Date

→ Design Patterns

- ~~[illegible]~~ choose the design pattern according to the problem faced and this will help in
  ↳ maintability, Scalability, reusability etc is increased
- Similar things are combined to increase efficiency
- "each pattern describe a problem which occur over and over again in your environment, software or application and then describe the core solution to that problem in such a way that you can use this solution a million time over without ever doing it the Same way twice"
- "design pattern's thumb rules are different concepts using which you can solve the problem of modelling real world examples into object oriented design"

↳ essential elements of patterns

① pattern name: it is a handle which we can use to describe a design problem in a word or two. Naming a pattern immediately increases our design vocabulary. It makes it easier to think about designs and to communicate them and their trade off to others. Finding good names has been one of the hardest part of developing our catalage. ~~The problem~~

② The problem describes when to apply the pattern. It explains the problem and it contexts. The problem will include a list of conditions that must be met before. It makes sense to apply the pattern.

③ Solution: The Solution describes the elements that make up the design, their relationship, responsibilities and collaborations. The Solution does not describe a particular concrete design or implementation because a pattern is like a template that can be applied in many different situations.

Date

④ Consequences: The consequences are the results and trade offs applying the pattern. Consequences are often invoiced when we describe design decisions. They are all critical for evaluating design and for understanding the cost and benefits of applying the pattern.

↳ Classification of design patterns

① Creational design patterns
② Structural design patterns
③ Behavioral design patterns

} These are different from each other on the basis of their level of detail, complexity and scale of applicababilty to the entire system being designed

① Creational: As the name ~~sufects~~ it provides the object of classes mechanism that enhance the flexibility and reuseability of existing code. It reduces the depenency and controlling how the user interaction with our classes so we would not deal with complex constructions. Its types are as follows:

↳ abstract factory.
↳ Builder
↳ factory method
↳ proto type
↳ ~~Sig~~ Singleton

② Structural: They are mainly responsible for assembling objects and classes into a larger structure making sure that these structures should flexible and efficient. They are very ~~[illegible]~~ echancing readability and maintainability of the code. It also ensures that functionalities are properly seperated and encapsulated. It reduces the minimal interface between independent things. Structural design are used for structuring more than one classes or objects together. Moreover we will

Date

develop deep inheritence and ISP (Interface Segregation) etc
(can be other solid principles as well). Types

↳ Adapter
↳ Composite
↳ Bridge
↳ Decorater
↳ facad
↳ flywait
↳ proxy

③ Behavioral : They are responsible for how one class communicates with others. It is also used for identifying and setting up Common communication pattern among objects. They deal with class and object interaction and distribute their responsibility

↳ Chain of responsibility
↳ Command
↳ interpreter
↳ iterator
↳ mediated
↳ memnto
↳ observer
↳ State
↳ Strategy
↳ template method
↳ Visitor

↳ Benefits :

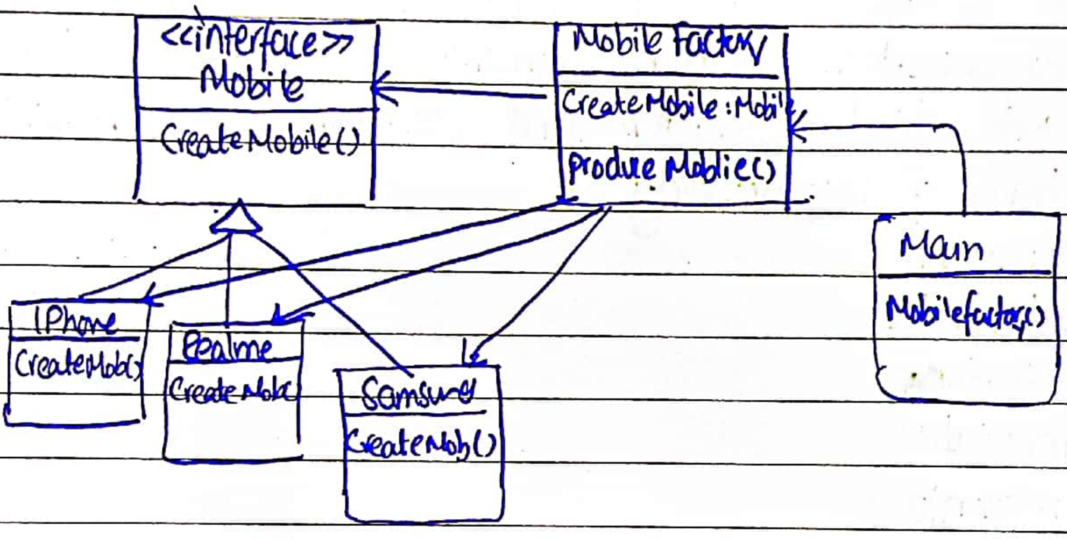
- improves the performance of the system
- ~~to~~ slove the bottleneck of the problem
- best design for the system is possible
- improve the code for writing in a more object oriented way like inheritance and encapsulation

Date

- development process is Speed up with well designed principles
- ~~sch Such as~~ Clear Seperation of modules and loosely Coupled system
- reuse things across all applications or modules

→ Factory Method

- Creational design pattern that provides an interface for creating object in a Super class but all Subclasses to alter the type of object that will be Created.



↳ Advantages

- you avoid the tight coupling between^ the creater and concrete^ classes (~~or concrete product~~
- Single responsibility Principle and Ocp not violated
- you can move the product creation code into 1 place in the program making the code easier to Support
- you can make new types of products into the program without breaking existing client code

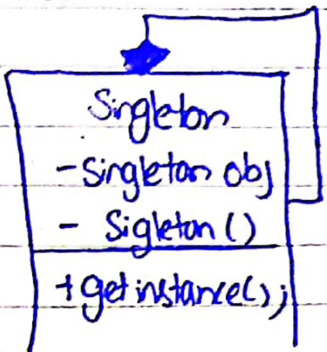
↳ Disadvantages

- the code may become complicated since you need to introduce alot of nuse of classes to implement the pattern.

Date

· the best Senario is when you ~~are~~ can introduce the pattern into an existing hierarchy of creater class

→ Singleton Pattern



· making private constructor would mean that It cannot be called from public space (outside the class)

```
Class Singleton{
     Singleton * obj;      to solve multithreading issue or use mutex lock or use "synchronized"
                           → Singleton *obj = new obj();
  private:
       Singleton() { };
  public:
  Singleton ~~Void~~ getinstance (Singleton obj){
            if (obj == null){
                Singletonobj = new obj;}
         return obj;}     };
```

```
Singleton obj1;
Obj1.getinstance();
Obj1.get instance(); }
```
will return sam as obj has already made

· only one Singleton object can be made
· Can be problem in multi~~threa~~threading if if(obj==null) run on the Same time ~~with~~ in multiple threads. Singleton pattern ~~Voited~~ Violated as more than one objects will be formed.

↳ Advantages
· flexibility: Since the classes control the intenciation of ~~problem~~ process
· Save meomory – no more that one Object can be formed
· ~~[illegible]~~

Date

→Structural Design Pattern
- Structural design pattern are ease to design by identifying a simple way to describe relationship between entities
- flexible interconnecting modules which can work in larger system

↳Adapter design Pattern
- match interfaces of different classes, the adoptor design patterns allows in compatible classes to interact with each other by converting the interface of one class into an interface expected by the client
- ~~It~~ improves reuseability of older functionality.
- A class needs to be reused which does not have an interface.
- We need to work through a seperate adoptor that adopts the interface of the existing class without ~~it~~ changing it.
- Client does not know whether the work with ~~a~~ target class. directly or through another alternative that does not have the target interface
- Advantages

① Allows us to use code from 3rd parties relatively
② ~~then without~~ No any need to change existing classes to use new code
③ Loose coupling and ~~Single~~ SRP is enforced in this pattern
④ A change in either the client interface or the adoptee only means a change to the adopter
⑤ It allows the reuseability of an existing code and functionality
⑥ We can isolate the interface from the data conversion code, thus supporting the SRP

Date

⑦ We can introduce new variants for adoptors in our application without breaking the exisiting client code
· Disadvantage
① It's some time easier to just manually refactor the client interface, than it is to use the adoptor pattern.
② If you are using the class adoptor pattern you will want to avoid dimonad inheritance at all costs. This is unlikely to happen but if it does, you will be in trouble.
③ This design pattern increases the overall complexity of the code.
④ It is relatively ~~the~~ simpler to change the service class the matches the rest of your code.

↳ Composite Pattern
· It lets you compose objects into tree structures and the work with these structures as if they were individual objects
· It is used where we need to treat a group of object in similar way as a single object
· Tree structure represents part as well as whole hierarchy. A thing made up of several parts or elements
· It is a partitioning design pattern and it describes that the group of object are treated same way as a single ~~et~~ instance as the same type of the object.
⇒ Implementation guidelines
· represents part whole hierarchy of objects
· clients ignore the difference between the compositions of objects and individual objects.

Date

· Advantages
① Reduces code complexity by eliminating many loops over the homogeneous collection of object
② This intern increases the maintainability and tesability of code with fewer chances to break existing running & tested code
③ The relationship is described in the composite design pattern isn't a sub class relationship, it a collection of relationship which means client or API user does not need to care about operations like translating, rotating, scaling and ~~drying~~, drawing whether it is a single object or a collection of objects
④ Simplified hierarcial objects
⑤ flexible object structure
⑥ encapsulation
⑦ recurssive opeerations
⑧ reuseable code
· Disadvantges
① limited type checking
② The use of common interface for both leaf and composite objects limits the type checking capabilities at compile time. It may not be possible to distinguish between individual objects and a group of objects at compile time
③ Performance overhead espically when the hierarcical structure is larger and deeper
④ The recurssive nature of operation can ~~zalt~~ result in multiple traversal of hierachy impacting the system performance.
⑤ Complexity in structure modification: modifying the structure of composite objects dynamically can be complex.

Date

Adding / removing objects from hierarchy may require careful management and updating of references.

⑥ lack of transperency: it hides the difference between individual objects and the groups of objects by providing a uniform interface. While this simplifies client code but it also reduces transperancy. Clients may not be aware of the internal structure of the hierarchy or the specific objects they are interacting with which can ~~to~~ make debugging and troubleshoting more challenging.

⑦ Diffculting in designing component interface: designing a component interface that caters to both indvidual and group of objects can be challenging. finding the right balance and defining operations that are relavent and meaningful for all components in the hierarchy requires careful consideration and design ~~conside~~ decisions