Software Design Patterns Project Report

Hanuman Chu, Caden Swartz, Pratik Asarpota, Kaelem Deng

Overview

For our project, we decided to create a spreading simulation. These spreaders, be it an alien invasion or a wildfire, move across a grid of tiles, depleting resources and potentially growing in numbers over a series of turns. In order to implement this idea, we had to tackle many challenges, including:

- Providing many ways (and an extensible model) for spreaders to behave
- Managing a large grid of tiles with potentially changing properties
- Simulating each step of a turn, with an extensible model for turn events
- Logging the simulation event-by-event
- Loading simulation parameters from a JSON file and turning the result of the simulation into a JSON file

In order to overcome these challenges, we have applied our learnings from this course, using various OOP design patterns to create an efficient, maintainable, and expandable simulation framework.

Design Patterns

Decorator

We wanted to add tile events which affected tiles in various ways. To do this we used decorators because we could add them randomly to tiles to affect their functionality. This would make it easy to create new events and ensure all the logic for an event is in that event and only that event. As an alternative, we could add a class that manages tile events and has a reference to a tile and a command list for each tile function where individual commands represent parts of an event. The benefit of this is that it's much simpler to add and remove events and when an event modifies a subset of the functions in the tile, we only need to process the event when a function it modifies is called whereas in the decorator pattern, we would have to go through all

the events each time even if all the decorator does for that function is call its inner's function with the same parameters. The downside here is that it would be slightly harder to maintain as we now have 2 classes directly subclassing tile, the event manager and base events so if we modified tile we would need to modify both of those instead of just the base decorator if we were using that pattern. In addition, Java has no way of ensuring that a function pointer's parameter and return value is linked to a regular function's return value so there are some situations where the function in tile could be changed and we would not receive a compile error in base events telling us to update it.

Builder

Our simulation acts upon a potentially very large grid of tiles, which we want to be able to be easily created from a file specification, without being able to insert/remove/change tiles during the simulation execution itself. The Builder pattern is an excellent way to achieve this. The director of the builder is able to specify grid size, which tiles should go where, and even default tiles to efficiently create the TileGrid object. Then, the TileGrid class itself does not need "addTile" or "setTile" methods which we would only want to be used at creation. The only real downside to this pattern is that we need to create another Builder object just to create the TileGrid itself, but this is a very small price to pay for a much more streamlined interface. Ultimately, the Builder object both streamlines TileGrid creation and more effectively controls access to the TileGrid.

Strategy

We wanted to have spreaders behave differently from each other in 2 ways: how they spread and how they extract resources. To do this, we used the strategy pattern allowing us to easily combine different ways spreaders behave. It also allows us to easily create new ways for spreaders to behave as well as test behaviours individually. We could have used command for this purpose instead but our intent was to be able to swap algorithms so command doesn't really apply.

The logging subsystem also uses the strategy pattern, as it allows for the client to both choose a filtering strategy (Filter outputted logs via their specified log level, etc), and an output strategy (printing to output stream or collecting the logs). This solves the problem of having an extendable output and filtering method while being able to test each filter/output individually.

Singleton

We use a PRNG for randomness and wanted to have it produce the same results given the same initial seed. To do this, we used the singleton pattern to ensure that only one instance is ever created and all requests for random numbers go through said instance. This makes it easier to test other parts of the code because we know everything is using the same source of randomness. Alternatively, we could just pass a local instance around to whoever needs it. This would allow us to have multiple PRNG streams which we could use to ensure certain decisions are not affected by others. However, this isn't something that we require currently and it makes it harder to verify that everything that should use the same stream does. If this was something we wanted, a better solution would be a singleton registry instead to keep it easy to verify.

The logger subsystem also uses the singleton pattern as we had a need to store logs in a single instance to ensure proper log tracking. At the end of the simulation, the singleton logger can output all the logs as the single source of truth.

Iterator

The Iterator Pattern is used throughout the "grid" package as a means of standardizing retrieval of a collection of tiles. As Java already includes the "Iterable" class, the Iterator structure is already created for us.

Prototype

We wanted the user to be able to specify tile types and where those types are located so they wouldn't have to create each tile individually. To do this we used the prototype pattern to be able to create tiles for each tile type and whenever a tile type is needed, create a copy of the corresponding tile to return. An alternative that doesn't really apply is flyweight because the tile values are changing pretty much immediately upon starting the simulation. We could do a lazy thing where we turn the flyweight into a real tile once something is done to it but this would have little if any benefit as the tile values change pretty much immediately and would have the big downside of needing to do extra processing for the flyweight. Another alternative that fits our situation a little more would be creating tiles purely based on the tile type information. This would mean we wouldn't need a temporary tile for each tile type and an extra function in tile but while small, there is some processing done in the creation of tiles that can be avoided when cloning.

Command

We leveraged a simple Command pattern in order to allow certain clients to decorate tiles in the TileGrid without being able to completely replace or remove tiles entirely. We simply created an "Applier" functional interface inside of "TileDecorator". Then, this "Applier" type can be passed as a parameter, allowing a client to specify what decorator to use so long as that decorator has a corresponding Applier. This use of the command pattern effectively prevents clients from haphazardly modifying the TileGrid. Unfortunately, there is a tradeoff in that every concrete Decorator needs to also extend Applier. This extension cannot be enforced via inheritance as Applier needs to be a static class.

Composite

As previously mentioned, the Logger uses the strategy pattern, in order to use multiple strategies at once, for example, using the PrintStreamOutput strategy with the CollectionOutput strategy to both print logs into the output stream and also to collect all the logs into a list, this requires a composite pattern to fulfill. Instead of the composite pattern, this system could have been managed directly in the singleton logger, however, this would mix the logic of the logger to both coordinate the use of other classes, and to also keep track of the used logs, which would violate the single responsibility principle. This is used to effectively implement the strategy design pattern, so that users can use multiple strategies at the same time.

Template Method

We wanted logged events to be on a per-subsystem basis, so each system can describe the format that they wanted to log with. To do this, we leveraged the template method pattern so that other systems can simply implement the LogEvent class that describes the log level their subsystem should log at and also the format that their logs will be in. This is easily extendable if another subsystem were to be added, as all they would have to do is create a logging class that extends LogEvent, and this would be polymorphic with the other event loggers.

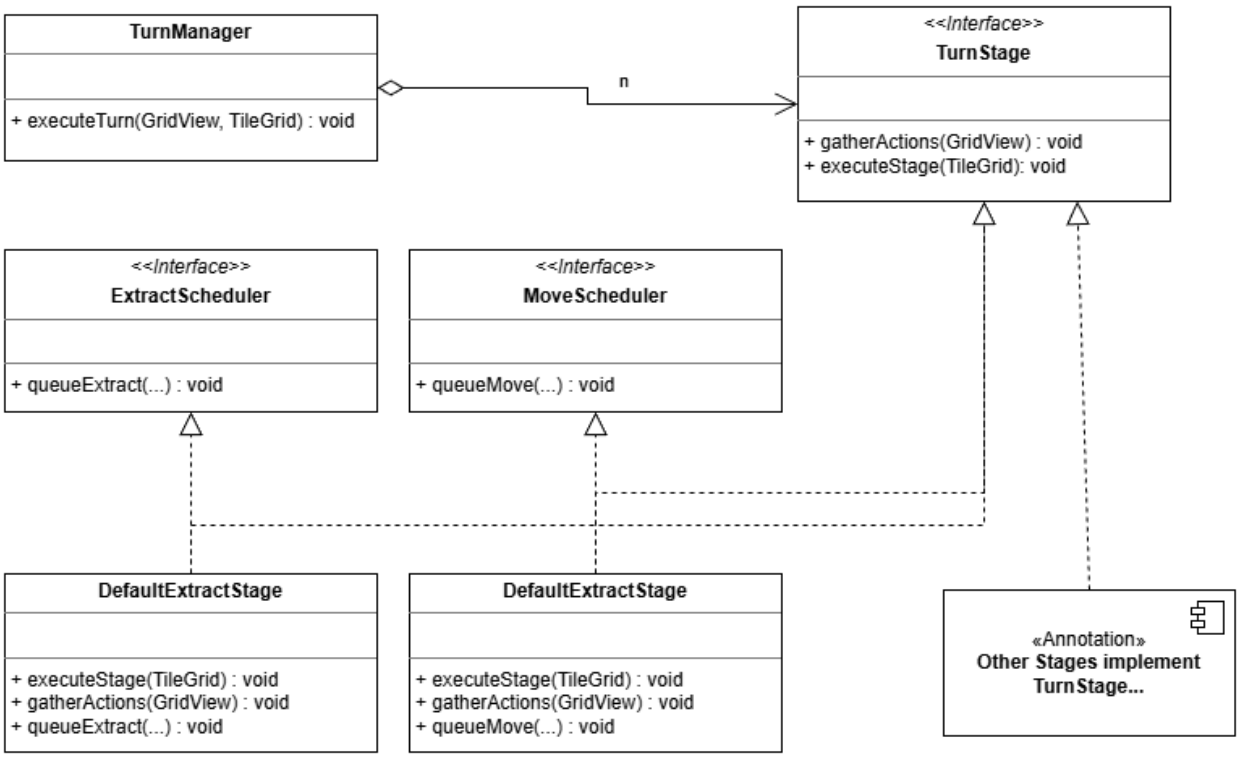
Facade

The logger coordinates the different logging abstractions (filtering, output strategy, and event logging). This allows the client code to simply call a "log()" function from the singleton

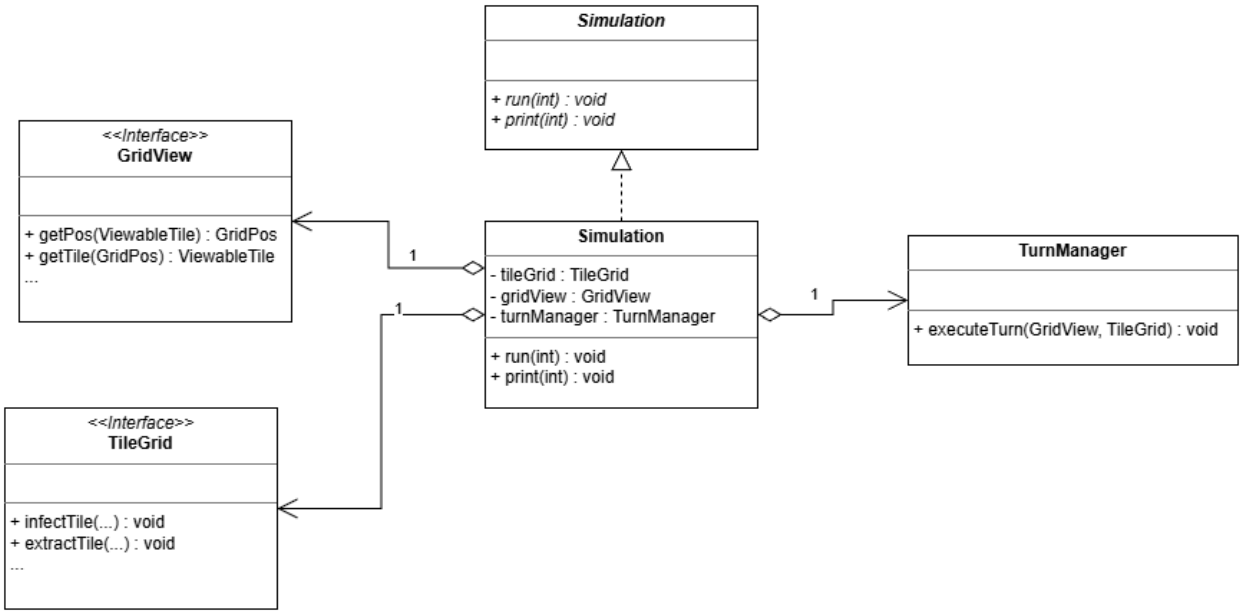
logger with filters/outputs set without having to worry about the implementation details. The alternative to this would be to just allow the clients to directly access the coordination of the logger themselves, however, this couples the internal implementation of the logger and the client.

UML Diagrams

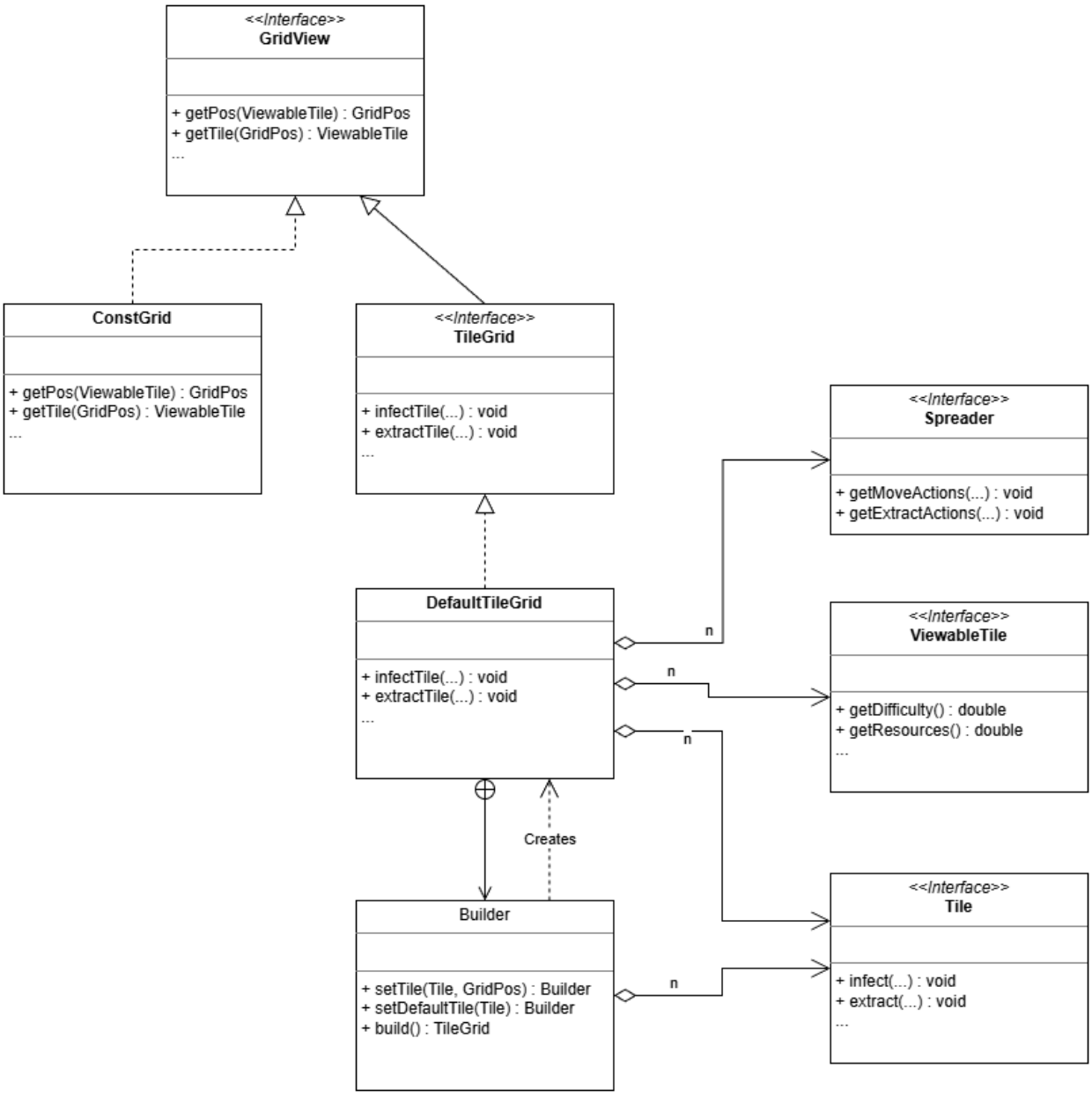
The "turn" package structure:



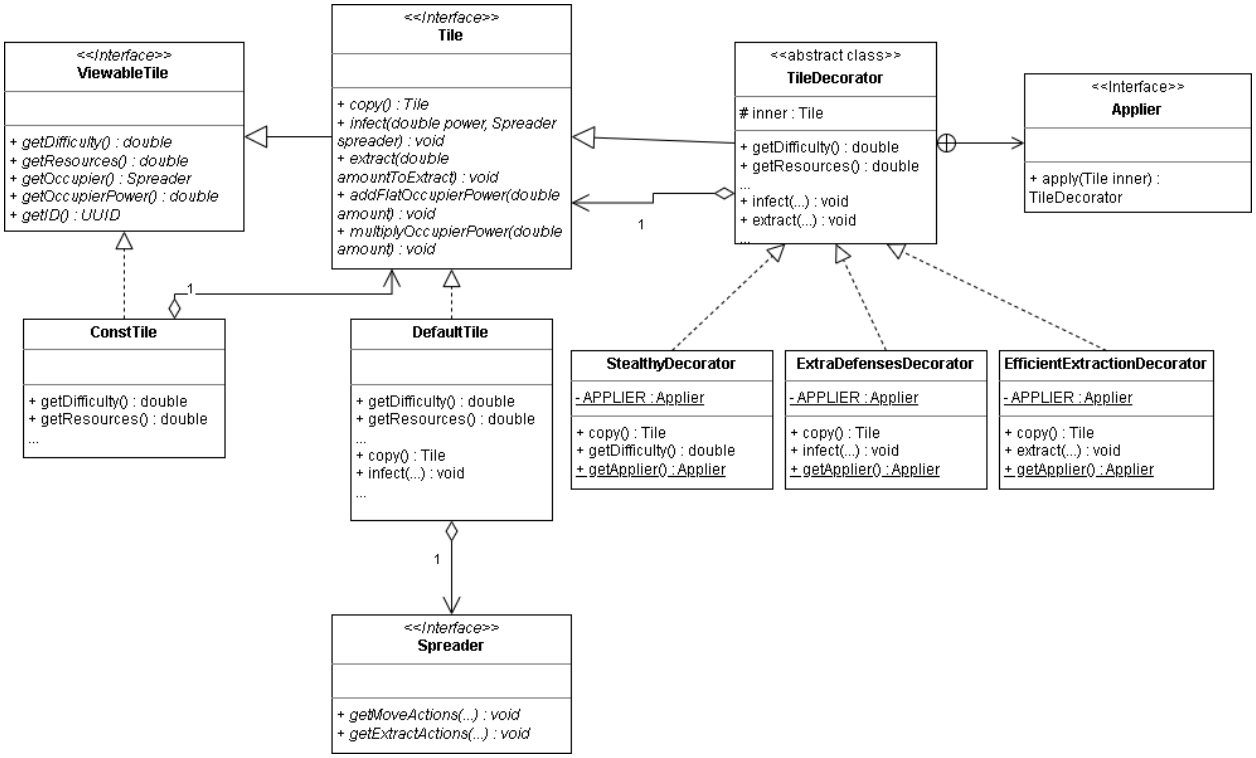
The “simulation” package structure:



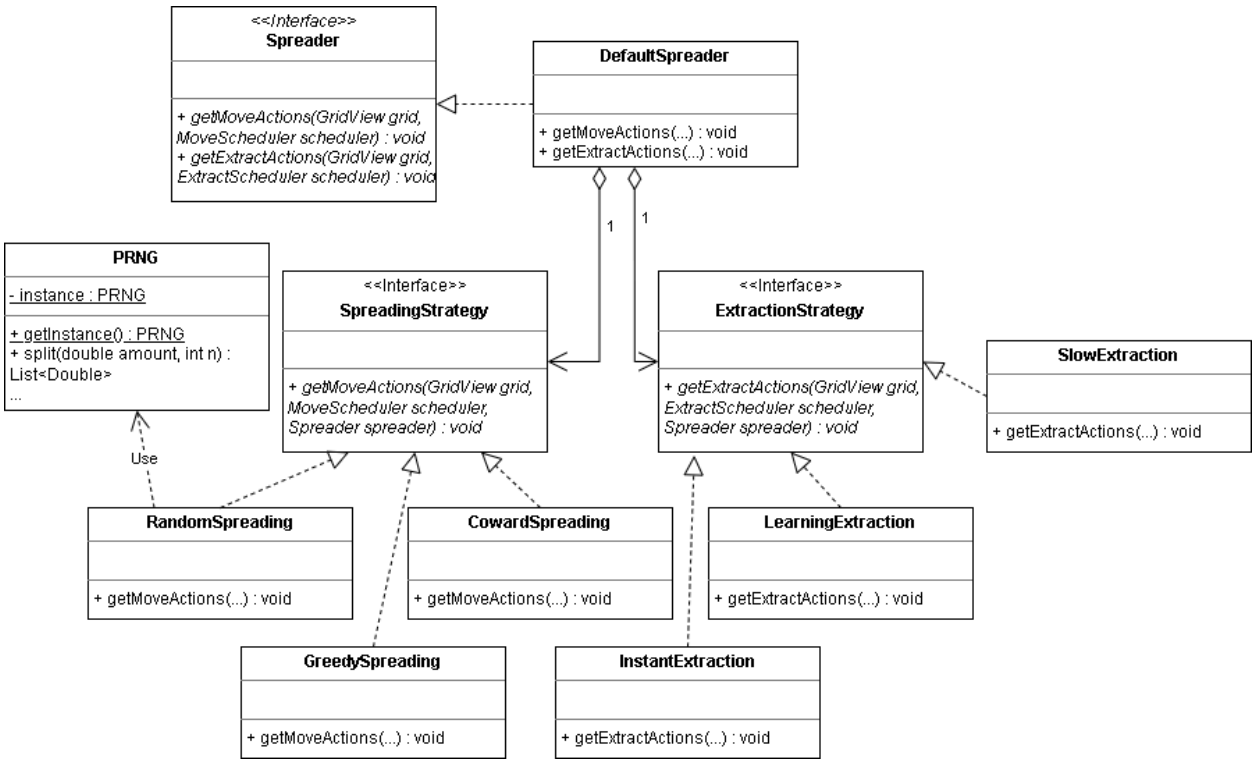
The “grid” package structure:



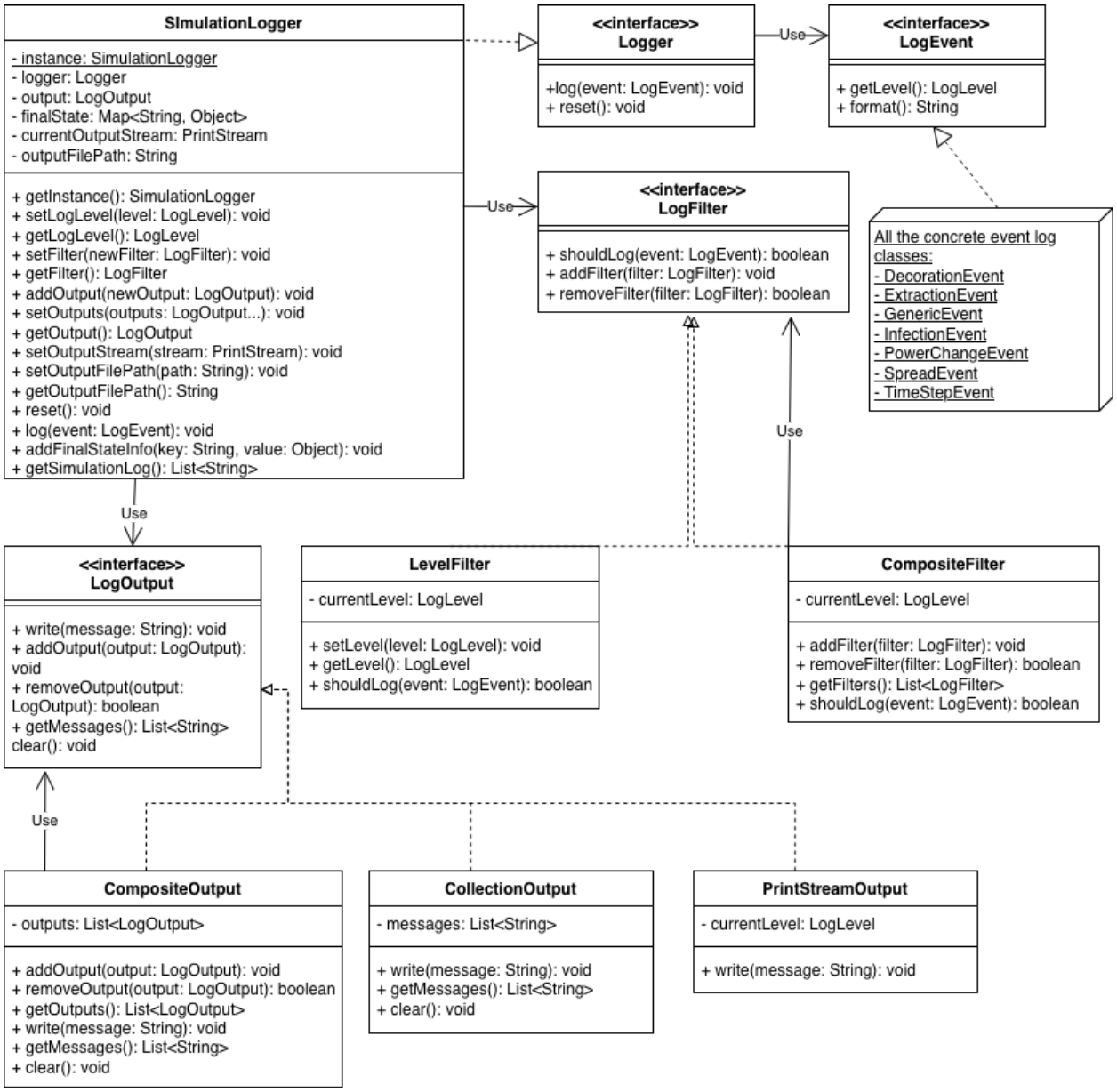
The “tile” package structure



The "spreader" package structure



The “logger” package structure



Design Evaluation

Our design is maintainable because it follows SOLID. More specifically, it is testable because classes depend on interfaces whenever possible which makes it easy to test individual classes by injecting implementations of said interfaces that behave how the test wants. This dependency on interfaces also makes our design extensible because to add new versions of

existing classes with different functionality, we can simply add new implementations of those interfaces.

Reflections

In this project we faced a lot of challenges, mainly from our choice of language. The main one was Java's lack of a const keyword. To fit with SOLID and ensure we don't expose classes to stuff they don't need and shouldn't use, having a const version of our classes was important. To workaround Java's lack of const we had to create a separate version of the class with only getters that had a reference to the real class. This meant that there would be an extra dereference every time and at the time we thought that was a small enough penalty if it allowed us to prevent classes from modifying tiles they weren't supposed to. However, as time went on we discovered more and more problems with this solution including but not limited to: java's lack of implicit conversion, object addresses being modifiable, and const tiles being unequal to their tiles. If we were to do it again and we had to keep Java, I wouldn't bother making a const version of a class because the maintainability added by ensuring classes that shouldn't don't get to modify a tile is offset by the performance cost and the extra time spent dealing with the issues resulting from a const tile and a tile being different types.

Another challenge we faced was with the base Java library's linked lists. The main benefit of a linked list is that removals are O(1) if you know the node to remove. Java's equivalent of this is an iterator but iterators become invalid after the list is changed in any way so this meant we couldn't store multiple iterators and remove just one of them because that would invalidate the others. The workaround to this was to use a hashset which has amortized O(1) removals and additions. It has the downside of not keeping order but we were able to work around this.

One challenge that would be difficult to do in some other languages as well was with the decorators. We wanted to be able to tell the owner of the tiles what decorator to apply which is possible in C++ with templates. There is a way to pass a class name in java but there is no way to ensure that said class is a decorator at compile time so people could pass in non decorator classes and we would only find out when the function is run. To fix this, we used the command pattern as described in the patterns section above. As seen in that section, this solution still has the issue that we can't ensure every decorator has an applier. In C++ this would be possible with

CRTP but Java appears to have no way of doing this so we had to accept this as the best we can get.

Another challenge was with the PRNG. We ran into an issue where multiple runs with the same seed produced different results. After a lot of investigating, we discovered that the issue was with HashMap which produced non deterministic. To fix this, we could force an order but this would affect the efficiency so we decided to use seeds for PRNG testing only. If we had more time, we could have created our own hash map implementation or found a library with one that was deterministic.