# Overview:

* Describes how classes and objects can be combined to form larger structures.
  + Utilizes inheritance to compose interfaces or implementations.
  + Structural object patters describe ways to assemble objects.
  + E.g. complex user interfaces and accounting data.
* This design patterns concern class and object composition.
* The composite design pattern
  + Describes how to build a class hierarchy made up of classes for two kinds of objects.
* The proxy design pattern acts as a convenient surrogate or placeholder for another object.
  + Provide a level of indirection to specific properties of objects.

# Class Patterns vs Object Patterns (sub-category):

* Class patterns describe how relationships between classes are defined:
  + Use inheritance to compose interfaces or implementations.
  + Relationships are established at compile time.
  + Adapter.
* Object patterns describe relationships between objects:
  + Describe ways to compose objects to realize new functionality.
  + Use composition.
  + Relationships are typically created at runtime.
    - More dynamic and flexible.
  + Bridge, composite, decorator, façade, flyweight and proxy patterns
* There are seven structural patterns that we will study:
  + Will highlight their similarities and differences.

# Summary:

# Adapter versus Bridge:

* There are many similarities between the structural patterns, especially in their participants and collaborations.
* **Adapter versus Bridge:**
  + Both promote flexibility by providing a level of indirection to another object.
  + Both involve forwarding requests to this object from an interface other than its own.
* The key difference between these patterns lies in their intents:
  + Adapter focuses on resolving incompatibilities between two existing interfaces.
  + The bridge bridges an abstraction and its (potentially numerous) implementations.
    - Provides a stable interface to clients even as it lets you vary the classes that implement it.
* The Adapter and Bridge are often used at different points in the software lifecycle.
  + An adapter often becomes necessary when you discover that two incompatible classes should work together.
    - To avoid replicating code.
  + The user of a bridge understand up-front that an abstraction must have several implementations, and both may evolve independently.
  + The Adapter pattern makes things work after they are designed and the Bridge makes them work before they are.

# Composite versus Decorator:

* Composite and decorator have similar structure diagrams.
  + Both rely on recursive composition to organize an open-ended number of objects.
* Decorator is designed to let you add responsibilities to objects without subclassing.
  + Avoids the explosion of subclasses that can arise from trying to cover every combination of responsibilities statically.
* Composite has a different intent than the decorator.
  + Focuses on structuring classes so that many related objects can be treated uniformly.
    - Multiple objects can be treated as one.
  + Focus is not on embellishment but on representation.
* The Composite and Decorator patterns are often used together:
  + Both lead to the kind of design in which you can build applications just by plugging objects together without defining any new classes.
  + Consist of an abstract class with some subclasses that are composites, some that are decorators, and some that implement the fundamental building blocks of the system.
    - Both composites and decorators will have a common interface:
      * A composite is a ConcreteComponent.
      * A decorator is a Leaf

# Decorator versus Proxy:

* Both patterns describe how to provide a level of indirection to an object.
* Both implementations keep a reference to another object to which they forward requests.
* Differ by their intent.
* The proxy pattern composes an object and provides an identical interface to clients.
  + Not concerned with attaching or detaching properties dynamically.
  + Not designed for recursive composition.
  + Its intent is to provide stand-in for a subject when it is inconvenient or undesirable to access the subject directly:
    - It lives on a remote machine, has restricted access, or is persistent.
* The decorator pattern is when the component provides only part of the functionality, and one or more decorators furnish the rest:
  + Addresses the situation where an object’s total functionality cannot be determined at compile time.
  + Recursive composition an essential part.
    - Not the case in Proxy:
      * Focuses on one relationship between the proxy and its subject and that relationship can be expressed statically.
* Differences do not mean that these patterns cannot be combined:
  + Might envision a proxy-decorator that adds functionality to a proxy, or a decorator-proxy that embellishes a remote object.