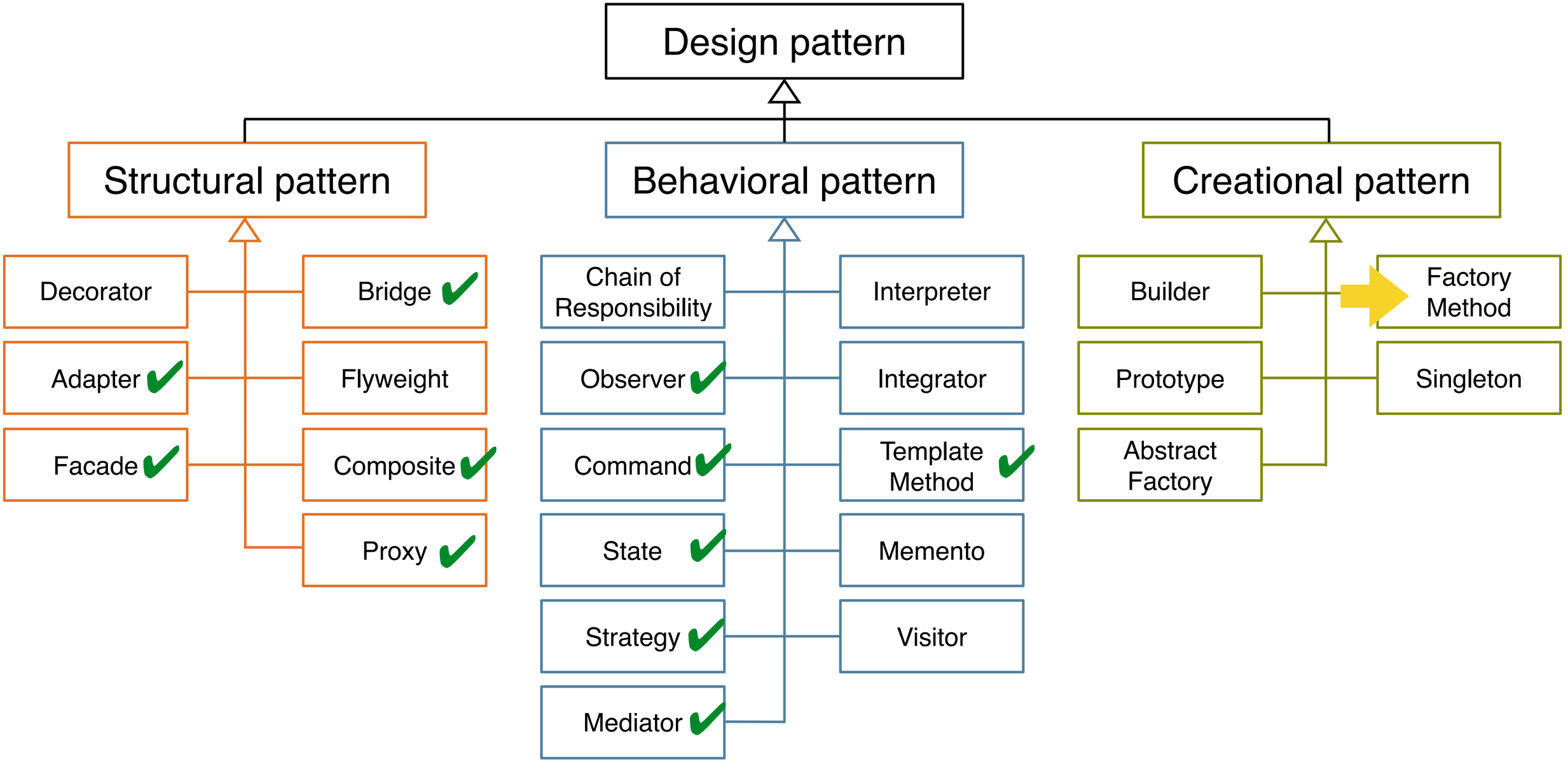
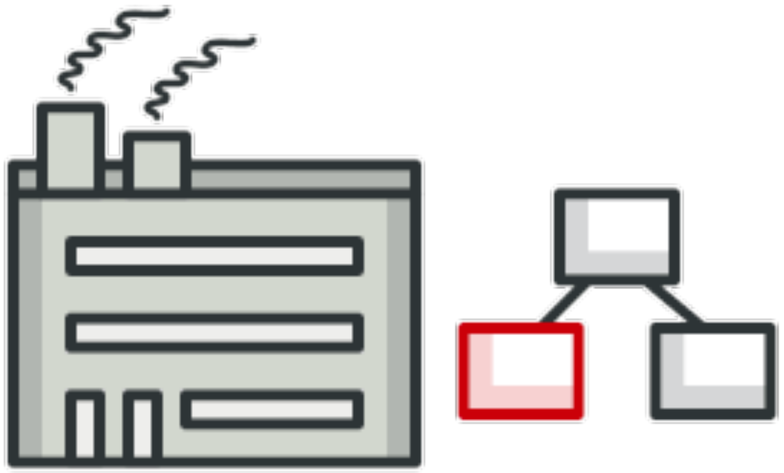
# Outline

- Factory method pattern
- Abstract factory pattern
- Flyweight pattern
- Builder pattern

# Design pattern overview



# Factory method pattern



# Problem

- Instantiating an object with the **`new`** operator creates a dependency on a concrete class, which means **high coupling**

➡ The **`new`** operator is considered **harmful**

# Motivation

- Object oriented design goals: **low coupling** and **high cohesion**

  ➡ Program against super classes / interfaces rather than against concrete implementations

- How can we instantiate new objects without depending on concrete implementations?
- Use a factory class: a class responsible for the instantiation of objects

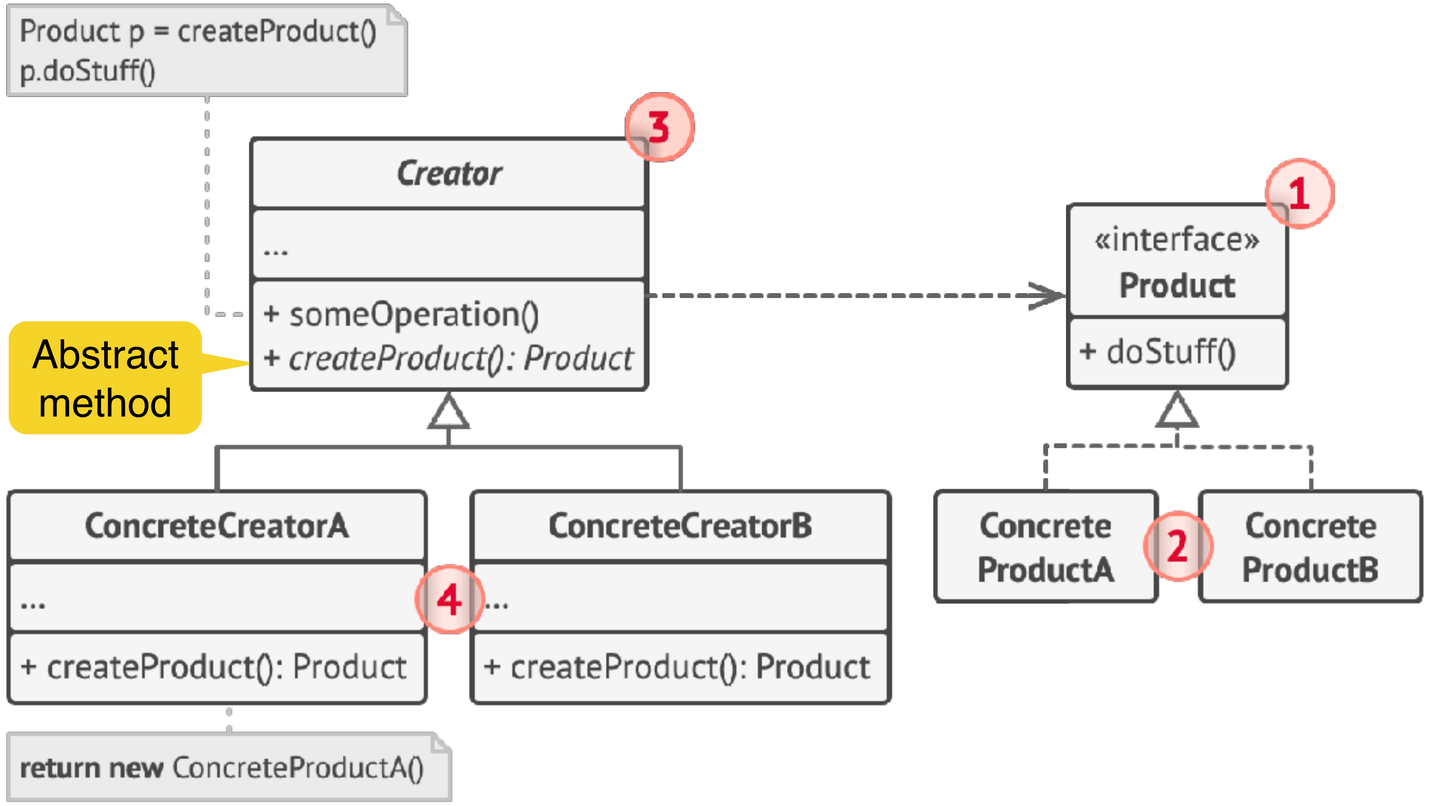
  ➡ Define an interface for creating an object, but let subclasses decide which class to instantiate (defer instantiation to subclasses)

  ➡ Defining a virtual constructor

  ➡ First step towards dependency injection

# Structure

1. **`Product`** declares the interface, which is common to all objects produced by the creator and its subclasses
2. **`ConcreteProduct`**s are different implementations of the product interface
3. **`Creator`** declares the factory method that returns **`new`** product objects
4. **`ConcreteCreator`**s override the factory method, so it returns a different type of product



# Benefits

- There is a difference between requesting an object and creating one
- The new operator always creates an object, and fails to encapsulate object creation
- A factory method enforces encapsulation, and allows an object to be requested without strong coupling to the act of creation
- Improved extensibility when adding new objects

# Example

```
public class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza = null;
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        pizza.bake();
        return pizza;
    }
}
```

- Problem: creating a pizza is coupled with baking the pizza
- How can we minimize the coupling?

# Example: decoupling by use of a factory class

```
public class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza = null;
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        pizza.bake();
        return pizza;
    }
}
```

```
public class PizzaStore {
    PizzaFactory factory;
    public PizzaStore(PizzaFactory factory) {
        this.factory = factory;
    }
    public Pizza orderPizza(String type) {
        Pizza pizza = factory.createPizza(type);
        pizza.bake();
        return pizza;
    }
}
public class PizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

# Example: UML model for the pizza factory

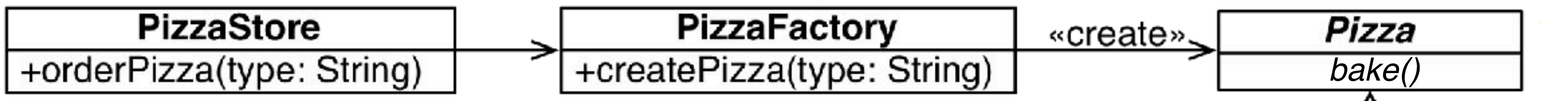
```java
public class PizzaStore {
    PizzaFactory factory;
    public PizzaStore(PizzaFactory factory) {
        this.factory = factory;
    }
    public Pizza orderPizza(String type) {
        Pizza pizza = factory.createPizza(type);
        pizza.bake();
        return pizza;
    }
}
```

Delegation

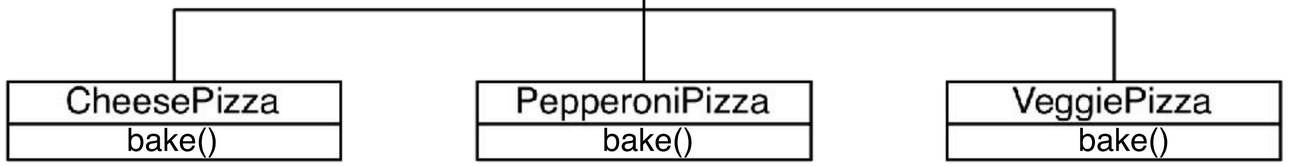
```java
public class PizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

Create concrete pizzas

Declare the interface for different pizzas

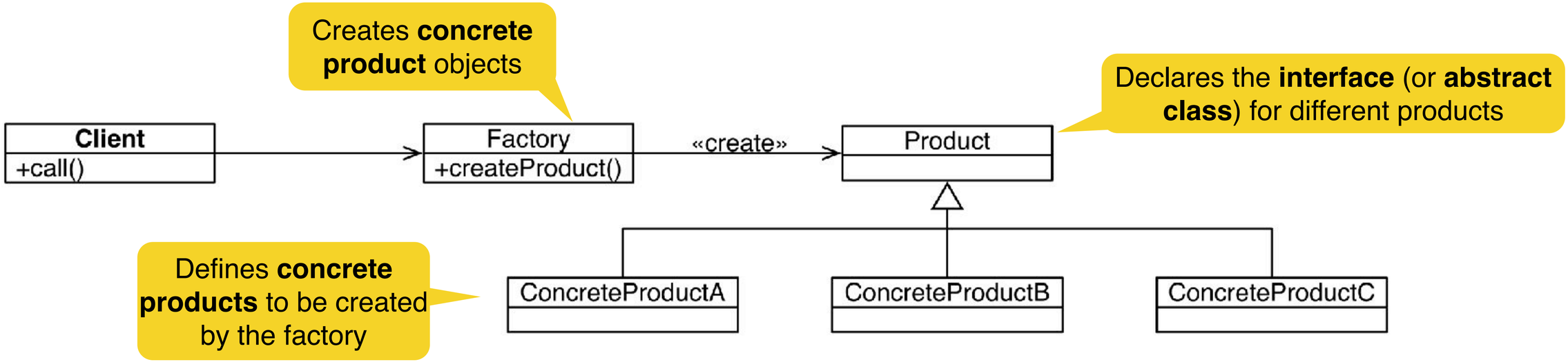



Pizza taxonomy



# Purpose of the factory class

- Acts as delegate for the creation of products
- Allows the **`Client`** to use a single interface to the products of a factory
- Polymorphism allows the client to use each of the concrete products in a uniform way
- Makes it easy to add additional products (extensibility)



# Check list

1. If you have an inheritance hierarchy, consider adding a polymorphic creation capability by defining a factory method in the base class
2. Design the arguments to the factory method: what qualities or characteristics are necessary and sufficient to identify the correct derived class to instantiate?
3. Consider designing an internal object pool that will allow objects to be reused instead of created from scratch (caching)
4. Consider making all constructors private or protected

# Factory method pattern vs. other patterns

- Factory methods are usually called within template methods
- Often, designs start using factory method (less complicated, more customizable, subclasses proliferate)
- Sometimes, they evolve towards abstract factory or builder when more flexibility is needed


**L03E01 Factory Method Pattern**

Not started yet.

Start exercise

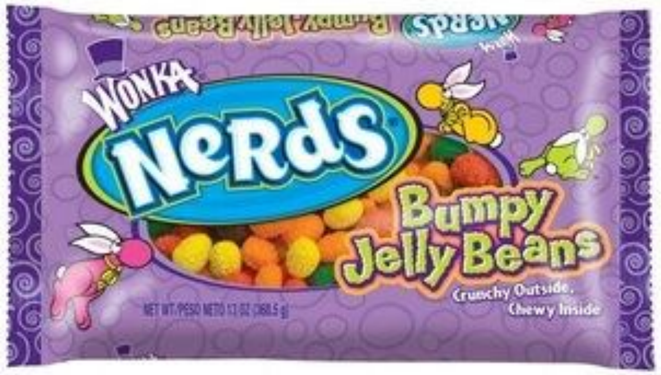
Easy

Due date in 7 days

15 min

5 pts

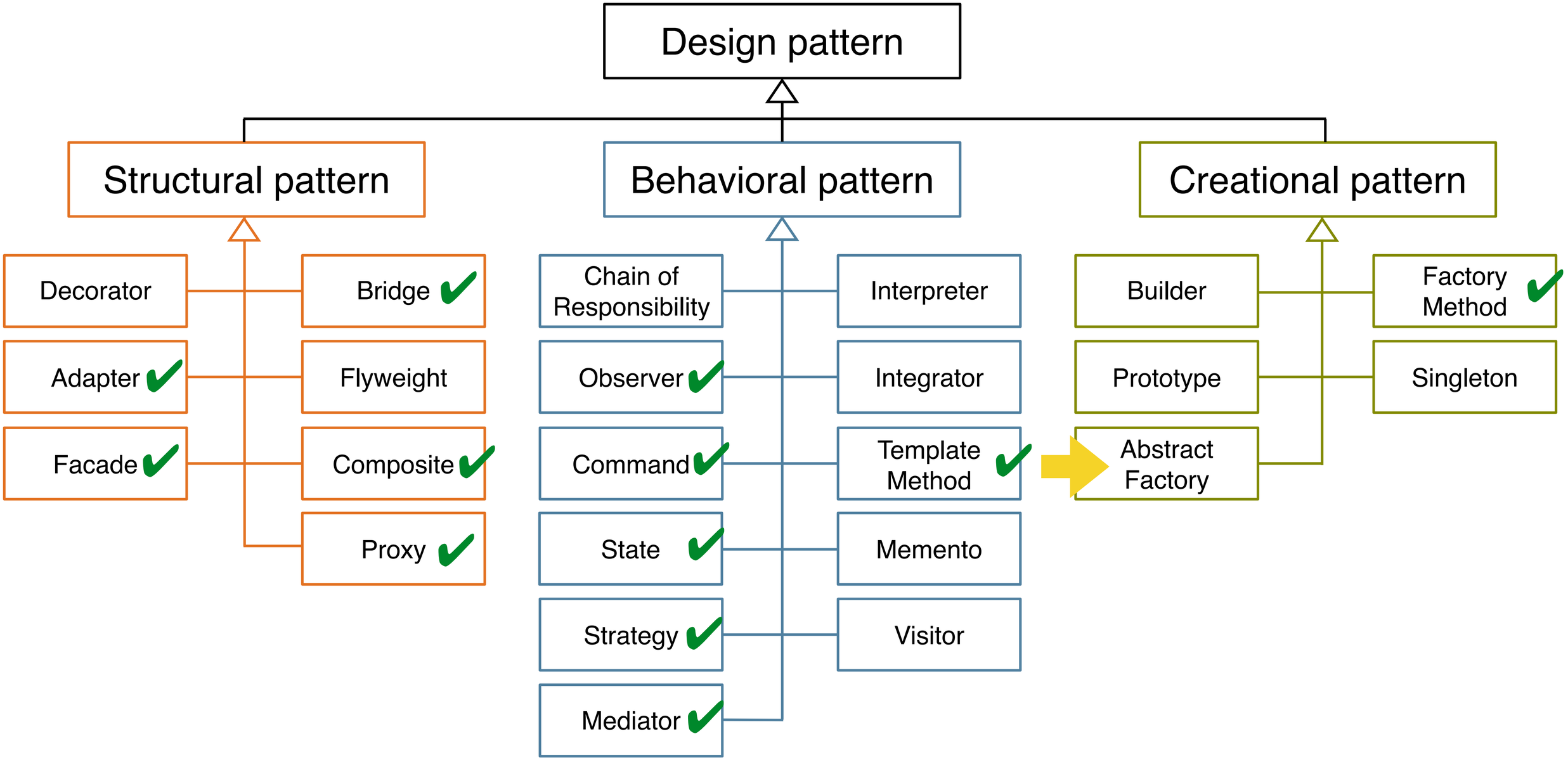
- Problem statement: Willy Wonka Sweet Factory
  - **Part 1**: Sweet Factory Assortment
  - **Part 2**: Production error handling
  - **Part 3**: Willy Wonka Factory
  - **Part 4**: FactoryGuest

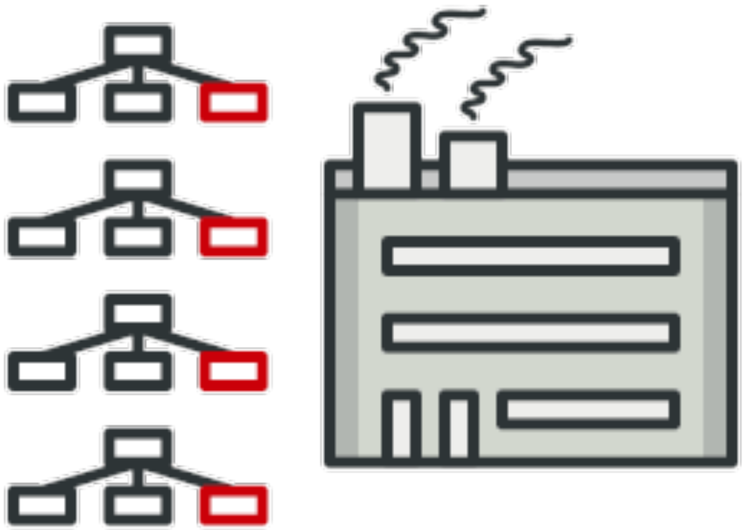
# Outline

- Factory method pattern
- Abstract factory pattern
- Flyweight pattern
- Builder pattern

# Design pattern overview

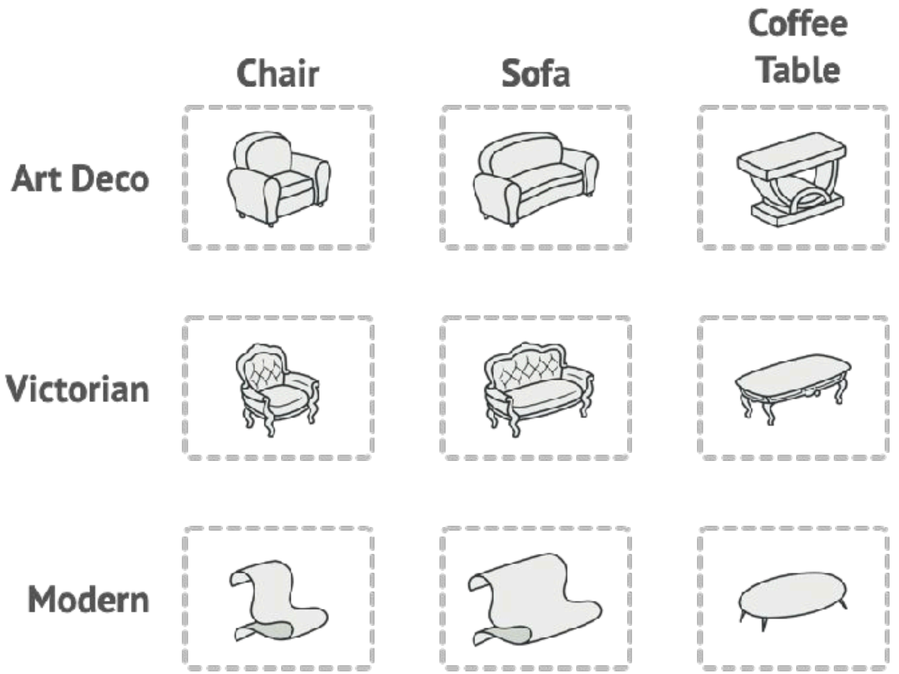


# Abstract factory pattern



# Problem

- Example: a family of related products: chair + sofa + coffee table
  - Several variants of this family: modern, victorian, art deco




How can we make sure to create individual furniture objects so that they match other objects of the same family?

# Motivation

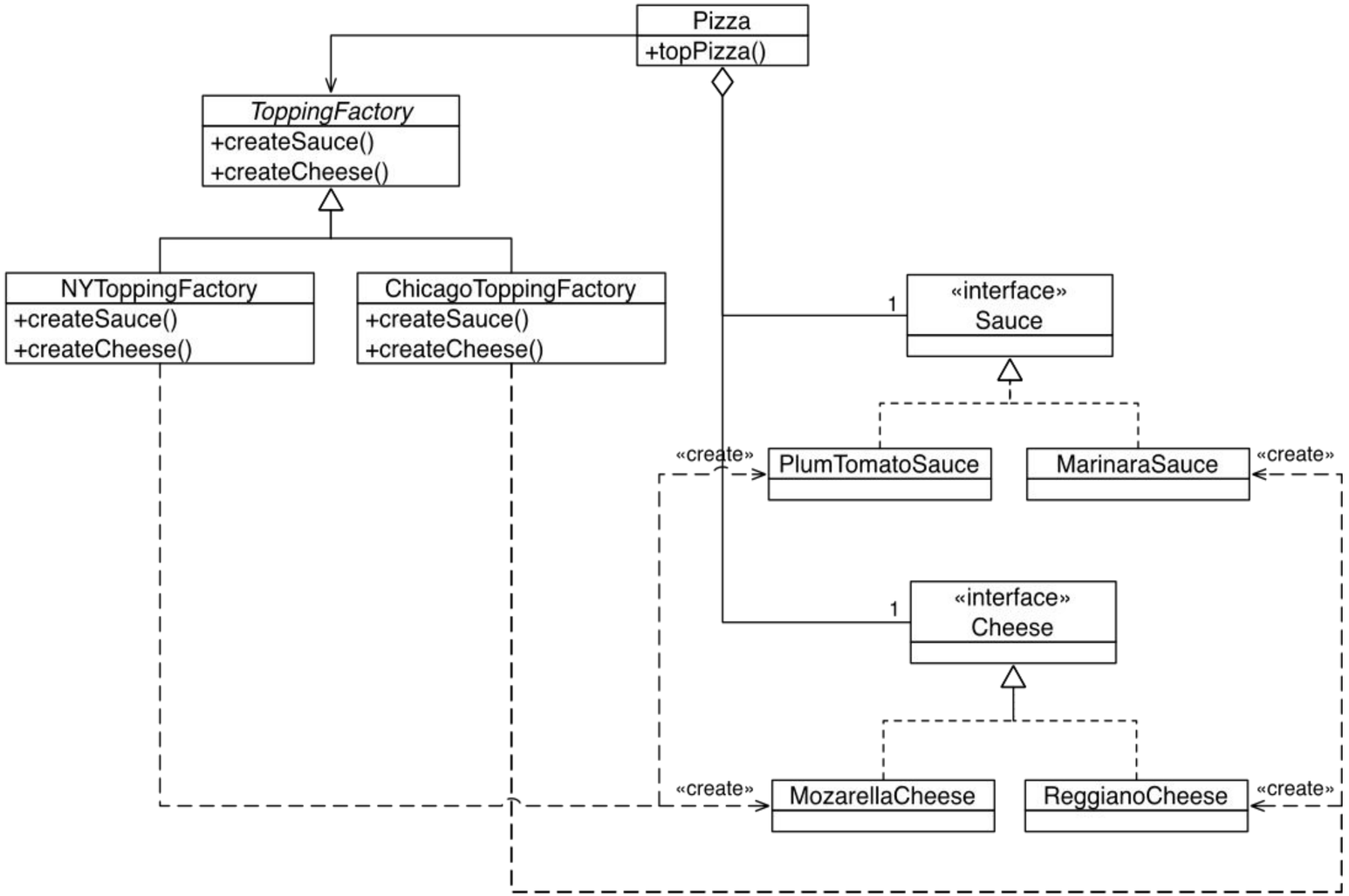
- Consider a user interface toolkit that supports multiple look and feel standards for different operating systems
  - Can you write a single user interface across the different look and feel standards for the window managers used by different operating systems?
  - Example: Chrome on Windows, Chrome on MacOS
- Consider a facility management system for an intelligent house that supports different control systems
  - Can you write a facility management application that encapsulates the manufacturer-specific control system from the rest of the application?
  - Examples of existing control systems: Zumtobel, KNX (originally called EIB)

# From the factory method to the abstract factory pattern

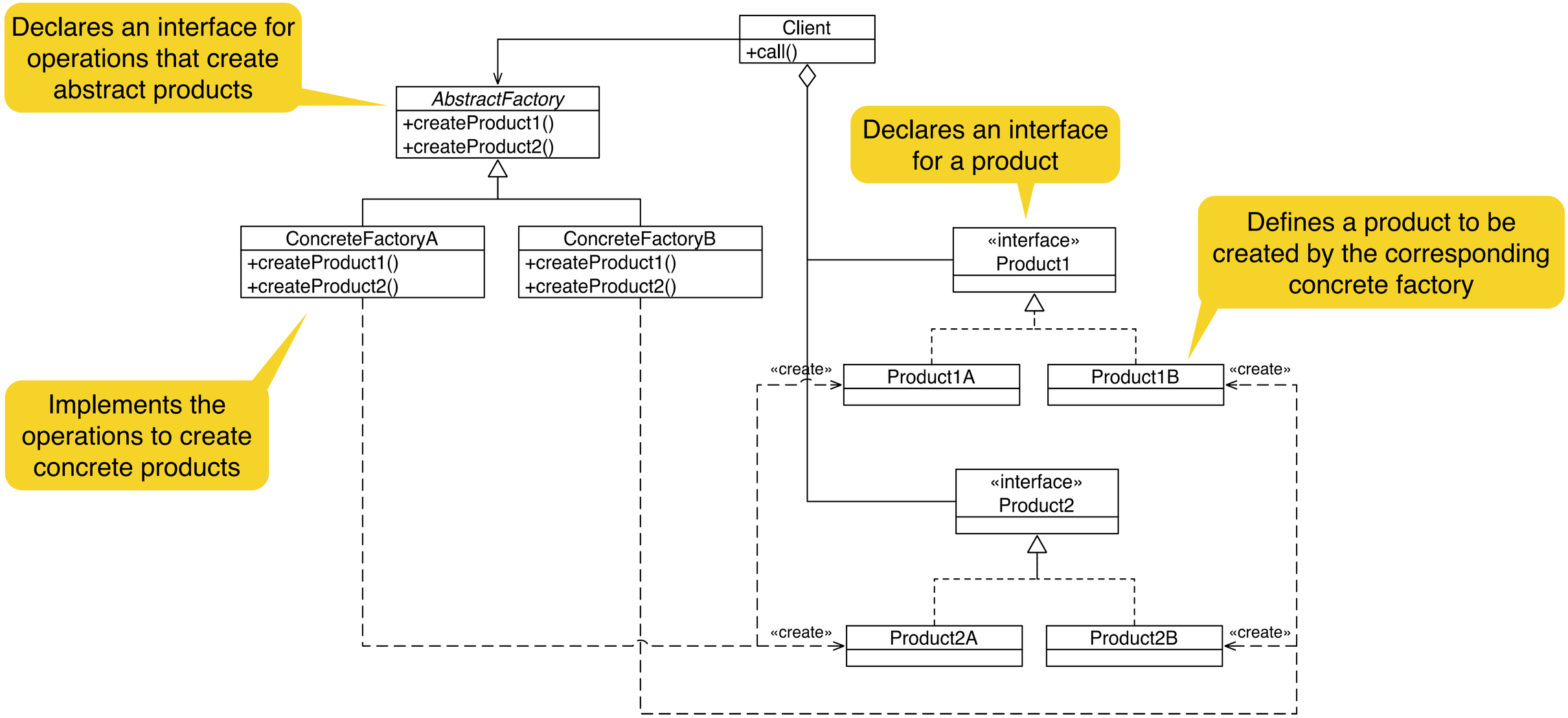
- The factory method pattern puts calls to **`new`** into the factory, thereby reducing the dependency of the client code on a concrete implementation
- But we are bound to a single factory and still have a lot of **`if statements`**
- How can we generalize this?
  - Example: New Yorkers like Mozzarella, people from Chicago prefer Reggiano cheese
  - New Yorkers prefer Plum Sauce, while Chicagoans like only Marina sauce
- Can we write an application for pizza toppings which can be customized for people living in different cities?
  - Can we create factories for Chicago style pizzas and NY style pizzas?

# Example: abstract factory pattern for pizzas

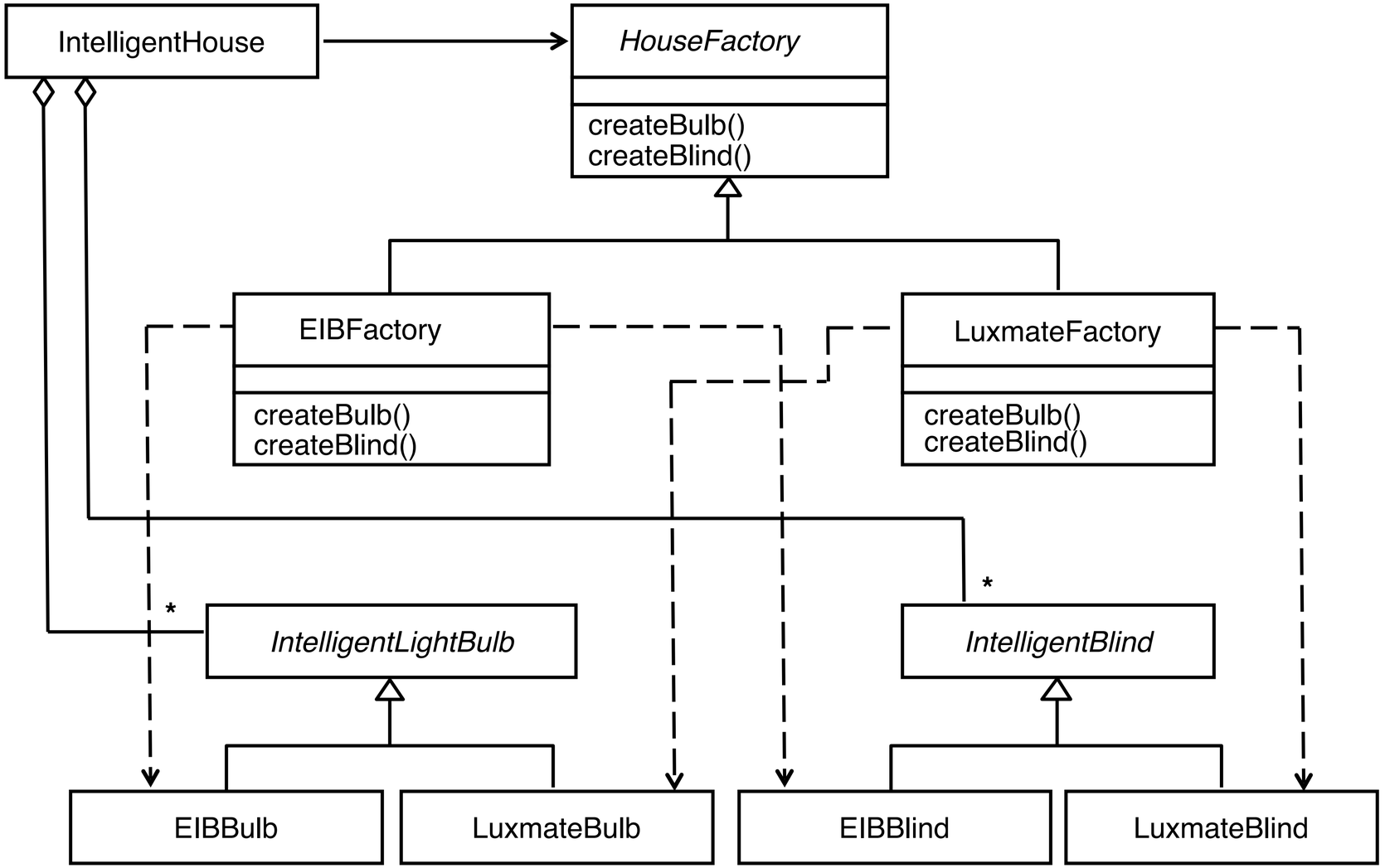
- A pizza is a family of related objects of toppings and sauces



# Structure


Declares an interface for operations that create abstract products
Client
+call()
AbstractFactory
+createProduct1()
+createProduct2()
Declares an interface for a product
Defines a product to be created by the corresponding concrete factory
ConcreteFactoryA
+createProduct1()
+createProduct2()
ConcreteFactoryB
+createProduct1()
+createProduct2()
«interface»
Product1
Implements the operations to create concrete products
«create»
Product1A
Product1B
«create»
«interface»
Product2
«create»
Product2A
Product2B
«create»

# Example: facility management system for a house



# Applicability

- Independence from initialization or representation
- Manufacturer independence
- Constraints on related products
- Cope with upcoming change

# Benefits and drawbacks

+ **Compatibility** of products produced by the factory
+ **Low 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 maintain
+ **Open / closed principle**: you can introduce new variants of products without breaking existing client code
– **Complex code**: a lot of new interfaces and classes are introduced along with the pattern

# Abstract factory vs. other patterns

- Factory method often evolves into an abstract factory
- 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 the bridge pattern: useful when some abstractions defined by **bridge** can only work with specific implementations
  - Abstract factory can encapsulate these relations and hide the complexity from the client code


## L03E02 Abstract Factory Pattern

Not started yet.

Start exercise

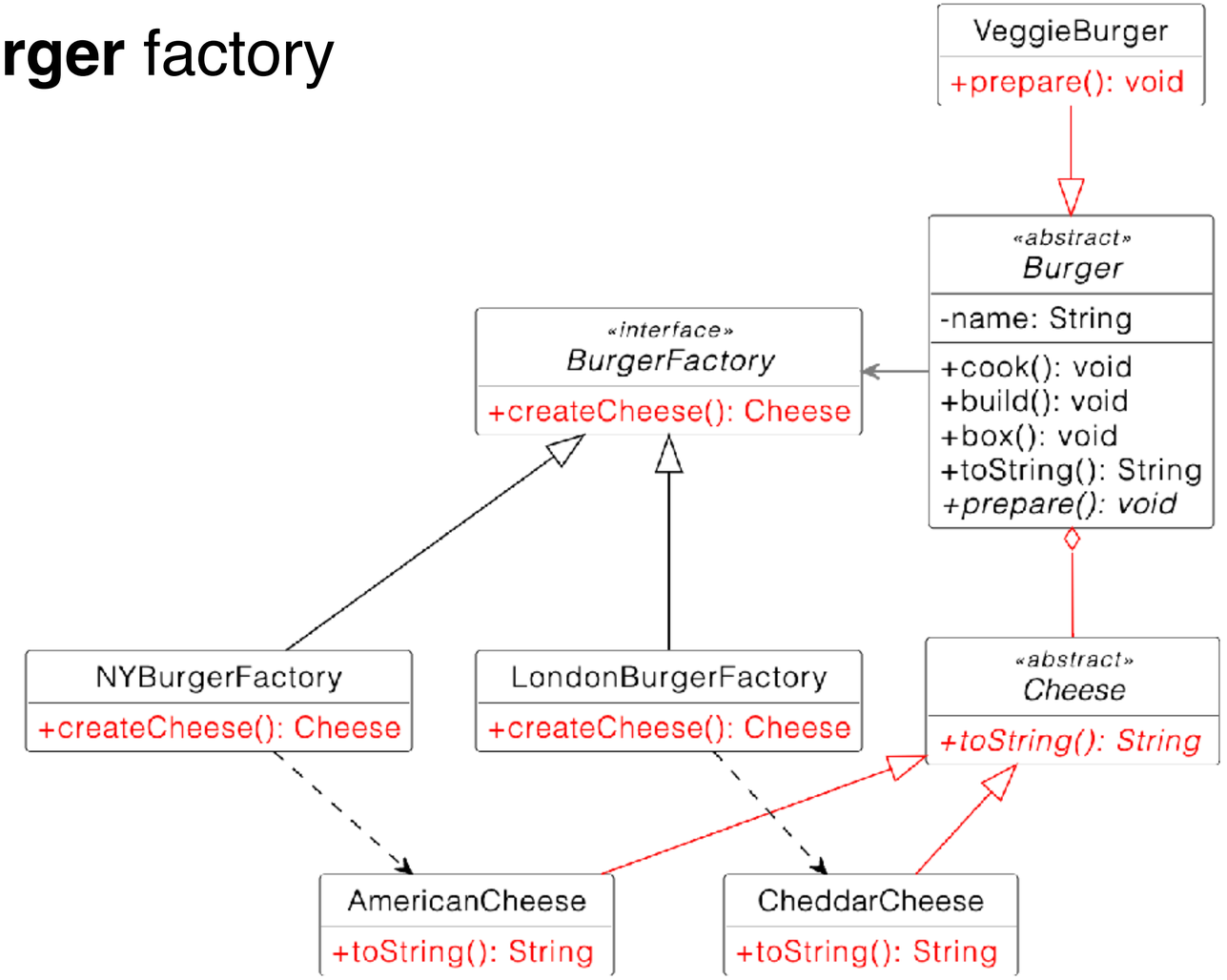
Easy

Due date in 7 days

20 min

5 pts

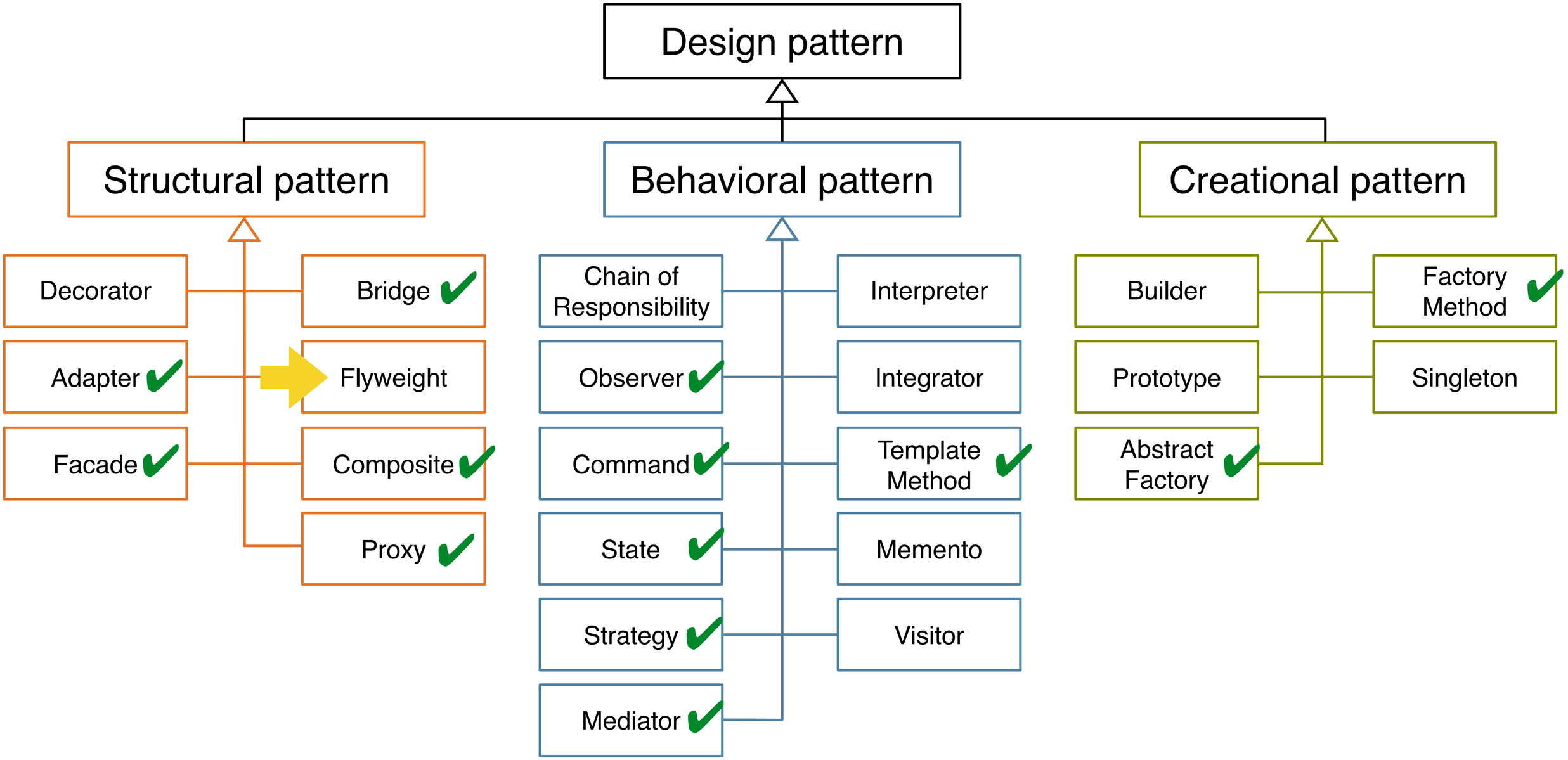
- Problem statement: Implement a **burger** factory
  - **Part 1:** different types of cheese
  - **Part 2:** London franchise store
  - **Part 3:** new type of burger: vegan
  - **Part 4:** distribute this new burger in the respective stores



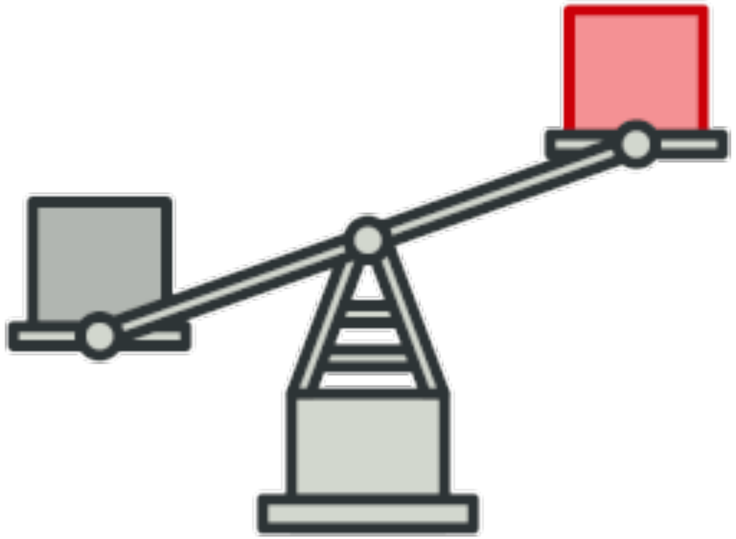
# Outline

- Factory method pattern
- Abstract factory pattern
- Flyweight pattern
- Builder pattern

# Design pattern overview

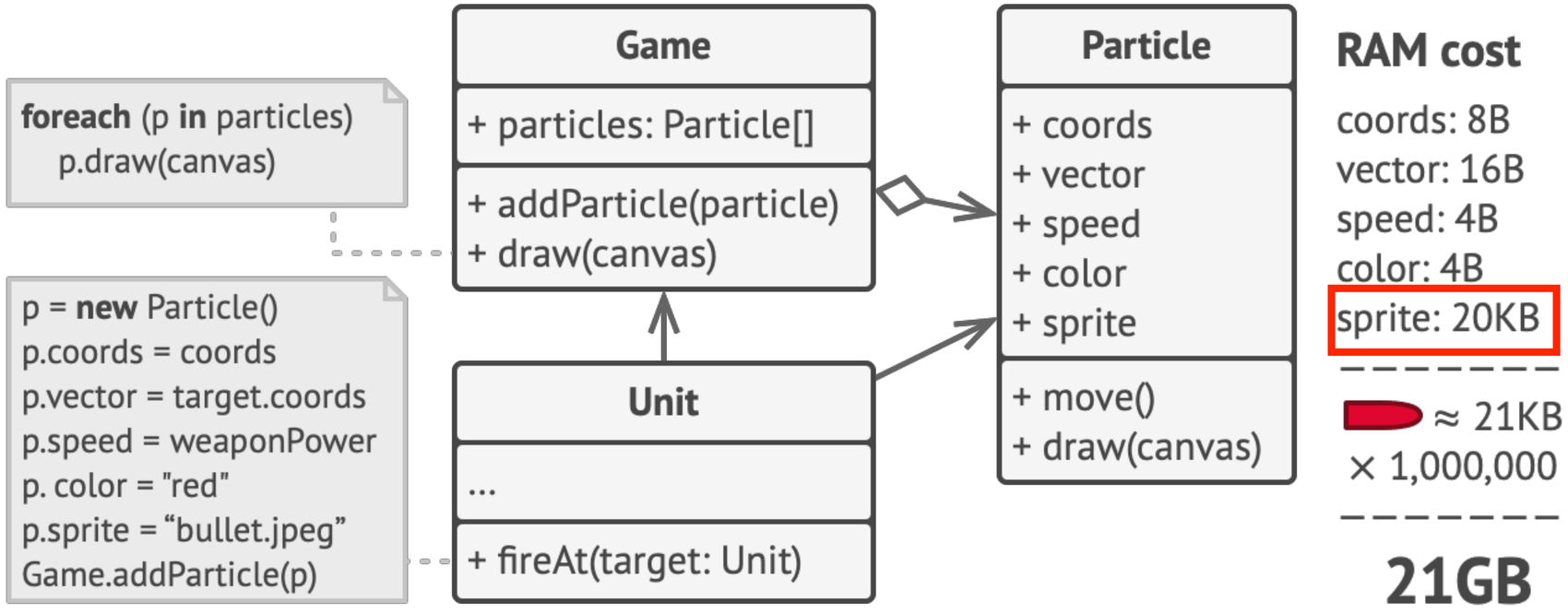


# Flyweight pattern



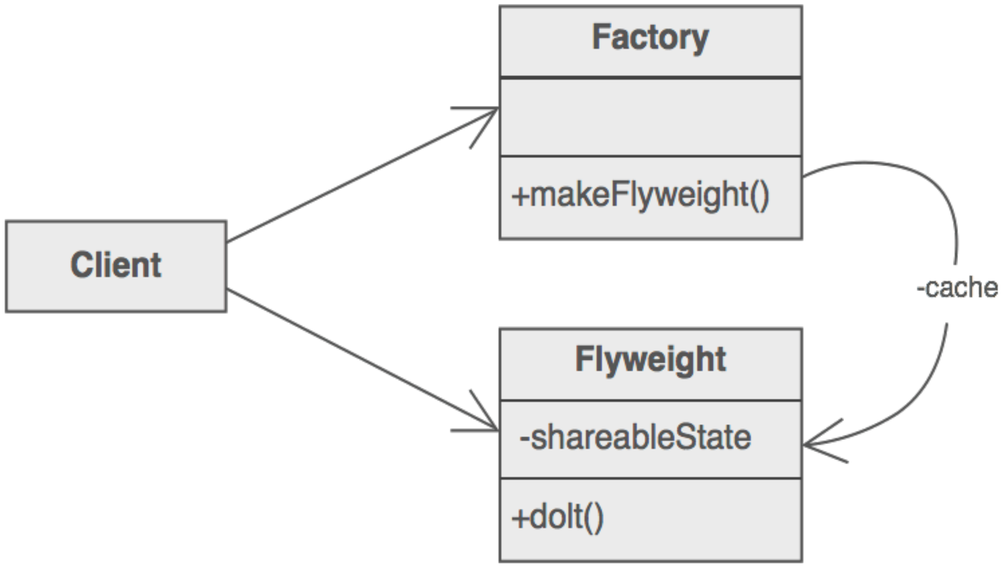
# Problem

- Designing objects down to the lowest levels of system granularity provides optimal flexibility
- But it can be **unacceptably expensive** in terms of **performance** and **memory usage**
- Example: particle system with 1,000,000 particles

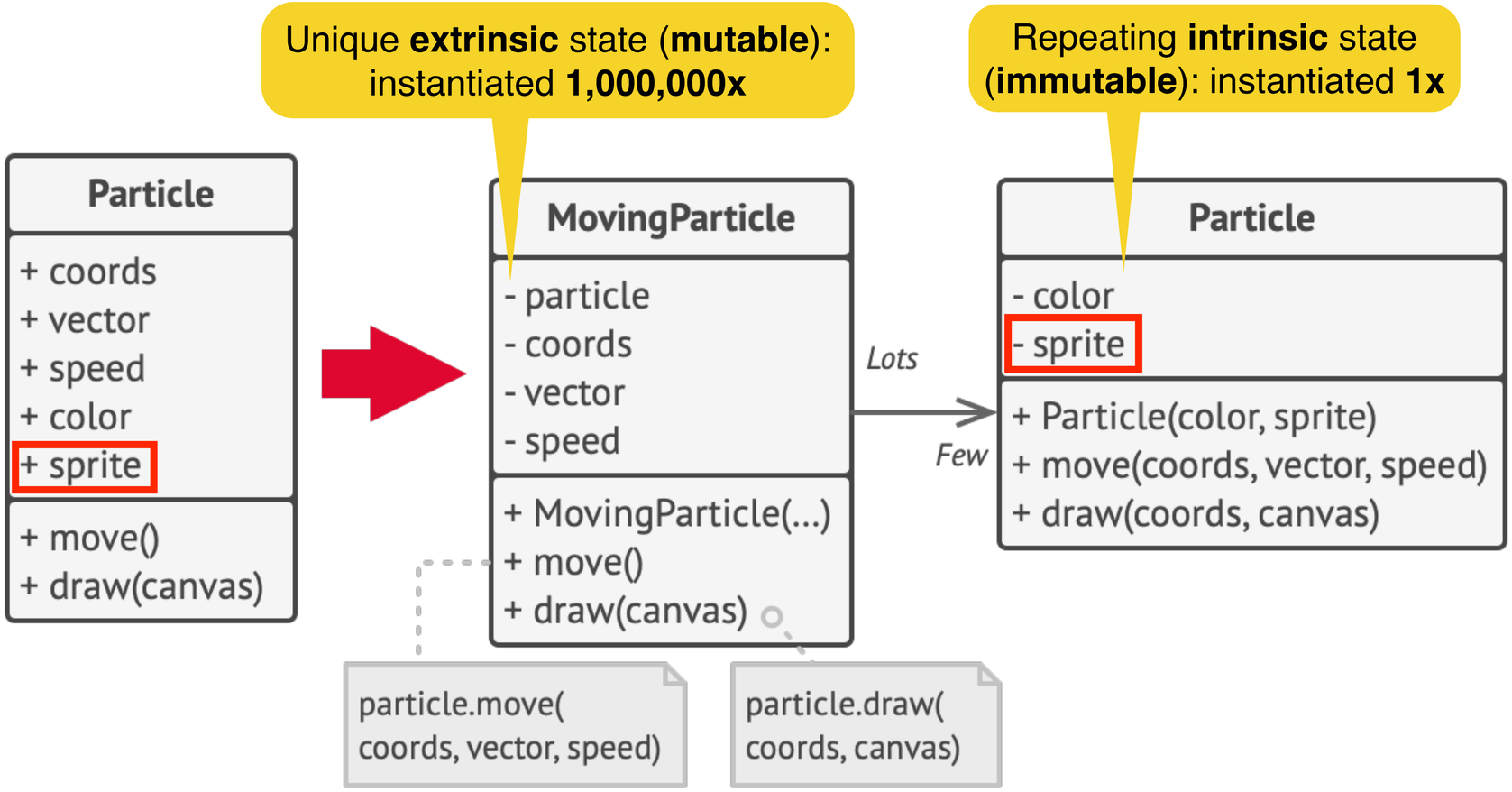


# Flyweight pattern

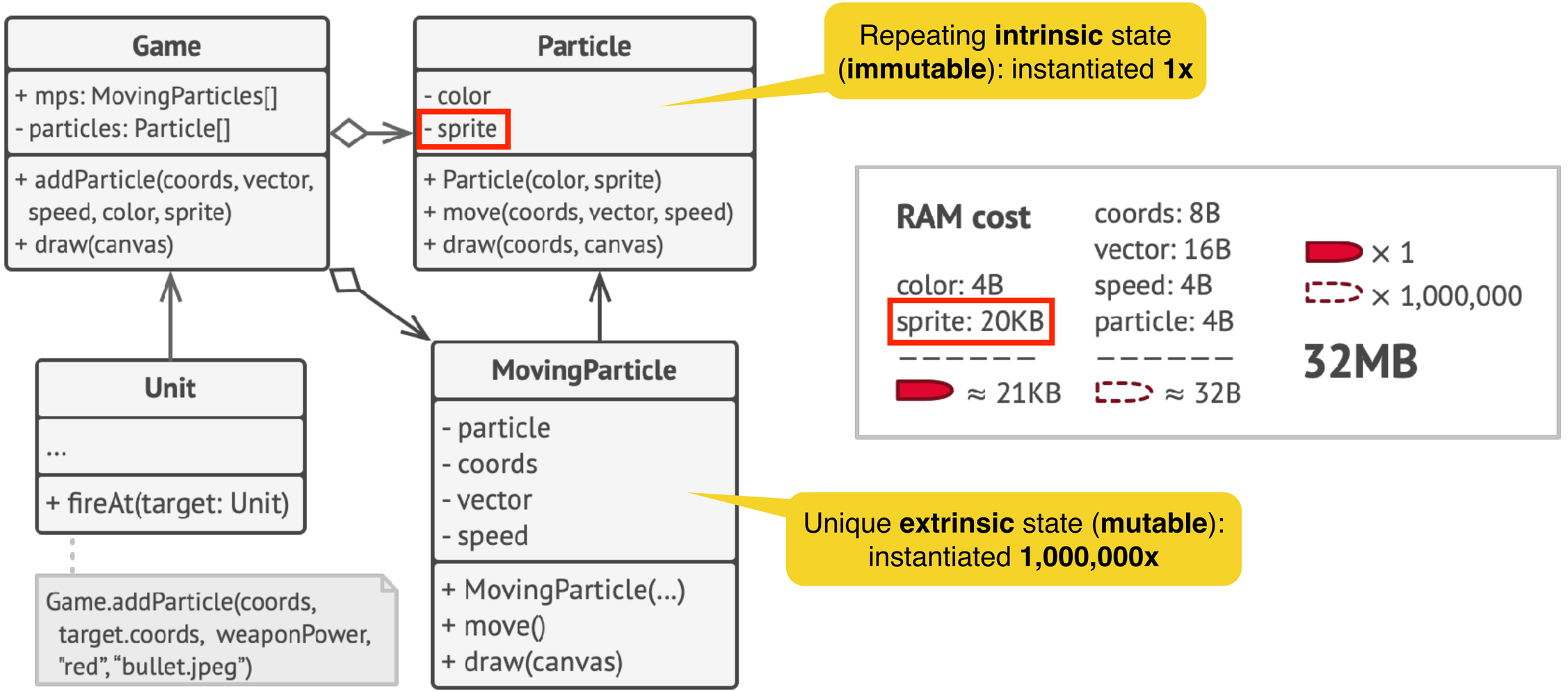
- Purpose: share objects to allow their use at fine granularity without prohibitive cost
- Each flyweight object is divided into **two** pieces
  1. **Extrinsic / mutable:** the state **dependent** part: stored or computed by client objects, and passed to the flyweight when its operations are invoked
  2. **Intrinsic / immutable:** the state **independent** part: stored (shared) in the flyweight object



# Example: **improved** particle system

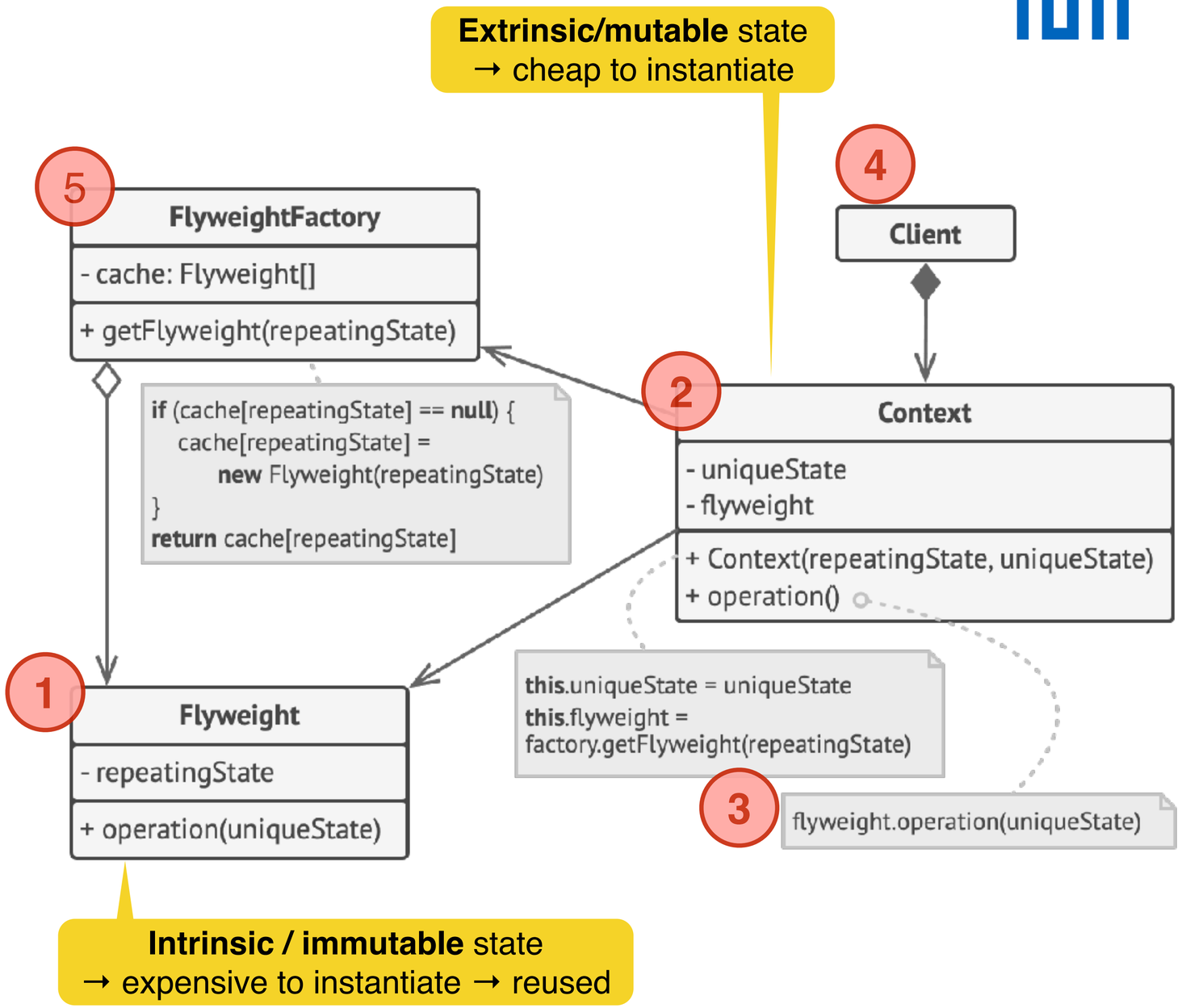


# Example: **improved** particle system



# Structure

1. `Flyweight` contains the portion of the state that can be shared between multiple objects **(immutable / intrinsic)**
2. `Context` contains **extrinsic / mutable** state, unique across all original objects
3. The behavior of the original object remains in the flyweight: a caller must also pass the **extrinsic** state
4. `Client` calculates or stores the **extrinsic** state of flyweights
5. `FlyweightFactory` manages a **cache** of existing flyweights



# Check list

1. Ensure that **object overhead** is an issue: the client of the class is able and willing to absorb responsibility realignment
2. Divide the target class's state into **2 objects**: **shareable** intrinsic state and **non-shareable** extrinsic state
3. Remove the **non-shareable** state from the class attributes, and add it to the calling argument list of affected methods
4. Create a **factory** that can cache and reuse existing class instances
5. The **client** must use the **factory** instead of the new operator to request objects
6. The **client** must look-up or compute the non-shareable state, and pass that state to class methods

# Flyweight vs. other patterns

- Whereas flyweight shows how to make lots of little objects, facade shows how to make a single object represent an entire subsystem
- Flyweight is often combined with composite to implement shared leaf nodes
- Flyweight explains when and how state objects can be shared


**L03E03 Flyweight Pattern**

Not started yet.

Start exercise

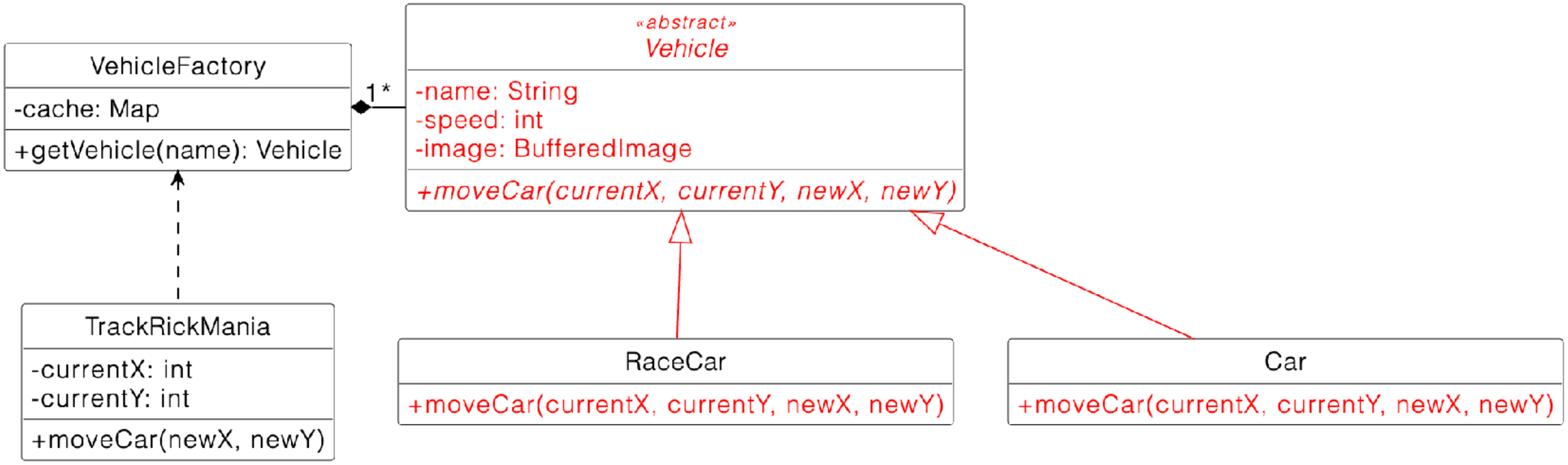
**Medium**

**Due date in 7 days**

20 min

5 pts

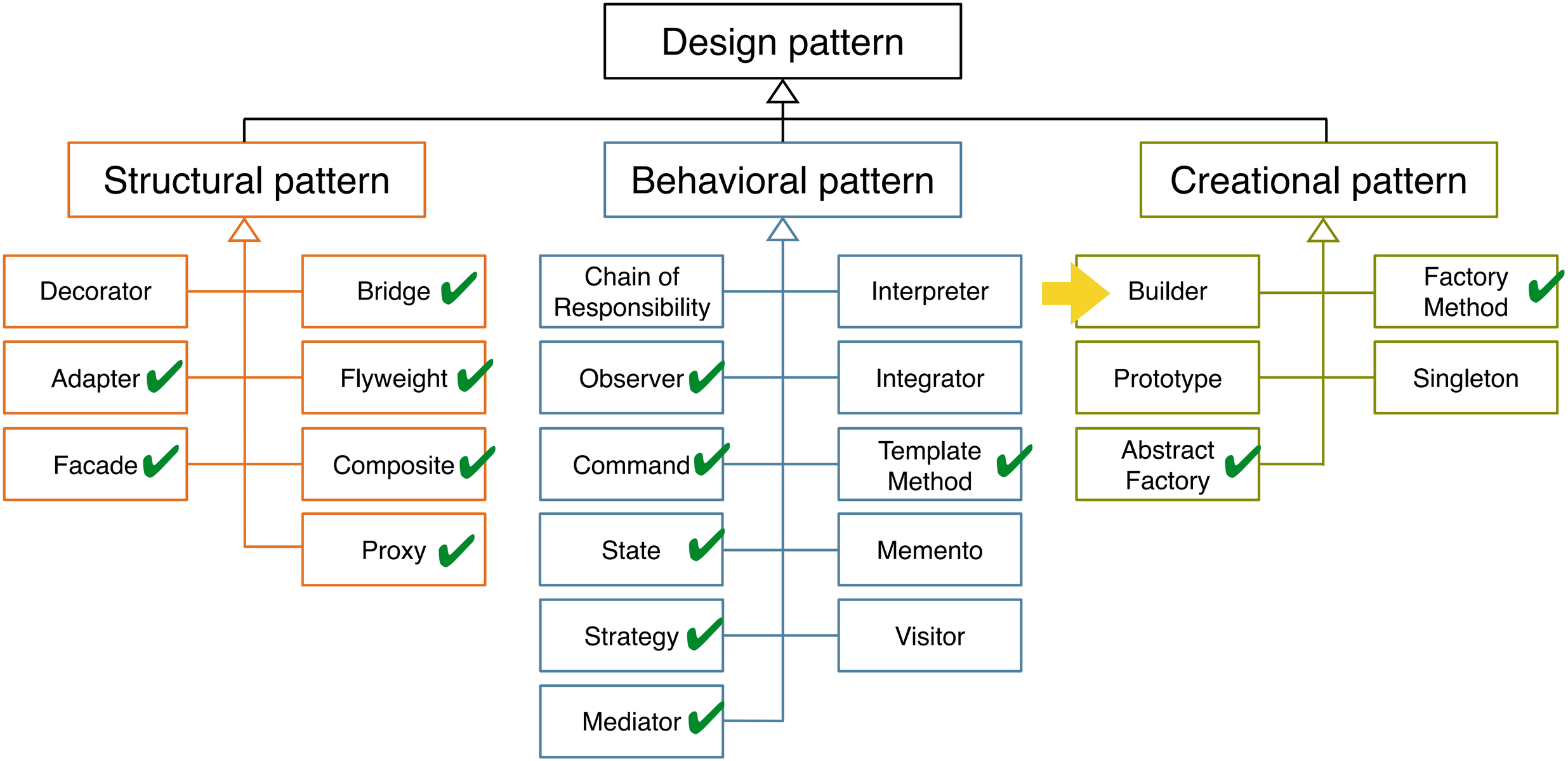
- Problem statement: car racing with 1000 cars



# Outline

- Factory method pattern
- Abstract factory pattern
- Flyweight pattern
- Builder pattern

# Design pattern overview

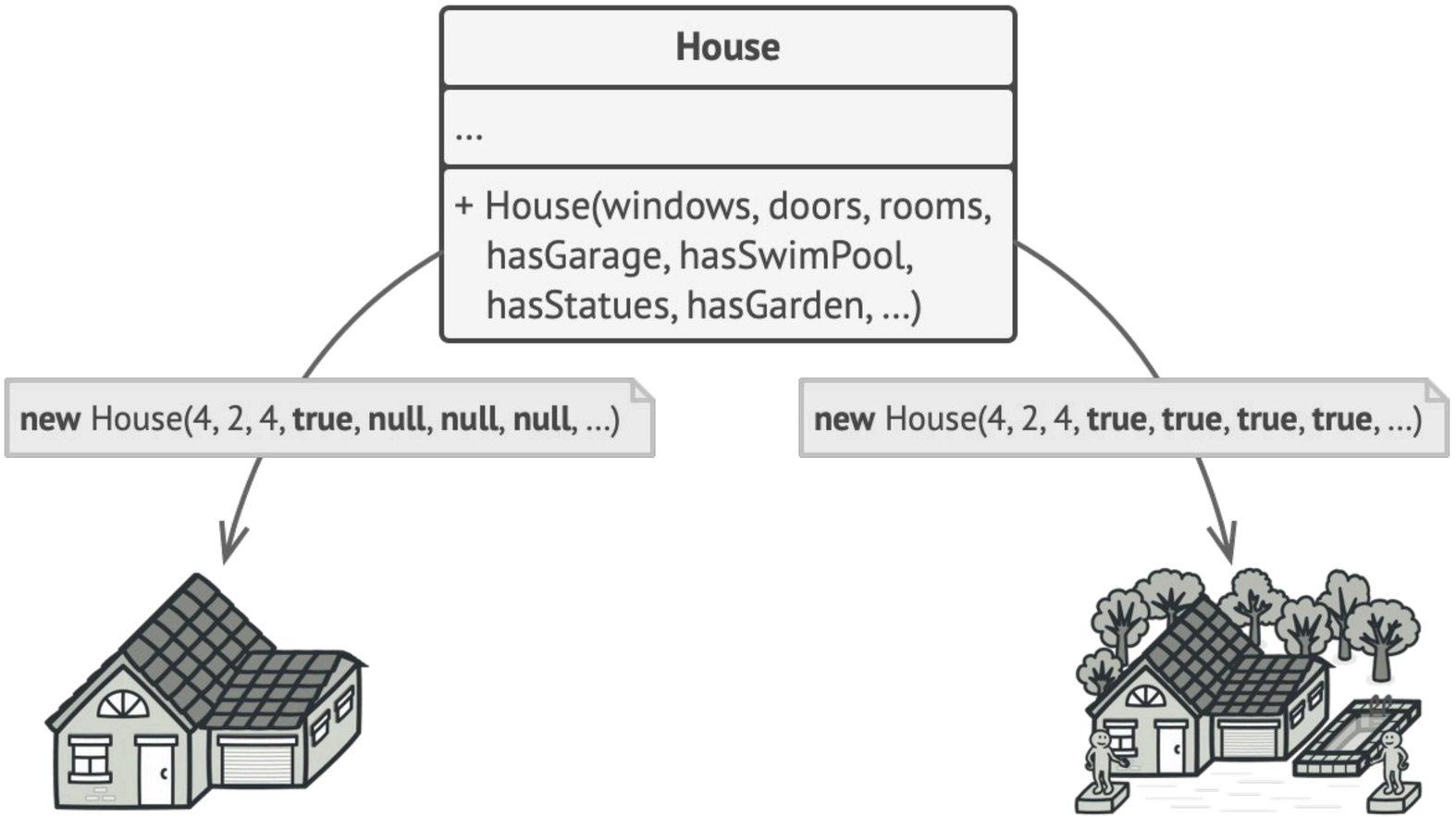


# Builder pattern

# Problem

- An application needs to create the elements of a complex aggregate
- The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage
- Intent
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations
  - Parse a complex representation, create one of several targets

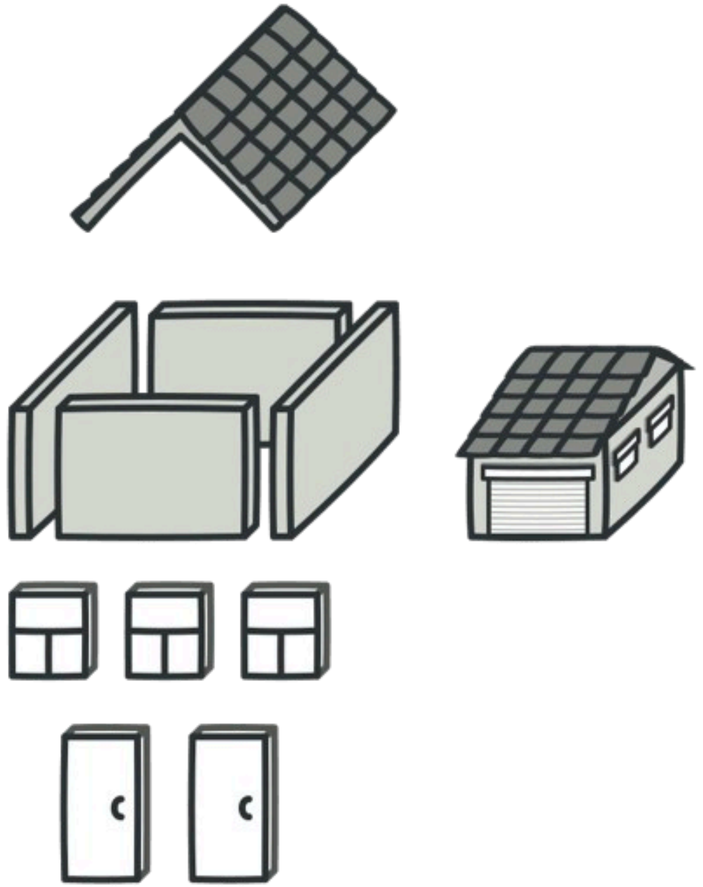
# Example




If most of the parameters are unused, this makes constructor calls pretty ugly

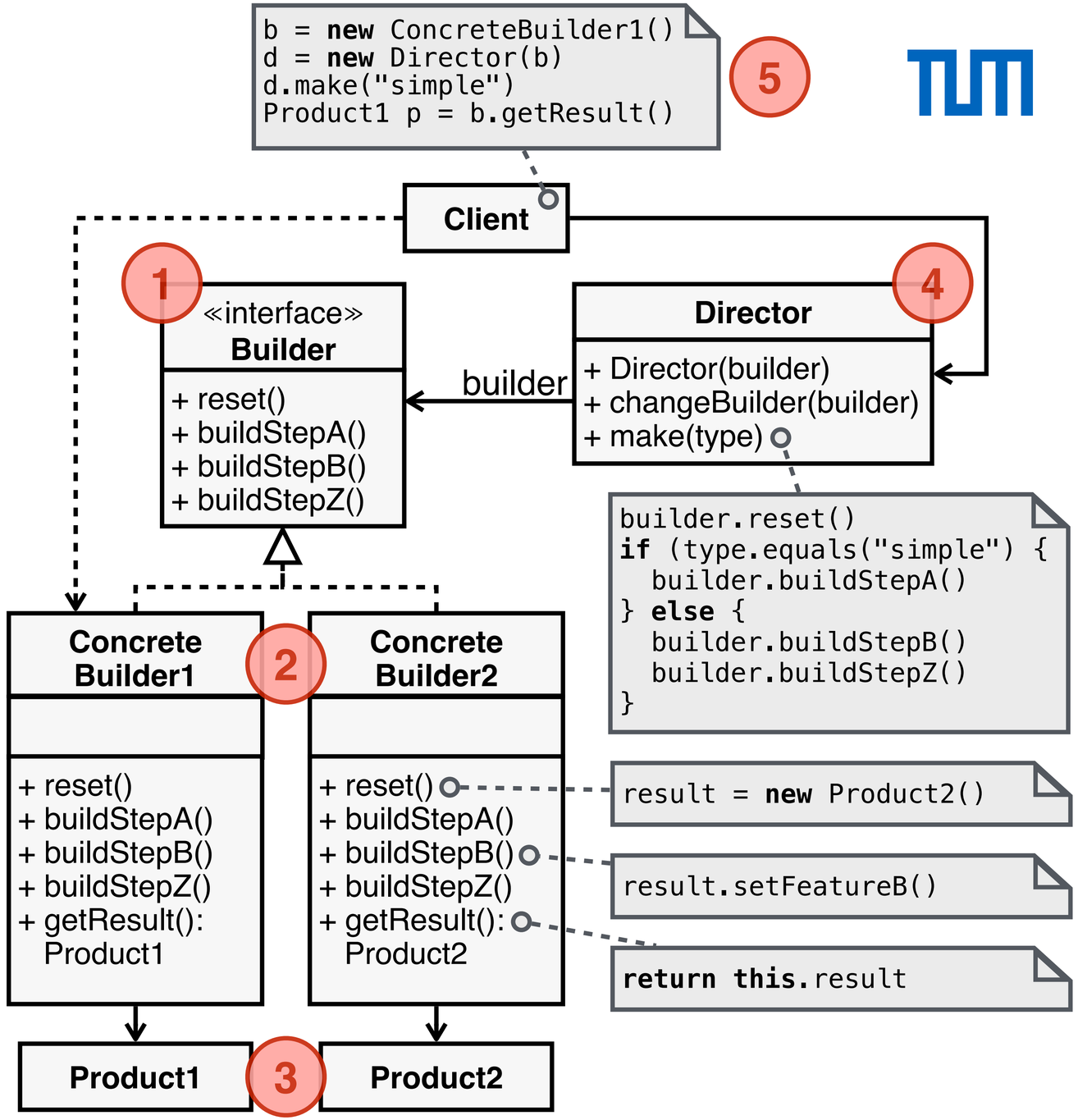
# Improved example

| HouseBuilder |
| --- |
| … |
| + buildWalls()<br>+ buildDoors()<br>+ buildWindows()<br>+ buildRoof()<br>+ buildGarage<br>+ getResult(): House |

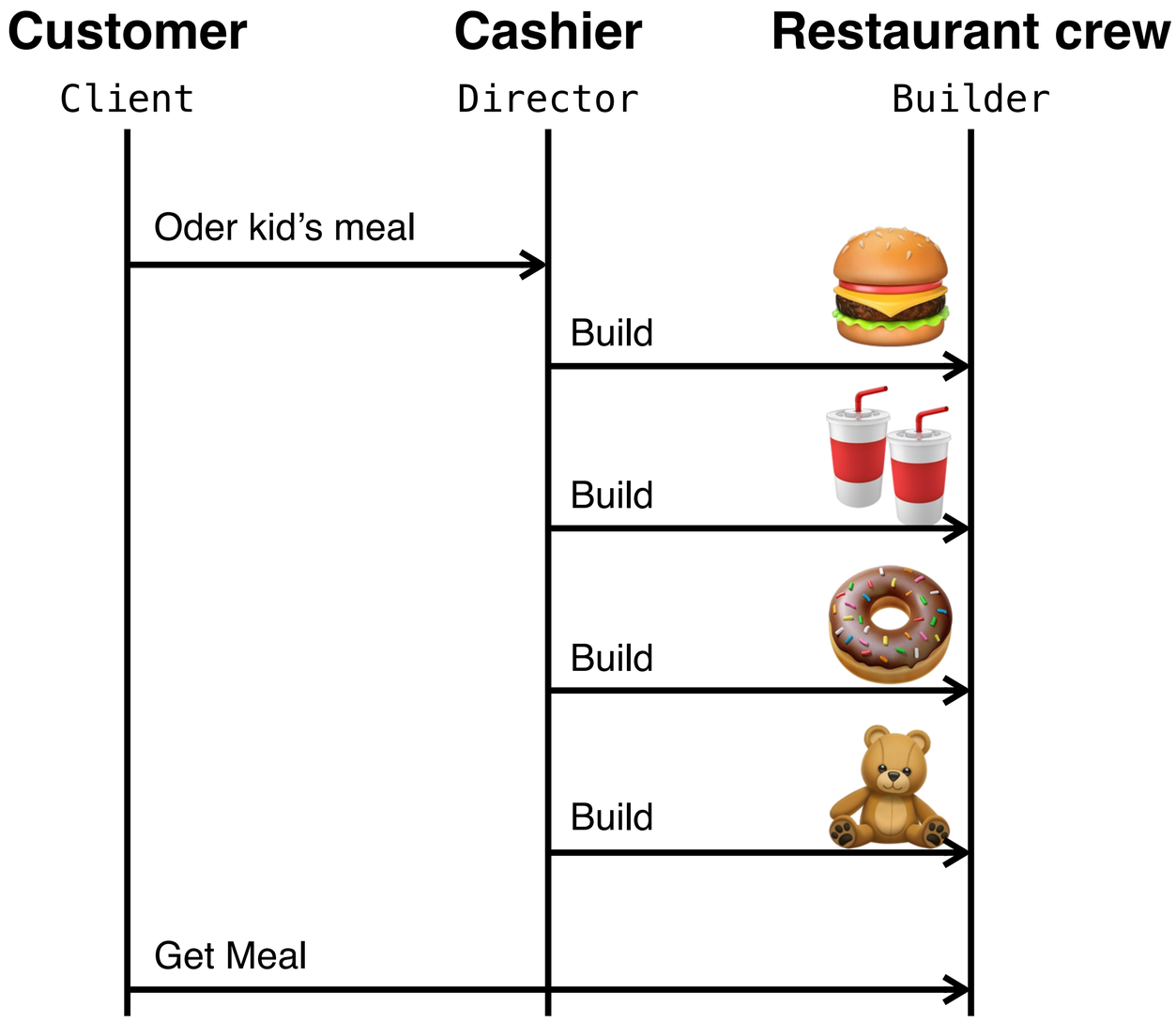


# Structure

1. **Builder** interface declares product construction steps common to all types of builders
2. **ConcreteBuilder**s provide different implementations of the construction steps and may produce products that don't follow the common interface
3. **Product**s are resulting objects constructed by different builders
4. **Director** defines the order in which to call construction steps: create and reuse specific configurations of products
5. **Client** associates builder object with director (usually in constructor)



# Example


Customer
Cashier
Restaurant crew
Client
Director
Builder
Oder kid's meal
Build
Build
Build
Build
Get Meal

# Check list

1. Decide if a common input and many possible representations (or outputs) is the problem at hand
2. Encapsulate the parsing of the common input in a **director** class
3. Design a standard protocol for creating all possible output representations: capture the steps of this protocol in a **builder** interface
4. Define a **builder** derived class for each target representation
5. The client creates a **director** object and a **builder** object, and registers the latter with the former
6. The client asks the **director** to **construct**
7. The client asks the **builder** to return the result

# Builder pattern vs. other patterns

- Builder focuses on constructing a complex object step by step
- Abstract factory emphasizes a family of product objects (either simple or complex)
- Builder returns the product as a final step, but as far as the abstract factory is concerned, the product gets returned immediately
- Builder often builds a composite
- Designs start using factory method (less complicated, more customizable, subclasses proliferate) and evolve toward abstract factory or builder


**L03E04 Builder Pattern**

Not started yet.

Start exercise

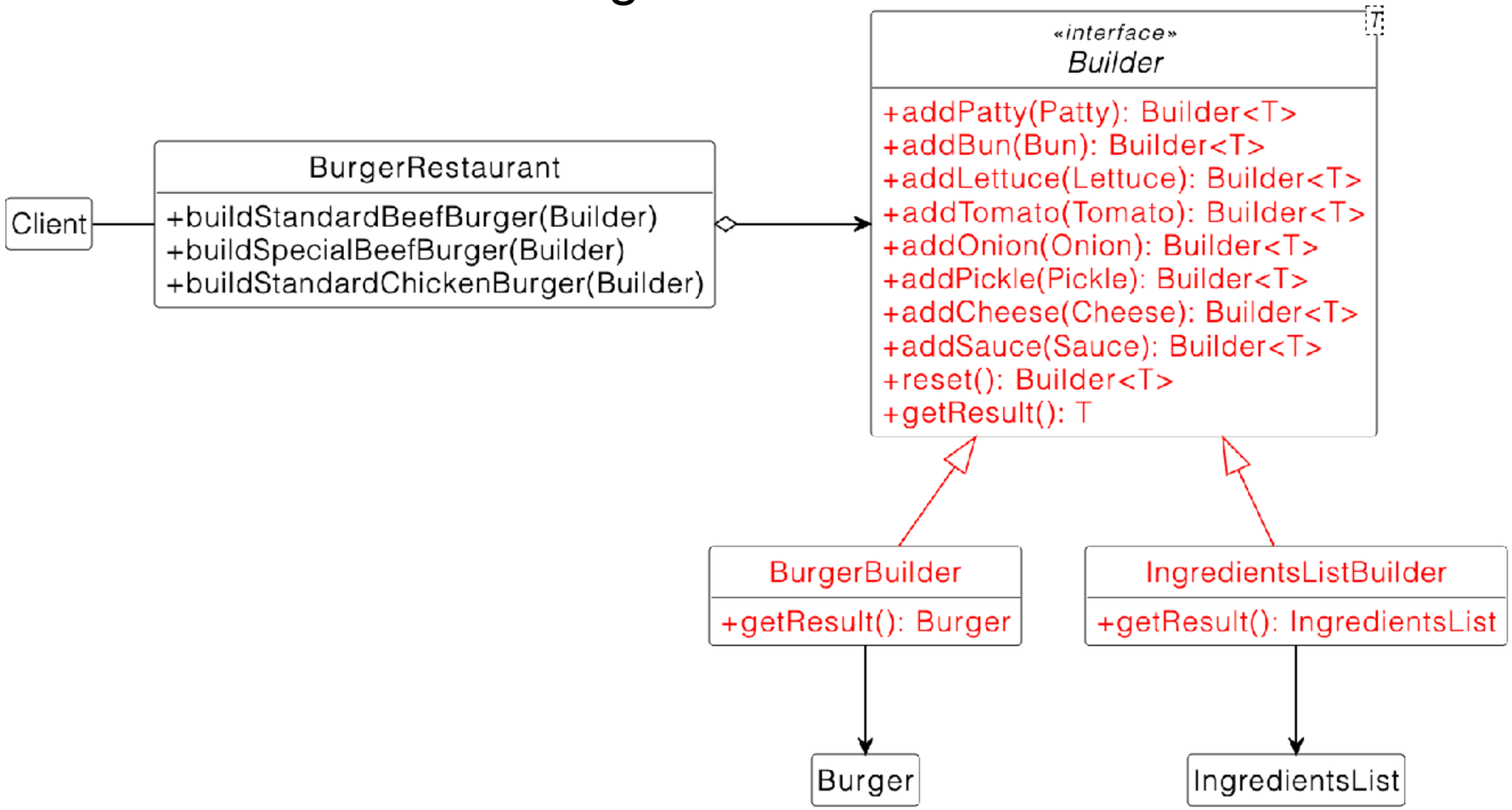
**Medium**

**Due date in 7 days**

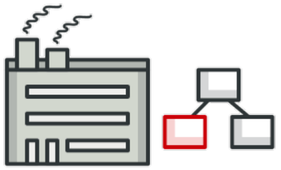
15 min

6 pts

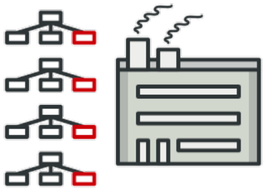
- Problem statement: build a burger



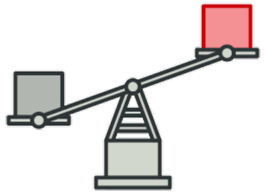
# Summary



The **factory method pattern** provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created



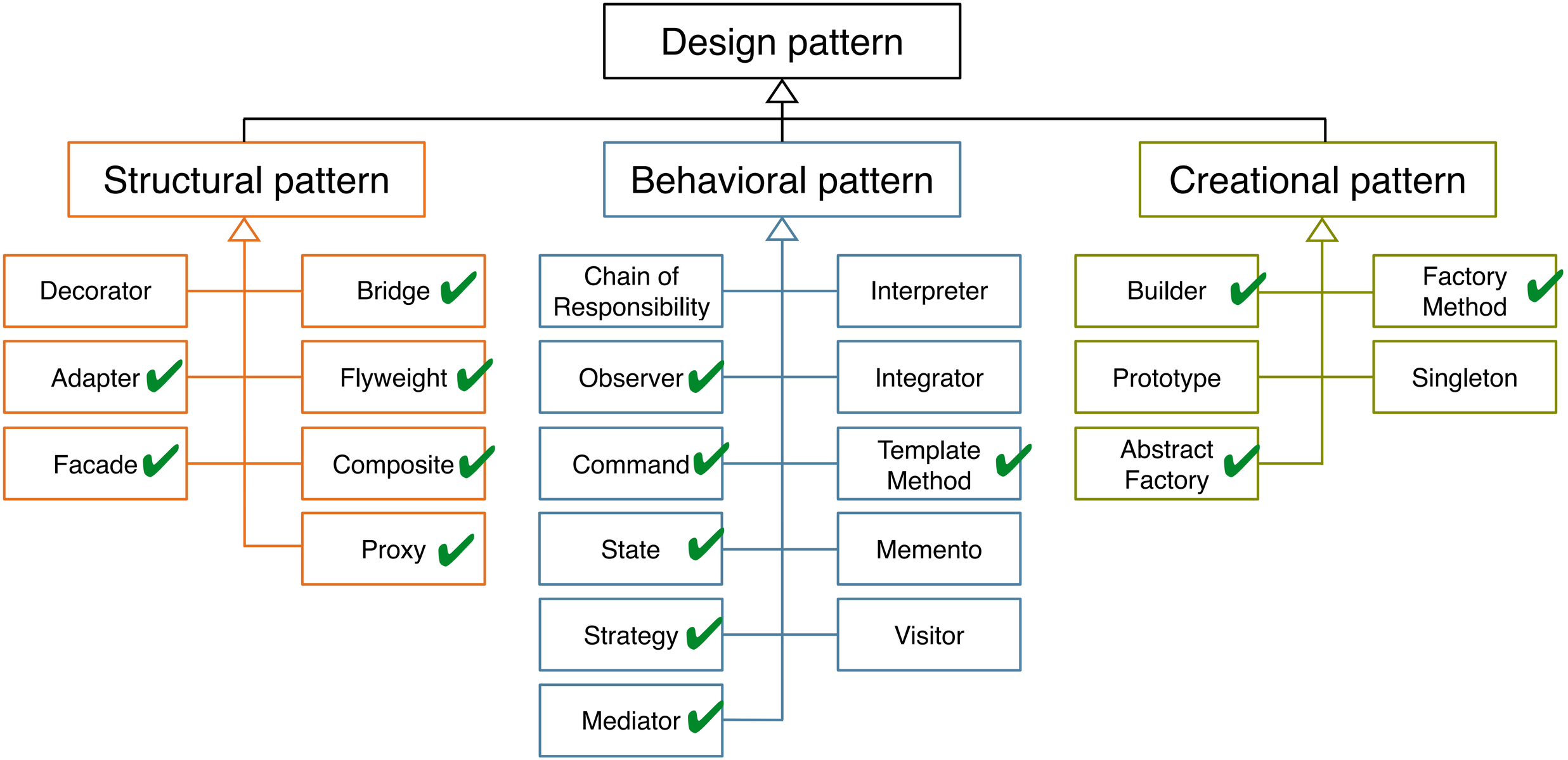
The **abstract factory method** produces families of related objects without specifying their concrete classes



With the **flyweight pattern**, more objects fit 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

The **builder pattern** constructs complex objects step by step and allows to produce different types and representations of an object using the same construction code

# Design pattern overview



# Literature

- Eric Gamma et al: Design patterns, Addison Wesley, 1995
- Elisabeth Freeman et al: Head First Design Patterns, O'Reilly 2004
- https://refactoring.guru/design-patterns
- https://sourcemaking.com/design_patterns