What, Why, SOLID

Tuesday, October 30, 2018 5:21 PM

Design patterns are basically best practices in software engineering using **object-oriented design**(**OOD**) that describe general reusable solutions to common problems in software design within a given context. These are not the finished or final designs readily transformed into live software product but a description or a template of how to solve a given problem that can be used in many different situations. Design patterns are formalized best practices that can be used to solve common problems when designing a solution.

Design patterns provide a way to solve problems related to software development using a proven solution. They not only make communication between designers more efficient, but they also make the code more tangible, clear, efficient, and reusable.
GoF design patterns are generally considered the foundation for all other patterns. Software professionals can immediately picture high-level design when they refer to the name of the pattern used to solve a particular problem.

SOLID design principles are a set of basic OOD design principles. These principles were first published by Robert C. Martin

S.O.L.I.D stands for five primary class design principles:

- **Single Responsibility Principle** (**SRP**): A class should have only one reason to change, which means that a class should have only one primary job to do; an extension can be added, for example, inheritance. Implemented by the decorator pattern.

- **Open Closed Principle** (**OCP**): A class should be designed in such a way that it is open for the extension of its behavior but is closed for modification in itself.

- **Liskov Substitution Principle** (**LSP**): All of the derived classes should be substitutable (replaceable) with their parent class.

- **Interface Segregation Principle** (**ISP**): Interfaces should be fine-grained and client-specific. Let's say that a client should never be forced to implement an interface that they don't need or use.

- **Dependency Inversion Principle** (**DIP**): Depend only on abstractions rather than on concretions. Abstractions should not depend on details (or implementations); instead, details should depend on the abstractions.

GoF design patterns

Thursday, November 01, 2018 6:54 PM

Design patterns were formally introduced in the book *Design Patterns: Elements of Reusable Object-Oriented Software*, first published in 1994 by the four authors Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, with a foreword by Grady Booch. These authors are usually referred to as the *Gang of Four*. The book contains the most popular 23 design patterns; hence, the 23 patterns are known as *GoF* design patterns. These design patterns are fundamentally the crux of **object-oriented analysis and design** (**OOAD**).

23 *GoF* design patterns are divided into three main categories.

Creational patterns		
These design patterns deal with object creational mechanisms in a suitable manner under a given situation		
1	Abstract factory	Provides an interface that can create families of related classes
2	Builder	Separates complex object construction from its representation
3	Factory method	An interface used to create an object without specifying the exact class to be created
4	Prototype	Creates fully initialized objects by copying or cloning an existing object
5	Singleton	A class whose only a single instance can exist

Structural Patterns		
These design patterns are the best practice used to identify a simple way to realize relationships between entitles in a given situation		
6	Adapter	Converts the interface of a class into another interface client except by wrapping it's own interface around that of an already existing class
7	Bridge	Separates an object's interface from its implementation so that the two can vary independently
8	Composite	Composes zero or more similar objects in a tree structure so that they can be manipulated as one object to represent part or whole hierarchies
9	Decorator	Dynamically adds/overrides behavior in an existing object
10	Facade	Provides a unified interface to a set of interfaces in a subsystem
11	Flyweight	Efficient sharing of a large number of similar objects
12	Proxy	An object acting as a placeholder for another object to control access to it and reduce complexity

Behavioral patterns		
They identify common communication patterns between objects and increase the flexibility in carrying out the communication between them		
13	Chain of responsibility	Decouples the request sender from the request receiver by chaining more than one receiver object in a way that if one receiver object does not handle the request, it passes on to the next until the request has been handled
14	Command	Encapsulates a request/action/event along with parameter as an object
15	Interpreter	Specifies a way to interpret/evaluate sentences in a given language
16	Iterator	Sequentially accesses the elements present in an aggregate object without exposing its underlying representation/Structure
17	Mediator	Promotes loose coupling by encapsulating (and often centralizing) the communication/interaction between various objects
18	Memento	Provides the capability to snapshot the object's internal state in order to restore to this state later
19	Observer	An observable object sends events to many observing objects; sort of defines a one-to-many dependency between objects
20	State	Allows the changes in the object's behavior based on the change in its state

Abstract Factory Pattern

Thursday, November 01, 2018 8:29 PM

Abstract Factory Pattern - Problems

- App needs to support multiple database types
  - SQL Server, Oracle, MySQL
  - Isolate database type from application
- Measurement data comes from multiple sources
  - Serial port, Ethernet, device driver
- Need to create different report types
  - PDF, Word doc, etc.

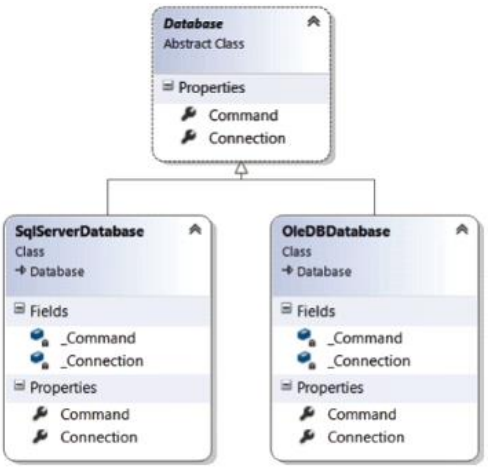
Abstract Factory Pattern

- Provides an abstract class
  - Generalized interface
  - Hides details from rest of application
  - Factory class creates an instance of a class that inherits or implements the abstract class
  - In conjunction with Factory Method pattern

Abstract Factory Pattern Example

- Database abstract class
  - Connection property (System.Data.Common.DBConnection)
  - Command property (System.Data.Common.DBCommand)
- Abstract class cannot be instantiated
  - Instance classes must inherit from it

DATABASE ABSTRACT FACTORY



Builder Pattern

Friday, November 02, 2018 1:18 PM

BUILDER PATTERN - PROBLEMS

- App needs to construct multiple type classes
- These classes have complicated construction methods
- Database
  - Connection string, Command type, timeout
- Measurement data sources
  - Serial port: port number, baud rate, parity, etc.
  - Ethernet: IP address, port, timeout, etc.
- Isolate construction process from application

BUILDER PATTERN

- Builder Interface or abstract class
  - Generalized object construction interface
  - Provides methods for each object construction step
- Create Builder classes that implement builder interface for each object type

BUILDER PATTERN

- Director class
  - Uses Builder class to create object
  - Hides creation details from rest of application

Builder Pattern Example

- Construct Database class
  - SQL Server and OleDB
  - Create Connection, Command objects
  - Set Command parameters

Builder Pattern Example Classes

- Database abstract factory classes
- Abstract Database class
- SqlServerDatabase class
  - Inherits from Database class
- OleDbDatabase class
  - Inherits from Database class

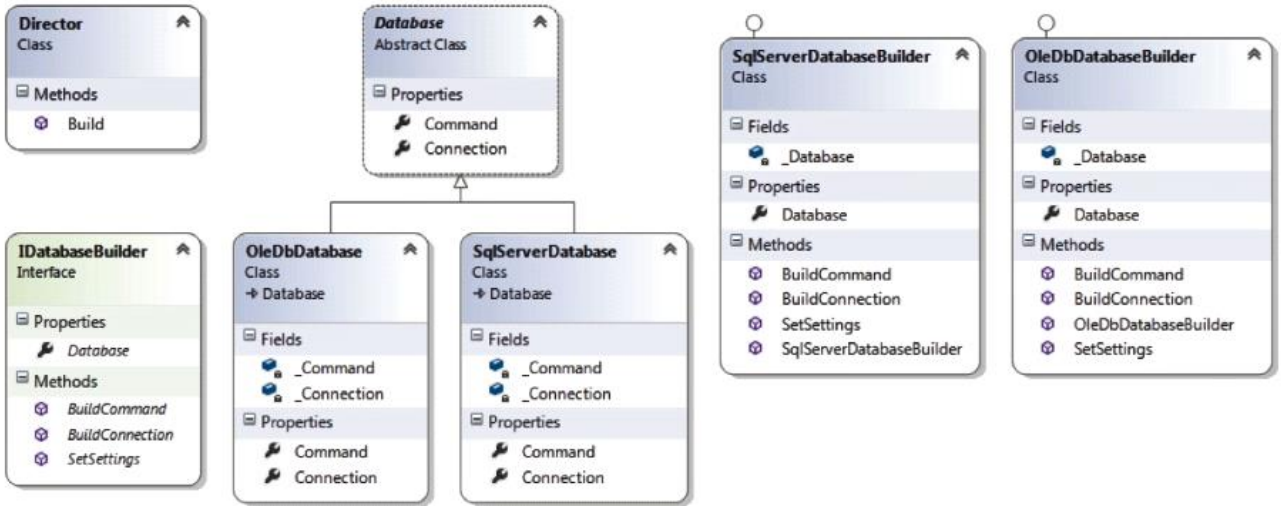
Builder Pattern Example Classes

- IDatabaseBuilder interface
  - BuildConnection method
  - BuildCommand method
  - SetSettings method
  - Database property
- SqlServerDatabaseBuilder and OleDbDatabaseBuilder classes
  - Implement IDatabaseBuilder

Builder Pattern Example Classes

- Director class
  - Build method
  - Takes IDatabaseBuilder parameter
  - Calls IDatabaseBuilder construction methods in sequence
- After Director.Build:
  - Builder.Database returns fully constructed Database instance

DATABASE BUILDER



Factory Method Pattern

Friday, November 02, 2018 6:14 PM

Factory Method Pattern - Problems

- App needs to support multiple database types
  - SQL Server, Oracle, MySQL
  - Isolate database type from application
- Measurement data comes from multiple sources
  - Serial port, Ethernet, device driver
- Need to create different report types
  - PDF, Word doc, etc.

Factory Method Pattern

- Provides a method to create class instances
- Often used in conjunction with Abstract Factory
  - Provides a single method for creating objects
  - Hides details of creating objects from rest of application
  - Factory class creates an instance of a class that inherits or implements the abstract class

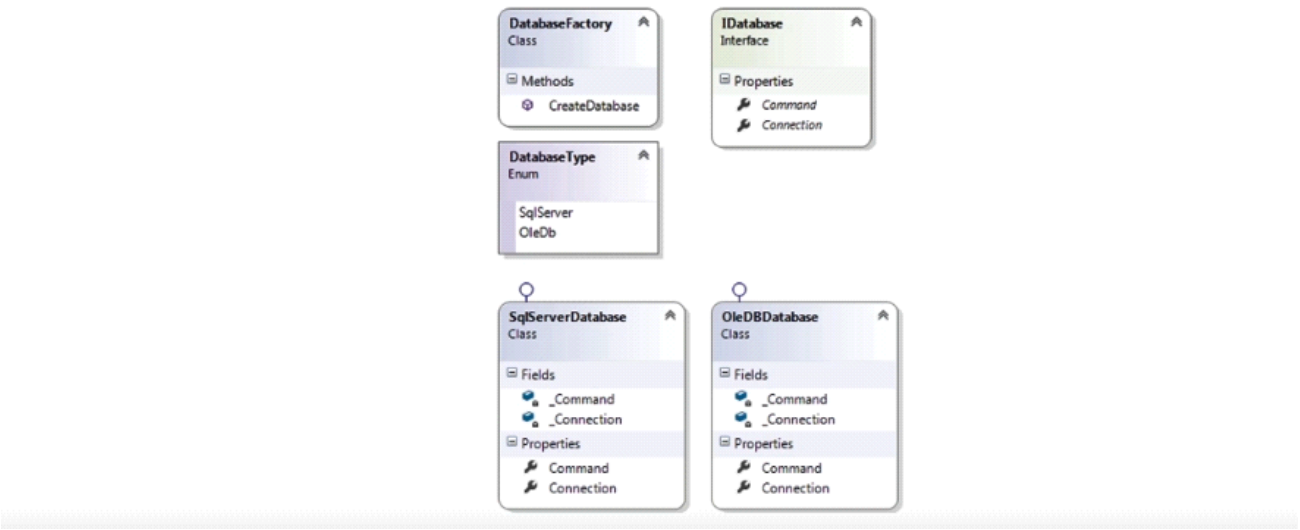
Factory Method Pattern Example

- Database abstract class
  - Connection property
  - Command property
- SQL Server and Ole DB derived classes
- Enum for database type
- Database Factory class for database class creation
  - Single static method

Factory Method Pattern Example

- Implementation Choices
  - Use of interface rather than abstract class
  - Interfaces for properties
- Interfaces require implementation in concrete classes
- No implementations in "base" interface
- Factory returns an interface rather than a class
- Use of abstract class may be appropriate as well

Database Factory Method Pattern



Prototype Pattern

Saturday, November 03, 2018 5:34 PM

PROTOTYPE PATTERN - PROBLEMS

- App needs copies of objects
  - Objects are not simple value types
  - May have other objects as properties
- User Object
  - Name, Username, Password properties
  - Address object property
- Isolate copy code from application

PROTOTYPE PATTERN

- "Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype" - GoF
- Provides a method to copy an object
- .Net iCloneable interface
- Deep and Shallow copies

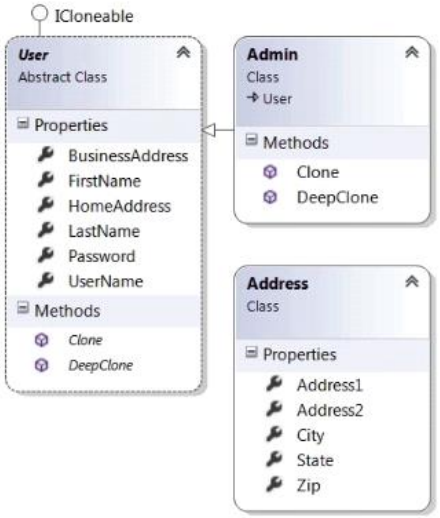
PROTOTYPE PATTERN EXAMPLE

- User abstract class
  - Name, Username, Password properties
  - Address object property
  - Clone methods
  - ICloneable interface
- Admin derived class
  - Implements ICloneable and Clone methods

PROTOTYPE PATTERN EXAMPLE

- Shallow Copy
  - Use MemberwiseClone method
  - References to object properties are copied
  - Changes to clone make changes to original!
- Deep Copy
  - Copies values of all properties, including properties of sub-objects

PROTOTYPE PATTERN



Singleton Pattern

Saturday, November 03, 2018 11:51 PM

SINGLETON PATTERN - PROBLEMS

- App needs a single instance of an object throughout
  - Exception logging
  - Database or business layer managers
  - Hardware access – socket, serial port

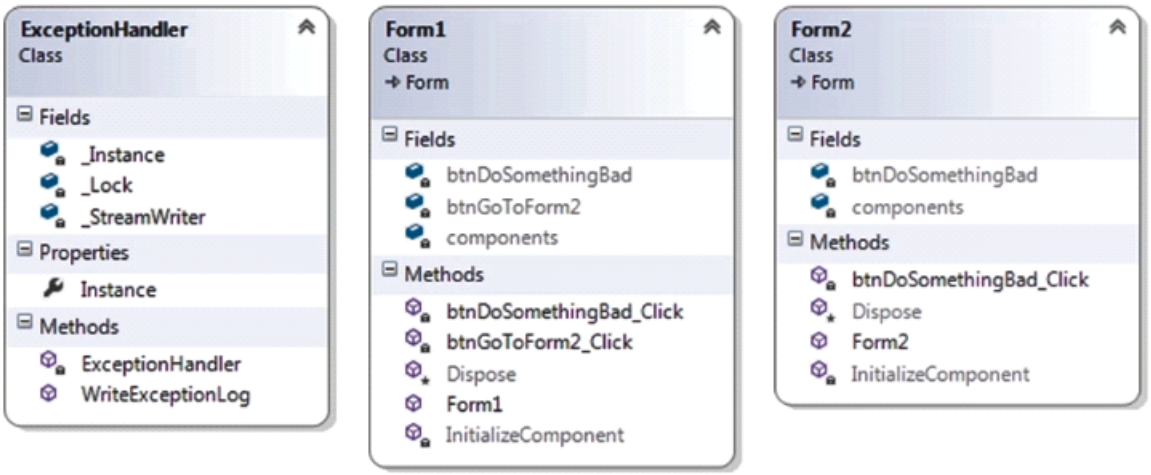
SINGLETON PATTERN

- Simplest, most common pattern
- Creates a single instance of an object
- Static private instance
- Private constructor
- Public Instance property
  - Could be GetInstance method

SINGLETON PATTERN EXAMPLE

- ExceptionHandler class
  - Manages writing to a single log file for all of the application
  - Private static Instance member
  - Private constructor
  - Public Instance property – type ExceptionHandler
  - WriteException public method

SINGLETON PATTERN



http://www.crazyforcode.com/cant-static-class-singleton/

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members

Adapter Pattern

Sunday, November 04, 2018 3:57 AM

ADAPTER PATTERN - PROBLEMS

- You have an object that needs converting to a different type
  - Types are likely similar
- Data transfer object (DTO) to business object
  - Data comes from web service, needs conversion
- Third party objects
  - Need to convert to application types

ADAPTER PATTERN

- Provides an interface for the application to use
  - Interface has methods adapter will implement
- Adapter class does conversion
  - Implements interface methods
  - Converts types, or implements methods

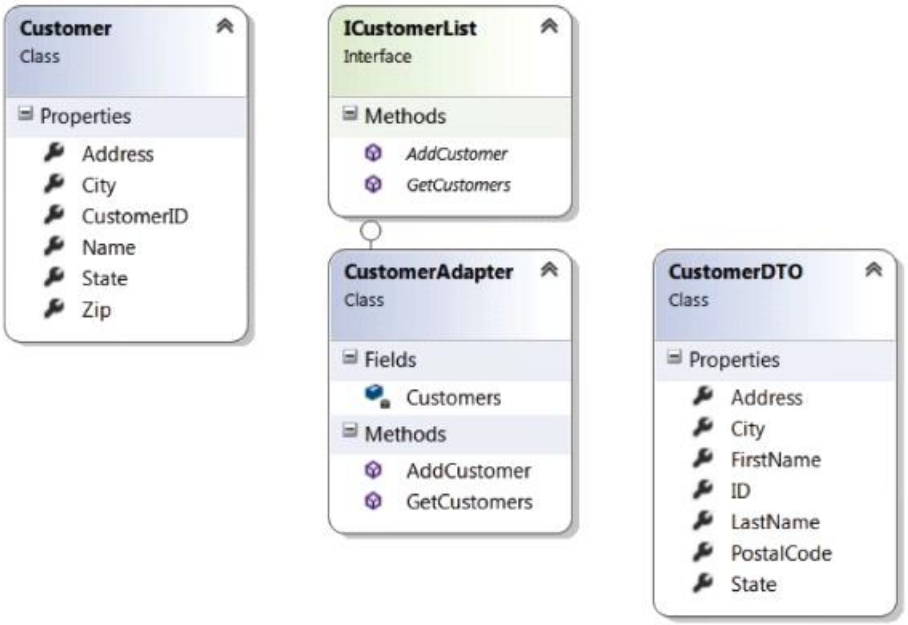
ADAPTER PATTERN EXAMPLE

- Customer Business Object
  - Used throughout application
- Customer DTO
  - Comes from web service
  - Different properties than Customer

ADAPTER PATTERN EXAMPLE

- ICustomerList
  - GetCustomers return list of business object customers
  - AddCustomer takes Customer DTO, converts to Customer, adds to list
- CustomerAdapter
  - Implements GetCustomers and AddCustomer

ADAPTER


Customer
Class
Properties
Address
City
CustomerID
Name
State
Zip
ICustomerList
Interface
Methods
AddCustomer
GetCustomers
CustomerAdapter
Class
Fields
Customers
Methods
AddCustomer
GetCustomers
CustomerDTO
Class
Properties
Address
City
FirstName
ID
LastName
PostalCode
State

Bridge Pattern

Sunday, November 04, 2018 3:57 AM

Bridge Pattern - Problems

- You want to decouple the application from a feature implementation
  - If feature changes, application won't
- Messaging
- Social Media Posting
- Logging
- Data Sources

Bridge Pattern

- Decouple an abstraction from its implementation so that the two can vary independently
- Provides an abstract class or interface
- Feature classes will be concrete implementations of abstract Bridge class
- Abstraction class used by application
  - Note, abstraction != abstract class

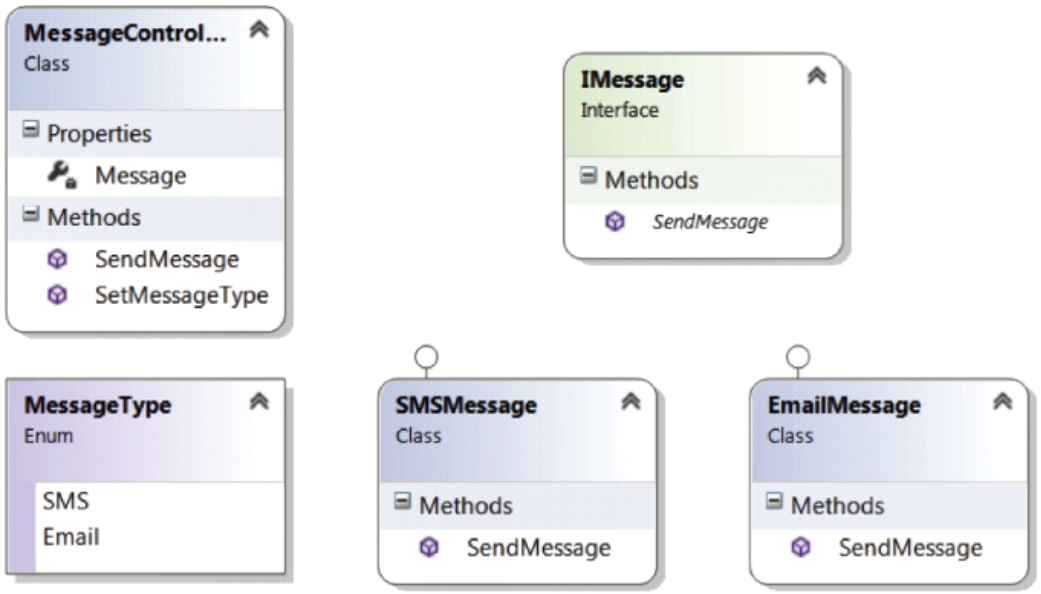
Bridge Pattern Example

- Messaging Feature
  - Currently, SMS or Email
  - May want other types in future
- IMessage interface abstraction
  - SendMessage method
- Concrete SMSMessage and EmailMessage classes

Bridge Pattern Example

- MessageController
  - Abstraction class
  - SendMessage method
  - SetMessageType method
  - Decouples IMessage and concrete implementations

Bridge



https://softwareengineering.stackexchange.com/questions/336951/what-is-the-difference-between-bridge-and-factory-pattern-in-c

Composite Pattern

Sunday, November 04, 2018 3:46 PM

COMPOSITE PATTERN - PROBLEMS

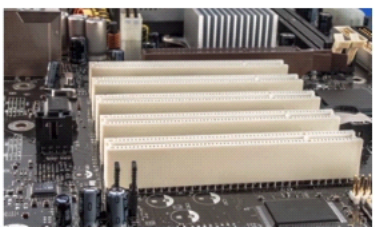
- You have an object that is composed of similar child objects
  - Each child may or may not have children
  - Tree View
- File System Folders
- Supervisors and Employees
- Users and Followers
- Manufacturing Assemblies

COMPOSITE PATTERN

- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Provides an abstract class or interface for each object
- Parent objects will have a collection of children

COMPOSITE PATTERN EXAMPLE

- Manufacturing Parts List
  - End product contains assemblies and individual parts
  - Each assembly contains parts
  - Each assembly may contain other assemblies



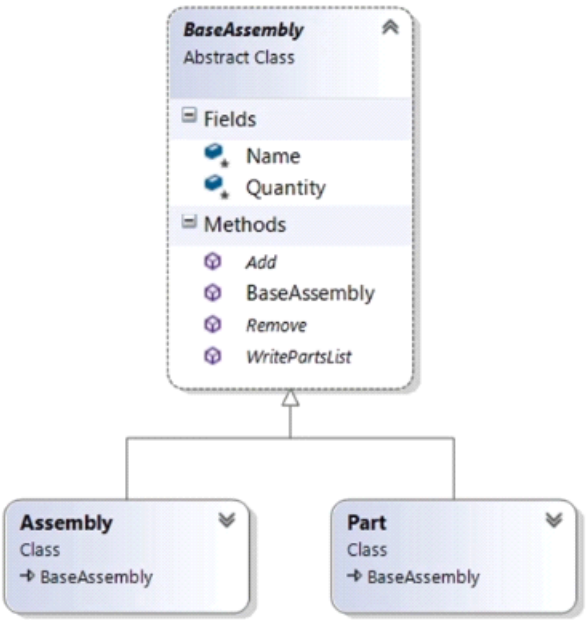
COMPOSITE PATTERN EXAMPLE

- BaseAssembly abstract class
  - Could be an interface
  - Quantity and Name members
  - WritePartsList method
  - Add and Remove methods in parent assemblies

COMPOSITE PATTERN EXAMPLE

- Assembly implementation class
  - Internal collection of child assemblies
  - Children are BaseAssembly type
  - WritePartsList calls same method on all children
- Part implementation class
  - No children
  - Add and Remove or not implemented – throw exception

COMPOSITE



Decorator Pattern

Sunday, November 04, 2018 5:09 PM

DECORATOR PATTERN - PROBLEMS

- You have a base object that may have a lot of options
  - You want to be able to add options dynamically
  - Not a lot of subclasses
- Options on product object
- Add class capabilities
  - Encrypt a stream
  - Add media to a social post

DECORATOR PATTERN

- Allows behavior to be added to an individual object, either statically or dynamically
  - Does not affect other objects from the same class
- Subclass the original Component class into a Decorator class
- In Decorator class, add a Component member
- Pass Component to Decorator constructor
- In Decorator, override any method(s) that need to be modified

Decorator Pattern Example

- Computer Ordering – Build your own
  - RAM Options
  - Storage Options
  - Discount Options

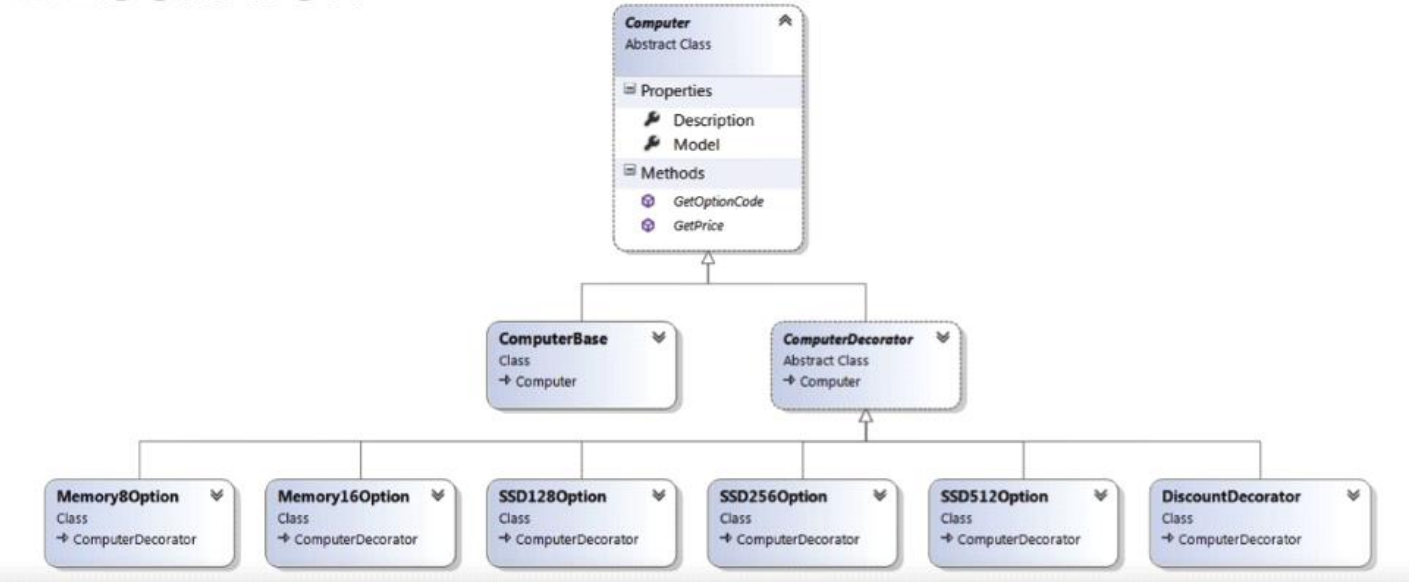
Decorator Pattern Example

- ComputerBase class
  - Class to be "decorated"
  - GetPrice and GetOptionCode methods
- ComputerDecorator abstract class
  - ComputerBase member set in constructor

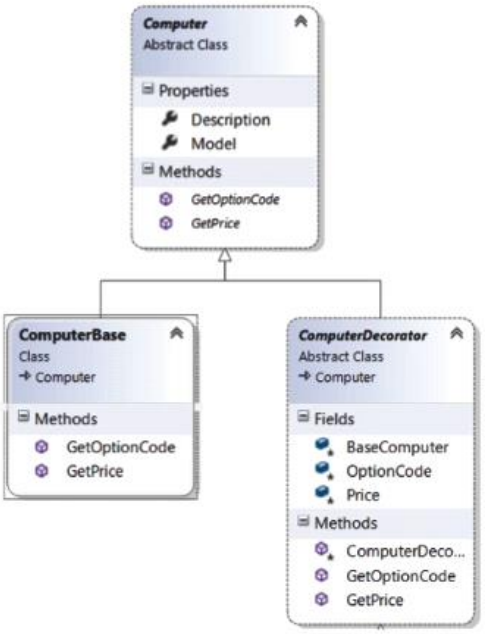
Decorator Pattern Example

- Decorators
  - RAM - 8, 16 GByte
  - Storage – 128, 256, 512 GByte
  - Discount
- Overrides
  - Override GetOptionCode by appending their option code to base
  - Override GetPrice by adding the option price to base
  - DiscountDecorator lowers price in GetPrice

DECORATOR



DECORATOR



Façade Pattern

Friday, November 09, 2018 10:37 PM

FAÇADE PATTERN - PROBLEMS

- You have a complex set of operations or objects to interface with
  - Want to simplify application code
- User registration process
- Process purchase order
- Follow a user in social app

FAÇADE PATTERN

- Provide a unified interface to a set of interfaces in a subsystem.
- Façade defines a higher-level interface that makes the subsystem easier to use.
- Ideally a single class or method for the application interface
  - Runs multi-step operations
  - Multiple classes / methods

Façade Pattern Example

- Social App Following
  - User follows another user
  - Process follow in database
  - Notify "followee"
  - Post the activity
  - Notify followers

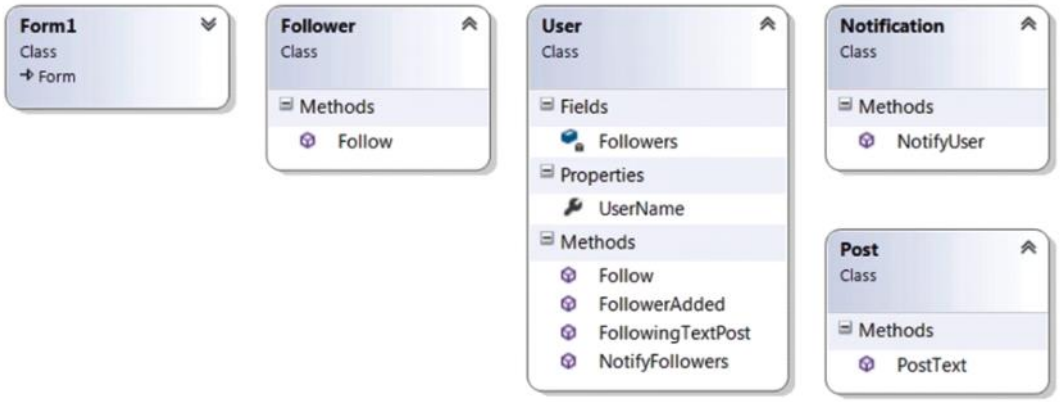
Façade Pattern Example

- User class
  - Multiple methods for following, posting, notifications
- Post class
  - Handles post operations
- Notification class
  - Handles notifications

Façade Pattern Example

- Follower Class
  - Façade class
- Follow method
  - Multi-step process for following
  - Add user as follower
  - Notify user that they have an added follower
  - Post that user is now following a new person
  - Notify all of user's followers of new following activity

FAÇADE


Form1
Class
Form
Follower
Class
Methods
Follow
User
Class
Fields
Followers
Properties
UserName
Methods
Follow
FollowerAdded
FollowingTextPost
NotifyFollowers
Notification
Class
Methods
NotifyUser
Post
Class
Methods
PostText

Flyweight Pattern

Saturday, November 10, 2018 1:17 PM

Flyweight Pattern - Problems

- You need a lot of copies of an object
  - Memory and performance concerns
- List of users
- Game objects
- Posts in a social app

Flyweight Pattern

- A flyweight is a shared object that can be used in multiple contexts simultaneously.
- Instead of re-creating instances, instances are re-used, or shared
- Dictionary of objects
  - Add object to dictionary if not already there
  - Re-use object if already created

Flyweight Pattern Example

- User List
  - Application has lots of users
  - Users are used throughout
  - Need to limit memory footprint
  - Only create user when used

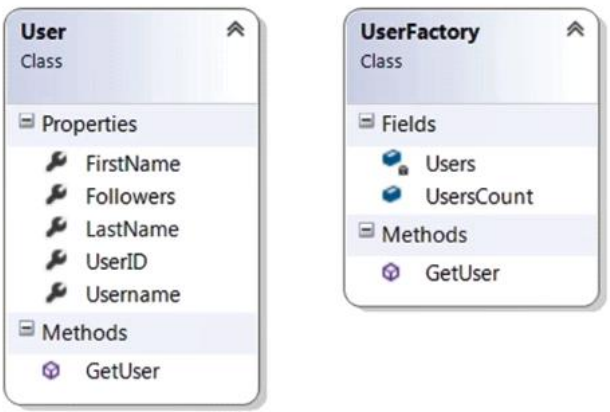
FLYWEIGHT PATTERN EXAMPLE

- User class
  - Normal properties, like username, etc.
  - Follower list
  - Heavy footprint

FLYWEIGHT PATTERN EXAMPLE

- UserFactory Class
  - Purpose is to supply users as needed
  - Internal user dictionary
- GetUser method
  - Checks dictionary – returns user if available
  - Creates user if necessary
  - Adds user to dictionary for future re-use

FLYWEIGHT



Proxy Pattern

Saturday, November 10, 2018 1:57 PM

PROXY PATTERN - PROBLEMS

- You have a process that is complex and/or time-consuming
  - Need a stand-in for application to be responsive
- Complex calculation
- Database on remote server
- Access web service
- Process payment

PROXY PATTERN

- A Proxy provides a surrogate or placeholder for another object to control access to it.
- Allows you to postpone operation until you need it
- Allows the application to be responsive during process
- Application can run while proxy runs asynchronous task

PROXY PATTERN EXAMPLE

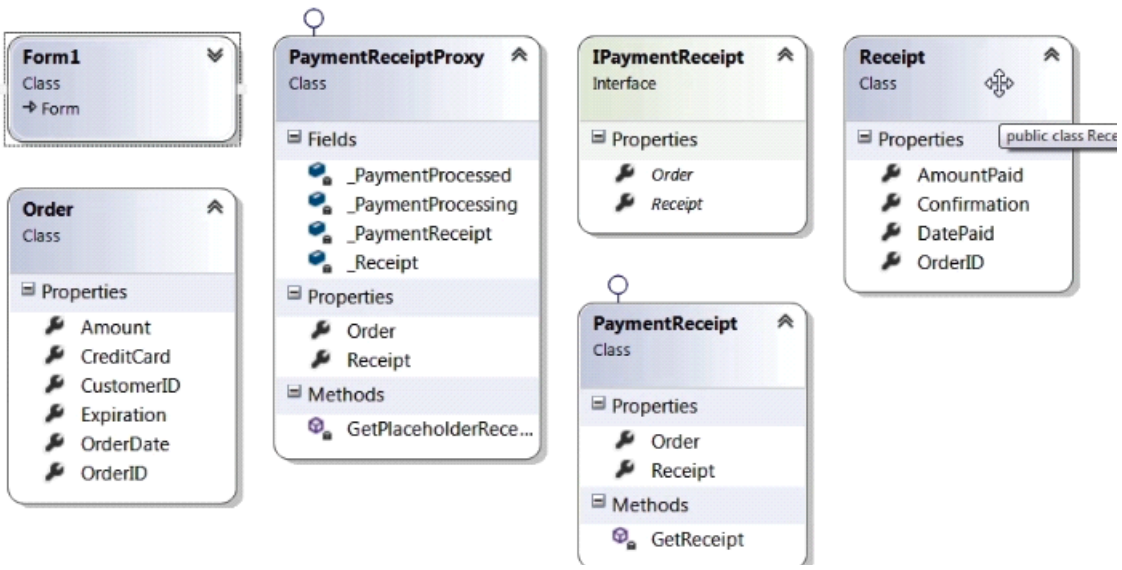
- Payment Processing
  - Ordering Application
  - Uses third party payment processor
  - Asynchronous web service call
  - Show "processing..." while calling service
  - When process is complete, show confirmation

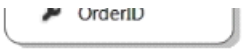
Proxy Pattern Example

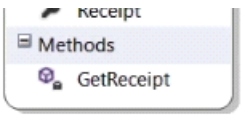
- IPaymentReceipt interface
  - Order and Receipt properties
- PaymentReceipt class
  - Implements IPaymentReceipt
  - Interfaces with payment processor
  - Does web service processing

Proxy Pattern Example

- PaymentReceiptProxy class
  - Implements IPaymentReceipt
  - PaymentReceipt member
- Receipt property
  - If payment processing not done:
    - Get receipt from PaymentReceipt
    - Placeholder receipt




OrderID


Receipt
Methods
GetReceipt

Chain Of Responsibility Pattern

Saturday, November 10, 2018 2:50 PM

Chain of Responsibility Pattern - Problems

- You have a multi-step request process
  - Steps must be done in sequence
  - Don't want application to manage process
- Approval process
  - Purchase
  - Document Review
  - Human Resource
  - Back Office

Chain of Responsibility Pattern

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Application initiates request – waits for answer
- Each receiver processes it and passes to next
  - May stop chain to accept or deny

Chain of Responsibility Pattern Example

- Back Office Material Approval
  - Request for a new material to be added to system
  - Lengthy approval process
  - Many departments

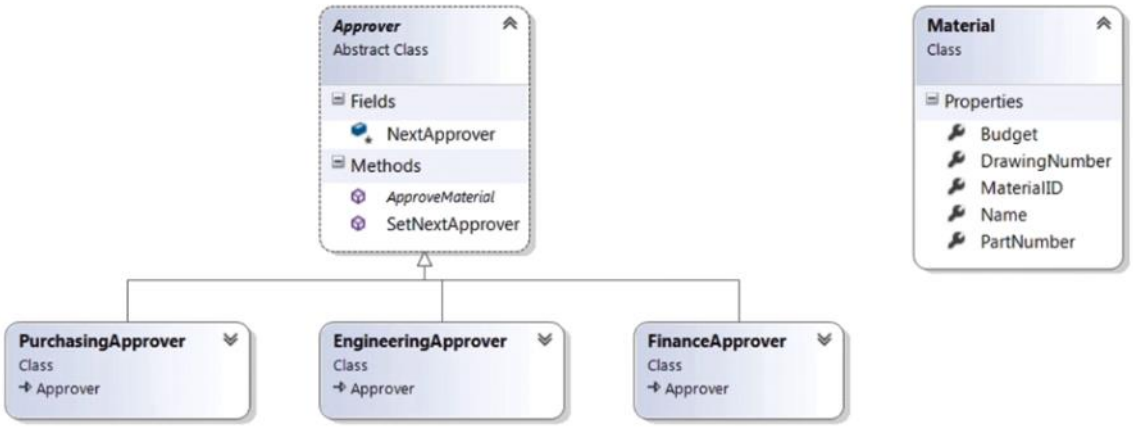
Chain of Responsibility Pattern Example

- Approver abstract class
  - ApproveMaterial method
    Return true or false
    Reason if false
  - SetNextApprover method
    Next in chain of command

Chain of Responsibility Pattern Example

- EngineeringApprover
  - Looks for part numbers and drawing numbers
- PurchasingApprover
  - Approves up to a budget limit
- FinanceApprover
  - Approves up to a larger budget limit

Chain of Responsibility



Command Pattern

Sunday, November 11, 2018 12:19 AM

Command Pattern - Problems

- You want to decouple method calls from application
  - Method = Command
  - Use an Invoker class
  - Command has no parameters
  - Invoker treats them all the same
- Menu
- Messaging
- Queue

Command Pattern

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Commands implement common interface
- Invoker executes command by calling interface method

Command Pattern Example

- Update UI
  - Command pattern used in WPF (Windows Presentation Foundation)
- ICommand interface
  - Control property
  - Color property
  - UpdateColor method

Command Pattern Example

- UpdateBackColor command
- UpdateForeColor command
- ColorInvoker
  - Calls UpdateColor
  - Does not set UI or Color properties

Command Pattern

Interpreter Pattern

Sunday, November 11, 2018 2:08 AM

Interpreter Pattern - Problems

- You need to translate an input into a different output type
  - Not casting or converting
  - More complex – hierarchical
- Language translation
- String command parsing

Interpreter Pattern

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Define grammar or translation instructions
- Hierarchy of expressions
  - Terminal expression – stand alone
  - Non-terminal – contains expressions

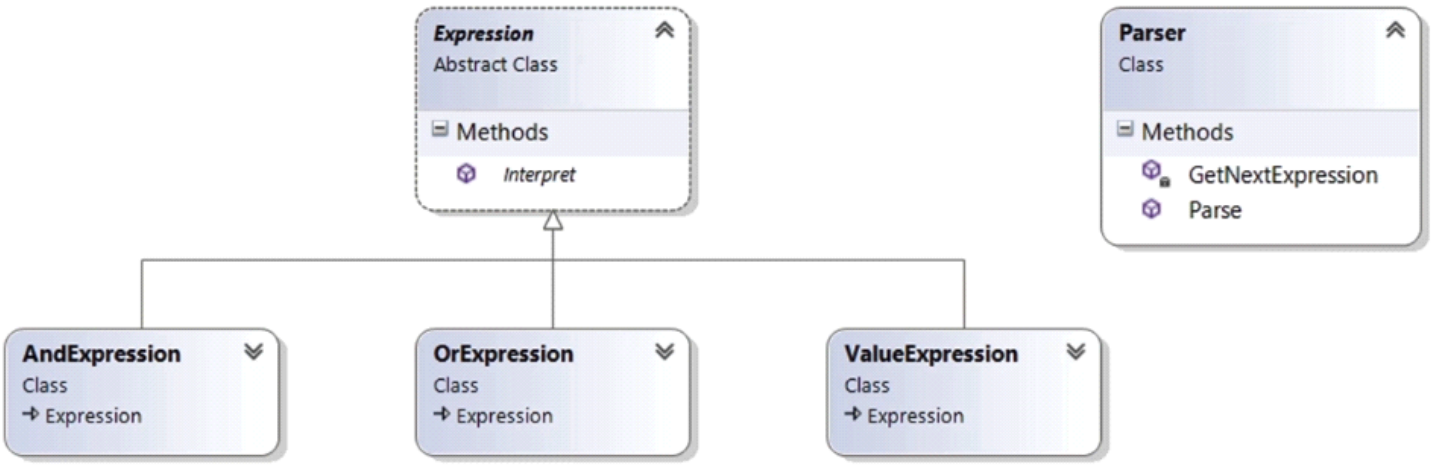
Interpreter Pattern Example

- Parse Simple Logic String
  - AND / OR operators
  - String or numeric literals
- Expression abstract class
  - Interpret method

Interpreter Pattern Example

- ValueExpression
  - Terminal expression
  - String or numeric
- AndExpression
  - And together two expressions
- OrExpression
  - Or together two expressions
- Parser Class

Interpreter Pattern



Iterator Pattern

Sunday, November 11, 2018 8:38 PM

Iterator Pattern - Problems

- You have a collection of objects you need to access
  - Sequential access only
  - No access to inner collection
  - Read only
- Used everywhere!

Iterator Pattern

- Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- Aggregate
  - Contains collection
  - Provides its Iterator
- Iterator
  - Access operations

Iterator Pattern Example

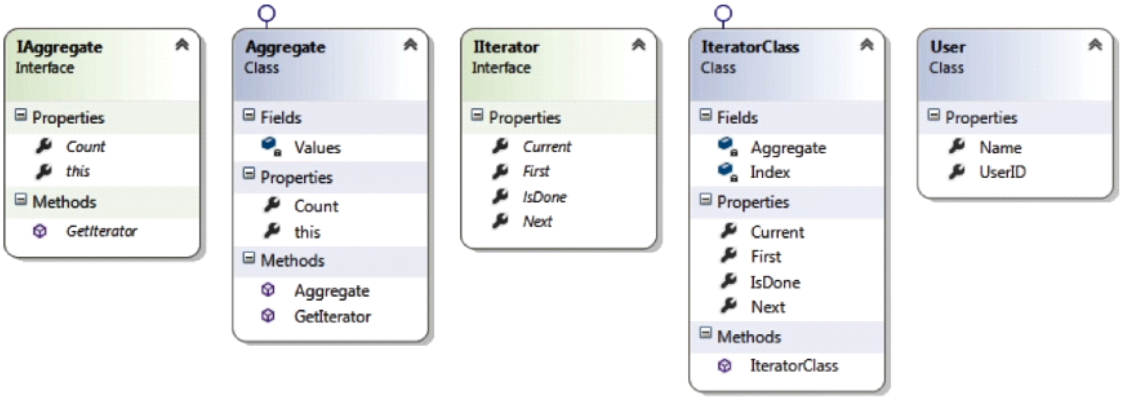
- Basic collection of users
- IAggregate interface
  - GetIterator method
  - Interface for raw collection
- Aggregate class
  - Implements IAggregate

- Aggregate class
  - Implements IAggregate

ITERATOR PATTERN EXAMPLE

- IIterator interface
  - First property – sets index at start
  - Current property – current item in collection
  - Next property – increments index
  - IsDone property – end of collection?
- Iterator class
  - Implements IIterator
  - Aggregate and Index members

ITERATOR PATTERN



Mediator Pattern

Sunday, November 11, 2018 10:12 PM

MEDIATOR PATTERN - PROBLEMS

- You have objects that communicate with each other
  - Sometimes in groups
  - Don't want objects to hold references of other objects
- Messaging
- Chat
- Command/Response

MEDIATOR PATTERN

- Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Mediator has list of objects (colleagues)
- Colleagues don't know other colleagues
  - Communicate with Mediator

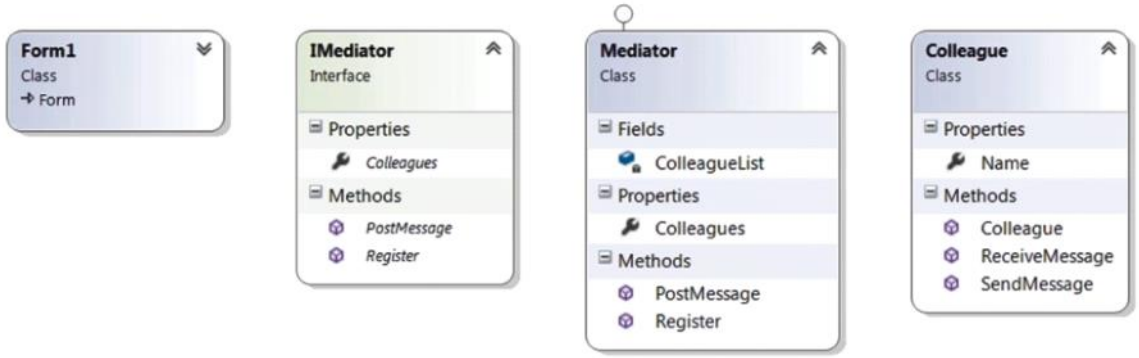
MEDIATOR PATTERN EXAMPLE

- Messaging
  - Group of Colleagues
  - Simple string messages
- IMediator interface
  - Register method
  - PostMessage method

MEDIATOR PATTERN EXAMPLE

- Mediator class
  - Implements IMediator
  - Holds list of Colleagues
- Colleague class
  - Name property
  - SendMessage method – IMediator parameter
  - ReceiveMessage method

MEDIATOR PATTERN



Memento Pattern

Monday, November 12, 2018 7:28 PM

Memento Pattern - Problems

- You want to be able to restore an object to a previous state
  - No outside access to internal state
- Undo operation
- Transaction rollback

Memento Pattern

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- Memento class stores state of object
- Originator is class to be saved
  - Creates Memento
  - Restore method
- Caretaker holds Memento

Memento Pattern Example

- Edit / Undo
  - Edit of user object
- User class
  - Originator
  - UserID, Name, etc. properties
  - SaveState method
  - RestoreState method

MEMENTO PATTERN EXAMPLE

- UserMemento class
  - UserID, Name, etc. properties
  - Constructor with properties as parameters
- Caretaker class
  - UserMemento property

MEMENTO PATTERN

Observer Pattern

Tuesday, November 13, 2018 2:42 AM

OBSERVER PATTERN - PROBLEMS

- You want to be able to see changes in an object
  - Multiple interested objects
  - Automatic notifications
- Data change notification to UI
- Price Watcher
- Monitoring app

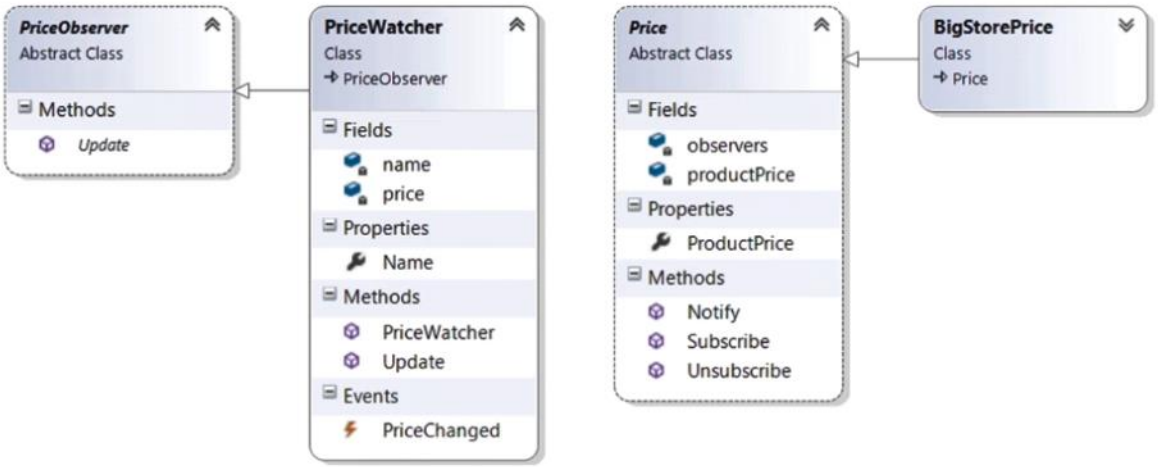
OBSERVER PATTERN

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Subject class
  - Class being observed
  - Contains / manages / notifies observers
- Observer
  - Provides method to call on notifications

Observer Pattern Example

- Price Watcher
- Price abstract class
  - ProductPrice property
  - Subscribe and Unsubscribe
  - Notify method
- PriceObserver abstract class
  - Update method
- PriceWatcher class
  - PriceChanged event

Observer Pattern



State Pattern

Tuesday, November 13, 2018 5:44 PM

STATE PATTERN - PROBLEMS

- You want an object's behavior to change depending on the condition, or state it's in
  - State machine implementation
  - Don't burden object
- Navigation
- Logic Flow
- Validation

STATE PATTERN

- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Context
  - Application interface - contains State object
- State – Abstract class
  - Interface for behavior changes
  - Concrete State classes for each possible state

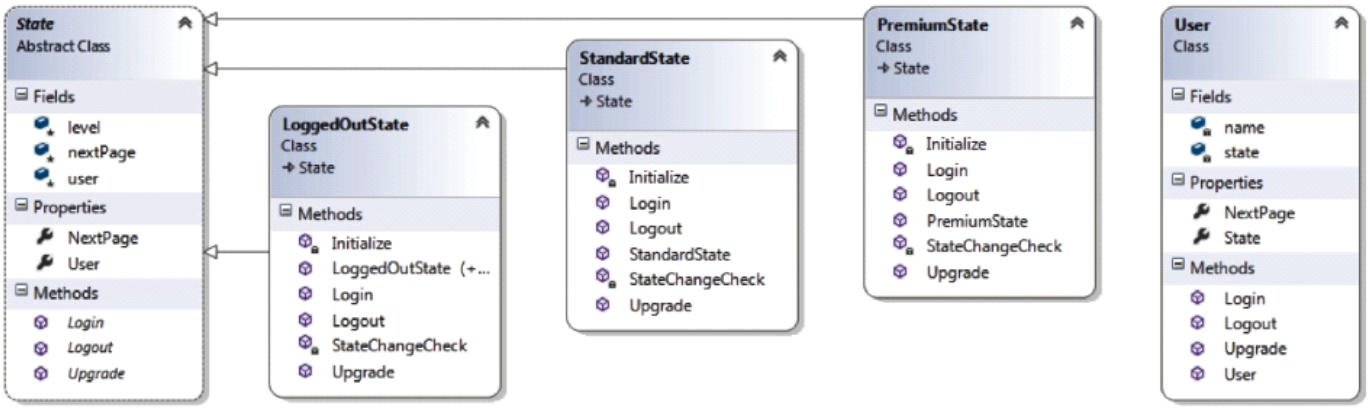
STATE PATTERN EXAMPLE

- App Navigation
  - State determines which page to go to
- User class
  - Can be logged out, standard, or premium
- State abstract class
  - Login, Logout, Upgrade methods
  - NextPage property

- Login, Logout, Upgrade methods
- NextPage property

State Pattern Example

- LoggedOutState class
  - NextPage is login page
- StandardState class
  - NextPage is main page
- PremiumState class
  - NextPage is premium page



Strategy Pattern

Tuesday, November 13, 2018 8:29 PM

STRATEGY PATTERN - PROBLEMS

- You want to be able to choose an algorithm or set of steps to run
  - Common interface
- Sorting type
- Calculations
- Transaction process

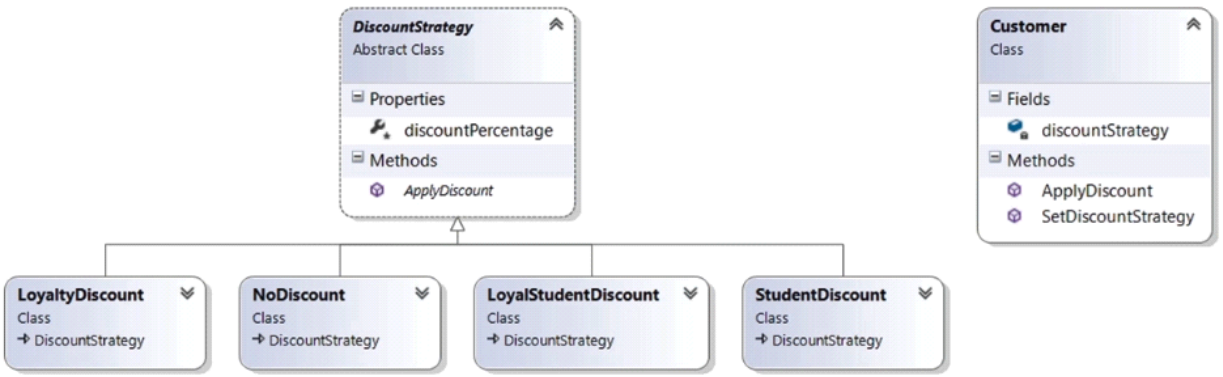
STRATEGY PATTERN

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Strategy abstract class
  - Algorithm interface
- Concrete Strategy classes
  - Implement appropriate algorithm

STRATEGY PATTERN EXAMPLE

- Discount Strategy
  - Apply different discounts
- Customer class
  - Contains discount strategy
  - ApplyDiscount method
- Discount Strategy abstract class
  - ApplyDiscount method

STRATEGY PATTERN



Template Pattern

Thursday, November 15, 2018 6:34 PM

TEMPLATE PATTERN - PROBLEMS

- You want to create a starting place for classes to inherit from
  - Abstract class
  - Inheritance
- One of main OO principles

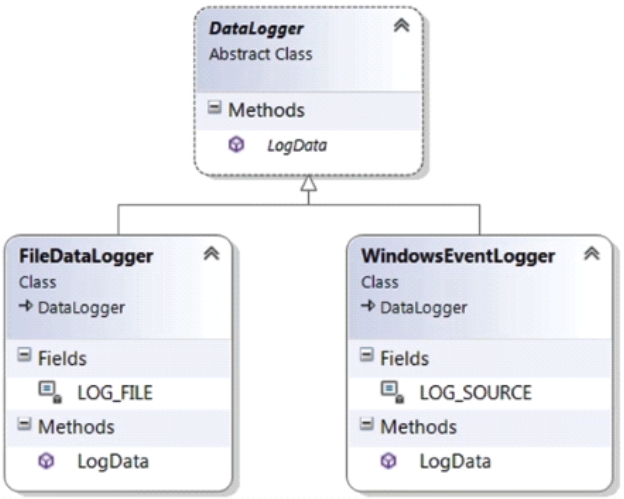
TEMPLATE PATTERN

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Abstract class
  - Defines public interface
  - Provides some implementation
- Inherited classes finish / alter implementation

TEMPLATE PATTERN EXAMPLE

- Data Logger
  - Log application info
- DataLogger abstract class
  - LogData method
- FileDataLogger
  - Logs to text file
- WindowsEventLogger
  - Logs to Windows event log

TEMPLATE PATTERN



Visitor Pattern

Thursday, November 15, 2018 7:26 PM

VISITOR PATTERN - PROBLEMS

- You want to add functions to an object dynamically
  - Separate object from operation
  - Limit object clutter
  - Add functions as needed
- Add different output formats
- Add functions to business objects

VISITOR PATTERN

- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Element class (client object)
  - Accepts visitors
- IVisitor interface
  - Visit method - applies function

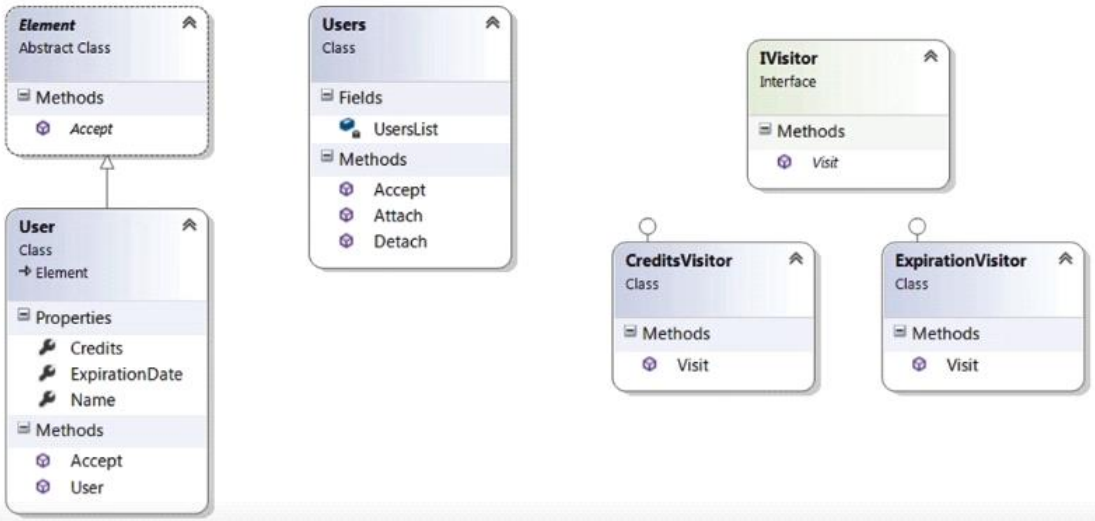
VISITOR PATTERN EXAMPLE

- User object
  - Update credits and expiration
- User class
  - Name, Credits, ExpirationDate properties
  - Accept method – takes IVisitor parameter

VISITOR PATTERN EXAMPLE

- IVisitor interface
  - Visit method
  - User parameter
- CreditsVisitor class
  - Updates credits
- ExpirationVisitor class
  - Updates credits

VISITOR PATTERN



GitHub link

Thursday, November 15, 2018 11:21 PM

https://github.com/adhikari-chayan/Design-Patterns.git