SCS 2203 – Mind Map

Software Design Patterns				
		Applicability	**Pros**	**Cons**
Creational Pattern	**Factory Pattern** – a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.	• When you don't know beforehand the exact types and dependencies of the objects your code should work with. • When you want to provide users of your library or framework with a way to extend its internal components. • When you want to save system resources by reusing existing objects instead of rebuilding	• Avoids the tightly coupling between the creator and concrete products. • Single responsibility principle. • Open/closed principle	• The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern

		them each time.		
	Singleton pattern – lets you ensure that a class has only one instance, while providing a global access point to this instance.	• Used for logging, drivers objects, caching, thread pool • Used in abstract factory, builder, prototype, façade • When a class in your program should have just a single instance available to all clients • When you need stricter control over global variables	• Can be sure that class has only one instance • You gain a global access point to that instance • Singleton object is initialized only when it's requested for first time	• Violates the singleton responsibility principle • Can mask bad design, for instance, when the components of the program know too much about each other. • Pattern requires special treatment in a multithreaded environment so that multiple treads won't create a singleton object several times.
	Abstract factory pattern - Lets you produce families of	• When your code needs to work with	• Can be sure that the products you're	• The code may become more

	related objects without specifying their concrete classes	various families of related products, but you don't want it to depend on the concrete classes of those products – they might be unknown beforehand, or you simply want to allow for future extensibility.	getting from a factory are compatible with each other. • Avoid tight coupling between concrete products and client code • Single responsibility principle • Open/closed principle	complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.
	Builder pattern – lets you construct complex objects step by step. Allows you to produce different types and representations of an object using the same construction code.	• To get rid of a "telescoping constructor" • When you want your code be able to create different representation of some product • To construct composite trees or	• Can construct objects step-by-step, differ construction steps or run steps recursively • Can reuse the same construction code when building various representations of products	• The overall complexity of the code increases since the pattern requires creating multiple new classes

		other complex objects	• Single responsibility principle	
Structural pattern	◊ Adapter pattern ◊ Bridge pattern ◊ Composite pattern ◊ Decoder pattern ◊ Façade pattern ◊ Flyweight pattern ◊ Proxy pattern			
Behavioural pattern	◊ Chain of responsibility pattern ◊ Command pattern ◊ Iterator pattern ◊ Mediator pattern ◊ Memento pattern ◊ Observer pattern ◊ State pattern ◊ Strategy pattern ◊ Template method pattern ◊ Visitor pattern			