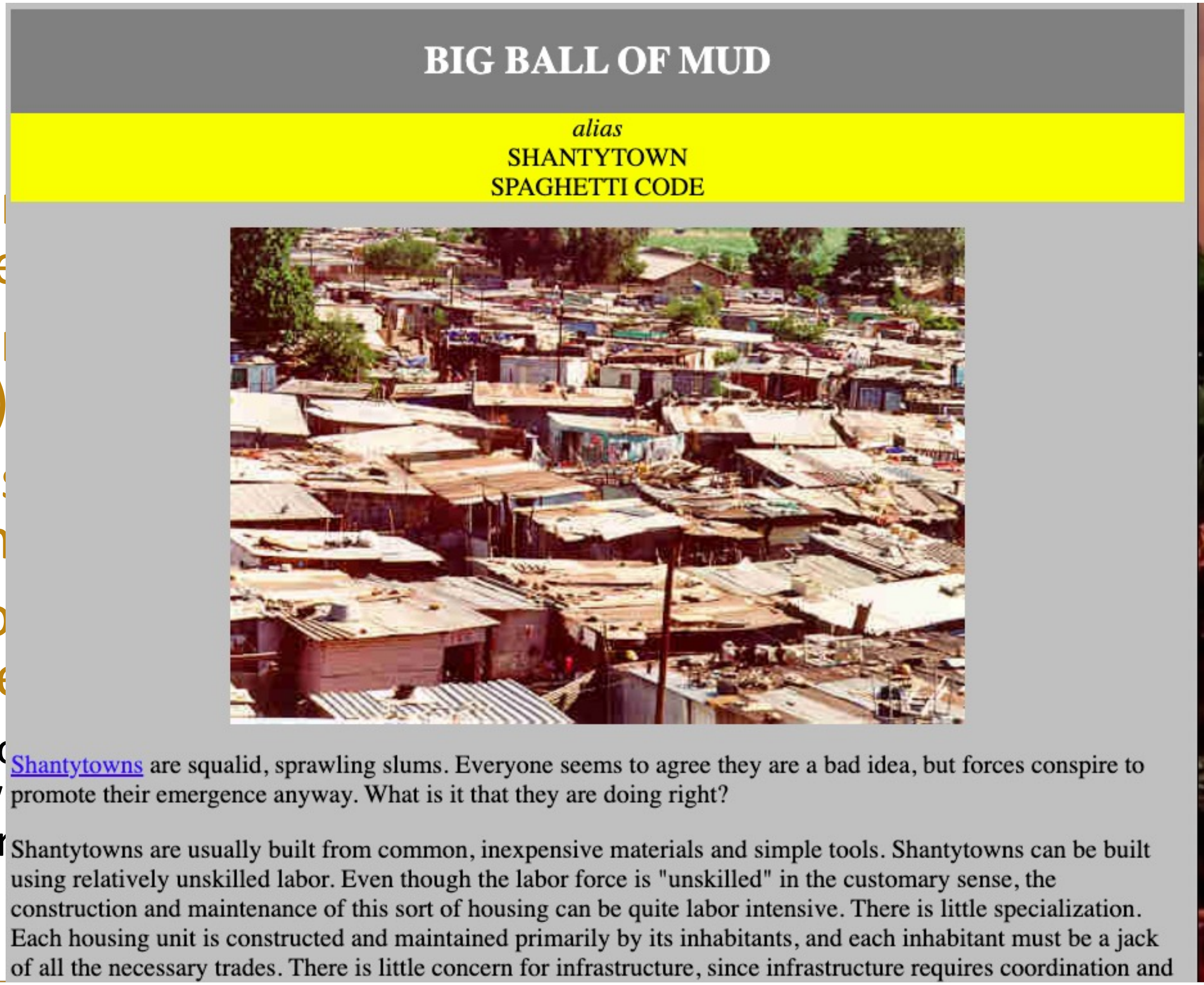
Design problems



- To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied.
  - Tell several objects that the state of some other object has changed (Observer pattern).
  - Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
  - Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
  - Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).

Refactoring

- Refactoring [...]xity without changing the[...]
- Refactoring [...]e 'reading complexity')
- It also make[...]uce the chances of n[...]es.
- The reality o[...] additions to existing code[...]
  - The code bec[...] operations that you started w[...]m in ways that you did not or[...]



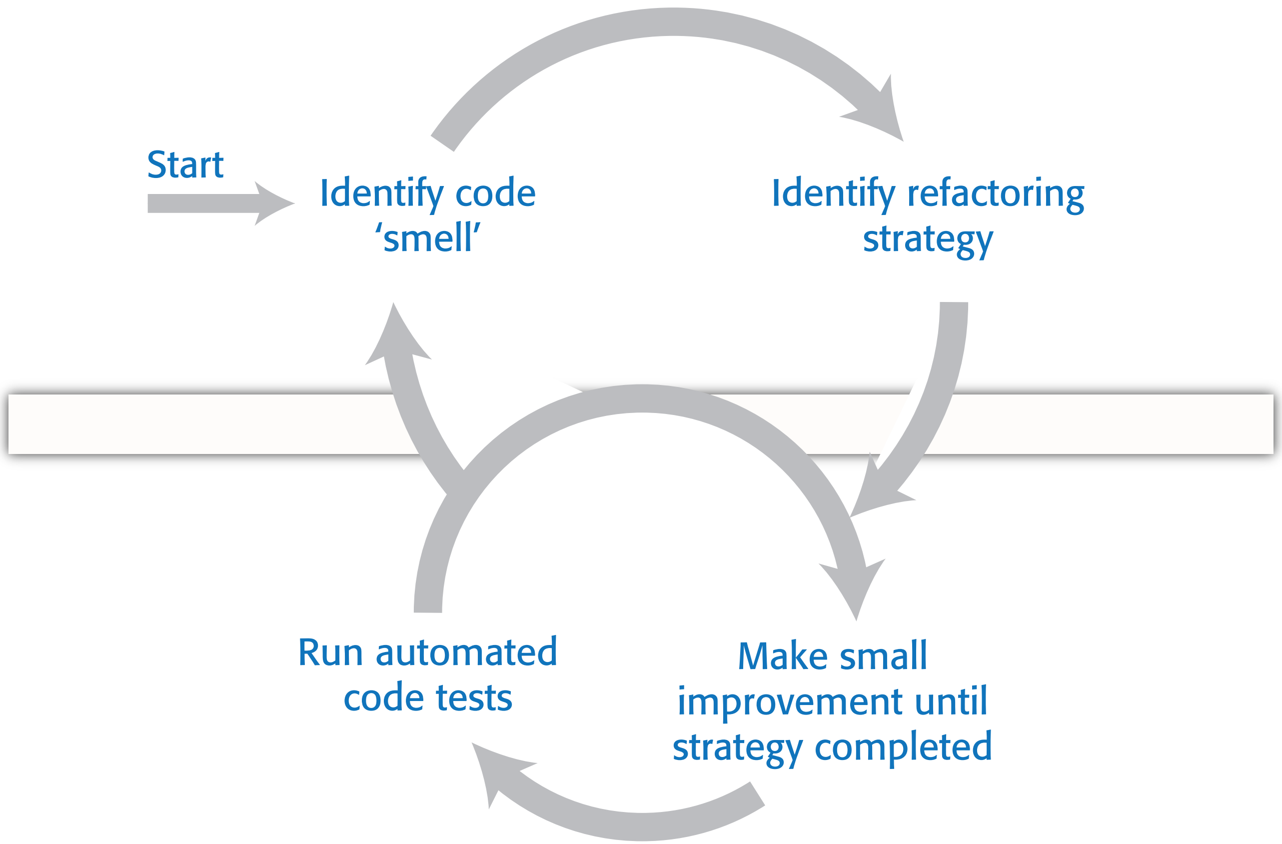
BIG BALL OF MUD

alias
SHANTYTOWN
SPAGHETTI CODE

Shantytowns are squalid, sprawling slums. Everyone seems to agree they are a bad idea, but forces conspire to promote their emergence anyway. What is it that they are doing right?

Shantytowns are usually built from common, inexpensive materials and simple tools. Shantytowns can be built using relatively unskilled labor. Even though the labor force is "unskilled" in the customary sense, the construction and maintenance of this sort of housing can be quite labor intensive. There is little specialization. Each housing unit is constructed and maintained primarily by its inhabitants, and each inhabitant must be a jack of all the necessary trades. There is little concern for infrastructure, since infrastructure requires coordination and

A refactoring process



Code smells

- Martin Fowler, a refactoring pioneer, suggests that the starting point for refactoring should be to identify code ‘smells’.
- Code smells are indicators in the code that there might be a deeper problem.
  - For example, very large classes may indicate that the class is trying to do too much. This probably means that its structural complexity is high.

Examples of code smells

- *Large classes*
  Large classes may mean that the single responsibility principle is being violated. Break down large classes into easier-to-understand, smaller classes.

- *Long methods/functions*
  Long methods or functions may indicate that the function is doing more than one thing. Split into smaller, more specific functions or methods.

- *Duplicated code*
  Duplicated code may mean that when changes are needed, these have to be made everywhere the code is duplicated. Rewrite to create a single instance of the duplicated code that is used as required

- *Meaningless names*
  Meaningless names are a sign of programmer haste. They make the code harder to understand. Replace with meaningful names and check for other shortcuts that the programmer may have taken.

- *Unused code*
  This simply increases the reading complexity of the code. Delete it even if it has been commented out. If you find you need it later, you should be able to retrieve it from the code management system.

Examples of refactoring for complexity reduction

- *Reading complexity*
  You can rename variable, function and class names throughout your program to make their purpose more obvious.
- *Structural complexity*
  You can break long classes or functions into shorter units that are likely to be more cohesive than the original large class.
- *Data complexity*
  You can simplify data by changing your database schema or reducing its complexity. For example, you can merge related tables in your database to remove duplicated data held in these tables.
- *Decision complexity*
  You can replace a series of deeply nested if-then-else statements with guard clauses, as I explained earlier in this chapter.

Input validation

- Input validation involves checking that a user’s input is in the correct format and that its value is within the range defined by input rules.
- Input validation is critical for security and reliability. As well as inputs from attackers that are deliberately invalid, input validation catches accidentally invalid inputs that could crash your program or pollute your database.
- User input errors are the most common cause of database pollution.
- You should define rules for every type of input field and you should include code that applies these rules to check the field’s validity.
  - If it does not conform to the rules, the input should be rejected.

Methods of implementing input validation

- *Built-in validation functions*
  You can use input validator functions provided by your web development framework. For example, most frameworks include a validator function that will check that an email address is of the correct format.
- *Type coercion functions*
  You can use type coercion functions, such as int() in Python, that convert the input string into the desired type. If the input is not a sequence of digits, the conversion will fail.
- *Explicit comparisons*
  You can define a list of allowed values and possible abbreviations and check inputs against this list. For example, if a month is expected, you can check this against a list of all months and recognised abbreviations.
- *Regular expressions*
  You can use regular expressions to define a pattern that the input should match and reject inputs that do not match that pattern.

Regular expressions

- Regular expressions (REs) are a way of defining patterns.
- A search can be defined as a pattern and all items matching that pattern are returned. For example, the following Unix command will list all the JPEG files in a directory:
- `ls | grep ..*\.jpg$`
- A single dot means 'match any character' and * means zero or more repetitions of the previous character. Therefore ..* means 'one or more characters'. The file prefix is .jpg and the $ character means that it must occur at the end of a line.
- In a program on the next slide, REs are used to check the validity of names.

Number checking

- Number checking is used with numeric inputs to check that these are not too large or small and that they are sensible values for the type of input.
  - For example, if the user is expected to input their height in meters then you should expect a value between 0.6m (a very small adult) and 2.6m (a very tall adult).
- Number checking is important for two reasons:
  - If numbers are too large or too small to be represented, this may lead to unpredictable results and numeric overflow or underflow exceptions. If these exceptions are not properly handled, very large or very small inputs can cause a program to crash.
  - The information in a database may be used by several other programs and these may make assumptions about the numeric values stored. If the numbers are not as expected, this may lead to unpredictable results.

Input range checks

- As well as checking the ranges of inputs, you may also perform checks on these inputs to ensure that these represent sensible values.
- These protect your system from accidental input errors and may also stop intruders who have gained access using a legitimate user's credentials from seriously damaging their account.
- For example, if a user is expected to enter the reading from an electricity meter, then you should
  - (a) check this is equal to or larger than the previous meter reading and
  - (b) consistent with the user's normal consumption.

Failure management

- Software is so complex that, irrespective of how much effort you put into fault avoidance, you will make mistakes. You will introduce faults into your program that will sometimes cause it to fail.
- Program failures may also be a consequence of the failure of an external service or component that your software depends on.
- Whatever the cause, you have to plan for failure and make provisions in your software for that failure to be as graceful as possible.

Failure categories

- Data failures
  - The outputs of a computation are incorrect. For example, if someone's year of birth is 1981 and you calculate their age by subtracting 1981 from the current year, you may get an incorrect result. Finding this kind of error relies on users reporting data anomalies that they have noticed.
- Program exceptions
  - The program enters a state where normal continuation is impossible. If these exceptions are not handled, then control is transferred to the run-time system which halts execution. For example, if a request is made to open a file that does not exist then an IOexception has occurred.
- Timing failures
  - Interacting components fail to respond on time or where the responses of concurrently-executing components are not properly synchronized. For example, if service S1 depends on service S2 and S2 does not respond to a request, then S1 will fail.

Failure effect minimisation

- Persistent data (i.e. data in a database or files) should not be lost or corrupted;
- The user should be able to recover the work that they've done before the failure occurred;
- Your software should not hang or crash;
- You should always 'fail secure' so that confidential data is not left in a state where an attacker can gain access to it.

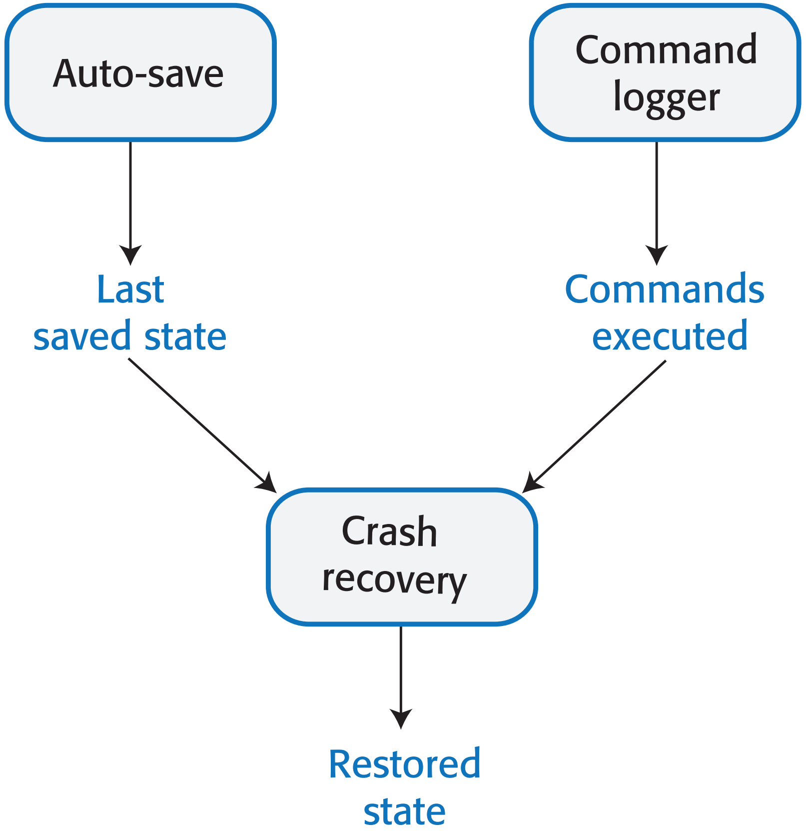
Exception handling

- Exceptions are events that disrupt the normal flow of processing in a program.
- When an exception occurs, control is automatically transferred to exception management code.
- Most modern programming languages include a mechanism for exception handling.
- In Python, you use **try-except** keywords to indicate exception handling code; in Java, the equivalent keywords are **try-catch.**

Auto-save and activity logging

- Activity logging
  - You keep a log of what the user has done and provide a way to replay that against their data. You don't need to keep a complete session record, simply a list of actions since the last time the data was saved to persistent store.
- Auto-save
  - You automatically save the user's data at set intervals - say every 5 minutes. This means that, in the event of a failure, you can restore the saved data with the loss of only a small amount of work.
  - Usually, you don't have to save all of the data but simply save the changes that have been made since the last explicit save.

Auto-save and activity logging



External service failure

- If your software uses external services, you have no control over these services and the only information that you have on service failure is whatever is provided in the service's API.
- As services may be written in different programming languages, these errors can't be returned as exception types but are usually returned as a numeric code.
- When you are calling an external service, you should always check that the return code of the called service indicates that it has operated successfully.
- You should, also, if possible, check the validity of the result of the service call as you cannot be certain that the external service has carried out its computation correctly.

Using assertions to check results from an external service


```
def credit_checker (name, postcode, dob):

    # Assume that the function check_credit_rating calls an external
service
    # to get a person's credit rating. It takes a name, postcode (zip
code)
    # and date of birth as parameters and returns a sequence with the
database
    # information (name, postcode, date of birth) plus a credit score
between 0 and
    # 600. The final element in the sequence is an error_code which
may
    # be 0 (successful completion), 1 or 2.
    NAME = 0
    POSTCODE = 1
    DOB = 2
    RATING = 3
    RETURNCODE = 4
    REQUEST_FAILURE = True
    ASSERTION_ERROR = False
```

```
    cr = ['', '', '', -1, 2]

    # Check credit rating simulates call to external service
    cr = check_credit_rating (name, postcode, dob)
    try:
        assert cr [NAME] == name and cr [POSTCODE] == postcode and cr
[DOB] == dob \
            and (cr [RATING] >= 0 and cr [RATING] <= 600) and \
            (cr[RETURNCODE] >= 0 and cr[RETURNCODE] <= 2)
        if cr[RETURNCODE] == 0:
            do_normal_processing (cr)
        else:
            do_exception_processing (cr, name, postcode, dob,
REQUEST_FAILURE)
    except AssertionError:
            do_exception_processing (cr, name, postcode, dob,
ASSERTION_ERROR)
```

Key Points

- The most important quality attributes for most software products are reliability, security, availability, usability, responsiveness and maintainability.
- To avoid introducing faults into your program, you should use programming practices that reduce the probability that you will make mistakes.
- You should always aim to minimize complexity in your programs. Complexity makes programs harder to understand. It increases the chances of programmer errors and makes the program more difficult to change.
- Design patterns are tried and tested solutions to commonly occurring problems. Using patterns is an effective way of reducing program complexity.
- Refactoring is the process of reducing the complexity of an existing program without changing its functionality. It is good practice to refactor your program regularly to make it easier to read and understand.
- Input validation involves checking all user inputs to ensure that they are in the format that is expected by your program. Input validation helps avoid the introduction of malicious code into your system and traps user errors that can pollute your database.

Key Points

- Regular expressions are a way of defining patterns that can match a range of possible input strings. Regular expression matching is a compact and fast way of checking that an input string conforms to the rules you have defined.

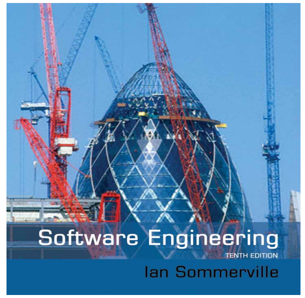
- You should check that numbers have sensible values depending on the type of input expected. You should also check number sequences for feasibility.

- You should assume that your program may fail and to manage these failures so that they have minimal impact on the user.

- Exception management is supported in most modern programming languages. Control is transferred to your own exception handler to deal with the failure when a program exception is detected.

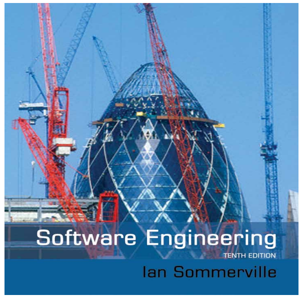
- You should log user updates and maintain user data snapshots as your program executes. In the event of a failure, you can use these to recover the work that the user has done. You should also include ways of recognizing and recovering from external service failures.

Implementation issues



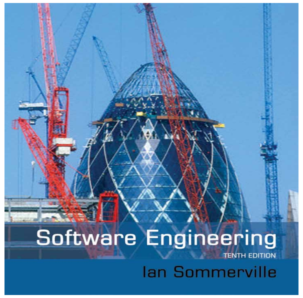
- Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
  - Reuse Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
  - Configuration management During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - Host-target development Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

Reuse



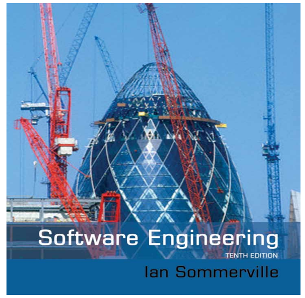
- From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  - The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

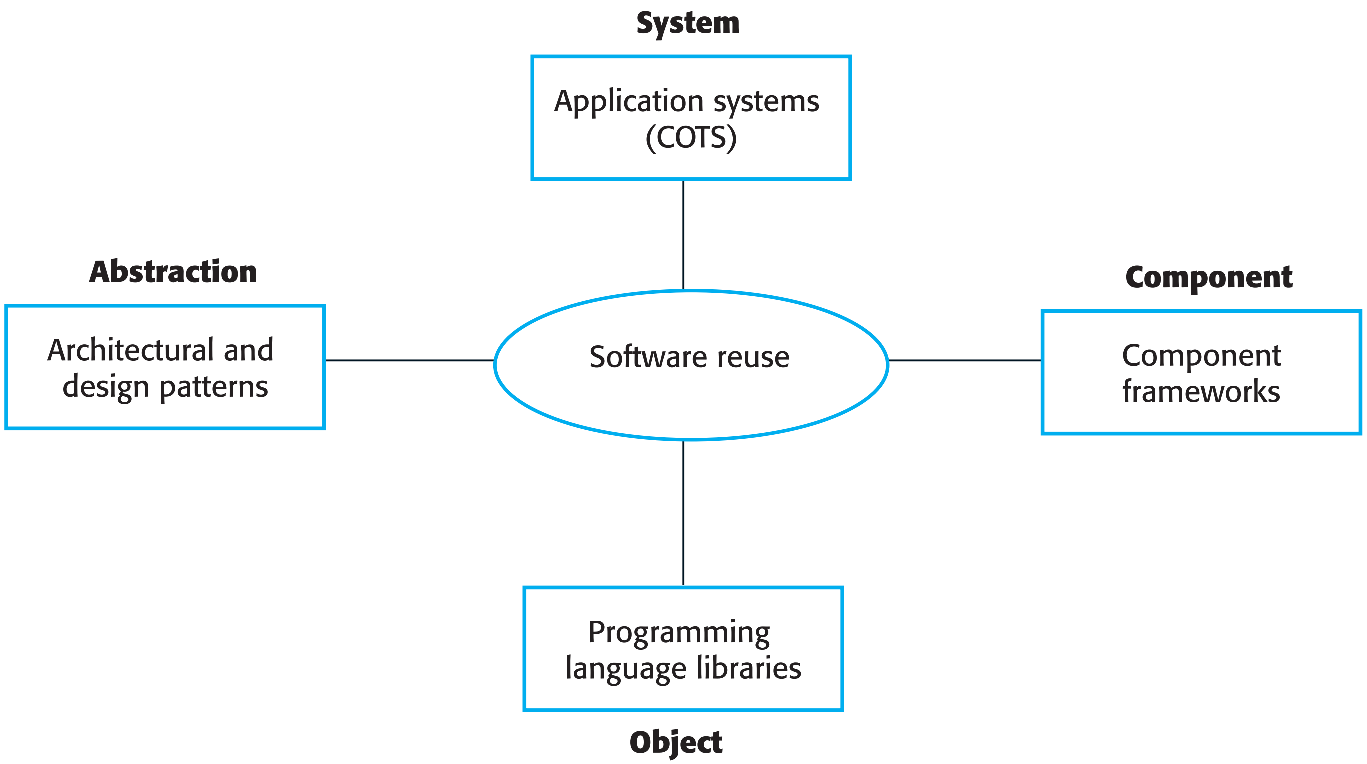
Reuse levels



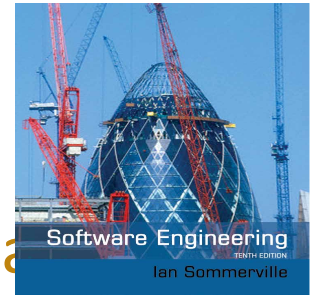
- The abstraction level
  - At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.
- The object level
  - At this level, you directly reuse objects from a library rather than writing the code yourself.
- The component level
  - Components are collections of objects and object classes that you reuse in application systems.
- The system level
  - At this level, you reuse entire application systems.

Software reuse



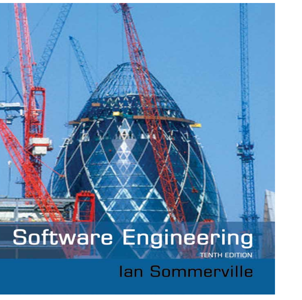


Reuse costs



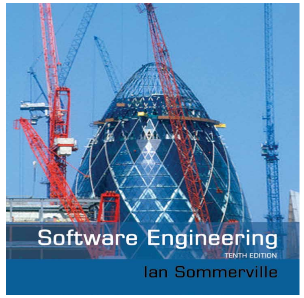
- The costs of the time spent in looking for software to reuse a assessing whether or not it meets your needs.
- Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

Configuration management



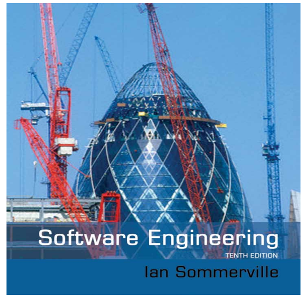
- Configuration management is the name given to the general process of managing a changing software system.
- The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

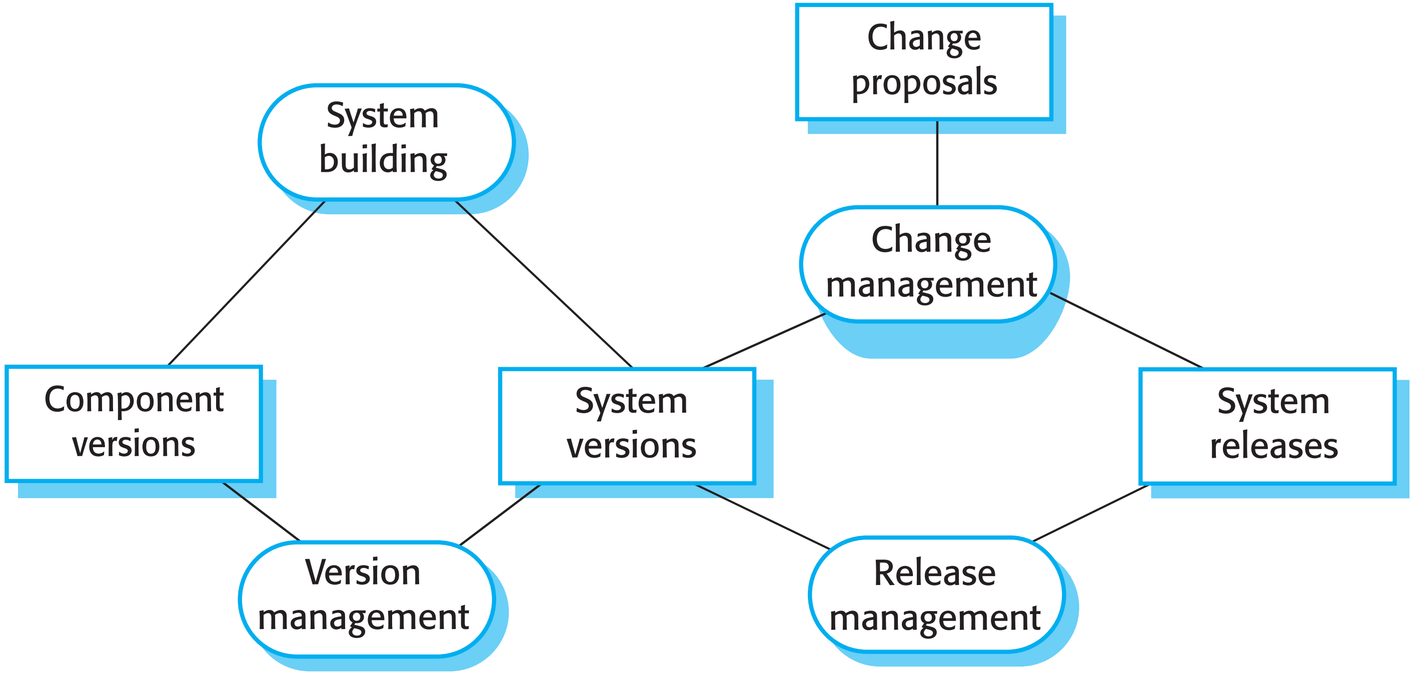
Configuration management activities



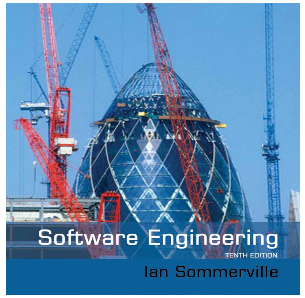
- Version management,
  - where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- System integration,
  - where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
- Problem tracking,
  - where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

Configuration management tool interaction



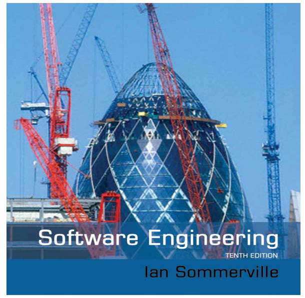


Host-target development



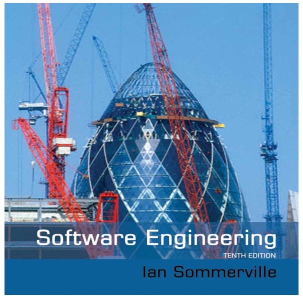
- Most software is developed on one computer (the host), but runs on a separate machine (the target).
- More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- Development platform usually has different installed software than execution platform; these platforms may have different architectures.

Component/system deployment factors



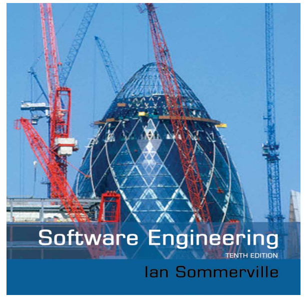
- If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
- High availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
- If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another.

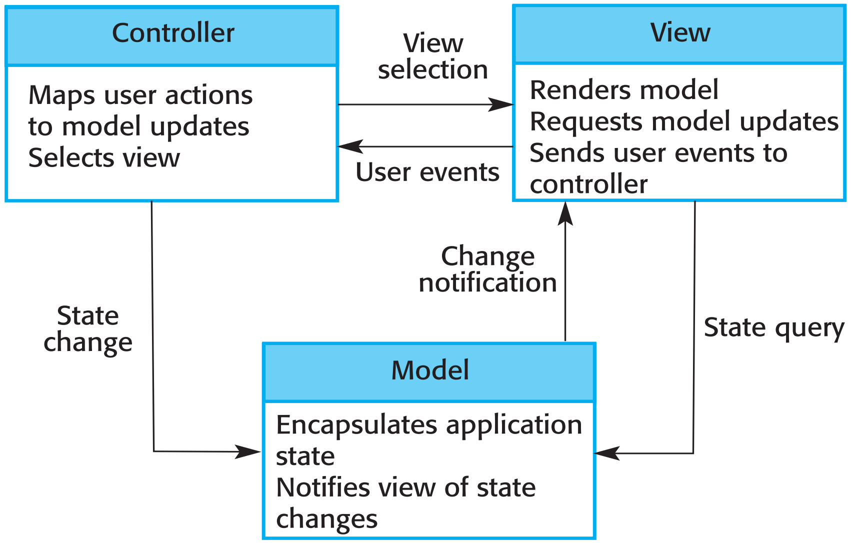
Architectural patterns



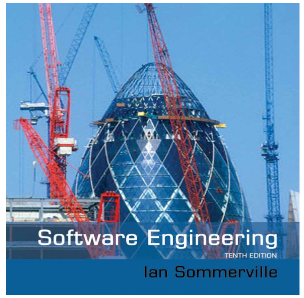
- Patterns are a means of representing, sharing and reusing knowledge.
- An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- Patterns should include information about when they are and when the are not useful.
- Patterns may be represented using tabular and graphical descriptions.

The organization of the MVC



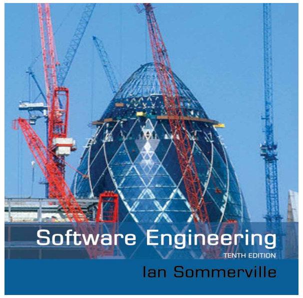


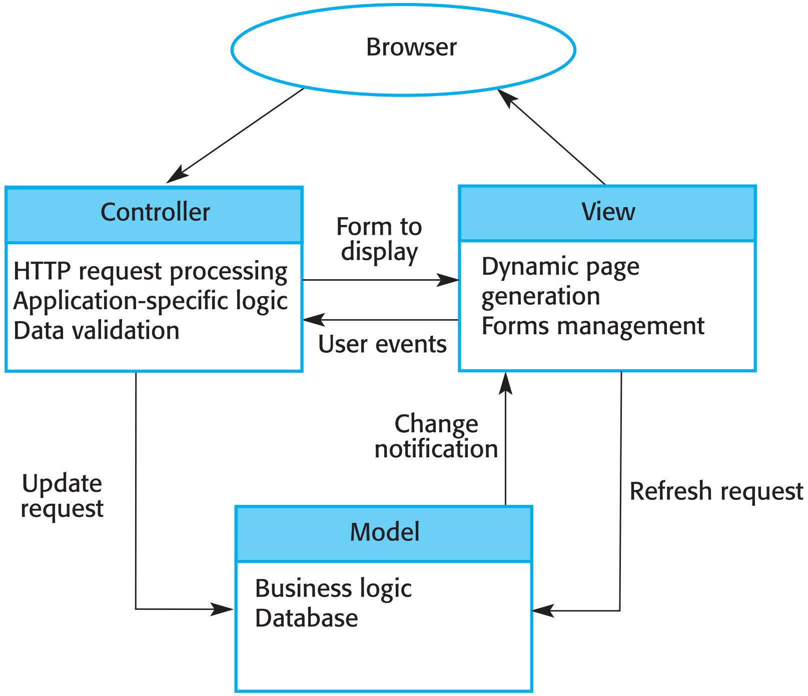
The Model-View-Controller (MVC) Design Pattern



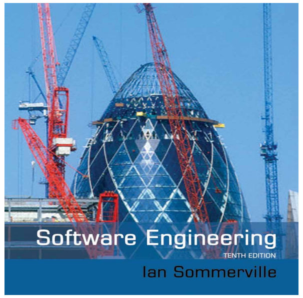
Name	**MVC (Model-View-Controller)**
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See previous slide.
Example	Next slide shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

Web application architecture using the MVC



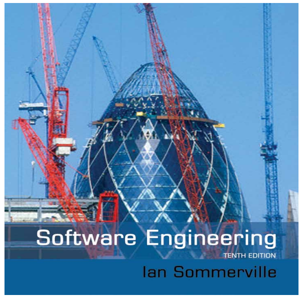


Layered Architecture



- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

The Layered architecture pattern



Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.
Example	A layered model of a system for sharing copyright documents held in different libraries.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

A generic layered architecture