# Unidade Curricular

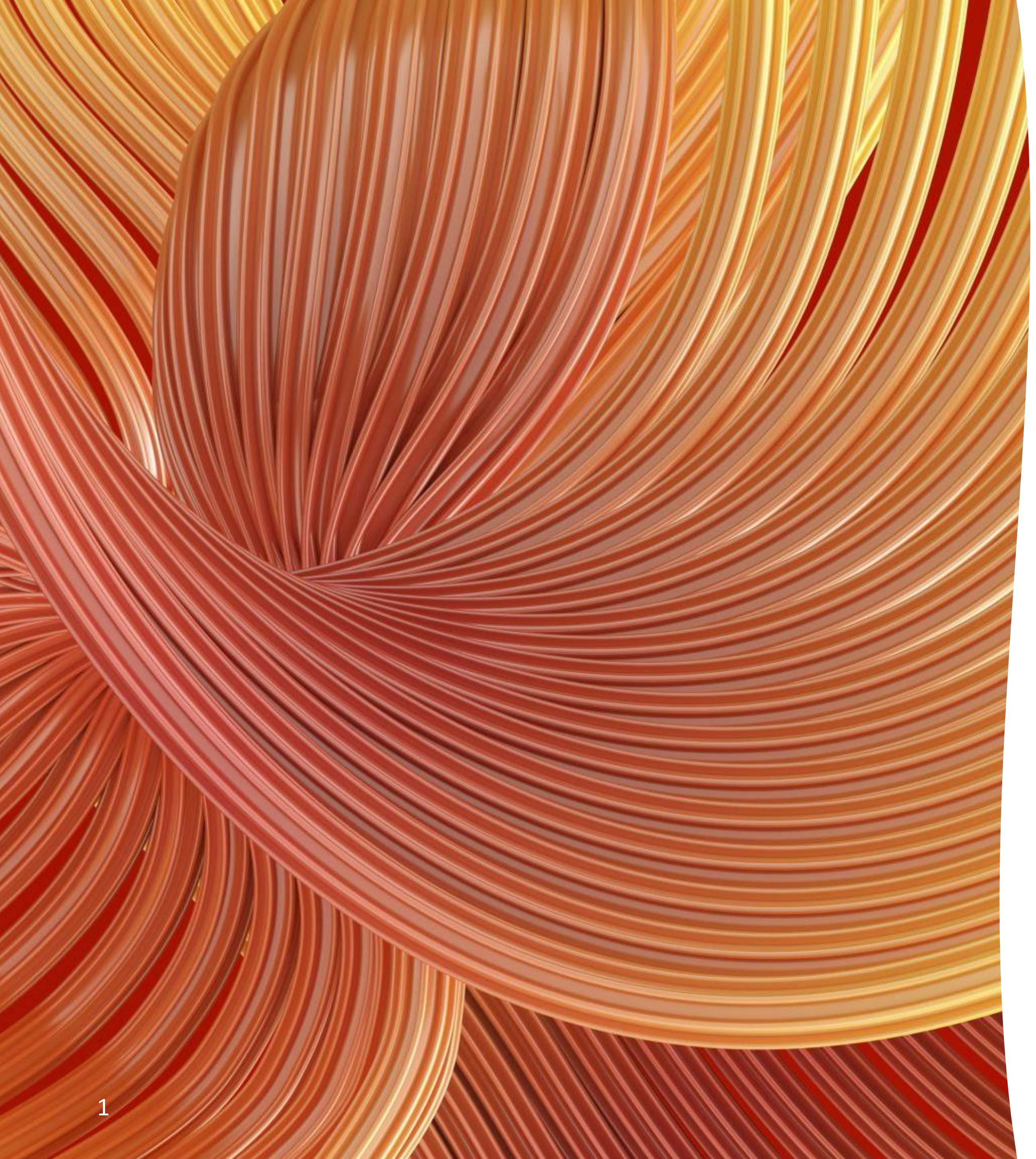
# “Padrões e Desenho de Software”

# #09 – Structural Patterns (3)

António José Ribeiro Neves

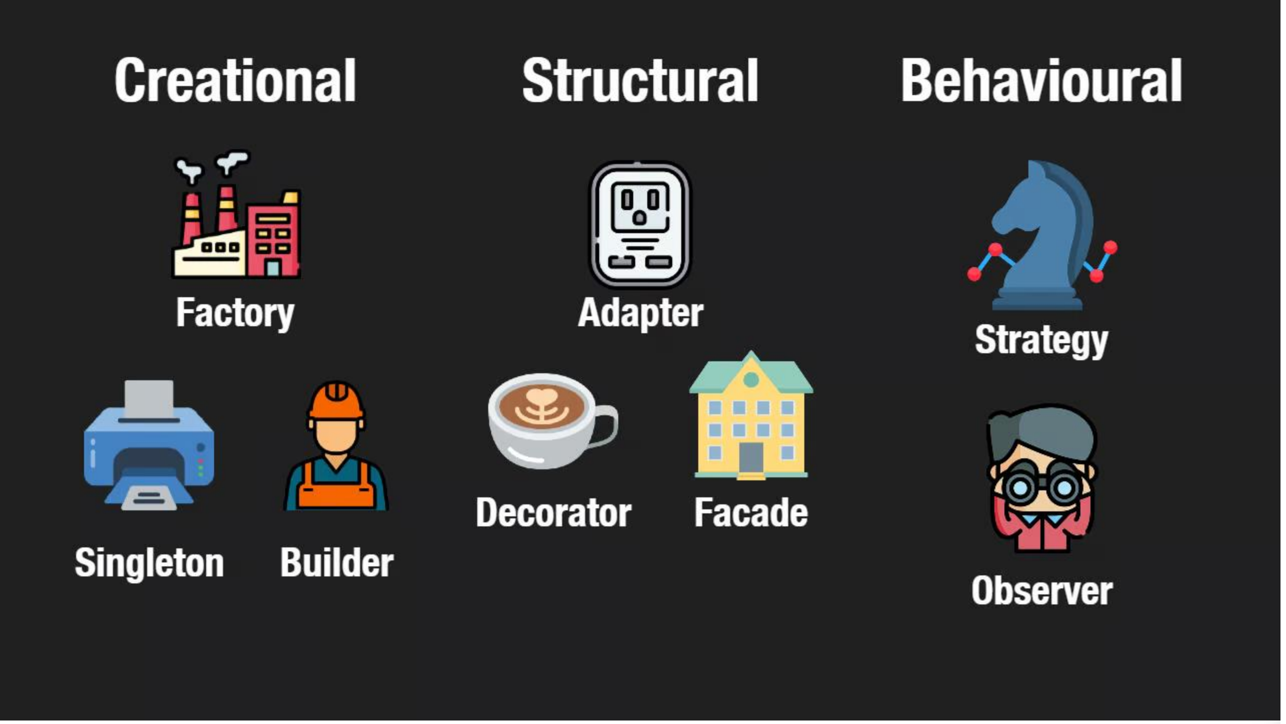
an@ua.pt

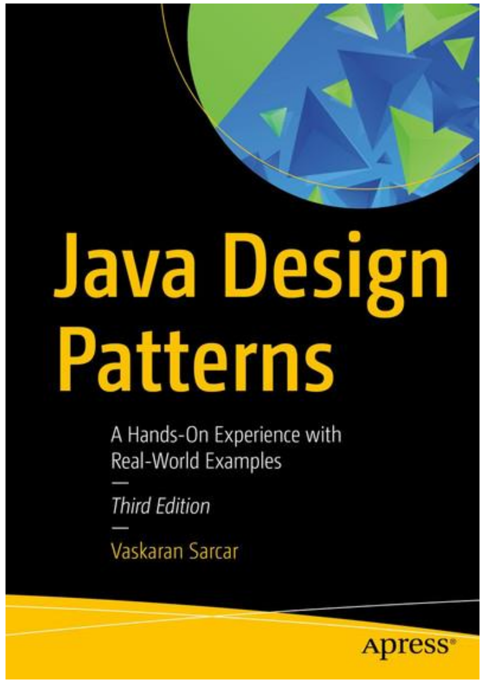
https://www.ua.pt/pt/uc/12275



## Outline

- Façade Pattern
- Flyweight Pattern
- Proxy Pattern





# Structural Design Patterns

**15 minutes Book Activity**

Explore the Façade Design Pattern in the following book and website:

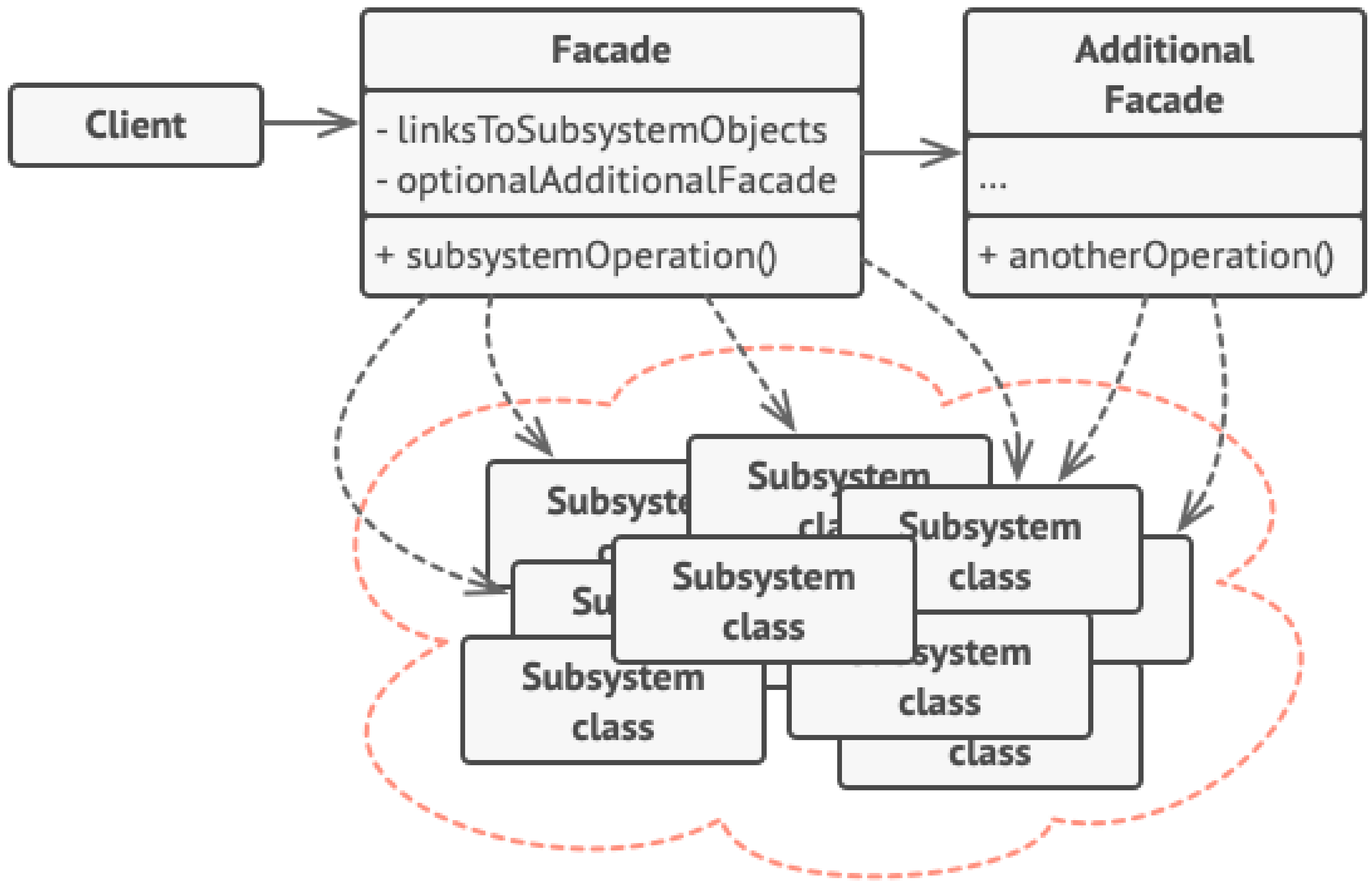
https://bit.ly/47m3nu8

**and**

https://refactoring.guru/design-patterns/facade

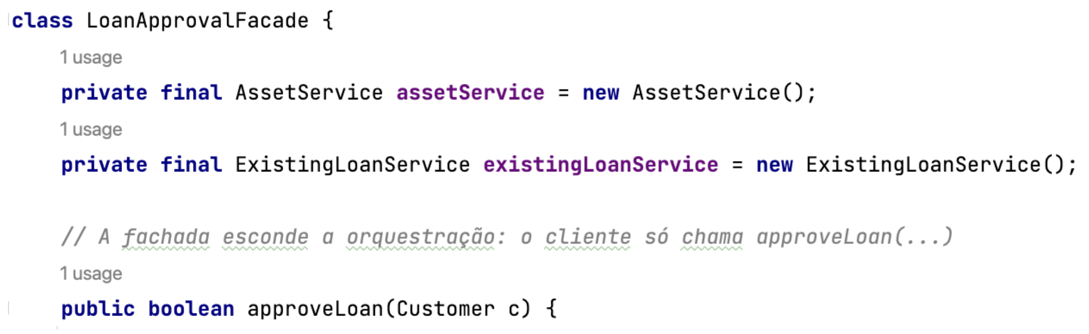
# Facade

- Structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes
- Provide limited functionality in comparison to working with the subsystem directly
- It includes only those features that clients really care about



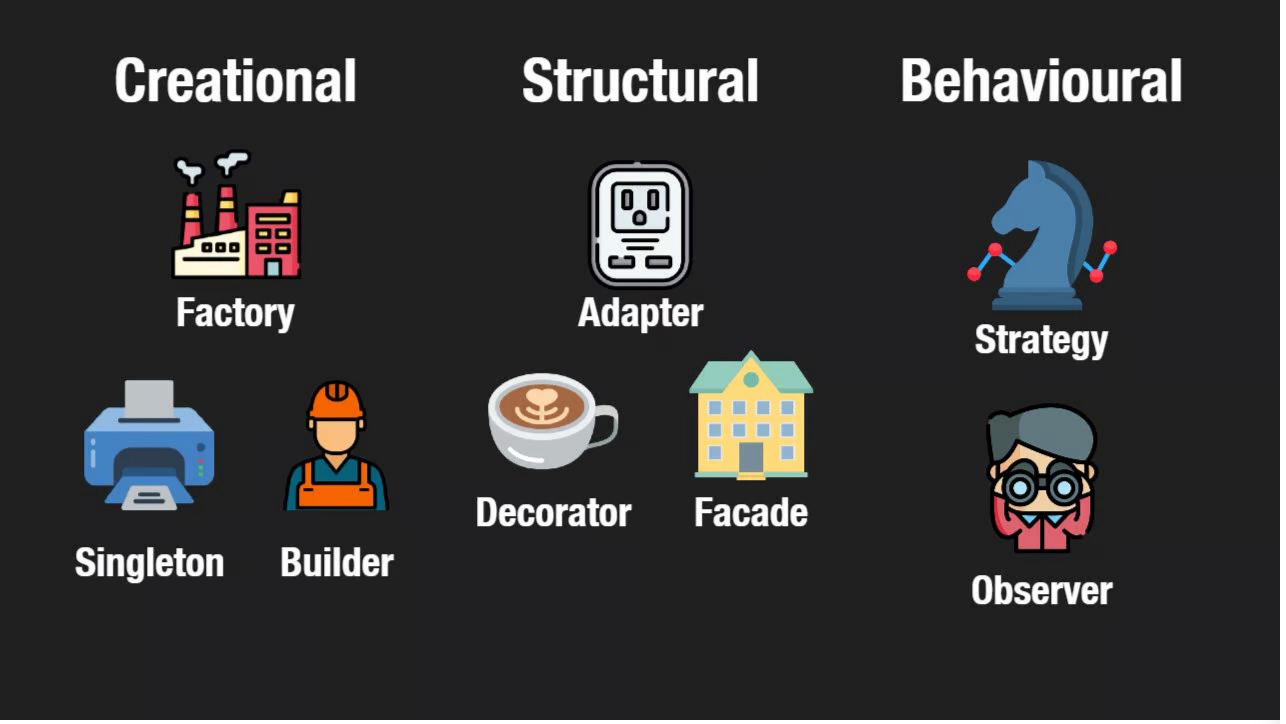
# How a Facade is typically implemented

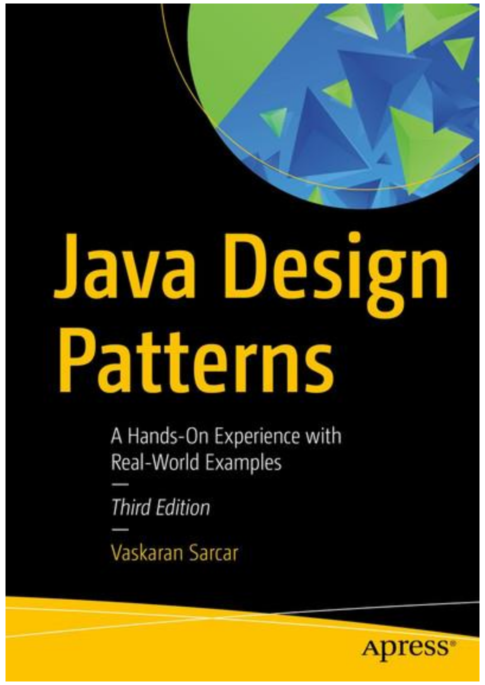
- The Facade class holds references to one or more subsystem classes that perform the actual work.
- The Facade's public methods (e.g., approveLoan(customer)) define high-level operations that combine multiple subsystem calls into a single, simple action.
- The client interacts only with the Facade, never directly with the underlying subsystem classes - this reduces coupling and hides implementation complexity.
- Each subsystem remains independent and unaware of the Facade - the Facade simply coordinates their use.
- The Facade can be implemented as:
    - an object with instance references to subsystems (most common), or
    - a static utility class exposing high-level methods when no state is required.
- The Facade doesn't add new logic — it only organizes and orchestrates existing logic in a convenient You can have multiple Facades in large systems, each covering a different domain.



```
class LoanApprovalFacade {
    1 usage
    private final AssetService assetService = new AssetService();
    1 usage
    private final ExistingLoanService existingLoanService = new ExistingLoanService();

    // A fachada esconde a orquestração: o cliente só chama approveLoan(...)
    1 usage
    public boolean approveLoan(Customer c) {
```





**15 minutes Book Activity**

# Structural Design Patterns

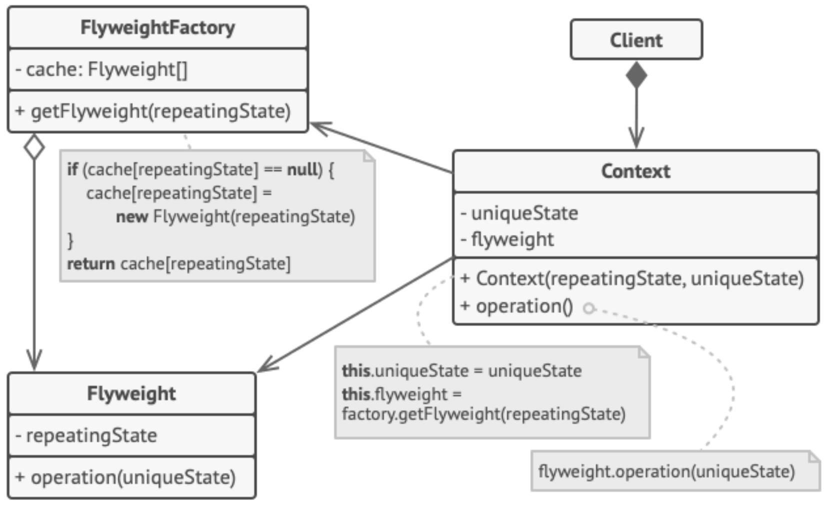
Explore the Flyweight Design Pattern in the following book and website:

https://bit.ly/4hIIbC8

**and**

https://refactoring.guru/design-patterns/flyweight

# Flyweight



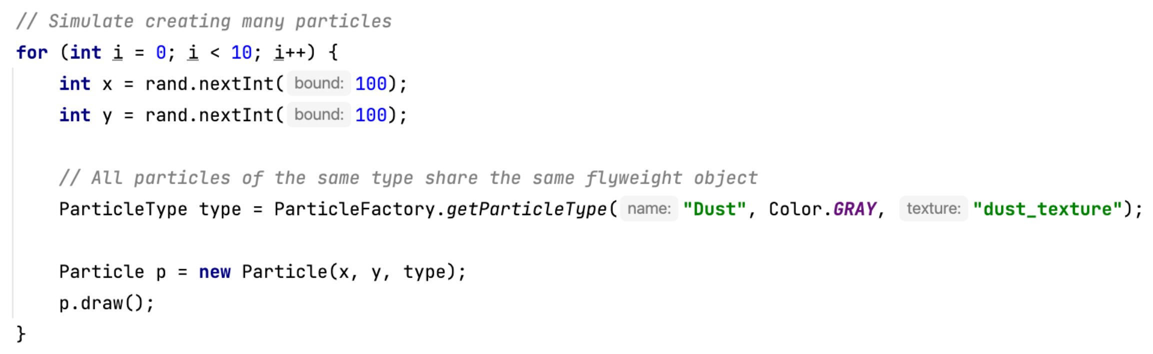
```
if (cache[repeatingState] == null) {
    cache[repeatingState] =
        new Flyweight(repeatingState)
}
return cache[repeatingState]
```

```
this.uniqueState = uniqueState
this.flyweight =
factory.getFlyweight(repeatingState)
```

```
flyweight.operation(uniqueState)
```

- It is a structural design pattern that 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.
- Constant data of an object is usually called the intrinsic state - it lives within the object; other objects can only read it, not change it.
- The rest of the object’s state, often altered “from the outside” by other objects, is called the extrinsic state.
- The Flyweight pattern suggests that you stop storing the extrinsic state inside the object.
- We pass this state to specific methods which rely on it. Only the intrinsic state stays within the object, letting you reuse it in different contexts.
- As a result, we need fewer of these objects since they only differ in the intrinsic state, which has much fewer variations than the extrinsic.

# How the Flyweight is typically implemented



```java
// Simulate creating many particles
for (int i = 0; i < 10; i++) {
    int x = rand.nextInt( bound: 100);
    int y = rand.nextInt( bound: 100);

    // All particles of the same type share the same flyweight object
    ParticleType type = ParticleFactory.getParticleType( name: "Dust", Color.GRAY,  texture: "dust_texture");

    Particle p = new Particle(x, y, type);
    p.draw();
}
```

- Identify which parts of the object's state are intrinsic (shared) and which are extrinsic (unique) - ( e.g., ParticleType holds shared data (color, texture), while Particle keeps position (x, y).
- Extract the intrinsic state into a Flyweight class - immutable, shared by many objects.
- Keep the extrinsic state outside the flyweight (in the context object or passed as parameters).
- Introduce a Factory (FlyweightFactory) that:
    - Creates and stores Flyweight objects (usually in a Map or HashMap);
    - Returns an existing flyweight when a request for the same intrinsic data appears;
    - Ensures that identical intrinsic states are reused instead of duplicated.
- The client never instantiates flyweights directly - it always requests them from the factory.
- The Flyweight objects should be immutable - since they're shared, they must not allow changes to intrinsic state.
- Often combined with other patterns:
    - Factory Method (to manage flyweight creation)
    - Singleton (to manage the factory instance)

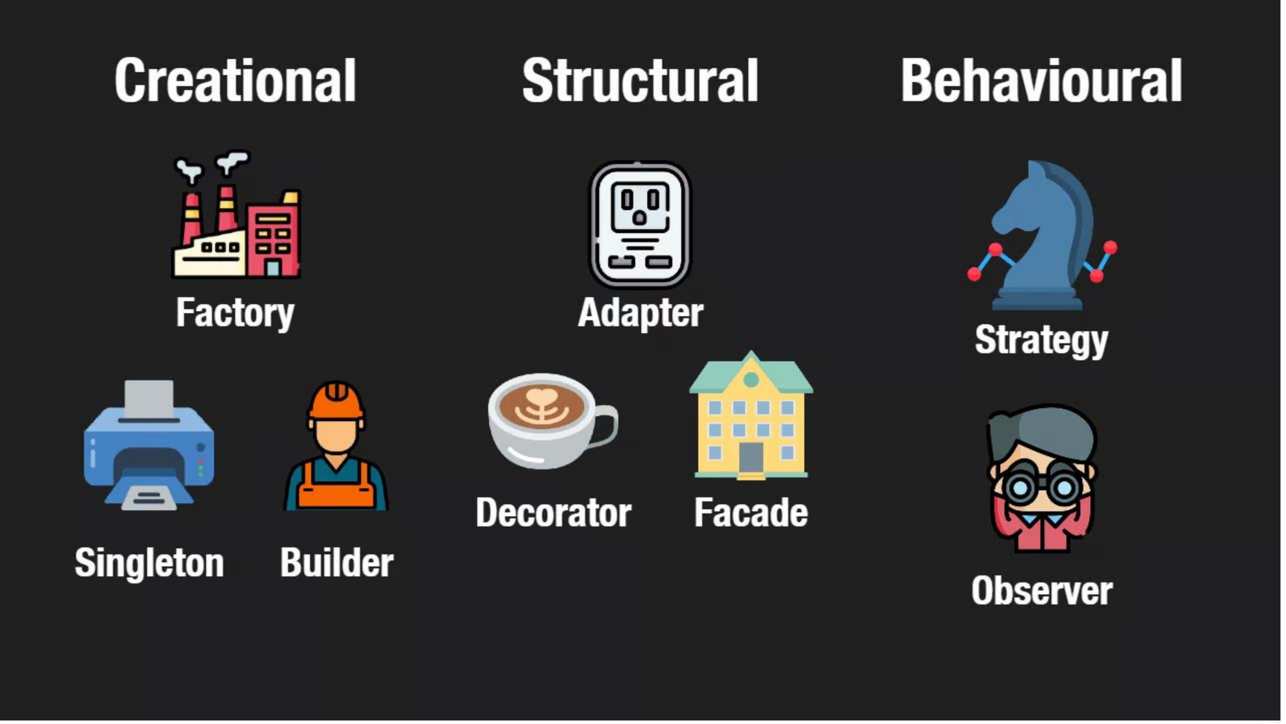
## Let’s take a short break

**10 Minutes**

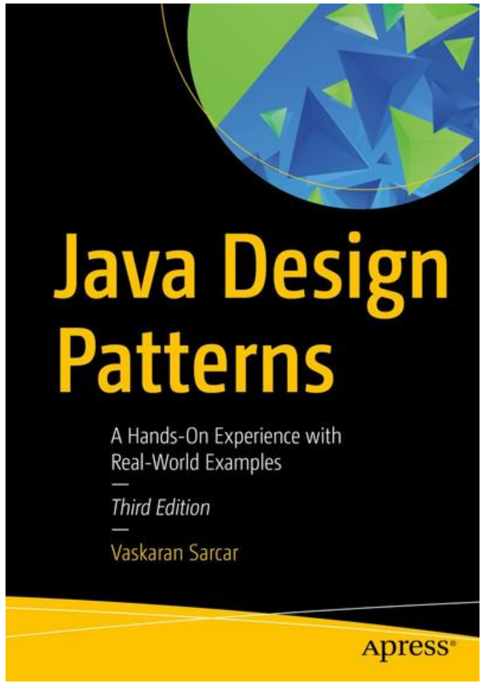
You are free to go grab
a coffee, water, etc.

But… 10 minutes **is 10 minutes** (600 seconds, **not 601 seconds!)**

10 minutes



# Structural Design Patterns



**15 minutes Book Activity**

Explore the Proxy Design Pattern in the following book:

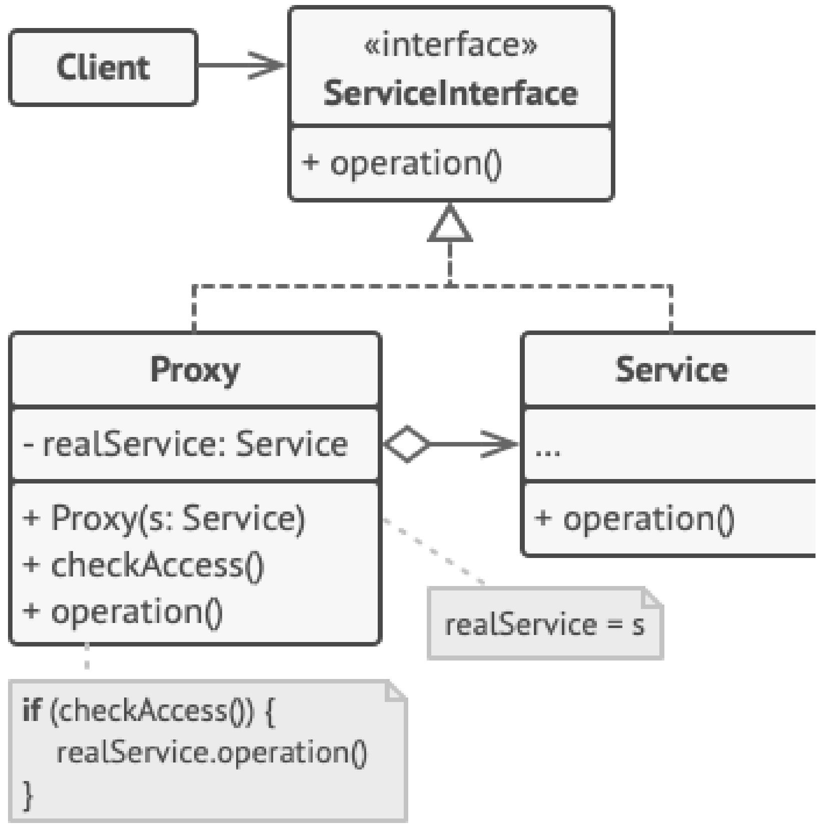
https://bit.ly/49AQcH8

**and**

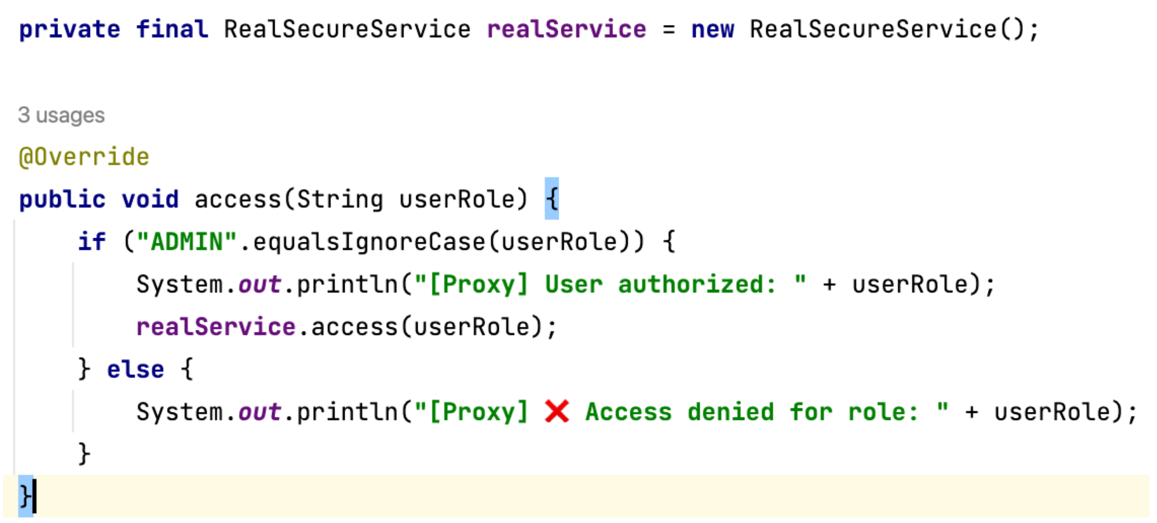
https://refactoring.guru/design-patterns/proxy

# Proxy

- It is a structural design pattern that 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.



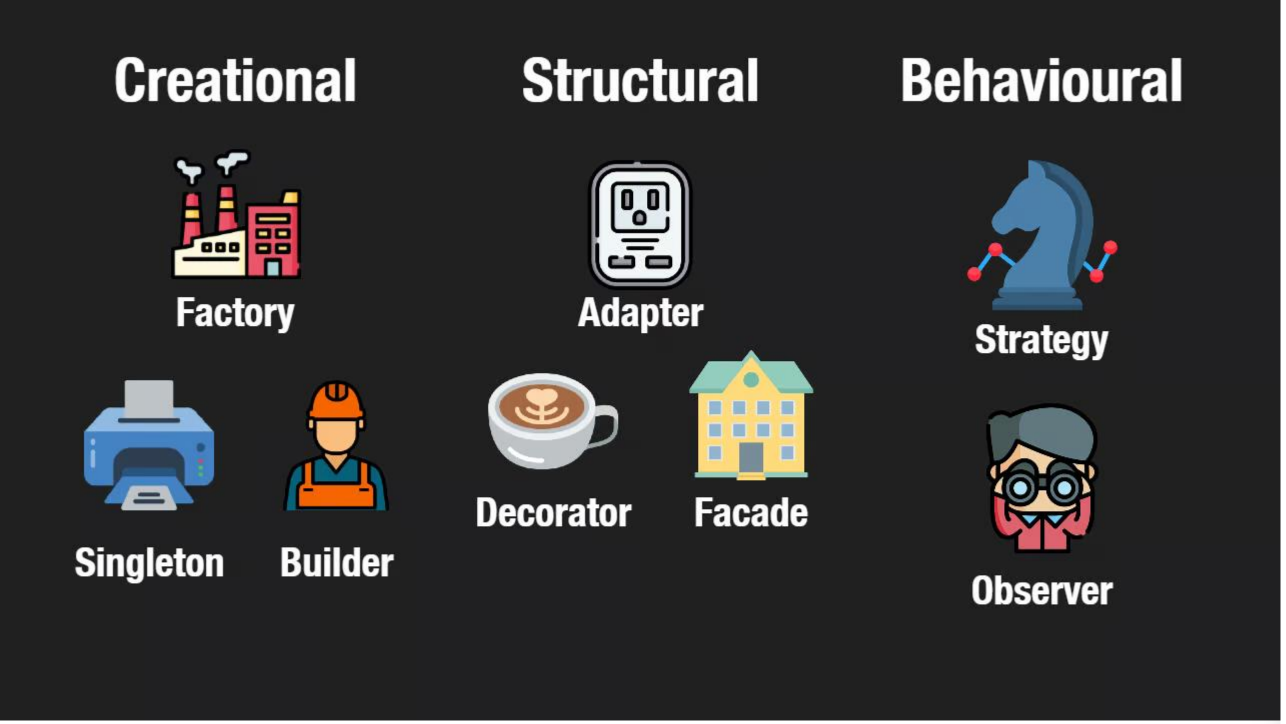
## How the Proxy is typically implemented

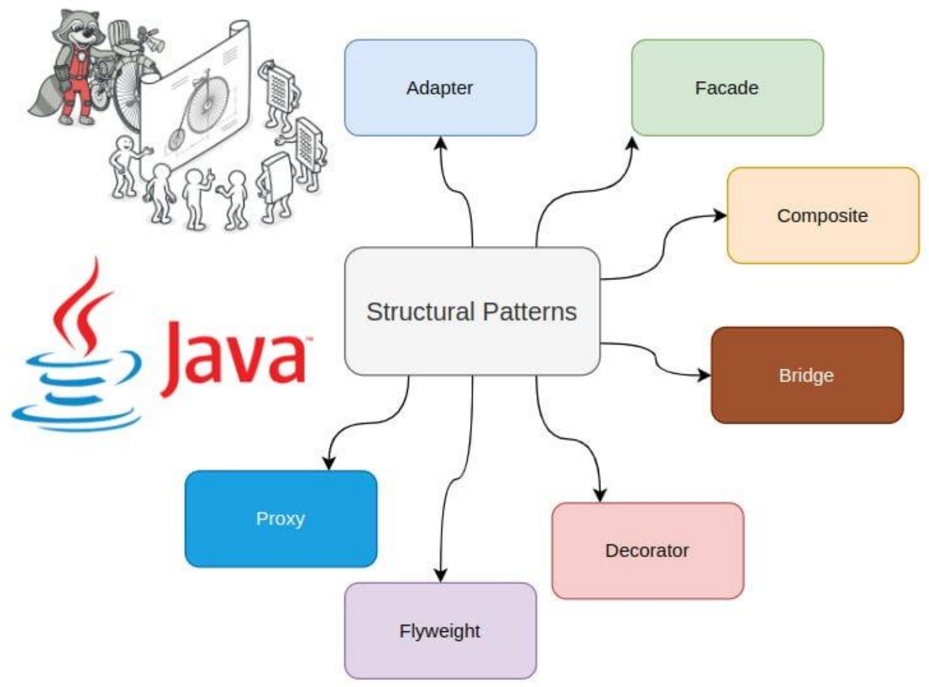


```
private final RealSecureService realService = new RealSecureService();

3 usages
@Override
public void access(String userRole) {
    if ("ADMIN".equalsIgnoreCase(userRole)) {
        System.out.println("[Proxy] User authorized: " + userRole);
        realService.access(userRole);
    } else {
        System.out.println("[Proxy] ❌ Access denied for role: " + userRole);
    }
}
```

- Define a common interface for both the Real Subject (the actual object) and the Proxy - this allows the client to interact with either one transparently.
- The Proxy class holds a reference to the Real Subject and implements the same interface.
- In its methods, the Proxy:
  - Performs pre-processing (e.g., access checks, logging, lazy initialization, caching, etc.);
  - Delegates the actual work to the real object (realSubject.method());
  - Optionally performs post-processing after delegation.
- The client code communicates only with the Proxy, unaware whether it's working with the real object or not.
- The Real Subject remains unchanged - the proxy adds behavior without modifying existing code.
- Proxies can be instantiated lazily, creating the real object only when needed.





# Structural Design Patterns

- **20 minutes** to answer the questions in the link:

https://forms.gle/BpJSyjRkpxyW9zta9