Cornell University
Computing and Information Science

CS 5150 Software Engineering

Object Oriented Program Design

William Y. Arms

Program Design

The task of **program design** is to represent the software architecture in a form that can be implemented as one or more executable programs.

Given a system architecture, the program design specifies:

- programs, components, packages, classes, class hierarchies, etc.
- interfaces, protocols (where not part of the system architecture)
- algorithms, data structures, security mechanisms, operational procedures

If the program design is done properly, all significant design decisions should be made before implementation. Implementation should focus on the actual coding.

Models for Program Design

Levels of Abstraction

The complexity of a model depends on its level of abstraction

- High-levels of abstraction show the overall system.
- Low-levels of abstraction are needed for implementation, particularly for:

  unusual or complex parts of the system

  interfaces

Two approaches

- Model entire system at same level of abstraction, but present diagrams with different levels of detail.
- Model parts of system at different levels of abstraction. In practice this is usually an efficient way to use the effort of the design team.

UML Models

UML models (**diagrams** and **specifications**) can be used for almost all aspects of program design.

- **Diagrams** give a general overview of the design, showing the principal elements and how they relate to each other.
- **Specifications** provides details about each element of the design. The specification should have sufficient detail that they can be used to write code from.

In heavyweight software development processes, the entire specification is completed before coding begins.

In lightweight software development processes, an outline specification is made before coding, but the details are completed as part of the coding process, using language based tools such as Javadocs.

List of Models in UML

Models used mainly for requirements

- Use case diagram shows a set of use cases and actors and their relationships.

Models used mainly for systems architecture

- Component diagram shows the organization and dependencies among a set of components.
- Deployment diagram shows the configuration of processing nodes and the components that live on them.

Models used mainly for program design

- Class diagram shows a set of classes, interfaces, and collaborations with their relationships.
- Object diagram shows a set of objects and their relationships.

List of Models in UML

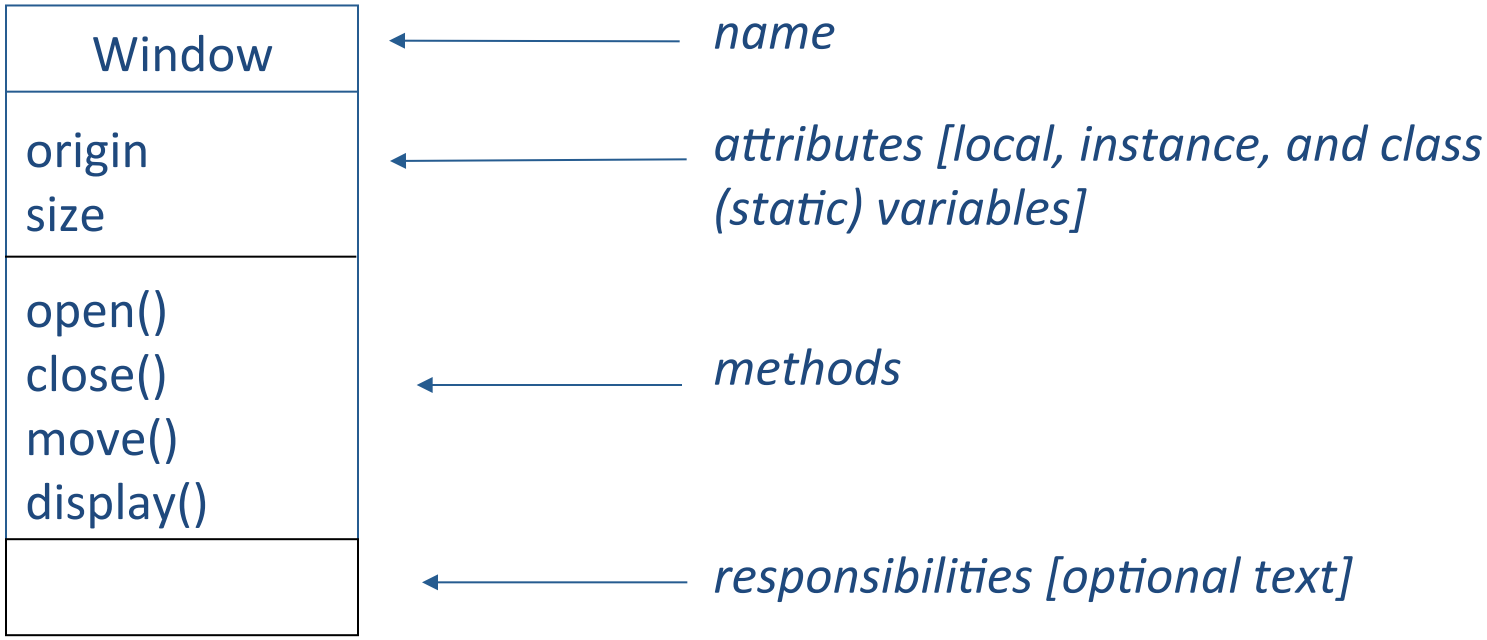
Models for interactive aspects of systems

These models can be used for requirements or program design.

- Interaction diagram: shows set of objects and their relationships including messages that may be dispatched among them
  - Sequence diagrams: time ordering of messages
  - Collaboration diagrams: structural organization of objects that send and receive messages
- Statechart diagram shows a state machine consisting of states, transitions, events, and activities.
- Activity diagram (flowchart) shows the flow from activity to activity within a system.

Class Diagrams

A **class** is a description of a set of objects that share the same attributes, methods, relationships, and semantics.



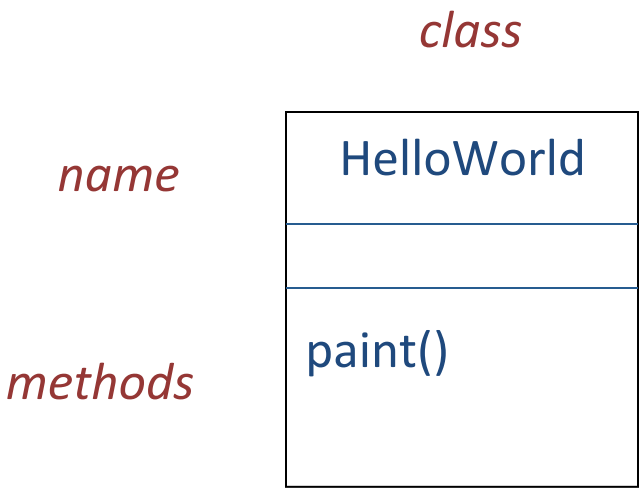
Note on terminology. This course uses the term methods for the operations that a class supports. UML uses the less familiar term operations for this purpose.

The "Hello, World!" Applet

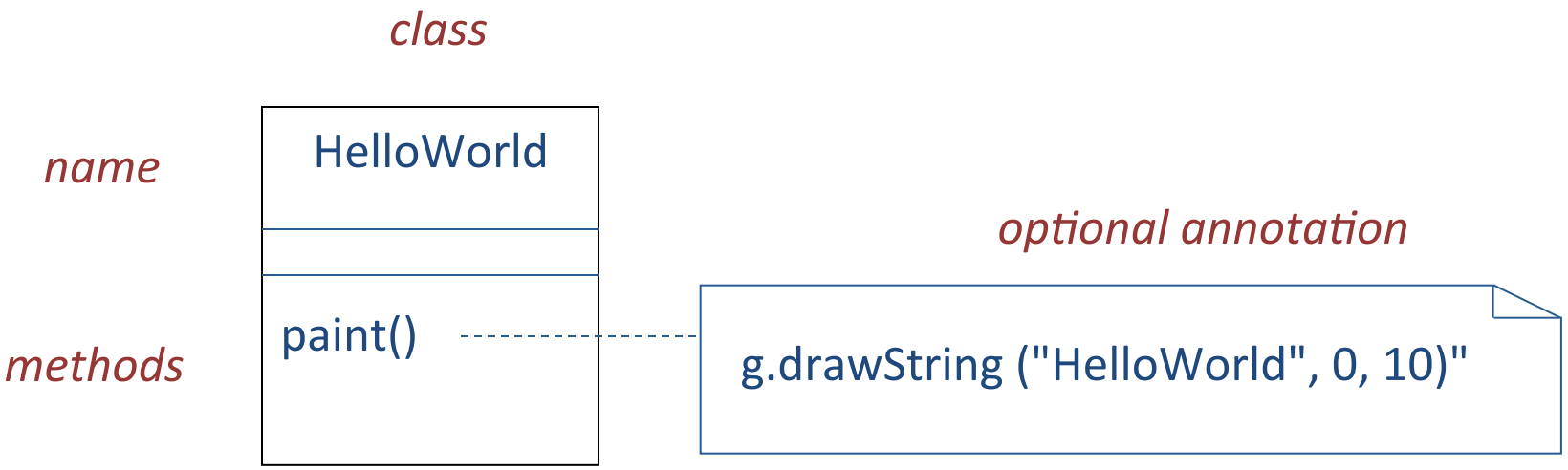
```
import java.awt.Graphics;
class HelloWorld extends java.applet.Applet {
    public void paint (Graphics g) {
        g.drawString ("Hello, World!", 10, 10);
    }
}
```

Example from: BRJ

The HelloWorld Class



The HelloWorld Class



Notation: Annotation or Note

A **note** is a symbol for attaching constraints and comments to an element or a collection of elements.

Notation: Relationships

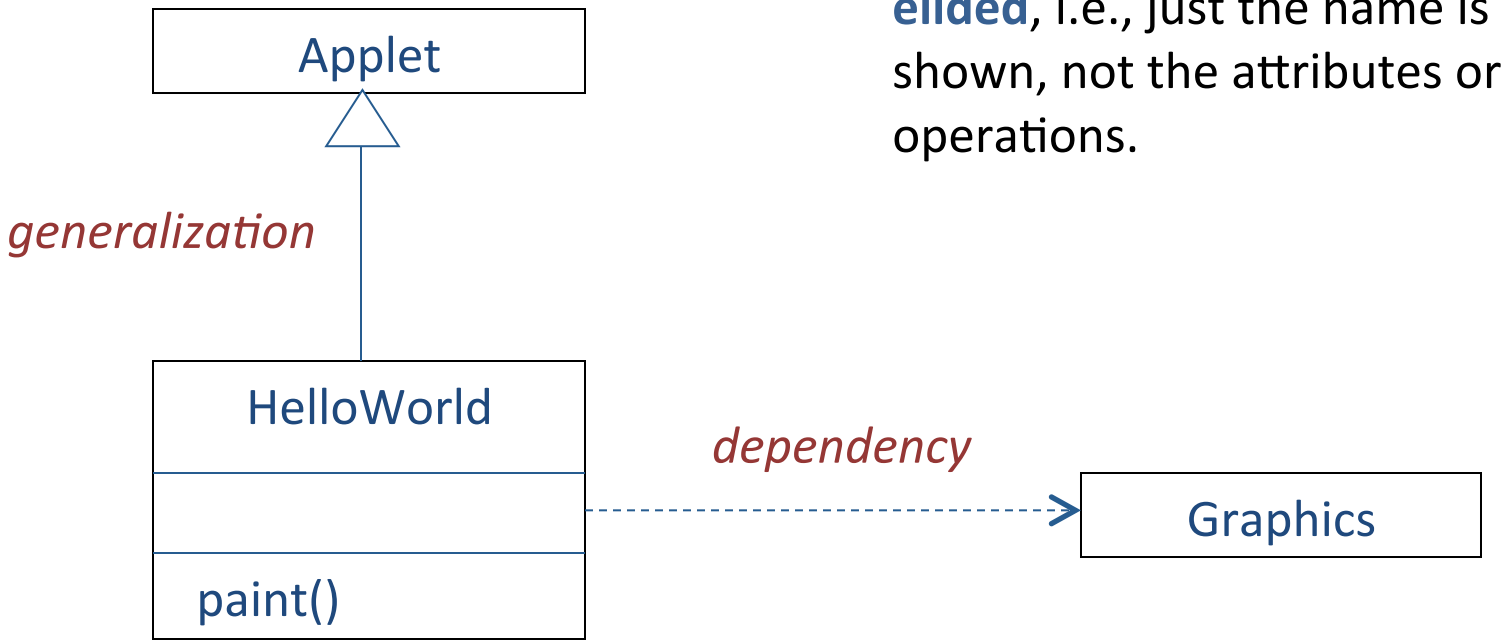
A **dependency** is a semantic relationship between two things in which a change to one may effect the semantics of the other.

A **generalization** is a specialization/generalization relationship is which objects of the specialized element (child) are substitutable for objects of the generalized element (parent).

A **realization** is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out.

The HelloWorld Class

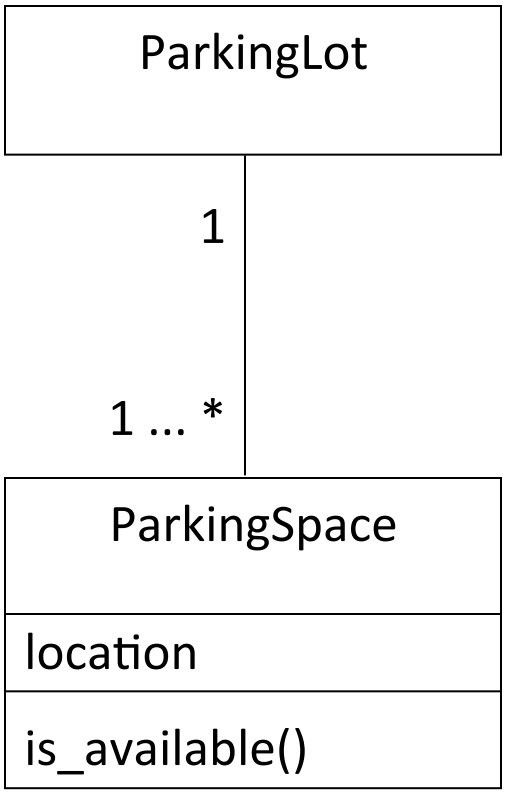
Note that the Applet and Graphics classes are shown **elided**, i.e., just the name is shown, not the attributes or operations.



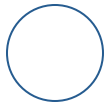
Notation: Relationships

An **association** is a structural relationship that describes a set of links, a link being a connection among objects.

Relationships



Notation: Interface



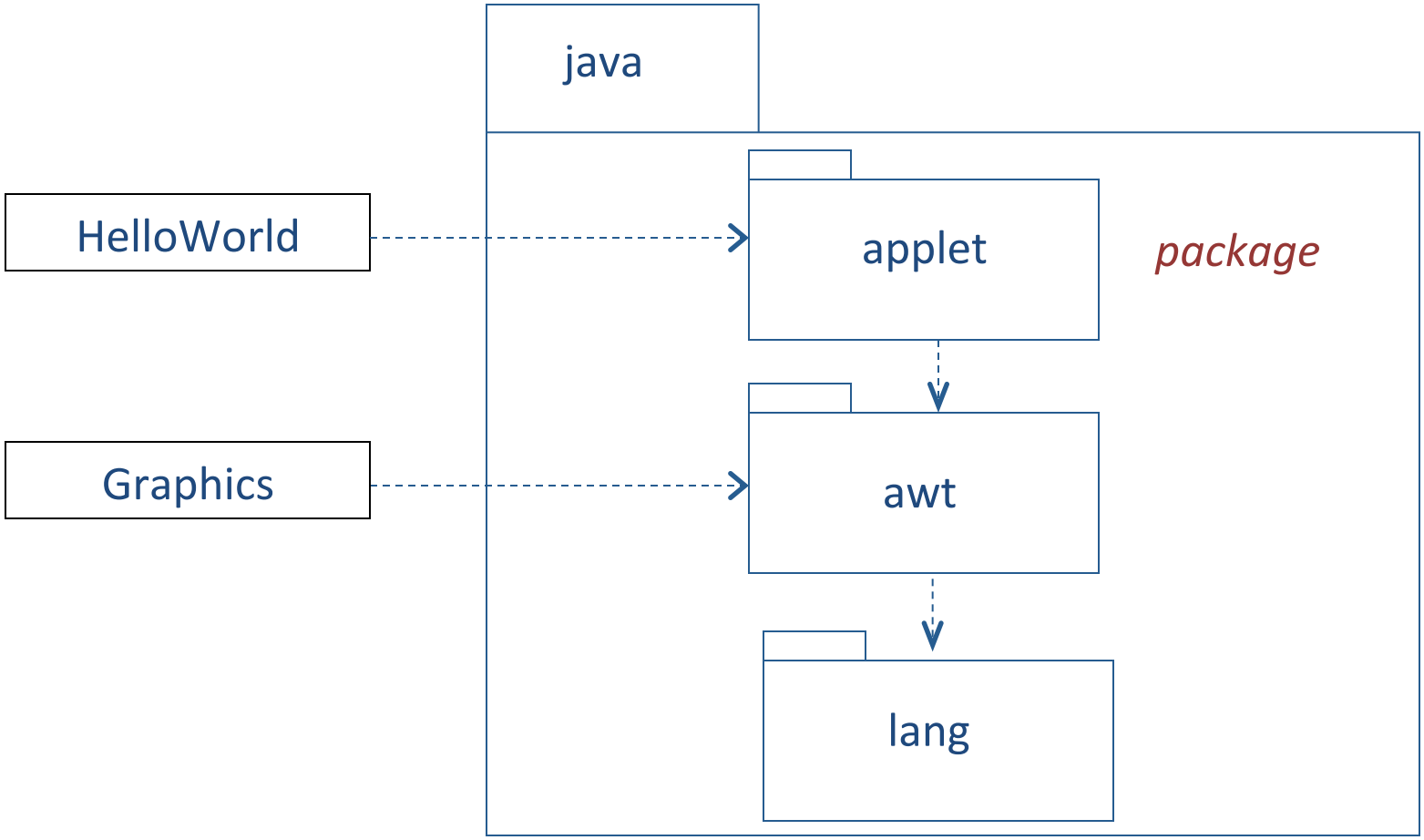
ISPX

An **interface** is a collection of methods that specify a service of a class or component, i.e., the externally visible behavior of that element.

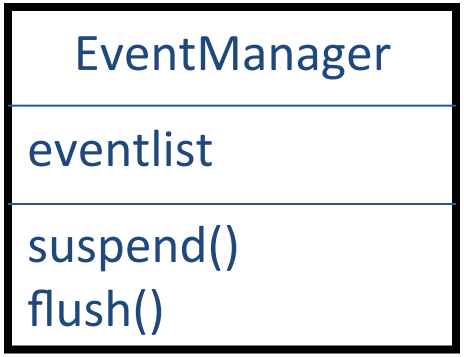
Notation: Package

A **package** is a general-purpose mechanism for organizing elements into groups.

Packaging Classes

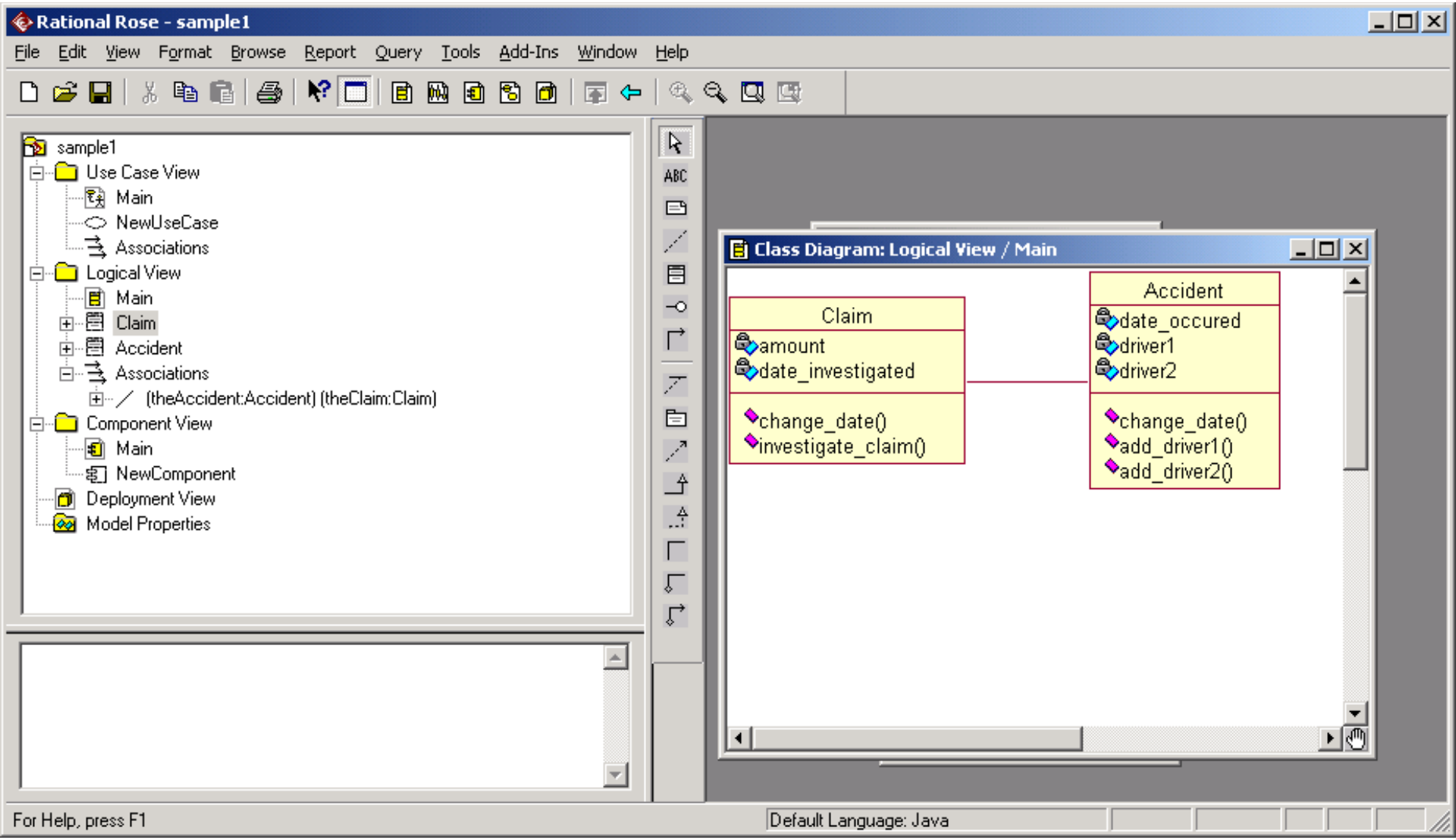


Notation: Active Class

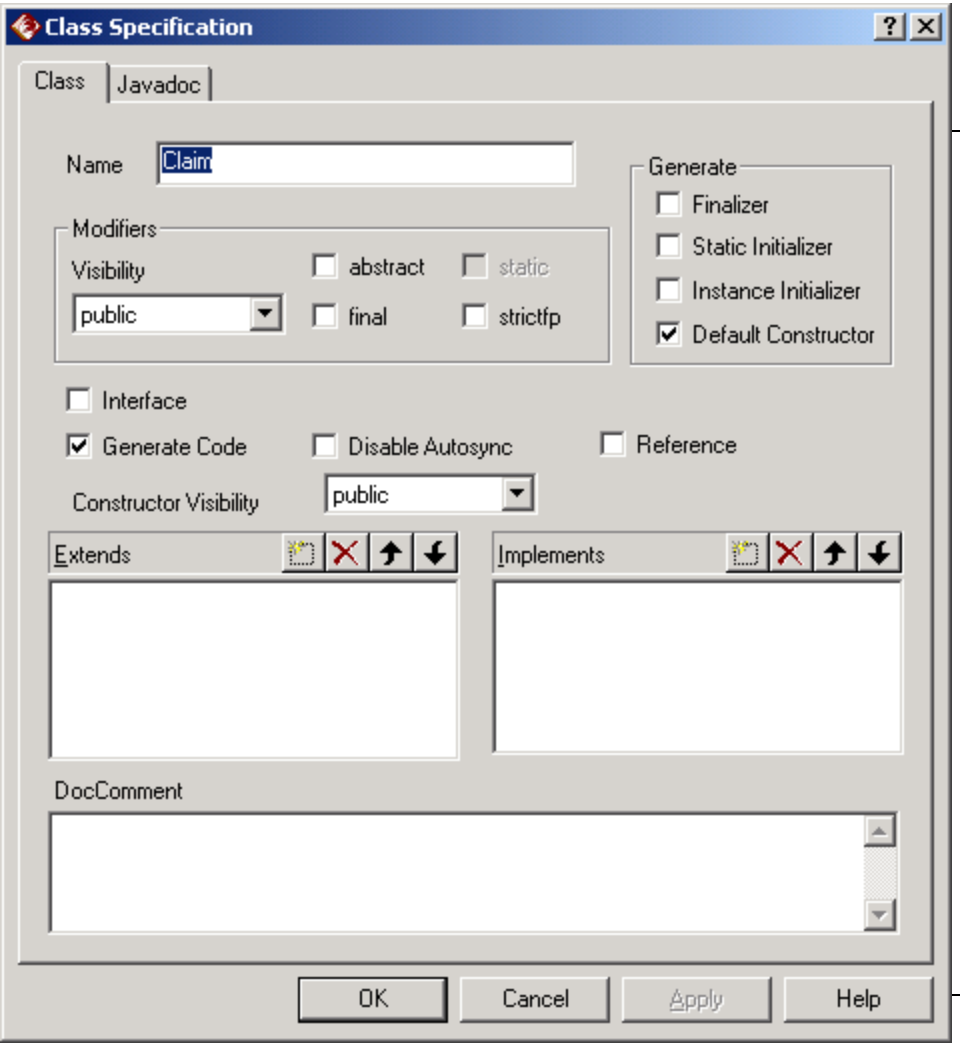


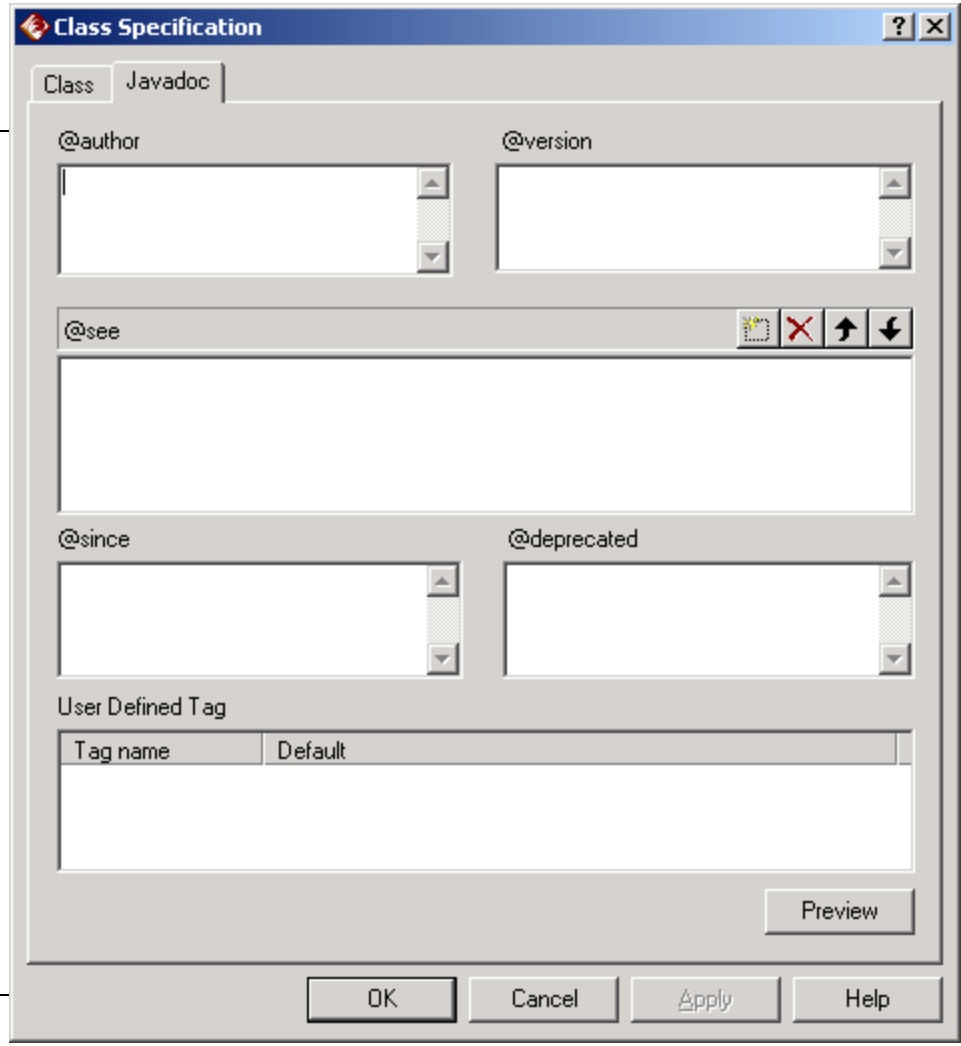
An **active class** is a class whose objects own one or more processes or threads and therefore can initiate control activity. When instantiated, the class controls its own execution, rather than being invoked or activated by other objects.

Rational Rose: A Typical Class Diagram



Specification Fields





Deciding which Classes to Use

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

Step 1. Identify a set of candidate classes that represent the system design.

- What terms do the users and implementers use to describe the system? These terms are candidates for classes.
- Is each candidate class crisply defined?
- For each class, what is its set of responsibilities? Are the responsibilities evenly balanced among the classes?
- What attributes and methods does each class need to carry out its responsibilities?

Deciding which Classes to Use

Step 2. Modify the set of classes

Goals:

- Improve the clarity of the design

  If the purpose of each class is clear, with easily understood methods and relationships, developers are likely to write simple code, which future maintainers can understand and modify.

- Increase coherence within classes, and lower coupling between classes.

  Aim for high cohesion within classes and weak coupling between them.

Application Classes and Solution Classes

A good design is often a combination of application classes and solution classes.

- **Application classes** represent application concepts.

  Noun identification is an effective technique to generate candidate application classes.

- **Solution classes** represent system concepts, e.g., user interface objects, databases, etc.

Noun Identification: A Library Example

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.

Noun Identification: A Library Example

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.

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

Relations between Classes

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

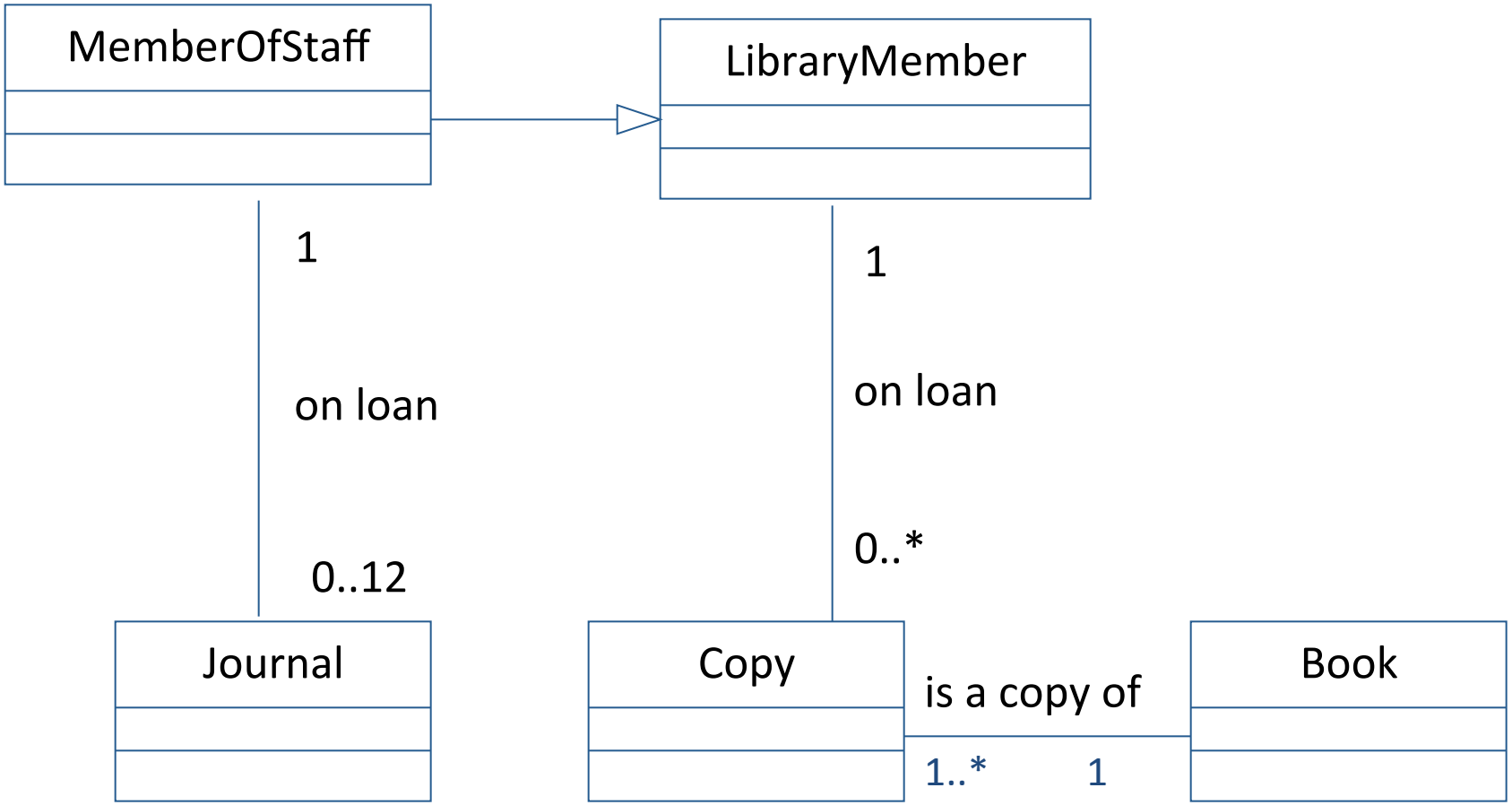
Is Item needed?

Methods

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

Item not needed yet.

A Possible Class Diagram



From Candidate Classes to Completed Design

Methods used to move to final design

Reuse: Wherever possible use existing components, or class libraries. They may need extensions.

Restructuring: Change the design to improve understandability, maintainability, etc. Techniques include merging similar classes, splitting complex classes, etc.

Optimization: Ensure that the system meets anticipated performance requirements, e.g., by changed algorithms or restructuring.

Completion: Fill all gaps, specify interfaces, etc.

Design is iterative

As the development 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.

Rough Sketch: Wholesale System

Design is empirical and iterative. The following very artificial example, gives an idea of the process.

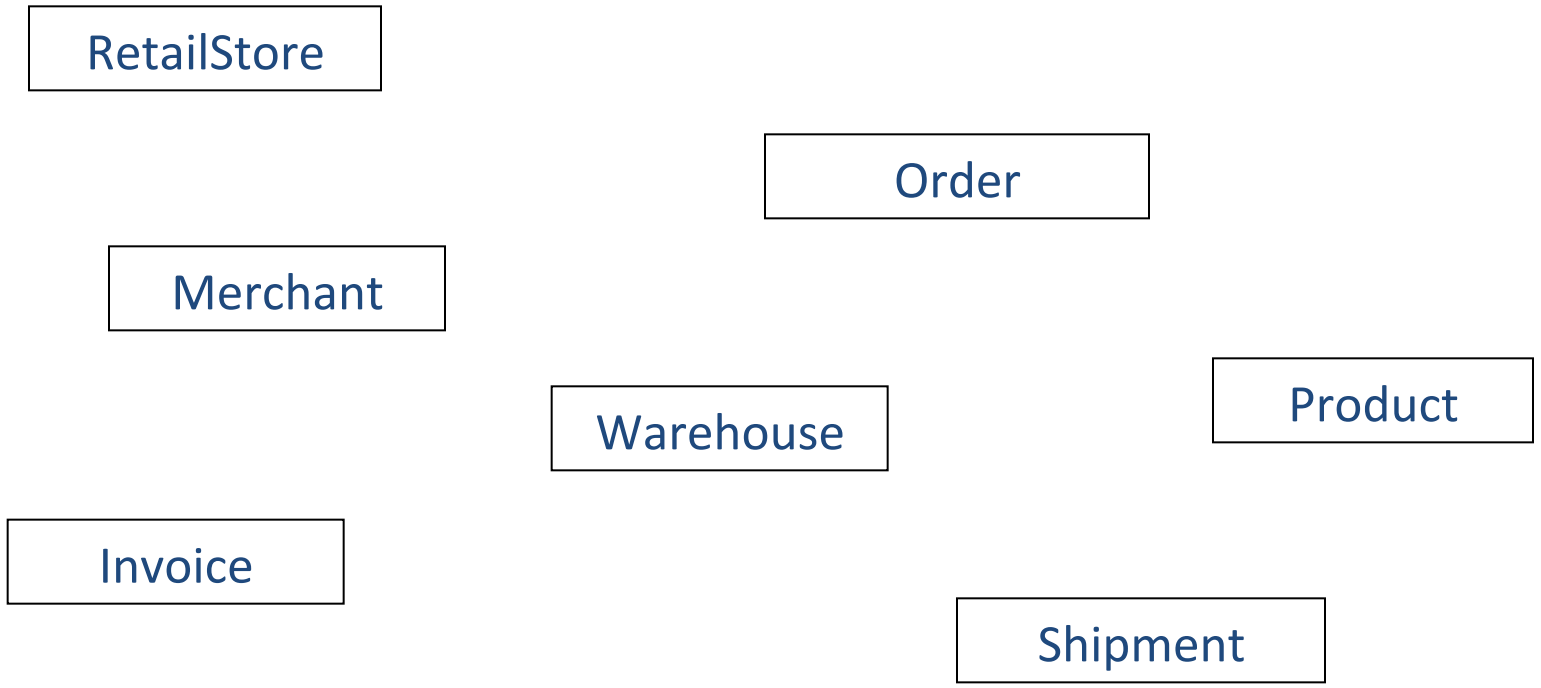
Example

A wholesale merchant supplies retail stores from stocks of goods in a warehouse. A comprehensive set of requirements have been identifier, and a preliminary software architecture defined, but the relationships among the various objects in the system have not been determined.

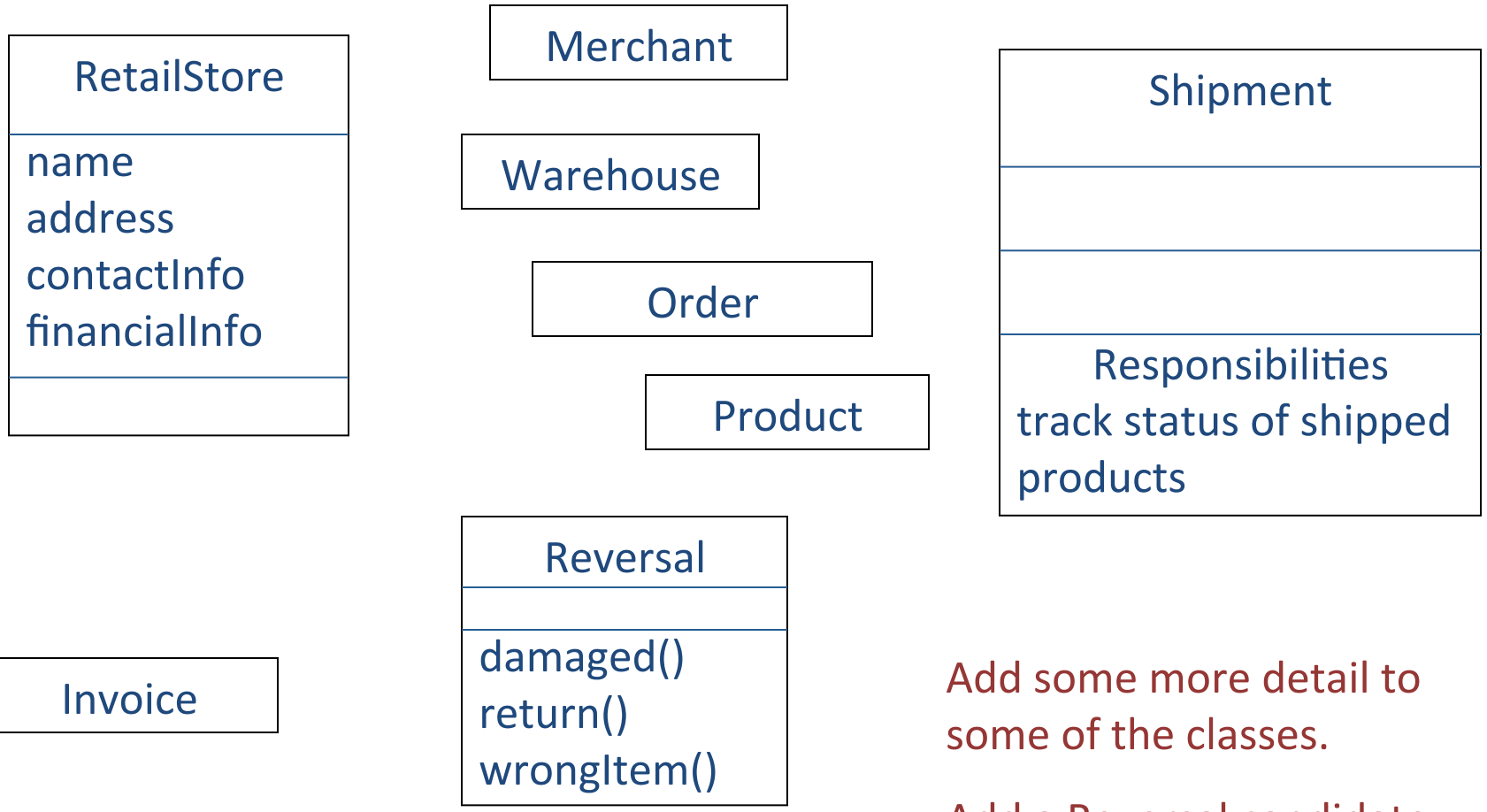
What classes would you use to model this business?

Rough Sketch: Wholesale System

Noun identification has found a large number of candidate classes. Here are some of them.



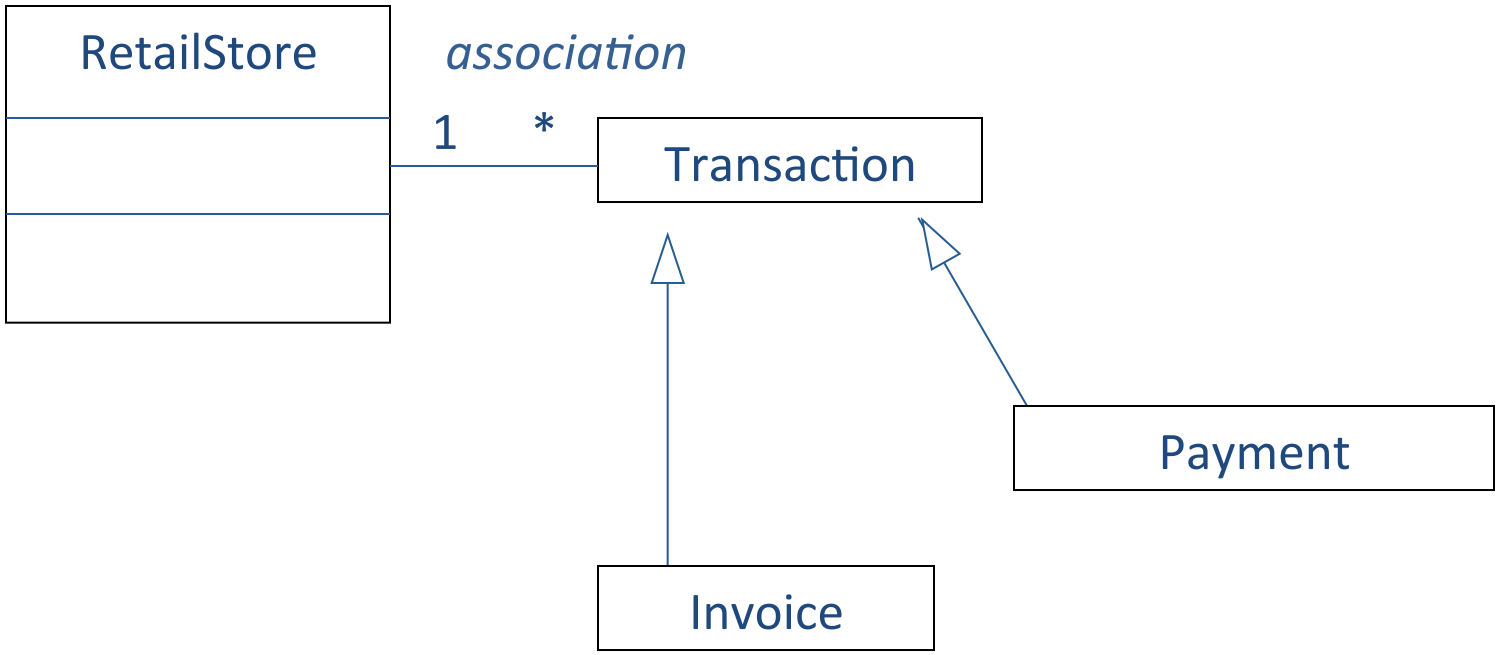
Rough Sketch: Wholesale System



Add some more detail to some of the classes.

Add a Reversal candidate class.

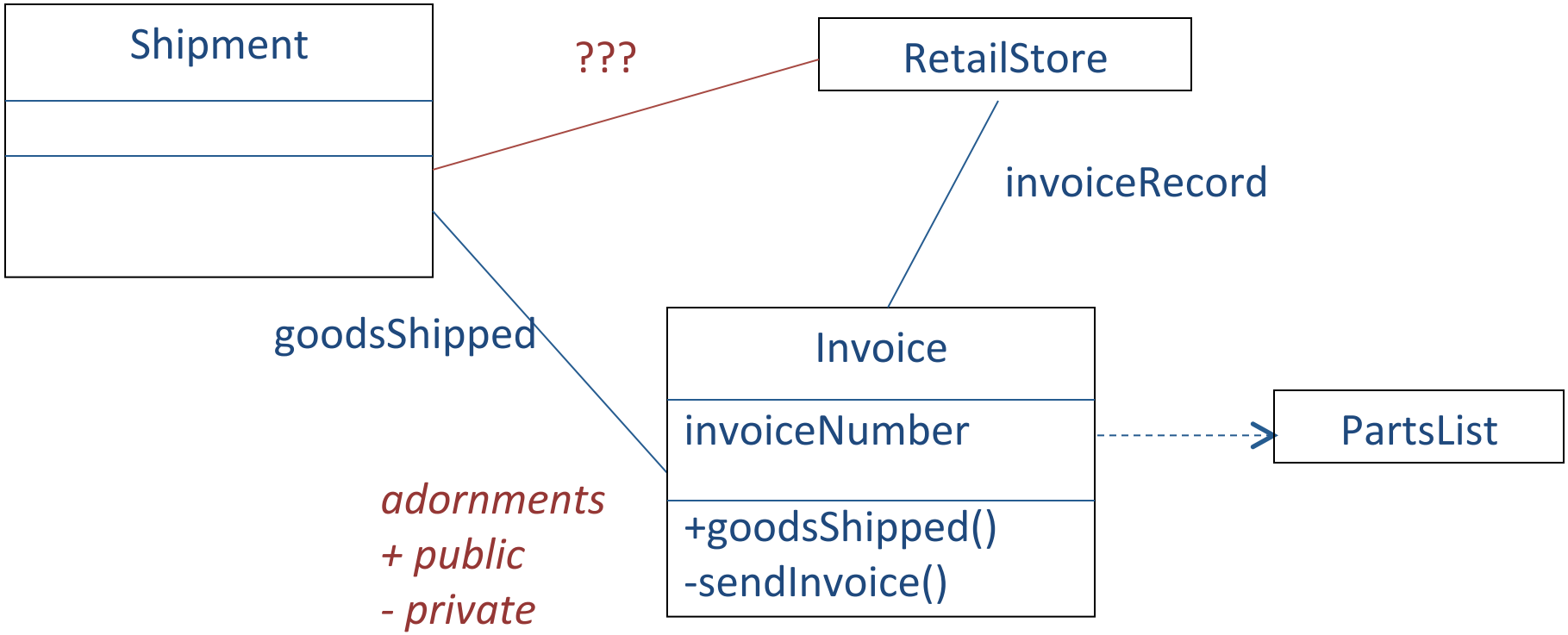
Rough Sketch: Wholesale System



Compare the candidate classes to a scenario that describes a common transaction type: an order from a retail store

Which class is responsible for the financial records for a store?

Rough Sketch: Wholesale System



Compare the candidate classes to a scenario that describes a different aspect of a common transaction type: a shipment to a retail store.

Lessons Learned

Design is empirical. There is no single correct design.

During the design process:

- Eliding: Elements are hidden to simplify the diagram
- Incomplete: During the early part of the design process, elements may be missing.
- Inconsistency: During the early part of the design process, the model may not be consistent

The diagram is not the whole design. Diagrams must be backed up with specifications.

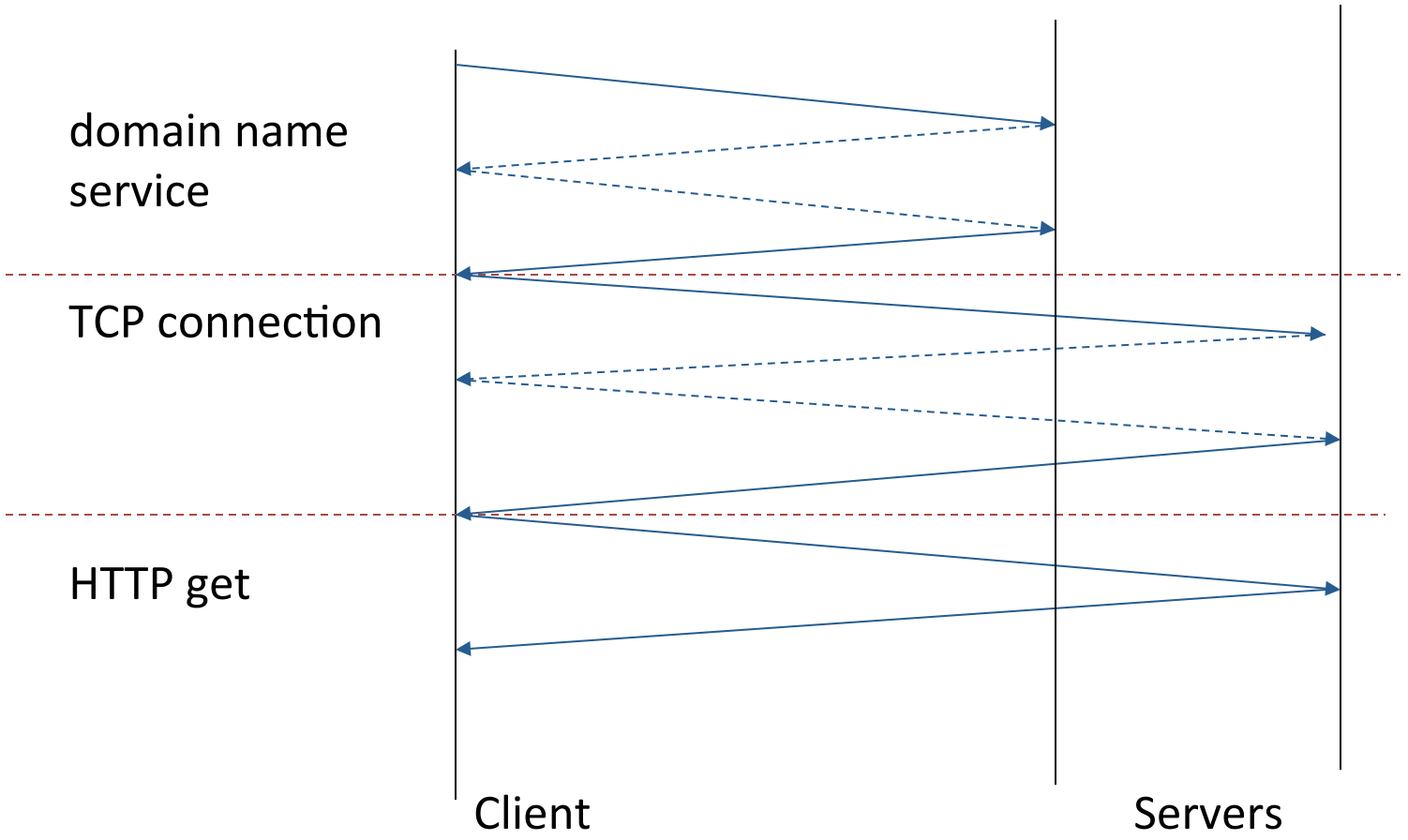
Modeling Dynamic Aspects of Systems

Interaction diagram: shows set of objects and their relationships including messages that may be dispatched among them

- Sequence diagrams: time ordering of messages

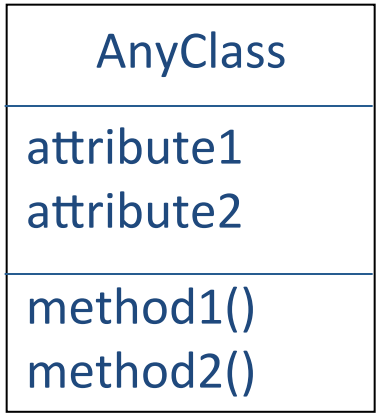
Interaction: Informal Bouncing Ball Diagrams

Example: execution of an HTTP get command,
e.g., http://www.cs.cornell.edu/



UML Notation for Classes and Objects

Classes



or

Objects

or

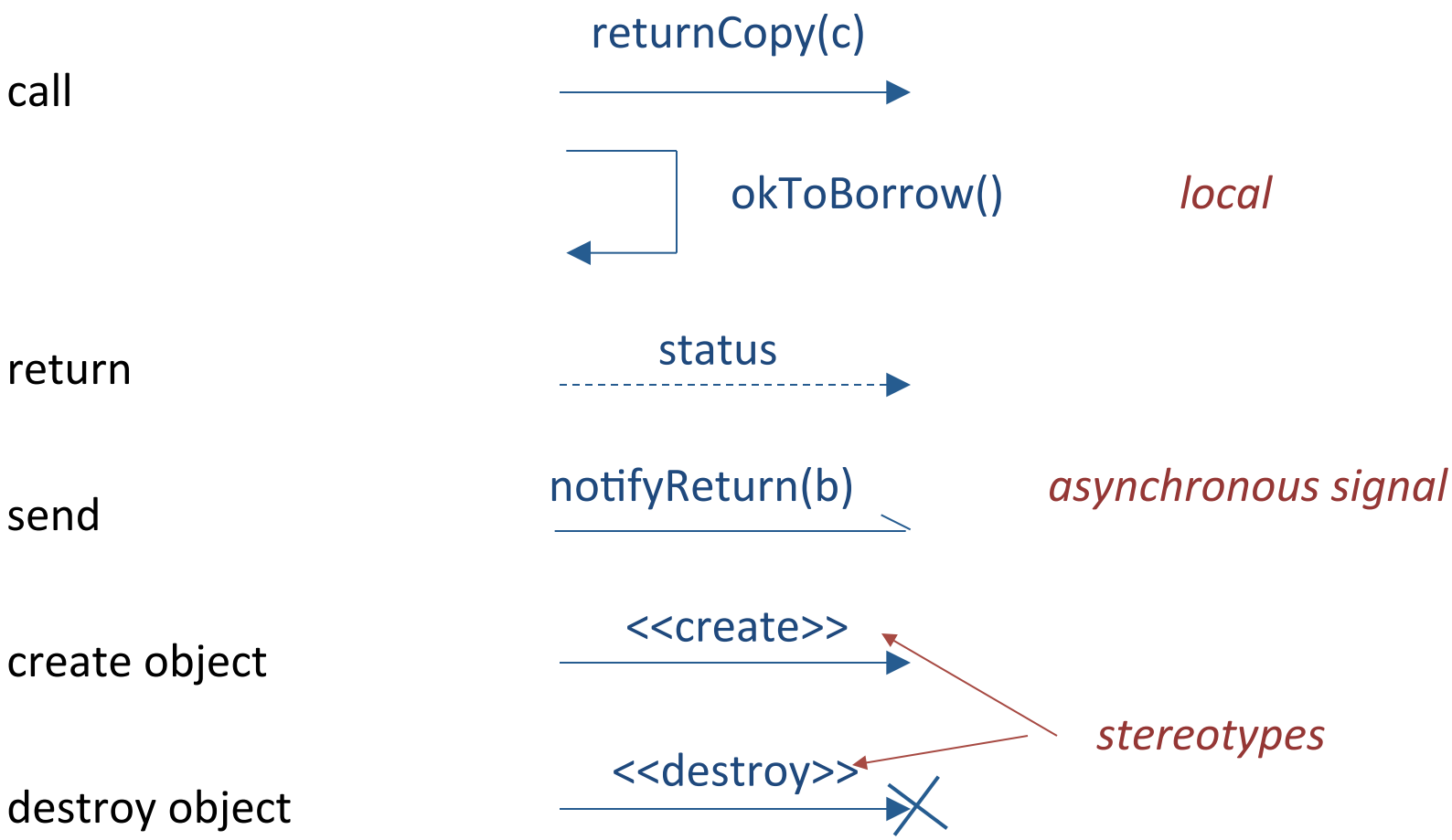
or

The names of objects are underlined.

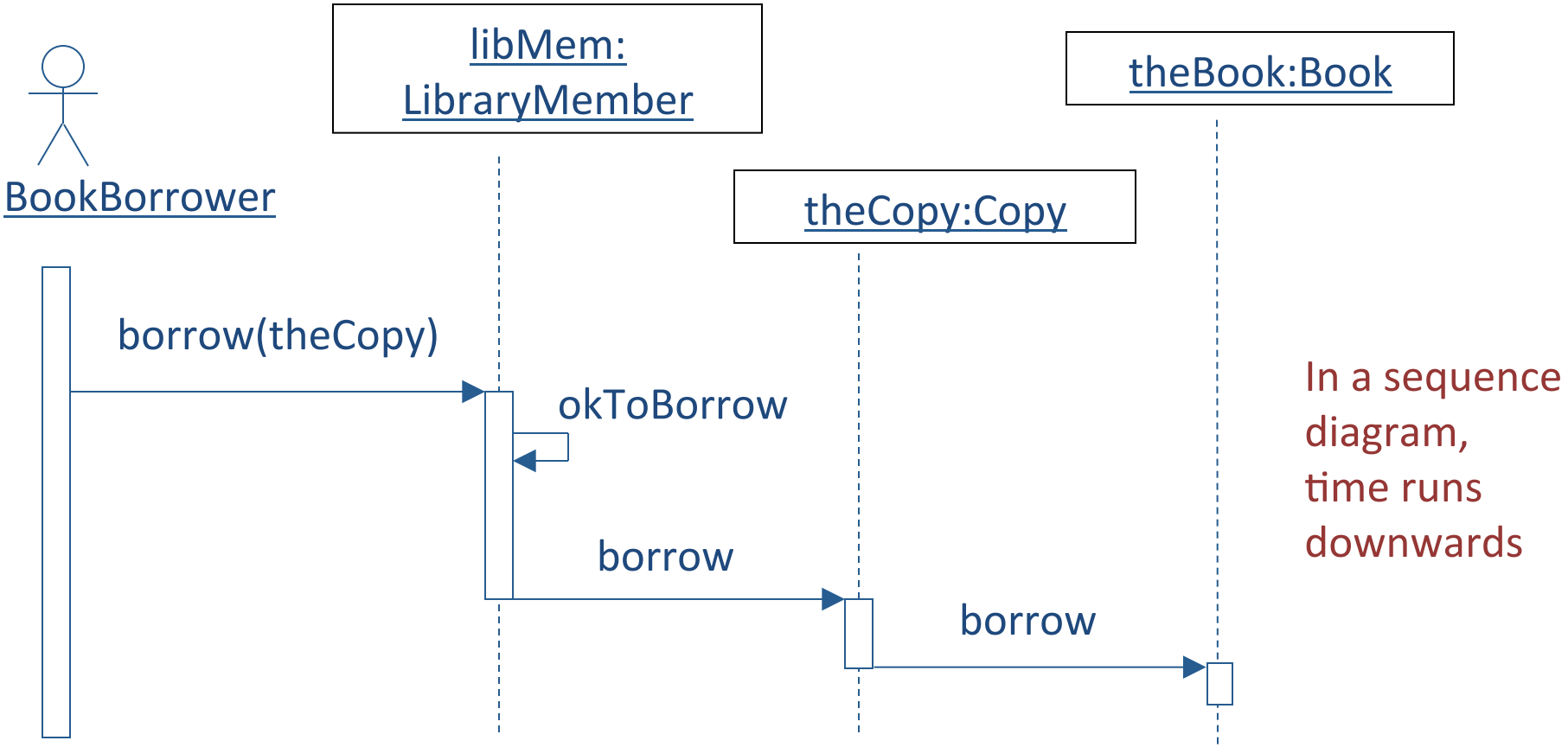
Notation: Interaction

An **interaction** is a behavior that comprises a set of **messages** exchanged among a set of objects within a particular context to accomplish a specific purpose.

Actions on Objects

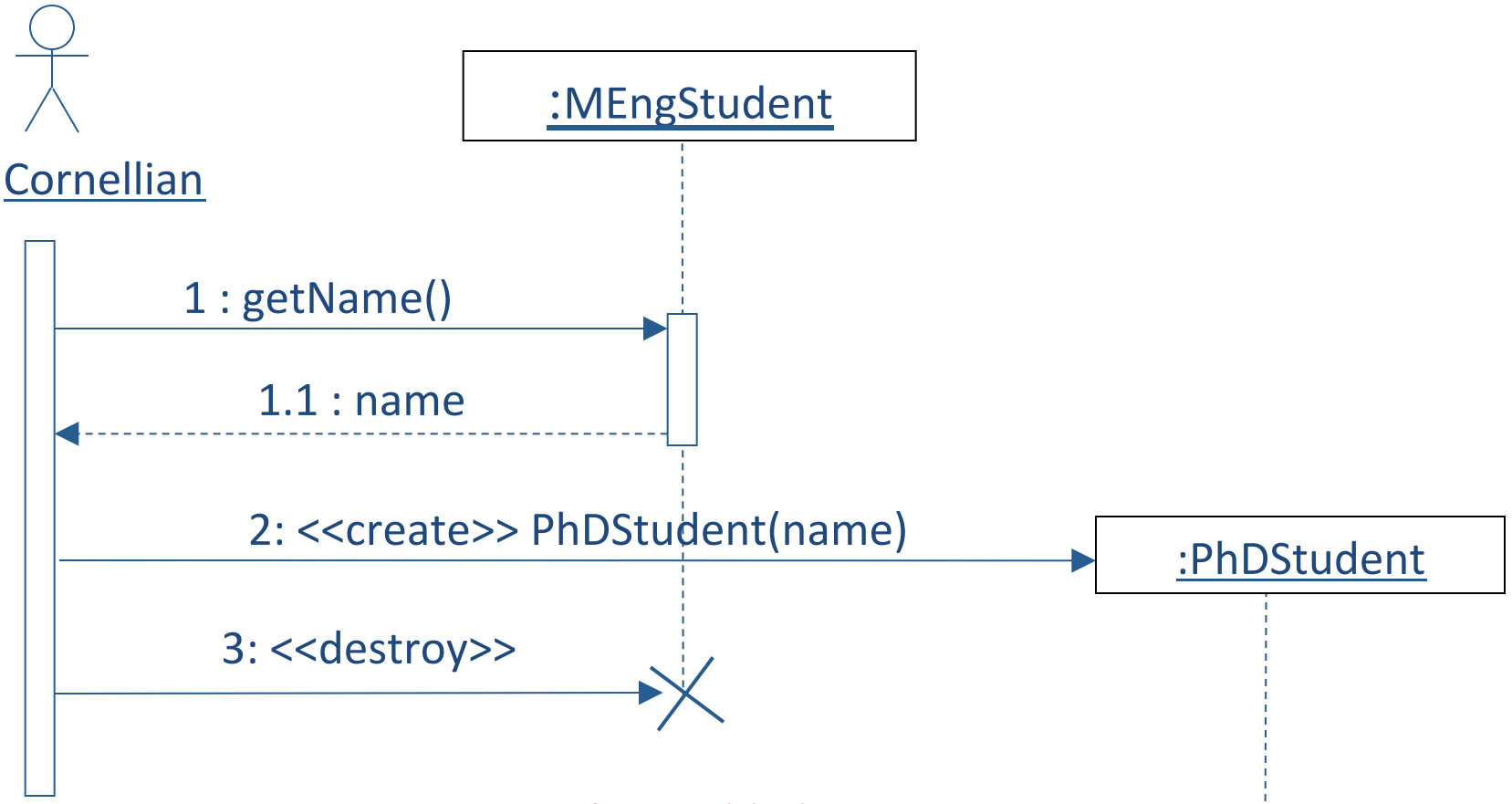


Sequence Diagram: Borrow Copy of a Book



In a sequence diagram, time runs downwards

Sequence Diagram: Change in Cornell Program



sequence numbers added to messages

Sequence Diagram: Painting Mechanism