Analysis, Design, and Justification (I): Introduction

Ruben Acuña

Spring 2022

Learning Objectives

The final goal:

- ADJ-CO1: Understand problem solving as a learning and growth process.
- ADJ-CO2: Construct a sound and evidence-based solution to a problem.
- ADJ-CO3: Create solutions to open-ended problems without a specific correct solution.
- ADJ-CO4: Create solutions to ill-defined problems with problems statements which interfere with problem solution.
- ADJ-CO5: Construct solutions based on contextual needs of a problem, team, or customer.

A Typology of Problems

Real Problems

Here’s a thought: how are real problems different than what we see in a class at school?

Educational Problems

- Here are a couple of answers of examples from SER222 problems: one short answer, and one programming.
- Questions:
  - What do you know immediately about these problems?
  - What the steps in solving it?
  - If someone handed you a “solution”, could you check it and make sure they are right?

Question 3

[Acuña] **What** are the tilde approximation of the following growth expressions?

1. $5n^2+3n^2+30$
2. $100n+30n^2+\log(n)$
3. $10n^2\log_2 n+n$

Use full expression notation (e.g., n+10 ~ n) in your answers.

2 Requirements [32 points]

For this assignment, you will be writing a Deque ADT (i.e., LastnameDeque). A deque is closely related to a queue - the name deque stands for "double-ended queue." The difference between the two is that with a deque, you can insert, remove, or view, from either end of the structure. Attached to this assignment are the two source files to get you started:

- Deque.java - the Deque interface; the specification you must implement.
- BaseDeque.java - The driver for Deque testing; includes some simple testing code.

Your first step should be to review these files and satisfy yourself that you understand the specification for a Deque. Your goal is implement this interface. Implementing this ADT will involve creating 9 methods:

- public LastNameDeque() - a constructor [3 points]
- public void enqueueFront(Item element) - see interface [5 points]
- public void enqueueBack(Item element) - see interface [6 points]
- public Item dequeueFront() - see interface [5 points]

Real-world Problems

"Just meet the course outcomes. Teach it like CSE310."

A *real-world problem* is typically characterized by being *open-ended* and *ill-defined*.

Same Questions:

- What do you know immediately about this problem?
- What are the steps in solving it?
- If someone handed you a "solution", could you check it and make sure they are right?

Open Ended

Close-ended ——— Open-ended

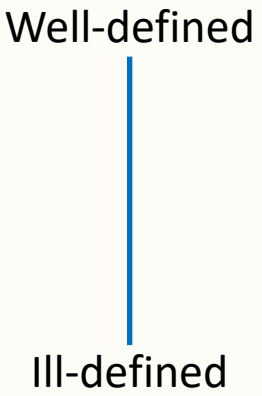
- **Closed-ended Problem:** An answer must look a very specific way (e.g., only one answer), and/or be achieved in a limited number of ways.
- **Opened-ended Problem:** An answer has no specific form that limits that it, rather there are some constrains that define what it means to be an answer.

- Example 1: What is the Tilde approximation of 5n^2+1?
- Example 2: Write a program to take the sum of the first n positive integers.
- Example 3: Consider designing a program where you need to store a fixed size collection of items. Would you use an array or linked list?

Design Problems

- Unfortunately for us, the types of problems that engineers are asked to solve are often design problems.
- A ***design problem*** has the flavor of picking one solution among many options, and where the "best" solution may not be immediately apparent and may involve tradeoffs.
  - An example of a design problem: converting the idea for this sentence into a written sentence.
- In this case, the open-endedness comes from the fact that some attributes of the solution are not predetermined. For example: the programming language you use, or even the statements on a particular line.
- It may also be the case that you're allowed to solve a problem differently: you can write code or just find an existing library that does it for you.

Ill-defined



- ***Well-defined:*** We know what we start with, we have options/ideas for what can be done, and we'll know the solution when we see it. (We say a solution is *verifiable*, perhaps with some additional information called a *certificate*.)
- ***Ill-defined:*** The basic elements of the problem, or the actions that can be taken, or what a solution looks like may be unclear.

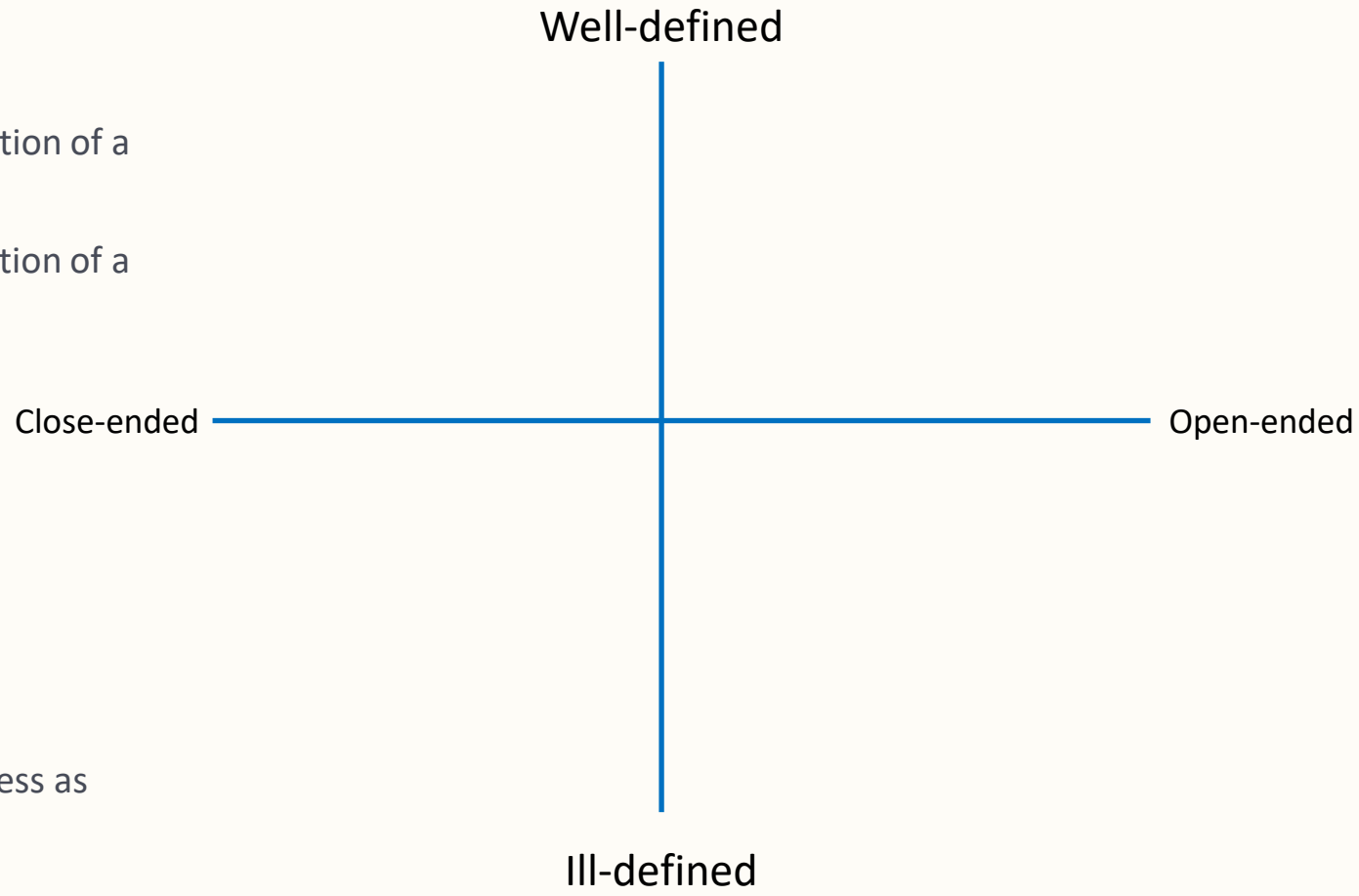
Related ideas: *underspecified problems*, *ill-structured problems* and *wicked problems*.

Practicing Classification

Let’s try plotting a few of these:

- Finding the tilde approximation of a function?
- Finding the tilde approximation of a method?
- You choose!
- Curing COVID-19?

- Question: is correct to view “defined”ness and “open”ness as orthogonal?



Ill-defined: Ambiguity

- Oftentimes parts of problems are ambiguous – both in the sense of the description and the problem itself.
- This could mean:
  - Something is unspecified/unknown.
  - Something is underspecified.
  - Something cannot be known.
- What are examples of all of these?
- This gives us all sorts of trouble – how can we have any confidence in our solution if it is not *well founded*?

Ill-defined: Faulty

A **faulty** problem is one where the prompt is misleading, contradictory, or simply incorrect.

For the following examples, are they fine/ambiguous/faulty?

- Example 1: Store a collection of contacts such that we can quickly access them by name as well as ID number.
- Example 2: Consider designing a program where you need to store a fixed size collection of items. What data structure would you use?
- Example 3: Create a linear time algorithm that: “Given a set of items, each with a weight and a value, determine[s] the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible”. (from Wikipedia)
- Example 4: Efficiently check if an image is black and white using a neural network.

Solving Real-World Problems

- Thinking about homework in your SER classes for a second, what do your answers look like? Just source code?
- To contrast: how are real-world solutions different?
  - Further: are real-world problems graded…?
- To be more specific: in homework we often jump straight to a ***design***: that is, a proposed solution that we have generated.
- Really, we are missing two extra steps: analysis, and justification.
  - ***Analysis***: sitting down and explicitly dealing with all the muddling of the problem and its description.
  - ***Justification***: arguing and supporting our proposed solution against alternatives.
    - A thought: we want to be right. All the time if possible!

Solving Real-World Problems

So, what happens when people go from solving educational problems to real-world ones? Well, it's complicated.

Consider for second: if you ask someone about the transition from school to work, one of the first things you'll here is: "I learned a lot!"

Basic problems:

- "Where do I start?"
- "What do I do next?"
- "Am I done?"
- Any others?

We can't solve this all at once but paying more attention to analysis and justification will help us greatly.

Introducing ADJ: Analysis, Design, and Justification

ADJ

- What is ADJ?
  - ADJ is a simple three step problem-solving process. (and those outcomes at the beginning!)
    - *It's just: analysis, design, and then justify. That's it.*
  - It serves as scaffolding for other tools and techniques that you learn.
- Why do something different?
  - Most class assignments are far away from real-world problems.
  - This is problematic since it means we aren't practicing all of the skills we really need.
    - No one can tell you how good you are at breaking down a problem unless they ask for more than a design!

Using ADJ ensures that we do our due diligence when solving a problem. Right now, it may not seem like a big deal, maybe even just cumbersome, but the goal is set you on trajectory of solving real problems.

ADJ

We now define the ADJ process as following these three stages in sequence:

- **Analysis (A)**: a translation of the original ill-defined problem into concrete requirements that can be evaluated.
- **Design (D)**: a solution to the problem being solved.
- **Justification (J)**: an argument which supports your design as being superior to other potential designs which also satisfy the analysis requirements.

- We wabt to show that a problem under analysis (A), soundly yields a design (D), such that the design is the/an optimal solution to the problem. The justification (J) is a sound argument that the design is the optimal solution for the problem given the analysis.
- The plan: we'll step through each of the stages, we'll talk about some useful (but minimal) techniques.

Example

- As the simplest possible example, let's take the following problem: "Consider designing a program where you need to store a fixed size collection of items and display them. Would you use an array or linked list? Justify your answer."
  - More educationally, we might attach "Analyze the problem, design a choice, and justify the choice." to the end of it.
- For now, we'll assume that the answer would be given as a write up.
- Here's a quick answer for it: "Linked list, because it will allow you to add or remove elements quickly." Any thoughts on this answer as good/bad?

Soundness

- Soundness means that the parts of our argument build upon each other logically.
  - Every piece of text should be linked conceptually to the text, otherwise it will introduce inconsistencies and lose soundness.
- The follow must be emphasized: **Answers must follow from problem statement.**
- *Consider designing a program where you need to store a fixed size collection of items and displaying them. Would you use an array or linked list? Justify your answer.* Consider some more sample answers:
  - "Linked list, because it will allow you to add or remove elements quickly." Is this sound?
  - "Array, because it will allow you to quickly access each of the elements." Is this sound?
  - "Linked list, because it will allow you to quickly access each of the elements." Is this sound?
  - "Array, because it will allow you to quickly display each entry." Is this sound?
  - "Linked list, because it will allow you to quickly display each entry." Is this sound?
- Random thought: should an answer have more or less raw text in it?

Process

- When doing engineering, we would expect to see some general flavor of process. Often people indicate process to explain how SE is different than CS.
  - There is another aspect as well: control. People don't like using this word (it sounds harsh), but I think it captures a difference between science and engineering.

  - Are we using a process here?
  - Why is using process good?
  - Why is using process bad?
  - What would justify the use of process for problem solving?

Analysis

Overview

Spoiler: this will be the most complicated section of ADJ that we talk about.

- Why?
- In Analysis, you should define your immediate problem, and any useful corollaries.
- The answer should be in terms of the problem itself, and not be biased in any towards a possible solution.
- **It is possible that some portions of your analysis will not be needed later** - this is expected since otherwise we do not have multiple inputs to a design process, meaning there might be only one class of possible choices.

Rough Steps:

1. Identify and address ambiguities in problem statement.
2. Generate definitions for ill-specified parts of the problem, along with useful corollaries.
3. Using the definitions to propose a more well-specified view of the problem.
4. Define metrics (more to come!)

Definitions

- Almost always there are ambiguities in your problems – fix them.
- Treat definitions as your guideposts. They tell you what is real.
  - If you find yourself asking: “Am I allow to do that?”, then look to the definitions to guide you.
- In general, this means we need to find places that are underspecified, and then detail them. For example:
  - if a problem says “display”, we might want to define more specifically “print each element in the collection exactly once in any order”.
  - If a problem says “efficiently”, we might want to define it as “run in a Big-Oh polynomial time on the number of operations”.

Fixing Ill-defined Problems

When faced with an ill-defined problem, there are three things we can think about:

- Making **reasonable** assumptions.
- Asking questions!
- **Performing experiments:** experimentation is a key tool with dealing with the unknown.
  - We want to take the scientific approach of evidence-based discovery.
  - Have we learned any experimental methods in this class?
    - *Are there any others you have seen that might be useful?*

Take the knowledge you gain from these and codify them in definitions which you can then build on.

If we do this properly, we won't waste time creating incorrect solutions.

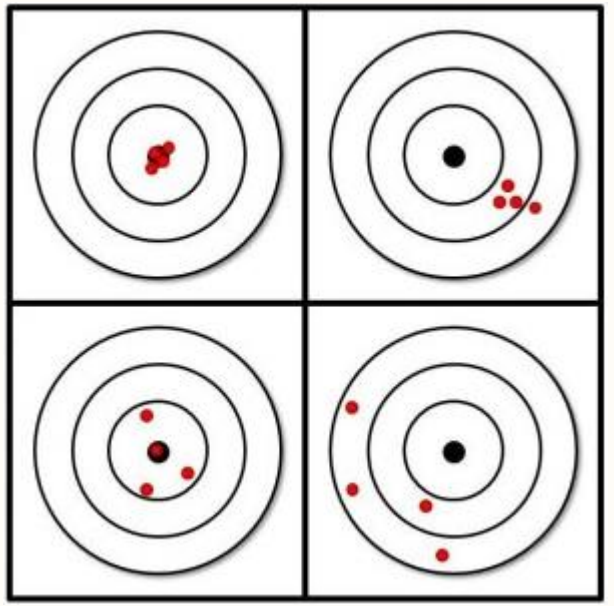
Asking Questions

- Asking questions is a big thing in industry!
  - Defining a problem is a back and forth! Not even the customer knows what they want! It's up to your team to gather enough information that you can take action and produce something useful.
  - Problems are too complex for someone to know everything about it.
  - Solutions are too varied for anybody to be able to create them all.
  - That said, someone somewhere has the knowledge or expertise you need. You just have to find them.

- When formulating a question to someone, help them help you:
  - Ask the appropriate person.
  - Make it specific.
  - Make it simple.
  - Make it actionable to you.

Metrics and Measurements

- Before we attempt to do any design work, we need to know what makes a solution good – we need to define metrics.
  - A metric will be a way for us to measure different solutions to the problem and compare them.
- Ideally, we would want:
  - Metrics that are useful relative to our problem statement.
  - Metrics that are reliable: high in accuracy and precision.
  - More practically: efficient metrics
- Have we learned any metrics in this class?
  - Are there any others you have seen that might be useful?



http://glsi.agron.iastate.edu/2015/01/18/accuracy-vs-precision/

Design

Overview

- This section should build on the results of the analysis approach to construct a solution which meets a metric (defined in analysis) by which a design may be judged to provide a good answer.
  - Design is subjective - in and of itself, we cannot state that a particular design is best without a metric.
- The design can look many ways:
  - For the simplest problems, it might be just be a selection: “array” or “list”.
  - For some problems, design will take the form of designing an explanation (later backed with an argument which states a position on the validity of the “claim”).
  - In others, it will literally mean designing an algorithm.
- BTW, what classes have you learned design skills in?

Justification

Overview

- Given that we have a design, we want to argue that this design is:
  a) well-founded, sound, and solves the problem, and
  b) is the optimal design for the problem.
- A *justification* is an easily verifiable piece of information (here: readable by a 3rd party) that shows that the claim for the design is correct.

- Rough Steps:
  1. Evaluate your metrics and show that they either are optimal or hold.
     - *Sometimes they need to be evaluated only for your solution and sometimes for alternatives as well (when you need to argue against them).*
  2. Done!*

* Provided that your analysis itself is sound.