**Creational Pattern:**

**Singleton:**

**Features:**

* Only One instance created
* By default, lazy loaded
* Do not overuse singletons. They are difficult to unit test and are not thread safe. An argument should not be passed to singleton.

**Examples:**

* Logger
* Spring Beans

**Design:**

* A singleton creates itself and manages its lifecycle
* It is static in nature but not necessarily a static class, the reason being static classes are not thread safe in nature.
* There is a private instance and private constructor because single itself should only be able to call the constructor.

Table

Description automatically generated with medium confidence

**Runtime:**

Allows application to interact with Environment.



**Eagerly Loaded Singleton Class:**



**Lazily Loaded Singleton Class:**



**Thread Safe Lazily Loaded Singleton Class:**

1. Make private static attribute volatile
2. Insert code in constructor to avoid reflection calls
3. In getInstance() implement doublecheck or synchronized check.



**Builder:**

This is used to construct objects with many parameters in constructor and then make it immutable. E.g., StringBuilder, DocumentBuilder and Locale.Builder.

**Design:** Often in OOP, we define multiple constructors with different sets of parameters (telescoping constructors). The builder pattern overcomes this by handling the construction with an object rather than using parameters. It is written with a static inner class which returns instance of object it is building.

**StringBuilder: More performant than StringBuffer**

****

**Builder Pattern Example:**

The below example has a lot of getters and setters (not immutable). It does not define a proper contract like what is a valid lunch order.

****

**Telescopic Constructors:** It creates immutable objects but does not allow different configs like only wheat and ham.



**Builder Demo:** We have a static class inside. The created objects are immutable. It is complex.



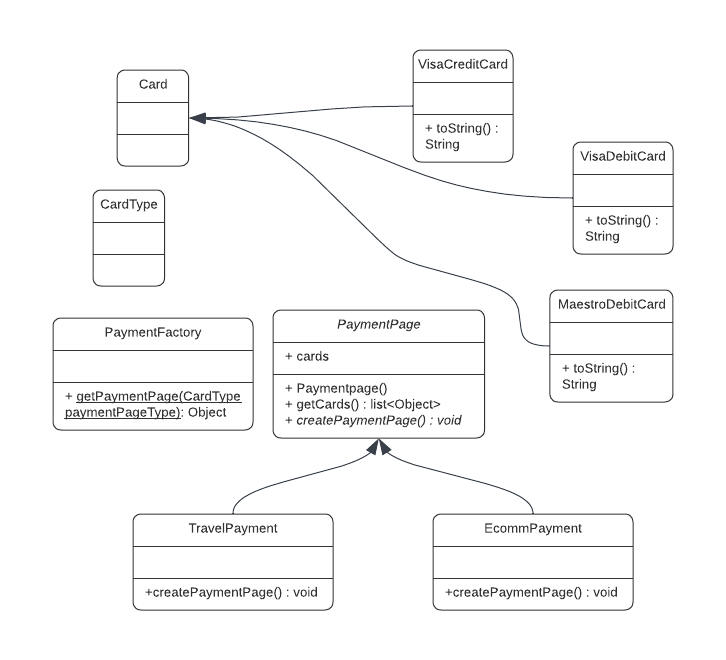
**Factory:**

|  |  |
| --- | --- |
| * Does not expose instantiation logic. * The client does not know anything about the type of object being created. * It defers creation to subclass. * The client only knows about the common interface that the factory has exposed. * Examples – Calendar, ResourceBundle and NumberFormat |  |
| **Design:**   * Factory is responsible for creating instance and managing lifecycle. * Objects created are referenced through common interface. * The method to request an object is parameterized where the parameter is the concrete type. |

**Calendar Demo:** Calendar is not singleton, it is factory.

****

**Factory Implementation:**

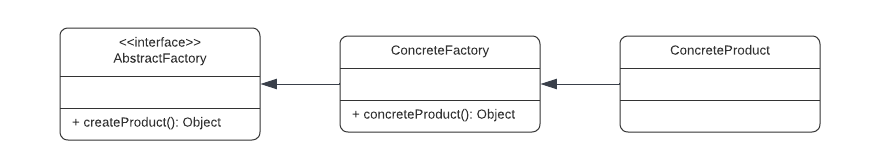




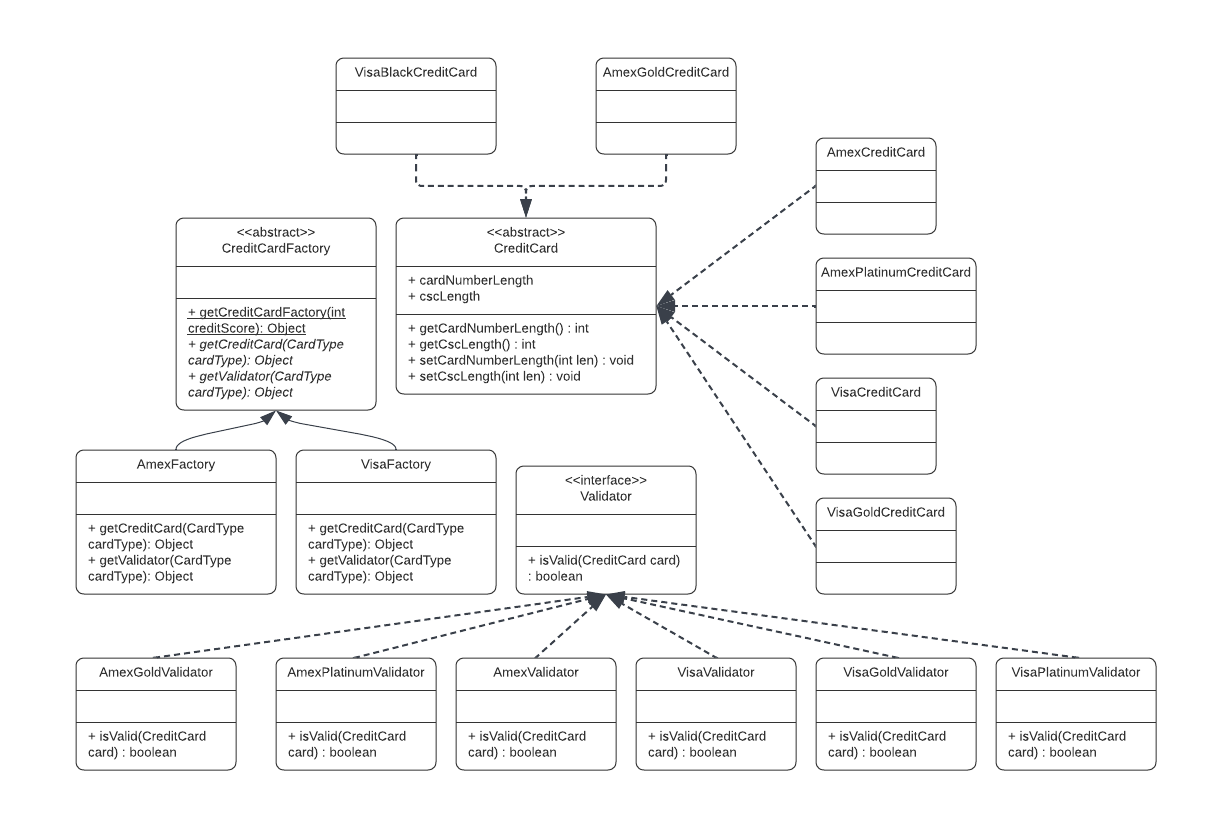
**Abstract Factory:**

* Factory of Factories or related objects
* Examples : DocumentBuilder
* Group Factories together

Design:



Implementation:









**SOLID**

Let’s say, we have a system comprising of many modules.

* If we make changes to one module and other modules break, then the code is fragile.
* If code changes to one module warrant code changes to other modules, then code is rigid.

The system has a lot of technical debt.

Technical Debt = overlooking code quality to achieve fast delivery.

Code quality = more time + complex code + maintainable and modular code.

If cost of change in system and response time to meet customer needs grow over time and become unmanageable, then system has high technical debt.

The best approach is to write code and then take some time to refactor and pay technical debt applying design patterns and solid and more unit tests.

SOLID

* **S**INGLE RESPONSIBILITY PRINCIPLE
* **O**PEN CLOSED PRINCIPLE
* **L**ISKOV SUBSTITUTION PRINCIPLE
* **I**NTERFACE SEGREGATION PRINCIPLE
* **D**EPENDANCY INVERSION PRINCIPLE

**SINGLE RESPONSIBILITY PRINCIPLE**

Every function, class or module should have only one reason to change or one responsibility. Some examples of responsibilities are Business Logic, Persistence, Orchestration, Logging and User Interfaces. Some benefits are as follows.

* Easily maintainable code
* Loosely coupled code more resilient to changes
* More testable design.

1. Avoid Monster Methods and God Classes.
2. In case of if else and case statements, if the different branches have different logic, it is better to extract the logic in a method.
3. Move helper methods like saveToDB, Serialize etc to helper methods.
4. Actor SRP – ReportGenerationForHR and ReportGenerationForEngineer should not be facilitated by a single generateReport() method.

**Dangers:**

* Difficult to read code
* Decreased quality due to difficulty in testing
* Side Effects
* Tight Coupling, specifically with concrete classes, use abstraction for the concrete class implementation.

**OPEN CLOSED PRINCIPLE**

Classes, Functions and Modules should be closed for modification but open for extension.

This means existing source code should not be modified while adding new features because other components may depend on it. Instead, new features must be implemented in isolation in new classes and have their own unit tests. This will reduce regression bugs.

There are 2 approaches:

1. Inheritance – Extend the existing class, MoneyTransfer to InternationalMoneyTransfer and override the method transferMoney() but it will result in coupling
2. Strategy Pattern – Create an interface e.g. MoneyTransferProcessor with the method transferMoney() and then have each class, MoneyTransfer and InternationalMoneyTransfer implement this interface. None of the classes have any relationship with each other. Then we create a factory MoneyTransferProcessorFactory and based on attributes return appropriate MoneyTransfer classes and invoke the method.







**LISKOV SUBSTITUTION PRINCIPLE**

If S is a subtype of T, then objects of type T in a program may be substituted with Objects of type S without modifying the functionality of program.

Keep base classes small and interfaces lean and focussed.

**INTERFACE SEGREGATION PRINCIPLE**

Clients using interfaces should not be forced to depend on methods they do not use. The interface can be an interface or an abstract class.

Lean interfaces reduce code coupling and makes code more focussed.

Things to lookout for:

* + Interfaces with many methods
  + Interfaces with low cohesion (all methods must be related to a function e.g., payment and stock related methods should not be in same interface or abstract class)
  + Empty implementation or throwing exception in methods instead of implementing the method
  + Forcing implementation

Corrective measures:

* Breakup interfaces
* For code we cannot control, use adapter pattern

**UML**



**DEPENDENCY INVERSION PRINCIPLE**

1. High level modules should not depend on low level modules; both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

High level modules (payment, user mgmt. etc.) map to business domain and are abstract in nature while low level modules (logging, data access, io, network communication, security like oauth, email and notification) are concerned with implementation details and are concrete in nature.

|  |  |
| --- | --- |
|  | Component A will not directly talk to Component B. Instead, it will speak to an abstraction of Component B. Component B Abstraction is an interface or an abstract class which Component B implements or extends. A component factory will return an instance of the concrete component based on conditions. This component factory will be used by Component A to create an instance of Component B and use its methods. |

****

**DEPENDANCY INJECTION**

A technique that allows creation of dependant objects outside of a class and provides those objects to a class.

Declare dependencies in Constructor. But it is to be noted that when there are many objects and dependencies, then it becomes harder to manage. For that we have IOC.



**INVERSION OF CONTROL**

It is a design principle where the control of object creation, configuration and lifecycle is passed to a container or framework.

The IOC container creates objects instead of the programmer. E.g., Spring Framework.

* XML Configuration
* Annotation

