# Design Problem

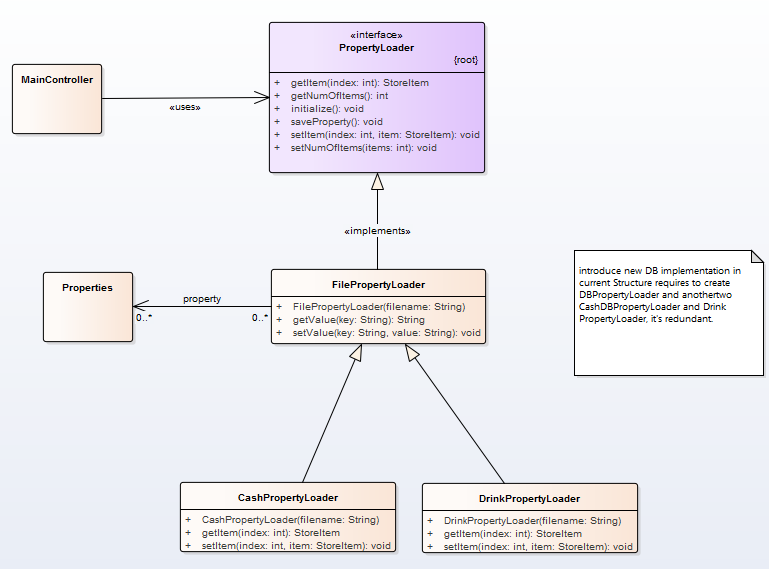
The vending machine currently use File Property Loader to load and initialize Cash and Drink, it does not support other loaders. For example, if the vending machine application is to be reused in another project which store all Cash and Drink information in Database instead of file.

Based on the current design structure, it is not easy to extend to support the additional ways of Property Loader feature.

# Motivation

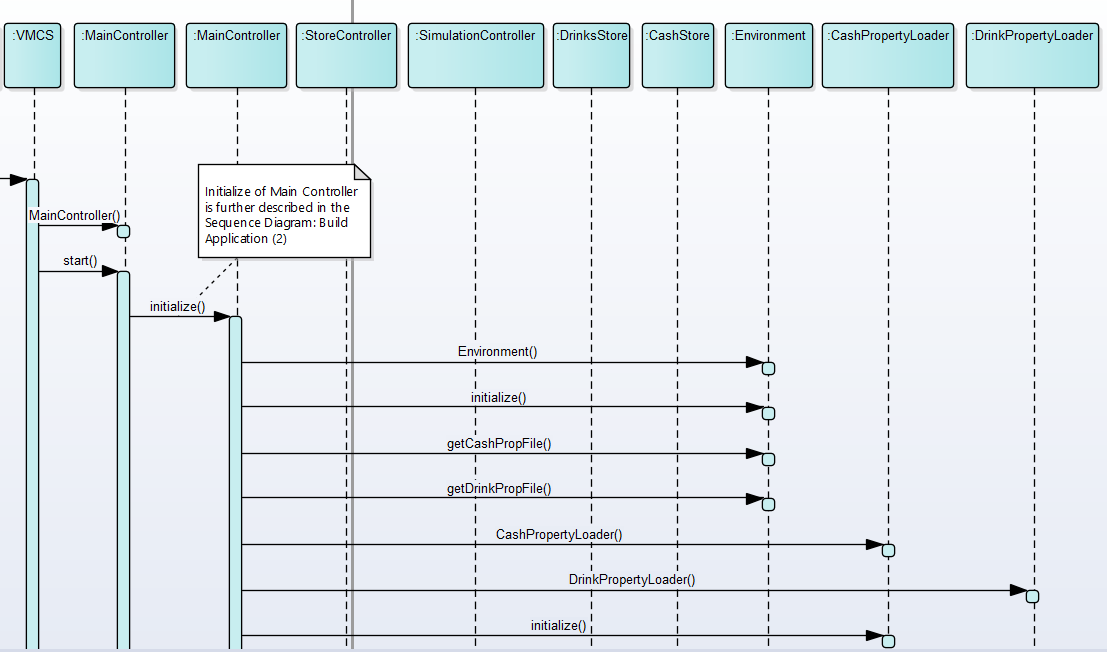
* The ability to reuse the vending machine system to store the Cash and Drink information in various ways including file without affecting existing design.
* The Cash/Drink Property Loader implementation is closely coupled with File Property Loader implementation. Whenever a client creates a Cash/Drink Property Loader, it instantiates a concrete class that has one specific implementation (File Property Loader). This, in turn, makes it harder to port the system using other Loader implementations.

# Current Design (Class Diagram)



Potentially need to support other Data Storing method such as store data in to Data base instead of file.

# Current Design (Sequence Diagram)



Both CashPropertyLoader and DrinkPropertyLoader are tightly coupled with FilePropertyLoader, they are subclass of FilePropertyLoader. It makes difficulty to extend to other Data Storing solution.

# Candidate Design Pattern Considered

The identified design problem is a Structural issue therefore the following Candidate Design Pattern are from Structural:

|  |  |  |
| --- | --- | --- |
|  | **Bridge** | **Strategy** |
| Intent | Decouple an abstraction from its implementation so that the two can vary independently. | Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. |
| Applicability | * You want to avoid a permanent binding between an abstraction and its implementation. * This might be the case, for example, when the implementation must be selected or switched at run-time. * Both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently. * Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled. * You want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client. | * Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors. * You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms. * An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures. * A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class. |
| Aspect | Implementation of an object | an algorithm |
| Decision | After study the candidate patterns, **the decision is to use Bridge pattern**. The reasons for choosing Bridge pattern over other patterns is because:   * It can avoid a permanent binding between an abstraction and its implementation. Property Loader should not permanent bind with File Loader Implementation. * It can hide the implementation of an abstraction completely from clients. Cash Property Loader must not know what Loader implementation is used to load Cash in system. * Changes in the implementation of an abstraction should have no impact on clients. That means adding a new Loader Implementation should don’t require any code change on Cash Property Loader. | |
|  |  | |

# Participants

The classes and objects participating in this pattern are:

* Client (MainController)
* It use the Abstractions
* Abstraction (PropertyLoader)
* Defines the abstraction’s interface
* Maintains a reference to an object of type Implementor
* Refined Abstraction (CashPropertyLoader, DrinkPropertyLoader)
* Extends the interface defined by PropertyLoader
* Implementor (PropertyLoaderImpl)
* Defines the interface for implementation classes. This interface doesn’t have to correspond exactly to Abstraction’s interface; in fact the two interfaces can be quite different. Typically the PropertyLoaderImpl interface provides only primitive operations (such as getValue or setValue), and PropertyLoader defines higher-level operations based on these primitives.
* ConcreteImplementor (FilePropertyLoaderImpl, DBPropertyLoaderImpl)
* Implements the PropertyLoaderImpl interface and defines its concrete implementation.

# Revised Design (Class Diagram)

Please note that the Concrete Implementor for DB is based on future enhancement and **are only added in for illustration purpose,** it will not store or load any data from/to data base.

# 

Concrete Implementor

Concrete Implementor

Refined Abstraction

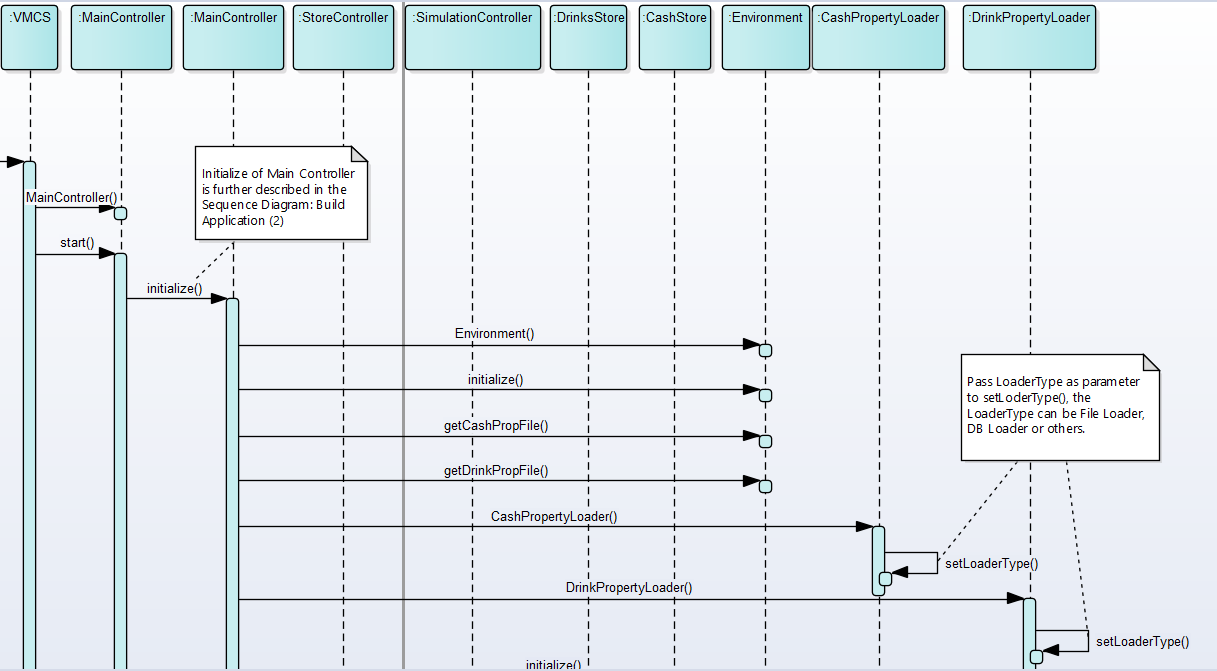
Refined Abstraction

Abstraction

Implementor

Client

# Revised Design (Sequence Diagram)



# Consequences

Using Bridge pattern has the following consequences (positive and negative).

|  |  |
| --- | --- |
| **Consequence** | **Rationale** |
| Decoupling interface and implementation. | Decoupling abstraction and Implementor also eliminates compile-time dependencies on the implementation. Changing an implementation class doesn’t require recompiling the Abstraction class and its clients. This property is essential when you must ensure binary compatibility between different versions of a class library. |
| Improved extensibility | Can extend the Abstraction and Implementor hierarchies independently. |
| Hiding implementation details from clients. | Can shield clients from implementation details, like Cash Property Loader do not need to know where the data is stored. |

# Implementation

There are several issues that were considered when applying the Bridge pattern.

|  |  |
| --- | --- |
| **Issue** | **Rationale** |
| Implementor switching at runtime cause loss of changed data | At the same running time, vending machine system is storing data in to either DB, File or other storage. Switching the Implementor at runtime will lose the changed data in prior implementor. |
| Creating the right Implementor object | How, when and where do you decide which Implementor class to instantiate when there’s more than one?  Although various implementors are provided, each vending machine system only need to instantiate one specific Implementor according to the chosen data storing method. |