System design document for Panic on TDAncefloor

Version: 1.1

Date: 2017-05-28

Authors: Farzad Besharati, Gustav Lahti, Rikard Teodorsson & Magnus Wamby

This version overrides all previous versions.

1 Introduction

Panic on TDAncefloor is a game designed to run on both PC and Android. The game is designed to be a wave-based top-down shooter, with a modular item system, which affects the implementation and requirements.

1.1 Design goals

The goal with *Panic on TDAncefloor* is to make a top-down shooter game that runs on both Windows computers and Android devices. *Panic on TDAncefloor* should also have items that affect the Characters or their Projectiles.

1.2 Definitions, acronyms and abbreviation

PoTDA is short for *Panic on TDAncefloor*.

The **Program** is the software, regardless of platform, equivalent to PoTDA.

The **Game** is a subsection of the software, divided into the **running** and **inventory management** state. Menus and similar elements are not directly considered a part of the game.

An **Actor** is an entity in the game world.

A **Character** is a more autonomous entity in the game world. Subtype of Actor.

A **Projectile** is a more temporary entity in the game world that collides with and damages Characters. Subtype of Actor.

An **Obstacle** is a static entity in the game world that blocks the movement of Projectiles and Characters. Subtype of Actor.

The **User** is the person playing the game.

The **Player** is the Character the user is controlling.

An **Inventory** is a collection of items used to modify a Character or a Character’s attack(s).

A **Storage** is a collection of the player’s unused items.

LibGDX is the framework used in the Program. LibGDX contains most of the libraries used, including the physics engine and I/O.

Box2D is the physics engine used in PoTDA and is included in LibGDX.

2 System architecture

The only machine involved in running the Program is the machine itself, no external connections are required once the Program is downloaded on said machine.

2.1 Platforms

The game runs on a Windows computer or Android device. Both platforms use the latest version of the Program. In most aspects the program is identical. However, resource management and input is on a platform basis and is handled by LibGDX.

The machine provides for up to three of the following control schemes: keyboard and mouse, keyboard only, and touch-screen joysticks.

2.1.1 System flow

The overlaying flow of the program consists of a top-level application as an entry point. It is further divided into a screen for each core program state: menu, game, inventory, pause and game over. The application has a single screen active at any given time, though the game screen can be sleeping in the background.

Each screen is fully responsible for its own functionality, excluding the parameters used in construction. Screens set the current screen of the program autonomously.

- The **menu, pause and game over screens** simply serves as views that allows the user to start the game and change settings.
- The **game screen** is the core of the Program: the game screen is responsible for creating the Game's MVC-system and updating it. It creates and switches to all other screens as necessary based on user input or Game state.
- The **inventory screen** is responsible for manipulating two Game elements, inventory and storage, via a menu-like UI.

The Game's MVC-system is as mentioned created by the game screen. A single Actor is composed of four objects: a controller, a model, a view and a physics actor. The controller, model and view components follow a passive MVC pattern, where the controller both updates and handles communication between the model and view. The physics actor is an interface towards a physics engine, currently the external library Box2D.

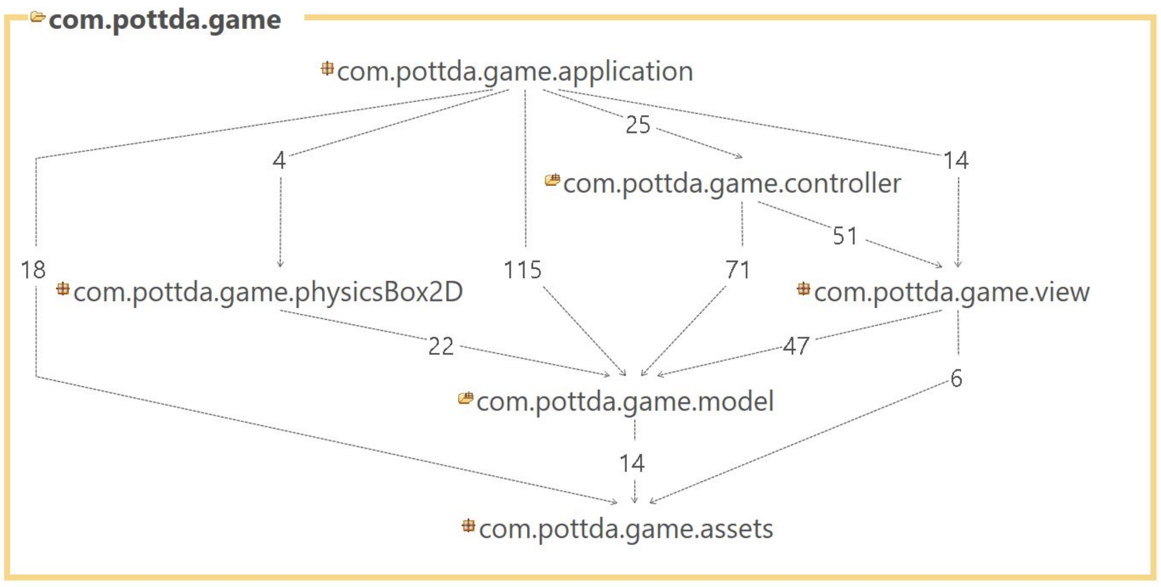
There are two parts of connecting the MVC: a controller hookup and a model actor builder. When the model creates a new actor, the builder notifies the controller hookup which in turn creates the appropriate view and controller.

3. Subsystem decomposition

PoTDA is divided into six main packages:

- Application
- Assets
- Controller
- Model
- PhysicsBox2D
- View

The following screenshots taken with STAN showcases package design.



3.1 Application

The application serves as entry-point and handles the overarching set up of the system. The application is split into a group of screens of varying complexity. The simple menu screens are limited in reach, displaying a UI, handles inputs and redirects to other screens. A screen is designed to be mostly self-contained and can be easily instantiated as needed.

The game screen is the main part of the system, and contains the game. It is responsible for creating and maintaining the model when the screen is rendered. A large part of the model-handling functionality is delegated to other classes in the controller and model-package and work mostly autonomously once set-up with listeners.

The inventory screen is reached from the game screen and serves as a user interface for managing the storage and inventory. It delegates its functionality to a passive MVC-system, with a controller, model and view.

When a screen is no longer needed, it is disposed. In the case of a game screen, the model is removed by garbage collection as nothing else references it.

3.2 Model

The Model is further divided into three packages:

3.2.1 Model

The model package contains model actors and concrete implementations for character, projectile and obstacle. The concrete implementations primarily differ in interaction with each other.

- Characters contain game play values and an inventory.
- Projectiles contains damage value and collides on impact with characters and obstacles. It is responsible for interacting with the character.
- Obstacles do nothing in the model layer (functionality is primarily in physics layer). It contains the abstract item and attack item implementation and inventory.

The package also contains blueprint and xml-classes that handle XML-assets to generate inventories, enemies and enemy groups. The blueprints are in essence a cache for the parsed XML-items to increase run-time performance. The xml-classes are data types containing the values required to create an inventory, enemy or enemy group.

3.2.2 Builders

Contains utility-like functionality for instantiating model actors. The builders have a common instance that notifies listeners when new actors are created. These are passed on to the controller hookup detailed in controllers.

3.2.3 Items

The classes in this package are all subclasses of Item. They are used by the Inventory to modify its stats and/or launch Projectiles and/or to modify Projectiles when they are spawned, destroyed, after a certain amount of time, or after the Projectile has travelled a certain distance.

Items has one abstract class called SizedItems, the classes that inherit this or any of its sub-classes use an enum to allow for three different sizes of the same Item-implementation. The size affects the size in the Inventory, the magnitude of its stat(s), and its drop chance.

3.3 Controller

The Controller package updates the Model and then the View. In the case of AIController it also updates EnemyHealthBarController, which in turn only updates views.

This package also has a class that takes an Actor and gives it together with a View to a Controller.

3.4 View

This package contains the view for most Actors, a factory for Obstacles' textures, and screen textures.

3.5 PhysicsBox2D

This subsystem is an adapter between the Model and Box2D, the physics engine used in PoTDA. PhysicsBox2D has PhysicsActors that hold a Box2D body and can take inputs from the Model to affect that body. This subsystem also has a factory to create PhysicsActors and their bodies.

Additionally this package has a listener that is added to the physics world. This listener is used to detect collisions and determine if further actions should be taken by either party.

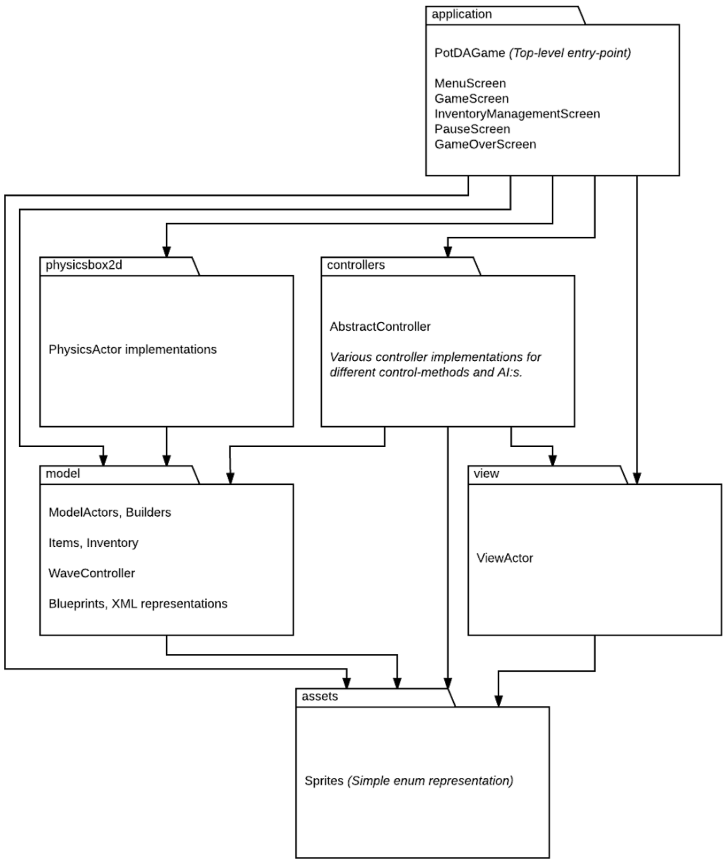
The purpose of having PhysicsBox2D instead of integrating it in the Model is to make it easier to change the physics engine in case of further development.

3.6 Assets

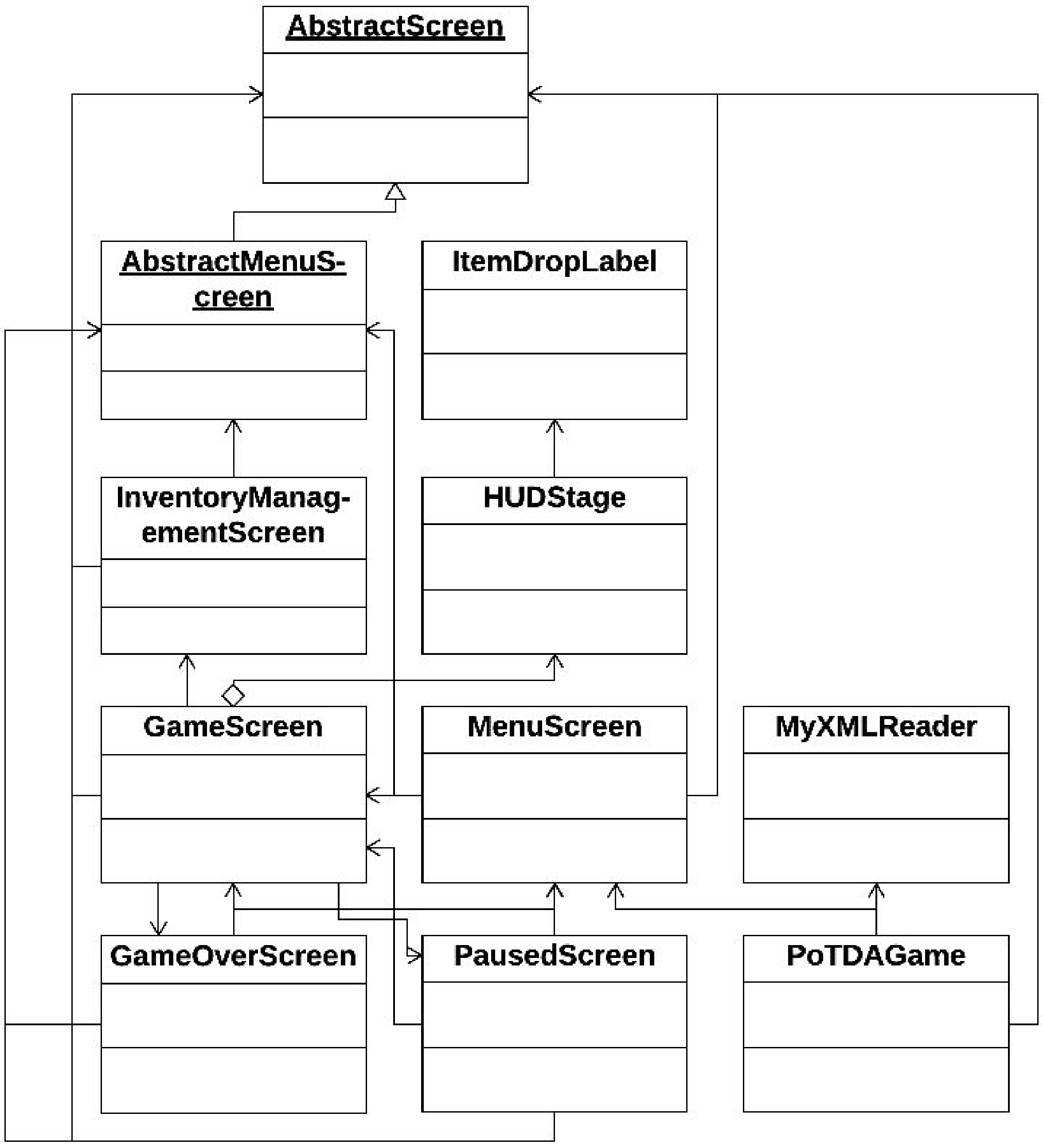
This package hold the Constants class that contains constants used by multiple classes throughout the Program.

The Sprites enum is stored here, it keeps relative paths to all sprites used in the Program.

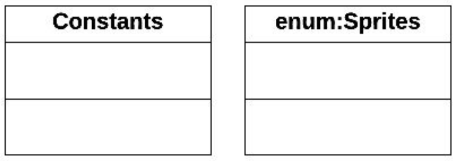
Appendix


application
PotDAGame (Top-level entry-point)
MenuScreen
GameScreen
InventoryManagementScreen
PauseScreen
GameOverScreen
physicsbox2d
PhysicsActor implementations
controllers
AbstractController
Various controller implementations for different control-methods and AI:s.
model
ModelActors, Builders
Items, Inventory
WaveController
Blueprints, XML representations
view
ViewActor
assets
Sprites (Simple enum representation)

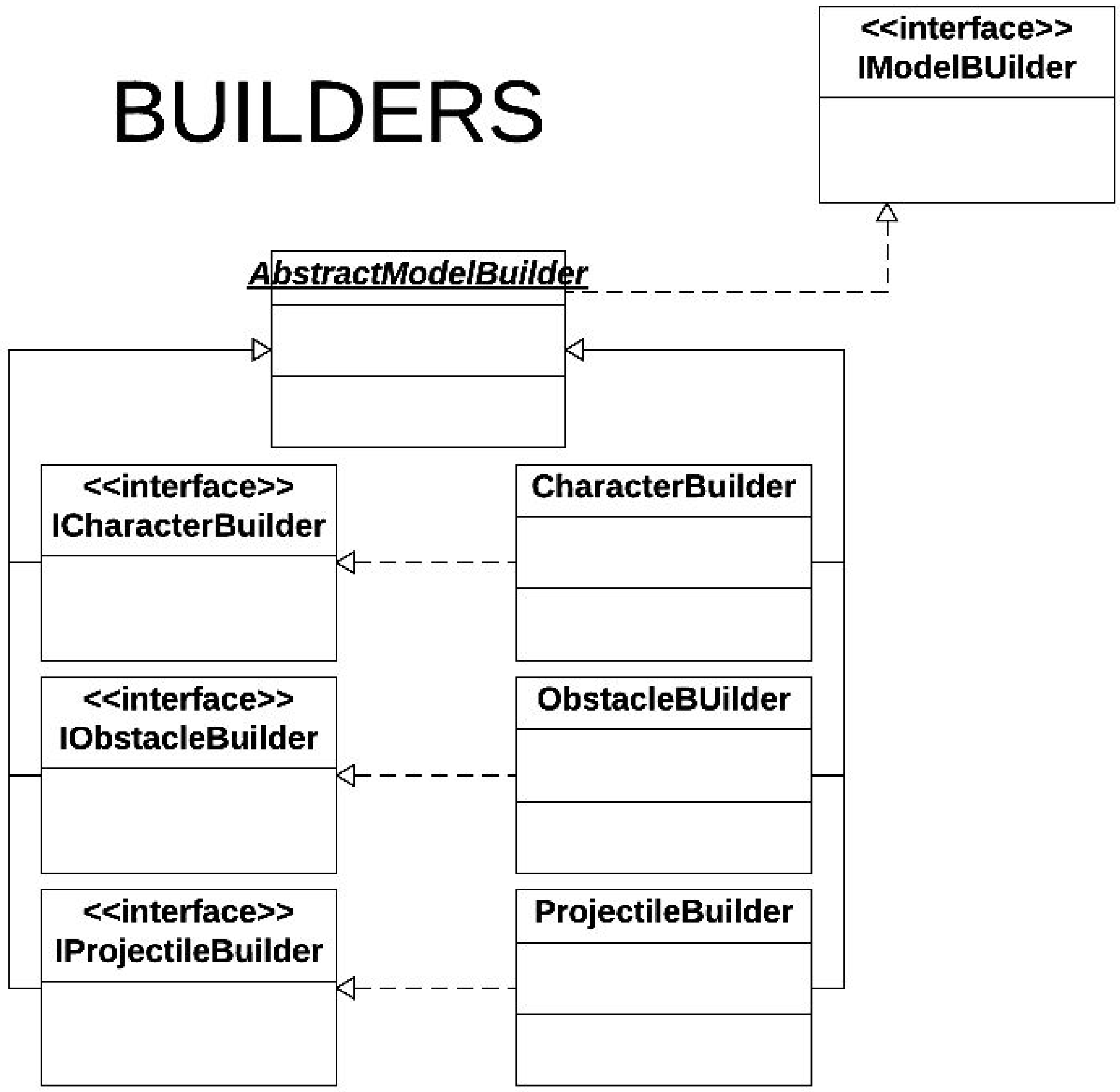
APPLICATION



ASSETS

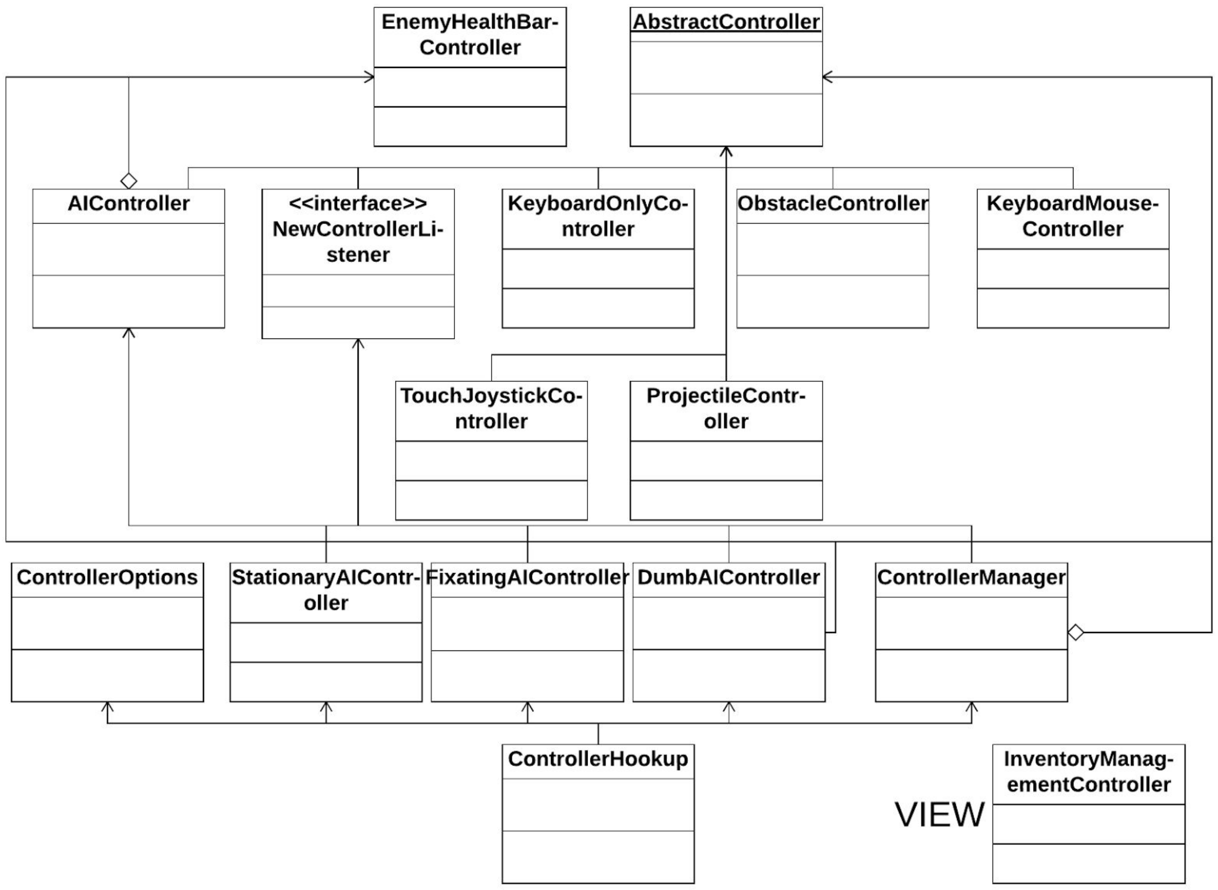


BUILDERS

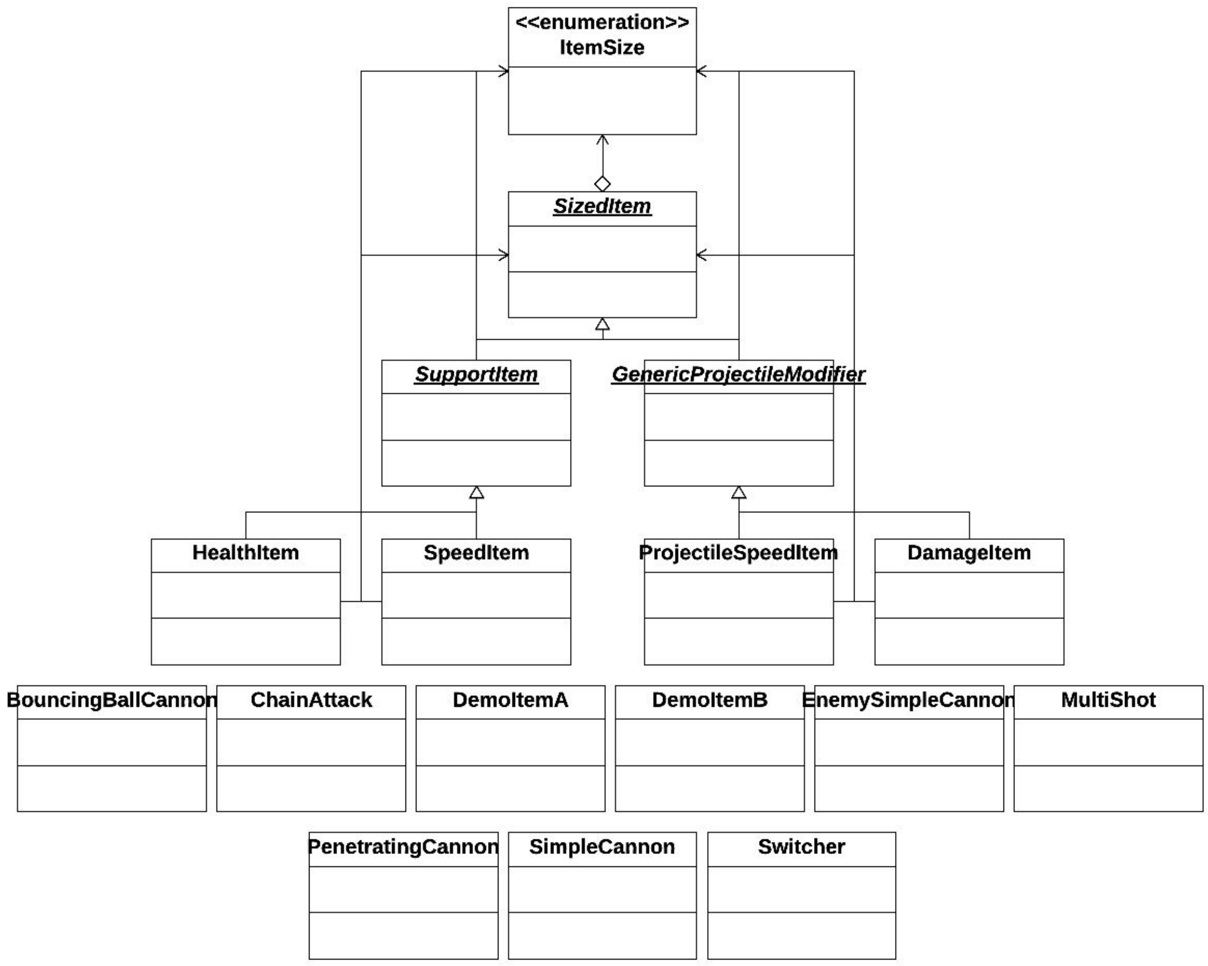


<<interface>>
IModelBUilder
AbstractModelBuilder
<<interface>>
ICharacterBuilder
CharacterBuilder
<<interface>>
IObstacleBuilder
ObstacleBUilder
<<interface>>
IProjectileBuilder
ProjectileBuilder

CONTROLLER

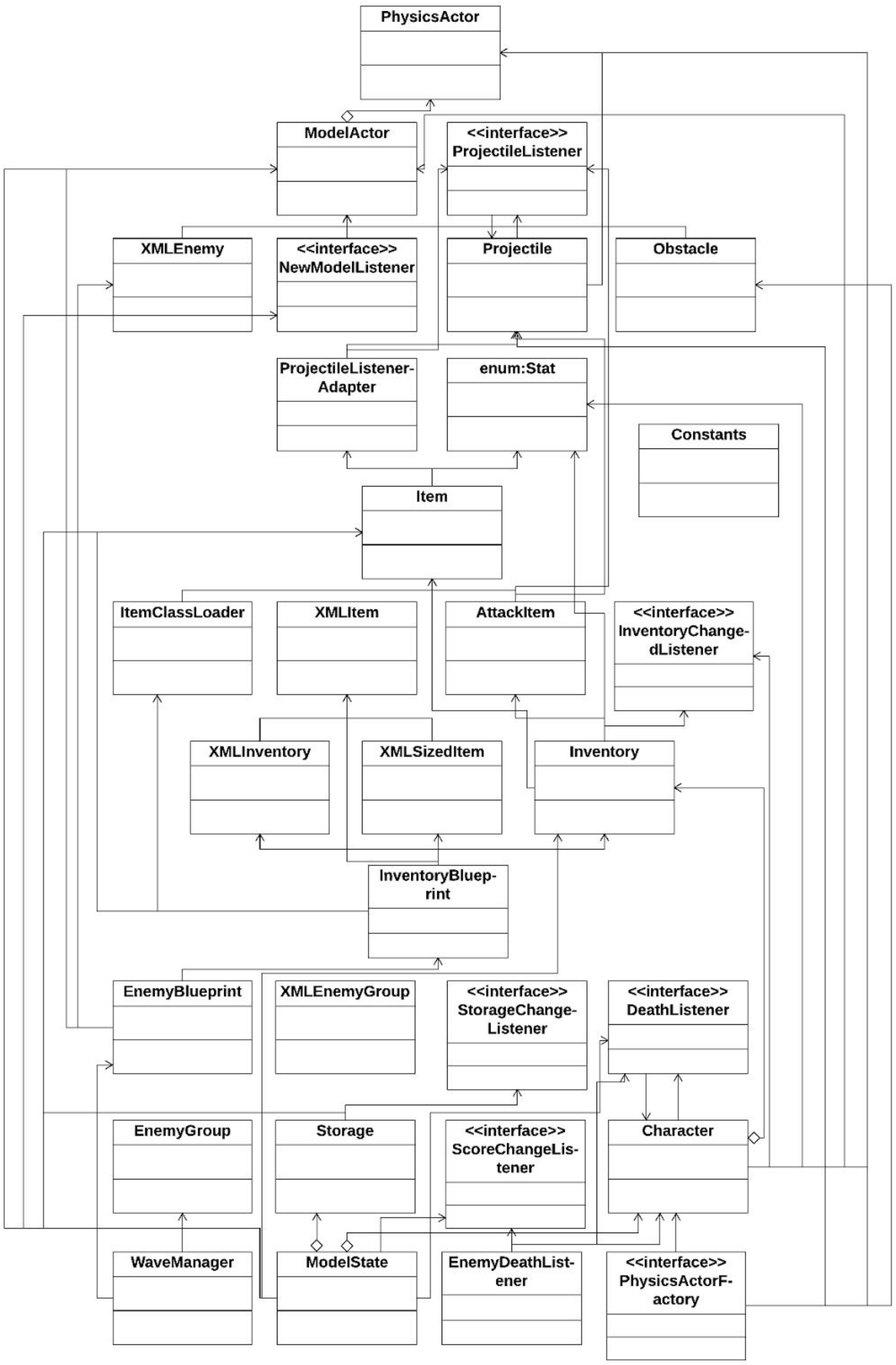

EnemyHealthBar-Controller
AbstractController
AIController
<<interface>> NewControllerLi-stener
KeyboardOnlyCo-ntroller
ObstacleController
KeyboardMouse-Controller
TouchJoystickCo-ntroller
ProjectileContr-oller
ControllerOptions
StationaryAIContr-oller
FixatingAIController
DumbAIController
ControllerManager
ControllerHookup
VIEW
InventoryManag-ementController

ITEMS

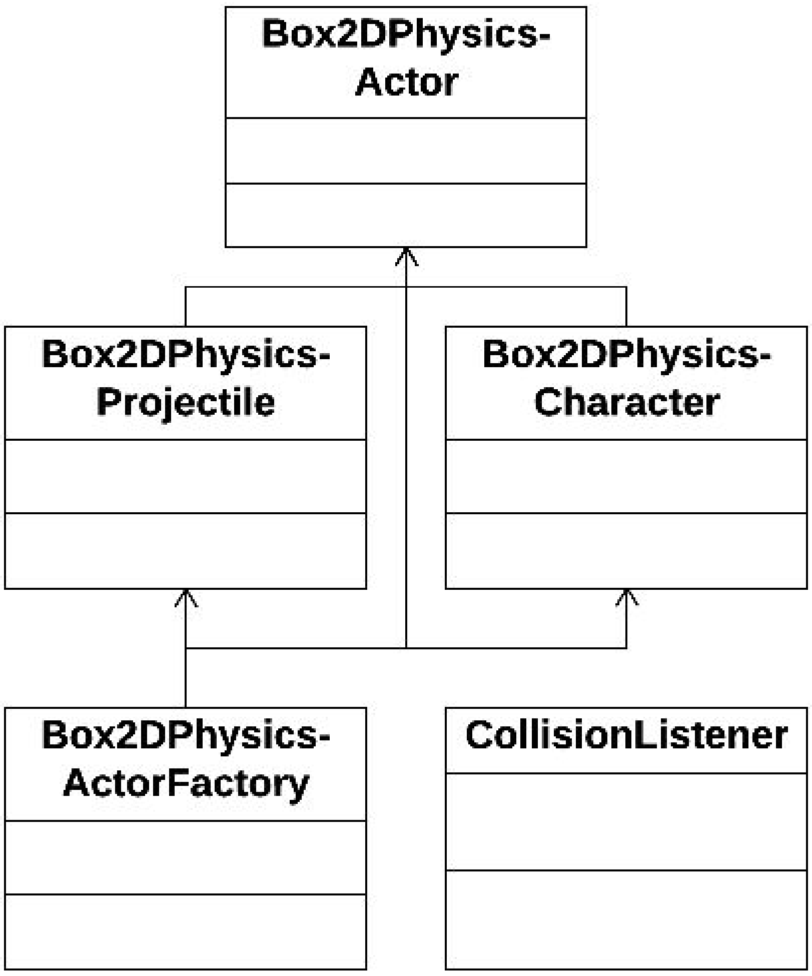


<<enumeration>>
ItemSize
SizedItem
SupportItem
GenericProjectileModifier
HealthItem
SpeedItem
ProjectileSpeedItem
DamageItem
BouncingBallCannon
ChainAttack
DemoItemA
DemoItemB
EnemySimpleCannon
MultiShot
PenetratingCannon
SimpleCannon
Switcher

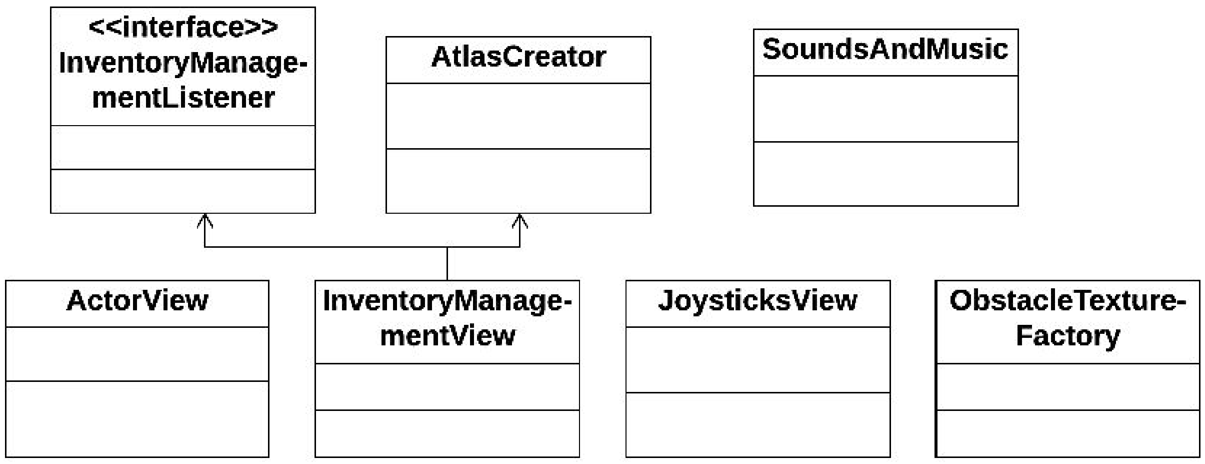
MODEL


PhysicsActor
ModelActor
<<interface>> ProjectileListener
XMLEnemy
<<interface>> NewModelListener
Projectile
Obstacle
ProjectileListener-Adapter
enum:Stat
Constants
Item
ItemClassLoader
XMLItem
AttackItem
<<interface>> InventoryChange-dListener
XMLInventory
XMLSizedItem
Inventory
InventoryBluep-rint
EnemyBlueprint
XMLEnemyGroup
<<interface>> StorageChange-Listener
<<interface>> DeathListener
EnemyGroup
Storage
<<interface>> ScoreChangeLis-tener
Character
WaveManager
ModelState
EnemyDeathList-ener
<<interface>> PhysicsActorF-actory

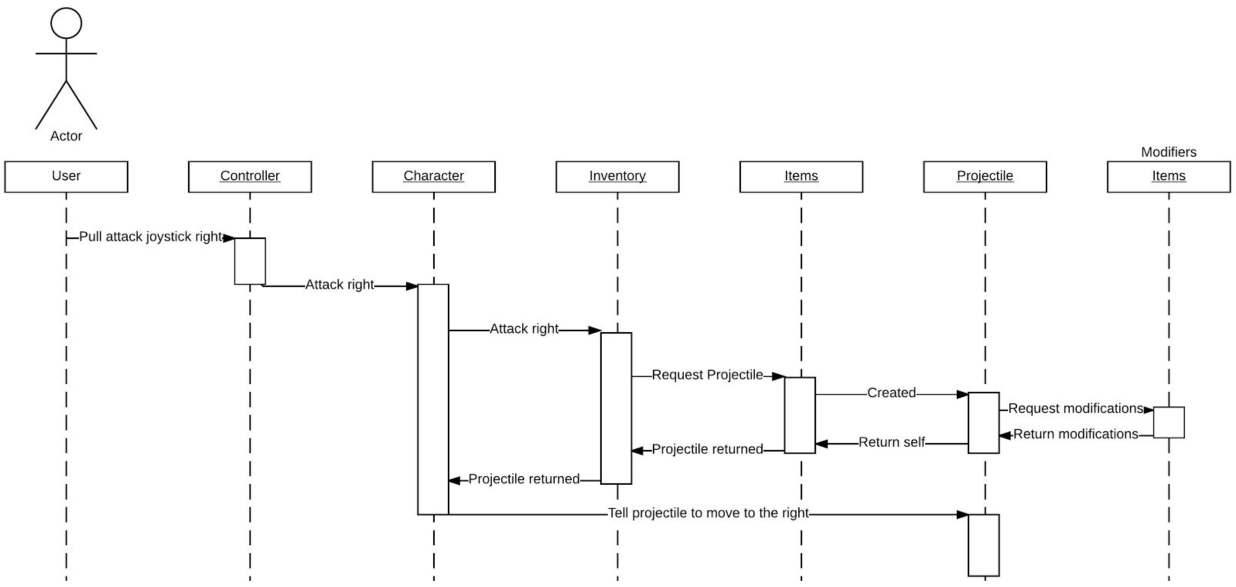
physicsBox2D



VIEW

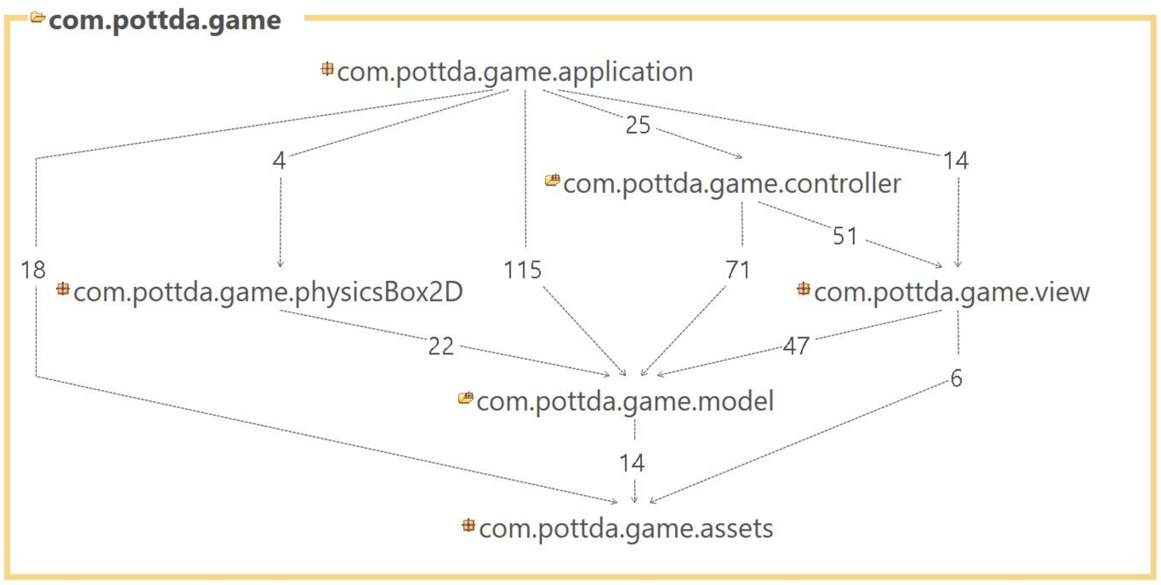


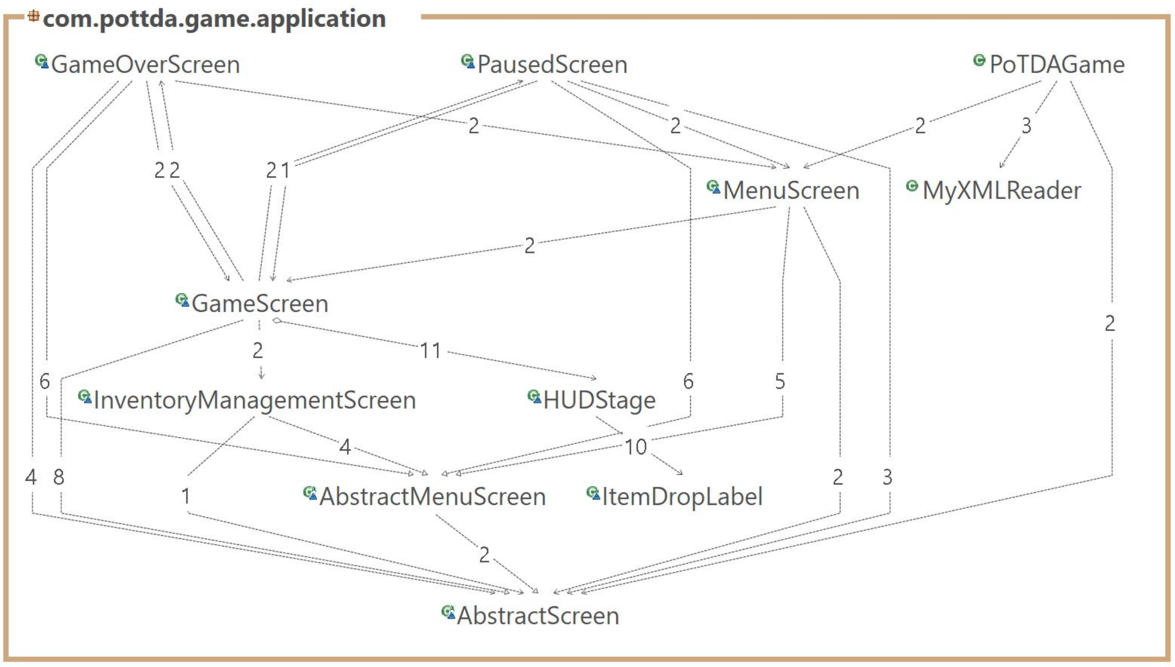
<<interface>> InventoryManage-mentListener
AtlasCreator
SoundsAndMusic
ActorView
InventoryManage-mentView
JoysticksView
ObstacleTexture-Factory

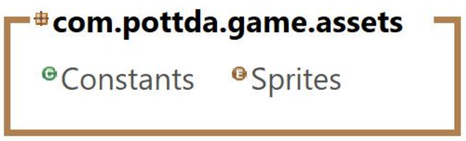


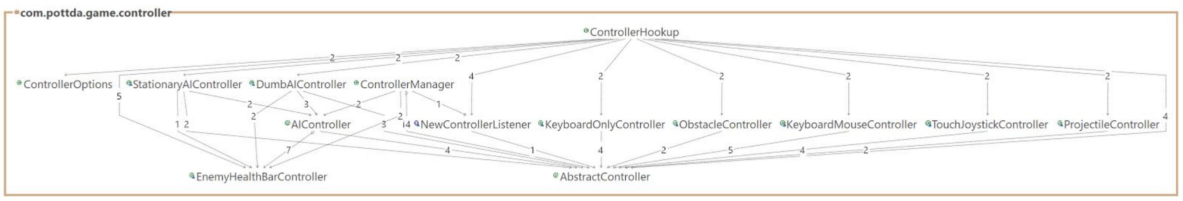
Actor
User
Controller
Character
Inventory
Items
Projectile
Modifiers
Items
Pull attack joystick right
Attack right
Attack right
Request Projectile
Created
Request modifications
Return modifications
Return self
Projectile returned
Projectile returned
Tell projectile to move to the right

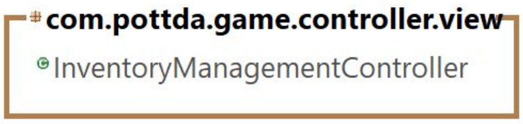
Package composition

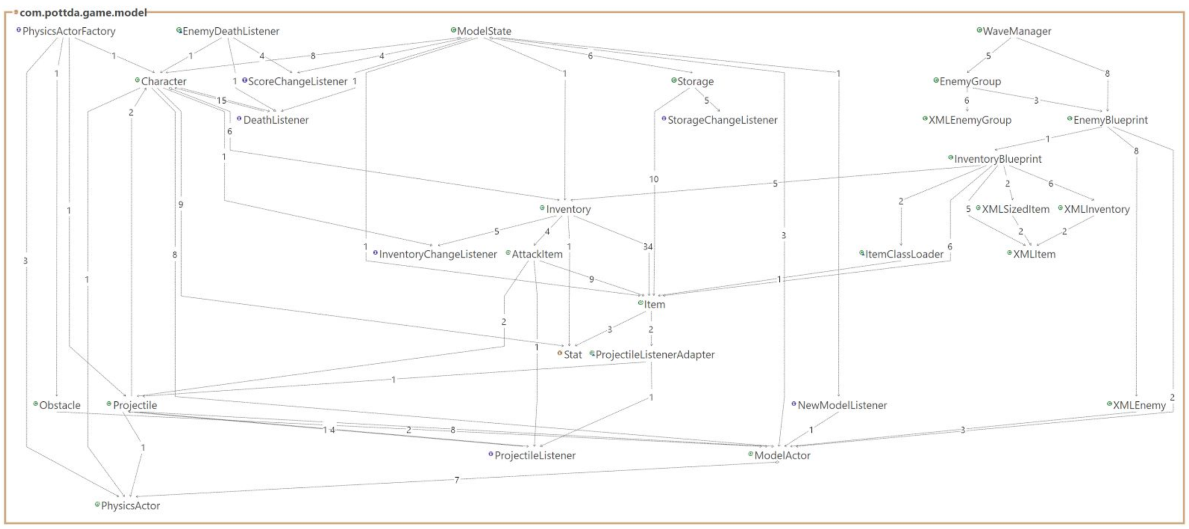


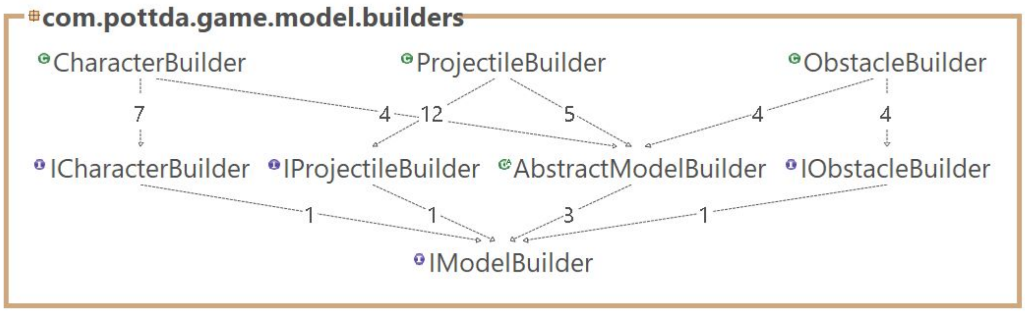


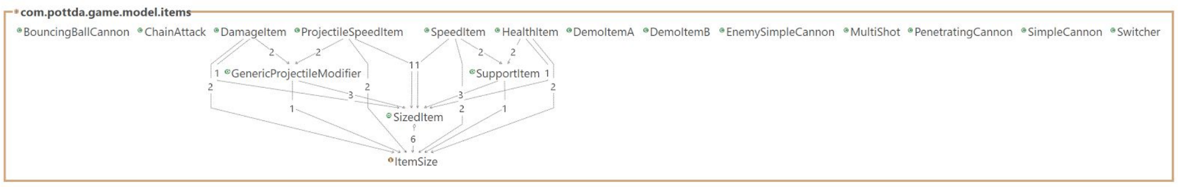


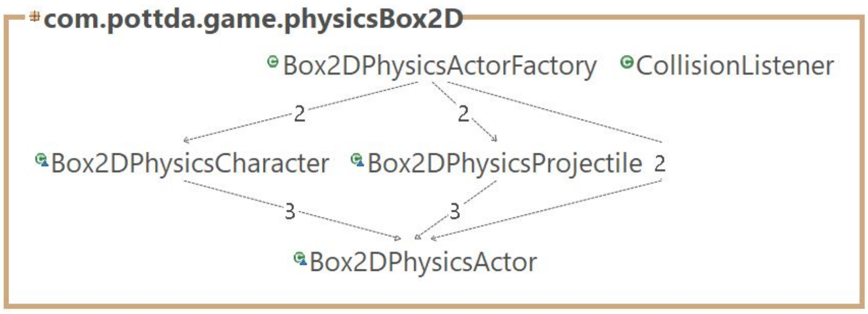


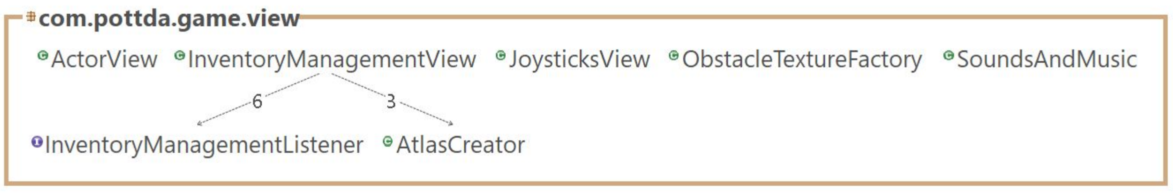

com.pottda.game.controller.view
InventoryManagementController


com.pottda.game.model
PhysicsActorFactory
EnemyDeathListener
ModelState
WaveManager
Character
ScoreChangeListener
Storage
EnemyGroup
DeathListener
StorageChangeListener
XMLEnemyGroup
EnemyBlueprint
InventoryBlueprint
Inventory
XMLSizedItem
XMLInventory
InventoryChangeListener
AttackItem
ItemClassLoader
XMLItem
Item
Stat
ProjectileListenerAdapter
Obstacle
Projectile
NewModelListener
XMLEnemy
ProjectileListener
ModelActor
PhysicsActor


com.pottda.game.model.builders
CharacterBuilder
ProjectileBuilder
ObstacleBuilder
ICharacterBuilder
IProjectileBuilder
AbstractModelBuilder
IObstacleBuilder
IModelBuilder


com.pottda.game.model.items
BouncingBallCannon
ChainAttack
DamageItem
ProjectileSpeedItem
SpeedItem
HealthItem
DemoItemA
DemoItemB
EnemySimpleCannon
MultiShot
PenetratingCannon
SimpleCannon
Switcher
GenericProjectileModifier
SupportItem
SizedItem
ItemSize


com.pottda.game.physicsBox2D
Box2DPhysicsActorFactory
CollisionListener
Box2DPhysicsCharacter
Box2DPhysicsProjectile
Box2DPhysicsActor


com.pottda.game.view
ActorView
InventoryManagementView
JoysticksView
ObstacleTextureFactory
SoundsAndMusic
6
3
InventoryManagementListener
AtlasCreator