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.

Concepts

Class and object

The class is like a template for objects.

Static and dynamic type

Static typed languages are those in which type checking is done at compile-time, whereas dynamic typed languages are those in which type checking is done at run-time

Encapsulation

Encapsulation means wrapping data together as a single unit. Specifically data should also be hidden and getters should be supplied.

Specification and Implementation inheritance

Implementing an interface or abstract class or inherting code from classes/abstract classes.

Static method and instance method

Static methods are a single copy of a method and can not access instance variables but can be used without an instance, complete opposite of an instance method which can access instance variables and requires an instance.

Variables and attributes

Attributes are instance variables while variables are declared in methods.

Abstract class and interface

An interface is a form of specification inheritance where what methods a class needs to have is defined, similarly an abstract class can do this while also having methods and attributes that are inherited. But can be used as a type but neither can be instanced.

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
Values that can be modified and or not modified(const/final).

Refactoring

Alterning internal code without changing its external behavior.

Defensive Copying
To ensure immutability, getter methods should return a copy instead of a reference the value in question.

Immutable Adapters
An implementation of the adapter pattern, where an interface is created to only expose immutable and safe methods of the underlying class.

Design by contract
This is a designphilosophy where a methods specifications are seen as a contract between the client and the method.
"If you promise to give me two positive integers, I promise to give you their biggest common divisor."

- Precondition(Förvilkor): Things that must be fullfiled for the method to be called.
- Postcondition(Eftervilkor): If the precondition is not fullfilled there is no guarantee on the return.
  Otherwise the return should be able to be guaranteed.
- Invariant: The needed requirements should be in a wellformated state.

Method Cascading
Calling multiple methods on the same object. For example:

$\boxed{\text{a.b().c();}}$ instead of $\boxed{\text{a.b();a.c();}}$

Mutate-by-copy
Instead of mutating an object's state it should be copied with the new state.

Lambdas and function interfaces
Short-form replacement for anonymous classes for interfaces with single methods:

```
interface k { int p(double g); }
List<k> l = new List<>();
l.push((double g) -> return (int)g;);
```

Exceptions
An error that is thrown whenever something that should not happen happens. These should be handled using try catches(varies between languages).

Polymorphism and code reuse

Polymorphism

Polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

Subtype-polymorphism

Being able to take the superclass or any of its subclasses and still work.

$\boxed{\text{functionThatTakesA(new BThatInheritsA())}}$

Parametric polymorphism

The use of generics, ie $\boxed{\text{T extends A}}$.

Ad hoc-polymorphism(VG)

Different types should be able to give defenitions of the same symbol/name without knowing of each other.

(See operator overloading in C# and type classes in Haskell)

Something like this:

$\boxed{\text{<T implements +> T add(T x, T y) = x + y;}}$

This is however not possible in Java.

Co-,contra-, and invariant

Covariant: ? extends T
Contravariant: ? super T
Invariant: T (No sub or supertypes)

Delegation

Delegation means handing over responsibilities for a task to another class or method.

Inheritance

Inherting means making a class a subclass of another, and inherting methods and such from the super.

Type constructor and type parameter

$\boxed{\text{class Box<T>\{\}}}$: The T here is a type parameter.

Type argument and Wildcards

$\boxed{\text{Box<Integer> b = new Box<>();}}$: The Integer is in this case a type argument.

Wildcards are used for creating generic declarations and method calls but not for defining generic type, for example: $\boxed{\text{List<? extends A>}}$ but not $\boxed{\text{class A<? extends B>}}$!

Upper and lower bounds

Upper bound wildcard meands $\boxed{\text{? extends Type}}$ and lower bound means $\boxed{\text{? super Type}}$.

Get-Put principle

Use $\boxed{\text{List<? super T>}}$ to only be able to put data in the list and $\boxed{\text{List<? extends T>}}$ to be able to get data out of the list.