S01 Software Development Tools

IDEs
Revision Control
Gitlab and Git

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

Integrated Development Environments

- An editor to do more than just *write* code.
  - Syntax highlighting, completion, continuous compilation, testing, debugging, packaging
  - Code analysis and refactoring capabilities
- Examples: Eclipse, IntelliJ, VisualStudio, XCode

Version Control (VCS, RCS, SCM)

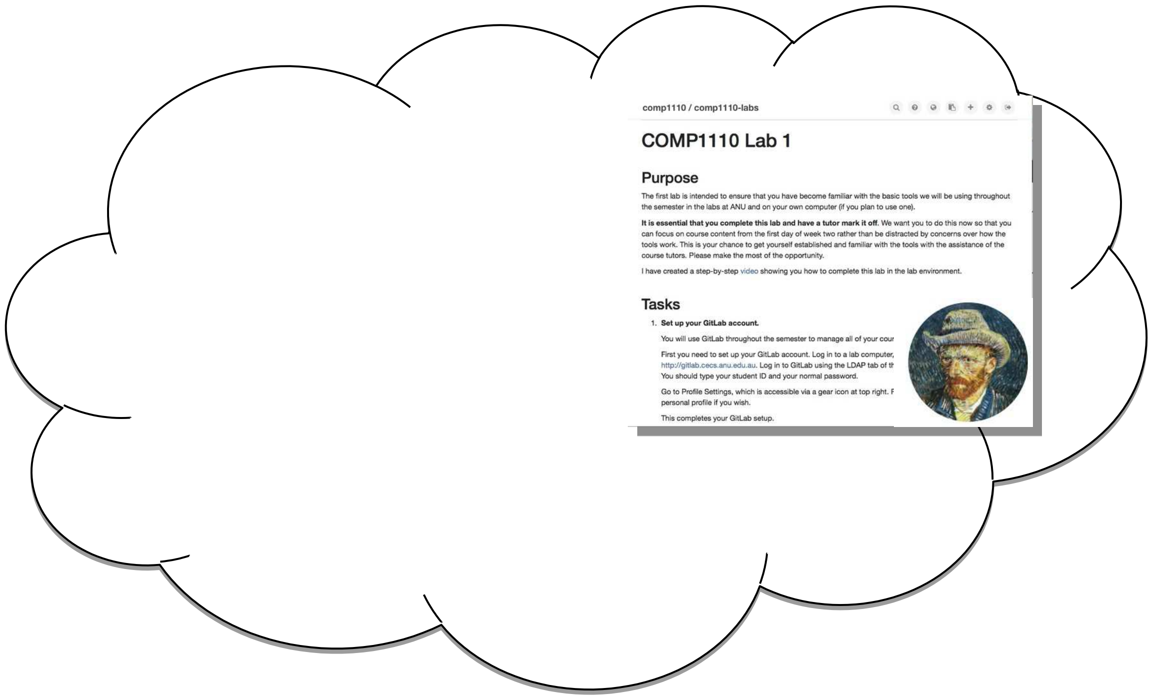
- Indispensable software engineering tool
- Solitary work
  - Personal audit trail and time machine
  - Establish when bug was introduced
  - Fearlessly explore new ideas (roll back if no good)
- Teamwork
  - Concurrently develop
  - Share work coherently

Git & Gitlab

- Distributed version control system
  - hg, git, and others
- Contrast with centralised version control
  - cvs, svn, others

We will use a distributed version control system – git – and a server – the ANU teaching gitlab – for sharing and submitting course work.

Git & GitLab



comp1110 / comp1110-labs

COMP1110 Lab 1

Purpose

The first lab is intended to ensure that you have become familiar with the basic tools we will be using throughout the semester in the labs at ANU and on your own computer (if you plan to use one).

It is essential that you complete this lab and have a tutor mark it off. We want you to do this now so that you can focus on course content from the first day of week two rather than be distracted by concerns over how the tools work. This is your chance to get yourself established and familiar with the tools with the assistance of the course tutors. Please make the most of the opportunity.

I have created a step-by-step video showing you how to complete this lab in the lab environment.

Tasks

1. **Set up your GitLab account.**

   You will use GitLab throughout the semester to manage all of your cour

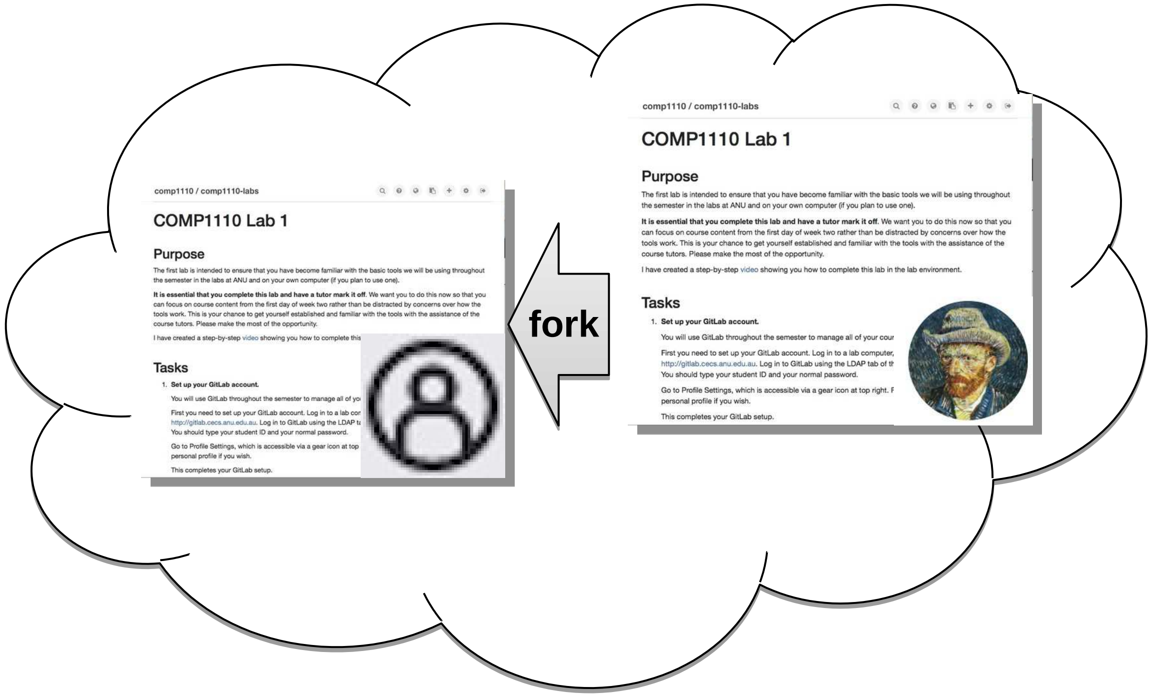
   First you need to set up your GitLab account. Log in to a lab computer, http://gitlab.cecs.anu.edu.au. Log in to GitLab using the LDAP tab of th You should type your student ID and your normal password.

   Go to Profile Settings, which is accessible via a gear icon at top right. F personal profile if you wish.

   This completes your GitLab setup.

(master) labs repo
(owned by comp1110)

Git & GitLab



comp1110 / comp1110-labs

COMP1110 Lab 1

Purpose

The first lab is intended to ensure that you have become familiar with the basic tools we will be using throughout the semester in the labs at ANU and on your own computer (if you plan to use one).

It is essential that you complete this lab and have a tutor mark it off. We want you to do this now so that you can focus on course content from the first day of week two rather than be distracted by concerns over how the tools work. This is your chance to get yourself established and familiar with the tools with the assistance of the course tutors. Please make the most of the opportunity.

I have created a step-by-step video showing you how to complete this lab in the lab environment.

Tasks

1. **Set up your GitLab account.**

   You will use GitLab throughout the semester to manage all of your cour

   First you need to set up your GitLab account. Log in to a lab computer, http://gitlab.cecs.anu.edu.au. Log in to GitLab using the LDAP tab of th You should type your student ID and your normal password.

   Go to Profile Settings, which is accessible via a gear icon at top right. F personal profile if you wish.

   This completes your GitLab setup.

comp1110 / comp1110-labs

COMP1110 Lab 1

Purpose

The first lab is intended to ensure that you have become familiar with the basic tools we will be using throughout the semester in the labs at ANU and on your own computer (if you plan to use one).

It is essential that you complete this lab and have a tutor mark it off. We want you to do this now so that you can focus on course content from the first day of week two rather than be distracted by concerns over how the tools work. This is your chance to get yourself established and familiar with the tools with the assistance of the course tutors. Please make the most of the opportunity.

I have created a step-by-step video showing you how to complete this

Tasks

1. **Set up your GitLab account.**

   You will use GitLab throughout the semester to manage all of yo

   First you need to set up your GitLab account. Log in to a lab cor http://gitlab.cecs.anu.edu.au. Log in to GitLab using the LDAP ta You should type your student ID and your normal password.

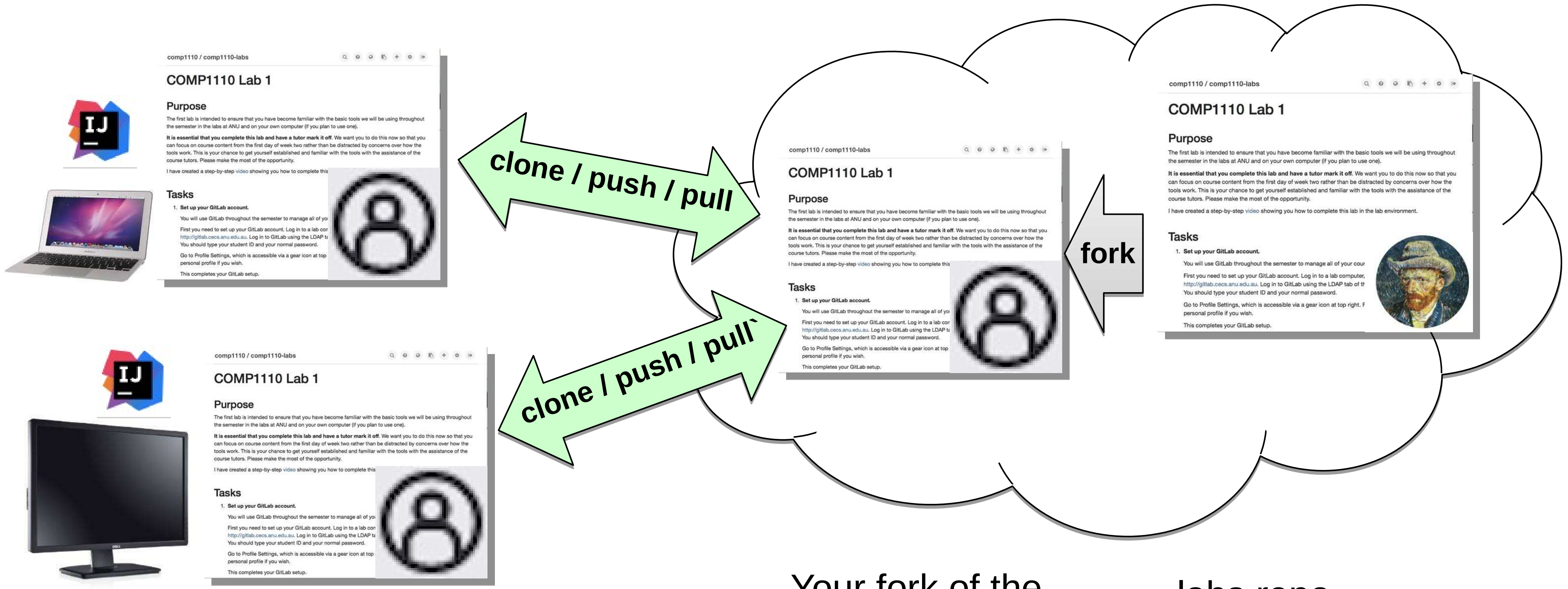
   Go to Profile Settings, which is accessible via a gear icon at top personal profile if you wish.

   This completes your GitLab setup.

Your fork of the labs repo (owned by you)

labs repo (owned by comp1110)

Git & GitLab



comp1110 / comp1110-labs

COMP1110 Lab 1

Purpose

The first lab is intended to ensure that you have become familiar with the basic tools we will be using throughout the semester in the labs at ANU and on your own computer (if you plan to use one).

It is essential that you complete this lab and have a tutor mark it off. We want you to do this now so that you can focus on course content from the first day of week two rather than be distracted by concerns over how the tools work. This is your chance to get yourself established and familiar with the tools with the assistance of the course tutors. Please make the most of the opportunity.

I have created a step-by-step video showing you how to complete this lab in the lab environment.

Tasks

1. **Set up your GitLab account.**

   You will use GitLab throughout the semester to manage all of your cour

   First you need to set up your GitLab account. Log in to a lab computer, http://gitlab.cecs.anu.edu.au. Log in to GitLab using the LDAP tab of th You should type your student ID and your normal password.

   Go to Profile Settings, which is accessible via a gear icon at top right. F personal profile if you wish.

   This completes your GitLab setup.

Clone(s) of your fork of the labs repo
(owned by local user)

Your fork of the
labs repo
(owned by you)

labs repo
(owned by comp1110)

Recap

- **Repository** (“repo”): A copy of a project and its history.
- **Gitlab**: A server (remote) that stores repos
  - ANU teaching gitlab: https://gitlab.cecs.anu.edu.au
- **Clone**: A working (local) copy of a repo.
- **Pull**: Fetch updates from a remote to a working copy.
- **Push**: Send updates from a working copy to a remote.
- **Commit**: An update to a repo.

IntelliJ Git Integration

- Clone an existing repository:
  - “Get from VCS” on splash screen
- Other operations:
  - Git menu
  - right mouse click > Git

S02 Revision Control

Git

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

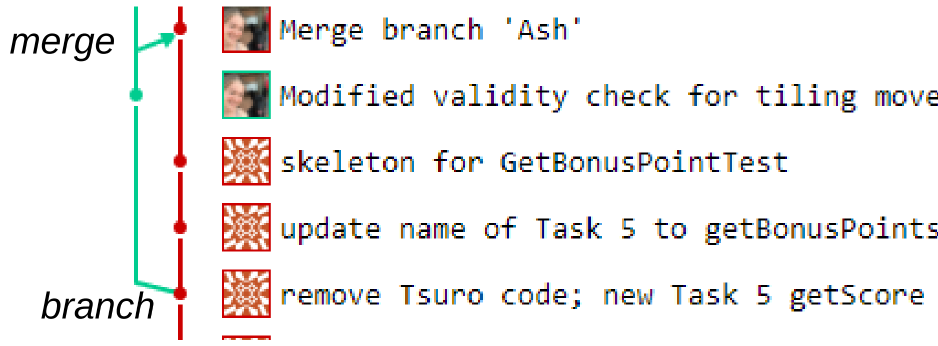
Git Concepts

- Commit (noun)
- Staging (IntelliJ allows you to more or less ignore this, so we will)
- ✓ Commit (atomically commit changes to your local repo)
- ✓ Push (push outstanding local changes to a remote repo)
- ✓ Pull (*fetch* new changes from a remote repo and merge / rebase locally)
- ✓ Update (in IntelliJ, otherwise known as checkout – update your working version)
- Merge / Rebase
- Reset and Revert

Git Commits

Captures a set of changes (e.g., modifications, additions, deletions) that may span multiple files.

- Globally unique commit ID (large hexadecimal number)
- Parent – child relationship
  - Single parent, single child is simple case
  - Multiple children indicates a **branch**
  - Multiple parents indicates a **merge**
- Commits are usually never deleted



```
Merge branch 'Ash'
Modified validity check for tiling move
skeleton for GetBonusPointTest
update name of Task 5 to getBonusPoints
remove Tsuro code; new Task 5 getScore
```

A Little More on Update

Update will by default take you to the “HEAD” (the most recent known commit).

You can, however, “update” to any particular revision, moving yourself back and forward in time. To do this, you need to specify the revision.

In IntelliJ you can do this by double-clicking on the revision (Git -> Show Git Log, select the revision right click “Checkout revision”)

Branches and Merging

- A **branch** occurs when a commit has more than one child.
- A **merge** is special commit with two parents (thus uniting branches).
- If branches are conflicting (changes to the same file), those conflicts must be **resolved** before merging.
- You can create named branches, and jump between them with update (**checkout**).

Rollback, Reset, Revert

- You can **rollback** (checkout) uncommitted changes (to “HEAD” state).
- You can **reset** your local HEAD state to any particular commit (throwing away un-pushed changes whether committed or not).
- You can also **revert** any particular commit. This amounts to applying a counteracting commit.

Amend and Rebase

WARNING: The following commands will cause trouble if they "modify" commits that have been previously pushed:

- You can amend a commit message, add more changes with **amend**.
- You can interactively remove, combine, reorder and edit commits with **rebase** *interactive*.

When All Else Fails

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

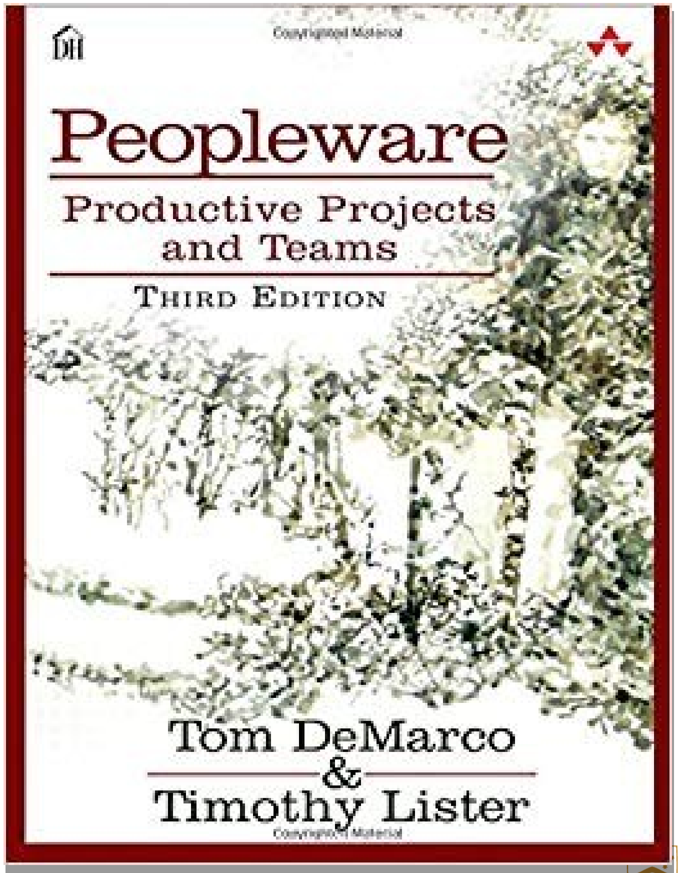
https://xkcd.com/1597/

S03 Software Development Teams

Importance of people in software engineering
Understanding team effectiveness
Conflict and conflict resolution
Code of conduct

Q: Why Do Software Projects Fail?

A: People



Understanding Team Effectiveness

Study of 180 google teams world-wide

- Gathered data on team members (attitudes, skills, personality, etc.)
- Identified factors that correlated with their measure of performance

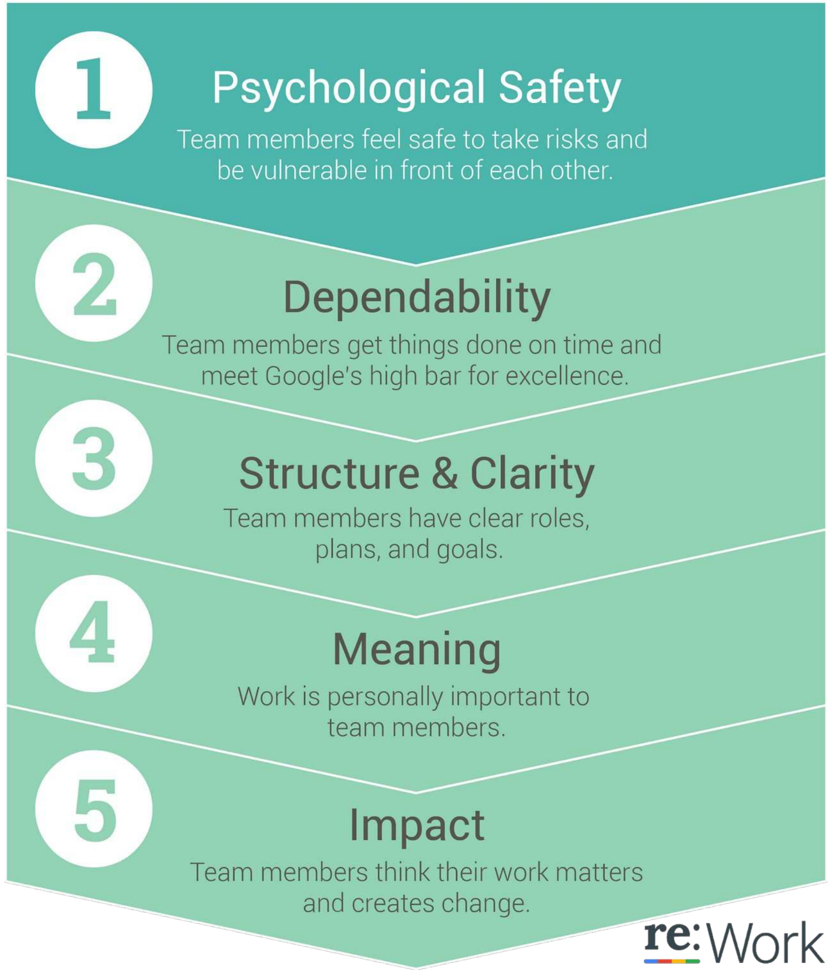
https://rework.withgoogle.com/guides/understanding-team-effectiveness/

Understanding Team Effectiveness

- Factors:
  - ~~Co-location of teammates~~
  - ~~Consensus-driven decision making~~
  - ~~Extroversion of team members~~
  - ~~Individual performance of team members~~
  - ~~Workload size~~
  - ~~Seniority~~
  - ~~Team size~~
  - ~~Tenure~~

These **did not** significantly impact the performance measure used by Google.

This does not mean that these are not important factors in other settings or other regards.



1 **Psychological Safety**
Team members feel safe to take risks and be vulnerable in front of each other.

2 **Dependability**
Team members get things done on time and meet Google's high bar for excellence.

3 **Structure & Clarity**
Team members have clear roles, plans, and goals.

4 **Meaning**
Work is personally important to team members.

5 **Impact**
Team members think their work matters and creates change.

re:Work

- Discuss up-front what you are aiming for in this project.
- Plan, and agree on your responsibilities.
  - Don’t forget to plan for contingencies!
- Contributions system
  - team mark x f(contrib) = individual mark
  - Team of 3, equal contrib (33/33/33): individual marks = team mark
  - Team of 2, equal contrib (50/50): team mark of 2/3 = max ind. mark
  - Contribution capped at 2/3 (66%).
  - Deciding contributions is the *team’s* responsibility.

Conflict Resolution Strategies

Conflict is a part of any work environment.

Working under stress is bound to cause problems.

- Stephanie Ray, 2018, 10 Conflict Resolution Strategies that Actually Work

1) Define Acceptable Behavior
2) Don't Avoid Conflict
3) Choose a Neutral Location
4) Start with a Compliment
5) Don't Jump to Conclusions
6) Think Opportunistically, Not Punitively
7) Offer Guidance, Not Solutions
8) Constructive Criticism
9) Don't Intimidate
10) Act Decisively

S04 Test-Driven Development

Test-driven development (TDD)
JUnit

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

Types of Tests

- **Unit tests**: testing individual “units” / “modules”
  - In OO a unit is at the level of a **method** or **class**
  - Check the “building blocks” are functioning correctly
- **Integration tests**: the integration of multiple modules
  - Expose problems with interface of modules and interactions between them
- **System tests**: end-to-end complete system
  - Checking it meets its requirements

Test Driven Development (TDD)

TDD "red, green, refactor"

1. Create test that defines new requirements
2. Ensure test fails
3. Write code to support new requirement
4. Run tests to ensure code is *correct****
5. Then refactor and improve
6. Repeat

Key element of *agile programming*

What Makes **Good** Unit Tests?

- Isolate behaviour / reduce dependencies
- Common path / usage
- Edge cases
- Touch on all branches
- Deterministic
- Limit false positives (test fails for correct code)
- Coverage

JUnit

Unit testing for Java

- Developed by Kent Beck
  - Father of extreme programming movement
- Integrated into IntelliJ
- Useful for:
  - TDD (Test driven development)
  - Bug isolation and regression testing
    - Precisely identify the bug with a unit test
    - Use test to ensure that the bug is not reintroduced

JUnit

- Methods marked with `@Test` will be tested
- When JUnit is called on a class, all tests are run and a report is generated (a failed test does not stop execution of subsequent tests).
- JUnit has a rich set of annotations that can be used to configure the testing environment, including:
- `@Test`, `@Ignore`, `@BeforeEach`, `@BeforeClass`, `@AfterEach`, `@AfterClass`, `@Timeout`
- JUnit can check that an exception is thrown if that is expected in a certain case

  -
    ```
    Assertions.assertThrows(
      ArithmeticException.class,
      () -> myMethod());
    ```

S05 Software Design

Software Complexity
Software Design

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

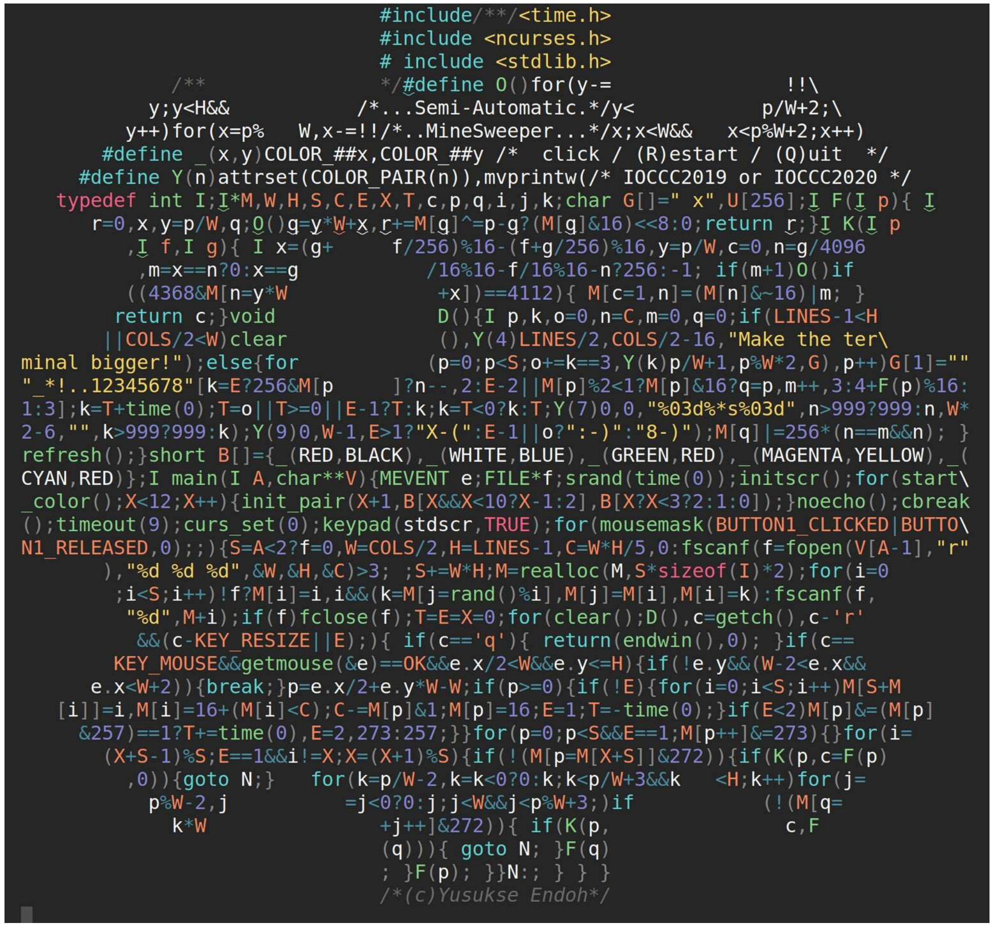
Software Complexity

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+
[<]<-]>>.>---.+++++++..+++.>>.<-.<.++
+.------.--------.>>+.>++.
```

- “Hello World” in the BrainF#@k language (apparently: source wikipedia)
- Syntax only 8 characters, *Turing complete*
- Simple or complex?

Software Complexity

- The International **Obfuscated** C Code Contest
- Yusuke Endoh one of the 2020 winners: Minesweeper Solver



```
                               #include/**/<time.h>
                               #include <ncurses.h>
                               # include <stdlib.h>
             /**               */#define O()for(y-=               !!\
           y;y<H&&           /*...Semi-Automatic.*/y<          p/W+2;\
         y++)for(x=p%   W,x-=!!/*..MineSweeper...*/x;x<W&&   x<p%W+2;x++)
       #define _(x,y)COLOR_##x,COLOR_##y /*  click / (R)estart / (Q)uit  */
     #define Y(n)attrset(COLOR_PAIR(n)),mvprintw(/* IOCCC2019 or IOCCC2020 */
   typedef int I;I*M,W,H,S,C,E,X,T,c,p,q,i,j,k;char G[]=" x",U[256];I F(I p){ I
      r=0,x,y=p/W,q;O()g=y*W+x,r+=M[g]^=p-g?(M[g]&16)<<8:0;return r;}I K(I p
         ,I f,I g){ I x=(g+     f/256)%16-(f+g/256)%16,y=p/W,c=0,n=g/4096
          ,m=x==n?0:x==g           /16%16-f/16%16-n?256:-1; if(m+1)O()if
         ((4368&M[n=y*W             +x])==4112){ M[c=1,n]=(M[n]&~16)|m; }
        return c;}void              D(){I p,k,o=0,n=C,m=0,q=0;if(LINES-1<H
       ||COLS/2<W)clear             (),Y(4)LINES/2,COLS/2-16,"Make the ter\
minal bigger!");else{for           (p=0;p<S;o+=k==3,Y(k)p/W+1,p%W*2,G),p++)G[1]=""
"_*!..12345678"[k=E?256&M[p    ]?n--,2:E-2||M[p]%2<1?M[p]&16?q=p,m++,3:4+F(p)%16:
1:3];k=T+time(0);T=o||T>=0||E-1?T:k;k=T<0?k:T;Y(7)0,0,"%03d%*s%03d",n>999?999:n,W*
2-6,"",k>999?999:k);Y(9)0,W-1,E>1?"X-(":E-1||o?":-)":"8-)");M[q]|=256*(n==m&&n); }
refresh();}short B[]={_(RED,BLACK),_(WHITE,BLUE),_(GREEN,RED),_(MAGENTA,YELLOW),_(
CYAN,RED)};I main(I A,char**V){MEVENT e;FILE*f;srand(time(0));initscr();for(start\
_color();X<12;X++){init_pair(X+1,B[X&&X<10?X-1:2],B[X?X<3?2:1:0]);}noecho();cbreak
();timeout(9);curs_set(0);keypad(stdscr,TRUE);for(mousemask(BUTTON1_CLICKED|BUTTO\
N1_RELEASED,0);;){S=A<2?f=0,W=COLS/2,H=LINES-1,C=W*H/5,0:fscanf(f=fopen(V[A-1],"r"
       ),"%d %d %d",&W,&H,&C)>3; ;S+=W*H;M=realloc(M,S*sizeof(I)*2);for(i=0
        ;i<S;i++)!f?M[i]=i,i&&(k=M[j=rand()%i],M[j]=M[i],M[i]=k):fscanf(f,
         "%d",M+i);if(f)fclose(f);T=E=X=0;for(clear();D(),c=getch(),c-'r'
          &&(c-KEY_RESIZE||E);){ if(c=='q'){ return(endwin(),0); }if(c==
        KEY_MOUSE&&getmouse(&e)==OK&&e.x/2<W&&e.y<=H){if(!e.y&&(W-2<e.x&&
      e.x<W+2)){break;}p=e.x/2+e.y*W-W;if(p>=0){if(!E){for(i=0;i<S;i++)M[S+M
   [i]]=i,M[i]=16+(M[i]<C);C-=M[p]&1;M[p]=16;E=1;T=-time(0);}if(E<2)M[p]&=(M[p]
     &257)==1?T+=time(0),E=2,273:257;}}for(p=0;p<S&&E==1;M[p++]&=273){}for(i=
       (X+S-1)%S;E==1&&i!=X;X=(X+1)%S){if(!(M[p=M[X+S]]&272)){if(K(p,c=F(p)
         ,0)){goto N;}   for(k=p/W-2,k=k<0?0:k;k<p/W+3&&k   <H;k++)for(j=
           p%W-2,j          =j<0?0:j;j<W&&j<p%W+3;)if          (!(M[q=
             k*W               +j++]&272)){ if(K(p,              c,F
                               (q))){ goto N; }F(q)
                               ; }F(p); }}N:; } } }
                               /*(c)Yusukse Endoh*/
```

What is Software Complexity?

- ***Accidental* Complexity**
  - Software that is designed or presented in a way that is more difficult for a **human to understand, use and modify** *than it needs to be*.
  - It is difficult to write elegant, clear, reusable code.
- ***Essential* Complexity**
  - Inherent to the problem being solved. Irreducible.
- **Not to be confused with** computational complexity.

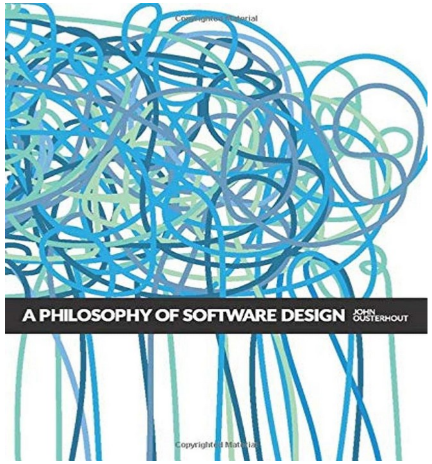
Software Complexity

- Some **contributing factors**:
  - Interlinking many components
  - Unstated assumptions
  - Non-local changes, unintuitive side-effects
  - Duplication / lack of encapsulation / exposure to details
  - Poor naming
  - Not following conventions / inconsistency
- Often **incrementally** works its way into a project, e.g., *feature creep*, dealing with *legacy*.

Good Software Design

- Many opinions. Conventions / preferences vary between communities.
- Recommendation:

  A Philosophy of Software Design, John Ousterhout



- Design principles
- Red flags

Some Principles (Ousterhout)

- **Deep “modules”** (method, class, package, or module)
  - Simple interfaces* (narrow)
  - Encapsulate lots of complexity (depth)
  - General-purpose
- Prefer **simple interface** over simple implementation
- Design **errors out of existence**
- Design for **ease of reading**, not ease of writing
- Extra: Don’t Repeat Yourself (**DRY**) and **SOLID** principles

* Interfaces in the broad sense, not just the Java keyword

Some Red Flags (Ousterhout)

- **Shallow module**: interface not much simpler than implementation
- **Overexposure**: user needs to be aware of rarely-used features
- **Repetition**: non-trivial code is repeated
- **Conjoined methods**: methods are so co-dependent that you have to understand implementation of both
- **Comment repeats code**
- **Hard to name entity**
- Extra: **Deeply nested control-flow blocks**

S06 Code Review

Code Review
Comments and Documentation

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

What is Software Complexity?

- ***Accidental* Complexity**
  - Software that is designed or presented in a way that is more difficult for a **human to understand, use and modify** *than it needs to be*.
  - It is difficult to write elegant, clear, reusable code.
- ***Essential* Complexity**
  - Inherent to the problem being solved. Irreducible.
- **Not to be confused with** computational complexity (about performance).

Code Review

- One or more people review code who are removed from the implementation.
- Commonly done for a specific change (e.g., set of git commits) but can also be done for a complete project / implementation.
  - **Fix** a specific bug
  - **Implement** a new feature
  - **Refactor** part of the code
- Gitlab offers a “merge request” workflow (“pull request” on github) where reviewers / maintainers review the changes **before** they are merged into the mainline branch.

Code Review Motivations

- Barrier to ensure project remains **maintainable**.
  - Improve implementation / quality.
  - Clarify code, double-check edge cases.
  - On-balance rejection of a feature (accidental or essential complexity).
- Second pair of eyes: potentially less biased, can consider bigger picture, can bring new insight.
- Effective way to **learn** a new code-base and a team’s processes / conventions. Highlights interrelated parts.
- Can catch some bugs before reaching production… but implementer really should have adequate tests developed and passing.

Doing a Code Review

- **Objective**: is it in scope of this project
- **Functionality** (for end-users and developers):
  - does it do what is intended
  - edge cases / bugs
  - might have to run code for UI changes etc
- **Tests**: present, appropriate
- **Complexity**: design minimises / encapsulates complexity
- **Good names**: convey information and not too long
- **Comments**: help to understand decisions and the why, not repeating code, appropriately documenting interfaces
- **Conformance** to project style guide / conventions.

Further Tips

- Be considerate.
- Point out things that are good!
- Clearly label *nitpicks* as such.
- No code is ever perfect. Tailor to circumstances:
  - flight control software
  - a game

Code Comments / Documentation

- **Class or method comments – always for** `public`
  - How to use, edge cases, side-effects, pre/post-conditions, invariants, explain abstraction, examples.
  - Should not leak the implementation details.
- **Implementation comments – as required**
  - Give intuition where implementation is non-obvious to a likely contributor / your future self
  - Highlight where edge cases are handled if hidden
  - Rationale for the design if not the obvious choice
  - Should not just repeat code