SE2 notes by Silvio Biasiol

Application design

EDD (Empathy Driven Design):

The design of a product or a service that focus on connecting emotionally the maker with his target users and the people around them. Differently from an engineering process in which there is no direct contact with the end user an user centered process focuses on the user and his needs. An example is the:

design thinking

Three modes:

- **empathize** it is the process of understanding people
  - interview : *observe* the user in his interactions, what he says and what he actually does. It can be useful to find *extreme users* (they often make a need apparent). *Engage* the user to speak asking questions like "Why?". Watch and listen, for example make the user complete a task, follow his steps and ask him why he is doing what he does. Do not suggest answers.
  - dig deeper: draw conclusion of your work on post-it, whiteboard etc. Get all informations in your head onto a wall, anything that captures impressions and information about your user and share it with your team.
- **define** this step is about making sense of the widespread information you have gathered, the goal is to create a meaningful and actionable problem statement
  - capture findings: develop an understanding of the type of person your *user* is. Try to synthesize the *needs* of the user and work to express *insights* you developed through the synthesis of the gathered information. For example you can create a *User need statement* (f.e. "I am seven-years-old and I hate doing homework because it takes me forever to finish.").
  - define problem statement: articulate a POV (point of view) by combining user, needs and insight.
- **ideate** is the mode of the design in which you concentrate on idea generation and create new solutions for your users.
  - sketch: step beyond obvious solutions and thus increase the innovation potential of your team. Surrounding yourself with inspiring related materials and embracing misunderstanding allow you to reach further than you could by simply thinking about a problem. A technique is directly building, prototyping. The key point is to separate the generation of the ideas from the evaluation of ideas. An useful technique is the HMW (How Might We...)
  - generate a new solution once you have the ideas design a 2-3 voting criteria and carry the two or three ideas that receive the most votes forward into prototyping.

- **prototype** the generation of artifacts intended to answer questions that get you closer to your final solution. A prototype can be anything that the user can interact with - be it a wall of post-it notes, a gadget you put together or even a storyboard.
    - <u>build</u> The useful part of prototyping is that you can test the possibilities and fail quickly and cheaply. The point is to not spend too long on a prototype and to build with the user in mind.
- **test** is when you solicit feedback, about the prototypes you have created, from you user, it is an opportunity to learn and understand you user again. Always prototype as if you know you're right, but test as if you know you're wrong! Ideally you can test within a real context of the user life. Ask people to use your solution in their normal routine or similar. Show but don't tell, watch how the user use (and misuse) your prototype. Ask users to compare if you have multiple prototypes.
- **iteration** iterate both by cycling through the process multiple times and also by iterating within a step

Questions:

- Descrivere a grandi linee il processo di design thinking
- In cosa consiste la fase "define" nel processo di design thinking, che scopo ha e come si realizza?
- In cosa consiste la fase "prototype" nel processo di design thinking, che scopo ha e come si realizza?
- In cosa consiste la fase "test" nel processo di design thinking, che scopo ha e come si realizza?
- In cosa consiste la fase "empathize" nel processo di design thinking, che scopo ha e come si realizza?
- In cosa consiste la fase "ideate" nel processo di design thinking e che scopo ha?

Agile development

Scrum

In a nutshell:
- **Split your organization** in small, cross-functional, self-organizing teams
- **Split your work** in a list of small deliverables. Sort the list by priority and estimate relative effort for each of them.
- **Split time** into short fixed-length iterations, with potentially shippable code after each iteration
- **Optimize the release plan** and update priorities based on insights gained inspecting the release after each iteration
- **Optimize the process** by having a retrospective after each iteration

So basically instead of having a big group spending a long time building a big thing we have many small teams spending short time buildings small things but integrating regularly to see the whole.

Scrum roles and teams

Scrum prescribes 3 roles:

- **Product owner** sets product vision and priorities
- **Team** implements the product (c.a. 7)
- **Scrum master** removes impediments and provides leadership

A scrum team should be able to eat all together a large american pizza without leaving anybody hungry :D

Scrum iterations (also called Sprints)

Scrum is based on timeboxed iterations which are composed by a cadence of planning, process improvement, and release:

- **Beginning of iteration** An iteration plan is created, i. e. team pulls out specific number of items from the product backlog based on the product owner's priorities and how much the team thinks they can complete in one iteration.
- **During iteration** Team focuses on completing items they committed to, The scope of iteration is fixed.
- **End of iteration** Team presents working code to the stakeholders, ideally potentially shippable. Then the team does a retrospective to discuss and improve their process.

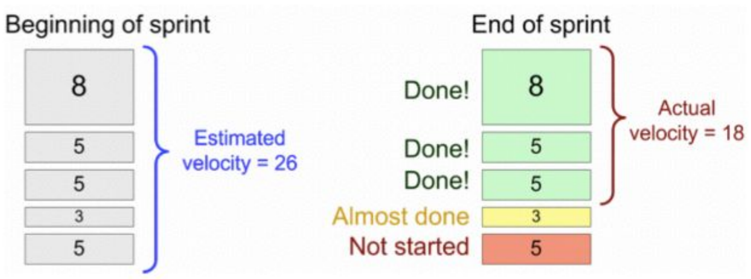
Sprint planning meeting

It generates:

- a sprint GOAL
- List of team members
- The sprint backlog
- Demo date
- Time and place for the daily meetings

Estimating velocity

focus factor = velocity / person days
Example: If my team has 7 members who are productive for 6 days each, and as a team they have a velocity of 31, then the focus factor is calculated as:
focus factor = 31 / (7 * 6) = 0.74



This focus factor can now be used to forecast the deliverables for the future iterations. So if only five members are available during an iteration, the achievable story points are calculated as:

estimated velocity = person days x focus factor = 0.74 * 5 * 6 = 22 [story points]

Sprint rules

No one outside the team can provide directions. Team members have the responsibility to:

- attend Daily Scrum Meeting
- keep the status of Sprint backlog (the table with to do , done , ...) up to date
- account for the job done and the one to be done

XP in Scrum

Collective ownership of the code: if somebody as introduced a bug is in your interest to fix it. The point is to write good code, the testing occurs throughout all the development process. Some new coding practices are introduced (such as pair programming and code reviews)

TDD (Test Driven Development)

It is a software development practice in which you write automated tests and then all the code you produce is oriented in passing those tests. Once your code passes those tests you refactor (improve the internal body of the code (add comments, documentations, write it better etc.) basically change the structure but not the functionality) it to make it meet some quality standards.

Pair programming

Don’t force it or use it full time is very tiring. In a nutshell 2 programmers 1 machine.

Scaling Scrum

One product, one backlog. The backlog should be manageable, people also can be shared across teams. Using scrum in distributed teams anyway will lead to lack of bonding and communication (you can’t turn your chair and speak f2f).

User story

Composed by:

- story name
- descrition
- how to demo
- story points

i.e

story name	description	how to demo	story points
Logout	As a logged in user, i want to be able to log-out	if not already logged in, go on trentoUdacity.com, login (credentials xxx and yyy), and then click on the logout button on the top right. the system goes to the home page trentoudacity.com and there is no name appearing in the top right part of the page. the menu item "my courses" is also gone.	2

Kanban

The key idea is to limit the WIP (Work In Progress). The only thing that Kanban prescribes is:

- **Visualize the workflow** split the work in pieces, write each item on a card and put on the wall
- **Limit WIP** assign explicit limits to how many items may be in progress at each workflow state
- **Measure lead time** (average time to complete one item), optimize the process to make lead time as small a predictable as possible

So how to decide the Kanban limits?
Too low Kanban limit → idle people → bad productivity
To high Kanban limit → idle tasks → bad lead time

technique/sector	Kanban	Scrum
velocity and prioritization	• Minimize lead time • estimation not prescribed • TODO column with items of equal importance • No prescription	• Sprint backlog must fit in a sprint • estimation required • totally ordered backlog • daily meetings and burndown charts
change	change is ok as long as you don't put more items in the backlog (todo column) than the limit	resist change within an iteration (if for example someone wants to add something it will be added to the backlog and maybe worked on in the next iteration)
WIP limit	per workflow state	per iteration
board	is persistent	resets every iteration
cross functional team	one per each board	optional

Questions:

- Quali sono le differenze principali tra scrum e kanban?
- Descrivi i punti fondamentali del processo software Scrum
- Come mi devo comportare, come regola generale, se mi accorgo di essere in ritardo in una sprint scrum?
  - chiedo uno sforzo aggiuntivo al team (lavoro straordinario) perché' dobbiamo rispettare i tempi
  - estendo la sprint in modo da consegnare comunque tutte le storie del backlog

    - finisco la sprint avendo completato un sottoinsieme delle storie pianificate inizialmente
    - aggiungo persone al team in modo da procedere più' velocemente
- Quale fra questi NON e' un ruolo scrum
    - Team
    - ScrumMaster
    - Product Manager
    - Product Owner
- Quale fra questi meeting non è' parte di scrum
    - Sprint planning meeting
    - Product review meeting
    - Sprint retrospective meeting
- Qual è' la dimensione "ideale" di un team di sviluppo software secondo scrum?
    - 10-12 persone
    - 5-7 persone
    - due persone, perché' da tre in su c'è' troppo overhead di comunicazione
    - 15-20 persone, per poter coprire tutti i ruoli necessari in un progetto con persone di opportuna competenza
- Cosa si intende per velocity in scrum, come la posso stimare a priori e come la calcolo a posteriori?
- Che cosa si intende per "focus factor" in un team scrum e come si calcola?
- Scrivi una user story a scelta (ma NON relativa a login/logout, e comprensiva di demo scenario, o "how to demo") relativa ad una applicazione tipo chat/messaging, quali whatsapp, facebook messenger, skype, etc

Versioning and collaboration

Git commands:

Useful overview of commands here:
https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud/git-branch-to-merge

git init initialize the repository locally
git clone <url> clone a repository in the current working directory
git remote add origin <url> adds the current repository to a remote one
git commit -am "...." add a new commit with the ... description
git log see the commits history
git pull --all pulls changes from remote repository
git push -u origin master pushes the local modifications to the remote repo (-u means that the push will be sent to the master branch)
git status show the status of the repository (files changes → staging area etc.)
git branch <branch_name> makes a branch, to begin working on the new branch, you have to check out the branch you want to use:
git checkout <branch_name> to start working on that branch (to return to the default one is git checkout master)

git add <filename/directory_name> add the file/directory to tracked files
git merge <branch_name> automatically merges the changes in the branch to master
git branch -d <branch_name> deletes the branch with branch_name
git reset --hard to undo local changes but not remove your last commit
git reset forget the staged files

Questions:

- A cosa servono strumenti tipo Git e github nel software development?
- Cos'è una "fork" in Github?
  - A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.
- Che differenza c'è tra git e github?
  - Git is a version control system, a tool to manage your source code history. GitHub is a hosting service for Git repositories. So they are not the same thing: Git the tool, GitHub the service for projects that use Git.
- Cos'è la staging area in git e a cosa serve?
  - The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit.
- A cosa serve una "branch" in git, quando si crea una nuova branch?
- Cosa significa fare checkout in git?
- Quando è' opportuno fare un git commit?
  - A intervalli regolari, ad esempio, ogni ora
  - quando ho completato un insieme di modifiche (nuova feature, o un bug fix)
  - quando ho portato il mio codice a un punto al quale voglio poter tornare, se serve
  - quando devo interrompere il lavoro (ad esempio, la sera

  A good rule of thumb is to make one commit per logical change. For example, if you fixed a typo, then fixed a bug in a separate part of the file, you should use one commit for each change since they are logically separate. If you do this, each commit will have one purpose that can be easily understood.

Software testing

Questions:

- Quali sono i motivi per cui una test case può fallire?
  - ci sono difetti nel sistema operativo o nell'hardware
  - il codice contiene bugs
  - il test case è sbagliato
- Cos'è la "fault injection" e perché la si usa?
- Un'azienda dà al suo staff un regalo di natale. Il regalo dipende dal numero di anni di permanenza nell'azienda, come segue:

    - Niente se < 12 mesi
    - playstation per da 12 a 23 mesi (esclusi)
    - 58” TV da 24 a 47 mesi esclusi
    - Porsche da 48 mesi in su

  L’applicazione che calcola il premio accetta solo numeri interi non negativi come input, quale “equivalence partition” ci servono per i test cases sul calcolo del regalo di natale?
    - 5 equiv. part.
    - 4
    - 3
    - 2
- Una funzione processa come input un valore di anno di nascita che è compreso tra 1900 e 2004 (inclusi). I boundary values per il test di questa funzione sono:
    - 1899,1900,2004,2005
    - 0,1900,2004,2005
    - 1900,2004
    - 1899,1900,1901,2003,2004,2005
- Uno dei campi di una web form è una textbox che accetta lettere dell’alfabeto, maiuscole o minuscole. Identifica quali valori appartengono a classi di equivalenza non valide
    - TESTING
    - tESTING
    - TEsting
    - TEst67g
- Quali fra queste NON è di norma un obiettivo del software testing?
    - stabilire se il software è pronto per una release
    - trovare errori nel software
    - accertarsi che il software non contenga errori
- Cosa si intende per regression testing?
- Quali fra queste affermazioni su equivalence partitioning sono corrette?
    - divide lo spazio di dati in input in partizioni, in modo tale che un elemento di una partizione sia rappresentativo dal punto di vista del testing, di tutta la partizione
    - per un testing accurato non posso avere meno di 3 classi di equivalenza
    - i test cases devono includere almeno due valori per ogni equivalence partition
- Quali sono le fasi tipiche di un approccio scientifico al debugging?
- ...

Udacity course about testing (parts 1 and 2)

test input (acceptability test)→ software under test → test outputs → ok ? cool : DEBUG
Bugs can be in:

- acceptability check
- SUT (Software Under Test)
- specification

- in OS, compiler, libraries,hardware
- ?

With a single input we can infer that if our test runs with it then it can run with a wider range of inputs (a domain) but being careful of the assumptions that we are making.

Creating testable software

- clean code
- refactor
- always be able to describe what a module does and how it interacts with other code
- no extra threads
- no swamp of global variables
- no pointer soup
- modules should have unit test
- when applicable support for fault injection
- assertions

Good test cases

If you have an interface, APIs.. then we have to test the APIs with all the possible representable values.

Fault injection

In software testing, fault injection is a technique for improving the coverage of a test by introducing faults to test code paths, in particular error handling code paths, that might otherwise rarely be followed. For example if I have a function

file = open(“tmp/foo”,”w”)

I’ll substitute it with :

file = my_open(“tmp/foo”,”w”)

the last one is almost identical to the first one but it will fail rarely, depends on the failure rate that can be effective in testing our SUT. In a nutshell we basically inject faults in the code 3:)

Timing of inputs

Sometimes the timing of inputs is important, so we have to check and send input with different times, take as an example the Therac 25 (the radiation killer machine)

Kinds of testing

- **white-box testing** the tester is using detailed knowledge about the internals of the system in order to construct better test cases
- **black-box testing** we don’t have detailed knowledge of the system
- **unit testing** we test small units in an isolated fashion. It’s objective is to find errors in the internal logic of the unit. It is used with *mock objects* that interact with the unit faking the system
- **integration testing** refers to taking multiple software modules that have already been unit tested and testing them in collaboration with each other.

- **system testing** does the system as a whole meet its goals? Usually done in combination with black-box testing (because the system may be too complex to use white-box testing)
- **differential testing:** we take the same input and we deliver it to two implementation of SUT and comparing them for equality
- **stress testing** we test the SUT due to assess the robustness and reliability, if for example we have a server we send to it a lot of requests.. or the Boeing 747 wings test
- **random testing** we use the results of a random number generator to chose random inputs to send to the SUT for example the crashme program of UNIX
- **A/B testing** it is the differential testing applied to the web interfaces to measure feasibility and other cool stuff
- **regression testing** is a type of software testing that verifies that software previously developed and tested still performs correctly even after it was changed or interfaced with other software.

Coverage metrics (also containing material from slides)

Test coverage allows us to assign a score to a collection of test cases but not necessarily gets all the bugs! The partitioning of the input domain is not a good way to spot bugs, better to use test coverage, *it is an automatic way of partitioning the input domain with some observed features of the source code*:

- **function coverage** we try to make test cases that call all the functions in our SUT. For example if with our test case we call 181 function of 250 total number of func. in our SUT than we got a score of 181/250. It is useful to spot dead code (functions that cannot be called or similar)
- **statement coverage** it is measured by the fault. It basically counts the lines of code being executed with certain inputs. For example if I have:

  ```
  if x > 1:
      x += 1
  if y > 1:
      x += 1
  ```

  if we have x=0, y=0 we get 50% statement coverage
  x=2, y=0 we get ¾ → 75% st. cov.
  x=2, y=0 and x=0, y=2 we get 100% st. cov.
- **branch coverage (also called decision coverage)** A branch is the outcome of a decision, so branch coverage simply measures which decision outcomes have been tested. Basically we have to get at each boolean operation both branches (the true one and the false one). Using the previous example: if we have x=2, y=2 we get 100% statement coverage but only 50% of branch coverage because the x<1 and y<1 branches were not executed, so in order to get 100% branch cover we should have x=2, y=2 and x=0, y=0
- **loop coverage** i just counts the number of times that an instruction was executed in a loop, it can be executed 0 times, 1 time, or multiple times. To get 100% loop coverage every instruction must be executed 0,1 and multiple times.

- **MC/DC coverage (Modified Condition Decision Coverage)** it is used in avionics to test the most critical parts of software. Basically is a branch coverage in which each condition in a decision takes every possible outcome
- **path coverage** In this the test case is executed in such a way that every logical path is executed at least once
- **boundary value coverage** is where test cases are generated using the extremes of the input domain, e.g. maximum, minimum, just inside/outside boundaries, typical values, and error values. It is similar to Equivalence Partitioning but focuses on "corner cases".
- **concurrent and synchronization coverage** tests concurrent programs stopping and reloading them with different time ranges

What about code not covered? 3 Possibilities:

- unfeasible code
- not worth covering
- incomplete test suite

Professor slides

Slides on test planning

It divides in

- **test plan** which defines the scope of work to be performed, it addresses
  - test strategy (which tests, how complete they should be, enviroment, ..)
  - test duration
  - test readiness criteria
  - test completion criteria
- **tests** composed by
  - collection of test scripts + manual tests
  - each test should return the expected result
  - each test document should contain a copy of all the tests used (to make them reusable)
- **test report** documents what occurred during tests, should contain
  - complete copy of each test script with proof of execution
  - copy of SRP (software problem report) with resolution
  - list of open unresolved SPRs
  - regression tests executed

Slides on whitebox testing

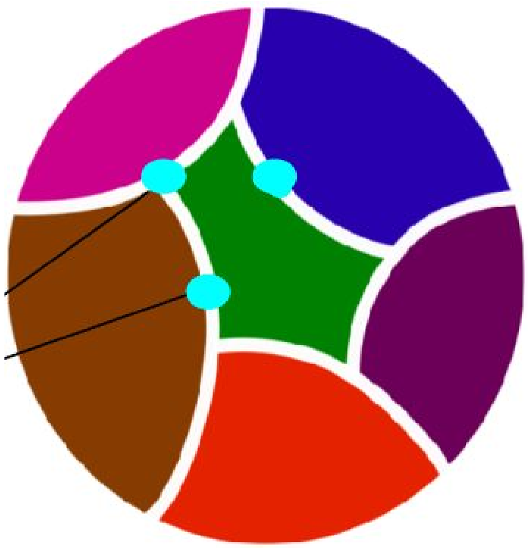
look at coverage metrics

Slides on testing

Testing is the process of executing a program with the intent of finding bugs

Test cases

A test case is a set of *input values*, *expected output*, and *preconditions* for executing a test with the aim of finding bugs. Ex. for sum(x,y) a test case might be sum(3,2) should give 5. Obviously a test case fails if it gets an output different from the expected one.



Equivalence partitioning (the partitions are the coloured zones in the image)

It is a software testing technique that divides the input data of a software unit into partitions of equivalent data from which test cases can be derived. So you can assume that:

- the program behaves analogously for inputs of the same partition (class)
- a test with a representative value for the partition is sufficient

Boundary values (the turquoise points in the image)

Testing boundary conditions of equivalence classes is more effective. An example strategy can be:

- choose an arbitrary value/s in each eq.class
- choose values near border of eq. classes
- test them

Udacity course about Javascript testing (complete)

Failing a test is not bad! The point is to make expectations and then validate them. These are the steps:

1. define expectations (results, errors...)
2. refactor the code in order to pass the tests

Jasmine

There are three basic constructs:

- **expect** each test starts with a call to expect which accepts a single actual value, than there is a matcher function:

  ```
  expect(<actual_value>).<matcher_function>;
  expect(<actual_value>).not.<matcher_function>;
  ```

  for ex.

  ```
  expect(add(2,3)).toBe(5);
  expect(add(2,3)).not.toBe(10);
  ```

- **it** is used to identify a specification or spec. A spec can contain multiple tests but to be validated all of the tests inside it must return true. for example:

  ```
  it("should be able to play a Song", function() {
      player.play(song);
      expect(player.currently Playing Song).toEqual(song);
      //demonstrates use of custom matcher
  ```

```
                expect(player).toBePlaying(song);
         });
```

- **describe** is used to identify a suite which is a group of related specs

```
        describe("Player", function() {
        <various specs here>
        });
```

Jasmine also defines a function called **beforeEach** function that will be executed before each and every one of our tests. Ex.

```
    beforeEach( function() {
        <initialization shit goes here>
    });
```

The red green refactor cycle

You write your tests first and than the code to pass them (so at the beginning they will all fail (red) and when you implement the fuck they will pass (green)).

Test asynchronous code

In asynchronous code the test (our expectation) can run before the function can complete its task. To fix this problem we can use the function beforeEach and a new function called **done** that signals to the framework when an asynchronous function has completed and the tests can be run. Ex.

```
describe('My object", function() {
    beforeEach( function( done) {
        myFunction.doStuff( function() {
            //init stuff goes here
            done();
        });
    });
    it('should do some neat stuff', function() {
        expect(<actual_value>).<matcher_function>;
        done();
    });
});
```

Udacity class on debugging (lesson 0,1,2)

Debug should be systematic and automatic. Debugging follows a scientific method composed by a series of experiments that allow us to gradually refine hypotheses until we understand why our program failed. Debugging is extremely expensive usually testing and debugging takes 50-75% of a software developing process.

The term bug is from the Harvard Mark II machine that got a relay broken by a moth (falena) stuck in it in 1947. The bug is now exposed in a museum LOL

- fix the problem not the symptom
- understand what the program does

- proceed systematically

failure error that is visible by the user
defect/fault/bug what turns an error in something that doesn't work
The aim of debugging is to find the defect that cause the failure/problem. The statement that has a defect can cause an infection (cascade effect in the program). The problem is that you may have a defect but not a failure and in that case you're not gonna find the defect.

Using the scientific method is a good way to debug:
initial observation → hypothesis → prediction → experiment → observation → diagnosis

Use assertion and bla bla bla

Writing good code

Udacity course on Js patterns

Requirements can change at all times. The key is to get an application that is

- bug free
- clearly written
- easily scalable
- extensible

Minimize our connections can be an effective way:

- **model** this is where all of the data is stored
- **view** all the stuff the user sees
- **controller** the connection in between model and view. It should allow to change the view/model without the need to change the other one

tb continued

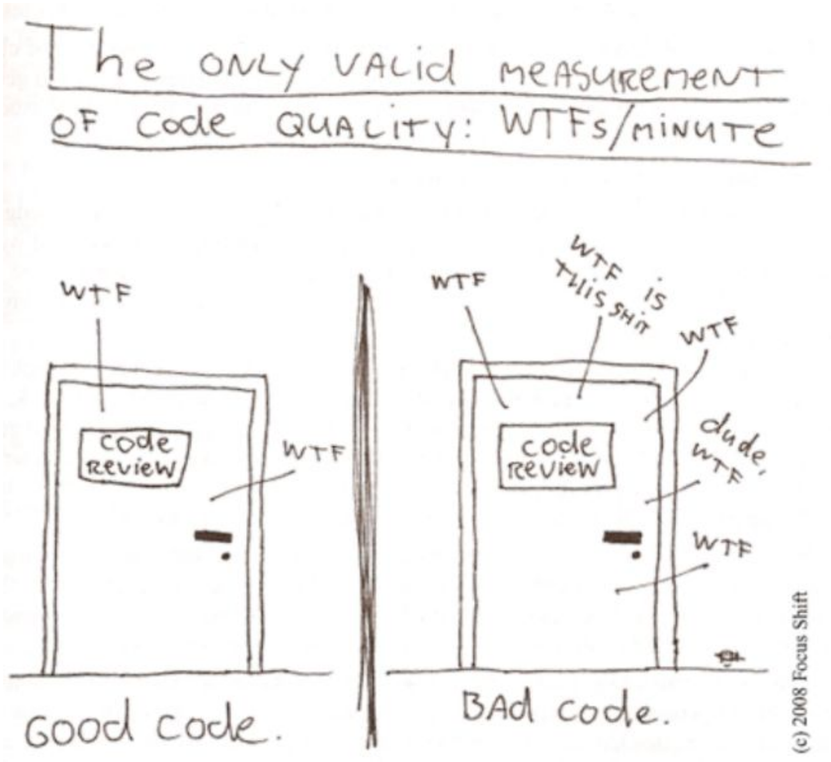
The quality software (prof. slides)

Internal and external qualities.

Product qualities

- **correctness** sw is correct if it satisfies the functional requirements specifications
- **reliability** pragmatically it means that user can rely on it, it is the probability of perform an operation without failure
- **robustness** sw behaves "reasonably" even in unforeseen circumstances
- **performance** speed in performing functions, responding to user inputs, etc.
- **usability** if the system is user friendly, intuitive..
- **generality** given a problem p the sw resolves a more general one including even p, this is useful for:
- **reusability** how parts of the sw can be reused
- **portability** multiplatform
- **maintainability** ease of maintenance, three categories of maintenance:

    - *corrective* removing residuals errors
    - *adaptive* adjusting to changes
    - *perfective* quality improvements
- **evolvability** basically continuous maintenance



Good code

Has these characteristics:

- **stroustrup** elegant, straightforward
- **minimal APIs/dependencies**
- **includes testing**
- **readable and consistent**
- **consistent naming of variables** i.e. not a, b but mothlyPay, payPerHour..
- **no side effects**
- **short functions** 20 lines average, few arguments
- **readable names**
- **keep code short and remove old code**
- **information hiding** make class and their members as inaccessible as possible (for testing purposes)
- **avoid global variables**
- **don't be Luca**