Lecture 9: Program Design

CS 5150, Spring 2025

Administrative reminders

- Assignment A2 due today
- Report #2 due Feb 28: progress, milestones, deliverables, architecture
- Don’t forget to set up meeting with your client
- Assignment A3 coming soon

Previously on 5150…

Design steps

- Given requirements, must design a system to meet them
  - **System architecture**
  - User experience
  - Program design

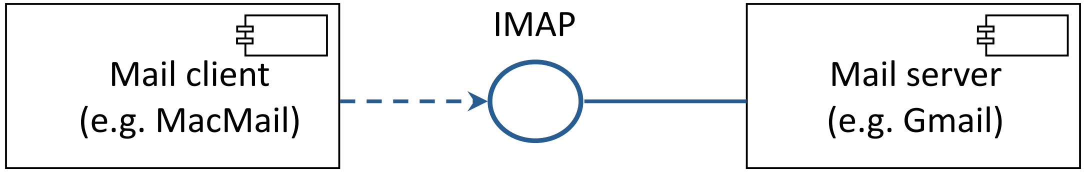
- **Ideal**: requirements are independent of design (avoid implementation bias)
- **Reality**: working on design clarifies requirements
  - Methodology should allow feedback (strength of iterative & agile methods)

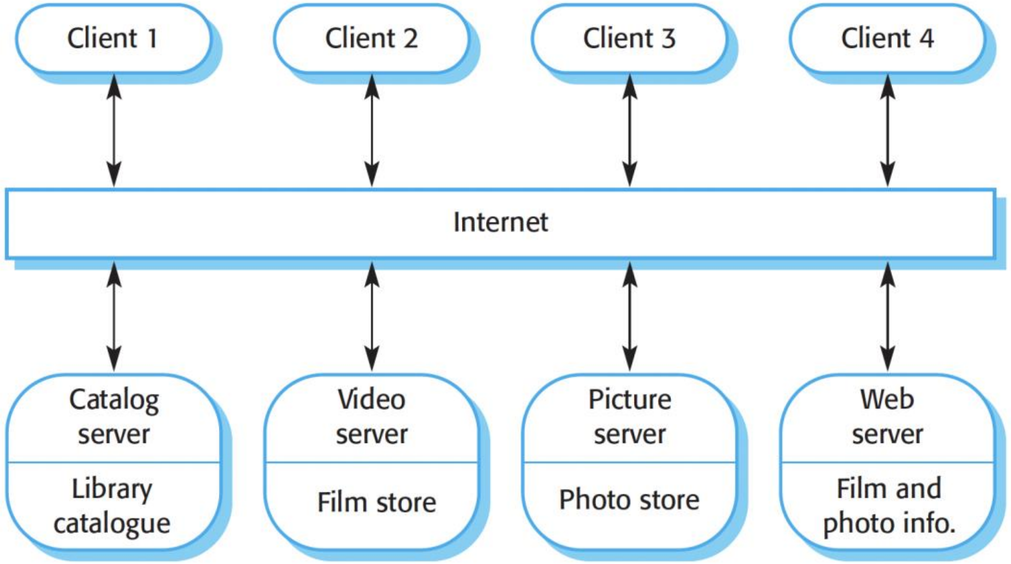
Design principles

- Design is an especially **creative** part of the software development process
  - More a "craft" than a science
  - Many tools are available; must select appropriate ones for a given project
- Strive for simplicity
  - Use modeling, abstraction to (hopefully) find simple ways to achieve complex requirements
  - Designs should be easy to implement, test, and maintain
- Easy to use correctly, hard to use incorrectly
- **Low coupling, high cohesion**

Client/Server

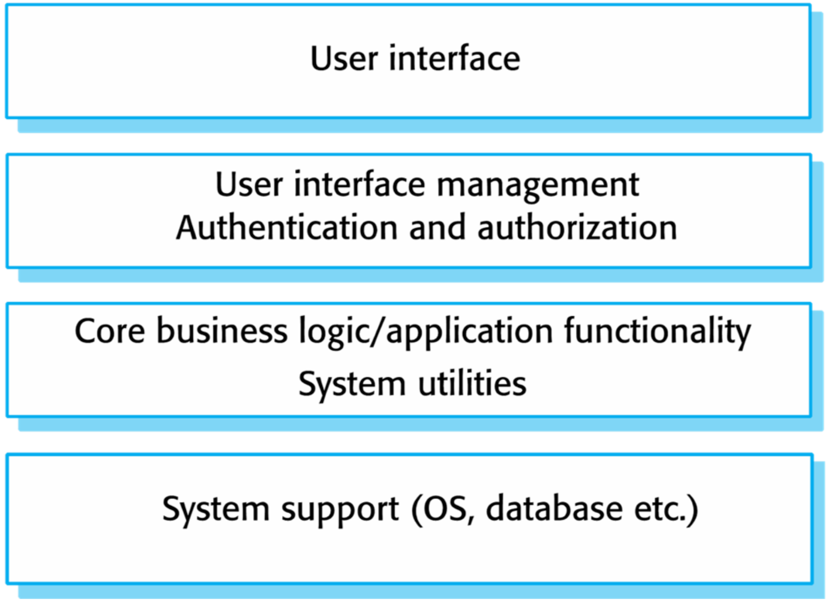
- Control flow in client and server are independent
- Communication follows a protocol
- If protocol is fixed, either side can be replaced independently
- Peer-to-peer: same component can act as both client and server





Layered Architecture

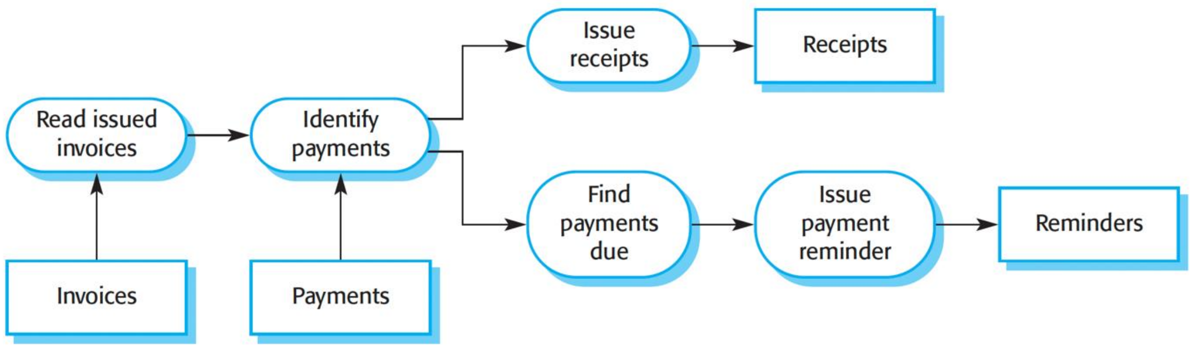
- Partition subsystems into stack of layers
  - Layer provides services to layer directly above
  - Layer relies on services to layer directly below
- Advantage: constrains coupling
- Danger: leaky abstractions
  - Clear separation is difficult
  - May need services of multiple lower layers
  - Performance



Sommerville, *Software Engineering*

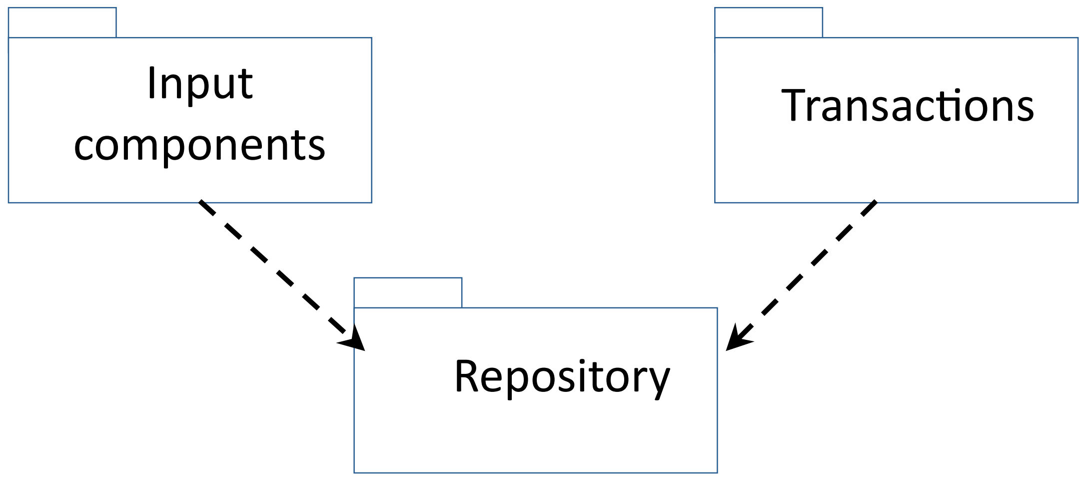
Pipe and Filter

- Transformation components process inputs to produce outputs
  - Subsystems coupled via data exchange
  - Good match for data flow models
  - May be dynamically assembled
  - Limited user interaction
- Applications:
  - Compilers
  - Graphics shaders
  - Signal processing
- Caveats:
  - Awkward to handle events (interactive systems)
  - Rate mismatches if branches merge



Repository

- Couple subsystems via shared data
  - Repository may need to support atomic transactions
- Advantages:
  - Components are independent (low coupling)
  - Centralized state storage (good for backups)
  - Changes propagated easily
- Dangers:
  - Bottleneck / single point of failure

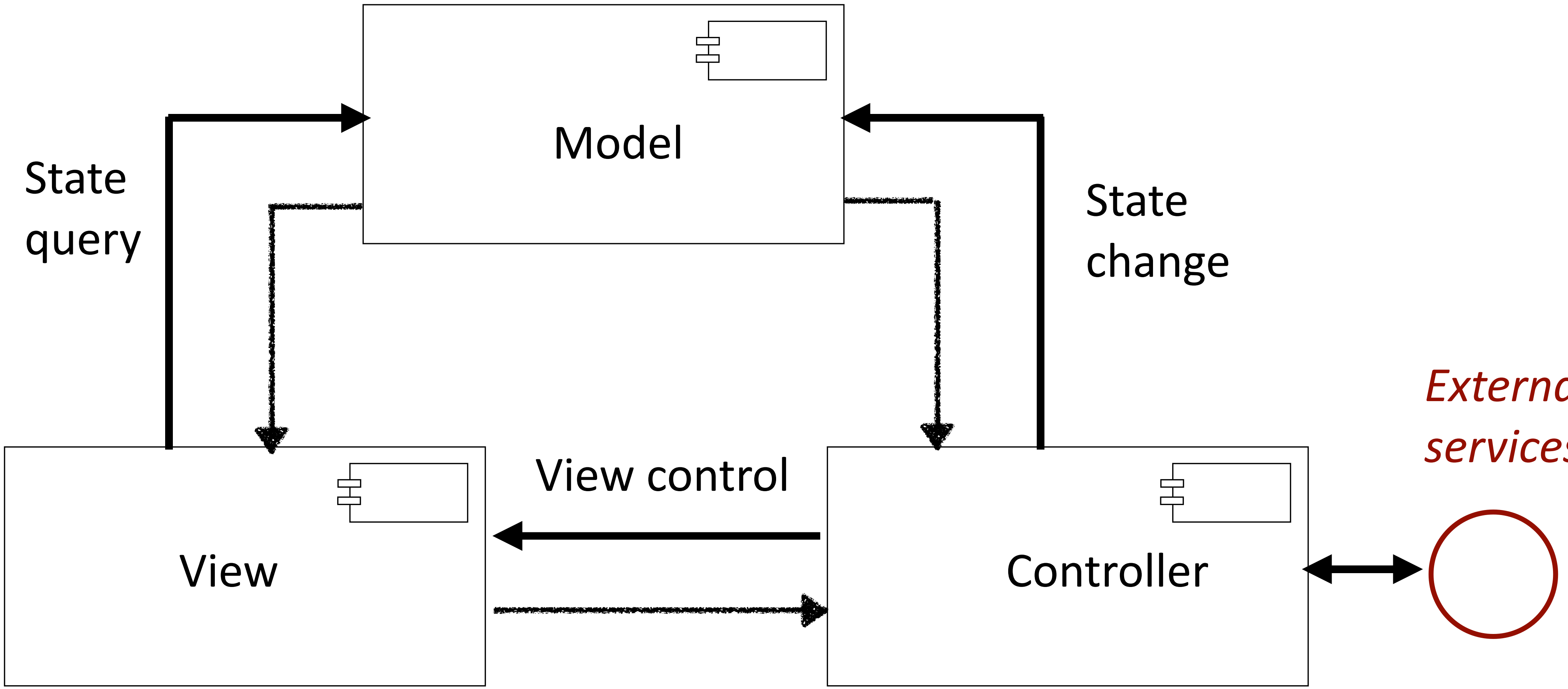


Model-View-Controller

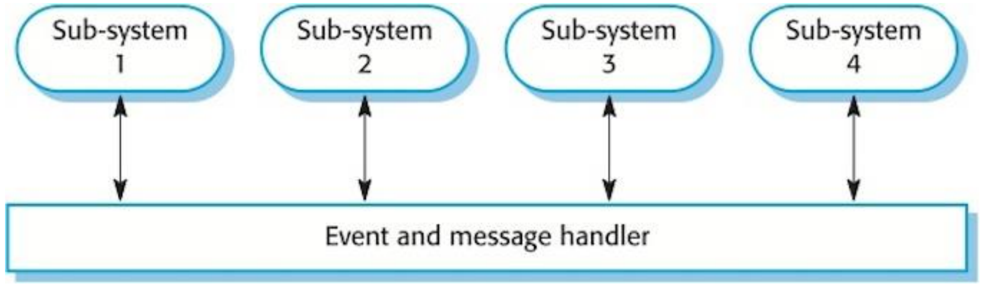
- Beware: many variations
  - Some are architectural styles: system-level responsibilities partitioned into different components
    - Example: **Play Framework** for building web apps
  - Some are program design patterns: functionality divided between different classes
    - Focus on reusable controls
    - Example: **Swing widgets**
    - Variation on which logic is widget-level vs. form-level (MVC vs. MVP)
    - Variation on which classes communicate directly (MVC vs. MVA)
    - Variations in model storage (domain objects, DB record sets, immutable store)

Read more: https://martinfowler.com/eaaDev/uiArchs.html

Component diagram



Publish-subscribe



- Event-driven control
  - Application responds to external stimuli and timeouts
  - No centralized orchestration
- Very loose coupling – components communicate via message broker
  - Easy to extend
  - Difficult to analyze (observer pattern)
    - No control over what (if any) code responds to an event
    - Potential for conflicts (multiple components respond in incompatible ways)
    - Potential for silently dropped events
    - Call stacks may not reflect causality

Deployment concerns

- Dependency conflicts
- Configuration, data sprawl
- OS portability
- Unintended interactions
  - Filesystem has same problems as global variables

- Solution: Encapsulation; but...
  - Deploying on separate machines risks under-utilization

Virtual machines

- Multiple OS instances running on one machine
  - Real hardware is managed by host OS or hypervisor
- Improves hardware utilization, reduces cost
  - Avoids energy consumption by redundant hardware
- Stateful – still risks data sprawl
  - Address with automated administration
- High overhead – software redundancy

- **Examples**: VMware, VirtualBox, Xen, Hyper-V

Containers

- Trade OS heterogeneity for reduced redundancy
- Still isolate filesystem, network without duplicating OS
- Lightweight – new instances start quickly
  - Improves elasticity
- Often encapsulates a single application
- Often treated as stateless (don't write to filesystem)

- Examples: Docker, LXC

"Serverless"

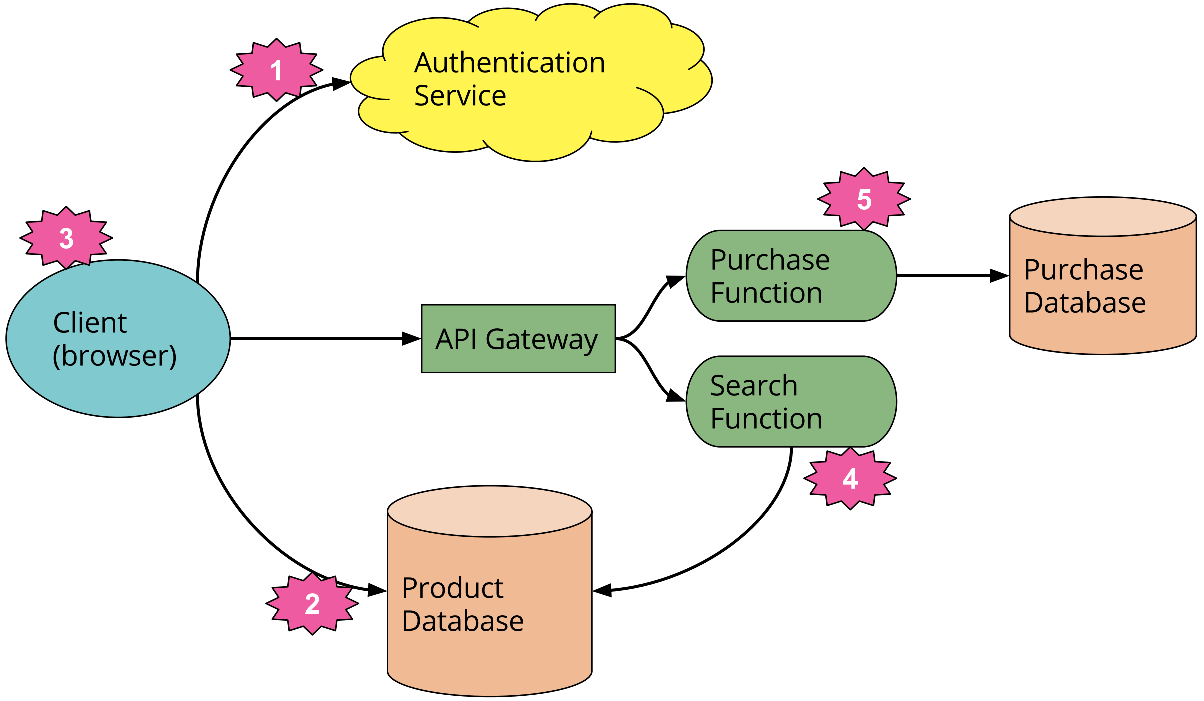
- Computation nodes are stateless, ephemeral, and event-triggered
  - Data store services still persist state, but are application-agnostic
- Application decomposed into event-handler functions
  - Event dispatch, container lifetime managed by platform
- Examples: Amazon Lambda, Azure Functions

Azure Functions

AWS Lambda

https://martinfowler.com/articles/serverless.html

Three-tier vs. serverless



https://martinfowler.com/articles/serverless.html

Microservices

- Components encapsulate services and expose them via standard interfaces. Are ideally binary-replaceable
  - In practice, many frameworks for managing modular applications are language-specific (e.g., OSGi for Java)
  - OOP abstractions like objects, methods are complicated at language boundaries and distributed deployment
- Microservices constrain component definition to **reduce coupling**
  - Language-agnostic protocols (e.g., RESTful HTTP)
  - Independently deployable
- Advantage: More scalable, fault tolerant, rapid roll out
- Disadvantage: Complex monitoring, more points of failure, network delays, testing is challenging
- Examples: Netflix, Amazon, Uber

Design steps

- Given requirements, must design a system to meet them
  - System architecture
  - User experience
  - **Program design**

- **Ideal**: requirements are independent of design (avoid implementation bias)
- **Reality**: working on design clarifies requirements
  - Methodology should allow feedback (strength of iterative & agile methods)

Lecture goals: Program Design

- Distinguish between heavyweight and lightweight design processes
- Document static and dynamic designs using UML diagrams
- Leverage design patterns to reuse solutions to common problems

Program design models

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.

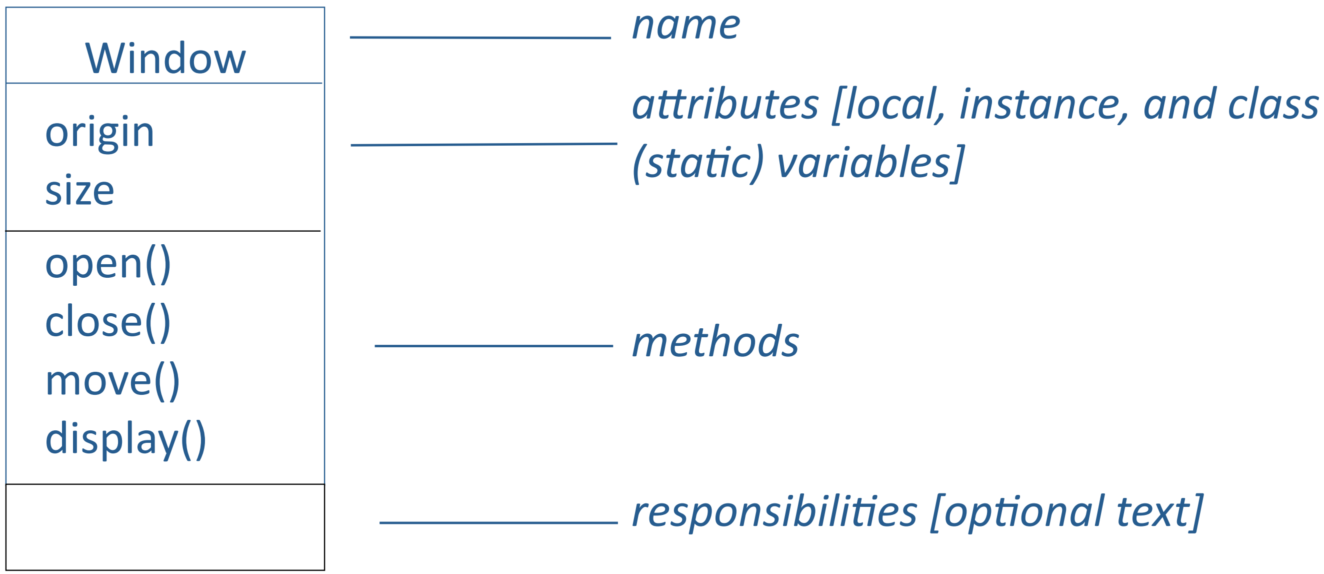
UML model choices

- **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

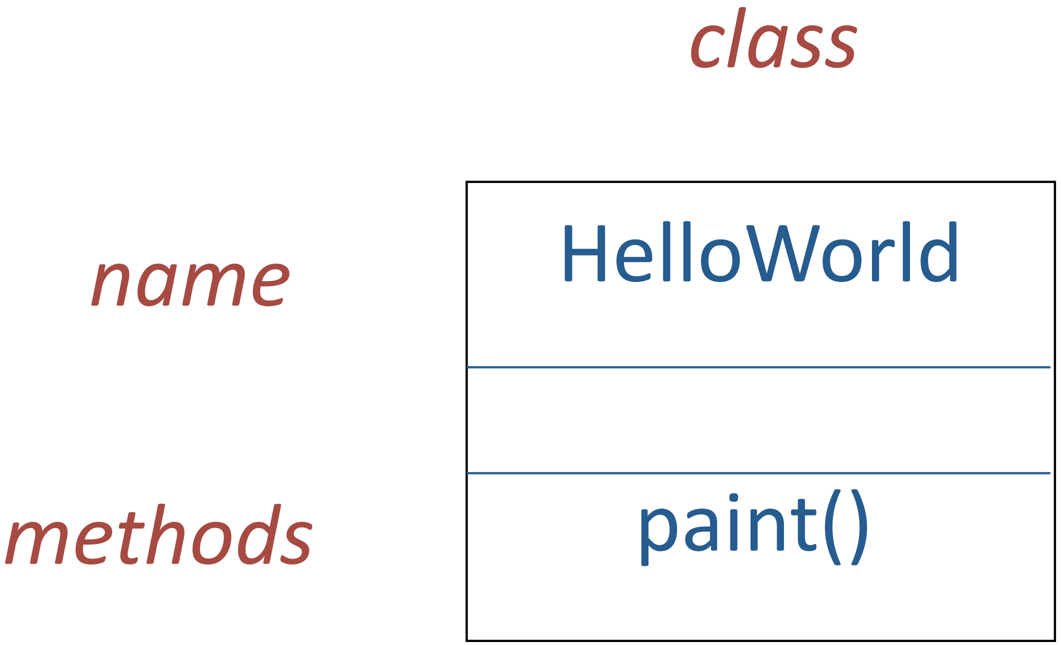
- Class: Set of objects with the same attributes, operations, relationships, and semantics

- "Operation" = "method"

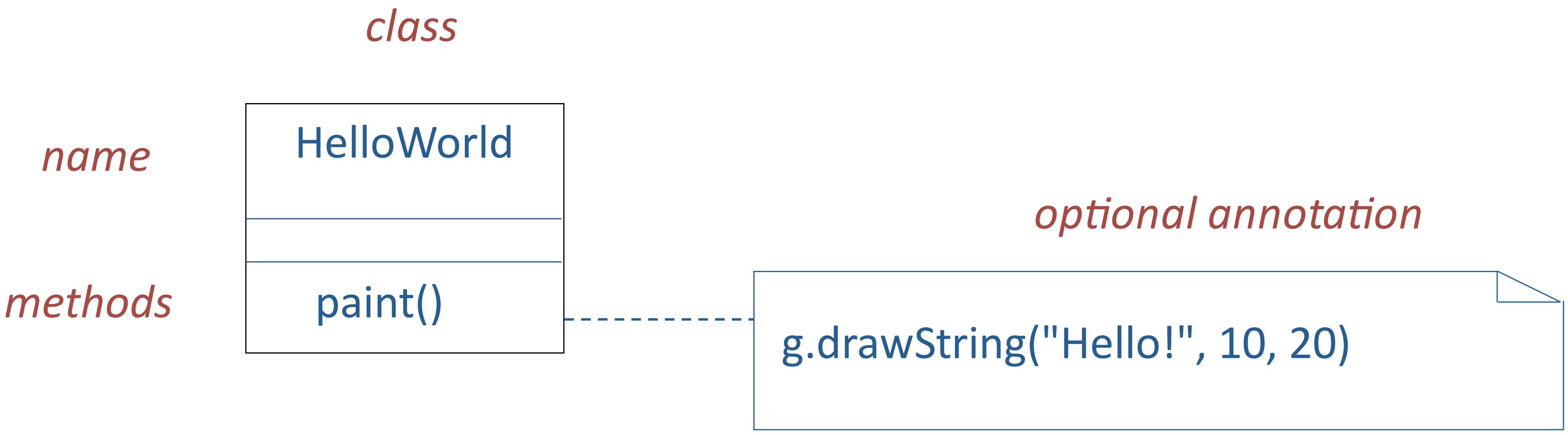


Example: Hello World applet

```java
import java.applet.Applet;
import java.awt.Graphics;
class HelloWorld extends Applet {
  public void paint(Graphics g) {
    g.drawString("Hello!", 10, 20);
  }
}
```



Annotations

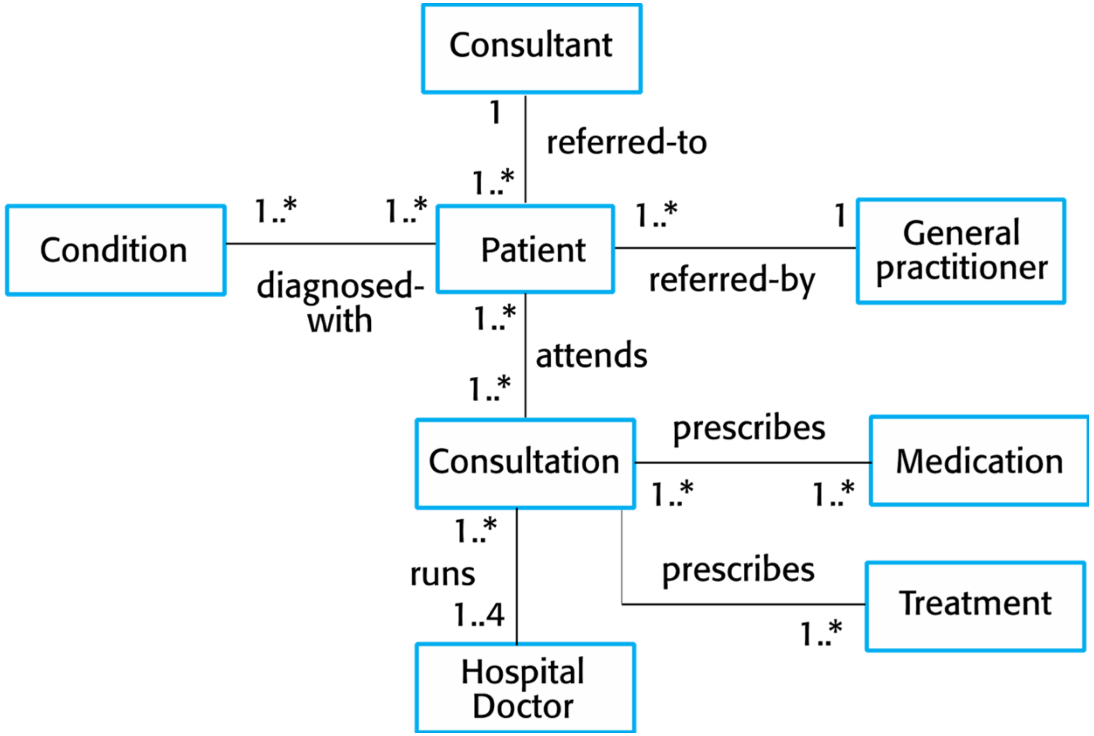


Relationships

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

0..1 *

employer employee



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

Relationships

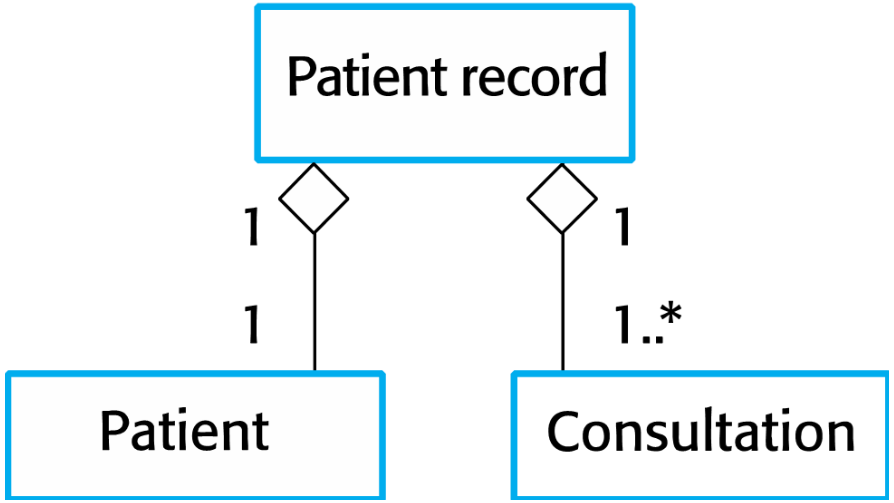
- Dependency
  - A change to one class may affect the semantics of another

- 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

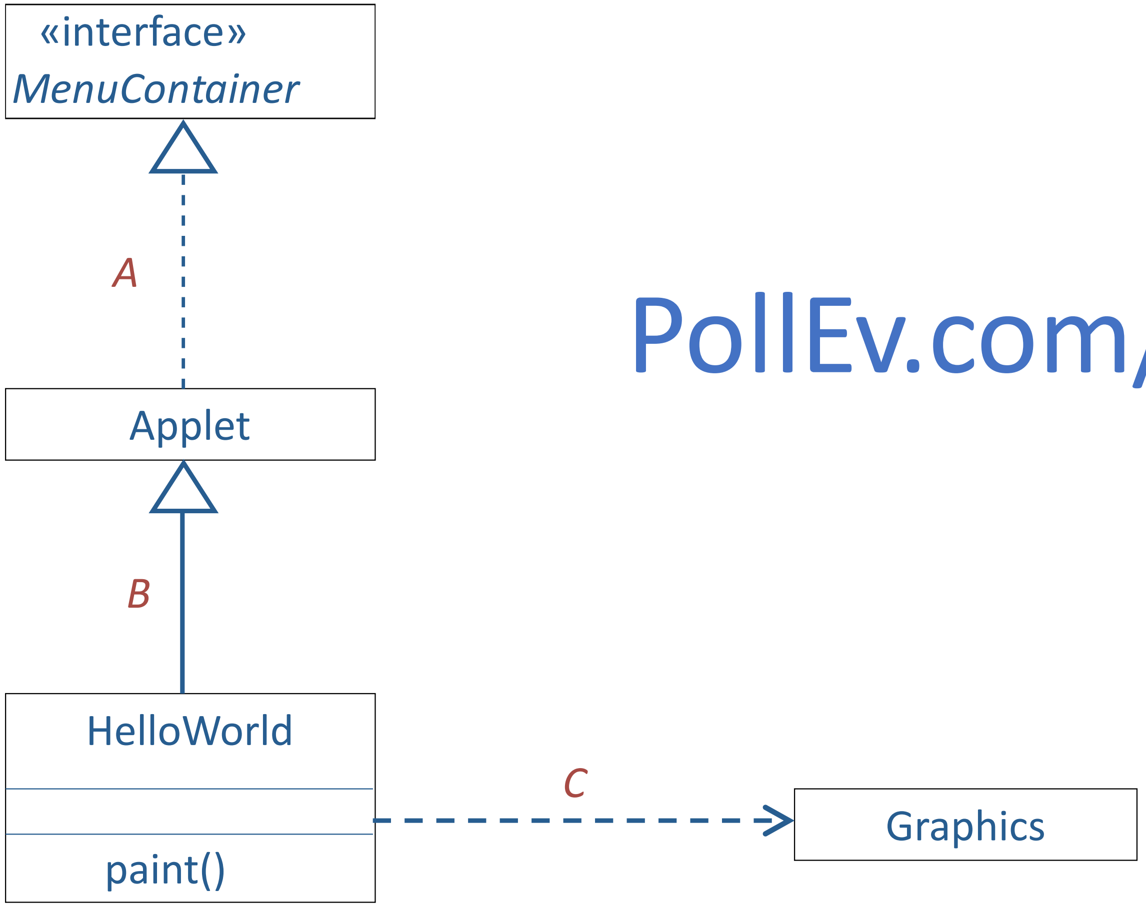
Relationships

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



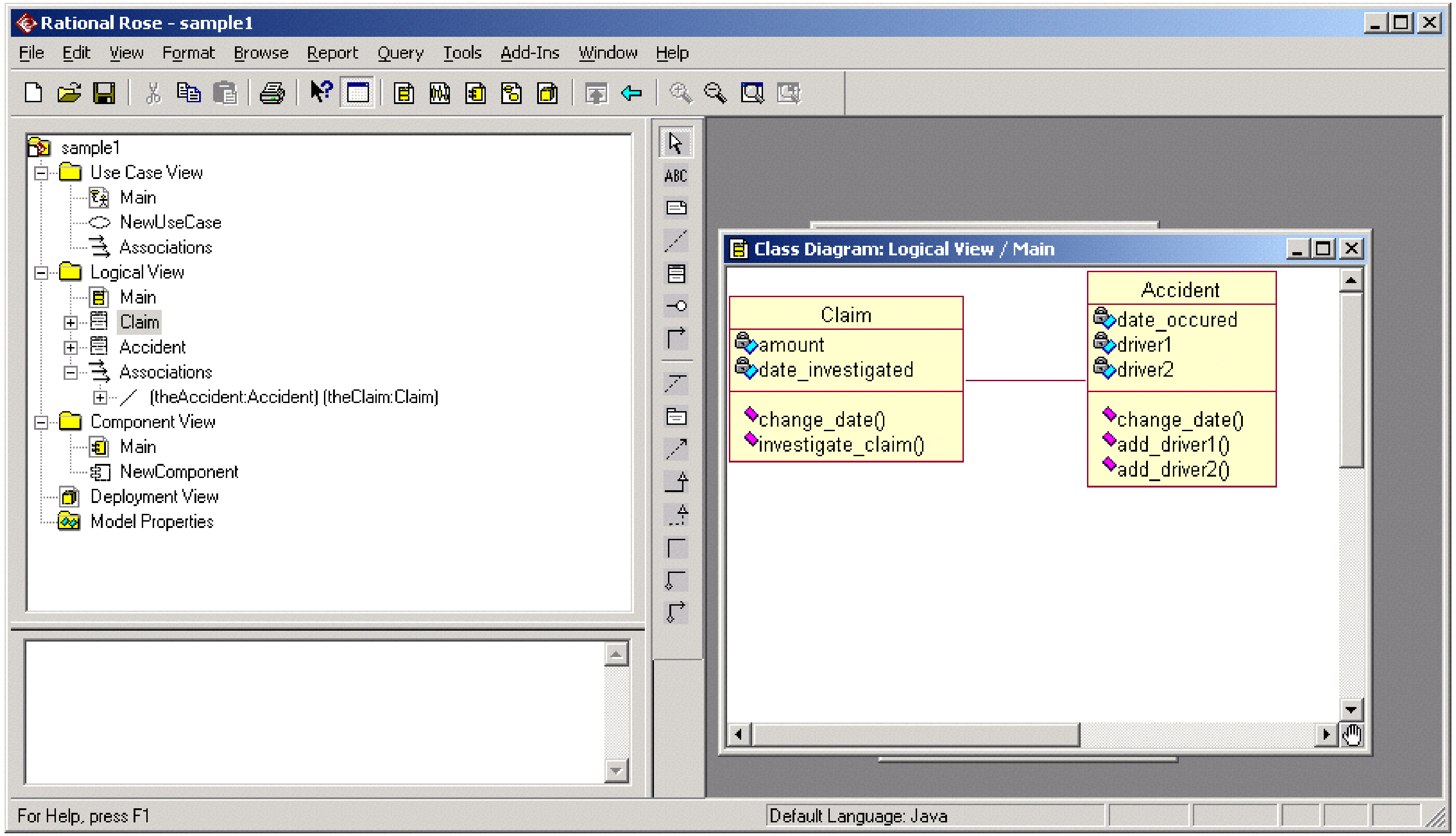
Sommerville, *Software Engineering, Tenth Edition*, Figure 5.13

HelloWorld relationships

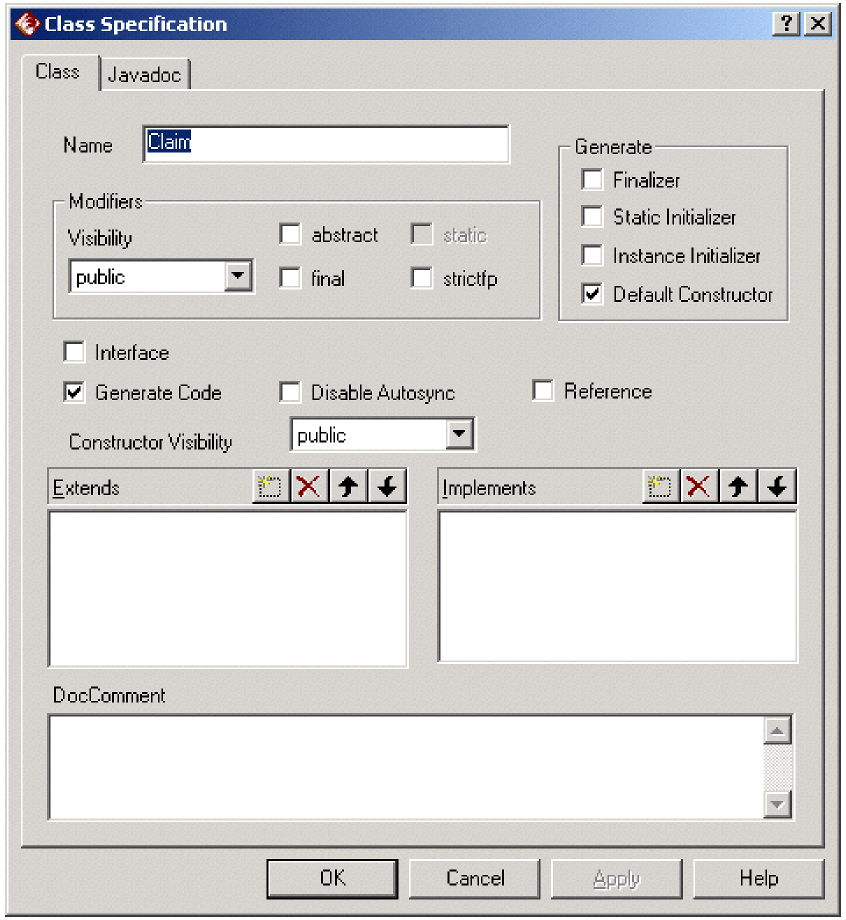


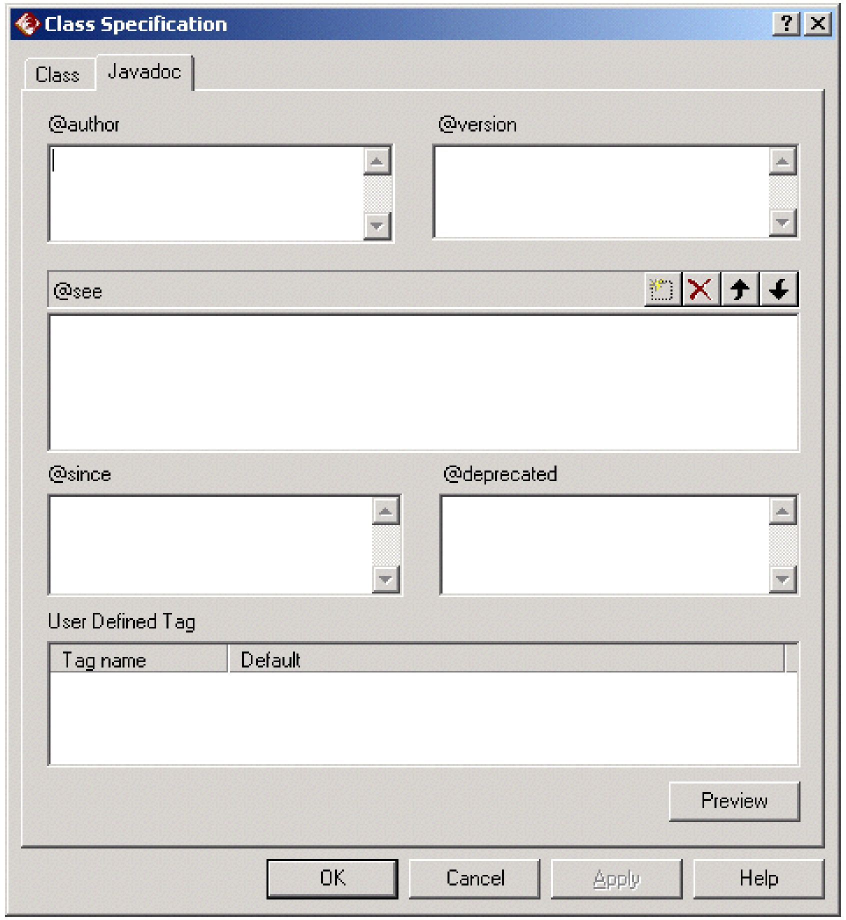
PollEv.com/cs5150sp25

Rational Rose



Rational Rose





Lightweight design

- Less detail
  - Only show "interesting" behaviors and attributes with ownership significance
- Less permanent
  - May only exist on whiteboard during design brainstorming
  - Reduces maintenance of keeping documents in-sync with code
- Less sequential
  - Only design what you need for current task
  - Use lessons from implementation to iterate on designs

- Leverage tooling and modern languages
  - Generate diagrams from source code
  - Generate specifications from comments
  - IDEs highlight attributes and methods

- Still need design activities, documentation to be successful

https://vtk.org/doc/nightly/html/classvtk3DWidget.html

Class design

Given a real-life system, how do you decide which classes to use?

- Step 1: Identify set of candidate classes
  - What terms do users and implementers use to describe the system?
  - Is each candidate class crisply defined?
  - What are the candidate classes' responsibilities? Are they balanced?
  - What attributes and methods does each class need to carry out its responsibilities?

Class design

- Step 2: Refine list of classes
  - Improve clarity of design
  - Increase **cohesion** within classes, reduce **coupling** between classes

Application and solution classes

- Application classes represent application concepts.
  - Use Noun Identification to generate candidate application classes
- Solution classes represent system concepts
  - User interface objects, databases, etc.

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

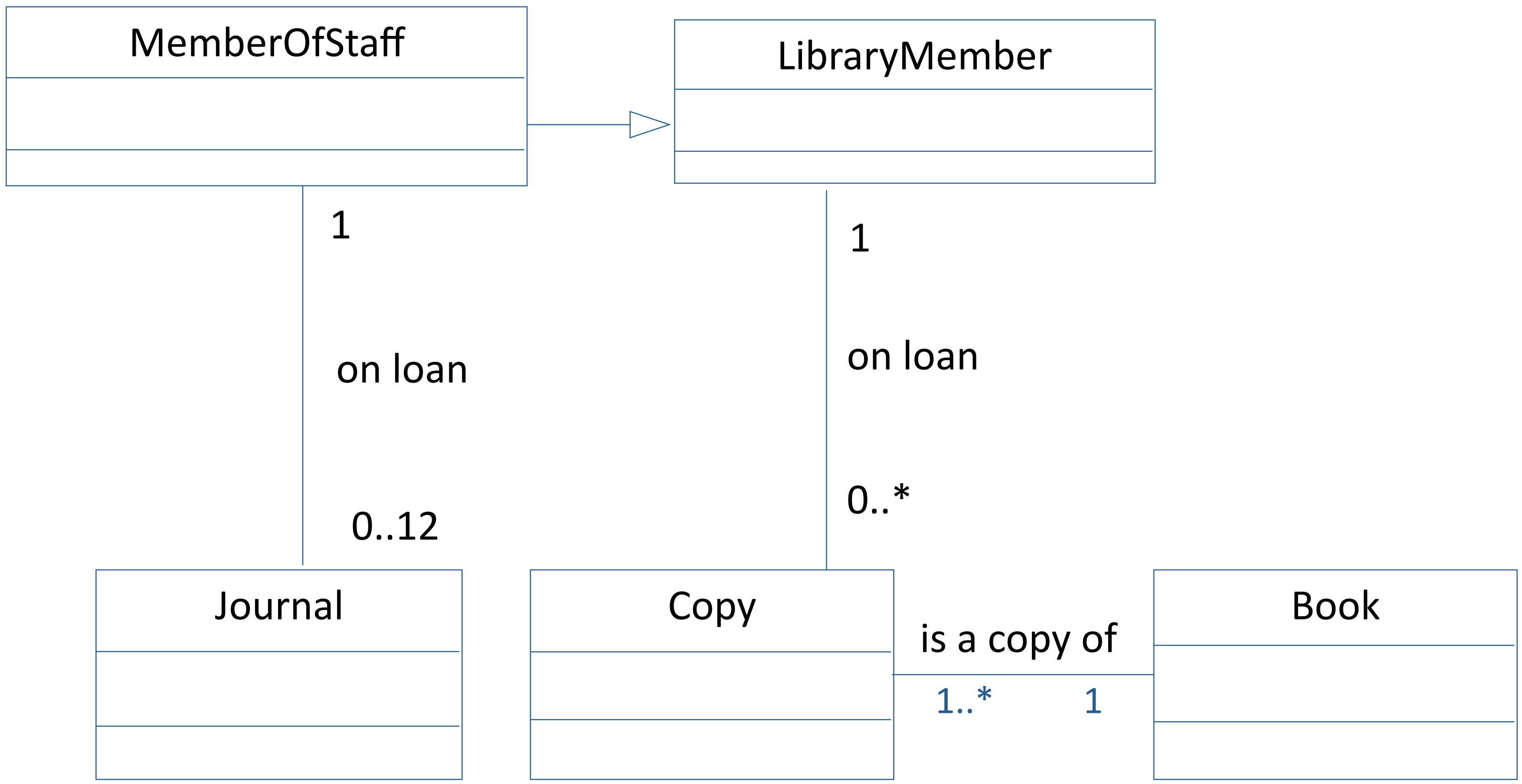
Example: Candidate relations

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.

#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

Classes

AnyClass

attribute1
attribute2

method1()
method2()

or

AnyClass

Objects

anObject:AnyClass

or

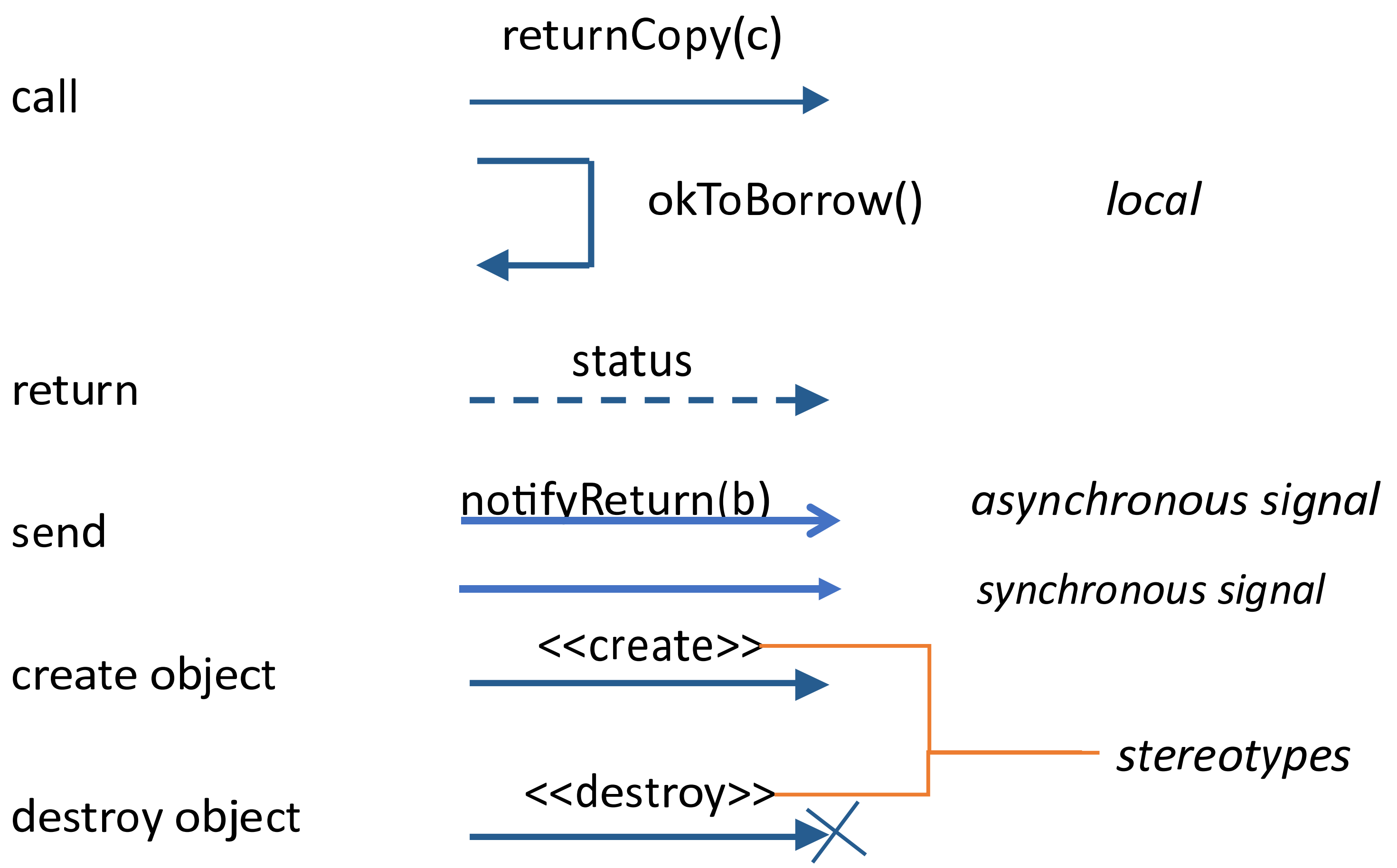
:AnyClass

or

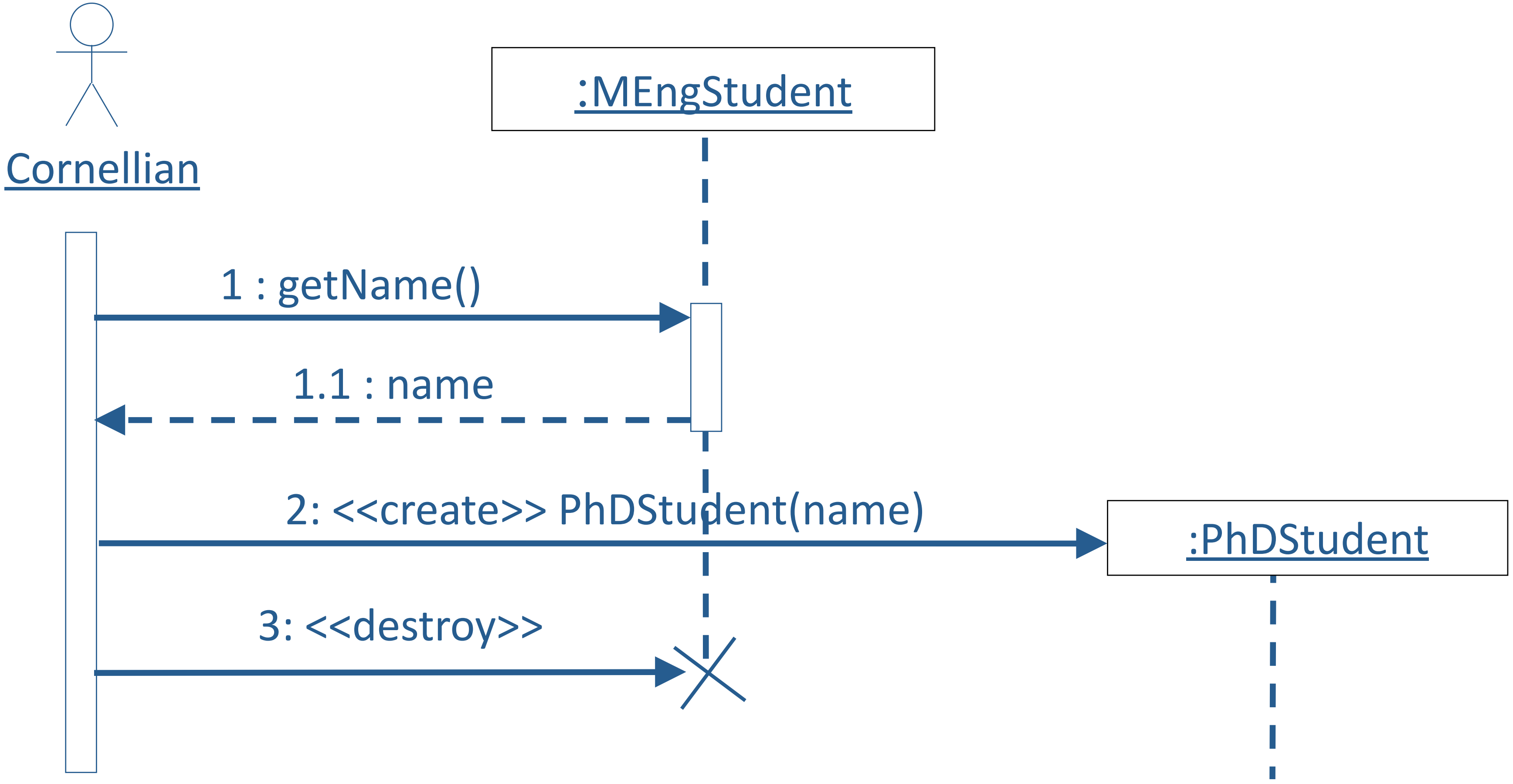
anObject

The names of objects are underlined.

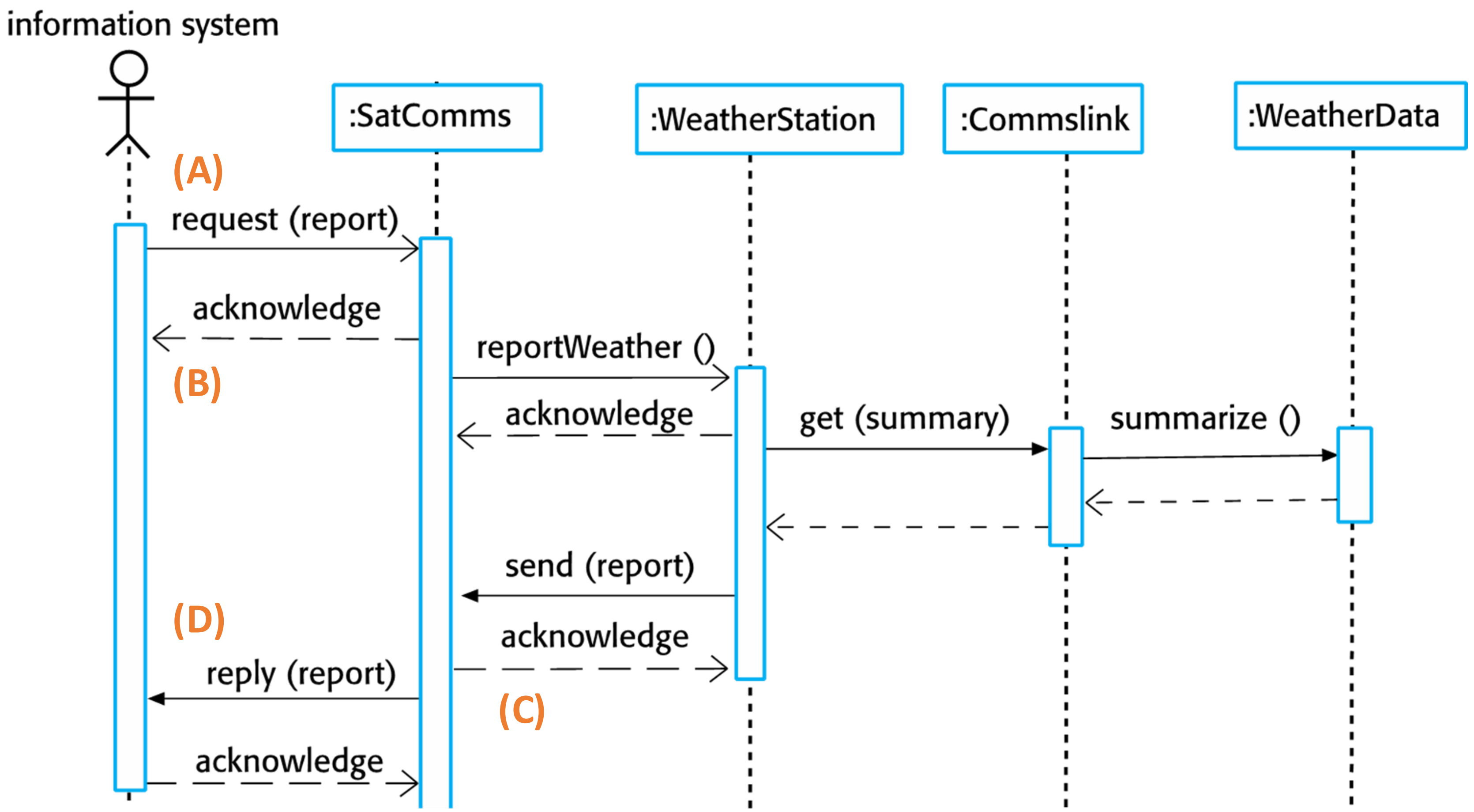
Message notation



Example: Changing student program



Poll: PollEv.com/cs5150sp25



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