Software Engineering

Lesson #12 - Practice

Lesson #12 - Practice

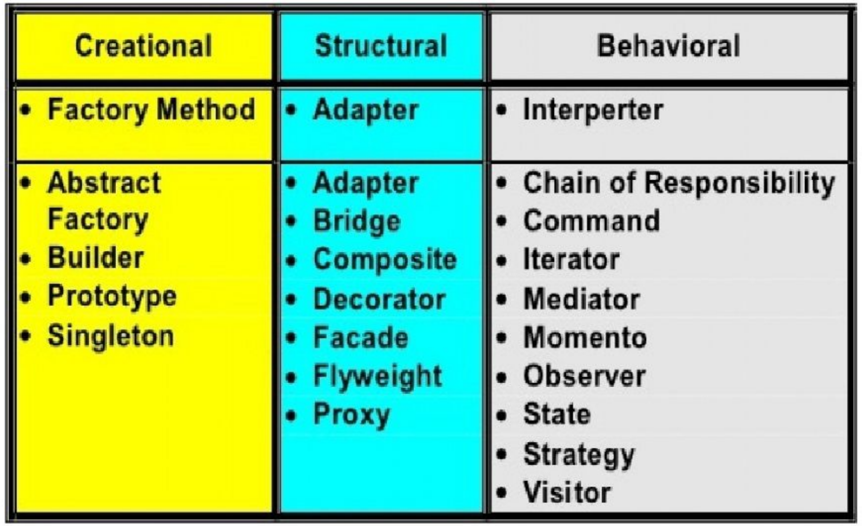
Agenda: Lesson #12 - Software Engineering - Practice

1	Creational Patterns
2	Class Work
3	Q & A

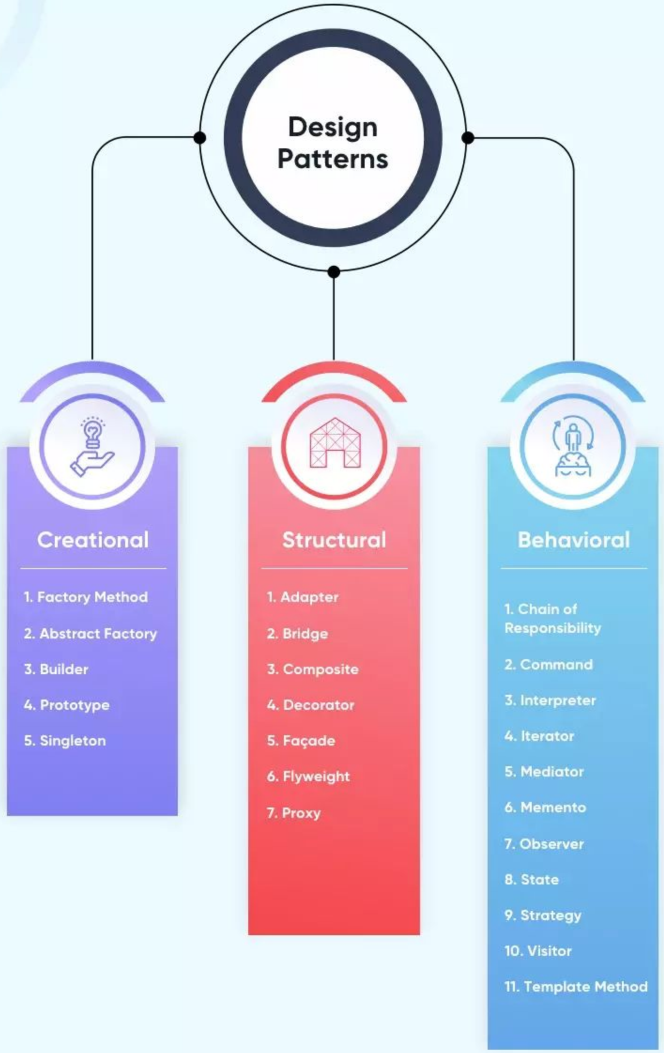
Agenda: Lesson #12 - Software Engineering - Practice

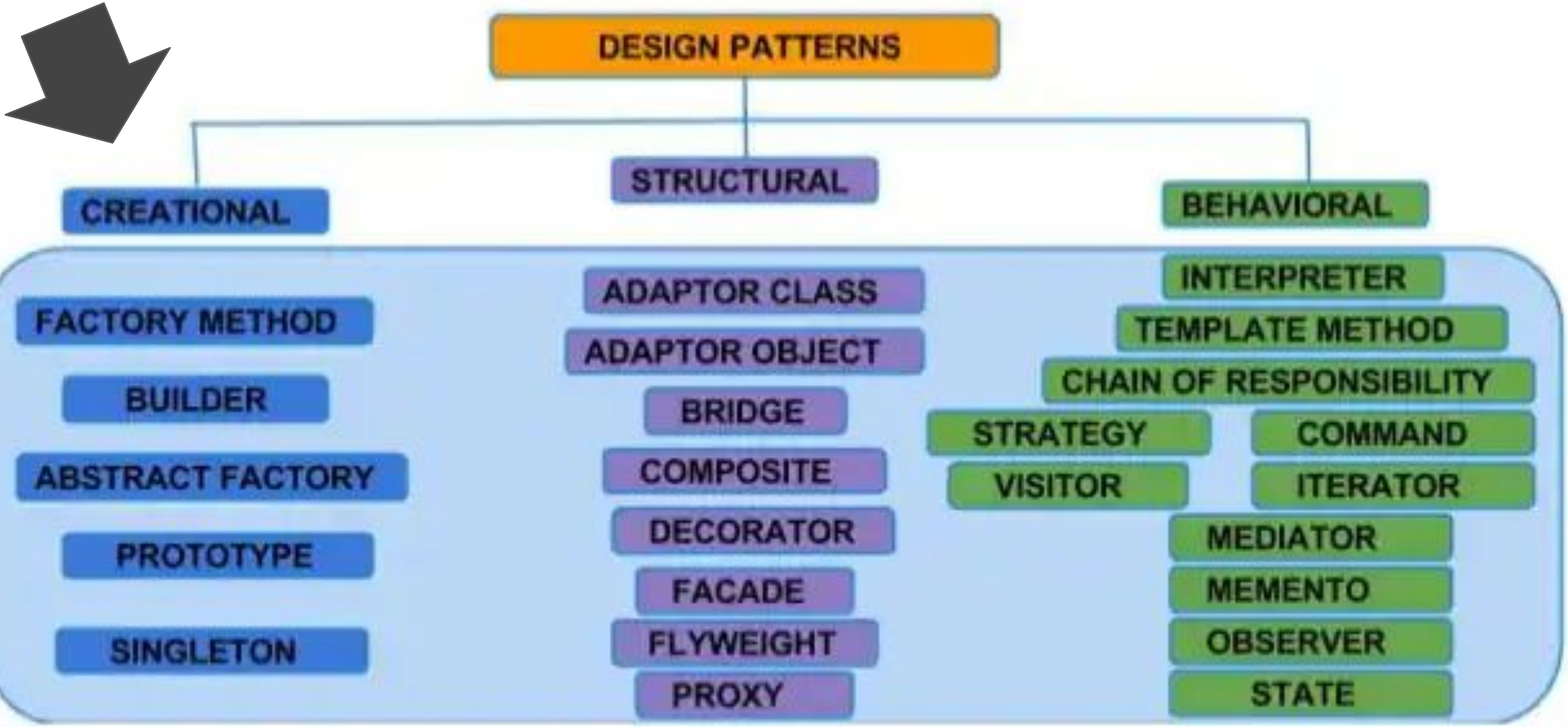
1	**Creational Patterns**
2	Class Work
3	Q & A

Design Patterns



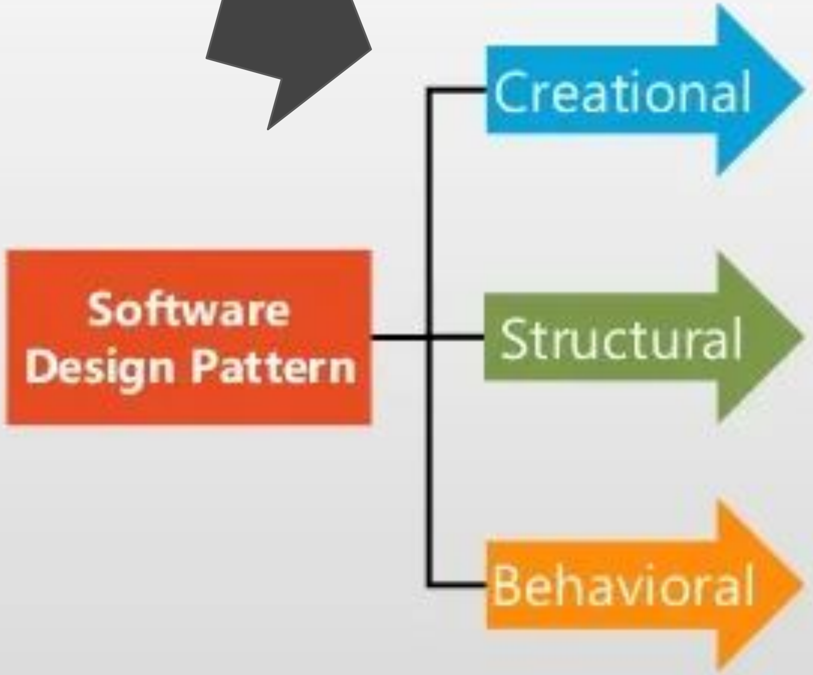
Creational	Structural	Behavioral
• Factory Method	• Adapter	• Interperter
• Abstract Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Momento • Observer • State • Strategy • Visitor




DESIGN PATTERNS
CREATIONAL
STRUCTURAL
BEHAVIORAL
FACTORY METHOD
BUILDER
ABSTRACT FACTORY
PROTOTYPE
SINGLETON
ADAPTOR CLASS
ADAPTOR OBJECT
BRIDGE
COMPOSITE
DECORATOR
FACADE
FLYWEIGHT
PROXY
INTERPRETER
TEMPLATE METHOD
CHAIN OF RESPONSIBILITY
STRATEGY
COMMAND
VISITOR
ITERATOR
MEDIATOR
MEMENTO
OBSERVER
STATE

Design Patterns

Types Of Design Patterns



- Used to construct objects such that they can be decoupled from their implementing system.
- Used to form large object structures between many disparate objects.
- Used to manage algorithms, relationships, and responsibilities between objects

Creational Patterns

Creational design patterns abstract the instantiation process

They help make a system independent of how its objects are created, composed, and represented

A class creational pattern uses inheritance to vary the class that instantiated, whereas an object creational pattern will delegate instantiation to another object

Creational Patterns

Creational patterns become important as systems evolve to depend more on object composition than class inheritance

Creational Patterns

As that happens, emphasis shifts away from hard-coding a fixed set of behaviors toward defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones

Thus creating objects with particular behaviors requires more than simply instantiating a class

Creational Patterns

There are two recurring themes in these patterns

First, they all encapsulate knowledge about which concrete classes the system uses

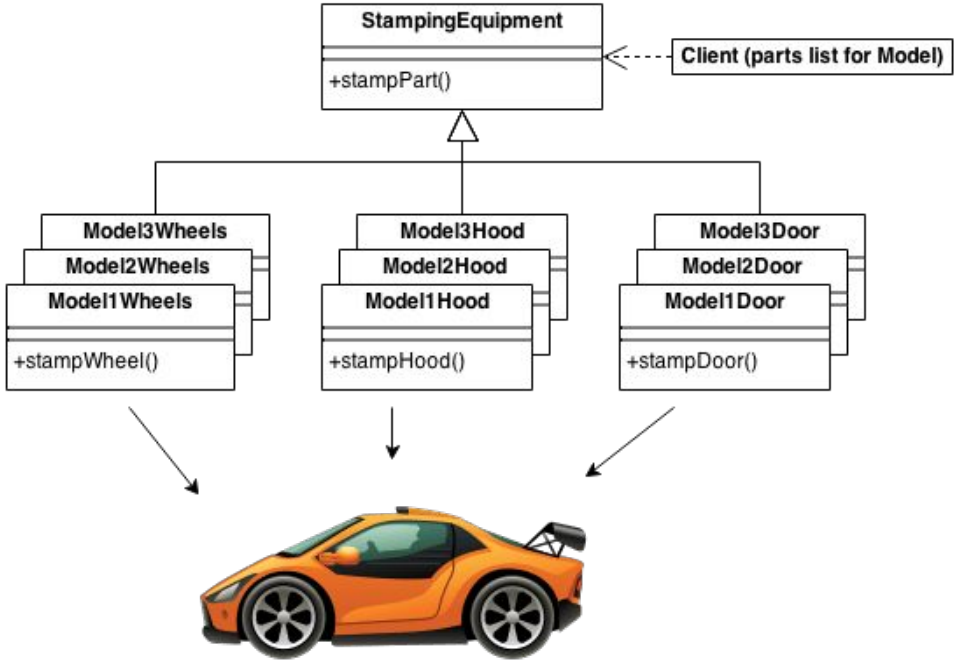
Second, they hide how instances of these classes are created and put together

All the system at large knows about the objects is their interfaces as defined by abstract classes

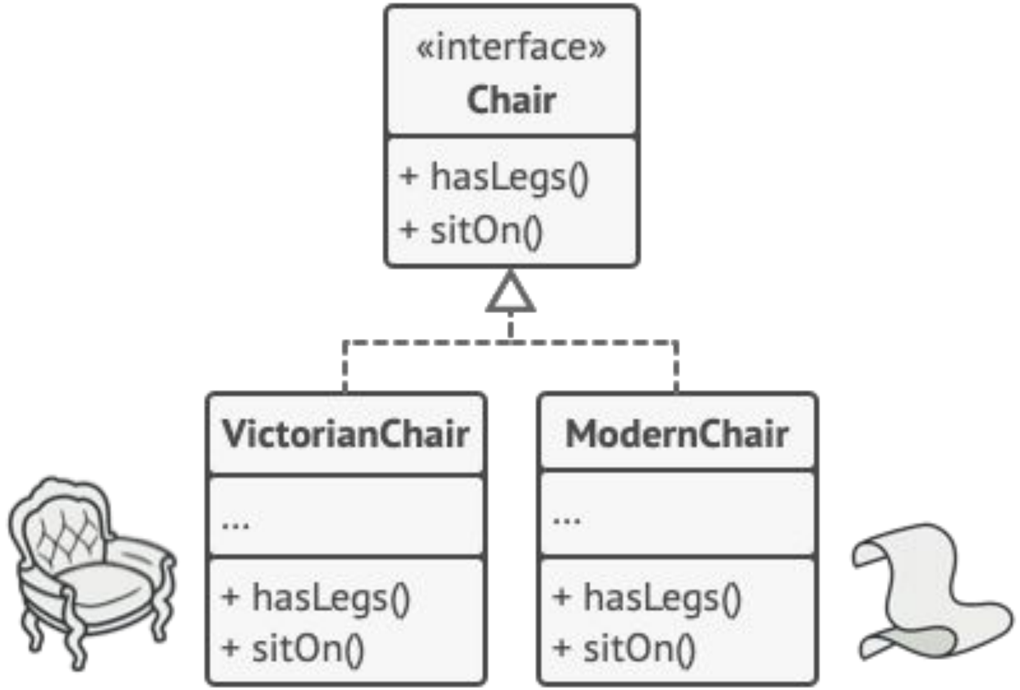
Abstract Factory Pattern

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Abstract Factory Pattern



Abstract Factory Pattern



Abstract Factory Pattern

Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager.

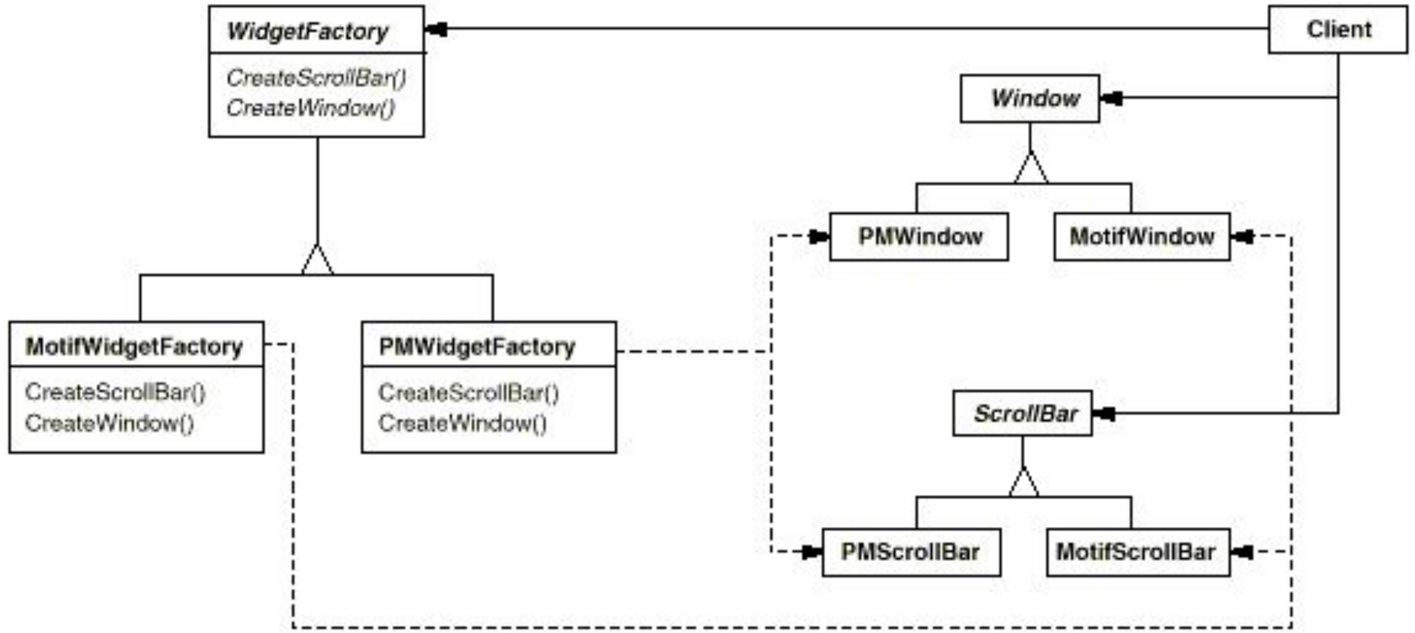
Different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons.

Abstract Factory Pattern

To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel.

Instantiating look-and-feel-specific classes of widgets throughout the application makes it hard to change the look and feel later.

Abstract Factory Pattern



Abstract Factory: Applicability

Use the Abstract Factory pattern when

- a system should be independent of how its products are created, composed, and represented.

- a system should be configured with one of multiple families of products.

Abstract Factory: Applicability

Use the Abstract Factory pattern when

- a family of related product objects is designed to be used together, and you need to enforce this constraint.

- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Creational Patterns

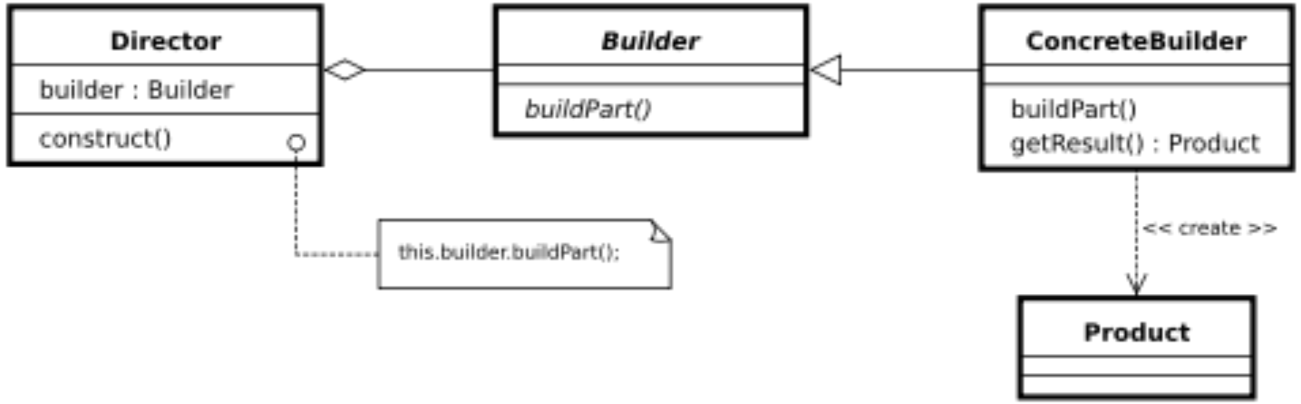
Builder Pattern

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

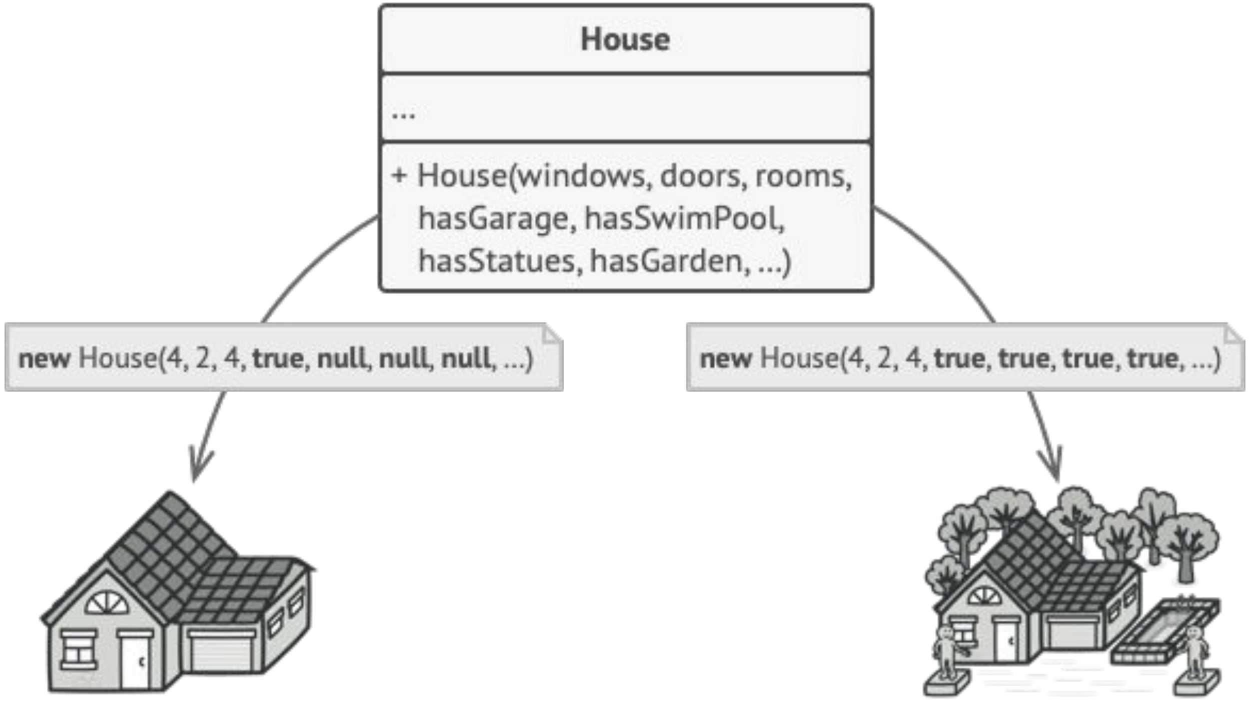
Builder Pattern

A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats. The reader might convert RTF documents into plain ASCII text or into a text widget that can be edited interactively. The problem, however, is that the number of possible conversions is open-ended. So it should be easy to add a new conversion without modifying the reader.

Builder Pattern



Builder Pattern


House
...
+ House(windows, doors, rooms, hasGarage, hasSwimPool, hasStatues, hasGarden, ...)
new House(4, 2, 4, true, null, null, null, ...)
new House(4, 2, 4, true, true, true, true, ...)

Builder: Applicability

Use the Builder pattern when

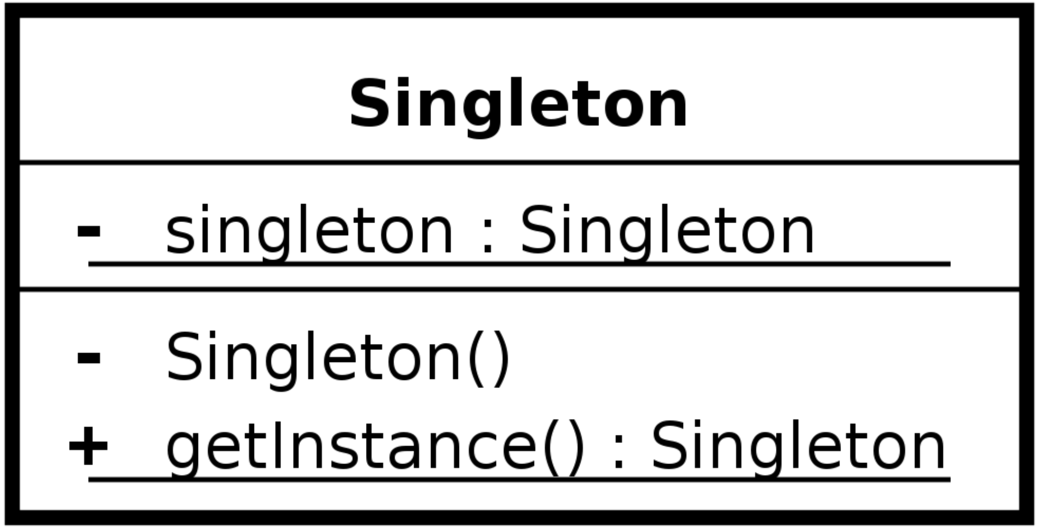
- the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.

- the construction process must allow different representations for the object that's constructed.

Singleton Pattern

Ensure a class only has one instance, and provide a global point of access to it

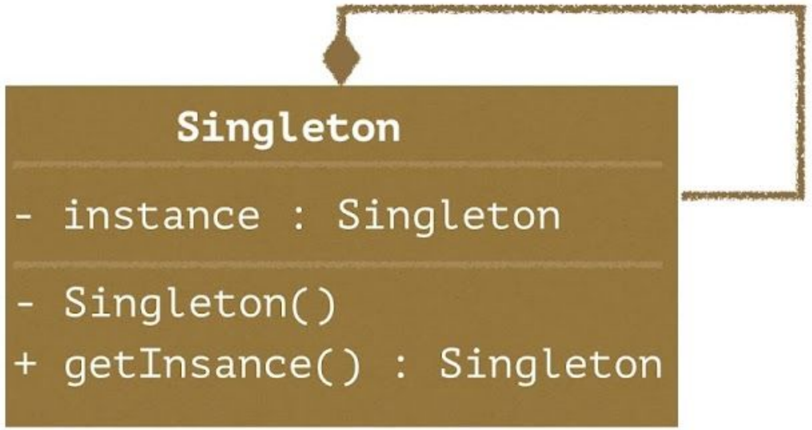
Singleton Pattern



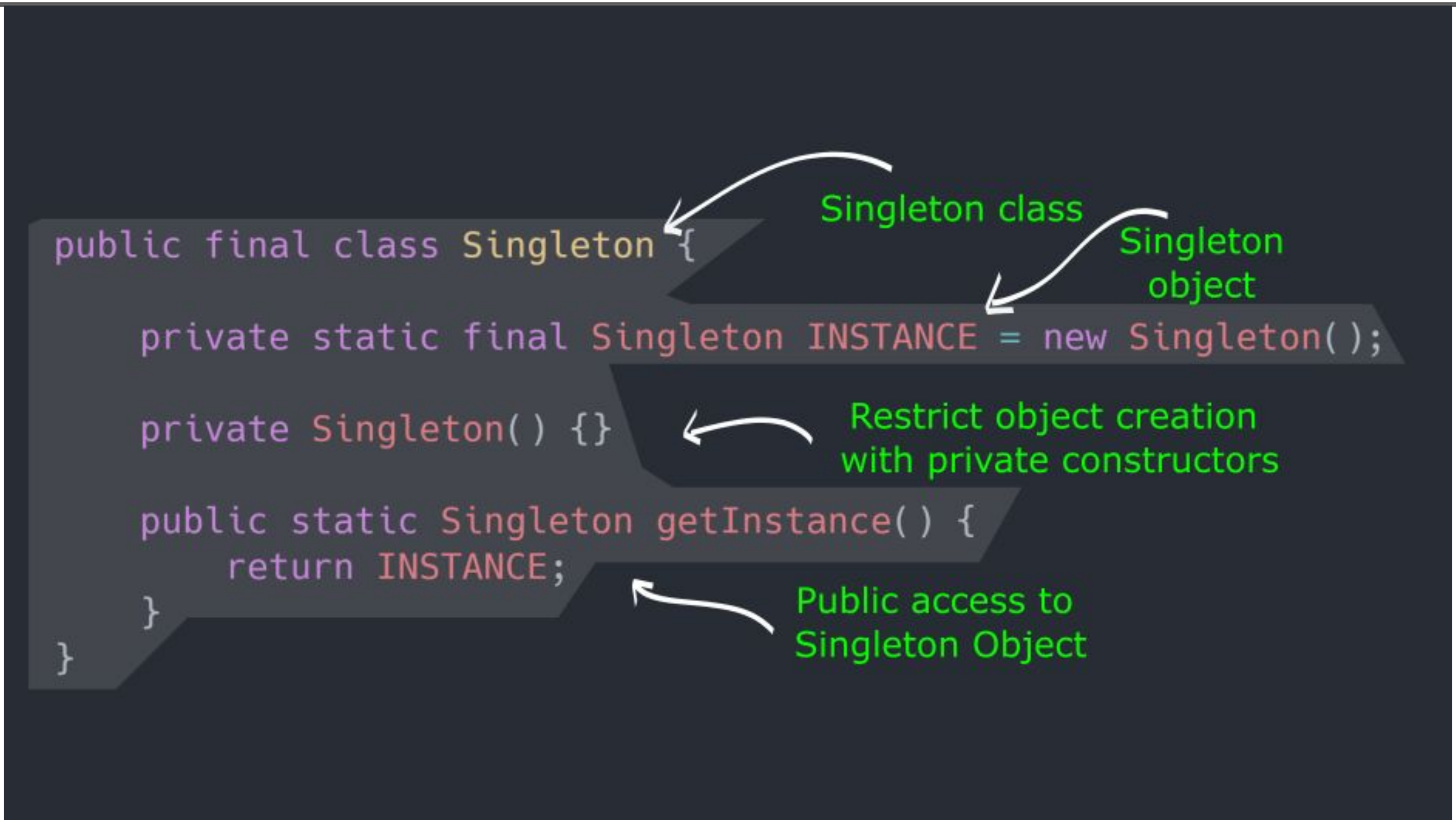
Creational Patterns

Singleton Design Pattern

"Ensure that a class has only one instance and provide a global point of access to it."



Creational Patterns



```
public final class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Singleton class
Singleton object
Restrict object creation with private constructors
Public access to Singleton Object

Singleton Pattern

It's important for some classes to have exactly one instance

Although there can be many printers in a system, there should be only one printer spooler

There should be only one file system and one window manager. A digital filter will have one A/D converter

An accounting system will be dedicated to serving one company

Singleton Pattern

How do we ensure that a class has only one instance and that the instance is easily accessible?

A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects

Singleton Pattern

A better solution is to make the class itself responsible for keeping track of its sole instance

The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance

This is the Singleton pattern

Singleton Pattern: Applicability

Use the Singleton pattern when

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

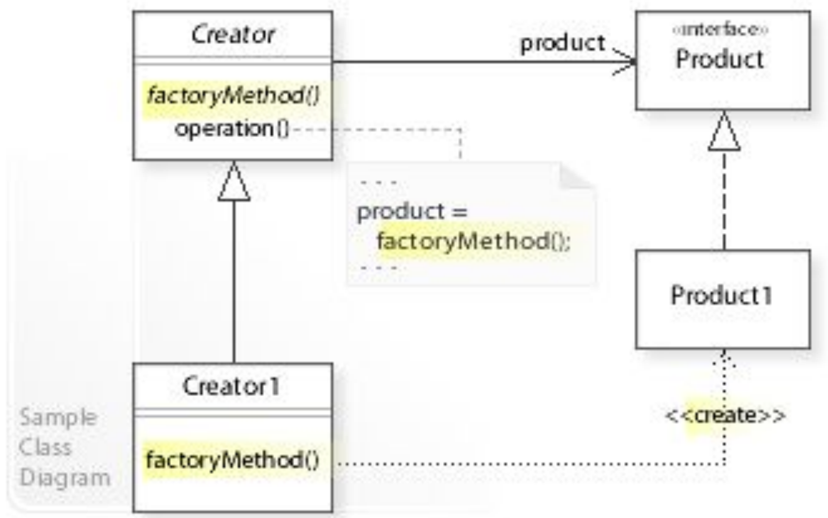
Factory Method Pattern

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

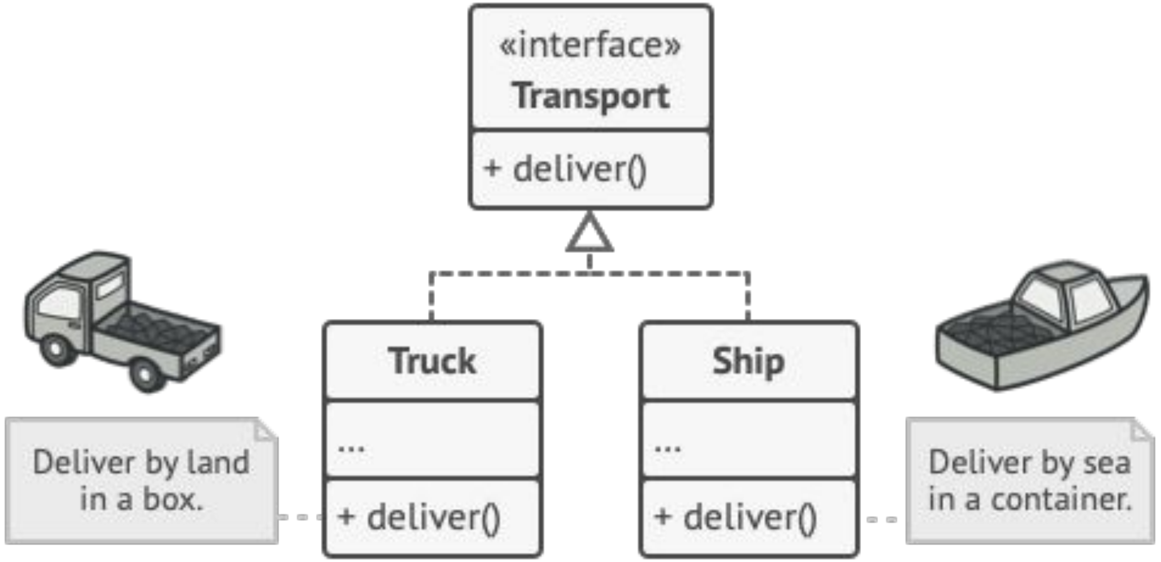
Factory Method Pattern

Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well.

Factory Method Pattern



Factory Method Pattern



Factory Method: Applicability

Use the Factory Method pattern when

- a class can't anticipate the class of objects it must create.
- a class wants its subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Prototype Pattern

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Prototype Pattern

You could build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves.

The editor framework may have a palette of tools for adding these music objects to the score.

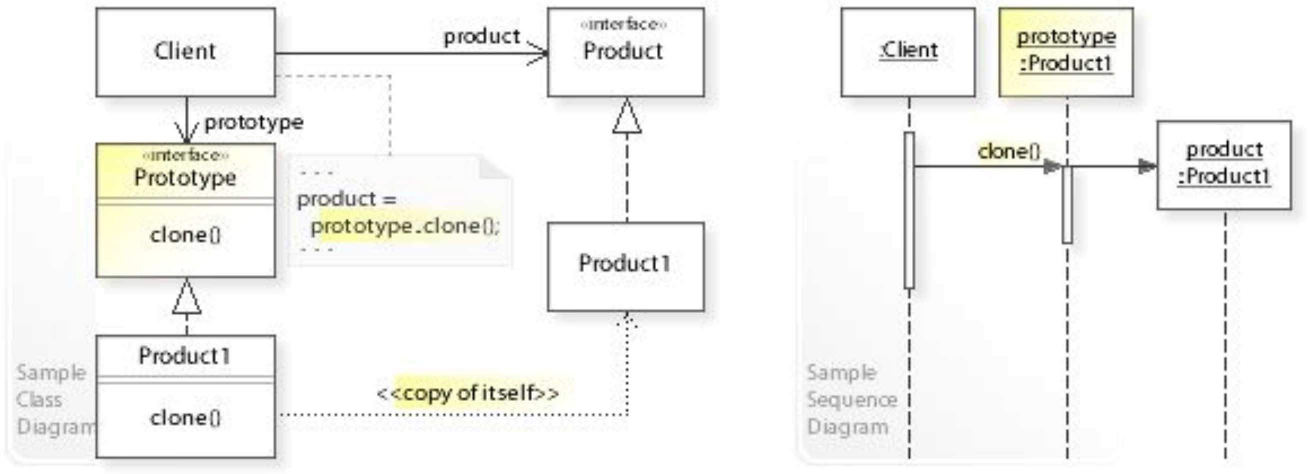
Prototype Pattern

The palette would also include tools for selecting, moving, and otherwise manipulating music objects.

Users will click on the quarter-note tool and use it to add quarter notes to the score.

Or they can use the move tool to move a note up or down on the staff, thereby changing its pitch.

Prototype Pattern



```
...
product =
  prototype.clone();
...
```

Prototype Pattern: Applicability

Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and

- when the classes to instantiate are specified at run-time, for example, by dynamic loading; or

Prototype Pattern: Applicability

Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and

- to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or

Prototype Pattern: Applicability

Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and

- when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Agenda: Lesson #12 - Software Engineering - Practice

1. Creational Patterns
2. **Class Work**
3. Q & A

Class work

Software Engineering, 10. Global Edition

No classwork for today

Agenda: Lesson #12 - Software Engineering - Practice

Agenda: Lesson #12 - Software Engineering - Practice

Q & A