EXERCISE 1

In the first exercise, the two principles that affected our code were:

The Open-Closed Principle (OCP)

In object-oriented programming, the open-closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behavior to be extended without modifying its source code.

The Dependency Inversion Principle (DIP)

In object-oriented design, a dependency inversion principle is a specific form of loosely coupled software modules. When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details.

It is prominent when looking at the classes that implement the interface TicketManager: following the open-closed principle, the interface is open to extension. Any additional parameter added to the class Ticket can be easily adapted by adding a new implementation without modifying any of the source code. Also, it respects the dependency inversion principle because the high-level function searchTicket, it’s independent of its actual implementation in SearchEngine.

In the first exercise, the design pattern we followed in our code was:

Composite pattern

The composite pattern describes a group of objects that are treated the same way as a single instance of the same type of object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.

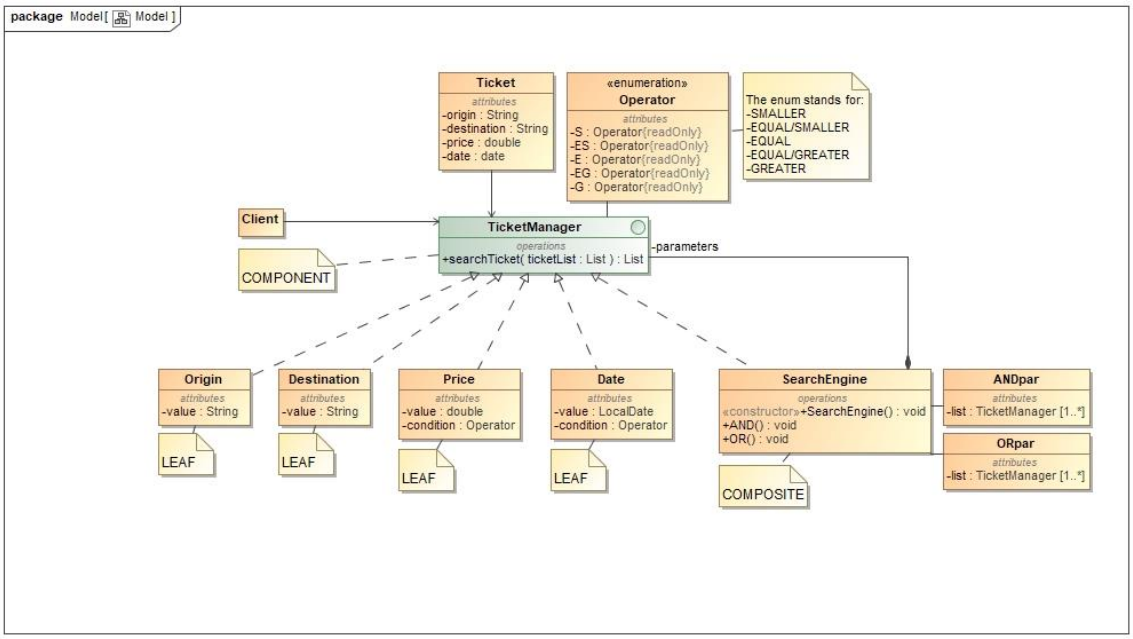

Figure 1.0

This pattern enables clients to work through the interface to treat any searchTicket implementation and the SearchEngine class uniformly: the implementations only perform the search in the list, while the class SearchEngine saves all the search the client wants to perform and forward the requests to their child components recursively to obtain the desired result. This makes client classes easier to implement, change, test, and reuse.

For the dynamic diagram, we chose the sequence diagram.

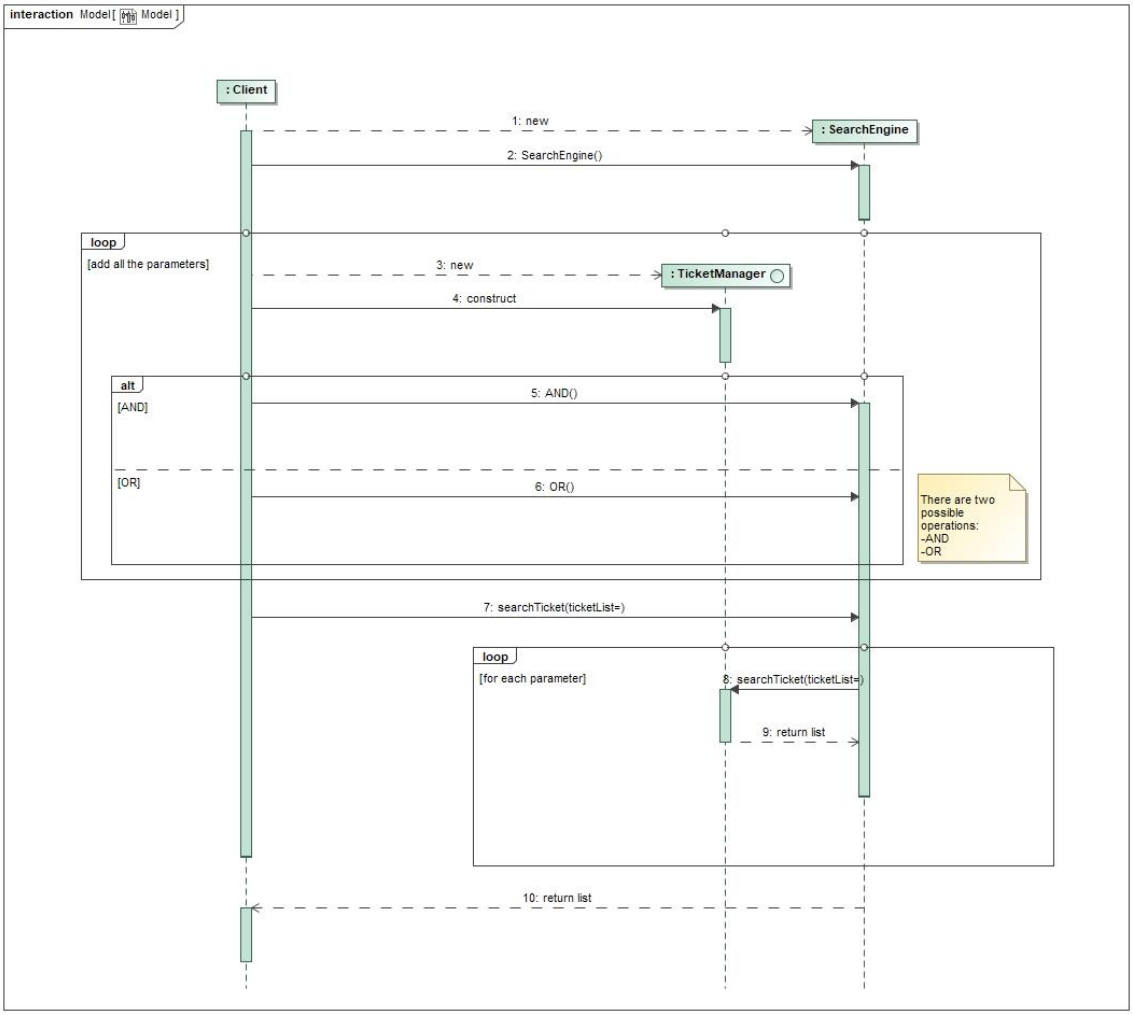

Figure 1.1

It depicts the objects involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario.

Because we thought that to depict the inner working of the code and its exchange between classes and implementations it would be best to show the function call and values returned for by each.

EXERCISE 2

In the first exercise, the two principles that affected our code were:

The Open-Closed Principle (OCP)

In object-oriented programming, the open-closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behavior to be extended without modifying its source code.

The Dependency Inversion Principle (DIP)

In object-oriented design, a dependency inversion principle is a specific form of loosely coupled software modules. When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details.

Both are visible in the implementation of the interface GraphIterator, it follows the open-closed principle as it is open to extensions such as introducing another sorting algorithm, while its source code remains closed to modification, and it also follows the dependency inversion principle as the interface is completely independent of its implementations, leaving the details to the implementation of the high-level function traverseGraph.

In the second exercise, the design pattern we followed in our code was:

Strategy pattern

The strategy pattern is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.

This pattern lets the algorithm vary independently from clients that use it.

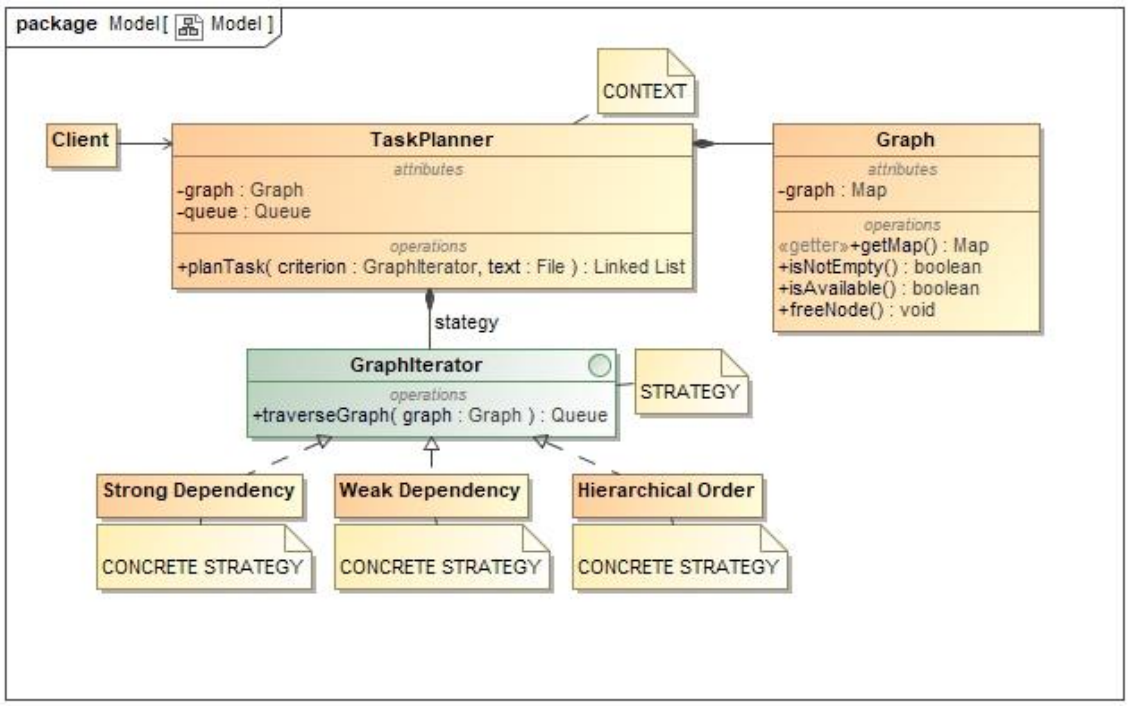

Figure 2.0

This pattern allows the client to choose what implementation to use in the program and to modify it on runtime, by changing which GraphIterator implementation is used.

For the dynamic diagram, we chose the sequence diagram.

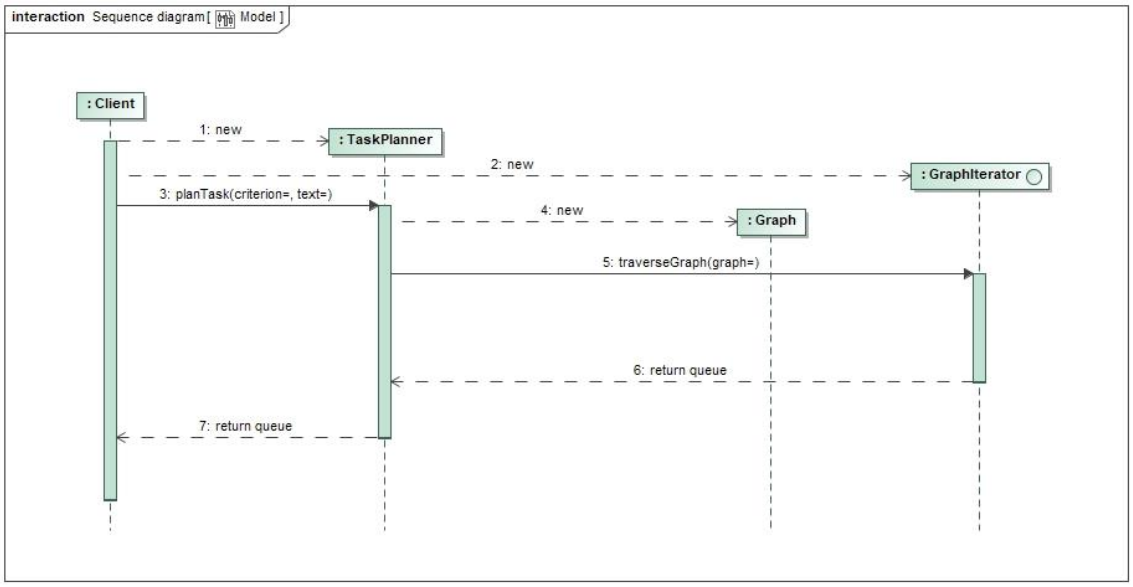

Figure 2.1

It depicts the objects involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario.

Because we thought that to depict the inner working of the code and its exchange between classes and implementations it would be best to show the function call and values returned for by each.

This is a simplified representation of the basic working of the code without entering in-depth about each algorithm. But we created more complex diagrams that represent each one of them.

Strong Dependency

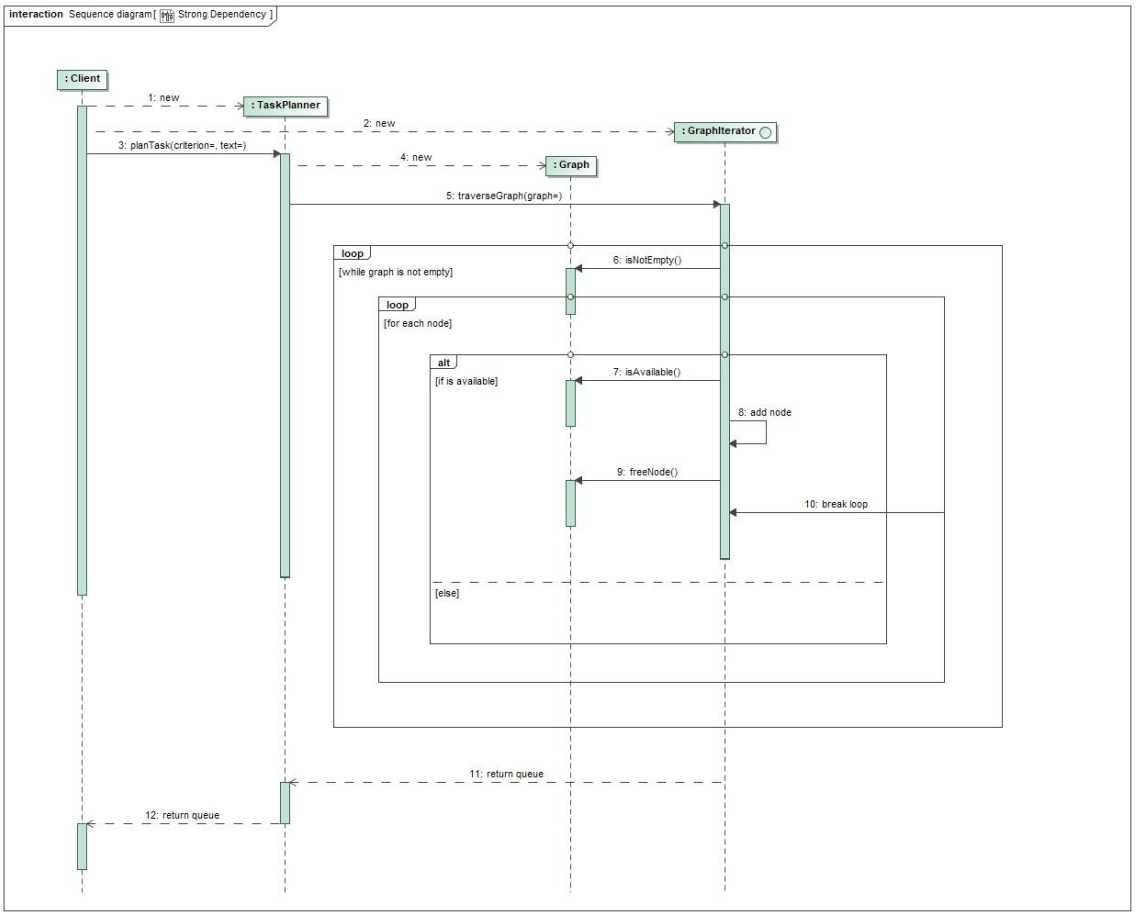

Figure 2.2

Weak Dependency

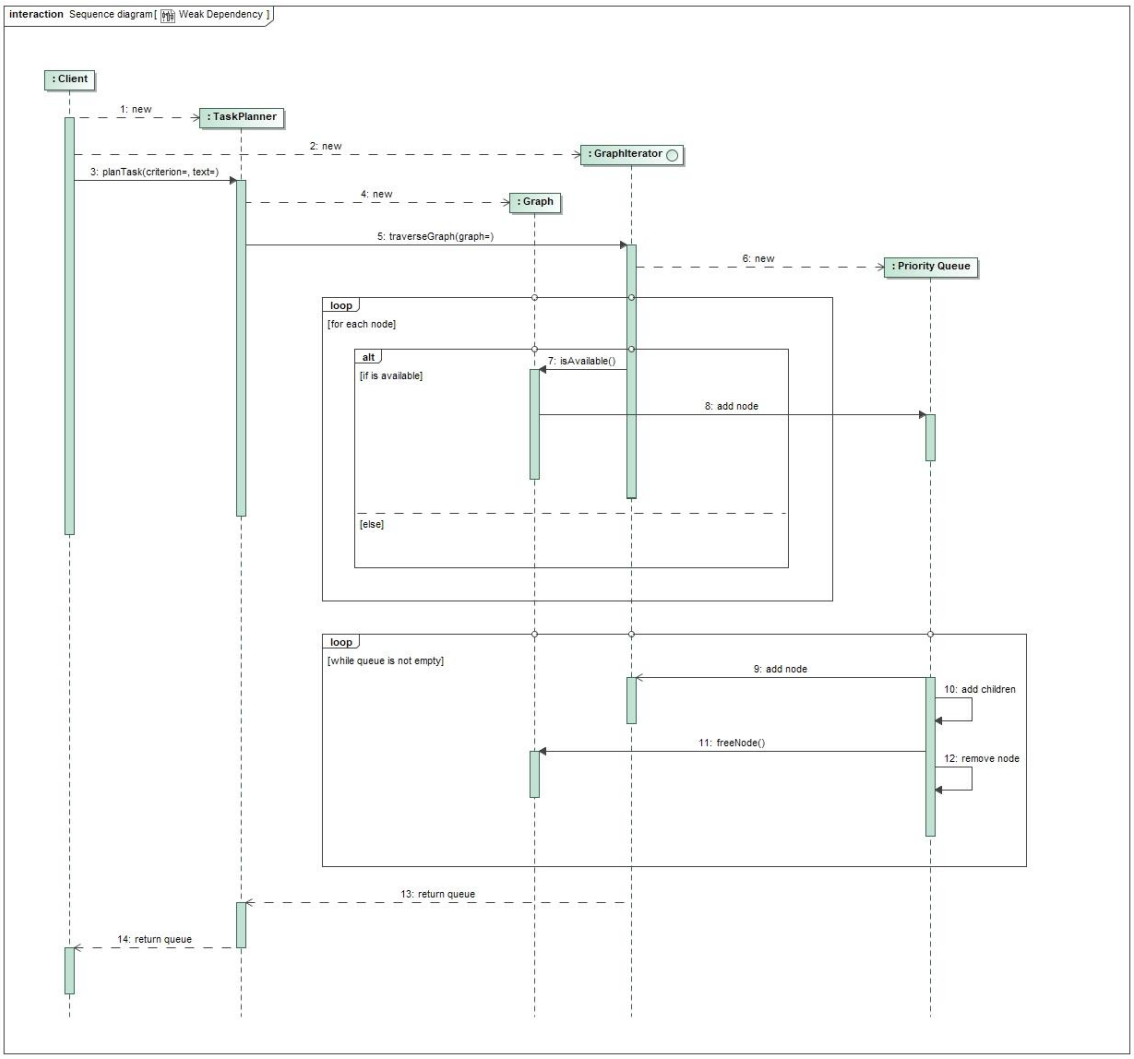

Figure 2.3

Hierarchical Order

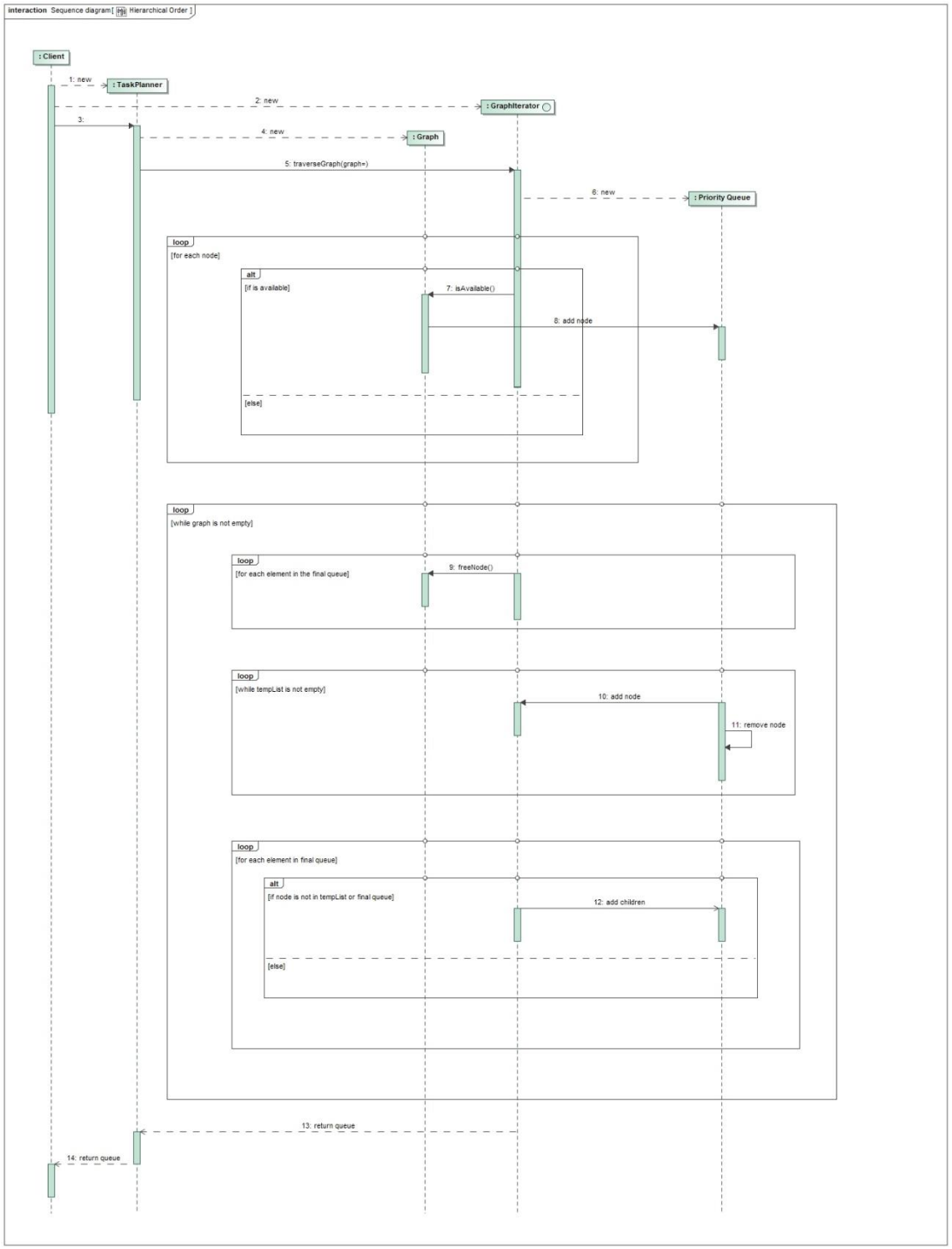

Figure 2.4