A. Creational

a. Simple Factory
    i. Simple Factories refer to one large conditional to decide which kind of object to make and when. It acts as an intermediary that can choose to call a constructor given a set of conditions are met. This provides increased separation between the code where the parameters are held and the code where the objects created are used.

b. Factory Method
    i. A Factory is a pattern that creates objects by referring to a small number of interfaces instead of only using a constructor to do so. Used to "increase the separation between the producers and the consumers of the objects constructed from certain designated classes." Factory Methods use method calls which call constructors to create new objects.

c. Abstract Factory
    i. An Abstract Factory is a factory that uses abstract classes in order to call constructors to create new objects. These patterns can produce families of closely related objects.

d. Prototype
    i. Prototypes are used to create objects based on an existing object, used in the same way a "prototype" would be used in real life. There is less room for error when creating objects this way instead of from the ground up because if the consumer doesn't specify certain parameters, etc. the object being used as the prototype will be able to fill in those blanks.

e. Builder
    i. A builder pattern is used to determine whether an object should be built or not, and if it should, whether to build it in its entirety or piece by piece. These are normally used to create complex objects. The builder uses conditionals to determine if certain object creation constraints are met, and will construct some or all of the object based on the specified conditions.

f. Singleton
    i. Singletons are used to limit the production of certain objects to make sure that too many aren't created. Generally this pattern is used to ensure that there is only one instance of a certain object in use at a given time, and that if further objects are created, it will return the already-created object.

B. Structural

a. Object Pool
    i. Object pools are used to keep a set of initialized objects on deck, ready to use, instead of destroying the objects after they are created. The main reason to use this pattern would be for object types that have a very high cost to make, ones that don't get used a lot, or both.

b. Adapter
    i. Adapters are used to "adapt" code from one method/object and use it for another without rewriting the original method/constructor. It allows objects with incompatible interfaces to interact.

c. Bridge
    i. Bridges are in charge of connecting the entities that specify concepts to the entities that implement these concepts. Bridges ensure that the implementation code hierarchy is not permanently bound to that of the concept abstraction code.

d. Composite
    i. Composite patterns are used as a "container" for 2 or more objects. It can be used to contain 2 or more subclasses that are children of a superclass, while itself inheriting all of the features and methods of the superclass, creating an easy and modular way of dealing with object types. (See "seat" example from book.)

e. Decorator
    i. Decorators are used to provide additional embellishment code to supplement the basic functionality. These are used to give the user the freedom to customize code and mix and match options for the code without hurting the underlying integrity of the original code.

f. Facade
    i. Facade patterns are used "to create different usage views of a system of classes for different categories of users." An example of a facade would be to have different modes on a program that correspond with different levels of experience.

g. Flyweight
    i. Flyweights are used to store "core forms" of an object and to take one of those forms, clone it, and add any user-specified embellishment on top of the copied core form. A reason to use flyweights is that it reduces the likelihood of an error occurring from making a new object from scratch, as the facade just keeps a copy of all the core forms of the objects that are needed. This is especially important in cases where the core object has a lot of

parameters/attributes that would have to be transferred from one object to another.

C. Behavioral

a. Chain of Responsibility
    i. Used to "create dynamic chains in software systems that are based on the query escalation metaphor." It should be used for when you don't know which objects will resolve a query at compile time because it allows for the query to get escalated until a good match is found.

b. Command
    i. Where "the actions are primary and the actors are secondary." Calls methods in other classes in order to synthesize a certain behavior. Synthesized behavior is composed of smaller behaviors from other classes. The objects that are being used to synthesize the larger behavior are called receivers. The object orchestrating the synthesis of the larger behavior is called the invoker.

c. Interpreter
    i. Used to examine input and parse it, normally into an Abstract Syntax Tree. Calls for the clients of an interpreter to be able to access all of the interpretation functionality through a common interface. "Interpreters are widely used in compilers that by design must translate a program, written obviously with a set of constraints on the syntax, into a structure suitable for conversion into executable machine code."

d. Iterator
    i. Used as an interface to access objects in a collection without the user being aware of where the objects are stored in memory.

e. Mediator
    i. An object that controls and brings order to the interactions of certain objects in a software system. Sometimes necessary to limit dependencies between classes and prevent methods from directly being called on other objects. Acts as an intermediary.

f. Memento
    i. Serves as a "recall mechanism for software objects in which the changes and their associated parameters are stored as 'mementos' in such a way that, while a user cannot peer inside them directly, the "mementos" are nevertheless available for restoring the objects to their previous states." Allows the user to save different states of an object/software system without violating encapsulation. Used to roll back the software, or as a save system.

g. Observer

    i. Used to disseminate information to other objects in a “subscription-based broadcasting” fashion. Observers take information that is needed by other objects and broadcasts it so that it is only available to objects that register with the Observer class. This is useful for situations where computational power must be maximized because the Observer only broadcasts the information to objects that are registered with it.

h. State
    i. Used to “efficiently represent context-based dependencies of behaviors in object-oriented software.” Each state becomes a different class, and these states present themselves to the rest of the code through a common interface. The state pattern allows global context to alter the functionality/behavior of a software system. A way to remember this is that it is like a save state in a video game, where all of the information and parameters for the context of a certain moment in the game are saved to a file, but in this case it gets stored in an object.

i. Strategy
    i. A pattern that addresses how to organize the code when the problem is complex and requires a multi-step solution. It is especially useful for when multiple different solutions exist for an individual step in the solution. The main difference between strategy and state is that with the state method, the states can transition from one to another. With the strategy pattern, the steps to solve a problem are supposed to run from beginning to end and then be complete. The Strategy pattern is more goal focused and the State pattern is more process focused.

j. Template Method
    i. Used to customize the behavior of a method by storing its invariable part in a superclass and its variables in subclasses extended from the superclass. Normally implemented when you want to use an algorithm for certain different things because you are able to leave the things that are not changed based on the case in a superclass and make subclasses to deal with changes associated with implementing the algorithm in the code. This allows algorithms to be performed on collections of different kinds of objects because each type of object will have a different subclass of the root template method class that specifies how to run the algorithm on that given object type.

k. Visitor
    i. A Visitor is used “to allow additional functionality to be associated with a class hierarchy without actually modifying the hierarchy in any significant way.” Provides a way for external code to add functionality to your

original code. Uses accept() hooks in order to add functionality to the code.

l. Null Object
    i. Used to encapsulate the absence of information. Useful when you want a placeholder for part of your code to do nothing. We used a Null Object as an UnsupportedOperationException in Homework 2 when working with LispLists so that we could create an empty exception to handle in the case of an error.
m. Proxy
    i. Provides an object that can be used as a substitute for a different, real service object used by the client. They are primarily used for security reasons so that the proxies can handle requests rather than the original objects themselves. An example of a real life proxy is the ATM, as it is only an interface to perform certain actions to your bank account, instead of controlling the bank account directly.
n. Model-View Controller (MVC)
    i. This is a compound pattern that separates a program into 3 parts: the model(s), the view(s), and the controller(s). This pattern is useful because it helps keep separation between the client and the code, and allows for easier “plug and play” and changing of the code without affecting the user as much.