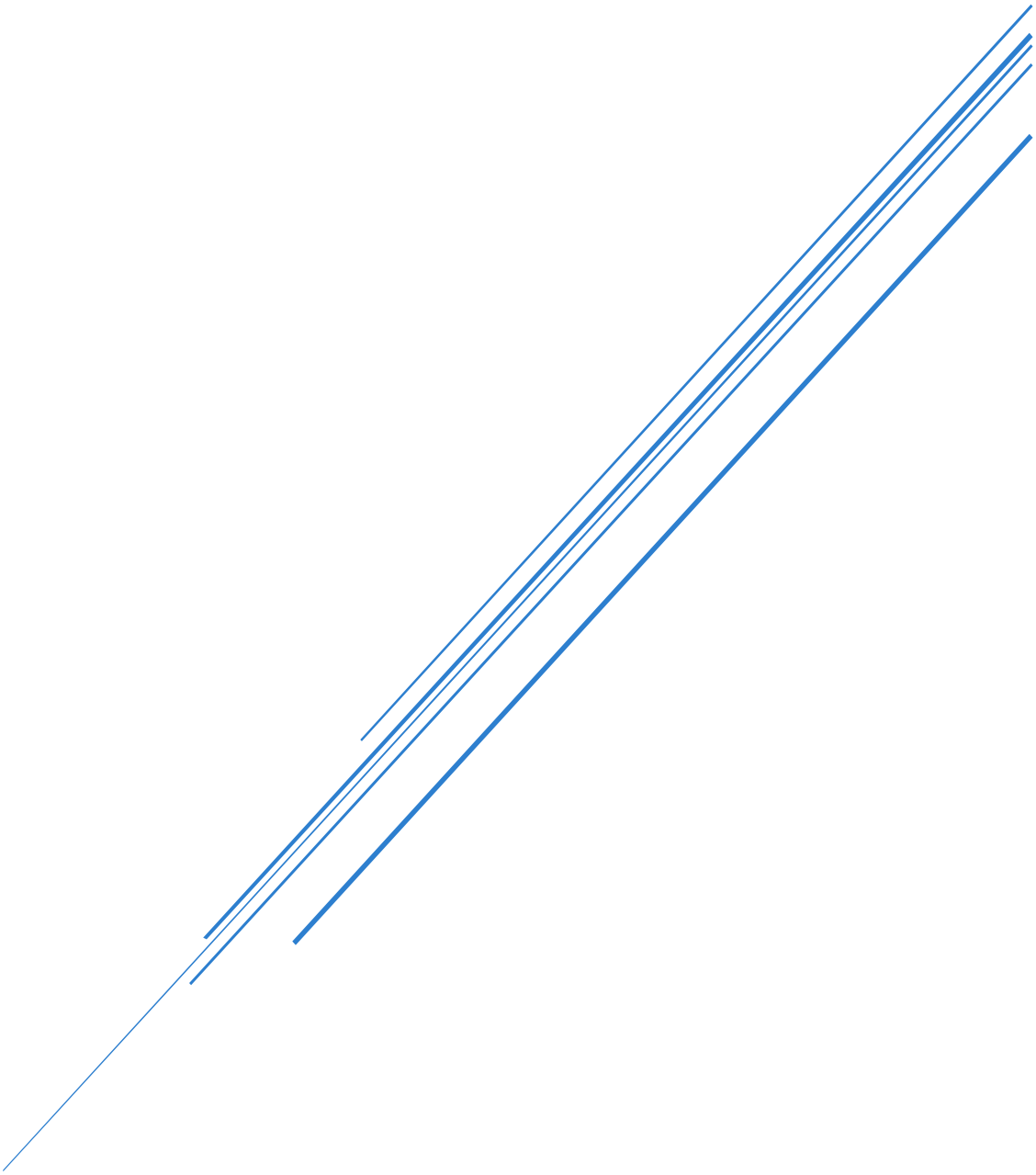
SOFTWARE QUALITY

Assignment 1



Group B

Indholdsfortegnelse

Test Strategy

- IDE: VS
- Sprog: C#
- Værktøjer og frameworks:
    - Unit test
- Typer af test:
    - Unit test
        - Hvad tester du?
        - Hvilken input giver du?
        - Hvad er det forventede output?
        - Hvad er det faktiske output?
        - Består eller fejler testen?
    - Integration test
    - Base test (Black-box test)

Annotations and explanations

Dette ligger i kodet.

- Github link: https://github.com/MrSalim86/GroupB

Reflection and discussion document

Gennem testprocessen har vi opdaget, at testenes effektivitet i høj grad afhænger af, hvor godt de forskellige dele af systemet dækkes, da hver testtype bidrager væsentligt til at sikre den samlede softwarekvalitet.

Vi har lært, at en vellykket teststrategi kræver en omhyggeligt balanceret kombination af forskellige testmetoder og processer for at opnå både høj intern kvalitet (såsom kodekonsistens, stabilitet og vedligeholde) og ekstern kvalitet (som brugervenlighed og overholdelse af brugerkrav).

Ved at anvende en bred vifte af tests og teknikker – herunder unit tests, integrationstests, mutationstests og brug af test doubles – kan vi sikre en omfattende testdækning, identificere og reducere potentielle fejl tidligt i udviklingsprocessen og bekræfte, at softwaren lever op til både tekniske specifikationer og brugernes forventninger.

Denne varierede tilgang styrker vores evne til at levere et produkt af høj kvalitet, der fungerer optimalt både teknisk og i praksis.

En af vores største udfordringer i testprocessen var vores begrænsede viden om de forskellige testmetoder, hvilket betød, at vi måtte bruge meget tid på at undersøge og forstå dem bedre. Dette læringsbehov gjorde, at vi måtte investere ekstra tid i at opbygge den nødvendige ekspertise for at kunne anvende testene effektivt

Testens effektivitet

Test double

Test doubles anvendes ofte, når man arbejder med eksterne tjenester for at undgå, at testene bliver langsomme eller upålidelige. Da eksterne tjenester ofte håndteres af andre teams og kan være udsat for netværksproblemer eller ustabilitet, er test doubles en effektiv måde at sikre, at dine tests forbliver stabile og hurtige.

Her er de fem forskellige test doubles

- Mock
  Mocks er vigtige, når vi skal kontrollere og verificere interaktionerne mellem komponenter. De sikrer, at metoder bliver kaldt med de korrekte parametre og i den rette rækkefølge. Mocks anvendes typisk i tests af metoder, der afhænger af eksterne systemer som databaser eller tredjeparts-API'er. Ved at bruge mocks kan vi simulere og styre den adfærd, vi forventer fra disse eksterne komponenter, og dermed sikre, at vores kode interagerer korrekt med dem.

- Stub
  Stubs er nødvendige, når vi vil isolere en specifik enhed under test ved at returnere foruddefinerede værdier fra metoder. Formålet med en stub er at sikre, at den enhed, der bliver testet, kan fungere korrekt under bestemte betingelser uden at afhænge af de faktiske implementeringer af afhængighederne. En stub kan implementeres med faste værdier for at kontrollere testbetingelserne, såsom at returnere en task med en fast titel. Ved at bruge stubs kan vi undgå kompleksiteten og usikkerheden forbundet med eksterne afhængigheder og fokusere på at teste den specifikke enhed.

- Fake
  Fakes tilbyder en realistisk, men letvægts version af en komponent, som kan simulere funktionaliteten af den rigtige afhængighed uden kompleksiteten. For eksempel, hvis vi bruger en fake service, kan den opretholde en intern liste af tasks og tillade tilføjelse og hentning af data. Fakes anvendes, når vi har brug for et mere realistisk testmiljø, men uden at implementere en fuld version af afhængigheden, som kunne være kompleks og tidskrævende.

- Spy
  Spy anvendes, når vi har brug for at inspicere og registrere interaktioner med enheden under test. Formålet med en spy er at opfange og gemme de kaldte metoder og deres parametre for efterfølgende verifikation. For eksempel, hvis vi bruger en spy til at registrere metoder, der blev kaldt på en afhængighed, kan vi kontrollere, hvilke metoder der blev kaldt, og med hvilke argumenter. Dette er nyttigt til at verificere, at interaktioner mellem komponenter sker som forventet uden at ændre deres adfærd.

- Dummy
  Dummy anvendes som placeholders, der opfylder kravene uden at have nogen funktionalitet. Dummies er nyttige, når vi har brug for at opfylde parameterkrav i en

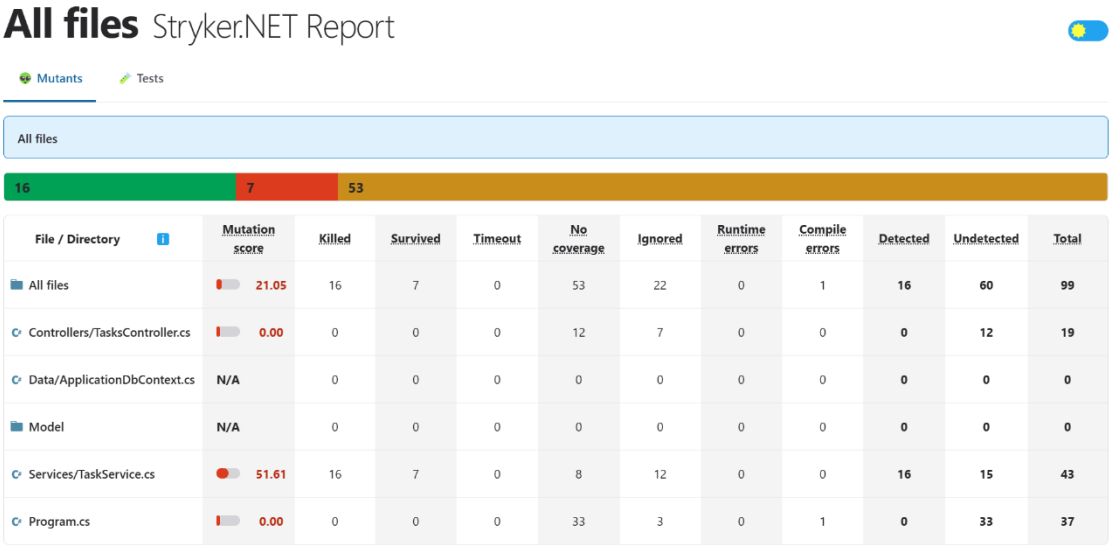
metode eller konstruktor, men ikke har behov for, at de faktiske objekter udfører nogen handlinger. For eksempel, en dummy service kan være nødvendigt for at opfylde et parameterkrav i en metode, men dens implementation vil ikke påvirke testresultaterne. Dummies sikrer, at vores testkode kan kompilere og køre korrekt uden at indføre unødvendig kompleksitet.

Samlet set er test doubles en uundgåelig del af en effektiv teststrategi, især når man arbejder med eksterne afhængigheder og komponenter. Test doubles hjælper med at isolere testene fra eksterne faktorer og sikrer, at de bliver kørt under kontrollerede og stabile forhold. Ved korrekt anvendelse af stubs, mocks, fakes, spies og dummies kan vi opnå pålidelige testresultater, som bidrager til en robust og fejlfri softwareudviklingsproces.

Mutation testning

Mutationstest er en teknik, der bruges til at evaluere kvaliteten af ens tests ved at indføre små ændringer i vores kode for at se, om vores tests kan opdage og "dræbe" disse fejl.

I vores rapport ser det således ud:



All files Stryker.NET Report

Mutants | Tests

All files

16 | 7 | 53

File / Directory	Mutation score	Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
All files	21.05	16	7	0	53	22	0	1	**16**	**60**	**99**
Controllers/TasksController.cs	0.00	0	0	0	12	7	0	0	**0**	**12**	**19**
Data/ApplicationDbContext.cs	N/A	0	0	0	0	0	0	0	**0**	**0**	**0**
Model	N/A	0	0	0	0	0	0	0	**0**	**0**	**0**
Services/TaskService.cs	51.61	16	7	0	8	12	0	0	**16**	**15**	**43**
Program.cs	0.00	0	0	0	33	3	0	1	**0**	**33**	**37**

Vi har testet vores *TaskServer.cs*, hvor vores forretningslogik ligger. Vores rapport viser, at en stor del af koden er dækket af tests, men de 7 overlevende mutanter indikerer svagheder i vores testdækning. Derudover er der 8 mutanter uden dækning, hvilket peger på, at visse dele af vores service-logik kræver yderligere tests. De 12 ignorerede mutanter kan tyde på, at filen er kompleks, og det vil være nødvendigt at analysere den nærmere. Samlet set viser resultaterne af vores mutationstest, at vores tests er delvist effektive, men der er markante muligheder for forbedring, især når det gælder testdækning.

Verification and validation

Refleksionen over forskellen mellem verifikation og validering, som beskrevet i Plutora-artiklen, understreger vigtigheden af begge processer i softwareudviklingen, da de fokuserer på forskellige mål, men begge er nødvendige for at opnå det endelige resultat

Verifikation sikrer, at produktet udvikles korrekt i henhold til de specificerede krav gennem aktiviteter som kodegennemgange og inspektioner. Det er en statisk proces, der foregår tidligt i udviklingen.

Validering sikrer, at det færdige produkt opfylder kundens behov og forventninger gennem testning. Det er en dynamisk proces, der sker sent i udviklingsforløbet.

Kort sagt handler verifikation om at bygge produktet rigtigt, mens validering handler om at bygge det rigtige produkt. Begge er nødvendige for at sikre både kvalitet og brugertilfredshed.

Verificering i Unit Tests:

Unittests kontrollerer små stykker af kode (som funktioner eller metoder) for at sikre, at de fungerer korrekt ifølge deres tekniske specifikationer.

Validering i Unit Tests:

Det er vanskeligere at dække validering i en enhedstest, da de normalt kun tester isolerede stykker kode. Vi har dog forsøgt at validere, om de enkelte metoder opfylder brugerkrav ved at inkludere tests for brugsscenarier, der svarer til virkelige situationer.

Softwarekvalitet:

Vores teststrategi dækker flere aspekter af softwarekvalitet, men der er plads til forbedring, især set i lyset af Peter Morlions synspunkter om softwarekvalitet.

Intern og Ekstern Kvalitet:

Vi har i høj grad fokuseret på testtyper som unit tests og integrationstests, som er essentielle for at opnå høj intern kvalitet. Disse tests sikrer, at de enkelte komponenter fungerer korrekt, og hjælper os med at vurdere kodekompleksitet og vedligeholdelighed. Ifølge Morlion bør vi dog også i højere grad overveje, hvordan vores teststrategi kan understøtte ekstern kvalitet – altså, hvordan slutbrugerne oplever systemet. For at opnå dette kunne vi inkludere tests, der simulerer realistiske brugsscenarier, såsom end-to-end tests, som kan identificere potentielle problemer i brugeroplevelsen.

Forbedring af vores teststrategi:

- **Automatisering af flere testtyper:** Udover unit- og integrationstests kunne vi udvide med flere specifikationsbaserede tests for at sikre, at softwaren opfylder kundens krav. Desuden bør vi overveje at implementere performance-tests for at måle, hvordan systemet opfører sig under belastning.

- **Mutationstestning:** Vores nuværende mutationstestning giver indsigt i, hvor vores testdækning kan forbedres. Vi kunne dog også fokusere på at skrive mere robuste tests, der kan "dræbe" mutanter og dermed øge testernes effektivitet.
- **Feedback loops:** I praksis kan vi anvende feedback fra mutationstestning samt brugerfeedback til løbende at forbedre både intern og ekstern kvalitet.
- **Tættere kobling mellem verifikation og validering:** Mens vi i høj grad fokuserer på verifikation gennem vores unit tests, er det vigtigt også at inkludere validering. Det vil sige, at vi i vores teststrategi bør integrere tests, der sikrer, at softwaren fungerer som forventet i praksis, ikke blot som specificeret i koden.

Test Categories

Martin Fowler identificer forskellige testkategorier, herunder unit tests, integration tests, component tests, broad stack tests, business facing tests, story tests, contract tests, subcutaneous tests, user journey tests, Og threshold tests.

Hver kategori har et specifikt formål, fra at teste individuelle funktioner og integration mellem komponenter til at sikre, at hele systemet fungerer som forventet fra brugerens perspektiv.

I vores teststrategi har vi brugt unit tests til at kontrollere individuelle funktioner, integration tests til at vurdere samspillet mellem vores systemer.

Vi har anvendt test doubles som mocks og stubs til at simulere eksterne afhængigheder. Mutationstestning har hjulpet med at evaluere testdækning og effektivitet, mens verifikation og validering har sikret både korrekt udvikling og opfyldelse af brugerkrav