**Newtons rapportmall**

**”Testning”**

**Kurs: Testning**

**Klass: SYSM8**

**Termin och år: VT/2025**

**Författare: Ture Christenson**

**Lärare: Benjamin Berglund**

1. Sammanfattning

Rapporten ser över kvaliteten och testbarheten i BudgetApp, en budgeteringsapplikation gjord med Razor pages. Genom att använda flera olika testmetoder som enhetstestning, BDD-scenariotestning och explorativ testning har jag sett både styrkorna i applikationen och områden där förbättringar behövs. Resultaten applikationen har en funktionell grundstruktur men skulle gynnas av en tydligare separation mellan logiken och presentationen, förbättrad felhantering och en mer robust verifiering av användardata.

Innehållsförteckning

[1. Sammanfattning 2](#_Toc535918580)

[2. Inledning 4](#_Toc535918581)

3. Arbetet och avslutande diskussion

[5. Referenslista 4](#_Toc535918584)

1. Inledning

Innan jag började med själva testningen så såg jag positiva aspekter och begränsningar i delar med testbarheten. De positiva aspekterna med appen är att det finns en tydlig användning av page model-mönster med en bra separation mellan dataåtkomst och presentation. Dependency injection används konstant genom konstruktorn för DBContext och SignInManager. Metoderna är bra separerade för varje åtgärd. Datamodellerna är tydliga.

Det finns saker i appen som begränsar testningen. Som att logiken är blandad med UI-logiken i page model, direkta databasanrop i page-model metoderna som gör att det försvårar enhetstestning. Applikationen har även en begränsad felhantering.

1. Arbetet och avslutande diskussion

De testmetoder som användes i applikationen var en explorativ testning genomfördes först för att förstå applikationens funktionalitet och struktur. Genom att utforska gränssnittet identifierades grundfunktioner, navigationsflöden och potentiella riskområden. Detta var viktigt för att förstå hur användaren använder applikationen.

Analytisk testning baserades på en analys av kodens struktur och identifierande risker. Det hjälpte till med att prioritera testfallen och potentiella problemområden i koden.

Sen så hade jag även gränssnittstestning med specflow som validerade att UI-komponenterna fungerade som det skulle och att användarflöderna var funktionella. Dessa tester var var datadriven testning där samma logik genomfördes med olika indata för olika kategorier av utgifter.

Enhetstester fokuserade på datamodeller och affärslogik:

* Validering av Expense-modellens egenskaper och relationer
* Test av GetExpensesByCategory-metoden för korrekt summering
* Validering av affärsregeln om kategorifördelning

BDD-tester med SpecFlow och Playwright för användarflöden:

* Scenario: Lägga till och sedan ta bort en utgift
* Scenario: Lägga till en utgift i Utilities-kategorin
* Scenario: Redigera en befintlig utgift
* Scenario: Ta bort alla utgifter och återställa budgeten

Det jag upptäckte i BDD testet var att grundläggande funktioner som att lägga till, redigera och ta bort utgifter fungerar som förväntat och att budget kan sättas och återställas korrekt.

De svårigheter jag hade i testningen var att hantera popups och JavaScript-dialoger som dök upp oväntat eller inte fångades korrekt. För att lösa detta använde jag en dialog-händelse i Playwright, där jag automatiskt accepterade dialogerna och satte en flagga för att hålla koll på om en dialog visades. Dessutom implementerade jag en fallback för att hantera bekräftelseknappar på sidan om ingen JavaScript-dialog dök upp. Denna lösning säkerställde att testerna inte blockerades av popups och gjorde dem mer robusta. Det var svårt att få rätt på det.

Jag stötte på problem med att hitta tillförlitliga selektorer för vissa element, särskilt när det gällde knappar utan tydliga identifierare. För att lösa detta skapade jag alternativa selektorer baserade på både text och CSS-klasser, vilket gav redundans i mina testscript.

För att förbättra applikationens testbarhet tänker jag:

1. Separera affärslogik från presentationslogik

Implementera ett service-lager som hanterar affärsregler och datamanipulation. Detta separerar logiken från presentation genom att skapa engagerade klasser (som exempelvis en ExpenseService) som innehåller all logik för datahantering, validering och beräkningar, vilket gör koden mer testbar. Låta Page Model fokusera på presentation och hantera användarinmatning.

1. Förbättra felhanteringen

Implementera en konsekvent felhantering i alla databasoperationer. Ge meningsfull återkoppling till användaren vid fel

1. Använda id-attribut för viktiga UI-element

Lägga till tydliga ID-attribut på alla interaktiva element för enklare testing. Skapa en konsekvent namngivningskonvention för DOM-element. En konsekvent namngivningskonvention skulle göra koden mer testbar

1. Referenslista

<https://sys8-newton.lms.nodehill.se/>

[*https://github.com/Svenpaj/shoptester*](https://github.com/Svenpaj/shoptester)