Dokumentation LAB 2 - Implementera enhetstest

Elena Kappler, Ruying Wei, Eric Nylander

1. Introduktion

Enhetstester spelar en avgörande roll för kvalitetssäkring i mjukvaruutveckling eftersom de säkerställer att varje del av koden fungerar som avsett och hjälper oss att upptäcka potentiella problem i tid, vilket i sin tur förbättrar kodens underhållbarhet. En bra skriven kod bör vara enkelt att testa, detta motiveras i boken "Att arbeta med systemutveckling" (Gustavsson & Görling, 2019, s.159) där de tar upp hur kritiskt det kan vara att hitta fel för sent, ett exempel kan vara att många, att ett system till peacemakers, kärnkraftverk och börsrelaterade problem. Att hitta fel i de miljöerna kan leda till allvarliga skador i samhället och privatpersoner eller de vanligaste scenariot att kunden behöver göra ett driftavbrott som kommer skapa stora intäktsbortfall och dålig PR (Gustavsson & Görling, 2019, s.159).

I vårt utvecklingsarbete har vi genomfört enhetstester för spelet Bysen. Denna rapport syftar till att ge insikt i hur enhetstester planeras, implementeras och analyseras, samt att reflektera över de lärdomar vi dragit av denna process i relation till de principer och metoder som behandlats i kursen. Genom att noggrant beskriva varje steg i arbetet och visualisera processen med ett flödesdiagram, strävar vi efter att få en tydlig översikt över vårt arbetsflöde, identifiera förbättringsområden samt stärka vår förmåga att kommunicera tekniska processer på ett strukturerat sätt.

2. Metod

2.1 Design och Skapande av Flödesdiagram

Flödesdiagrammet visar processen som vi arbetar med (se bild 1). Steget som tas först är att identifiera enheter som kan testas, till exempel kan vi leta efter publika metoder att skapa testfall för. Om ett testfall identifieras dokumenterar vi vad som ska testas och resultatet som förväntas för att säkerställa att vi har en tydlig förståelse. Nästa steg är att skapa ett testscenario i en separat miljö för att undvika påverkan på den befintliga koden under testningen. Till exempel kan man skapa en miljö där man testar funktioner isolerat, utan att påverka hela systemet, genom att använda mock-objekt för att simulera olika delar av programmet. Om ett test skulle misslyckas ska problemet identifieras. Det kan vara olika

anledningar som till exempel inmatningsfel, logiska fel eller glömt felhantering. Dessutom kan det hända att enheten inte kan testas individuellt för att den gör för många saker samtidigt och måste delas upp i mindre, testvänliga delar. Att identifiera orsaken till problem hjälper oss inte bara att snabbt återgå till testningen, utan också att identifiera och åtgärda brister i koden, vilket förbättrar testens effektivitet och kodens tillförlitlighet. Oberoende på om testet lyckas dokumenterar vi hur det gick. Genom dessa steg kan vi systematiskt genomföra tester, säkerställa kodens tillförlitlighet och funktionens korrekthet.

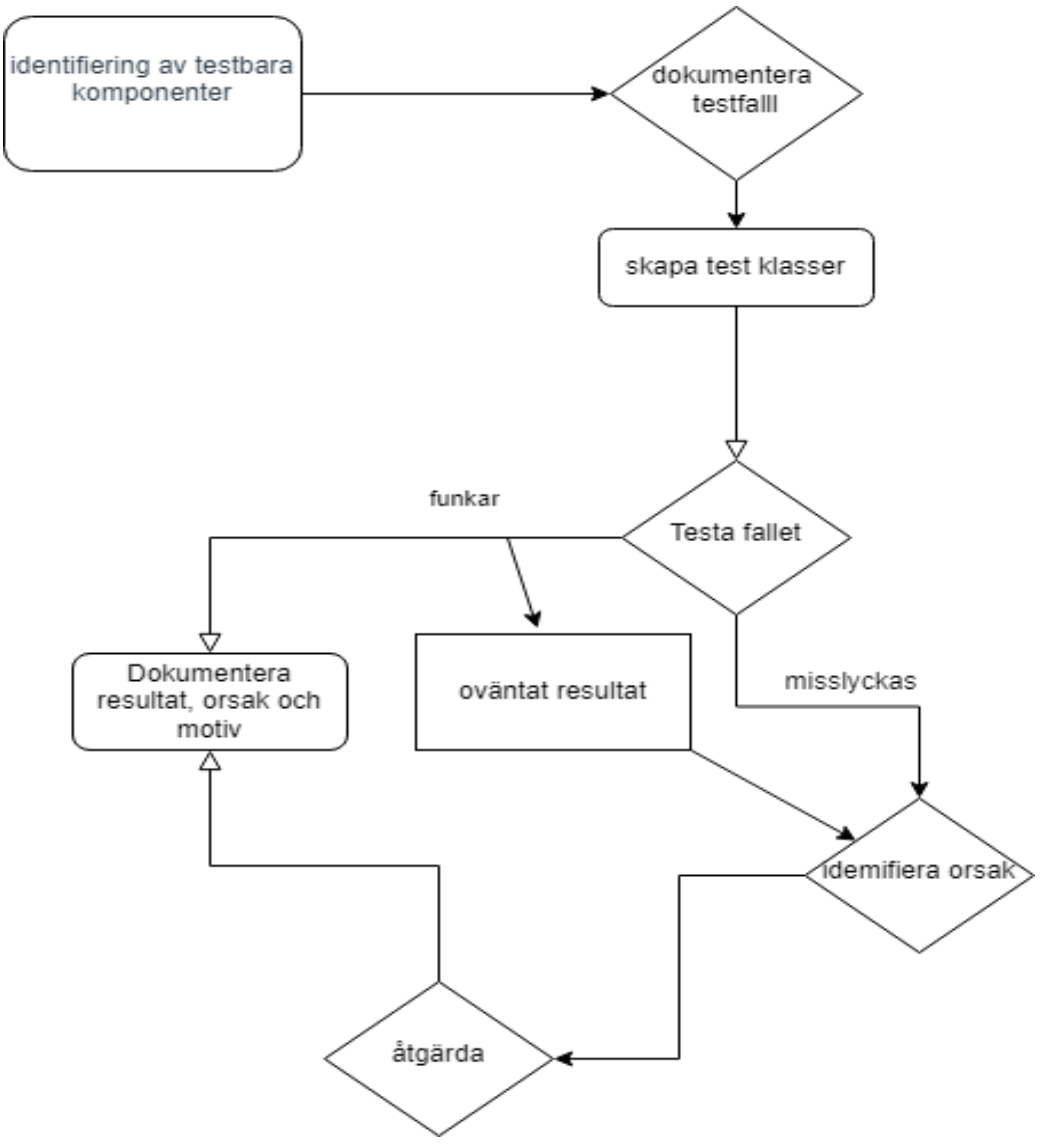

Bild 1. Flödesdiagram - Arbetsprocess.

2.2 Arbetsprocess för Enhetstester

Vi använder IntelliJ som IDE för utvecklingen och JUnit ramverket för att implementera våra tester. Vår arbetsprocess skulle kunna beskrivas som en mix av Code & Fix och parprogrammering. Vi sitter två stycken som programmerar och en som antecknar. Genom att arbeta i par kan vi dela erfarenheter med varandra samt diskutera och lösa problem som uppstår i tid, vilket förbättrar både kodens kvalitet och utvecklingseffektivitet. Antecknaren säkerställer att alla ändringar och diskussioner dokumenteras i detalj. Detta ger oss värdefulla referenser för att senare granska och analysera arbetsframstegen.

För att identifiera testbara komponenter går vi igenom koden för att hitta funktioner som returnerar ett värde då dessa är enklast att testa. Vi identifierar några men kommer sedan ihåg att privata metoder inte behöver testas, vilket gör att vi behöver tänka om. Här upptäcker vi att vår kod innehåller mest privata metoder och vi har svårt att hitta publika metoder som är "testbara". Detta beror på strukturen i koden som vi förändrade i samband med föregående uppgift. Även om vi tycker att vi har förbättrat den originala koden är den fortfarande långt ifrån bra vilket visas nu när vi försöker hitta testfall.

Vi upptäcker även många beroenden i koden, som till exempel publika variabler i Game klassen. Vi bestämmer oss för att redigera koden och skapa publika getter och setter metoder för 'messages' variabeln för att kunna testa dem.

Vidare går vi igenom delar av spelet som borde testas för att de är integrala för spelet. Det kommer upp grafiska spelkomponenter och frågan hur de kan testas. Det är här vi försöker använda Mockito-ramverket som dock inte fungerar som vi önskar. Därför använder vi ett 'riktigt' objekt istället för en mock. Vi upptäcker även att metoderna i GameGraphics klassen är väldigt stora och gör för många saker samtidigt vilket gör det svårt att testa. För de metoder som innehåller mycket kod delar vi upp dem i mindre delar för att bättre kunna testa enskilda funktioner. Till exempel extraherar vi en funktion för att beräkna spelarens position som vi kan testa på ett enkelt sätt.

3. Resultat

I GameGraphics klassen testar vi metoden calculatePlayerPosition() som ska beräkna spelarens position i rummet (se Bild 2). Detta gör vi för att kunna säkerställa att spelaren är på rätt position på spelfältet. Vi matar in fiktiv data för att testa att beräkningen stämmer överens med vår resultat. Testet var framgångsrik direkt från början och det behövdes inga ändringar.

```
@Test
void calculatePlayerPosition() {
    Room room = new Room( x: 100,  y: 100);  // Fiktiv konstruktor
    int roomSize = 50;
    int playerSize = 10;

    Point result = gameGraphics.calculatePlayerPosition(room, roomSize, playerSize);

    assertEquals( expected: 120, result.x);
    assertEquals( expected: 138, result.y);
}
```

Bild 2. Testmetod för calculatePlayerPosition().

I Game klassen testar vi getMessages(), setMessages och startNewGame(). Här stöter vi på lite mer problem från början. Vid första försöket misslyckades testet för getMesages() för att vi testar att listan som returneras inte är null vilket den dock är från början spelet. Listan skapas först när spelet initieras. Därför initierar vi messages i setUp metoden först och testar att den inte är null sedan. Vi testar även storleken av messages som ska vara ett vid tillfället. Båda dessa är positiva tester. Sedan lägger vi till ett negativt test som ser om metoden fungerar även om listan är tom.

```
//positive test
new *
@Test
void getMessages() {
    assertNotNull(game.getMessages());
    assertEquals( expected: 1, game.getMessages().size());
}

//negative test
new *
@Test
void testGetMessagesWhenEmpty() {
    game.setMessages(List.of());
    assertNull(game.getMessages());
}
```

Bild 3. Testmetoder för getMessages().

För att testa setMessages() skapar vi en lista med en sträng som vi sedan använder för att uppdatera game.messages. Testet ser till att den skapade listan är densamma som den som har uppdaterats i spelet vilket gör det till ett positivt testfall. Här lyckas testet direkt. Vårt negativt test provar att man inte kan sätta messages till null. Det misslyckas först för att vi har glömt att hantera fallet i metoden. Efter att vi la till null exception handling gick testet igenom.

```
//positive test
new *
@Test
void setMessages() {
    List<String> messages = new ArrayList<>();
    messages.add("Test message 2");
    game.setMessages(messages);
    assertEquals(messages, game.getMessages());
}

//negative test
new *
@Test
void testSetMessagesWithNull() {
    Game game = new Game();
    assertThrows(NullPointerException.class, () -> game.setMessages(null));
}
```

Bild 4. Testmetoder för setMessages().

Till slut testar vi startNewGame() metoden genom att anropa den och sedan prova att gameOver är false och playerName inte är null, alltså att initieringen av variablerna fungerar korrekt. Vid körning måste man mata in ett namn vilket inte är optimalt. Här skulle vi kunna implementara fler testfall vilket vi dock skulle behöva mer tid för. Vårt negativt test provar att inga fel uppstår när man startar ett nytt spel medan spelet inte är över än, vilket kan även anses som gränsfall. Detta test lyckas också direkt och vi behöver inga fler åtgärder.

```
//positive test
new *
@Test
public void startNewGame() {
    game.startNewGame();
    assertFalse(game.gameOver);
    assertNotNull(game.getPlayerName());
}

//negative test, edge-case
new *
@Test
void testStartNewGameWhenGameIsNotOver() {
    assertDoesNotThrow(() -> game.startNewGame());
}
```

Bild 5. Testmetoder för startNewGame().

4. Diskussion

4.1 Analys av Arbetsprocessen

Det framgår tydligt att vår kod inte är särskilt “test vänlig”. Nästan alla testbara metoder är privata och behöver därför inte testas, vilket tyder på brister i processens kvalitetsaspekter. De flesta metoder i Game klassen är privata som skapar problem att testa de olika enheterna individuellt. Detta är ej en bra process för att bygga program då det gör att testa individuella funktioner betydligt svårare och ett krav på flera verktyg. Detta gör det svårare att upptäcka individuella fel i metoder som inte kan ses i sammankoppling resten av systemet.

En bra struktur i koden är viktigt för att kunna utföra lämpliga enhetstester. Vi har inte lyckats med att dela upp koden tillräckligt tidigare vilket gör det ytterligare svårare att skriva tester.

4.2 Förbättringsmöjligheter

Det krävs bättre förarbete eller mer kvalitativ bra kod för att kunna skriva bra tester. Därför kan det vara smidigare att analysera kvaliteten innan man börjar identifiera tester och om så behövs förbättra kvaliteten. Det kan innebära t.ex. lösa kopplingar mellan klasser för mindre ömsesidig beroende såsom mindre metoder som följer Single Responsibility principen.

Det kan vara av fördel att använda GitHub för att enklare dela uppdaterad kod med alla gruppmedlemmar. Anledningen till att vi inte använde det är att vi ofta jobbar på koden samtidigt och inte alla är väl bevandrade i användningen av GitHub. Därför bestämde vi att det tar för mycket tid att sätta upp.

5. Slutsats

I denna uppgift har vi genomfört enhetstester för spelet Bysen och analyserat de utmaningar och förbättringsmöjligheter som uppstod under testprocessen. Vi upptäckte att privata klasser och dålig kodstruktur hade en betydande påverkan på effektiviteten av enhetstesterna. Privata klasser begränsar möjligheten att testa enskilda funktioner. Därför har vi insett att en bra kodstruktur och åtkomstkontroll för funktioner (public eller private) är avgörande för att göra enhetstester.

Att förbättra testprocedurer är viktigt från ett samhällsperspektiv och för kunden för att spara pengar då kostnader för att korrigera en bugg blir lägre om det hittas tidigt och ibland kan det

vara omöjligt att åtgärda om produkten är i drift då att uppdaterad programvara utan stora kostnader, ett exempel på företag som drabbats av detta är bilföretag som får återkalla tusentals bilar på grund av fel programvara (Gustavsson & Görling, 2019, s.160). Därför borde koden som skrivs vara lätt att testa för att kunna hitta individuella funktionsproblem. Men också att utveckla verktyg för att testa kod.

Referenser

Gustavsson, T. & Görling, S., 2019. Att arbeta med systemutveckling. Upplaga 1.

Skolmaterial (PDF, Läsmaterial, Presentationer)