Software Design Patterns

Lecture 8

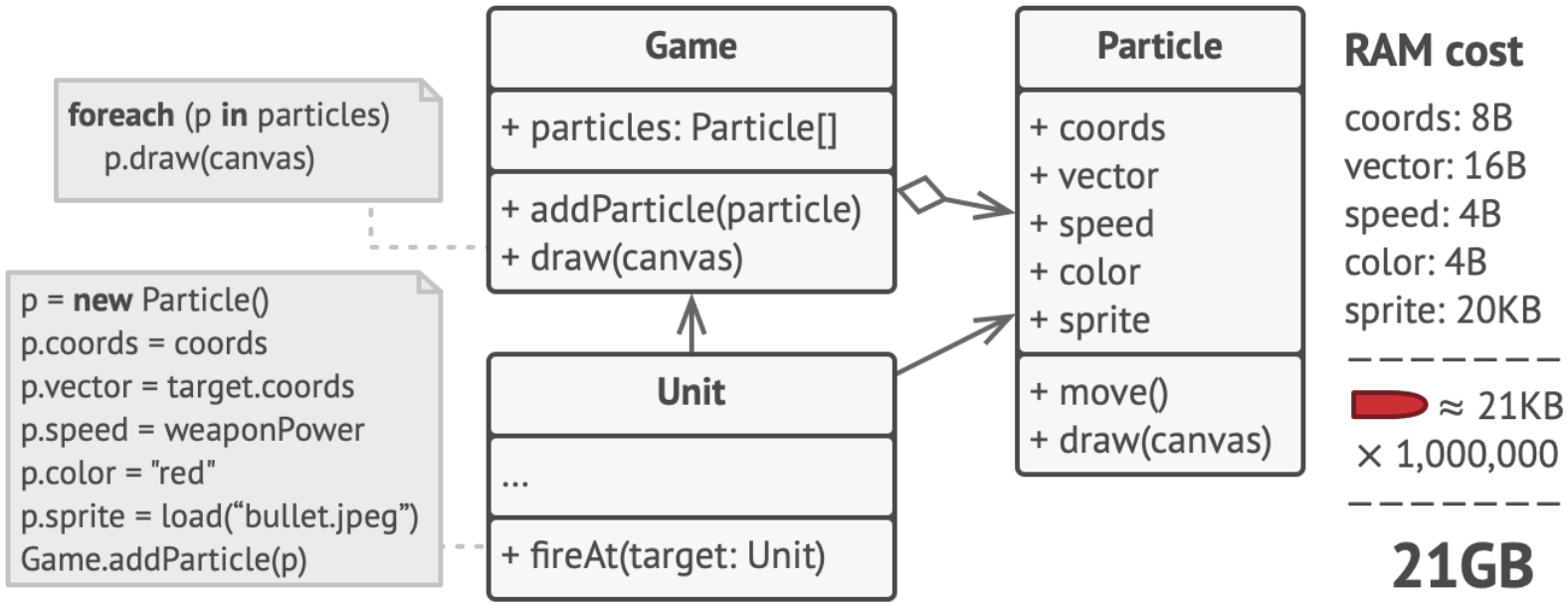
Flyweight
Proxy

Dr. Fan Hongfei

24 October 2024

Flyweight: Problem

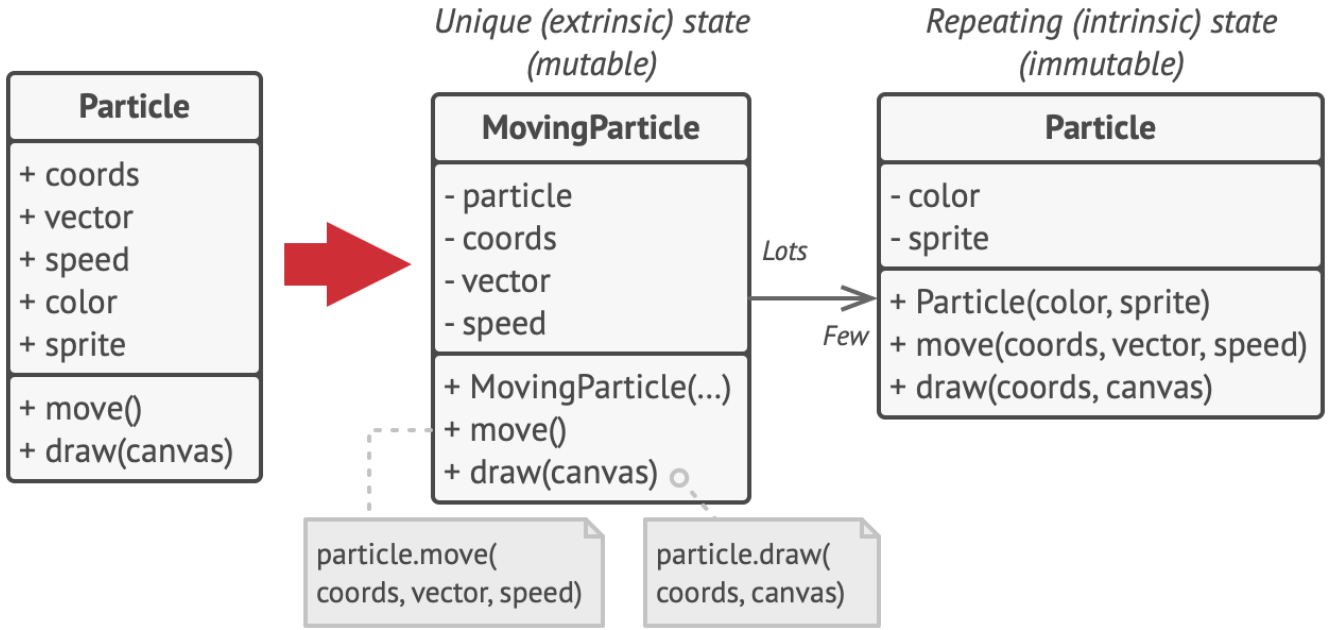
- **A video game: players moving around a map and shooting each other**
- Implementing a realistic particle system: bullets, missiles and shrapnel



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

Flyweight: Solution

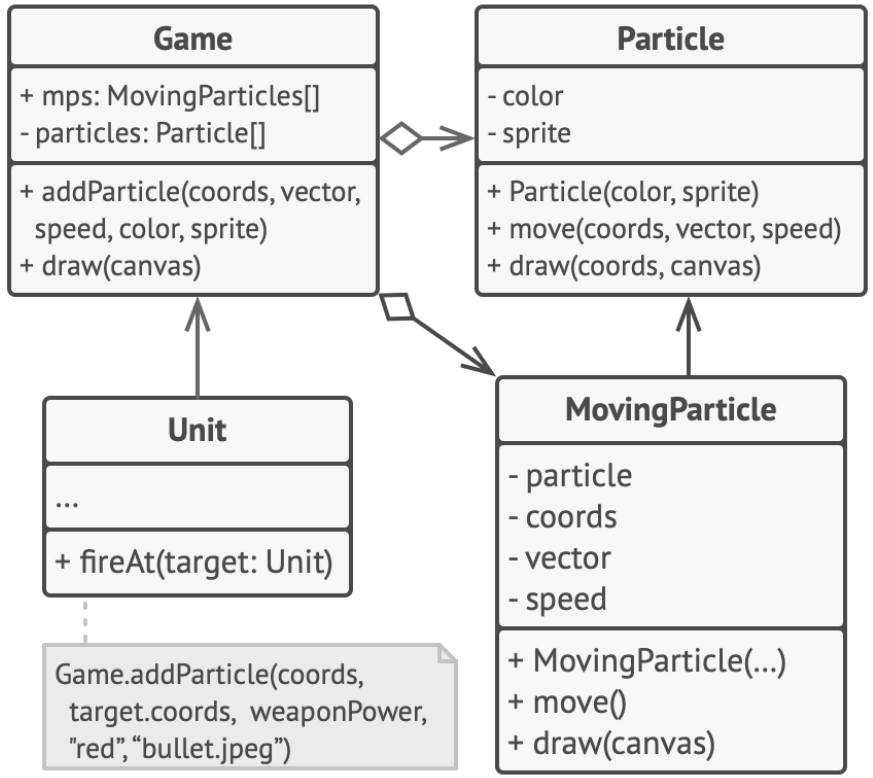
- **Problem analysis**
  - Some fields consume more memory than others, and store almost **identical** data
  - Other parts are **unique** to each instance, and the values **change** over time
  - **Intrinsic state and extrinsic state**
- **Solution**
  - The **Flyweight** pattern (aka Cache)

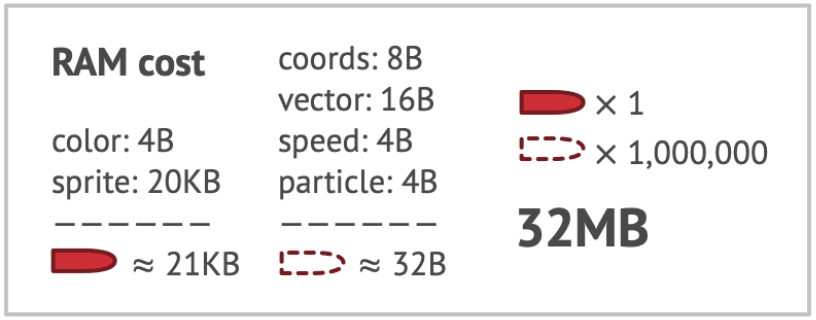


- Stop storing the extrinsic state inside the object
- Only intrinsic state stays within the object, for supporting reuse

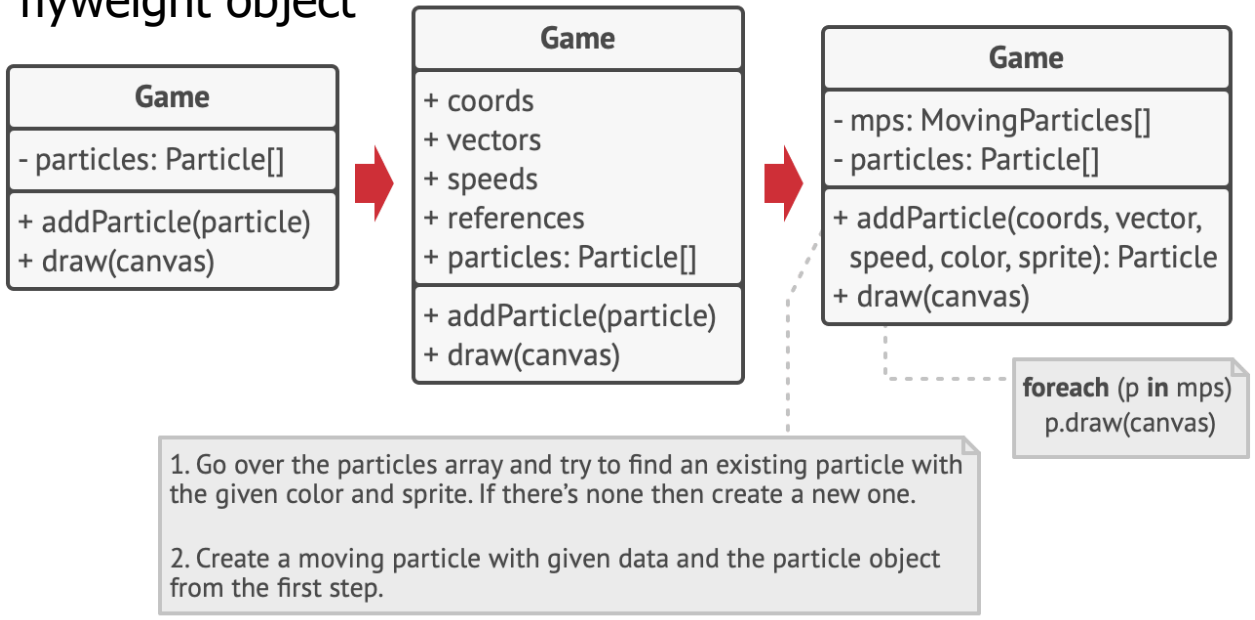
[Figures in this slide are extracted from https://refactoring.guru/design-patterns/flyweight]

Flyweight: Solution (cont.)





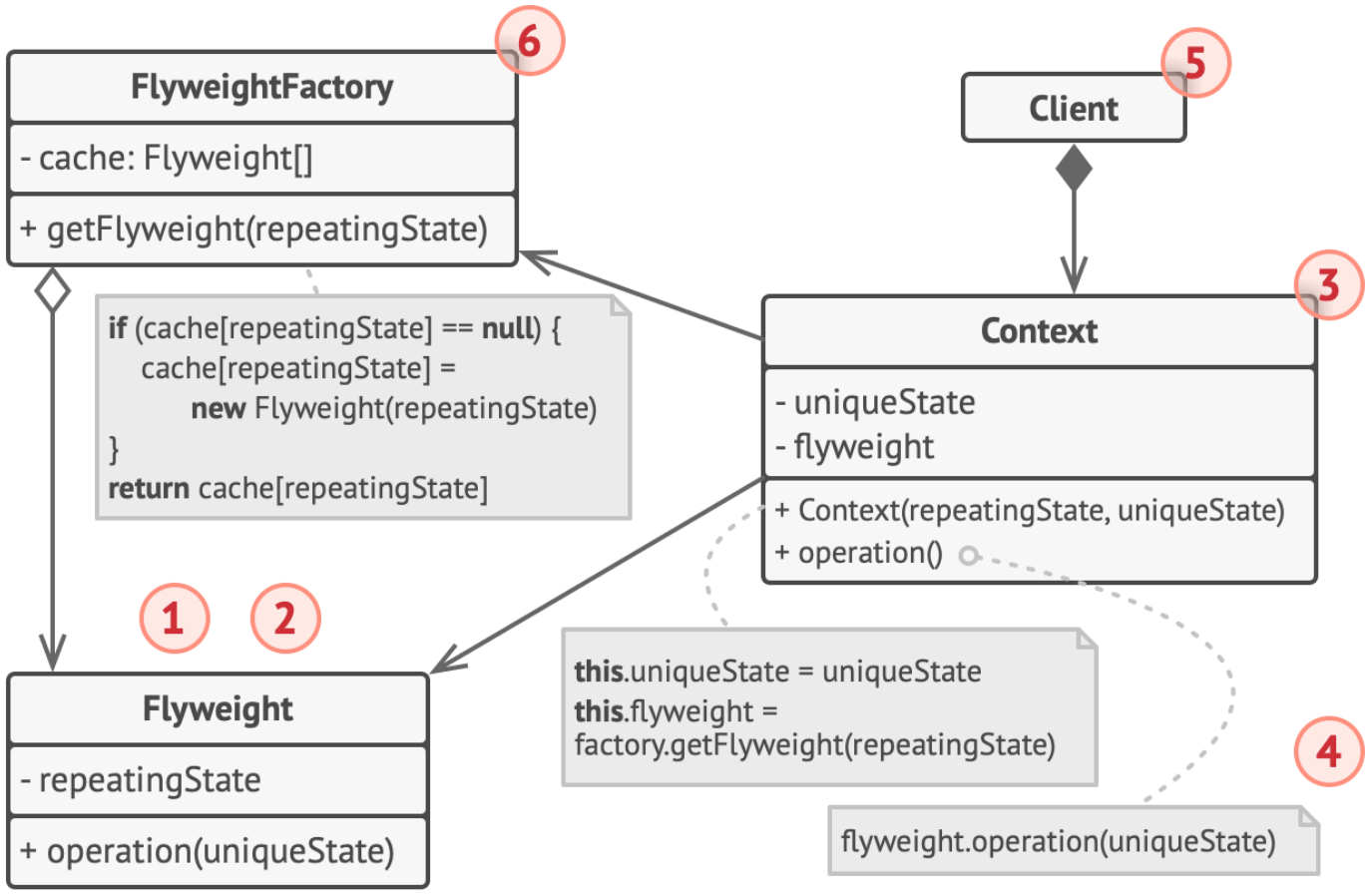
- **Extrinsic state storage**
  - The container object stores fields for extrinsic state
  - A better solution: create **a separate context class** that would store the extrinsic state along with reference to the flyweight object



- **Flyweight and immutability**
- **Flyweight factory**

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

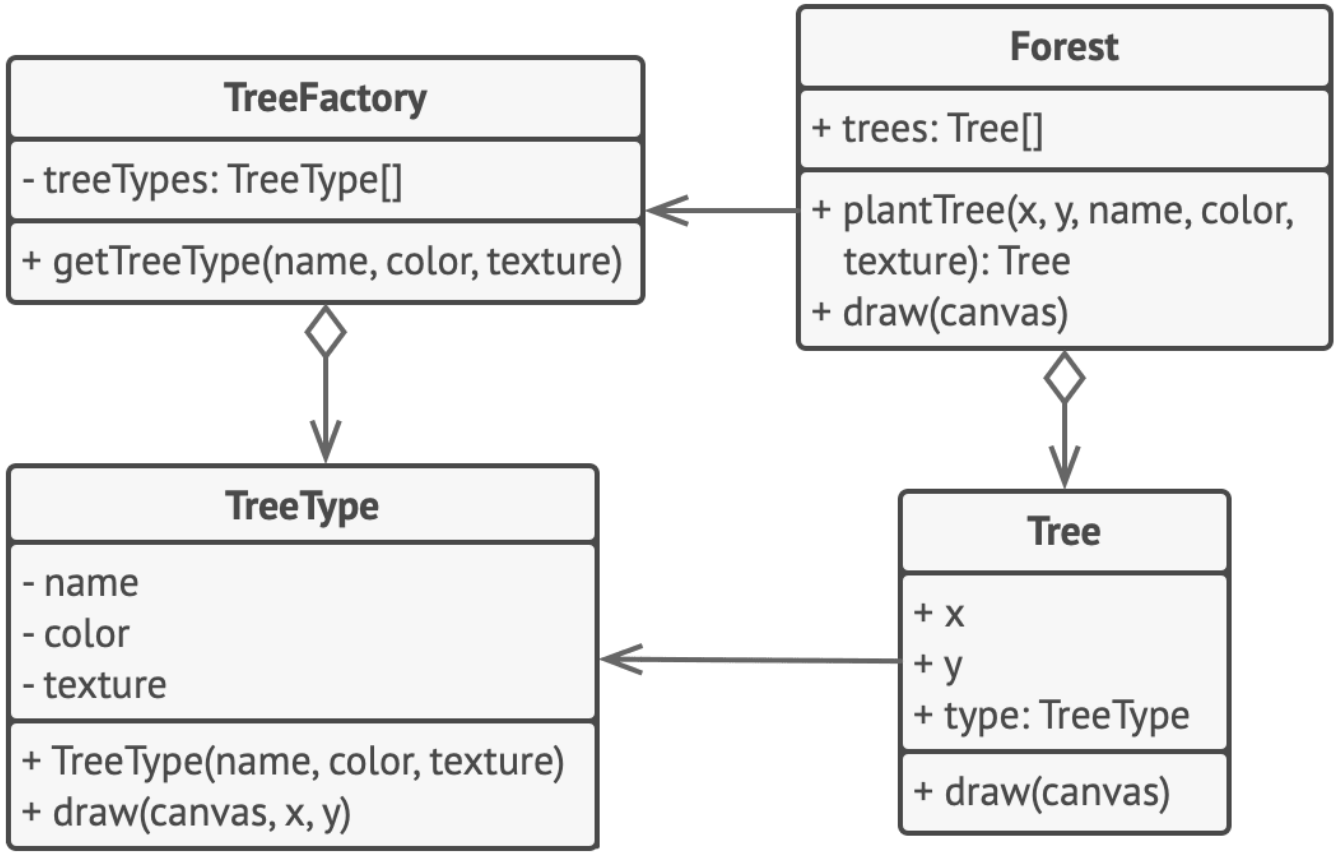
Flyweight: Structure



- Flyweight pattern is an **optimization**
- A context paired with one flyweight object represents the **full state** of the original object
- **Behavior** of the original object
  - Usually remained in the flyweight
  - May also be moved to the context

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

Flyweight: Example



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

Flyweight: Applicability

- **Only** when your program must support a huge number of objects which barely fit into available RAM
- Most useful when:
  - An application needs to spawn a huge number of similar objects
  - This drains all available RAM on a target device
  - The objects contain duplicate states which can be extracted and shared between multiple objects

Flyweight: Implementation

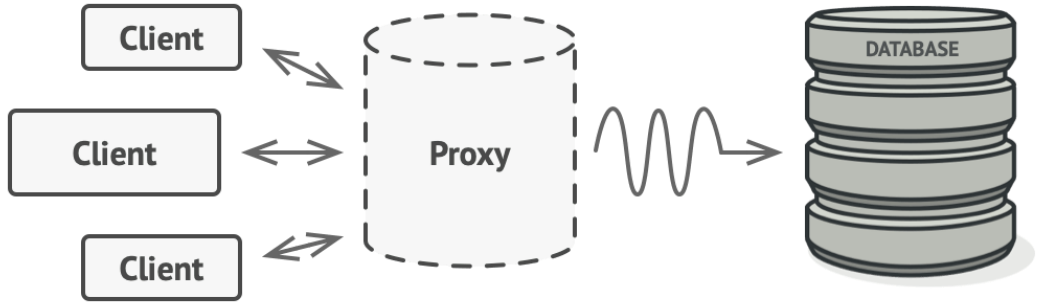
1. Divide fields of a class that will become a flyweight into two parts:
   - The **intrinsic** state: fields that contain unchanging data duplicated across many objects
   - The **extrinsic** state: fields that contain contextual data unique to each object
2. Leave the fields that represent the intrinsic state in the class, and make sure they are **immutable**
   - Initialized only inside the **constructor**
3. Go over methods that use fields of the extrinsic state
   - For each field used, introduce a new parameter and use it instead of the field
4. Optional: create a **factory class** to manage the **pool of flyweights**
5. The client must store or calculate values of the extrinsic state (context) to be able to call methods of flyweight objects
   - The extrinsic state along with the flyweight-referencing field may be moved to a **separate context class**

Flyweight: Pros and Cons

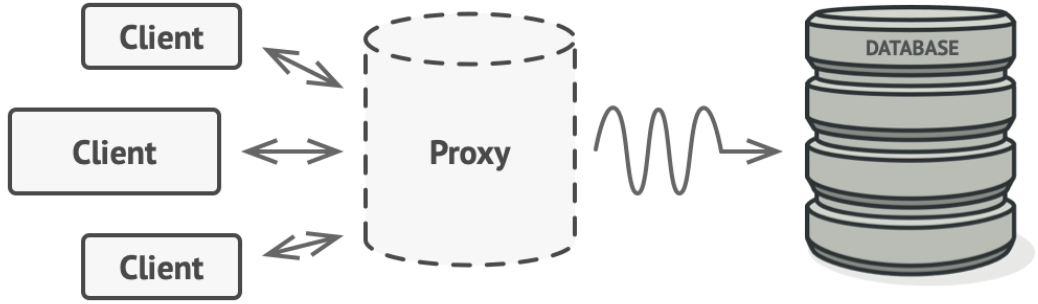
- **Pros**
  - Saving lots of RAM
- **Cons**
  - Might be trading RAM over CPU cycles, if some of the context data needs to be recalculated each time when a flyweight method is called
  - Code becomes much more complicated: the state of an entity is separated

Proxy: Problem and Solution

- **Problem**
  - A massive object consuming a vast amount of resources, needed from time to time, but not always

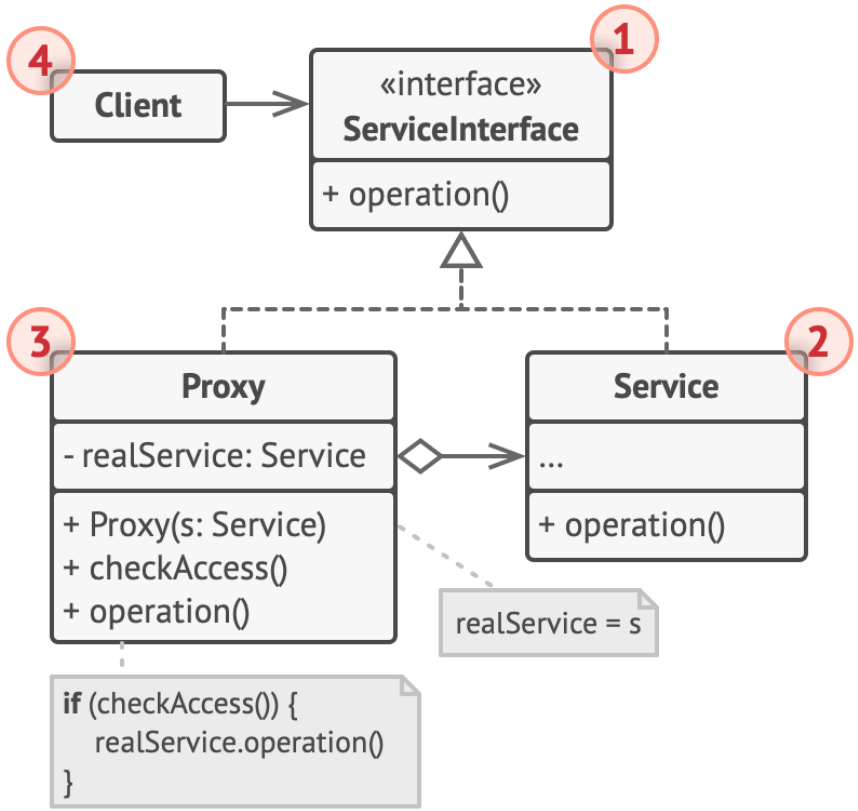


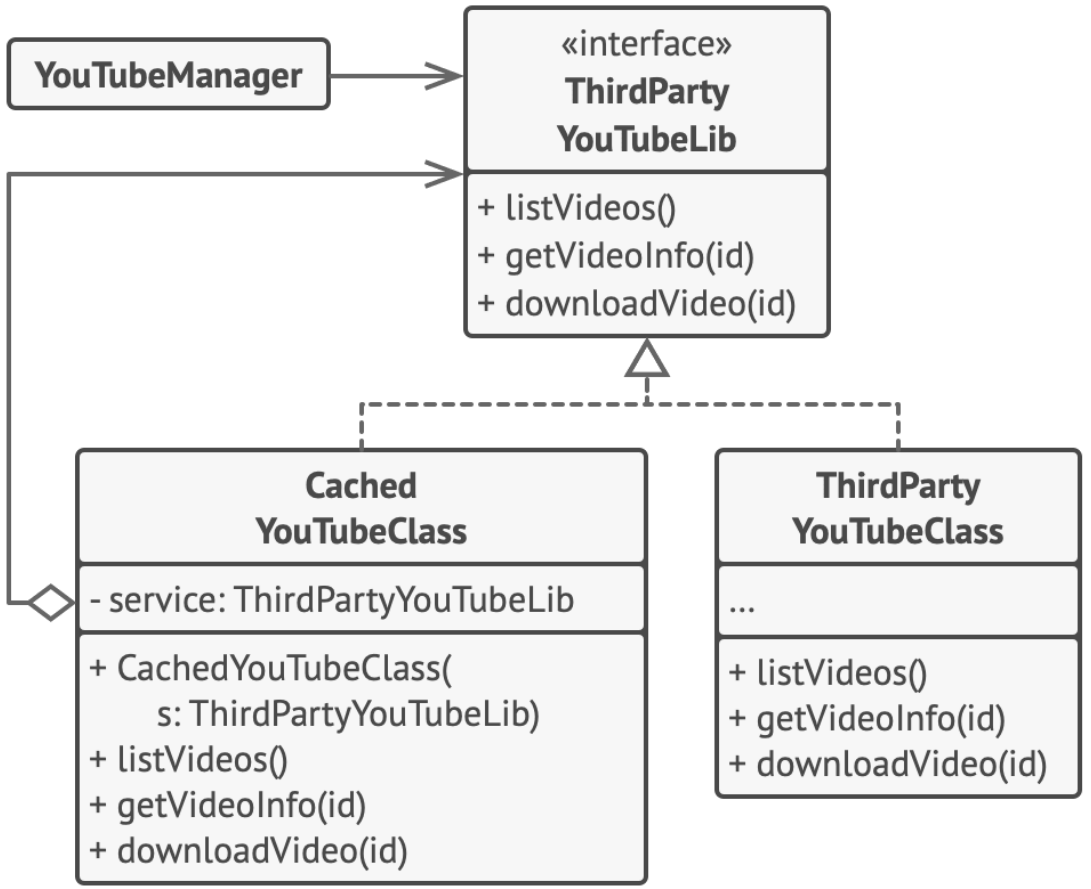
- **Solution**
  - **The Proxy pattern:** a new class with the **same interface** as an original service object
  - Benefit: executing additional tasks before/after the primary logic



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

Proxy: Structure and Example





- Usually, proxies manage the full lifecycle of their service objects

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

Proxy: Applicability

- **Lazy initialization (virtual proxy):** a heavyweight service wastes system resources by being always up, needed from time to time
- **Access control (protection proxy):** only specific clients are able to use the service
- **Local execution of a remote service (remote proxy):** the service object is located on a remote server
- **Logging requests (logging proxy):** keep a history of requests to the service object
- **Caching request results (caching proxy):** cache results of client requests and manage the life cycle of this cache
- **Smart reference:** dismiss a heavyweight object once there is no client that uses it

Proxy: Implementation

1. If there is no pre-existing **service interface**, create one
   - Plan B: make the proxy a subclass of the service class
2. Create the **proxy class**, with a field for storing a reference to the service
   - Usually, proxies create and manage the whole life cycle of their services
   - On rare occasions, a service is passed to the proxy via a constructor
3. Implement the **proxy methods** according to their purposes
4. Consider introducing a **creation method** that decides whether the client gets a proxy or a real service
5. Consider implementing **lazy initialization** for the service object

Proxy: Pros and Cons

- **Pros**
  - Control the service object without clients knowing about it
  - Manage the lifecycle of the service object when clients do not care about it
  - The proxy works even if the service object is not ready or is not available
  - Open/Closed Principle: introducing new proxies without changing the service or clients
- **Cons**
  - The code may become more complicated
  - The response from the service might get delayed