## Testning og debugging: (Software udvikler)

Hej alle sammen i denne her uge skal du beskæftige dig med testning og debugging af kodebaser. Fordi IT-teams er sammensat af mange folk med forskellig baggrund og kompetencer på tværs af flere kodesprog og teknologier er kvalitetssikring mere vigtigt end det nogensinde har været. For software at fungerer korrekt, kræver det systematiske og strukturerede metoder til både fejlfinding og testning. Dette inkluderer alt fra at finde fejl i specifikke funktioner til at sikre, at hele systemet fungerer som forventet, selv efter ændringer. Her er en introduktion til nogle af de vigtigste koncepter og teknikker, som professionelle softwareudviklere bruger til at teste og debugge kode.

Til denne uges opgave skal i få bytte kodebase med en af jeres medkursister og få udleveret PDF-downloaderen opgaven som i lavede i uge 5. Herefter har i til opgave:

1. Fork deres github repo, derefter clone det til ned sin egen computer
2. Når I har fået åbnet projektet i jeres egen IDE, så skal i for programmet til at køre på jeres computer og se om det virker som det var beskrevet i PDF-downloader opgaven.
   * Det her step kræver at man Manuel debugger
3. I skal noter de rettelser i laver koden og hvad der vil nødvendig at have med i readme-filen for at kunne køre programmet uden at nogen dyb forståelse for programmet. Det vil være blandt andet de her rettelser i skal have med i jeres præsentation.
4. Efter I have noteret de rettelserne, så skal i til at extende projektet med UNIT tests.
5. Efter I have extendet projektet med UNIT tests, så skal i extende projektet med INTERGRATION tests.
6. Efter I have lavet UNIT og INTERGRATION tests, så skal I lave en CI/CD pipline til projektet, det kunne fx være github actions eller i Jenkins

Jeres fund og tiltag skal i slutningen af ugen præsenteres for resten af holdet, her har i 10 minutter til at præsentere og 5 til at svare på spørgsmål og få feedback. Det er som udgangspunkt valgfrit hvilken form for testning i vælger at systematisere for at give jer et kort overblik bliver i her præsenteret for nogle aktuelle måder at teste på i kan inkludere i jeres løsning.

Husk at feedbacken og præsentationen af testning af kodebasen skal være konstruktiv og rettet for at give produktet en højere kvalitetssikring. I er velkomne til at tilføje direkte til kodebasen hvis i eksempelvis ser nogle potentielle udvidelser der kunne kræve yderligere testning eller kvalitetssikring i fremtiden

**Automatiserede tests**

Automatiserede tests gør det muligt at validere, at kode fungerer korrekt, uden at det kræver manuel gennemgang. Disse tests kan udføres på forskellige niveauer for at sikre funktionalitet, integration mellem moduler og hele brugeroplevelsen.

**1. Unit tests:**  
Unit tests fokuserer på individuelle funktioner eller metoder og tester, om de giver de forventede resultater for forskellige input.

* **Formål:** Sikre, at hver komponent fungerer korrekt i isolation.
* **Eksempel:** Test, at en funktion, der beregner momsen på en vare, returnerer de korrekte værdier for forskellige priser og skattesatser.

**2. Integrationstests:**  
Integrationstests kontrollerer, hvordan forskellige moduler i applikationen fungerer sammen. Fokus er på at opdage fejl i grænsefladerne mellem komponenterne.

* **Formål:** Sikre, at moduler kan kommunikere korrekt og udveksle data, uden at der opstår fejl.
* **Eksempel:** Test, at en databaseopdatering korrekt reflekteres i brugergrænsefladen.

**3. End-to-end tests:**  
End-to-end tests simulerer brugerinteraktioner med applikationen for at teste hele arbejdsgangen fra start til slut.

* **Formål:** Sikre, at applikationen fungerer som forventet for brugeren, inklusiv eventuelle bagvedliggende systemer.
* **Eksempel:** Test, at en bruger kan logge ind, foretage et køb og modtage en kvittering.

**4. Test frameworks:**  
Automatiserede tests oprettes og køres med test frameworks, der tilbyder strukturerede metoder til at skrive og eksekvere tests.

* + JUnit for Java-projekter.
  + NUnit til C# i Visual Studio.
  + Pytest for Python.
  + Jest for JavaScript.

**5. Continuous Integration (CI):**  
Automatiserede tests bliver ofte integreret i en Continuous Integration (CI)-pipeline, hvor kode automatisk testes ved hver ændring, inden den sendes videre i udviklingsprocessen.

* **Formål:** Sikre, at ny kode ikke introducerer fejl, og at eksisterende funktionalitet stadig fungerer.
* **Eksempel:** Opsætning af en GitHub Actions-pipeline til at køre tests, når der laves en pull request.

**Manuel debugging**

Manuel debugging bruges, når der opstår fejl i koden, som kræver en mere detaljeret undersøgelse. Det er en praktisk proces, der ofte udføres sammen med automatiserede tests for at finde og rette fejl.

**1. Fejlfinding med breakpoints:**  
Breakpoint-debugging giver udviklere mulighed for at stoppe kodeeksekveringen ved bestemte punkter for at inspicere værdier og finde fejl.

* **Værktøjer:**
  + IntelliJ til Java.
  + Visual Studio til C#.
  + VS Code til Python.

**2. Fejlsøgning i logfiler:**  
Logfiler kan give indsigt i, hvad der sker i applikationen, og hvor der opstår problemer.

* **Eksempel:** Brug af logniveauer (fx INFO, DEBUG, ERROR) til at spore, hvor en applikation fejler.

**3. Brugerinput-testing:**  
Test af edge cases (grænsetilfælde) og corner cases (kombinationer af uventede input) afslører ofte fejl, der ikke er opdaget i udviklingsprocessen.

* **Eksempel:** Test, hvordan en funktion håndterer null-værdier eller meget store tal.

**4. Regression testing:**  
Når ny funktionalitet tilføjes, er det vigtigt at sikre, at tidligere funktionalitet stadig virker. Dette kaldes regression testing og kan udføres manuelt eller automatiseret.

**Teststrategier og -principper**

**1. Testdækning:**  
Det er vigtigt at prioritere højrisikokode og sikre, at alle kritiske dele af applikationen er dækket af tests.

**2. TDD og BDD:**

* **TDD (Test-Driven Development):** Udvikleren skriver tests først og derefter koden, der opfylder testene.
* **BDD (Behavior-Driven Development):** Fokus på at skrive tests baseret på brugerhistorier og forretningskrav.

**3. Black box vs. white box testing:**

* **Black box testing:** Testeren kender ikke til den interne implementering og tester kun funktionaliteten.
* **White box testing:** Testeren har kendskab til den interne struktur og kan teste specifikke kodeveje.

**Ressourcer:**

<https://www.leapwork.com/blog/manual-vs-automated-testing>

<https://www.geeksforgeeks.org/software-engineering-debugging/>