|  |
| --- |
| Högskolan i Gävle |
| Laborationsrapport |
| Objektorienterad Design och Programmering, 7,5 hp  HT 19, period 1  Laboration 3  av  *Thomas Lundgren*  *900802-1835*  [*thomaslundgren@live.com*](mailto:thomaslundgren@live.com) |
|  |
| **Institutionen för matematik, natur- och datavetenskap**  **Högskolan i Gävle**  S-801 76 Gävle, Sweden |
|  |
|  |

**Innehållsförteckning**

[1 Inledning 1](#_Toc20223754)

[1.1 Syfte och riktlinjer 1](#_Toc20223755)

[1.2 Specifikation av alarmklocka 1](#_Toc20223756)

[2 Metod och utförande 2](#_Toc20223757)

[2.1 Utökat beteende hos Counter-klasserna 2](#_Toc20223758)

[2.2 Gränssnittet TimeType och klassen Time 2](#_Toc20223759)

[2.3 Gränssnittet AlarmType och klassen Alarm 2](#_Toc20223760)

[2.4 Klassen AlarmManager 2](#_Toc20223761)

[2.5 Gränssnittet AlarmActionType och klassen PrintAlarmAction 3](#_Toc20223762)

[2.6 Gränssnitten ClockType och AlarmClockType, klasserna WeekClock och WeekAlarmClock 3](#_Toc20223763)

[2.7 UML-klassdiagram 3](#_Toc20223764)

[2.8 Enhetstester med JUnit och testtäckning med EclEmma 3](#_Toc20223765)

[3 Resultat 4](#_Toc20223766)

[3.1 Counter 4](#_Toc20223767)

[3.2 Time 4](#_Toc20223768)

[3.3 Alarm 5](#_Toc20223769)

[3.4 Clock 6](#_Toc20223770)

[3.5 Beroenden mellan klasser 8](#_Toc20223771)

[3.6 Enhetstester med JUnit och testtäckning med EclEmma 9](#_Toc20223772)

[3.7 Designprinciper och riktlinjer 9](#_Toc20223773)

[3.8 Testkörning av WeekAlarmClock 10](#_Toc20223774)

[4 Diskussion 11](#_Toc20223775)

[4.1 Gränssnittet Comparable i gränssnittet TimeType 11](#_Toc20223776)

1. Inledning
   1. Syfte och riktlinjer

I denna laboration skulle ett grafiskt användargränssnitt (GUI)) skapas som använder och exponerar funktionaliteten hos klassen WeekAlarmClock och alla dess beroenden som skapades i laboration 1 och 2. Syftet med denna laboration var att studenten skulle:

* Lära sig att bygga ett grafiskt gränssnitt med layout och komponenter.
* Lära sig att skapa egendefinierad grafik genom att definiera om metoden paintComponent() i klassen JPanel.
* Lära sig att hantera händelser med lyssnare och att behärska………

Generella riktlinjer och principer som skulle tas i beaktande och i största möjliga mån följas (utöver de som specificerats i laboration 1 och 2) var:

* Separera presentation (view) från modellen.
* Håll GUI-kod så fri från modellens logik som möjligt.
* Håll GUI-kod fri från icke-GUI-kod genom att delegera till controller(s)
* Så få beroenden som möjligt. Delegera när det är möjligt.
* Tänk på och utgå från ett användarperspektiv – vad är lämpligaste användargränssnitt?
  1. Specifikation av alarmklocka

Laborationen skulle resultera i en klass som beskriver en alarmklocka. Denna skulle använda sig av den kod som skrevs i laboration 1.

Alarmklockan skulle kunna ha flera alarm som ställs och aktiveras vid olika tidpunkter.

Koden skulle testas med JUnit-tester (enhetstester).

1. Metod och utförande
   1. Utökat beteende hos Counter-klasserna

I laboration 1 skapades gränssnitt (interfaces) och klasser som specificerade och implementerade räknare. Dessa behövde utökas så att räknaren kunde ställas till en specifik tid.

Interfacet SettableCounterType skapades. Detta fick ärva specifikationen i CounterType genom att utöka (extend) interfacet. Metoden setCount(int value) lades till.

En ”ställbar” version av klassen AbstractCounter skapades. Denna döptes till SettableCounter. Denna klass utökar AbstractCounter och implementerar interfacet SettableCounterType. Således måste klassen implementera metoden setCount(int value) samt alla metoder som definieras i AbstractCounter.

Då klassen LinkedAbstractCounter hade skapats i laboration 1 skapades en ställbar, utökad version av den. Denna döptes till SettableLinkedCounter. Denna klass utökar AbstractLinkedCounter och implementerar interfacet SettableCounterType.

Klasserna Counter24 och Counter60 ändrades så att de nu istället utökar SettableLinkedCounter. De är alltså räknare som har en inre räknare och som kan ställas. Klassen Counter7, som skulle användas för att räkna veckodagar, skapades och fick utöka klassen SettableCounter.

* 1. Gränssnittet TimeType och klassen Time

Gränssnittet TimeType och klassen Time skapades för att representera tid i alarmklockan som skulle skapas. All kod för dessa var given. Klassen Time utökades något med felhantering. Vid en tidpunkt i utvecklingen fick klassen Time implementera interfacet Comparable<Time> för att möjliggöra jämförelser av instanser av klassen. Senare i utvecklingen gjordes upptäckten att detta inte var nödvändigt, men funktionaliteten fick kvarstå.

* 1. Gränssnittet AlarmType och klassen Alarm

Metodsignaturerna i gränssnittet AlarmType var given. Gränssnittet specificerar funktionaliteten hos ett alarm. En liten del av koden för klassen Alarm var given och resterande kod implementerades. Även här lades felhantering till.

* 1. Klassen AlarmManager

Klassen AlarmManager är den del av alarmklockan som är ansvarig för att lägga till, ta bort och handha alarm. Den har också som uppgift att kontrollera om det är dags att aktivera något alarm när klockans tid avancerar. Koden för klassen var given.

* 1. Gränssnittet AlarmActionType och klassen PrintAlarmAction

Gränssnittet AlarmActionType innehåller endast två metoder, alarmActivated() och alarmDeactivated(). Dessa metoder tillhandahåller de aktioner, de *funktioner* som ska utföras då ett alarm aktiveras. En implementation av gränssnittet, klassen PrintAlarmAction, skrevs.

* 1. Gränssnitten ClockType och AlarmClockType, klasserna WeekClock och WeekAlarmClock

Gränssnittet AlarmClockType var givet. I detta specificerades sju metoder som beskriver funktionaliteten hos en alarmklocka enligt nedan:

public interface AlarmClockType { public void tickTock(); public void setTime(TimeType time); public void addAlarm(AlarmType larm); public void removeAlarm(AlarmType alarm); public Collection<AlarmType> getAlarms(); public TimeType getTime(); public String toString(); }

Metoderna tickTock, setTime(), getTime() och toString() bröts ut till ett mer generellt interface kallat ClockType och tre metoder lades till: startClock(), stopClock() och resetClock(). ClockType skapades för att erbjuda en mer generell implementation av en klocka. Gränssnittet AlarmClockType fick sedan utöka ClockType och specificera metoderna addAlarm(), removeAlarm() och getAlarms().

* 1. UML-klassdiagram

Ett UML-klassdiagram skapades för varje paket/modul med ritverktyget draw.io.

* 1. Enhetstester med JUnit och testtäckning med EclEmma

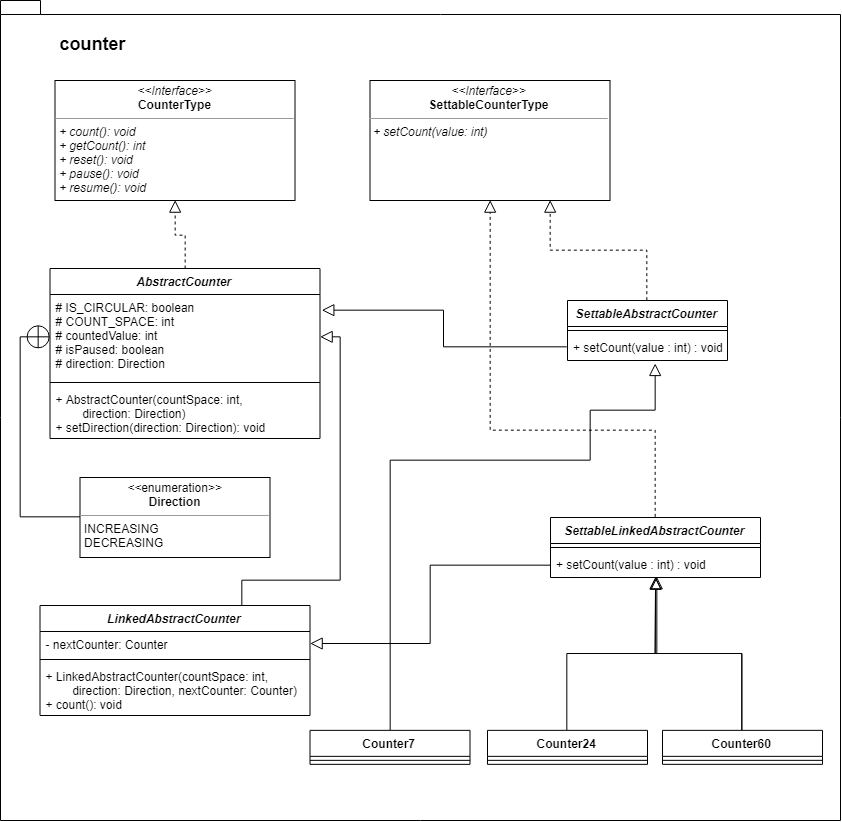
Enhetstester skrevs för varje klass. Enhetstesterna skrevs för att största möjliga mån täcka in gränsvärden och för att säkerställa att felhanteringen fungerar som den ska.

Det i Eclipse inbyggda plug-in-programmet EclEmma användes för att avgöra hur stor del av koden som täcks av enhetstesterna.

1. Resultat

Resultaten av laborationen redovisas här under rubriker som motsvarar de fyra paket/moduler som skapats, nämligen counter, time, alarm och clock. Efter en beskrivning av de fyra modulerna följer en mer översiktlig genomgång av laborationens resultat.

* 1. Counter

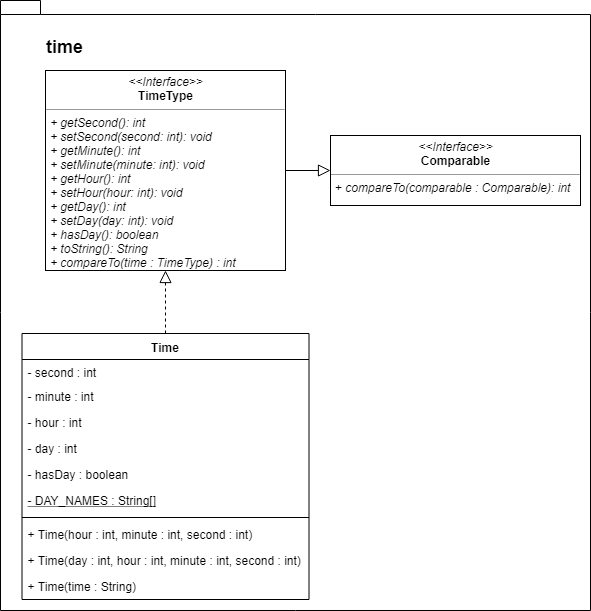
Figur 1 visar alla gränssnitt och klasser i paketet counter.

Figur 1: Klassdiagram för alla gränssnitt och klasser i paketet counter.

All funktionalitet som specificeras i de abstrakta datatyperna utökas och implementeras genom arv och implementation av gränssnitt.

* 1. Time

Figur 2 visar gränssnittet TimeType och klassen Time som utgör time-paketet.



Figur 2: Klassdiagram för paketet time.

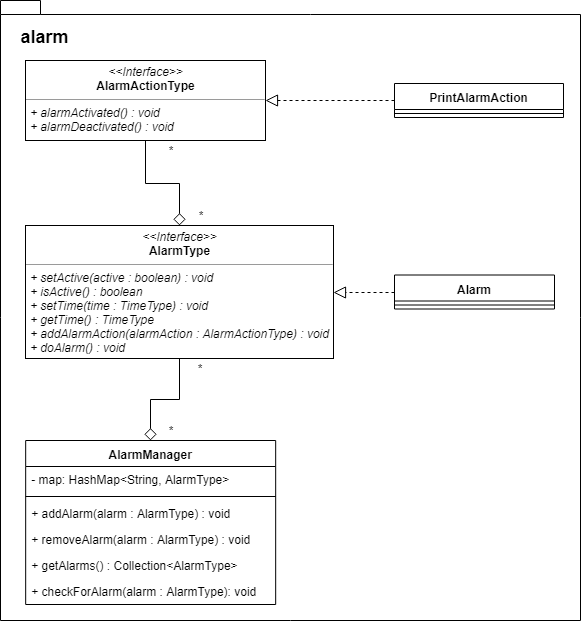
Utöver den givna koden i klass Time lades följande metod till:

private boolean isNegative(int number) {  
 return number < 0;  
}

Denna metod anropas i metoderna setDay(), setHour(), setMinute() och setSecond() för att säkerställa att ett positivt värde har givits. Om inte så kastas ett IllegalArgumentException.

* 1. Alarm

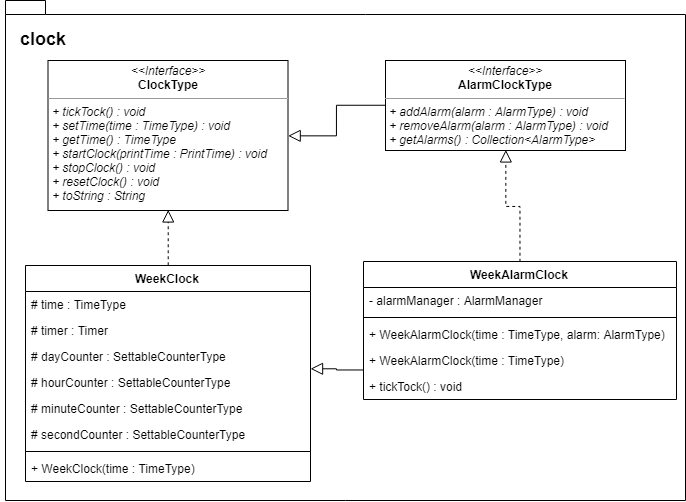
Figur 3 visar klassdiagrammet för paketet alarm . Gränssnittet AlarmType är beroende av gränssnittet TimeType i paketet time. Detta visas inte i klassdiagrammet för att hålla det överskådligt. För en beskrivning av beroenden mellan klasser i olika paket, se *3.5 - Beroenden mellan klasser*.

I paketet alarm ser vi två exempel av aggregat. AlarmManager är ett aggregat av AlarmType-objekt, som i sin tur är aggregat av AlarmActionType-objekt. Objekten är inte hårt kopplade som i ett kompositaggregat då ett enskilt objekt kan vara kopplade till flera instanser av aggregatet.

Figur 3: Klassdiagram för paketet alarm.

* 1. Clock

Figur 4 visar klassdiagrammet för paketet clock. Klassen WeekAlarmClock är det viktigaste resultatet i laborationen. Den erbjuder funktionaliteten som skulle uppnås. WeekAlarmClock har beroenden av klasser utanför paketet clock, nämligen AlarmManager i paketet alarm och SettableCounterType i paketet counter. WeekAlarmClock delegerar hanteringen av Alarm-objekt till AlarmManager.

För en beskrivning av beroenden mellan klasser i olika paket, se *3.5 - Beroenden mellan klasser*.

Figur 4: Klassdiagram för paketet clock.

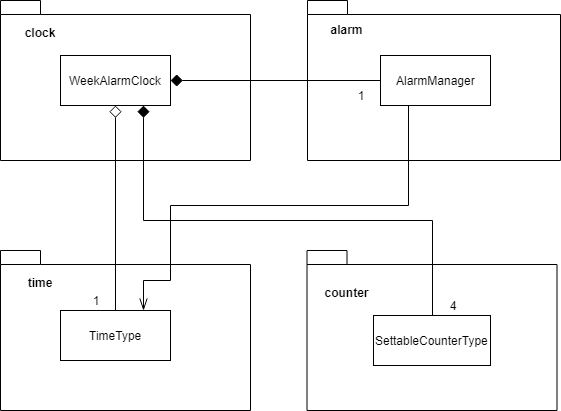
Metoden startClock() är definierad enligt följande:

@Override  
 public void startClock(PrintTime printTime) {  
 if (timer == null) {  
 timer = new Timer();  
 timer.scheduleAtFixedRate(new TimerTask() {  
 public void run() {  
 tickTock();  
 if (printTime == PrintTime.YES) {  
 System.out.println(getTime().toString());  
 }  
 }  
 }, 0, 1000);  
 }  
 }

Metoden skapar ett nytt Timer-objekt och ett nytt TimerTask-objekt. Koden inuti TimerTask-objektet exekveras med en sekunds mellanrum genom metodanropet timer.scheduleAtFixedRate(). startClock() tar emot ett Enum-objekt av typen PrintTime, som endast kan ha två värde, PrintTime.YES och PrintTime.NO. Om PrintTime.YES har skickats in som parameter till metoden kommer den nuvarande tiden att skrivas ut varje gång koden i TimerTask-objektet exekveras.

Metoden stopClock() avbryter exekveringen av TimerTask-objektet genom metodanropet timer.cancel().

* 1. Beroenden över paketens gränser

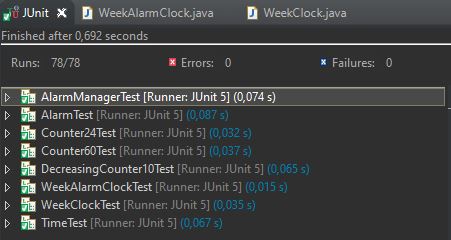
Figur 5 visar en något förenklad bild över de beroenden som förekommer över paketens gränser.

Figur 5: Beroenden över paketens gränser.

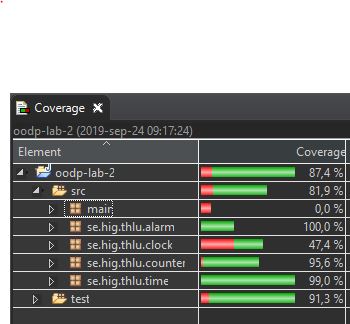
Beroendena mellan WeekAlarmClock och SettableCounterType respektive AlarmManager beskrivs här som komposita aggregat. Detta för att de instanser av dessa klasser som WeekAlarmClock är beroende av *skapas* inuti WeekAlarmClock. Det finns inget sätt för andra klasser att komma åt dessa instanser.

Beroendet av TimeType-objekt är istället ett lösare kopplat aggregat då de instanser av TimeType-klassen som WeekAlarmClock använder sig av kan vara tillgängliga utanför klassen.

* 1. Enhetstester med JUnit och testtäckning med EclEmma

Enhetstester skrevs för alla klasser. Totalt skrevs 78 enhetstester. Figur 6 visar att alla enhetstester exekveras utan fel.

Figur 6: Resultatet av att köra alla JUnit-tester i Eclipse.

EclEmma användes för att mäta hur stor del av källkoden som täcks av enhetstesterna, se Figur 7. EclEmma visar att 81,9 % av källkoden testas i enhetstesterna. Paketet clock har sämst testtäckning. Detta beror på att metoderna startClock(), stopClock() och resetClock() inte testats.

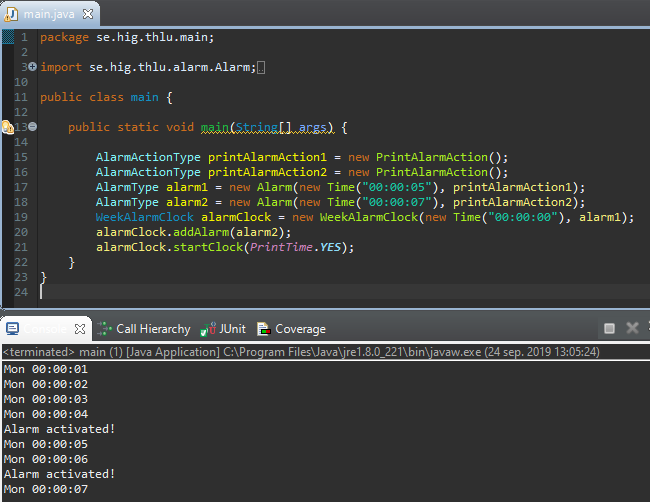
Figur 7: Resultatet av exekvering av EclEmma.

* 1. Designprinciper och riktlinjer

Genom dessa designval har de riktlinjer och designprinciper som specificerats följts:

* Program to an interface: Gränssnitt/abstraktioner har skapats för alla abstrakta datatyper där det har ansetts fördelaktigt.
* Alla objektreferenser är av typen med allra högst abstraktionsnivå (gränssnitt/abstrakt klass).
* Information hiding har åstadkommits genom att alla instansvariabler har definierats med de mest restriktiva tillgångsmodifierarna möjliga.
* Koden är robust då inga null-referenser tillåts och enhetstester finns för en stor del av koden och många use-cases.
* Funktionalitet som utökar funktionaliteten i den givna koden har definierats i ett nytt Interface/en ny klass i enlighet med Open-Closed Principle.
* Varje Interface/klass har ett begränsat, sammanhängande ansvarsområde i enlighet med Single Responsibility Principle.
* Delegering har använts där det är passande. Ett exempel på detta är i klassen WeekAlarmClock där hanteringen av AlarmType-objekt delegeras till AlarmManager.
  1. Testkörning av WeekAlarmClock

I paketet main finns klassen Main som innehåller systemets main-metod. Figur 3 visar resultatet av exekvering av main-metoden.



Figur 8: Resultatet av exekvering av main-metoden.

1. Diskussion
   1. Gränssnittet Comparable i gränssnittet TimeType

Min första lösning på funktionaliteten för att avgöra när ett alarm skulle aktiveras var att låta gränssnittet TimeType ärva gränssnittet Comparable<Time>. Denna kunde sedan användas i klassen WeekAlarmClock enligt följande:

private TimeType time; // instansvariabel för tid  
private Collection<AlarmType> alarm; /\* instansvariabel innehållande alla AlarmType-objekt \*/  
  
@Override  
public void tickTock() {

super.tickTock();  
 alarms  
 .stream()  
 .filter(alarm -> alarm.isActive() && alarm.getTime().compareTo(time) == 0)  
 .forEach(alarm -> alarm.doAlarm());  
}

Denna lösning förutsatte att WeekAlarmClock hade ett Collection-objekt innehållande alla AlarmType-objekt. När jag sedan läste implementationen av AlarmManager mer noggrant insåg jag att den kunde användas istället och att den innehöll en mer elegant lösning på funktionaliteten, nämligen att, för varje gång klockan tickar, söka efter den nuvarande tiden i en HashMap innehållande alla AlarmType-objekt. Lösningen i AlarmManager har också en amorterad tidskomplexitet av *O*(1) jämfört med den iterativa lösningen som utförs i *O*(*n*) tid, alltså betydligt mer effektiv då många AlarmType-objekt finns i samlingen.

Jag ändrade lösningen så att WeekAlarmClock delegerar hanteringen av AlarmType-objekt till AlarmManager, men lät fortfarande TimeType utöka Comparable. Detta för att det framstår som en rimlig funktionalitet hos TimeType-klassen.