Singleton

Uses static attributes to bind vars to the class rather than the class instances. When accessed uses class name without prior instantiation. Uses <<static>> in uml, uses private constructor and private final instance with a getter to fetch the data

```
class Singleton {
  private static Singleton INSTANCE;
  private Singleton() {}
  public static Singleton getInstance() {
    if ( instance == null ) instance = new Singleton();
    return instance;
  }
}
```

Observer, publisher/subscriber

Uses a subject to notify observers of changes. Subject has a list of observers and notifies them when it changes. Observer has an update method that is called by the subject when it changes. Subject has methods to add and remove observers.

```
interface IObservable { /* add, remove, notify */ }
class Observable implements IObservable {
  private List<Observer> observers;
  public void notifyObservers() { observers.forEach(o -> o.update()); }
}
interface IObserver { /* update */ }
class Observer implements IObserver {
  private IObservable observable;
  public Observer(IObservable o) { observable = o; }
  public void update() { /* do something */ }
}
```

Proxy pattern

Uses a proxy to control access to an object. Proxy has the same interface as the object it is proxying. Proxy can be used to control access to the object, to cache the object, or to delay instantiation of the object.

```
interface ISubject { /* doSomething */ }
class Subject implements ISubject { /* doSomething */ }
class Proxy implements ISubject {
  private ISubject subject;
  public Proxy(ISubject s) { subject = s; }
  public void doSomething() { subject.doSomething(); }
}
```

Command pattern

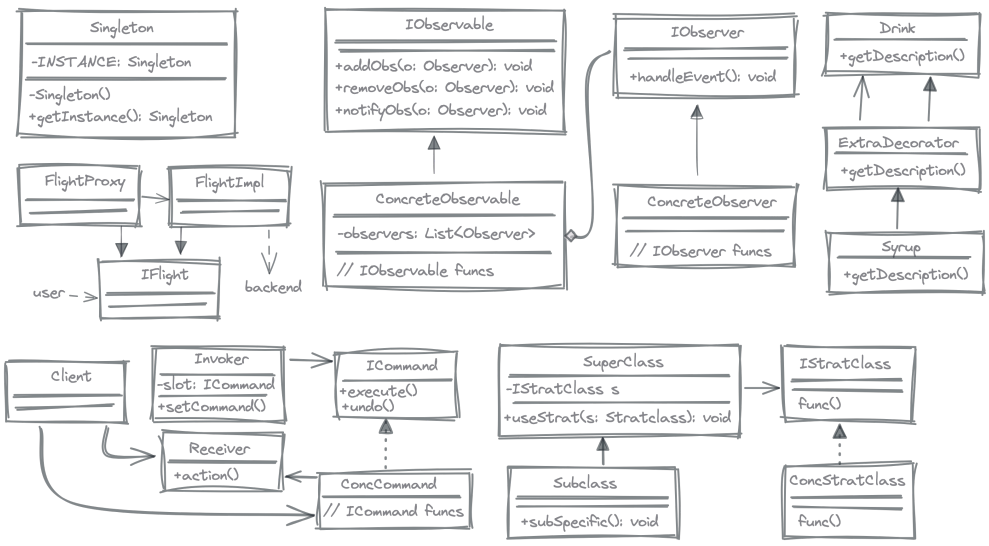
Uses a command object to encapsulate a request. Command object has an execute method that is called by the invoker. Invoker has a setCommand method that sets the command to be executed.

```
interface ICommand { /* execute */ }
class Command implements ICommand { /* execute */ }
class Invoker { /* ex. remote */
  private ICommand command;
  public void setCommand(ICommand c) { command = c; }
  public void executeCommand() { command.execute(); }
}
```

Strategy pattern

Encapsulates behaviors into separate classes. Context has a strategy interface and a method to set the strategy. Context has a method to execute the strategy. This is done to allow for different behaviors to be used at runtime.

```
interface IQuack { quack(); }
class QuackWithSound implements IQuack { /* execute */ }
class Context { /* ex. duck */
  private IStrategy strategy;
  public void setStrategy(IStrategy s) { strategy = s; }
  public void executeStrategy() { strategy.execute(); }
}
```



Rational Unified Process

Inception: Gather requirements, define project goals, identify risks and develop a project plan.

Elaboration: Develop a detailed project plan, identify use cases and develop a domain model.

Construction: Develop the system, run system tests and integrate the system.

Transition: Deploy the system, train users, and provide support.

Vision statement

summary of purpose, reflection of stakeholder expectations, hopeful but realistic, and a guide for decision making.

for [target customer(s)] who [need / opportunity], the [product name] is a [product category] that [key benefit, reason to buy]. Unlike [primary competitive alternative], our product [primary differentiation].w

---

creator principle

The responsibility of creating an object falls to the class that has the necessary information to do so.

ex. A Document class is responsible for creating new Page objects, as it has the necessary information about page layout and formatting.

high cohesion

A class should have a single responsibility and its methods should focus tightly on that responsibility.

ex. A Character class should only handle character-related functions like movement, attacks, and health management, rather than handling game logic or user input.

polymorphism

Objects should respond to the same method call in different ways based on their type.

ex. An Animal class can have a makeSound() method which returns 'woof' for a Dog object but 'meow' for a Cat object.

expert principle

Responsibilities should be assigned to the class that holds the most relevant/complete information.

ex. A product class would be responsible for calculating its own price and discounts since it holds the most relevant info.

low coupling

Minimizing dependencies between classes by designing loosely coupled relationships.

ex. Instead of directly accessing data within another class, use dependency injection or interfaces to avoid tight coupling.

indirection

introduces an intermediary layer between two classes to reduce direct coupling and provide flexibility in communication.

ex. A Controller class can act as an intermediary between a UI and DATA, handling user interactions and updating the view accordingly.

fabrication principle

Creating complex objects should be delegated to a separate factory class or method.

ex. In a web application, a UserFactory class can be responsible for creating User objects with appropriate access levels and roles based on user credentials.

protected variations principle

When a class needs to be extended to accommodate variations, it should be done by creating subclasses rather than modifying the original class.

ex: A base Shape class can be extended to create specific shapes like Circle or Square allowing for variations without altering the base class.

controller principle

A separate controller class should be responsible for handling user interactions, managing application flow, and coordinating between other objects.

ex. In a web application, a UserController can handle user registration, login, and profile management, separating these tasks from the presentation layer (views) and data access layer (models).

Feature	Unified Process (UP)	Waterfall Model	Agile Methodologies
Development approach	Iterative and incremental	Sequential	Iterative and incremental
Phases	Inception, Elaboration, Construction, Transition	Requirements, Design, Implementation, Testing, Deployment	No defined phases, but with well-defined practices (e.g., Scrum, Kanban)
Deliverables	Working software at each phase	Detailed documentation at each phase	Working software increments
Change management	Extensible; changes can be accommodated at each phase	Rigid; changes are difficult to implement after requirements are finalized	Embraces change; changes are expected and welcomed
Customer involvement	High customer involvement	Low customer involvement	High customer involvement
Documentation	Detailed documentation	Minimal documentation	Minimal documentation
Suitability	Complex projects with changing requirements	Well-defined projects with stable requirements	Projects with uncertain requirements or where quick feedback is needed