1. **Design Patterns Overview**

Design Patterns are blueprints or commonly used reusable solutions for reoccurring problems that occur when designing software systems. Design patterns are typically not drop in code but more a generally accepted best practice for implementing a solution for a given problem. As a generally accepted best practice, design patterns serve as an informal vocabulary or shorthand between software engineering practitioners allowing a high-level exchange of information about a design by stating the particular patterns being used without having to delve into the low level details/code to get the point across. In addition to expressing information about a design more concisely, the use of design patterns help enhance the maintainability of a system by allowing for more loosely coupled designs through the use of OOP techniques and in particular the use of interfaces between the various objects involved in the particular pattern.

1. **Project Overview**

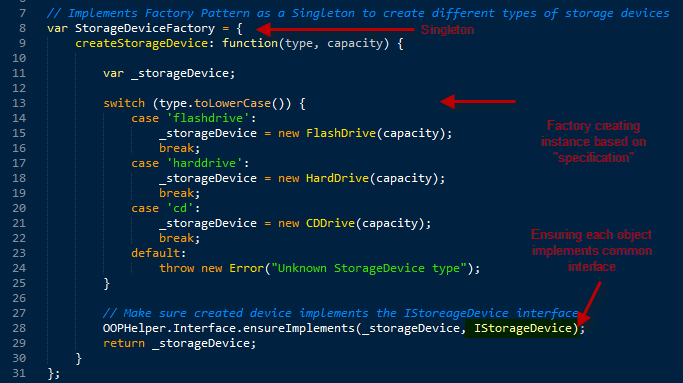
For this project I chose to implement a system for mimicking different types of storage devices (Hard Drives, CD, Flash etc.) including a basic file system. Some noted features of the design to help support the use of patterns include:

1. Interfaces: To help enforce the design and keep in-spirit with the “interface”-based practice embedded in design patterns, in particularly the Factory and Composition patterns, I utilize mechanism for defining “pseudo” interfaces and testing that objects implement expected interfaces.
2. Data Encapsulation / Hiding  
   I take advantage of Javascript Closure and Lexical Scoping features to enforce private vs public members and object vs static functions. All private member attributes are accessed using getters and setters with setters being occasionally private and called via a constructor.

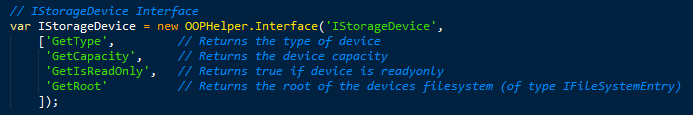
For the design I chose the following design patterns:

**Creational: Factory Pattern w/Singleton Pattern**The factory pattern mimics the notion of a real-life factory where we instruct the factory to create a particular type of class instance without actually being exposed to the creation logic and then access that object using an “interface” (IStorageDevice) common to all of the different entities the factory is capable of creating.   
  
Because there is typically not a need to have multiple factories, my StorageDeviceFactory is implemented as a Singleton. The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance.

The system supports creating three different types of storage devices, “hard drive”, “flash drive” and “cd” which are created through the use of a StorageDeviceFactory object which itself is implemented as a Singleton (Javascript object literal).



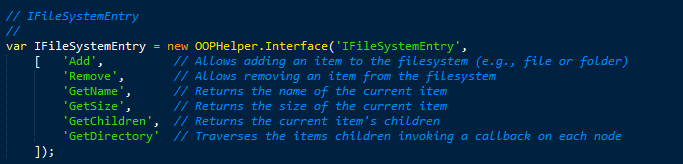
Each of the support concrete StorageDevice classes inherits from a StorageDeviceAbstract class which implements an IStorageDevice interface:

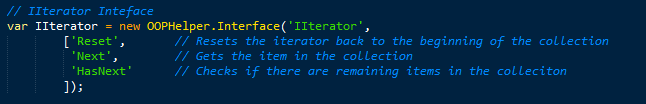


**Structural: Composite Pattern**

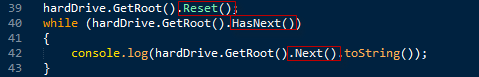
Each of the storage devices implements / exposes a filesystem represented as a tree of folders (Branches) and /or files (Leafs). A folder object usually contains one or more file or folder objects and thus is a complex object where a file is a simple object. Additionally files and folders typically have many operations and attributes in common, such as moving and copying a file or a folder, listing file or folder attributes such as file name and size, making it easier and more convenient to treat both file and folder objects uniformly by defining an IFileSystemEntry Interface.

The intent of this composite pattern is to compose objects into tree structures to represent part-whole hierarchies while letting clients treat individual objects (files) and components of objects (folders) uniformly.



**Behavioral: Iterator Pattern**Since the tree-like structure of the filesystem can be unwieldy to navigate, the FolderEntry class implements the IIterator pattern to provide a very simple mechanism for traversing the entire filesystem tree sequentially without exposing the implementation or internal structure of the tree.  


The iterator object maintains the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated allowing the client it use a simple while-loop which is the perfect complement to the Composition pattern.



1. **Project Output**

The following is a screenshot of the application running under node.js on Invengion.org. In the output, you see that the system creates to storage devices (1GB Hard Drive and 1MB Flash Drive) and then adds a collection of folders/files to each devices filesystem and then iterates over the filesystem of each device.

The source code can be found in /home/student/msk7/design-pattern-hw on invengion.org. The code can be executed using the command:



Additionally the source can be found on github at : [git@github.com:msk7/DesignPatterns.git](mailto:git@github.com:msk7/DesignPatterns.git) (<https://github.com/msk7/DesignPatterns.git>)

