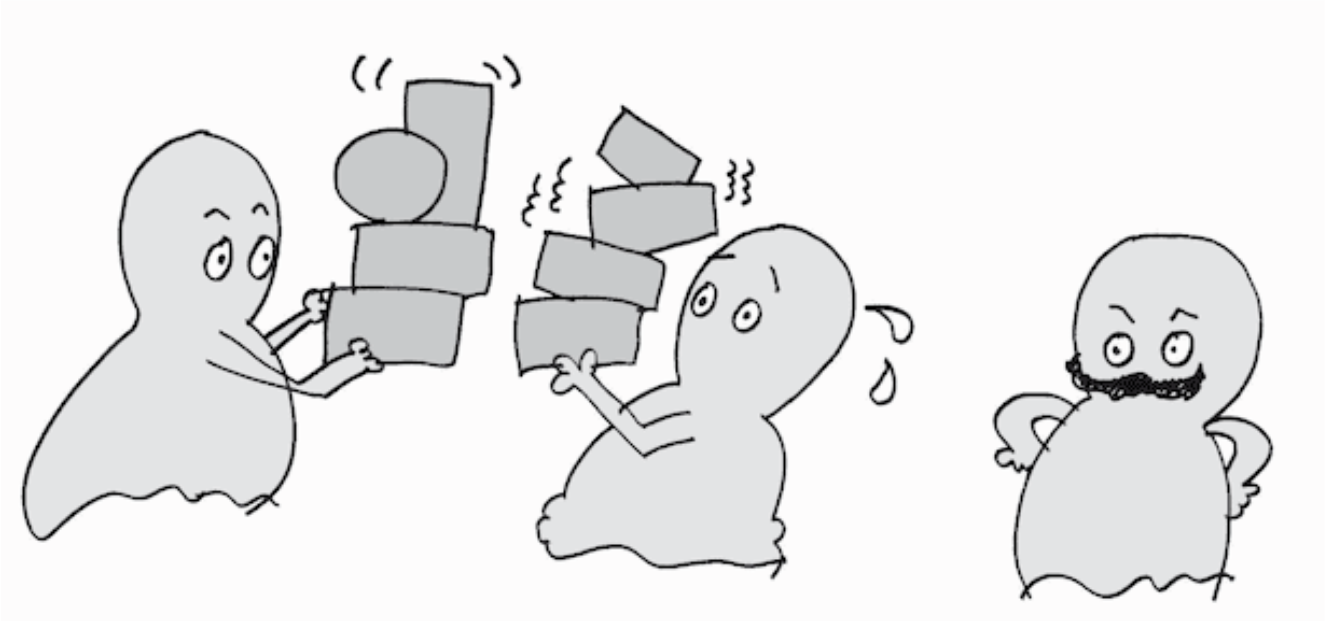
Proxy Pattern

Provide a Placeholder to Control Access to Another Object



Proxy Pattern

Think of a **representative** or **substitute** that controls access to something else:

- **Virtual assistant**: Handles simple requests, escalates complex ones to the manager
- **Bank ATM**: Proxy for bank teller, handles basic transactions
- **Web cache**: Serves cached content, fetches from the server if not available

The Problem

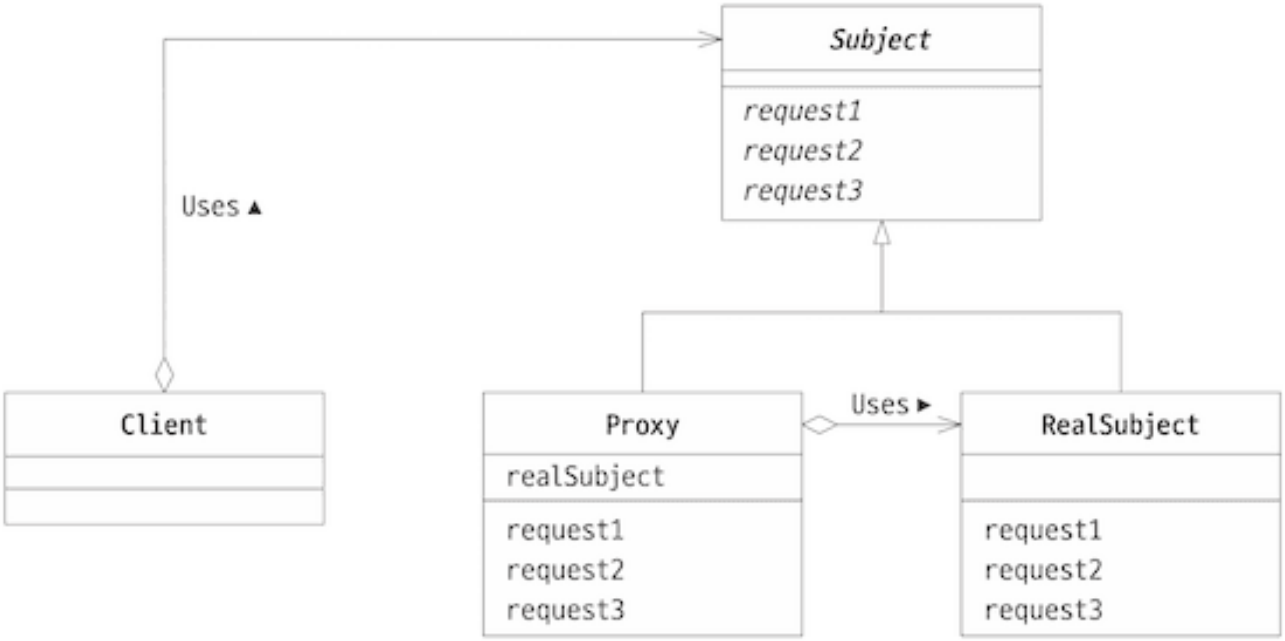
- Printer setup is **expensive** (hardware, drivers).
- Clients may only need **name/status**, not full printing.
- Delay heavy initialization until needed

The challenge: how to provide **immediate access** to object interface while **deferring expensive operations**?

The *Proxy* as the Solution

- **Proxy** provides the same interface as the real object but **controls when** the real object is created.
- **Lightweight operations** handled by proxy directly.
- **Heavy operations** trigger creation of a real object and delegate to it.

The Solution (Design)



Step 1: Understand the Players

In this design, we have key components:

- *Subject* (Printable): Common interface for RealSubject and Proxy

These two players use the Subject interface:

- **RealSubject** (Printer): The real object that the proxy represents and controls access to
- **Proxy** (PrinterProxy): Placeholder that controls access to RealSubject

Clients:

- **Client** uses the *Subject*.

Step 2: Same interface principle

- **Proxy** and **RealSubject** implement same interface.
- **Clients** interact with proxy **transparently**.

Step 3: Understand the Subject Interface

- *Subject* defines **common interface** that both proxy and real subject must implement.
- **Ensures** clients can treat proxy and real subject **uniformly**.
- In our example: `set_printer_name()`, `get_printer_name()`, `print()`.

Step 4: Understand the RealSubject

- **RealSubject** is the **actual object** that proxy represents.
- **Contains** the real business logic and **expensive operations**.
- **Heavy initialization**: Takes time/resources to create.
- **Full functionality**: Implements complete behavior.

Step 5: Understand the Proxy

- **Proxy** acts as **substitute** that controls access to RealSubject.
- **Lazy instantiation**: Creates RealSubject only when necessary.
- **Lightweight operations**: Handled directly without creating RealSubject.
- **Heavy operations**: Delegated to RealSubject after creation.

Code

- Main Method
- Subject Interface
- Proxy Implementation
- RealSubject Implementation

Main Method

The Proxy Pattern provides a placeholder for expensive objects and controls access to them through lazy initialization.

Scenario: Printer objects are expensive to create (simulated with delays).
Solution: Use a proxy that creates the real printer only when needed.

```python
from printer_proxy import PrinterProxy

def main():
    print("Creating printer proxy (fast operation)...")
    proxy = PrinterProxy("LaserJet Pro")

    # Lightweight operations - no real printer needed
    print(f"Name: {proxy.get_printer_name()}")
    proxy.set_printer_name("Updated LaserJet")

    # Heavy operation - now creates real printer
    print("Printing document (creates real printer)...")
    proxy.print("Important document")

    # Subsequent operations reuse the existing real printer
    proxy.print("Another document")
```

Step 1: Create a proxy quickly

```
proxy = PrinterProxy("LaserJet Pro")
```

- **Proxy creation** is **fast** - no expensive initialization.
- **Real object** not created yet, just the proxy placeholder.

Step 2: Use lightweight operations

```
print(f"Name: {proxy.get_printer_name()}")
proxy.set_printer_name("Updated LaserJet")
```

- **Simple operations** handled by **proxy directly**.
- **No real object** created - proxy maintains its own state for simple operations.

Step 3: Trigger heavy operation

```
proxy.print("Important document")  # Real printer created here
```

- **Heavy operation** triggers **creation** of real object.
- **Subsequent operations** reuse the **already created** real object.

Subject Interface

```
from abc import ABC, abstractmethod

class Printable(ABC):  # Subject interface
    @abstractmethod
    def set_printer_name(self, name):
        pass

    @abstractmethod
    def get_printer_name(self):
        pass

    @abstractmethod
    def print(self, string):
        pass
```

Key Points: Subject Interface

1. **Common interface**: Both proxy and real subject implement this
2. **Transparent access**: Clients use the same methods regardless of implementation
3. **Operation definition**: Defines all operations that may be proxied
4. **Polymorphism**: Enables treating proxy and real subject uniformly

Proxy Implementation

```
class PrinterProxy(Printable):
    def __init__(self, name="No Name"):
        self.name = name      # Lightweight state
        self.real = None      # Real subject not created yet

    def get_printer_name(self):
        return self.name      # Handled directly by proxy

    def set_printer_name(self, name):
        self.name = name      # Update proxy state
        if self.real is not None:  # Update real object if exists
            self.real.set_printer_name(name)

    def print(self, string):
        self._realize()       # Create real object if needed
        self.real.print(string)  # Delegate to real object
```

Step 4: Lazy realization

```
def _realize(self):
    if self.real is None:
        print("Creating expensive real printer...")
        self.real = Printer(self.name)  # Heavy operation here
```

- **Realize method** creates real object **only when needed**.
- **Expensive operations** deferred until necessary.

RealSubject Implementation

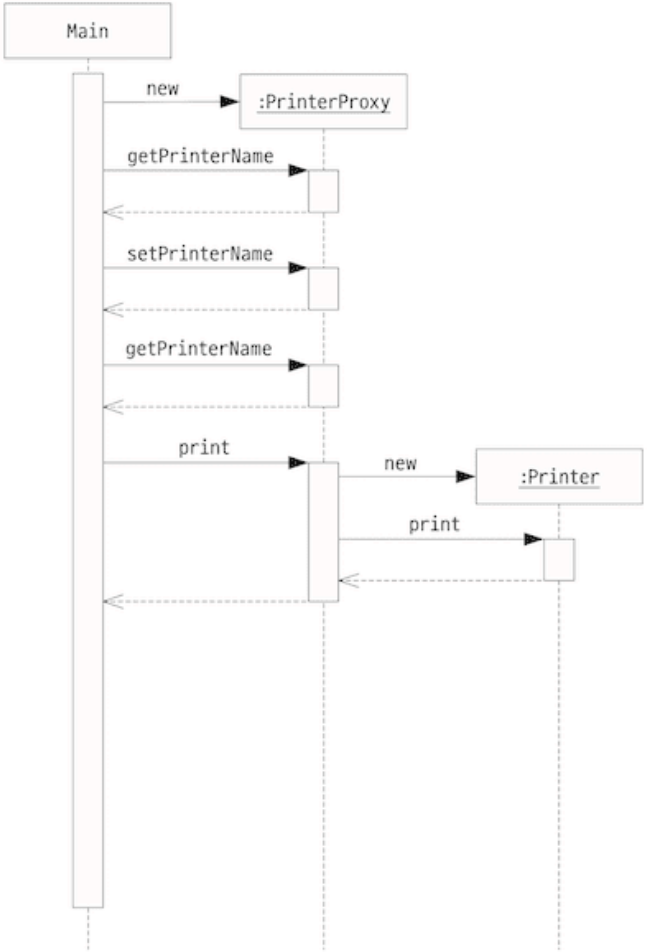
```
class Printer(Printable):  # RealSubject
    def __init__(self, name="Unknown"):
        self.name = name
        self._heavy_job(f"Initializing printer {name}")  # Expensive!

    def _heavy_job(self, msg):
        print(msg, end='')
        for i in range(5):  # Simulate expensive initialization
            time.sleep(0.1)
            print('.', end='')
        print(" Done!")

    def print(self, string):
        print(f"=== {self.name} ===")
        print(string)  # Actual printing functionality
```

Discussion

Sequence of Operations



1. **Client** requests operation from **Proxy**
2. **Proxy** checks if the operation can be handled locally
3. If **heavy operation**: Proxy creates **RealSubject** (if not exists)
4. **Proxy** delegates to **RealSubject**
5. **RealSubject** performs actual operation

Key Benefits

1. **Performance**: Avoids expensive operations until needed
2. **Resource management**: Controls when/how resources are allocated
3. **Transparency**: Same interface as real object - clients unaware
4. **Control**: Additional behavior without modifying the real object

When to Use Proxy

- **Expensive object creation**: Object costly to create/initialize
- **Access control**: Need to control access to the object
- **Remote objects**: Need a local representative for the remote object
- **Additional behavior**: Want to add behavior without changing the original

Virtual Proxy Benefits

- **Lazy loading**: Create objects only when actually needed
- **Memory efficiency**: Avoid loading unnecessary objects
- **Performance**: Fast startup, slower first access to heavy features
- **Resource optimization**: Better resource utilization

Related Patterns

- **Decorator**: Both wrap objects, but Decorator adds behavior while Proxy controls access
- **Adapter**: Proxy keeps the same interface, Adapter changes the interface
- **Facade**: Proxy represents a single object, Facade represents a subsystem
- **Singleton**: Protection proxy might use Singleton for access control

Proxy vs Related Patterns

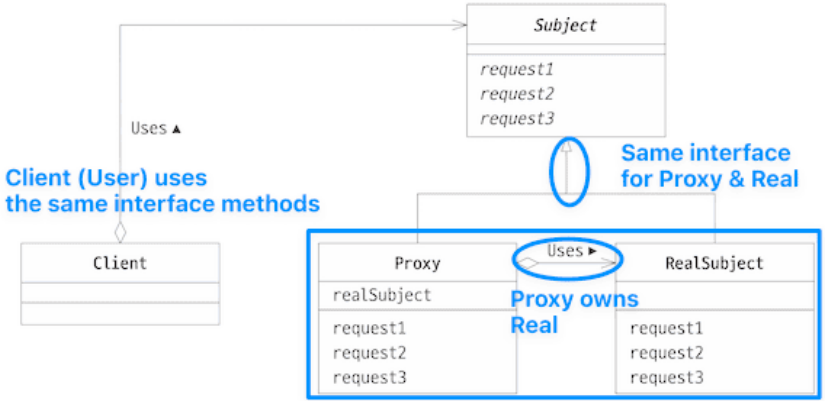
Adapter Pattern:

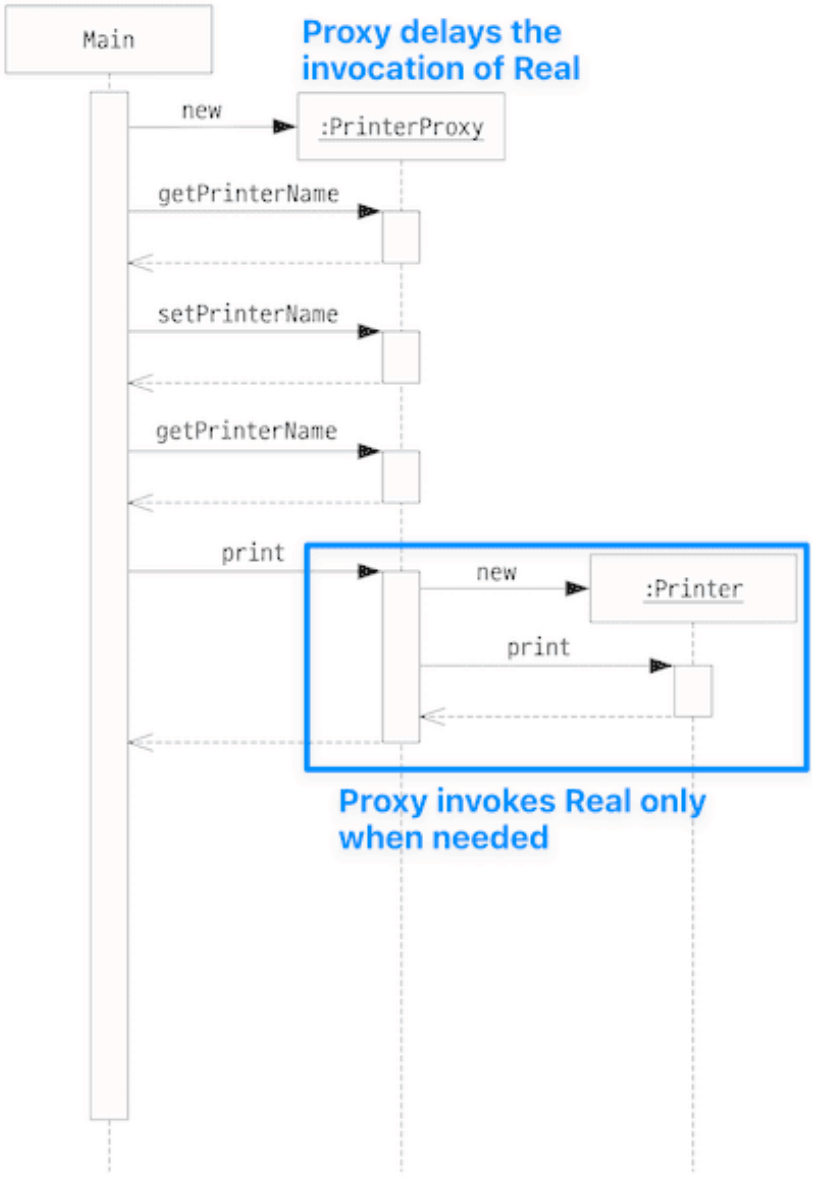
- **Adapter**: **Changes** interface to make incompatible classes work together
- **Proxy**: **Same** interface as real object, adds control layer

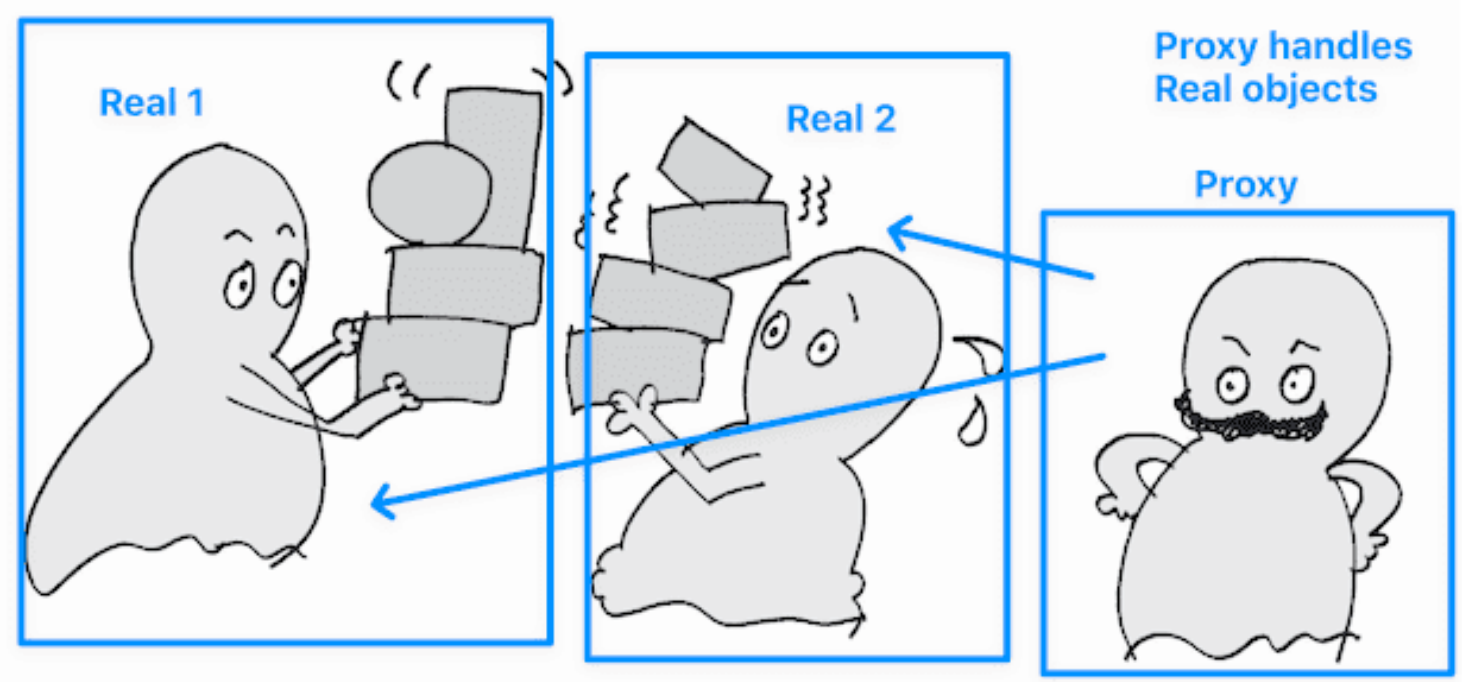
Decorator Pattern:

- **Decorator**: **Adds** behavior to existing object
- **Proxy**: **Controls access** to object, may create it

UML


Subject
request1
request2
request3
Uses ▲
Client (User) uses
the same interface methods
Same interface
for Proxy & Real
Client
Proxy
Uses ►
RealSubject
realSubject
request1
request2
request3
Proxy owns
Real
request1
request2
request3


Main
Proxy delays the invocation of Real
new
:PrinterProxy
getPrinterName
setPrinterName
getPrinterName
print
new
:Printer
print
Proxy invokes Real only when needed


Real 1
Real 2
Proxy handles
Real objects
Proxy