**Single Responsibility Principle**

A class should only have one reason to change

Separation of concerns – different classes handling different, independent tasks/problems

**Open-Closed Principle**

Classes should be open for extension but closed for modification

Tested classes should not be touched

**Liskov Substitution Principle**

You should be able to substitute a base type for a subtype

* Make methods virtual to force calls on child

**Interface Segregation Principle**

Don’t put too much into an interface; split into separate interfaces

* YAGNI – You Ain’t Going to Need It

**Dependency Inversion Principle**

High-level modules should not depend upon low-level ones; use abstractions

**Builder Design Patterns**

* Some objects are simple and can be created in a single constructor call
* Other objects require a lot of ceremony to create
* Having an object with 10 constructor arguments is not productive
* Instead, opt for piecewise construction
* Builder provides an API for constructing and object step-by-step

When piecewise object construction is complicated, provide an API for doing it succinctly

**Factory Design Pattern**

* Object creation logic becomes to convoluted
* Constructor is not descriptive
  + Name mandated by name of containing type
  + Cannot overload with same sets of arguments with different names
  + Can turn into ‘optional parameter hell’
* Object creation (non-piecewise, unlike Builder) can be outsourced to
  + A separate function (Factory Method)
  + That may exist in a separate class (Factory)
  + Can create hierarchy of factories with Abstract Factory (enterprise)

A component responsible solely for the wholesale (not piecewise) creation of objects

* Two constructor same type of argument (string, string) (string, string) but do completely different things

**Strategy Design Patter**

* Many algorithms can be decomposed into higher-and lower-level parts
* Making tea can be decomposed into
  + The process of making a hot beverage (boil water, pour into cup); and
  + Tea-specific things (put teabag into water)
* The high-level algorithm can then be reused for making coffee or hot chocolate
  + Supported by beverage-specific strategy
* Enables the exact behavior of a system to be selected either at run-time (dynamic) or compile-time (static)

**Events**

- mechanism for communication between objects

- used in building Loosely Coupled Applications

- helps extending applications

**Decorator** - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility

in our designs.

- In our designs we should allow behavior to be extended without the need to modify existing code.

- Composition and delegation can often be used to add new behaviors at runtime.

- The Decorator Pattern provides an alternative to subclassing for extending behavior.

- The Decorator Pattern envolves a set of decorator classes that are used to wrap concrete

components.

- Decorator classes mirror the type of the components they decorate. (In fact, they are the

same type as the componentes they decorate, either through inheritance or interface implementation.)

- Decorators change the behavior of their components by adding new functionality before and/or

after (or even in place of) method calls to the component.

- You can wrap a component with any number of decorators.

- Decorators are typically transparent to the client of the component; that is, unless the client is relying on the component’s concrete type.

- Decorators can result in many small objects in our design, and overuse can be complex.

