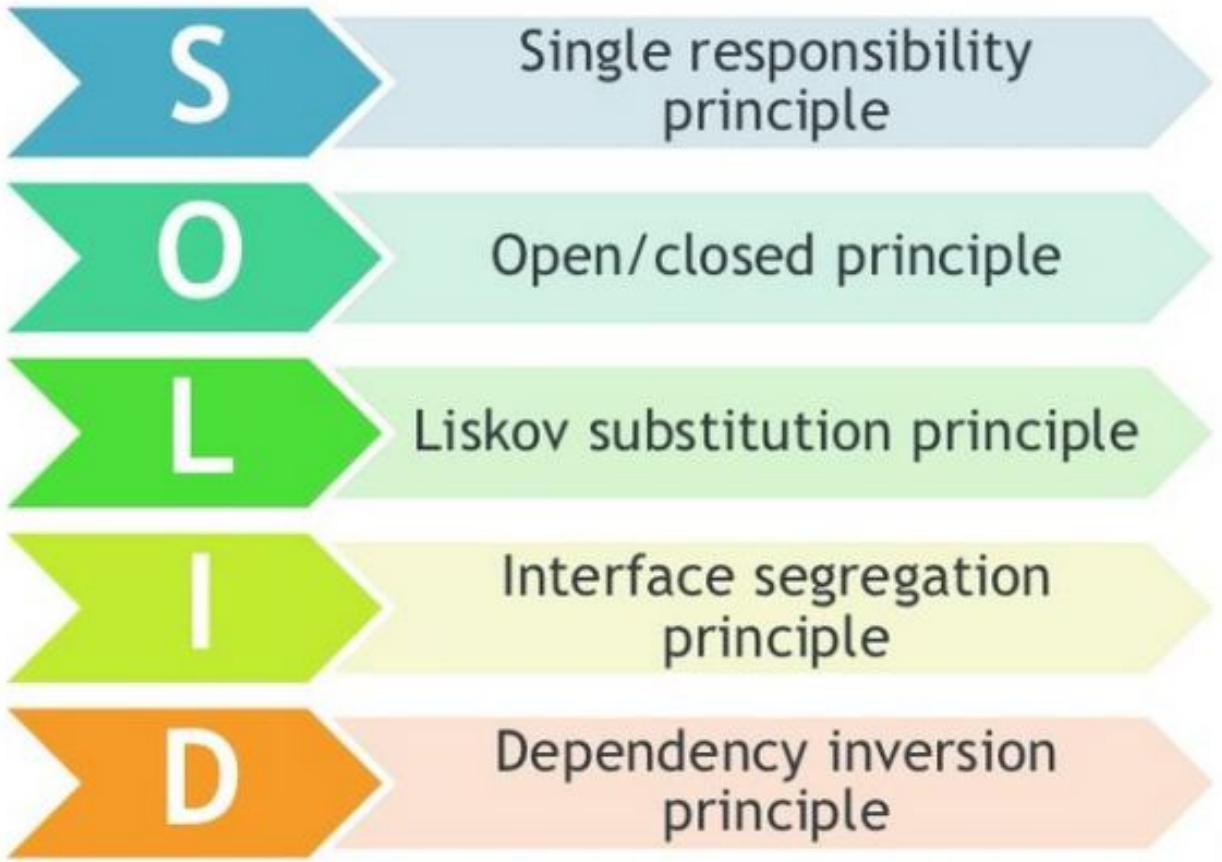
ECE444: Software Engineering

Design Patterns 3

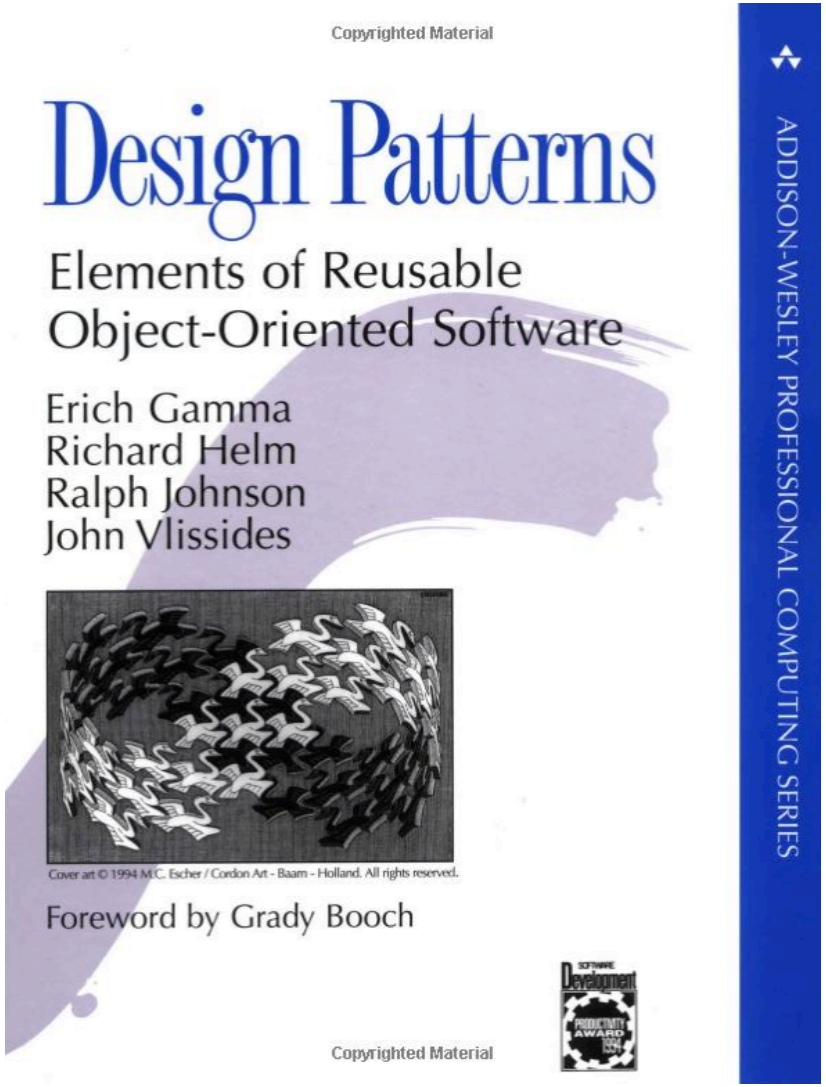
Shurui Zhou

The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

OO Design Principles

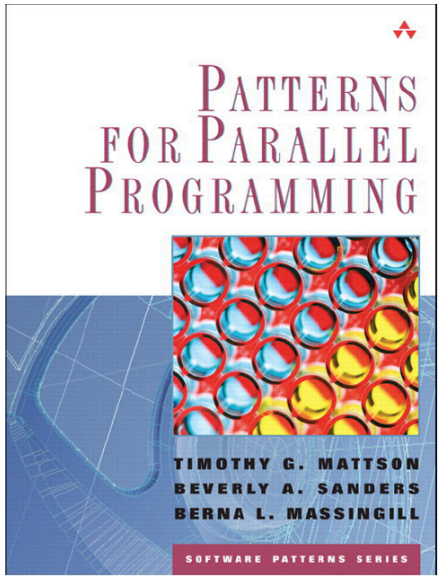


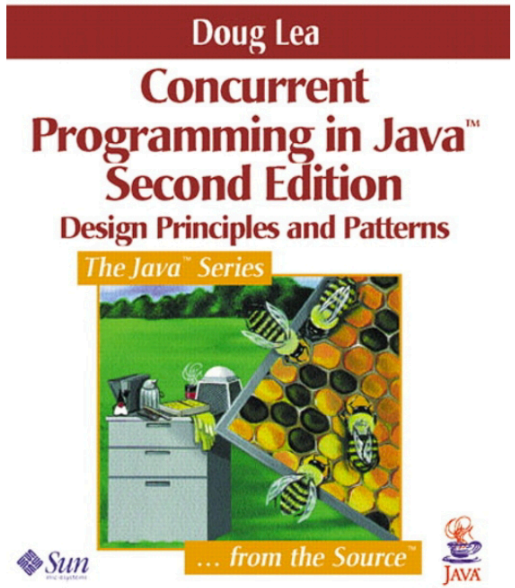
Building stable and flexible systems

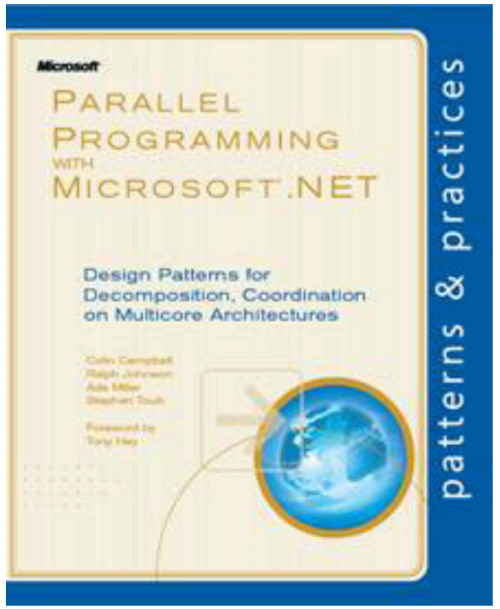


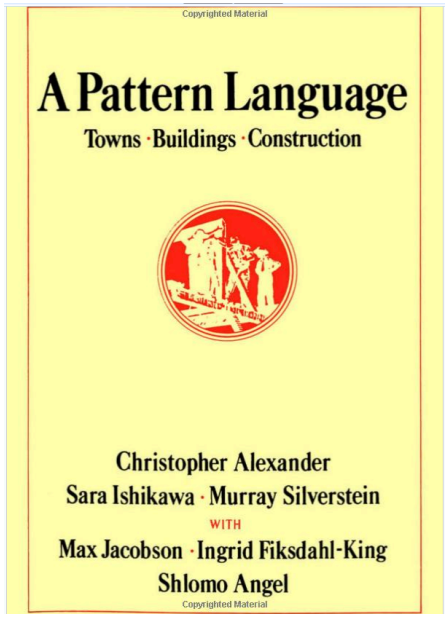
- the GoF book
- Elements of Reusable Object-Oriented Software
- 23 OO patterns

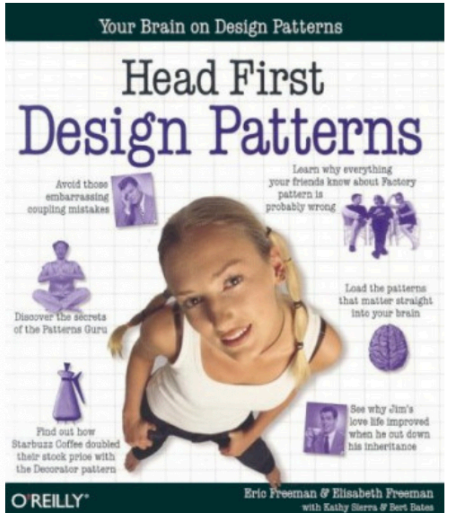
Lots of books on patterns

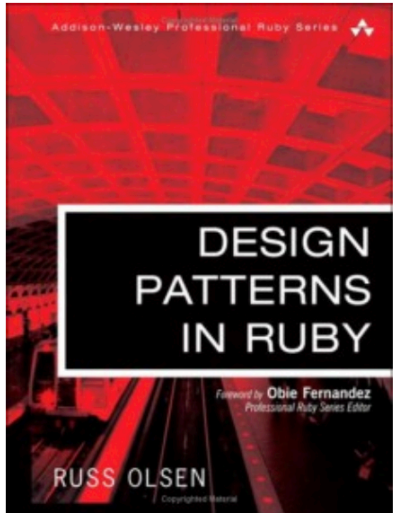


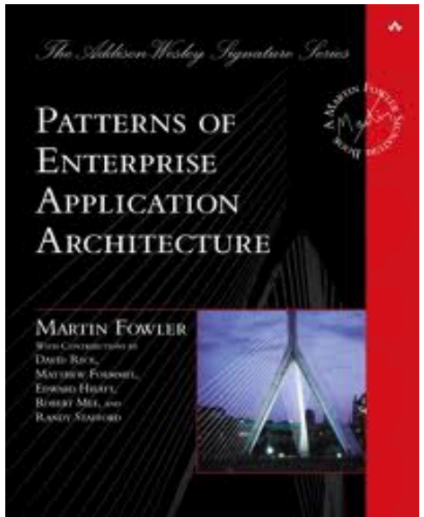


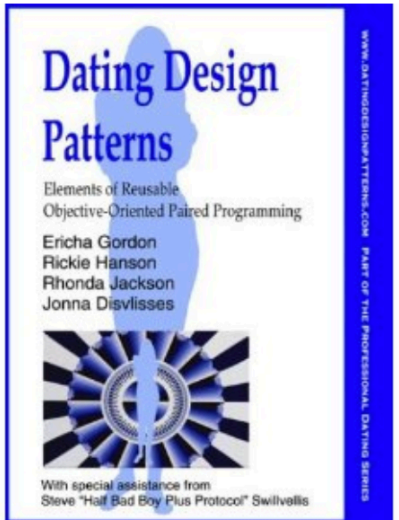












Design Patterns

- Design Patterns – expert solutions to recurring problems in a certain domain
- Description usually involves problem definition, driving forces, solution, benefits, difficulties, related patterns.
- Pattern Language - a collection of patterns, guiding the users through the decision process in building a system
- Patterns are related (high level-low level)

What does the pattern consist of?

- **Intent** of the pattern briefly describes both the problem and the solution.
- **Motivation** further explains the problem and the solution the pattern makes possible.
- **Structure** of classes shows each part of the pattern and how they are related.
- **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

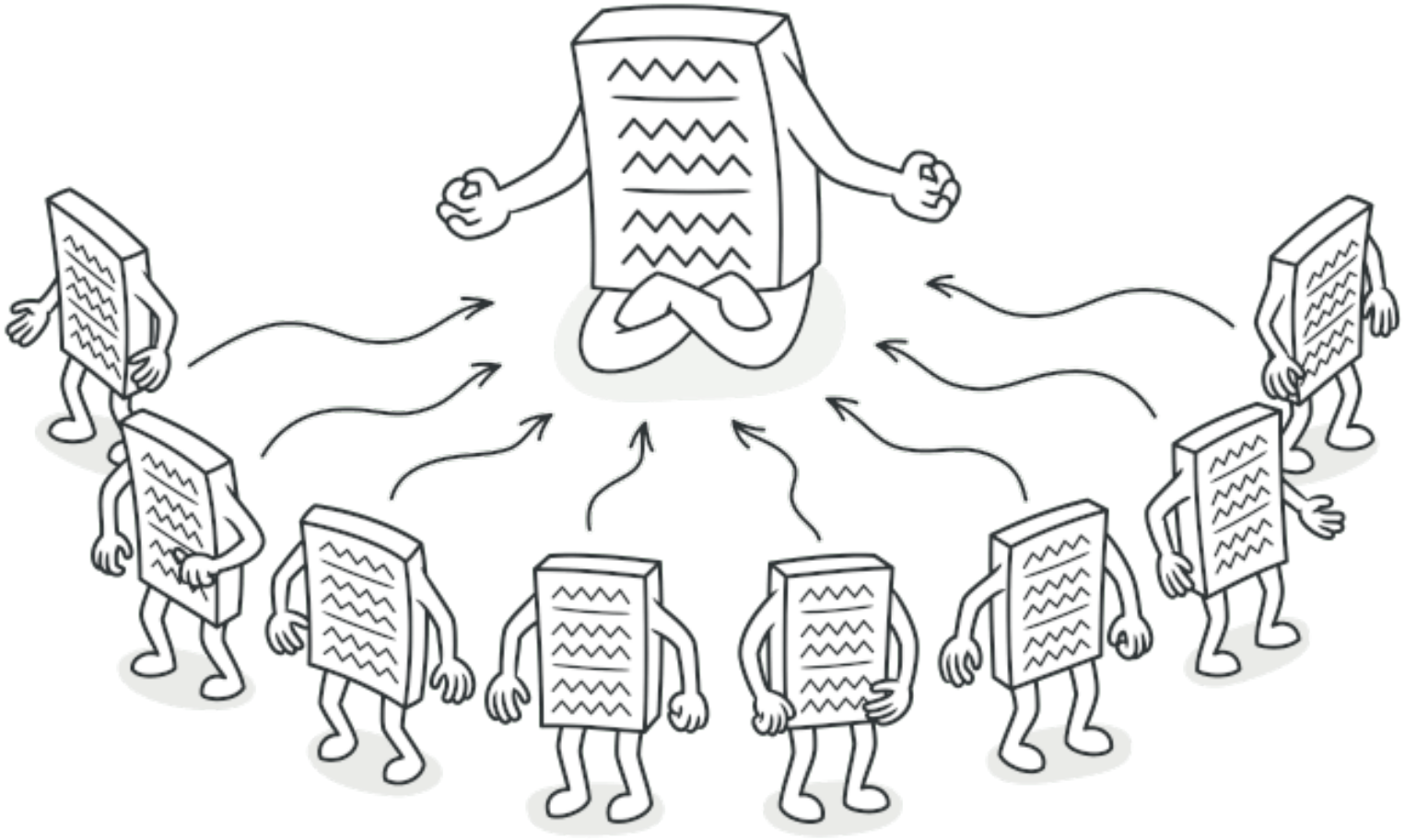
Classification of patterns

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

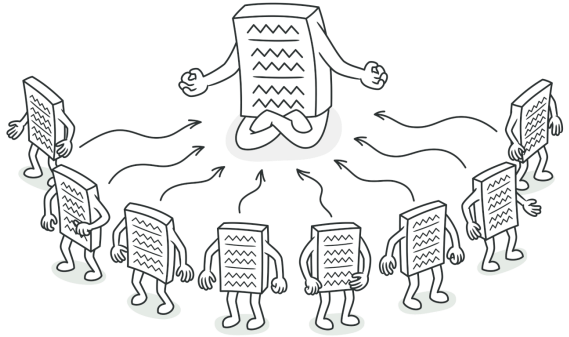
Classification of patterns

- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
- **Behavioral patterns**
  - **Strategy**

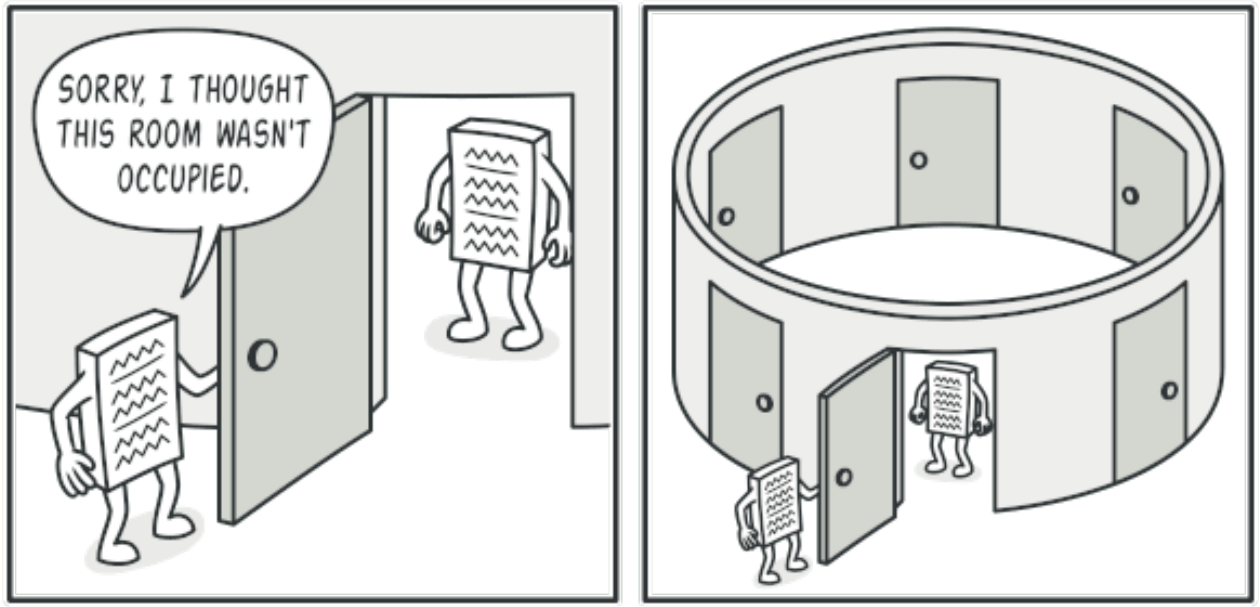
Singleton



Singleton

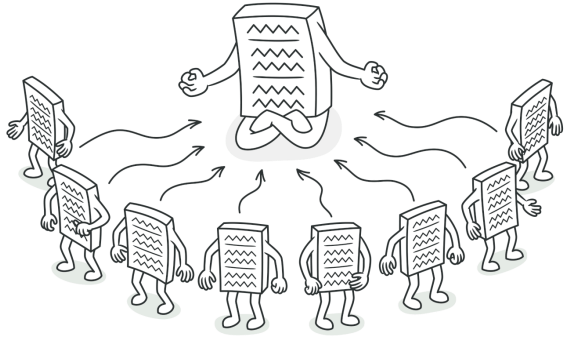


- Intent:
  - Ensure that a class has just a single instance
  - Provide a global access point to that instance



Clients may not even realize that they're working with the same object all the time.

Singleton

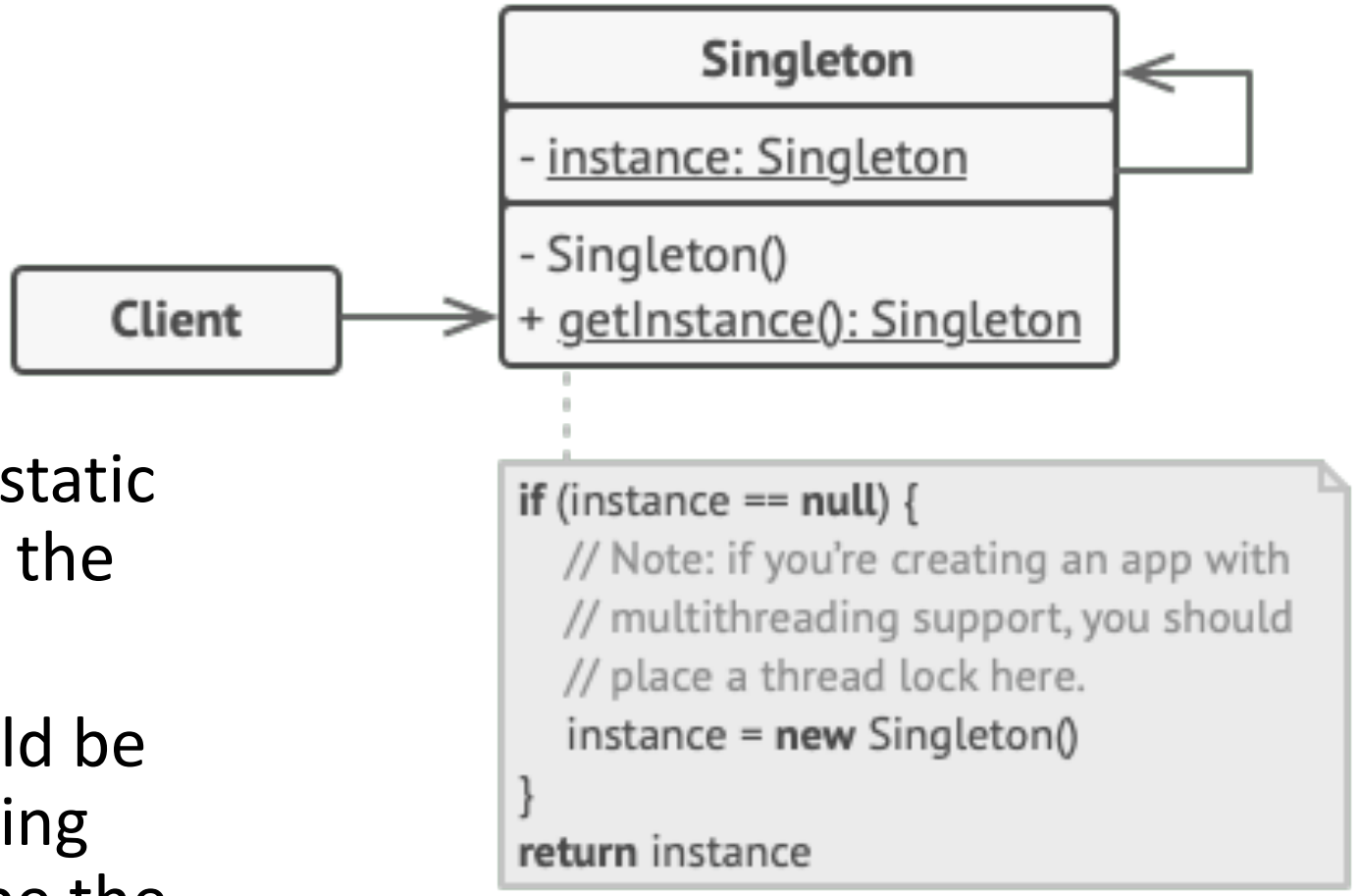


- a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.
- Example:
  - cache
  - thread pools
  - registries

Singleton

- How?
  - Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
  - Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

Singleton



- The **Singleton** class declares the static method getInstance that returns the same instance of its own class.
- The Singleton's constructor should be hidden from the client code. Calling the getInstance method should be the only way of getting the Singleton object.

Singleton Implementation (Python)

```python
class SingletonMeta(type):
    """
    The Singleton class can be implemented in different ways in Python. Some
    possible methods include: base class, decorator, metaclass. We will use the
    metaclass because it is best suited for this purpose.
    """

    _instances = {}

    def __call__(cls, *args, **kwargs):
        """
        Possible changes to the value of the `__init__` argument do not affect
        the returned instance.
        """
        if cls not in cls._instances:
            instance = super().__call__(*args, **kwargs)
            cls._instances[cls] = instance
        return cls._instances[cls]
```

```python
class Singleton(metaclass=SingletonMeta):
    def some_business_logic(self):
        """
        Finally, any singleton should define some business logic, which can be
        executed on its instance.
        """
```

```python
if __name__ == "__main__":
    # The client code.

    s1 = Singleton()
    s2 = Singleton()

    if id(s1) == id(s2):
        print("Singleton works, both variables contain the same instance.")
    else:
        print("Singleton failed, variables contain different instances.")
```

Singleton - Example

- java.lang.Runtime

Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the *getRuntime* method.

- **java.awt.Desktop#getDesktop()**
- **java.lang.System#getSecurityManager()**

Singleton - Applicability

- Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.
- Use the Singleton pattern when you need stricter control over global variables.

Singleton: Pros and Cons

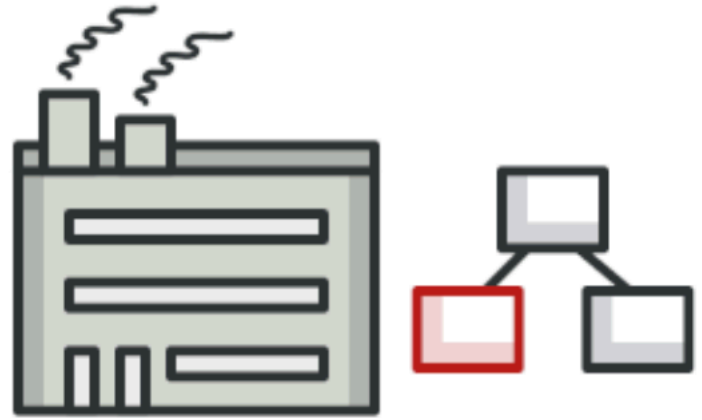
- ✓ You can be sure that a class has only a single instance.
- ✓ You gain a global access point to that instance.
- ✓ The singleton object is initialized only when it's requested for the first time.

- ✗ Violates the *Single Responsibility Principle*. The pattern solves two problems at the time.
- ✗ The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
- ✗ The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
- ✗ It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern.

Classification of patterns

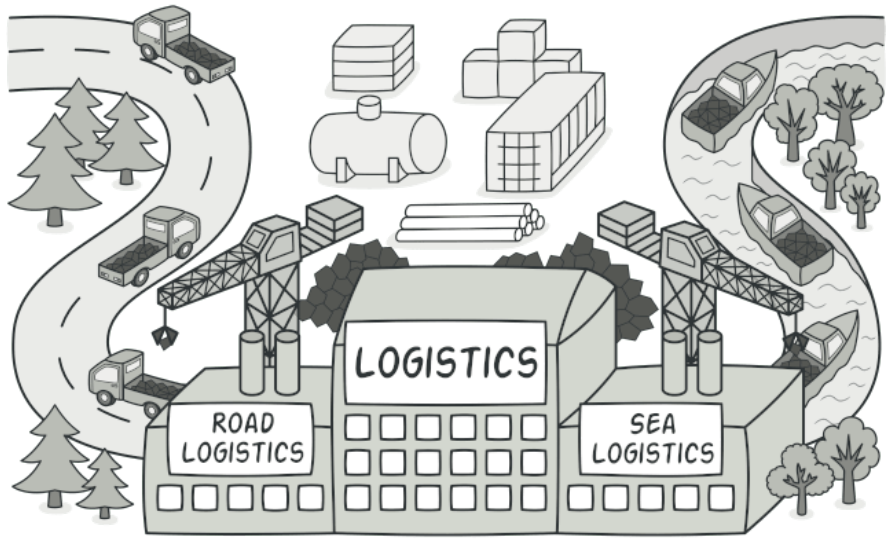
- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
- **Behavioral patterns**
  - **Strategy**

Factory Method



Factory Method

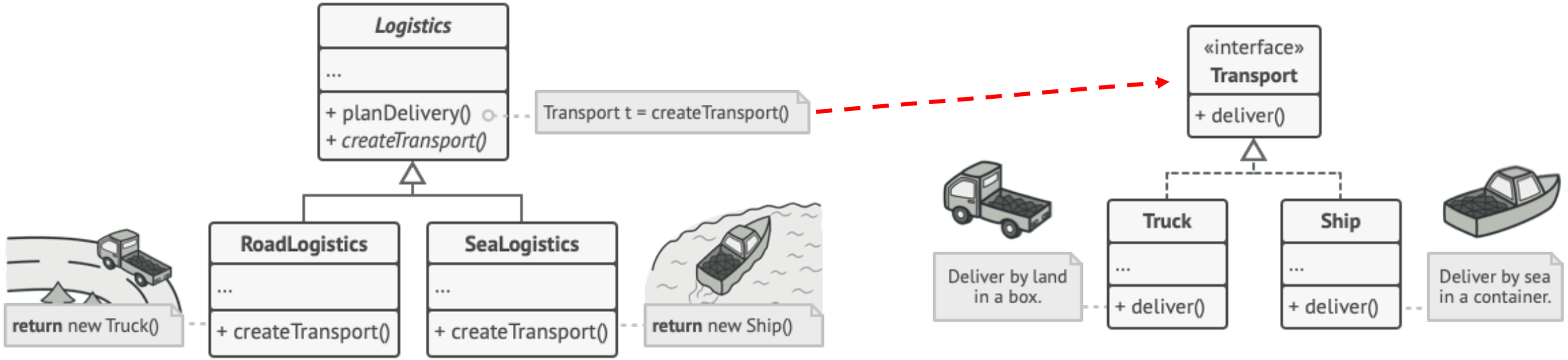
- **Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



Factory Method

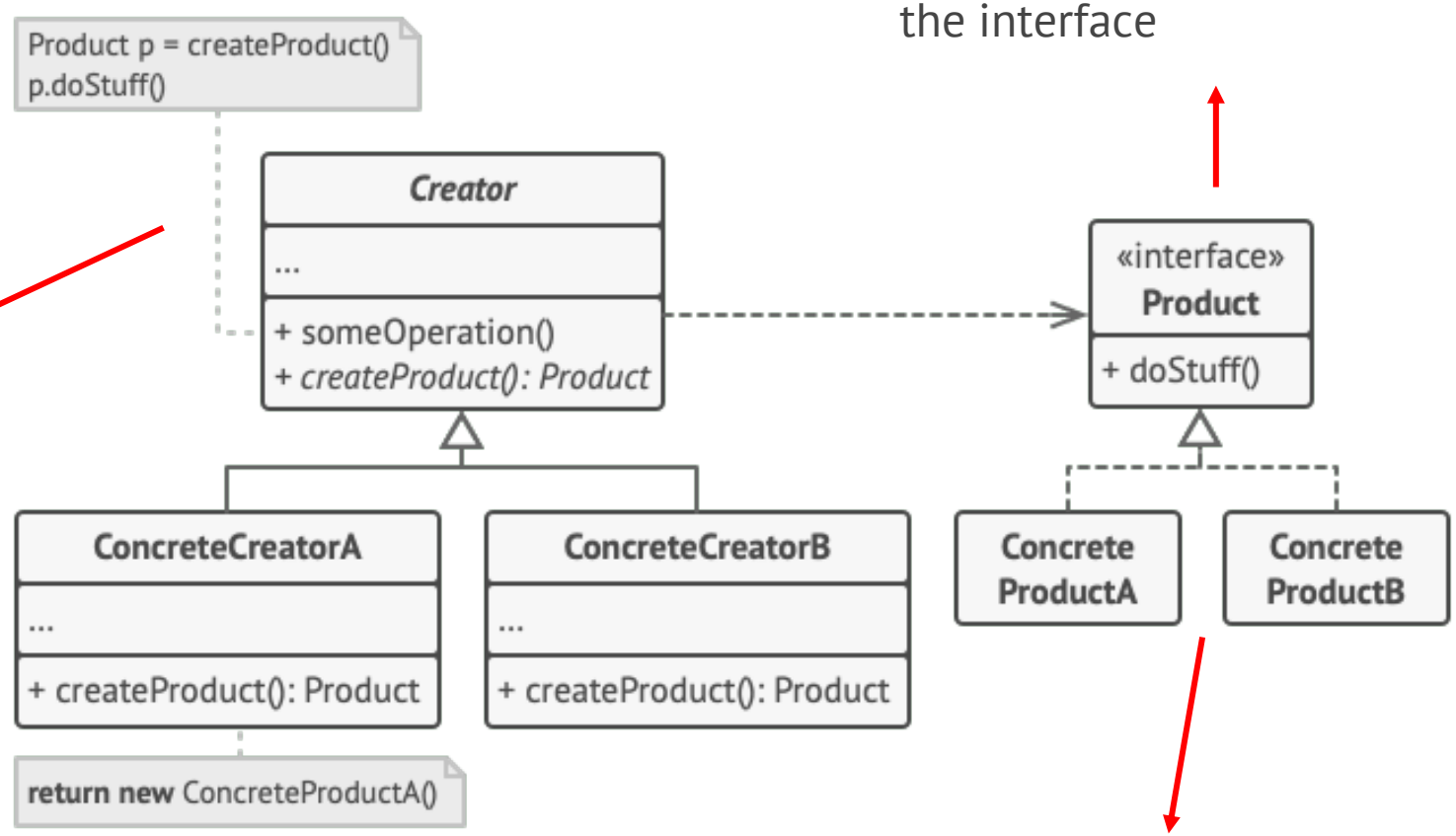
Creator

Products



Factory Method

The **Product** declares the interface

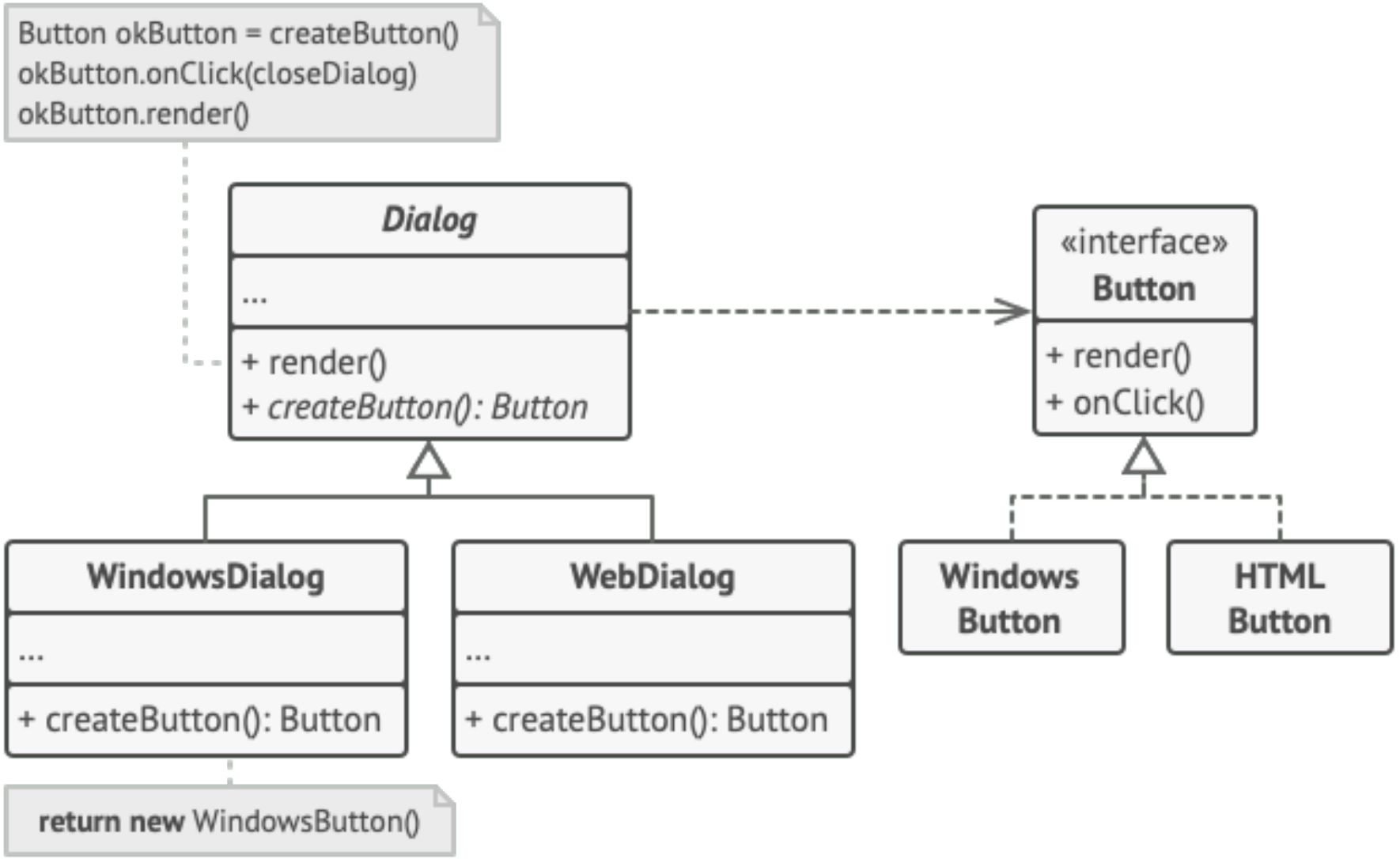


The **Creator** class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.

Concrete Creators override the base factory method so it returns a different type of product.
Note that the factory method doesn't have to **create** new instances all the time. It can also return existing objects from a cache, an object pool, or another source.

Concrete Products are different implementations of the product interface.

Factory Method - Example



Factory Method - Applicability

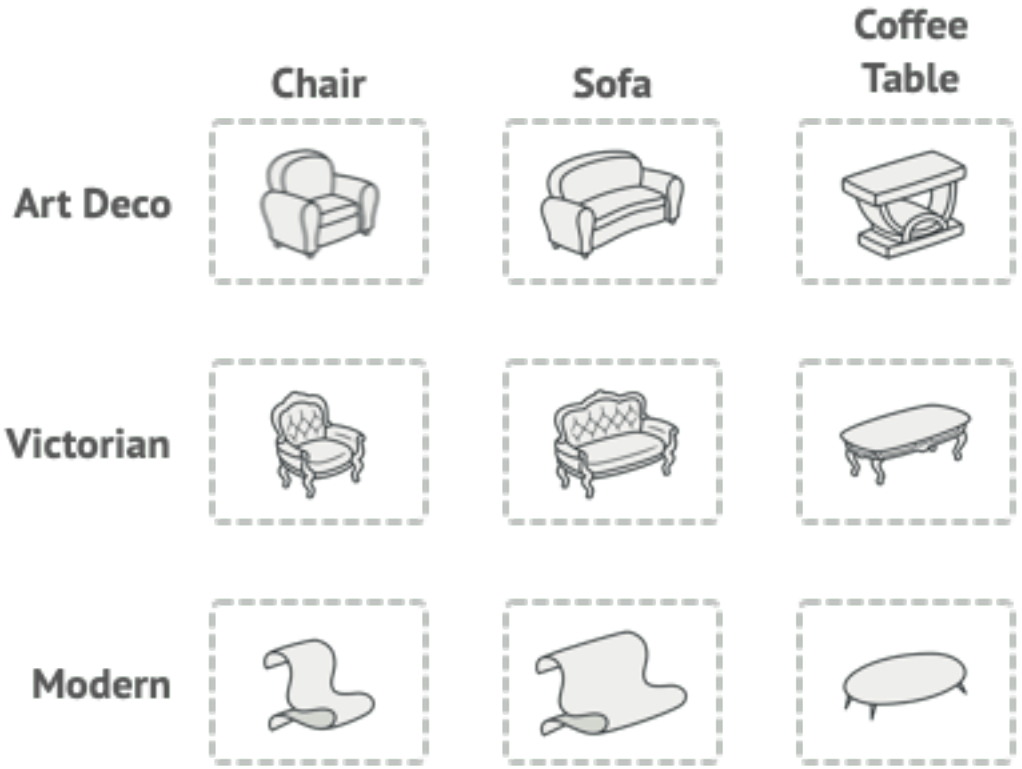
- when you don’t know beforehand the exact types and dependencies of the objects your code should work with.
- when you want to provide users of your library or framework with a way to extend its internal components.
- when you want to save system resources by reusing existing objects instead of rebuilding them each time.

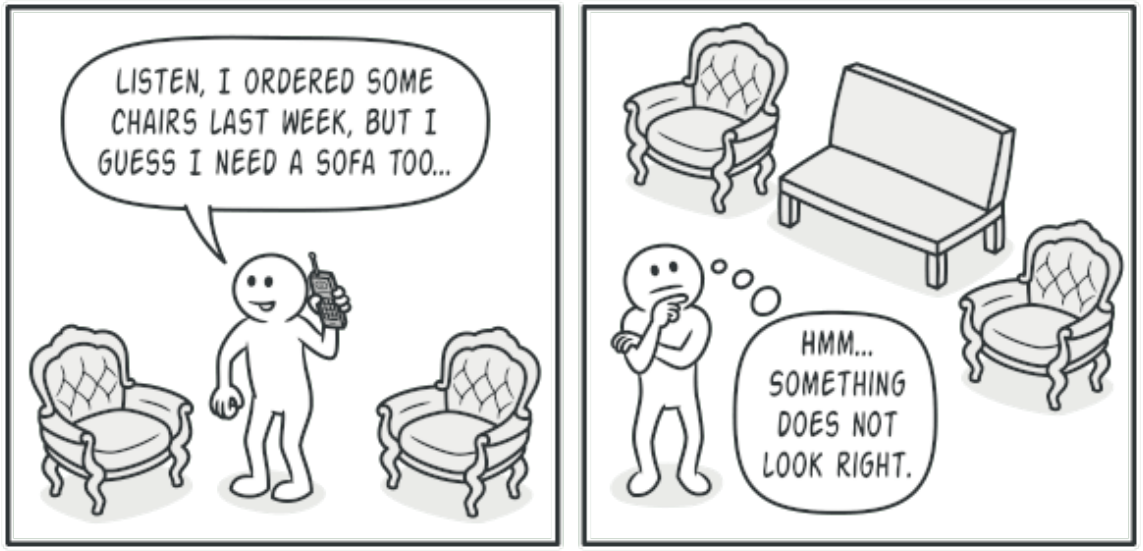
Factory Method – Pros and Cons

- ✓ You avoid tight coupling between the creator and the concrete products.
- ✓ *Single Responsibility Principle*. You can move the product creation code into one place in the program, making the code easier to support.
- ✓ *Open/Closed Principle*. You can introduce new types of products into the program without breaking existing client code.
- ✗ The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.

Abstract Factory

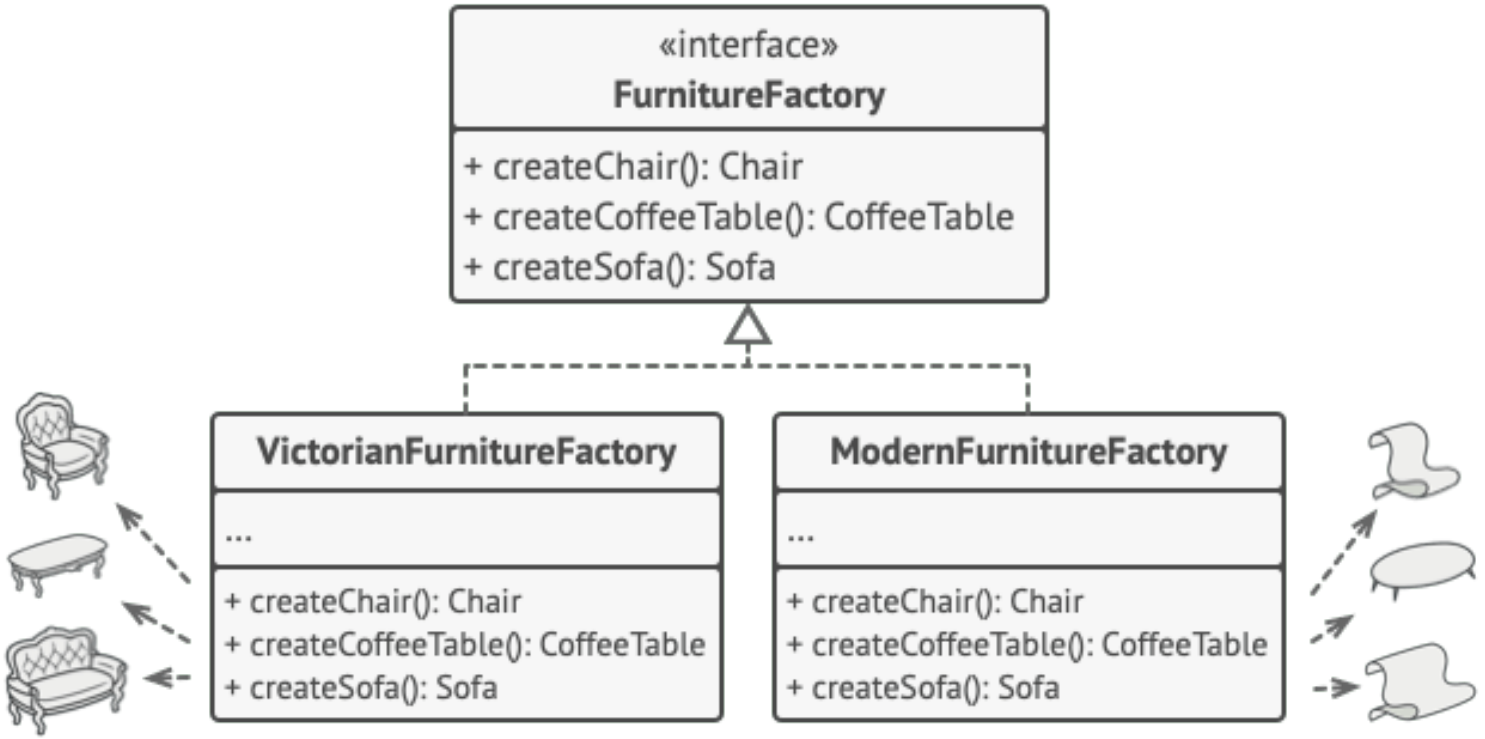
- Many designs start by using Factory Method (less complicated and more customizable via subclasses) and evolve toward Abstract Factory (more flexible, but more complicated).





Abstract Factory

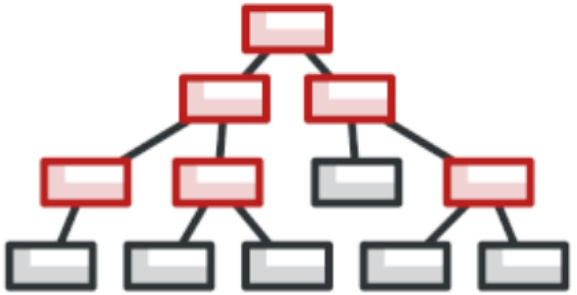
- Many designs start by using Factory Method (less complicated and more customizable via subclasses) and evolve toward Abstract Factory (more flexible, but more complicated).

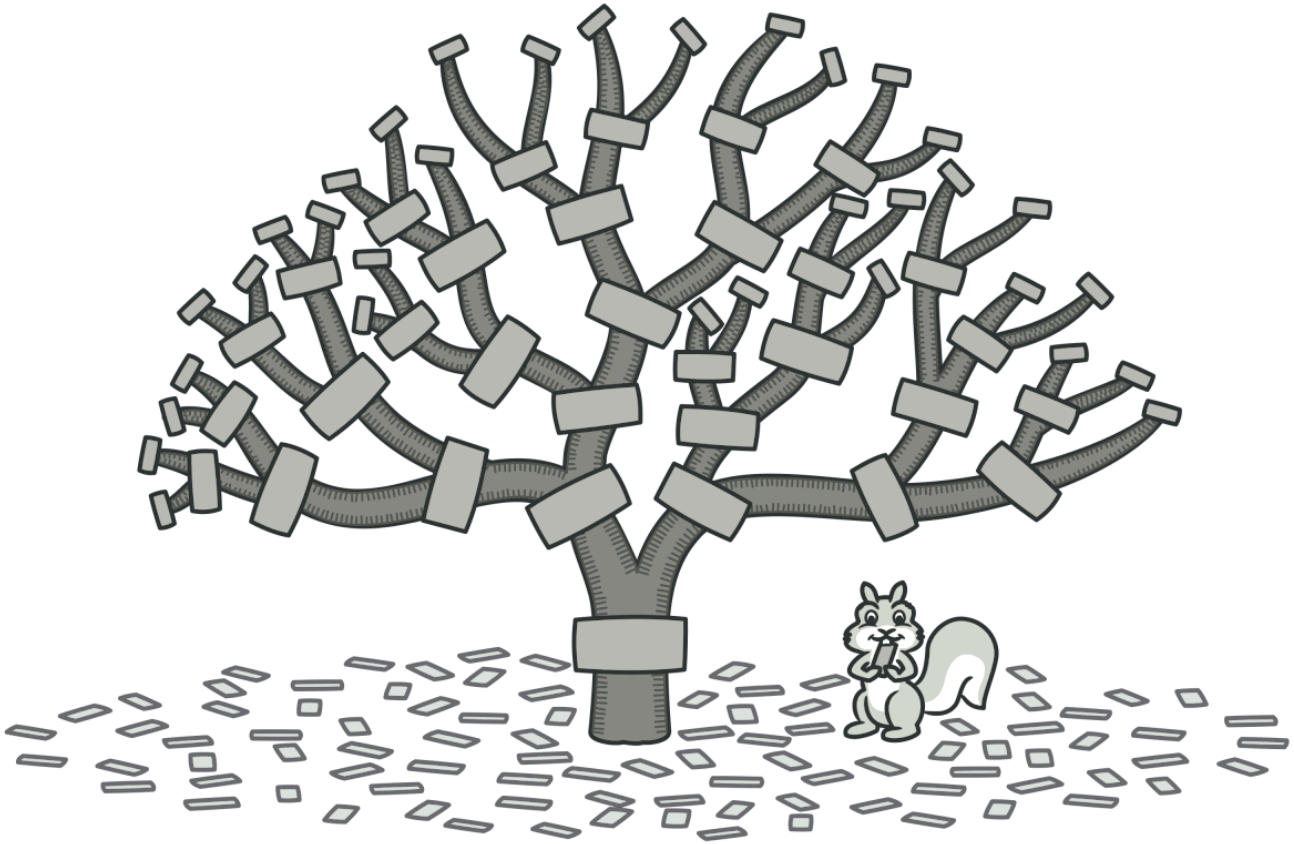


Classification of patterns

- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
- **Behavioral patterns**
  - **Strategy**

Composite Pattern





Composite Pattern

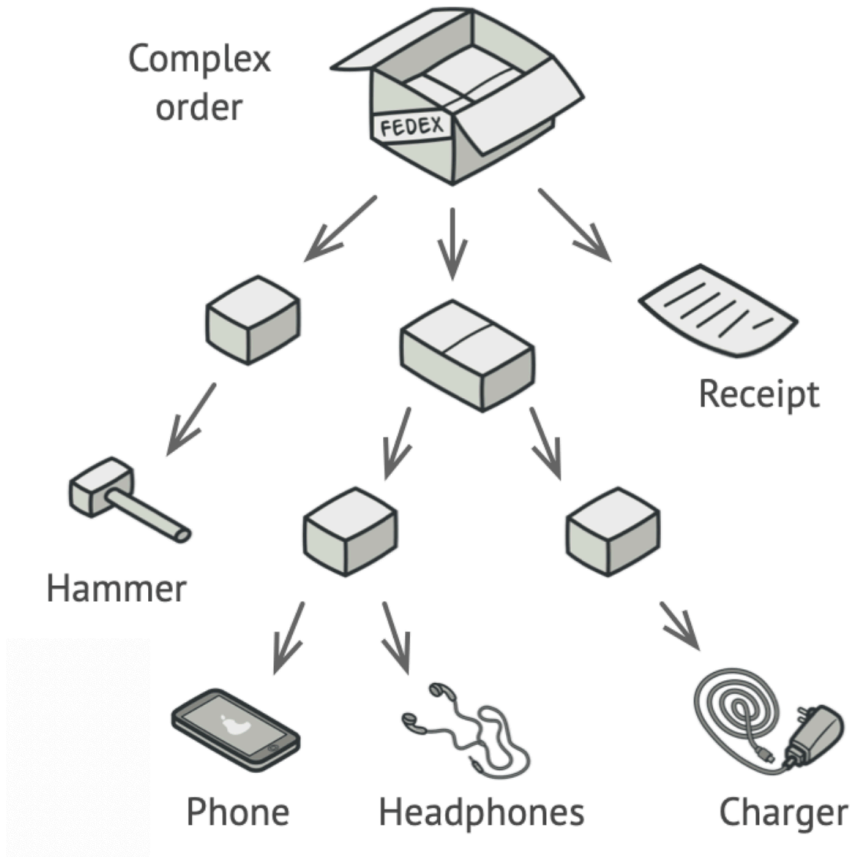
- Intent

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

- **Problem**

Using the Composite pattern makes sense only when the core model of your app can be represented as a tree.

Composite Pattern

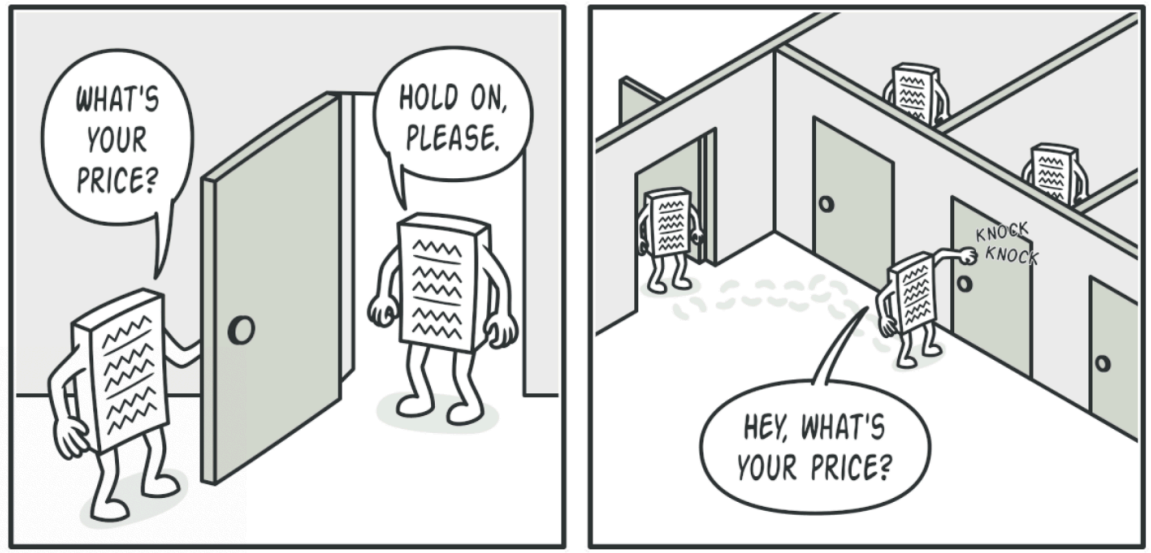


- 2 types of Objects
  - Products
  - Boxes

Composite Pattern

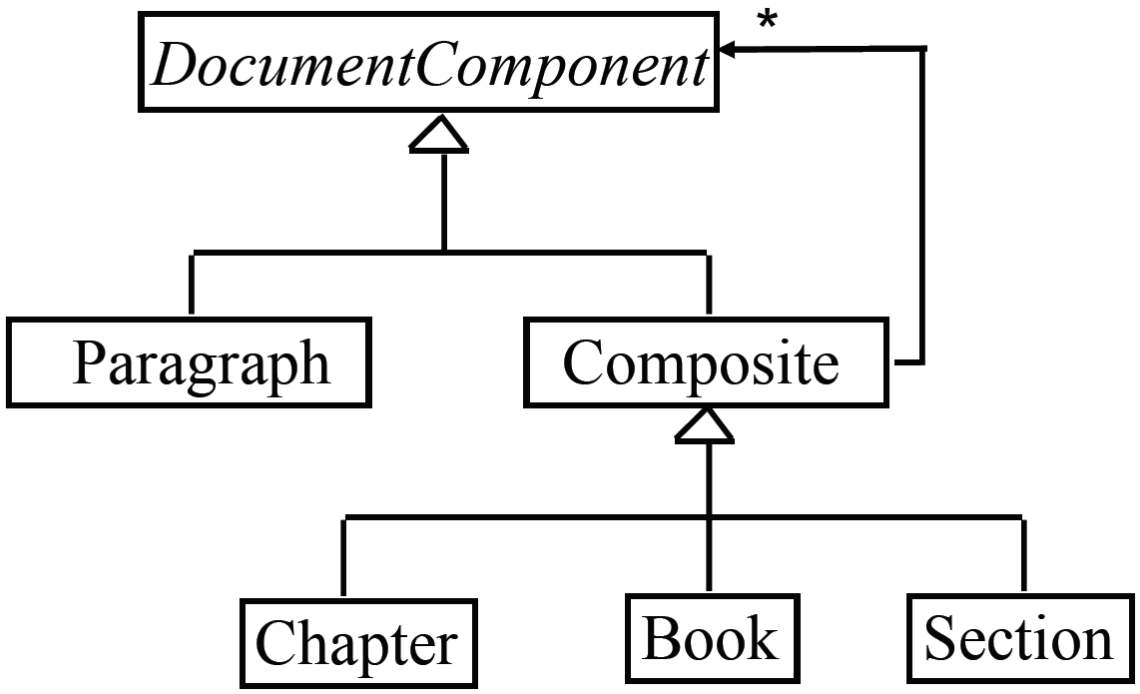
- Solution

Work with Products and Boxes through a common interface which declares a method for calculating the total price. (Recursively)



Composite Example

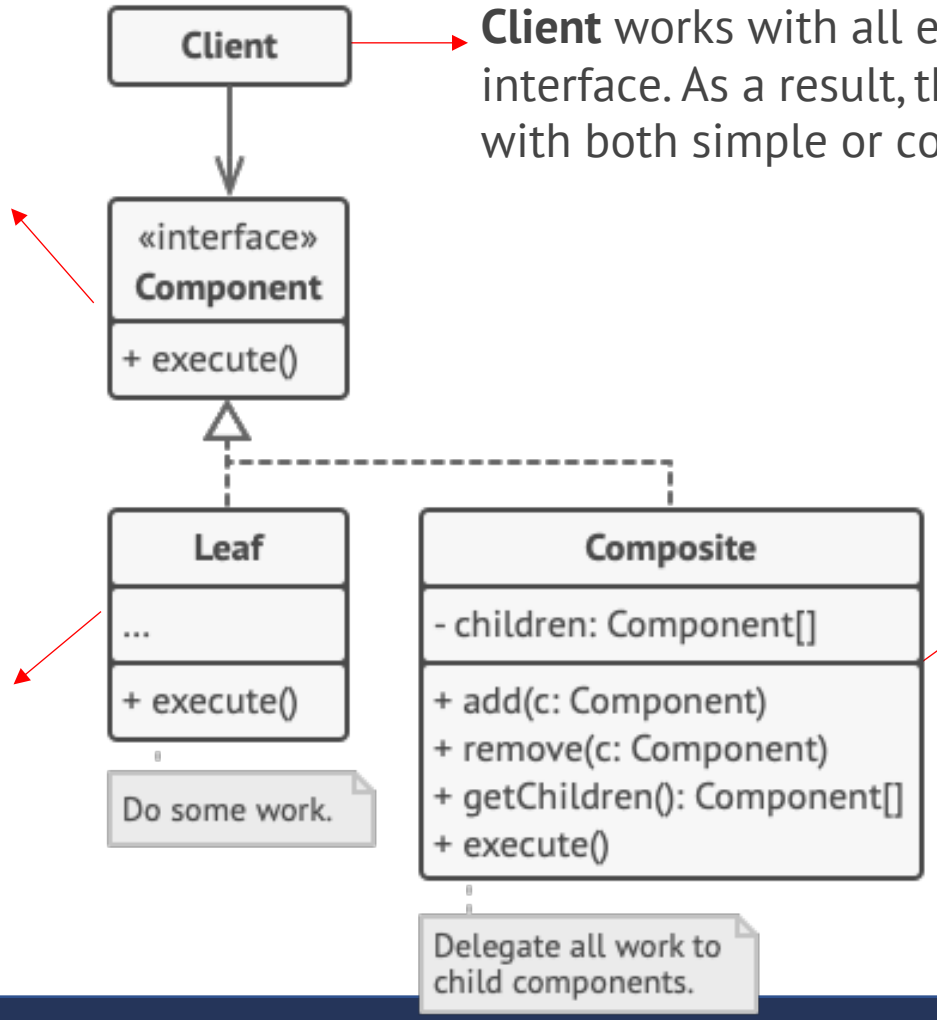
- Book



- Book
  - Chapter
    - Section
      - Paragraph
      - Paragraph
    - Section
      - Paragraph
  - Chapter
    - Section
      - Paragraph

Composite Design Pattern - Structure

The **Component** interface describes operations that are common to both simple and complex elements of the tree.



Client works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.

The **Leaf** is a basic element of a tree that doesn't have sub-elements.

The **Composite/container** is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface

Implementation

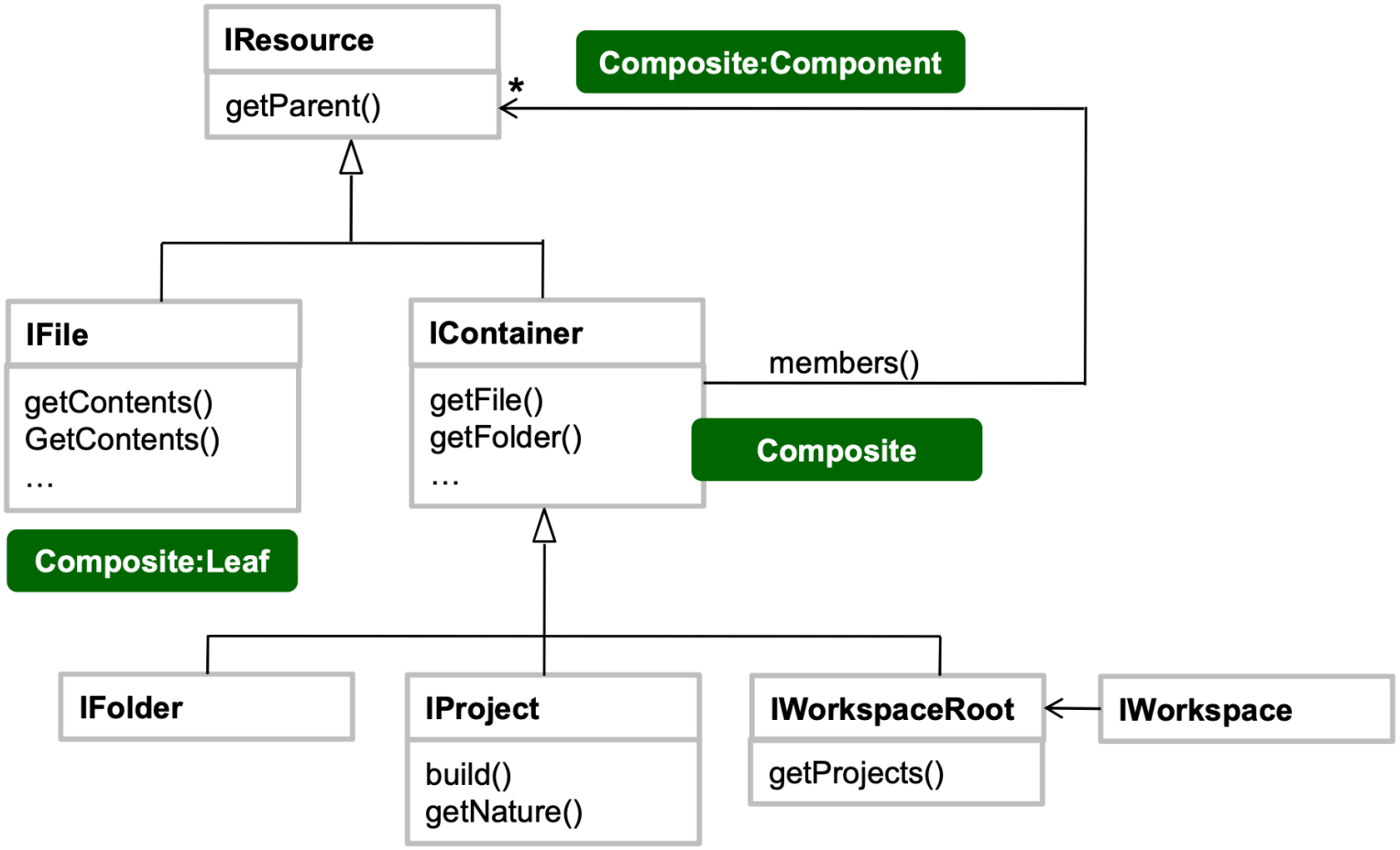
1. Make sure that the core model of your app can be represented as a tree structure. Try to break it down into simple elements and containers. Remember that containers must be able to contain both simple elements and other containers.
2. Declare the component interface with a list of methods that make sense for both simple and complex components.
3. Create a leaf class to represent simple elements. A program may have multiple different leaf classes.
4. Create a container class to represent complex elements. In this class, provide an array field for storing references to sub-elements. The array must be able to store both leaves and containers, so make sure it's declared with the component interface type.
5. While implementing the methods of the component interface, remember that a container is supposed to be delegating most of the work to sub-elements.
6. Finally, define the methods for adding and removal of child elements in the container.

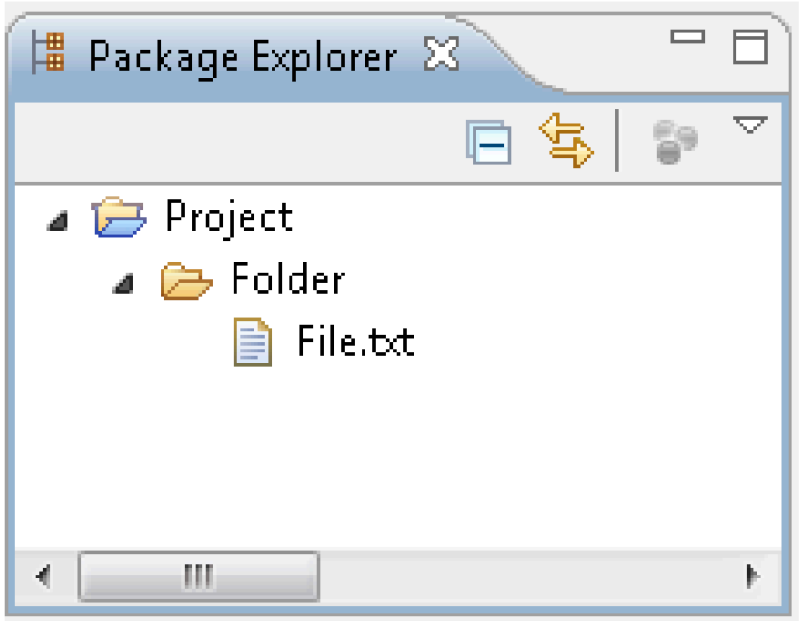
Usage of the pattern in Python

- **Usage examples:** The Composite pattern is pretty common in Python code. It's often used to represent hierarchies of user interface components or the code that works with graphs.
- **Identification:** If you have an object tree, and each object of a tree is a part of the same class hierarchy, this is most likely a composite. If methods of these classes delegate the work to child objects of the tree and do it via the base class/interface of the hierarchy, this is definitely a composite.

Real work application - Eclipse workspace, SWT

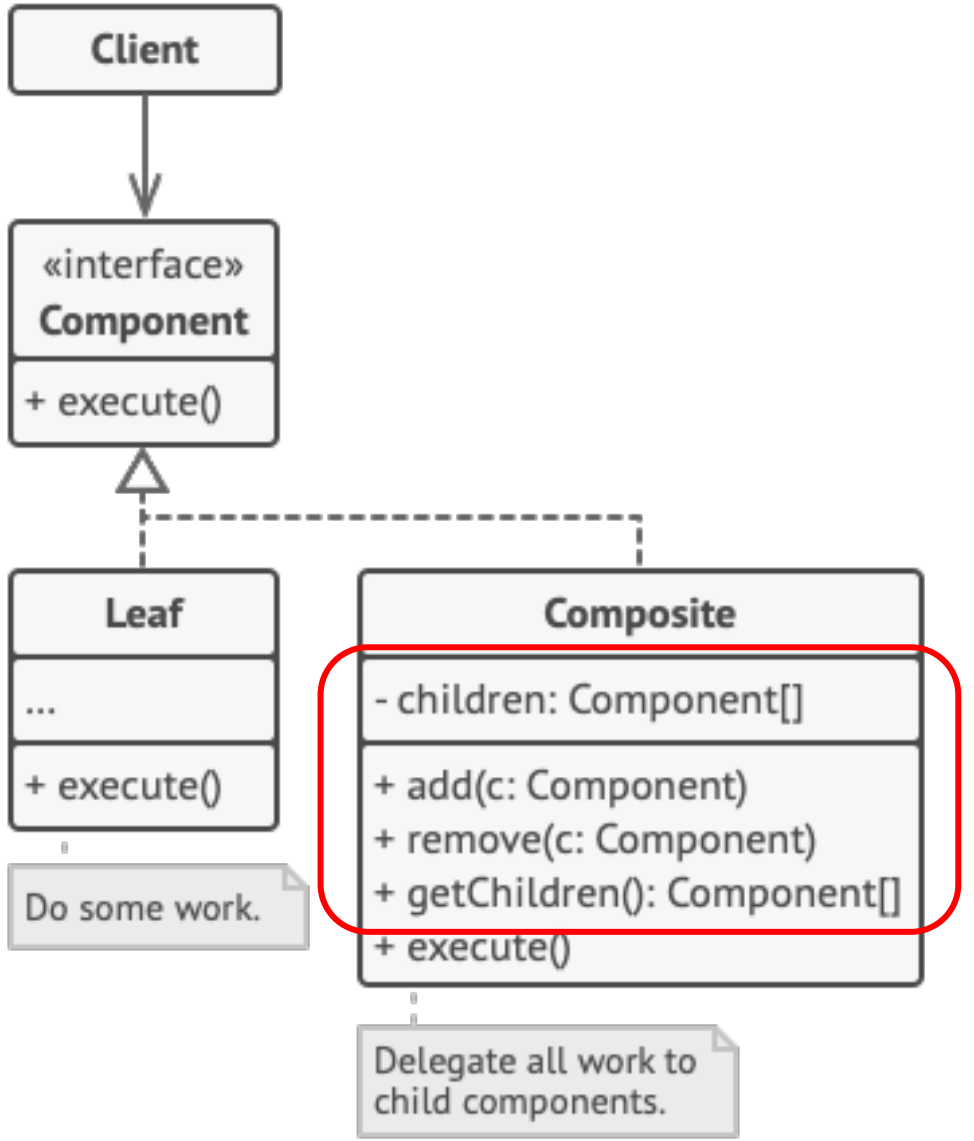
- IWorkspace is the root interface and it is a Composite of IContainers and IFiles.





Composite Pattern vs SOLID

- Which classes *declare* add and remove children operation?
- Trade-off between safety and transparency
  - **Component**: transparency, because you can treat all components uniformly. It costs you safety, however, because clients may try to do meaningless things like add and remove objects from leaves.
  - **Composite**: safety, because any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language. But you lose transparency, because leaves and composites have different interfaces.



Composite – Pros & Cons

- ✓ You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
- ✓ *Open/Closed Principle*. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

- ✗ It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.