Foundations of Software Engineering

Summer I 2024

Staff

Jan
(Instructor)

Pierre
(Instructor)

Revanth
Java

Anikesh
TS

Ji-min
TS

Shubh
TS, Java

Sanjana
TS, Java

Karan
TS, Java

Mehul
TS

Vihar
TS, Java

Mansi
C++, Java

Ngoc
TS, Java

Sindhu
TS, Java

Nishy
TS, Java

Contact

All course communication is to be done through private notes on Piazza.

In case of emergency, the following email is shared with all course staff: Email.

Times

Lectures: MTWR, 9:50 am - 11:30 am Location: Zoom

Office hours: MTWR, 11:30 am - 12:30 am Location: Zoom

Schedule and deliverables

M	05/06	Course Orientation Software Development Processes	
T	05/07	Course Project Overview Requirements Engineering	
W	05/08	Test-Driven Development Agile Planning and Estimation	
R	05/09	Teams and Code Walks Code-level Design Principles	Sprint0
M	05/13	Object-Oriented Design What makes a good test suite?	
T	05/14	Interaction-level Design Patterns	
W	05/15	**No class**	
R	05/16	Testing effectful code Beyond unit testing	Sprint1
M	05/20	Refactoring and Technical Debt	
T	05/21	Code walks	
W	05/22	**Midterm (9am - 3pm)**	
R	05/23	**Midterm (9am - 3pm)**	Sprint2

M	05/27	**No class**	
T	05/28	Distributed Systems REST APIs	
W	05/29	Collaborative design and implementation	
R	05/30	Continuous Development Processes	Sprint3
M	06/03	Collaborative design and implementation	
T	06/04	Collaborative design and implementation	
W	06/05	Collaborative design and implementation	
R	06/06	Security & Software Engineering	Sprint4
M	06/10	Engineering Software for Equity	
T	06/11	Open source	
W	0612	Collaborative design and implementation	
R	06/13	Collaborative design and implementation	Projects due
M	06/17	**Project presentations (9am - 3pm)**	
T	06//18	**Project presentations (9am - 3pm)**	

Teams

The supported languages for the project are Java, TypeScript, C++, Dart. Note that we have limited capacity for C++ and Dart. Other strongly typed languages such as Rust or Haskell may be supported, ask the instructors. Untyped languages such as JavaScript, Python, Lua or R are not supported.

Midterm

The midterm evaluates your understanding of course material in a "job interview" format. These notes will help you prepare for the midterm and improve your next job interview. Interviews are technical conversations in which the interviewer evaluates your technical depth and fit for the given team/company. Interviews have relatively predictable formats. You can prepare and practice to improve your performance. My four Bees:

1. **Be Relaxed**. Fear is a bad co-pilot. Speak clearly and slowly. Pause every minute or so. Look at the interviewer for signs you are off-track.
2. **Be brief.** Short answers are better than long ones. Always.
3. **Be precise.** Do not give vague answers. Ban fluff words. Use the right technical terms.
4. **Be humble.** If you don't know, say it and stop talking.

Last year, I interviewed candidates interested in working at a startup. Folks with 5+ years of SE experience who performed poorly. One thing to understand is that your interviewer is an engineer with another job, interviews are interruptions in their schedule that is why they are strictly time-boxed. When time runs out, the interviewer is eager to return to their work. My usual routine was to start with boilerplate to get the candidate comfortable then go through a series of increasingly open-ended technical questions. Here are some failure modes:

- Sea-of-words. Questions have simple short answers. A flood of words is often correlated with a lack of understanding.
  Mitigation: Think before starting your answer. Pause to make sure the interviewer is with you. Have a glass of water, drinking helps slow down. Stop when you start to be unsure of what you are saying.
- Langweiligkeitsverursachend: Speak for too long on one question, well past exhaustion of the topic. Some candidates belabor the obvious, speaking for minutes on things that need seconds. This eats time they could have spent on the following questions.
  Mitigation: try to be concise, if the interviewer needs more detail, they will ask.
- Babble-stan: When speaking of technical topics, use the correct terms. Misuse of vocabulary is a subtle indication you do not know what you are speaking of.
  Mitigation: learn the vocabulary. If you must speak of what you don't know, acknowledge it (without apologizing).
- Drowning in the weeds: Trying to write an entire algorithm and get tangled up in the details.
  Mitigation: There is little time for coding. Rather than wasting time on low-level algorithmics, define abstract functions and only detail the ones the interviewer asks you to.
- Silent running: Sometimes one gets stuck and can't make progress. This happens, but don't do it silently.
  Mitigation: Speak through your problem solving process. Verbalize what you are thinking. If the interviewer hears that you are going down the wrong path, they may choose to help you out.

For the boilerplate questions, prepare and rehearse answers. You should be able to introduce yourself and say why you are interested in the position in 30 seconds. You should have some anecdotes about your past work rehearsed and ready to go. They should be short and have a point. Self-deprecation is fine but in small doses. Ambition is also good, but tempered with realism. Always look at the interview and take cues from them. They mean well and will try to help when you leave them the chance.

For the midterm, we can ask you to introduce yourself and why you want to work on the Husksheets project. We can ask you questions that test your understanding of process, requirements, and planning in the abstract, as for design, refactoring, testing we will ask

questions that are grounded in code fragments. You will not have to write code, but rather talk about code. Remember, you have a fixed time budget for a number of questions, speaking too long on one question may cause you to not be able to answer the last (few) questions.

If any of the concepts discussed in the slides is unclear, well we are in the ChatGPT age, ask your friend. Or even better read some of the linked resources. We are always happy to answer questions. Topics include Process (waterfall vs agile, risk management, XP, scrum), Requirements (user stories, functional vs. non-functional), Planning (backlogs, sprints, tasks), Test driven development (deriving tests from user stories), Teams (approach, reviews), Testing (picking test inputs, black- v white-box, coverage, oracles, effects, doubles, beyond), Design patterns (motivation, factory, singleton, adaptor, visitor), Refactoring.

Project Presentation

Project presentations are held June 17 and 18. A signup sheet will be published here.

The presentation is done in groups.

Husksheets

Overview

The course project is a *distributed collaborative spreadsheet* application called **Husksheets.** It consists of (1) a *server* with a persistent store, (2) a *client* able to create and open spreadsheets, and (3) a *user interface* that displays sheets and allows editing them.

You are to design and implement Husksheets following best software engineering practices.

It is up to each team to decide what practices from the lecture are helpful.

What we ask you is to **document** your practices, and **argue** why and how your team used them.

Deliverables

The following are required

- Team code repository [Deadline Sprint0]
- Team notebook repository [Deadline: Sprint0]
- Notebook entries for each sprint (group) and each work session (individual)

- Statement of Work [Deadline: Sprint0]
- Final Product [Deadline: June 15]
- Demo [Deadline: June 17]

Code Repository

The repository is to be hosted by https://github.com. Set it to Private and add course staff as collaborators.

The repository should include the following:

- A **design** directory with all design documents
- A **Readme.rmd** with information describing the status of the project
- A **Makefile** with targets
  - test to run the tests
  - build to compile
  - docker to build a version of the project in Docker and run the tests

  Note that we insist on a Makefile (even if you project uses other build technology, you can simply call the other build tool from the Makefile)
- A **src** directory with the code of your project

Notebook Repository

The repository is to be hosted by https://github.com. Set it to Private and add course staff as collaborators.

The repository should contain one .Rmd file per team member and one .Rmd file for group meetings.

Notebooks should have this format (these are examples):

```
# Entry
12
## Start time
04/14/24 13:45
## Purpose
I need to fix the code that implements the file selection user interface. For this I need to learn
more about JavaFX
## Commit(s)
https://github.com/janvitek/JuliaGenerable/commit/846489905f0543b2307e586bda421f4629090f58?diff=sp
lit&w=1
## Ownership
+134 -23 =665
## Outcomes
Successfully added tests and implementation. Fixed a but that caused the UI to not display the
last element in a list, added a test for that case. Committed to the repo and removed the feature
flags.
## End time
04/14/24 15:00

# Entry
11
```

```
## Start time
04/14/23 11:50
## Purpose
Learn JavaFX -- the library that will be used in the graphical user interface of the project.
## Commit(s)
none
## Ownership
=465
## Outcomes
Read the JavaFX documentation and experimented with some code examples provided by ChatGPT. A lot
of small details but overall relatively straightforward. The visual style of the GUI is not
particularly pretty, but it will do for a MVP.
## End time
04/14/23 13:00

# Entry
10
## Start time
04/14/23 10:00
## Purpose
Group meeting with Pierre and Jane. Frank was absent. Planning for the week's sprint.
## Commit(s)
none
## Ownership
=465
## Outcomes
We agreed to split the work with me and Pierre pair programming the GUI, Jane will focus on
refactoring and Frank will continue working on the server. Next meeting Monday after class. We
will do a code review then.
## End time
04/14/23 13:00
```

Each entry has start and end times, a **Purpose** section where you describe what you plan to do, or, if this is a group meeting, summarizes attendance and goals of the meeting, a **Commit(s)** section with links to any commits that you made, an **Ownership** which lists the lines of code added and removed by the commits as well as the running total of code that contributed, and an **Outcome** section that summarizes what was achieved.

The level of details may vary. But do not forget to record your work. And **always push** the notebook to the repository as soon as you are done with an entry!

Statement of Work

Before starting to code, each team should write a SOW that describes what they expect to implement in terms of MVP (minimal viable product), desirable features and additional features. You will be graded based on meeting that SOW. Use best-practices to document the SOW.

Final Product

The tag of the code repository when you are done with development.

Please take into account that we emphasize code quality in this class. This means: clean code, frequently refactored, documented and tested. This is just as important as functionality -- better

have good code that does a little, than bad code that does a lot. Undocumented code will not be reviewed. Untested code will not be reviewed.

As a rough guideline for size, aim to contribute 1000 lines of quality code as measured by additions minus deletions on GitHub. This is a soft goal as it is more important that the code be of good quality (i.e. one could add one thousand empty lines or lines of comments it would meet the letter of the law but not its spirit -- we grade on the spirit). Furthermore, each code unit (file, class, method) should have a comment indicating its owner. We expect that the owner is the one committing changes to their code. Others may ask for changes or make pull requests with proposed changes, the final approval is with the owner.

Demo

A demonstration of the project and a code walk explaining your contributions.

Server Specification V1

The Husksheet Server accepts REST API requests from clients. The following endpoints are supported by the current version of the specification:

- `Result register()`
- `Result getPublishers()`
- `Result createSheet(Argument)`
- `Result getSheets(Argument)`
- `Result deleteSheet(Argument)`
- `Result getUpdatesForSubscription(Argument)`
- `Result getUpdatesForPublished(Argument)`
- `Result updatePublished(Argument)`
- `Result updateSubscription(Argument)`

Where Result is a JSON object returned by the REST call, and Argument is an object provided in the body of the request.

The format of both objects are:

```
Result {
  boolean  success
  String message
  List<Argument> value
}
Argument {
  String publisher, sheet, id, payload
}
```

The meaning of the various fields are: if `success` is true, the result object has a `value`, else the `message` holds the reason for the failure. The `publisher` is the name of a registered client, a

`sheet` is the name of a sheet belonging to a publisher, an `id` identifies an update to a sheet, and a `payload` contains the data for an update.

The server uses *Basic authentication*: you should send a header with key Authorization and the value `Basic auth`, where `auth` is `username:password` encoded with base 64. The set of client names and passwords is pre-assigned. Status code 401 is returned when the request is not authorized. HTTPS is used.

All endpoints may report an authentication failure, otherwise they should yield a result object. The first two are GETs, the others are POSTs.

All endpoints start with `/api/v1/`, so, to connect to endpoint `getPublishers` on host `localhost` on port `9443` in https, one would use `https://localhost:9443/api/v1/getPublishers`.

Endpoints

`register` causes the server to create a publisher with the client name. No value is returned.

`getPublishers` returns a list of argument objects with the `publisher` field set to all registered publishers.

`getSheets` takes an argument object with field `publisher` set to the name of a publisher and returns a list of argument objects with the `publisher` and sheet fields set to all sheet names for the given publisher.

`createSheet` takes an argument object with fields `publisher` and `sheet` set to the name of the client and the name of a sheet to create. No value is returned.

`deleteSheet` takes an argument object with fields `publisher` and `sheet` set to the name of the client and the name of a sheet to delete. No value is returned.

`getUpdatesForSubscription` takes an argument object with fields `publisher`, `sheet` and `id` set to the name of a publisher, a sheet, and an id. It returns an argument object with the `payload` set to all updates that occurred after `id`, and the `id` field set to the last `id` for those updates. The sheet is owned by a publisher different from the client. An empty `payload` is returned if no updates occurred after the given id. The initial id is "0".

`getUpdatesForPublished` takes an argument object with fields `publisher`, `sheet` and `id` set to the name of a publisher, a sheet, and an id. It returns an argument object with the payload set to all the requests for updates that occurred after id, and the id field set to the last id for those requests for updates. The sheet is owned by the client. An empty payload is returned if no updates occurred after the given id. The initial id is "0".

`updatePublished` takes an argument object with fields `publisher`, `sheet` and `payload` set to the name of a publisher, a sheet, and updates for that sheet. No value is returned. The sheet is owned by the client.

`updateSubscription` takes an argument object with fields `publisher`, `sheet` and `payload` set to the name of a publisher, a sheet, and requests for updates for that sheet. No value is returned. The sheet is owned by a publisher different from the client.

Sheet Update

A sheet is a set of cells indexed by references. Cells can have numeric or character values, or can hold formulas which determine how the value of the cell is computed. The largest number of columns and rows that is expected is in the thousands.

A **`Ref`** (or reference) is a string like `$A1` that consists of three parts, `$ . A . 1`, the first part is a mandatory dollar sign, it is followed by a column identifier and a row identifier. Column identifiers are, case-insensitive, sequences of alphabetic letters that denote a column. Thus `A` denotes column 1, and `AB` denotes column 28 (1×26^1+2×26^0). Row identifiers are non-zero positive integers.

An **`Update`** is a newline separated sequence of (Ref,Term) pairs, where a Term can be either a value or a formula. For example:

```
$A1 1
$a2 "help"
$B1 -1.01
$C4 ""
$c1 = SUM($A1:$B1)
```

A **`Term`** is either one of a number (signed, floating point), a double-quote-delimited string (possibly with \" escapes), or a formula.

A **`Formula`** starts with an = and is followed by an expression.

The grammar for expressions is:

```
E ::=  E Op E | '('  E ')' |  Fun '(' [E [',' E]* ] ')' |
['+'|'-']<int> | <string> | Ref
Op ::= '+' | '-' | '*' | '/' | '<' | '>' | '=' | '<>' | '&' | '|' | ':'
Fun ::= 'IF' | 'SUM' | 'MIN' | 'AVG' | 'MAX' | 'CONCAT' | 'DEBUG'
```

In the above square braces denote optional elements. Precedence is unspecified, parenthesis should be used for disambiguation.

The meaning of a Ref is the value of the corresponding cell, if the cell contains a formula that value is the result of evaluating the formula, if the cell has not been assigned a value then nothing is returned.

The meaning of a range $A1:$B2 is the sequence of values of the cells in that range. Cells without a value are ignored.

The meaning of operations is as follows:

x + y, returns the sum of x and y if they are numbers, error otherwise
x - y, returns the subtraction of x and y if they are numbers, error otherwise
x * y, returns the multiplication of x and y if they are numbers, error otherwise
x / y, returns the division of x by y if they are numbers and y is not zero, error otherwise
x < y, returns 1 if x is smaller than y and 0 if they are other numbers, error otherwise
x > y, returns 1 if x is larger than y and if they are other numbers, error otherwise
x = y, if x and y are numbers, returns 1 x is y, 0 otherwise; if they are both strings, return 1 if they are equal else 0; error otherwise
x <> y, if x and y are numbers, returns 0 x is y, 1 otherwise; if they are both strings, return 0 if they are equal else 1; error otherwise
x & y, if x and y are numbers, returns 1 if x and y are not 0, and 0 otherwise; error if either of them is a string
x | y, if x and y are numbers, returns 1 if x or y is 1, and 0 otherwise; error if either of them is a string
x : y, if x and y are Ref and x <= y, return the range of cells denoted; otherwise error.

The meaning of functions is as follows:

IF(e1, e2, e3) if e1 is not zero return the value of e2, if it is zero return e3's value; error if e1 is not a number
SUM(e1 ... en) if all values are numbers then return their sum; otherwise error
MIN(e1 ... en) if all values are numbers then return the smallest value; otherwise error
MAX(e1 ... en) if all values are numbers then return the largest value; otherwise error
AVG(e1 ... en) if all values are numbers then return their average; otherwise error
CONCAT(e1 ... en) coerce all values to strings and concatenate them
DEBUG(e) return the value of e