Note til 62532 Versionsstyring og testmetoder

Indhold

Git og Github

Git:

Git er et distribueret versionsstyringssystem, der bruges til at spore ændringer i kildekoden under softwareudvikling. Det giver udviklere mulighed for at arbejde sammen, bevare en historik over ændringer og håndtere parallelle udviklingsgrene.

GitHub:

GitHub er en webbaseret platform, der bygger på Git og tilbyder hosting af Git-lager samt værktøjer til samarbejde. Det fungerer som en centraliseret platform, hvor udviklere kan gemme, dele og bidrage til projekter. GitHub indeholder yderligere funktioner som pull requests, issues og projektstyring for at forbedre samarbejdet i udviklingsteams.

Git Kommandoer:

- Initialisering af et nyt Git Repository:
  - ***git init***
- Kloning af et Eksisterende Repository:
  - ***git clone <repository_url>***
- Arbejde med Ændringer:
  - Tilføj ændringer til staging area dette tracker også en fil:
    - ***git add <filnavn>***
  - Gem ændringer med en besked:
    - ***git commit -m "Beskrivelse af ændringer"***
- Opdatering og Synkronisering:
  - Hent seneste ændringer fra fjernrepository:
    - ***git pull***
  - Send lokale ændringer til fjernrepository:
    - ***git push***
- Branch Management:
  - Opret en ny gren:
    - ***git branch <gren_navn>***
  - Skift til en anden gren:
    - ***git checkout <gren_navn>***
  - Flet ændringer fra en gren til en anden:
    - ***git merge <kildegren>***
  - Opret en ny gren og skift til den samtidig:
    - ***git checkout -b <ny_gren_navn>***
  - Slet en gren (lokalt):
    - ***git branch -d <gren_navn>***
  - Slet en gren (fjernrepository):
    - ***git push origin --delete <gren_navn>***
- Status og Log:
  - Vis status for lokale ændringer:
    - ***git status***
  - Vis commit-log:
    - ***git log***

- Arbejde med fjernrepository:
    - Fjernrepository information:
        - ***git remote -v***
    - Tilføj en fjernrepository:
        - ***git remote add <navn> <repository_url>***
- Fletning af grene:
    - Flet ændringer fra en gren til den aktuelle gren:
        - ***git merge <kilde_gren>***
- Håndtering af ændringer:
    - Fortryd seneste commit (bevar ændringer i arbejdsområdet):
        - ***git reset HEAD~1***
    - Fortryd seneste commit (slet ændringer i arbejdsområdet):
        - ***git reset --hard HEAD~1***
- Visning af historik og forskelle:
    - Vis ændringer mellem to commits:
        - ***git diff <commit_sha1>..<commit_sha2>***
    - Vis forskellen mellem arbejdsområdet og seneste commit:
        - ***git diff HEAD***
- Tilbagekaldelse af ændringer:
    - Tilbagekald en fil til seneste commit:
        - ***git checkout -- <filnavn>***
    - Fortryd ændringer før commit:
        - ***git restore --source=HEAD --staged --worktree <filnavn>***
- Opdatering af fjernrepository:
    - Opdater fjernrepository-reference uden at downloade ændringerne:
        - ***git fetch***
- GitHub-**samarbejdskommandoer:**
    - Lav en kopi af et repository under dit GitHub-konto.
        - ***git fork***
    - Opret en pull request for at foreslå ændringer til det originale repository.
        - ***git pull-request***
- Ignorer
    - ***.gitignore***
- Hvis du vil oprette en ny fil og så tilføje den til et remote repo, så er den korrekte rækkefølge:
    - opret fil --> git add --> git commit --> git push
- Git Tag:
    - Opret en tag på det nuværende commit:
        - ***git tag <tag_navn>***
- **Git Stash:**
    - Gem midlertidige ændringer uden at commite dem:
        - ***git stash***
    - Anvend de gemte ændringer:
        - ***git stash apply***
- Git Cherry-pick:
    - Anvend en enkelt commit fra en anden gren:

        - ***git cherry-pick <commit_sha>***
- **Git Rebase:**
    - Omarranger commits for at forbedre historik:
        - ***git rebase <base_gren>***
- **Git Bisect:**
    - Brug binærsøgning til at finde en specifik fejlforårsagende commit:
        - ***git bisect start***
        - ***git bisect good <commit>***
        - ***git bisect bad <commit>***
- Git Submodule:
    - Arbejd med Git-undermoduler indlejret i dit projekt:
        - ***git submodule add <repository_url>***
        - ***git submodule update --init --recursive***
- **Git Log (med grafer):**
    - Se commit-log med grafiske repræsentationer:
        - ***git log --graph --oneline –all***
- Git Amend:
    - Tilføj ændringer til den seneste commit:
        - ***git commit –amend***
- Git Reflog:
    - Se en historik over git-handlinger, herunder branchskift og rebasing:
        - ***git reflog***
- Git Blame:
    - Find ud af, hvem der ændrede en bestemt linje i en fil:
        - ***git blame <filnavn>***
- **Git Archive:**
    - Opret en tar- eller zip-fil fra et bestemt commit:
        - ***git archive --format=zip --output=udgangsfil.zip <commit_sha>***
- Git Config:
    - Konfigurer Git-indstillinger, f.eks. brugeroplysninger:
        - ***git config --global user.name "Dit Navn"***
        - ***git config --global user.email "din@email.com"***

Test typer:

- Unittests:
    - **Definition:** En unittest er en type test, der fokuserer på at validere, om individuelle komponenter eller moduler af softwaren fungerer korrekt. Det er ofte udført af udviklere og er designet til at identificere fejl på det laveste niveau af softwaren, såsom funktioner eller metoder.
    - **Unittests består af.**
        - Setup-delen:
            - Dette er det første trin i testen, hvor du forbereder det nødvendige miljø for testen. Dette kan omfatte oprettelse af objekter, initialisering af variable

eller andre forberedende handlinger. Målet er at skabe det nødvendige grundlag for at udføre selve testen.
        - Udførelsesdelen:
            - Dette trin indebærer udførelse af den specifikke funktionalitet eller metode, der skal testes. Dette kan omfatte kald til en metode, udførelse af en algoritme eller manipulation af objekter. Formålet er at udføre den specifikke del af koden, der skal testes.
        - Kontrol- eller Valideringsdelen:
            - Efter udførelsen af den pågældende funktionalitet vurderes resultatet. Dette indebærer ofte sammenligning af det faktiske resultat med det forventede resultat. Hvis resultatet opfylder de forventede kriterier, anses testen for at være vellykket. Hvis ikke, indikerer det, at der kan være en fejl i koden, og det hjælper udvikleren med at identificere og rette fejlen.
    - Hvornår: Under udviklingsfasen.
    - Hvem: Udvikleren.

- Integrationstest:
    - **Definition:** Integrationstesten evaluerer samspillet mellem forskellige komponenter eller moduler i softwaren. Formålet er at afgøre, om de integrerede dele fungerer korrekt sammen. Integrationstests kan afsløre problemer som kommunikationsfejl, datatab eller inkompatibiliteter mellem komponenter.
    - Hvornår: Efter unittests og før systemtesten under udviklingsfasen.
    - Hvem: Testere og udviklere.
- Systemtest:
    - **Definition:** Systemtesten tester den fulde applikation som en samlet enhed for at sikre, at den opfylder de specificerede krav. Den omfatter test af alle systemets funktioner og kan omfatte forskellige testtyper som ydeevne, sikkerhed og brugervenlighed.
    - Hvornår: Efter integrationstesten.
    - Hvem: Dedicerede testteam.
- Alfatest:
    - **Definition:** Alfatesten er den første fase af ekstern test, hvor softwaren testes i et kontrolmiljø af interne brugere inden frigivelsen til en bredere brugergruppe. Formålet er at identificere tidlige problemer og fejl, før softwaren når ud til en større brugerbase.
    - Hvornår: Efter systemtesten og før beta-testen.
    - Hvem: Udviklere, testere og interne interessenter.
- Beta-test:
    - **Definition:** Betatesten er en ekstern test, hvor softwaren frigives til en begrænset gruppe af eksterne brugere. Disse brugere evaluerer softwaren i deres egne miljøer og giver feedback til udviklerne. Formålet er at identificere eventuelle problemer eller mangler, før softwaren lanceres officielt.
    - Hvornår: Efter accepttesten.
    - Hvem: Slutbrugere udenfor udviklingsteamet.
- Accepttest:
    - **Definition:** Accepttesten er en type test, der udføres for at verificere, om softwaren opfylder de specificerede krav og kriterier, som kunden eller brugeren har fastlagt. Det er den sidste fase af testprocessen, inden softwaren godkendes til implementering.

    - Hvornår: Efter systemtesten.
    - Hvem: Kunden eller kunderepræsentant.
- Brugertest:
    - **Definition:** Brugertesten fokuserer på at evaluere softwaren ud fra brugernes perspektiv. Det indebærer ofte at lade faktiske brugere udføre opgaver eller scenarier for at identificere eventuelle brugervenlighedsproblemer eller forbedringsområder i softwaren.
    - Hvornår: Efter implementeringen.
    - Hvem: Slutbrugere.

Tidslinje over softwareudvikling

- Kravspecifikation (Requirements Analysis):
    - Tidspunkt: Før udviklingsfasen.
    - Aktiviteter:
        - Indsamle og forstå kundens krav og forventninger.
        - Specificer kravene klart og præcist.
        - Opret en kravspecifikation.
- Design:
    - Tidspunkt: Følger kravspecifikationen.
    - Aktiviteter:
        - Udvikle systemarkitektur og design.
        - Identificere komponenter og deres relationer.
        - Opret en detaljeret designspecifikation.
- Implementering (Coding):
    - Tidspunkt: Efter designfasen.
    - Aktiviteter:
        - Skriv kode baseret på designspecifikationen.
        - Udfør enhedstests på individuelle komponenter.
        - Opret en fungerende version af softwaren.
- Unittests:
    - Tidspunkt: Under implementeringen.
    - Aktiviteter:
        - Opret og udfør tests for at validere korrektheden af individuelle komponenter.
        - Identificer og ret fejl i koden.
- Integrationstests:
    - Tidspunkt: Efter implementering og unittests.
    - Aktiviteter:
        - Test samspillet mellem komponenter og systemet som helhed.
        - Identificer og ret eventuelle konflikter eller inkompatibiliteter.
- Systemtests:
    - Tidspunkt: Efter integrationstests.
    - Aktiviteter:
        - Test hele systemets funktionalitet.
        - Sikre, at det opfylder kravspecifikationen.
        - Identificer og ret eventuelle mangler eller fejl.
- Accepttests:

    - Tidspunkt: Efter systemtests.
    - Aktiviteter:
        - Test for at sikre, at softwaren opfylder kundens krav.
        - Godkendelse af softwaren til implementering.
- Implementering (Deployment):
    - Tidspunkt: Efter accepttests.
    - Aktiviteter:
        - Implementer softwaren i produktionsmiljøet.
        - Overfør data og konfigurationer fra testmiljøet.
- Vedligeholdelse:
    - Tidspunkt: Efter implementering og driftsættelse.
    - Aktiviteter:
        - Overvåg og løs problemer i produktionsmiljøet.
        - Udfør opdateringer og rettelser efter behov.
        - Evaluer ydeevne og planlæg eventuelle forbedringer.

Udviklingsmetode

Vandfaldsmodel:

- Beskrivelse:
    - Sekventiel og lineær udviklingsproces, hvor hver fase skal afsluttes, før den næste starter.
- Fordele:
    - Klare og veldefinerede faser og leverancer.
    - Passer godt til små projekter med klare krav.
- Ulemper:
    - Rigid struktur, svært at tilpasse sig ændringer.
    - Risiko for, at kunden først ser produktet ved projektets afslutning.

Prototyping:

- Beskrivelse:
    - Skaber en foreløbig version af systemet til brugerfeedback og evaluering.
- Fordele:
    - Hurtig feedback og forbedringer.
    - Giver brugerne mulighed for at forstå systemet tidligt.
- Ulemper:
    - Kræver klare krav og design før prototyping.
    - Kan føre til manglende fokus på grundlæggende struktur.

Iterativ og inkrementel udvikling:

- Beskrivelse:
    - Bygger systemet gradvist i små dele og forbedrer det gennem iterationer.
- Fordele:
    - Hurtige levereable og hyppig feedback.
    - Fleksibilitet til at håndtere ændringer.
- Ulemper:

    - Kan kræve mere opmærksomhed på planlægning.
    - Risiko for kompleksitet ved gentagen integration.

Spiralmodel:

- Beskrivelse:
    - Kombinerer elementer fra prototyping og vandfaldsmodellen med en betydelig vægt på risikostyring.
- Fordele:
    - Effektiv risikostyring.
    - Fleksibel og kan tilpasses ændringer.
- Ulemper:
    - Kompleks og kræver ekstra ressourcer.
    - Kan blive dyrt og tidskrævende.

RAD (Rapid Application Development):

- Beskrivelse:
    - Fokuserer på hurtig udvikling og iterationer med minimal planlægning.
- Fordele:
    - Hurtig udvikling og tidlig levering.
    - Stærkt brugerengagement.
- Ulemper:
    - Kræver veldefinerede krav.
    - Kan føre til mindre stabil software.

V-Model:

- Beskrivelse:
    - En udvidelse af vandfaldsmodellen, hvor hver udviklingsfase har en tilsvarende testfase.
- Fordele:
    - Klare sammenhænge mellem udvikling og test.
    - Struktureret og let at forstå.
- Ulemper:
    - Stiv struktur, svært at tilpasse sig ændringer.
    - Risiko for at miste fleksibilitet.

Big Bang Model:

- Beskrivelse:
    - Uformel og enkel model uden specifikke processer.
- Fordele:
    - Fleksibel og tilpasser sig ændringer.
    - Passer til små projekter med uklare krav.
- Ulemper:
    - Risiko for kaos og dårlig styring.
    - Manglende struktur og planlægning.

DevOps:

- Beskrivelse:
    - Kombinerer udvikling og drift for at forbedre samarbejdet og levere software hurtigere og mere pålideligt.
- Fordele:
    - Hurtigere levering af software.
    - Automatisering reducerer fejl og øger effektiviteten.
- Ulemper:
    - Kræver organisatoriske ændringer.
    - Kompleks implementering.

Kanban:

- Beskrivelse:
    - Fokuserer på visuel styring af arbejdsprocessen, hvor opgaver repræsenteres som kort på et Kanban-board.
- Fordele:
    - Fleksibel og let at implementere.
    - Optimerer arbejdsstrømmen og reducerer spild.
- Ulemper:
    - Mindre struktureret end andre metoder.
    - Kræver klare processer.

Lean Development:

- Beskrivelse:
    - Inspireret af lean produktion og fokuserer på eliminering af spild og kontinuerlig forbedring af processer.
- Fordele:
    - Effektiv ressourceudnyttelse.
    - Fokus på kvalitet og kontinuerlig forbedring.
- Ulemper:
    - Kan kræve kulturelle ændringer.
    - Mere kompleks implementering.

Agile Metode:

- **Beskrivelse:**
    - Agile er en iterativ og inkrementel udviklingsmetode, der fremhæver fleksibilitet og samarbejde. Det opdeler projektet i små, inkrementelle leverancer kaldet "sprints" og fremmer hyppig tilpasning til ændringer i krav.
- Fordele:
    - *Fleksibilitet:* Evne til at håndtere ændringer og tilpasse sig dynamiske krav.
    - *Kundetilfredshed:* Tæt samarbejde med kunden sikrer, at produktet opfylder deres behov.
    - *Tidlig levering:* Hyppige leverancer sikrer tidlig værdi og feedback.
    - *Kontinuerlig forbedring:* Iterative cyklusser giver mulighed for konstant forbedring.

    - *Højere produktkvalitet:* Hyppige tests og rettelser forbedrer produktets kvalitet.
- **Ulemper:**
    - *Manglende klare krav:* Agile kræver ofte, at kravene udvikles løbende, hvilket kan være udfordrende.
    - *Usikkerhed:* Nogle projekter kan have usikkerhed, især i forbindelse med planlægning og estimering.
    - *Kræver erfarent team:* Succesfuld implementering kræver ofte et erfarent og velkoordineret team.
    - *Dokumentation:* Agile kan blive anklaget for at have mindre omfattende dokumentation sammenlignet med mere traditionelle metoder.

Semantisk Versionering og Versionsnummer Skift (Eksempel: 5.3.1 til 5.4.0)

Når et bibliotek anvender semantisk versionering, angiver versionsnumrene normalt MAJOR.MINOR.PATCH-formatet. Lad os se på betydningen af hvert nummer og hvordan det relaterer til skiftet fra version 5.3.1 til 5.4.0:

- MAJOR versionnummer (5.x.x):** Øges, når der foretages inkompatible API-ændringer. Dette betyder, at hvis MAJOR-nummeret stiger, kan der være brudte funktioner, og den nye version kan kræve ændringer i din kode for at forblive kompatibel.
- MINOR versionnummer (x.4.x):** Øges ved tilføjelse af funktionalitet på en bagudkompatibel måde. Dette betyder, at mindre opdateringer og tilføjelser er blevet introduceret, men eksisterende funktionalitet forventes at forblive intakt.
- PATCH versionnummer (x.x.0):** Øges for bagudkompatible fejlrettelser. Dette betyder, at der er udført rettelser til fejl, men uden at tilføje nye funktioner eller ændre eksisterende funktionalitet.
- Med dette i tankerne, når versionsnummeret går fra 5.3.1 til 5.4.0, indikerer det normalt, at der er tilføjet nye funktioner på en bagudkompatibel måde. Så svaret ville være:

Det er en mindre tilføjelse eller ændring, som er bagudkompatibel med den tidligere version.

Dette betyder, at du kan opdatere til den nye version og forvente, at din eksisterende kode og funktionalitet forbliver kompatible, mens du drager fordel af de nye tilføjelser eller ændringer.

Code Coverage

Code coverage refererer til målingen af, hvor meget af koden i et softwareprojekt der bliver udført under automatiserede tests. Det måles i procent og giver en indikation af, hvor godt koden er blevet testet. Høj code coverage betyder ikke nødvendigvis, at koden er fejlfri, men det indikerer, at mange af dens stier er blevet udforsket under testprocessen.

Typer af Code Coverage:

- Linje-dækning (Line Coverage):
    - Måler, hvor mange kodelinjer der er blevet udført under test.
    - Hver linje markeres som enten udført eller ikke udført.

- Gren-dækning (Branch Coverage):
    - Evaluerer, om alle beslutningsgrene i koden er blevet testet.
    - Det sikrer, at både "sand" og "falsk" delen af if-sætninger er blevet udført.
- Sti-dækning (Path Coverage):
    - Fokuserer på at teste alle mulige stier gennem koden.
    - Identificerer kombinationer af betingelser, der fører til forskellige udførelsesveje.

Fordele ved Code Coverage:

- Identifikation af Manglende Testdækning:
    - Hjælper med at identificere områder af koden, der ikke er blevet testet.
- Kvalitetsindikator:
    - Fungerer som en indikator for testkvaliteten og giver tillid til koden.
- Fejlopfølgning:
    - Gør det lettere at identificere og rette fejl, da dårligt testede områder er mere tilbøjelige til at indeholde fejl.

Udfordringer ved Code Coverage:

- Fokus på Mængde frem for Kvalitet:
    - Høj code coverage betyder ikke nødvendigvis, at alle mulige scenarier er blevet testet grundigt.
- Ignorerer Eksekveringstid:
    - Måler ikke nødvendigvis, hvor lang tid koden tilbringer under test.
- Kan Skabe Falsk Tryghed:
    - Høj code coverage kan føre til en falsk følelse af, at koden er fejlfri.

Code covrage i procenter

Hvis vi har opnået en statement coverage på 80%, betyder det, at 80% af de udførbare kodelinjer i softwareprojektet er blevet udført under automatiserede tests. Lad os bryde det ned:

- Statement Coverage: Dette måler, hvor mange kodelinjer (statements) der er blevet udført i løbet af testprocessen.
- 80% Statement Coverage: Dette angiver, at 80 ud af hver 100 kodelinjer er blevet udført under testene.

Dette kan fortolkes som:

- 80% af kodelinjerne er blevet udført mindst én gang.
    - Dette betyder, at der er blevet udført mindst én test, der har berørt eller kørt hver af disse 80 kodelinjer.
- 20% af kodelinjerne er ikke blevet udført under testene.
    - Der er 20 kodelinjer, som ikke er blevet berørt af testene, og det er derfor uklart, hvordan de reagerer under forskellige scenarier.

Det er vigtigt at bemærke, at selvom 80% statement coverage er en indikator for testdækningen, garanterer det ikke nødvendigvis, at alle grene og stier i koden er blevet testet. Det er derfor en god praksis at

kombinere statement coverage med andre typer dækning som gren- og sti-dækning for at opnå en mere fuldstændig testdækning.

Formel Testcase:

- **Test Case ID:** Et unikt identifikationsnummer, der bruges til at organisere og referere til testcasen.
- **Præconditions (Preconditions):** De nødvendige betingelser, der skal være opfyldt, før testen udføres. Dette sikrer, at testmiljøet er korrekt opsat.
- **Postconditions:** De forventede betingelser, der skal være opfyldt, efter testen er udført. Dette beskriver, hvordan systemet eller miljøet forventes at ændre sig som følge af testen.
- **Test Procedure:** En detaljeret beskrivelse af trinene, der skal udføres under testen. Dette inkluderer de specifikke handlinger, testeren skal udføre, og de forventede resultater efter hver handling.
- **Forventet Resultat:** Den forventede udgang fra testen. Dette er, hvad der forventes at ske, hvis systemet eller funktionen fungerer korrekt. Det bruges til at vurdere, om testen har bestået eller fejlet.
- **Testkoden (Test Code):** Den kode eller det script, der bruges til at udføre testen. Dette kan omfatte inputdata, testprocedurer og forventede resultater.
- **Dato/Tidspunkt for Udførsel**: Tidspunktet og datoen for, hvornår testen blev udført. Dette giver sporbarhed og historisk reference.
- **Ækvivalensklasser:** Opdeling af inputdata i klasser, hvor hvert input i en klasse forventes at have den samme virkning på systemet. Dette kan hjælpe med at identificere repræsentative inputscenarier.
- **Testdata:** De specifikke data, der bruges under testen. Dette kan omfatte både gyldige og ugyldige input.
- **Fejlrapporthåndtering:** Instruktioner eller forventninger til, hvordan testeren skal håndtere fejl, der opstår under testen. Dette kan omfatte dokumentation og rapportering af fejl.

Test Driven Development (TDD):

- **Definition:**
  - TDD er en udviklingsmetode, hvor tests skrives, før selve implementeringen af koden.
- **TDD-cyklus:**
  - Red: Skriv en test, der fejler, for at illustrere en manglende funktion eller fejl i koden.
  - Green: Implementer den minimale kode, der er nødvendig for at få testen til at bestå.
  - Refactor: Forbedre koden uden at ændre dens adfærd for at gøre den mere effektiv, læsbar eller vedligeholdelsesvenlig.
- **Fordele ved TDD:**
  - Tidlig fejlidentifikation: Fejl opdages tidligt i udviklingsprocessen, hvilket reducerer omkostningerne ved fejlrettelse senere.
  - Kodeforståelse: Forbedrer kodens forståelse, da udviklere skriver tests for at specificere den ønskede funktionalitet.

    - Automatiserede tests: Skaber en omfattende testpakke, der kan køres automatisk for at sikre, at eksisterende funktionalitet ikke er brudt ved senere ændringer.
- **TDD-principper:**
    - "You aren't done until your code is clean": TDD opfordrer til regelmæssig refaktorering for at bevare og forbedre kodekvaliteten.
    - Skriv tests i små trin: Start med små og enkle tests for at opbygge gradvist mere kompleks funktionalitet.
- **Tre love i TDD:**
    - First Law (You may not write production code until you have written a failing unit test): Ingen implementeringskode skrives, medmindre der allerede er en fejlfri test.
    - Second Law (You may not write more of a unit test than is sufficient to fail): Skriv kun så meget testkode, der kræves for at få testen til at fejle.
    - Third Law (You may not write more production code than is sufficient to pass the currently failing test): Implementer kun nok produktionskode til at få testen til at bestå.
- **Udfordringer ved TDD:**
    - Kræver disciplin og praksis for at mestre.
    - Nogle udviklere kan finde det udfordrende at skrive tests først.

Hvornår man bruger TDD:

- **Ny funktion skal implementeres:**
    - Når der kræves ny funktionalitet, kan TDD bruges til at specificere og sikre, at koden opfylder de krævede specifikationer.
- **Fejlretning:**
    - Ved fejl i eksisterende kode kan TDD hjælpe med at isolere og rette fejlen ved at skrive tests, der demonstrerer problemet, og derefter implementere en løsning.
- **Refaktorering:**
    - Under refaktorering af eksisterende kode kan TDD hjælpe med at sikre, at adfærden forbliver intakt, mens koden forbedres.
- **Usikre krav eller design:**
    - Når kravene eller designet ikke er klart defineret, kan TDD hjælpe med at afklare og strukturere implementeringen trin for trin.
- **Samarbejde i teams:**
    - TDD kan være gavnligt i teams, da det skaber klare og testbare grænseflader og hjælper med at forstå, hvordan forskellige dele af systemet interagerer.
- **Integration af tredjepartsbiblioteker:**
    - Når du integrerer tredjepartsbiblioteker eller værktøjer, kan TDD hjælpe med at validere, at integrationen fungerer som forventet.
- **Sikre vedligeholdelse:**
    - Hvis du ønsker at opretholde og videreudvikle software over tid, hjælper TDD med at sikre, at ændringer ikke utilsigtet bryder eksisterende funktionalitet.
- **Agil udvikling:**
    - TDD er en integreret del af agile udviklingsmetoder, hvor hurtige iterationer og regelmæssige tests er afgørende.

A/B Tests:

- **Definition:**
  - En metode inden for softwareudvikling og markedsføring, hvor to versioner (A og B) af et produkt eller en funktion testes for at evaluere, hvilken der udfører bedre.
- **Anvendelse:**
  - Ofte brugt i webudvikling, markedsføring og brugergrænsefladeoptimering.
- **Udførelse:**
  - To grupper (A og B) vælges tilfældigt.
  - Version A og B implementeres, hvor kun én variabel adskiller dem (f.eks., farve, tekst, layout).
  - Resultater sammenlignes for at afgøre, hvilken version der opnår bedre mål (f.eks., konverteringsrate, brugerinteraktion).
- **Fordele:**
  - Objektiv måling af effektivitet.
  - Muliggør dataunderstøttet beslutningstagning.
  - Identifikation af optimale løsninger.
- **Ulemper:**
  - Kræver tilstrækkelig trafik eller data for at være statistisk signifikant.
  - Resultater kan påvirkes af eksterne faktorer.
- **Eksempel:**
  - A/B-tester to versioner af en købsknap for at se, hvilken der resulterer i flere køb.

Debuggerens Funktioner og Termer

- **Breakpoint:**
  - Definition: Et punkt i din kode, hvor udførelsen midlertidigt stoppes, så du kan inspicere variabler og undersøge programmets tilstand.
  - Anvendelse: Bruges til at identificere fejl, undersøge variable eller følge udførelsesstien.
- **Step In:**
  - Definition: Debuggeren går ind i en funktion eller metode, der bliver kaldt på det nuværende breakpoint.
  - Anvendelse: Nyttig, når du vil undersøge, hvad der sker inde i en specifik funktion.
- **Step Over:**
  - Definition: Debuggeren udfører det næste trin uden at gå ind i eventuelle funktioner eller metoder, der bliver kaldt på det nuværende breakpoint.
  - Anvendelse: Bruges, når du ikke ønsker at dykke ned i detaljerne af kaldte funktioner, men ønsker at fortsætte udførelsen trin for trin.
- **Step Out:**
  - Definition: Debuggeren udfører resten af den nuværende funktion eller metode og stopper ved det næste breakpoint udenfor den.
  - Anvendelse: Nyttig, når du er dykket ind i en funktion ved et tidligere breakpoint, og nu vil fuldføre udførelsen af den funktion.
- **Watchpoint:**

    - Definition: En type breakpoint, der stopper udførelsen, når en bestemt variabel eller hukommelsesadresse ændres.
    - Anvendelse: Bruges til at spore ændringer i specifikke variabler eller hukommelsesområder.
- **Condition:**
    - Definition: En betingelse, der kan tilføjes til et breakpoint for at specificere, hvornår det skal udløses.
    - Anvendelse: Giver mulighed for at definere mere komplekse logikker for, hvornår debugging skal aktiveres.
- **Call Stack:**
    - Definition: En oversigt over de funktioner eller metoder, der er blevet kaldt, og som stadig venter på at blive afsluttet.
    - Anvendelse: Bruges til at forstå programflowet og identificere, hvor i koden du befinder dig.
- **Variable Inspection:**
    - Definition: Evnen til at inspicere værdierne af variabler på bestemte punkter i din kode under debugging.
    - Anvendelse: Hjælper med at identificere fejl og forstå programmets tilstand.
- **Restart/Resume:**
    - Definition: Funktionen, der giver mulighed for at genoptage udførelsen af programmet efter et breakpoint.
    - Anvendelse: Bruges til at fortsætte gennemførelsen efter at have identificeret et problem eller undersøgt en bestemt del af koden.

Junit

JUnit Testing Cheat Sheet

JUnit Grundlæggende:

1. **Installation af JUnit:**

   - Tilføj JUnit-afhængigheder til dit projekt (for eksempel, via Maven eller Gradle).

2. **Testklasse Struktur:**

   - Opret en testklasse for hver klasse, du vil teste.

   - Annotér testmetoder med `@Test`.

   ```java
   import org.junit.Test;
   ```

```
public class MinKlasseTest {

    @Test
    public void minMetode_ShouldDoSomething() {
        // Testkode her
    }
}
```

3. **Assertion Metoder:**
   - Brug `assertEquals`, `assertTrue`, `assertFalse`, osv., for at validere forventede resultater.

```java
import static org.junit.Assert.*;

@Test
public void minMetode_ShouldDoSomething() {
    assertEquals(forventetResultat, faktiskResultat);
    assertTrue(udtrykErSandt);
    // Andre assertion metoder
}
```

4. **Før og Efter Metoder:**
   - Annotér metoder med `@Before` for at udføre noget før hver testmetode.
   - Annotér metoder med `@After` for at udføre noget efter hver testmetode.

```java
import org.junit.Before;
import org.junit.After;
```

```
public class MinKlasseTest {

    @Before
    public void setUp() {
        // Initialiser ressourcer før hver test
    }

    @After
    public void tearDown() {
        // Ryd op efter hver test
    }
}
```

Test Afvigelser og Exceptions:

5. **Forventede Afvigelser:**
   - Brug `@Test`-attributten til at specificere forventede undtagelser.

   ```java
   @Test(expected = MinException.class)
   public void minMetode_ShouldThrowException() {
       // Kode der kaster MinException
   }
   ```

6. **Timeout for Test:**
   - Angiv en timeout for testmetoder.

```java
@Test(timeout = 1000)
public void minMetode_ShouldFinishInOneSecond() {
    // Kode der skal udføres inden for 1 sekund
}
```

Parametrized Testing:

7. **Parameterized Tests:**
   - Udfør testmetoder med forskellige inputværdier ved at anvende `@RunWith(Parameterized.class)`.

```java
@RunWith(Parameterized.class)
public class MinParametrizedTest {

    @Parameterized.Parameters
    public static Collection<Object[]> data() {
        // Returner en samling af testdata
    }

    @Parameterized.Parameter
    public int input;

    @Parameterized.Parameter(1)
    public int forventetResultat;

    @Test
    public void minMetode_ShouldProduceExpectedResult() {
        // Testkode her med input og forventetResultat
```

```
    }
}
```

Annotations og Test Lifecycle:

8. **Annotationer til Test Lifecycle:**
   - Udnyt annotations som `@BeforeClass` og `@AfterClass` for at udføre noget før og efter testklassen køres.

```java
import org.junit.BeforeClass;
import org.junit.AfterClass;

public class MinKlasseTest {

    @BeforeClass
    public static void setUpClass() {
        // Kode der udføres én gang før testklassen køres
    }

    @AfterClass
    public static void tearDownClass() {
        // Kode der udføres én gang efter testklassen er kørt
    }
}
```

Denne JUnit Testing Cheat Sheet giver en oversigt over de grundlæggende koncepter og metoder inden for JUnit-testning. Brug disse retningslinjer til at oprette og udføre test i Java-projekter ved hjælp af JUnit.