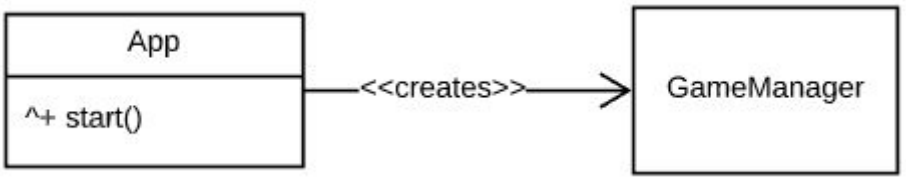
STICKMAN STAGE 3 REPORT - Codebase A

OOP DESIGN PRINCIPLE ANALYSIS

App Class



The start() method in the App class breaches the dependency inversion principle. The dependency inversion principle states that “High-level modules should not depend on low-modules. Both should depend on abstractions” (Martin, 2002, p.127).

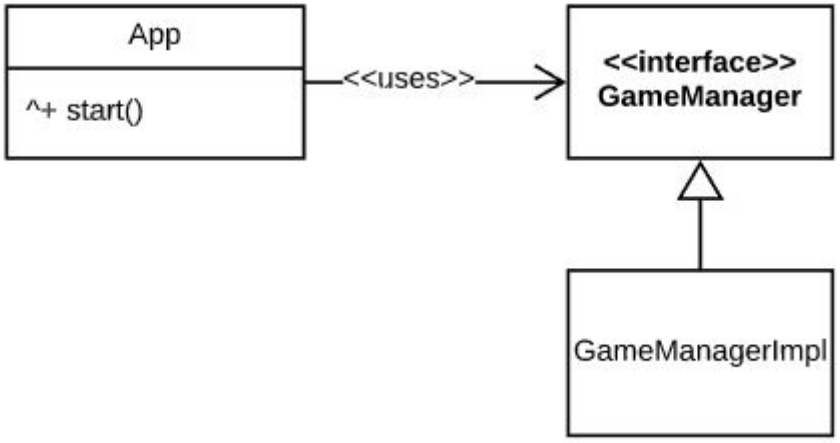
```
public void start(Stage primaryStage) {
      …
      GameManager window = new GameManager(model, 640, 400);
      window.run();
      …
}
```

The start method has a variable ‘window’ that holds a reference to a concrete class, GameManager. This means it depends on concretions rather than abstractions, hence it breaches dependency inversion principle.

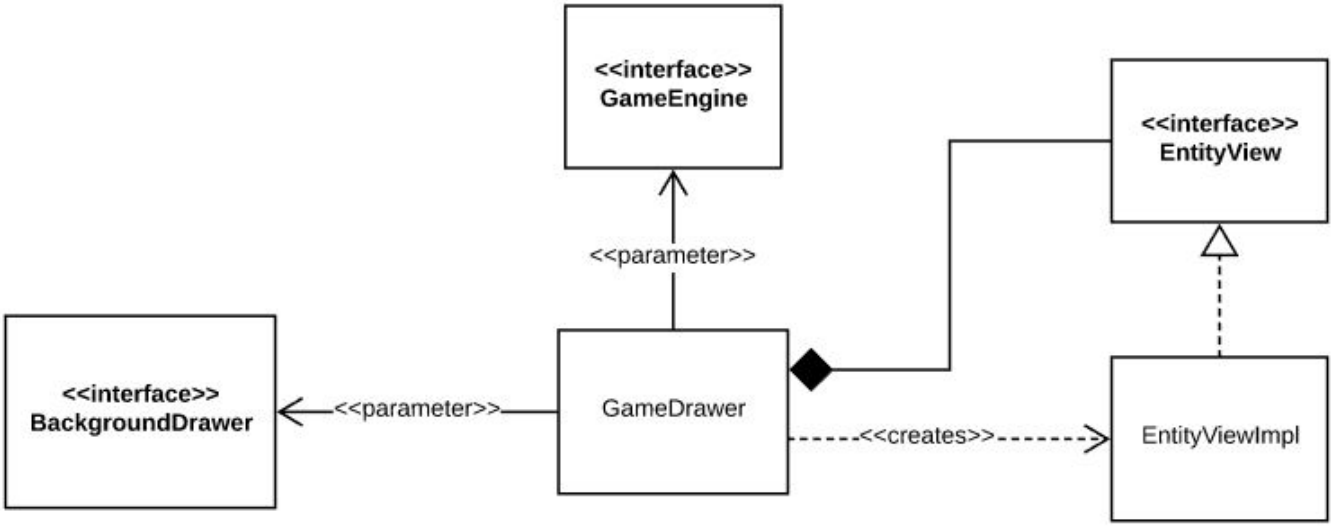
The impact of this breach is that the high level class, App, is coupled to a lower level class, GameManager. Any changes to GameManager has a direct effect on App making maintenance more difficult. It also breaches the open-closed principle because the behaviour of App cannot be extended with different GameManager classes since it depends on a concretion.

One way to fix this breach would be to add a GameManager interface and pass GameManagerImpl through App’s constructor. App now holds an association to GameManager instead. GameManagerImpl implements the GameManager interface. Now App can be reused with other GameManager implementations and new GameManager implementations can be created without modifying existing code and violating the open-closed principle. App is no longer coupled to a concrete class so changes to this class does not directly affect App.

```
public void start(Stage primaryStage, GameManager window) {
      …
      window.run();
      …
   }
                                        }
```



GameDrawer



The GameDrawer class breaches the single responsibility principle. This principle states that “a class should only have one reason to change” (Martin, 2002, p.95). That is, only one particular change in requirements should force modification of the class.

In GameDrawer, the draw method() updates the model, checks the view offsets, deletes old frames, updates the background *and* creates new frames. This is not done through calling appropriate methods, but rather the actual logic behind these processes are implemented in GameDrawer. Changes to any related class such as EntityViewImpl or BackgroundDrawer could potentially force changes in GameDrawer. Therefore, it has multiple responsibilities and thus breaches single responsibility

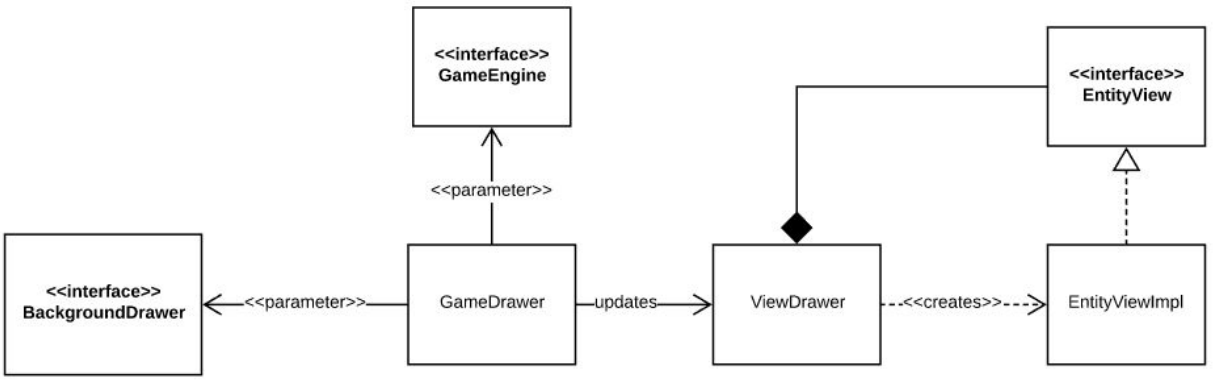
principle.

```
model.tick();
…
xViewportOffset += heroXPos - (width - VIEWPORT_MARGIN);
…
      if (entityView.isMarkedForDelete()) {
         pane.getChildren().remove(entityView.getNode());
       }
…
backgroundDrawer.update(xViewportOffset);
…
EntityView entityView = new EntityViewImpl(entity);
```

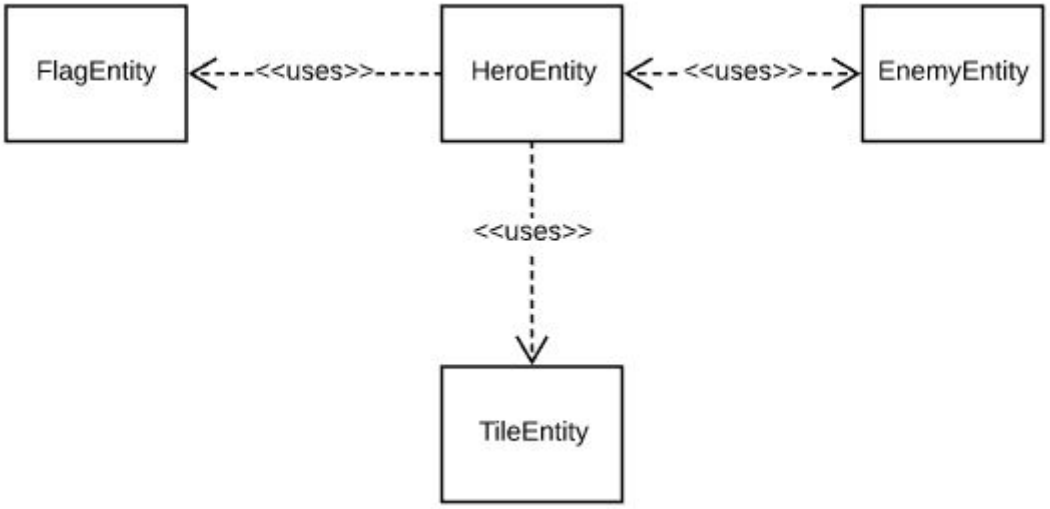
As a result of breaching single responsibility principle, GameDrawer is coupled to unrelated classes such as EntityView. Changes in any of these classes could potentially force changes in GameDrawer, such as a change in the parameters BackgroundDrawer.draw() takes. So unexpected bugs may appear in GameDrawer as a result of a seemingly unrelated change elsewhere.

To separate the responsibilities GameDrawer should call update() methods in appropriate helper classes instead of implementing the actual logic. Furthermore, it is possible that some BackgroundDrawer implementations have different methods of updating. For example, a background implementing a parallax effect would have different update logic to a simple scrolling background. Since only the background itself would know how it should update, the information expert principle states it should be responsible for updating itself.

```
public void draw(){
     model.tick();
     viewDrawer.update(model);
     backgroundDrawer.update(model);
}
```



Entity Collision Handling



The collision handling design shown in the HeroEntity and EnemyEntity classes breaches dependency inversion and open-closed principle.

```
public boolean handleCollision(Entity other, CollisionDirection
direction, Level level) {
    if (other instanceof EnemyEntity) {
     // some collision handling
    }
    if (other instanceof FlagEntity) {
     // some other collision handling
    }
    if (other instanceof TileEntity) {
     // some more collision handling
    }
  }
```

Dependency inversion states classes should depend on abstractions not concretions (Martin, 2002). The code snippet shown is taken from the handleCollision() method in HeroEntity. HeroEntity checks the type of each Entity that it collides with. It checks for concrete subtypes of Entity so HeroEntity is now dependent on concretions not abstractions, thus breaching dependency inversion.

Open-closed principle states a class should be “open for extension” while being “closed for modification”(Martin, 2002, p.100). Suppose that a new Entity was to be added to the level. Now another IF-statement is needed to handle that collision. So HeroEntity is not open for extension.

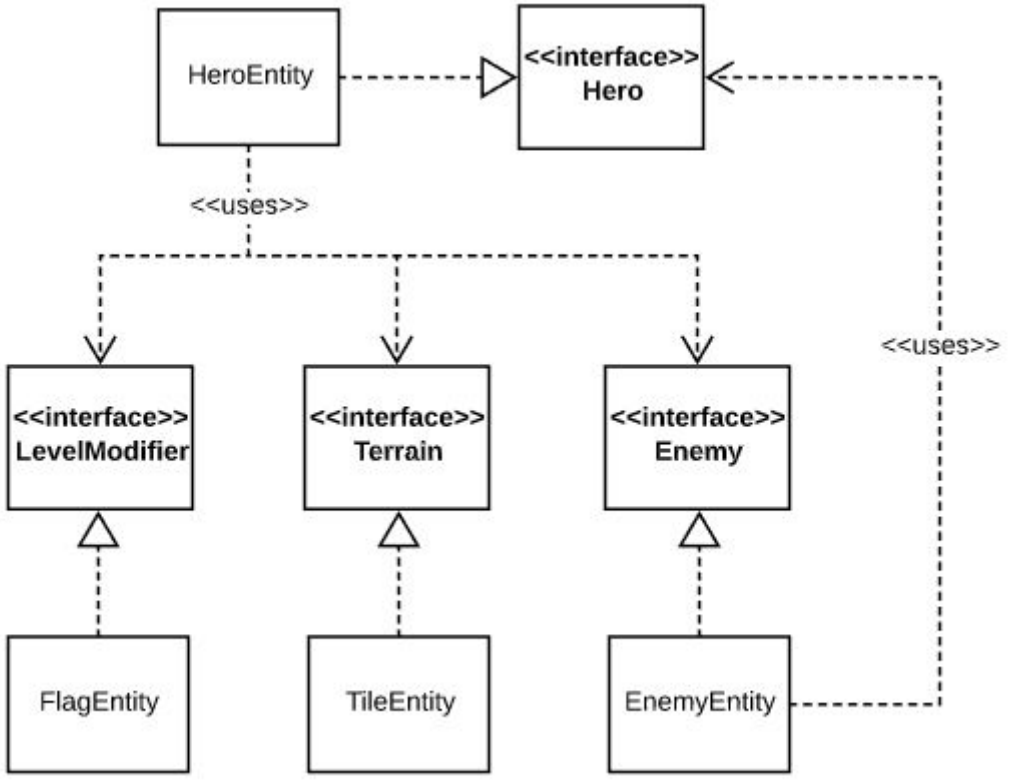
There is also a breach of information expert in EnemyEntity's die() method. Rather than calling a method in LevelImpl to remove itself from the list of entities, it gets the list by calling getEntities() then operates directly on the list. Information expert says LevelImpl should manage removal of an entity from the list, since it contains the necessary information, i.e the list, to perform this process. By getting the list and operating directly on it, EnemyEntity circumvents the encapsulation of data in LevelImpl.

One impact of these two breaches is HeroEntity is now coupled to every Entity subtype that it has to handle collisions with. From a maintenance and extension viewpoint, this requires retesting of HeroEntity every time another Entity, that it is coupled to, changes. If the game was extended to include many different terrain types such as walls and trees, HeroEntity could quickly be coupled to an unreasonable number of classes.

Another impact is the difficulty in extending the code. If another terrain Entity is to be added to the system, it is likely that the collision handling logic would be quite similar to that of a TileEntity. However, because the method depends on concretions, another IF-statement would be needed. This change would also need to be added to any Entity that collides with terrain. In this way, code is unnecessarily repeated leading to memory and computational power wastage.

One way to fix these breaches is to have handleCollision depend on interfaces, which are abstractions. Classify every collidable Entity under interfaces based on how they should behave in collisions such as Terrain, Enemy, Projectile and so forth. Now, unless a new type of interaction is needed that is not covered by existing interfaces, extensions can also be made without modification. E.g. RockEntity can be classified under Terrain so HeroEntity would not need to be modified.

```
public boolean handleCollision(Entity other, CollisionDirection
direction, Level level) {
    if (other instanceof Terrain) {
     // some collision handling
    }
    if (other instanceof Enemy) {
     // some other collision handling
    }
    if (other instanceof Projectile) {
     // some more collision handling
    }
  }
```



LevelImpl

The finish() method in LevelImpl() breaches single responsibility principle. The single responsibility states classes should only have one responsibility or "reason to change" (Martin, 2002).

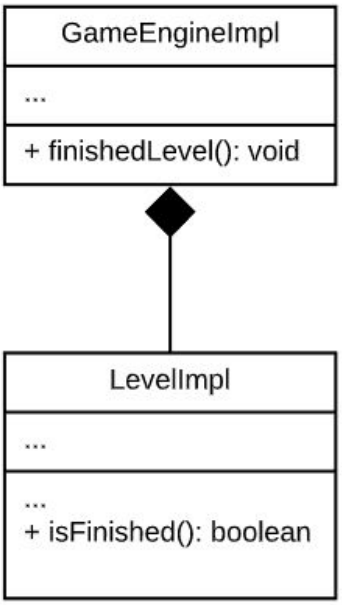
```
public void finish(String outcome) {
     …
    if ("DEATH".equals(outcome.toUpperCase())) {
      // what to do on death
    } else {
      // what to do in every other instance
    }

    System.exit(0);
  }
```

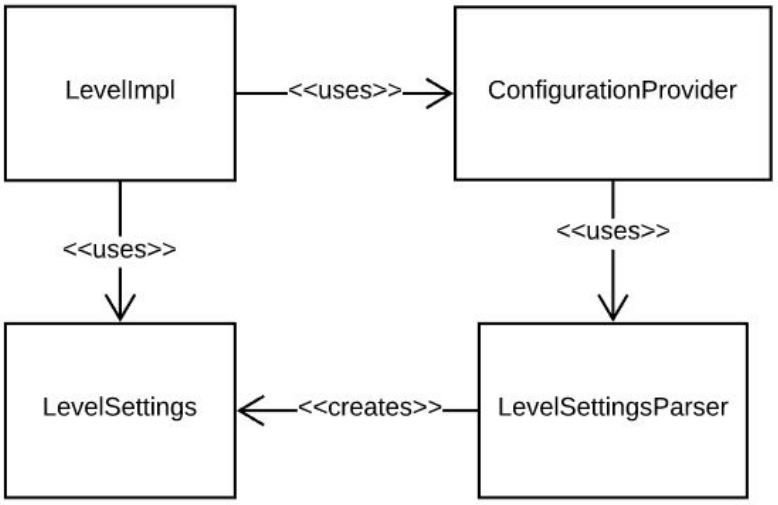
LevelImpl has the responsibility of managing entities in the game. In the finish() method it also has the responsibility of deciding what happens to the game when this particular level is finished. As a result, both changes to how entities are managed, *and* changes to what happens when a level finishes can force changes in LevelImpl, thus breaching single responsibility principle. This also violates the information expert principle. A level should not have the necessary game information, such as knowledge of other levels or end game menu, to make a decision on when to end the game so should not have responsibility for doing so.

The impact of this breach can be seen if requirements change around what happens when a level finishes. If the game should return to a start menu when the level finishes, then LevelImpl will need knowledge of higher level classes which indirectly couples LevelImpl to numerous other classes. It also allows LevelImpl to access methods and properties that are beyond the scope of its responsibilities. This makes maintenance more difficult since changes in LevelImpl could potentially introduce bugs in higher level classes and vice versa.

The simplest fix for this breach is to move the responsibility for handling a level finish to the GameEngine. Since the GameEngine already has responsibility for loading the level, it would remain cohesive for it to also handle finishing the level. The LevelImpl would simply have a method to let GameEngine know it has finished but lets GameEngine decide what should happen next.



Level

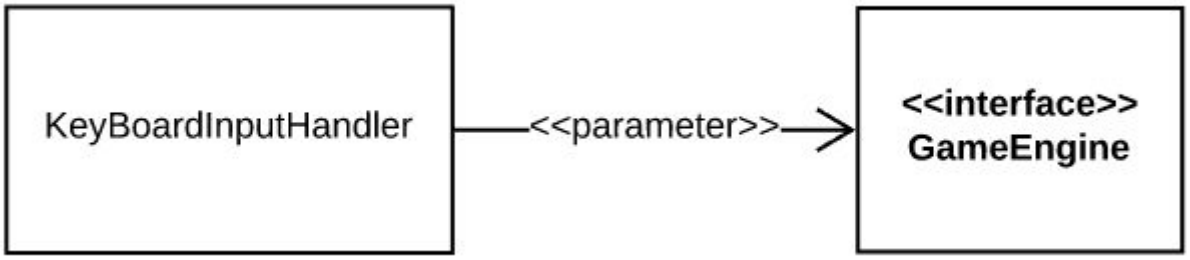


The design choices used to initialise the values of a level exemplifies good single responsibility design and high cohesion. Rather than having LevelImpl handle parsing the initialising values from the JSON file, the ConfigurationProvider uses a LevelSettingsParser to create a LevelSettings object which is then returned to LevelImpl to use for initialising.

LevelImpl simply gets the LevelSettings object from ConfigurationProvider and gets the initialising values. ConfigurationProvider instantiates a LevelSettingsParser instance to parse the JSON file. LevelSettingsParser reads the relevant values from the JSON file and creates a LevelSettings object containing those values. It would have been simpler, but not better design, to use the same code but have everything in ConfigurationProvider. As a result of following the single responsibility principle, the classes involved also exhibit high cohesion since each class is highly focused.

Using these principles have several benefits. The LevelImpl class is not bloated by having to parse the JSON file. This makes it more manageable and easier to maintain as there is less and more cohesive code. It also allows reuse of LevelImpl in situations where data does not need to be parsed from a JSON file. The ConfigurationProvider and LevelSettingsParser can all be reused for initialising other levels.

KeyboardInputHandler



The KeyBoardInputHandler breaches polymorphism, as well as dependency inversion and the single responsibility principle and the protected variations principle.

KeyBoardInputHandler handles inputs with a conditional IF-statement. A polymorphic method could be used since each condition would have similar execution. This breach is a result of KeyBoardInputHandler depending on hard coded behaviour within the handlePressed() method rather than abstracting the process, thus breaching dependency inversion principle. This also breaches the single responsibility principle as KeyBoardInputHandler should should not also be responsible for handling the logic for each key press.

The impact of these breaches can be seen with extensions. Consider later extensions that added 20+ key inputs which is not unreasonable. There would be 20+ conditions

to check. Every time a key is pressed, KeyBoardInputHandler could possibly go through every if condition which has a high performance cost. Furthermore any desired changes to keyboard input triggered processes will necessitate changes in KeyBoardInputHandler.

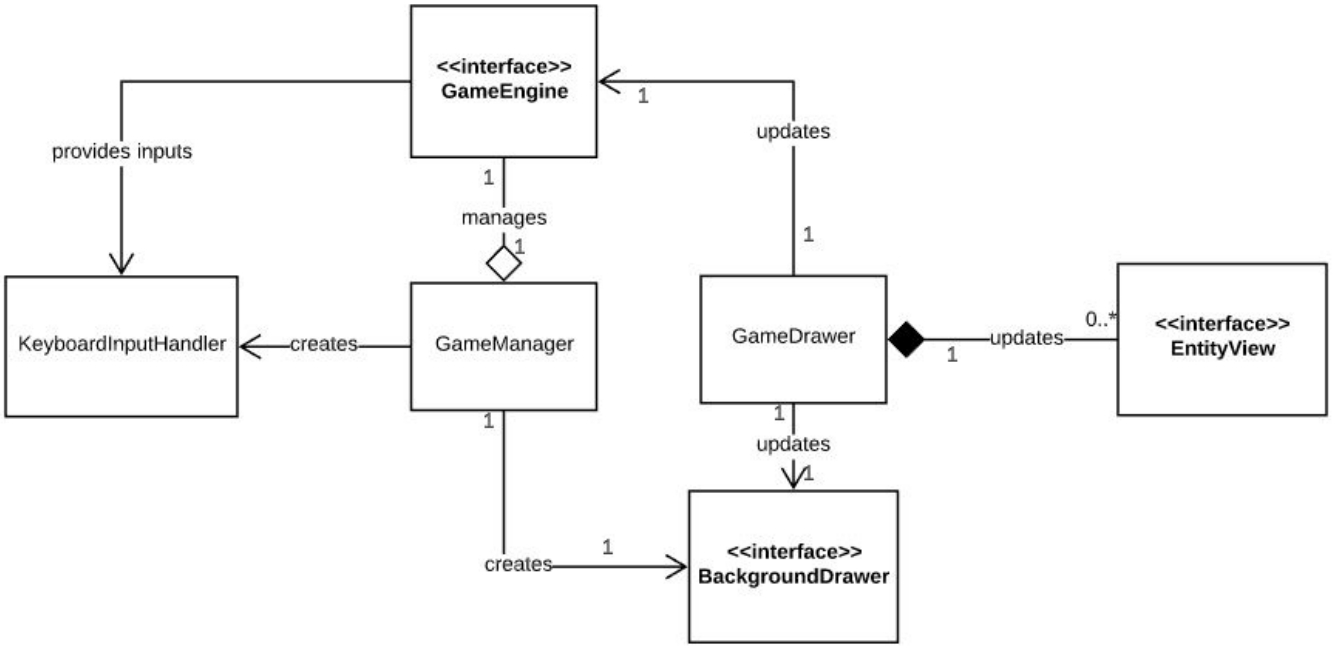
```
void handlePressed(KeyEvent keyEvent) {
     ...
    pressedKeys.add(keyEvent.getCode());
    if (keyEvent.getCode().equals(KeyCode.UP)) {
          //do something
    }
    if (keyEvent.getCode().equals(KeyCode.LEFT)) {
          //do something else
    } else if (keyEvent.getCode().equals(KeyCode.RIGHT)) {
          //do something else
    } else {
      return;
    }
    if (left) {
      if (right) {
        model.stopMoving();
      } else {
        model.moveLeft();
      }
    } else {
      model.moveRight();
    }
  }
```

A simple solution would be to use a functional interface. The logic for each keypress can be decoupled from the KeyboardInputHandler thus protecting it from variations. This also removes the responsibility of the logic implementation from KeyboardInputHandler. By using a map of keyEvents and functions, it also removes the need to have extensive IF-statements.

```
public interface KeypressAction() {
     void execute(GameEngine model);
}

void handlePressed(KeyEvent keyEvent) {
     if(actions.containsKey(keyEvent)){
          actions.get(keyEvent).execute(model);
     }
}
```

Separation of the model and view package with indirection

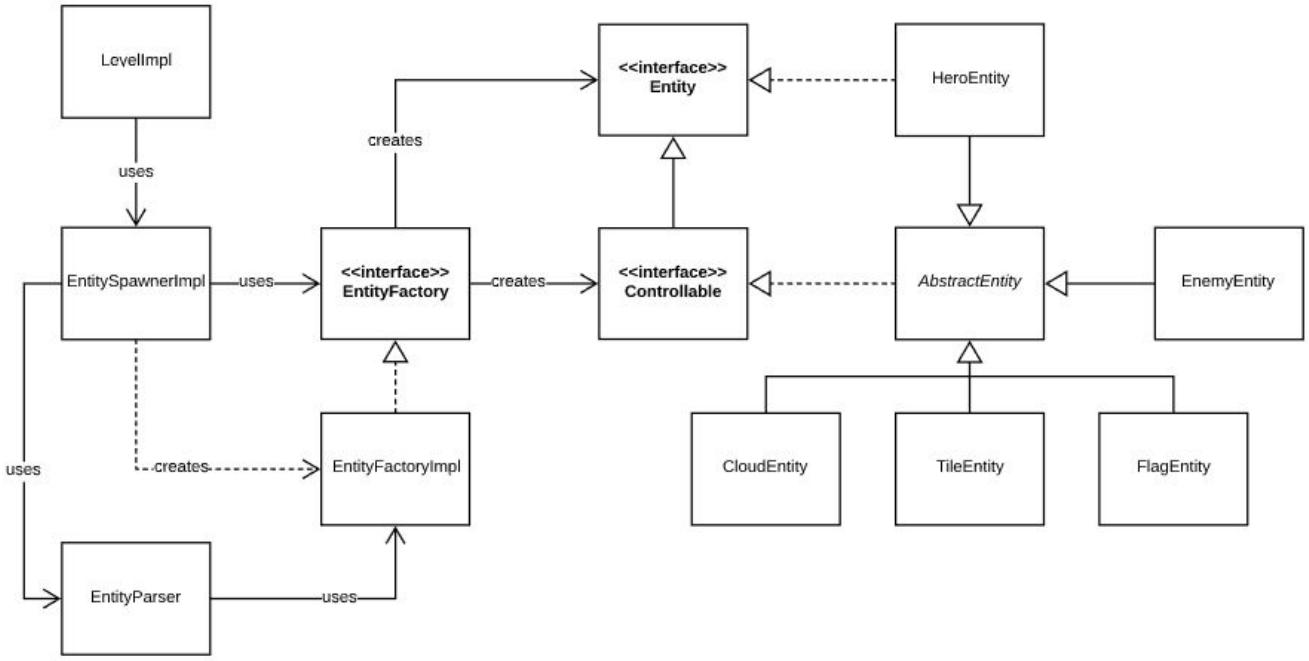


The general design of the system aims to lower coupling through indirection. The GameEngine class provides a mediating class between the model package and the view package of the system. GameEngine controls the initialisation, creation and functioning of the level and related classes, however none of those classes, except Entity, is coupled to the view package. GameManager, GameDrawer and KeyboardInputHandler are all coupled to GameEngine in order to pass necessary data through but not to any other class.

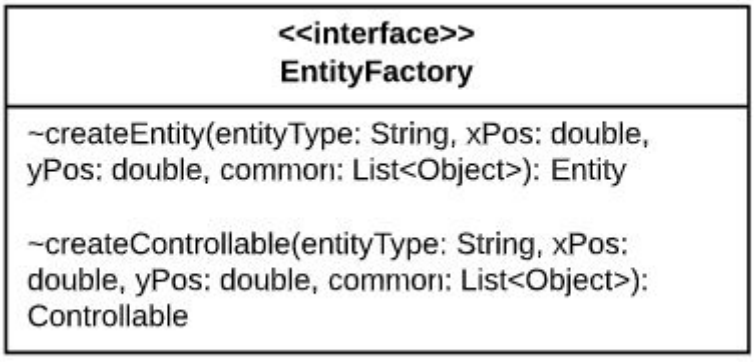
Lower coupling allows for reuse of both the model and view packages and supports protected variations principle by protecting the view package from variations of classes within the model package which improves maintainability and extensibility.

DESIGN PATTERN ANALYSIS

Factory Method



The code attempted to implement a factory method pattern with the EntityFactory interface and the EntityFactoryImpl class. However there are some flaws in the implementation of the pattern. Firstly, the factory pattern involves a class delegating the responsibility of creation to one of several helper classes, called factories. It is clear that the EntityFactory interface is designed for this role, and that EntityFactoryImpl is an implementation of this.



However, as seen in the above UML snippet, both EntitySpawnerImpl and EntityParser uses methods in EntityFactoryImpl to create entities. The following segment of code is from the EntitySpawnerImpl class which holds a reference to an EntityFactory and calls its method to create a TileEntity. Apart from this one instance, every other instance of creating an entity is delegated to EntityParser which supplies EntityFactory with the

necessary information to create Entity instances. This violates the single-responsibility principle as EntitySpawner has the responsibility for aggregating created entities to pass through to LevelImpl but also decides the parameters for TileEntity but not any other Entity subclasses. Consequently, it also reduces the cohesiveness of the class as it now has two separate responsibilities. It also unnecessarily couples EntitySpawnerImpl to EntityFactory. Simply by delegating the creation of TileEntity to EntityParserImpl, this can be avoided. It appears as though this decision was made because all instances of LevelImpl will have floor tiles along the length of the entire level but this violates the open-closed principle, as it prevents any future levels from not having this feature, e.g. an underwater level with no floor, without modification.

Another flaw in this design is that the EntityFactory interface has two separate methods for creation. The factory method the client should not know which subtypes of Entity it needs to create, and by having a separate method, it breaches this assumption. This violates the single responsibility of the factory to create just Entity types. A better solution would be to have a separate factory for creating Controllables.

```
// create floor tiles
for (int i = 0; i <= provider.getLevelData().getWidth() * 2; i += Constants.TILE_SIDE) {
  created.add(
      entityFactory.createEntity("TILE", i, provider.getLevelData().getFloorHeight(), null));
}
```

Secondly, a key characteristic of the full factory method is that the client depends on a Factory interface so that the specific implementation of the Factory can be decided at runtime and can be extended. In the EntityParser constructor, there is no parameter for passing an EntityFactory subclass. Instead it always instantiates an instance of EntityFactoryImpl. This breaches the open-closed principle of being open for extension, a key feature of the factory method.

```
public EntityParserImpl() {
  this.entityFactory = new EntityFactoryImpl();
}
```

Finally, EntitySpawner knows which type of entities to create as it reads the configuration file first. Yet it delegates the responsibility of parsing the configuration of each Entity to EntityParserImpl which then delegates the responsibility of actually creating the entities to EntityFactoryImpl. The factory method is for delegating the responsibility of creation to a class that know the type of entity to create and no further. So this also violates the information expert principle, which states that the class responsible for a function should be the one with the necessary information to carry out that function.

The impact of this deviation is slightly better cohesiveness since two classes process the JSON file and one creates it, but this could be achieved with just an EntityParser and an EntityFactory class. The drawbacks of this is unnecessary complexity and higher coupling which makes it more difficult to maintain and reuse classes. It also makes it significantly more difficult to extend the types of entities. Suppose friendly entities are added into the game. It would first need a new spawnFriendly() method in EntitySpawner, then a new case in EntityParserImpl and another new case in EntityFactory versus a correct implementation of factory method which would only require modification to the EntityFactoryImpl as EntityFactory interface, and consequently LevelImpl, would not know which Entity types exist.

```
spawnClouds(data, created, level);
spawnEnemies(data, created, level);
spawnPlatforms(data, created, level);
```

Despite some flaws in its implementation, the factory method used here is effective. By using an EntityFactory and other helper classes, the LevelImpl does not need to know which Entity subtypes it needs to create. This is the consequence of the factory method since it follows dependency inversion principle. The EntityFactory only returns Controllable and Entity types which are abstract types. This stops the LevelImpl from being coupled to every concrete Entity type and hence improves the maintainability and extensibility of code since changes to concrete Entity types won't affect LevelImpl.

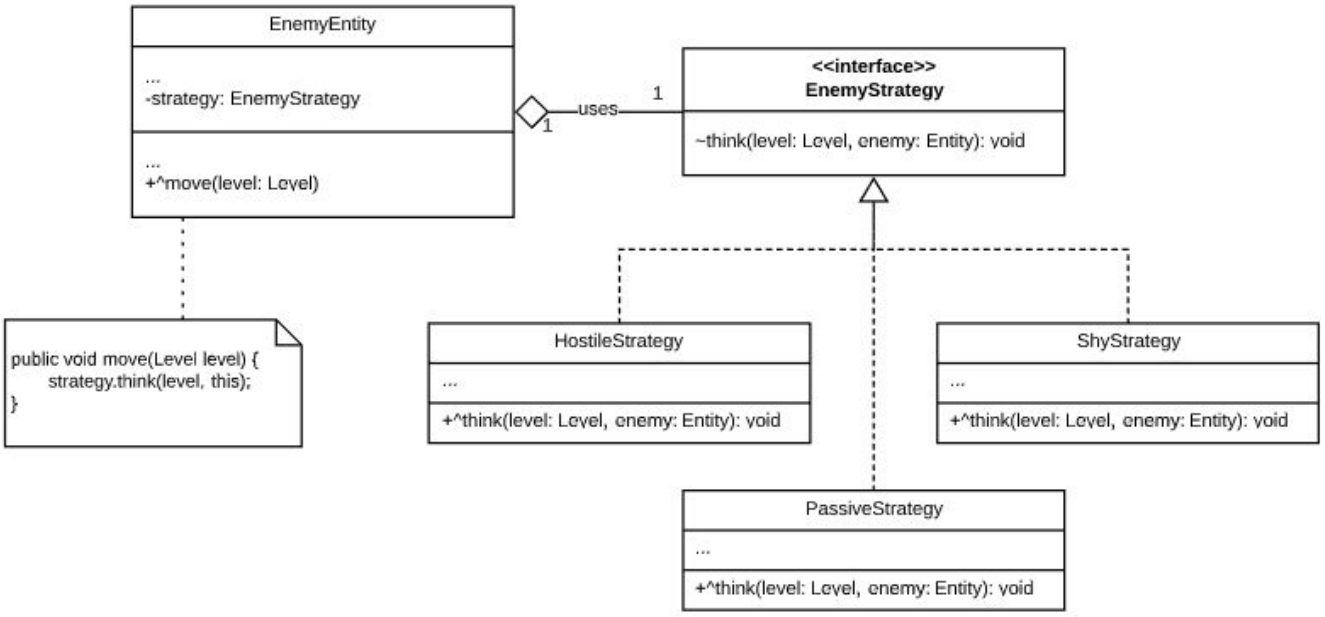
Using the factory method here also has the added advantage of improving class cohesiveness. If the factory method was not used, LevelImpl would have to read the data from the JSON file and then also create new Entity types based on that data. That would additional responsibilities on top of managing interactions within the level itself. This would violate the single responsibility principle as well.

This pattern also ensures that any objects that interact within the level all implements the Entity interface, and thus allows polymorphic behaviour. For example, the level can call move() and detectCollision on all Entity types without knowing the implementation of these methods or which specific subtype of Entity it is. However, the downside of this is that the LevelImpl class cannot call class specific actions on the Entity. This is probably why EntityFactory had two creator methods and thus allowing the level to treat one as a controllable 'hero' class instead of simply an Entity.

However, as stated above, the EntityParser always uses an EntityFactoryImpl class as its EntityFactory implementation. This prevents the LevelImpl from using different logic for Entity creation. For example, suppose it was desired that all Entity instances should start at least 100 pixels away from the Enemy. This could easily be done by swapping

in a new EntityFactory implementation but this is not possible with the current design. So it is not as effective as it could be.

Strategy Pattern



The EnemyStrategy interface and its concrete implementations are an example of a strategy design pattern. The strategy pattern is used correctly. According to Design Patterns Elements of Reusable Object-Oriented Software, the strategy pattern has a Strategy interface (EnemyStrategy), ConcreteStrategy classes (HostileStrategy, PassiveStrategy, and ShyStrategy) and a context (EnemyEntity) which maintains a reference to a concrete strategy and has varying behaviour according to the concrete strategy it uses.

The strategy design pattern is used to encapsulate algorithms that changes the behaviour of a context object (EnemyEntity) and is interchangeable. The EnemyEntity class implements a method called move() which calls the think() method in whatever strategy it holds a reference to. Since it interacts with concrete strategies like HostileStrategy and PassiveStrategy through the EnemyStrategy interface, these are interchangeable depending on which behaviour is desired to create hostile enemies, shy enemies and passive enemies. Furthermore, the data used in individual concrete EnemyStrategy implementations is hidden from Enemy. Therefore, this is a correct implementation of the strategy pattern.

```
@Override
Public void move(Level level) {
     strategy.think(level, this)
}
```

The use of the strategy pattern here is effective. The problem being solved is how to implement different behaviours into EnemyEntity to create different EnemyEntity types.

One possible alternative could be to have an AbstractEnemyEntity class and have ShyEnemyEntity, HostileEnemyEntity and PassiveEnemyEntity override the move() method. However this method hardcodes behaviour into concrete classes and does not allow for interchangeability. For example, perhaps later on it is desired to have an enemy that would switch from passive to hostile depending on whether the player has attacked it. The strategy pattern allows for this possibility. Using the abstract method would also create many different classes that differ only by its move() method which creates unnecessarily complex EnemyEntity classes for little benefit.

If all the strategies were kept within the EnemyEntity class, it would create a much more complex object that is difficult to maintain. It would also create unnecessarily large objects as some instances of EnemyEntity would likely not need to switch strategy. Furthermore, it would require conditional statements to select the desired behaviour which has a computational power cost.

Using the strategy pattern also follows the open-closed principle. Extension of further strategy is simply creating another class that implements EnemyStrategy and does not require much modification of existing code. Having separate strategy classes also allows for reusability. If different EnemyEntity types were introduced later on with more variations than just move(), these strategies could be used as well.

The effectiveness of the strategy pattern is diminished here because there are only three concrete implementations of EnemyStrategy. By using the strategy pattern, the system becomes more complicated and has more classes which requires more effort to maintain. A tradeoff that comes with the strategy pattern is the computational power overhead of passing the EnemyEntity and Level to all strategies. Some strategies may be simple and not need all the data passed to it, but regardless of which it will be passed. For example, the PassiveStrategy contains no behaviour implementation. If a conditional statement in EnemyStrategy was used instead, this would not even be a condition to check for. Instead a whole new class is created.

```
public class PassiveStrategy implements EnemyStrategy {
     @Override
     public void think(Level level, Entity enemy){}
}
```

DESIGN PATTERN ANALYSIS

There are some breaches of the style guide. In HeroEntity, some of the methods that are implementing the Controllable interface methods are missing the @Override annotation. The style guide used, Google java style guide, states that @Override is always used whenever legal, which it is in this case.

The corrections that need to be made is adding @Override after the javadocs for the methods implemented from Controllable. The impact of this breach is improper documentation that might lead to confusion over the origin of the method. Without the @Override annotation, one might come to the conclusion that HeroEntity declares these methods itself which would not allow polymorphism or abstraction. For example, a client might incorrectly deduce that a HeroEntity could not be allowed to be generalised as a Controllable and call getJumpStrength().

```
 /**
   * Returns whether or not the hero will move right at the next tick
   *
   * @return Whether the hero will move right
   */
  public boolean isMovingRight() {
    return this.movingRight;
  }
```

The overall code base has appropriate javadocs where dictated by the style guide. However, it lacks sufficient inline commenting. Especially in more complicated sections of code, inline commenting to explain why certain choices were made would be helpful in assisting understanding of the logic. For example, the draw() method in GameDrawer contains code that does many different processes. Not only is this a breach of single responsibility principle (as mentioned above) but the lack of inline commenting makes it even harder to follow what is happening, especially since the code is not split up into related sections with line breaks in between.

```
public void draw() {
     model.tick();
     List<Entity> entities = model.getCurrentLevel().getEntities();
     for (EntityView entityView : entityViews) {
           entityView.markForDelete();
     }
     // Some code updating viewport offsets
     // rest of the code updating entity view
}
```

OVERALL IMPACT ANALYSIS

Implementing the level transition was complicated by the breach of single responsibility in LevelImpl's finish() method. A level transition requires the current level to finish first, however since finish() is responsible for managing what happens after the level finishes, it is impossible to transition to another level without fixing the breach of single responsibility, or granting LevelImpl inappropriate access to GameEngine which stores the current level.

However, the separation of the level initialisation process into cohesive classes allowed reuse of ConfigurationProvider in creating new levels. This made it much simpler to create and initialise the next level in level transitions. If single responsibility was not followed here, the code for parsing the next level's configuration data would not be reusable.

For the implementation of the score, some way of tracking EnemyEntity deaths was necessitated. However, as mentioned above, EnemyEntity violated the encapsulation of LevelImpl's data in order to remove itself from the list of entities. As a result, any way of recording the death would have to occur in EnemyEntity but this would couple a score tracking class to a concrete low level implementation of Entity. This would also not be extensible since any new Entity deaths that need to be tracked would also require modification of their die() code to track it. As such, it was much more difficult to implement a robust solution to the second extension.

Having a separate LevelSettings parser also made it much easier to implement parsing the targetTIme from the JSON file. Since everything was already set up, it was a simple matter of adding a segment of code to also check for a targetTime value in the JSON file.

For the save and load feature, the design flaws of the KeyboardInputHandler class mentioned above meant there was no way of implementing this feature without continuing the violations of object oriented principles without refactoring the class. However, the separation of system into a view and model package with GameEngine acting as a controller between the two packages meant the keyboard inputs could be used to control save and load without having to be coupled to the lower level classes of the model package. Implementing this feature through dependence on the GameEngine interface rather than dependence on LevelImpl made it much simpler to follow design principles like open-closed principle and dependency inversion principle.

CODE REVIEW OF OWN CHANGES

LevelTransition

Description of feature:

A JSON configuration file contains an array of the filenames of all levels in the game. After each level finishes, the next level loads and the player would be prompted to start the next level. At the conclusion of the final level, a screen containing the winner text and some additional information is shown.

Description of extension work:

- A levels.json file storing the filenames of each of the levels in sequential order was added.
- A LevelsParser() class was added to parse the JSON file.
- An instance of LevelsParser was added to GameEngineImpl.
- App was modified to instantiate GameEngineImpl with “levels.json” instead of “config.json”.
- A loadNextLevel() and hasNextLevel() method was added to the GameEngine interface and GameEngineImpl.
- Victory and finished booleans were added to LevelImpl
- Changed finish() in LevelImpl to simply set victory and finished booleans.
- Added isFinished() and isVictory() to Level and LevelImpl
- Changed tick() method in GameEngineImpl to check if it needs to load next level
- Made timeline in GameManager a private attribute
- Added a pause() and play() method in GameManager that pauses and plays the timeline respectively.
- Added a UIDrawer interface and StandardUI class implementing this interface in the view package to draw the UI.
- Added UIDrawer to GameDrawer and called UIDrawer.update() in GameDrawer.draw()
- Added a setStartTime() method to Level and LevelImpl

LevelsParser is used by GameEngineImpl to parse the JSON file and return a list of strings containing the level filenames. Every tick, GameEngineImpl checks if the current level is finished. If it is, and it finished in a victory, GameEngineImpl loads the next level in the levels list. At each transition, a window pops up confirming player is ready to start the next level. During this, the timeline is paused so timer does not start counting down. If the player dies, a death screen is shown using the StandardUI class. The same is true for a victory screen.

O-O design principles analysis:

Open-closed and code reusability principles was used in the design choice to use a separate configuration file storing all the levels. Open-closed principle states that

modules should be open for extension. A basic object-oriented concept is designing modular code that can be reused. By having a separate configuration file, level configurations can be reused simply by adding the level name twice to the file.

Single-responsibility and information expert principle was used when refactoring the finish() method in LevelImpl. The principle states that classes should only have one responsibility, and information expert states that classes should have a responsibility if it has the necessary information to fulfil it. Since managing level transitions would give LevelImpl multiple unrelated responsibilities, and GameEngineImpl is a higher level class that composes Level instances, the responsibility was moved to GameEngineImpl.

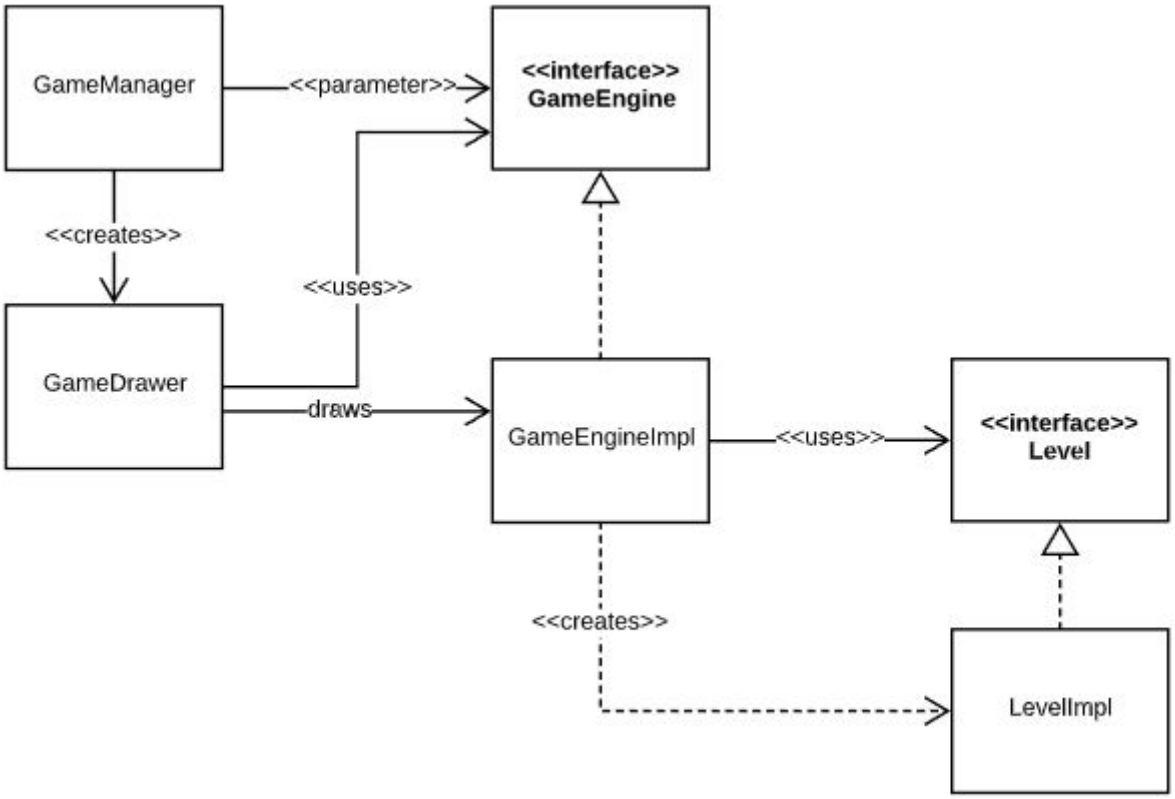
The impact of this is more robust code as LevelImpl is not dependent on other Level implementations which it would be if it managed transitions. It also allows reuse of LevelImpl for other level configurations which would not have been possible if LevelImpl had to transition to another level. The last level would not transition this would have to be checked with a condition or another class needed. The code is also more cohesive since LevelImpl focuses only on level responsibilities and GameEngine manages levels.

Single responsibility principle prompted the creation of a separate LevelsParser class. Instead of putting the code for parsing the values from levels.json in GameEngine, a separate class was created for this purpose to maintain single responsibility in GameEngine. The impact of this is reusable LevelsParser, and a cohesive and more maintainable GameEngine class. Changes to how the sequence of levels is parsed would not affect GameEngine.

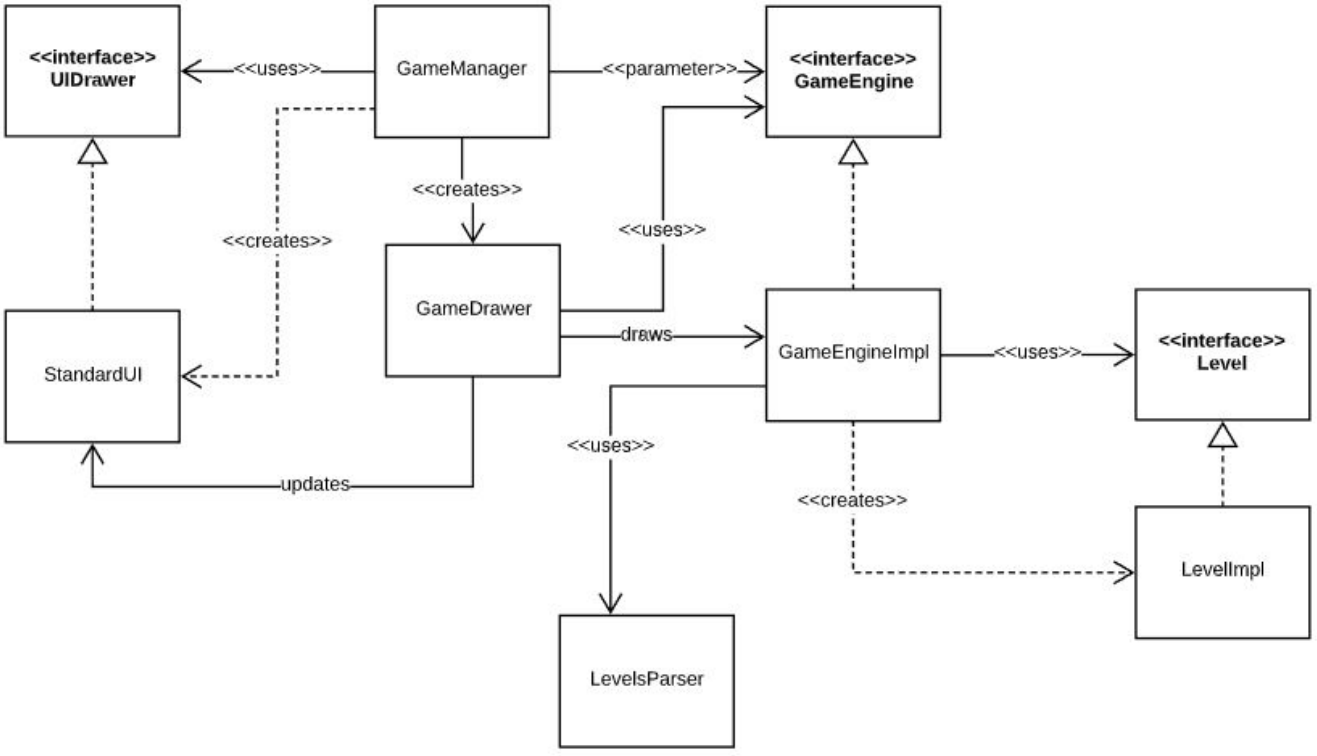
Open-closed and dependency inversion principle prompted the creation of the UIDrawer interface and StandardUI implementation. Open-closed principle states that classes should be open for extension but closed for modification. By having GameDrawer interact with an interface rather than a concrete class, extensions can be made simply by creating another implementation of UIDrawer. This also follows dependency inversion principle since GameDrawer depends on an abstraction rather than a concretion. The impact of this is extensions to the UI can be made easily and changes to concrete UIDrawer classes would not force changes in GameDrawer.

No design patterns were used here.

Before UML



After UML



Reflections on work:
Issues that exist are

- Death screens and level screens are simply overlaid on top of the current level.
- Level transitions prompted with a window instead of using the ingame UI

Improvements that could be made:

- Have a separate transition, death and victory screen.
- Create custom UI elements such as button to fit into the appearance of the game better

Things I would've done differently

- Added a separate transition, death and victory screen.

Things I wish I could've done

- Added custom UI elements

SCORE IMPLEMENTATION

Description of feature:
Levels have a target time in their configuration file. A score is shown on the screen starting at this time. The score counts down until it hits zero. At zero, it stops counting down but internally keeps decrementing to negative scores. When an enemy is killed, the score increases its internal counter by 100, so if time taken is more than 100 + target time, and an enemy is killed, score remains at 0. The total cumulative score for previous levels is also shown.

Description of extension:

- Added code to parse level targetTime in LevelSettingsParser
- Added targetTime to LevelSettings constructor
- Added getTargetTime() method to LevelSettings
- Added targetTime property to LevelImpl
- Added getTargetTime to Level interface and LevelImpl
- Added ScoreStrategy interface and TimedStrategy implementation
- Added parse level score strategy code to LevelSettingsParser
- Added scoreStrategy field to JSON configuration files
- Added scoreStrategy to LevelSettings constructor and as a field
- Added getScoreStrategy() method to LevelSettings
- Added scoreStrategy field to LevelImpl
- Added setScoreStrategy() to Level interface and LevelImpl
- Added a score update in tick() in LevelImpl
- Added getTotalScore() method in GameEngine and GameEngineImpl by storing level scores in a private field.

LevelSettings is constructed containing a ScoreStrategy implementation. LevelImpl takes that ScoreStrategy and sets its own field to that ScoreStrategy. The getScore()

method in LevelImpl calls the getScore() method from the ScoreStrategy. GameEngineImpl stores the completed level scores in a private field. StandardUI updates a label with the getScore() value of the current level every tick. It also calls getTotalScore() in GameEngine to display the total score so far.

O-O design principles analysis:
Single responsibility, open-closed principle and dependency inversion principle lead to implementing this feature using a strategy pattern. A LevelImpl should have responsibility for calculating its own scores, since the method for calculating score could change depending on clients. Open-closed principle lead to using an interface for LevelImpl to interact with. This way, different ways of calculating scores can be extended simply by adding another implementation of ScoreStrategy interface. Dependency inversion also pushed for LevelImpl to depend on an abstraction for calculating scores rather than a concretion.

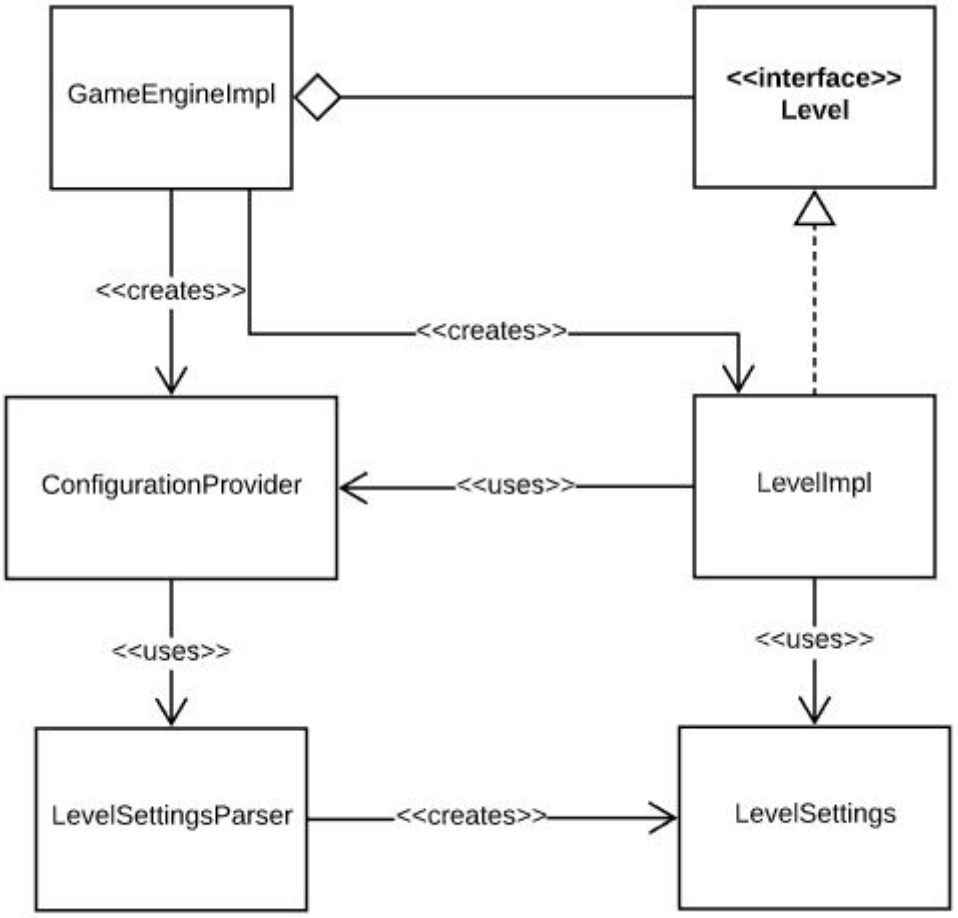
Patterns used - Strategy Pattern:
Correctness
I believe this pattern was implemented correctly. According to Gamma, Helm, Johnson, & Vlissides (1994), the strategy pattern aims to define a group of algorithms using a common interface, the ScoreStrategy interface, and make them interchangeable, which is true since LevelImpl references ScoreStrategy rather than a concrete class, and algorithms can change independently to the client, which is also true so LevelImpl is not affected by changes to any concrete strategy as long as it still implements ScoreStrategy correctly.
Effectiveness
This pattern was implemented effectively as it helps the system follow O-O design principles. As mentioned above, using ScoreStrategy helps LevelImpl maintain single responsibility. If LevelImpl implements the logic for calculating its own score, not only does it have an additional responsibility, changes to the score calculating requirements will force changes in LevelImpl. Furthermore, LevelImpl would not be reusable if it had concrete score calculation implementation. For example, if two levels were identical except one had enemies worth 100 points but another had enemies worth 200 points, then an entirely new Level implementation would be required for mostly the same code. This is memory inefficient.

Separating score calculation into a separate class is enough to fix single responsibility, but the dependency inversion issue of depending on concrete classes still exists. The aforementioned consequences are also relevant here. Using the ScoreStrategy interface avoids all these problems. One other method might've been to use conditional statements to check which score calculating strategy LevelImpl should use. However,

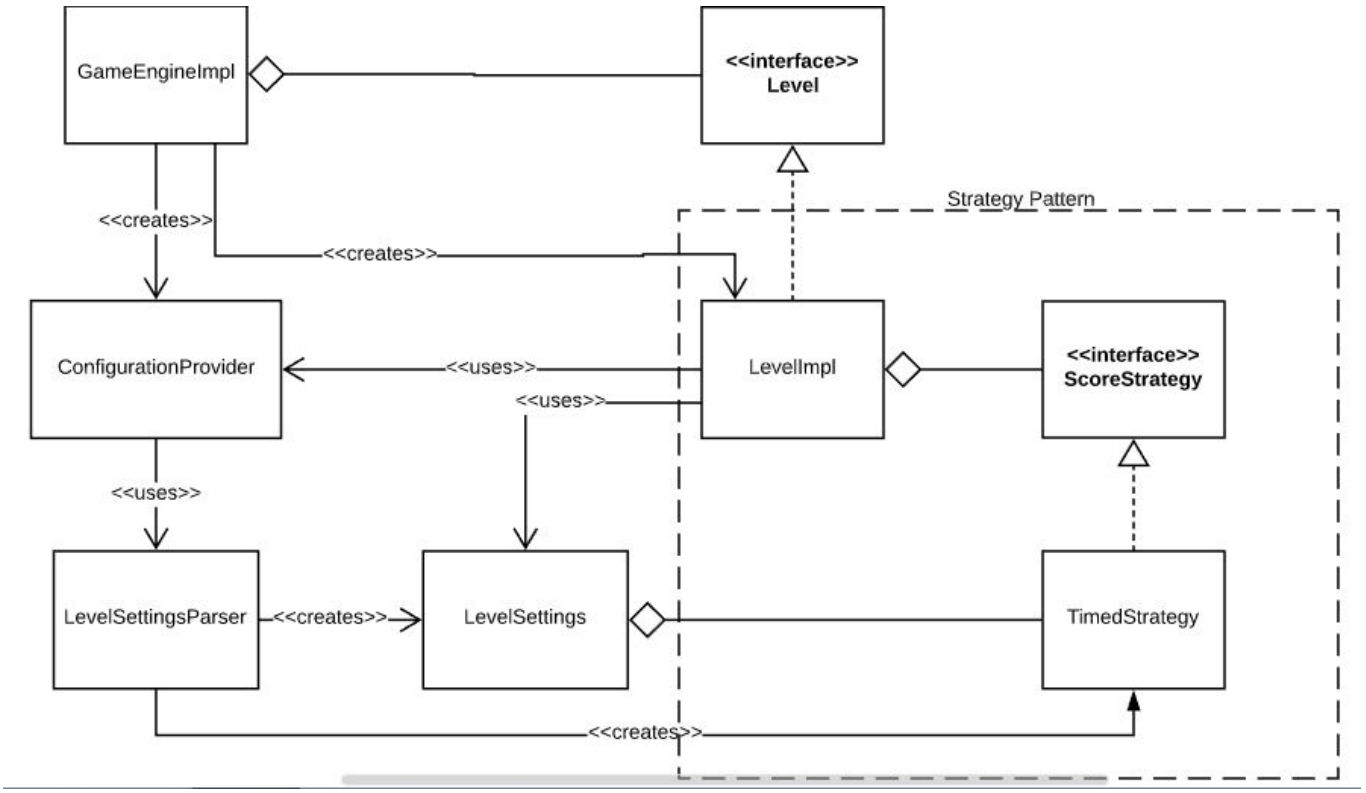
this bloats the class with code that it may never need to use. This approach would also breach open-closed principle as extension is not possible without modifying the conditional statement.

The disadvantages of a strategy pattern however includes a more complex architecture, increased number of objects and memory overhead for communicating between LevelImpl and ScoreStrategy. However, the benefits of following these design principles far outweigh the disadvantages which are insignificant in this scenario. So the strategy pattern used is effective.

Before UML



After UML



Issues that exist

- TimedStrategy calculates the number of dead entities by checking changes in the number of entities in LevelImpl. This won't work if entities get removed for other reason e.g entities are despawned.

Improvements that could be made

- Allowing a number of different score strategies so that total code can be reused. For example, having score count down based on time and having score increase based on enemy kills could be separated into two different strategies and the total score could be the combination of the two. This way, they could be used separately or reused elsewhere as well.

Things I would've done differently

- Use a combination of an observer and strategy pattern. Strategy for different ways of calculating the score as done but also have LevelImpl maintain a list of observers including ScoreStrategy that will be notified of changes. This would allow the aforementioned improvement to be implemented.

<u>Saving</u>

Description of feature

The player is able to save the game state at any time. Pressing the S key will save a game state. Pressing the Q key will load the saved game state. Only one save can be saved at a time. Saving again will overwrite the previous save. Scores, current level and current level state are all saved. If player was moving left or right when saving, the restored save will have player unmoving.

Description of extension work

- Added a Memento interface, and LevelMemento class that implements the interface
- Added a Caretaker interface, and GameCaretaker class that implements the interface.
- Added save() and restore() methods in Level and LevelImpl
- Added a Prototype interface to add copy() functionality
- Changed Entity interface to extend Prototype.
- Added save() and restore() methods to GameEngine and GameEngineImpl
- Changed keyboard input handling to include q and s handling
- Changed ScoreStrategy to implement Prototype

When S is pressed, the current level state is copied, including score and entity, and stored in a LevelMemento. At the same time, several fields in GameEngineImpl is also saved recording which level the game is up to, and the total score so far. The GameCaretaker stores the LevelMemento and handles passing the LevelMemento back to the LevelImpl. The LevelImpl then accesses the getters from LevelMemento to restore its own state.

O-O design principles analysis:
Single responsibility prompted a GameCaretaker that would only interact with the Memento interface. The Memento interface does not contain getters, hence GameCaretaker would never be able to access the stored state of LevelMemento. This was done because its sole responsibility is to store the Memento and restore the Level, not access the stored state. This also prevents GameCaretaker from being coupled to concrete classes which would be a breach of dependency inversion principle.

In order to create a Memento, the game state had to be copied. Rather than simply having every element add a copy() method, interface segregation and polymorphism was supported by adding a Prototype interface. The Prototype interface contained a copy() method. This meant only elements that needed to be copied to save the game state would implement the interface and copy() can be called on abstract Prototype instances rather than specific subtypes.

Open-closed principle is supported by the Caretaker interface. This allows for later extension, such as having three save files instead of one, without modifying existing caretakers. This would be done simply by implementing a new Caretaker class that stored Memento instances in an array instead of as a singular field.

Pattern Used - Memento:

Correctness

I believe the pattern was designed and implemented correctly. The Memento pattern is meant to “capture and externalise an object’s internal state so that the object can be restored to this state later”(Gamma, et al., 1994) without violating encapsulation. The LevelMemento stores the level properties, a copy of every Entity and a copy of the ScoreStrategy. The GameEngineImpl stores a copy of the current level progression in a private variable. So the implementation definitely stores the state correctly.

GameEngineImpl and LevelImpl also contain the appropriate methods for restoring its internal state. GameCaretaker passes the LevelMemento back to LevelImpl for restoration and GameEngineImpl handles its own restoration. Since GameCaretaker holds a reference to the Memento interface not the LevelMemento class, it does not have access to LevelMemento’s getters. Therefore it does not violate encapsulation potentially accessing LevelImpl’s internal state. Since GameEngineImpl and LevelImpl handles the actual restoration of their internal state, encapsulation is not breached here either. Thus this is a correct implementation of the Memento pattern.

Effectiveness

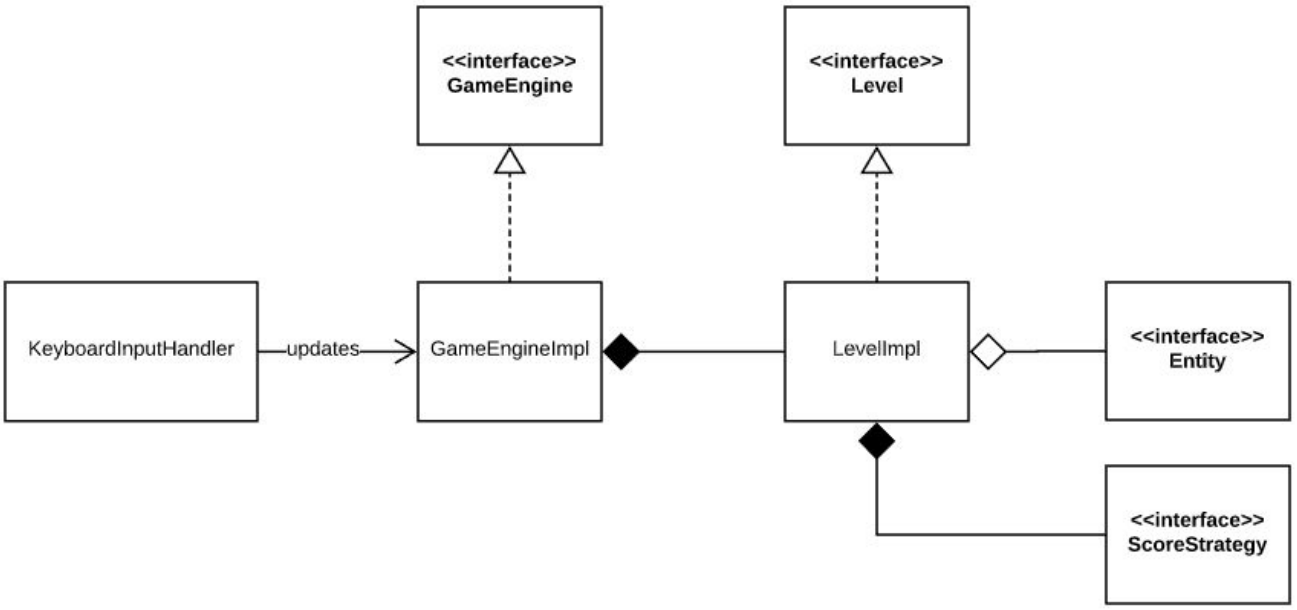
The issue with saving the internal state of LevelImpl is that it encapsulates all its data. Only its list of entities can be retrieved by outside classes. Exposing its data to create a copy would violate encapsulation which compromises the security and reliability of the class since external classes could change its internal state. Furthermore, not only does the internal state of LevelImpl need to be saved, the GameEngineImpl’s state, the Entity states and the ScoreStrategy state also needs to be saved. There needs to be a process for ensuring a deep copy of the relevant elements can be created.

The Memento pattern can be used here. As mentioned above, the implementation created of the Memento pattern does not violate encapsulation. Besides solving the aforementioned issues, the Memento pattern also has several benefits over alternative solutions. Other methods of saving the internal state would probably involve LevelImpl storing a copy of all its fields in another variable. For example, copiedEntities list would be a copy of the entities list. However, this bloats the LevelImpl class and could potentially cause memory related issues when every LevelImpl stores multiple saves. Furthermore, it overcomplicates LevelImpl which makes it harder to maintain.

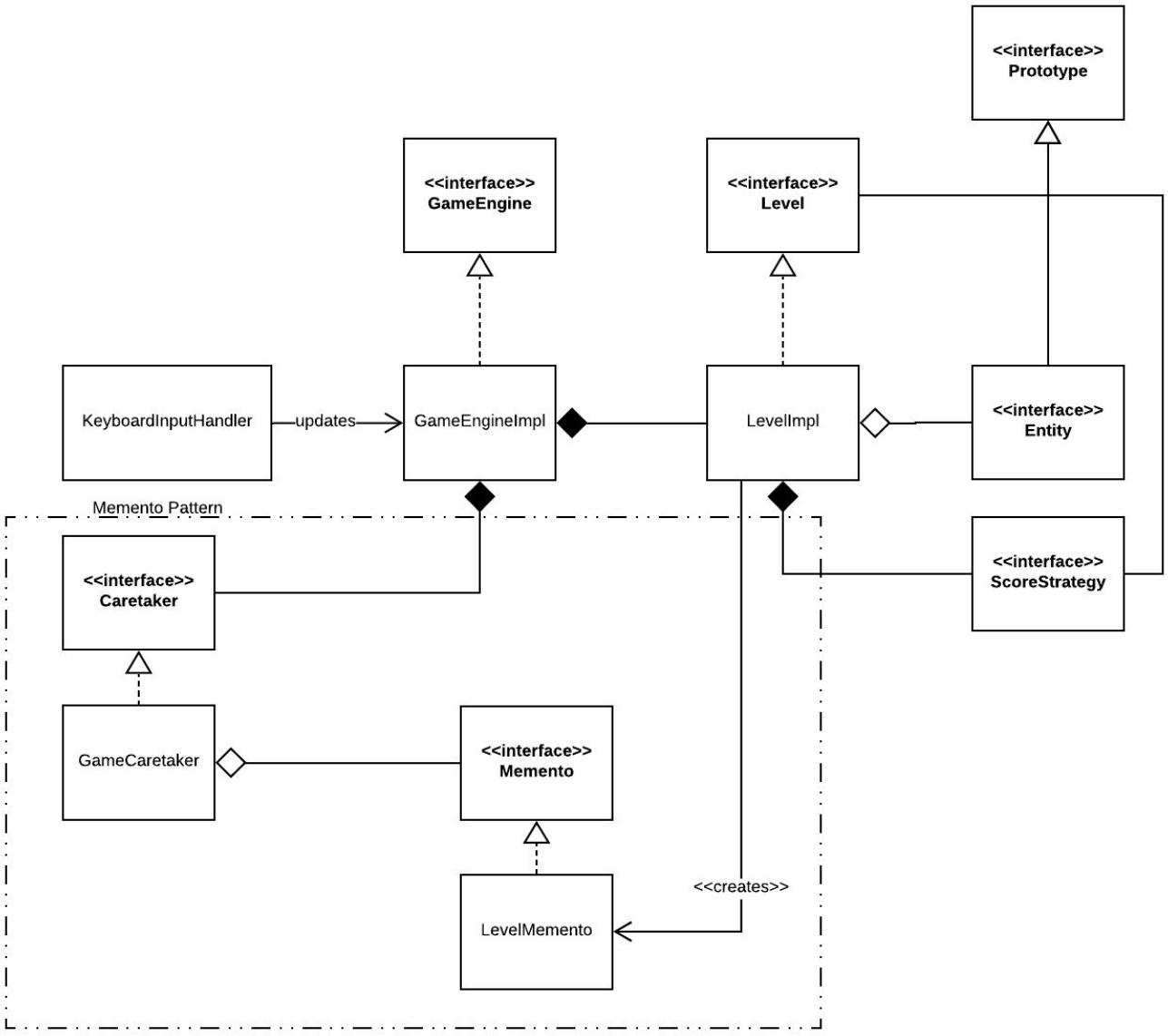
The implementation of the Memento pattern with a Caretaker class supports indirection and protected variations principle. The client, GameEngine, would never need to know about LevelMemento, hence protecting it from the instability of LevelMemento and the restoration process. This means changes to LevelMemento and how it is created does not affect GameEngine.

The disadvantage of Memento is the extra cost of copying the entire state of LevelImpl and moving it into a Caretaker class. However, the benefits of Memento in solving the aforementioned issues with saving, far outweighs the disadvantages, especially considering there is not a feasible alternative. Therefore Memento is implemented effectively here.

Before UML



After UML



Existing Issues

- GameEngineImpl's state is stored as a copy of its fields. This means if GameEngineImpl ends up with more fields, it will make the class much more bloated and hard to maintain if this method of saving is continued.
- Levels are loaded by taking the saved state and restoring it into the current level. This only works if it is the same LevelImpl.

Things to Improve

- The aforementioned issues.

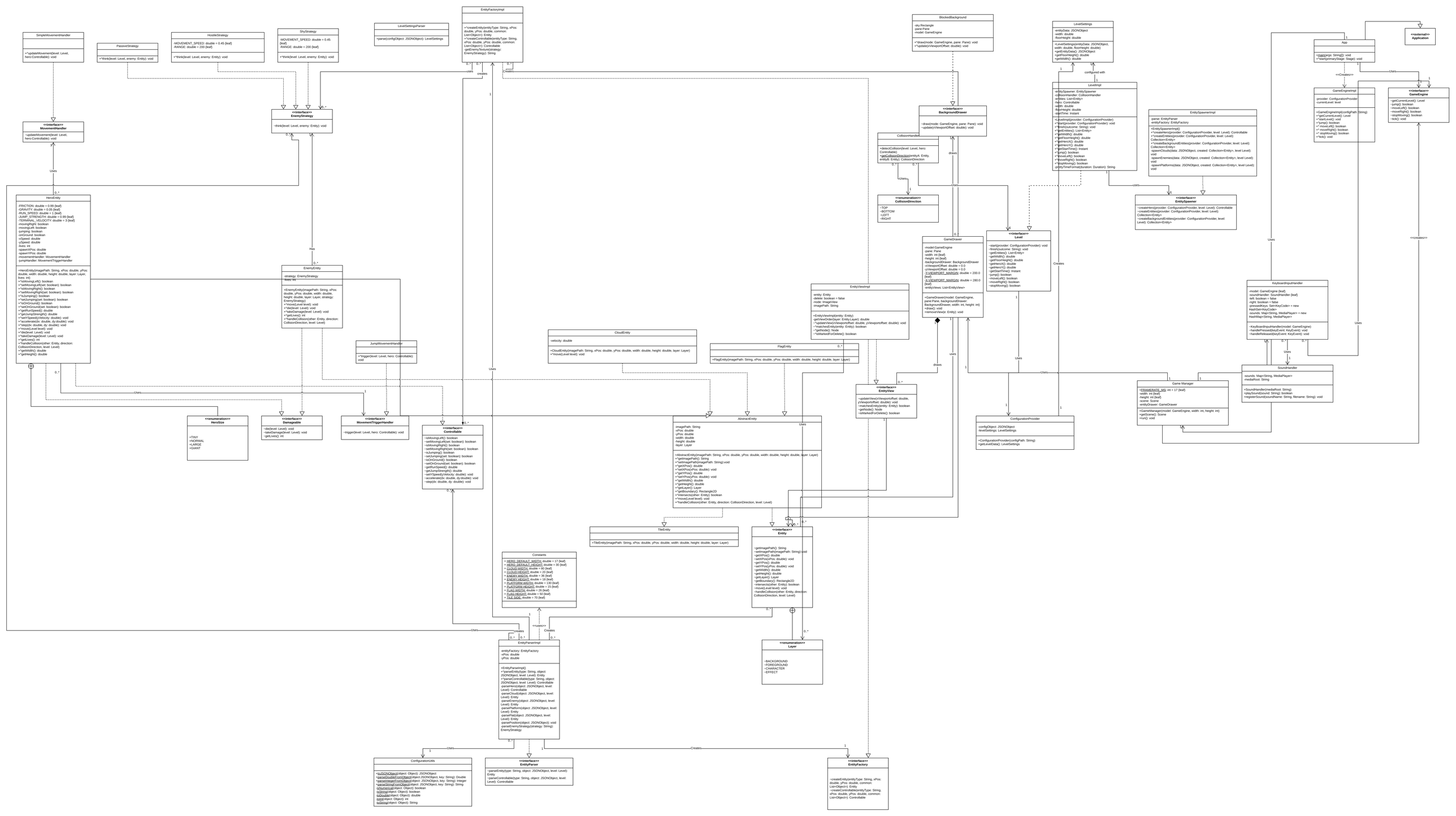
Things to have done differently

- Have loading a save go back to the previous level rather than updating the current level with the previous saved state
- Create a GameMemento object to save GameEngine's state instead of saving it within the class.

References

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Canada: Addison-Wesley

Martin, R.C. (2002). *Agile Software Development, Principles, Patterns, and Practices* (1st ed.). Upper Saddle River, NJ: Pearson Prentice Hall.


SimpleMovementHandler
+*updateMovement(level: Level, hero:Controllable): void
PassiveStrategy
+*think(level: Level, enemy: Entity): void
HostileStrategy
-MOVEMENT_SPEED: double = 0.45 {leaf}
-RANGE: double = 200 {leaf}
+*think(level: Level, enemy: Entity): void
ShyStrategy
-MOVEMENT_SPEED: double = 0.45 {leaf}
-RANGE: double = 200 {leaf}
+*think(level: Level, enemy: Entity): void
LevelSettingsParser
+parse(configObject: JSONObject): LevelSettings
EntityFactoryImpl
+*createEntity(entityType: String, xPos: double, yPos: double, common: List<Object>): Entity
+*createControllable(entityType: String, xPos: double, yPos: double, common: List<Object>): Controllable
-getEnemyTexture(strategy: EnemyStrategy): String
BlockedBackground
-sky:Rectangle
-pane:Pane
-model: GameEngine
+*draw(mode: GameEngine, pane: Pane): void
+*update(xViewportOffset: double): void
LevelSettings
-entityData: JSONObject
-width: double
-floorHeight: double
+LevelSettings(entityData: JSONObject, width: double, floorHeight: double)
+getEntityData(): JSONObject
+getFloorHeight(): double
+getWidth(): double
configured with
LevelImpl
-entitySpawner: EntitySpawner
-collisionHandler: CollisionHandler
-hero: Controllable
-exit: Controllable
-width: double
-floorHeight: double
-startTime: Instant
+LevelImpl(provider: ConfigurationProvider)
+*start(provider: ConfigurationProvider): void
+*finish(outcome: String): void
+*getEntities(): List<Entity>
+getWidth(): double
+*setFloorHeight(): double
+*getHero(x): double
+*getHero(Y): double
+*getStartTime(): Instant
+*jump(): boolean
+*moveLeft(): boolean
+*moveRight(): boolean
+*stopMoving(): boolean
-printlyTimeFormat(duration: Duration): String
EntitySpawnerImpl
-parse: EntityParser
-entityFactory: EntityFactory
+EntitySpawnerImpl()
+*createHero(provider: ConfigurationProvider, level: Level): Controllable
+*createEntities(provider: ConfigurationProvider, level: Level): Collection<Entity>
+*createBackgroundEntities(provider: ConfigurationProvider, level: Level): Collection<Entity>
-spawnClouds(data: JSONObject, created: Collection<Entity>, level Level): void
-spawnEnemies(data: JSONObject, created: Collection<Entity>, level Level): void
-spawnPlatforms(data: JSONObject, created: Collection<Entity>, level Level): void
App
+start(args: String[]): void
+*start(primaryStage: Stage): void
<<external>>
Application
GameEngineImpl
-provider: ConfigurationProvider
-currentLevel: level
+GameEngineImpl(configPath: String)
+*getCurrentLevel(): Level
+*startLevel(): void
+*finish(): void
+*moveLeft(): boolean
+*moveRight(): boolean
+*stopMoving(): boolean
+*tick(): void
<<interface>>
GameEngine
-getCurrentLevel(): Level
-jump(): boolean
-moveLeft(): boolean
-moveRight(): boolean
-stopMoving(): boolean
-tick(): void
<<interface>>
MovementHandler
+updateMovement(level: Level, hero:Controllable): void
<<interface>>
EnemyStrategy
+think(level: Level, enemy: Entity): void
<<interface>>
BackgroundDrawer
+draw(mode: GameEngine, pane: Pane): void
+update(xViewportOffset: double): void
CollisionHandler
+detectCollision(level: Level, hero: Controllable)
+getCollisionDirection(entityA: Entity, entityB: Entity): CollisionDirection
<<enumeration>>
CollisionDirection
-TOP
-BOTTOM
-LEFT
-RIGHT
<<interface>>
EntitySpawner
-createHero(provider: ConfigurationProvider, level: Level): Controllable
-createEntities(provider: ConfigurationProvider, level: Level): Collection<Entity>
-createBackgroundEntities(provider: ConfigurationProvider, level: Level): Collection<Entity>
<<interface>>
Level
-start(provider: ConfigurationProvider): void
-finish(outcome: String): void
-getEntities(): List<Entity>
-getWidth(): double
-getFloorHeight(): double
-getHero(X): double
-getHero(Y): double
-getStartTime(): Instant
-jump(): boolean
-moveLeft(): boolean
-moveRight(): boolean
-stopMoving(): boolean
HeroEntity
-FRICTION: double = 0.99 {leaf}
-GRAVITY: double = 0.05 {leaf}
-RUN_SPEED: double = 1 {leaf}
-JUMP_STRENGTH: double = 0.99 {leaf}
-TERMINAL_VELOCITY: double = 3 {leaf}
-movingRight: boolean
-movingLeft: boolean
-jumping: boolean
-onGround: boolean
-xSpeed: double
-ySpeed: double
-lives: int
-spawnXPos: double
-spawnYPos: double
-movementHandler: MovementHandler
-jumpHandler: MovementTriggerHandler
+HeroEntity(imagePath: String, xPos: double, yPos: double, width: double, height: double, layer: Layer, lives: int)
+*isMovingLeft(): boolean
+*setMovingLeft(set: boolean): boolean
+*isMovingRight(): boolean
+*setMovingRight(set: boolean): boolean
+*isJumping(): boolean
+*setJumping(set: boolean): boolean
+*isOnGround(): boolean
+*setOnGround(set: boolean): boolean
+*getRunSpeed(): double
+*getJumpStrength(): double
+*setYSpeed(yVelocity: double): void
+*accelerate(dx: double, dy:double): void
+*step(dx: double, dy: double): void
+*moveLeft.level level): void
+*die(level: Level): void
+*takeDamage(level: Level): void
+*getLives(): int
+*handleCollison(other: Entity, direction: CollisionDirection, level: Level)
+*getWidth(): double
+*getHeight(): double
EnemyEntity
-strategy: EnemyStrategy
-lives: int
+EnemyEntity(imagePath: String, xPos: double, yPos: double, width: double, height: double, layer: Layer, strategy: EnemyStrategy)
+*die(level: Level): void
+*takeDamage(level: Level): void
+*getLives(): int
+*handleCollison(other: Entity, direction: CollisionDirection, level: Level)
JumpMovementHandler
+*trigger(level: Level, hero: Controllable): void
CloudEntity
-velocity: double
+CloudEntity(imagePath: String, xPos: double, yPos: double, width: double, height: double, layer: Layer)
+*moveLevel level): void
FlagEntity
+*FlagEntity(imagePath: String, xPos: double, yPos: double, width: double, height: double, layer: Layer)
EntityViewImpl
-entity: Entity
-delete: boolean = false
-node: ImageView
-imagePath: String
+EntityViewImpl(entity: Entity)
-getViewOrder(layer: Entity.Layer): double
-*updateView(xViewportOffset: double, yViewportoffset: double): void
-*matchesEntity(entity: Entity): boolean
-*getNode(): Node
-*isMarkedForDelete(): boolean
GameDrawer
-model:GameEngine
-pane: Pane
-width: int {leaf}
-height: int {leaf}
-backgroundDrawer: BackgroundDrawer
-xViewportOffset: double = 0.0
-yViewportOffset: double = 0.0
-X_VIEWPORT_MARGIN: double = 200.0 {leaf}
-Y_VIEWPORT_MARGIN: double = 280.0 {leaf}
-entityViews: List<EntityView>
+GameDrawer(model: GameEngine, pane:Pane, backgroundDrawer: BackgroundDrawer, width: int, height: int)
+draw(): void
+removeView(e: Entity): void
<<interface>>
EntityView
-update(xViewOrViewportoffset: double, yViewportoffset: double): void
-matchesEntity(entity: Entity): boolean
-getNode(): Node
-isMarkedForDelete(): boolean
KeyboardInputHandler
-model: GameEngine {leaf}
-soundHandler: SoundHandler {leaf}
-left: boolean = false
-right: boolean = false
-pressedKeys: Set<KeyCode> = new HashSet<KeyCode>
-sounds: Map<String, MediaPlayer> = new HashMap<String, MediaPlayer>
-KeyBoardInputHandler(model: GameEngine)
-handlePressed(keyEvent: KeyEvent): void
-handleReleased(keyEvent: KeyEvent): void
SoundHandler
-sounds: Map<String, MediaPlayer>
-mediaRoot: String
+SoundHandler(mediaRoot: String)
+playSound(sound: String): boolean
+registerSound(soundName: String, filename: String): void
Game Manager
+FRAMERATE_MS: int = 17 {leaf}
-width: int {leaf}
-height: int {leaf}
-scene: Scene
-entityDrawer: GameDrawer
+GameManager(model: GameEngine, width: int, height: int)
+getScene(): Scene
+run(): void
ConfigurationProvider
-configObject: JSONObject
-levelSettings: LevelSettings
+ConfigurationProvider(configPath: String)
+getLevelData(): LevelSettings
<<enumeration>>
HeroSize
+TINY
+NORMAL
+LARGE
+GIANT
<<interface>>
Damageable
-die(level: Level): void
-takeDamage(level: Level): void
-getLives(): int
<<interface>>
MovementTriggerHandler
+trigger(level: Level, hero: Controllable): void
<<interface>>
Controllable
-isMovingLeft(): boolean
-setMovingLeft(set: boolean): boolean
-isMovingRight(): boolean
-setMovingRight(set: boolean): boolean
-isJumping(): boolean
-setJumping(set: boolean): boolean
-isOnGround(): boolean
-setOnGround(set: boolean): boolean
-getRunSpeed(): double
-getJumpStrength(): double
-setYSpeed(yVelocity: double): void
-accelerate(dx: double, dy:double): void
-step(dx: double, dy: double): void
AbstractEntity
-imagePath: String
-xPos: double
-yPos: double
-width: double
-height: double
-layer: Layer
+AbstractEntity(imagePath: String, xPos: double, yPos: double, width: double, height: double, layer: Layer)
+*getImagePath(): String
+*setImagePath(imagePath: String):void
+*getXPos(): double
+*setXPos(xPos: double): void
+*getYPos(): double
+*setYPos(yPos: double): void
+*getWidth(): double
+*getHeight(): double
+*getLayer(): Layer
+*getBoundary(): Rectangle2D
+*intersects(other: Entity): boolean
+*moveLevel level): void
+*handleCollison(other: Entity, direction: CollisionDirection, level: Level)
TileEntity
+TileEntity(imagePath: String, xPos: double, yPos: double, width: double, height: double, layer: Layer)
<<interface>>
Entity
-getImagePath(): String
-setImagePath(imagePath: String):void
-getXPos(): double
-setXPos(xPos: double): void
-getYPos(): double
-setYPos(yPos: double): void
-getWidth(): double
-getHeight(): double
-getLayer(): Layer
-getBoundary(): Rectangle2D
-intersects(other: Entity): boolean
-moveLevel level): void
-handleCollison(other: Entity, direction: CollisionDirection, level: Level)
Constants
+ HERO_DEFAULT_WIDTH: double = 17 {leaf}
+ HERO_DEFAULT_HEIGHT: double = 30 {leaf}
+ CLOUD WIDTH: double = 80 {leaf}
+ CLOUD HEIGHT: double = 20 {leaf}
+ ENEMY WIDTH: double = 36 {leaf}
+ ENEMY HEIGHT: double = 18 {leaf}
+ PLATFORM WIDTH: double = 130 {leaf}
+ PLATFORM HEIGHT: double = 15 {leaf}
+ FLAG WIDTH: double = 26 {leaf}
+ FLAG HEIGHT: double = 50 {leaf}
+ TILE SIDE: double = 70 {leaf}
<<enumeration>>
Layer
-BACKGROUND
-FOREGROUND
-CHARACTER
-EFFECT
EntityParserImpl
-entityFactory: EntityFactory
-xPos: double
-yPos: double
+EntityParserImpl()
+*parseEntity(type: String, object: JSONObject, level: Level): Entity
+*parseControllable(type: String, object: JSONObject, level: Level): Controllable
-parseHero(object: JSONObject, level: Level): Controllable
-parseCloud(object: JSONObject, level: Level): Entity
-parseEnemy(object: JSONObject, level: Level): Entity
-parseFlag(object: JSONObject, level: Level): Entity
-parsePlatform(object: JSONObject, level: Level): Entity
-parseTile(object: JSONObject, level: Level): Entity
-parsePosition(object: JSONObject): void
-parseEnemyStrategy(strategy: String): EnemyStrategy
ConfigurationUtils
+toJSONObject(object: Object): JSONObject
+parseDoubleFromObject(object:JSONObject, key: String): Double
+parseIntegerFromObject(object: JSONObject, key: String): Integer
+parseStringFromObject(object: JSONObject, key: String): String
-isNumerical(object: Object): boolean
-isString(object: Object): boolean
-isDouble(object: Object): double
-toInt(object: Object): int
-toString(object: Object): String
<<interface>>
EntityParser
-parseEntity(type: String, object: JSONObject, level: Level): Entity
-parseControllable(type: String, object: JSONObject, level: Level): Controllable
<<interface>>
EntityFactory
-createEntity(entityType: String, xPos: double, yPos: double, common: List<Object>): Entity
-createControllable(entityType: String, xPos: double, yPos: double, common: List<Object>): Controllable
Uses
creates
Has
draws
configured with
<<creates>>
<<uses>>