Jimdustry
Programming.

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
Abstractions should be decoupled from it's implementation, so the two can vary independently.

Return to this!

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
Ko

Module
Ko

Facade
Ko

Model-View-Controller
Ko

Observer
Ko

Design Patterns (VG)

Entity
Ko
Aggregate
Ko
Chain of Responsibility
Ko
Visitor
Ko
Abstract Factory
Ko

Command
Ko
Servant
Ko
Builder
Ko
Mediator
Ko
Publish-Subscribe
Ko