Högskolan i Gävle

Laborationsrapport

Objektorienterad Design och Programmering, 7,5 hp

HT 19, period 1

Laboration 1

av

Thomas Lundgren
900802-1835

thomaslundgren@live.com

Institutionen för matematik, natur- och datavetenskap
Högskolan i Gävle
S-801 76 Gävle, Sweden

Innehållsförteckning

1 Inledning

Syftet med denna laboration var att, utgående från given Java-kod, designa interface och klass(er) för en räknare. Detta för att studenten skulle:

- ta ett steg mot att uppnå de övergripande kursmål, se kursplanen.
- förstå och använda begrepp såsom: arv, abstrakt klass, interface, kommunikation mellan objekt, konstruktor, overload, override, super, this.
- förstå och använda klassdiagram i UML och träna på att identifiera/analysera relationer mellan klasser.
- lära sig att skapa tester (JUnit) och därigenom att avlusa programvara och säkerställa robust kod.

Räknaren skulle kunna räkna uppåt eller nedåt, ett steg i taget, likt en sådan räknare som används för att räkna antalet passagerare som kliver på ett flygplan. Användaren skulle också kunna ange ett tröskelvärde, som, när räknaren når detta tröskelvärde så börjar den om att räkna ifrån startvärdet.

Instanser av räknar-klassen skulle kunna kopplas ihop så att när en räknare har nått sitt tröskelvärde så räknar den upp ett steg på en annan räknare. Tre stycken räknare skulle kopplas ihop på detta sätt för att bilda en klocka, där en räknare räknar sekunder, en annan räknar minuter och en tredje räknar timmar.

All kod skulle enhetstestas med JUnit-tester.

2 Metod och utförande

2.1 Gränssnittet Counter och de abstrakta klasserna AbstractCounter och LinkedAbstractCounter

Arbetet inleddes med att studera den givna koden . I den givna koden gavs en delvis implementerad abstrakt klass, `AbstractCounter`, som beskrev en räknare. Alla metoder (utom `toString()`, som ärvs ifrån klassen `Object`) som annoterats med `@Override` antogs ingå i gränssnittet (eng. interface) `Counter` som den abstrakta klassen implementerade. Ett interface skapades innehållande dessa metoder.

Klassen `AbstractCounter` saknade små delar av implementationen, men framför allt var det tillgångsmodifierare (eng. access modifiers) för klassens instansvariabler som saknades. Här valdes den mest begränsande tillgångsmodifieraren som var möjlig, nämligen `protected`.

I den givna koden hade `AbstractCounter` funktionalitet för att hålla en referens till ett annat `Counter`-objekt som instansvariabel så att `Counter`-objekt skulle kunna länkas ihop. Denna funktionalitet togs bort ur `AbstractCounter` och flyttades till en egen abstrakt klass vid namn `AbstractLinkedCounter`. Detta för att undvika `Null`-referenser i `AbstractCounter` om en användare önskar en fristående, icke-sammankopplad, räknare. `AbstractLinkedCounter` tillåter inte `Null`-referenser, vilket innebär att användaren tvingas använda `AbstractCounter` för fristående räknare.

2.2 Klasserna Counter60, Counter24 och DecreasingCounter10

Tre konkreta implementationer av räknare skapades: `Counter60`, `Counter24` och `DecreasingCounter10`.

`Counter60` utökar (eng. extends) klassen `AbstractLinkedCounter` och tar således emot en referens till en annan räknare genom beroendeinjicering (eng. dependency injection) genom konstruktorn. `Counter60`-objektet räknar upp den injicerade räknaren när den själv når 60.

`Counter24` utökar klassen `AbstractCounter` och är således en fristående räknare som slår om till noll efter att den har räknat till 23.

`DecreasingCounter10` skapades enbart för att kunna testa beteendet hos räknare som räknar nedåt.

2.3 Klassen ClockCounter

Klassen `ClockCounter` är en klass som fungerar som en klocka, ihopbyggd av tre `Counter`-objekt. Ett `Counter60`-objekt används som räknare för sekunder och minuter. Sekundräknaren har en referens till minuträknaren. När sekundräknaren har räknat till en minut räknar den upp minuträknaren ett steg. Minuträknaren har en referens till ett `Counter24`-objekt som får agera som timräknare. När minuträknaren har räknat till 60 minuter räknar den upp timräknaren ett steg.

I `ClockCounter`-klassen finns en metod `startCount()` sätter igång `ClockCounter`-objektet. Det kommer då att räkna upp en sekund varje sekund. Genom att skicka in ”enumeration”-objektet `PrintTime.YES` eller `PrintTime.NO` kan användaren välja att skriva ut tiden till `System.out`, standardströmmen för utskrift av data.

2.4 Enhetstester med JUnit

Enhetstester skrev för klasserna `Counter24`, `Counter60`, `DecreasingCounter10` och `ClockCounter`.

Alla klasserna testades i en egen testfil. Testerna tester de flesta intressanta aspekter hos räknarnas beteenden. Alla testfiler innehåller tester som testar beteendet hos räknarna när de når sitt tröskelvärde.

2.5 Klassdiagram

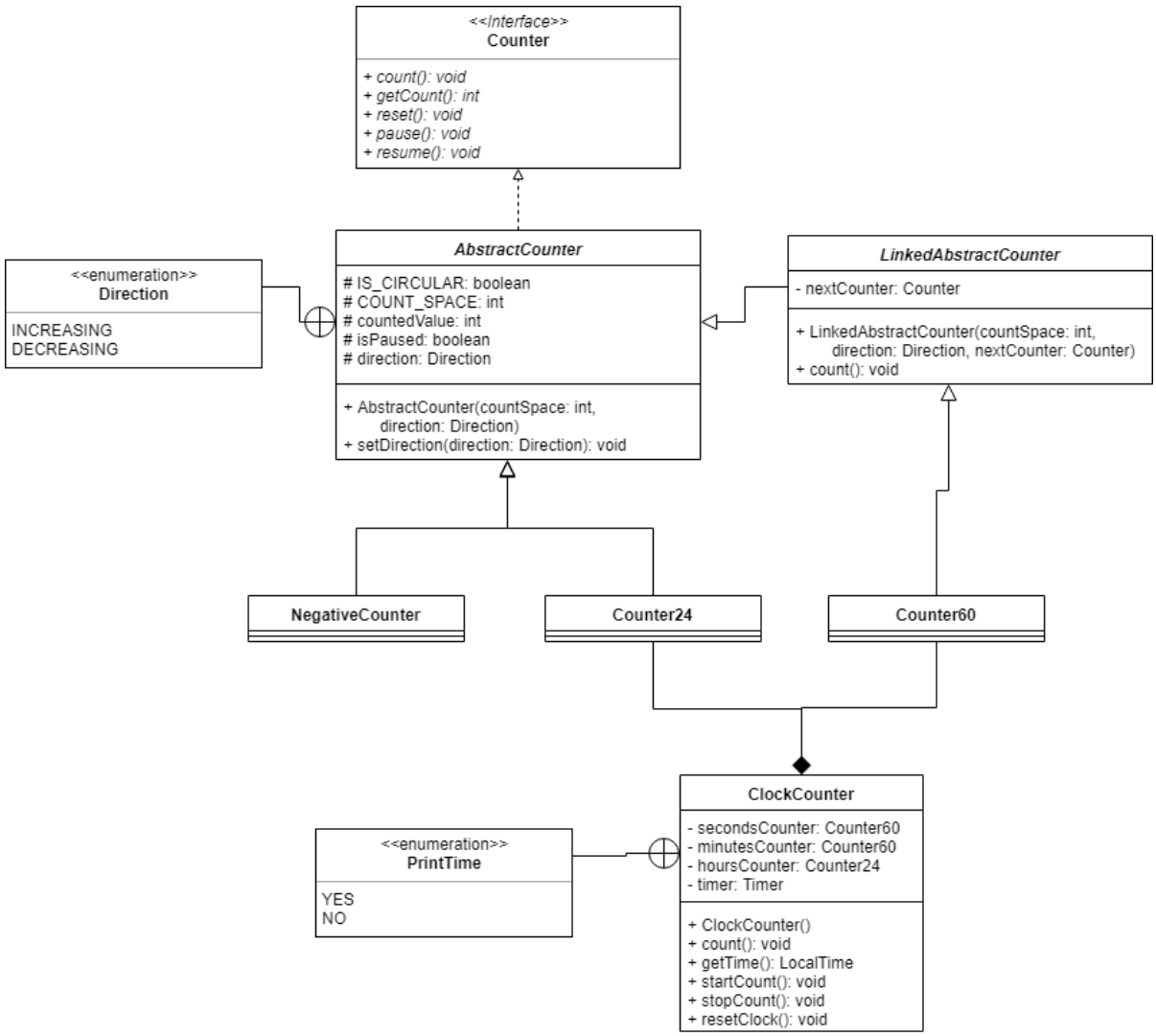
Ett klassdiagram över alla klasserna (utom testklasserna) gjordes med hjälp av ritverktyget `draw.io`.

3 Resultat

Implementationen av koden redovisas inte närmare här. För källkod, se bifogad `zip`-fil.

3.1 Klassdiagram

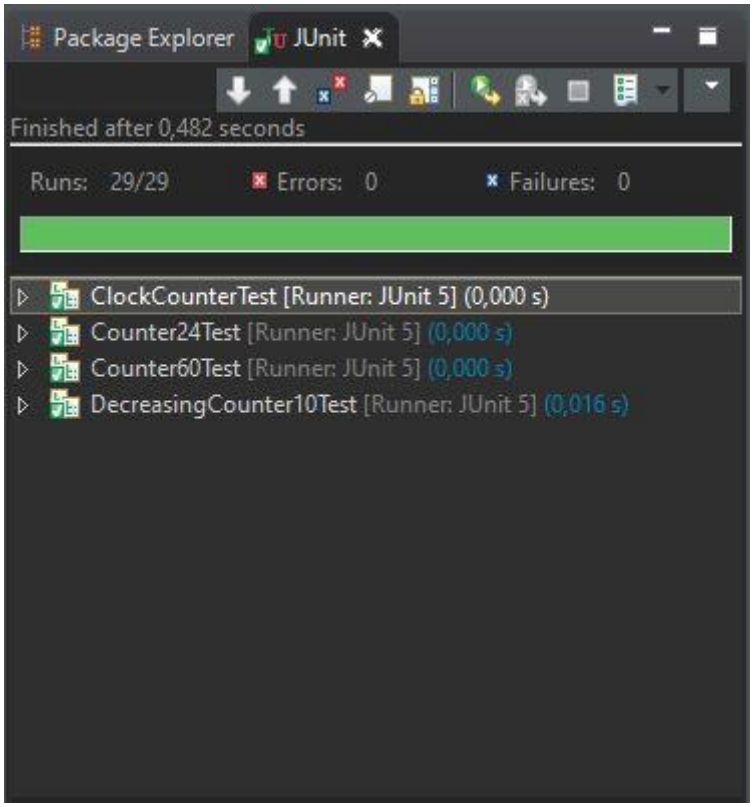
UML-klassdiagrammet nedan beskriver systemet som har producerats, se Figur 1. Nedärvda variabler och metoder har utelämnats ur klassdiagrammet för att göra det mer överskådligt. Observera att `LinkedAbstractCounter` har en egen implementation av metoden `count()` varför den ingår i klassdiagrammet.



Figur 1: UML-klassdiagram för systemet som har producerats.

3.2 Enhetstester med JUnit

29 enhetstester skrevs. Figur 2 visar resultatet av exekvering av testerna med JUnit i Eclipse.



Figur 2: Exekvering av alla enhetstester.

4 Diskussion

4.1 Implementation av AbstractCounter och LinkedAbstractCounter

Funktionaliteten för att länka samman räknare extraherades från `AbstractCounter` och lades i `LinkedAbstractCounter` istället. Detta för att undvika `Null`-referenser i instansvariabeln `nextCounter` då användaren önskar skapa en fristående räknare. På detta sätt tvingas användaren vara mer explicit i vilken sorts räknare som önskas, men potentiella problem med `Null`-referenser elimineras.

4.2 Enhetstester med JUnit

I enhetstesterna används Roy Osheroves föreslagna namngivningskonvention. Den går ut på att ett enhetstests namn delas in i tre delar. Han beskriver det som:

> [UnitOfWork_StateUnderTest_ExpectedBehavior]

vilket kan läsas på hans blogg [1].

5 Referenser

[1] R. Osherove, “Naming standards for unit tests - Blog - Osherove,” 2005. [Online]. Available: http://osherove.com/blog/2005/4/3/naming-standards-for-unit-tests.html. [Accessed: 09-Sep-2019].