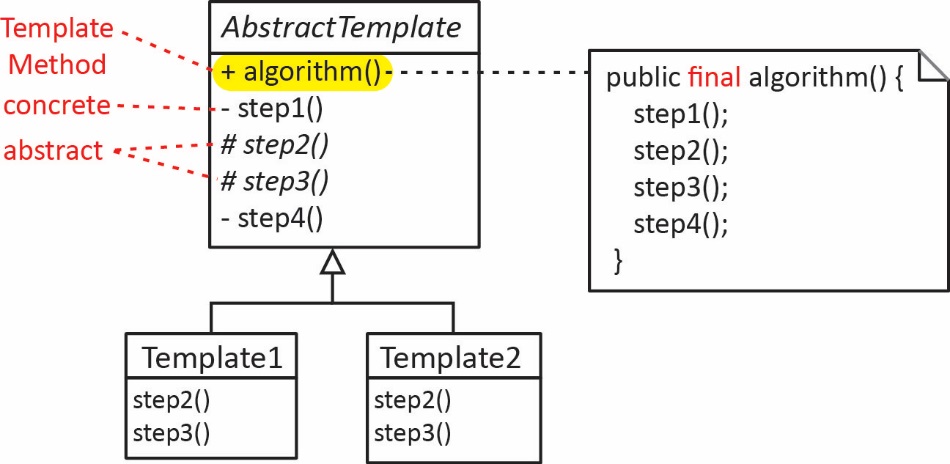
**Chapter 8 Notes – Template Method Pattern**

**General Solution**

1. The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure. [HFDP]

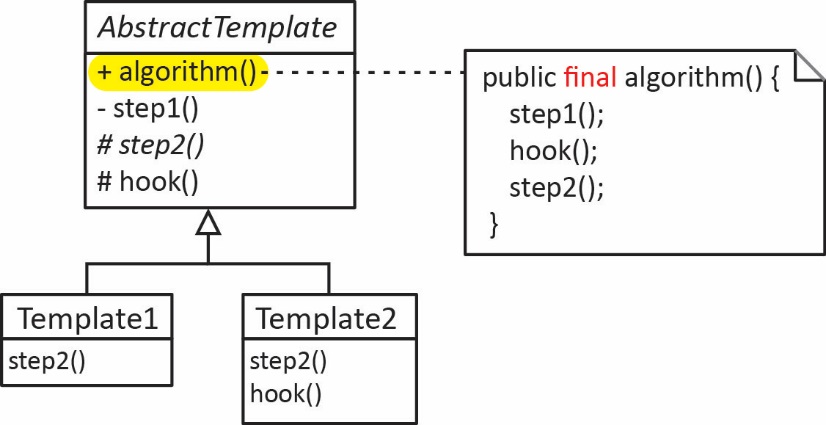
In other words, the Template Method Pattern is useful when we want to make the steps in an algorithm concrete, but allow the implementation of individual steps to vary.

1. The solution is to make the template method, *algorithm final* and the steps in the algorithm that can vary, *abstract*. Then, sub-classes can implement the abstract steps.



1. How is Template Method different from Strategy?

* Template Method – encapsulates the steps in an algorithm by using sub-classing.
* Strategy – encapsulates an algorithm through composition allowing clients to swap algorithms at runtime.

1. Usually, it will be easy to spot when to use the Template Method pattern. Anticipating use for it is a good skill so that your design more easily incorporates changes and variations. Also, in a refactoring project, sometimes we look for places to apply the Template pattern. When two pieces of code are very similar, it is sometimes easy to abstract things out and apply the pattern.
2. *Hook* methods are sometimes used in an algorithm to allow subclasses to provide optional behavior. A *hook* method is a method with a null implementation and can be overridden.

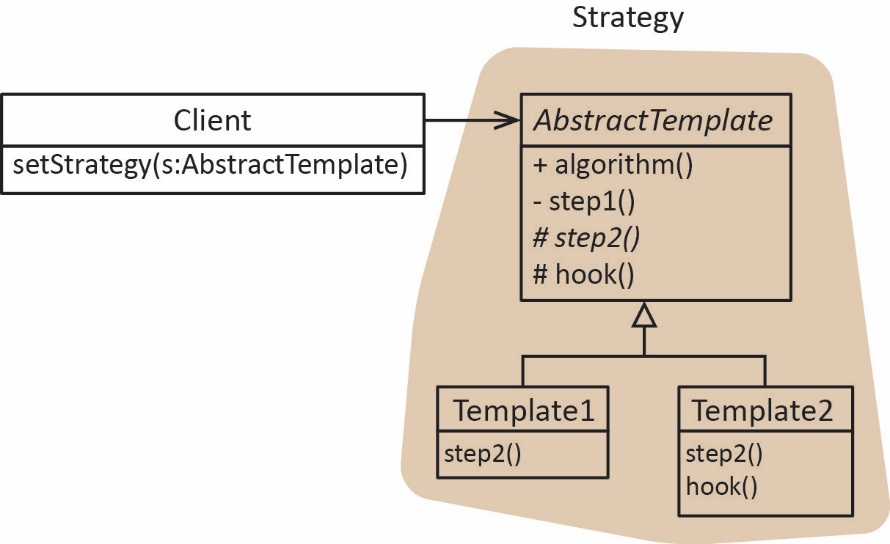
**protected** **void** hook() {}

1. Use the Template Method pattern:

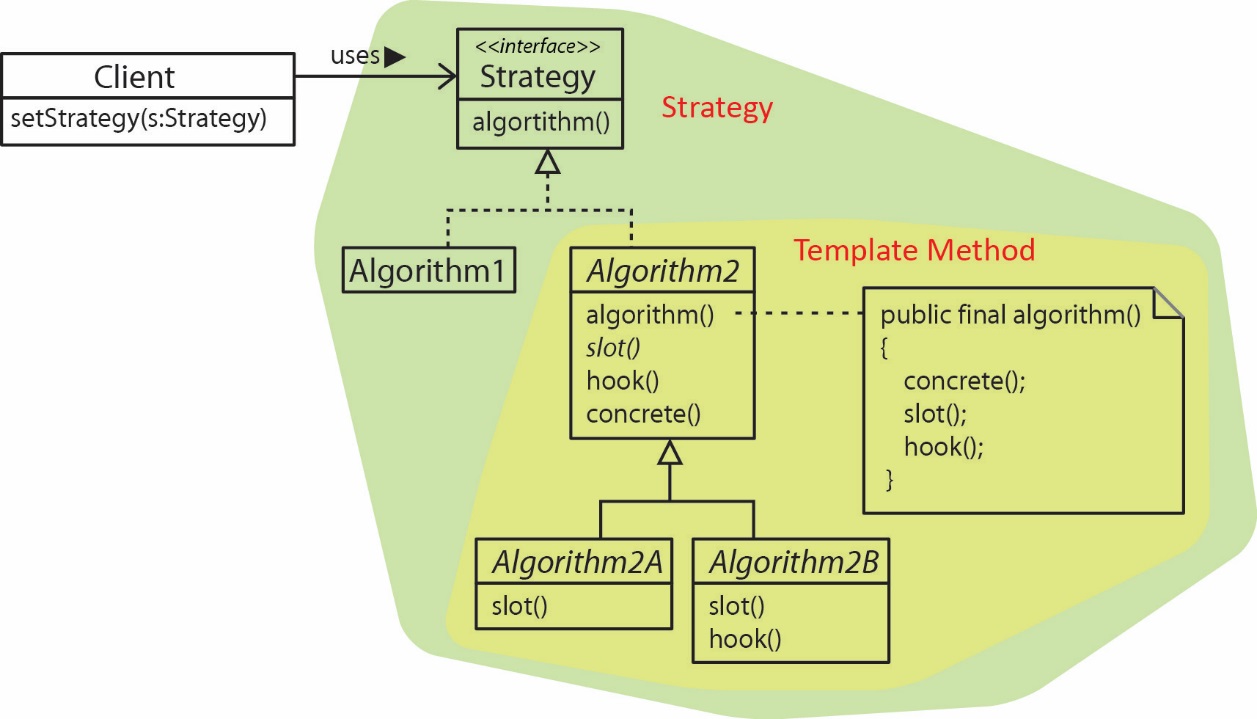
* To define the structure of an algorithm that cannot change, but allow individual steps to vary through their implementation in subclasses.
* To localize common behavior among subclasses and place it in a common class (in this case, a superclass) to avoid code duplication. This is a classic example of “code refactoring.”

**Combining Patterns**

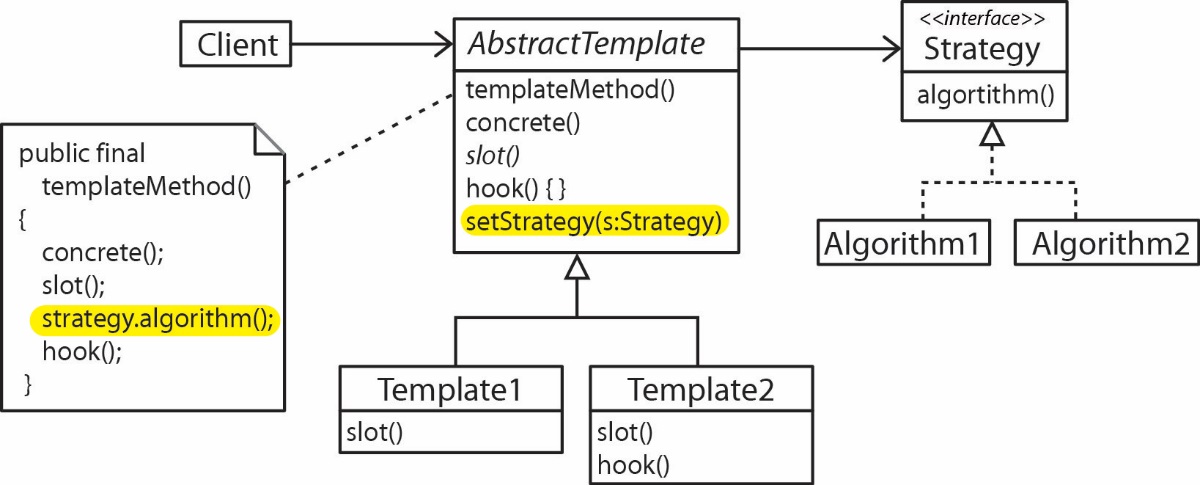
1. A strategy can be implemented as a template method.



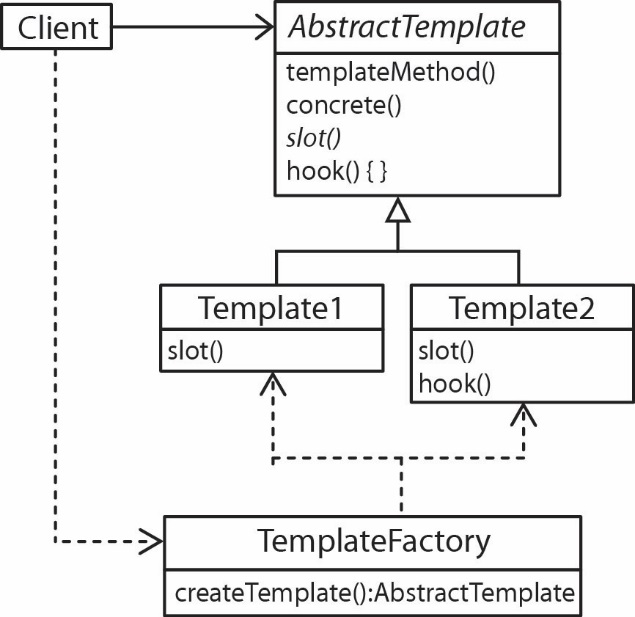
1. An algorithm in the strategy pattern can be a template method.



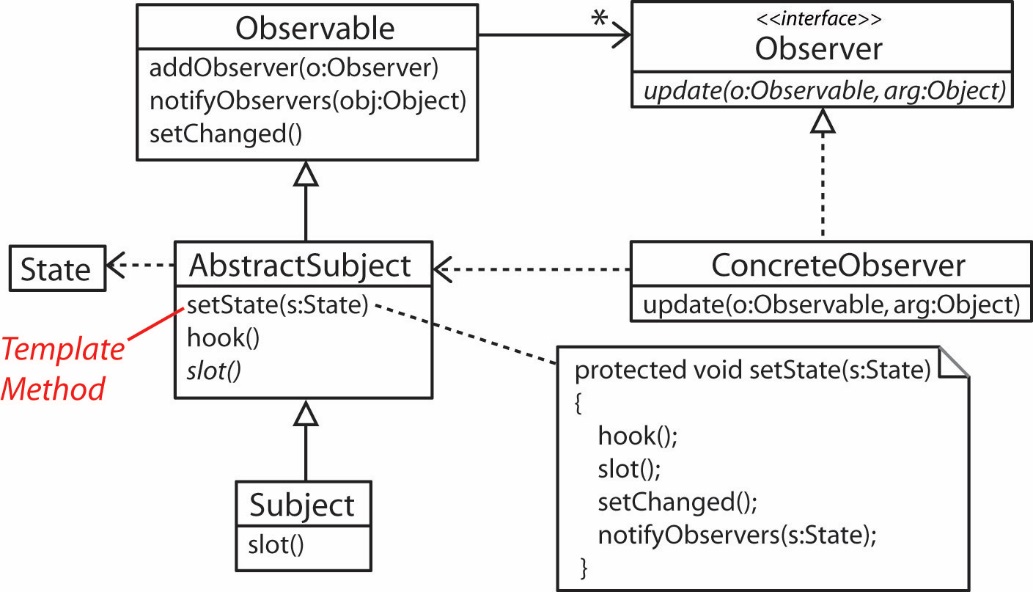
1. A step in a template method can be a strategy.



1. A factory could be used to generate concrete templates.

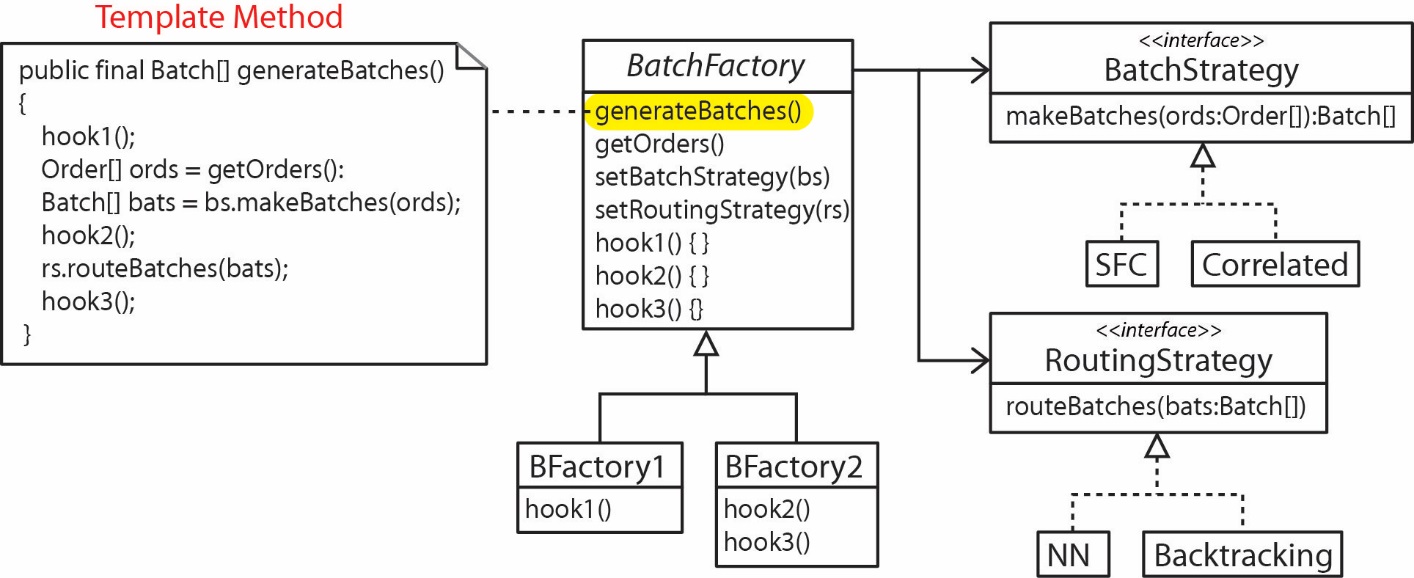


1. Template Method can be used in Observer in a number of ways. Below, we have an *AbstractSubject* (Abstract Template) class which is an *Observable*. Its *setState* method is a template method defining the sequence of actions that take place when the state is set on an observable. A slot method implementation in the subclass determines what happens when the state is set. Notice that the last two lines of the Template Method call setChanged() and notifyObservers() on the Observable.

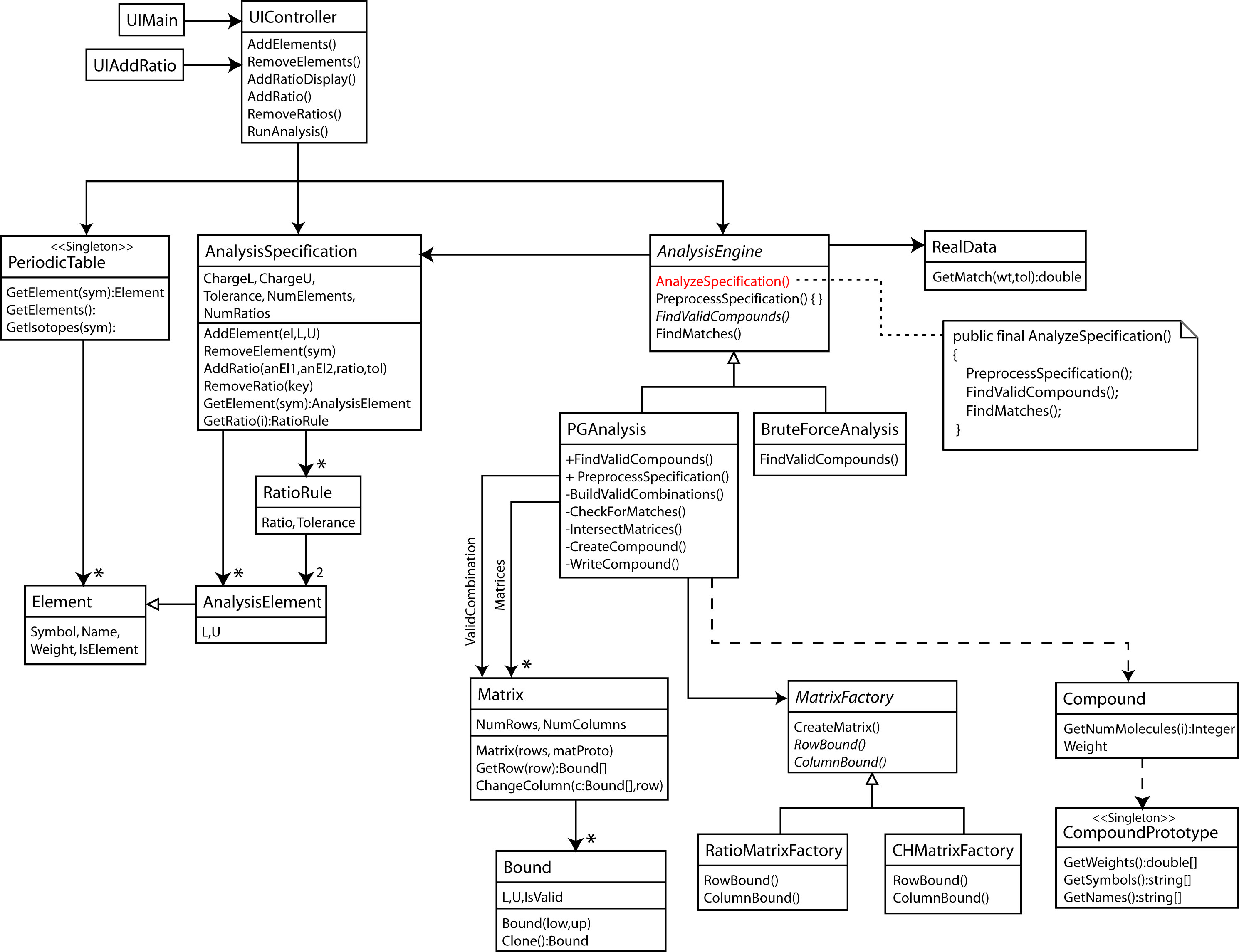
****

**Example**

1. Consider a large online merchant such as LLBean. Orders arrive continuously and each order contains a request for items in the warehouse. Orders are grouped into batches and then an order picker is sent out to retrieve all the items in all the orders in a batch. Order batching can be achieved in a number of different ways. Ideally, we’d like to have batches that contain mostly orders with items that are close to one another so that the order picker doesn’t have to travel too far between picks. Similarly, there are different ways to route an order picker once a batch is determined and different travel rules. For instance, in some warehouses, once you enter an aisle, you must exit from the other end. In other situations, an order picker is allowed to enter and exit an aisle at the same end. A template method for this situation might be:



1. Below is the UML for a system to perform a type of analysis in chemistry. Input data is collected in an AnalysisSpecification. An AnalysisEngine abstract class contains a Template Method, AnalyzeSpecification().

****

1. The *orderPizza* method from the abstract factory chapter is a template method which uses a factory method as one of the steps to create a pizza.

**Homework**

1. A class *Foo* has a method, *blah* that implements an algorithm composed of three steps: *a, b, c* that must be called in that order. Steps *a* and *c* are implemented in *Foo*; however, the implementation of step *b* can vary and there is no default implementation. Further, between steps *a* and *b* an optional behavior *go* needs to be called to allow subclasses the option of doing this additional step. (a) Model this situation with a class diagram. (b) Write all code (in a proof of concept way). (c) Write code to show how to use this.