Software Design Patterns

Lecture 4

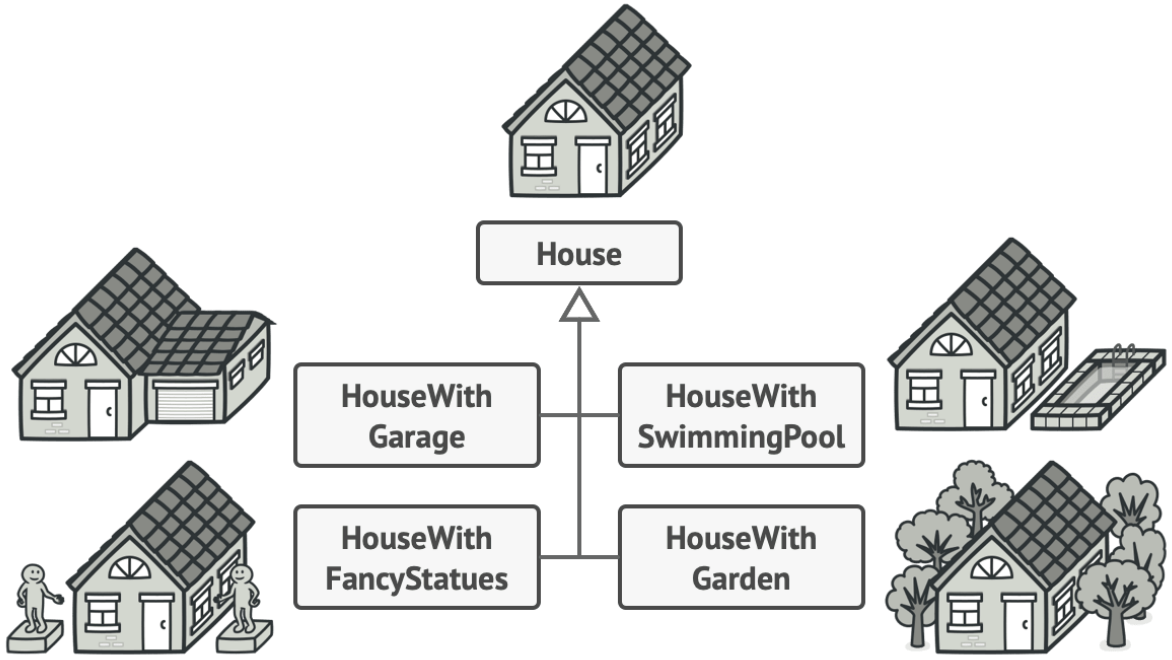
Builder
Prototype

Dr. Fan Hongfei

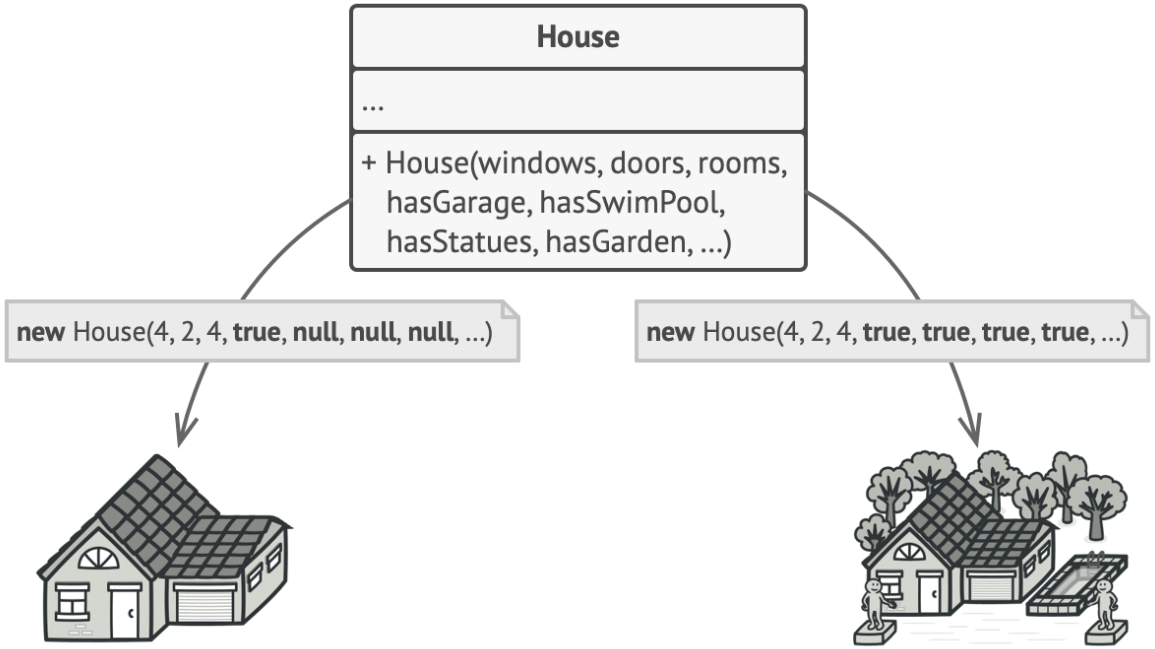
26 September 2024

Builder: Problem

- **Example:** a complex object that requires laborious, step-by-step initialization of many fields and nested objects



Problem: a large number of subclasses

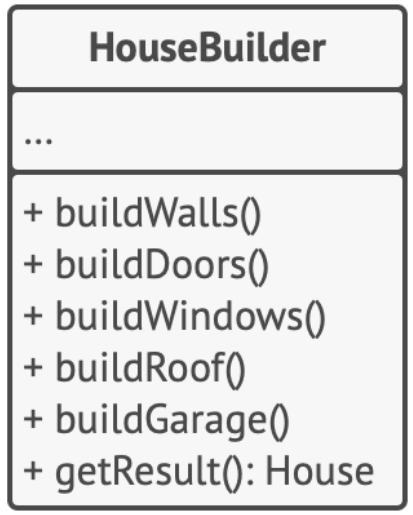


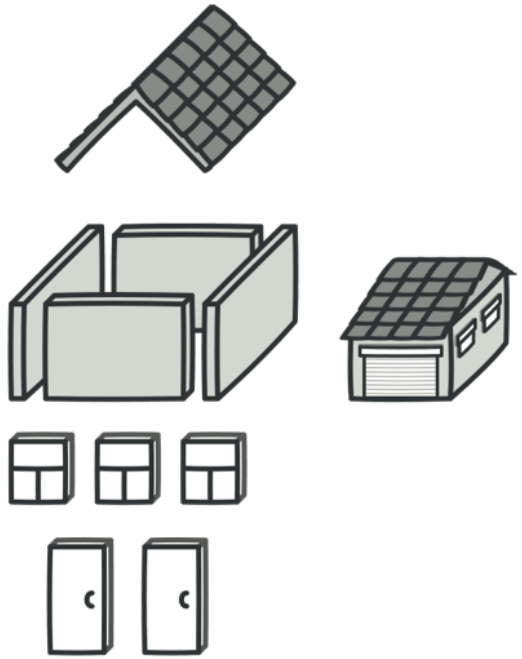
Problem: in most cases most of the parameters will be unused

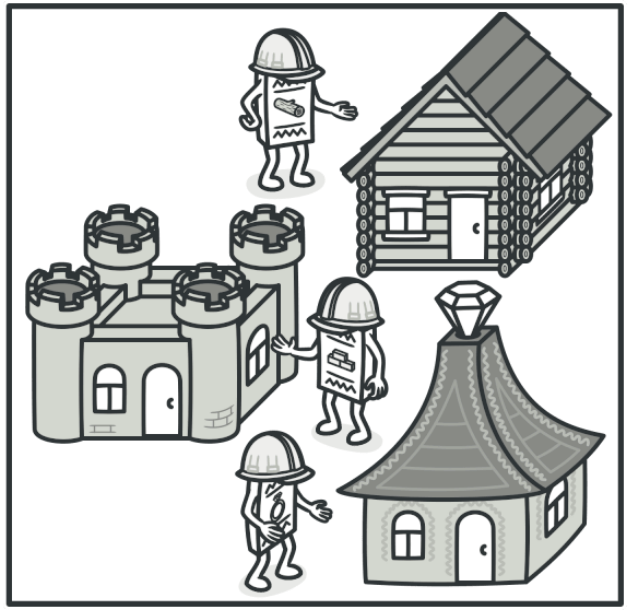
[Figures in this slide are extracted from https://refactoring.guru/design-patterns/builder]

Builder: Solution

- Extract the object construction code out of its own class and move it to separate objects called **builders**
  - The builder does not allow other objects to access the product while it is being built
  - Object construction is organized into a set of steps, and call only those steps that are necessary
  - Some construction steps might require different implementation: create different builder classes







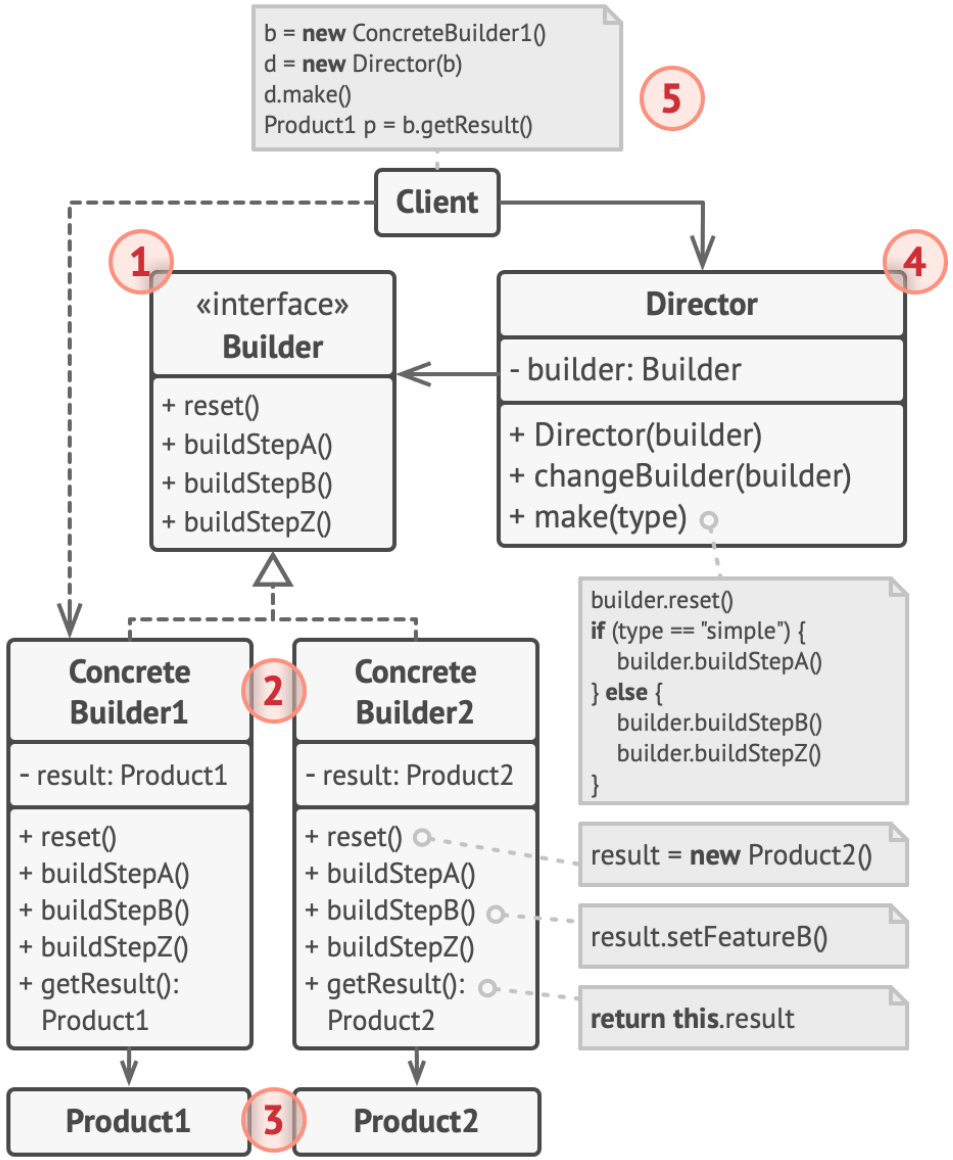
[Figures in this slide are extracted from https://refactoring.guru/design-patterns/builder]

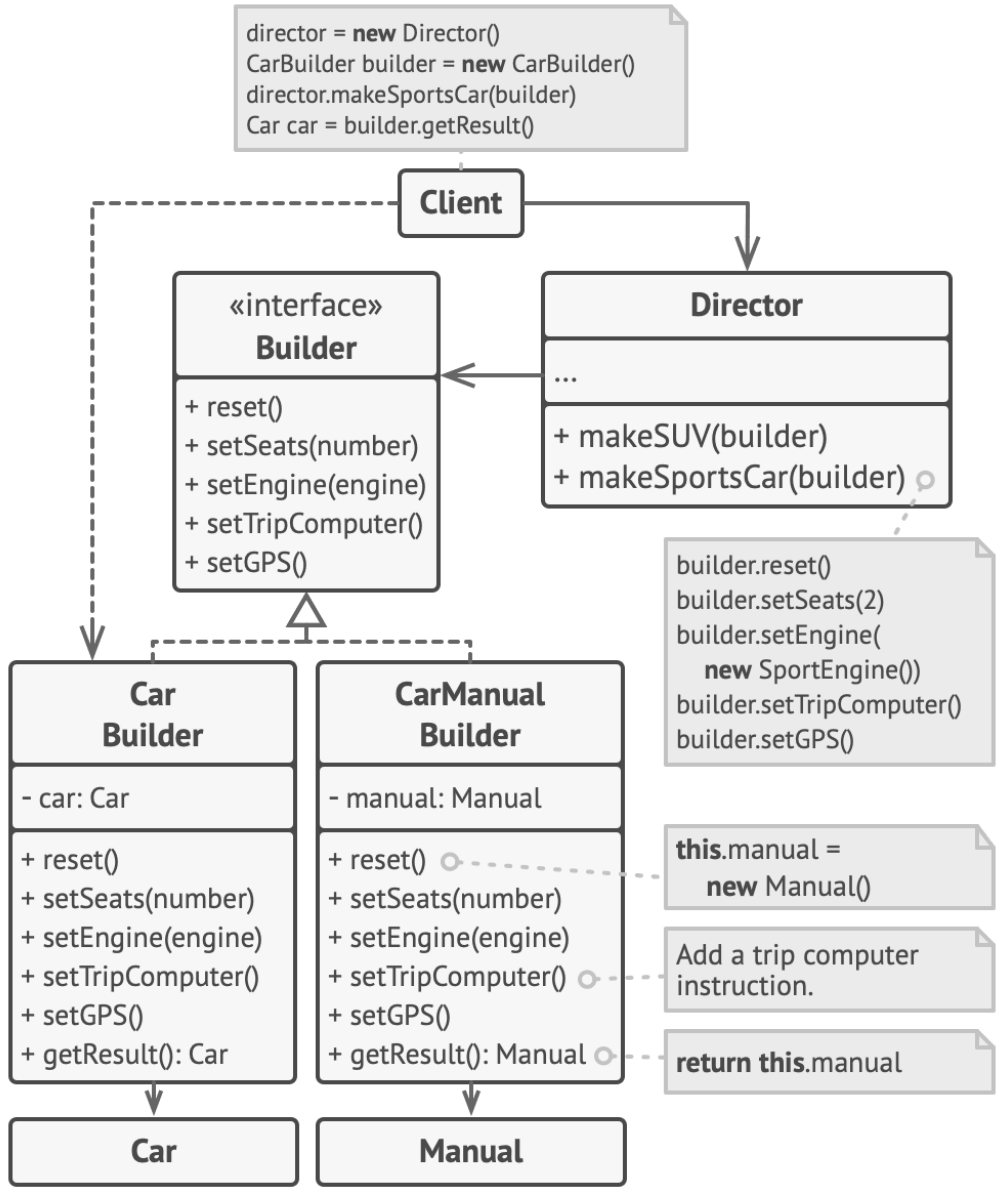
Builder: Solution (cont.)

- Further extract the series of calls to the builder steps into a separate class called **director**
  - The director defines **the order** in which to execute the building steps, while the builder provides concrete implementation
  - Having a director class is **not strictly necessary**
  - Might be a good place to put various construction routines for reuse
  - Completely hiding the details of construction from the client code

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/builder]

Builder: Structure and Example





[Figures in this slide are extracted from https://refactoring.guru/design-patterns/builder]

Builder: Applicability

- Get rid of a "telescoping constructor"
  - Build objects step by step, using only those steps that are needed
- When you want your code to be able to create different representations of some products
  - Applied when construction of various representations of the product involves similar steps that differ only in the details
- Construct Composite trees or other complex objects
  - A builder does not expose the unfinished product while running construction steps

Builder: Implementation

1. Clearly define the **common construction steps** for building all available product representations
2. Declare these steps in the base **builder interface**
3. Create a **concrete builder class** for each of the product representations and implement their construction steps, and also implement **a method for fetching the result of construction**
4. Consider creating a **director class** (not necessarily)
5. The client code creates both the builder and the director objects
6. The construction result can be obtained directly from the director only if all products follow the same interface; otherwise, the client should fetch the result from the builder

Builder: Pros and Cons

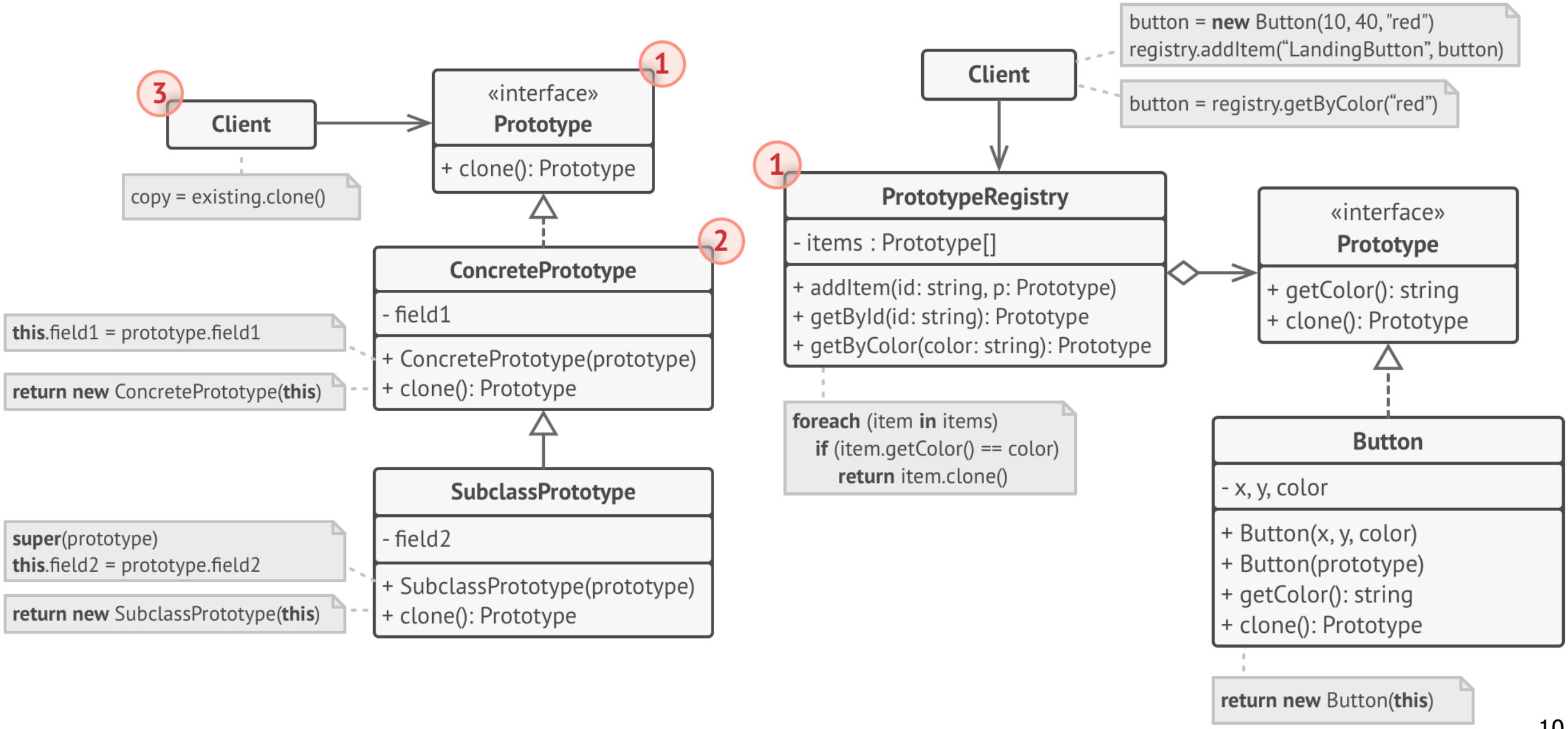
- **Pros**
  - Construct objects step-by-step, defer construction steps or run steps recursively
  - Reuse the same construction code when building various representations of products
  - Single Responsibility Principle
- **Cons**
  - The overall complexity of the code increases since the pattern requires creating multiple new classes

Prototype: Problem and Solution

- **Problem: creating an exact copy of an object**
  - Some fields may not be visible from the outside
  - The code becomes dependent on that class
  - Sometimes you only know the interface but not the concrete class
- **Solution: the Prototype pattern**
  - Delegating the cloning process to the actual objects being cloned
  - A common interface for all objects supporting cloning, with a single ***clone* method**
  - An object supporting cloning is called a **prototype**

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/prototype]

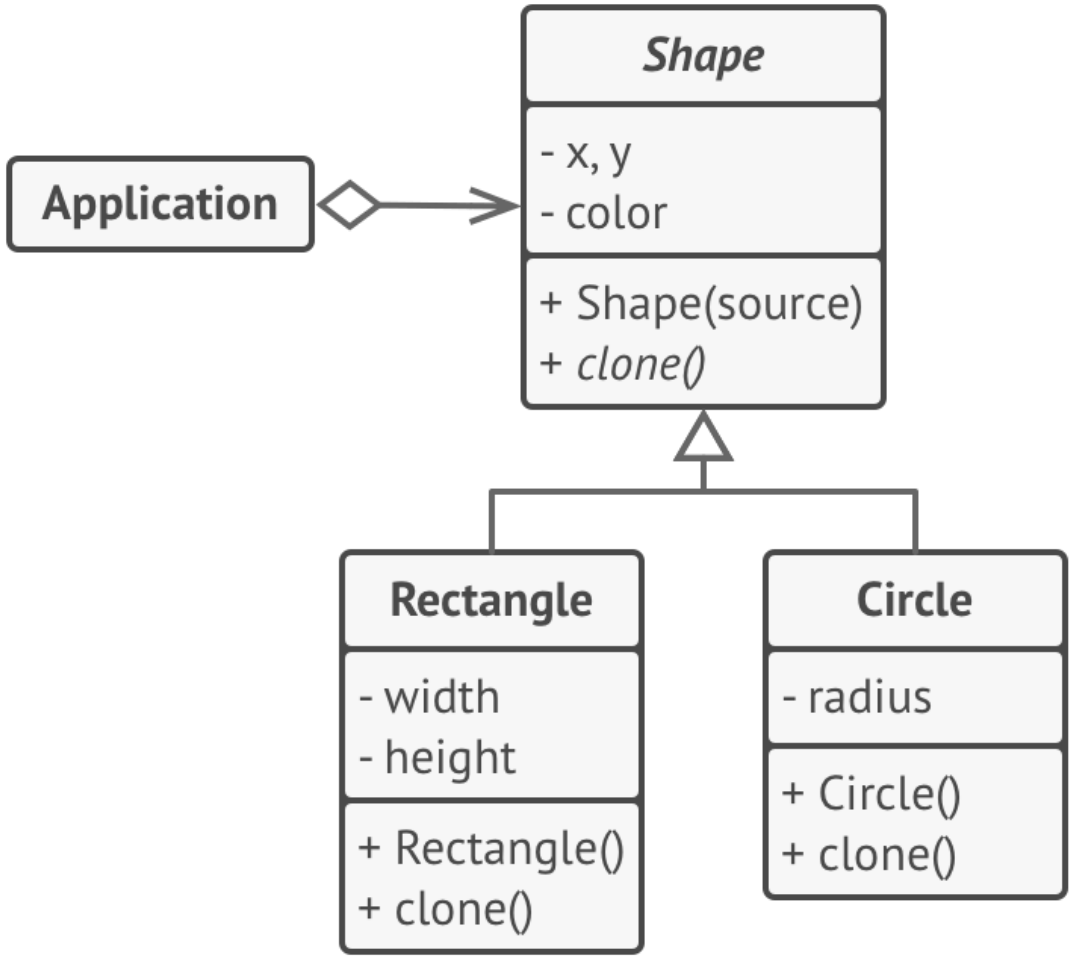
Prototype: Structure



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/prototype]

Prototype: Example

- Produce exact copies of geometric objects, without coupling the code to their classes



- *Note: a subclass may call the parent's cloning method before copying its own field values to the resulting object.*

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/prototype]

Prototype: Applicability

- When your code should not depend on the concrete classes of objects that you need to copy
  - Your code works with objects passed to you from 3rd-party code via some interfaces, while the concrete classes are unknown
  - This interface makes the client code independent from the concrete classes of objects that it clones
- When you want to reduce the number of subclasses that only differ in the way they initialize their respective objects
  - Use a set of pre-built objects configured in various ways as prototypes
  - Instead of instantiating a subclass, the client can simply look for an appropriate prototype and clone it

Prototype: Implementation

1. Create the **prototype interface** and declare the **clone method**, or just add the method to all classes of an existing class hierarchy
2. A prototype class must **define the alternative constructor** that accepts an object of that class
   - Call the parent constructor in the subclass
3. Implement the clone method by using the **new operator**
4. Optionally, create a **centralized prototype registry** to store a catalog of frequently used prototypes
   - Either as a new factory class, or in the base prototype class with a static method

Prototype: Pros and Cons

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