INLEDNING

UML-ANALYS

**UML/Kommentarer**

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

Övergripande designkvalitét

Programmet tar ingen direkt hänsyn till de underliggande principer som Objekt-Orienterad programmering förespråkar. Dessutom förekommer flera ”Code Smells” i form av: långa nästlade if-satser som dessutom gör koden svår att läsa och förstå (se klassen TowerComponent), konceptuella problem i arvshierarkier (t.ex. klassen Rock ärver från klassen Entity, dvs Rock har metoden fight()…), svårt att återanvända koden (t.ex. hårdkodad placering och funktionalitet för alla knappar i spelet), svårt att förändra problem i koden (t.ex. att lösa det konceptuella problemet med rock hade inte fungerat utan stora refaktoriseringsinsatser).

Programmet påvisar dock smart och imponerande funktionalitet vilket går till spillo i den omodulära designen.  Följande avsnitt kommer att diskutera SOLID principerna och diverse andra för att sedan följas upp med en analys av övergripande användningar av Design-Mönster och sedan avslutas med möjliga förbättringar/refaktoriseringar.

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.

Man 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 man 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 man 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 man 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 man 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 man kan uppnå om man 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):**

Programmet har ingen paketindelning överhuvudtaget så fokus ligger mest på coupling och cohesion mellan de olika klasserna.

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.

Det är svårt att hitta konkreta brott mot Low coupling principen då ingen paketindelning existerar överhuvudtaget. Denna brist på paketuppdelning är inte heller rättfärdigad då klassernas cohesion är låg och en paketuppdelning och refaktorisering med avseende på ansvarsområden och kontext hade varit högst möjligt, om inte nödvändigt. Mer om det diskuteras i senare avsnitt.

**Law of Demeter (LoD):**

Law of Demeters syfte är att reducera beroende och på så vis endast ge en klass den högst nödvändiga informationen den behöver för att uppfylla sitt syfte. Eftersom flera klasser redan bryter mot SRP så har klasser otydliga ansvarsområden vilket i sin tur leder till att det är svårt att avgöra vad som är ”högst nödvändig information” för klassen. (se exempel från SRP-avsnittet) Man kan dock konstatera att LoD inte har tagits i hänsyn vid skrivandet av koden. Hade kunnat vara värre då Island som anropar tick() för alla entiter och uppdaterar således entities som genom polymorfism kör egna render() metoder i de individuella subklasserna. Detta används dock snarare för fördelen med polymorfism än med hänsyn på LoD då principen bortses från på så många andra ställen.

**Command-Query Separation Principle(CQSP):**

Det främsta syftet med CQSP är att man aldrig skall ha en metod som både ändrar på ett tillstånd, och returnerar ett värde. I kodbasen finner man flera instanser där detta inte följs.

I både Rock, Tree och Tower så finns metoder vid namn gatherResources() som inte bara returnerar ett booleskt värde, utan även ändrar på instansvariabler.

Vi ser även liknande brott mot principen i: House.submitResource(), House.build(), HouseType.setAcceptsResource(), Job.hasTarget(), Job.Plant.hasTarget(intern klass i Job) och Sound.read(). Den sistnämnda tar in en Array som argument, fyller den och returnerar ett booleskt värde. De andra metoderna tenderar att ändra på instansvariabler. Detta kommer att leda till problem om man skulle vilja bygga ut programmet. Framtida utvecklare kan eventuellt komma att förvänta sig ett booleskt värde när man kallar på en boolesk funktion, och inte inse att programmets tillstånd faktiskt ändras av detta.

Designmönster

I sitt nuvarande stadie uppenbaras ett stort behov av designmönster i kodbasen. Man kan dock identifiera och dra paralleller med ett antal olika designmönster. Den abstrakta klassen sound har en mycket god implementation av **Template Pattern**, där metoden read() använder sig av den abstrakta metoden fill(). Var och en av subklasserna implementerar sedan fill() för få ut ett unikt ljud. Vidare kan man dra paralleller med ett antal andra designmönster, antingen i funktion, eller i implementation. HouseType är ett mycket intressant exempel. Sättet som HouseType är designat på är som sådant att man aldrig kan instansiera det, och det existerar ett bestämt antal instanser av det. Denna funktionalitet påminner mycket om **Singleton Pattern.** Vad man dock valt att implementera är statiska variabler som ger ut en instans av HouseType, och där dessa instanser skapas vid runtime, istället för vid ett metodanrop. Vid instansiering av klassen House, så ges instansen en specifik instans av HouseType. Alla instanser av HouseType har olika booleska värden och primitiva variabler. House i sin tur delegerar sedan vissa av dess metoder till instanser av HouseType. Denna implementation av ett Singleton-liknande mönster har dock nackdelar då man t.ex. skulle vilja utöka programmet med fler hustyper. Varför har man t.ex. inte valt att bygga en arvshierarki med House som superklass till fler interna klasser? Fler konkreta förbättringsförslag återfinns senare i detta dokument.

Som nämnt finns viss delegation i House till HouseType, men detta går inte att knyta till något specifikt designmönster. Denna delegation är nödvändig med tanke på hur House och HouseType är skrivna, men den borde inte behöva finnas. Man kan inte heller, trots namnet på superklassen Entity, knyta några paralleller till **Entity Pattern.** Inga entiteter har nämligen egna identiteter, någonting som är ett strikt krav till Entity Pattern. Programmet kräver dock aldrig att man skall behöva komma åt någon specifik entitet, så det finns ingen speciellt bra motivation till att tilldela egna identiteter till entiteter i programmet.

Refaktoriseringsmöjligheter

**Designmönster**

Programmet har flera möjligheter för implementation vid refaktorisering. Nedan har ett flertal olika designmönster diskuterats som vid refaktorisering skulle innebära stora förbättringar ur ett objekt-orienterat perspektiv, t.ex. högre modularitet, kodåteranvändning och läslighet.

Ett bra mål hade i slutändan varit att implementera ett **Model-View-Controller (MVC)** mönster. Men för att nå detta är det tydligt att stora refraktoriseringar med andra mönster krävs.

Att avgränsa koden i olika paket för gränssnitt, modell och kontroller hade gett ett stort antal fördelar. Om korrekt implementerat skulle det gett moduler klara ansvarsområden, vilket leder till att SRP följs till en högre grad. Önskvärt är också att alla paket endast har avsiktliga publika klasser och interface. Detta skulle lett till lägre coupling och möjliggjort parallell utveckling av programmet.

Updateloopen hade kunnat konstruerats på lite olika sätt men i detta program funkar **Observer-pattern** utmärkt. Andra metoder som Polling där View paketet hela tiden frågar efter uppdatering hade kunnat användas men då t.ex. ljudeffekter endast spelas vid specifika tillfällen är detta överflödigt. Däremot Observer-pattern låter andra moduler “lyssna” på Modellen och således uppdatera sig då Modellen kallar på det. Nackdelarna här är bland annat att man inte har direkt kontroll över ordningen moduler uppdaterar sig men i detta fall är det en acceptabel kompromiss som stöds med rätt dokumentation för framtida utvecklare.

Ytterligare ett designmönster som kan appliceras vid refaktoriseringen är ett designmönster för skapandet av alla Entity-objekt, nämligen ett **Factory-Pattern**. Detta minskar och försvagar i sin tur beroendena mellan klasser utan effekt på funktionaliteten och binder dom istället till . Framförallt ger designmönstret dock stora möjligheter till kodåteranvändning. Abstraktionen genom Factory-pattern kommer dock inte utan nackdelar, programmet får ytterligare lager av komplexitet. Man kan dock återigen argumentera för att detta är en berättigad kompromiss då mängden av instanser där samma kod återanvänds är substantiell nog. Slutligen ger oss också Factory-pattern en viss framtidssäkring då vi modulärt enkelt kan lägga till nya t.ex. Entity-subklasser.

Slutligen kan en form av **State Pattern** implementeras. En av anledningarna till att renderGame(), och andra metoder är så väldigt långa är att spelet skall bete sig olika beroende på vilka booleska värden som är sanna. Betrakta specifikt won, running, paused och titleScreen. Metoderna i TowerComponent kontrollerar hela tiden tillståndet på dessa variabler och utför olika operationer beroende på dessa. Här kommer **State Pattern** in händigt. Att dela upp de olika tillstånden i “States” och låta dessa utföra sina egna metoder, skulle kraftigt ha minskat komplexiteten hos många metoder, samtidigt som det uppmuntrar till utökning av programmet. Man skulle t.ex. kunna lägga till svårighetsgrader, samtidigt som en klar ansvarsuppdelning skulle ha skett mellan de olika State-klasserna, vilket vid en god implementation, följer SRP till en högre grad. Man bör dock fråga sig själv om states verkligen är nödvändigt. Om man inte kommer bygga ut programmet på ett sätt där states är applicerbart, utan endast vill adressera nuvarande tillstånd så bör man ta hänsyn till att programmet är i samma tillstånd i princip hela tiden, och det finns andra ta hand om många av de problem som tagits upp här. Bland annat genom god funktionell nedbrytning.

**Generell kod**

Det finns många metoder som använder sig av långa, ibland flera, if-satser vilket gör dessa metoder väldigt långa, invecklade och svåra att förstå. Ett bra sätt att förbättra koden på skulle vara att funktionellt bryta ner metoderna och använda dessa i huvudmetoderna. Nackdelen med detta skulle då vara att det skulle vara mycket arbete att få det att fungera på grund av komplexiteten.

En väldigt udda metod, init(), finns i praktiskt taget alla spelobjekt. Den agerar som en konstruktor för att binda objektet, som varit skapat sen innan, till ön och grafiken. Enda stället i koden som den blir använd är i metoden addEntity() som också lägger till den i öns lista av objekt. Detta kan optimeras genom att ha init() i konstruktorn och att skapa spelobjektet genom addEntity() istället, så undviker man onödiga steg.

Själva klassen Entity är master klassen för alla spelobjekt, inklusive icke-spelobjekt, utan att vara abstrakt eller ens ha flera lager av arv inom “Familjeträdet”. Detta skapar problem då klasser som inte riktigt har något gemensamt med någon annan klass utanför att de båda ska finnas i spelet. Till exempel att infotext eller rök, effekter i praktiken, har hälsa, kan därmed dö, och en fight() metod. Sen har också statiska objekt, alltså icke rörliga spelobjekt, såsom hus och träd en fight() metod som inte ens används. Ett sätt att “städa upp” detta är att antingen göra ett eller flera interfaces för olika sorters spelobjekt beroende på vad de gör. Eller så kan man utöka arvet och ha flera abstrakta klasser som alla ärver av Entity baserat på funktion. Till exempel en för spelobjekt som ska röra på sig, som Peon eller Monster, kontra stillastående spelobjekt, som House eller Tree. Sedan utifrån de stillastående kan man ha en abstrakt klass som är för resurser, Rock och Tree.

Slutsats

Vid första anblick är BreakingTheTower ett roligt och imponerande tidsfördriv med trevliga grafiska interface. Men bakom dessa fina användargränssnitt och denna tidsfördrivande spellogik utbreder sig en stor uppsjö av hastigt fattade beslut och bristfälliga designbeslut. Programmet visar alla tecken på småskalig programmering, helt utan en avsiktlig objekt-orienterad grund. Vi finner dessutom väldigt få tecken på designmönster. Koden är en mycket bra läxa för hur det kan gå när man inte planerar utvecklingen av sin kod, och agerar praktexempel för hur någonting kan vara fint på utsidan med “spaghetti” och “code smells” på insidan. Mycket av detta kan dock försvaras av det faktum att programmet är hastigt skrivet under 48 timmar, för att användas i en tävling.