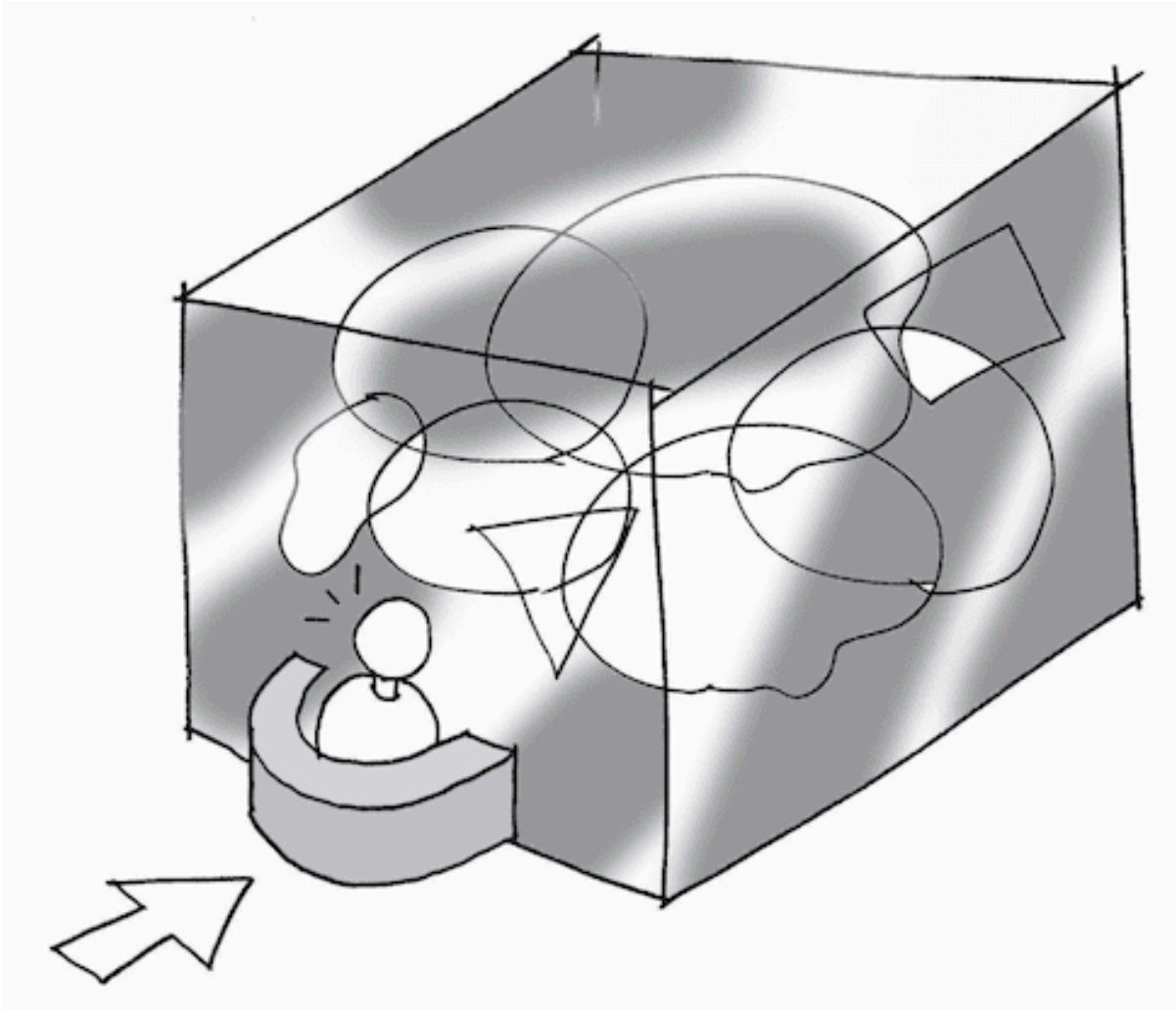
Facade Pattern

Provide Unified Interface to Subsystem



Facade Pattern

Think of a **hotel concierge** as a facade to the hotel:

- **Guest**: "I need dinner reservations, theater tickets, and a taxi"
- **Concierge**: Coordinates with restaurants, theaters, taxi companies
- **Behind scenes**: Multiple phone calls, bookings, confirmations
- **Guest experience**: One simple request gets everything done

The **concierge** *hides complexity* and provides a **unified interface** to city services.

The Problem

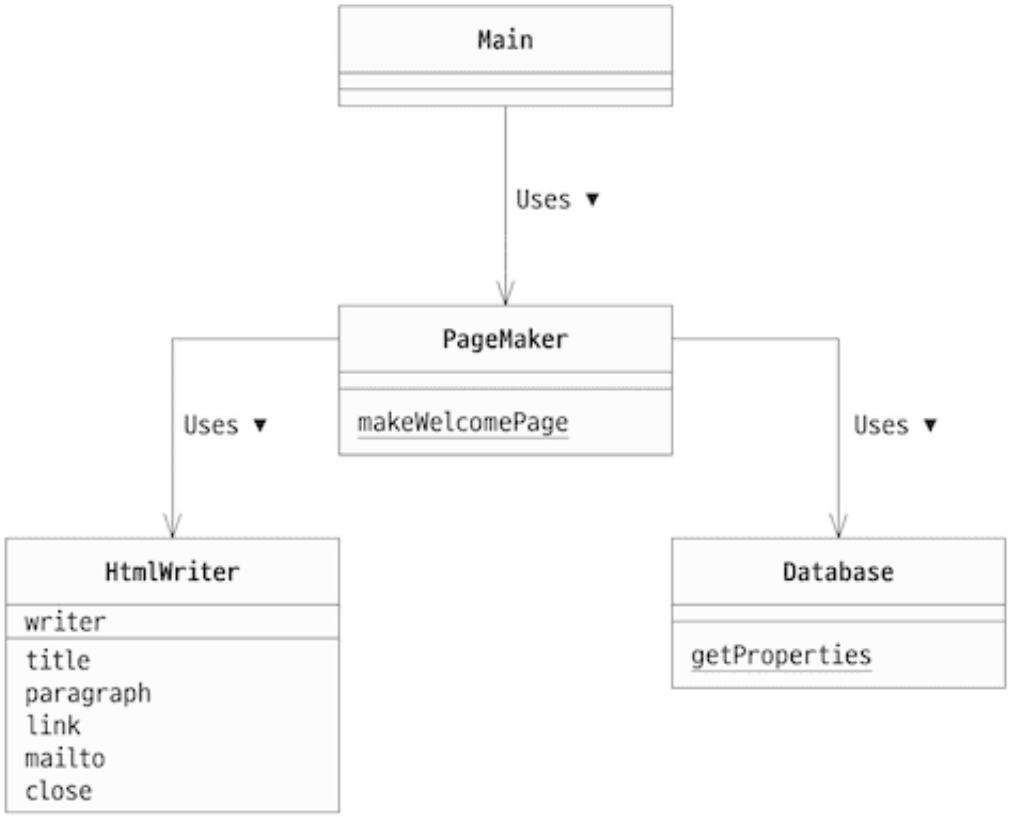
- We have **complex subsystems** with many interconnected classes.
- **Clients** need to coordinate multiple classes for common **workflows**.
- Direct usage creates **tight coupling** between clients and subsystem internals.

Challenge: How to simplify interaction with complex subsystems?

The *Facade* as the Solution

- We create a **facade class** that provides *simple methods*.
- **Facade** *coordinates* calls to multiple **subsystem classes**.
- **Clients** use the simple facade interface instead of dealing with subsystem complexity.

The Solution (Design)



Step 1: Understand the Players

In this design, we have players:

- **Facade** (PageMaker)
- **Subsystem Classes** (Database, HtmlWriter)

We have Clients that use a facade instead of the subsystem directly.

- **Client**

Step 2: Interface

- **Facade** provides high-level methods for common workflows.
- **Clients** call one facade method instead of multiple subsystem calls.

Step 3: Understand abstractions (Facade-Subsystem)

- We have a **Facade** that `coordinates` interactions with multiple **subsystem classes**.
    - **Facade** (PageMaker) - provides a simplified interface
    - **Subsystem Classes** - do the actual work
- **Facade** knows which subsystem classes to call and in what order.

- Notice that **Facade** *composes* multiple **subsystem classes**.
    - It acts as a coordinator, not just a wrapper.
- Notice that **clients** can still access **subsystem classes directly** if needed.
    - Facade provides convenience, not restriction.

Step 4: Understand concretion (Facade-Subsystem)

- We have **PageMaker** (facade) that coordinates **Database** and **HtmlWriter**.
    - **PageMaker**: Simple methods for creating web pages
    - **Database**: Handles data retrieval operations
    - **HtmlWriter**: Handles HTML generation

Code

- Main Method
- Facade Class
- Subsystem Classes

Main Method

```
from page_maker import PageMaker

def main():
    print("=== Facade Pattern Example ===\n")

    # Simple facade interface hides complexity
    print("Creating welcome pages...")

    # One method call orchestrates multiple operations
    PageMaker.make_welcome_page("alice@example.com", "welcome_alice.html")
    PageMaker.make_welcome_page("bob@example.com", "welcome_bob.html")

    print("All pages created successfully!")
```

Step 1: Simple client interface

```
PageMaker.make_welcome_page(
  "alice@example.com",
  "welcome_alice.html")
```

- **Client** makes a straightforward call.
  - **Client** doesn't need to know about Database, HtmlWriter, etc.
- **Facade** handles all the complexity behind the scenes.

Step 2: Facade orchestrates subsystem

```
# What facade does internally:
# 1. Database.get_properties() - get user data
# 2. HtmlWriter() - create HTML structure
# 3. writer.title(), writer.paragraph() - add content
# 4. File operations - save to disk
# 5. Error handling - manage exceptions
```

- **Multiple subsystem operations** coordinated in correct order.
- **Error handling** centralized in facade.

Facade Class

```
# page_maker.py
from database import Database
from html_writer import HtmlWriter

class PageMaker:
    @staticmethod
    def make_welcome_page(mailaddr, filename):
        try:
            # Step 1: Get user data from the database
            mail_properties = Database.get_properties("maildata")
            username = mail_properties.get(mailaddr, "Unknown User")
            # Step 2: Create an HTML file
            with open(filename, 'w', encoding='utf-8') as f:
                writer = HtmlWriter(f)
                # Step 3: Generate an HTML content
                writer.title(f"{username}'s web page")
                writer.paragraph(f"Welcome to {username}'s web page!")
                writer.paragraph("Nice to meet you!")
                writer.mailto(mailaddr, username)
                writer.close()
            print(f"{filename} created for {mailaddr} ({username})")

        except Exception as e:
            print(f"Error creating page: {e}")
```

Subsystem Classes

```
# database.py
class Database:
    @staticmethod
    def get_properties(dbname):
        filename = f"{dbname}.txt"
        properties = {}
        with open(filename, 'r') as f:
            for line in f:
                if '=' in line:
                    key, value = line.strip().split('=', 1)
                    properties[key] = value
        return properties

# html_writer.py
class HtmlWriter:
    def __init__(self, file):
        self.file = file
        self.file.write("<html><head></head><body>\n")
    def title(self, title): self.file.write(f"<h1>{title}</h1>\n")
    def paragraph(self, msg): self.file.write(f"<p>{msg}</p>\n")
    def close(self): self.file.write("</body></html>\n")
```

Discussion

Common Misunderstanding

It is wrongly known that Facade pattern prevents clients from accessing subsystem classes directly if needed.

Wrong! Clients can still access subsystem classes directly; Facade just provides a convenient alternative

Without Facade (Complex)

The client must coordinate multiple classes.

```
properties = Database.get_properties("maildata")
username = properties.get("alice@example.com", "Unknown")

with open("alice.html", 'w') as f:
    writer = HtmlWriter(f)
    writer.title(f"{username}'s page")
    writer.paragraph(f"Welcome {username}!")
    writer.close()
```

With Facade (Simple)

Facade handles all coordination.

```
PageMaker.make_welcome_page("alice@example.com", "alice.html")
```

Key Benefits

1. **Simplification**: Easy-to-use interface for complex subsystem
2. **Decoupling**: Clients isolated from subsystem implementation details
3. **Consistency**: Standardized way to perform common operations
4. **Maintainability**: Changes in the subsystem don't affect the client code

Key Drawbacks

1. **Limited flexibility**: Facade may not expose all subsystem capabilities
2. **Another layer**: Adds indirection between client and subsystem
3. **Potential bottleneck**: All operations must go through facade
4. **God object risk**: Facade might become too complex if overused

When to Use Facade

- When you want **simple interface** to complex subsystem
- When there are **many dependencies** between clients and subsystem classes
- When you want to **layer your subsystems**
- When the subsystem is **complex** but clients need only a subset of functionality

When NOT to Use Facade

- When the subsystem is **already simple**
- When you need **fine-grained control** over subsystem classes
- When **performance** is critical (extra layer adds overhead)
- When subsystem **changes frequently** (facade becomes maintenance burden)

Real-World Examples

- **Compiler**: A Simple compile command hides the lexer, parser, optimizer, and code generator
- **Operating System**: System calls provide a simple interface to the complex kernel
- **Libraries**: jQuery facades complex DOM manipulation
- **APIs**: REST APIs facade complex backend systems

Related Patterns

- **Adapter**: Adapter changes the interface, Facade simplifies the interface
- **Singleton**: Facade often implemented as Singleton

Facade vs Adapter

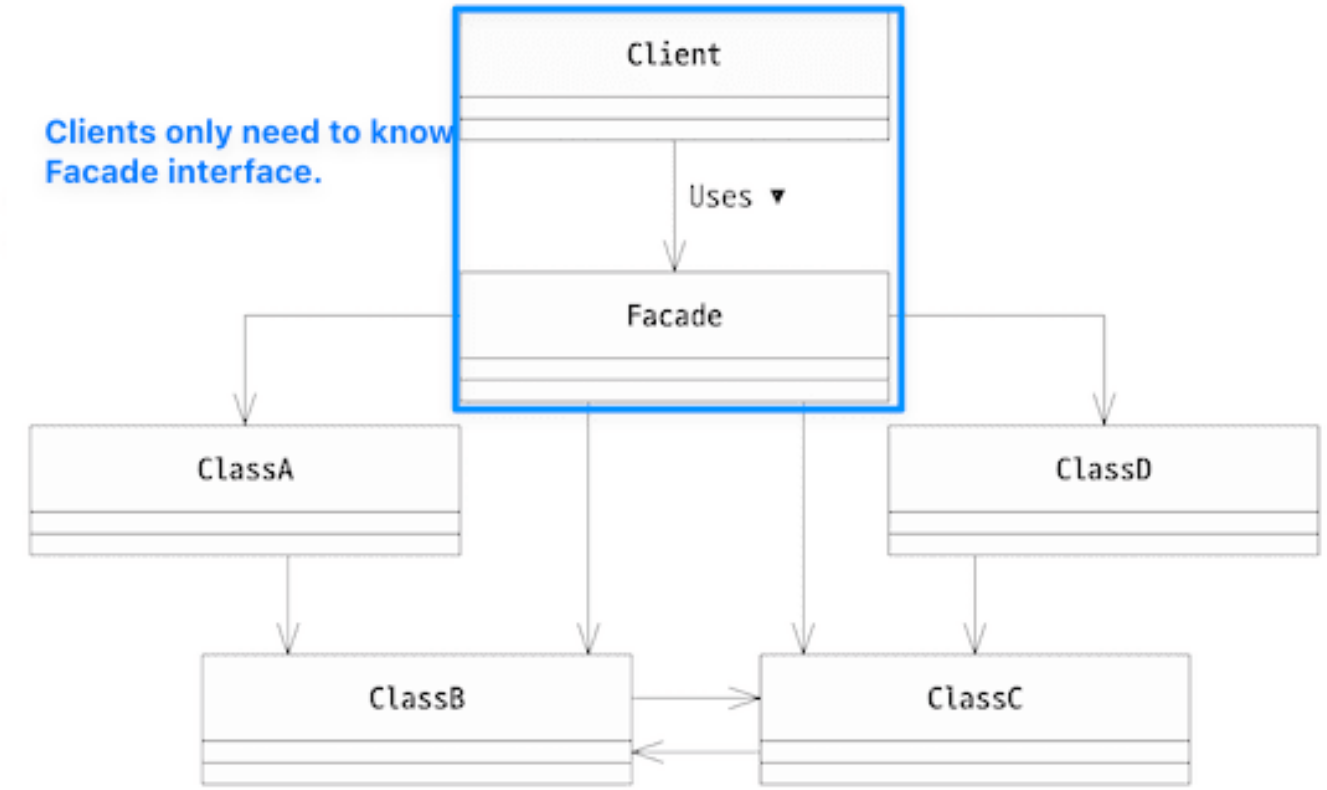
Facade:

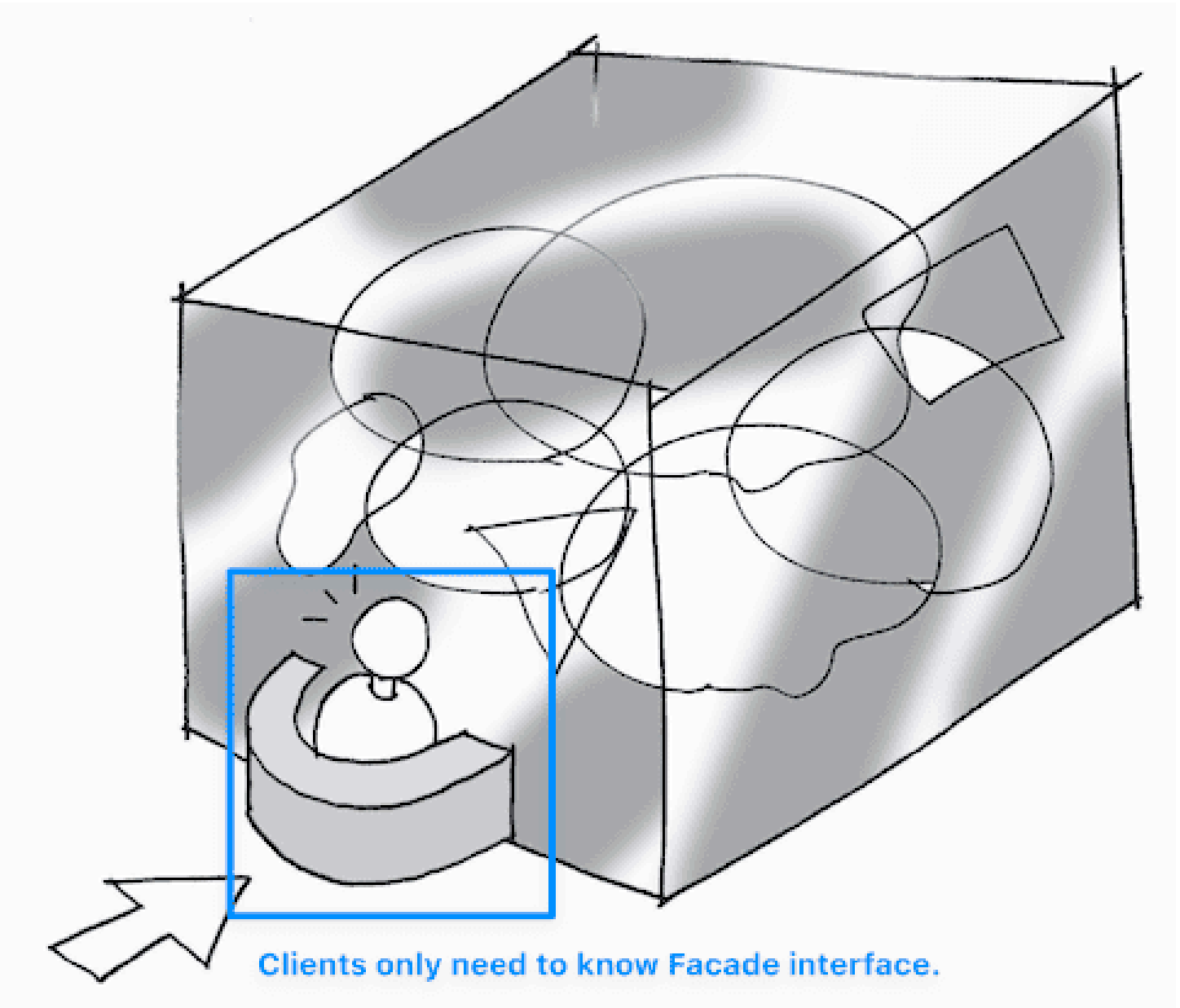
- **Simplifies** complex interface
- **Multiple** subsystem classes
- **New** interface for existing system

Adapter:

- **Changes** incompatible interface
- **Single** adaptee class
- **Compatible** interface for existing class

UML


Client
Clients only need to know
Facade interface.
Uses
Facade
ClassA
ClassD
ClassB
ClassC


Clients only need to know Facade interface.