Draft 1

INLEDNING

ANALYS

**UML/Kommentarer**

**[uml och kommentarer kommer ligga här]**

**Designprinciper**

* SOLID-principer

**Single Responsibility Principle (SRP):**

Koden tar allmänt ingen avsiktlig hänsyn till SRP vilket leder till många ”uppblåsta” klasser och metoder med mycket ansvar. Detta leder i sin tur till att de flesta klasserna inte är ordentlig avgränsade och hanterar mycket utanför sitt konceptuella ansvar.

Vi kan se exempel på detta då alla klasser som är ansvariga för ett visuellt objekt har en render()-metod. Detta i sig går emot många andra Patterns som annars hade varit applicerbara (MVC, tar upp senare) men dessutom ger detta klasserna low cohesion, vilket är något uppnår med SRP, då klasserna hanterar både grafiska aspekter och funktionalitet som är en mycket stark avgränsning inom Objekt-Orienterad Programmering.

I klassen TowerComponent vars syfte är att agerar som en typ av main-klass som får spelet att köras och uppdateras hittar vi ytterligare dåliga avgränsningar. Både i själva klassen som tillsammans med många andra ärver från klassen Entity och därmed render()-metoden för grafisk visualisering men dessutom ser vi stora brott mot SRP i dess långa och komplexa metoder. Ponera metoden renderGame(). Metoden är 200 rader lång och innehåller flera for-loopar och nästlade if-satser. Metoden i sig håller reda på tiden i spelet, styr ”kamera-vinkeln” för skärmen, renderar alla spel-objekt, kontrollerar win-conditions, skapar det visuella gränssnittet för användaren (kontrollpanelen) och mycket mer. Detta visar starkt på att SRP inte hållits i åtanke vid skrivandet av metoden. Detta reflekteras tydligt, inte bara i metoden i fråga, utan i övriga delar av programmet där, likt denna metod, funktionell nedbrytning för högre cohesion hade kunnat applicerats.

Övriga avsteg från principen kan finnas i bland annat HousType-klassen, där HouseType representerar en specifik variant av ett hus samtidigt som den skapar instanser av sig själv via globala variabler. Andra exempel på avsteg kan finnas i de klasser vars handlingar skall spela upp ljud vilket då görs direkt i metoden via klasserna Sound och Sounds, t.ex. Peon.Die(), Monster.Die(), House.tick(), Job.gotoAndConvert() m.fl..

Dessa är endast några av de flertal avstegen från principen vilket leder till att all klasserna får högre beroenden/coupling mellan varandra (syns t.ex. i att många klasser hanterar ljud själva) och lägre sammanhållning/cohesion (syns i t.ex. att alla Entity subklasser hanterar både logik och rendering).

**Open Closed Principle (OCP):**

OCP är kärnan i en objektorienterad modell och genomsyrar därav strukturen av ett modulärt program. Tyvärr kan man inte se några större tecken på att OCP legat som grund vid skrivningen av programmet vars struktur använder mycket ”just-do-it” metodik.

Skulle programmet utökas hade stora refaktoriseringar behövts eftersom mycket logik är hårdkodad och inga modulära design Patterns är implementerade. Som tydligt exempel på denna brist på ”extensibility” är det hårdkodade grafiska interaktions-gränssnittet där knapparna renderas på ett koordinat-intervall och sedan logik för knapparna ”klistrats” där på med samma koordinater utan att vara kopplade till de faktiska knapparna. En utökning av knapparna hade därav inte varit möjligt utan att hårdkoda om knapparnas position och sedan den motsvarande logiken.

Liknande modularitets-problem kan hittas i Entity där klassen är alldeles för generell för att representera alla objekt (vilket programmeraren i dagsläget kommer runt genom att t.ex. lämna ärvda metoder tomma och därav ignorera LSP). Problem uppstår då den modularitet som finns i logiken endast renderar hårdkodade objekt av klassen Entity, dvs inget objekt som inte konceptuellt bör ärva från Entity kan renderas utan omskrivning. Finns alltså inget interface motsvarande t.ex. IDrawable. Ingen extensibility för objekt som inte ärver från Entity som dessutom är en konceptuellt ifrågasättbar abstraktion.

Två utmärkta sätt att uppnå modularitet genom OCP är polymorfism och kodåteranvändning.

Polymorfism återfinns i t.ex. TowerComponent klassen som renderar men den är restriktiv till Entity som nämnt ovan. I själva Entity-klassen finns också polymorfism i form av metoder som används och överskrivs i dess subklasser. Den är dock högst ifrågasättbar och ingen ”ren” Polymorfism då superklassen Entity t.ex. lämnar fight(), tick() och render() tomma för att sedan overrideas i dess subklasser. I (den konceptuellt ifrågasättbara) subklassen Rock till Entity har också fight() metoden lämnats tom för att super-klassen Entity är för bred .

Finns några tecken på kodåteranvändning genom abstraktion men dessa överskuggas av det faktum att det finns så mycket duplicerad kod och långa metoder där ”Code Reuse” knappast legat i fokus. Detta kan ses i metoden isFree() i Entity som kontrollerar om en plats är ledig på marken eller inte som används i flera andra klasser för att kontrollera att inga kollisioner skapas. Dock genomsyrar inte denna kodåteranvändnings-tankegång resten av programmet då

Trots att man på vissa ställen kan finna tecken på ett försök till modularitet, så misslyckas dessa försök på grund av dålig implementering.

* Ingen extensibility då programmets logik är till större del hårdkodad, gränssnittet speciellt. Svårt att lägga till något.
* Tappar flexibiliteten på grund av långa metoder som dessutom är alldelles för specifika för att kunna återanvändas. Ingen readability heller.
* Väldigt svårt att utöka pga starka beroenden mellan klasser vilket skapar en stelhet i programmet.
* Finns en fattig variant av polymorfism som kanppast förtjänar att kallas polymorfism.
* Finns några tecken på att man tänkt på återanvändning av kod genom abstraktion men dessa överskuggas av det faktum att det finns så mycket duplicerad kod och långa metoder där code reuse knappast legat i fokus.

**Liskovs Substitution Principle (LSP):**

**Interface Segregation Principle (ISP):**

**Dependency Inversion Principle (DIP):**

* Övriga principer
* Övergripande designkvalitét

**Designmönster**

* [Titlar på alla designmönster vi hittar]

Refaktoriseringsmöjligheter

**Designprinciper**

* [Förbättringmöjlighet, med fördelar&nackdelar]

**Generell kod**

* [Förbättringmöjlighet, med fördelar&nackdelar]

Sammanfattning