Design Patterns 2

CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

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

Recall: design pattern

Definition: design pattern

A **design pattern** is a reusable solution to a recurring design problem.

- Or: working around flaws in language design

Three kinds of pattern

Creational

- static factory method
- Factory Method
- singleton
- interning
- builder

Structural

- adapter
- decorator
- proxy

Behavioral

- observer
- visitor

Wednesday today December 2

Gamma, Helm, Johnson, and Vlissides, Design Patterns (1995)

Which lines print `true`?

```
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());
```

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

The pattern: interning

- **Name**: interning. `Integer.valueOf` and `String.intern` do it
- **Problem**: `new` makes a new object even when an equal one exists
- **Solution**: keep one object for each value, and hand that one out
- **Gains**: no duplicate objects for equal values
- **Costs**: safe only for immutable objects; sharing is not promised
  - `Integer.valueOf` promises only -128 to 127: compare with `equals`

The order of the arguments

```
19  public Message(String author, String owner, String text) {
```

```
 8  String asker = "bob@example.com";
 9  String owner = "ada@example.com";
10  System.out.println(new Message(asker, owner, "hi ada"));
11  System.out.println(new Message(owner, asker, "hi ada"));
```

```
$ java Signing
bob@example.com signs ada@example.com's book: hi ada
ada@example.com signs bob@example.com's book: hi ada
```

A builder names each part

```
25  public static class Builder {
32      public Builder author(String author) {
33          this.author = author;
34          return this;
35      }
51      public Message build() {
52          if (author == null || owner == null || text == null) {
53              throw new IllegalStateException("a part is not set");
54          }
55          return new Message(this);
56      }
```

Can you see the mistake now?

```
11   System.out.println(new Message.Builder()
12       .author(asker).owner(owner).text("hi ada").build());
13   System.out.println(new Message.Builder()
14       .author(owner).owner(asker).text("hi ada").build());
```

```
$ java Signing
bob@example.com signs ada@example.com's book: hi ada
ada@example.com signs bob@example.com's book: hi ada
⋮
```

And with no owner set?

```
15   System.out.println(new Message.Builder()
16       .author(asker).text("hi ada").build());
```

```
$ java Signing
⋮
Exception in thread "main" java.lang.IllegalStateException:
    a part is not set
⋮
```

Half sheet, question 1

Which bugs does the compiler catch when we use the constructor?

When we use the builder?

```
10   System.out.println(new Message(asker, owner, "hi ada"));
```

```
11   System.out.println(new Message.Builder()
12       .author(asker).owner(owner).text("hi ada").build());
```

What `javac` checks in each call

- **Constructor**: `javac` checks that three `String`s are passed
  - a part left out does not compile
  - a swap compiles
- **Builder**: `javac` checks that each method gets a `String`
  - a part left out compiles: `build()` checks, at run time
  - a swap compiles too, but the call reads `author(owner)`

The pattern: builder

- **Name**: builder. `Message.Builder` is one
- **Problem**: many parts of one type, passed by position
- **Solution**: a named method per part, then `build()`
- **Gains**: the call names each part; `build()` can check them
- **Costs**: a second class; a part left out compiles
  - an IDE can often write the second class at a click

Bloch, Effective Java, 3rd edition (2018).

Can empty() return an ArrayDeque?

```
4   public static HelpQueue empty() {
5       return new ListQueue();
5       return new java.util.ArrayDeque<String>();
```

```
$ javac HelpQueues.java
HelpQueues.java:5: error: incompatible types:
    ArrayDeque<String> cannot be converted to HelpQueue
⋮
1 error
```

- `ArrayDeque` never says `implements HelpQueue`

DequeQueue, an adapter

```
 6  public class DequeQueue implements HelpQueue {
 7      private ArrayDeque<String> deque =
 8          new ArrayDeque<String>();
14      public void join(String netid) {
15          if (deque.contains(netid)) {
16              throw new IllegalArgumentException(netid);
17          }
18          deque.addLast(netid);
19      }
```

- Holds an `ArrayDeque` in a field, and forwards each call to it

Will javac accept it without the check?

```
15      if (deque.contains(netid)) {
16          throw new IllegalArgumentException(netid);
17      }
18      deque.addLast(netid);
```

```
$ javac DequeQueue.java
$ java Twice
[ada, ada]
```

- **Not checked**: a second `join` of a student throws

The pattern: adapter

- **Name**: adapter. `DequeQueue` is one
- **Problem**: a class does the work but lacks the interface
- **Solution**: implement the interface; hold the object; forward
- **Gains**: the class and its clients stay as they are
- **Costs**: a class to write; `javac` checks only its types

Structural patterns: wrappers

- A **wrapper** holds an object in a field and calls its methods
- Three kinds of wrappers:

	Behavior	Interface
adapter	same	changed
decorator	changed	same
proxy	same	same

Composition and forwarding

- The subtyping lecture's words, for what every wrapper does:
  - **composition**: building a class from an object in a field
  - **forwarding**: passing a method's work on to that object

LoggingQueue, a decorator

```
 4  public class LoggingQueue implements HelpQueue {
 5      private HelpQueue inner;
 7      public LoggingQueue(HelpQueue inner) {
 8          this.inner = inner;
 9      }
15      public void join(String netid) {
16          inner.join(netid);
17          System.out.println("log: join " + netid);
18      }
```

- Holds any `HelpQueue` in a field, and forwards each call to it

Any HelpQueue, even a wrapped one

```
10   hour(new LoggingQueue(new ListQueue()));
11   hour(new LoggingQueue(new FastQueue()));
12   hour(new LoggingQueue(new LoggingQueue(new ListQueue())));
```

```
$ java Wrapped
log: join ada
log: next ada
⋮
log: join ada
log: join ada
log: next ada
log: next ada
```

Half sheet, question 2

What guarantees do we get from the fact that the compiler accepts `LoggingQueue`?

```
 4  public class LoggingQueue implements HelpQueue {
 5      private HelpQueue inner;
 7      public LoggingQueue(HelpQueue inner) {
 8          this.inner = inner;
 9      }
15      public void join(String netid) {
16          inner.join(netid);
17          System.out.println("log: join " + netid);
18      }
```

What `javac` checks about a wrapper

- **Checked**: it has `HelpQueue`'s four methods, with their types
- **Checked**: `inner` has each method the wrapper calls on it
- **Not checked**: that it satisfies `HelpQueue`'s specification
  - `LoggingQueue` without `inner.join(netid)` compiles
  - `DequeQueue` without its check compiled, and let ada in twice

The pattern: decorator

- **Name**: decorator. `LoggingQueue` is one
- **Problem**: add a behavior to every implementation of an interface
- **Solution**: implement it; hold any object of it; forward, and add
- **Gains**: one class for every implementation; decorators stack
- **Costs**: more objects; a wrapper is not `==` to what it wraps

The same logging, by subclassing

```
 2  public class LoggingListQueue extends ListQueue {
 3      @Override
 4      public void join(String netid) {
 5          super.join(netid);
 6          System.out.println("log: join " + netid);
 7      }
15  }
```

- `hour(new LoggingListQueue())` prints the same lines
- **Can it log a `FastQueue`?** No: only a `ListQueue`

Can a subclass log the one `OfficeQueue`?

```
2  public class LoggingListQueue extends ListQueue {
2  public class LoggingOfficeQueue extends OfficeQueue {
```

```
$ javac LoggingOfficeQueue.java
LoggingOfficeQueue.java:2: error:
    OfficeQueue() has private access in OfficeQueue
public class LoggingOfficeQueue extends OfficeQueue {
       ^
1 error
```

- A wrapper can hold the one queue: it never calls the constructor

Composition against subclassing

	`LoggingListQueue`	**`LoggingQueue`**
Logs	a `ListQueue`	any `HelpQueue`
Writes	`join` and `next`	all four methods
Relies on	how `ListQueue` is written	`HelpQueue`'s specification
Chosen	when the class is written	when an object is wrapped

Subtyping, "Is a Square a Rectangle?"

Bloch, Effective Java, 3rd edition (2018)

Saved books, read at the start

```
15   for (Path file : files.sorted().toList()) {
16       String name = file.getFileName().toString();
17       String login = name.replaceFirst("[.]txt$", "");
18       books.put(login, new SavedBook(file));
19   }
```

- Each user's book is saved in a file, one file per user
- SavedBook reads its whole file when it is made

Ada asks for her book. Which files are read?

```
$ java Startup
reading ada@example.com.txt
reading bob@example.com.txt
started
[ada@example.com: hello, bob@example.com: hi ada]
[ada@example.com: hello, bob@example.com: hi ada]
```

LazyBook stands in for a SavedBook

```
 6  class LazyBook implements Book {
 7      private final Path file;
 8      private SavedBook real = null;
30      @Override
31      public List<String> messages() throws IOException {
32          return real().messages();
33      }
34  }
```

- Both classes implement Book, now an interface

`real()` reads the file once

```
18      private SavedBook real() throws IOException {
19          if (real == null) {
20              real = new SavedBook(file);
21          }
22          return real;
23      }
```

- `real()` reads the file the first time it is called

Books makes a LazyBook for each file

```
18   books.put(login, new SavedBook(file));
18   books.put(login, new LazyBook(file));
```

```
$ java Startup
started
reading ada@example.com.txt
[ada@example.com: hello, bob@example.com: hi ada]
[ada@example.com: hello, bob@example.com: hi ada]
```

- Only ada's file, when she asks, and only once

Will javac accept Book without throws?

```
14  List<String> messages() throws IOException;
14  List<String> messages();
```

```
$ javac *.java
LazyBook.java:31: error: messages() in LazyBook cannot implement
    messages() in Book
⋮
  overridden method does not throw IOException
1 error
```

- javac checks each method's types and exceptions

The pattern: proxy

- **Name**: proxy. `LazyBook` is a virtual proxy
- **Problem**: control access to an object, such as when it is made
- **Solution**: a stand-in with its interface forwards each call
- **Gains**: only `Books` changes; a file is read only if it is used
- **Costs**: forwarders, unchecked; `Book` declares `IOException`

Today's patterns

- **interning**: `javac` checks nothing
- **builder**: `javac` checks each part's name and type
  - not that every part is set
- **adapter**, **decorator**, **proxy**: wrappers
  - `javac` checks their types, not the specification
- **composition** over subclassing
  - `javac` refuses a subclass of `OfficeQueue`
  - not whether `ListQueue`'s methods call one another

Three kinds of pattern

Creational

- static factory method
- Factory Method
- singleton
- interning
- builder

Structural

- adapter
- decorator
- proxy

Behavioral

- observer
- visitor

Wednesday today December 2

Coming up

- **Monday**: Reading 4 discussion, and upgrading apps
- **Tuesday**: section on generics, with a project proposal workshop
  - project proposal due, 11:59pm
- **Wednesday to Friday**: no class (Thanksgiving)
- Hand in your half sheet at the door