Software Engineering Techniques

What makes software "good"? Today, we start building some guidelines, object-oriented design principles (OODPs), that will help us quickly plan software systems with desirable qualities and rule out designs that are likely to cause us (or our teammates, or worse, our clients) frustration in the future.

Content Learning Targets

After completing this activity, you should be able to say:

- I can list several attributes that make software "good".
- I can draw a UML class diagram representing given Java classes.
- I can create stubs of Java classes from a given UML class diagram.
- I can use the PlantUML service to create simple UML class diagrams.
- I can apply object-oriented design principles (OODPs) 1 and 2 to evaluate design options.
- I can improve candidate designs by fixing flaws with respect to OODPs 1 and 2.

Process Skill Goals

During the activity, you should make progress toward:

- Communicating object-oriented design ideas to team members using precise language and diagrams. (Communication)
- Reading system requirements descriptions and extracting key information for object-oriented design. (Information Processing)
- Building consensus from different design ideas and including all team members in design discussions. (Teamwork)

Facilitation Notes

Model 1 introduces software design principles and motivates UML class diagrams.

Model 2 defines the first two object-oriented design principles and walks through the Book Tracker design problem.

Model 3 structures additional practice design problems: Company Accounts and Card Game.

Key questions: #5, e), f), #12, #13

Model 1 The Software Design Process

Questions (30 min) Start time:

1. Summarize from the classwide discussion: What makes software good?

Answers will vary slightly based on student input, but should be along the lines of: works correctly (functional), easy to extend/modify/reuse, readable, simple (as possible), efficient

Interlude: Requirements and Design

The software development process typically begins with abstract ideas, such as "dog-walking gig economy marketplace". These ideas need to be formed into ***requirements***—what the clients want from the software. (CSSE371 will teach you this ***requirements analysis*** step.) Once we have clear requirements, we need to design software to meet them. In CSSE220, we start practicing the design process. You will build on these skills in future courses, including CSSE280 (designing web applications), CSSE333 (designing databases and applications which interact with them), CSSE374 (applying more advanced software design patterns), and senior capstone.

2. If we come up with multiple design options, how can we figure out which one to use? Brainstorm some ideas.

We could implement several possible designs, then compare them at the end, but that would waste a lot of time. Instead, we could "sketch out", in some way, several candidate designs, and try to see which is best ***without*** implementing them all.

3. Software design is an ***iterative*** process in which we ***learn from failure***, refining our design until we land on something workable. To make the process efficient, we need some concise way of describing candidate designs. In the specific context of object-oriented design, how could we accomplish this? (Hint: we have seen a useful type of diagram in previous classes.)

We can use UML diagrams to summarize designs and visualize several classes together. We will add arrows between classes in the UML diagram to represent their relationships.

4. Sketch (on a whiteboard or paper) a UML diagram for the two classes below. Refer to past worksheets as needed. Make sure to capture the relationship between the two classes.

```
public class Ninja {
    private String name;
    private int level;

    public Ninja(String name) {
        this.name = name;
        this.level = 1;
    }

    public void setLevel(int level) {
        this.level = level;
    }
}
```

```
public class Pirate {
    private Ninja enemy;

    public Pirate(Ninja enemy) {
        this.enemy = enemy;
    }

    public Pirate() {
        this.enemy = new Ninja("Foo");
    }
}
```

5. Compare your UML diagram against this solution. What differences do you notice?

Answers will vary. You may have included the constructors as methods, but we often omit these. The biggest difference is probably that you included `enemy` as a field in `Pirate` instead of drawing an arrow from `Pirate` to `Ninja`. We read this arrow as "Pirate ***has-a*** Ninja". This arrow implies the existence of the field of type `Ninja` in `Pirate`.

6. Add the below `Wizard` class to your UML diagram.

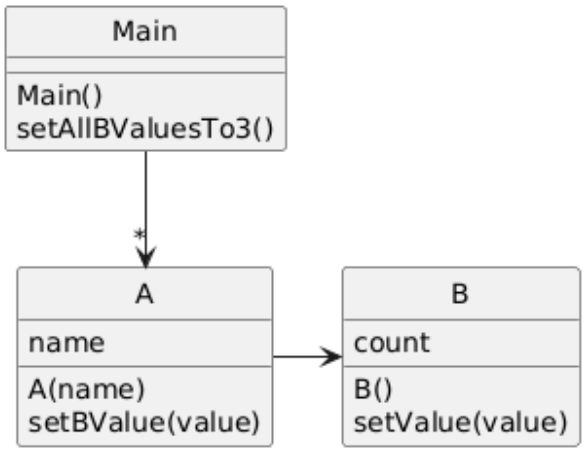
```
public class Wizard {
    private String name;
    private ArrayList<Ninja> ninjaFriends;

    public Wizard(String name) {
        this.name = name;
        this.ninjaFriends = new ArrayList<>();
    }

    public void addNinjaFriend(Ninja newFriend) {
        this.ninjaFriends.add(newFriend);
    }
}
```

See solution. The "*" on the arrow means `Wizard` stores ***0 to many*** references to `Ninja` objects.

7. We will use the PlantUML web app to create UML class diagrams throughout the course. **In pairs** (or solo if you only have three today), practice using this tool by reproducing the below UML class diagram. Use the above example and your UML cheatsheet for reference. (Note: We will always add `skinparam style strictuml` as a first line to keep our UML diagrams simple.)



PlantUML solution

Model 2 Object-Oriented Design Principles 1 and 2

Object-oriented design principles (OODPs) are guidelines (think, "rules of thumb") that help developers write good software. Today, we will get familiar with the first two OODPs from our CSSE220 list (see the OODP reference sheet in Moodle).

Questions (25 min) Start time:

8. The most important object-oriented design principle (OODP #1) is…

functionality! Our design must (a) be able to ***store required information*** (one/many to one/-many relationships), (b) be able to ***access the required information*** to accomplish tasks, and (c) ***not duplicate data*** (IDs/object references are OK)

9. A second priority (OODP #2) is to structure the design ***around the data*** to be stored. Based on the examples of classes we have created so far, which part of speech usually signals we should consider creating a class?

Nouns, because each class is a blueprint for some type of object. Note that we will generally not use plural nouns as class names; instead, we will create multiple instances of an object when needed.

10. Part (b) of OODP #2 states, "Classes should have intelligent behaviors (methods) that may operate on their data." Why do you think this is part of OODP #2?

It makes sense to put methods that use certain data in the same place as those data for easy access. And most interesting real-world objects have not only internal "data" but also behaviors using that data. (For example, a coffee maker is more than just a container for grounds.)

Interlude: Design Problems
Throughout the course, we will practice object-oriented design using story problems: (multi-)paragraph descriptions of system requirements. We will practice the design process by

a) identifying flaws in designs we give you, and

b) developing your own design(s), following the OODPs.

Since object-oriented design is such an important skill, there will be several opportunities to practice in class, extra practice problems that you can work through on your own time, and assignment/exam design problems, including your final project design.

11. **Design Problem 1: Book Tracker**

> A website tracks books and the kids that read them. For each book the system stores the name and author. For each kid the system stores name and grade level. The teacher enters when a kid reads a particular book. It should be possible to print a report on a book that includes all kids who have read a particular book (with their grade level). It should be possible to print a report on a kid that includes the books (with authors) a particular kid has read.

Suppose we start with a class called `BookMain` that takes care of user input via three methods: `handleNewReading(String bookName, String kidName)`,`handlePrintReportForKid(String kidName)`, and `handlePrintReportForBook(String bookName)`.

a) What classes should we add? And with what fields?

Looking at the nouns in the description, "book" and "kid" both make sense to turn into classes. Our `Book` class should have `name` and `author` fields. Our `Kid` class should have `name` and `gradeLevel` fields.

b) What relationships do we need to represent between classes?

`BookMain` should probably keep a list of all `Kid` objects and a list of all `Book` objects. Each `Kid` might need to keep a list of its read `Book` objects, and each `Book` might need to keep a list of the `Kid` objects who have read it.

c) What methods should our new classes have?

Both `Kid` and `Book` need a `printReport()` method. Also, `Kid` needs an `addBook(Book book)` method, and `Book` needs an `addKid(Kid kid)` method.

d) Create a UML class diagram for your design using PlantUML. (Remember to add the `skinparam style strictuml` line at the top.) To save your design, click the "Decode URL" button, then copy/paste the updated URL from your browser's address bar.

PlantUML solution

e) Consider Bad Design A. Applying OODPs 1 and 2, what is wrong with this design?

This design does not function. There is no (sane) way to look up a book for printing a report or for associating with a Kid.

f) Consider Bad Design B. Applying OODPs 1 and 2, what is wrong with this design?

This design functions but there is a lot of data duplication, which in general we want to avoid. In particular, the author/title information in Kid is duplicated, and the name/-grade level information in the book is duplicated.

g) Revisit your original design. If it had similar flaws to those in Bad Designs A and B, fix them. If it didn't have those design flaws, double-check that your design adheres to OODPs 1 and 2. Provide a link to your final PlantUML design below.

PlantUML solution

Aside: If you want a detailed, step-by-step process for parsing story problems and identifying classes, fields, and methods, see the BookTracker slides in today's class materials folder.

Model 3 More Design Practice

We will end today with two more design problems. For each design problem, split into pairs (or individuals, if you only have three today) for parts (a) and (b), then regroup for the rest.

Questions (30 min) Start time:

12. Design Problem: Company Accounts

A particular company keeps a variety of different accounts for its projects. Each account has an account number and a balance. When a deposit or withdrawal occurs, the transaction occurs immediately, and the current balance should be updated. The system should support getting the current balance.

The system should also support getting the balance as it existed at ***any*** date/time in the past. Note the input historical date/time may not correspond to a particular transaction time—for example, if the system had a balance of $1 at 1 p.m. and then was changed to $2 at 3 p.m., a request for the balance at 2 p.m. should return $1.

a) Suppose our main class is called `CompanyExpenseMain`. What `handle` methods will it need?

```
handleDeposit(accountID, amount),handleWithdrawal(accountID, amount),
handleGetCurrentBalance(accountID),
handleGetHistoricalBalance(accountID, dateTime)
```

b) Determine what classes to add, what fields and methods they need, and how they should relate to each other. Sketch (on a whiteboard or paper) a UML diagram for your design.

c) (***Regroup.***) Compare your team's UML diagrams against Bad Design A. What differences do you notice? And what is wrong with Bad Design A, in terms of OODPs 1 and 2?

Bad Design A violates OODP 1(a), because the main class only has one `Account`, but it may need to have many. It is also moderately difficult to compute historic balances, so you could argue it violates 1(b) too.

d) Compare your team's UML diagrams against Bad Design B. What differences do you notice? And what is wrong with Bad Design B, in terms of OODPs 1 and 2?

Bad Design B violates OODP 1(a). `Account` doesn't have enough data to make `getHistoricalBalance` work. Perhaps `oldBalances` is a list of balances? But transaction dates are not stored, so we can't look up the balance on a particular day/time.

e) Combine your team's original designs into one final design, applying any lessons learned from Bad Designs A and B. Create a UML class diagram for your design using PlantUML. (Remember to add the `skinparam style strictuml` line at the top.) Save your link.

Solution A: parallel ArrayLists for balances and transaction dateTimes
Solution B: modified `Transaction` class

13. Design Problem: Card Game

In a particular card game, players have hands of cards. Each card is worth some points and also has a color (red, blue, green). During play, players accrue bonuses that mean cards of a particular color are worth bonus points. During play, sometimes a random card is selected from one player's hand and moved to another player's hand. At the end of game, it is necessary to compute the total points for each player's hand.

Reminder: Do parts (a) and (b) in pairs/solo, then regroup.

a) Suppose our main class is called `GameMain`. What `handle` methods will it need?

```
handleMoveRandomCard(startPlayerName, endPlayerName),
handleAddBonus(playerName, colorName, bonusAmount),
handleComputeHandValueForPlayer(playerName)
```

b) Determine what classes to add, what fields and methods they need, and how they should relate to each other. Sketch (on a whiteboard or paper) a UML diagram for your design.

c) (***Regroup.***) Compare your team's UML diagrams against Bad Design A. What differences do you notice? And what is wrong with Bad Design A, in terms of OODPs 1 and 2?

Bad Design A violates OODP 1(a), because a player's color bonuses can't be preserved if they lose all their cards of that color. Also, this design makes use iterate over all cards to get a player's hand to move cards or compute final totals. (Alternatives: 1(c), cards duplicate player names and color bonuses; 2(a), missing `Player` class.)

d) Compare your team's UML diagrams against Bad Design B. What differences do you notice? And what is wrong with Bad Design B, in terms of OODPs 1 and 2?

Bad Design B violates OODP 1(a), because once a card is added to a player's hand, its specific point value is lost, so the card cannot be randomly moved to another player's hand. You could also argue for 2(a), since there is no `Card` class.

e) Combine your team's original designs into one final design, applying any lessons learned from Bad Designs A and B. Create a UML class diagram for your design using PlantUML. (Remember to add the `skinparam style strictuml` line at the top.) Save your design below by clicking the "Decode URL" button, then copy/pasting the updated URL from your browser's address bar.

Solution: each `Player` maintains a list of their `Card` objects. (Remember we implicitly assume we have the necessary constructors, getters, and setters.)