Design Patterns solve a problem and try to eliminate "code smells", promoting good design and best practices. Most design patterns emphasize polymorphic behavior. If we practice good programming we may be using design patterns and don't even know what pattern we are using! Design patterns can seem repetitive also because since they emphasize sound conventions and principles they revolve around the concepts:

1. Code to the interface and not the implementation.
2. Code by convention; reveal by need.
3. DRY principle – Do not repeat yourself.
4. Use abstractions and not concretions – Client does not need how behavior is implemented.
5. YAGNI – You Aien’t Going to Need It
6. KISS – Keep It Simple Stupid (kind of harsh)
7. SOLID principles discussed later

1. https://github.com/tmaronson-hcl/sandbox Projects with patterns **are DecoratorExample2** refactor of **DecoratorExample, AdapterExample, JDBCTemplateDemo, Liskov, PatternStarters (factory, decorator), SpringFactory, SpringProxy, SpringSingleton, SpringTemplate, StackSimulation** eliminating a code smell.

2. **zz-demo-JDBCTemplate project( like JDBCTemplateDemo**) - show repository Java files (refactored to eliminate deprecated warnings)

- Good use of Template pattern where JDBCTemplate avoids boilerplate code of JDBC (load driver, get connection with url and properties, write SQL queries, create PreparedStatement, get ResultSet or similar, exception handling, close connections.) All these methods and behaviors are handled in proper order.

Some Patterns used in Java API

- Bridge Pattern with JDBC driver classes - database agnostic with DriverManager, DataSource

- Iterator pattern with Iterator interface in Java API for Collection

- Observer pattern with Observer, Observable in Java API – see MVVM

- Command pattern was useful with Swing

- Template pattern with JDBCTemplate Spring JDBC – see above

- Factory pattern many places – ex.: Spring Framework, NumberFormat, DateFormat, etc.

- Decorator – Java IO

- MVC – Spring MVC

MVC vs. MVVM

<https://www.guru99.com/mvc-vs-mvvm.html>

MVC – Model View Controller pattern

MVVM – Observer pattern (Model View -View-Model) When application has updated its state, notifies subjects (observables) Observer update(Observable, Object) See Java API. Ex: Change profile on Linkedin (Observable object hasChanged) and all contacts (or Observers -notifyObservers) are contacted about update (Observer update).

* MVC framework is an architectural pattern that separates an applications into three main logical components Model, View, and Controller. MVVM facilitates a separation of development of the graphical user interface with the help of mark-up language or GUI code
* In **MVC, controller** is the entry point to the Application, while in **MVVM, the view** is the entry point to the Application.
* MVC Model component can be tested separately from the user, while MVVM is easy for separate unit testing, and code is **event-driven**.
* MVC architecture has "**one to many**" relationships between **Controller & View** while in MVVC architecture, "**one to many**" relationships between **View & View Model**.

3. Solid principles - PHP code link but examples are good <https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>

Java code link - many including <https://www.baeldung.com/solid-principles>. Liskov project in repo to show and discuss Liskov principle

Single Responsibility - class has one responsibility.

Open/Closed - Classes open for extension and closed for modification.

Liskov Substitution - Subtypes can be substituted wherever there is a supertype and not cause programs errors.

Interface Segregation - implementing classes are only interested in methods/behaviors of importance to them.

Dependency Inversion - decoupling, both high and low-level modules depend on abstractions (interfaces) rather than concrete classes.

4. Get Head First Design Patterns book, register for free, download code at

1. <https://wickedlysmart.com/head-first-design-patterns/>
2. <https://learning.oreilly.com/library/view/head-first-java/0596009208/>
3. <https://www.oodesign.com/>

Java API code in earlier versions that seemed to violate SOLID principles

1. Swing API - good for event handling but many interfaces like WindowListener had many methods (7 or so) and many did not have to be used in all cases (stubs) but had to be implemented due to rules of interfaces. Then for every xxxListener type interface Sun created a corresponding xxxAdapter class so the developer could add just methods of interest (but of course can only extend one class so nothing is perfect). Ex. Interface Segregation

2. Older versions of EJB - home and remote interfaces, Entity beans (data) and Session Beans (business logic), many unused methods that had to be implemented (stubs) due to rules of interfaces. Ex. Interface Segregation

Anti-Patterns [https://sahandsaba.com/nine-anti-patterns-every-programmer-should-be-aware-of-with-examples.html#premature-optimization](https://sahandsaba.com/nine-anti-patterns-every-programmer-should-be-aware-of-with-examples.html%23premature-optimization)

6. Code smells [https://refactoring.guru/refactoring/smells](https://refactoring.guru/refactoring/smells%20) Look at state/strategy pattern(s) for a code smell such as <https://refactoring.guru/replace-type-code-with-state-strategy>