Java:
Learning to Program with Robots

Chapter 11: Building Quality Software
After studying this chapter, you should be able to:

- Identify characteristics of quality software, both from the users’ and programmers’ perspectives
- Follow a development process that promotes quality as you develop your programs
- Avoid common pitfalls in designing object-oriented programs
- Include defensive programming measures to make errors more likely to expose themselves so they can be fixed
- Explain characteristics of quality user interfaces and describe an iterative methodology for developing them
Requirements:
What the program must do.

Architecture:
A description of the most important classes and how they relate to each other.

Scenario:
A specific task a user may want to perform with the program.
A program is required to help with selling tickets in a concert hall. The program must show a list of upcoming concerts and a list of existing patrons. New concerts may be added to the list of concerts. Concerts may also be deleted. Similarly, patrons may be added to or deleted from the list of patrons.

When a particular concert is identified, the program must display the tickets that are still available for that concert. The user should then be able to select a patron from the patron list and one or more tickets to sell to the patron. A patron’s past ticket purchases are kept for marketing purposes.

Tickets are divided into three groups for pricing. Gold is the most expensive, followed by Silver, and finally Bronze. When a concert is entered into the system, the prices for Gold, Silver, and Bronze tickets are specified. Tickets are labeled with numbers for rows (1…15) and letters for seats (A…T).

Information must be saved in a file so the program can be stopped and restarted.

A possible user interface is sketched on the next slide.
Case Study 1: Defining Requirements (2/2)

```
Concert Hall

Concerts
- ID: 109, Date: Sept 1, Time: 8:00, Performer: K-W Symphony Orchestra
- ID: 110, Date: Sept 2, Time: 8:00, Performer: K-W Symphony Orchestra
- ID: 111, Date: Sept 3, Time: 8:30, Performer: Janet Lopez

Patrons
- ID: 1000, Name: Byron Weber Becker, Phone: 579-0001
- ID: 1001, Name: Karen Anderson, Phone: 883-1234
- ID: 1002, Name: Arnie Dyck, Phone: 886-5678

Tickets
- Sell
```
Designing the architecture consists of five tasks:

- Identify the most important classes and methods for the program using an analysis of the nouns and verbs in the requirements.
- Summarize the responsibilities and collaborators for each class on index cards.
- List scenarios in which the software will be used.
- Walk through the scenarios using the index cards to further develop the responsibilities and collaborators.
- Develop a class diagram based on the responsibilities and collaborators listed on the index cards.

The result is a class diagram used to guide implementation.
The program sells tickets.
The program shows a list of upcoming concerts.
The program shows a list of existing patrons.
The program adds new concerts (to the list of concerts).
The program adds patrons (to the list of patrons).
The (user, patron?) identifies a concert.
The program displays available tickets (for an identified concert).
The user selects a patron from the patron list.
The user selects one or more tickets.
A patron has past ticket purchases (kept for marketing purposes).
Tickets have a price group (Gold, Silver, Bronze).
Concerts have prices for Gold, Silver, and Bronze tickets.
Tickets have a row number (1..15).
Tickets have a seat letter (A..T).
The program saves information to a file.
CRC (Class-Responsibilities-Collaborators) Cards

- are usually 4”x6” index cards
- summarize one class’s responsibilities and collaborators
- are an inexpensive (in materials and time to develop) precursor to the class diagram

<table>
<thead>
<tr>
<th>ConcertHall</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Responsibilities</strong></td>
<td><strong>Collaborators</strong></td>
</tr>
<tr>
<td>sell tickets</td>
<td>ConcertList</td>
</tr>
<tr>
<td>add a concert to list of concerts</td>
<td>PatronList</td>
</tr>
<tr>
<td>add a patron to list of patrons</td>
<td></td>
</tr>
<tr>
<td>save information to a file</td>
<td></td>
</tr>
</tbody>
</table>
11.2.3: Iterative Development

Define the Requirements
- Describe the desired system

Design the Architecture
- Identify classes and methods
- Create CRC cards
- List scenarios
- Walk through scenarios
- Develop a class diagram

Choose a Set of Scenarios

Refine the Design
- Refine requirements
- Add new scenarios
- Walk through new scenarios

Implement the Scenarios
- Choose one scenario
- Write tests for that scenario
- Write code to pass tests
- Refactor

Evaluate with Users
• Document classes and methods.
• Avoid nested loops.
• Keep methods short.
• Make helper methods private.
• Put duplicated code in a helper method.
• Make instance variables private.
• Write powerful constructors.
• Keep data and processing together.
• Write immutable classes.
• Delegate work to helper classes.
Many of these rules of thumb (heuristics) relate to four features that have been long recognized as crucial to quality code:

- **Encapsulation**: Grouping data and related services into a class.
- **Cohesion**: The extent to which each class models a single, well-defined abstraction and each method implements a single, well-defined task.
- **Information Hiding**: Hiding and protecting the details of a class’s operation from others.
- **Coupling**: The extent to which interactions and dependencies between classes are minimized.
11.3.2: Managing Complexity (2/2)
11.3.2: Encapsulation
11.3.2: Cohesion
11.3.2: Information Hiding
Exceptions

- Immediately alert programmers to things that have gone wrong, including helpful information for debugging the program.

Design by Contract

- Write down exactly what a method expects of its caller and what it will do in return (a contract for the method).

Assertions

- Specify what should be true at critical points in your code. To the extent that it’s possible, check that the assertions actually are true. If they aren’t, throw an exception.
The Pharmacy program is poorly written and ignores many of the rules of thumb discussed earlier. As a result, its encapsulation, cohesion, information hiding, and coupling are all very poor.

**Refactor** the program. Refactoring does not change the program’s functionality. It changes its internal structure so that it has higher quality from a programmer’s perspective.
11.5.1: Iterative User Interface Design (1/2)

Design

User Testing and Evaluation

Prototyping
The five E’s of User Interface Evaluation:

- **Effective**: The completeness and accuracy with which users achieve their goals.

- **Efficient**: The speed and accuracy with which users can complete their tasks.

- **Engaging**: The degree to which the tone and style of the interface makes the product pleasant or satisfying to use.

- **Error Tolerant**: How well the design prevents errors or helps with recovery from those that do occur.

- **Easy to Learn**: How well the product supports both initial orientation and deepening understanding of its capabilities.
Well-designed user interfaces are:

- **Controlled by the user**: Give the user as much control over the process as is consistent with the user’s knowledge and skill level.

- **Responsive**: Give the users constant feedback, and give it fast enough so the user can work at full speed without waiting for the computer.

- **Understandable**: Some of the techniques for understandable interfaces include consistency, good visual structure, and information recognition rather than recall.

- **Forgiving**: Prevent as many mistakes as possible; make it easy to correct the mistakes that do happen.
an iterative development process

defines
requirements

designs
the architecture

defines
important classes and methods

nouns and verbs

help discover

walk throughs

refine

can be evaluated from

software quality

implements and evaluates scenarios

walk throughs

is enhanced by using

repeatedly

software quality

is improved by

can be evaluated from

proactively responding to exceptions

is improved by

important classes and methods

a programmer's perspective of quality

includes

testable programs

includes
maintainable programs

includes

understandable programs

includes

weak coupling

information hiding

high cohesion

encapsulation

classes, responsibilities, and collaborators

proactively responding to exceptions

is improved by

important classes and methods

can be evaluated from

a user's perspective
We have learned:

• that software “quality” involves understandability, testability, and maintainability.

• that a development process featuring iteration helps lead to quality software.

• how to use the specification’s nouns and verbs to design a program.

• how to create CRC cards and use them in a walk-through.

• rules of thumb that provide guidance in implementing quality software.

• that strong encapsulation, high cohesion, information hiding, and weak coupling help manage software complexity.

• how to use exceptions, assertions, and “design by contract.”

• that iteration, prototyping, and evaluation have strong roles in developing quality user interfaces.