1. General notes

* Gamma categorization of design patterns:
  + Creational patterns
    - Deal with the creation (construction) of objects
    - Explicit (constructor) vs implicit (DI, reflection etc.)
    - Wholesale (singe statement) vs piecewise (step by step)
  + Structural patterns
    - Concerned with the structure (e.g., class members)
    - Many patterns are wrappers that mimic the underlying class interface
    - Stress the importance of good API design
  + Behavioral patterns
    - They are all different, no central theme

1. SOLID

* Single responsibility principle
  + Class should have only 1 reason to change
  + Different tasks handle different independent tasks, problems
  + Example is having a journal with title and add\_entry() method. Bad idea is to add new functionality to this class for saving journal to file. In case we have a lot of classes operating on strings, it would lead us to copying this functionality to other classes. It is better to create separate class for new concern (interfacing with files).
* Open – close principle
  + Classes should be open for extensions (by inheriting for example) but closed for modifications
  + It is better to not come back to same class as it is already tested and also client maybe would have to recompile client program to use new library
* Liskov substitution principle
  + You should be able to substitute a base type with a inherit class
  + If we have class square and rectangle, it’s better to make those 2 classes inherit from shape instead of square inherit from rectangle (square only 1 member, rectangle 2)
* Interface segregation
  + Don’t put to much into 1 interface, split into separate interfaces
  + Example with IMachine witch were able to print(), scan() and fax(). Every class inheriting from this interface has to implement all of these function even if it is only printer
* Dependency inversion principle
  + High level modules should not depend on low level ones, use abstractions

1. Builder

* Instead of creating a component in user code, you create specialized class/structure to create (build) this component (encapsulate this component) and work with whis component
* If you want to force user to not use component, you can make ctrors private and add friend class builder into component
* You can either give builder a constructor and initialize components members in it, or you can return builder via static function (components member)

1. Factory

* Used when creation logic is too complicated and/or constructors would be not descriptive (constructors has the same name as component, you cannot overload them with same args with diffrent names, can turn into ‘std::optional hell’)
* Object creation (non-piecewise, unlike Builder) can be outsourced to:
  + Separate function (Factory method)
  + Separate class (Factory)
  + Hierarchy of factories (Abstract Factory)

1. Prototype

* Interface needs to implement prototype in means partially create object and store it somewhere to copy
* User has to copy prototype (already created object – then he can customize fields of this object in his favour, but for all other fields will be already filled with some values)
* To make copy possible, interface has to implement copy constructor or implement serialization methods

1. Singleton

* Only one component in the system (e.g. database or object factory)
* E.g. when constructor is very expensive
* Can be implemented with hiding ctrs (copy as well) in private part. Then providing member of class Singleton\* singleton and static method if (singleton == nullptr) -> singleton = make\_unique<Singleton>() and return singleton beside if. This way every caller of this static method will get same instance of Singleton (first will allocate memory for it).