# 4.1 Object Design Trade-Offs

This section defines the different tradeoffs in object design and the need to make decisions about them. In the making of a text-based computer game, aspects such as memory usage, availability, durability and maintainability are important. Compromises need to be made from these in order to come up with the optimal design.

1. Memory space vs. response time

Text-based games often need a good share of memory even though no graphics are involved. As more memory is required to store all the user’s data, response time may deteriorate; this, however, should not happen. The management classes involved in sending and receiving data, in turn, take care of this problem by limiting data transfer to only when it is required.

1. Buy vs. build

Usually, a build policy may be sought because of the simple design of most objects. The parser component, however, may be supplied from elsewhere, which will in fact save time. The compromises needed to be made here is between cost and time, and one component may be bought if the cost is not too high, and if it will save time.

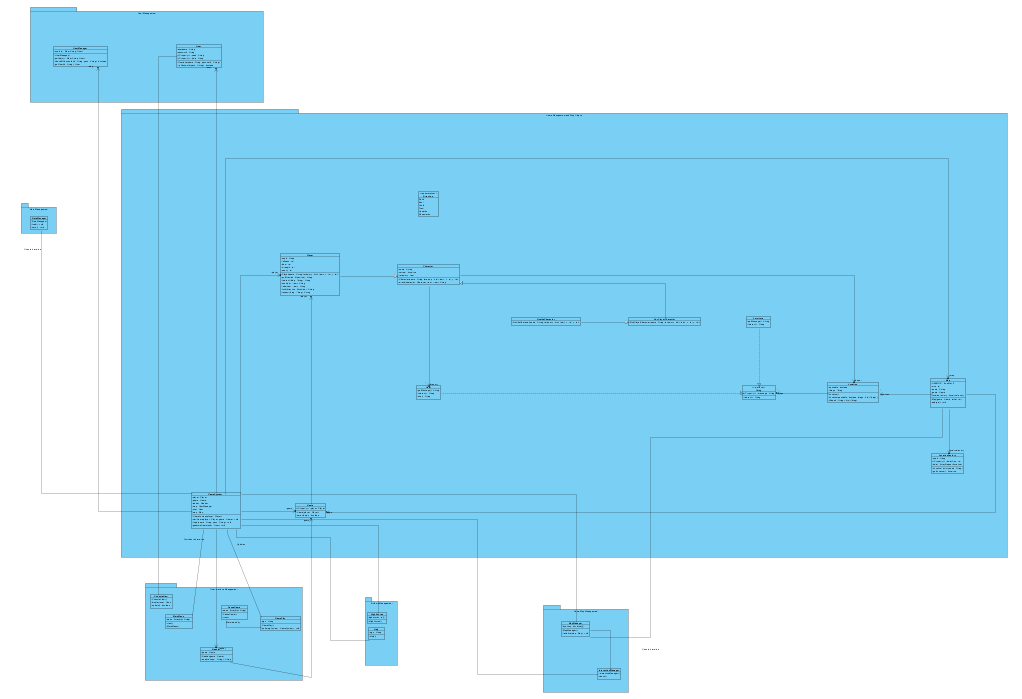
1. Platform dependence vs. flexibility

The game needs an optimal platform, such as the operating system and even architecture. This optimal platform will allow perfect programmability for the game, but it also needs to be flexible, such that only subtle changes to the design will grant it new platform dependency properties. Most games in the market are optimized for one platform, however since this is a text-based game, the switch between platforms is as simple as an integrated pattern.

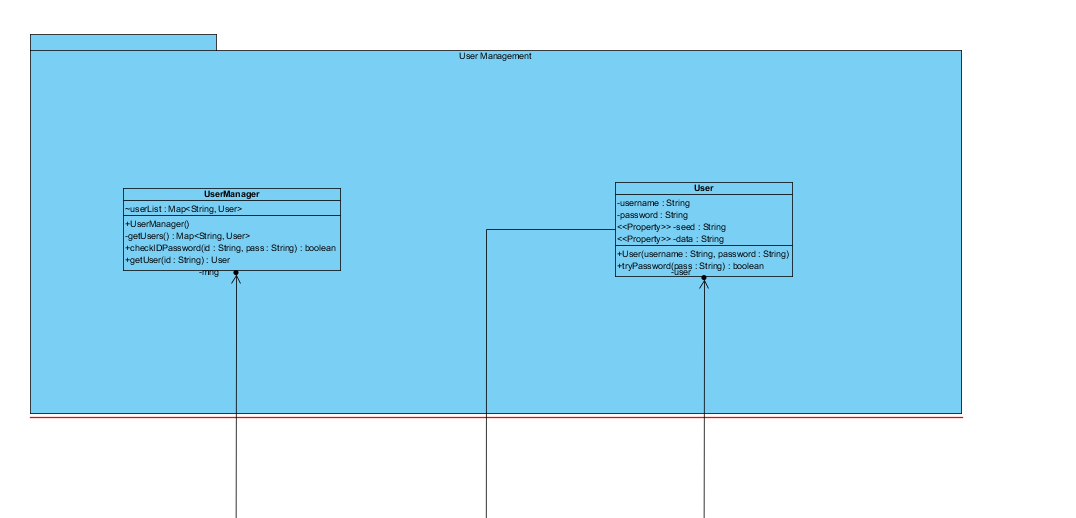
1. Design patterns

Several design patterns have also been compared in terms of tradeoffs. The Bridge pattern was seen to provide more flexibility in terms of modifying existing code. Examples of this pattern can be seen in the figure provided in the following page. Since a general algorithm was not required for the game, a strategy pattern was not considered. In fact, several design patterns were used in different subsystems. These subsystems and the design patterns will be discussed in the upcoming section.

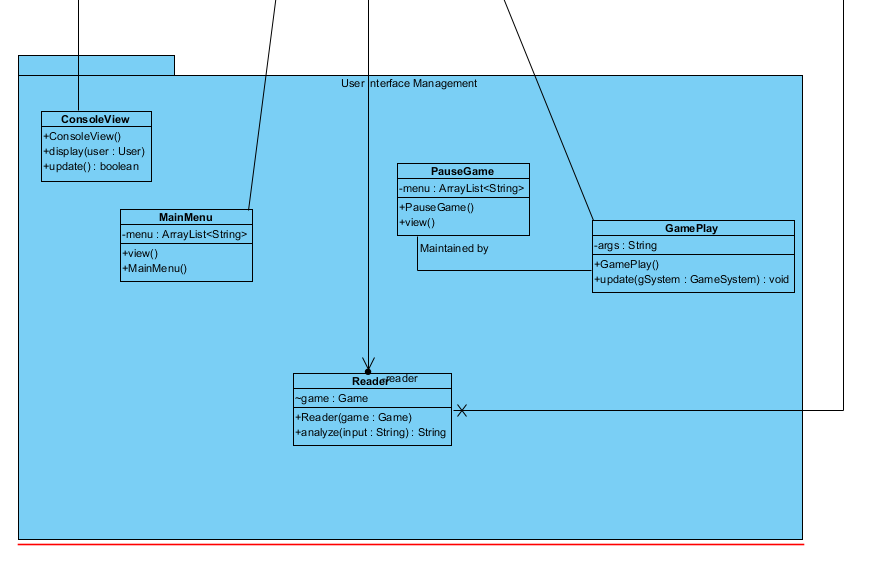
# 4.2 Final Object Design

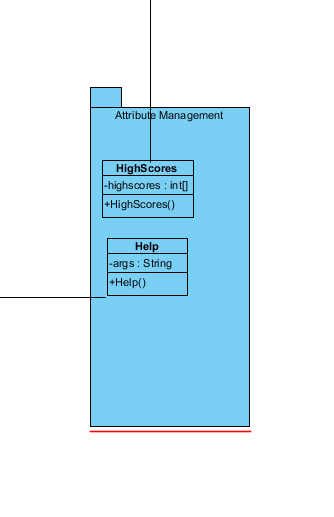
 As the final object design is too large to make every component legible, the document will display the object design as composed of its subsystems.

*Figure shows object diagram*



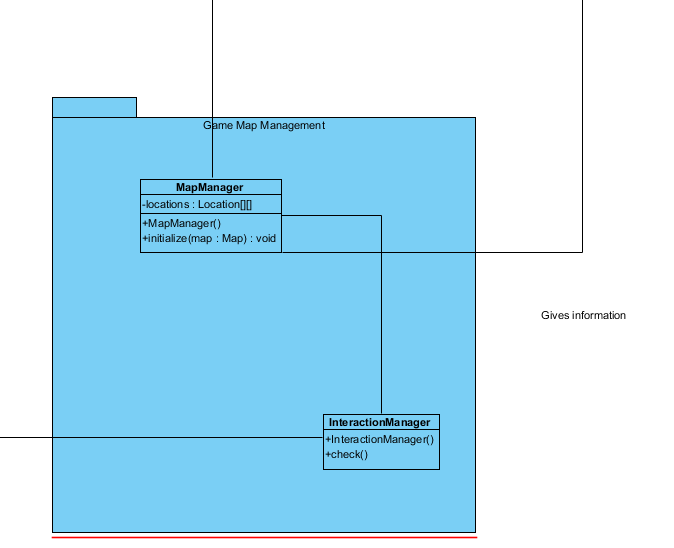
*Figure shows User Management Subsystem*

**

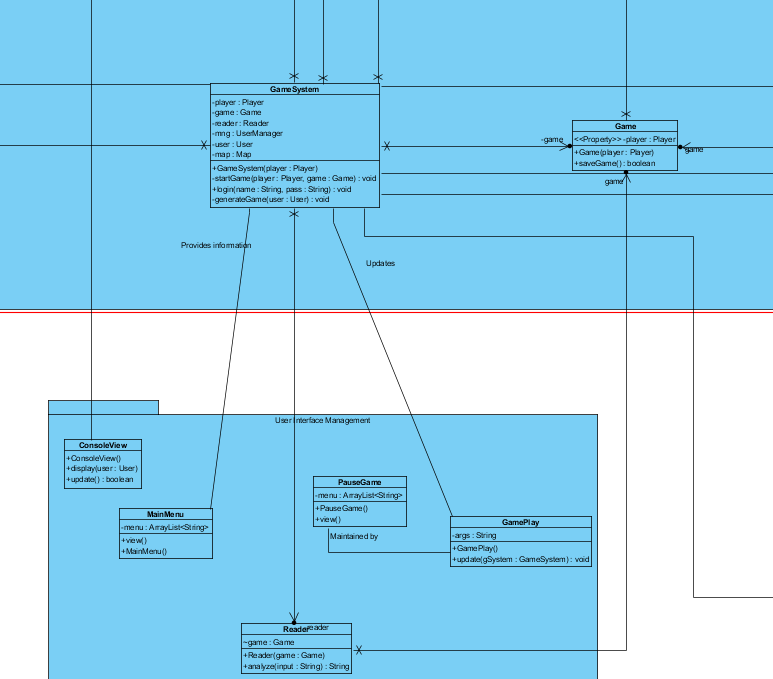
*Figure shows UI Management Subsystem*

*Figure shows Attribute Management*

*Subsystem*

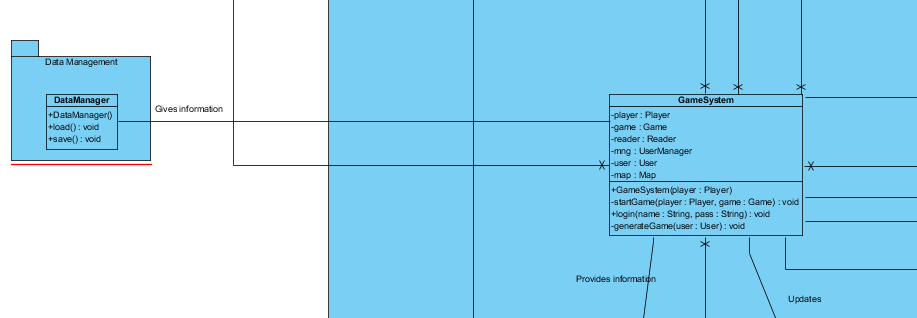
**

*Figure shows Game Map Management Subsystem*

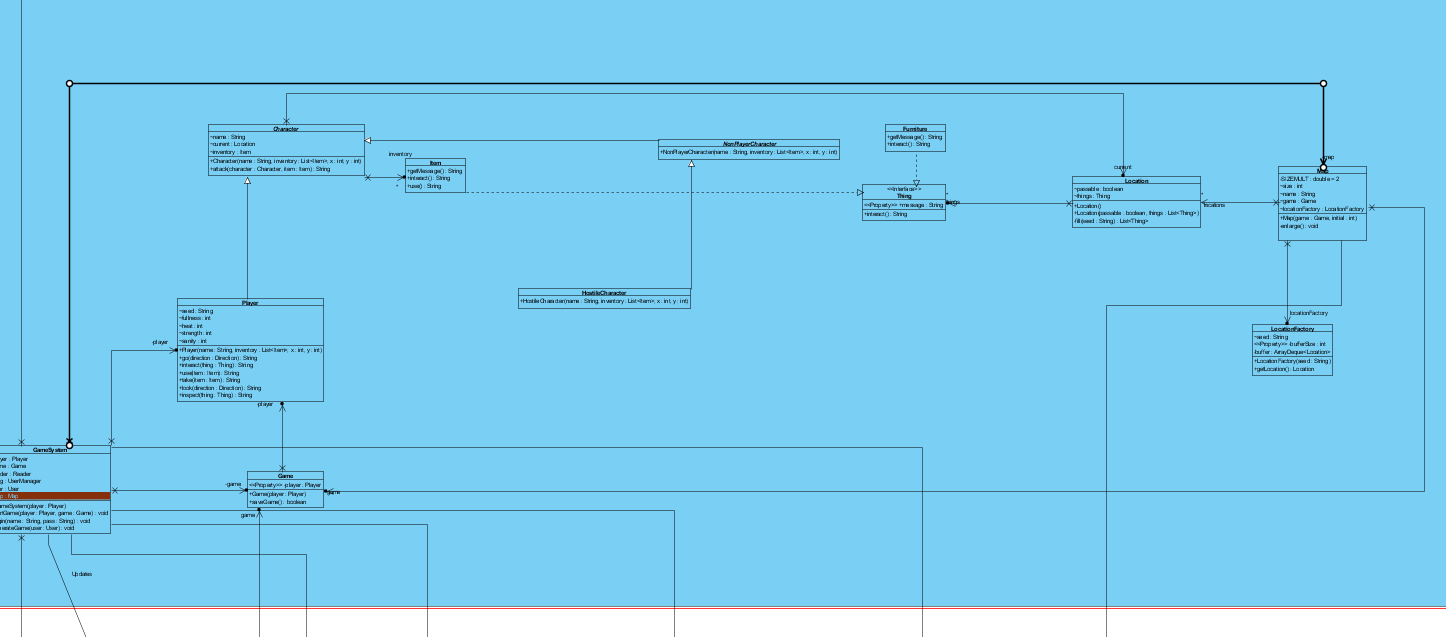
**

*Figure shows Game Management Subsystem and its relations with UI Management Subsystem*

At this point one can see that the Bridge pattern has been used; new code may be written for classes in the UI Management subsystem flexibly. Also, note that delegation has been used before inheritance.



*Figure shows the interactions of the Data Manager subsystem with class GameSystem*

**

*Figure shows the remaining objects and their interactions with GameSystem.*

This part of the design was done choosing Abstract Factory design, hence the multiple inheritance relationships. The GameSystem class provides methods to classes like Map and Player. This design was superior to others in the sense that it allows compatibility; there is a single solution for implementing the methods of these classes. It also allows quick modifications.