SWEN30006 Project 2 Report

[Fri 13:00] Team 07

- Angel He, 1264462
- Ziming Wang, 1180051
- Weida Fan, 1238958

Table of Contents

Note:

- The resolution of some diagrams is limited to the resolution of the page. For a clearer version, please see the relevant file in `pacman/documentation/`.
- The colors in the report body match those of the packages illustrated in the Static Design Model (Figure 3.1). They serve as a visual guidance.
- The report and diagrams both focus on the design of the new application/functionalities, and less on the refactoring of the game component, which is based on Project 1.
  As a recap, in Project 1, we categorized `Actors` into `Items` and `MovingActors` (superclass of `PacPlayer` and `Monsters`). The `Items` were managed by a specialized `ItemManager` (renamed for the current design).

Part 1: Design of Editor

1.0: Application Breakdown

As Figure 3.0 shows, the application incorporates three main components at the domain level: Game, Editor, and the Checkers. An all-encompassing application `TorusVerseApp`, is also required to:

1. Bind these components together: By the principle of Indirection and Low Coupling, the app serves as a mediator between these components, and controls their interactions.
2. Act as the application's façacde controller, providing users with a single, convenient entry point.
3. Facilitate extensibility: By the Open-Closed Principle, each component should be modified without influencing others. They should be easily detachable from and adaptable to the app.

1.1: Façade and Adapter Pattern: MapEditorAdapter and Controller

A primary problem is how the editor should be adapted for the TorusVerseApp. Fortunately, the 2D map editor created by Matachi already provides a façade controller (`Controller`) that wraps the editor subsystem. Therefore, this was the only class that required small modifications. Yet, considering the use of a different editor as a future variation point, a stable interface would be useful to minimize the impacts of such changes on the app. Therefore, the Adapter pattern was employed: The editor `Controller` is now contained within a concrete `MapEditorAdapter` class (see Figure 3.1), which implements the `EditorAdapter` interface. The `EditorAdapter` interface consists of a single method, `runEditor()`, with the file path as the parameter. That is, if a new editor is to be incorporated, all that's needed is to build a new concrete adapter class which implements this method. Protected Variation is thus achieved by the use of a combination of Façade and Adapter pattern.

1.2: Strategy and Visitor Pattern: Map format conversions

Delving deeper into the map editing functionality, other incompatibilities were revealed, especially regarding the map "format":

1. Whereas `Game` utilized the `CellType` enum for `Items`, the map editor uses strings and characters to denote the different `Tiles`. Thus, how do we map between the internal representations used by the map editor, and the representations used by `Game`?
2. The new map format contained initial positions for not only the `Items`, but also the `MovingActors`. Thus, how to initialize the `MovingActors` based on the map but not the properties file?
3. As above, how do we preserve the original game's behavior, of using `PacManGameGrid` and a properties file for initialization, whilst adopting the new map format?

Before all, the first two problems required us to *unify* the representations, which is really what the "Grid" conceptual class in Figure 3.0 represents. In the new application, the `MovingActors` will be initialized according to the map file, just like the `Items`. Hence, Polymorphism naturally comes into play. The `CharacterType` enum was thus created, which implements the interface `ActorType` just as `CellType` does. Now, the `Actors` do not have to be 'put' separately, but can be managed together under the `ActorType` reference. Moreover, now a single HashMap (`ACTOR_TYPE_TO_CHAR_DICT` in `Controller`) is sufficient to unify the game's representations of its `Actors` and the editors' character symbols for its `Tiles`. Accounting for the non-functional requirements of consistency and maintainability, the lengthy if-else block

in the original `loadFile()` method in `Controller` is also replaced by a simple look-up of `STR_TO_CHAR_DICT` – a static final mapping between XML strings and `Tiles` in `Controller`.

The other problems become solvable once a common reference is settled. Targeting question 2 and 3, by the principle of Low Coupling, `Level` should only 'see' a `MapReader` and a `PacManMap`, without concerning their underlying implementations. Specifically, regardless of the 'source' of information, all that's needed by `Level` to set up the `Actors` is a mapping from `Location` to `ActorType`. That is, a conceptual "Grid" could be as simple as a dictionary with location as key, and `ActorType` as value. These needs are expressed by the `getItemLocations()` and `getCharacterLocations()` methods in the `MapReader` interface. Also, for High Cohesion, since the doing-responsibility of map-reading is now assigned to the `MapReaders`, the map-reading logic is moved from `loadFile()` in the original `Controller`, to `EditorMapReader`.

The separate consideration of how this information is extracted from each map format, condenses to one problem: How to design for varying, but related algorithms? The Strategy Pattern meets this objective. As Figure 3.1 illustrates, the Workers package contained two concrete implementers of the `MapReader` interface – `EditorMapReader` and `PropertyMapReader`. Whilst the former handles the new XML map format (`EditorMap`) for the editor, the latter retains the game's original design of attaining location information from both the `PacManGameGrid` and properties file.

To complement this, the Visitor Pattern was applied to the maps themselves. Abiding by the principle of High Cohesion, the `MapReaders` should simply be a helper for reading maps, whilst the maps themselves are the focus. Thus, instead of adding a composite-aggregation association between each `MapReader` and `PacManMap`, the `PacManMaps` themselves simply 'accept' being read by the `MapReaders`. Hence, the `readMyItemLocations()` and `readMyCharacterLocations()` methods in the `PacManMap` interface both take a `MapReader` as its argument. Now, the game can easily switch between its original mode of operation, and the new mode designed for the `TorusVerseApp`. To see this, `Game` could be run via the `OrginalDriver` program in the `TestPrograms` package, independently of the new app.

1.4: Template Method and Composite Pattern: The Checker family

Also visible from Figure 3.1 is the `checker` package, which constitutes the level and game checking logic used by the application. Whilst the rationales of level checking and game checking are different, they do share two common behaviors:

1. Checking the validity of a file path (either a map file or a directory), and
2. Reporting the errors to the log file.

The Strategy Pattern, using delegation, serves as a valid resolution, as the `Checkers` indeed have different checking algorithms. Yet, a closer inspection revealed that both checkers should store a collection of error messages (i.e. a common attribute), and their error-logging behavior (`inspectAndLogErrors()`) is exactly the same. Therefore, whilst also solving the problem of separating a generic algorithm from a detailed design, the Template Method pattern facilitates better code reuse here. This settled the hierarchical structure where `GameChecker` and `LevelChecker` both inherit the abstract class `Checker`.

On the other hand, there is one important distinction between `GameChecker` and `LevelChecker`. The `GameChecker` follows a sequence of *steps*: from filtering the file names in the directory, to extracting the numerical prefixes, to collecting the valid filenames. Contrastingly, `LevelChecker` applies multiple small, individual checks, each on a different aspect of the map file. This in turn inspired the use of the Composite

Pattern: Using Pure Fabrication, a series of small checkers (PacStartChecker, PortalPairChecker, NumGoldPillChecker, GoldPillAccessibilityChecker), which did not exist at the domain level (not in Figure 3.0), was created. These became the “leaves” of the CompositeLevelChecker. Moreover, all of them inherit the LevelChecker superclass, so that they could be accessed via a common interface. Under this design, not only could each of these checks be separately applied, making unit testing convenient; but they can also be easily combined, making composite policies configurable.

Similarly, although GameChecker is not a composition, it acts as a “wrapper” for MapNameSequenceChecker. With this provision of Indirection, the external classes using GameChecker (i.e. TorusVerseApp) becomes less vulnerable to changes in the game checking logic, such as the addition of new rules. Notably, an improvement that could be made in the future is to further reduce the coupling between Controller and CompositeLevelChecker. For example, the current checks are invoked within saveFile() and loadFile(), because by Information Expert, these methods offer easy access to the filename of the loaded/saved files, which the error-logging needs. Considering future extensibility, these could well be migrated to MapEditorAdapter.

Another design considered was to use the Decorator Pattern in place of the Composite Pattern. In fact, the decorator’s recursive structure may even better suit the implicit sequential structure amongst some of the checkers. For example, the accessibility of golds and pills should only be applied once the map has passed the validity checks for PacPlayer’s starting point and the number of golds and pills, so GoldPillAccessibilityChecker could wrap around NumGoldPillChecker and PacStartChecker. However, conceptually, the decorator ought to embellish a core object. Yet, in this application, these individual checkers all exist at the same ‘level’ with equal importance – they are more like “leaf” objects. It is really the composite itself, CompositeLevelChecker, that should be the primary focus. Therefore, comparatively, the composite design provides a lower representational gap for this application.

1.6: Singleton Factory: AppComponentFactory

According to the Creator and Information Expert principles, TorusVerseApp is a suitable creator of its components, as it ‘contains’ and closely uses these components. However, the creation logic could be a variation point, since A24 is inclined to merge the TorusVerse and Multiverse in the future. Therefore, following the Protected Variation and High Cohesion principles, the creation responsibility may well be assigned to a specialized, Pure Fabrication object – AppComponentFactory. The immediate succeeding question is then: How should this factory be accessed? Since only one such factory instance is needed, the natural solution is to adopt the Singleton Pattern. That is, AppComponentFactory can itself provide a global, single point of access to the external bodies.

With object creations being delegated to AppComponentFactory, TorusVerseApp now also has a more cohesive set of responsibilities – namely, to control the flow of the application. Figure 3.2 illustrates the partial process of application set-up based on the different command-line arguments. Unlike the conventional builder pattern used to construct complex *objects* in a step-by-step manner; here, TorusVerseApp serves as the director who controls the complex *procedure* of construction. As an example, depending on the argument, TorusVerseApp directs AppComponentFactory (i.e. like a concrete builder) to create the type of checker (CompositeLevelChecker or GameChecker) it needs. This clear separation of responsibilities allows the construction logic to be modified without influencing the components under construction.

1.7: Chain of Responsibility Pattern: It's your responsibility!

The behavioral pattern, Chain of Responsibility, enables requests to be passed along a chain of potential handlers until one of them handles it. Using indirection and delegation, this effectively reduces the coupling between the sender and receivers. This was helpful in several implementation-wise issues.

The first broader issue is: How to split up the multiple responsibilities of the façade class, `Level`? The original `Game` class (precursor of Level) was already bloated, and the additional functionalities in TorusVerse would only exacerbate the problem of high coupling and low cohesion. The `Workers` package was the key to resolve this problem. It formulates a chain of responsibility, whereby `Level` employs the `SettingManager` (like an executive manager), which in turn employs a `PropertyReader`, `MapReader`, and `ItemManager`. Table 1.0 outlines the set of responsibilities assigned to each class.

Table 1.0: Responsibilities of each class in the Workers package.

Class	Responsibility
`SettingManager`	The façade class for this package. Allocates (delegates) jobs to the sub-workers below via wrapper methods.
`ItemManager`	Draws the grid, manages (creates, puts, removes) the items.
`PropertyMapReader`	Together with `PropertyReader`, reads the location information from the `PacManGameGrid` and properties file (original behavior).
`EditorMapReader`	Reads the location information from an `EditorMap`.
`PropertyReader`	Reads and parses values from the properties file.
`LocationIndexConverter`	Converts Location to an Integer for indexing purposes.

The next issue was: How does `Game` handle the boundaries of the `Levels`? `Game` needs to wait for a `Level` to be finished before starting the next one, which is an asynchronous operation that a for loop cannot easily satisfy. This prompted the introduction of the `LevelCompletionHandler` functional interface. As Figure 3.3 illustrates – on completion of one level (detected by `act()` in `Level`), the `handleCompletion()` method of the `completionHandler` in `Level` is invoked, which in turn triggers `Game` to run the next level. An alternative considered here was to use the Observer Pattern: `Game` could be a subscriber of `Level`, monitoring the state changes of, say, an `isLevelOver` attribute in `Level`. However, the multiplicity of the relationship would be inverted – `Game` the observer would observe multiple `Levels` (subjects), unlike the traditional pattern where `Level` the subject should be observed by multiple subscribers. Therefore, the Observer pattern appears as a slight overkill compared to using a handler.

The chain of responsibility is also evident in the handling of `IOException` and `JDOMException`, which are mostly triggered by the failure of `Level` and `Game` checking. All classes which encounter these exceptions simply `throw` it, and it is only at the end of the chain, at the `Driver` program, where such exceptions are explicitly handled. Considering usability as a non-functional requirement, a `JOptionPane` message dialogue box is displayed upon catching these exceptions, notifying the users of the failed checks. This method of exception-handling is applied in several cases, such as when the user does not select a file to load, or attempts to save/load a check-failed map.

Part 2: Design of Autoplayer & Game internals

2.0: Singleton Factory and Template Method Pattern: PacPlayers

The conceptual distinction of Manual Player (user-controlled) and Auto Player already existed at the domain level (see Figure 3.0), which are both specialized `PacPlayer`s. They indeed share lots of commonalities, including behaviors upon eating `Items`, as well as attributes like the number of pills eaten (score). The only difference lies in how they select the direction of their next move. Thus, the Template Method Pattern using inheritance suits this setting. In particular, the `act()` template method is placed in the (grand-)parent class, `MovingActor`, and the specific implementation of `setNextDirection()` is deferred to these concrete `PacPlayers`. Following, as with the `AppComponentFactory`, the `PacPlayer` creation logic could also be delegated to another Singleton Factory, `PlayerFactory`, which constructs the right player depending on the `isAuto` property read from the properties file.

2.1: Strategy Pattern and Polymorphism: PathFindingStrategy and LocationExpert

A ‘smarter’ player necessitates ‘smarter’ strategies than the original behavior of following property moves. For extensibility, it should also be configurable in the strategy it uses. Therefore, the Strategy Pattern which allows creating an interchangeable family of algorithms (`PathFindingStrategy`) is applicable. The concrete strategy currently used, `OptimalPathFindingStrategy`, adapts the Breadth-First-Search algorithm. Remarkably, the arguments of `findPath()` also involve two new interfaces: `LocationPredicate` and `LocationExpert`. The former is a functional interface, which enables the specification of constraints on the target location – for example, the current `AutoPlayer` prioritizes locations with golds and pills (and additionally avoids monsters if possible). This also enhances Protected Variation – for the evolution point of extending the autoplayer to deal with (or favour) ice-cubes, specifying a new `LocationPredicate` to prioritize locations of `IceCubes` would suffice. The source of this locational information, in turn, would be a `LocationExpert`.

The `LocationExpert` interface is implemented by both `ItemManager` and the `PacManMaps`. This categorization did not exist at the domain level, as its necessity was only discovered during the implementation stage. The trigger was the distinction between static and dynamic location information: whilst the initialization of the game setting requires a static map (`PacManMap`), `AutoPlayer`’s path-finding strategy requires a dynamic, adaptive view of the item locations (e.g. `PacPlayer` could eat items). Further complexity is introduced by the fact that `OptimalPathFindingStrategy` is also used by `GoldPillAccessibilityChecker` to check the initial accessibility of pills. This means that, `PathFindingStrategy` should be able to attain the location information from both static and dynamic ‘maps’. Hence, incorporating Polymorphism, this gave rise to the new interface, `LocationExpert`, a common reference to all information experts of item locations.

2.2: Idea for Extensions on IceCube, Monsters and Pills

With the current, refactored design of the original game component, the additional freezing effect of ice cubes becomes simple to implement. Figure 3.4 illustrates the process after `PacPlayer` eats an `IceCube`, whilst Figure 3.5 presents a possible structure of the abstract base class, `Monster`. Firstly, `Monster` would

require an additional `state` and `stateTimer` attribute. A small modification to `eatItem()` in `PacPlayer` would also be required – if `PacPlayer` eats an ice, then the timer for all monsters would be started. On timeout, the monsters would return to their normal state. Additionally, the current design utilizes the Template Method Pattern, where the generic method `act()` in `MovingActor` defers the implementation of `move()` to its subclasses. Thus, to enforce the rule that `Monsters` cannot move in frozen state, adding a simple state checking to `move()` in Monster would suffice.

As previously mentioned in Section 1.7, under the current implementation, each class has a cohesive set of responsibilities, and predominantly uses delegation to obtain their required information. Further, the `ch.aplu.jgamegrid.Actor` class already has a public reference to `GameGrid`, which `Level` inherits. Following, `Level` is the Creator and Information Expert of `SettingManager`, which is in turn the Creator and Information Expert of the other 'workers'. Thus, for `AutoPlayer` to work in the presence of monsters and pills, it simply has to follow the chain of responsibilities:

- If `AutoPlayer` wants to 'know' about the Monsters, it passes the request to `Level`.
- If it wants information about the pills, it first enquires `Level`, which would pass this request to `SettingManager`, which in turn asks `ItemManager` for the information.

In fact, `setNextDirection()` in `AutoPlayer` already provides a sample use of this pattern – because `AutoPlayer` initially tries to find a path that avoids monsters whilst targeting locations of golds and pills, it needs the information about `Monsters` and `Item` locations.

2.3: Portal

Building upon the refactored version of `Game`, the implementation of the new type of `Item`, `Portal`, becomes quite simple. All that was necessitated was to add some new types to the enum, `CellType`; and by Polymorphism, let `Portal` extend the `Item` class. To imitate the portal's paired nature, each portal simply stores the location of its partner portal. Then, as Figure 3.6 presents, the addition of a simple conditional check to getNextMoveLocation() sufficiently fulfills the portal's capability. Notably, the initial design considered storing a direct reference to the partner portal. However, this would form a circular dependency between the two `Portal` objects, and possibly cause memory leaks. Hence this idea was falsified. Nevertheless, taking inspiration from it, a recursive structure may indeed conform well to this representation. Thus, as a future improvement, if `Portal` is to have additional capabilities, the Decorator or Composite pattern could be applied.

Part 3: Diagrams

3.0: Domain Model

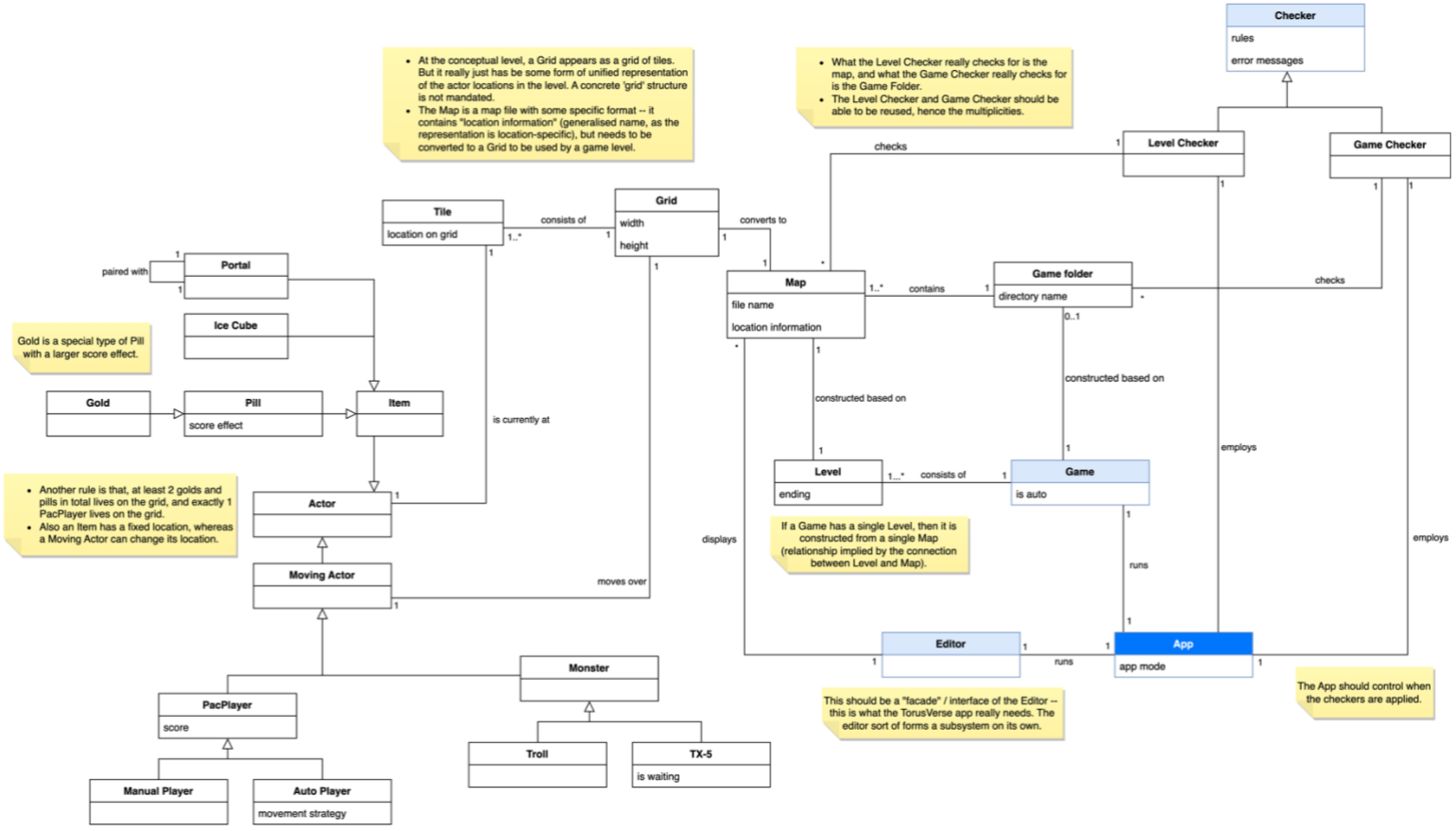

Figure 3.0: A Domain Class Diagram capturing the domain concepts relating to the auto-player and game levels/maps for PacMan in the TorusVerse.

3.1: Static Design Model

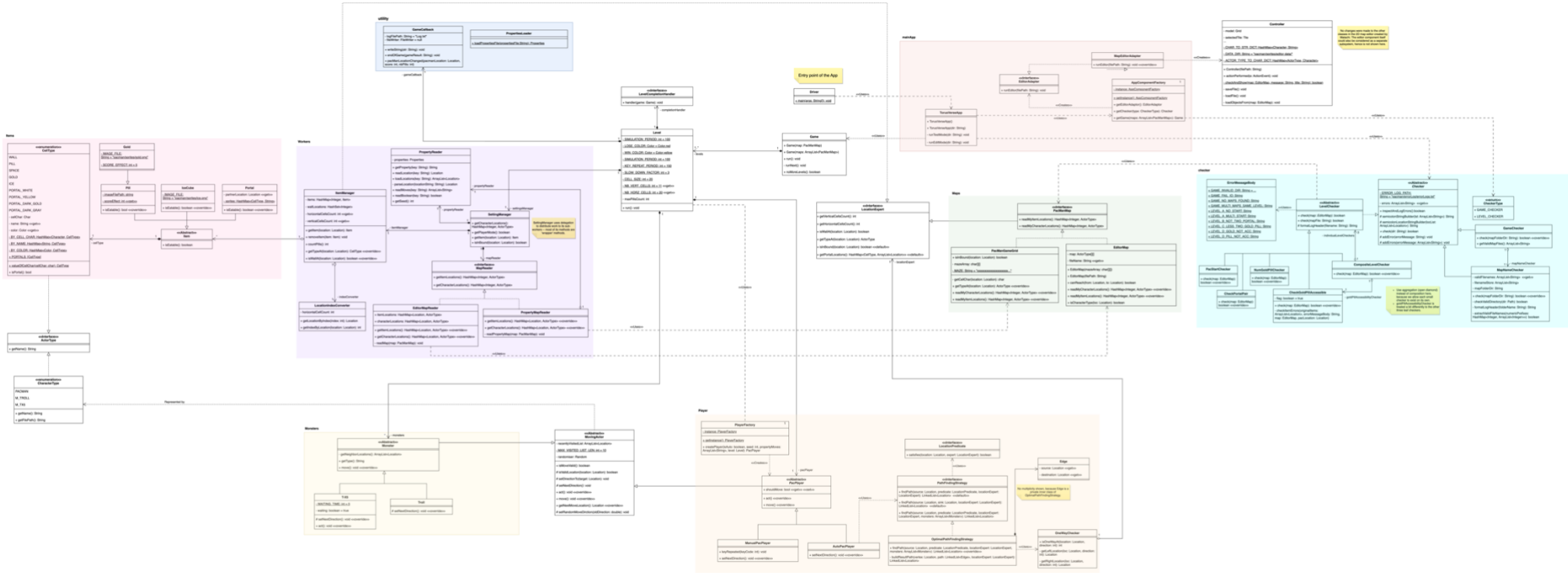

Figure 3.1: A Design Class Diagram documenting the design relating to the auto-player and game levels/maps for PacMan in the TorusVerse. Some less important methods (e.g. private, getter, setter methods) and dependencies are omitted for brevity.

** The page limits the resolution of the diagram. For a clearer version, please see documentation/StaticDesignModel.pdf.*

3.2: Additional Diagrams

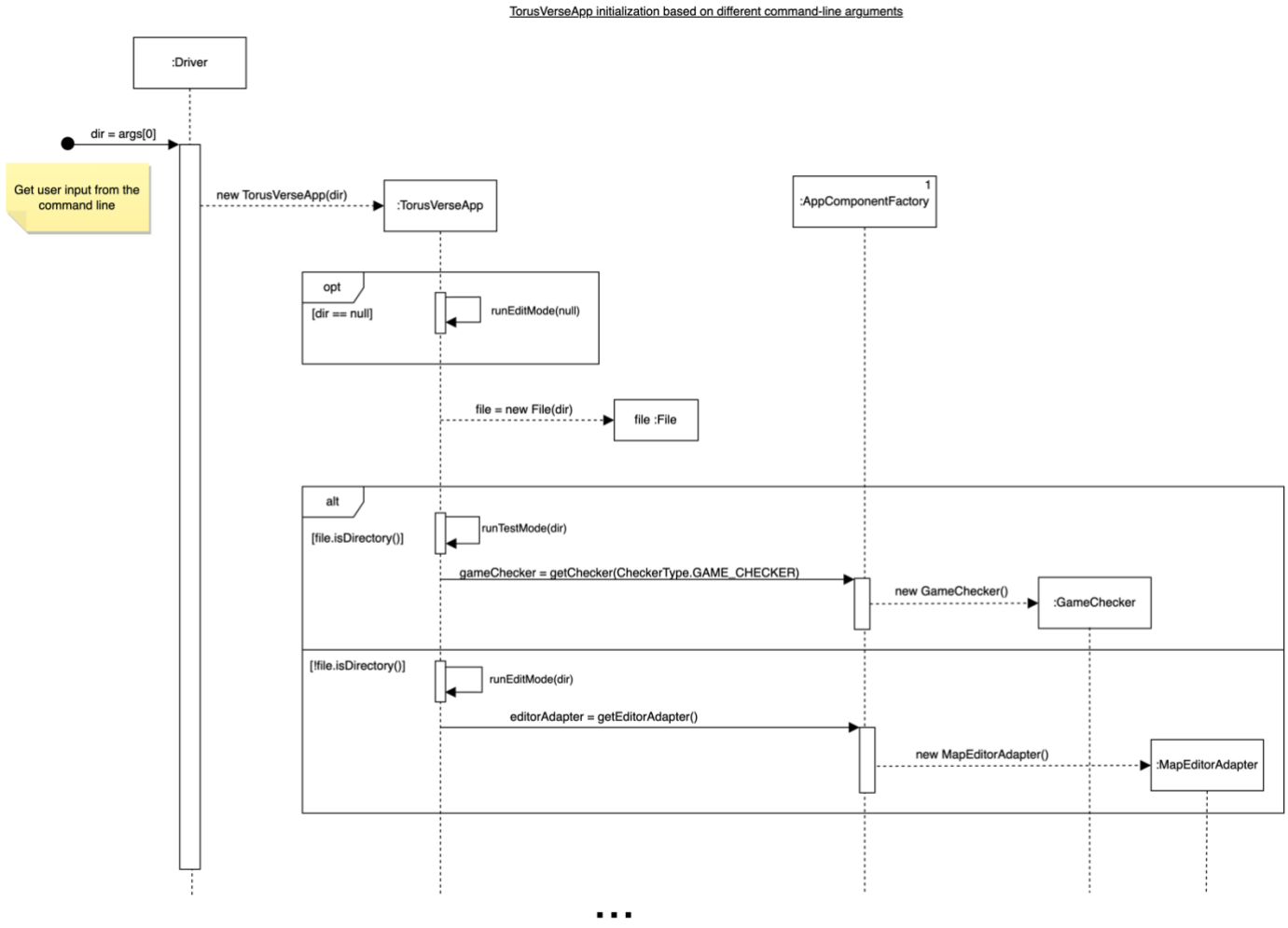

Figure 3.2: A Design Sequence Diagram showing the initialization of `TorusVerseApp` based on different command-line arguments from the user.

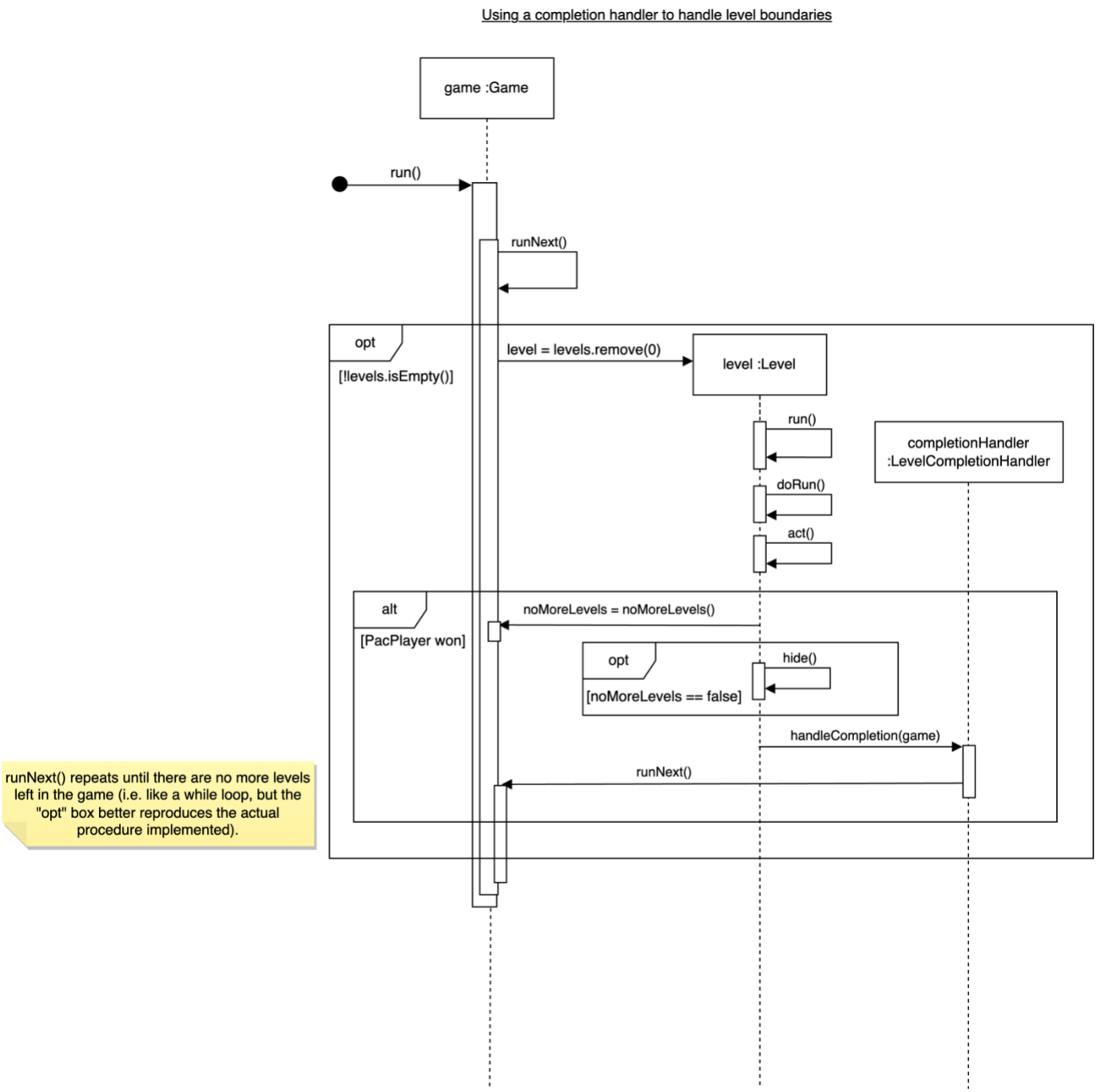

Figure 3.3: A Design Sequence Diagram showing the use of `LevelCompletionHandler` to handle level boundaries.

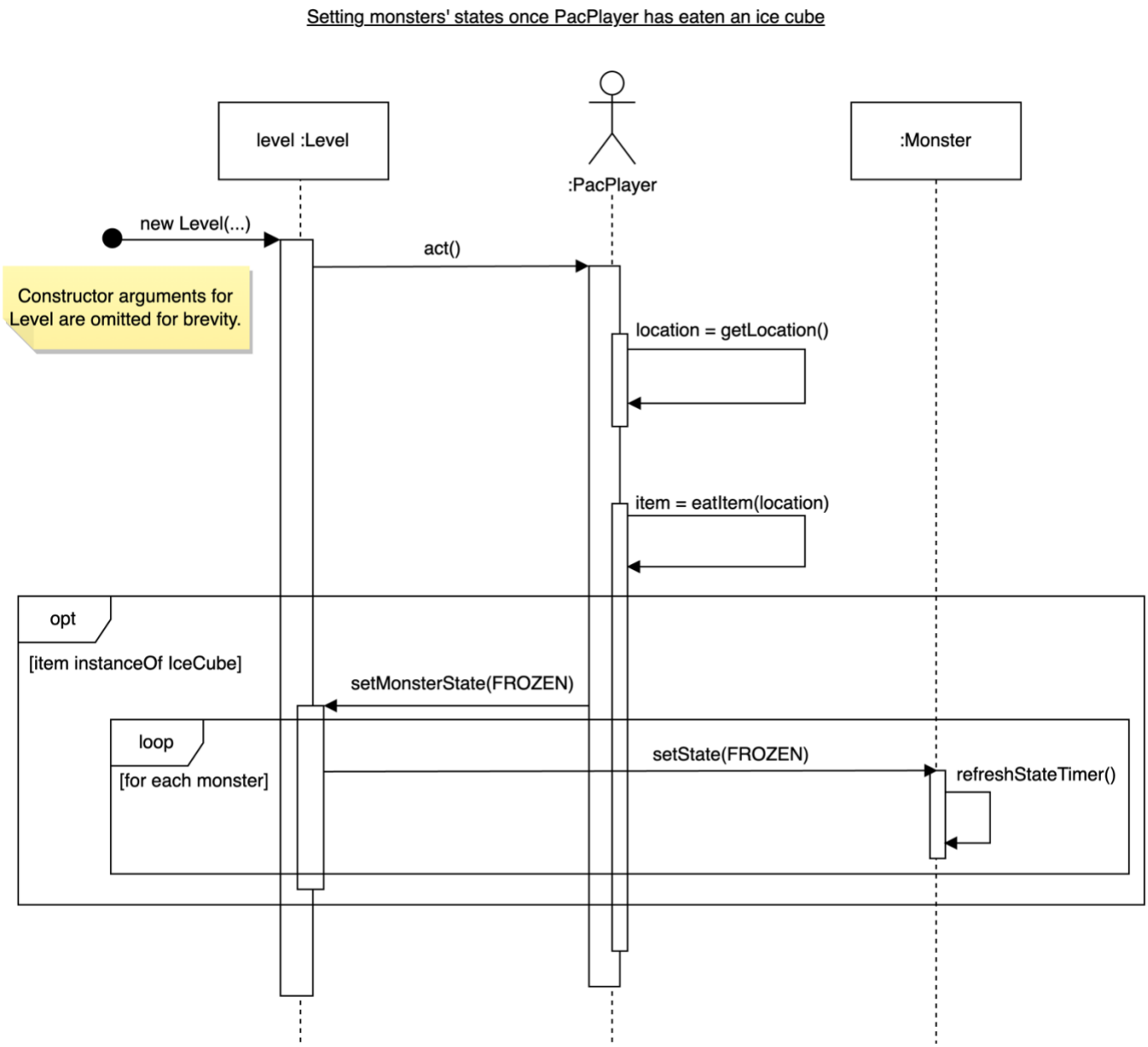

Figure 3.4: A Design Sequence Diagram showing the (proposed) effect on `Monsters` once `PacPlayer` has eaten an `IceCube`, for the extension in Section 2.2.

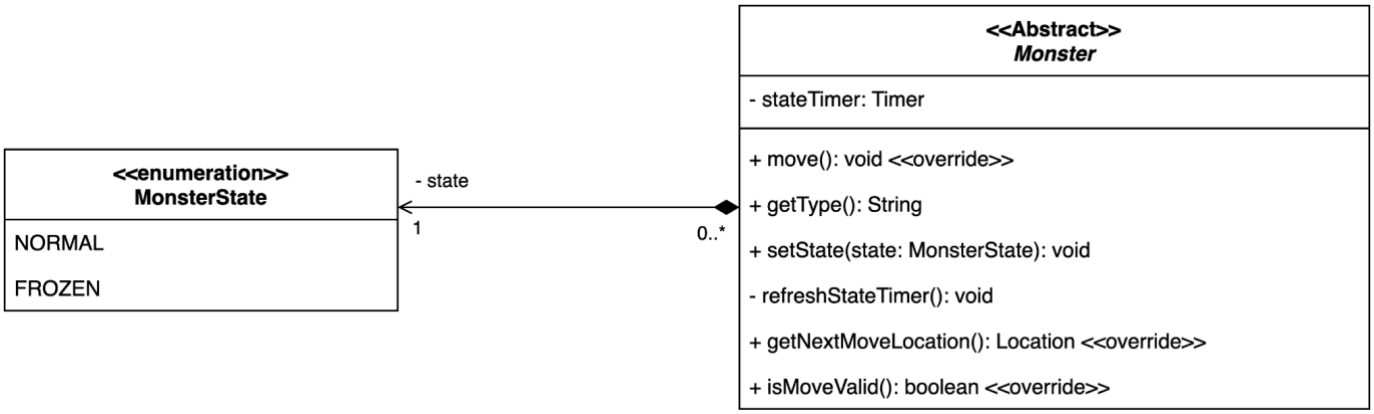

Figure 3.5: A partial Design Class Diagram showing the possible structure of the `Monster` class, under the proposed solution for the extension in Section 2.2.

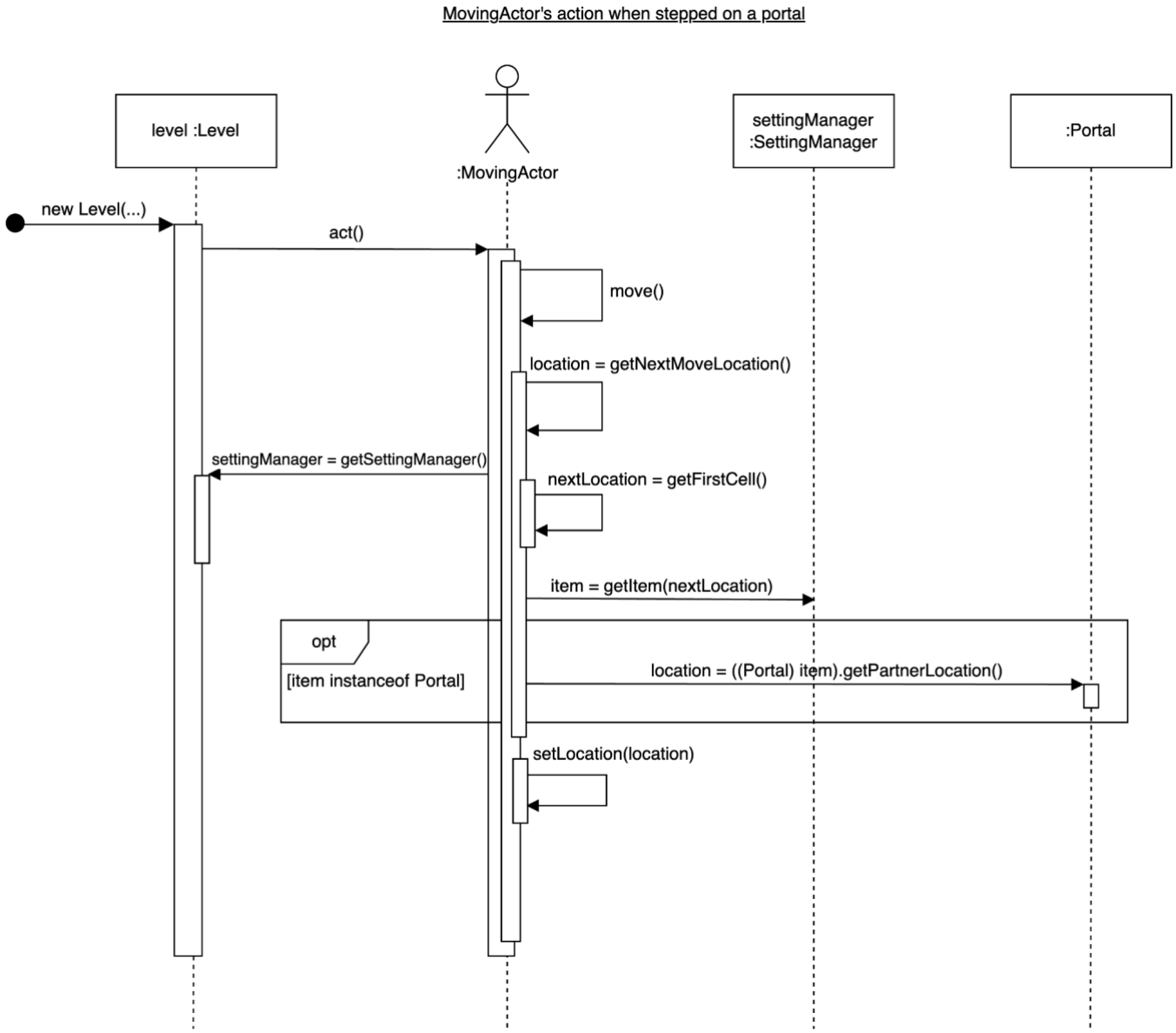

Figure 3.6: A Design Sequence Diagram showing the process after a `MovingActor` steps on a portal.