Skriven 211014 av grupp HEJAP: Hanna Boquist, Amanda Cyrén,
Johan Selin, Emma Stålberg, Philip Winsnes

Peer-Review av Handymen, Alchemy Defense

Generellt sett är det en väldigt bra kodstruktur och väldokumenterat. MVC följs mycket väl och koden är lättförståelig. SOLID-principerna efterföljs bra i stort med undantag för "Open-Closed Principle". Se kommentarer nedan för förslag på förbättringar.

- Följer design och implementation designprinciper?
    - Håller projektet en konsekvent kodstruktur? **Ja.**
    - Är koden återanvändbar? **Ja till stor del. Problem: se punkt 1, 2, 6, 8, 10 & 13.**
    - Är den lätt att underhålla? **Ja till stor del. Problem: se punkt 1, 2, 5, 6, 7, 10 & 13.**
    - Kan man enkelt lägga till och ta bort funktionalitet? **Ja till stor del. Problem: se punkt 1, 2, 5, 6, 7, 9, 10 & 13.**
    - Används designmönster? **Ja. Singleton, Module, Observer, MVC, (Marker Interface Pattern, används inkorrekt, se punkt 12).**
- Är koden dokumenterad? **Mestadels, saknas javadoc i t.ex. *ConcreteBoard.java*.**
- Används bra och tydliga namn? **Ja.**
- Är designen modulär? Finns det några onödiga beroenden? **Ja, designen är modulär men det finns onödiga beroenden. Problem: se punkt 1, 6, 7, 8 & 13.**
- Använder koden korrekta abstraktioner? **Ja, i nästan samtliga fall. Problem: se punkt 1 & 9.**
- Är koden bra testad? **Nej, täckningsgraden av testerna kan förbättras, se punkt 4.**
- Bidrar koden till en säker användning gällande information? Finns problem gällande prestanda? **Inget som är relevant.**
- Är koden enkel att förstå? Har den en MVC-struktur och är modellen isolerad från de andra delarna? **Koden är lättförståelig och följer MVC-strukturen nästan till fullo, men problem finns, se t.ex. punkt 8.**
- Kan designen eller koden bli förbättrad? Finns det bättre lösningar? **I stort sätt är programmet välstrukturerat och genomtänkt, men vissa saker kan tänkas över och förbättras. Se nedanstående punktlista.**

---

1. Klassen *Tower* ger möjlighet för bra kod-återanvändning men ingen av dess subklasser implementerar eget beteende vilket innebär att de är onödiga. I nuläget kan alla subklasser skapas som varianter av superklassen Tower (förutsatt att den inte vore en abstrakt klass). Det kanske är tanken att det ska implementeras senare? Däremot kan kanske ett *Tower* bättre konstrueras med hjälp av komposition och ge den egenskaper via "dependency injection". Underlättar vid uppgradering, ändring av attackmönster, återanvändning. Undviker klassexplosioner.
2. *TowerPrices* specificerar köp- och säljpriser för alla tornen. Gör dock så att ett pris för ett torn är frånkopplat från själva tornet. Finns för och nackdelar men om man skulle vilja uppgradera ett torn (finns inte nu verkar det som) så är prisen redan satta och det blir knepigt att hålla koll på vilket pris varje torn ska ha.

3. *TowerPrices* kan möjligtvis bara hålla sina "private static final ints" som publika istället och låta komponenter använda dem genom klassen direkt. Ingen metod för det behövs utan *TowerPrices* agerar endast som en dataklass. Ett pris skulle då nås genom att skriva `TowerPrices.RED_TOWER_BUY_PRICE` för ett rött torn.
4. Inte tillräcklig täckningsgrad på tester. Programmet uppnår endast 56% täckningsgrad i testerna av modellen.
5. Designmönstret Singleton används i klassen *Player*. En spelare kommer alltid skapas med startvärdena 100 guld och 100 hp och det blir krångligt om man skulle vilja lägga till någon slags förmån eller köra i ett specialläge där spelaren exempelvis vill skapas med 10000 hp och 1 miljon guld. Går visserligen att använda metoder för att öka guld och hp efter skapelse men blir knepigt vid flerspelarläge. Känt för singleton är att JUnit-tester kan bli svåra att utföra då mönstret introducerar starka couplings och oönskade sidoeffekter. Mer om detta går att läsa om här.
6. Klassen *Tile* har två instanser *Tower* och *Foe*, men en tile kan endast ockuperas av endera, alltså har den en instans som klassen inte "behöver" vid ett givet tillfälle. Både *Tower* och *Foe* implementerar *BoardObject* och det borde vara möjligt att låta *Tile* ha ett *BoardObject* istället. Underlättar även vid implementation av fler typer, säg någon slags blockad.
7. *Tile* använder "reaching through" men borde kanske snarare ha en metod "getTowerDamage" istället för att metoden *damageFoe()* kallar direkt på metoden vilket skapar ett beroende direkt från *ConcreteBoard* (klassen *damageFoe()* finns i) till *Tower*.

```
private void damageFoe(Tile cellTower, Tile cellFoe) {
  int damage = cellTower.getTower().getDamage();
  cellFoe.getFoe().takeDamage(damage);
}
```

8. "Interface" *BoardObject* som många klasser implementerar har krav på en metod som hämtar klassens sökväg till dess bild vilket bryter mot MVC mönstret. Modellen får en direkt koppling till vyn.

```
public interface BoardObject {
  String getImageFilePath();

  BoardObjectType getBoardObjectType();
}
```

   Dessutom ska modellen vara överförbar till vilket GUI-ramverk som helst och där behövs eventuellt ingen sökväg till den klassens bild. Dessutom är dessa sökvägar "private final Strings" vilket innebär att om man skulle vilja ändra hur något ser ut när programmet körs så blir det väldiga problem. Exempelvis om man skulle vilja flasha rött när en skurk blir skadad eller om ett torn uppgraderas.
9. *Pathfinder* skulle kunna abstraheras. *ConcreteBoard* har just nu en *Pathfinder* som kan utföra "breadth-first search" (BFS) som endast har två publika metoder: *generateNewPath* och *blocksPath*. Låt istället *Pathfinder* vara en "interface" som definierar dessa två metoder och låt *ConcreteBoard* ha en attribut av den. Då kan

konkreta klasser av typen Pathfinder konstrueras som exempelvis BFSPathfinder, DFSPathfinder (depth-first), AstarPathfinder (A*) osv. Kan ses som överflödigt men skulle kanske bli nödvändigt om man skapar ett större tilegrid / mindre rutor. Liknande funktionalitet bör kunna uppnås med Strategy Pattern.

10. I klasserna *TowerFactory* och *TowerPrices* används många "switch"-satser vilket inte är optimalt då det bryter mot designmönstret "Open Closed Principle". Ska det exempelvis utökas med fler torn blir det svårt eftersom de nya tornen måste läggas in på alla ställen. Glöms ett ställe märks inte detta förrän programmet redan körs och det blir ett exekveringsfel.
11. Inte särskilt användarvänligt t.ex. när det kommer till att högerklicka för att ta bort objekt. Man skulle isåfall vilja få den infon på något sätt. Svårt att hänga med i spelet p.g.a. långsam uppdateringsfrekvens.
12. "Interface" *Foe* beskrivs använda sig av designmönstret "Marker Interface Pattern" men implementerar inte detta korrekt eftersom *Foe* definierar metoder för de som implementerar interfacet. För att få önskat beteende som "Marker Interface Pattern" ger kan istället "annotations" användas i Java men så som *Foe* är i nuläget behöver den ingetdera utan kan helt enkelt vara ett "vanligt" "interface".
13. Delar av programmet skulle nyttjas av "Dependency Injection"-designmönstret (DI). Exempelvis i klassen *ConcreteBoard* som just nu har hårdkodade värden för dess *TileGird*-instans och skapar objekt av andra klasser i sin konstruktor. Vill man i nuläget ändra storleken på brädet så måste man ändra på klassen i sig. Genom DI eller flera konstruktorer så görs kopplingen mellan *ConcreteBoard* och de andra klasserna svagare samt det får en mer flexibel och återanvändbar klass.