1.1 We have looked at some **static analysis tools** like StyleCop, PMD, FindBugs and SonarLint. Explain how static analysis can improve code quality. Explain how it helped you or could have helped you in your project.

Static analysis tools can help us with improving our code by analysing it before it runs. It points out potential bugs, memory leaks, conventions, coding-styles, typos, bloated classes, cyclomatic complexity and so on. If we follow the suggestions made to us by the tool, we end up with a code base that is more uniform, which could help make it more readable

- StyleCop, PMD, FindBugs

  Static Analysis tools as the ones mentioned can help us get rid of Code Smell from our code base. It finds errors or bad practises in our code, such as not closing implementations that implement the Closeable interface, or double assigned variables.

- Linters, SonarLint

  Linters go an extra step and other than just detecting bad practises, it also detects typos or inconsistent indentation.

- Security

  An SA tool will also point out if passwords are empty or plain strings, as this can cause security issues if we were to push it to a version control system.

- Code smell

  Can be anything from code standard that is not being followed or forgotten, to breaking design patterns. Eg: A class that does too much, a class that does too little, inconsistent variable naming convention, cyclomatic complexity is too high,

- Technical debt

  Technical debt arises when we miss or ignore Code Smell.

1.2 Explain test levels, and what characterizes the individual levels. Then, relate to your own project.

- unit testing

    - *Det niveau, der er tættest på implementeringen.*
    - *Koden deles ind i units, som testes hver for sig, og sikres kun at gøre lige præcis dét den er tiltænkt.*
- integration testing
    - *Tester integrationen mellem units.*
    - *(Der kan være gråzoner mellem integration- og systemtests. Nogle gange er en integration så stor, at den må betragtes som et system.)*
- system testing
    - *Man tester systemet som helhed. Lidt ala en kæmpe integrationstest. Bortset fra at en system-test er black box og integrations test er white boks.*
    - *Verificerer at HELE systemet virker*
- load testing
    - *Finde ud at hvordan softwaren opfører sig, når den bliver eksponeret for fx mange samtidige brugere.*
- static testing
    - *Kører ikke kode.*
        - *Det betyder også at man kan køre statisk test på alle typer dokumenter; rapporter, planer, marketing materiale -- testplaner, projektplaner ...*

- Acceptance testing
    - *"Øverste lag" i test levels.*
    - *"Bruger"-test. (Udføres af QA frem for bruger.)*
    - *Tester at krav er opfyldt.*
    - *Selenium (browser driver interface) ıı BDD (cucumber)*

1.3 Explain what kinds of tests can be carried out without running any code. Explain how it can be used on non-code documents as well.

Static testing kan blive brugt uden at køre kode.

Kan bruges på andre ting end kode fx. strategier, planer, rapporter & mange andre ting

Gælder om at finde fejl i systemet

Flere forskellige standarder, vi fokusere på IEEE 1028 da den er bedst til software

IEEE 1028 har flere forskellige test niveauer:

- **Reviews**

    Reviews/informal reviews

    Hvor man tilspørger en medarbejde om de kan kigge ens kode igennem

    Uformelt

    Typisk bliver der kigget på grammatik, stavning/navngivning, struktur og indhold

- **Walk-through**

    Stadig uformelt, men i mødeform

    Forfatter/udvikler gennemgår sin kode step-by-step

    Er mere om man har forstået sin opgave “Er det rigtigt forstået at…?”

- **Technical reviews**

    Formelt

    Gennemgåelse af teknisk dokument

    White-box testing teknik

    Forgår uden en “manager”

    Forfatter/udvikler styrer ikke mødet

    Slutter med en rapport der opsumere fundende problemer

- **Management reviews**

    Review af projekt relateret planer

    Review af produkt relateret planer

Review of rapporter

Om man er godt med i processen eller man skal ændre på scope, “Are we on schedule?”

- **Audit**

  Udført af udefrakommende

  De skal have special certification

- **Static analysis**

  Automatisk analyse af kode

  Kan ikke bruges på andet end kode

  Er med til at forhindre fejl

  Fjerner “technical debt” og “code smell”

- **Linters**

  Linters er betegnelsen på statiske analyse værktøjer

  Fanger dårlige vaner

  Fanger stavefejl

1.4 Explain test activities, and how they are related to each other. Then explain the test activities you carried out in your project.

- Unit testing

  Unit testing handler om at teste den mindste del som det giver mening at teste dette er ofte en methode.

- Integration testing

  Integration testing handler om at man gruppere flere moduler sammen som giver mening at gruppere og man så tester dem som en helhed.

Der findes flere måder at integrations teste på nogle af de mest populære er

- *Sandwich*
- *Top-down*
- *Buttom op*
- *Big bang*

- Refactoring

  Refactoring handler om at man ændre den eksisterende kode dette kan være med flere formål fx når man køre tdd skal man refactorere når man har fået sin test til at pase, refactorering kan også foregå for at give gøre sin kode mere læsbar eller nemmere at bygge vidre på.

- Maintenance

- Continuous Integration

  Handler om at man ofte pusher til sit versions styrings system typisk har man også ofte lavet tests her og så bliver det automatisk kørt når der bliver pushet

- Code reviews

1.5 Testing is related to ensuring higher code quality. Elaborate on what characterizes high code quality, and what makes code testable.

When we test our code, it makes it so if a bug or error should occur, we are made aware of it immediately when running the test, instead of at run-time in a production environment. It also makes it easier to make estimations on tasks. If we don't make tests, and something breaks down the line, we have unforeseen time and effort that has to be expended on this problem. If we test always and on everything, then we catch the bugs or errors straightaway, and our estimations remain more accurate.

- Testable code

  To ensure that a class is testable, the dependencies of the class should be interfaces instead of concrete implementations to make it not dependant on that certain implementation.

- Names of tests

  When we write the test method name, we should either add “test” as a prefix or suffix. We should also have descriptive names since each test should only do exactly one thing. Ex: shouldAddNewUserToDatabaseWithCorrectSignupFormTest.

- “sufficient” tests of a method or class

  A test is only as sufficient as a developer can make it. There might be oversights, but we can always ascertain the valid (and invalid) boundary values, as well as equivalence partitions, how often a method is called, with what parameters it was called, etc. It should also not be necessary to test any of the built-in Java libraries, as we can assume these work as intended.

- Assertions, defensive programming

  When writing test code, we use *Assertions* to verify whether two values are equal, less than, larger than, *not* equal, and so on. We can also assert whether errors are thrown.

  Defensive programming means checking your invariants, preconditions and postconditions.

- Dependency injection

  A form of Inversion of Control. Instead of a certain class instantiating its own implementations, it should take an interface in the constructor, so we’re not bound to a certain implementation. This makes it so we can mock the interface when we test our class.

1.6 Explain the concept of maintainable code, and how it's related to testing. Explain how to find out if a code base is maintainable.

- Maintainability
  - *Lav kobling, lav kompleksitet, dækket af tests, god navngivning af tests*
  - *Testbar kode: Let at reparere en bug, hvis man blot skal lokalisere den, ændre i koden og køre nogle tests. Ingen gætteleg om andre funktioner er beskadiget af det*
  - *Hvis man er nervøs for at røre ved koden, er der rød alarm på teknisk gæld*
  - *Lettere tidsestimere, når kode er dækket af test*
  - *Kan blive meget dyrt at ændre noget i ikke-testbar kode*
- Product quality
  - *UI test, har vi forstået kundens behov, har vi testet kravspecifikationen? (statisk test)*
  - *Produktets kvalitet bliver afspejlet i kodens kvalitet*
  - *Kodekompleksitet -*
  - *Når man tester bliver man tvunget til at skrive noget kode på en god struktureret måde. Og kode der gør det samme hver gang.*
  - *Testbar kode = lavere kobling, kodeuden temporal kobling*
  - *Testbar kode = ingen temporal kobling*
  - *Testbar kode = gør brug af Dependency Injection for at tilføre inversion of control*
- Temporal coupling
  - *Tidsmæssig kobling*
  - *afhængig af at ting skal kaldes i en særlig rækkefølge*
  - *Setter er et eksempel på temporal = man kan kun kalde den EFTER constructoren er kaldt*
  - *"Lav din kode på en måde, så programmøren kun kan gøre det rigtigt" (dvs ingen rækkefølger på metoder)*
  - *Testbar kode er oftest fri for temporal kobling*
  - *Eks på fail: Assignment3 - RoomServiceTest*

- Continuous Integration

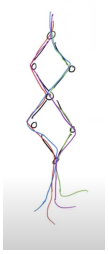
    - *ALLE tests køres ved hvert commit, hvilket bidrager til mere bug free kode, da det minimerer regressioner (fejl der kommer tilbage) -- ("det virker stadig, efter de nye ændringer/tilføjelser")*
    - *Udover at hver udvikler sidder og tester sin kode, så bliver tests'ne (sammen med resten af projektets tests) også kørt på CI-serveren hver eneste gang udvikleren committer sin kode.*

- Static Analysis
    - *Automatisk code review*

- Dependency injection, inversion of control
    - *Det er let at skrive og vedligeholde unittests med dependency injections, da dependencies er lette at mocke*
    - *Computerklasse med processorafhængighed-eks*

- Low coupling, high cohesion
    - ***LØS KOBLING:***
    - *Når ting er løst koblede sammen, er det også lettere at frakoble en komponent og putte en ny ind.*
    - *Hvis man fx vil udskifte den måde man gemmer data på, så er det nemt at tage fat i den komponent der gemmer data og skifte den ud med noget andet.*
    - *Når et system er opdelt i løst koblede komponenter, er de mere uafhængige af hinanden, hvorfor man mere sikkert kan ændre i de enkelte uden at det påvirker de andre.*
    - ***HIGH COHESION***
    - *Der skal ikke være kode indblandet, som ikke har noget med modulet at gøre: Et loggingmodul skal kun tage sig af logging, et storage modul skal kun tage sig af storage. => har med lav kobling at gøre*

- Cyclomatic code complexity
    - *Lange metoder er svære at forstå og derfor svære at vedligeholde. De er også svære at teste ordentligt.*
    - *En slags målestok for hvor kompleks en metode er:*
    - *Antallet af uafhængige stier i en metode:*



<-- fire stier gennem to if-sætninger:

(to stier = code coverage | fire stier = path coverage)

- *Antallet af tests der skal til at teste en metode*
- *Antallet af "if" + 1*
- *(hører under statisk analyse)*

1.7 Explain unit testing, and what characterizes it in contrast to other types of test.

- **What and why**

  Et stykke kode.

  Skal teste en unit

  Unit skal være isoleret

  Skal være automatisk

  Skal vise/kommunikere de fejl der bliver fundet

  For at teste den mindst mulige unit

  Skal kunne køre samtidig med andre tests

  Hjælper med at find bugs tidligt

  Hjælper med at forstå code basen

  Kan bruges som dokumentation

  Hjælper med at genbruge kode

- **Unit Under Test _ System Under Test_**

  (Vis kode)

  Vis unit under test

  Vis system under test

- **Unit test lifecycle(BeforeAll, AfterAll _ SetUp, TearDown)_**

  (Vis booking, med opret & fjern customer før og efter test)

  BeforeAll - for den kører test

  AfterAll - efter den har kørt test

- **Test doubles (mock, fake, stub, spy)**

  Test double objekt der erstatter noget produktionscode

  Mock:

  Når man verifyer om noget er blevet gjort

  Fake:

  Teste funktionalitet (when/then)

  Stubs:

  En lille mock klasse

  Spy:

  Når man tjekker noget, f.eks. om man har sendt en sms X antal gange

- **Test Driven Development**

  Test først kode efter

Red-green-refactor

Rappoterede fordele:

- Højere produktivitet
- Mindre tid på at debug
- Mere loosely coupled code

Reppoterede ulemper:

- Stort antal unit tests kan give en falsk trykhed

- **Dependency Injection**

  Når du giver er interface med som en parameter i en constructor eller metode

  Decoupling er essential for Unit testing, hvilket Dependency Injection hjælper med.

- **Equivalence classes, boundary value analysis, equivalence partitions**

  Equivalence partitioning teknik til at opdele data ind i units.

  Kan test hver unit med én test og ikke al dataen

  Boundary value analysis test partition grænserne

1.8 Explain test driven development, and how it affects the development process and code quality.

- Red, Green, Refactor

  Test driven development fungerer på den måde at man tester før man koder altså selvom man er bevidst om at testen kommer til at fejl.

  Bagefter skriver man det mindst mulige kode der skal til for at testen består.

Og til sidst fakturere man sin kode så den passer med det den skal bruges til.

- Testable code

  Når man gerne vil lave tests og ikke har lavet lavet tdd og måske heller ikke haft test i tankerne da man udviklede koden kan man stå i en situration hvor at det ikke er muligt at teste koden på en ordenlig måde dette problem kommer man uden om ved at have testet koden fra starten og derved have testable kode

- Maintainable code

  Når man har en høj code coverage som man har når man bruger tdd så vil det i fremtiden være lettere at veligeholde og vidre udvikle systemet.

- Equivalence partitions
- Positive, negative tests

  Når man tester er det en fordel måde at teste at noget virker når det bruges rigtigt og at noget fejler når det bruges forkert

  Har man fx en function der tester om et tal er lige vil man både teste med et lige og et ulige tal for at være sikker på at det virker.

1.9 Explain about test doubles. Explain how and why mocking is useful, and in what test areas.

Using mocks is useful, because it allows us to “mock away” the actual functionality, so we do not use the actual implementation, and can still feel safe that we’ve tested a “unit under test”, without having affected some other parts of our application by using concrete implementations.

- Mockito, mocks, spies, stubs, fakes, dummies

  Mockito is the framework we use to make mocks. On these mocks we can use methods like *verify(), when(), thenReturn()*, to make our test-flow behave the way it should, without using the methods on the concrete class, and potentially mess something up. This way we also ensure that we’re only testing *one* unit, as we’re just mocking an interface.

- Dependency injection

  When we supply an Interface into a method's parameters, or a class's constructor.

- Interfaces, contracts

  Interfaces help us abstract implementations away into abstractions, so we're not bound to a specific implementation, but can instead supply some interface in an argument or constructor, instead of having the class instantiate its own dependency (dependency injection).

- Black-box vs white-box

  With black-box testing, it means we can't look inside the box. That means a black-box test only looks at what goes in and what comes out. We're interested in our inputs, and we expect certain outputs.

  With white-box testing, we can suddenly see what's inside the box. That means we're interested in how the method or function is being carried out. We're interested in the flow of a certain method or component, rather than what is returned or changed.

1.10 Characterize high quality software. Explain how writing tests can increase code quality.

Test øger kvaliteten, da de fordrer pure funktioner, lav kobling

- Defensive programming
  - **check your invariants, preconditions and postconditions**
    - Test i runtime koden (tjek input)
  - Test at alle regler i kontrakt overholdes - tjek også at det fejler
  - *Trust nobody, Doubt yourself! -- vid at du er fejlbarlig*
- Black-box development
  - *Outsourcing*
  - *Fx arbejde med et interface, uvidende om implementationen (som måske er outsourced til Indien)*
- Interfaces, contracts

    - *Abstrakterer implementation, så vi kan give interfacet som argument - i stedet for at klassen selv skal instantiere sin egen dependency*
    - *Med kontrakten i hånden kan man begynde at lave sine tests, selvom systemet ikke er færdigimplementeret. For vha interfaces kan man mocke det der ikke er implementeret. (fx sms-service)*
    - *Interface er også en kontrakt (interface = hvad skal den kunne)*
- Inversion of control
    - *Brug interfaces, som kan mock'es. Derved kan vi være lykkeligt uvidende om den faktiske implementation.*
    - *Lav kobling*

- dependency injection
    - *Gøre metode testbar ved at trække dependencies ud. Opretter metoden noget selv, så ryk det ud i dens argumentet (eller klassens constructor)*
    - *Giv interface med som parameter*

- Components
    - *En komponent er en del af noget andet.*
    - *A reusable piece of software*
    - *gennemtestede komponenter => man skal ikke teste dem igen, når man bruger dem i et system*

1.11 Elaborate on dependencies in software, and how it's related to the subject of tests.

- **Dependencies between layers**

  Model lag, service lag, data lag

  Model lag afhængigt af service lag, service afhængigt at data.

  Lag altid afhængigt af lag nedenunder

  Skal bruge mocks/stubs for at teste mellem lag

- **System resources**

  Systemet kan have system afhængigheder, som System.nanotime

  Skal wrappes i et interface for at test

  Skal injectes ind i metoden/klassen

- **Relations between objects**

  Er meget fundamentalt i objekt-orienteret programmering

  Kan test det ved hjælp af dependency injection

  Skal mockes så vi kun test den en unit

- **Dependency inversion, Inversion of Control, Dependency Injection**

  ### Dependency inversion:

  Når man wrapper et “højere lag” i et interface og laget under kaldet interfacet.

  ### Inversion of Control:

  Er med til at skabe et loosely coupled program

  Opnås ved at wrapper klasser i interfaces

  ### Dependency Injection:

Når du giver er interface med som en parameter i en constructor eller metode

Decoupling er essential for Unit testing, hvilket Dependency Injection hjælper med.

- **Mocks**

  Mocke forbindelser mellem units, så vi kun tester én unit.

  Mockito er et godt framework

1.12 Explain problems in test automation, and how a continuous integration tool can help.

- What is Continuous Integration?

  I bund og grund betyder Continuous Integration at man ofte skal til pushe sin kode til det samlede kodebase på sit versionsstyringssystem.

  Når man arbejder flere sammen på det samme projekt er versionsstyring et meget vigtigt værktøj men hvis der ikke pushes særlig ofte kan der let opstå merge konflikter fordi 2 udviklere har arbejdet på det samme kode og derfor er de 2 ikke længere ens.

  Merge konflikter kan være svære at håndtere især hvis det er længe siden der er blevet arbejdet på det som er blevet pushet.

  Dette kan man løse ved følge ci princippet om at der ofte skal pushes der er ikke nogen fast tidsgrænse på hvor ofte dette skal gøres men hvis man gør det efter hver gang man har får lavet noget funktionelt som det giver mening at pushe det behøver ikke være færdigt.

  Vil man ikke kunne undgå merge konflikter men de ville være sjælnere og lettere at løse.

- How can a CI help regarding tests?

  Når der så ofte bliver pullet og pushet til ens vcs kan det også være svært at havde det fulde overblik over koden så måske man kommer til at ændre noget som man ikke var bevidst om var nødvendigt for at feature som en anden udvikler havde lavet i mellemtiden men hvis man har lavet tests af sit program ville man kunne få ens vcs til at builde og køre disse test og ud fra resultatet bestået / ikke bestået kunne sende fx en besked om at der er en fejl i programmet dette vil betyde at der er støre chance for at finde bugs tideligt.

- What is a regression?

  An error that comes back after having been fixed once.

- What test levels can be covered by a CI system?

  Unit, integration, system.

1.13 Explain specification-based testing, and how you can be more confident that you have written a sufficient amount of tests.

- Equivalence partitioning
  - De input som forårsager samme output, de tilhører een ækvivalenspartition. Når man har testet med een værdi fra den partition, så er hele ækvivalensklassen testet.
  - Ex: forsikring -> Hvis alle med alder ml 15-50 skal have den samme præmie udbetalt, så er det nok at teste med ét tal ml 15-50
- Boundary value analysis
  - Tester grænseværdier
  - Man erkender at rigtig mange fejl sker ude i kanterne af de her equivalence partitionering.
  - Man tjekker på hver side af boundary:
    - Interval: 50-60 -> tjekker 49, 50, 60, 61
- Edge cases
  - En ekstrem min- og maks-værdi, for at sikre ydre boundaries
- Decision tables
  - Hvis man har mange/indviklede conditions, så kan man gennemgå alle muligheder systematisk
  - Tabeller hvor man skriver alle kombinationer af input, så man kan få overblik og teste alle kombinationer
- Code coverage
  - Er alle kodelinjer dækket af tests?
    - Jacoco

  Hvad med path coverage?
- Mutation testing
  - Tester kvaliteten
  - Forsøg at få tests til at fejle:
    - Ændrer noget i koden (lav en mutation)

        - kører unit test
        - hvis unit test **ikke** fejler, så er den del af koden tydeligvist ikke dækket af test

- Finder ud af om tests er dækkende
- PITest