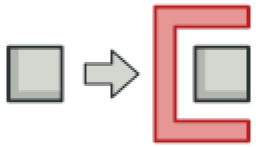
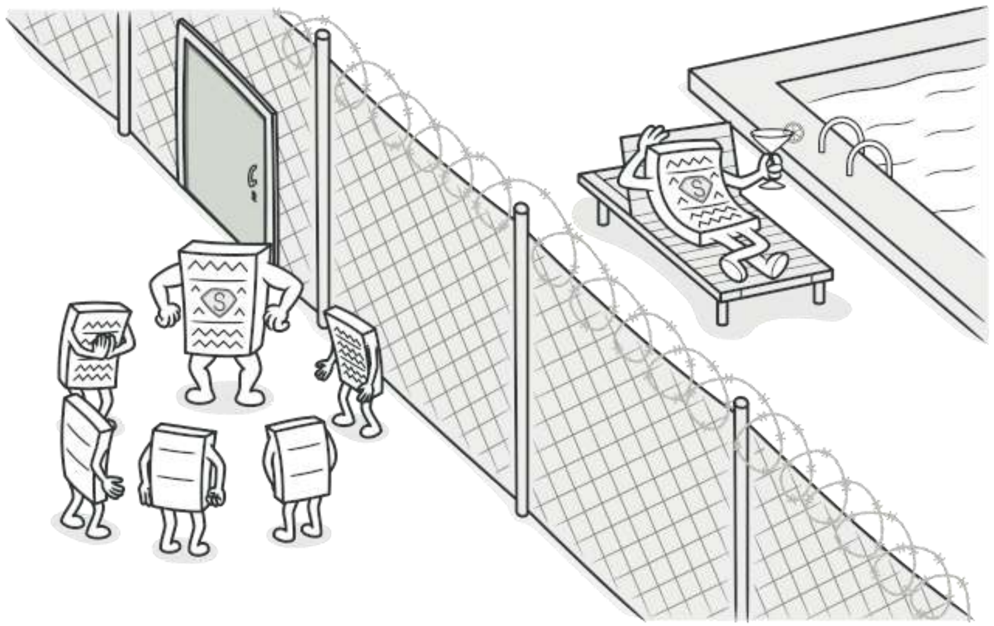
Proxy



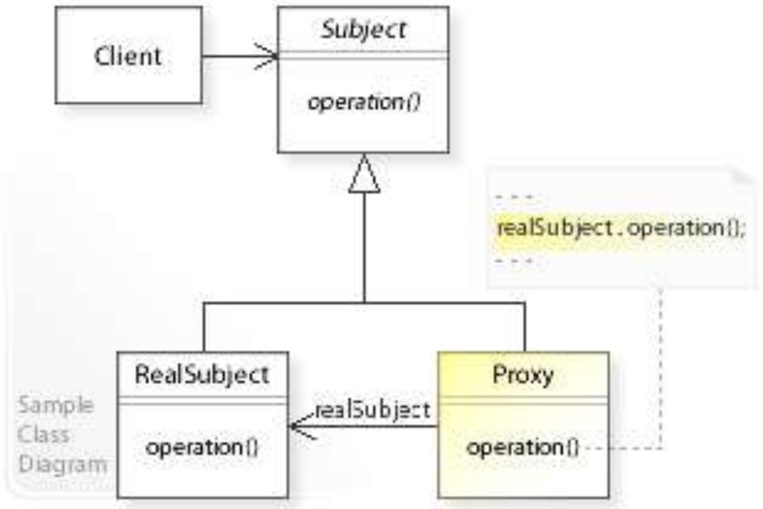
Proxy: What is it?

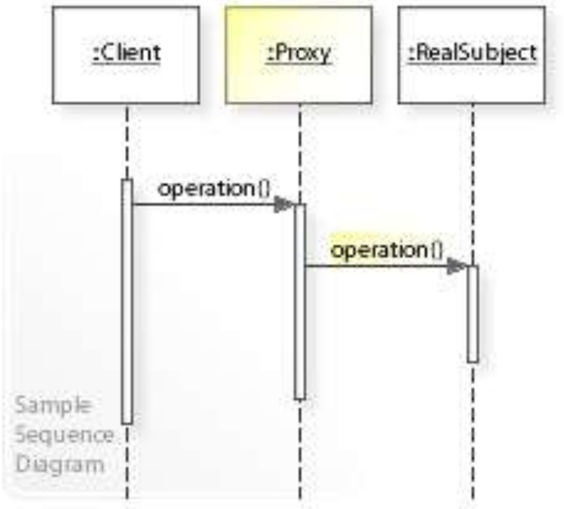
- **Object Structural** pattern
- Alternative name: Surrogate
- Base ideas:
  - A “middleman” or substitute when working with an object
  - Has (almost) the same interface as the original object
  - Has (almost) the same observable behavior as the original object
  - Code using the proxy (usually) doesn’t even know it is a proxy
  - Some additional behavior/checks before and/or after the action



Basic principle

- Proxy and the original object share same interface
- Proxy relays calls to the original object and returns the results
- User doesn't know if they have the "real thing" or just a proxy



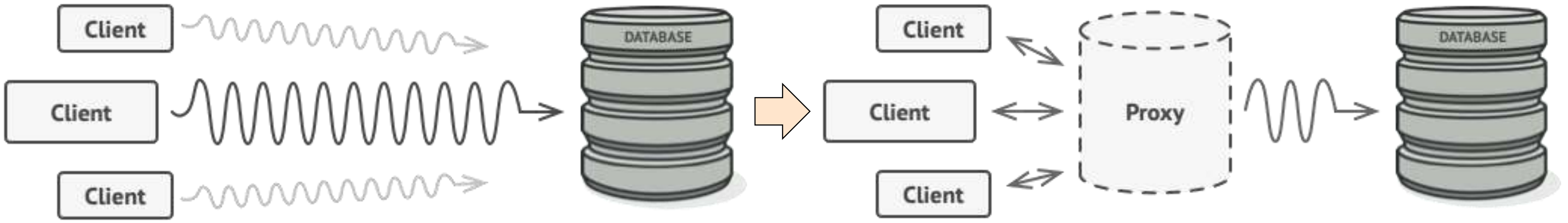


Proxy: Common use-cases

- Real-world analogies:
  - Spokesperson relays questions to their boss
  - Ambassador represents a country
- Common kinds of proxies:
  - Remote proxy – Local representation for an object that is somewhere else
  - Virtual proxy – Delays creation/loading of expensive objects
  - Protection proxy – Adds additional checks when accessing the original object
  - Logging proxy – Logs access to the original object
- What if proxy could change the behavior/interface?
  - Smart reference/pointer
  - Mutex synchronizing the access

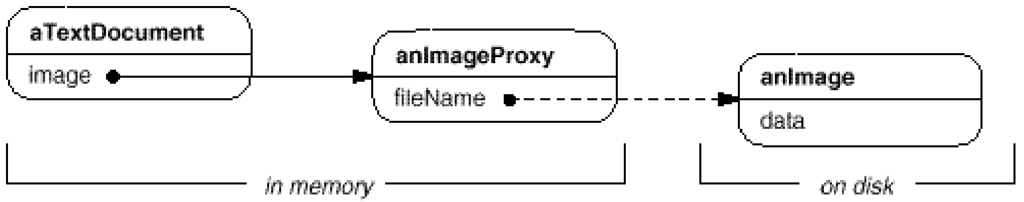
Remote proxy

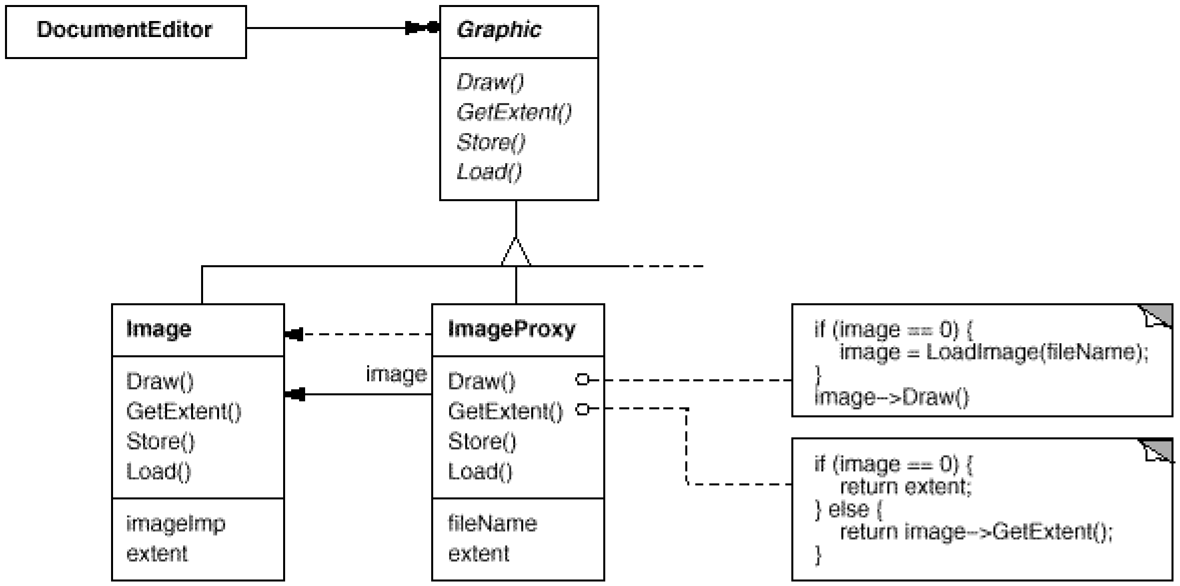
- Handles access to objects in another address space or on another machine
- Can add supporting behavior
  - Lazy initialization
  - Logging
  - Caching
- Examples:
  - "File" object accessing data in a file
  - Database object relaying queries to actual database and caching results



Virtual proxy

- Acts as a lazy initialization wrapper around the object
- Delays doing expensive operations until they are needed
- Example: Loading full image data only when needed





```
if (image == 0) {
    image = LoadImage(fileName);
}
image->Draw()
```

```
if (image == 0) {
    return extent;
} else {
    return image->GetExtent();
}
```

Protection/Logging proxy

- Adds extra actions or checks before and/or after the action

```
interface IDoor {
    abstract void Open(Character c);
    …
}

// The actual door
class Door : IDoor {
    …
}

// Proxy to the door
class DoorProxy : IDoor {
    private IDoor actualDoor;

    override void Open(Character c) {
        if (!c.CanUseHands()) return;

        log($"{c} opened a door!");
        actualDoor.Open(c);
    }
}
```

(almost) Proxy: Smart pointer

- Proxy has extra feature of counting instances that exist
- Exposes the original object directly when asked (or forwards all actions without change)
- Likely has slightly different interface or type

```
// The actual room
class Room {
    …
}

// Pointer to the room
class RoomPointer {
  public:
    RoomPointer(…);
    ~RoomPointer();

    Room* operator->();
    Room& operator*();
}
```

(almost) Proxy: Mutex

- Changes the observable behavior – call might block
- Synchronizes access to the object

```
class IDoor {
  public:
    virtual void Open(Character& c) = 0;
    …
}

// The actual door
class Door : public IDoor { … }

// Proxy to the door
class ThreadSafeDoor : IDoor {
  private:
    std::mutex mutex;
    IDoor actualDoor;

  public:
    void Open(Character& c) override {
        std::lock_guard lock(mutex);

        actualDoor.Open(c);
    }
}
```

Benefits and drawbacks

- Benefits
  - Separates responsibilities
  - Easily add behavior to existing classes/interfaces
  - Can be completely transparent to the user
- Drawbacks
  - More complex code
  - Adds level of indirection
  - Harder to account for or enforce when working with it

Related design patterns

- Adapter, Facade
  - Both add an indirection level
  - Proxy doesn’t change the interface
- Decorator
  - Can have similar implementation
  - Different purpose: decorator adds new responsibilities, proxy at most limits access