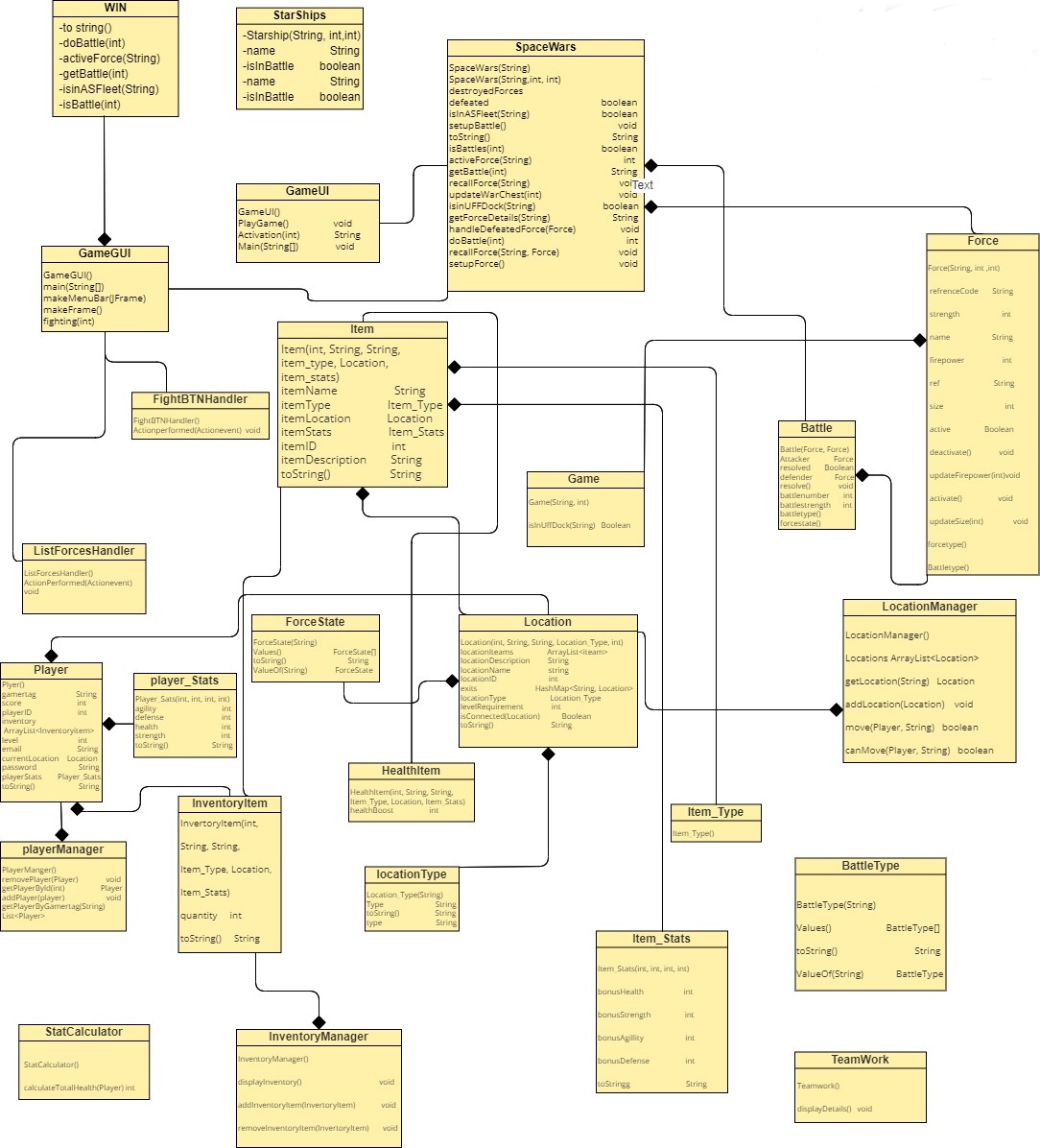
**WIN Technical Report**

**Aim:**

The primary objective of this game is to win battles and increase the number of coins in the war chest. To activate different forces in the Active Star Fleet (ASF), players must pay an activation fee. These forces are then available to engage in battles with opponents. If the player is victorious, the winnings are added to the war chest. However, if the player loses a battle, the losses are subtracted from the war chest and may result in the destruction of a force if it has lost all its strength. Therefore, players must make strategic decisions to succeed in the game. This report will provide an overview of the significant design decisions taken by our team during the game's development. Moreover, we will present a UML diagram that illustrates the class relationships.

**UML diagram:**

**Design decision 1: Unit class Inheritance:**

Our team decided to utilize class inheritance to establish various types of units in the game. Initially, we created a base Unit class that encompassed common properties and methods, and then generated specific unit classes that inherited from the base class.

Several alternative implementations were considered, including using composition to create different unit types or utilizing interfaces to define common properties and methods for units.

The benefits of inheritance, however, ultimately proved more compelling. For instance, using inheritance simplifies adding new unit types in the future, as any new classes will only need to inherit from the base Unit class and override any necessary methods. Furthermore, inheritance enables a clear hierarchy of classes, making the code more organized and easier to comprehend.

Nevertheless, inheritance has potential drawbacks. A complex and rigid class hierarchy may emerge, making it challenging to modify or extend later. Additionally, a subclass may inherit methods that are not required, resulting in unnecessary code bloat and reduced performance.

Nonetheless, our team's decision to use class inheritance was justified based on the clarity and organization it provided in representing different unit types. The benefits of code reuse and extensibility outweighed potential drawbacks and aligned with sound object-oriented design principles.

**Design decision 2: Using the Factory pattern to create new Forces:**

Our team decided to employ the Factory pattern to create new Forces in the game, providing a centralized point for object creation and affording flexibility in the creation process. Our implementation of the Factory pattern utilizes the Force class to generate new Force objects based on the type of force specified by the player.

While an alternative to the Factory pattern is the Prototype pattern, which clones existing Force objects to create new ones, this approach may be less flexible since it relies on existing objects. However, it can be simpler to implement and potentially have better performance, as new objects are not created from scratch.

The benefits of utilizing the Factory pattern include a simplified codebase with reduced duplication and the ease of adding new types of forces in the future. In contrast, potential drawbacks include increased complexity in implementation and more memory usage since new objects are created each time a Force is generated.

Ultimately, the team decided on the Factory pattern as it provides more flexibility and simplified future force addition, outweighing any added complexity. We determined that the additional complexity of implementing the Factory pattern was manageable and would not adversely affect game performance.

**Design decision 3: Using an ASF class to manage active forces:**

Our team decided to implement an Active Star Fleet (ASF) class to manage the active forces in the game. This dedicated class keeps track of which forces are currently active and handles the activation and deactivation of forces. Additionally, our ASF class provides an interface for checking the status of each active force, making it easier to implement certain game features.

An alternative approach to managing active forces is using a list instead of a dedicated ASF class. While this method may be simpler to implement and potentially have better performance by not managing an additional object, it may be less flexible than using a dedicated ASF class as it could be challenging to track each active force's status.

Benefits of using the ASF class include providing a centralized point for managing active forces, simplifying the code and reducing duplication. Additionally, it offers a convenient interface for checking the status of each active force, making it easier to implement certain game features. Potential drawbacks of using the ASF class include a complex and rigid class hierarchy that may be challenging to modify or extend later, but this can be managed with careful planning.

Ultimately, our team chose to implement the ASF class as it aligns with our project's object-oriented design principles and provides a clear and organized representation of active forces. The benefits of the ASF class, such as code reuse and extensibility, outweigh potential drawbacks, and the added complexity of implementing the ASF class is manageable without adversely affecting game performance.

**Conclusion:**

In conclusion, the design decisions made by our team during the implementation of the game were carefully considered and aimed at achieving the ultimate goal of winning battles and accumulating coins in the war chest. Our choice to utilize class inheritance, the Factory pattern, and the ASF class were all justified based on their benefits of code reuse, extensibility, and organization, even though they may have had potential drawbacks. Our team managed to strike a balance between complexity and simplicity to deliver a well-designed game that aligns with sound object-oriented design principles. We are confident that our decisions will contribute to the success of the game and enhance the player's overall experience.