

Dungeons and Dragons Game Library & Example Client Interface

Calvin Withun

Index

Abstract	3
Dungeons and Dragons Background Information.	4
Key Terms & Definitions	5
System Requirements	8
Design	9
Client Graphical Interface.	12
Test Cases	15
Summary & Conclusions.	16
References	17

Abstract

Dungeons and Dragons is a classic tabletop roleplaying game which was first publicized in 1974. This game has remained popular over the years, and continues to receive updates in the modern day. However, the ongoing Covid-19 pandemic has made it challenging for players to engage with the game. Dungeons and Dragons was designed to be played around a table with a group of other people, which is problematic for those who are attempting to maintain social distancing. There are, of course, certain online resources such as www.roll20.com or www.dndbeyond.com which provide a virtual platform for playing the game, and these resources have been available since long before the outbreak began. While these resources typically attempt to provide some degree of automation for common events in the game, such as making attack rolls and saving throws, there is a great deal of game content which still requires user maintenance. The objective of this project is to create a game library which eliminates this user maintenance by automating all interactions between core game elements. This automation will serve many purposes, such as allowing newer players to engage with the game without needing an in-depth knowledge of the game mechanics, allowing players to focus on the role-playing aspect of the game without worrying about the math governing character interactions, or opening the door to certain game mechanics which typically bog down gameplay due to their scale or complex nature. This project also includes a client program which serves as an example and proof of functionality for interfacing with the library.

Dungeons and Dragons Background Information

Dungeons and Dragons, despite being a roleplaying game which exists primarily in the mind of the players, is governed by a mathematical probability system referred to as the D20 system. In this system, the success or failure of any activity is typically determined by having the actor roll a 20-sided die. If the rolled value meets or exceeds a certain threshold, the activity is a success. The rolled value may be augmented by a series of bonuses or penalties, such as applying a +2 bonus to the roll or re-rolling the die for a new number entirely. An augmentation may come from any number of sources – it may be a product of the actor's intrinsic characteristics, it may be a product of malignant or benevolent magic, it may be a product of toxins or other environmental hazards, etc. The central objective of this project is to create a library which is capable of automating all such augmentations, regardless of the source from which it originates or the specific nature of the augmentation, and then comparing the final value of the roll to the target threshold and facilitating the appropriate fallout.

Key Terms & Definitions

- **AttackRoll** – an attack roll in Dungeons and Dragons is how a player can attempt to overcome the defenses of a target, normally in order to deal damage. For example, if an Orc wishes to deal damage to a player, it must make an attack roll using its weapon contested by the defensive properties of the player's armor. If the attack roll surpasses the value of the player's defenses, the player takes damage. If it does not surpass the value of the player's defenses, nothing happens.

AttackRoll is a subclass of DiceContest, and it facilitates all relevant calculations required to make the offensive attempt described above. It does not calculate the defenses of the target, however.
- **Damage** – damage in Dungeons and Dragons is typically represented by a collection of dice and a damage type. The dice are rolled, and the sum of the dice is the damage dealt by an attack. The damage also has a damage type, such as fire or cold or slashing. This damage type determines how effective a particular quantity of damage is against a target. For example, a Yeti may be entirely unaffected by any damage whose damage type is cold, but suffers extra damage whenever the damage type is fire.

Damage is a class which represents a potential range of damage originating from a single Event, even if that Event deals damage of multiple types. This class contains a collection of DamageDiceGroup objects.
- **DamageDiceGroup** – a subclass of DiceGroup which represents a damage roll of a particular damage type. This class contains a list of Die objects which specify the range and spread of damage, and a DamageType enum which determines the type of that damage.
- **DamageType** – an enum which represents a damage type such as Acid, Fire, Slashing, or Thunder.
- **DiceContest** – a dice contest in Dungeons and Dragons refers to any time when a 20-sided die is rolled and then compared to a target value. This includes attack rolls, saving throws, skill checks, and ability checks. The die may have various bonuses or penalties applied to it, according to the circumstance in which it is rolled.

DiceContest is a subclass of Event which represents some form of contest between two GameObject objects. One GameObject provides a value which defines the threshold at which the contest succeeds or fails, and the other GameObject generates an attempt to match that threshold. The fallout for each DiceContest object is loaded from an external Lua script upon construction.
- **DiceGroup** – a class which represents a collection of dice of various sizes, and a bonus value to be applied when the dice are rolled.
- **Die** – a class representing a physical N-sided die. The size of the Die is defined upon construction.
- **Effect** – this class is an abstract description of any lingering or static modifiers that are applied to a creature in Dungeons and Dragons. The standard game rules have no explicit

reference to what this class represents. This class is intended to represent things such as class features, spell effects, racial features, feats, special equipment effects, terrain effects, and more.

Effect is a class representing an ongoing collection of logic applied to an Entity object. The particular logic contained in each Effect object is loaded from an external Lua script upon construction. Effect objects have the ability to modify Event objects which an Entity is in the process of invoking. Later updates to the library may grant Effect objects the ability to similarly modify Task objects which an Entity is in the process of invoking.

- **Entity** – this class represents any game piece in Dungeons and Dragons which is assigned a stat block (e.g. any game piece which has the capacity to take actions). Entity is a subclass of GameObject which represents any game piece with the capacity to invoke a Task or an Event, or to sustain an Effect. Entity objects may represent game pieces such as Player Characters, Zombies, or Traps, but not such things as buildings, doors, or chests.
- **Event** – this class represents any particular activity which a game piece is prepared to perform. For example, a Zombie might be able to perform the *Slam* event. Entity is a class representing some activity which an Entity is prepared to perform. An Event may be modified by various Effects from various Entity objects before it is invoked. Note that this class is distinct from Task because a Task is aware of the resource cost necessary to be invoked, whereas an Event is not, and a Task grants an Entity the ability to invoke Events.
- **EventGroup** – a class representing a collection of alternative Event objects. If any of the contained Event objects are invoked from an EventGroup, all contained Event objects become unavailable to the invoking Entity. This allows a Task to grant an Entity the option to either make a melee weapon attack or a thrown weapon attack, for example, but not both, upon being invoked.
- **GameObject** – a class representing any game piece which might normally appear on a physical game board.
- **SavingThrow** – a saving throw, in contrast to an attack roll, is how Dungeons and Dragons allows a player to attempt to avoid the effects of some harmful or dangerous phenomenon. This entails the target or targets of the phenomenon rolling a saving throw using the specified ability score contested by some threshold defined by the offending party. If the saving throw meets or surpasses that threshold, the target is able to avoid the brunt of the phenomenon (though it may still suffer reduced damage or lesser effects). If the saving throw falls short of that threshold, the target suffers the full force of the phenomenon.
SavingThrow is a subclass of DiceContest which represents a saving throw. This class facilitates all relevant calculations, including all relevant dice rolls and bonuses, and then applies the final saving throw score to the dice contest value of the targeted Entity.

- **Task** – this class represents a particular way which a game piece may expend its action economy and other resources in order to do things in-game. Tasks available to game pieces may typically be found towards the bottom of that game piece’s stat block as listed under *Action*, *Bonus Action*, *Reaction*, etc.

Task is a class which represents the potential for an Entity to exchange resources for the ability to invoke Event objects. At this time, all Task objects have no resource cost, as this is yet to be implemented in the library.
- **VirtualBoard** – this class represents the physical tabletop and gridded map upon which Dungeons and Dragons is played.

VirtualBoard is a final class which represents a three-dimensional game board / game space. This class contains a list of GameObject objects, and possesses functions dedicated to searching for Entity objects located within particular three-dimensional geometries. It may be treated as a two-dimensional plane if this is preferred simply by having the client software ignore one of the three dimensions.
- **ItemAttackGroup** – this class represents all the ways which a weapon may be used to make an attack roll in Dungeons and Dragons. A weapon might be used to make a melee attack (swinging it at a target), a thrown attack (throwing the weapon at a target), or a ranged attack (using the weapon to launch ammunition at a target). However, only one of these options may be chosen when a game piece makes a weapon attack. That collection of options is represented by this class.

WeaponAttackGroup is a subclass of EventGroup which specifically represents the EventGroup generated by making a single weapon attack. The Entity invoking instances of this class notifies those instances whenever the item in its main hand slot changes, in order to ensure that the EventGroup always stays up to date with the selected weapon.

System Requirements

The library shall support the following tasks at the user level (unimplemented features are italicized):

- Introduce new GameObject objects to the library.
- Assign Task objects to Entity objects.
- Invoke Task objects assigned to Entity objects.
- Invoke Event objects from an EventGroup queued by an Entity.
- *Move GameObject objects within the library's virtual space.*
- Manipulate the inventory of an Entity:
 - Equip weapon in main-hand.
 - Equip weapon in off-hand.
 - Stow weapon in main-hand.
 - Stow weapon in off-hand.
 - Equip a versatile weapon with two hands.
 - Equip a versatile weapon with one hand.
 - Equip armor.
 - Doff armor.
 - *Move Item objects into an Entity object's inventory.*
 - *Move Item objects out of an Entity object's inventory.*
- View Entity data:
 - View hit points, maximum hit points, ability scores, Entity name, etc.
 - View available Tasks.
 - View queued EventGroups and Events.
 - View active Effects.
 - View inventory contents.
- Search certain geometries in the library's virtual space for GameObject objects:
 - Cone search.
 - Cube search.
 - Line (cylinder) search.
 - Nearest-coordinate search.
 - Sphere search.

Design

The relationships between all of the core data types implemented in this library can be found in Diagram 1.

Many of the core data types implemented in the library were created with the purpose of being explicitly defined by client-provided files. In this way, the client may introduce customized game content to the library without needing to modify the library itself. The library provides a small collection of default derivations of these general classes for common Dungeons and Dragons phenomena such as attack rolls, saving throws, and item proficiencies, but introduces no particular game content on its own.

The external files the library loads are Lua scripts. Lua is its own programming language, and as such Java does not normally know how to interact with lua files. However, there is an opensource project called LuaJ which allows for Lua files to be loaded and processed by a Java program. LuaJ creates virtual machines which know how to process Lua scripts, and then permits this virtual machine and the Java program to interface with each other.

The library has a single abstract class dedicated to interacting with LuaJ features, called Scriptable. All core data types which load Lua scripts are derived from Scriptable. Because of this, if a different external file type is ever desired, minimal changes would be required in the library. See Diagram 2 for a more detailed depiction of the class hierarchy within the library.

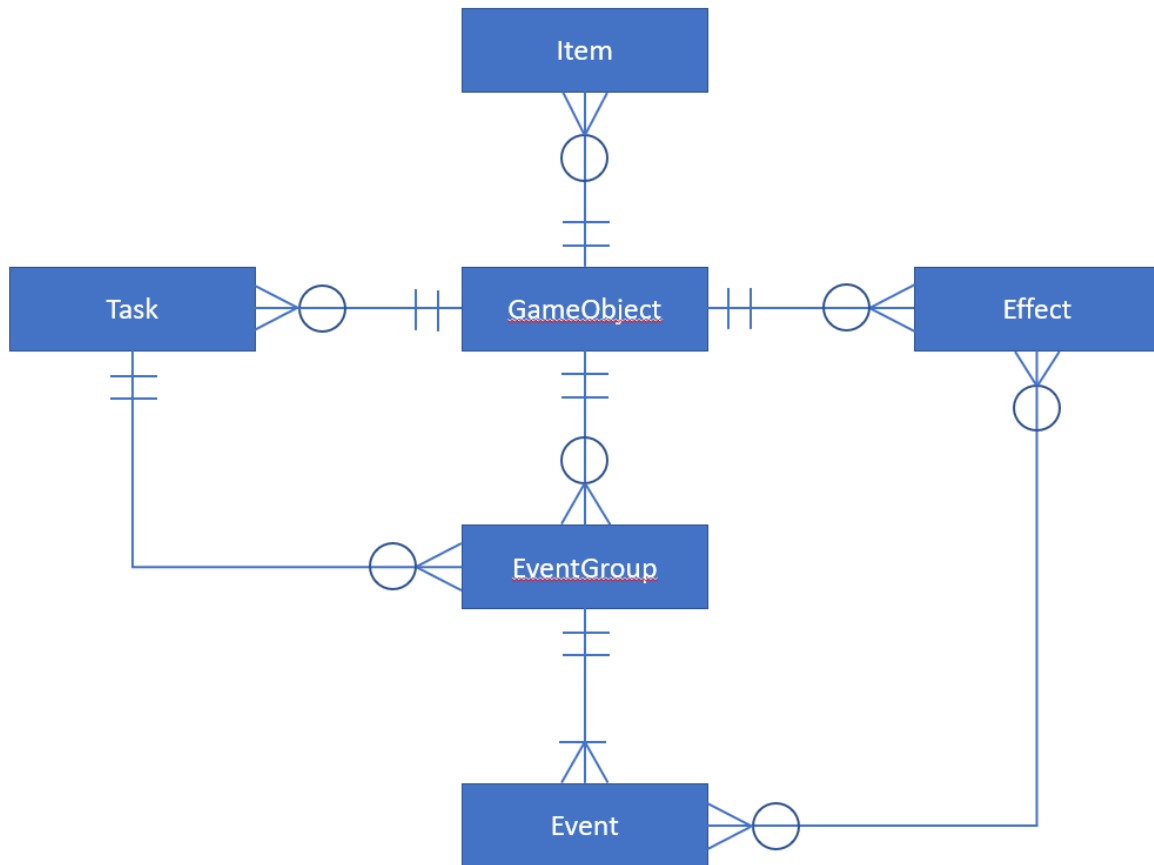


Diagram 1: Entity Relationship Diagram for Core Data Types

Note that certain derived classes may not fit this pattern as presented. For example, ItemAttackGroup is derived from EventGroup, and it contains references to an Item object and a GameObject (Entity) object which are not contained by the parent class.

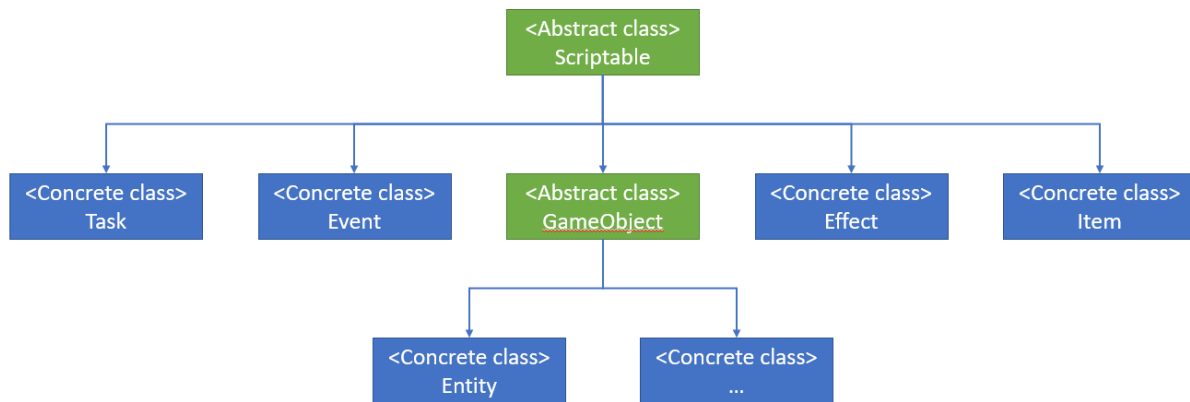


Diagram 2: Core Data Types Class Diagram

As GameObject is an abstract class, but not everything which might be found on a game board is an Entity (e.g. buildings and chests), there must be other yet-unimplemented derivations of GameObject. These are represented by the “...” concrete class block. There are non-core derivations of the other classes not shown in this diagram.

Client Graphical Interface

The client program provides a 2-dimensional rendering of the library's virtual space. Every GameObject which the library is aware of is rendered as a blue square. A GameObject may be selected by clicking on it, at which point it turns red. If the user clicks off of the selected GameObject, it will return to its normal blue color. While a GameObject is selected, the user may view its Tasks and queued Events. To invoke a Task or queued Event, the user may simply click on that icon. If a Task is invoked, the selected GameObject will acquire new queued Events. If a queued Event is selected, the selected GameObject will invoke that Event targeting itself, and all Events sharing an EventGroup with the selected Event will be removed from the Event queue. To see these changes render on screen, the GameObject will have to be deselected and reselected (this is a bug which has not yet been fixed). A screenshot of the client program may be seen in Image 1.

At this time, the client does not display any data indicating what happens when an Event is invoked. This data is currently only viewable through the console running the program. However, if this data were viewable, it would resemble the content of Image 2.

The client program does not implement the majority of the features available in the most recent library release. This is because the client program is a secondary program, intended only as a proof of concept for an interface between the user and the library, as well as because of the general lack of experience developing graphical interfaces available to the developer during this project. However, using the client program as a starting point for interacting with the library, it would not be difficult to see how to include additional features into a more refined graphical user interface.

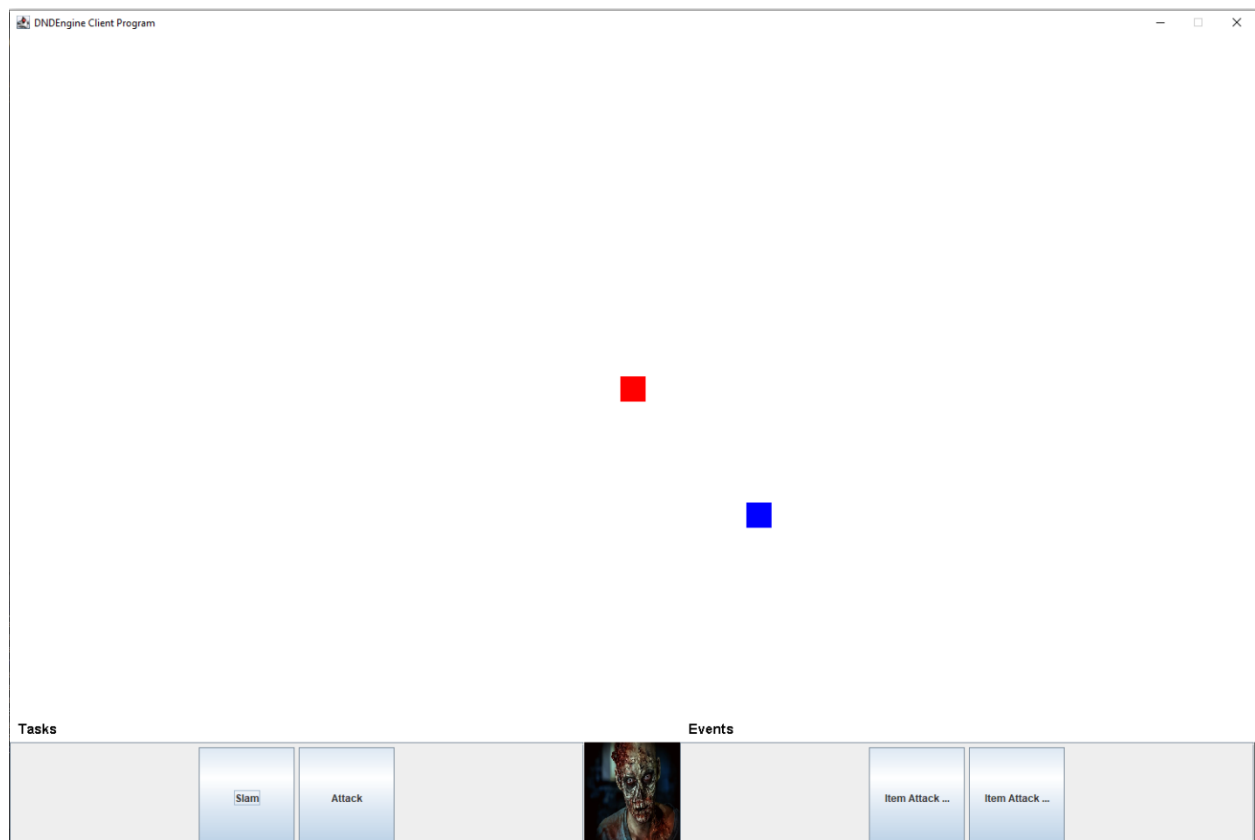
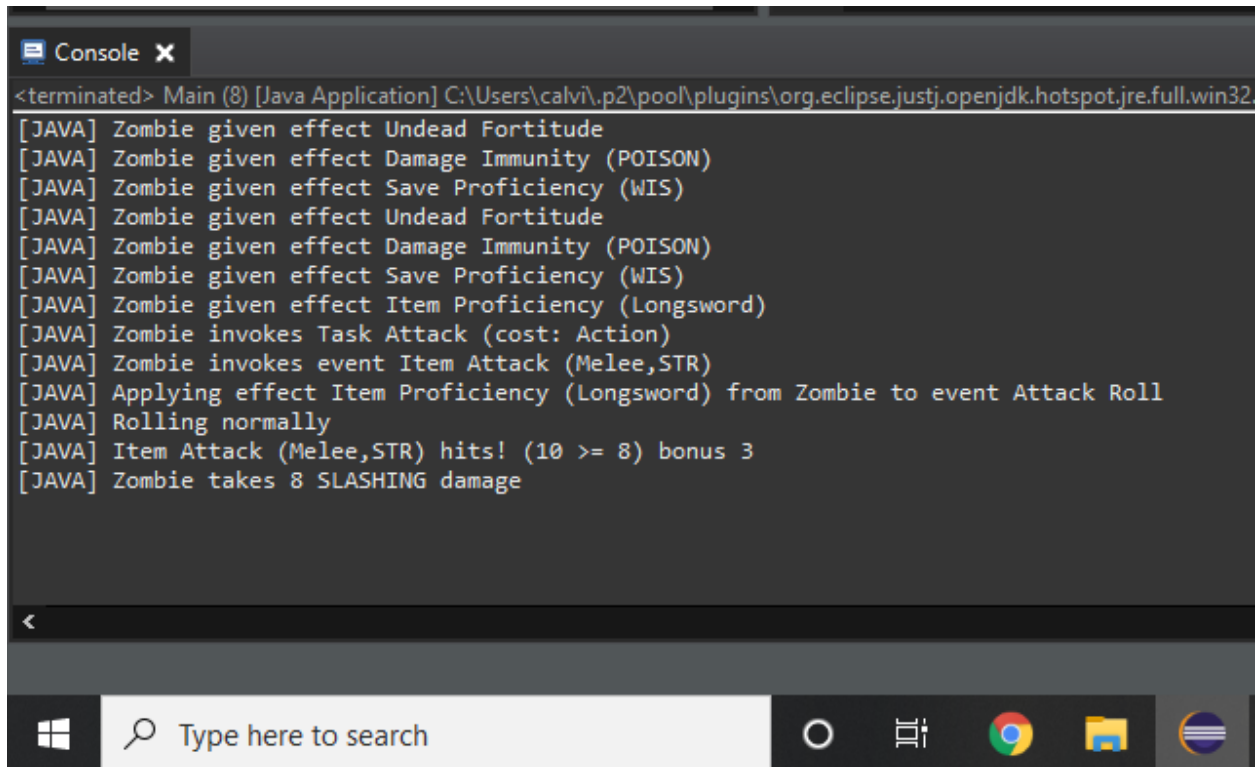


Image 1: Client GUI Program for the DNDEngine library.



```

<terminated> Main (8) [Java Application] C:\Users\calvi\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32
[JAVA] Zombie given effect Undead Fortitude
[JAVA] Zombie given effect Damage Immunity (POISON)
[JAVA] Zombie given effect Save Proficiency (WIS)
[JAVA] Zombie given effect Undead Fortitude
[JAVA] Zombie given effect Damage Immunity (POISON)
[JAVA] Zombie given effect Save Proficiency (WIS)
[JAVA] Zombie given effect Item Proficiency (Longsword)
[JAVA] Zombie invokes Task Attack (cost: Action)
[JAVA] Zombie invokes event Item Attack (Melee,STR)
[JAVA] Applying effect Item Proficiency (Longsword) from Zombie to event Attack Roll
[JAVA] Rolling normally
[JAVA] Item Attack (Melee,STR) hits! (10 >= 8) bonus 3
[JAVA] Zombie takes 8 SLASHING damage
  
```

Image 2: DNDEngine Library Console Output

This output is eventually intended to be recorded in log files to be viewed for debugging purposes. It is not intended to be directly presented to the user, which is why it does not appear anywhere in the client program's graphical display.

Test Cases & Results

Since the client program is inadequate for the purpose of performing manual tests on the library, automated test cases were devised to verify that the library behaves as expected. To accomplish this, JUnit 5 was incorporated into the library.

JUnit 5 is a testing interface which allows the developer to program a suite of scenarios, and then assert what certain variables and values are expected to be when interrogated by JUnit. If there is ever a mismatch in actual value from expected value, JUnit will report a failure. In this way, arbitrarily many tests may be performed on arbitrary sections of a program. JUnit also tracks path coverage, allowing the developer to view which lines of code were or were not traversed over the course of running the test suite.

Ideally, the test cases will have 100% path coverage and 0 failures or errors. However, as JUnit was introduced to this project only after the architecture was stable enough to justify creating such tests, it does not yet have 100% path coverage. Of the tests which have been defined, all but one pass without failure or error. The single failure is due to a feature which has not yet been implemented in the library, but which has a test case defined. It is a premature test, rather than a test which reveals a mistake or bug in the code. The results of running the full JUnit test suite can be found in Image 3 (below).

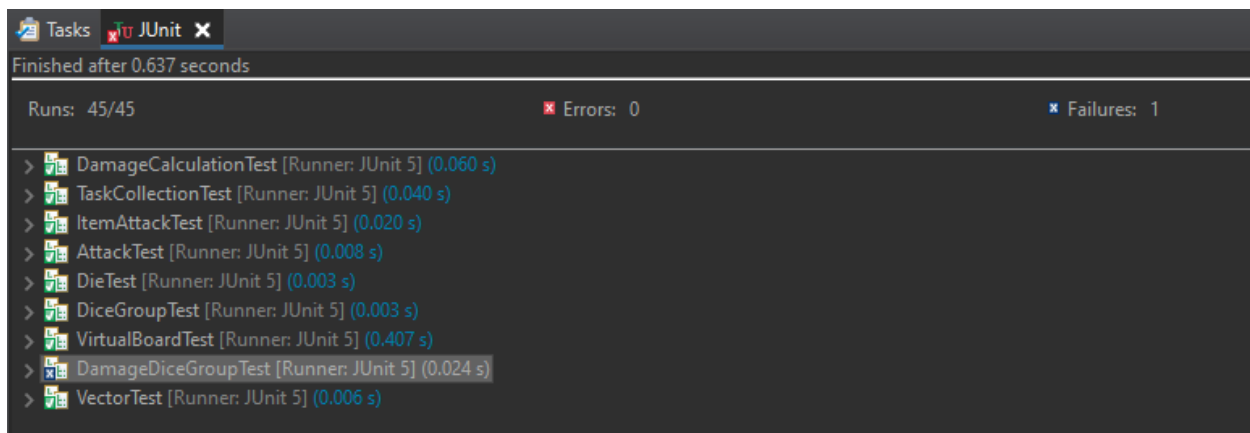


Image 3: JUnit Test Suite Results

The particular failure results from a test which is designed to verify that damage values can never be reduced below 1 (all damage rolls must always deal at least 1 point of damage after modifiers have been applied). This feature was removed over the course of development, but the test case remains as a reminder to re-implement it at a later time.

Summary & Conclusions

This project has identified a set of core data types and has successfully established a system which allows them to interact with each other as needed. This set of data types and their interactions constitutes a pattern which is applicable to many games beyond Dungeons and Dragons (for example, most Pokémon games could be represented using this pattern), so this library requires specific default derivations of these data types in order to give it the desired theme. Not all of these derivations have been implemented at this point in the library development, meaning that the project is in a state of incompleteness. Specific next steps include the introduction of resources (the prerequisites for invoking a Task) and spellcasting functionality (this will require derivations of Task and EventGroup), as well as making the library aware of GameObject movement and of Item objects which have been dropped and are not in an inventory.

The client program functions as expected, essentially, thus demonstrating that the library can be effectively interfaced with by client software. Furthermore, the client software introduces an Entity script for a Zombie, which includes all Zombie features that are currently intended to be supported by the library, and it functions without complications, thus demonstrating that Entity objects can effectively be defined by Lua scripts, as well as Task, Event, and Effect objects. Finally, the client software provides a collection of Lua scripts meant to be loaded as Item objects, and these items all function as expected when used by the Zombie Entity objects. Lua scripts, therefore, are demonstrated to effectively drive the particular behaviors of the core data types as Java objects, meaning that any client can introduce customized Lua scripts for novel game content.

The developer intends to continue this project to completion. If the project becomes popular upon publication, it might be possible to create a code repository of some sort for storing these novel Lua scripts. This would allow users and enthusiasts to create their own custom game content and publish it for public use, potentially drawing together a new community of creative individuals to help expand the overall influence of this project.

References

- Bechtold, Stefan, et al. *JUnit 5 User Guide*, 26 Oct. 2020, 22:06:33, junit.org/junit5/docs/current/user-guide/.
- Gamma, Erich, et al. *Design Patterns Elements of Reusable Object Oriented Software*. 1st ed., Addison Wesley, 1998.
- Roseborough, James, et al. "Luaj/Luaj." *GitHub*, 1 Apr. 2020, github.com/luaj/luaj.