1 Assignment Overview

For this assignment, you will develop a reusable class library for sorting objects in a type-safe manner. The purpose of this assignment is three-fold. First, you will gain some experience recasting algorithms as reusable components. Second, you will gain some experience working with C++ templates. Finally, you will gain some experience implementing (at least) two of the design patterns discussed in class.

2 System Overview

The design of your sorting library will be based on the Prioritizer interface discussed in class, and will consist of two hierarchies. The first hierarchy will contain templates that encapsulate implementations of various sorting algorithms. The hierarchy will also include a helper template that provides code common across a family of prioritizers that do not amortize the cost of sorting over time. (i.e., All of the sorting work will take place within the body of ChangeToExtractionPhase().)

This template will simplify the task of developing new prioritizer implementations. The second hierarchy will contain comparator templates and classes that implement comparison strategies. These templates and classes will be used to control the order imposed by the prioritizers contained in the first hierarchy.

Like each of the previous two assignments, a key goal of this assignment is for you to gain some design experience. As a result, I won’t provide a complete description of the system design. You have some flexibility in how to complete this assignment. I will, however, describe some of the key requirements that your system design must satisfy. In particular, I will describe some of the key system classes that your design must include, as well as some of the responsibilities that these classes must satisfy.

2.1 The Prioritizer Template

The Prioritizer template will serve as the root of your first hierarchy, and will define the interface of a type-safe prioritizer. The template implementation will be parameterized by the type of object to be sorted. Your template should define an interface only.
2.2 The PrioritizerBase Template

The PrioritizerBase template will derive from Prioritizer, and will implement the code common across a family of prioritizers that do not amortize the cost of sorting over time. This template must satisfy the following responsibilities:

- The PrioritizerBase template will be parameterized by the type of object to be sorted, as well as the maximum number of objects that may be sorted.
- The PrioritizerBase template will not allocate any memory on the heap. Instead, the template will use a statically allocated array, with a length equal to the second parameter value used to instantiate the template.
- The PrioritizerBase template will add and return object copies, not pointers or references.
- The PrioritizerBase template will make use of the Strategy pattern to factor the comparison strategy out of the prioritizer implementation.
- The PrioritizerBase template will make use of the Template Method pattern to defer the sorting task to a derived class. The template should have exactly one pure virtual method.

2.3 The PrioritizerImpl1 and PrioritizerImpl2 Templates

The PrioritizerImpl1 and PrioritizerImpl2 templates will derive from PrioritizerBase, and will provide the virtual method implementation responsible for the actual sorting task. The templates will implement distinct sorting algorithms of your choice. Like PrioritizerBase, each template will be parameterized by the type of object to be sorted, and the maximum number of objects that may be sorted by a prioritizer object.

2.4 The Comparator Template

The Comparator template will serve as the root of your second hierarchy, and will define the interface of a type-safe comparator. Comparators in this hierarchy will serve as strategies to objects instantiated from the Prioritizer hierarchy. The template implementation will be parameterized by the type of object to be compared. Your template should define an interface only.

2.5 The ComparatorImpl1 and ComparatorImpl2 Classes

The ComparatorImpl1 and ComparatorImpl2 classes will derive from Comparator (instantiated appropriately), and will provide the virtual method implementations responsible for comparing objects. You are free to impose any total ordering that you wish. ComparatorImpl1 will compare objects of a different type than ComparatorImpl2.

3 Assignment Grading

Your assignment will be graded based on the quality of your final design and implementation. You should follow the design principles discussed in class, and at a minimum, satisfy all of
the requirements described in the preceding sections. If your design follows all the principles discussed in class, and satisfies all of the requirements discussed above, you will receive a 100%.

4 Hand-In Instructions

As before, you are required to submit an archive containing all of the source code for this assignment, as well as a hard copy of the source. Your assignment is due in class on November 21\textsuperscript{st}.

5 A Warning

Although this lab assignment will not be as rigorous as your previous assignment, it will be time-consuming. Please start early! Late assignments will not be accepted, nor will I grade assignments that do not compile.

As always, you are expected to work on this assignment individually. Please do not consult anyone other than me on any aspect of this assignment.