Omtentamen för TDA551
Objekt-orienterad Programmering och Design

Dag: 2017-04-10, *Tid:* 8.30–12.30

Ansvarig:	Alex Gerdes
Examinator:	Niklas Broberg
Förfrågningar:	Alex Gerdes (alexg@chalmers.se, 031 772 6154)
Resultat:	Erhålls via Ladok
Betygsgränser:	3:a 24 poäng 4:a 36 poäng 5:a 48 poäng max 60 poäng
Siffror inom parentes:	Anger maximal poäng på uppgiften
Granskning:	Tentamen kan granskas på studieexpeditionen. Vid eventuella åsikter om rättningen eposta och ange noggrant vad du anser är fel så återkommer vi.
Hjälpmedel:	Inga (en UML cheat sheet är bifogat)
Var vänlig och:	Skriv tydligt och disponera papperet på lämpligt sätt. Börja varje uppgift på nytt blad. Skriv ej på baksidan av papperet.
Observera:	För full poäng krävs ett läsbart, begripligt och *heltäckande* svar. Svar kan ges på antingen svenska eller engelska (eller blandat). Skriv tydligt om du gör antaganden som inte står i uppgiftsbeskrivningen. Tips: Läs igenom alla frågor innan du börjar svara på uppgifter. Notera oklarheter så du kan fråga när någon kommer förbi (ca 9:30). De flesta uppgifter är designade för att ni ska ge korta svar. Få svar, om något, bör kräva mer än ett A4-ark.

Lycka till!

Börja med att betrakta och bekanta dig med en bifogade koden för "TowerDefence" (sammanlagt 10 klasser och interfaces). De flesta av uppgifterna baseras på olika delar av denna kodbas.

Uppgift 1

(4 poäng)

Redogör för följande tekniska termer, med en eller ett par meningar per svar:

a) Konstruktor (constructor)

b) Subtyp (subtype)

c) Generisk typ (generic type)

d) Gränssnitt (interface)

Uppgift 2

(6 poäng)

Med utgångspunkt i koden för TowerDefence, ange för varje deluppgift nedan om kodstycket är korrekt, ger ett statiskt fel, eller ger ett runtime-fel. Om det är korrekt, ange för varje metodanrop i kodstycket vilken eller vilka versioner av metoden `getDescription` (klass och signatur) som kommer att exekveras. Om det är inkorrekt, förklara varför.

a)
```
Tower orthanc = new Tower(new Position(1,1));
System.out.println(orthanc.getDescription());
```

b)
```
Tower orthanc = new Piece();
System.out.println(orthanc.getDescription());
```

c)
```
Piece orthanc = new Tower(new Position(1, 1));
System.out.println(orthanc.getDescription());
```

d)
```
Tower baraddur = new PowerTower(new Position(1, 1), 666);
System.out.println(baraddur.getDescription());
```

Uppgift 3

Antag att vi har följande variabler:

```
List<? extends Piece> listExtendsPiece = ...;
List<Piece> listPiece = ...;
List<Tower> listTower = ...;
List<? super Tower> listSuperTower = ...;
```

a) Ange för varje deluppgift nedan om kodstycket är korrekt, ger ett statiskt typfel, eller ger ett runtime-fel. Om det ger ett fel, förklara varför. Antag att alla listor innehåller minst ett element. (4 poäng)

i.
```
listExtendsPiece = listPiece;
Piece piece      = listExtendsPiece.get(0);
```

ii.
```
listExtendsPiece = listTower;
Tower orthanc    = listExtendsPiece.get(0);
```

iii.
```
listPiece        = listTower;
Piece piece      = listPiece.get(0);
```

iv.
```
listSuperTower   = listPiece;
Piece piece      = listSuperTower.get(0);
```

b) Antag att vi vill ha en simulerings-motor för våra värld, och att denna ska hålla reda på alla föremål (pieces) i världen, oavsett om de är torn eller andra sortes föremål. Vilken av de fyra ovanstående listorna bör vi då välja för att lagra dessa föremål? (2 poäng)

c) Typen `List<? super T>` är contra-variant i typen `T`. Förklara vad det innebär, och visa med ett exempel vad det får för konsekvenser. (2 poäng)

Uppgift 4

a) Förklara kort vad en 'race condition' är och ge ett exempel av en situation där en sådan 'condition' kan uppstå. (2 poäng)

b) Förklara vad modifieraren `synchronized` innebär för en method och varför man måste vara restriktiv med denna modifieraren. (2 poäng)

Uppgift 5

Betrakta klassen `World`:

a) Beskriv tre olika sätt som object av denna klass kan förändras genom att dess attribut förändras, trots att dessa är privata. (3 poäng)

b) Hur kan man skydda sig mot oväntade förändringar? (3 poäng)

Uppgift 6

Redogör för följande design-principer, och ge ett exempel för var och en av dem på kod som följer respektive inte följer principen.

a) Liskov Substitution Principle (3 poäng)

b) Command-Query Separation Principle (3 poäng)

Uppgift 7

Redogör för följande design patterns (design-mönster), inklusive syfte och effekt. Visa exempel som UML-diagram:

a) Strategy Pattern (4 poäng)

b) Decorator Pattern (4 poäng)

Uppgift 8 (8 poäng)

Betrakta kodbasen för TowerDefence. Notera alla design patterns som använts. För varje pattern, peka konkret i koden på de klasser, interfaces och metoder som bidrar. Visa gärna med hjälp av UML-diagram för relevanta delar.

Uppgift 9 (6 poäng)

Klassen `World` har en getter för instansvariabeln `canvas`. Vi kan få alias problem med detta eftersom `canvas` är en referensvariabel. Applicera *Iterator* designmönstret för att förbättra koden. Visa resultatet som UML-diagram samt koden för en klass som implementerar `Iterator<T>` gränssnittet. *Tips:* följande metoderna ska vara med i implementationen:

- `boolean hasNext()`
- `T next()`

Uppgift 10

Betrakta kodbasen för TowerDefence. För följande design-principer, hitta minst ett fall där koden bryter mot principen:

a) Dependency Inversion Principle (2 poäng)

b) Law of Demeter (“Don’t talk to stranges”) (2 poäng)

TowerDefence kodbas

TowerDefence

```
1  import java.util.*;
2
3  public class TowerDefence {
4    enum State {WON, LOST, RUNNING}
5    private State state;
6    private World world;
7
8    private ArrayList<StateObserver> observers = new ArrayList<>();
9
10   TowerDefence() {
11     world = new World();
12     state = State.RUNNING;
13   }
14
15   public void tick() {
16     fireTowers();
17     world.getMonster().move(world);
18     updateState();
19     notifyStateObservers();
20   }
21
22   private void fireTowers() {
23     for (Tower tower : world.getTowers()) {
24       int damage = tower.shoot(world.getMonster());
25       world.getMonster().shot(damage);
26     }
27   }
28
29   private void updateState() {
30     if (world.getMonster().getHealth() <= 0) {
31       state = State.LOST;
32     } else if (world.atEndOfWorld(world.getMonster())) {
33       state = State.WON;
34     }
35   }
36
37   private void notifyStateObservers() {
38     for (StateObserver observer : observers)
39       observer.update(state);
40   }
41
42   public void addStateObserver(StateObserver observer) {
43     observers.add(observer);
44   }
45
46   public World getWorld() {
47     return world;
48   }
49 }
```

World

```
import java.util.*;

public class World {
  enum Tile {ROAD, TERR}
  private Monster monster;
  private ArrayList<Tower> towers;
  private Tile[][] canvas;
  private int rows;
  private int cols;
  private Position endOfWorld;

  public World() {
    monster = new Monster(100);
    towers = new ArrayList<>();
    // make world
  }

  public boolean atEndOfWorld(Monster monster) {
    return endOfWorld.equals(monster.getPosition());
  }

  public boolean onRoad(Position position) {
    int x = position.getX();
    int y = position.getY();
    return x < rows && x >= 0 &&
           y < cols && y >= 0 &&
           canvas[x][y] == Tile.ROAD;
  }

  public int getRows() { return rows; }

  public int getCols() { return cols; }

  public Tile[][] getCanvas() { return canvas; }

  public Monster getMonster() { return monster; }

  public ArrayList<Tower> getTowers() { return towers; }

  public void addTower(Tower tower) { towers.add(tower); }
}
```

TowerDefenceFactory

```
public class TowerDefenceFactory {
  public static TowerDefence buildGame() {
    return new TowerDefence();
  }
}
```

Tower

```java
import java.util.Random;

public class Tower extends Piece {
  private Random random;
  protected int range;
  protected int power;

  Tower(Position position) {
    this.position = position;
    range       = 2;
    power       = 22;
    random      = new Random();
  }

  public int shoot(Piece piece) {
    int damage = power / Position.distance(position, piece.getPosition());
    return inRange(piece.getPosition()) ? damage : 0;
  }

  public boolean inRange(Position position) {
    return Position.distance(this.position, position) <= range;
  }

  @Override
  public String getDescription() { return "Tower : " + power; }
}
```

PowerTower

```java
public class PowerTower extends Tower {
  public PowerTower(Position position, int extraPower) {
    super(position);
    power = super.power + extraPower;
  }

  @Override
  public String getDescription() {
    return "Power" + super.getDescription();
  }
}
```

StateObserver

```java
public interface StateObserver {
  void update(TowerDefence.State state);
}
```

Monster

```
public class Monster extends Piece {
  private int health;
  private Direction direction;

  Monster(int health) {
    direction = Direction.UP;
    this.health = health;
  }

  public void move(World world) {
    turn(1);
    Position next = new Position(position);
    next.move(direction);
    if (world.onRoad(next))
      position = next;
    else {
      turn(2);
      move(world);
    }
  }

  private void turn(int n) {
    if (n > 0) {
      switch (direction) {
        case LEFT:  direction = Direction.DOWN;  break;
        case RIGHT: direction = Direction.UP;    break;
        case UP:    direction = Direction.LEFT;  break;
        case DOWN:  direction = Direction.RIGHT; break;
        default: break;
      }
      turn(n - 1);
    }
  }

  public void shot(int damage) {
    health -= damage;
  }

  public int getHealth() { return health; }

  @Override
  public String getDescription() { return "Orc"; }
}
```

Direction

```
public enum Direction {
  UP, DOWN, RIGHT, LEFT
}
```

Position

```
public class Position {
  private int x;
  private int y;

  Position(int x, int y) {
    this.x = x;
    this.y = y;
  }

  Position(Position p) {
    x = p.getX();
    y = p.getY();
  }

  public void move(Direction direction) {
    switch (direction) {
      case UP:    x--; break;
      case DOWN:  x++; break;
      case RIGHT: y++; break;
      case LEFT:  y--; break;
    }
  }

  public int getX() { return x; }

  public int getY() { return y; }

  public static int distance(Position p, Position q) {
    return Math.abs(p.getX() - q.getX()) + Math.abs(p.getY() - q.getY());
  }
}
```

Piece

```
public abstract class Piece {
  protected Position position;

  public Position getPosition() { return position; }

  void setPosition(Position position) { this.position = position; }

  public abstract String getDescription();

  @Override
  public String toString() {
    return "This piece is a " + getDescription();
  }
}
```

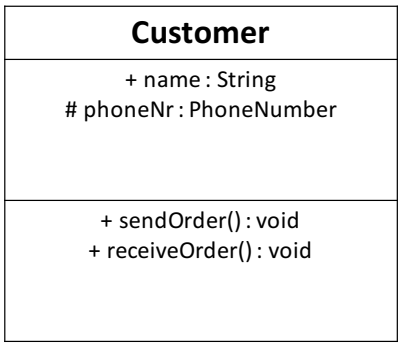
UML cheat sheet

Klassdiagram: Classes

- En class består av:
  - Dess namn
  - Dess variabler
  - Dess metoder (och konstruktorer)

+ = public
- = private
= protected
~ = none (i.e. package)

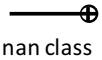
italics = abstract
underline = static
<<interface>>



Klassdiagram: Relationer

- Fyra grundläggande typer av relationer mellan classes och interface:
  - Association (Has-A) 
    - En class har attribut (fields) som håller objekt av en annan class (interface).
  - Beroende (usage dependency) 
    - En class (interface) använder eller skapar objekt av en annan class (interface).
  - Generalisering (Is-A) 
    - En class (interface) är en direkt subclass (subinterface) till en annan class (interface).
  - Realisering (implements) 
    - En class implementerar ett interface.

Klassdiagram: Relationer specialfall

- Specialfall av association:
  - Aggregation (parts-of-a-whole) 
    - En class består av en eller flera delar, som representeras av en annan class.
  - Komposition (subpart-of) 
    - En starkare form av aggregation, där ”delen” bara får förekomma i en ”helhet”, och existerar endast så länge dess helhet existerar
  - Inkapsling (encapsulation) 
    - En class existerar inuti en annan class