Objektorienterad analys av BreakingTheTower

Grupp 17

Oscar Sanner
Olof Sjögren
Robert Sahlqvist

Innehållsförteckning

OBS! UML-Diagram bifogas separat

Inledning

Detta projekt är en Objektorienterad analys av kodbasen för spelet BreakingTheTower. Spelet skapades till kodningstävlingen Ludum Dare 2008 och kom på 3:e plats. Det är ett strategispel där man bygger diverse byggnader och hanterar enheter för att förstöra ett torn på andra sidan ön. Det finns ett grafiskt gränssnitt med knappar och informativ text och diverse ljud spelas upp vid vissa interaktioner.

Följande avsnitt kommer att diskutera övergripande designkvalité, SOLID principerna och diverse andra principer. Detta följs sedan upp med en analys av övergripande användningar av Design-Mönster för att sedan avslutas med möjliga förbättringar/refaktoriseringar. Dessutom finns ett UML-diagram med kommentarer bifogat.

Övergripande designkvalitét

Programmet tar ingen direkt hänsyn till de underliggande principer som Objektorienterad 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 kommer att visa sig svår att återanvända vid framtida utökningar.

SOLID-principer

Single Responsibility Principle (SRP):

Koden tar allmänt ingen avsiktlig hänsyn till SRP vilket leder till många ”uppblåsta” klasser 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 andra designmönster som annars hade varit applicerbara (MVC, tas upp senare) men dessutom ger detta klasserna low cohesion, vilket är något man uppnår med SRP, eftersom klasserna hanterar både grafiska aspekter och funktionalitet som ofta är en mycket stark avgränsning inom objektorienterad 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. 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 klassen. För övrigt är specifikt metoden ett stort brott mot Separation of Concern.

Övriga avsteg från principen kan finnas i bland annat HouseType-klassen, där HouseType representerar en specifik variant av ett hus samtidigt som den skapar instanser av sig själv via globala variabler. Den påvisar alltså något som liknar Singelton Pattern (Diskuteras i senare avsnitt) som i grunden bryter mot SRP. Här får man göra en avvägning då både SRP och Singleton fundamentalt är motstridiga i sina principer och designfilosofier. Men då “Singleton Pattern”:et (om

det ens går att kalla det det) är dåligt implementerat i detta fall är det högst tveksamt om brottet mot SRP går att rättfärdiga.

Dessa är endast några av de flertal avstegen från principen vilket leder till att all klasserna får högre beroenden mellan varandra (syns t.ex. i att många klasser hanterar ljud själva, bland annat de flesta klasser i Entitet hierarkin.) 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.

Man kan dock påpeka att viss mån har tagits till OCP vid skrivande av programmet. I klassen Island återfinns en lista med Entiteter, vilka läggs till via addEntity(). Och det enda som krävs av ett Entity-objekt är att det faktiskt skall ärva från klassen "Entity". All logik som rör dessa objekt programmeras direkt i klasserna, och trots att hierarkin med Entity bryter mot många andra principer så leder dess design till viss öppenhet för utbyggnad av programmet. Dessutom anropar endast TowerComponent uppdatering på Entity-objekt och behöver därför inga beroende på de direkta Entity subklasserna. Det leder dock inte till att programmet är helt stängt för modifikation då man fortfarande kommer behöva ändra i "Bitmaps" och "Island". Ytterligare är programmet uppbyggt av dubbla och cirkulära beroenden, vilket leder till att det också blir väldigt invecklat att bygga ut programmet. De dubbla beroenden och kedjeberoenden som finns gör programmet väldigt stelt, och allt annat än stängd för modifikation.

Finns några tecken på kodåteranvändning (t.ex. isFree()) i Entity som återanvänds 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.

Fler exempel på avvikelser från OCP är den stora mängden av hårdkodning som förekommer i koden. Betrakta sättet som knappar och andra grafiska widgets ritas upp på. Istället för att låta knapparna vara någon form av objekt, så har man valt att rita upp knapparna på specifika koordinater i spelet. Man har därefter hårdkodat att om musen är på samma specifika koordinater som den uppritade ikonen är så kommer viss logik ske. Logiken är alltså skild från ikonen på ett sätt som gör det svårt att lägga till nya knappar. Denna typ av hårdkodning återfinns på många andra ställen i programmet och gör programmet mycket svårförståeligt. Som exempel på en annan typ av hård kodning används “random.nextDouble() * 256 - 128) * 1.5;” för initial utplacering av flertalet spelobjekt, trots att det helt saknas någon förklaring av vad värdet faktiskt innebär.

Liskov Substitution Principle (LSP):

En av de första egenskaperna man noterar när man analyserar koden är den breda användningen av subklasser. Vad som dock också noteras är att majoriteten av klasserna, med undantag för interna klasser i Job och Sound, är subklasser till klassen Entity.

För att ge ett exempel där LSP effektivt har följts kan man titta på klassen Sound. 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 subklasserna. Detta kan ses som en bra implementation av LSP. Dock sker detta på grund av att inga metoder överskuggas i subklasser, vilket gör det enklare att följa LSP.

Kodbasen innehåller 3 arvshierarkier. Vid första anblick påpekar man att många metoder i superklasserna för dessa arvshierarkier lämnas tomma, eller returnerar booleska värden per default. Metoderna har alltså varken några förvillkor eller direkta eftervillkor och det blir svårt att diskutera huruvida detta avviker från LSP. Man kan dock strikt argumentera för att dessa metoder, tillsammans med hela dess superklasser, borde vara abstrakta. Alternativt borde dessa metoder vara del av ett interface.

Man kan dock finna exempel på metoder som har funktionalitet i superklassen och överskuggas med ändrad funktionalitet i subklassen. Mer specifikt finns det en metod i Job, vid namn hasTarget(). Metoden lovar som eftervillkor att ge det booleska värdet true om och endast om instansvariabel "Entity e" inte är null och flera andra villkor uppfylls. Metoden hasTarget() i subklassen Job.Plant har dock helt andra eftervillkor, givet att den booleska instansvariabeln "hasSeed" är falsk. Detta leder till att LSP inte riktigt följs, då hasSeed inte är ett förvillkår i superklassens metod, eller ens existerar. Vad man bör fråga sig är vad detta får för konsekvenser. Att samma metod gör helt olika saker beroende på vart den körs är självklart mycket förvirrande. En klar dokumentation hade säkerligen hjälpt mycket i detta fall, men det bästa vore om metoderna kunde hållas separerade.

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 annat exempel där denna kodande med denna princip i åtanke hade förbättrat programmet är i superklassen Entity där flera metoder lämnas tomma för att sedan överskuggas i de specifika subklasserna. Alla klasser överskuggar dock inte alla metoder (t.ex. klassen Rock som har metoden fight()) och här hade interfaces kunnat implementeras för att abstrahera de metoder som i dagsläget är konceptuellt ifrågasättbara eller tomma och lämna kvar de som garanterat är gemensamma för alla subklasser. Ett direkt brott mot ISP återfinns i klassen TowerComponent där uppdateringen av spelet och logiken ligger. Denna klass implementerar bland annat Java-interfacet MouseListener. Här används dock aldrig metoderna mouseEntered() och mouseClicked() och de har således lämnats tomma. Implementation av interfaces för “listeners” är dock nödvändig vid användning av awt-biblioteket och i detta fall är det ett skulle man kunna försvara det som ett nödvändigt ont.

Trots detta så är valet fortfarande högst ifrågasättbart då man enligt god objektorienterad 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 det inte implementerats några interfaces visar att det inte finns några abstrakta beroenden vilket leder till hög sammankoppling mellan klasserna. 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 beroendekedjor.

Dessa brott mot DIP gör att programmet har både många och starka beroenden mellan klasser på alla nivåer, 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. Det 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 så att det blir väldigt låg kontextuell cohesion för flera klasser och otydliga ansvarsområden.

Det är svårt att hitta konkreta brott mot low coupling principen då ingen paketindelning existerar överhuvudtaget.

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. Detta leder som tidigare nämnt till otydliga avgränsningar av klasserna.

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

Man 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.

Vad man dock bör notera är att vissa metoder är formulerade som om de skall utföra en uppgift, de är inte formulerade som om de skall returnera ett värde t.ex. gatherResources(). Här gör det inte så mycket att det “hänger med ett booleskt värde”. Vad som är värre är metoder som hasTarget() och Sound.read() som är formulerade på sätt som gör att man inte förväntar sig att något skall ändras. Här vore det återigen mycket passande med god dokumentation. Någonting som saknas genom hela kodbasen.

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 vilket är typiskt för ”äkta” Singelton Pattern. Vid instansiering av klassen House, så ges instansen en av de specifika instanserna 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. Slutligen finns inget tecken på, eller paralleller mellan, Entity-Pattern med tanke på Entity klassen. Inga entiteter har nämligen egna identiteter, någonting som är ett strikt krav till Entity Pattern.

Refaktoriseringsmöjligheter

Designmönster

Programmet har flera möjligheter för implementation av designmönster vid refaktorisering. I följande paragrafer kommer flera olika designmönster diskuteras som vid refaktorisering skulle innebära stora förbättringar ur ett objektorienterat perspektiv, t.ex. högre modularitet, kodåteranvändning och läsbarhet.

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 Model View och Controller hade gett ett stort antal fördelar. Om korrekt implementerat skulle det gett moduler klara ansvarsområden, vilket leder till att SRP kan följas 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 låter Observer-Pattern 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 vilka moduler som uppdaterar sig men det går att implementera flera Observer-Patterns för olika uppdateringar (t.ex. en för grafisk uppdatering och en för ljud uppdatering).

Ytterligare ett designmönster som kan appliceras vid refaktoriseringen är ett designmönster för skapandet av alla Entity-objekt, nämligen ett **Factory-Pattern**. Detta knyter beroendena till en abstraktion och inkapslar skapandet av objekt. Dessutom ger designmönstret 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 man 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 ger en viss möjlighet 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. Detta leder in på nästa segment.

Kodförbättringar

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.

Vad som också bör adresseras är den ofta totala bristen på förklarande namn på variabler. För att ge exempel så ges oftast koordinater namn så xr istället för xRotation, och xe istället för xEntity. Detta i kombination med en mycket god dokumentation skulle göra kodbasen mycket enklare att förstå, bygga ut och underhålla.

Klassen Entity är superklassen till alla spelobjekt utan att vara abstrakt eller att ha flera lager av arv inom arvshierarkin. Detta skapar problem då klasser som inte riktigt har något konceptuellt gemensamt med någon annan klass utöver att de båda ska finnas i spelet. Till exempel att InfoPuff eller Puff, effekter i praktiken, har en fight() metod, hälsa och kan därmed dö. Dessutom har också statiska objekt, dvs icke rörliga spelobjekt såsom hus och träd, en oanvänd fight() metod som de ärver från Entity. Ett sätt att lösa dessa konceptuella och strukturella problem är att antingen skapa interfaces i enlighet med ISP. Eller så kan man utöka arvshierarkin och ha flera abstrakta klasser som alla ärver av Entity baserat på funktion. (t.ex. indelning efter natur, byggnader och varelser). Det finns dock ingen gräns på hur många interfaces en klass kan implementera men en klass däremot endast ha en superklass.

Utöver detta finns oanvända metoder (t.ex. cut() i Tree), variabler (t.ex. yield i Tree) och klasser (t.ex. Vec) utspridda i källkoden. Dessa bör tas bort eller ändras och/eller implementeras.

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 objektorienterad grund. Man 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.