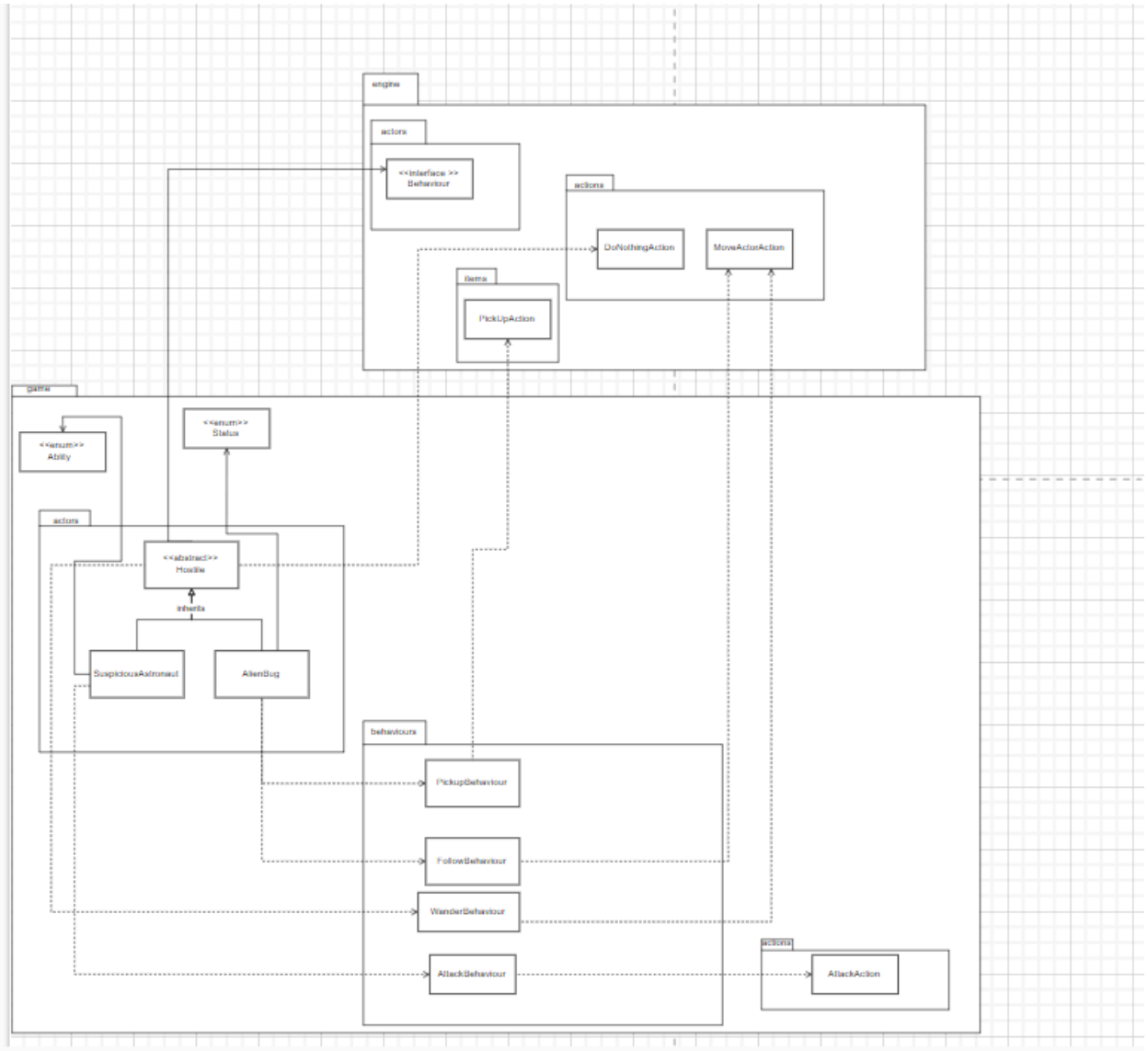
Assignment -2
Requirement 2

UML diagram



DESIGN GOAL

The objective of this task is to create distinct behavioral patterns for the AlienBug and SuspiciousAstronaut characters within a gaming environment. These behaviors should enhance gameplay and adhere to the principles of object-oriented programming

ALIEN BUG class inherits from hostile class and the main aim is to prioritize behavior where the bug first picks up the scrap in its current position , then follows the player/intern when it is one exit away and finally wanders around .

Suspicious astronaut class inherits from hostile class and its main goal is to attack the player with maximum damage which instantly kills them

Design decision

The AlienBug is an HOSTILE game creature that adapts its actions based on the game's surroundings. It builds upon the Hostile class and utilizes a Map<Integer, Behaviour> to handle its behaviors, enabling it to react dynamically as the game situation changes

AlienBug is a subclass of the Hostile class, allowing it to use similar hostile traits like health and damage levels, as well as common actions such as attacking other characters. This helps cut down on repetitive code.
To handle its behavior dynamically , AlienBug uses a Map<Integer, Behaviour> to store different behaviors according to their priority. This enables AlienBug to adjust its actions based on specific situations, like collecting items or following other characters, without changing its fundamental logic structure.
Each distinct action such as FollowBehaviour or PickUpBehaviour is contained in separate classes that follow the Behaviour interface. This approach complies with the Single Responsibility Principle, guaranteeing that each behavior class handles a particular AlienBug action. This simplifies code maintenance and expansion.

AlienBug's design enables it to adjust its behaviors based on interactions with other game actors. For example, when encountering an actor with the Status.HOSTILE_TO_ENEMY capability, it dynamically adds a FollowBehaviour to its behavior map. This feature improves AlienBug’s ability to react to changes in the game environment, making it a more interactive and unpredictable opponent.

When specific actions are needed, like when the AlienBug dies, the class changes inherited methods like unconscious() to do custom things, such as dropping items. This way, it makes sure that the AlienBug's distinct characteristics and responses really come through in how the game works.
Overall , the design of the AlienBug class cleverly brings together inheritance, interface implementation, and dynamic behavior management to make a challenging and interesting enemy in the game that can be easily adapted and maintained.

The SuspiciousAstronaut class inherits standard hostile attributes and behaviors from the Hostile class. It is also endowed with exclusive restricted entry capability (Ability.RESTRICTED_ENTRY), which influences its interactions with players and other entities in the game environment. By extending the Hostile class and incorporating this custom feature, the SuspiciousAstronaut class maintains clarity in its purpose while enabling potential modifications or expansions to its behavior without directly altering existing code. This design choice aligns with SOLID principles, particularly demonstrating adherence to the Single Responsibility Principle by encapsulating the suspicious astronaut's behavior within its own class and ensuring clear definition within the game architecture. Furthermore, it upholds the Open/Closed Principle by facilitating addition of new capabilities or `behaviors to↵the SuspiciousAstronaut class without necessitating changes to existing code, thereby promoting flexibility and maintainability in the game's design

Alternate design
An alternate method could include developing distinct classes for each character without deriving from a shared superclass. This would permit individual modification and development of each class without impacting the others.

```
public class AlienBug { private String name; private char displayChar; private int hitPoints; ... }
public class SuspiciousAstronaut { private String name; private char displayChar; private int hitPoints; ... }
```

The alternative design does not align with several Design and SOLID principles
1. **DRY: -** This approach involves duplicating similar code across multiple classes, which heightens the risk of errors and complicates maintenance of the codebase.
2. **Open/Closed Principle:-** Extending functionality would necessitate modifying existing classes instead of extending them, conflicting with the principle of being open for extension but closed for modification.
3. **Single Responsibility Principle:-** Failure to separate functionalities (common vs unique) results in each class having more than one reason to change, leading to complications in updates and testing.

Principles used

SRP: Single Responsibility Principle
Application: The SRP is followed in the AlienBug class by encapsulating behaviors in specific classes that implement the Behaviour interface. Each behavior, such as PickUpBehaviour and FollowBehaviour, is responsible for managing a different aspect of the AlienBug's functionality. The SuspiciousAstronaut class is designed with a single responsibility: representing a unique type of hostile entity with specific capabilities (Ability.RESTRICTED_ENTRY)
Why: Adhering to the SRP is crucial for maintaining a clean and modular codebase. By assigning each behavior to a separate class, the AlienBug class becomes more focused and easier to understand, reducing complexity and enhancing maintainability.
Adhering to the SRP ensures that each class has a clear and focused purpose, which promotes maintainability and reduces the risk of bugs or unintended side effects.
Pros/Cons: The main advantage of following the SRP in this context is the clarity and simplicity The primary benefit is improved code organization and clarity, as each class has a clearly defined purpose. This approach also facilitates code reuse and promotes easier testing. However, the downside is the potential proliferation of small, specialized classes, which can increase the complexity of the overall system if not managed effectively.

OCP IN ALIEN BUG

APPLICATION : The AlienBug class adheres to the Open/Closed Principle by dynamically managing behaviors with a Map<Integer, Behaviour>.

The SuspiciousAstronaut class upholds the Open/Closed Principle by leveraging capabilities and potential polymorphism to override a method in the base class.

Why? This approach enables the addition of new functionalities without making changes to the current source code, supporting flexibility and ease of maintenance. As well as prioritizes which behavior the alien bug should implement first

By leveraging capabilities and inheritance, additional functionalities can be incorporated without modifying the current codebase, fostering flexibility and expandability.

Pros / cons : Benefits involve the ability to easily incorporate new behaviors and reducing the need for extensive code changes, but it can lead to challenges in handling dynamic behavior mapping.

Connsance

Application The AlienBug class reduces connascence by employing interfaces such as Behaviour and abstract classes like Actor. More specifically, FollowBehaviour engages with the generic methods of GameMap and Actor without requiring specific implementation knowledge.

The Suspicious Astronaut class avoids connascence by inheriting from the Hostile abstract class and utilizing well-defined interfaces for communication. This guarantees that the astronaut class follows the anticipated behaviors and characteristics specified by Hostile, making use of inherited methods

Why ?By using interfaces to reduce connascence, the components of a system become less dependent on each other, aiding in easier maintenance and scalability due to the increased decoupling.

By using inheritance in this manner, it standardizes interactions among various classes that have a common base. This ensures uniform behavior and lowers the risk of errors resulting from inconsistent implementations.

Pros /cons The main advantage is the decrease in direct connections between components, which improves flexibility and simplifies testing. However, a possible downside is that it may lead to more complicated design and require more comprehensive documentation for understanding the system's structure.

This method makes the development process simpler by reusing code and ensuring a consistent interface. However, it may create challenges with adaptability, as any changes to the standard behavior may necessitate further adjustments or restructuring, potentially resulting in inflexible class setups.

Future extensions

Integration of New Features: When the game introduces new character types or items, we can seamlessly add new behaviors as separate classes without having to change the existing AlienBug class. This approach follows the Open/Closed Principle.

Prioritizing Behaviors with Logic: We could extract the logic used for prioritizing behaviors into a separate strategy pattern. This would enable us to use different prioritization algorithms depending on the specific game scenarios.

Conclusion

Overall, our chosen design provides a robust framework for achieving the assignment objective. By carefully considering design principles, requirements, and constraints, we have developed a solution that is efficient, scalable, and maintainable, paving the way for future enhancements, extensions, and optimizations. This approach not only improves the gameplay experience but also aligns with best practices in software design