Design Patterns 1

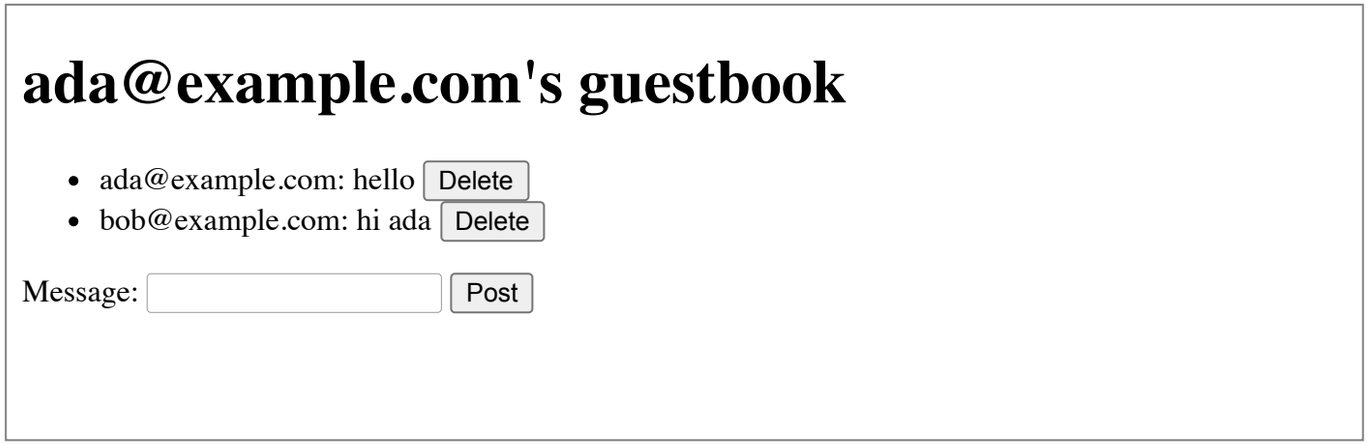
CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

- Reading 4: due Friday, November 20, 11:59pm
- Project proposal: due Tuesday, November 24, 11:59pm

A guestbook for each user



1. Each user who has signed in has a book: this is ada's, as ada sees it
2. Any user who has signed in may read any book and sign it: bob did
3. Only a book's owner may delete a message from it

A delete has two people in it

- **The asker**: the user who sent the request
- **The owner**: whose book it is
  - The request names the book, so its sender chooses
- **Bob asks to delete a message from ada's book. What should happen?**
- The asker, bob, is not the owner, ada: bob is refused

`delete` decides (block A)

```
308   public void delete(String asker, int index)
309           throws NotOwnerException {
310       if (!asker.equals(owner)) {
311           throw new NotOwnerException(
312                   asker + " does not own this book");
313       }
314       messages.remove(index);
```

- Line 310 compares the asker with the book's owner
- **What does `delete` need from its caller?** Who is asking
- **What will `javac` hold the caller to?** A `String`: any will do

Will `javac` accept this call? (block A)

```
124   try {
125       book.delete(owner, index);
126   } catch (NotOwnerException e) {
```

```
308   public void delete(String asker, int index)
```

- The call. Line 125's `owner`: the book the request named
- Is the call right?

javac accepts it, and bob deletes

```
125   book.delete(owner, index);
```

```
310   if (!asker.equals(owner)) {
```

- Bob asks to delete from ada's book: the handler's `owner` is ada
- Line 125 passes it along, so `delete`'s `asker` is ada, not bob
- Line 310 compares ada with the book's owner, ada: bob deletes
- **Testing**: a test with two users catches this handler
  - but not a handler written next year

User, a type of its own

```
209  class User {
210      private final String login;
211      // RI: login is not null, and not empty.
212      // AF(this) = the user whose login is login.
213
214      private User(String login) {
215          this.login = login;
216      }
```

- A private field: only code inside the class can touch it
- A private constructor: only code inside `User` can make a `User`

Will javac accept line 125 now? (block C)

```
125   book.delete(owner, index);
```

```
308   public void delete(User asker, int index)
309           throws NotOwnerException {
```

```
$ javac Guestbook.java
Guestbook.java:125: error: incompatible types:
    String cannot be converted to User
                book.delete(owner, index);
                            ^
⋮
1 error
```

The quick fix (block D)

- It is late, and the author wants line 125 to compile:

```
125   book.delete(new User(owner), index);
```

- **Now the argument is a `User`. Will `javac` accept it?**

```
$ javac Guestbook.java
Guestbook.java:125: error: User(String) has private access in User
                book.delete(new User(owner), index);
                            ^
1 error
```

- Outside `User`, no code can turn a `String` into a `User`

So how does anyone get a User?

```
223   public static Optional<User> from(HttpExchange exchange) {
224       String login = exchange.getRequestHeaders()
225           .getFirst("Tailscale-User-Login");
226       if (login == null || login.isEmpty()) {
227           return Optional.empty();
228       }
229       return Optional.of(new User(login));
230   }
```

- `from` is static, so it is called on the class: `User.from(exchange)`
- Line 229 makes a `User`

The call, done right (screen only)

```
61   Optional<User> signedIn = User.from(exchange);
67   User asker = signedIn.get();
```

```
125   book.delete(asker, index);
```

- The handler asks `User.from` for the user who sent this request
- That `User` is what it hands to `delete`, and `javac` accepts it

Half sheet, question 1 (3 minutes)

To be sure that every `User` in this program came from a request, which lines do you have to read? Why are those enough?

- Look at block B of the sheet

Every `User` came from a request

- Read `User`, lines 209 to 250, and nothing outside it
- Line 214's constructor is private: only code inside `User` can call it
- The class has one `new User`: line 229, inside `from`
- `from` makes that `User` from the request's login
- From then on, `javac` checks that every `delete` is handed a `User`
- It does not check which request, or what `delete` does with it

Know that it works

Your job is to deliver code that you know works.

- **Reasoning**: we read one class, once
- **A tool**: `javac` checks every call, at every compile
- That is one thing a design pattern can do for us

That was a design pattern

Definition: design pattern

A **design pattern** is a solution to a design problem that comes up again and again, written down so that others can reuse it.
It has four parts: a name, the problem it solves, the solution, and the consequences of using it, both what it gains and what it costs.

- `User.from` is a **static factory method**

The four parts: Gamma, Helm, Johnson, and Vlissides, Design Patterns (1995), pages 2 and 3.

The help queue, again

```
 8  public interface HelpQueue {
10      int size();
18      void join(String netid);
26      String next();
29      List<String> waiting();
30  }
```

- Students join at the back; a TA helps the student at the front
- Four classes implement it:
  - `ListQueue`, `LinkedQueue`, `FastQueue`, `FasterQueue`

On the screen only.

A constructor has no name of its own

```
21  public FastQueue() {
22      queue = new ArrayList<String>();
23      members = new HashSet<String>();
24      checkRep();
25  }
26  /** @requires no NetID is in netids twice
27   *  @effects makes a queue of the students given, front first */
28  public FastQueue(List<String> netids) {
```

- `new FastQueue(netids)` does not say "no NetID twice". A name could
- Both are named `FastQueue`: only their arguments tell them apart

On the screen only.

What can new do here? (screen only)

```
6   HelpQueue queue = new FastQueue(List.of("ada", "ada"));
```

```
$ java BadList
Exception in thread "main" java.lang.IllegalStateException:
    someone is in queue twice
 at FastQueue.checkRep(FastQueue.java:41)
 at FastQueue.<init>(FastQueue.java:33)
 at BadList.main(BadList.java:6)
```

- A constructor returns a new object of its own class, or throws
- `from` returned an empty `Optional` for no login: no constructor can

The pattern: static factory method

- **Name**: static factory method. `User.from` is one
- **Problem**: a constructor has no name, and always makes a new object
- **Solution**: a static method makes the object and returns it
- **Gains**: a name; need not make a new object; may return a subtype
- **Costs**: harder to find in the documentation; may rule out subclasses

The name, problem, gains, and costs: Bloch, Effective Java, 3rd edition (2018), Item 1.

A client names its class

```
4   ListQueue queue = new ListQueue();
5   // Students join at the back.
6   queue.join("ada");
7   queue.join("bob");
```

- `OfficeHours` calls only `HelpQueue`'s methods
- But line 4 names `ListQueue`, as each client that makes a queue does
- To change to `FastQueue`, we edit each of those lines

Lecture 10's `OfficeHours.java`, before today. Not on the sheet.

One place names the class

```
2  public class HelpQueues {
3      /** @return a new help queue, with no students waiting */
4      public static HelpQueue empty() {
5          return new ListQueue();
5          return new LinkedQueue();
6      }
7  }
```

```
4  HelpQueue queue = HelpQueues.empty();
```

- A static factory method again: a client asks `HelpQueues.empty()`
- To change the class, edit line 5: `javac` checks nothing new here

Half sheet, question 2 (3 minutes)

Suppose line 5 of `HelpQueues.java` changes to `return new LinkedQueue()`.
Without rereading `OfficeHours.java`,
what can you be sure of about `OfficeHours.java`? What can you not be sure of?

- Look at block E of the sheet

What `javac` can tell us, and what it cannot

- **Sure**: `OfficeHours` still compiles
  - `javac` checks that `LinkedQueue` is a `HelpQueue`
  - and `OfficeHours` calls only `HelpQueue`'s methods
- **Not sure**: that `OfficeHours` behaves the same
  - that needs `LinkedQueue` to satisfy `HelpQueue`'s specification
  - `javac` does not check that

emptyQueue is a Factory Method

```
 9  public abstract class HelpQueueTest {
10      // A subclass says which implementation to test.
11      abstract HelpQueue emptyQueue();
```

```
1  public class ListQueueSpecTest extends HelpQueueTest {
2      HelpQueue emptyQueue() {
3          return new ListQueue();
4      }
5  }
```

- A subclass overrides it to choose the class: overridable, not static

Gamma, Helm, Johnson, and Vlissides, Design Patterns (1995), page 107. This code is on the screen only.

Static factory methods you have seen

- `List.of("ada", "bob")`
- `Optional.of(user)` and `Optional.empty()`
- `Path.of("messages.txt")`
- Each is a static factory method: a name, and no `new` at the call

One queue for office hours? (block F)

```
6   door.arrive("ada");
7   door.arrive("bob");
8   ta.help();
```

```
13  class Door {
14      private HelpQueue queue = new ListQueue();
22  class Ta {
23      private HelpQueue queue = new ListQueue();
```

- **Ada and bob arrive at the door. Who does the TA help?**
- `nobody to help`: the door and the TA each made a queue

`OfficeQueue`, a singleton (block G)

```
10  private static final OfficeQueue THE_QUEUE = new OfficeQueue();
14  private OfficeQueue() {
15  }
18  public static OfficeQueue theQueue() {
19      return THE_QUEUE;
20  }
```

- A program has one `OfficeQueue`: line 10 makes it
- The constructor is private, and `theQueue` hands out the one

The door and the TA ask for it (block F)

```
14   private HelpQueue queue = new ListQueue();
14   private HelpQueue queue = OfficeQueue.theQueue();
23   private HelpQueue queue = new ListQueue();
23   private HelpQueue queue = OfficeQueue.theQueue();
```

- Both ask for the one queue: the program prints `helping ada`

Will `javac` accept a second queue?

- At the door, line 14, someone writes `new OfficeQueue()` anyway

```
$ javac Session.java
Session.java:14: error: OfficeQueue() has private access in OfficeQueue
    private HelpQueue queue = new OfficeQueue();
                              ^
1 error
```

- We read one class, once: there is one `OfficeQueue`
- `javac` checks that no client makes a second, at every compile

Gamma, Helm, Johnson, and Vlissides, Design Patterns (1995), page 129: the client "will get an error at compile-time".

Singleton's cost

- "One" is now written into the type
- A second office hours needs a second queue, and the type forbids it
- Its clients are hard to test: a test cannot put a stand-in in its place
- So choose it only for a one that is decided, and say so

The testing cost: Bloch, Effective Java, 3rd edition (2018), Item 3.

Interning (block H)

```
4   System.out.println(Boolean.valueOf(true) == Boolean.valueOf(true));
5   System.out.println(Integer.valueOf(100) == Integer.valueOf(100));
6   System.out.println("hello" == new String("hello"));
7   System.out.println("hello" == new String("hello").intern());
```

- **Each line prints `true` or `false`. Which?**

```
$ java Sharing
true
true
false
true
```

Interning: nothing checks it

- **Interning**: keep one object for each value, and hand that one out
- It is safe for objects that never change
- No tool checks it
- Sharing is promised only for -128 to 127: compare with `equals`

Today's patterns

- **static factory method**: a constructor has no name
  - `javac` checks who can make the object, if the constructor is private
- **Factory Method**: code knows only an abstract type
  - `javac` checks nothing new

Today's patterns, continued

- **singleton**: nothing stops a second object
  - `javac` checks that no client makes a second
- **interning**: `new` makes an object even when an equal one exists
  - `javac` checks nothing

Reusing an object already made: Bloch, Effective Java, 3rd edition (2018), Item 1.

Three kinds of pattern

- **Creational**: about making objects. Today's, and builder on Friday
- **Structural**: Friday, after builder
- **Behavioral**: Wednesday, December 2

Coming up

- **Friday**: design patterns 2, and Reading 4 due, 11:59pm
- **Monday**: Reading 4 discussion, and upgrading apps
- **Tuesday**: section on generics, with a project proposal workshop
  - project proposal due, 11:59pm
- Hand in your half sheet at the door