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):**

En av de första egenskaperna man noterar när man analyserar koden är den breda användningen av subtyps-indelning[annat ord behövs]. Vad som dock också noteras är att majoriteten av klasserna, med undantag för interna klasser i Job och Sound, är subtyper till klassen Entity.

Då alla metoderna i Entity, som överskuggas av subklasser, är tomma är det svårt att identifiera brott mot LSP. De metoder som innehar funktionalitet ändras aldrig, och de metoder som överskuggas har ingen funktionalitet. Sammanlagt finns det väldigt lite funktionalitet att ta bort, men väldigt mycket funktionalitet att lägga till. Vad man kan anmärka på är att vid överskuggning av metoderna så har de överskuggande metoderna ibland olika parametrar. Detta kan man anse är ett brott mot LSP då man inte kan substituera upp klassen som har andra parametrar. Som ett exempel kan man titta på metoden fight() i klassen Monster(extends Entity), som till skillnad från Entity tar in en Peon som argument, istället för Monster. Väljer man då att byta ut ett objekt av typen Entity mot ett objekt av typen Monster, så kommer fel faktiska parametrar skickas in i objektet, vilket skulle leda till fel.

Trots att det är svårt att finna rena brott mot LSP så kan det här vara relevant att adressera andra opassande designval relaterade till subtypshierarkier. Som tidigare nämnt så är många av metoderna lämnade tomma i superklassen Entity. Det går därav att argumentera att Entity bör vara abstrakt, och att dess metoder också bör vara abstrakta. Det råder inte heller något tvivel om att Entity ur ett rent konceptuellt perspektiv har fler egenskaper än vad en entitet har. Klassen Entity har bland annat metoder för att slåss, och en boolesk variabel som är true om entiteten lever, detta trots att allt från stenar till hus ärver av Entity, och en sten kan varken leva eller slåss. Konceptuellt har en sten och en människa väldigt lite gemensamt, och detta bör reflekteras i en tunnare superklass, eventuellt med fler klasser mellan för att dela in de 8 klasser som ärver från Entity i mindre hierarkier.

De 2 andra arvhierarkierna man kan analysera är Sound och Job: Sound är en abstrakt klass med olika interna subklasser som alla är ljud. Vad klasserna gör olika är hur de implementerar metoden fill(), som är abstrakt i Sound. Ingen funktionalitet tas bort ifrån Sound, utan man finner endast extra funktionalitet hos subtyperna. Detta kan ses som en bra implementation av LSP. Man kan dock inte säga samma sak om Job klassen. Job är en icke abstrakt klass som aldrig instansernas, och på samma sätt som Entity så innehåller den många tomma metoder som överskuggas av subklasser för att ge dessa en faktisk funktion. Vad man dock ser mer tydligt i Job är att funktionalitet tas bort. Betrakta till exempel metoden hasTarget. I superklassen Job så används denna metoden till hitta en peon, och ta reda på om den har ett nuvarande mål att gå till, samt säkerställa att detta målet inte är null, medans den skrivs om i klassen Plant till att hitta en Peon som kan plantera ett träd, utan att faktiskt göra några kontroller på peonen. [Säkerställande av exakt funktionalitet behövs]. I övrigt delar Job-klassen många av de problem som återfinns i Entity-klassen.

**Interface Segregation Principle (ISP):**

I programmet används inga interfaces vilket tydligt visar att man inte har följt ISP vid skapandet av programmet. På grund av detta får programmet hög sammankoppling mellan klasser då klasser tvingas få direkta beroenden av varandra och inte genom olika abstraktionslager vi kan skapa med ISP.

Ett tydligt exempel på detta är faktumet att varje renderbart spelobjekt är en subclass till klassen Entity och ärver således metoden render() (som förvisso lämnas tom) som sedan anropas för alla enteties i huvud-tråden. Detta gör att inga objekt som inte ärver direkt från Entity inte kan renderas.

Men kanske huvudpoängen med ISP är dock att abstrahera starka beroenden genom att skapa och dela upp interfaces och i sin tur bero på dessa istället. Ett exempel där detta hade kunnat implementeras hade varit att skapa interfacet IDrawable med metoden render() och låta huvud-tråden bero på detta interface istället som implementeras på relevanta ställen.

I dagsläget så beror många klasser på metoder de inte implementerar, vilket endast har lösts genom att lämna metoden tom där den inte skall implementeras. I Entity har render() metoden som sagt lämnats tom overrideas sedan i relevanta klasser men även metoderna fight() och tick() har lämnats tomma. Förutom det som sagt konceptuella problemet med att t.ex. klassen Rock ärver en metod tick() så lämnas även denna metoden tom i Rock, dvs Rock beror på en metod den inte implementerar. Nu är visserligen inte Entity ett interface men samma typ av problem som ISP syftar på återfinns i denna arvshierarki.

Ett mera direkt brott mot ISP återfinns dock i klassen TowerComponent där uppdateringen av spelet och logiken ligger. Denna klass implementerar bland annat Java-interfacen MouseListener vilket ger ett diverse metoder som läser interaktioner genom musen. Metoderna mouseEntered() och mouseClicked() är dock ej implementerade i programmet och har därför lämnats tomma. Dock så kan man mena att i just detta fallet så är brottet försvarbart då metoderna kommer från ett importerat bibliotek där metoderna i fråga inte har någon direkt inverkan på programmets funktion och att lämna dom tomma påverkar inte strukturen eller funktionerna.

Trots detta så är valet fortfarande högst ifrågasättbart då man enligt god objekt-orienterad design alltid ska skapa med underhåll och framtidssäkring i åtanke och att lämna tomma metoder som en annan utvecklare inte vet är tomma kan skapa stora problem och mycket förvirring i framtiden. Dessutom fanns det ingen dokumentation kring programmet som understryker dessa tomma metoder så i slutändan hade annan metodik varit lämpligare.

**Dependency Inversion Principle (DIP):**

UML-diagrammet visar tydligt att DIP inte har följts då i princip alla klasser har flera direkta beroenden av andra. Faktumet att inga interfaces har implementerats visar också att inga beroenden abstraherats och således en extremt hög sammankoppling mellan alla klasser existerar. Flera direkta starka dubbel-beroenden existerar mellan centrala klasser (t.ex. Entity ”has a” Island som ”has a” lista med Entity-objekt, Island ”has a” TowerComponent som ”has a” Island) alltså formas både långa och starka beroende kedjor.

Dessa brott mot DIP gör att programmet har både många och starka beroenden, tvärtemot det som vi kan uppnå om vi följer DIP. Ingen kontroll finns över beroendena vilket gör att det inte finns några modulära kopplingspunkter vid utökning av programmet, utan enda sättet att utöka är att gå in i den redan starka och långa kedjan av beroenden.

* Övriga principer

**High cohesion, Low coupling (HcLc):**

Som tidigare nämnt hanterar flera klasser, som ärver från klassen Entity, både grafisk representation och ljud utöver sin faktiska funktionalitet (t.ex. Peon, Monster och House). Det blir alltså svårt att avgränsa klassernas ansvar, som nämnt i SRP avsnittet, vilket gör den kontextuella cohesionen för flera klasser **låg**, dvs otydliga ansvarsområden. Eftersom en **high** cohesion, dvs sammanhållning, lägger grunden till att moduler kan återanvändas i andra sammanhang så blir programmet orörlig (”immobile”) på grund av en **låg** cohesion.

**Law of Demeter (LoD):**

* Övergripande designkvalitét

**Designmönster**

* [Titlar på alla designmönster vi hittar]

Refaktoriseringsmöjligheter

**Designprinciper**

* [Förbättringmöjlighet, med fördelar&nackdelar]

**Generell kod**

* [Förbättringmöjlighet, med fördelar&nackdelar]

Sammanfattning