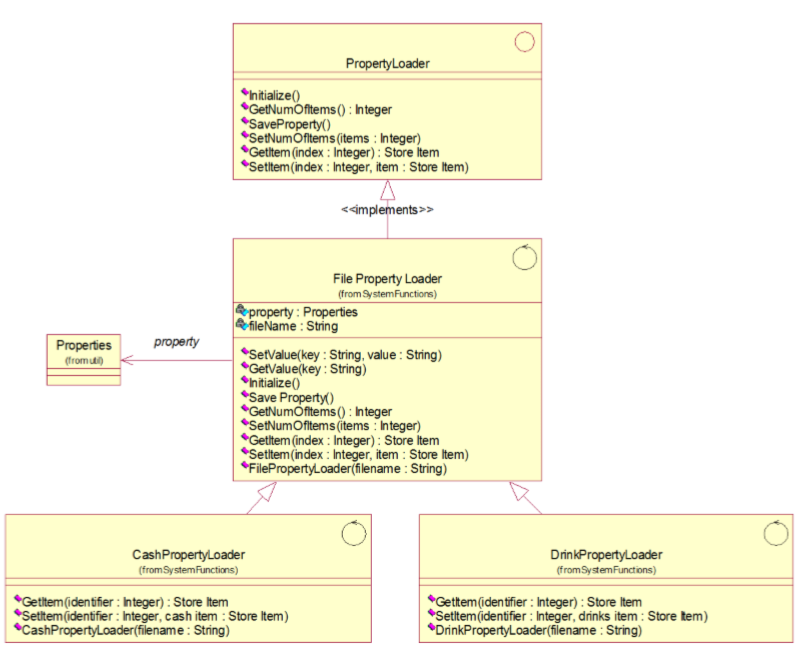
**SITHIVINAYAGAM KOKULAKUMARAN (A0120490U)**

**Bridge Design Pattern**

1. **Design Problem**

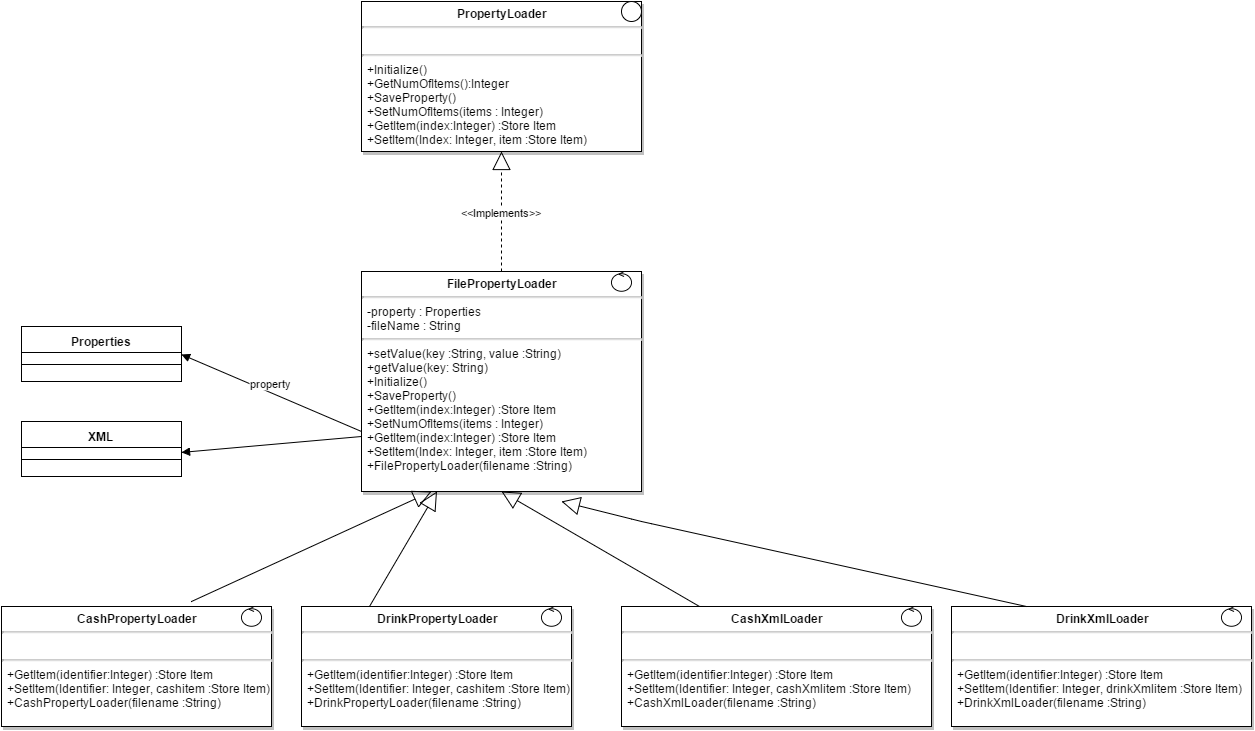
It is difficult to add or change the persistence mechanism in the existing system of VMCS. Currently it is done only using properties file. There is no possibility given to user to change the storing mechanism. If we need to add XML based data store mechanism, we need to add 3 classes. Hard to extend Business Objects are having persistence logic.

The following is the class diagram of existing design of the persistence mechanism.



Current design doesn’t provide the expandability, easy maintainability which should have been expected in the design stage. Though this design is well satisfied the current requirement, there is no room to extend, if requirement for changes comes in future to collaborate different kind of storing mechanism.

The following is the brute forced class diagram of implementing XML persistence mechanism to current design. Each additional mechanism inclusion will add more classes and code repetitions.



1. **Candidate Patterns**

The following design patterns have been analyzed to solve this design problem.

1. Adapter Design Pattern

Adapter pattern converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

1. Bridge Design Pattern

Bridge patterns decouples an abstraction from its implementation so that the two can vary independently

1. Abstract Factory

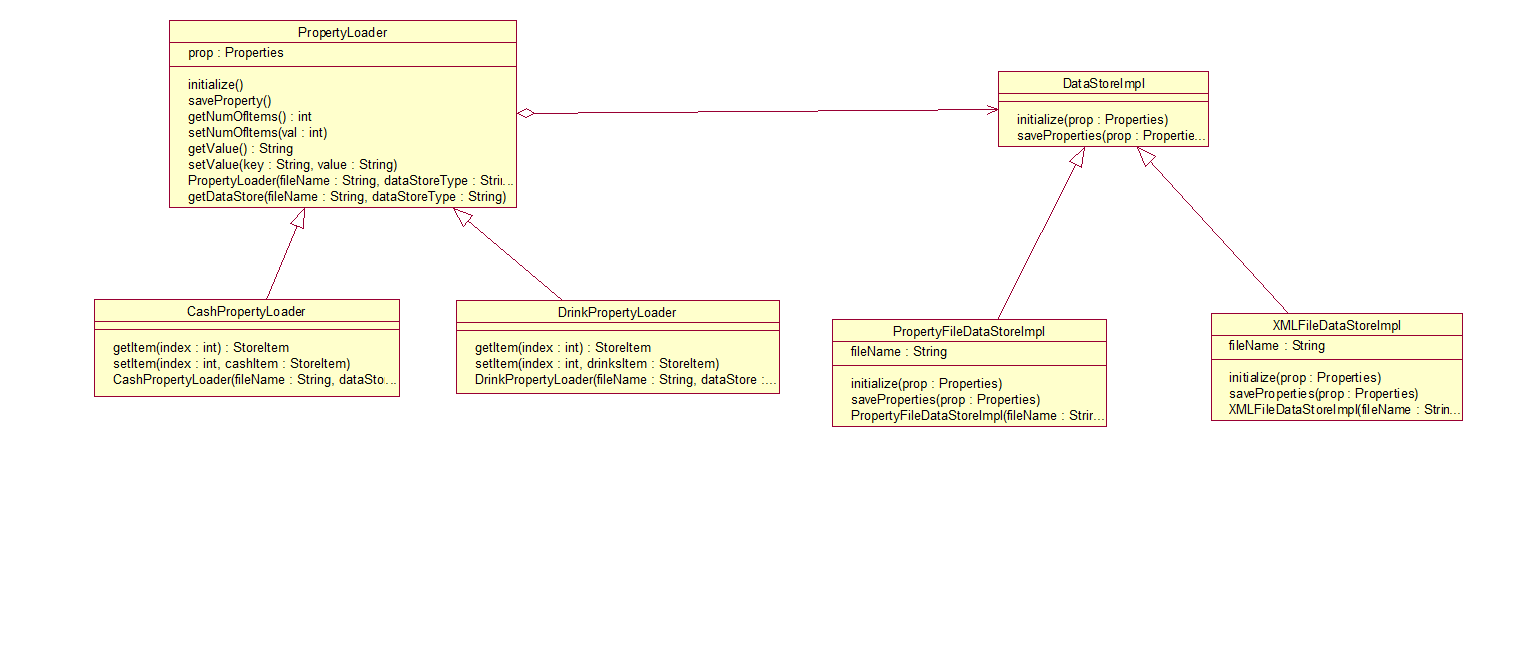
Abstract Factory pattern provides an interface for creating families of related objects without specifying their concrete class. For current design issue, will need only one storage mechanism at a time.

1. **Motivation to select bridge pattern**

Bridge pattern has been selected to solve this design issue out of three candidate design. According to current design, the implementation related functionalities has to be removed from the CashPropertyLoader and DrinkPropertyLoader classes. So that, they don’t need to stick with other abstract functionalities. So implementation detail can be decided in run time. Further it is not required to expose the implementation to clients. And only one class is needed to be added when adding different storage mechanism.

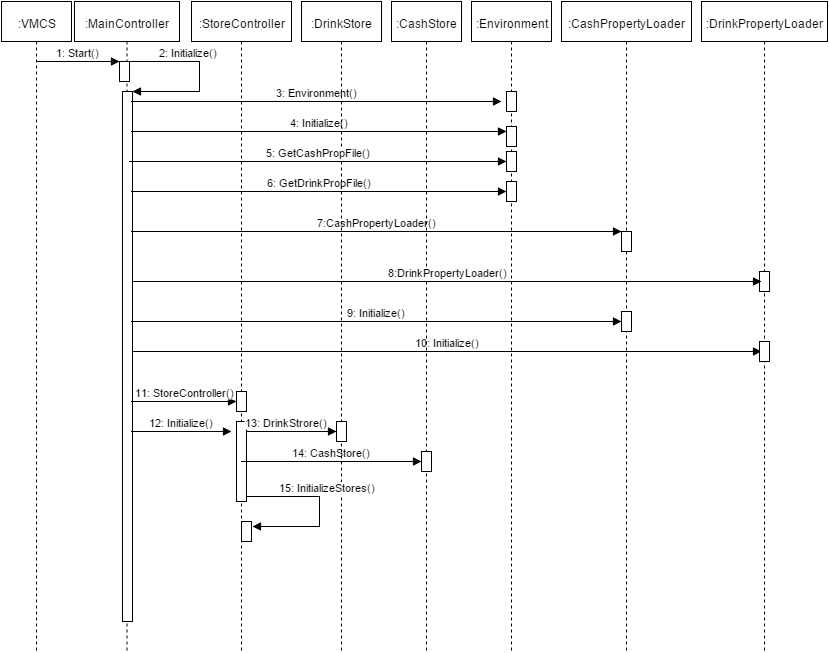
1. **Class Diagram after applying the design pattern**

The following is the class diagram after applying the bridge design pattern with two persistence mechanisms such as properties file and XML file.

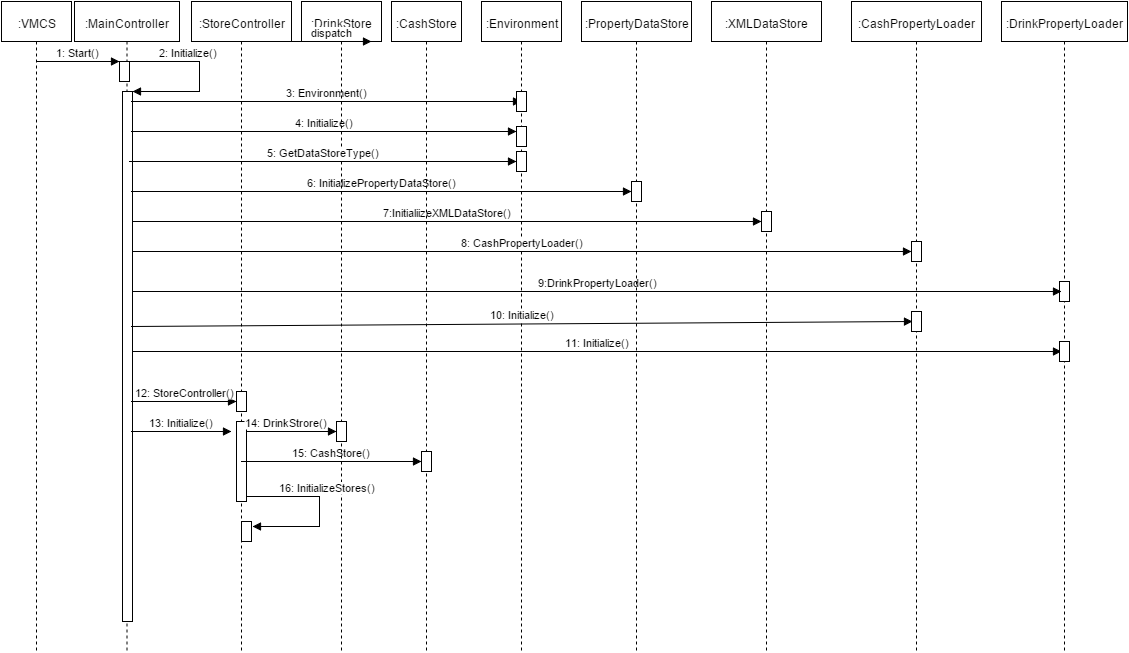
****

1. Sequence Diagrams

Following sequence diagram illustrates system start up before applying the bridge design pattern



Following sequence diagram illustrates system start up after applying the bridge design pattern



1. **Implementation Decisions**
2. Only one Implementer

In some scenarios, only one Implementer class should be needed, for those cases no need to define abstract implementer, concrete implementer is enough. But in this case, currently we decided to have two concrete implementers and this might get increased based on the future needs.

1. Creating the implementer object

Though abstraction knows the concrete implementer classes, to enable the desired restrictions (not to use some implementer in concrete abstraction) within concrete abstraction classes, we decided to put the concrete implementer creation in the subclasses of Loader those are CashLoader, DrinkLoader. Factory pattern can be used for externalization of implementation class creation. DataStore type will be retrieved from vcms properties file

1. Sharing implementers

Implementers are not shared in this context.

1. Using multiple Inheritance

Multiple inheritance are not supported directly in Java. So it is not relevant in this context.