Lecture 9: Program Design Continued …

CS 5150, Spring 2026

Administrative Reminders

- Project Report #1 Due Feb 28 11.59 PM
- Midpoint presentation (5-7 mins): 12/17 Mar, leave 1 min for questions. Sign up! (link on canvas/modules/projects)
  - Check if your client is available
- Assignment 2 and Project plan grades are out
- In-class exam 1: Mar 10

- Mar 19 External Talk!

External Talk (Mar 19)

- **Rajdeep Mukherjee**: Tech Lead and Senior Applied Scientist at AWS, PhD Oxford Univ.

- **Topic**: Spec Driven Development with AI Agents/Kiro (https://kiro.dev/)

Previously…

Reusable design patterns

- Design templates that solve recurring problems in a variety of different systems
- Popularized by "Gang of Four"
  - E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994
- Avoid reinventing the wheel; adopt proven solutions with known tradeoffs
- When developers are familiar with design patterns, they can be used to quickly communicate complex relationships between classes

Observer pattern

- **Setting**: A variety of entities (for example, different graphical views) need to be updated whenever the state of an object changes
- **Solution**
  - Observers: notified when Subject state changes; should update (i.e. display) accordingly
  - Subject: notifies Observers when its state changes
- **Consequences**
  - Subject not coupled to concrete Observers
  - Lack of coupling may impede performance optimizations
  - Redundant updates may be triggered
  - Control flow for Observers is inverted, can be hard to trace

Builder

- **Setting**: Want flexibility in constructing a complex object without sacrificing encapsulation (or immutability)
  - Constructors limit flexibility (must be distinguished by signature)
  - Might want to share a partial configuration
- **Solution**:
  - Define a Builder class associated with the class of object to be created
  - Mutate builder with imperative code
  - Builder is responsible for constructing object on demand
- **Consequences**:
  - Object creation decoupled from representation
  - Requires separate Builder class for each type

Builder examples

- Java's StringBuilder (yields immutable Strings)
- Java's HttpRequest.Builder

```
var b = HttpRequest.newBuilder();
b.uri(new URI("http://www.nasa.gov/"));
b.version(HTTP_1_1);
b.GET().header("DNT", "1");
HttpRequest r = b.build();
var r2 = b.timeout(Duration.ofSeconds(10)).build();
```

https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries

Builder notes

- Builder methods often return `this`, enabling chaining
- Complex class constructor typically trivial (accepts full set of field values), may be private

Factory method

- **Setting**: Want to create an object (fulfilling some interface) without specifying its exact (sub)class
- **Solution**:
  - Define a factory method whose return type is the interface/superclass
  - Method implementation selects appropriate subclass, defers initialization logic to its constructor
- **Consequences**:
  - Can modify subclass constructors, add new subclasses without affecting client code
  - Supporting new subclasses requires *registering* with factory
  - Leads to polymorphic factories ("abstract factory" pattern), service providers

Resource acquisition

- Heap memory: `malloc()`/`free()`, `new()`/`delete()`
- Files & devices: `open()`/`close()`
- Mutexes: `lock()`/`unlock()`

- Concerns
  - Use before allocation
  - Use after deallocation
    - Deallocation in wrong order
  - Resource leaks (never deallocated)
    - Common around Exceptions (need `finally` block)
    - Can lose data if buffers are never flushed

RAII (resource acquisition is initialization)

- Manage resources using object lifetimes (e.g., scopes)
  - Resource is acquired when object is created
  - Resource is returned when (last) object is destroyed
    - If resource can be shared, copying object increments reference count
    - Obeys stack ordering
  - Holding a resource is a *class invariant*
  - No object leaks -> no resource leaks
- Examples
  - Smart pointers (`shared_ptr`, `unique_ptr`)
  - Mutexes (`lock_guard`)

Examples (Unique ptr/shared ptr)

- std::unique_ptr<int> p = std::make_unique<int>(42);
  - Exactly one owner
- P owns the integer; when p goes out of scope, memory is freed
- Copying not allowed!
  - std::unique_ptr<int> p2 = p1; Invalid!

- std::shared_ptr<int> p1 = std::make_shared<int>(42);
- Allowed: std::shared_ptr<int> p2 = p1; => Ref count ++
- Uses reference counting
- Intuition: The object lives as long as someone is using it
- Problem: Cycles!
  - Solution: weak ptr!

(Anti)pattern: Singleton

- **Setting**: Want to enforce that a class only has a single instance, which can be easily accessed
- **Solution**:
  - Make constructor private
  - Construct single instance in static storage (possibly lazily)
  - Provide static method to get instance
- **Consequences**
  - Allows easy access to infrastructure without complexities of dependency injection
  - Most of the disadvantages of global variables
    - Can't configure differently for different subsystems
    - Thread safety/contention concerns if mutable
    - Can't replace instance for testing

Singleton example

```
public class Singleton {
    private static volatile Singleton instance = null;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized(Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```


https://en.wikipedia.org/wiki/Singleton_pattern

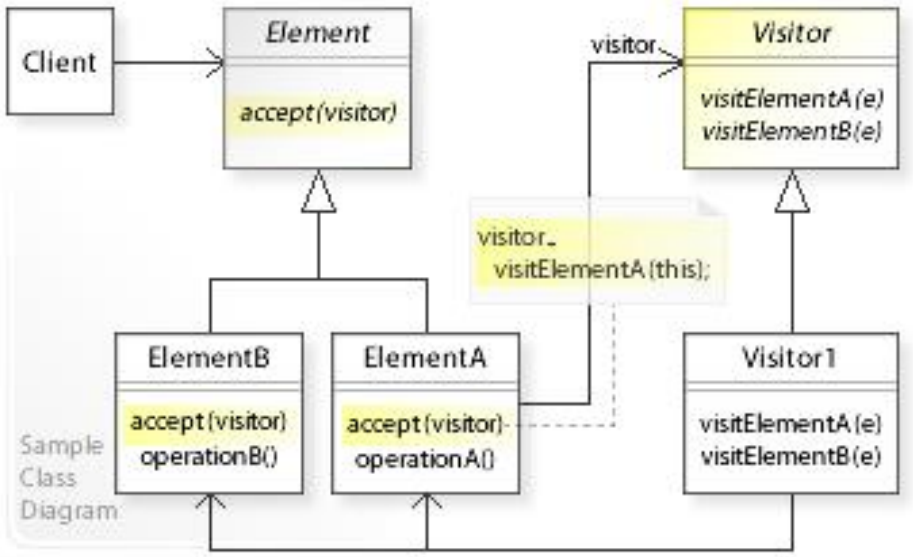
Composite

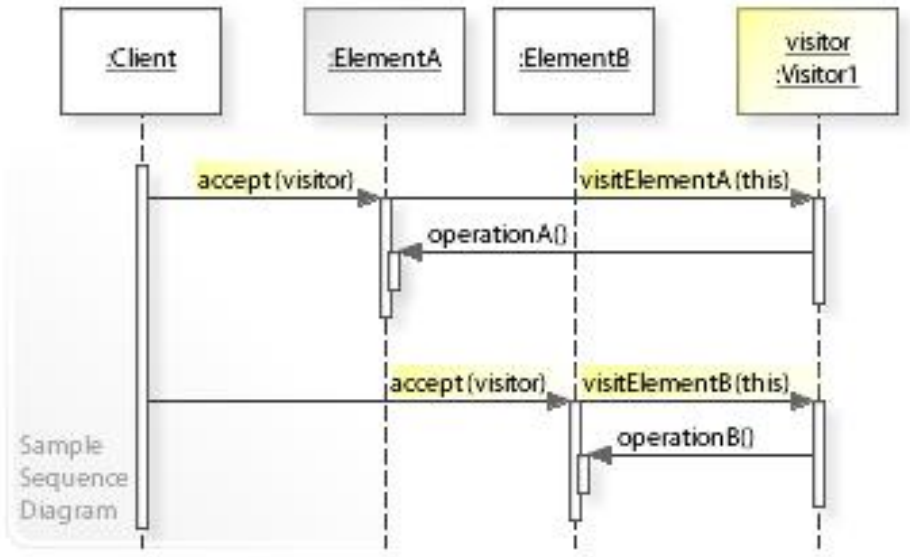
- **Setting**: Hierarchy of elements; want to treat parent and leaf nodes uniformly
- **Solution**:
  - Component: common interface
  - Leaf: concrete implementation of Component; performs actual work
  - Composite: concrete implementation of Component, composed of Components; delegates operations to constituent Components
- **Consequences**
  - Can add additional leaves without affecting client code

Visitor

- **Setting**: Add new functionality to many classes without modifying them; useful when new operations are needed frequently
- **Solution**
  - Visitor interface declares visit method for each class
  - Concrete Visitor implementations provide new functionality
  - Target classes have method to "accept" a visitor, which just calls the appropriate method for its type
- **Consequences**
  - Adding new functionality only requires a new Visitor subclass
  - Adding new types requires expanding Visitor interface, updating all subclasses (but can catch unhandled types at compile time)
  - Avoids duplicating dispatch logic

Visitor UML

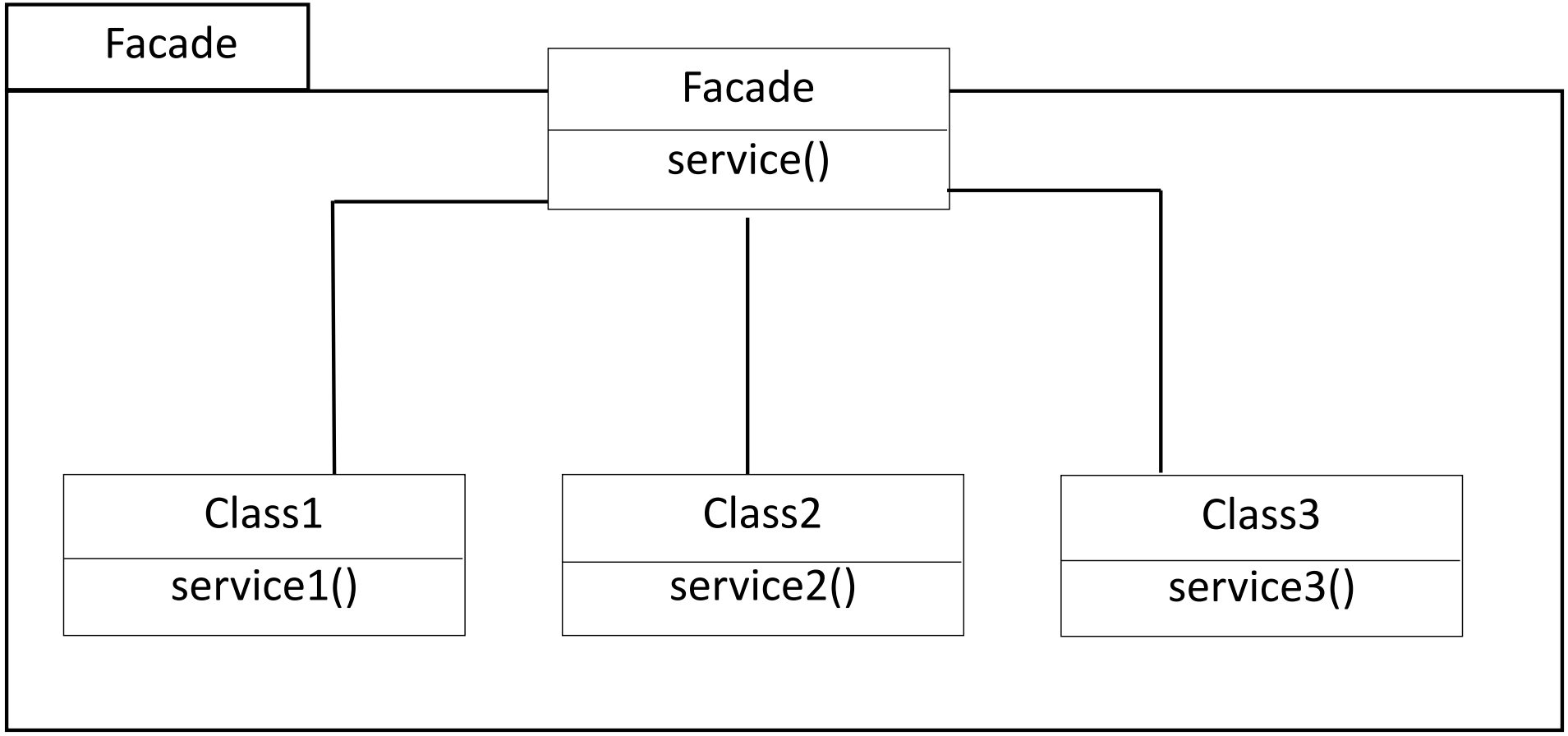




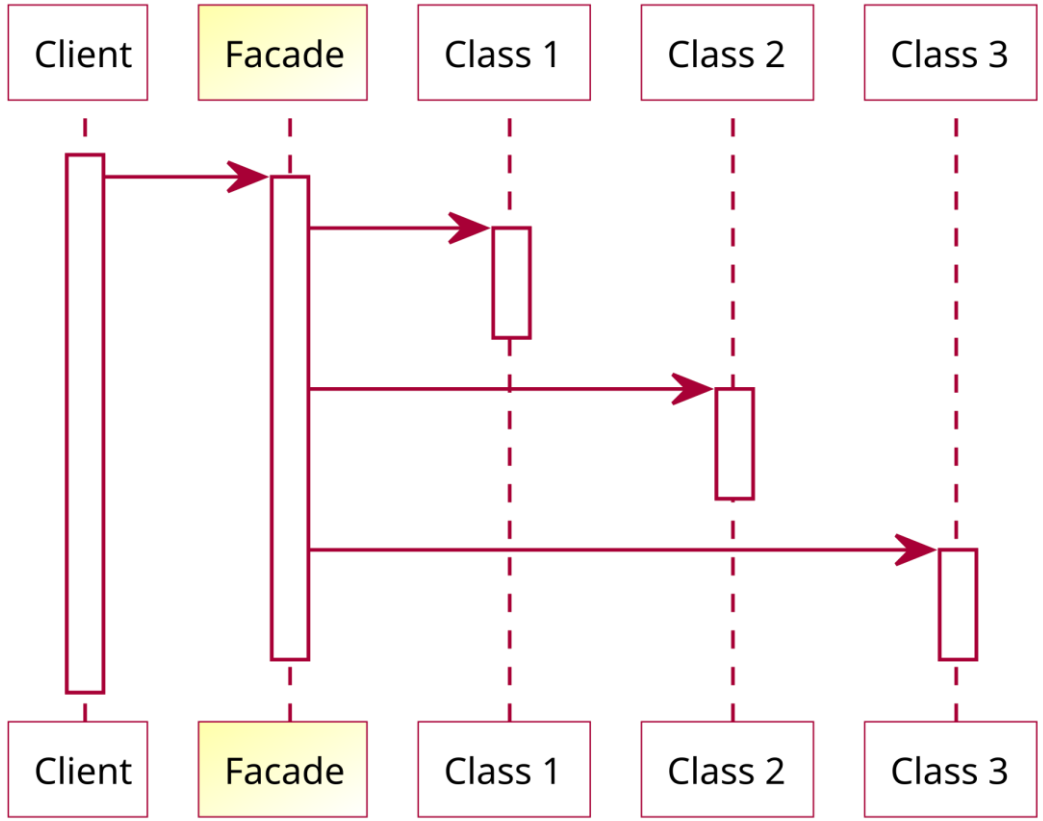
Façade

- **Situation**: Expose high-level functionality while avoiding coupling to low-level classes
- **Solution**:
  - Façade class declares high-level interface and implements it by invoking methods of low-level classes in appropriate sequence
- **Consequences**:
  - Encapsulates lower-level functionality, reducing coupling
    - Lower-level classes can be refactored without affecting users of high-level service
  - Simplifies and standardizes way that high-level functionality is performed
  - Repeated application yields a layered design
  - Improves readability/usability of complex software library

Façade delegation diagram



Façade Sequence Diagram



Sample sequence diagram

Façade examples

- Real-world examples:
  - Payment processing APIs (Stripe, Paypal)
    - stripe.charge(…)
  - Compiler toolchains
    - gcc main.c
  - Placing orders over phone via an *operator*

Design Pattern Categories

- **Creational Patterns**: how to create objects?
  - Factory
  - Builder
  - Singleton
- **Structural Patterns**: how to organize classes and objects to form larger structures to provide new functionality
  - Composite
  - Façade
  - Adapter
- **Behavioral Patterns**: algorithms, responsibilities
  - Visitor
  - Observer
  - Iterator

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

Poll: PollEv.com/cs5150sp26

The application needs to estimate the cost (execution time per sample) for an arbitrary filter chain on different kinds of hardware. New hardware is benchmarked frequently, and the filter classes themselves should not be responsible for cataloging it all. Which design pattern could help provide this capability?

- Observer
- Composite
- Visitor
- Façade

Filter chain: ordered sequence of filters, performing some operations

Poll: PollEv.com/cs5150sp26

- You're developing a distributed web application that runs across multiple servers. The application needs a centralized logging system that captures errors, warnings, and informational messages from all components of the application. This logging system must ensure that log entries are consistent, properly timestamped, and written to the appropriate destination (file, database, or external service) without duplicating records or creating race conditions when multiple components attempt to log simultaneously. Which design pattern would you choose?
- Visitor, Façade, Singleton, or Builder?

Poll: PollEv.com/cs5150sp26

- You're building an e-commerce application where users can add different types of products to their shopping cart - physical items (requiring shipping address and delivery options), digital downloads (requiring email delivery), subscription services (requiring recurring billing setup), and gift cards (requiring recipient information). The system needs to handle the creation of these different product types and their specific checkout processes through a unified interface. Which design pattern would you choose?

- Builder, Factory, Singleton, or Composite?

References

- Wikipedia: Software design pattern
- *Design Patterns* (Gamma et al, 1994); aka "GoF"
- *Object-Oriented Software Engineering* (Bruegge & Dutoit, 2004)
- Stack Overflow: Examples of GoF Design Patterns in Java's core libraries

Programming

Beyond code review

- How to ensure a healthy body of source code and preserve quality over time?
  - Explicit style guides and rules
  - Static analysis
  - Continuous enforcement

Past CS 5150 advice

- Write simple code
- Avoid risky programming constructs
- If code is difficult to read, rewrite it
- Include runtime verification
  - Verify **class/data invariants** after modification
  - Verify **preconditions** for parameter values
- Eliminate all warnings from source code
- Have a thorough set of test cases
- Expect to take longer to write and test production code in a production environment than in an academic one

Static analysis

- Checks that can be done on the source code (without running it)
  - Syntax errors during compilation
  - Linters & compiler warnings
  - Style checks
  - Complexity measurement
- Notable tools
  - clang-static-analyzer (C++)
  - FindBugs, ErrorProne (Java)
  - CodeSonar
- Keep false positives low (ideally zero)
  - Allows checks to be run continuously without risking desensitization

What bugs can static analysis find?

- Dead code
  - Many subtle ways to introduce (bad ordering of if-statements, poorly-scoped early returns)
- Typos in names (indicated by unused parameters)
- Misleading indentation
- Unintentional overloads, risky implicit conversions (abs vs. std::abs)
- Unhandled cases, unintended fallthrough in switch statements
- Use of deprecated functionality
- Common mistakes
  - Using == when operand types override equals()
  - delete vs. delete[]
- Missing null pointer checks
- …

Style guides

- Improve consistency of code
- Avoid unproductive arguments

- Popular style conventions:
  - PEP 8 for Python
  - Google Style Guides
  - Microsoft's C# Conventions

Style automation

Advantages

- Zero human effort
- Uniform enforcement
- Prevent accidentally misleading style
- Can be applied after refactoring, synthesizing code
- Can update entire codebase when style rules change

Disadvantages

- Can't reproduce all reasonable style rules
- Special-case exceptions are awkward
- Reformatting pollutes blame history