SENG301: Practical Work

Lab 1 2022

Overview

Many of the concepts we'll be looking at have some “soft” aspects. Applying them in realistic contexts highlights issues such as whether a particular principle applies and whether different principles are giving us conflicting advice. It is important not only to have a good understanding of the theory behind these principles, but also to be able to recognise opportunities to apply them in practice.

Our practical work exercises are intended to provide you with opportunities to identify examples of particular principles (and their violations) in the wild, to form and express professional opinions about them, and to justify your opinions with references to the software engineering literature. Some will focus primarily on design, some on code, and others on the combination.

Reference system

In order to provide some context to help us achieve this, you should choose some Java code you have written, contributed to, or are otherwise familiar with — this will be your *reference system*. The codebase doesn't have to be huge but should have sufficient elements that we're likely to find things of interest.

Many of you will have code from SENG202, SENG302, hobby projects etc. that will be suitable. If not, then there are many places (GitHub, SourceForge, …) where open-source projects can be found. Please talk to your tutor if you aren’t sure what to do.

How (and why) to lab

The most important thing about practical work is actually turning up and attempting it! History tells us that skipping labs or lectures (or “deferring them”) is a predictor of sub-optimal performance. Assignments and exam questions are likely to be based on material covered in labs so it’s important not to fall behind.

On a more positive note, we strongly encourage you to collaborate with others in the practical work sessions. As well as helping with learning, it’s also a great way to get to know more people in the class. Being able to express, justify and modify your opinions about design issues is an important skill for professional software engineers.

As always, we have an excellent group of tutors this year. They are approachable and keen to help so please engage with them so they can share their expertise with you.

The exercises we'll cover over the next few weeks are intended as starting points for discussion, exploration and experimentation. Please feel free to attempt them in any order. There may be more exercises than you have time to complete in the lab. If you are already confident on some topics then it's good to challenge yourself with something new while your tutors are around to help you.

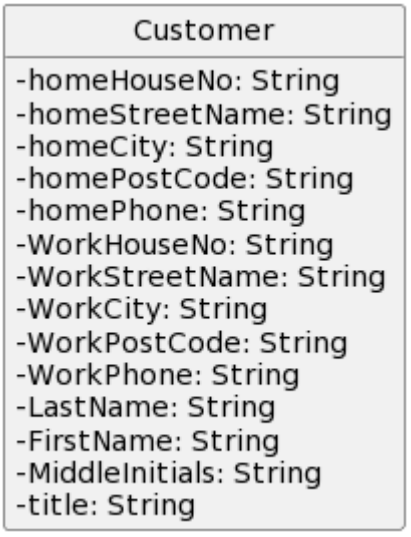
Outcomes

This year, students are not required to submit formal practical work reports. However, as you work through the practical work sessions this term, please make sure that you record appropriately the outcomes of your work. For example, you may have sketched UML diagrams to describe proposed design modifications, recorded design pattern occurrences by annotating source code, or used a revision control system to store before-and-after versions of code elements.

This will not only help you to discuss exercises with course staff but will also be helpful as you study for the exam. It is common for exam questions to be based on material students are expected to be familiar with from practical work exercises.

Design principles

1. Getters (also referred to as accessors) feature in a number of design principles including "Tell, don't ask."
    a. How can you quickly identify methods which might be getters? Use your knowledge of your reference system to find some occurrences of getters in your code.
    b. Can you use these to help you find examples where "Tell, don't ask" is violated? If so, how could the code be refactored to remove the problem? Can you find examples where "Tell, don't ask" is used correctly? Note these examples for future reference.
2. Consider this proposed design for the customers of a business.
    a. Do you think this is a satisfactory design? Do you foresee any difficulties in implementing and using it in future? Why?
    b. What design principles are relevant here? What changes would you suggest after considering these principles?
    c. Sketch a UML class diagram representing your suggested changes (if any).

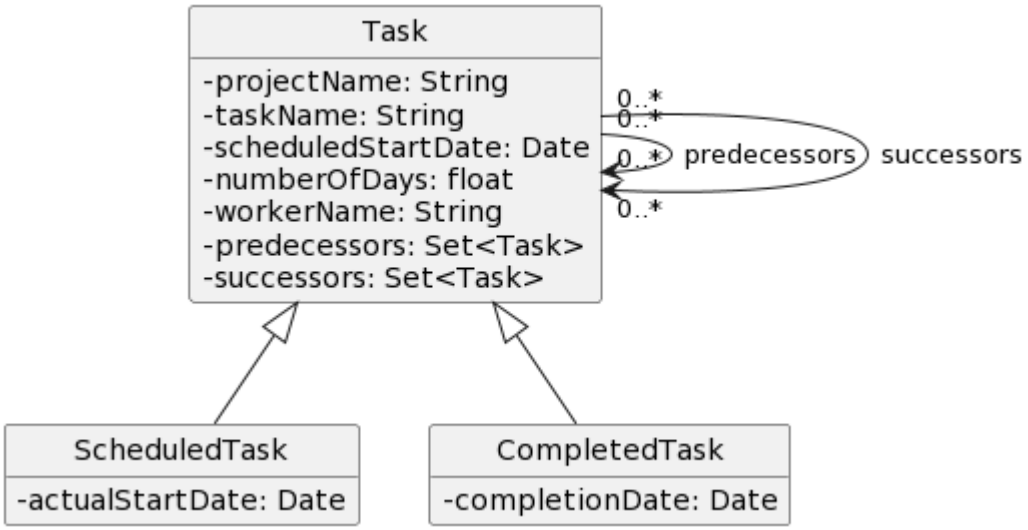


3. Ron is planning to build a simple application to teach children about finance. The only unit of currency is the (digital) coin. Each user will have a wallet (also digital) to keep their coins in.

Credit/debit operations allow coins to be added/removed from a wallet. Some users will be allowed to go into overdraft (i.e. after a debit operation the number of coins in their wallets might be negative). All other wallets must have zero or more coins in them at all times. Ron begins by designing and implementing simple Java code for wallets—including credit & debit operations as well as overdrafts.

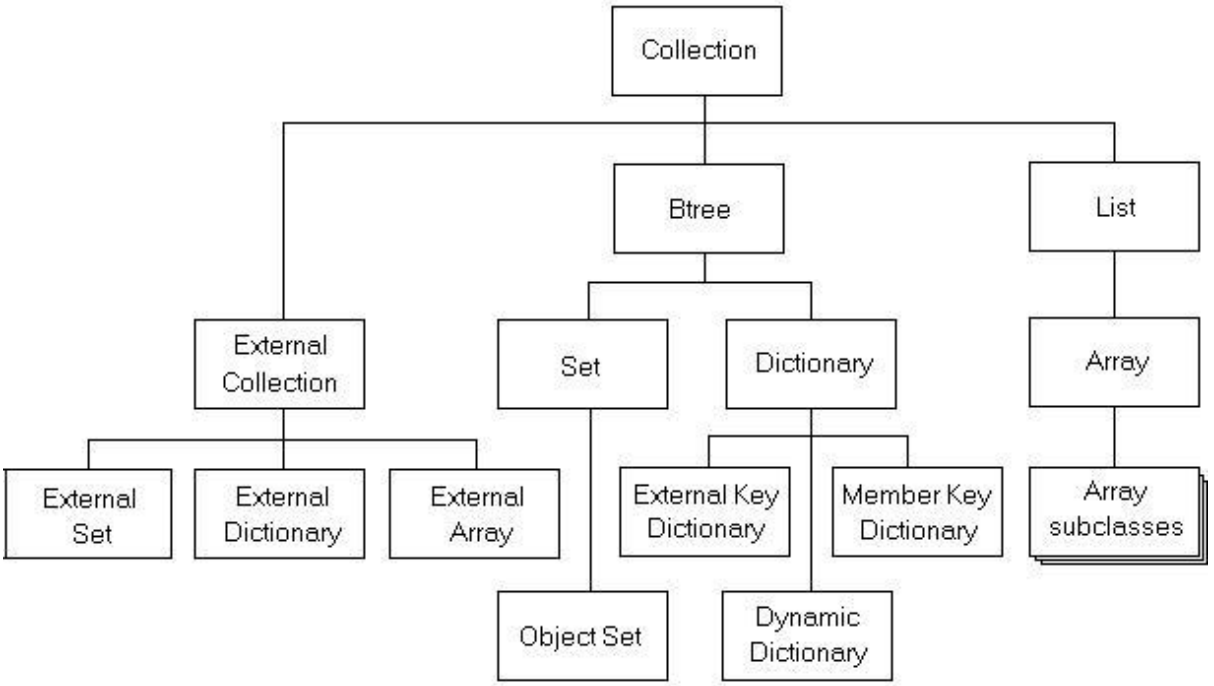
a. Ron wants to impress Hermione by showing her this code, but knows that she'll want to see it working. What would be some suitable ways for him to demo his code without having to finish writing the rest of the application? Which would you prefer yourself?
b. Ron makes his choice and demonstrates his code going through its paces. Everything seems to work perfectly but he is disappointed by Hermione's reaction. She says "You've violated the *tell, don't ask* principle—I'll refactor it for you," grabs his keyboard, and in a few minutes triumphantly announces that she's finished the task.
    i. Design and implement the digital wallet the way Ron did. Make sure you (using your answer to part a.) include a way to demonstrate it.
    ii. Now develop a second version that would make Hermione happy.
    iii. Compare the two—what, if any, advantages might Hermione's "tell" version have?

4. You've been asked to review and complete the proposed design for a project scheduling system. The system must model Tasks for projects, and record Task dependencies (i.e. which Tasks must come before others). Each task can have any number of workers assigned to it.



a. Is the existing design (the part shown above) OK? Why or why not?
b. What design principles apply? How might you apply them to improve the data model?
c. Document your proposed solution with a UML class diagram and/or Java code. Do you foresee any practical difficulties implementing your design in Java?

5. Look for instances of inheritance in your reference system. You might start with a class diagram, if you have one, or by searching for “extends” and “implements” in the code. How can you tell if the LSP has been followed? Has it? What if any improvements would you suggest?
6. In previous courses you have looked at collections — including those Java provides.



The figure above shows the collections hierarchy of another OO language, A-lang[1]. A-lang's collection design is roughly similar to Java's, but A-lang's collections include features such as automatic filtering and maintenance of inverse relationships. Note that “Dictionary” is a synonym for “Map”.

Ignoring, for now, Array classes, can you identify a design principle (or principles) NOT followed by this design? What are the implications of ignoring them? How could the design be changed to accommodate the principles?

Note that “Array subclasses” is not a real class, it indicates that a subclass exists for each kind of array: BooleanArray, CharacterArray, DateArray, … ObjectArray, etc. Unlike Java, A-lang arrays are integrated into the collections hierarchy. Is this a good idea? What is the difference between a List and an Array (make reasonable assumptions about what operations List and Array might support)? Does it make sense for Array to extend List? If not, how should it be incorporated into the collections hierarchy?

“External” collections acquire data from outside the system, e.g. from a relational database. Note that ExternalSet, ExternalDictionary and ExternalArray are similar to

[1] Actual name may/does differ

Set, Dictionary and Array, but do not allow updating of their contents. Is this a good design? Can you formulate arguments to back up your position?

The names and short descriptions of all methods provided by Collection are shown in the table below. Making reasonable assumptions about what the methods do, comment on the design. Do any methods surprise you? Can you apply design principles to support your arguments? Can you suggest improvements?

Method	Description
add	Adds an object to a collection
clear	Removes all entries from a collection
copy	Copies entries from the receiver to a compatible collection
countOf	Returns the number of times the specified entry occurs in the collection
createIterator	Creates an iterator for the Collection class and subclasses
deleteEmpty	Deletes a shared or exclusive collection if it is empty
first	Returns the first entry in the collection
getOwner	Returns the object that is the owner of the collection
getStatistics	Analyzes the collection and returns structural statistics
includes	Returns true if the collection contains a specified object
indexNear	Returns an approximate index of an object in a collection
indexOf	Returns the index of a specified entry if it exists in the collection
inspect	Inspects a collection of objects
inspectModal	Opens a modal inspector window for the receiver object
instantiate	In exclusive collections only, ensures that an object is created before it is used
isEmpty	Returns true if the collection has no entries
last	Returns the last entry in the collection
maxSize	Returns the maximum number of entries that the collection can contain
purge	Deletes all object references in a collection
remove	Removes an item from a collection
setBlockSize	Specifies or changes the block size of the receiver
size	Returns the current number of entries in the collection

7. Select some significant classes in your reference system, and choose some of the design principles covered in class but not already in this lab. Perform a design audit. If you find possible violations of design principles, what refactoring/redesign would fix the problems? Pay particular attention to cases where there are competing solutions.