Object Oriented principles and design.

I am in pain.

Solid:

Single Responsibility Principle (SRP):
A class should only have a single responsibility, and only one reason to change.

Open-Closed Principle (OCP):
Objects should be open to extension but closed for modification.

Liskov Substitution Principle (LSP)(VG):
Every subclass or derieved class should be able to substitue for their base or super class.

Interface Segregation Principle (ISP):
A client should never be forced to implement an interface that it doesn't use, nor should clients be forced to depend on methods they do not use. Many client-specific interfaces are better then one general-purpose interface.

Dependency Inversion Principle (DIP):
One should depend on abstractions, rather than on concretions.

Other:

Seperation of Concern (SoC):
Concerns of the software should be split up into individual parts. One should not write their software as one giant block and should instead split the code into smaller more distinctive parts.

Command-Query Separation Principle (CQSP):
A method should either return data or modify data(void in most cases), but not both.
Something that breaks this principle is for example the `pop()` method, that exists for lists in many languages, because it both modifies the list by removing the last index and then it returns that data. This is however allowed because there simply is no better solution.

High Cohesion, Low Coupling (HCLC):
Low Coupling: Modules and classes should be as independent from each other as possible.
High Cohesion: Related code should be close together. For example all methods in a module should be related to the modules distinct task.

Composition over Inheritance (CoI):
Classes should preferably contain other classes that implement needed behavior rather then inheriting behavior.

Law of Demeter (LoD):
To avoid loose coupling classes should only talk to their immediate "friends".
For example: instead of writing `a.b.c.d()` one should do `a.tellBtoTellCtoDoD()`, as one should avoid relying on other classes implementation. This also ties into information hiding, as in for example if something doesn't need to be `public` it shouldn't be `public`. Package- or class-private should always be prefered.

Defensive Programming (DP):
Software and libraries should be written as if the user **will** use it wrongly. For example, if data should only gettable instead of settable, a getter method should be implemented and the data should be hidden.

Design Patterns

Template Method
This pattern suggests that you break down your algorithms into smaller steps, and each step is it's own method. These may either be abstract or inheritable, but the idea is that the client supply their own subclasses with their own implementations of the steps.

Bridge
The bridge pattern is a design pattern used in software engineering that is meant to "decouple an abstraction from its implementation so that the two can vary independently". The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

Strategy
This suggests that algorithms should be able to be selected during runtime. Instead of implenting a single algorithm directly, an algorithms should be abled to be selected during runtime.

State
This allows objects to change their behaviour when its internal state changes. This could be seen as an implementation of the Strategy Pattern.

Decorator
Decorator pattern allows for objects to implement behavior dynamically without affecting other objects of the same class. This can be useful for the Single Responsibility Principle as it allows for objects to have dynamic different areas of concerns, without the need for subtyping.

Adapter
This is used to wrap an interface of another class to be used as another interface. It's used to make existing classes work with each other without modifying their source code.

Factory (Method)
The factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

Singleton
This restricts a class to only be created once. This is used to coordinate actions across the whole program easily. Could be considered a "Global State". Often critized however.

Iterator
In object-oriented programming, the iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements. The iterator pattern decouples algorithms from containers; in some cases, algorithms are necessarily container-specific and thus cannot be decoupled.

Composite
This is a structural pattern where a group of objects are treated in a similar way as a single object. This is often represented as a tree structure.

Module
This design pattern is used to wrap a set of function, variables and classes in a module, similarly to packagess in Java.

Facade
A facade is an object that serves as a front-facing inteface used to mask more complex underlying code.

Model-View-Controller
Model
The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.
View
Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
Controller
Accepts input and converts it to commands for the model or view.

Observer
The observer pattern is a software design pattern in which an object, named the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

Design Patterns (VG)

Entity

Ko

Aggregate

Ko

Chain-of-Responsibility

The Chain-of-Responsibility pattern consists of a source of command objects and a series of processing objects. Each processing objects contains logic that defines the types of command objectss that it can handle, and the rest are passed to the next procesor in the chain. There should also be a mechanism for adding new processing objects to the end of the chain.

Visitor

This pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying the structures. It is one way to follow the open/closed principle.

Abstract Factory

This provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes. In normal usage,the client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme. The client does not know (or care) which concrete objects it gets from each of these internal factories, since it uses only the generic interfaces of their products.

Command

This is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

Servant

A servant object is one that offers some functionality to a group of objects without defining that functionality in them. The object in question are taken as a parameter by the servant. For example, an object that moves polygons, instead of the polygons doing that themselves as that is not their area of concern.

Builder

This suggets that the creation and assembling of complex objects should be in its own builder object, and thuss a class delegates it's creation to this object. By doing this the construction and representation can be separated, and the same construction process can be used to create different representations.

Mediator

To reduce coupling objects should not communicate directly with each other, but instead communicate through a mediator object.

Publish-Subscribe

This is a messaging pattern where senders of messages, called publishers, do not program the message to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classses without knowledge of hich subscribers there may be. Similarly, subscribers express interest in one or more classes and only recieve mesages that are of interest, without knowledge of which publishers, if there are any.

Concepts

Class and object
The class is like a template for objects.
Static and dynamic type
Ko
Encapsulation
Ko
Specification and Implementation inheritance
Ko
Static method and instance method
Ko
Variables and attributes
Ko
Abstract class and interface
Ko
Dynamic Binding
Ko
Primitive and referencetypes
Ko
Alias
Ko
Super and subclass
Ko
Overloading and overriding
Ko
Constructor
Ko
Initialization
Ko

Mechanisms and techniques

Mutability and immutability
Ko
Refactoring
Ko
Defensive Copying
Ko
Immutable Adapters
Ko
Exceptions
Ko
Method Cascading
Ko
Mutate-by-copy
Ko
Lambdas and function interfaces
Ko
Design by contract
Ko
Algebraic data type and object hierarchy(VG)
Ko
Streams
Ko
Threads and threadsecurity
Ko

Polymorphism and code reuse

Polymorphism
Ko
Subtype-polymorphism
Ko
Parametric polymorphism
Ko
Ad hoc-polymorphism(VG)
Ko
Co-,contra-, and invariant
Ko
Delegation
Ko
Inheritance
Ko
Type constructor and type parameter
Ko
Type argument and Wildcards
Ko
Upper and lower bounds
Ko
Get-Put principle
Ko