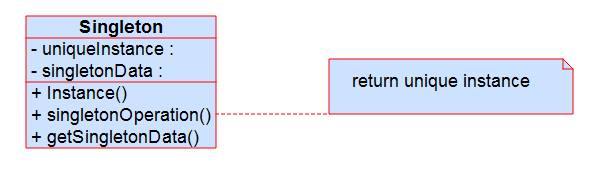
**Design Patterns**

The improved Peachy Galaxy application makes use of two important design patterns. The Singleton pattern for the MainWindow class, and the prototype pattern for the TreeModel class.

**Singleton Pattern**

The goal of the Singleton pattern is to ensure that a class only ever has one instance created and to provide a global point of access to it. While a singleton can be created by using a single global variable, it does not prevent someone from creating several new instances of the class. A better way to implement the singleton pattern is to have the class itself be responsible for keeping track of its single instance. This way, the class itself can be used to ensure that no more than once instance of the class can is ever created and the class itself can act as a way to access the single instance.



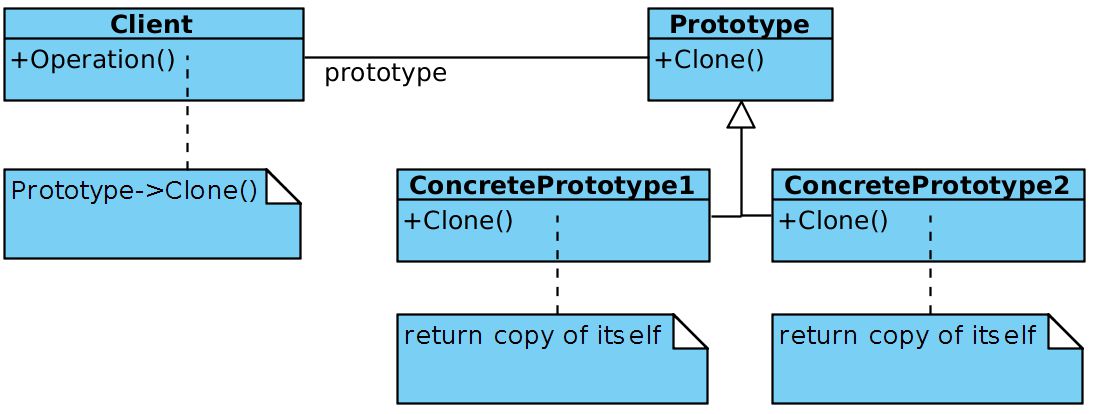
The core benefits of the singleton pattern are that it provides:

* **Controlled access to the single instance**: Since the class itself encapsulates its single instance, the singleton pattern provides strict control over how and when the instance is used.
* **Reduced Name Space:** The singleton pattern avoids polluting the namespace with global variables that’s only purpose is to store a single instance of a class.
* **Permits refinement of operations and representations:** A class using the singleton structure can be extended and subclassed easily. This has the benefit of allowing one to configure an application at runtime by selecting which subclass one needs.
* **Permits a variable number of instances:** If required a singleton implementation can be changed easily to permit the creation of more than once instance of a Singleton class while still maintaining control over the number of instances created.

A Singleton is implemented by enforcing that only once instance can ever be created. The most common way to do this is to hide the constructor behind a class or static method that is responsible for insuring that only once instance is every created. The method has access to a private variable that stores the unique instance and enforces that the private variable is initialized before its first use. This ensures that the singleton has been created and initialized before its first use. From the beginning, our group had the idea that our application would only ever be creating one single instance of the MainWindow class and using that instance to control all behaviour of the applications GUI. Since our application’s MainWindow class only uses a single of instance of itself and must be accessible via a well-known access point we found it appropriate to follow the singleton pattern as we extended the code from the stage 1 system.

**Prototype Pattern**

The objective of the Prototype pattern is to specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. The prototype pattern is extremely useful when creating several new objects which all may differ slightly. The pattern improves application performance by implementing a prototype interface that creates clones of an instance of the desired class instead of creating an entirely new object. This performance increase is especially noticeable in applications that create a sizable number of new objects. This cloned instance is called a prototype and its structure can be depicted with this diagram:



The core benefits of the prototype pattern are that it provides:

* **The ability to add and remove products at run-time:** The prototype pattern allows for the incorporation of a new class into the system by registering a prototype instance with the client. This provides more flexibility when compared to other creational patterns as a client can add or remove prototypes during run-time.
* **Specifying new objects by varying value:** The prototype pattern allows for one to create new behaviour by specifying different values for an objects variable’s instead of creating several different classes and subclasses. Instead the prototype pattern defines new types of objects by registering new instances of objects as prototypes.
* **Specifying new objects by varying structure:** Using the prototype pattern one can build objects out of several parts and subparts. Then using the prototype pattern, you can define new groupings of objects that can be reused by simply adding a group of instances it to the pallet of available prototype objects.
* **Reduced Sub classing:** Many alternatives to the Prototype pattern tends to provide a large hierarchy of Creator classes that parallel the product class hierarchy. The prototype pattern allows one to clone a prototype instead of requiring an entire creator class ultimately reducing the overall number of classes needed for the application.

Our group decided that the TreeModel class would benefit highly from using the Prototype pattern. By using the prototype pattern one can avoid creating a huge class hierarchy of creator classes which will simplify our team’s codebase. Additionally, the performance improvements gained by simply cloning objects instead of initializing new ones is a big plus as the TreeModel class must initialize a huge amount of TreeItem’s. The TreeModel uses a “Prototype Manager” which is when a prototype can be added and remove dynamically during runtime. The class keeps track of a registry of available prototypes and stores / retrieves them when needed.