Note: Please unzip the File before running Breakout.jar. Please run Configurator.jar before running Breakout.jar.

Erklärung des MVC-Patterns und Problem Decomposition:

Für Eigenschaften die langlebiger sein müssen als bloß ein Methodenaufruf verwenden wir Instanzvariablen. Für Zwischenergebnisse und Sachen die nach dem Methodenaufruf nicht gespeichert werden müssen benutzen wir lokale Variablen. Die Eigenschaften der Modelle sind als Instanzvariablen gespeichert, damit diese langlebig sein können und jederzeit während der Laufzeit abrufbar sind.

Unser Programm ist (wie gefordert) unter Anwendung des MVC-Schemas geschrieben worden.
Es gibt Models, Views, und Controller.

Phasen und grober Ablauf:

1. Init: Der Controller schickt an die View den init() Befehl und initiiert das MainModel mit dem Starttext. (Model wird von Controller manipuliert). Die View kriegt einen update() Befehl (View stellt Model dar) und zeigt den Text an der richtigen Stelle an. Keyboard und Mouse Listeners werden geaddet.
2. Start Game: Wenn die Leertaste gedrückt wird wird das Model durch ein neues Model mit Blöcken aus dem LevelManager, einem Ball und einem Paddle ersetzt und update() der view aufgerufen. Weiterhin wird jedes KeyEvent geblockt um einen Neustart während des Spiels zu verhindern.
3. Der GameLoop wird gestartet
4. Gameloop: pro Loop wird die Position des Balls geupdatet. Abhängig davon ob gerade ein Bossfight stattfindet oder nicht wird:
   (a: Kein BossFight | b: Bossfight)
   a. Collision mit Blöcken überprüft und kollidierte Blöcke aus dem Model entfernt
   b. Der BossMove wird ausgeführt und die entstehende Laser dem Model hinzugefügt. Außerdem werden Laser im Model geupdatet.

   Am Ende wird der Ball final bewegt und die view geupdatet.
5. Falls der Spieler gewonnen oder verloren hat wird Abhängig von 2 lokalen Variablen winScreen oder looseScreen aufgerufen.
6. Hier wird das Model mit entsprechenden Text ersetzt und und der Hintergrund entsprechend eingefärbt und die View geupdatet. Außerdem werden KeyEvents wieder freigeschaltet.

Model:

Zum Model gehören die Packages net.lighthouse.levels und net.lighthouse.model . Ihr Scope ist public, da das Model und alle Modelobjekte in Controller, Collider und View benutzt werden.

Model Package:

Im Model Package sind:

1. alle Objekte, die in unserem Breakout vorhanden sind (BBall, BBlock mit Collection BlockList, BBoss, Blaser, BPaddle und BText, sowie das Grundobjekt BObject.
2. Eine Klasse, die den momentanen Status des Spiels darstellen kann, indem sie Informationen über momentan vorhandene Objekte speichert und Methoden zum manipulieren stellt (MainModel)

Zu 1.:
Das Grundobjekt ist das BObject. Es hat eine Position, Dimensionen (Höhe/Breite) und eine Farbe.
Alle Objekte im Spiel erben von BObject. Daher haben sie auch die oben genannten Eigenschaften. Außerdem hat

- BBall eine interne Repräsentation der Geschwindigkeit des Balls, Getter und Setter dafür, eine move Methode und und eine Methode die die voraussichtliche nächste Position returned.
- BBoss eine interne Darstellung seiner HP
- BPaddle eine move Methode auf x.
- BText eine interne Darstellung von seinem Text und Getter+Setter dafür.

BBlock ist ein BObject und hat keine zusätzlichen Methoden. Blaser ist ein BBall und hat keine zusätzlichen Methoden.
BlockList extended ArrayList<BBlock> und stellt die Methode getBlockAtXY, mit der man einen Block an einer bestimmten Stelle holen kann.

Zu 2.:
MainModel ist die Klasse die bei uns den momentanen zustand des Spiels darstellt (Model). Sie

- hat Listen für alle Objekte, die im Spiel vorkommen können (außer Paddle, es gibt nur ein Paddle), sowie Getter und Setter für diese Listen.
- merkt sich den userScore.
- stellt mit addObject eine Methode etwas dem Model hinzuzufügen.
- Stellt mit contains eine Methode um zu testen, ob ein bestimmtes Objekt im Model ist.

- Ist iterable. Wenn iterator() aufgerufen wird, werden alle Objekte, die momentan im Model sind mit toArrayList Methode zu einer Arraylist hinzugefügt und der iterator dieser ArrayList zurückgegeben.

LevelManager:

Die Klasse ist public da sie im MainModel benutzt wird, um neue Levels zu bekommen. Da es keine Instanzen vom LevelManager geben soll, ist der Konstruktor private. Die Levels werden intern als 2d Array von Colors gespeichert. Die Klasse getRandomLevel() wählt “zufällig” ein Level aus und lässt aus diesem eine BlockList bauen, welche zurückgegeben wird. Wenn eine Color null ist, wird für diese kein Block erstellt. Ansonsten wird ein BBlock mit der Color erstellt.

View:

Beschreibung

Zur View gehört das Package net.lighthouse.view.
In diesem Package ist das Interface „View“. Dieses Interface definiert das Verhalten einer MainView. Eine solche MainView muss

- Eine init() Methode haben, die das GraphicsProgram Fenster rescaled und das lighthouse initialisiert (falls Settings das sagen)
- Eine update() Methode haben, die die Anzeige auf den Stand des Models bringt.

Wir haben in unserem Projekt 2 verschiedene Views.
Die erste und ältere ist die legacyMainView im Package net.lighthouse.view.legacy . Sie löscht bei aufrufen von update() alle Objekte vom GCanvas des GraphicsPrograms mit dem sie konstruiert wird, und zeichnet dann den gesamten Frame nach vorgaben des Models neu. Durch diese vorgehensweise flackert der Bildschirm allerdings stark und es ist ein bisschen brute-force und nicht besonders schön. Deshalb wurden die legacyMainView und die legacyClientView mittlerweile durch einen rewrite ersetzt. Die legacyMainView hat keinen support für BBoss, BText, BLaser.
I only keep it around in case something happens to the new Main View and I want to test it in the old Systems.

Die 2. und nach allen Maßstäben bessere View ist die MainView im net.lighthouse.view.rewrite Package. Sie hat Support für alle Objekte im Model und client-only Objekte. Ebenfalls im rewrite Package sind die Klassen BLink und BLinkList.
BLink ist eine Klasse, die ein Objekt im Model and ein GObject bindet und es somit möglich macht, pro frame nur Objekte aus der View zu entfernen/bewegen, die auch im Model geändert sind.

Die BLinkList extended ArrayList<BLink> und gibt zusätzlich hilfreiche Methoden when dealing with multiple BLinks.
Ein update in der neuen MainView sieht ungefähr so aus:

1. wenn wir im ersten Aufruf sind werden alle Objekte aus dem Model dem Screen und der BLinkList links hinzugefügt.
2. Es wird geguckt, Ob es objekte gibt, die noch in links sind, aber nicht mehr im Model. Diese Objekte werden vom screen und aus links gelöscht.
3. 3. Es wird geguckt, ob es Objekte gibt, die im Model hinzugefügt wurden, aber noch nicht auf dem Screen / in den links sind. Wenn ja, werden sie dem Screen und den links geadded.
4. Es wird geguckt, ob es Objekte gibt, die sich im Model bewegt/ihre Farbe geändert haben. Wenn ja, werden sie auf dem Screen entsprechend geupdated.
5. Falls das Lighthouse benutzt werden soll, wird ein update ans lighthouse gepusht (Dazu gleich mehr).
6. Das Canvas vom Client wird gerepainted.

Zu 5.:
Ich war dabei, eine “vernünftige” Art zu schreiben, das Lighthouse zu updaten. Doch dann hatte ich eine bessere Idee: Screenshots! And thus, DarkhouseView was born.
Die DarkhouseView funktioniert wie folgt:

1. Die update Methode des LighthouseScaler nimmt sich das GCompound der MainView, in dem die GObjects drin sind, die auf dem Lighthouse angezeigt werden dürfen (AKA alles ohne Text).
2. Es wird ein BufferedImage erstellt und der Grafikkontext des GCompound darauf gepainted.
3. Dieses BufferedImage wird auf 14x28 downgescaled.
4. Ein davon wird ein GImage erstellt. Falls der Framebuffer gespeichert werden soll wird das Image auf die Platte gespeichert.
5. Der LighthouseHandler macht weiter: seine Update Methode bekommt das GImage, rechnet es mithilfe des Converters in Lighthouse-Format um und schickt das Ergebnis an das Lighthouse.

Scopes:

BLink und BLinkList sind Package private da sie nur in der MainView verwendet werden. MainView ist public weil sie im Controller benutzt wird. Same with

legacyMainView. legacyClientView ist Package private weil sie nur von legacyMainView im selben Package benutzt wird.
DarkhouseScaler ist public weil er von Klassen in net.view.lighthouse.legacy und net.view.lighthouse.rewrite benutzt wird. DarkhouseHandler ist Package private weil er nur vom Scaler im selben Package benutzt wird.
Die Converter Klasse hat einen statischen Constructor weil es keine Objekte der Klasse geben soll. Ihre Methoden sind aus diesem Grund static. Sie ist Package private weil sie nur von DarkhouseScaler benutzt wird.

Controller:

Das spielt hat 4 Controller. Der MainController ist der “Benutzer” der 3 Anderen Controller.

Der MainController:

Scope: Pulbic da diese Klasse die main methode enthält, welche beim Programmstart ausgeführt wird.

Der Main Controller kontrolliert den Fluss des Spiels. Er extended auch GraphicsProgram wie in den VorlesungsFolien. Der MainController stellt allen anderen Komponeten die das Spiel beeinflussen und kontrollieren die benötigten Modelle zur Verfügung. Er verarbeitet weiterhin jegliche Usereingaben. Zum Spielstart und Win/Loose Senarios ändert er das MainModel entsprechend, damit ausschließlich Text angezeigt wird. Wenn ein Spiel läuft wird das MainModel mit Spiel Objekten bestückt.
Die GameLoop befindet sich ebenfalls im MainModel. Die FPS können variabel durch die Settings bestimmt werden. Ein Loop Durchlauf beeinflusst mit Hilfe der 3 anderen Kontroller den Ball, Paddle, Blöcke, Boss und Laser.

Der CollisionChecker:

Scope: Wird von den Controllern verwendet, hat aber ein eigenes Package. Deshalb public damit die Methoden von den Kontrollern ausgeführt werden können.

Der CollisionChecker operiert für BBalls. Objekte von ihm prüfen bei jeden Gameloop Durchlauf die Collisions mit Borders, Paddle, Blöcken und Boss. Da BLaser von BBall erbt, operiert der CollisionChecker auch für die Laser und prüft im BLaserController die Colission mit dem Paddle und Borders.
Gegebenenfalls wird außerdem ein boolean zurück gegeben der definiert ob eine Kollision stattgefunden hat. Mit dem kann der MainController dann den Spielverlauf weiter beeinflussen.

Der BossController:

Scope: Package-Private da er nur vom MainController benutzt wird.

Instanzen des BossController beeinflussen das Verhalten des Bosses. Sie werden bei jedem GameLoop Durchlauf aufgerufen. Ein BossMove ändert die Farbe des Bosses, wenn dieser Schaden genommen hat oder fast Tot ist. Außerdem gibt er bei Schaden ein BLaser zurück welcher vom MainController
im MainModel gespeichert wird.

Der BLaserController:
Scope: Package-Private da er nur vom MainController benutzt wird.

Dieser Controller stellt eine statische Methode zur Verfügung um alle BLaser zu updaten. Die Methode ist statisch, da keine Objekte von BLaser benötigt werden, da keine einzelnen LaserKontroll Eigenschaften in einen Objekt gespeichert werden muss. Ein Update führt CollisionChecking mit Hilfe des CollisionControllers durch und entfernt Laser wenn nötig. Außerdem wird es dem MainController mitgeteilt, falls ein Laser das Paddle trifft und somit das Spiel verloren ist.

Settings

Unser Spiel stellt Settings zur Verfügung welche in einer settings.txt definiert werden können. Diese Datei muss im Projectroot Ordner (jedenfalls bei IntelliJ) liegen, sofern das Projekt durch ein IDE gestartet werden soll. Wenn keine settings.txt gefunden wird, werden Default Settings verwendet. Zur Not können auch diese verändert werden.
Die Settings sind public da sie überall verwendet werden sollen. Von den Settings können keine Instanzen erstellt werden, das es während des Spiels keine unterschiedlichen Settings Werte geben sollte und somit alle Settings als ArrayList in einer Class Variable gespeichert werden…
Mögliche Settings:

Key	**Value**	**Default Value**
user-name	UserName in der Web-View	‘Empty’
token	API Token fürs Hochhaus	‘Empty’
web-view	Ob die Web-View gerendert werden soll. true or false	false
use_new_viewport	Ob der neue Version der Client-View benutzt werden soll. true or false	true
frametime	Abstand zwischen GameLoop Durchläufe in Millisekunden. Empfohlener Wert: 40	40
print-frametimes	Gibt die Zeit aus die benötigt wurde, bis ein neuer Frame berechnet wird. true or false	false
save-framebuffer	Speichert jeden gerenderten Lighthouse Frame in dem Projekt Ordner. true or false	false