Principles of Software Construction: Objects, Design, and Concurrency

Designing (sub-) systems

A formal design process

Josh Bloch **Charlie Garrod**

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 3 due Sunday, September 25th
- Midterm exam next Thursday (September 29th)
  - Review session Wednesday, September 28th, 7-9 pm, HH B103

Key concepts from Tuesday…

Iterator design pattern

- Problem: Clients need uniform strategy to access all elements in a container, independent of the container type
  - Order is unspecified, but access every element once
- Solution: A strategy pattern for iteration
- Consequences:
  - Hides internal implementation of underlying container
  - Easy to change container type
  - Facilitates communication between parts of the program

The decorator design pattern

- Problem: You need arbitrary or dynamically composable extensions to individual objects.
- Solution: Implement a common interface as the object you are extending, add functionality, but delegate primary responsibility to an underlying object.
- Consequences:
  - More flexible than static inheritance
  - Customizable, cohesive extensions
  - Breaks object identity, self-references

A generic implementation of pairs

```
public class Pair<E> {
  private final E first, second;
  public Pair(E first, E second) {
    this.first  = first;
    this.second = second;
  }
  public E first()  { return first;  }
  public E second() { return second; }
}
```

- Better client code:

```
Pair<String> p = new Pair<>("Hello", "world");
String result = p.first();
```

Some Java Generics details

- Can have multiple type parameters
  - e.g., `Map<String, Integer>`
- Wildcards
  - e.g. `List<?>` or `List<? extends Animal>` or `List<? super Animal>`
- Generics are type invariant
  - `ArrayList<String>` is a subtype of `List<String>`
  - `List<String>` is not a subtype of `List<Object>`
- Generic type info is erased (i.e. compile-time only)
  - Cannot use `instanceof` to check generic type
- Cannot create Generic arrays

```
Pair<String>[] foo = new Pair<String>[42]; // won't compile
```

Generic array creation is illegal

```
                                    // won't compile
List<String>[] stringLists = new List<String>[1];
List<Integer> intList = Arrays.asList(42);
Object[] objects = stringLists;
objects[0] = intList;
String s = stringLists[0].get(0); // Would be type-safe
```

Generic design advice: Prefer lists to arrays

```
// Fails at runtime
Object[] oArray = new Long[42];
oArray[0] = "I don't fit in"; // Throws ArrayStoreException

// Won't compile
List<Object> ol = new ArrayList<Long>(); // Incompatible type
ol.add("I don't fit in");
```

Wildcard types provide API flexibility

- `List<String>` is not a subtype of `List<Object>`
- `List<String>` is a subtype of `List<? extends Object>`
- `List<Object>` is a subtype of `List<? super String>`
- `List<Anything>` is a subtype of `List<?>`

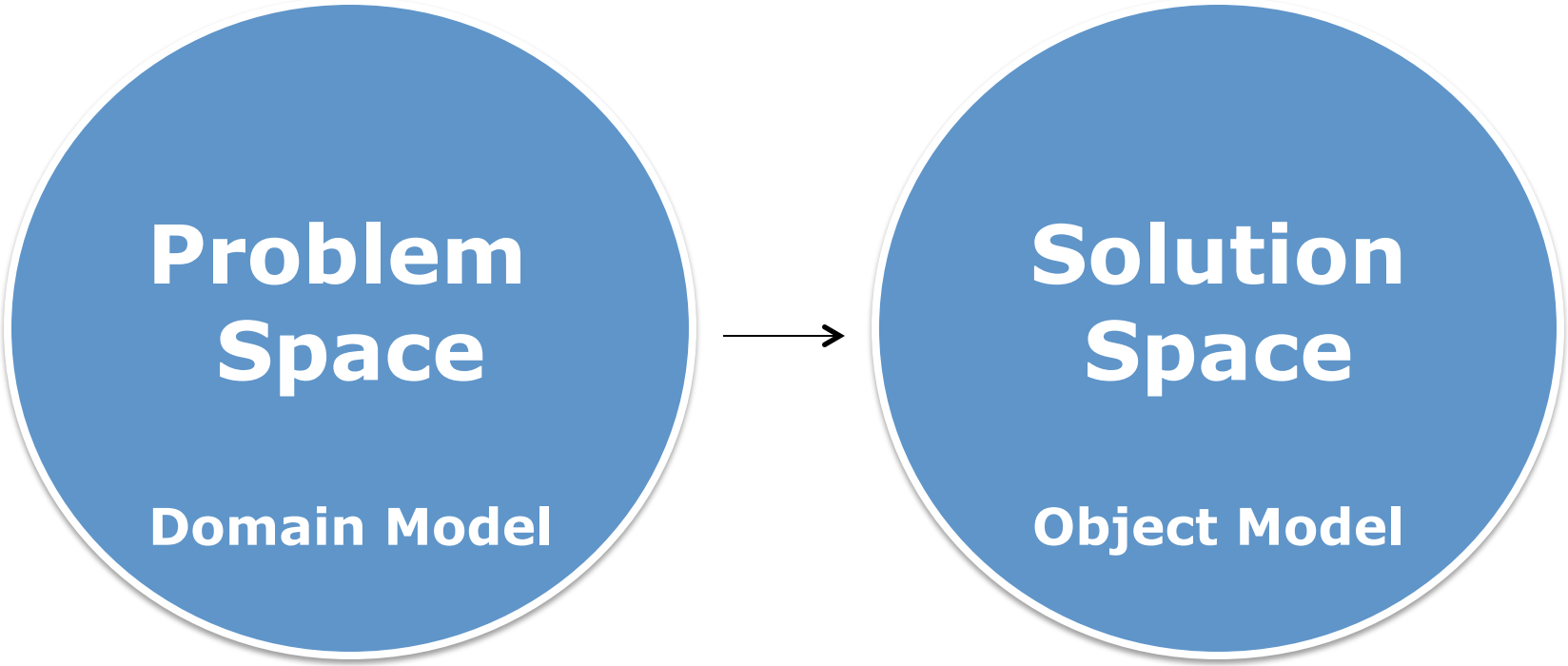
Wildcards in the `java.util.Collection` API

```
public interface Collection<E> … {
  boolean     add(E e);
  boolean     addAll(Collection<? extends E> c);
  boolean     remove(Object e);
  boolean     removeAll(Collection<?> c);
  boolean     retainAll(Collection<?> c);
  boolean     contains(Object e);
  boolean     containsAll(Collection<?> c);
  void        clear();
  int         size();
  boolean     isEmpty();
  Iterator<E> iterator();
  Object[]    toArray()
  <T> T[]     toArray(T[] a);
  …
}
```

Today: Tools, goals, and understanding the problem space

- Visualizing dynamic behavior with interaction diagrams
- Design goals and design principles
- Understanding a design problem: Object oriented analysis

Our path toward a more formal design process



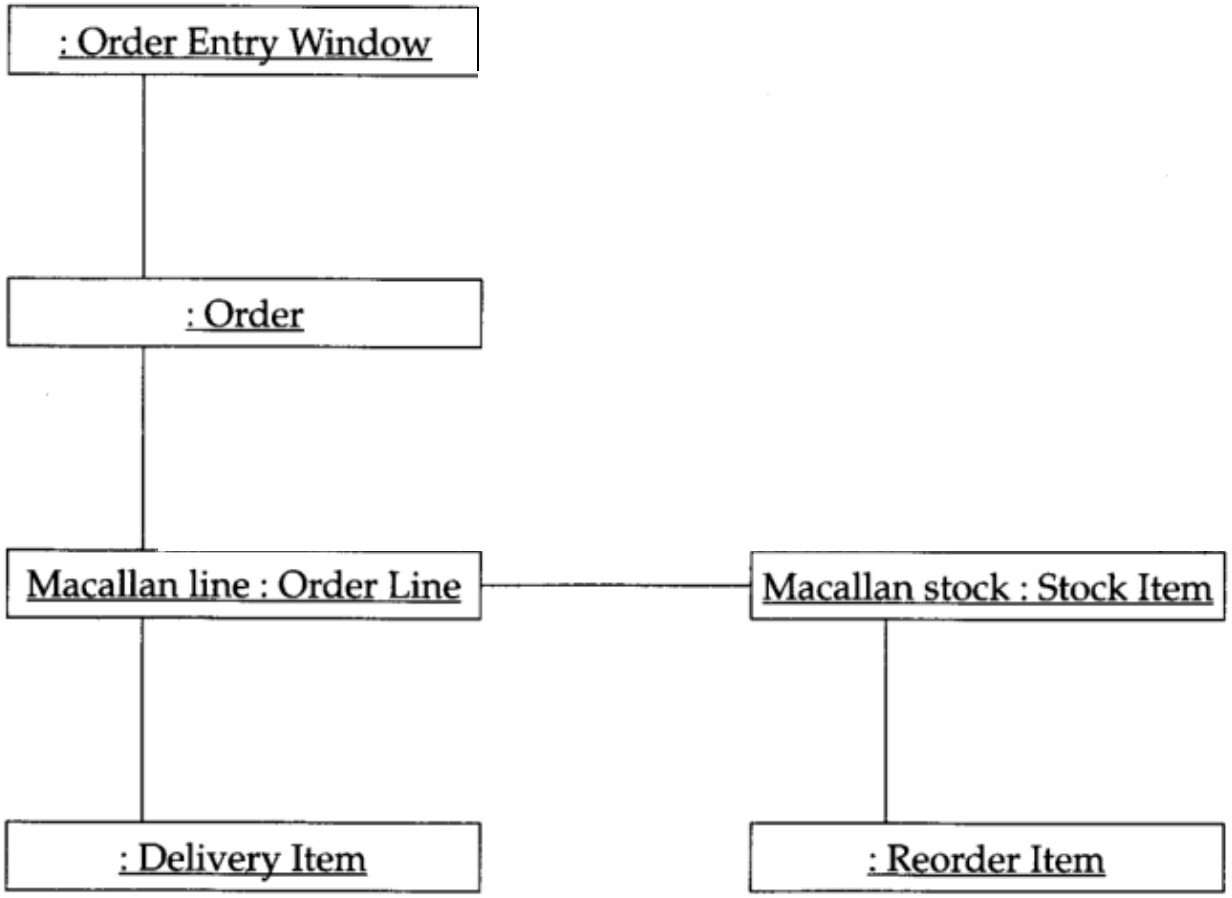
- Real-world concepts
- Requirements, concepts
- Relationships among concepts
- Solving a problem
- Building a vocabulary

- System implementation
- Classes, objects
- References among objects and inheritance hierarchies
- Computing a result
- Finding a solution

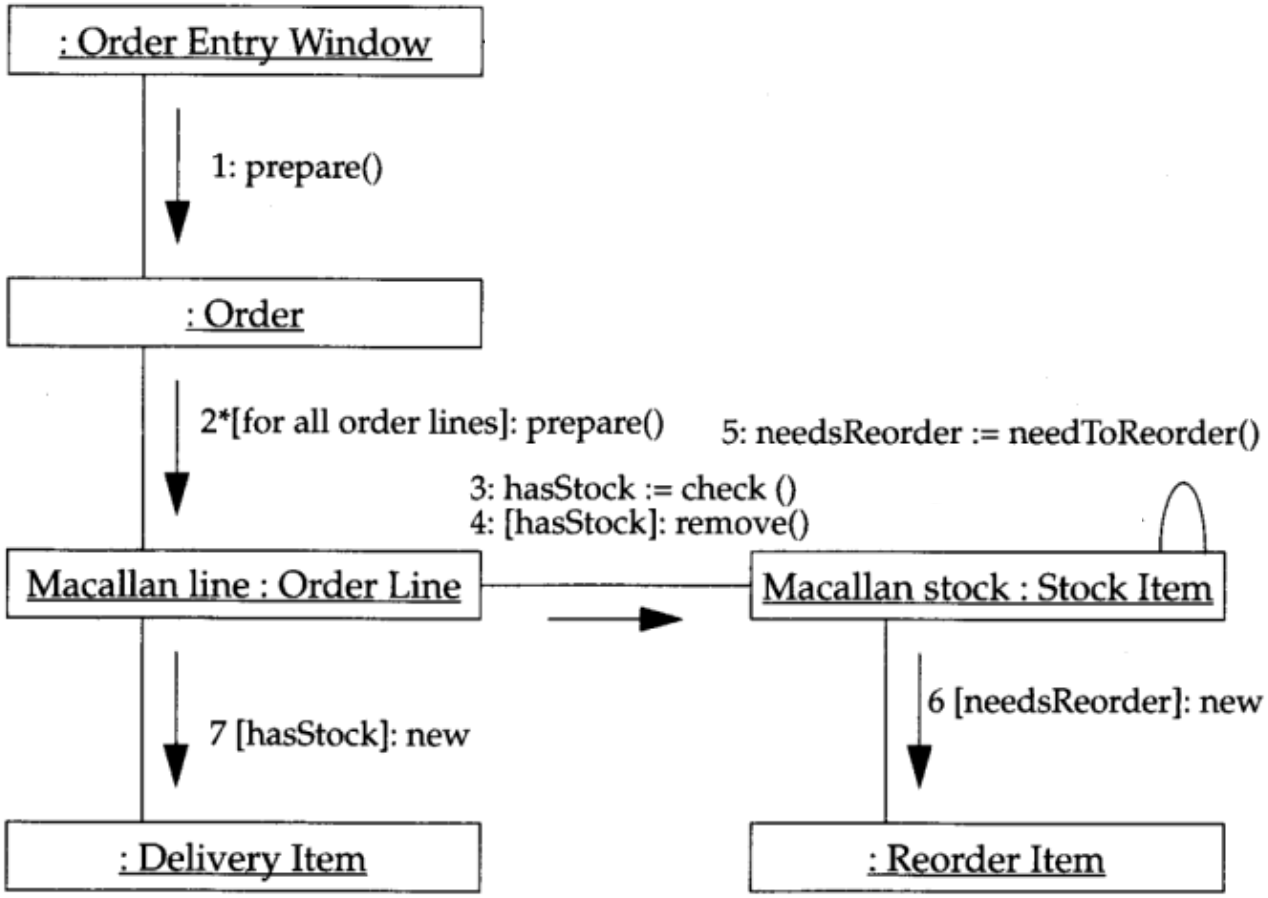
Visualizing dynamic behavior: Interaction diagrams

- An *interaction diagram* is a picture that shows, for a single scenario of use, the events that occur across the system's boundary or between subsystems
- Clarifies interactions:
  - Between the program and its environment
  - Between major parts of the program
- For this course, you should know:
  - Communication diagrams
  - Sequence diagrams

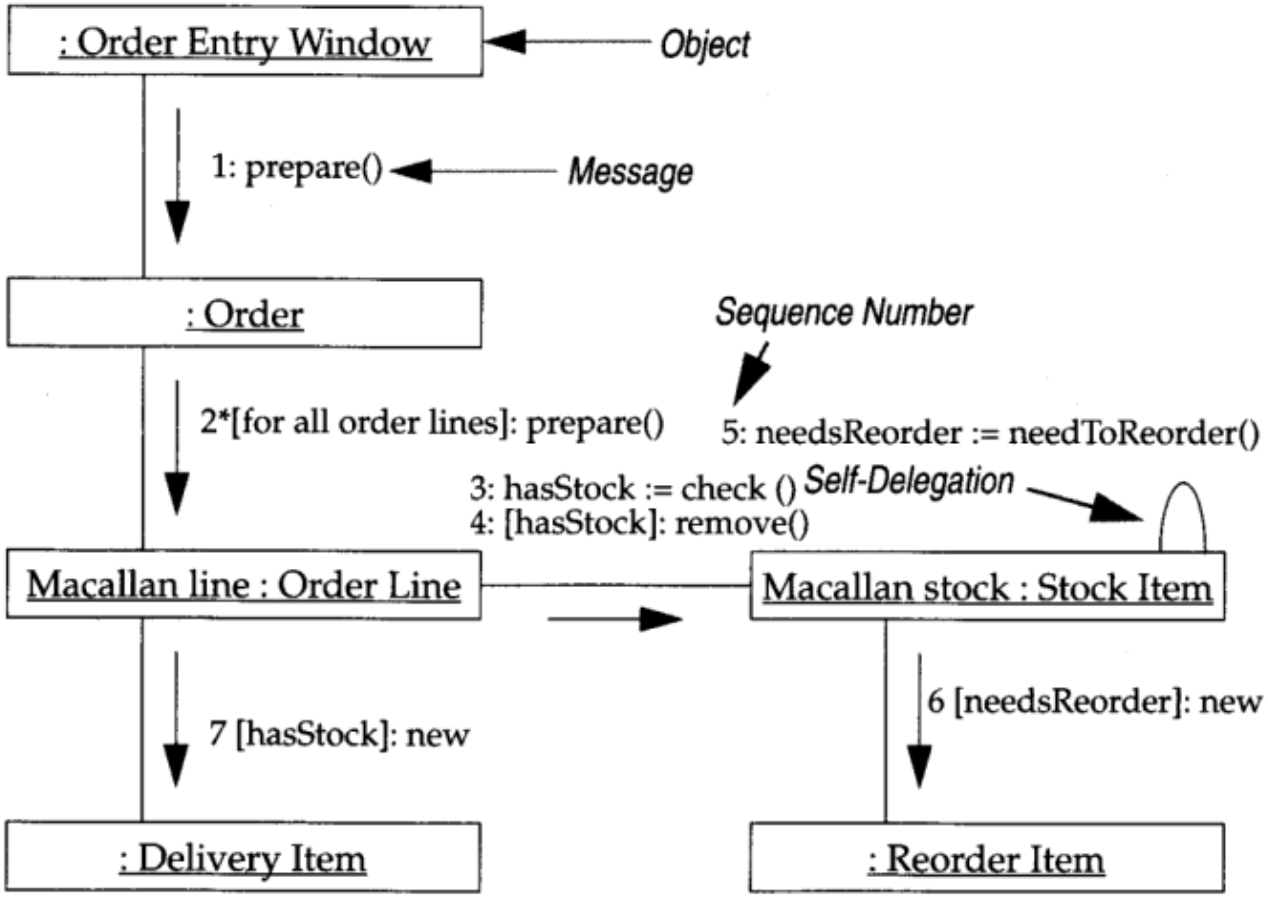
Creating a communication diagram



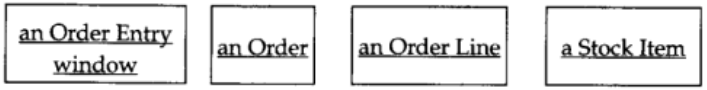
An example communication diagram



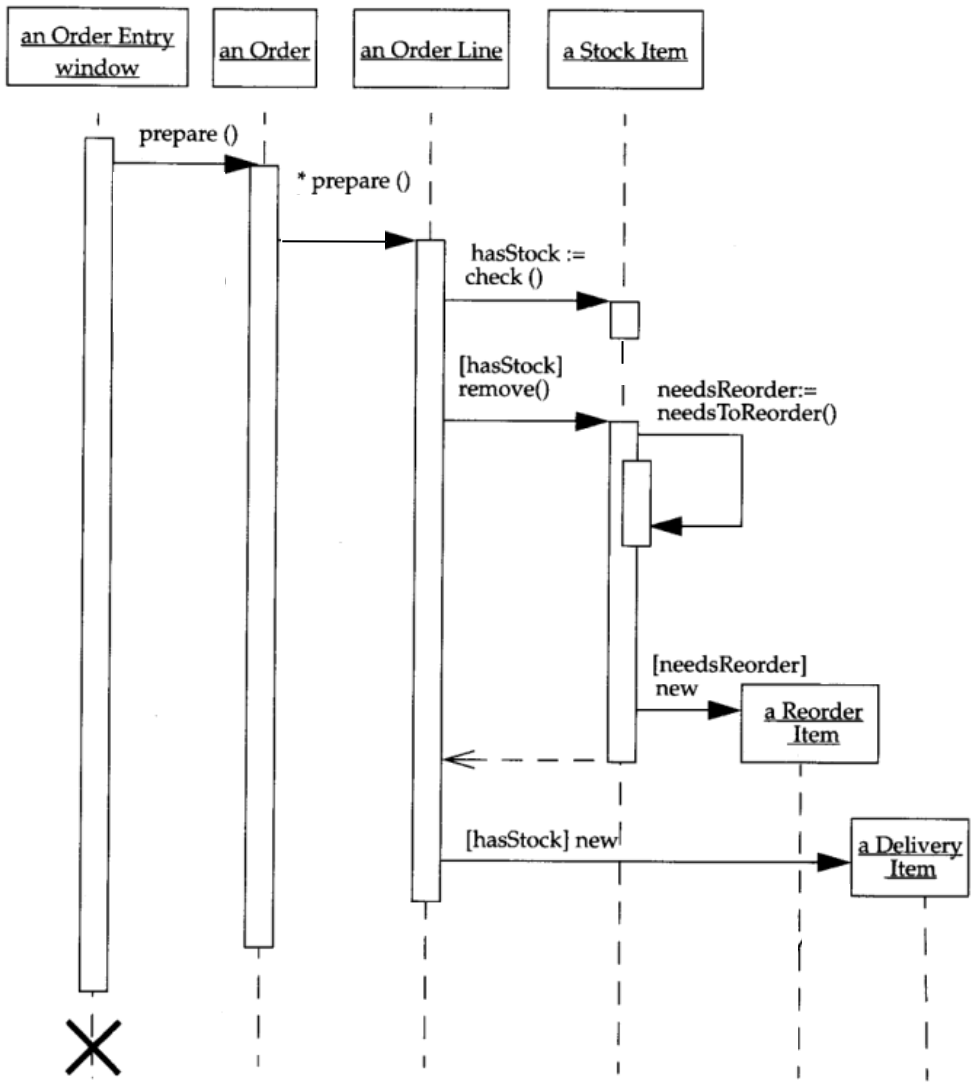
(Communication diagram with notation annotations)



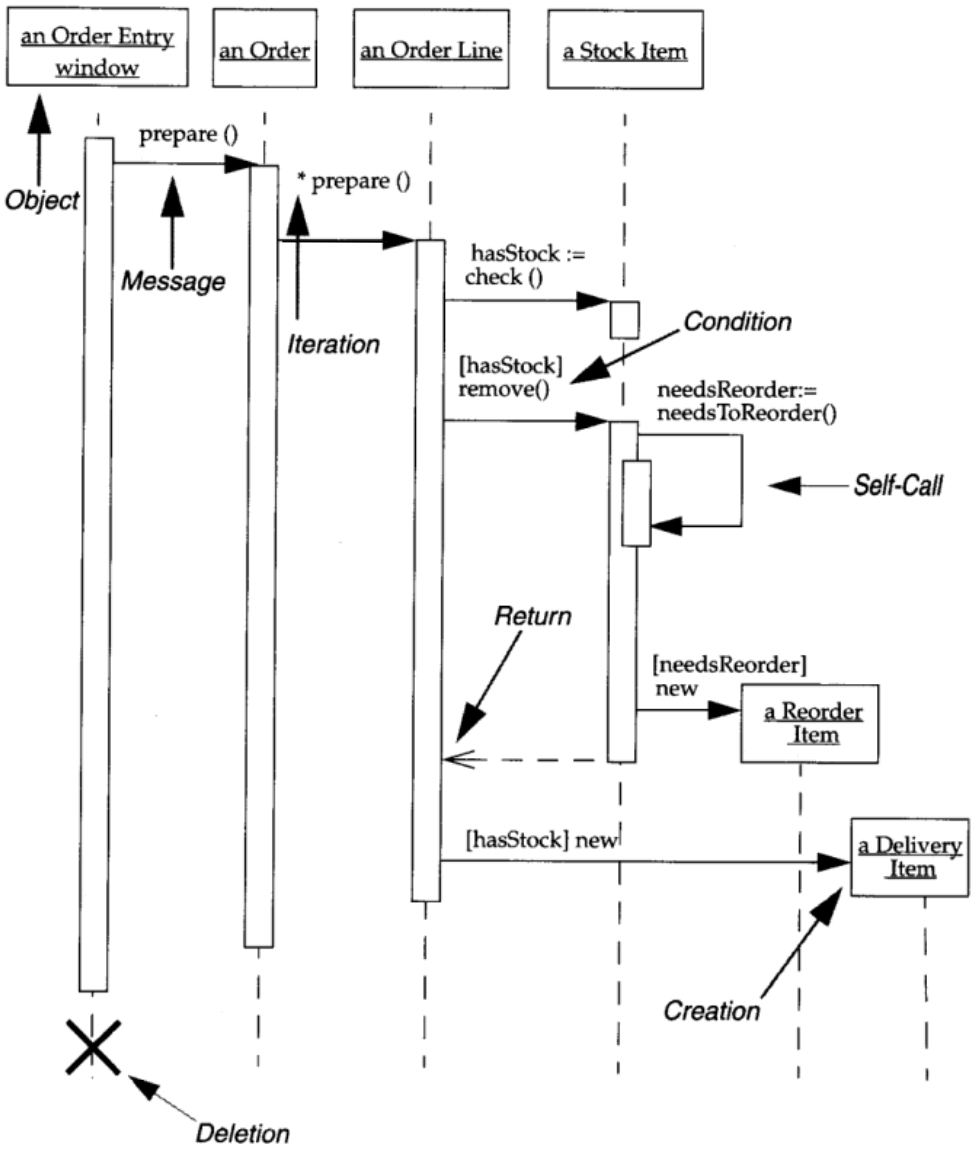
Constructing a sequence diagram



An example sequence diagram



(Sequence diagram with notation annotations)

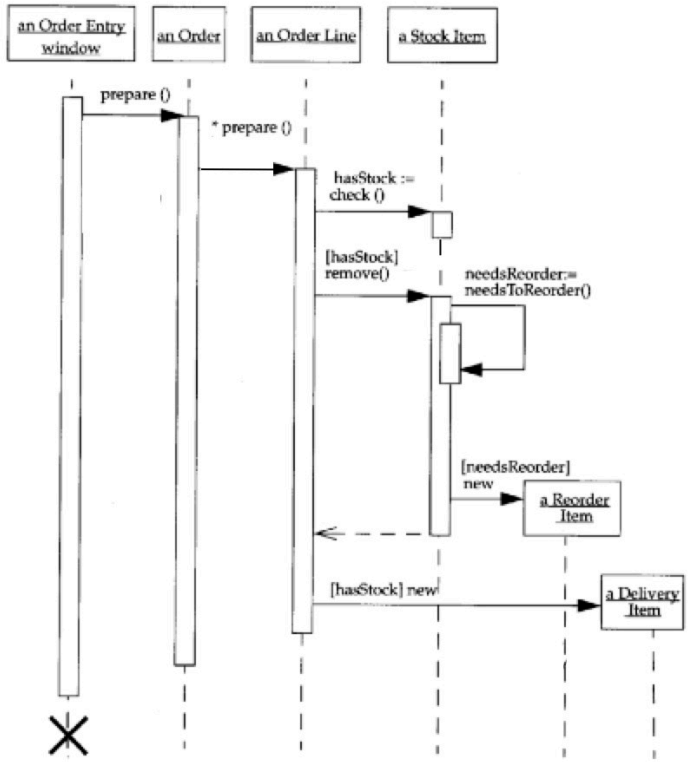


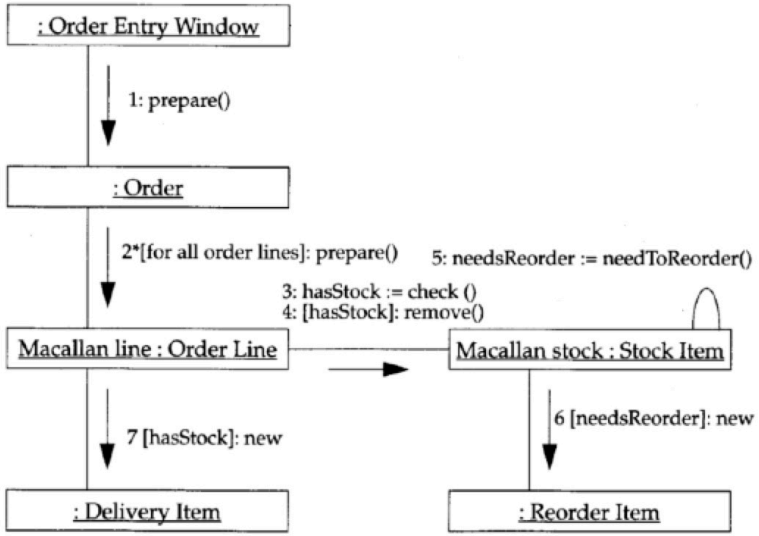
Draw a sequence diagram for a call to `LoggingList.add`:

```
public class LoggingList<E> implements List<E> {
  private final List<E> list;
  public LoggingList<E>(List<E> list) { this.list = list; }
  public boolean add(E e) {
      System.out.println("Adding " + e);
      return list.add(e);
  }
  public E remove(int index) {
      System.out.println("Removing at " + index);
      return list.remove(index);
  }
  …
```

Sequence vs. communication diagrams

- Relative advantages and disadvantages?





Today: Tools, goals, and understanding the problem space

- Visualizing dynamic behavior with interaction diagrams
- Design goals and design principles
- Understanding a design problem: Object oriented analysis

Metrics of software quality

Source: Braude, Bernstein, Software Engineering. Wiley 2011

- Sufficiency / functional correctness
  - Fails to implement the specifications … Satisfies all of the specifications
- Robustness
  - Will crash on any anomalous event … Recovers from all anomalous events
- Flexibility
  - Must be replaced entirely if spec changes … Easily adaptable to changes
- Reusability
  - Cannot be used in another application … Usable without modification
- Efficiency
  - Fails to satisfy speed or storage requirement … satisfies requirements
- Scalability
  - Cannot be used as the basis of a larger version … is an outstanding basis…
- Security
  - Security not accounted for at all … No manner of breaching security is known

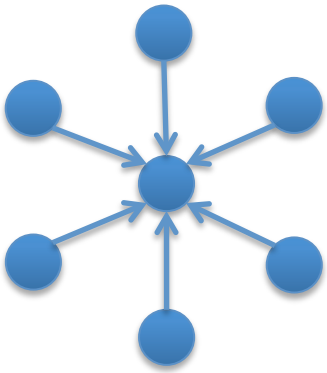
Design challenges/goals

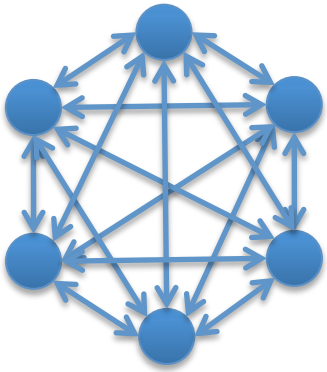
Design principles

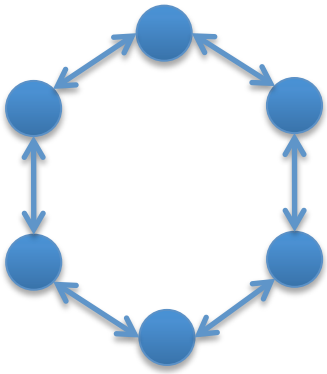
- Low coupling
- Low representational gap
- High cohesion

A design principle for reuse: *low coupling*

- Each component should depend on as few other components as possible



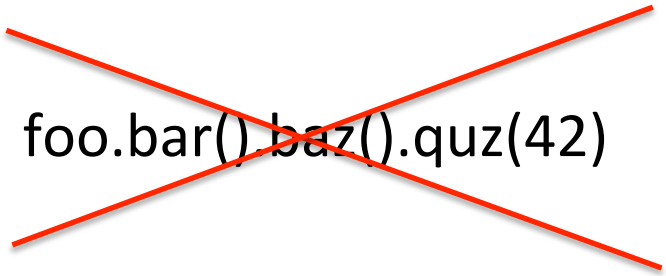




- Benefits of low coupling:
  - Enhances understandability
  - Reduces cost of change
  - Eases reuse

Law of Demeter

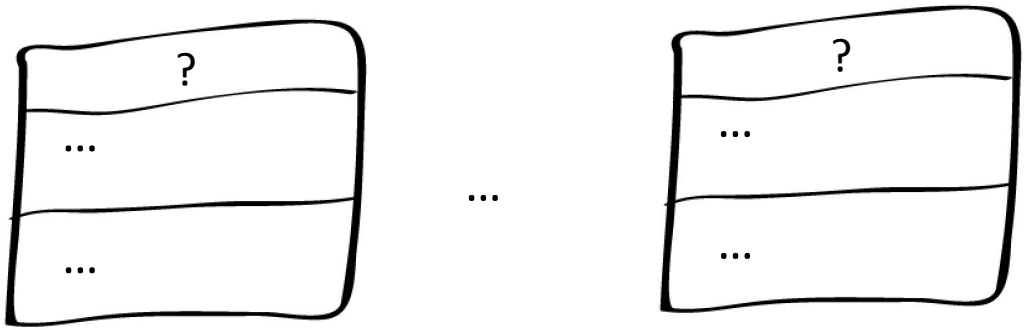
- "Only talk to your immediate friends"



Representational gap

- Real-world concepts:

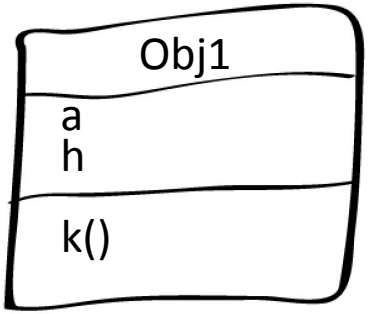
- Software concepts:

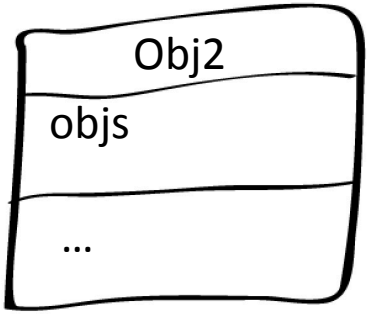


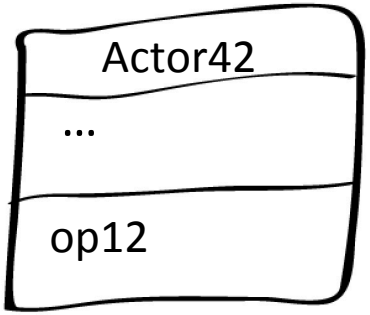
Representational gap

- Real-world concepts:

- Software concepts:



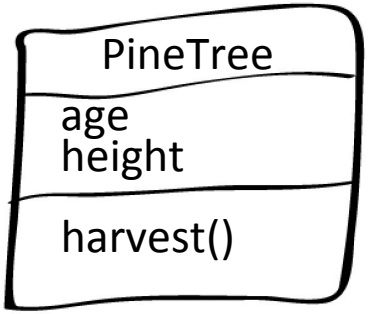


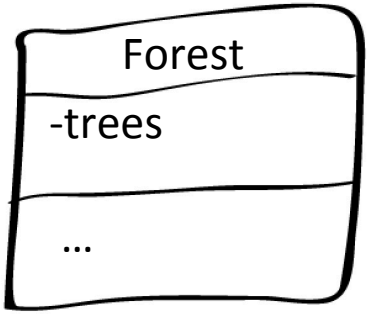


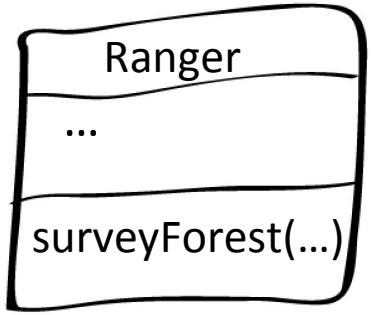
Representational gap

- Real-world concepts:

- Software concepts:





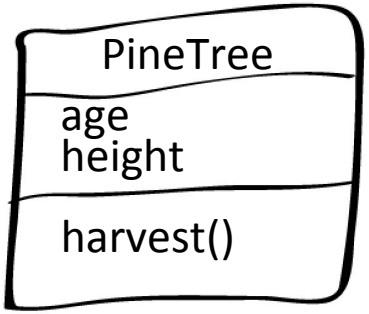


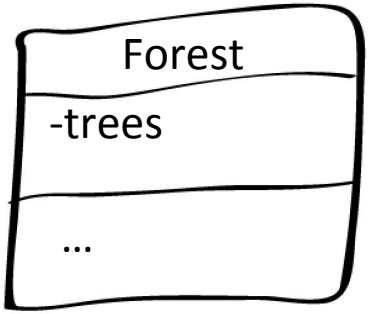
Benefits of low representational gap

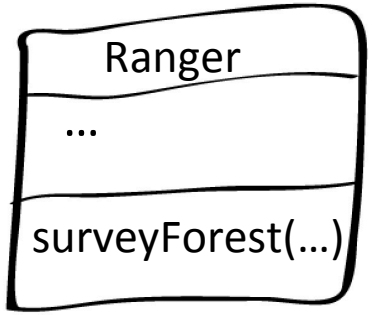
- Facilitates understanding of design and implementation
- Facilitates traceability from problem to solution
- Facilitates evolution

A related design principle: high cohesion

- Each component should have a small set of closely-related responsibilities
- Benefits:
  - Facilitates understandability
  - Facilitates reuse
  - Eases maintenance







Coupling vs. cohesion

- All code in one component?
  - Low cohesion, low coupling
- Every statement / method in a separate component?
  - High cohesion, high coupling

Today: Tools, goals, and understanding the problem space

- Visualizing dynamic behavior with interaction diagrams
- Design goals and design principles
- Understanding a design problem: Object oriented analysis

A high-level software design process

- Project inception
- Gather requirements
- Define actors, and use cases

15-313

- Model / diagram the problem, define objects
- Define system behaviors
- Assign object responsibilities
- Define object interactions
- Model / diagram a potential solution
- Implement and test the solution

15-214

- Maintenance, evolution, …

…

Artifacts of this design process

- Model / diagram the problem, define objects
  - Domain model (a.k.a. conceptual model)
- Define system behaviors
  - System sequence diagram
  - System behavioral contracts
- Assign object responsibilities, define interactions
  - Object interaction diagrams
- Model / diagram a potential solution
  - Object model

Artifacts of this design process

- Model / diagram the problem, define objects
  - Domain model (a.k.a. conceptual model)
- Define system behaviors
  - System sequence diagram
  - System behavioral contracts
- Assign object responsibilities, define interactions
  - Object interaction diagrams
- Model / diagram a potential solution
  - Object model

Today: understanding the problem

Defining a solution

Input to the design process: Requirements and use cases

- Typically prose:

A public library typically stores a collection of books, movies, or other library items available to be borrowed by people living in a community. Each library member typically has a library account and a library card with the account's ID number, which she can use to identify herself to the library. A member's library account records which items the member has borrowed and the due date for each borrowed item. Each type of item has a default re

is borrowed

member mu

member's li

Use case scenario: A library member should be able to use her library card to log in at a library system kiosk and borrow a book. After confirming that the member has no unpaid late fees, the library system should determine the book's due date by adding its rental period to the current day, and record the book and its due date as a borrowed item in the member's library account.

Modeling a problem domain

- Identify key concepts of the domain description
  - Identify nouns, verbs, and relationships between concepts
  - Avoid non-specific vocabulary, e.g. "system"
  - Distinguish operations and concepts
  - Brainstorm with a domain expert

Modeling a problem domain

- Identify key concepts of the domain description
  - Identify nouns, verbs, and relationships between concepts
  - Avoid non-specific vocabulary, e.g. "system"
  - Distinguish operations and concepts
  - Brainstorm with a domain expert
- Visualize as a UML class diagram, a *domain model*
  - Show class and attribute concepts
    - Real-world concepts only
    - No operations/methods
    - Distinguish class concepts from attribute concepts
  - Show relationships and cardinalities

Building a domain model for a library system

A public library typically stores a collection of books, movies, or other library items available to be borrowed by people living in a community. Each library member typically has a library account and a library card with the account's ID number, which she can use to identify herself to the library.

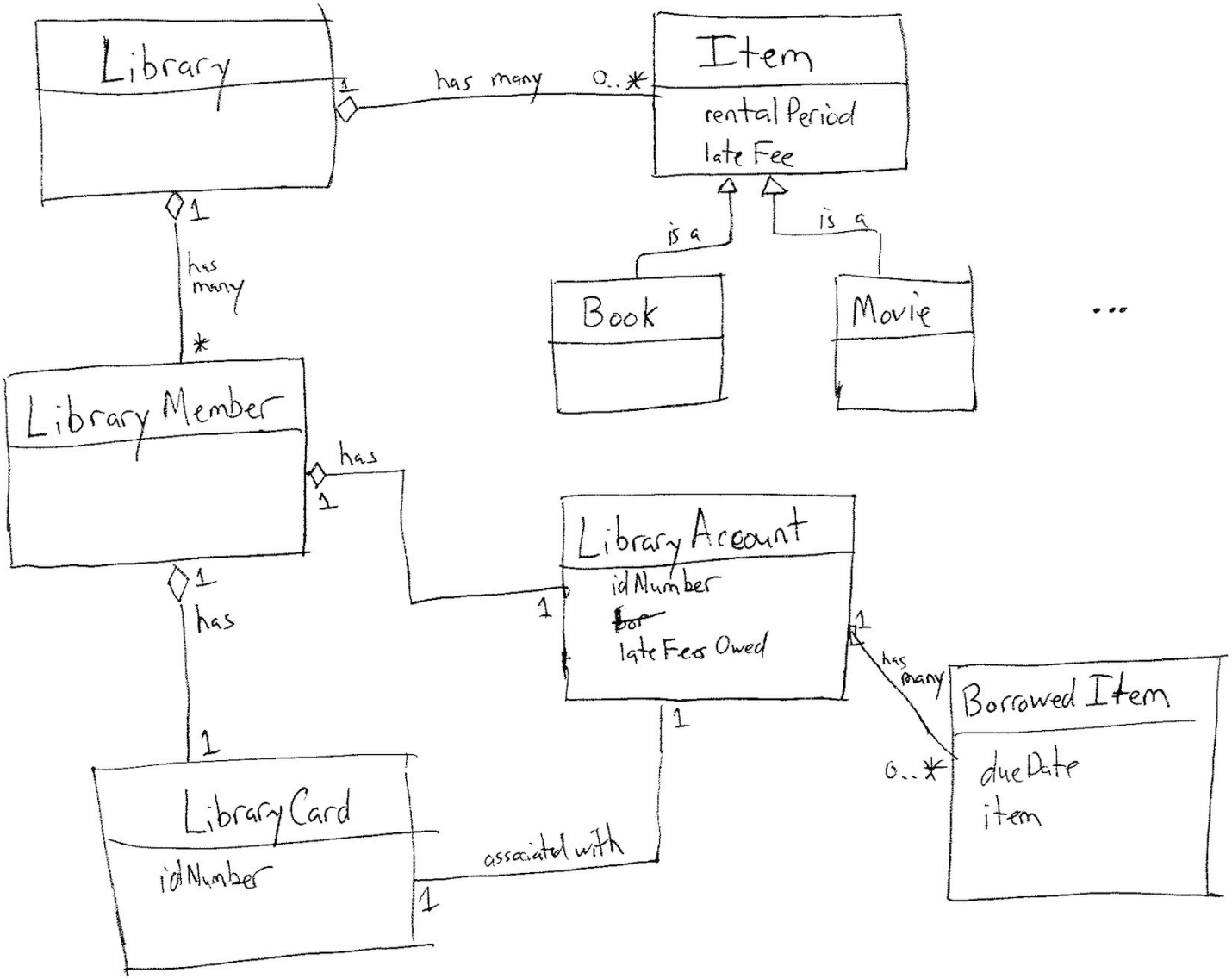
A member's library account records which items the member has borrowed and the due date for each borrowed item. Each type of item has a default rental period, which determines the item's due date when the item is borrowed. If a member returns an item after the item's due date, the member owes a late fee specific for that item, an amount of money recorded in the member's library account.

Building a domain model for a library system

A public library typically stores a collection of books, movies, or other library items available to be borrowed by people living in a community. Each library member typically has a library account and a library card with the account's ID number, which she can use to identify herself to the library.

A member's library account records which items the member has borrowed and the due date for each borrowed item. Each type of item has a default rental period, which determines the item's due date when the item is borrowed. If a member returns an item after the item's due date, the member owes a late fee specific for that item, an amount of money recorded in the member's library account.

One domain model for the library system



Notes on the library domain model

- All concepts are accessible to a non-programmer
- The UML is somewhat informal
  - Relationships are often described with words
- Real-world "is-a" relationships are appropriate for a domain model
- Real-word abstractions are appropriate for a domain model
- Iteration is important
  - This example is a first draft. Some terms (e.g. Item vs. LibraryItem, Account vs. LibraryAccount) would likely be revised in a real design.
- Aggregate types are usually modeled as classes
- Primitive types (numbers, strings) are usually modeled as attributes