Software technology

08 – Design Patterns (Part I)

Frontend Architecture

Lecture: Zoltán GERA / Practice: László GRAD-GYENGE

Editor & Presenter: Dr. Attila GLUDOVÁTZ

Online catalog – every week

- https://catalog.inf.elte.hu/
- Log in
- Username: yourUsername (@inf.elte.hu)
- Password: your email password
- Captcha: I generate a number for you…
- Lecture attendance is ***not*** optional! Max 3 misses and you are out

What’s a design pattern?

- Design patterns are typical solutions to commonly occurring problems in software design
- They are like pre-made blueprints that you can customize to solve a recurring design problem in your code
- Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems
  - While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution
  - The code of the same pattern applied to two different programs may be different

What does the pattern consist of?

- Most patterns are described very formally so people can reproduce them in many contexts
- Here are the sections that are usually present in a pattern description:
  - **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
- Some pattern catalogs list other useful details, such as applicability of the pattern, implementation steps and relations with other patterns

Why should I learn patterns?

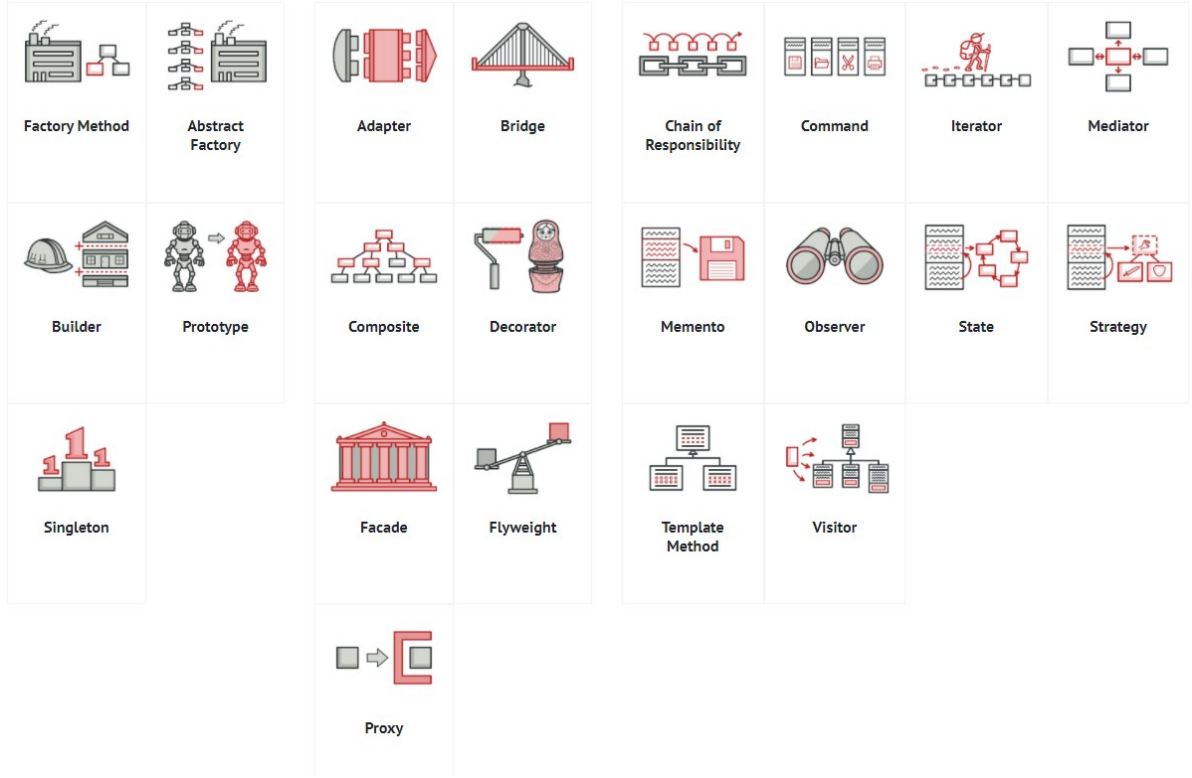
- *The truth is that you might manage to work as a programmer for many years without knowing about a single pattern. A lot of people do just that. Even in that case, though, you might be implementing some patterns without even knowing it. So why would you spend time learning them?*
  - Design patterns are a toolkit of tried and tested solutions to common problems in software design
  - Design patterns define a common language that you and your teammates can use to communicate more efficiently

Design Pattern

- Best practices
- Good examples
- Never code it directly!
  → Always write custom code tailored to your specific needs
- Common way of referring problems
- Naming convention

Design Pattern Types

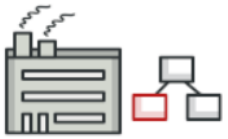
I. Creational
II. Structural
III. Behavioral
IV. Concurrency
V. Architectural
VI. Distributed
VII. Algorithm Strategy
VIII. Implementation Strategy



I. Creational Design Patterns

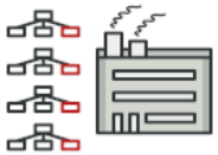
provide object creation mechanisms that increase flexibility and reuse of existing code

I. Creational Design Patterns



Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

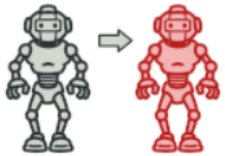


Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.

Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



Prototype

Lets you copy existing objects without making your code dependent on their classes.

...and:

- Lazy Initialization
- Object Pool
- RAII

Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.

I. Creational Design Patterns

- Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation
- The basic form of object creation could result in design problems or in added complexity to the design
- Creational design patterns solve this problem by somehow controlling this object creation

I. Creational Design Patterns

- Creational design patterns are composed of two dominant ideas:
  1. Encapsulating knowledge about which concrete classes the system uses
  2. Hiding how instances of these concrete classes are created and combined

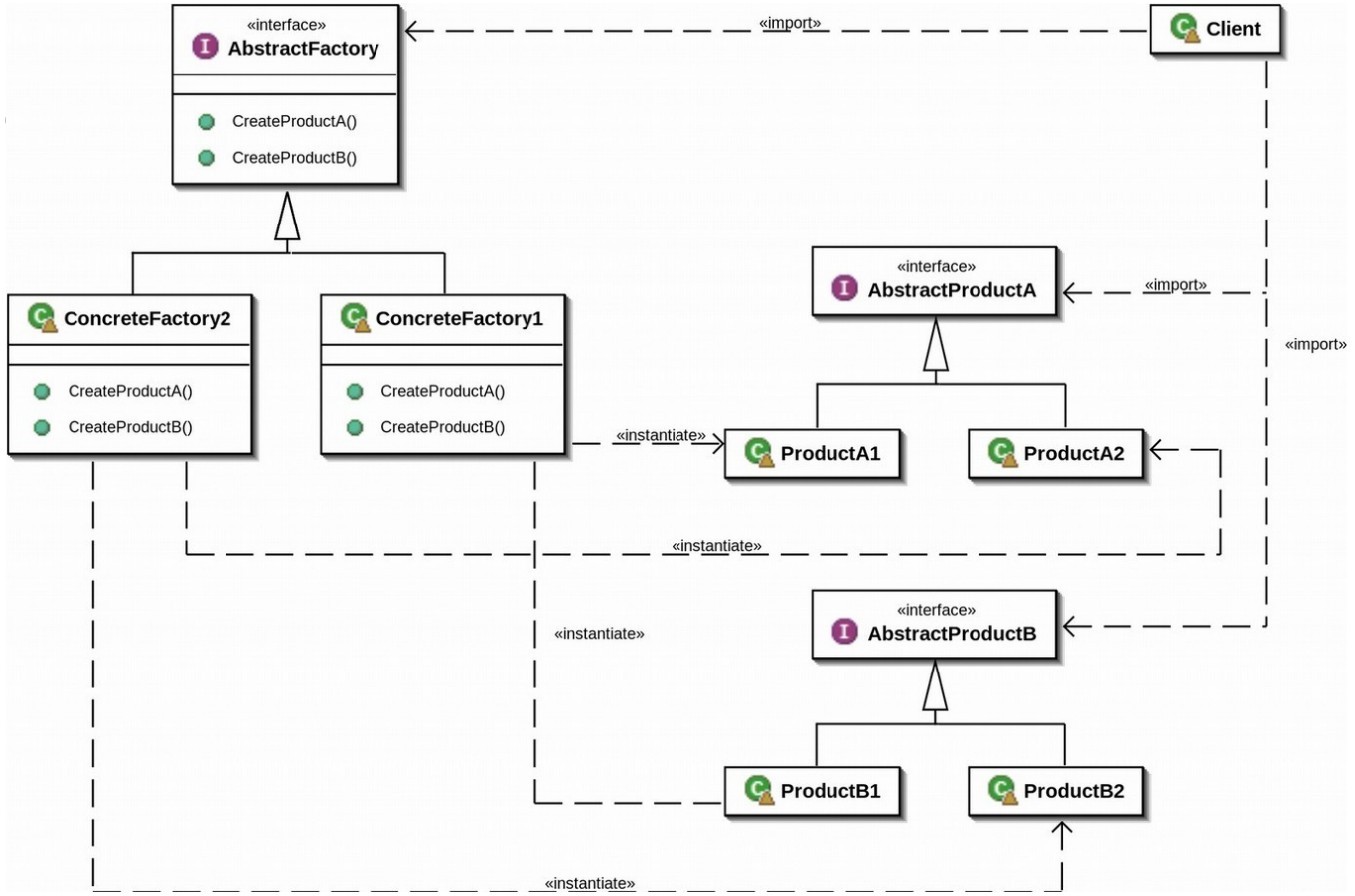
- Creational design patterns are further categorized into **Object-creational patterns** and **Class-creational patterns**,
  - Object-creational patterns deal with Object creation
  - Class-creational patterns deal with Class-instantiation

I. Creational Design Patterns

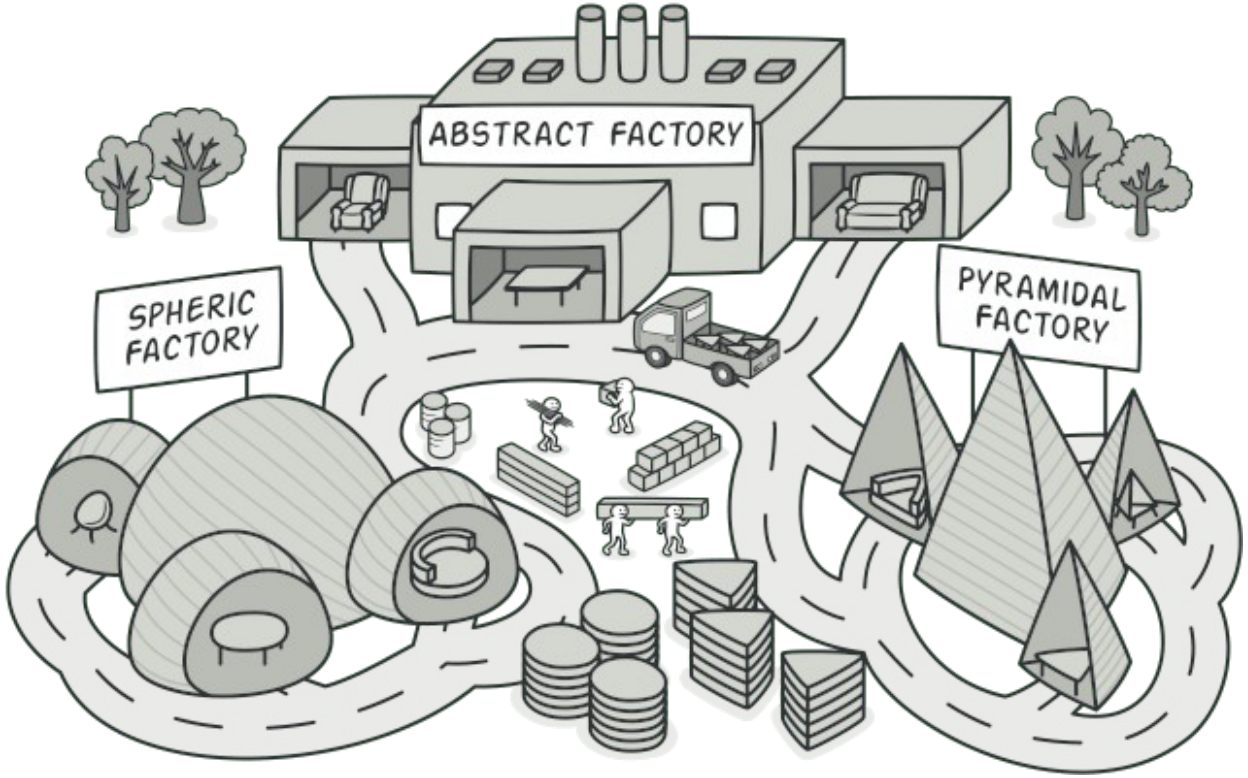
- Five well-known design patterns that are parts of creational patterns are the
  1. **Abstract factory pattern**, which provides an interface for creating related or dependent objects without specifying the objects' concrete classes
  2. **Builder pattern**, which separates the construction of a complex object from its representation so that the same construction process can create different representations
  3. **Factory method pattern**, which allows a class to defer instantiation to subclasses.
  4. **Prototype pattern**, which specifies the kind of object to create using a prototypical instance and creates new objects by cloning this prototype
  5. **Singleton pattern**, which ensures that a class only has one instance, and provides a global point of access to it

I. Creational Design Patterns
1. Abstract factory pattern

- Encapsulate multipl
  factories hiding
  implementation

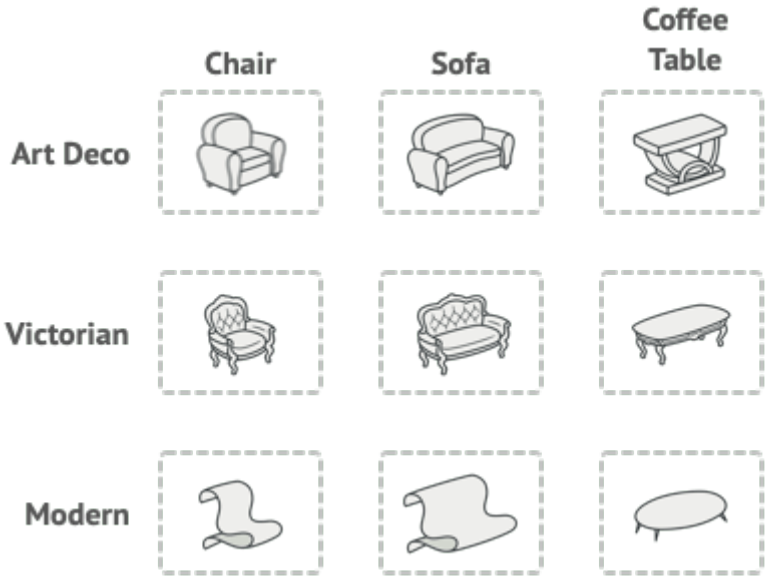


I. Creational Design Patterns
1. Abstract factory pattern

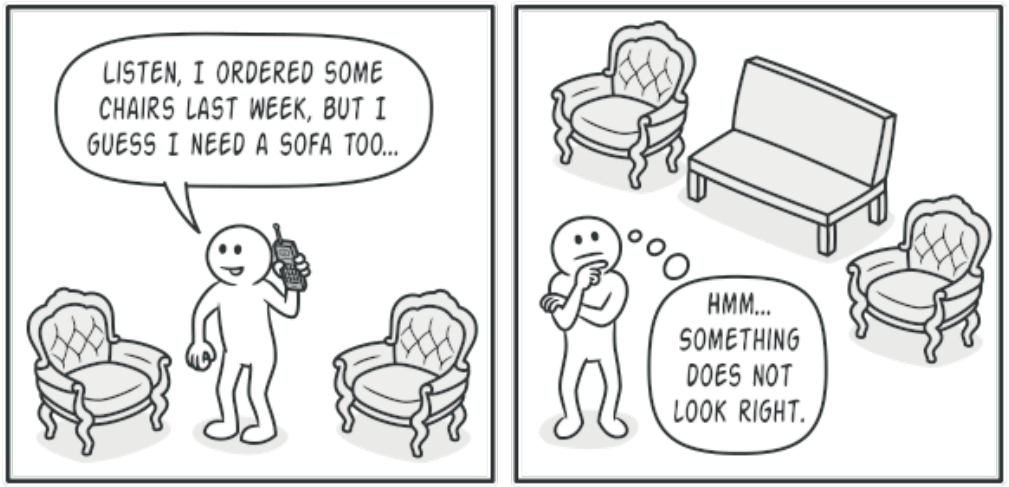


I. Creational Design Patterns
1. Abstract factory pattern – Problem

Product families and their variants

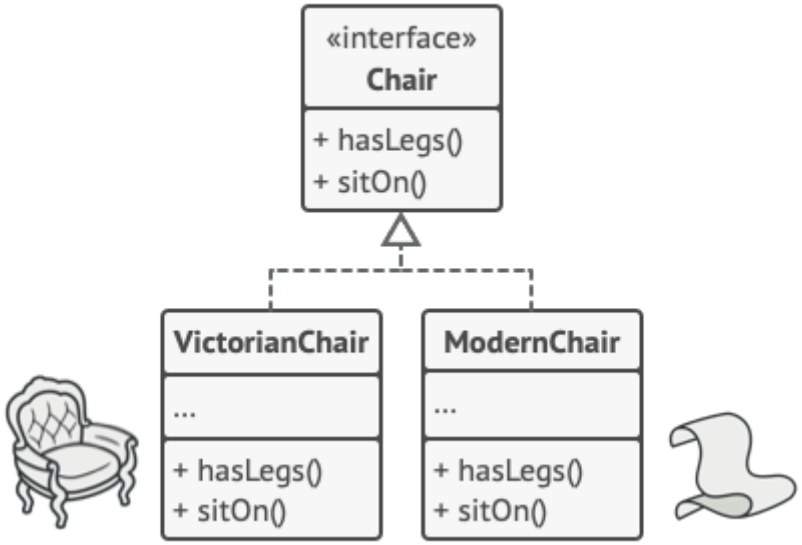


A Modern-style sofa doesn’t match Victorian-style chairs

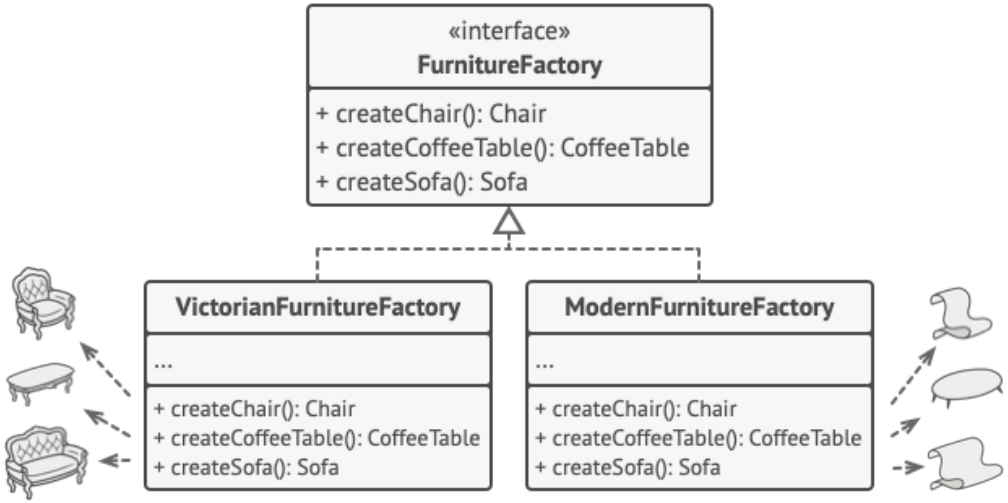


I. Creational Design Patterns
1. Abstract factory pattern – Solution

All variants of the same object must be moved to a single class hierarchy

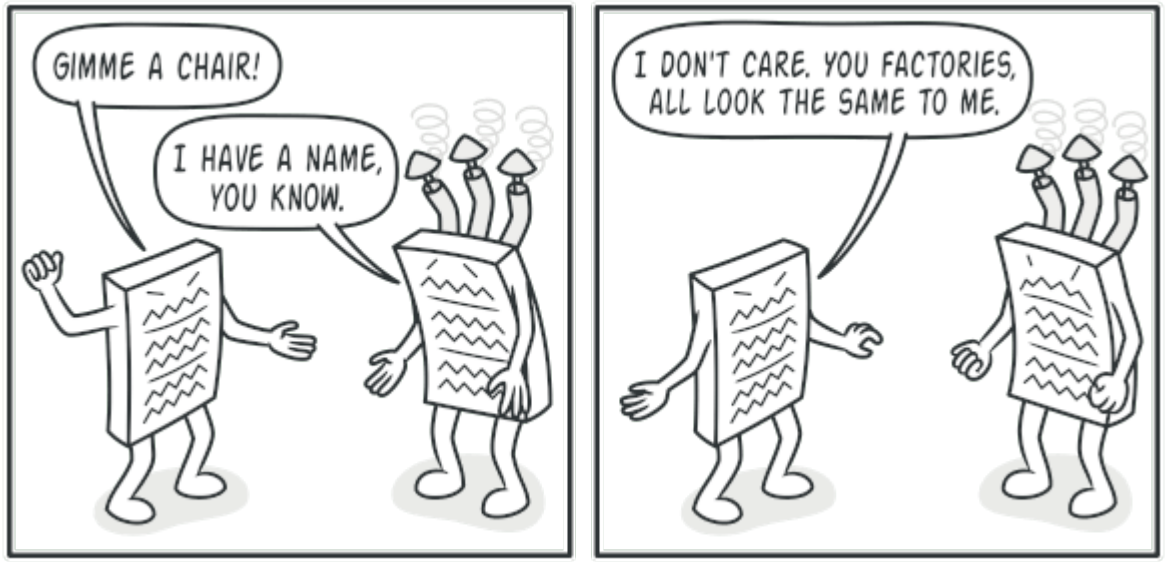


Each concrete factory corresponds to a specific product variant



I. Creational Design Patterns
1. Abstract factory pattern – Solution

- The client shouldn’t care about the concrete class of the factory it works with



I. Creational Design Patterns
1. Abstract factory pattern – Applicability

- *Use the Factory Method when …*
  - your code needs to work with various families of related products, but you don’t want it to depend on the concrete classes of those products—they might be unknown beforehand, or you simply want to allow for future extensibility

I. Creational Design Patterns
1. Abstract factory pattern – Pros and Cons

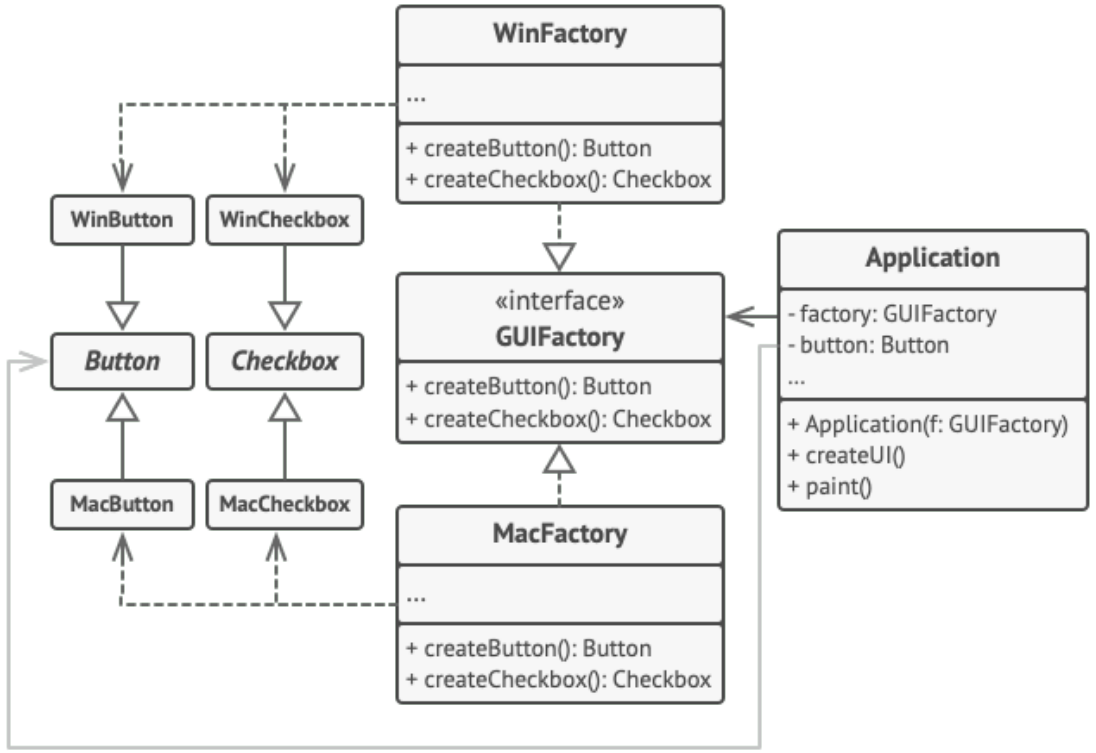
- You can be sure that the products you're getting from a factory are compatible with each other
- You avoid tight coupling between concrete products and client code
- Single Responsibility Principle. You can extract the product creation code into one place, making the code easier to support
- Open/Closed Principle. You can introduce new variants of products without breaking existing client code

- The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern

I. Creational Design Patterns
1. Abstract factory pattern – Relations with Other Patterns

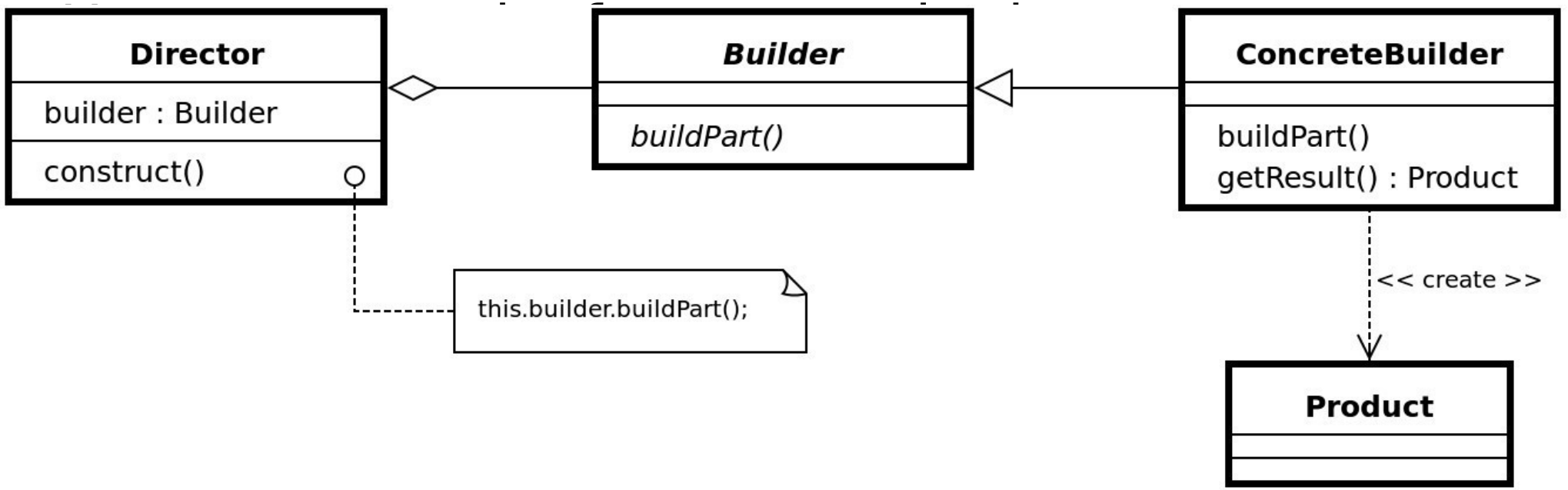
- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory, Prototype,** or **Builder** (more flexible, but more complicated).
- Abstract Factory specializes in creating families of related objects. Abstract Factory returns the product immediately, whereas Builder lets you run some additional construction steps before fetching the product.
- **Abstract Factory** classes are often based on a set of **Factory Methods**, but you can also use **Prototype** to compose the methods on these classes.
- **Abstract Factory** can serve as an alternative to **Facade** when you only want to hide the way the subsystem objects are created from the client code.
- You can use **Abstract Factory** along with **Bridge**. This pairing is useful when some abstractions defined by Bridge can only work with specific implementations. In this case, Abstract Factory can encapsulate these relations and hide the complexity from the client code.
- **Abstract Factories, Builders** and **Prototypes** can all be implemented as **Singletons**

I. Creational Design Patterns
1. Abstract factory pattern – The cross-platform UI classes example



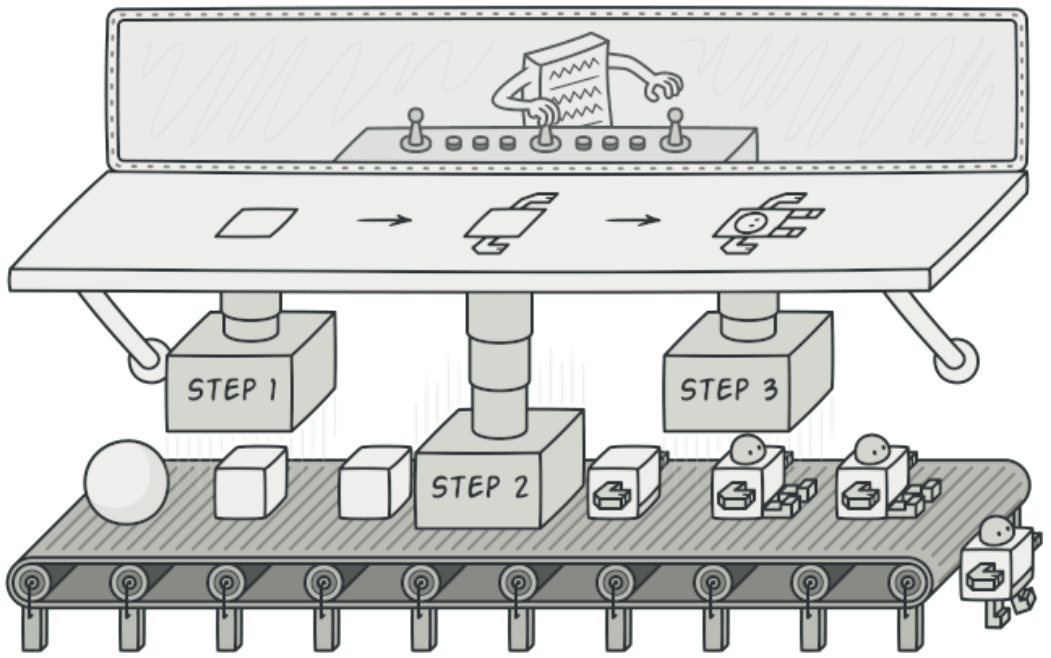
I. Creational Design Patterns
2. Builder pattern

- Instead of constructors of long / different parameter lists



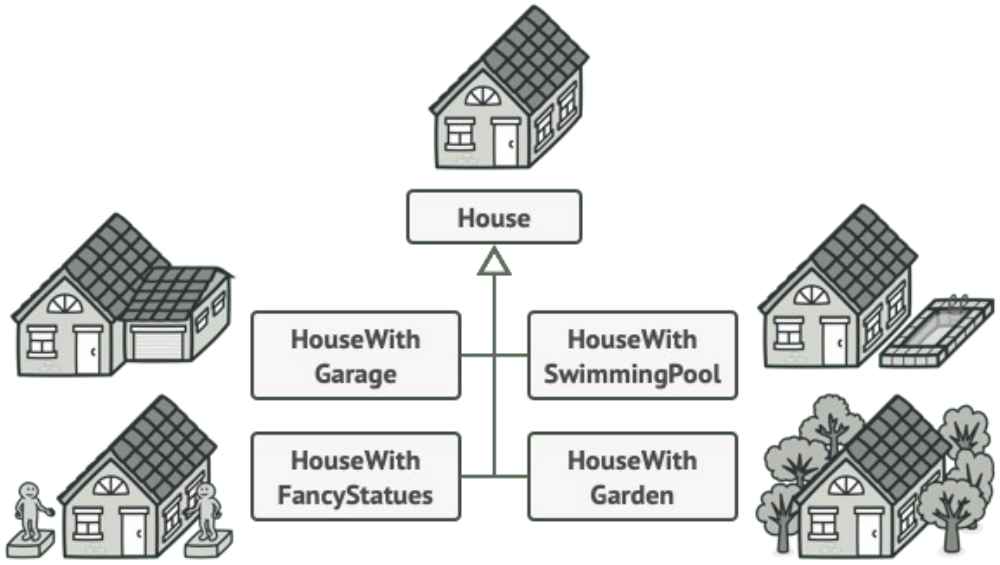
I. Creational Design Patterns
2. Builder pattern

- **Builder** is a creational design pattern that lets you *construct complex objects step by step*
- The pattern allows you to produce different types and representations of an object using the same construction code

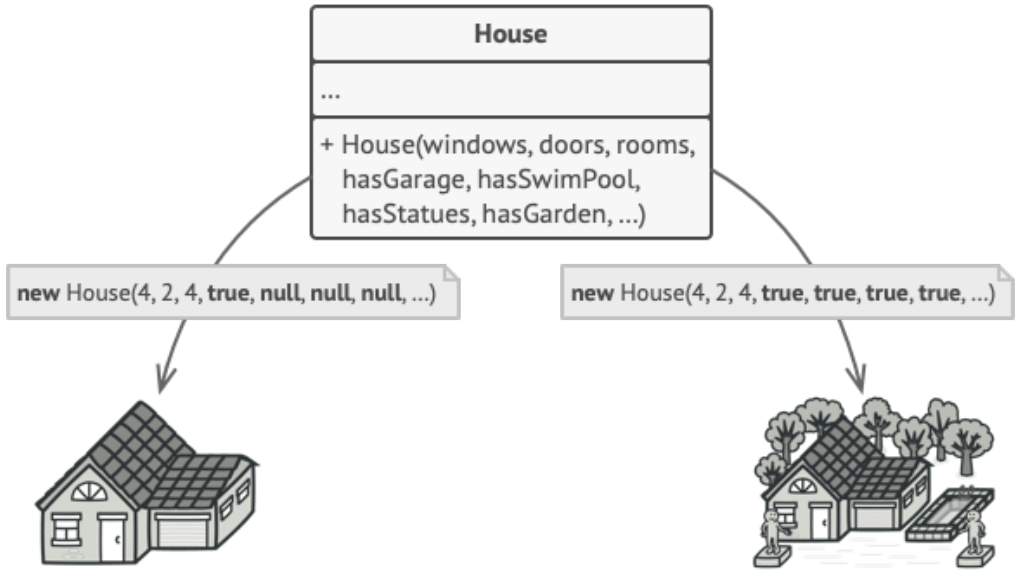


I. Creational Design Patterns
2. Builder pattern – Problem

You might make the program too complex by creating a subclass for every possible configuration of an object

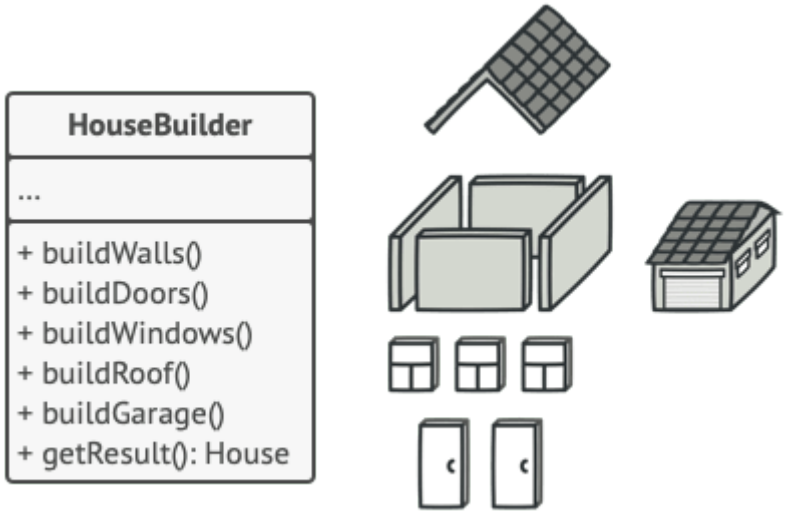


The constructor with lots of parameters has its downside: not all the parameters are needed at all times

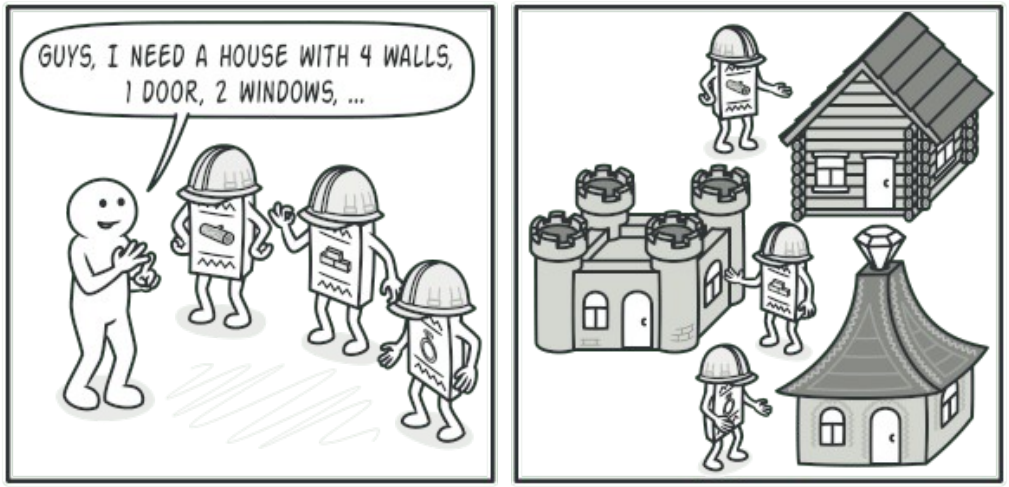


I. Creational Design Patterns
2. Builder pattern – Solution

The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built



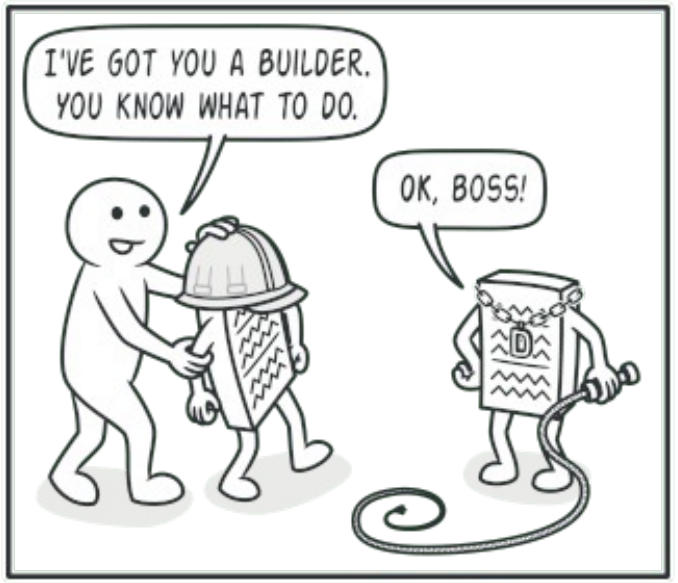
Different builders execute the same task in various ways



I. Creational Design Patterns
2. Builder pattern – Solution (Director)

- You can go further and extract a series of calls to the builder steps you use to construct a product into a separate class called **director**
- The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps

The director knows which building steps to execute to get a working product



I. Creational Design Patterns
2. Builder pattern – Applicability

```
class Pizza {
    Pizza(int size) { ... }
    Pizza(int size, boolean cheese) { ... }
    Pizza(int size, boolean cheese, boolean pepperoni) { ... }
    // ...
```

Creating such a monster is only possible in languages that support method overloading, such as C# or Java.

- Use the Builder pattern to get rid of a “telescopic constructor”
  - Say you have a constructor with ten optional parameters. Calling such a beast is very inconvenient; therefore, you overload the constructor and create several shorter versions with fewer parameters. These constructors still refer to the main one, passing some default values into any omitted parameters
  - The Builder pattern lets you build objects step by step, using only those steps that you really need. After implementing the pattern, you don’t have to cram dozens of parameters into your constructors anymore

I. Creational Design Patterns
2. Builder pattern – Applicability

- Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses)
- Use the Builder to construct (II.3.) Composite trees or other complex objects
  - The Builder pattern lets you construct products step-by-step. You could defer execution of some steps without breaking the final product. You can even call steps recursively, which comes in handy when you need to build an object tree.
  - A builder doesn’t expose the unfinished product while running construction steps. This prevents the client code from fetching an incomplete result

I. Creational Design Patterns
2. Builder pattern– Pros and Cons

- You can construct objects step-by-step, defer construction steps or run steps recursively
- You can reuse the same construction code when building various representations of products
- *Single Responsibility Principle*. You can isolate complex construction code from the business logic of the product

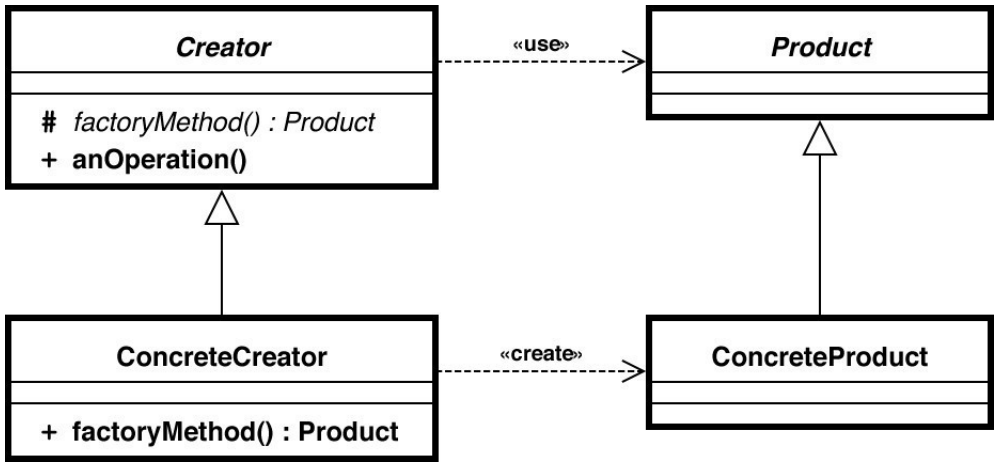
- The overall complexity of the code increases since the pattern requires creating multiple new classes

I. Creational Design Patterns
2. Builder pattern – Relations with Other Patterns

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory, Prototype**, or **Builder** (more flexible, but more complicated).
- **Builder** focuses on constructing complex objects step by step. **Abstract Factory** specializes in creating families of related objects. Abstract Factory returns the product immediately, whereas Builder lets you run some additional construction steps before fetching the product.
- You can use **Builder** when creating complex **Composite** trees because you can program its construction steps to work recursively.
- You can combine **Builder** with **Bridge**: the director class plays the role of the abstraction, while different builders act as implementations.
- **Abstract Factories, Builders** and **Prototypes** can all be implemented as **Singletons**

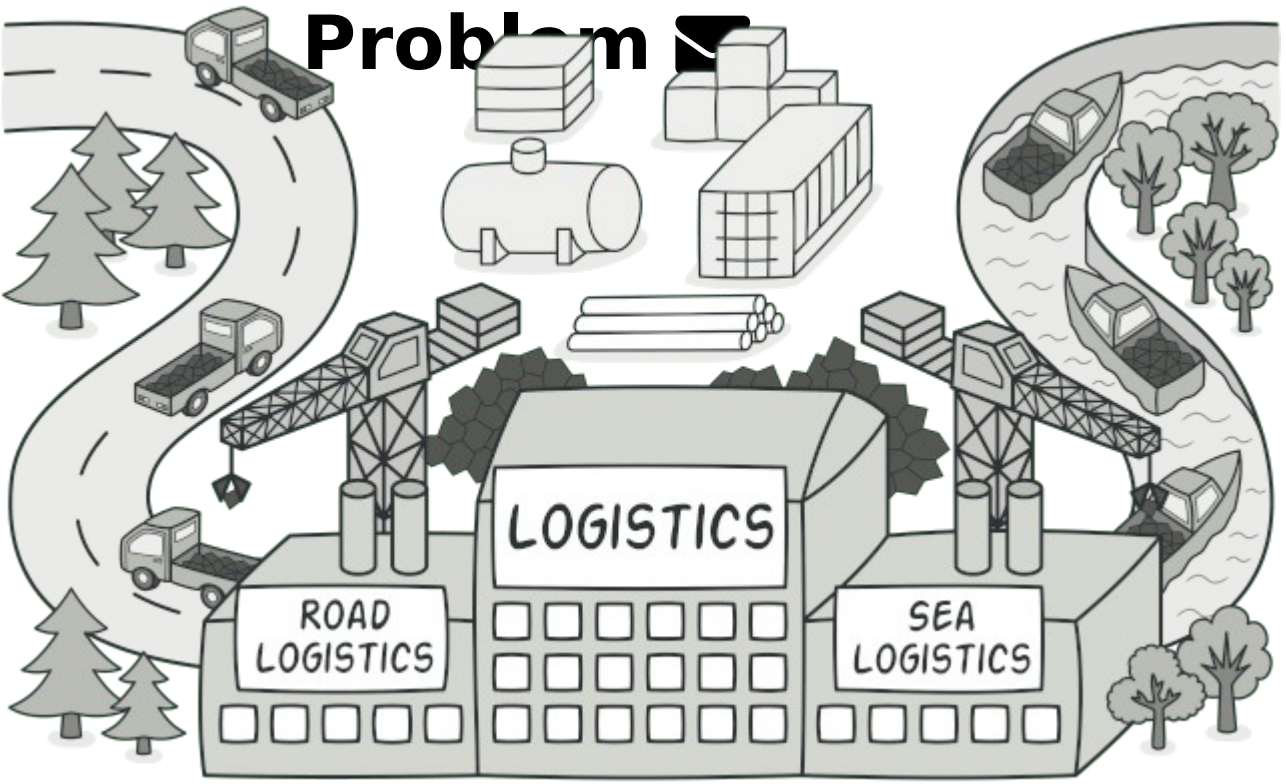
I. Creational Design Patterns
3. Factory method pattern

- Also known as: Virtual Constructor
- 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



I. Creational Design Patterns
3. Factory method pattern – Situation & Problem

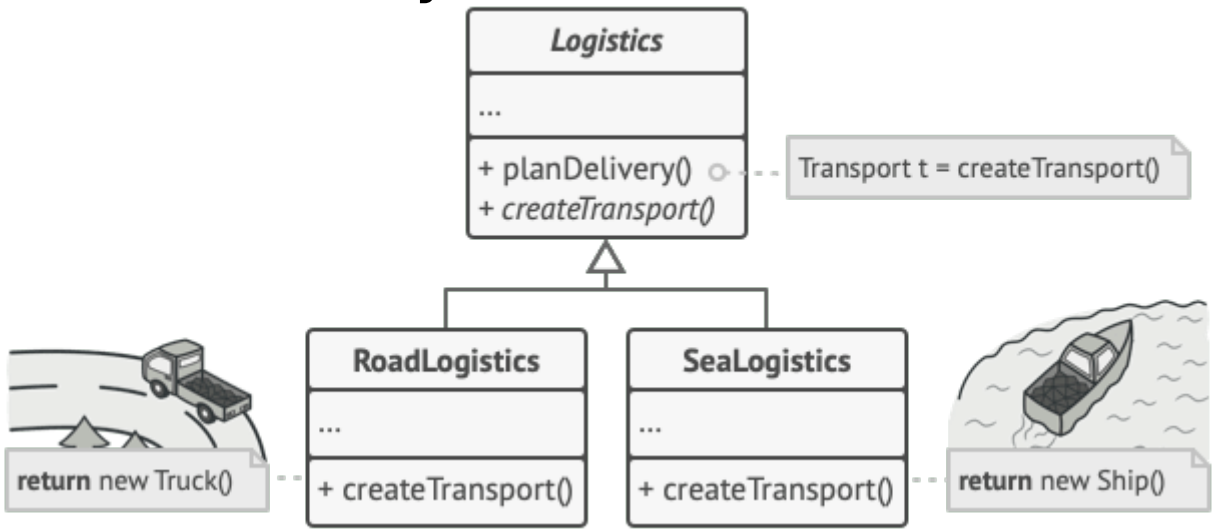
- **Situation & Problem**



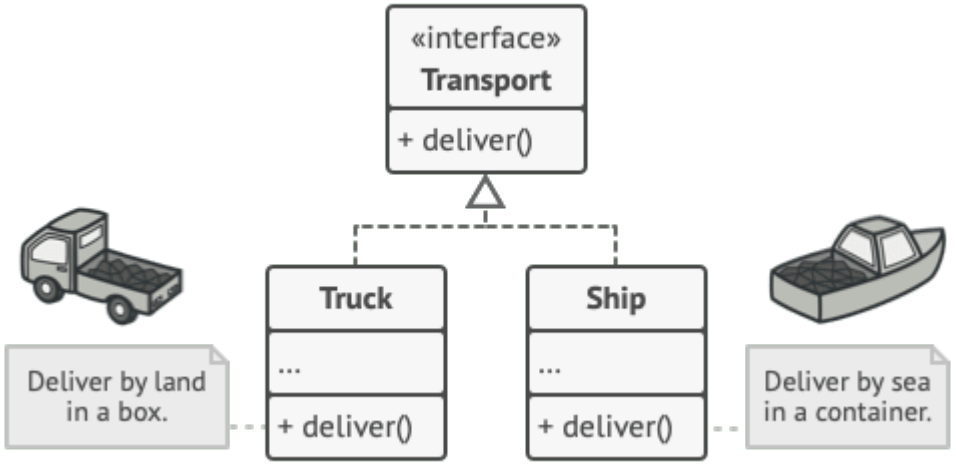
- Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes

I. Creational Design Patterns
3. Factory method pattern – Solution

- Subclasses can alter the class of objects being returned by the factory method

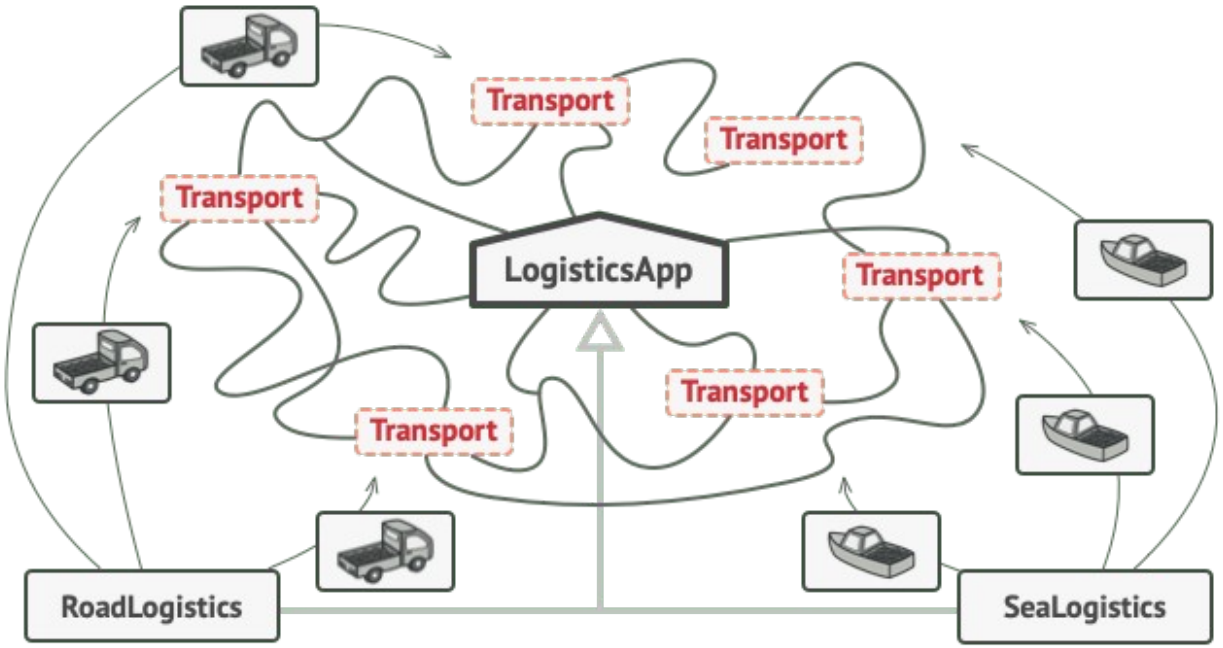


- All products must follow the same interface



I. Creational Design Patterns
3. Factory method pattern

- As long as all product classes implement a common interface, you can pass their objects to the client code without breaking it



I. Creational Design Patterns
3. Factory method pattern – Applicability

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

I. Creational Design Patterns
3. Factory method pattern – 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

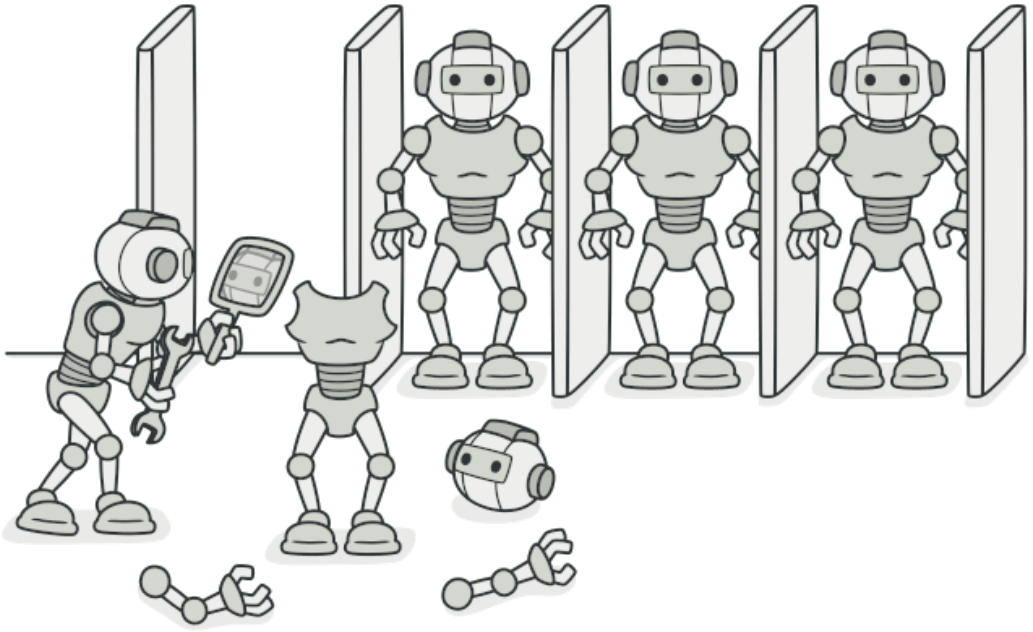
- The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best-case scenario is when you’re introducing the pattern into an existing hierarchy of creator classes

I. Creational Design Patterns
3. Factory method pattern – Relations with Other Patterns

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory, Prototype,** or **Builder** (more flexible, but more complicated)
- **Abstract Factory** classes are often based on a set of **Factory Methods**, but you can also use Prototype to compose the methods on these classes
- You can use **Factory Method** along with **Iterator** to let collection subclasses return different types of iterators that are compatible with the collections
- **Prototype** isn’t based on inheritance, so it doesn’t have its drawbacks. On the other hand, Prototype requires a complicated initialization of the cloned object. **Factory Method** is based on inheritance but doesn’t require an initialization step
- **Factory Method** is a specialization of **Template Method**. At the same time, a *Factory Method* may serve as a step in a large *Template Method*

I. Creational Design Patterns
4. Prototype pattern (Also known as: Clone)

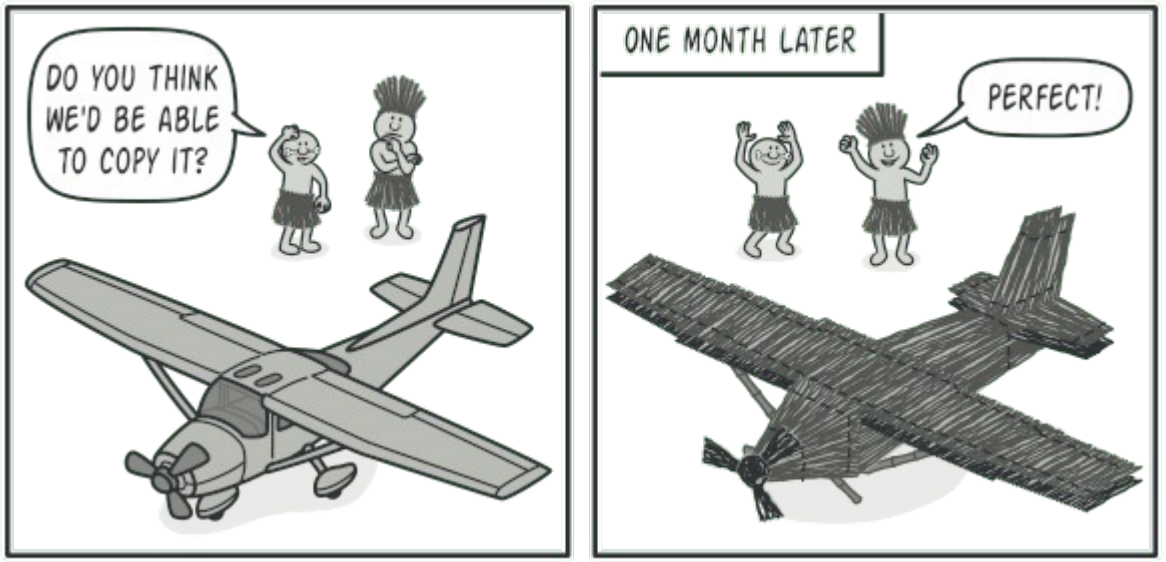
- Prototype is a creational design pattern that lets you *copy existing objects without making your code dependent on their classes*



I. Creational Design Patterns
4. Prototype pattern – Problem

- Say you have an object, and you want to create an exact copy of it. How would you do it? First, you have to create a new object of the same class. Then you have to go through all the fields of the original object and copy their values over to the new object.
- Nice! But there's a catch. Not all objects can be copied that way because some of the object's fields may be private and not visible from outside of the object itself

Copying an object "from the outside" isn't always possible



I. Creational Design Patterns
4. Prototype pattern – Solution

- The Prototype pattern delegates the cloning process to the actual objects that are being cloned
- The pattern declares a common interface for all objects that support cloning
  - This interface lets you clone an object without coupling your code to the class of that object.
- Usually, such an interface contains just a single **clone** method
  - The implementation of the clone method is very similar in all classes
  - The method creates an object of the current class and carries over all of the field values of the old object into the new one
- You can even copy private fields because most programming languages let objects access private fields of other objects that belong to the same class

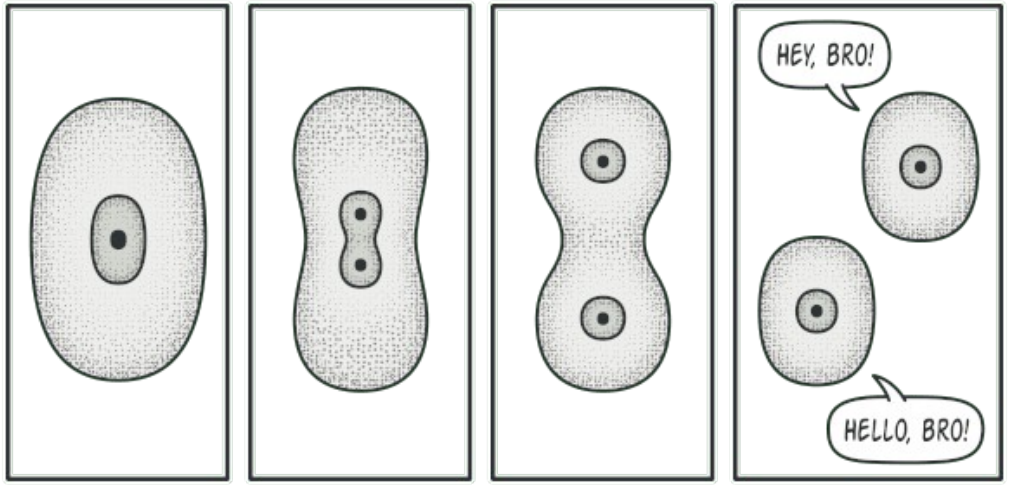
I. Creational Design Patterns
4. Prototype pattern – Solution

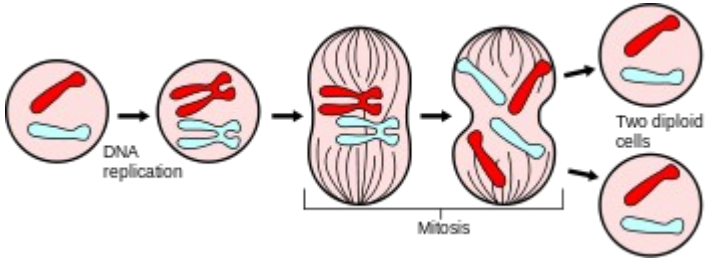
- An object that supports cloning is called a **prototype**
- When your objects have dozens of fields and hundreds of possible configurations, cloning them might serve as an alternative to subclassing
- Here's how it works: you create a set of objects, configured in various ways. When you need an object like the one you've configured, you just clone a prototype instead of constructing a new object from scratch

Pre-built prototypes can be an alternative to subclassing

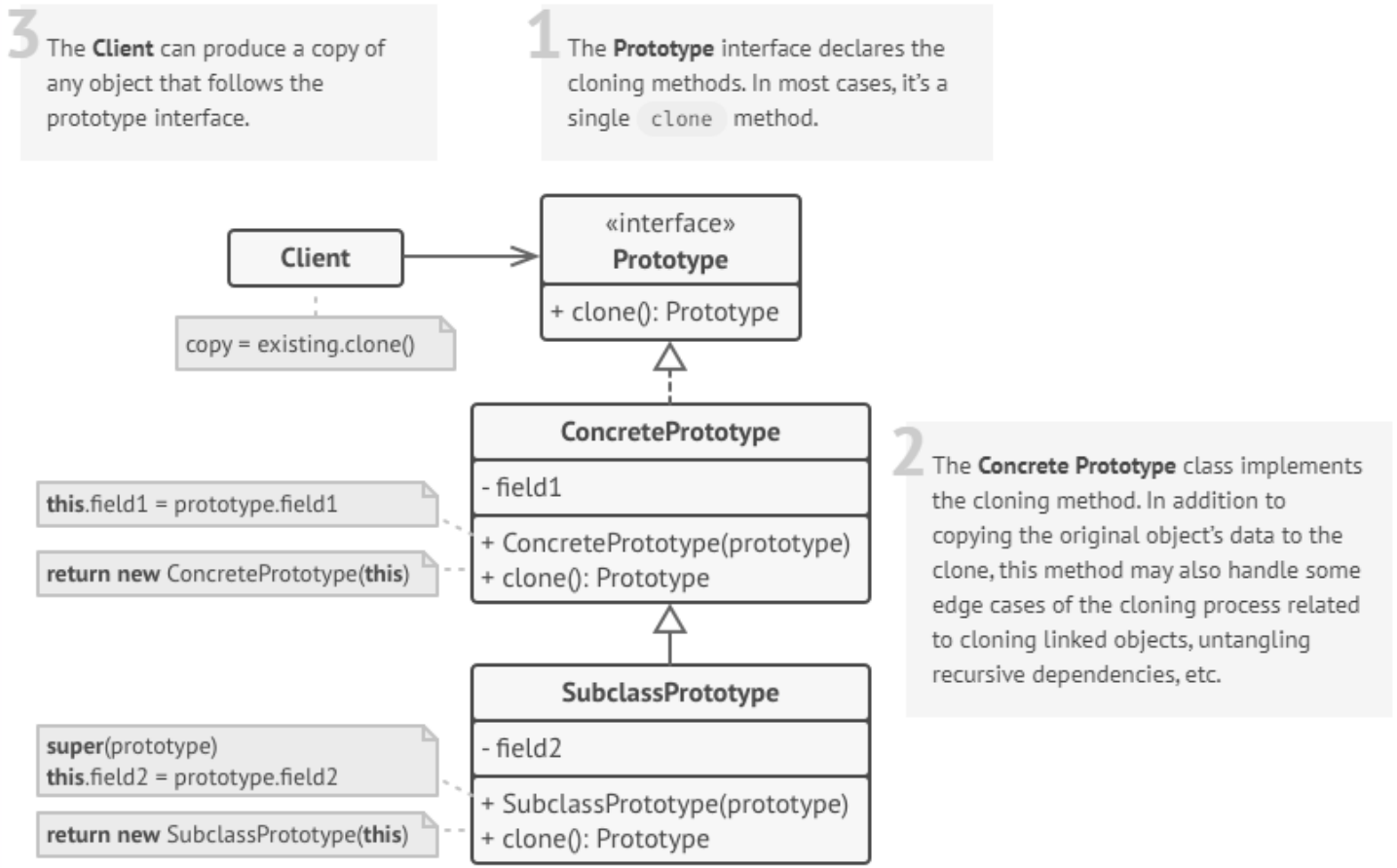
I. Creational Design Patterns
4. Prototype pattern – Real-world analogy

- The process of mitotic cell division (biology)
  - After mitotic division, a pair of identical cells is formed
  - The original cell acts as a prototype and takes an active role in creating the copy
- In real life, prototypes are used for performing various tests before starting mass production of a product. However, in this case, prototypes don't participate in any actual production, playing a passive role instead





I. Creational Design Patterns
4. Prototype pattern – Structure



I. Creational Design Patterns
4. Prototype pattern – Applicability

- *Use the Factory Method when …*
  - your code shouldn’t depend on the concrete classes of objects that you need to copy
  - you want to reduce the number of subclasses that only differ in the way they initialize their respective objects. Somebody could have created these subclasses to be able to create objects with a specific configuration

I. Creational Design Patterns
4. Prototype pattern – Pros and Cons

- You can clone objects without coupling to their concrete classes.
- You can get rid of repeated initialization code in favor of cloning pre-built prototypes.
- You can produce complex objects more conveniently.
- You get an alternative to inheritance when dealing with configuration presets for complex objects

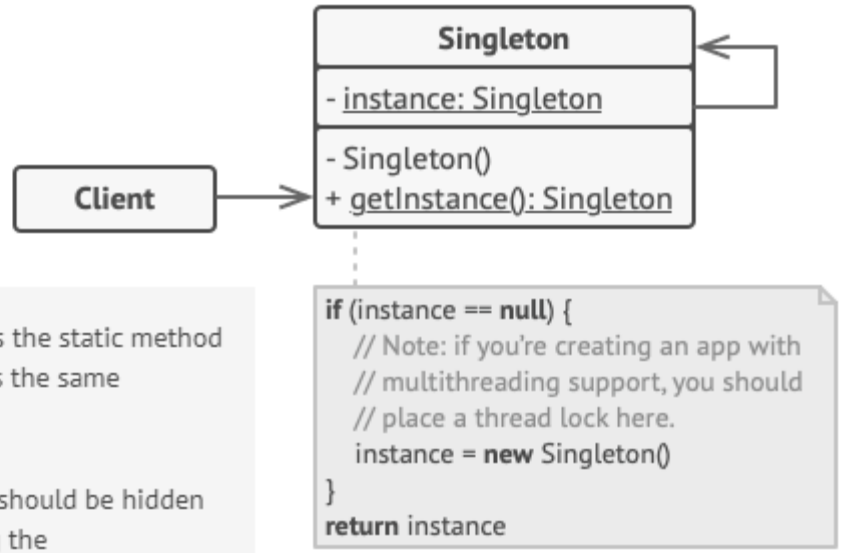
- Cloning complex objects that have circular references might be very tricky

I. Creational Design Patterns
4. Prototype pattern – Relations with Other Patterns

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory, Prototype,** or **Builder** (more flexible, but more complicated)
- **Abstract Factory** classes are often based on a set of **Factory Methods**, but you can also use Prototype to compose the methods on these classes
- **Prototype** can help when you need to save copies of **Commands** into history.
- Designs that make heavy use of **Composite** and **Decorator** can often benefit from using **Prototype**. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.
- **Prototype** isn’t based on inheritance, so it doesn’t have its drawbacks. On the other hand, Prototype requires a complicated initialization of the cloned object. **Factory Method** is based on inheritance but doesn’t require an initialization step.
- Sometimes **Prototype** can be a simpler alternative to **Memento**. This works if the object, the state of which you want to store in the history, is fairly straightforward and doesn’t have links to external resources, or the links are easy to re-establish
- **Abstract Factories, Builders** and **Prototypes** can all be implemented as **Singletons**

I. Creational Design Patterns
5. Singleton pattern

- Can be an anti-patterr
- Testing

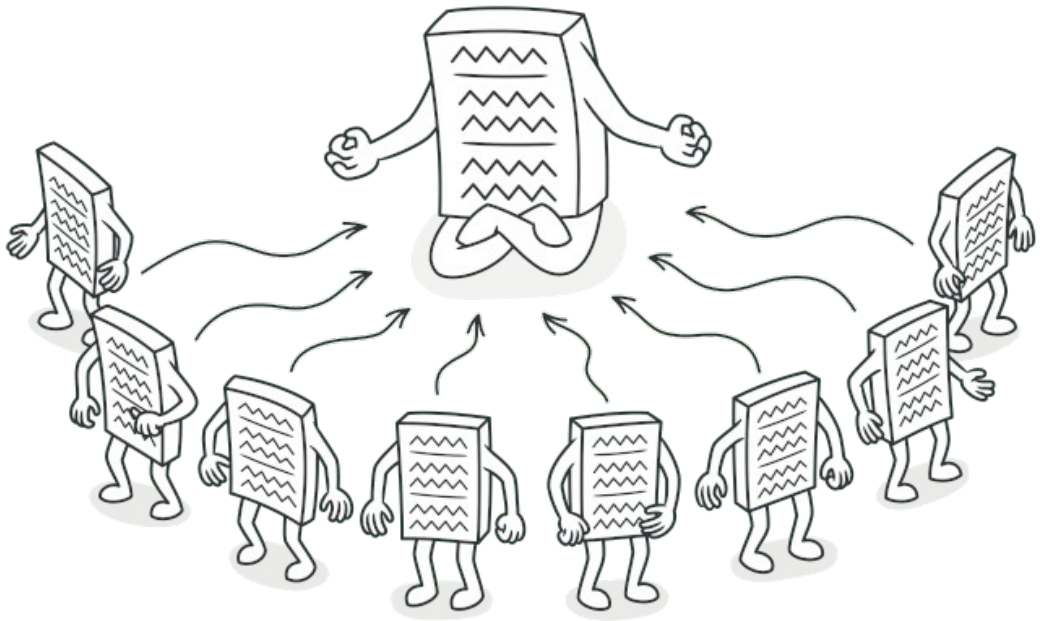


1 The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.

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.

I. Creational Design Patterns
5. Singleton pattern

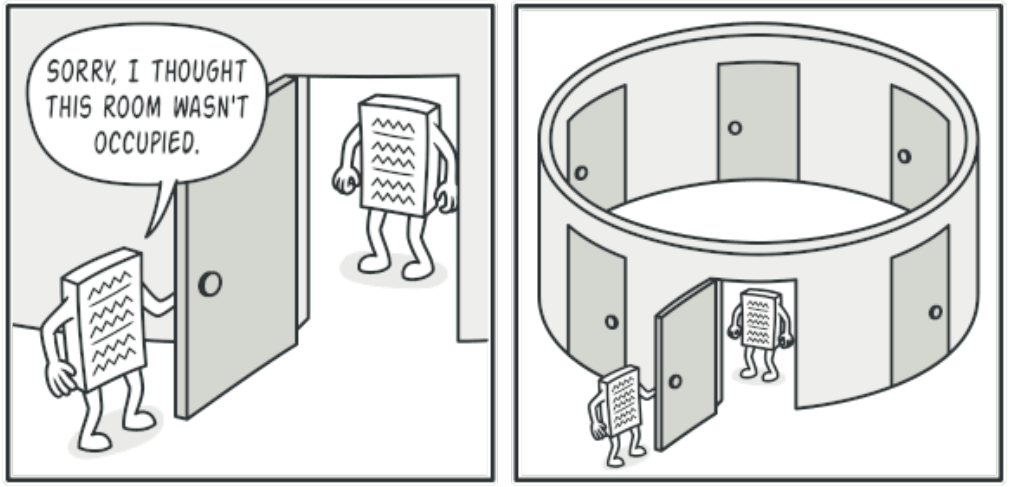
- **Singleton** is a creational design pattern that lets you *ensure that a class has only one instance, while providing a global access point to this instance*



I. Creational Design Patterns
5. Singleton pattern – Problem #1

- **Ensure that a class has just a single instance**. Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file

- Here's how it works: imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you'll get the one you already created
  - Note that this behavior is impossible to implement with a regular constructor since a constructor call must always return a new object by design



I. Creational Design Patterns
5. Singleton pattern – Problem #2

- **Provide a global access point to that instance**
  - *Remember those global variables that everyone used to store some essential objects? While they're very handy, they're also very unsafe since any code can potentially overwrite the contents of those variables and crash the app.*
  - Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.
  - There's another side to this problem: you don't want the code that solves problem #1 to be scattered all over your program. It's much better to have it within one class, especially if the rest of your code already depends on it

I. Creational Design Patterns
5. Singleton pattern – Solution

- All implementations of the Singleton have these two steps in common:
  1. Make the default constructor private, to prevent other objects from using the new operator with the Singleton class
  2. 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
- If your code has access to the Singleton class, then it’s able to call the Singleton’s static method. So whenever that method is called, the same object is always returned

I. Creational Design Patterns
5. Singleton pattern – Real-world analogy

- The government is an excellent example of the Singleton pattern
- A country can have only one official government
- Regardless of the personal identities of the individuals who form governments, the title, “The Government of X”, is a global point of access that identifies the group of people in charge

I. Creational Design Patterns
5. Singleton pattern – 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
  - you need stricter control over global variables

I. Creational Design Patterns
5. Singleton pattern – 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

I. Creational Design Patterns
5. Singleton pattern – Relations with Other Patterns

- A **Facade** class can often be transformed into a **Singleton** since a single facade object is sufficient in most cases
- **Flyweight** would resemble **Singleton** if you somehow managed to reduce all shared states of the objects to just one flyweight object. But there are two fundamental differences between these patterns:
  1. There should be only one Singleton instance, whereas a Flyweight class can have multiple instances with different intrinsic states
  2. The Singleton object can be mutable. Flyweight objects are immutable
- **Abstract Factories, Builders** and **Prototypes** can all be implemented as **Singletons**

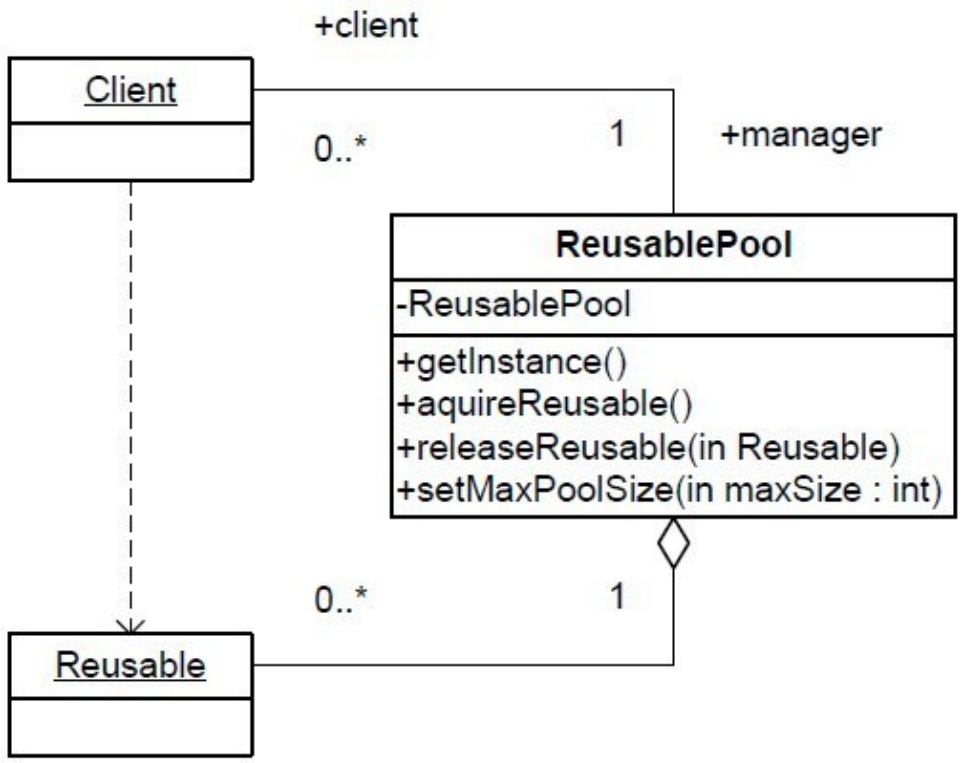
I. Creational Design Patterns
6. Lazy Initialization

See Singleton!

- In computer programming, **lazy initialization** is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. It is a kind of lazy evaluation that refers specifically to the instantiation of objects or other resources
- In a software design pattern view, lazy initialization is often used together with a factory method pattern. This combines three ideas:
  1. Using a factory method to create instances of a class (factory method pattern)
  2. Storing the instances in a map, and returning the same instance to each request for an instance with same parameters (multiton pattern)
  3. Using lazy initialization to instantiate the object the first time it is requested (lazy initialization pattern)

I. Creational Design Patterns
7. Object Pool

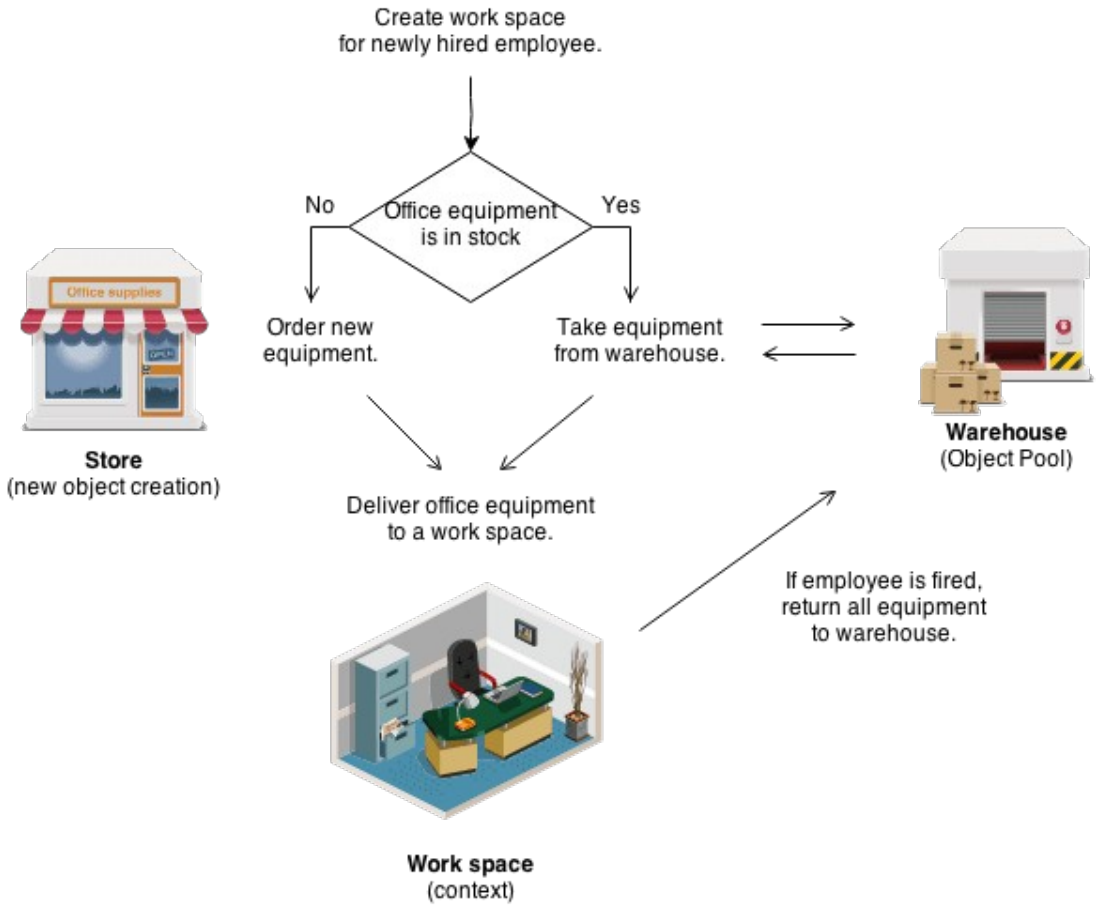
- Object pooling can offer a significant performance boost; it is most effective in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instantiations in use at any one time is low



I. Creational Design Patterns
7. Object Pool – Problem

- Object pools (otherwise known as resource pools) are used to manage the object caching. A client with access to an Object pool can avoid creating a new Objects by simply asking the pool for one that has already been instantiated instead
- Generally the pool will be a growing pool, i.e. the pool itself will create new objects if the pool is empty, or we can have a pool, which restricts the number of objects created.
- It is desirable to keep all Reusable objects that are not currently in use in the same object pool so that they can be managed by one coherent policy. To achieve this, the Reusable Pool class is designed to be a singleton class

I. Creational Design Patterns
7. Object Pool – Example



I. Creational Design Patterns
7. Object Pool – Relations with Other Patterns

- The **Factory Method** pattern can be used to encapsulate the creation logic for objects. However, it does not manage them after their creation, the **Object Pool** pattern keeps track of the objects it creates
- **Object Pools** are usually implemented as **Singletons**

I. Creational Design Patterns
8. RAII (Resource Acquisition Is Initialization)

- Constructor Acquires, Destructor Releases
- Scope-Based Resource Management
- Resource Acquisition Is Initialization or RAII, is a C++ programming technique which binds the life cycle of a resource that must be acquired before use (allocated heap memory, thread of execution, open socket, open file, locked mutex, disk space, database connection—anything that exists in limited supply) to the lifetime of an object

```cpp
void write_to_file (const std::string & message) {
    // mutex to protect file access (shared across threads)
    static std::mutex mutex;

    // lock mutex before accessing file
    std::lock_guard<std::mutex> lock(mutex);

    // try to open file
    std::ofstream file("example.txt");
    if (!file.is_open())
        throw std::runtime_error("unable to open file");

    // write message to file
    file << message << std::endl;

    // file will be closed 1st when leaving scope (regardless of exception)
    // mutex will be unlocked 2nd (from lock destructor) when leaving
    // scope (regardless of exception)
}
```

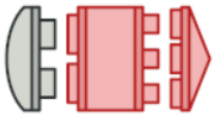
I. Creational Design Patterns
8. RAII (Resource Acquisition Is Initialization)

- RAII can be summarized as follows:
  - encapsulate each resource into a class, where
    - the *constructor* acquires the resource and establishes all class invariants or throws an exception if that cannot be done,
    - the *destructor* releases the resource and never throws exceptions;
  - always use the resource via an instance of a RAII-class that either
    - has automatic storage duration or temporary lifetime itself, *or*
    - has lifetime that is bounded by the lifetime of an automatic or temporary object

II. Structural Design Patterns

explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient

II. Structural Design Patterns



Adapter

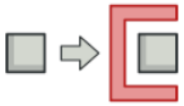
Allows objects with incompatible interfaces to collaborate.

Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

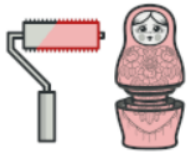
Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.



Proxy

Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

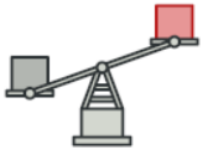


Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.

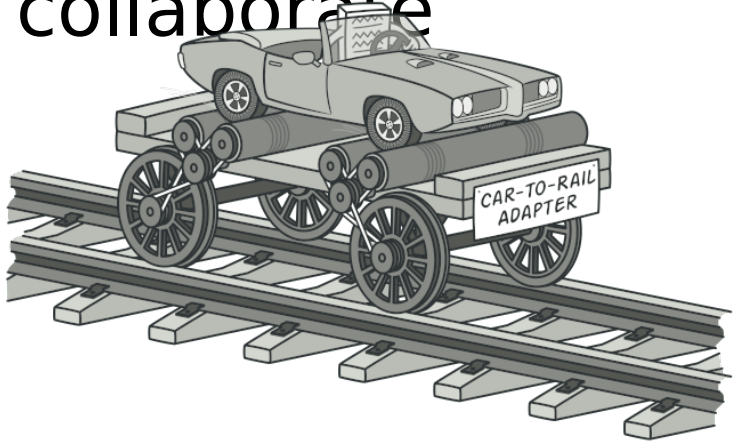


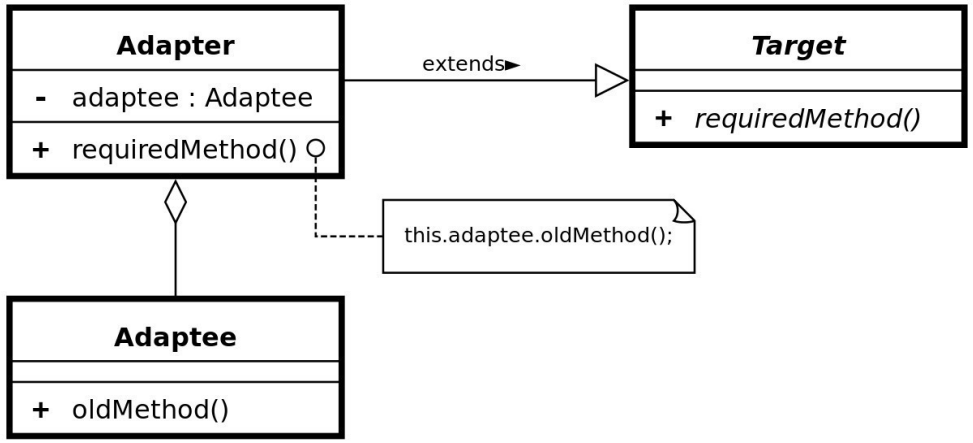
Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

II. Structural Design Patterns
1. Adapter (also known as Wrapper or Translator)

- **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate

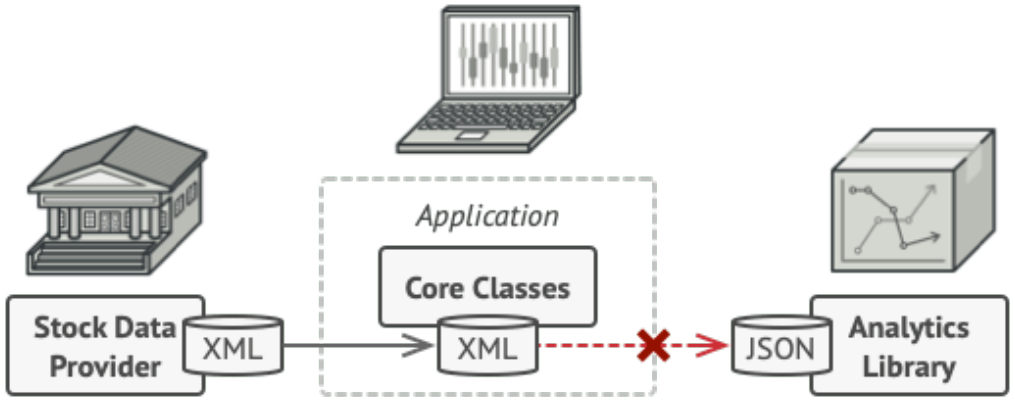




II. Structural Design Patterns
1. Adapter – Problem

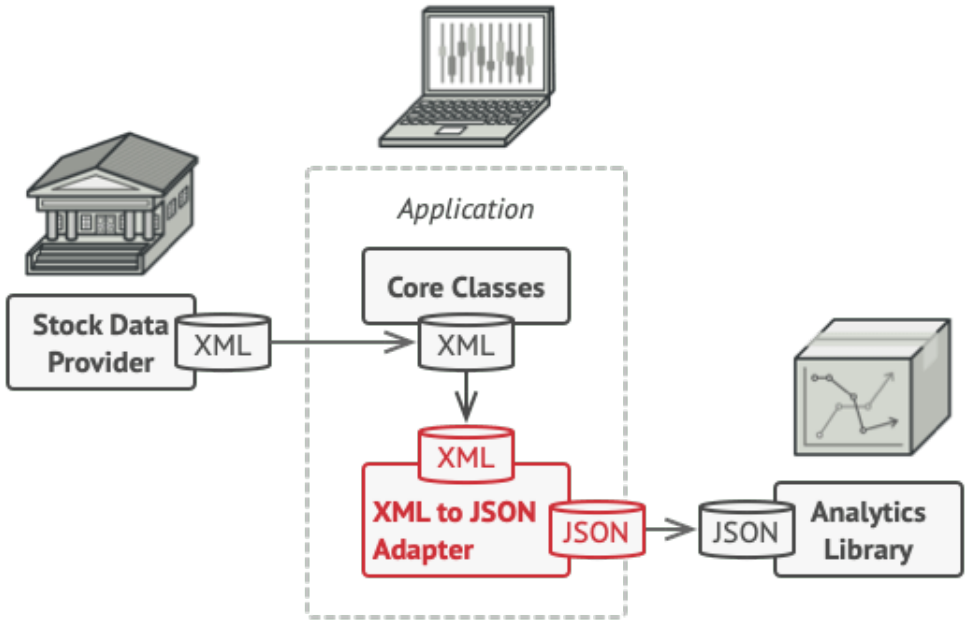
- Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user
- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format

You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app



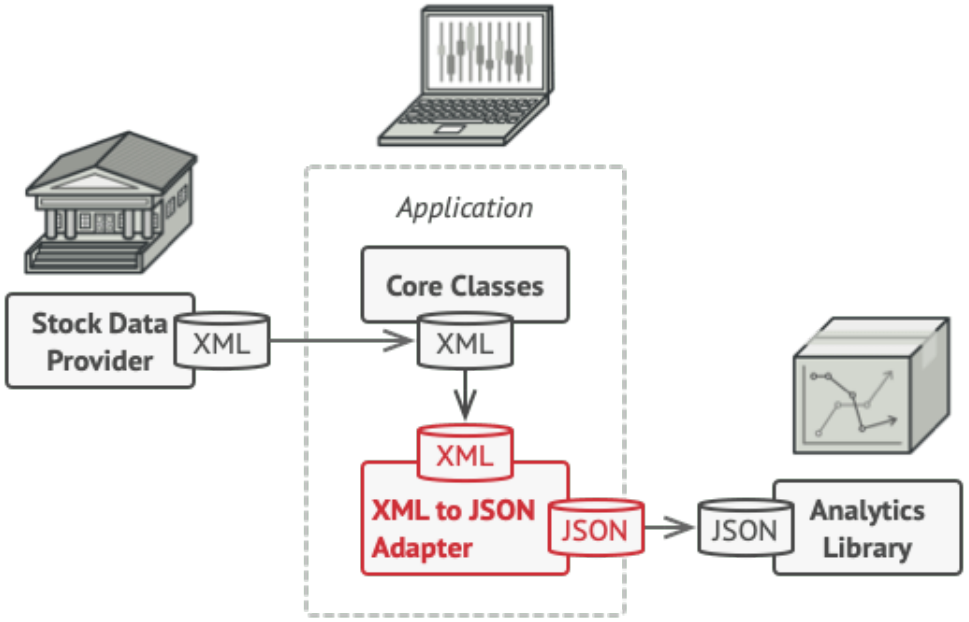
II. Structural Design Patterns
1. Adapter – Solution

- You can create an **adapter**. This is a special object that converts the interface of one object so that another object can understand it.
- An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter. For example, you can wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles

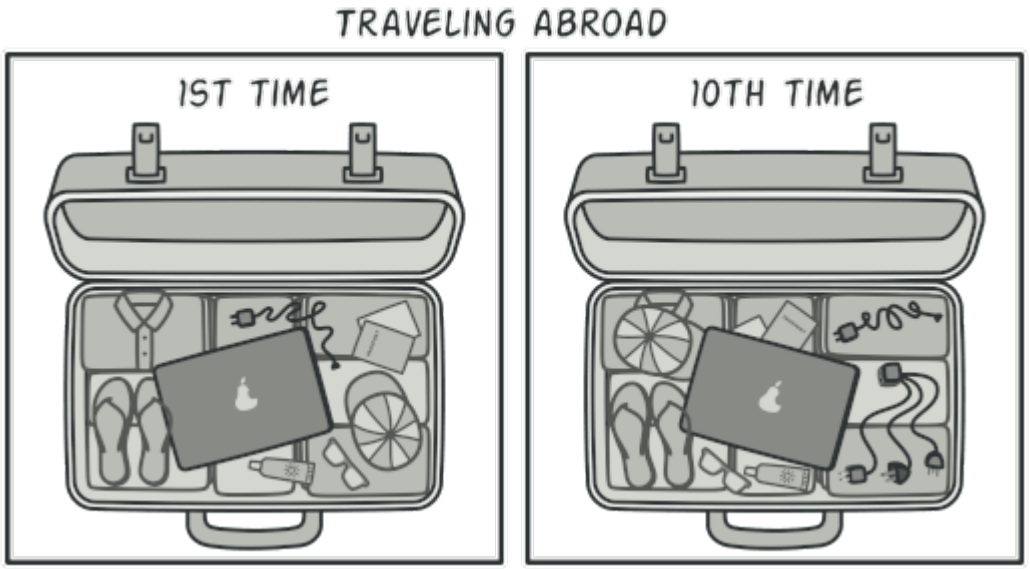


II. Structural Design Patterns
1. Adapter – Solution

- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:
  1. The adapter gets an interface, compatible with one of the existing objects.
  2. Using this interface, the existing object can safely call the adapter's methods.
  3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.
- Sometimes it's even possible to create a two-way adapter that can convert the calls in both directions



II. Structural Design Patterns
1. Adapter – Real-world analogy



II. Structural Design Patterns
1. Adapter – Applicability

- Use the Adapter pattern when you want to…
  - use some existing class, but its interface isn’t compatible with the rest of your code
  - reuse several existing subclasses that lack some common functionality that can’t be added to the superclass

II. Structural Design Patterns
1. Adapter – Pros and Cons

- *Single Responsibility Principle*. You can separate the interface or data conversion code from the primary business logic of the program
- *Open/Closed Principle*. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface

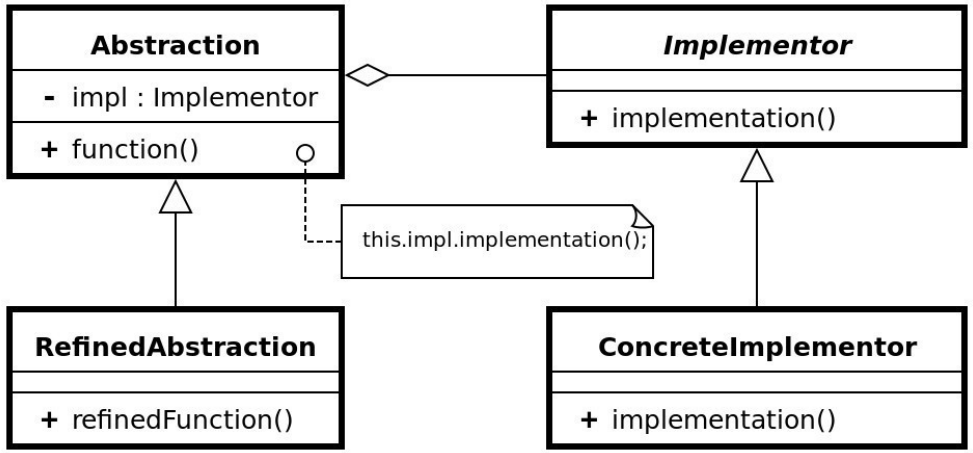
- The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it’s simpler just to change the service class so that it matches the rest of your code

II. Structural Design Patterns
1. Adapter – Relations with Other Patterns

- **Bridge** is usually designed up-front, letting you develop parts of an application independently of each other. On the other hand, **Adapter** is commonly used with an existing app to make some otherwise-incompatible classes work together nicely
- **Adapter** changes the interface of an existing object, while **Decorator** enhances an object without changing its interface. In addition, *Decorator* supports recursive composition, which isn't possible when you use *Adapter*
- **Adapter** provides a different interface to the wrapped object, **Proxy** provides it with the same interface, and **Decorator** provides it with an enhanced interface
- **Facade** defines a new interface for existing objects, whereas **Adapter** tries to make the existing interface usable. Adapter usually wraps just one object, while *Facade* works with an entire subsystem of objects
- **Bridge, State, Strategy** (and to some degree **Adapter**) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn't just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves

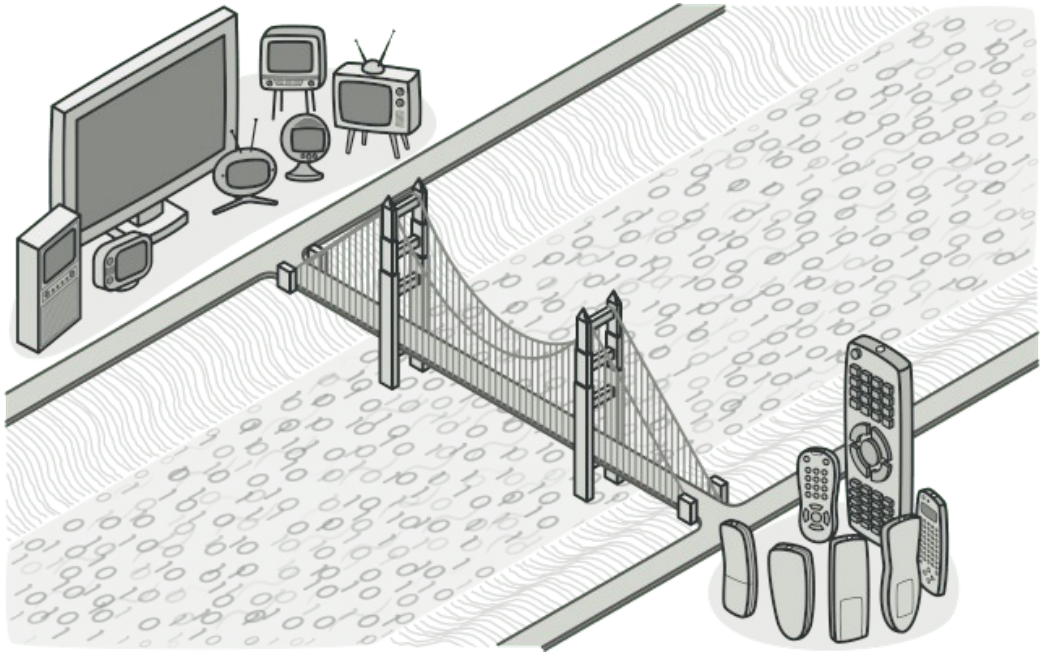
II. Structural Design Patterns
2. Bridge

- Decouple abstraction and implementation to make them independent
- Pimpl idiom

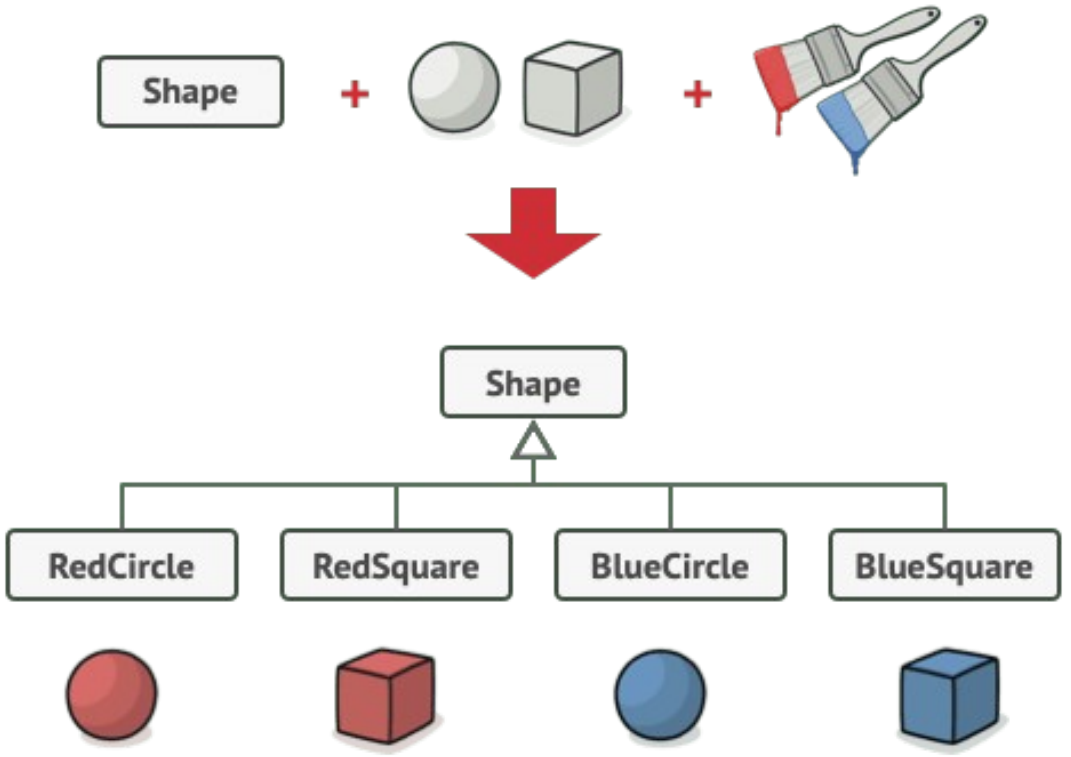


II. Structural Design Patterns
2. Bridge

- **Bridge** is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies — abstraction and implementation — which can be developed independently of each other

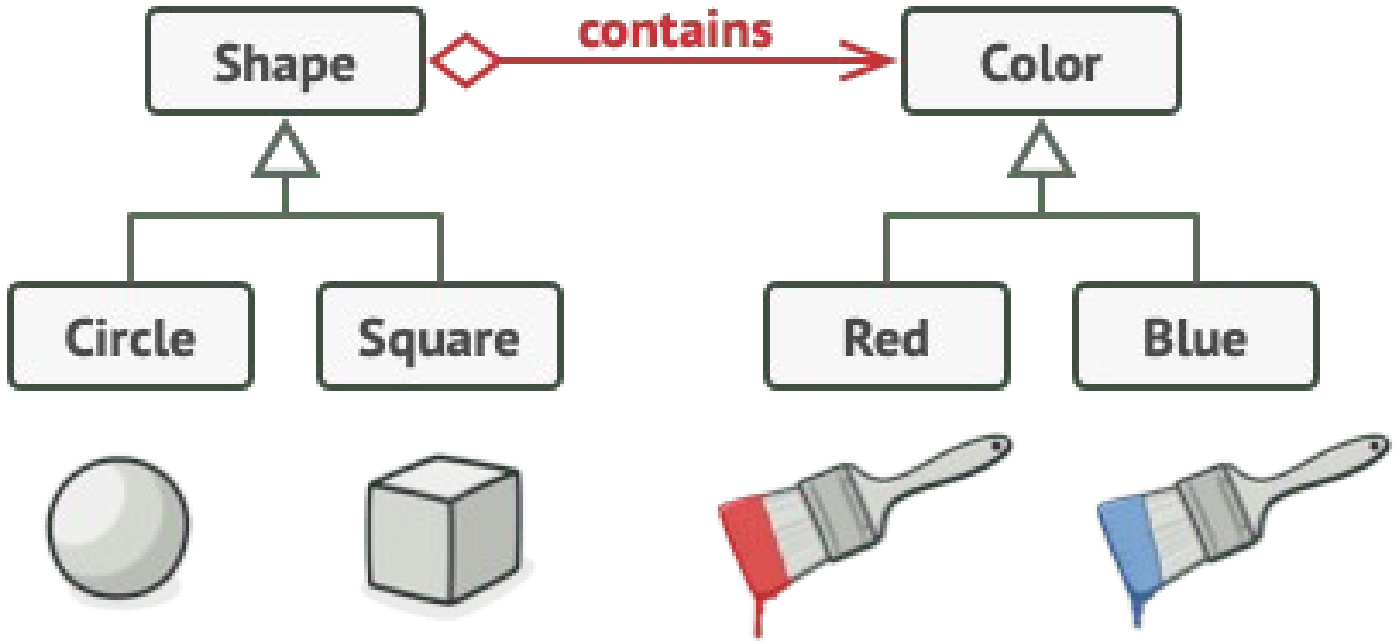


II. Structural Design Patterns
2. Bridge – Problem



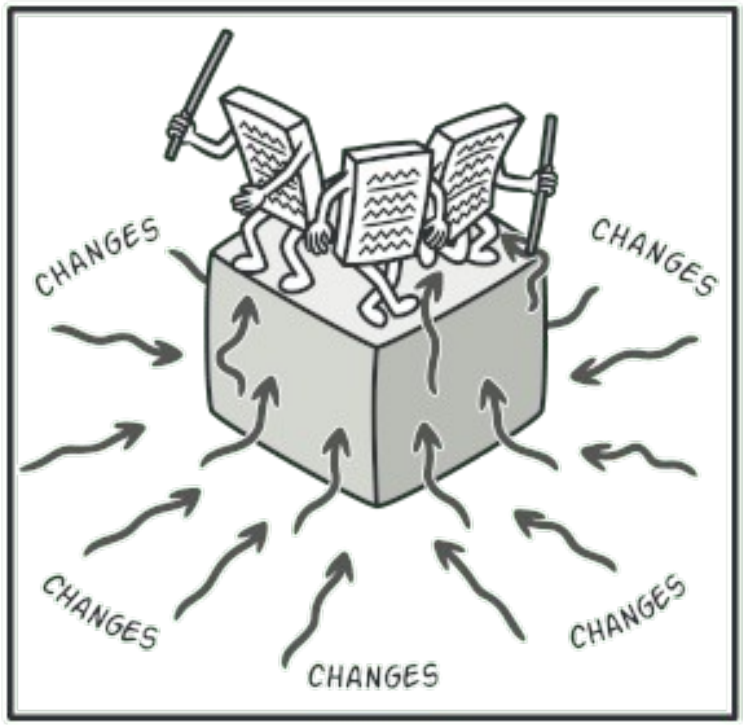
Number of class combinations grows in geometric progression

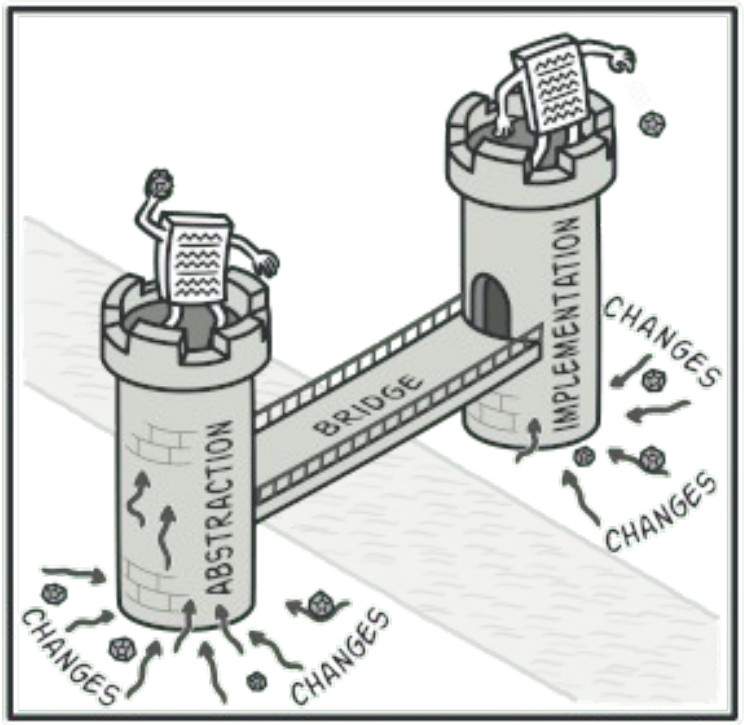
II. Structural Design Patterns
2. Bridge – Solution



You can prevent the explosion of a class hierarchy by transforming it into several related hierarchies

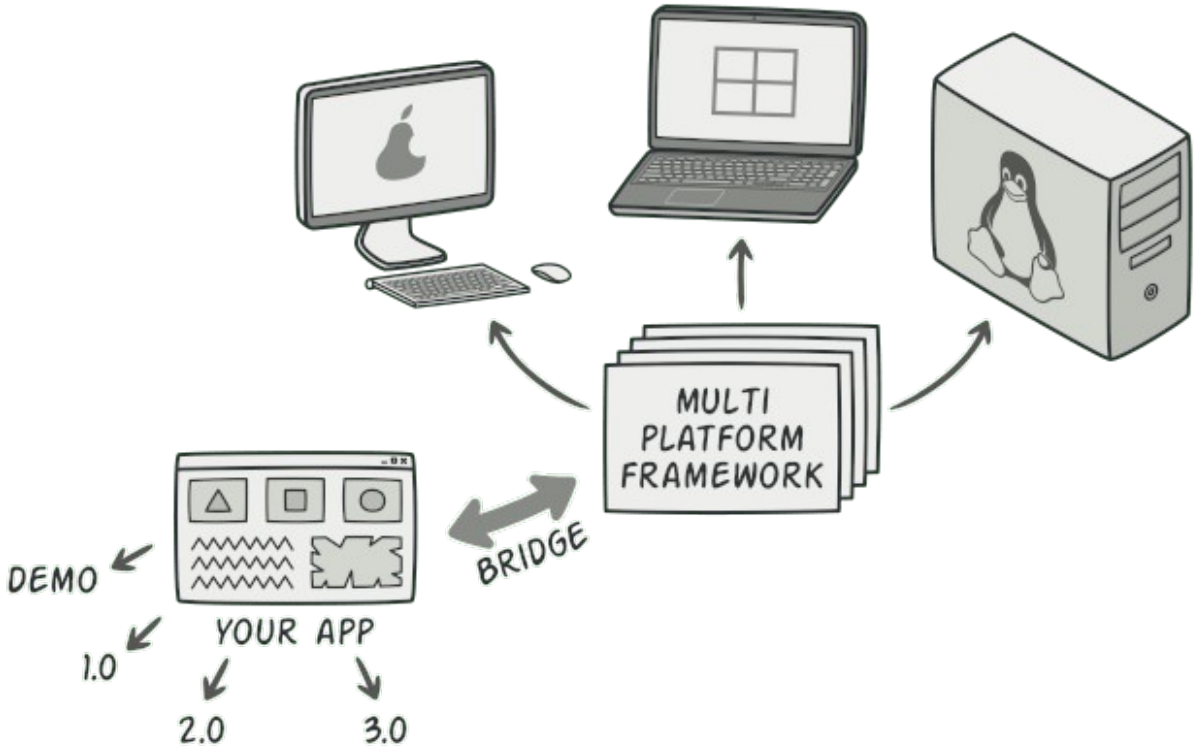
II. Structural Design Patterns
2. Bridge – Solution





Making even a simple change to a monolithic codebase is pretty hard because you must understand the entire thing very well.
Making changes to smaller, well-defined modules is much easier

II. Structural Design Patterns
2. Bridge – Solution



One of the ways to structure a cross-platform application

II. Structural Design Patterns
2. Bridge – Applicability

- Use the Bridge pattern when you …
  - want to divide and organize a monolithic class that has several variants of some functionality (for example, if the class can work with various database servers)
  - need to extend a class in several orthogonal (independent) dimensions
- Use the Bridge if you need to be able to switch implementations at runtime

II. Structural Design Patterns
2. Bridge – Pros and Cons

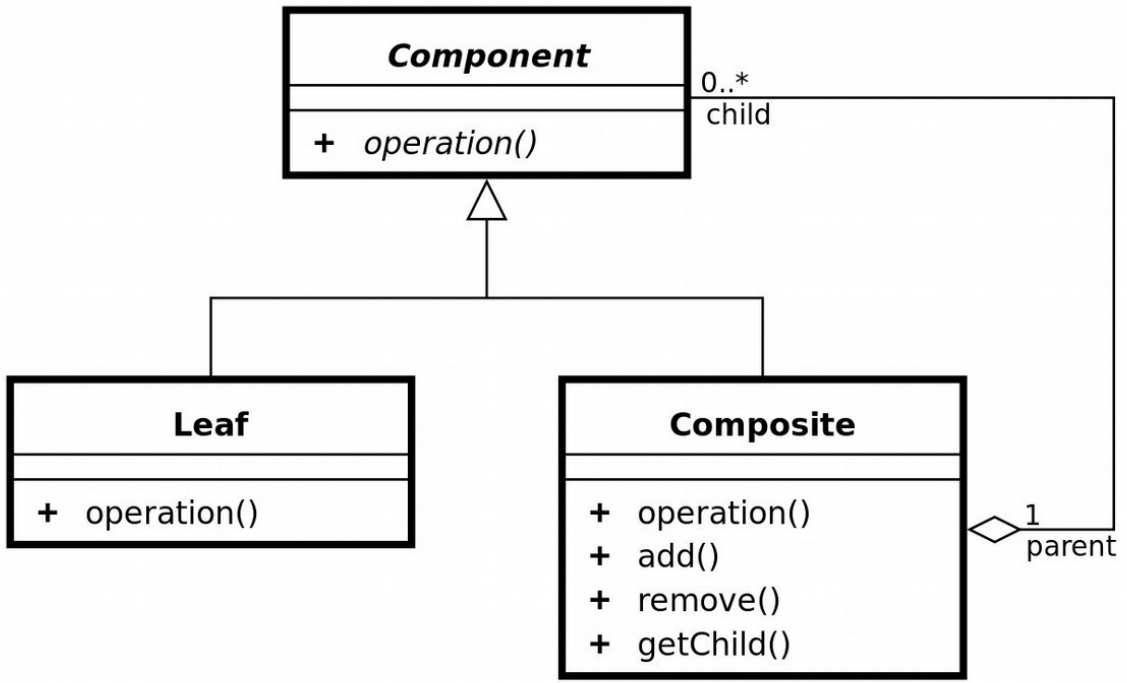
- You can create platform-independent classes and apps
- The client code works with high-level abstractions. It isn’t exposed to the platform details
- *Single Responsibility Principle*. You can separate the interface or data conversion code from the primary business logic of the program
- *Open/Closed Principle*. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface

- You might make the code more complicated by applying the pattern to a highly cohesive class

II. Structural Design Patterns
2. Bridge – Relations with Other Patterns

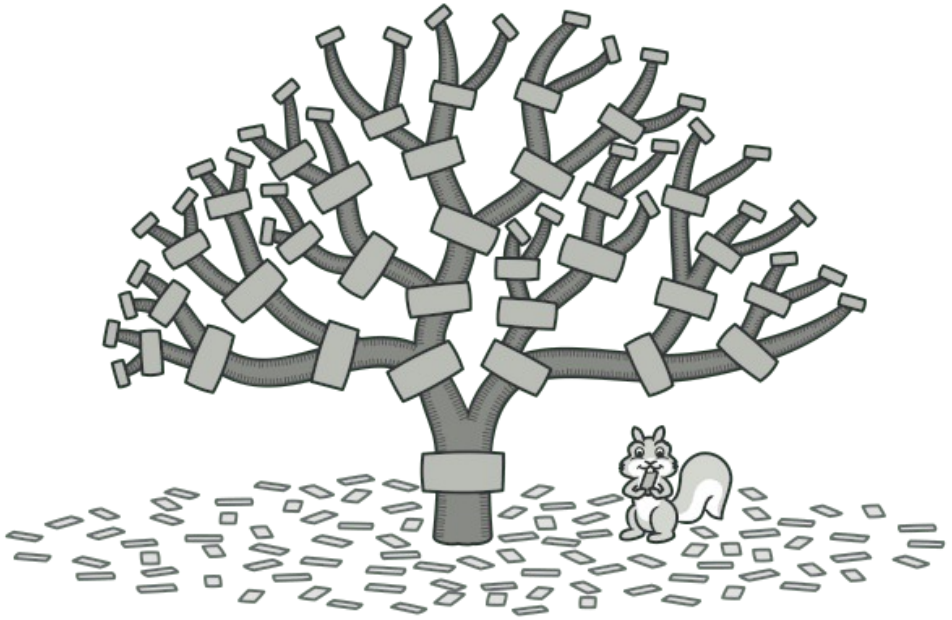
- **Bridge** is usually designed up-front, letting you develop parts of an application independently of each other. On the other hand, **Adapter** is commonly used with an existing app to make some otherwise-incompatible classes work together nicely
- **Bridge, State, Strategy** (and to some degree **Adapter**) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn't just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves
- You can use **Abstract Factory** along with **Bridge**. This pairing is useful when some abstractions defined by *Bridge* can only work with specific implementations. In this case, *Abstract Factory* can encapsulate these relations and hide the complexity from the client code
- You can combine **Builder** with **Bridge**: the director class plays the role of the abstraction, while different builders act as implementations

II. Structural Design Patterns
3. Composite

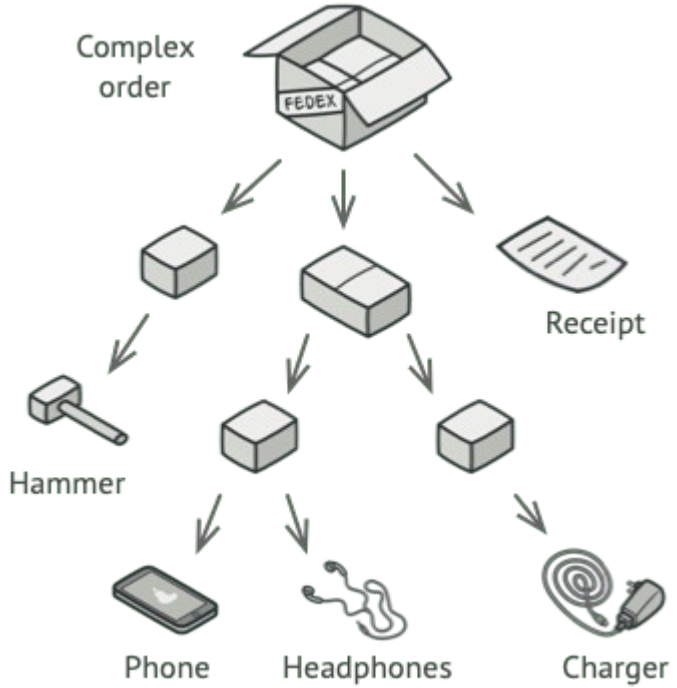


II. Structural Design Patterns
3. Composite (Also known as: Object Tree)

- **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

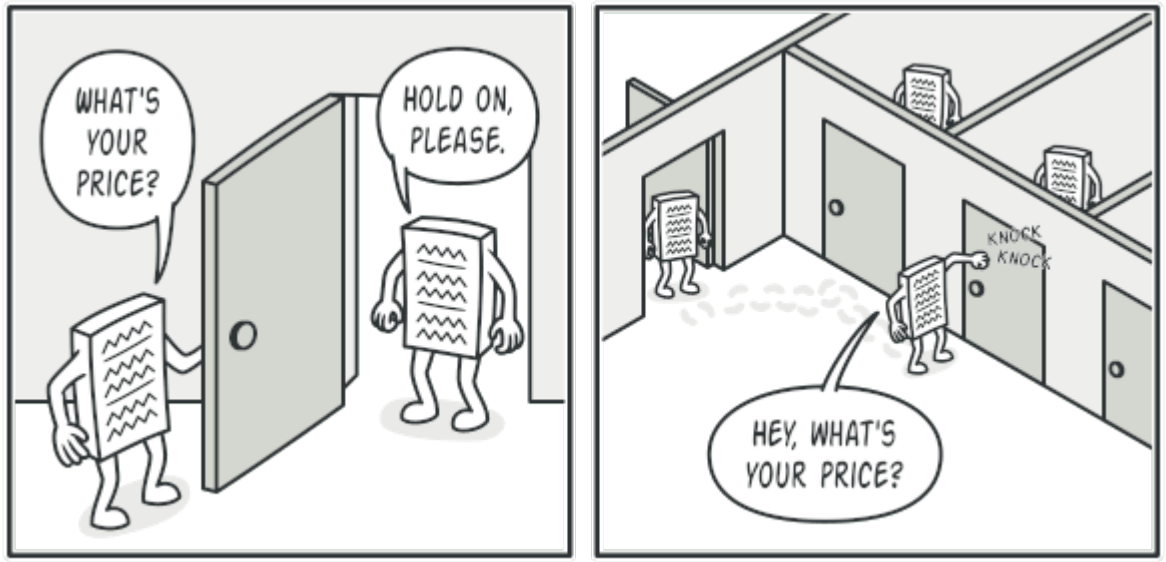


II. Structural Design Patterns
3. Composite – Problem



n order might comprise various products, packaged in boxes, which are packaged in bigger boxes and so o
The whole structure looks like an upside-down tree

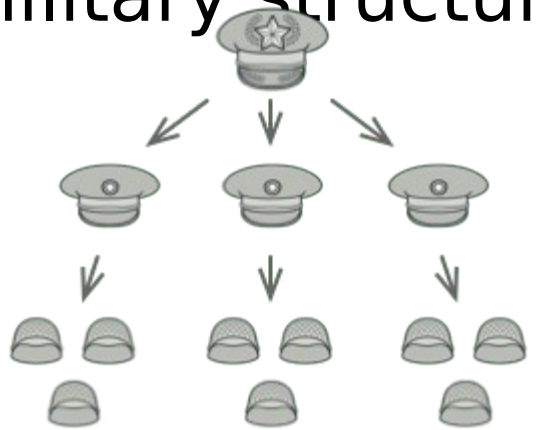
II. Structural Design Patterns
3. Composite – Solution



The Composite pattern lets you run a behavior recursively over all components of an object tree

II. Structural Design Patterns
3. Composite – Real-world analogy

An example of a military structure



Armies of most countries are structured as hierarchies. An army consists of several divisions; a division is a set of brigades, and a brigade consists of platoons, which can be broken down into squads. Finally, a squad is a small group of real soldiers. Orders are given at the top of the hierarchy and passed down onto each level until every soldier knows what needs to be done

II. Structural Design Patterns
3. Composite – Applicability

- Use the Composite pattern when you …
  - have to implement a tree-like object structure
  - want the client code to treat both simple and complex elements uniformly

II. Structural Design Patterns
3. Composite – Pros and 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

II. Structural Design Patterns
3. Composite – Relations with Other Patterns

- You can use **Builder** when creating complex **Composite** trees because you can program its construction steps to work recursively
- **Chain of Responsibility** is often used in conjunction with **Composite**. In this case, when a leaf component gets a request, it may pass it through the chain of all of the parent components down to the root of the object tree
- You can use **Iterators** to traverse **Composite** trees
- You can use **Visitor** to execute an operation over an entire **Composite** tree
- You can implement shared leaf nodes of the **Composite** tree as **Flyweights** to save some RAM
- **Composite** and **Decorator** have similar structure diagrams since both rely on recursive composition to organize an open-ended number of objects
  - A Decorator is like a Composite but only has one child component. There's another significant difference: Decorator adds additional responsibilities to the wrapped object, while Composite just "sums up" its children's results
  - However, the patterns can also cooperate: you can use Decorator to extend the behavior of a specific object in the Composite tree
- Designs that make heavy use of **Composite** and **Decorator** can often benefit from using **Prototype**. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch

III. Behavioral Design Patterns

take care of effective communication and the assignment of responsibilities between objects

Questions?

- …
- Or write me an email to gla@inf.elte.hu

System Architecture

Questions?

- …
- Or write me an email to gla@inf.elte.hu