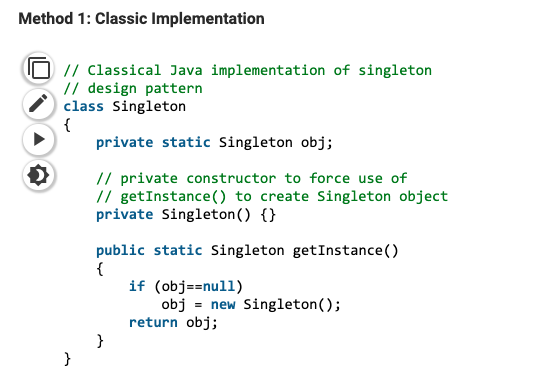
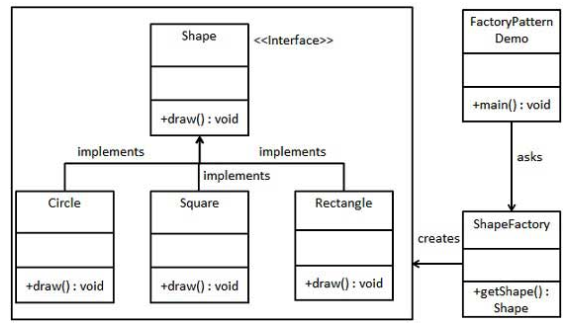


**Common Design Patterns:**

* Singleton: only one instance for the class; can be accessed globally
  + **pro**: only one instance 🡪 save memory; no need to create&delete instance repeatly
  + **con:** no interface, cannot inherit; not **thread safe**

****

* Factory Method: create object without exposing logic to the client using the common interface
  + when? we want to make different instance basing on the condition
  + how: **children class** make the factory interface



* Abstract Factory: factory of factories; best way to create an object
  + purpose: provide an interface to create multiple dependent objects; no need to explicitly specifying their classes
  + solve: interface choosing problem
* Observer:
  + purpose&when: when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically
  + solve: one object change; others are notified 🡪 high cohesion & low coupling

**Creational:**

* **singleton**: one instance for one class; can be accessed globally
  + when? Eg: *logging page*
  + pro: save memory; no need to create/delete instance repeatly
  + con: no interface, cannot inherit; not **thread safe**
* **factory:** 
  + when? Superclass has multiple subclasses, we need to return one subclass
  + how? Defines an interface for **object creating**, the subclass decides which one to use to create the instance
  + pro: perfect example of **Open-Close Principle**
    - **Open**: Writing extendable code by using inheritance: adding subclasses
    - **Close**: do not change the original code
  + Con: complex system, since 2 classes are created when adding new item
* **Abstract Factory:**
  + How? Provide API, clients can create objects using the API without knowing detail
  + Pro: high-cohesion & low coupling **(Dependency Inversion Principle)**
  + Con: hard to add new behavior (need to change the interface and all subclasses)
* **Builder:** separates the complex object structure **from representation??**
  + When? For complex object structure; objects with similar structure
  + How? Encapsulate the creating process into different methods
  + Pro: the object/product creating process clear
  + Con: result in a huge system, hard to ctrl
* **Prototype:** 
  + When? Huge object, high creating cost; but we need to use them frequently
  + Pro: simple object creation process; more efficient
  + Con: ---

**Structural:**

* **Adaptor:** *change the interface of one class to adapt another interface*
  + When?
  + How?
  + Pro: improve the **reusability** of the class
  + Con: ---
* **Bridge:** separate objects interface from its implementation
  + How? Changes inheritance to related
  + Pro: encapsulation, low coupling, reduce the number of class (reduce amount of code)
* **Composite:** a tree structure of simple and composite objects (*recursive*)
  + Pro: clearly define hierarchy structure classes; easier to add new class
  + Con: **abstract** class structure
* **Decorator**: add responsibility to objects dynamically
  + Pro: perfectly inherit parents’ behavior
  + Con: creating many small objects 🡪 increase the complexity of the system
* **Façade**: single class represents the entire subsystem
  + How? Provide unified interface for the entire subsystem
  + Pro: client can easily access subsystem thru the interface; decoupling (client & subsystem)
  + Con: contradict to Open-close principle; **newly added subsystem requires change on Façade class**
* **Flyweight:** Sharing same/similar objects
  + When? Too many same/similar take too much rescources
  + Pro: reduce the number of objects
  + Con: result in more complex system; external access: longer loading time
* **Proxy**: an object represents another object
  + How? Client cannot access the object directly
  + Pro: protect the class (security); reduce coupling

**Behavioral:**

* **Visitor:** 
  + When? We need to perform an operation on a group of similar objects; if the system data structure is stable, but operation always change
  + Pro: easier to add algorithms
  + Con: ---
* **Template:** separate steps and encapsulate into functions or classes; inherit same behaviors
  + How? Superclass: same logic; subclasses: different logics
* **State**: change object’s behavior when its state changes
  + Pro: reduce if…else…; easier to add new states
  + Con: increase the number of classes; complex system and implementation
* **Observer**: notify change to several classes
  + How? One object changes, all observers will be notified
  + Pro: low coupling (observer & subject)
  + Con: ---
* **Memento**: restore the previous state for objects
  + How? Capture object’s internal state; save it outside of the object
  + Pro: encapsulating unnecessary information from the user
  + Con: high memory/resource consuming
* **Mediator**: defined simplified communication between classes
  + When **NOT**? Many to many relationship **NOT** mediato
  + Pro: classes do not need to interact with each other (they only connect with **Mediator**)
  + Con: **Mediator** is overwhelming complicated; it crashes the whole system crashes
* **Interpreter**: define syntax for language
  + When? Used in OOP compiler
* **Command:** encapsulate command
  + How? Hide command processing process, only knows who is the receiver
  + Pro: **decoupling** between **invoker** and **receiver**
  + Con: system contains too many **commands (used)**
* **Responsibility chain:** passing requests along chains of objects until 1 object can process the request appears
  + Pro: decouple **(invoker and receiver);** simpler object structure (just refer the subclass)
  + Con: cannot make sure every request can be processed; hard to observe

**AGILE Software Development (mindset)**

* **Related Framework**: Scrum, Extreme Programming, Future Driven Development
* **Agile is a Mindset:**
* **12 Agile Manifesto**
  + checking *iNotes/development concepts*