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)(*):
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 method 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
Ko

Strategy
Ko

State
Ko

Decorator
Ko

Adapter
Ko

Factory (Method)
Ko

Singleton
Ko

Iterator
Ko

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