Lecture 9: Program Design

CS 5150, Spring 2026

Administrative Reminders

- Project Report 1 (for Sprint 1) Instructions are out now on course website: https://www.cs.cornell.edu/courses/cs5150/2026sp/projects/report-1.html
  - We will share a canvas assignment for uploading it later.
  - Due Feb 28, 11.59 PM

- Assignment 2 due Feb 20, 11.59 PM
- Project Plan is being graded – released by tomorrow
- Midpoint presentation – 5-7 mins presentation in class (dates TBD)
  - Mar 12/17 tentative

Previously…

Heavyweight vs. Lightweight design

Heavyweight

- Program design and coding are separate
  - Use models to specify program in detail, before beginning to code
  - UML provides modeling notation

Lightweight

- Program design and coding are interwoven
  - Development is iterative
  - Assisted by integrating multiple development tools (IDEs)

Mixed approach

- Use models to specify outline design
- Work out details iteratively during coding

Program design

- **Goal**: represent software architecture in form that can be implemented as one or more executable programs

- **Specifies**:
  - Programs, components, packages, classes, class hierarchies
  - Interfaces, protocols
  - Algorithms, data structures, security mechanisms, operational procedures

- Historically (e.g., aerospace), program design done by domain engineers, implementation done by *programmers*

UML models for design

- Diagrams give general overview
  - Principal elements
  - Relationships between elements
- Specifications provide details about each element

In a heavyweight process, specifications should have sufficient detail so that corresponding code can be written unambiguously.

Ideally, specification is complete before coding begins.

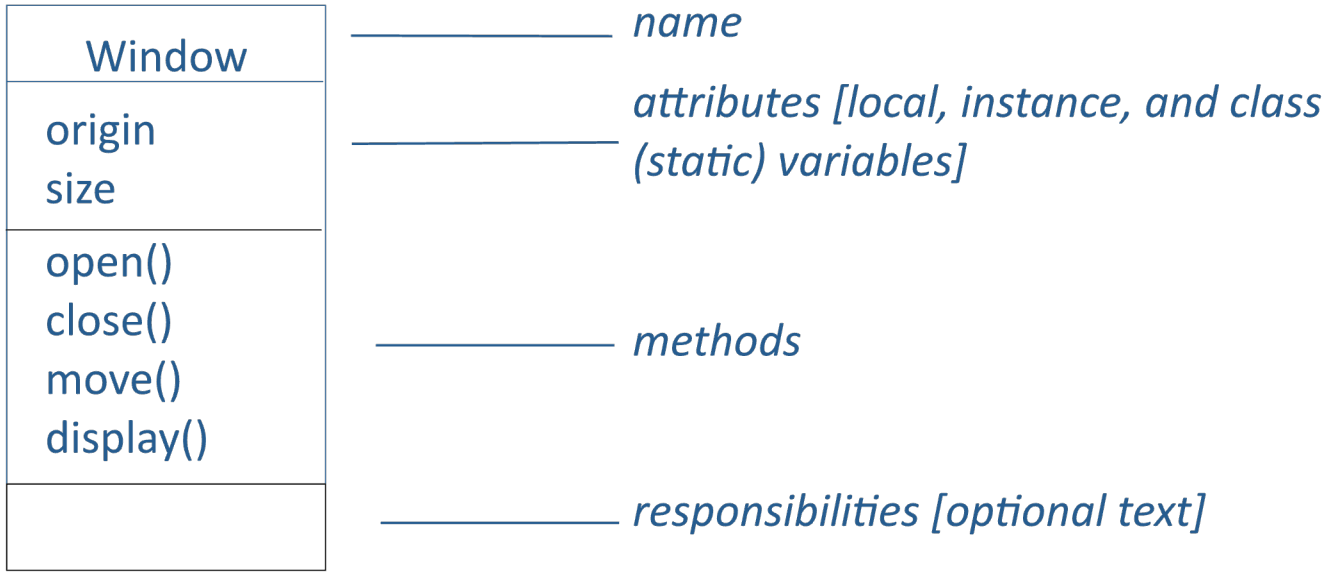
UML model choices

- **Requirements**
  - Use case diagram: use cases, actors, and relationships
- **Architecture**
  - Component diagram: interfaces and dependencies between components
  - Deployment diagram: configuration of processing nodes and the components that execute on them
- **Program design**
  - Class diagram (structural): classes, interfaces, collaborations, and relationships
  - Sequence diagram (dynamic): set of objects and their relationships

Class diagram

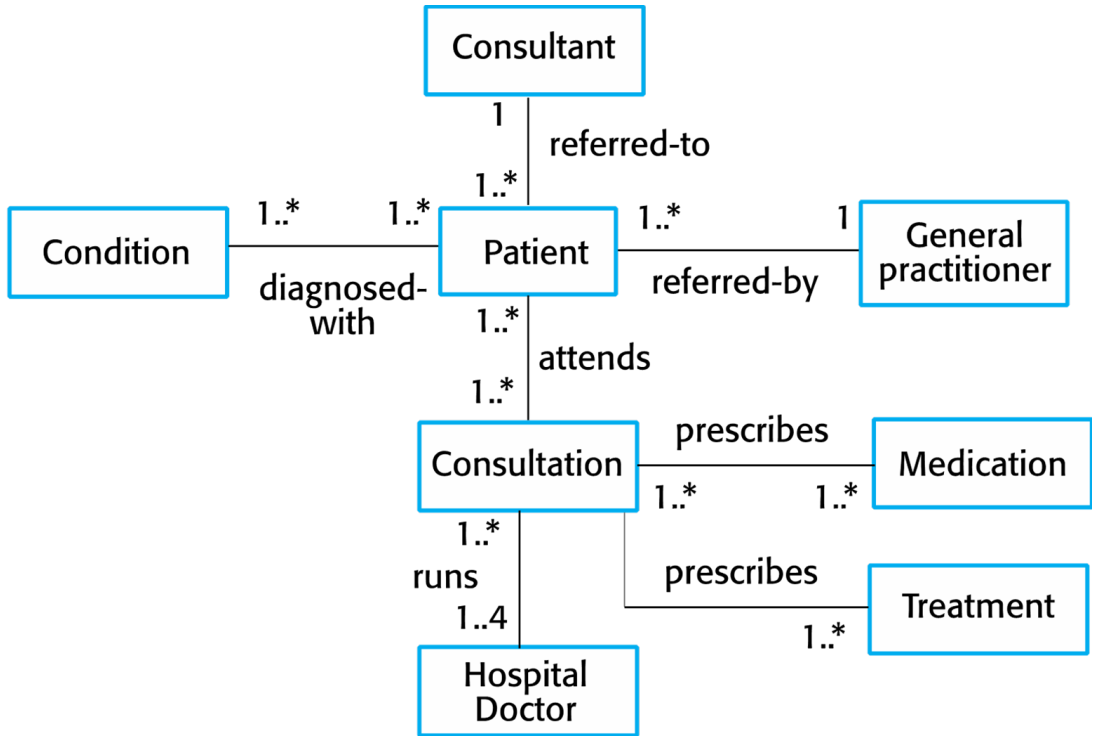
- Class: Set of objects with the same attributes, operations, relationships, and semantics

- "Operation" = "method"



Relationships

- Association: show multiplicity of links between instances of classes
  - Analogous to relations in entity-relation diagrams
  - Bidirectional – doesn't imply ownership or composition



Sommerville, *Software Engineering, Tenth Edition*, Figure 5.9

Relationships

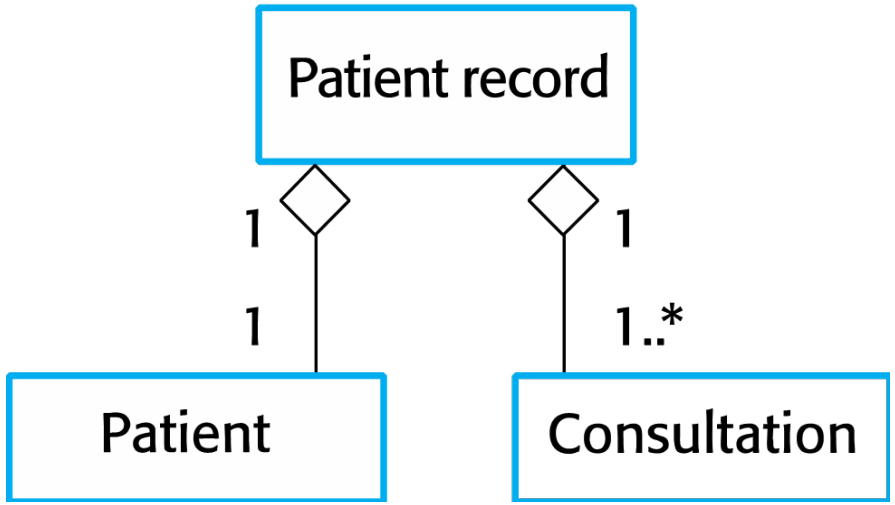
- Dependency
  - A change to one class may affect the semantics of another

- Generalization (inheritance)
  - Objects of a specialized (child) class are substitutable for objects of a generalized (parent) class

- Realization (interfaces)
  - A class is guaranteed to fulfil a contract specified by another class

Relationships

- Aggregation
  - An instance of one class (the whole) is composed of objects of other classes (the parts)
  - To reduce coupling, prefer composition over inheritance



Sommerville, *Software Engineering, Tenth Edition*, Figure 5.13

Example: noun identification

The library contains books and journals. It may have several copies of a given book. Some of the books are reserved for short-term loans only. All others may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

The system must keep track of when books and journals are borrowed and returned, and enforce the rules.

Example: Candidate classes

Noun	Comments	Candidate
Library		
Book		
Journal		
Copy		
ShortTermLoan		
LibraryMember		
Week		
MemberOfLibrary		
Item		
Time		
MemberOfStaff		
System		
Rule		

Example: Candidate classes

Noun	Comments	Candidate
Library	*the name of the system*	no
Book		yes
Journal		yes
Copy		yes
ShortTermLoan	*event*	no (?)
LibraryMember		yes
Week	*measure*	no
MemberOfLibrary	*repeat of LibraryMember*	no
Item	*book or journal*	yes (?)
Time	*abstract term*	no
MemberOfStaff		yes
System	*general term*	no
Rule	*general term*	no

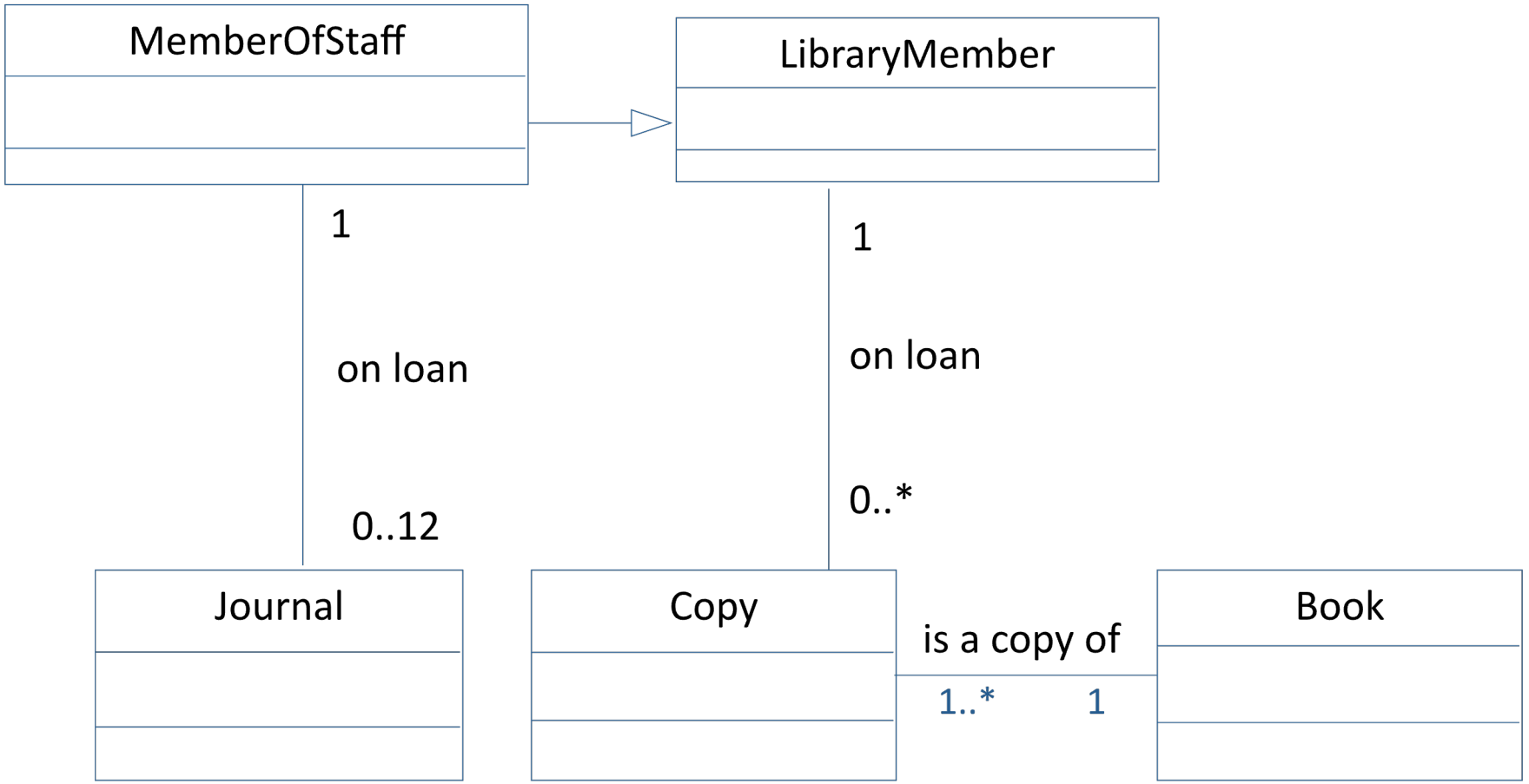
Example: Candidate relations

Book	is an	Item
Journal	is an	Item
Copy	is a copy of a	Book
LibraryMember		
Item		
MemberOfStaff	is a	LibraryMember

Example: candidate methods

LibraryMember	borrows	Copy
LibraryMember	returns	Copy
MemberOfStaff	borrows	Journal
MemberOfStaff	returns	Journal

Example: candidate class diagram



Moving towards final design

- Reuse: Wherever possible use existing components, or class libraries
  - They may need extensions.
- Restructuring: Change the design to improve understandability, maintainability
  - Merge similar classes, split complex classes
- Optimization: Ensure that the system meets anticipated performance requirements
  - Change algorithms, more restructuring
- Completion: Fill all gaps, specify interfaces, etc.

- Design is *iterative*
  - As the process moves from preliminary design to specification, implementation, and testing it is common to find weaknesses in the program design. Be prepared to make major modifications.

Lecture Goals

- Continue with class design notations
- Discuss various design patterns, their motivations, pros and cons

#1 rule of class design

- Classes should be easy to use correctly and hard to use incorrectly
  - See Effective C++, Third Edition

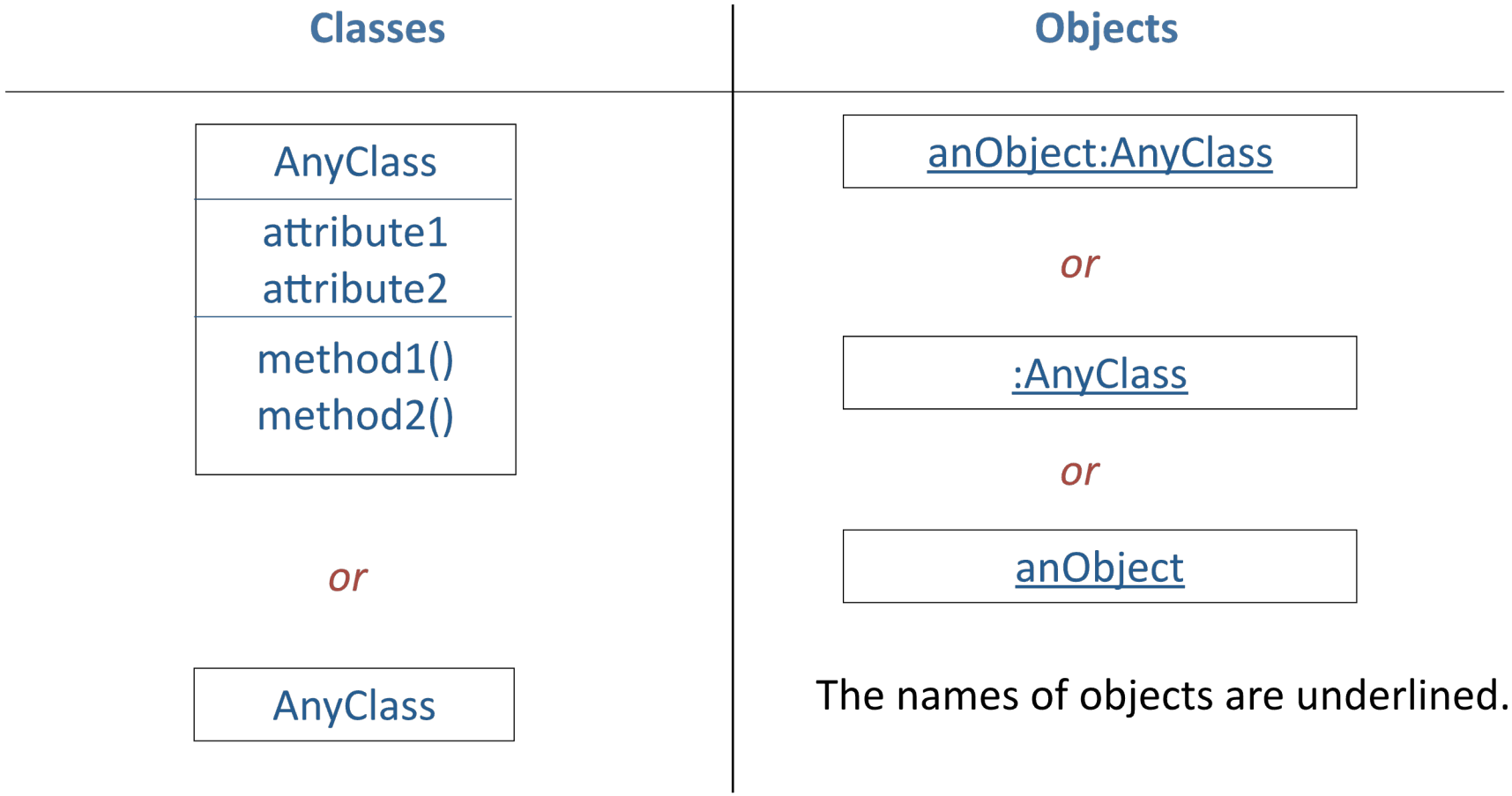
- Other good rules of thumb:
  - Avoid cyclic dependencies (tight coupling)

Modeling dynamic aspects of systems

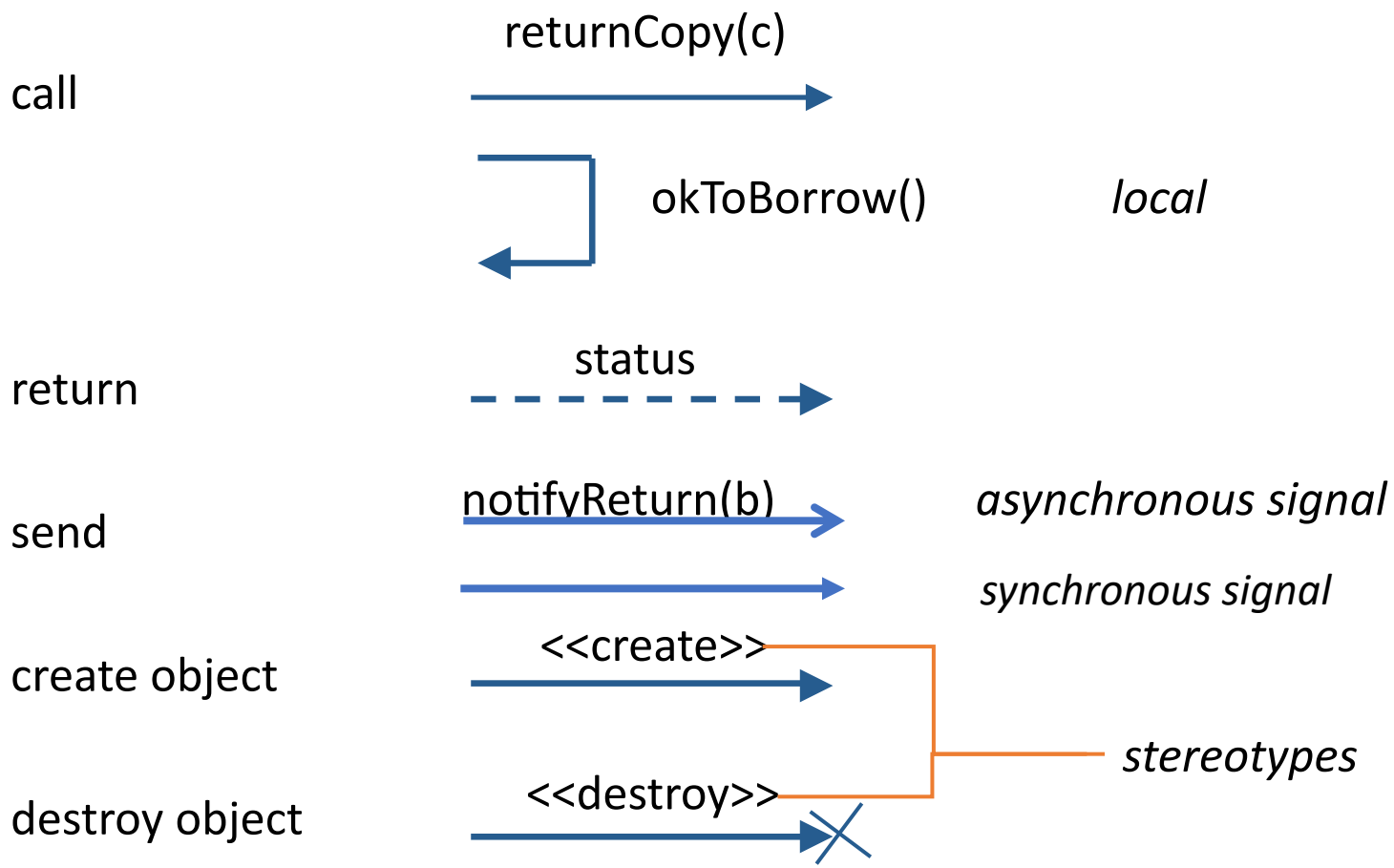
- Interaction diagrams: show a set of *objects* and their relationships
  - Includes messages sent between objects

- Sequence diagrams: time ordering of messages

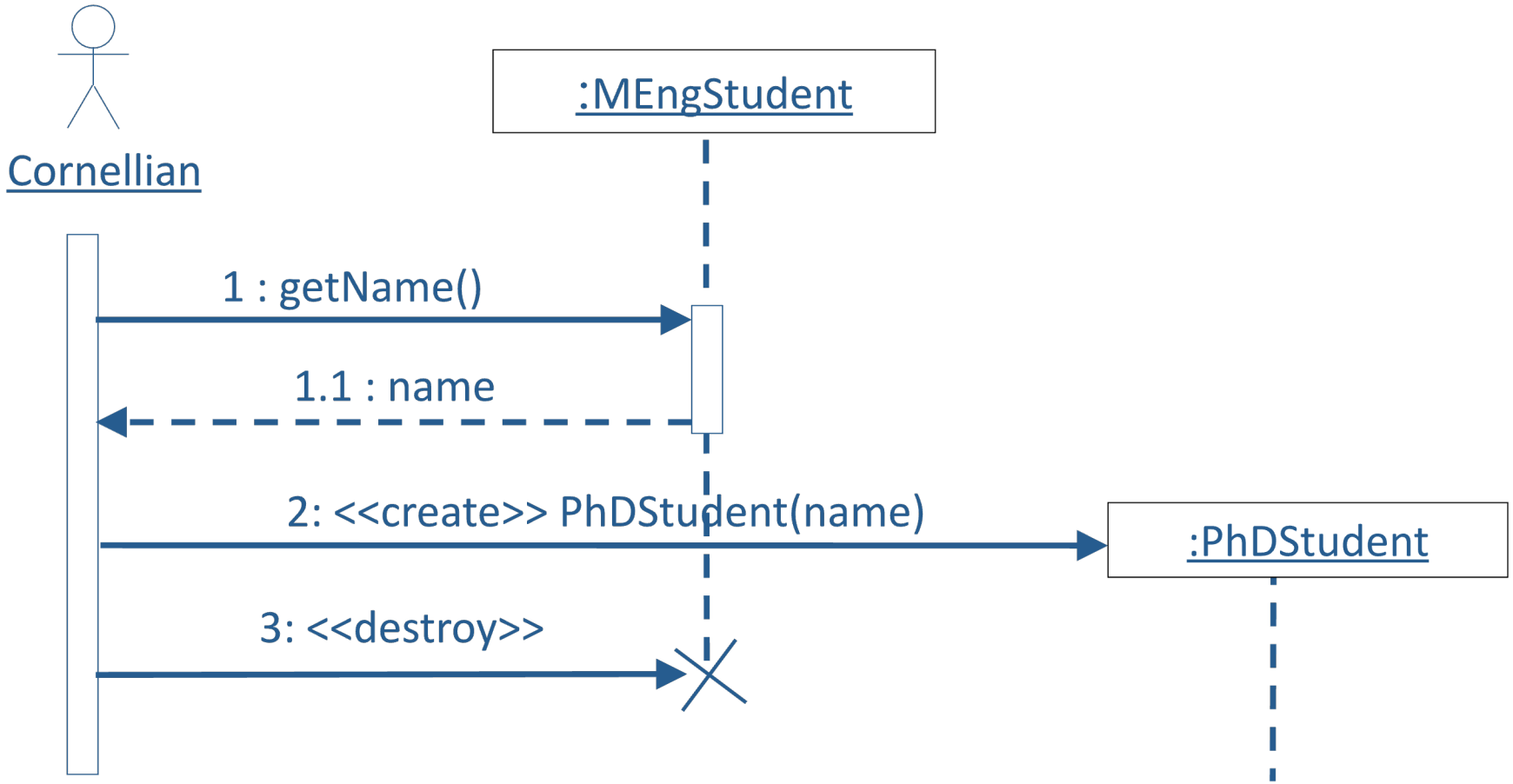
Object notation



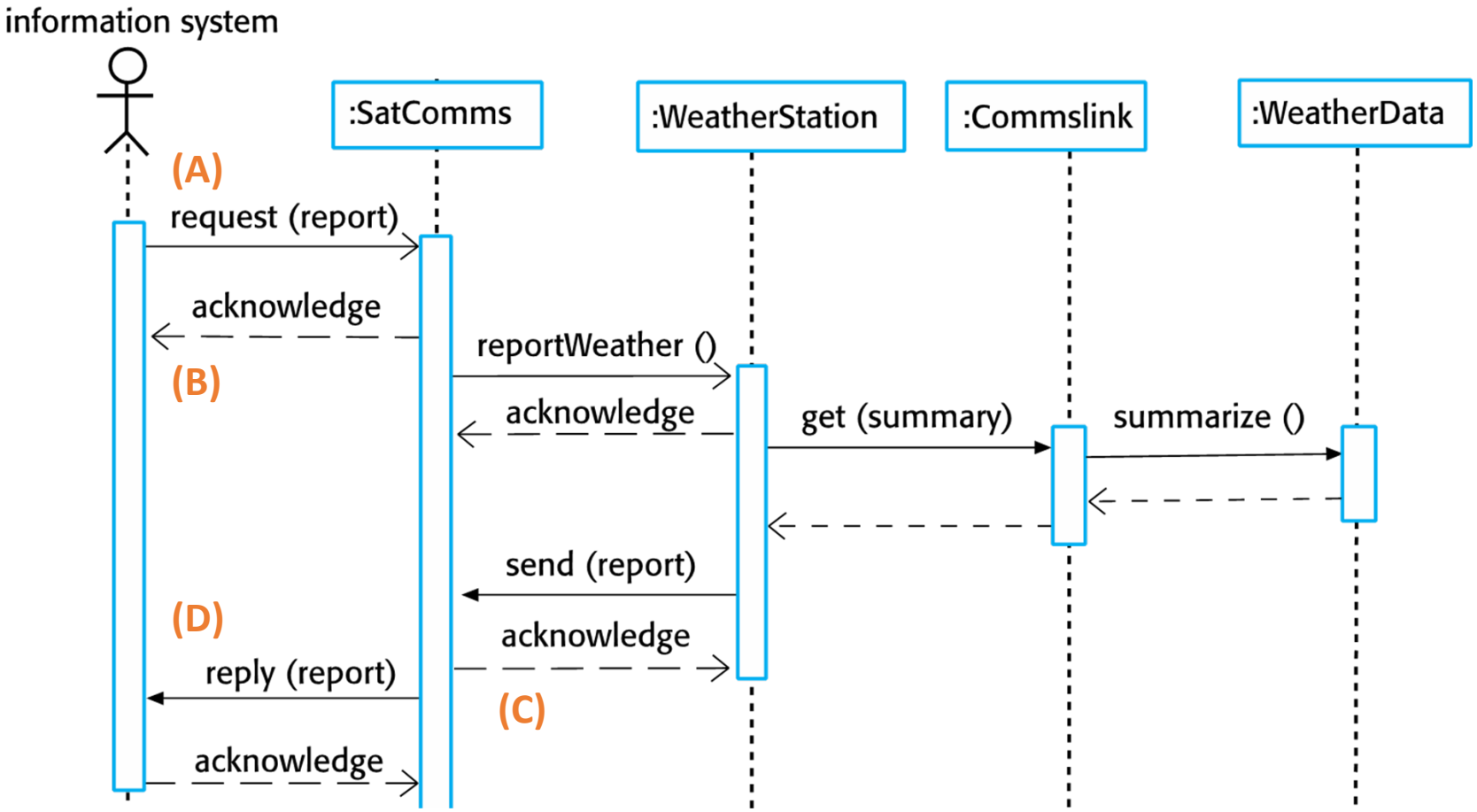
Message notation



Example: Changing student program



Poll: PollEv.com/cs5150sp26



Sommerville, *Software Engineering, Tenth Edition*, Figure 7.7

Design patterns

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

Properties of patterns

- Meaningful name
- Description of the problem setting
  - Explains where pattern may be applied
- Description of solution
  - Not a library, but a "design template"; can be instantiated in different ways
  - Often expressed graphically
- Statement of consequences
  - Results and tradeoffs of applying the pattern in the problem setting

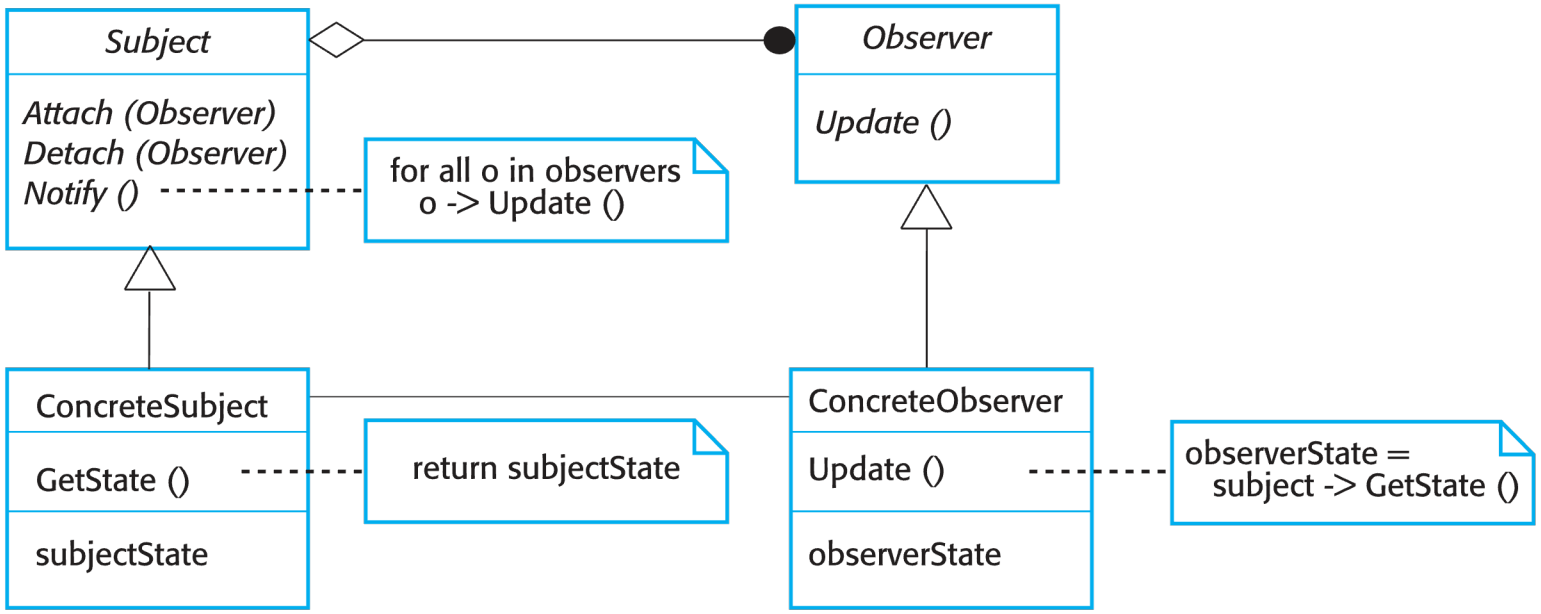
Implementation

- Design patterns make extensive use of inheritance and abstract classes/interfaces
  - Classes that provide concrete implementations for abstract methods can participate in the pattern

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

Observer pattern



Sommerville, *Software Engineering, Tenth Edition*, Figure 7.12

Example: Swing events

PollEv.com/cs5150sp26

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

Factory examples

- Java's Charset.forName()

```
c = Charset.forName("UTF-8");
c = Charset.forName("US-ASCII");
```

- Java’s Collections class

```
List<String> list = Collections.emptyList();
List<String> list2 = Collections.unmodifiableList(new ArrayList<>());
```

Homework: Check what classes are returned by different invocations?

Creation patterns (Factory, Builder)

- More flexible than constructors
  - Can perform imperative operations prior to assigning const fields (C++) (e.g., for validation)
  - Can return other types (union types, subtypes)
  - Can reuse instances ("interning"): memory efficient, caching
- Reduces dependencies of core classes
  - Move string parsing, file I/O, networking code to helper classes
    - User u = UserFactory.fromJsonFile("user.json");
      instead of new User("user.json");
  - 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
  - Holding a resource is a *class invariant*
  - No object leaks -> no resource leaks
- 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;
}
```

Derived from https://en.cppreference.com/w/cpp/language/raii

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!

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` …)

Limitations of RAII

- Resource tied to object lifetime -> lexical scope. What if:
  - You want the resource to survive across function boundaries?
  - You need dynamic lifetime control?
- Depends on well-definedness of object lifetime
  - Heap-based objects: need to be deleted across all execution paths,
  - Relies on exceptions being caught “somewhere” so that “stack unwinding” can happen
  - If no matching handler is found in a program, the function terminate() is called -> so stack unwinding may not happen -> then its up to the OS to kill objects

https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization (see limitations)

(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

Poll

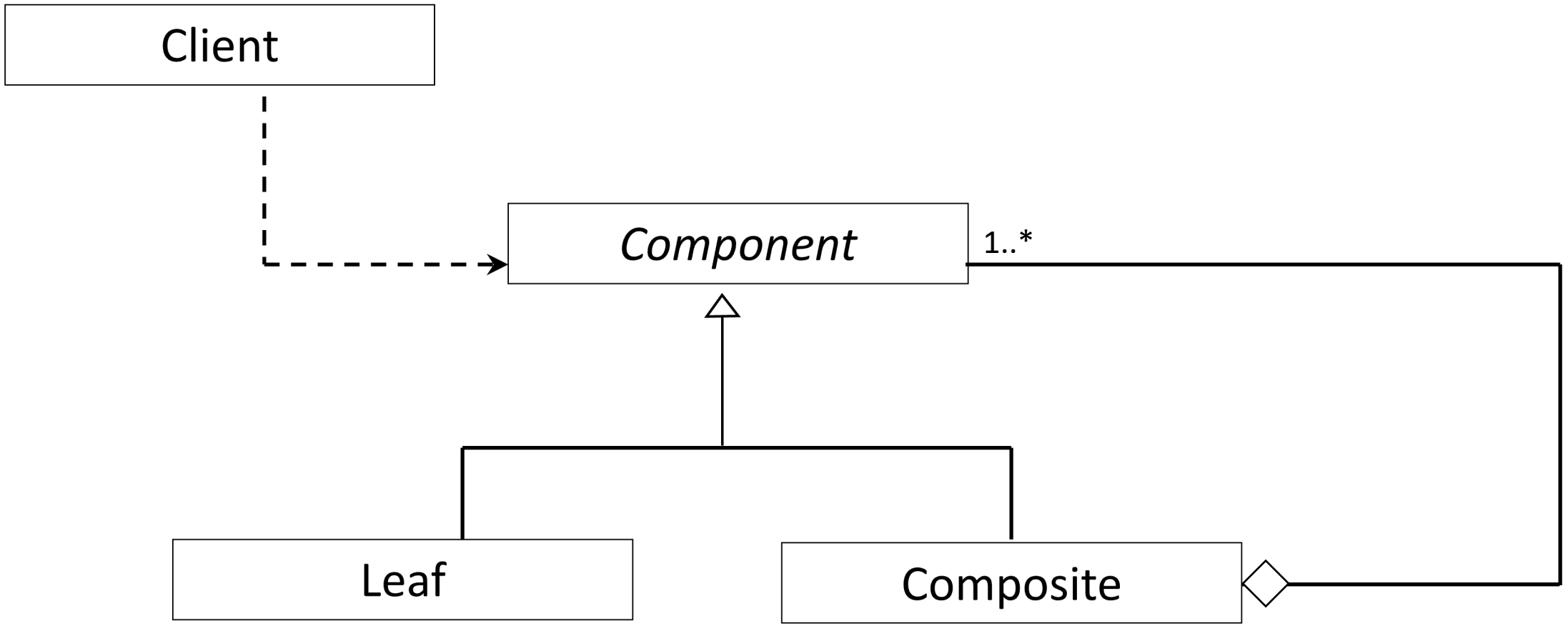
A signal processing application provides many different filters with a common interface. The application can load a text file specifying a filter chain, with one filter type (with parameters) on each line. Which design pattern would help us construct the appropriate filter subclass as each line of the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



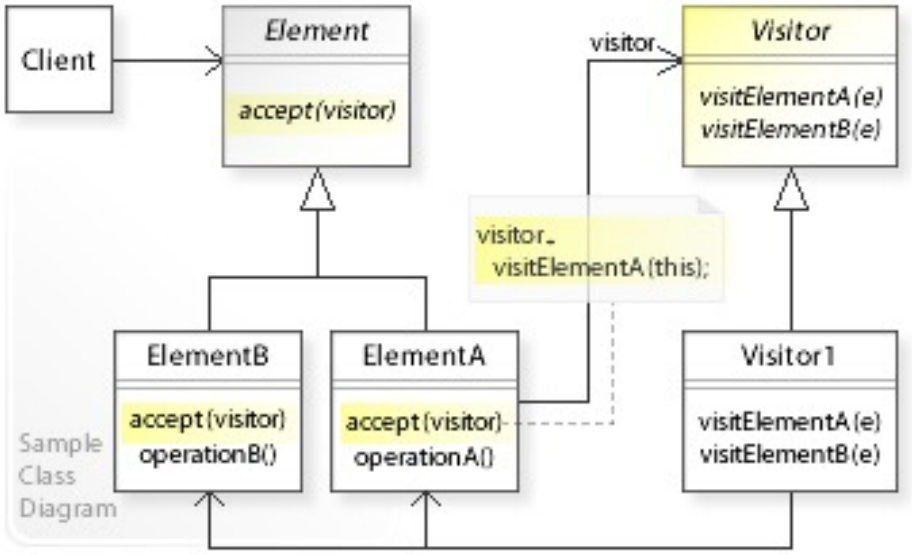
Composite example

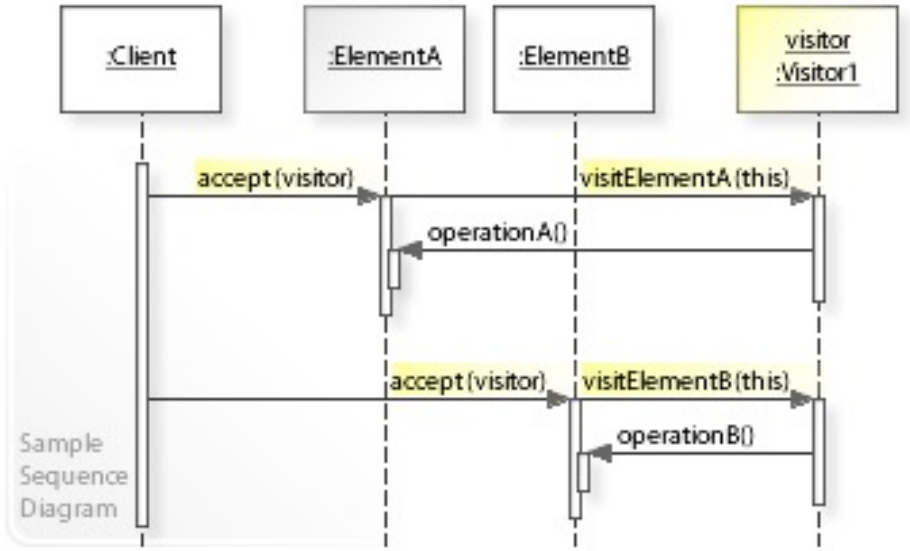
- CS 2110: Tree node
  - Depth
    - Leaf: 1
    - Composite: max([c.depth() for c in children])
  - Contains(x)
    - Leaf: value == x
    - Composite: any([c.contains(x) for c in children])
- File System, Menu Systems

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





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

- Follows Open/Closed principle:
  - *“software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*"