Lecture 17: Design patterns

CS 5150, Spring 2022
Lecture goals

• Leverage design patterns to reuse solutions to common problems
• Justify uniformity of coding conventions and style
Design patterns

... continued from Lecture 16
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
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
  
  ```java
  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();
  ```
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
Factory examples

• Java's `Charset.forName()`
  ```java
  var c = Charset.forName("UTF-8");
  ```
Creation patterns

• More flexible than constructors
  • Can perform imperative operations prior to assigning const fields (C++)
  • Can return other types (union types, subtypes)
  • Can reuse instances ("interning")

• Reduces dependencies of core classes
  • Move string parsing, file I/O, networking code to helper classes
  • Allows reusing core classes in more constrained contexts
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

• Examples
  • Smart pointers (shared_ptr, unique_ptr)
  • Mutexes (lock_guard)
RAII example

Manual resource management
mutex m;
void bad() {
    m.lock();
    f();  // Throws
    if(!check()) return;
    m.unlock();
}

RAII
mutex m;
void good() {
    lock_guard<> lock(m);
    f();  // Throws
    if(!check()) return;
}
Resource acquisition advice

• C++
  • If working with resources from C APIs (e.g. Unix file descriptors), use or write an RAII wrapper
  • Use smart pointers to track ownership of heap objects
    • Reserve references, raw pointers for "borrowing"
  • Avoid manual resource management APIs
    • Unless deferred acquisition or early release are absolutely required

• Java
  • Use try-with-resources to automatically close resource objects in finally block

• Python
  • Use context managers (with ... as ...)
(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

```java
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
Poll

A signal processing application provides many different filters with a common interface. The application can load a text file specifying a filter chain, with one filter type (with parameters) on each line. Which design pattern would help us construct the appropriate filter subclass as each line of the file is read?

• Builder
• Factory method
• RAII
• Singleton
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
Composite class diagram

Client

Component

Leaf

Composite

1..*
Composite example

• CS 2110: Tree node
  • Depth
    • Leaf: 1
    • Composite: max([c.depth() for c in children])
  • Contains(x)
    • Leaf: value == x
    • Composite: any([c.contains(x) for c in children])
Visitor

• Setting: Add new functionality to many classes without modifying them

• 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
Visitor example

• Java's `javax.lang.model.element.{Element, ElementVisitor}`
  • Strict example of pattern

• Java's `FileVisitor`
  • Elements lack "accept" method; instead, `Files.walkFileTree` handles dispatch
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
Façade delegation diagram

Façade

Facade

service()

Class1

service1()

Class2

service2()

Class3

service3()
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
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
Example

```java
if (a != null)
    if (a.x != x) return false;
    if (a.y == y) return true;
    else return false;
```

• Will this always return?
• Could this throw an exception?
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
Case study: C++ reference parameters

• `foo(Bar b)` // Copy argument
• `foo(Bar & b)` // Borrow mutable argument
  • Indistinguishable from copy at call site
• `foo(Bar const & b)` // Borrow immutable argument
  • Indistinguishable from copy at call site
• `foo(Bar * b)` // Borrow or share mutable argument, may be null
  • No lifetime guarantee for shares
• `foo(Bar const * b)` // Borrow or share immutable argument, may be null
  • No lifetime guarantee for shares
• `foo(shared_ptr<Bar> b)` // Share ownership of argument
• `foo(unique_ptr<Bar> b)` // Transfer ownership of argument
• `foo(Bar && b)` // Transfer ownership of argument
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
• Reformattting pollutes blame history