Principles of Software Construction: Objects, Design, and Concurrency

A formal design process 2

Josh Bloch    Charlie Garrod    Darya Melicher
Administrivia

• Reading due today: UML & Patterns Ch. 14, 15, and 16
• Midterm exam Thursday in class
  – Review session today 5–7 pm in Doherty 2315
• Homework 4a due next Thursday
  – Mandatory design review meeting before the homework deadline

It's National Voter Registration Day!
Key concepts from Thursday
Create a domain model for an airport self-check-in system

An airport self-check-in system is used to check in passengers for their upcoming flights. Each passenger is assigned a seat on each of their flights. During the check-in procedure, the passenger may choose to check in one or more bags as their baggage. At the end of a check-in procedure, the passenger receives a boarding pass for each of the flights of their trip they checked into.
Notes on creating a domain model

• All concepts are accessible to a non-programmer
• The UML is somewhat informal
• Real-world abstractions and "is-a" relationships are appropriate
• Aggregate types are usually modeled as classes
• Primitive types are usually modeled as attributes
• Iteration is important
A system sequence diagram for the airport self-check-in system

Use case scenario: After a passenger is authenticated with the system using their booking ID, they should be able to change a seat on any of their flights to any unoccupied seat in the cabin of the same type.
A system behavior contract for the airport self-check-in system

Use case scenario: After a passenger is authenticated with the system using their booking ID, they should be able to change a seat on any of their flights to any unoccupied seat in the cabin of the same type.

Operation: changeSeat(flight, newSeat)

Pre-conditions: Passenger has already authenticated with the system. Chosen seat is on the flight passenger has a booking for. Seat is not currently occupied by another passenger.

Post-conditions: Authenticated passenger's account records the new seat assignment. New seat is now marked as occupied. Old seat is now marked as free.
Distinguishing domain vs. implementation concepts

• Domain-level concepts
  – Almost anything with a real-world analogue

• Implementation-level concepts
  – Implementation-like method names
  – Programming types
  – Visibility modifiers
  – Helper methods or classes
  – Artifacts of design patterns
Summary: Understanding the problem

• Domain models — understand domain and vocabulary
• System sequence diagrams + behavioral contracts — understand interactions with environment
• Be fast and (sometimes) loose
  – Elide "obvious" details
  – Iterate, iterate, iterate, ...
• Domain classes often turn into Java classes
  – Lower representational gap for better understandability and maintainability
  – Some domain classes don't need to be modeled in code; other concepts only live at the code level
• Get feedback from domain experts
Thursday and today you did / will do...

- 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

Understand the problem (Thursday)

Define a solution (today)
Thursday and today you did / will do...

- 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

Understand the problem (Thursday)

Define a solution (today)
Thursday and today you did / will do...

- 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
Object-oriented programming

- Programming based on structures that contain both data and methods

```java
public class Bicycle {
    private int speed;
    private final Wheel frontWheel, rearWheel;
    private final Seat seat;
    ...
    public Bicycle(...) { ... }
    public void accelerate() {
        speed++;
    }
    public int speed() { return speed; }
}
```
Two types of object responsibilities

- **Data ("knowing")**
  - Private or otherwise encapsulated data
  - Data in closely related objects
- **Methods ("doing")**
  - Private or otherwise encapsulated operations
  - Object creation, of itself or other objects
  - Initiating actions in other objects
  - Coordinating activities among objects
Assign object responsibility using interaction diagrams

- For a given system-level operation, create an object interaction diagram at the *implementation-level* of abstraction
- Implementation-level concepts:
  - Implementation-like method names
  - Programming types
  - Helper methods or classes
  - Artifacts of design patterns
An interaction diagram for the library system 1

Use case scenario: A library member should be able to use her library card to log in at a library system kiosk and ...
An interaction diagram for the library system 2

Use case scenario: ... 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 loan period to the current day, and record the book and its due date as a borrowed item in the member's library account.
Create an interaction diagram for the airport self-check-in

Use case scenario: After a passenger is authenticated with the system using their booking ID, they should be able to change a seat on any of their flights to any unoccupied seat in the cabin of the same type.
An example domain model for an airport self-check-in system

```
<table>
<thead>
<tr>
<th>Bag</th>
<th>0..* has many</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Passenger</th>
<th>1</th>
<th>associated with</th>
<th>0..*</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flight</th>
<th>1</th>
<th>has many</th>
<th>0..*</th>
</tr>
</thead>
<tbody>
<tr>
<td>gate</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BoardingPass</th>
<th>1</th>
<th>has a</th>
<th>1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Seat</th>
<th>1</th>
<th>has a</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>number row</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```


Interaction diagrams help evaluate design alternatives.

Create two possible interaction diagrams:

1. Printing a boarding pass, assuming that the main method (or a delegated method or class) has responsibility for printing the boarding pass.
2. Printing a boarding pass, assuming that the boarding pass class has responsibility for printing itself.
Heuristics for responsibility assignment

- Controller heuristic
- Information expert heuristic
- Creator heuristic

**Goals**
- Understandability
- Reuse
- Maintainability
- Extensibility
- ...

**Principles**
- Low coupling
- High cohesion
- Low representational gap

**Patterns**
- Strategy
- Template method
- Decorator
- Iterator

**Heuristics**
- Controller
- Information expert
- Creator
Controller heuristic

• Assign responsibility for all system-level behaviors to a single system-level object that coordinates and delegates work to other objects
  – For complex use case scenarios, consider specific sub-controllers

• Design process: Extract interface from system sequence diagrams
  – Key principles: Low representational gap and high cohesion
Controller heuristic used in the library system

```plaintext
libraryController

loginMember(libraryCardNumber)

borrow(item: Item)

success?, dueDate
```

```
:LibrarySystem

LibraryController

loginMember(libraryCardNumber: Int)

borrow(item: Item)
```
Use the controller heuristic

authenticating(bookingId)

changeSeat(flight)

seatsAvailable

chooseSeat(newSeat)

success?, newBoardingPass

SelfCheckInController

authenticate(bookingId)

changeSeat(flight)

chooseSeat(newSeat)
Information expert heuristic

• Assign responsibility to the class that has the information needed to fulfill the responsibility
  – Initialization, transformation, and views of private data
  – Creation of closely related or derived objects

• Design process: Assignment from domain model
  – Key principles: Low representational gap and low coupling
Use the information expert heuristic

• In the airport self-check-in system, what object should have the responsibility to update the passenger’s seat?
• What is the relevant information?
  – Who knows the passenger’s seat?
  – Who can check whether the new seat is valid (e.g. exists on the flight)?
  – Who can check whether the new seat is in the same type of cabin?
Interaction diagrams help evaluate design alternatives

Create two possible interaction diagrams:

1. Printing a boarding pass, assuming that the main method (or a delegated method or class) has responsibility for printing the boarding pass
2. Printing a boarding pass, assuming that the boarding pass class has responsibility for printing itself
Creator heuristic: Who creates an object Foo?

• Assign responsibility of creating an object Foo to a class that:
  – Has the data necessary for initializing instances of Foo
  – Contains, aggregates, or records instances of Foo
  – Closely uses or manipulates instances of Foo

• Design process: Extract from domain model, interaction diagrams
  – Key principles: Low coupling and low representational gap
Use the creator heuristic

• In the airport self-check-in system, what object should have the responsibility for creating a boarding pass?
Thursday and today you did / will do...

- 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

Understand the problem (Thursday)
Define a solution (today)
Creating an object model

• Extract data, method names, and types from interaction diagrams
  – Include implementation details such as visibilities
Creating an object model for the library system
An example object model for a library system

**LibrarySystem**
- currentSession: LibraryAccount
- loginMember(libraryCardNumber: int)
- borrow(item: Item)
- retrieveAccount(libraryCardNumber: int)
- logoutMember()
- payLateFees(dollars: int)

**LibraryAccount**
- libraryCardNumber: int
- firstName: String
- lastName: String
- lateFeesOwed: int
- getFirstName(): String

**Book**
- title: String
- author: String
  ...
  + getLoanPeriod()

**Movie**
- title: String
- author: String
  ...
  + getLoanPeriod()

**Item**
- loanPeriod: int
- lateFee: int
  + getLoanPeriod()

**BorrowedItem**
- dueDate: Date
- returned: Date
  ...
  + hasBeenReturned(): boolean
  + isOverdue(currentDate: Date): boolean

**members**
0..*

**items**

**borrowedItems**
0..*
Object-level artifacts of this design process

- **Object interaction diagrams** add methods to objects
  - Can infer additional data responsibilities
  - Can infer additional data types and architectural patterns
- **Object model** aggregates important design decisions
  - Is an implementation guide
Summary:

- Domain-level models help you understand the problem domain.
- Object-level interaction diagrams and object model systematicaly guide the design process.
  - Convert domain model, system sequence diagram, and contracts to object-level responsibilities.
- Use heuristics to guide, but not define, design decisions.
- Iterate, iterate, iterate, ...