***Decorator*** – Wraps an object to provide new behavior

***State*** – Encapsulates state-based behaviors and uses delegation to switch between behaviors.

***Iterator*** – Provides a way to traverse a collection of objects without exposing its implementation.

***Facade*** –Wraps an object to provide new behavior.

***Strategy*** – Encapsulates interchangeable behaviors and uses delegation to decide which one to use.

***Proxy*** – Wraps an object to control access to it

***Factory Method*** – Subclasses decide which concrete classes to create.

***Adapter*** – Wraps an object and provides a different interface to it.

***Observer*** – Allows object to be notified when state changes

***Template Method*** – Subclasses decide how to implement steps in an algorithm

***Composite*** – Clients treat collections of objects and individual objects uniformly.

***Singleton*** – Ensures one and only one object is created.

***Abstract Factory*** – Allows a client to create families of objects without specifying their concrete classes.

***Command*** – Encapsulates a request as an object.

***Creational Patterns:***

Creational Patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

***Example*** – Abstract Factory, Factory Method, Singleton, Builder, Prototype

***Behavioral Patterns:***

Any pattern that is a Behavioral Pattern is concerned with how classes and objects interact and distribute responsibility.

***Example***: State, Command, Observer, Template Method, Visitor, Mediator, Iterator, Interpreter, Memento, Strategy, Chain of Responsibility

***Structural Patterns:***

Structural Patterns let you compose classes or objects into larger structures.

***Example***: Facade, Decorator, Proxy, Composite, Flyweight, Bridge, Adapter

Patterns classified based on classes and objects,

***Class Patterns*** describes how relationships between classes are defined via inheritance. Relationships in class patterns are established at compile time.

***Example***: Template Method, Factory Method, Adapter, Interpreter

***Object Patterns*** describe relationships between objects and are primarily defined by composition. Relationships in object patterns are typically created at runtime and are more dynamic and flexible.

***Example***: Composite, Visitor, Decorator, Command, Memento, Proxy, Façade, Observer, Strategy, Chain of Responsibility, Bridge, Mediator, Flyweight, Prototype, State, Abstract Factory, Builder, Singleton

Patterns classification based on ***different domain***s,

**Architectural Patterns** are used to create the living, vibrant architecture of buildings, towns and cities. This is where patterns got their start.

***Habitat***: Found in buildings you like to live in, look at and visit.

***Application Patterns*** are patterns for creating system-level architecture. Many multi-tier architectures fall into this category.

***Habitat***: Seen hanging around 3-tier architectures, client-server systems and the web, MVC Pattern

***Domain-Specific Patterns*** are patterns that concern problems in specific domains, like concurrent systems or real-time systems.

***Habitat***: J2EE patterns

***Business Process Patterns*** describe the interaction between businesses, customers and data and can be applied to problems such as how to effectively make and communicate decisions.

***Habitat***: Seen hanging around corporate boardrooms and project management meetings.

***Organizational Patterns*** describe the structures and practices of human organizations. Most efforts to date have focused on organizations that produce and/or support software.

***Habitat***: Like development team, customer support team

***User Interface Design Patterns*** address the problems of how to design interactive software programs.

***Habitat***: Seen in the vicinity of video game designers. GUI builders and producers.

***Anti-Patterns:***

An Anti-Pattern tells you how to go from a problem to a BAD solution. An anti-pattern always looks like a good solution, but then turns out to be a bad solution when it is applied. By documenting anti-patterns, we help others to recognize bad solutions before they implement them. Like patterns, there are many types of anti-patterns including development, OO, organizational, and domain-specific anti-patterns. Elements of an anti-pattern,

* An anti-pattern tells you why a bad solution is attractive.
* An anti-pattern tells you why that solution in the long term is bad
* An anti-pattern suggests other patterns that are applicable which may provide good solutions.

***Bullet Points:***

* Let Design Patterns emerge in your designs; don’t force them in just for the sake of using a pattern.
* Design Patterns are not set in stone; adapt and tweak them to meet your needs.
* Always use the simplest solutions that meets your needs, even if it doesn’t include a pattern.
* Study Design Patterns catalogs to familiarize yourself with patterns and the relationships among them.
* Pattern classifications (or categories) provide groupings for patterns. When they help, use them.
* You need to be committed to be a patterns writer; it takes time and patience, and you have to be willing to do logs of refinement.
* Remember, most patterns you encounter will be adaptations of existing patterns, not new patterns.
* Build your team’s shared vocabulary. This is one of the most powerful benefits of using patterns.
* Like any community, the patterns community has its own lingo. Don’t let that hold you back. Having read this book, you now know most of it.