Our implementation of the Acuity STAR application makes use of two design patterns, Singleton and Prototype, both described below.

Singleton

The intention of using a Singleton pattern is to ensure a class only has one instance and provide a global point of access to it. One way to do this is by creating a global variable to make an object accessible, but this does not prevent one from instantiating multiple objects. A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the motivation behind the Singleton pattern illustrated in Figure 1.

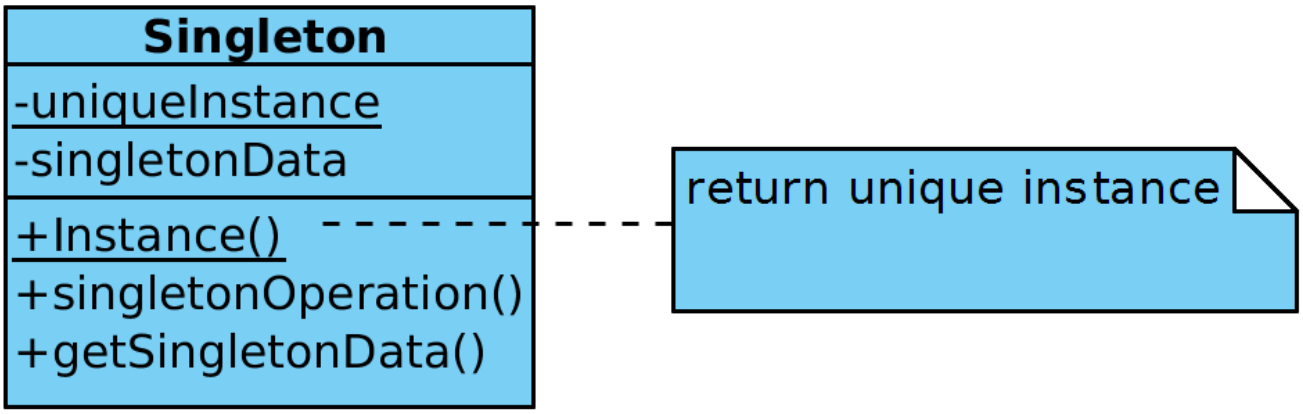

Figure 1: Diagram representation of the Singleton pattern structure. It defines a class operation Instance() that lets clients access its unique instance. Singleton may be responsible for creating its own unique instance. Clients access a Singleton instance solely through Singleton's Instance() operation.

The Singleton pattern has several benefits:

i. *Controlled access to sole instance*: because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
ii. *Reduced name space*: the Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
iii. *Permits refinement of operations and representation*: the Singleton class may be subclassed, and it is easy to configure an application with an instance of this extended class. One can configure the application with an instance of the class you need at run-time.
iv. *Permits a variable number of instances*: this pattern makes it easy to change one's mind and allow more than one instance of the Singleton class. Moreover, one can use the same approach

to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.

v. *More flexible than class operations*: another way to package a Singleton's functionality is to use class operations such as a static member function in C++. However this technique makes it hard to change a design to allow more than one instance of a class. Moreover, static member functions in C++ are never virtual, so subclasses can't override them polymorphically.

Although we do not require a variable number of instances, we acknowledge our flexibility in modifying our instance should the need arise. We use the Singleton design pattern in the implementation of MainWindow, which controls the runtime behaviour of the graphical user interface. This design decision is a result of our belief that there must be exactly one instance of this class and it must be accessible to clients from a well-known access point.

There most important issue with the Singleton pattern we consider when implementing it is ensuring a unique instance. The Singleton pattern makes the sole instance a normal instance of a class, but that class is written so that only one instance can ever be created. A common way to do this is to hide the operation that creates the instance behind a class operation (that is, either a static member function or a class method) that guarantees only one instance is created. This operation has access to the variable that holds the unique instance, and it ensures the variable is initialized with the unique instance before returning its value. This approach ensures that a singleton is created and initialized before its first use. One can define the class operation in C++ with a static member function Instance of the Singleton class. Singleton also defines a static member variable uniqueInstance that contains a pointer to its unique instance.

Prototype

The goal of using a Prototype pattern is to specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. Suppose our system has many objects which, although differ slightly from each other, exhibit almost identical behaviour. We know object composition is a flexible alternative to subclassing. We would like our framework to take advantage of this to parameterize instances based on the type of class it will create. The solution to this dilemna lies in making the framework create a new instance by copying or "cloning" an instance of the desired class. We call this instance a prototype and depict its structure in Figure 2.

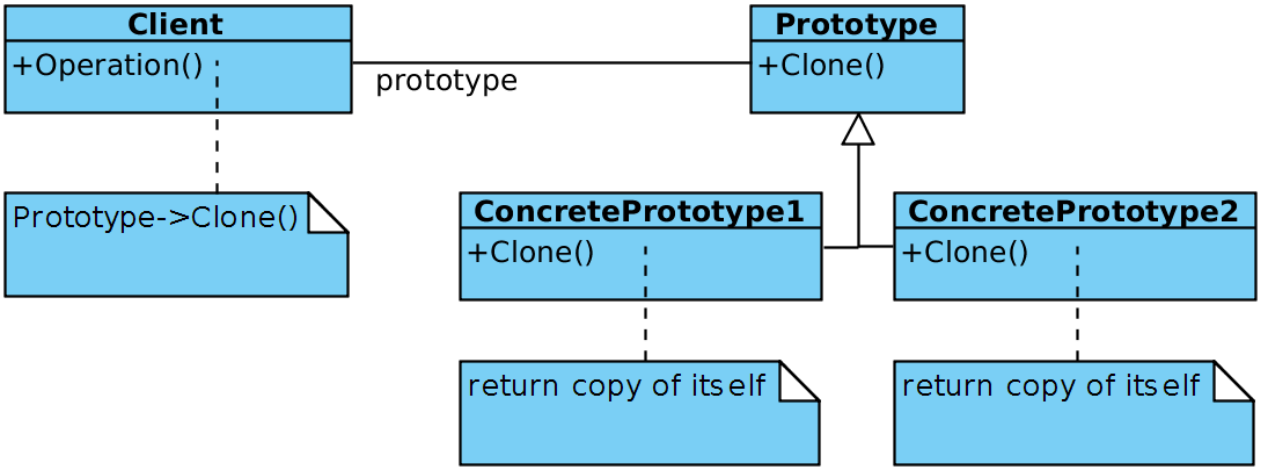

Figure 2: Diagram representation of the Prototype pattern structure. Prototype declares an interface for cloning itself, while ConcretePrototype implements an operation for cloning itself. Client creates a new object by asking a prototype to clone itself.

The Prototype pattern shares many of the same consequences with other creational design patterns: It hides the concrete product classes from the client, thereby reducing the number of names clients know about. Moreover, these patterns let a client work with application-specific classes without modification. However, their are additional benifits unique to Prototype:

i. *Adding and removing products at run-time*: prototypes allow the incorporation of a new concrete product class into a system simply by registering a prototypical instance with the client. This is slightly more flexible than other creational patterns because a client can install and remove prototypes at run-time.

ii. *Specifying new objects by varying values*: highly dynamic systems permit new behavior through object composition—by specifying values for an object's variables, for example—and not by defining new classes. One effectively defines new kinds of objects by instantiating existing classes and registering the instances as prototypes of client objects. A client can exhibit new behavior by delegating responsibility to the prototype. This kind of design lets users define new "classes" without programming. In fact, cloning a prototype is similar to instantiating a class. The Prototype pattern can greatly reduce the number of classes a system needs.

iii. *Specifying new objects by varying structure*: many applications build objects from parts and subparts. Our graphical user interface, for example, is built from widgets, dialog boxes, radio buttons, etc. For convenience, such applications often let one instantiate complex, user-defined structures, say, to use a specific widget again and again. The Prototype pattern supports this as well. We simply add this widget as a prototype to the palette of available graphical user interface elements.

iv. *Reduced subclassing*: other alternatives to the Prototype pattern, such as the Factory Method, often produce a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern allows one to clone a prototype instead of asking a factory method to make a new object. Hence one des not need a Creator class hierarchy at all. This benefit applies primarily to languages like C++ that don't treat classes as first-class objects.

v. *Configuring an application with classes dynamically*: some run-time environments, like that of our application, enables one to load classes into an application dynamically. The Prototype pattern is the key to exploiting such facilities in a language like C++. An application that wants to create instances of a dynamically loaded class will not be able to reference its constructor statically. Instead, the run-time environment creates an instance of each class automatically when it's loaded, and it registers the instance with a prototype manager. Then the application can ask the prototype manager for instances of newly loaded classes, classes that weren't linked with the program originally.

Our decision to use the Prototype pattern follows from the conclusion that our system should be independent of how its products are created, composed, and represented. We consider it the best choice for the TreeModel class, the instantiation of which is specified at run-time by dynamic loading. In general we like to avoid building a class hierarchy of factories that parallels the class hierarchy of products, which is why we do not opt for the factory method. Furthermore, our MainWindow, PieChartWidget, and CustomSort classe instances can have one of only a few different combinations of state. It is thus more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

The main liability of the Prototype pattern is that each subclass of Prototype must implement the Clone operation, which may be difficult. For example, adding Clone is difficult when the classes under consideration already exist. Implementing Clone can be difficult when their internals include objects that don't support copying or have circular references.

In the case of TreeModel, we use what is referred to as a prototype manager. When the number of prototypes in a system isn't fixed (that is, they can be created and destroyed dynamically), we keep a registry of available prototypes. Clients will not manage prototypes themselves but will store and retrieve them from the registry. A client will ask the registry for a prototype before cloning it. Unfortunately our prototype classes do not define operations for (re)setting key pieces of state. If not, then you may have to introduce an initialization operation that takes initialization parameters as arguments and sets the clone's internal state accordingly. An example of this is the setupModel() operation in TreeItem.