Midterm Examination Solutions — June 2023

CSC207H1Y — Software Design

Robert Wu
rupert.wu@utoronto.ca

Department of Computer Science
University of Toronto

June 21, 2023

Question 1. Version Control [11 MARKS]

(a) [2 MARKS] Briefly give two reasons for using version control like `git`.
Any two of the following are acceptable: backup/restore, synchronization, short-term undo, long-term undo, track changes, sandboxing, branching/merging.

(b) [2 MARKS] What is a non-trivial merge conflict between two branches? Be precise.
Branches diverged (conflicting commits) (1.0) and their subsequent commits modified the same files (1.0).

(c) [2 MARKS] Name two common ways that merge conflicts are resolved.
Any two of the three are acceptable.

- Local merge/rebase to the feature branch and push.
- Local merge/rebase the `main` branch and push.
- Merge request (i.e. GitHub pull request).

(d) [5 MARKS] After cloning, `Main.java` is modified to support the French language. Use a typical `git` workflow in command-line to update only `Main.java` in the remote repository. Include a command to check tracked/staged file changes before staging. You're not required to use every line of space.

```
# suppose repository has been cloned
$ ls
src test README.md
# files including Main.java have been modified.
$ git status
$ git add src/Main.java
$ git commit -m "added French language support"
$ git push
```

- 1.0 for `git status`.
- 1.0 for `git add` and an additional 0.5 for correct path `src/Main.java`.
- 1.0 for `git commit` and additional 0.5 for meaningful commit message.
- 1.0 for `git push`.

Question 2. Basics of Java [14 MARKS]

(a) [3 MARKS] What's different between compiled and interpreted languages? Give an example of each.
An interpreter (like Python) translates/executes code one statement at a time, while a compiler (like C) translates the entire program once and executes any number of times.

- 2.0 for the explanation.
- 0.5 for each example.

(b) [1 MARK] How would you describe Java on this spectrum? What does it run on?
Hybrid/in-between (0.5) on the Java Virtual Machine (JVM) (0.5).

(c) [1 MARK] What's the difference between a function and a method?
Methods belong to classes, functions don't.

(d) [4 MARKS] Declare and initialize a `float f`, `boolean b`, `long l`, and `char c` in ascending order of their memory sizes (i.e. how many bits to represent each type) with any valid values. Use the code block on the right.

```
boolean b = false;
   char c = '8';
  float f = 32.0f;
   long l = 64; // L is optional
```

- 0.5 for each correctly initialized.
- 0.5 for each correct position.

(e) [3 MARKS] Program `Bill` doesn't compile. Point out **three** *different* potential compile errors.

```
class Bill {
    private double rate = 0.13;
    private double tax(double principle) {
        return (1 + rate) * principle;
    }
    public static void main(String[] args) {
        float fee = 5.78;
        float fee = tax(fee);
    }
}
```

There might be more, but accept any three of the following:

1. variable `fee` is already defined
2. lossy conversion from `double` to `float`
3. non-static method `tax(float)` cannot be referenced in a static context
4. `fee`'s initial value 5.78 doesn't end with `f`

Wording need not be exact.

(f) [2 MARKS] Briefly give one reason for and one reason against using `float` instead `double`.
Smaller memory footprint but less precision.

Question 3. Object-Oriented Programming (OOP) [13 MARKS]

Class `Constant` (right) always returns a. We want to write a program to model linear functions $f(x) = bx + a$ and quadratic functions $g(x) = cx^2 + bx + a$, each with its own `compute()` method. Look closely at what they have in common.

```
class Constant {
    private double a;
    public Constant(double a) { this.a = a; }
    public double compute(double x) { return a; }
}
```

(a) [6 MARKS] Use chained inheritance to write `Linear` and `Quadratic` while reusing code when possible. Follow best practices when overriding methods and shadowing variables.

```
class Linear extends Constant { // 0.5
    private double b; // 0.5
    public Linear(double a, double b) {
        super(a);
        this.b = b;
    } // 1.0
    public double compute(double x) {
        return b * x + super.compute(x);
    } // 1.0
}
```

```
class Quadratic extends Linear { // 0.5
    private double c; // 0.5
    public Quadratic(double a, double b, double c) {
        super(a, b);
        this.c = c;
    } // 1.0
    public double compute(double x) {
        return c * x * x + super.compute(x);
    } // 1.0
}
```

(b) [2 MARKS] So far, functions descend from `Constant` but really, we only need a type that provides the `compute(double x)` behaviour. Write such a type `OneVar` and adapt `Constant`'s header to it.

```
interface OneVar { // cannot be a class
    // public is optional, but no other modifiers allowed
    public double compute(double x);
} // worth 1.0
```

```
class Constant implements OneVar {
    // give 0 if they modified the
        class definition
} // worth 1.0
```

(c) [5 MARKS] Construct a function of each of the above classes given `a`, `b`, `c`. Load them into a single `ArrayList` that stores any arbitrary functions. Finally, compute and print their their values at x.

```
import java.util.ArrayList;
public class PolyInheritance {
    private static double a = 1.73, b = 0.2, c = -2.88, x = 82.3;
    public static void main(String args[]) {
        ArrayList<OneVar> functions = new ArrayList<OneVar>();
        functions.add(new Constant(a));
        functions.add(new Linear(b, a));
        functions.add(new Quadratic(c, b, a));
        for (OneVar func : functions) System.out.println(func.compute(x));
    }
}
```

- 1.0 for a mostly right `ArrayList` initialization and 0.5 for using type `OneVar` in the `ArrayList`.
- 0.5 for adding each function.
- 1.0 for the `for` loop with correct `OneVar` type (if they used iterator).
- 1.0 for the correct `compute()` and `println()` calls in the loop.

Question 4. Exceptions & Errors [10 MARKS]

(a) [3 MARKS] Explain in detail the difference between a `RuntimeException` and a regular `Exception`.
A regular `Exception` must be declared in methods that throw them, and declared or caught/handled in code that calls to methods that throw. A `RuntimeException` can but doesn't have to be declared or caught. You should never catch an `Error`.

(b) [5 MARKS] Wrap the following file-opening code in an exception handling structure.

```
String filePath = "/home/rhubarb/file.md";
BufferedReader reader = new BufferedReader(new FileReader(filePath)); // FileNotFoundException?
reader.close() // IOException?
```

Consider also,

- Any exception `e` should be handled with `e.printStackTrace()`.
- `reader` is not guaranteed to be non-`null` at all times.
- Treat `reader.close()` as tear-down/clean-up code.
- Exception handling blocks can be nested.

```
String filePath = "/home/rhubarb/file.md";
// start here
BufferedReader reader;
try {
    reader = new BufferedReader(new FileReader(filePath));
} catch (FileNotFoundException e) {
    e.printStackTrace();
} finally {
    if (reader != null) {
        try {
            reader.close()
        } catch {IOException e} {
            e.printStackTrace();
        }
    }
}
```

- 1.0 for each of the outer `try` and `catch` blocks.
- 0.5 for wrapping the closing code in the outer `finally`.
- 0.5 for gating the `reader.close()` with `if (reader != null)`.
- 1.0 for the inner `try` and `catch` with the aforementioned gate.
- 1.0 for keeping the `BufferedReader reader` declaration outside of `try`.

(c) [2 MARKS] If the above code were in a method, and you don't want to handle these exceptions here, **briefly** describe how you would modify your code. Be sure to use the exact Java keywords.
Declare the exceptions with `throws IOException, FileNotFoundException` (1.0 each).

Question 5. Design Principles [8 MARKS]

(a) [2 MARKS] What is persistence? Give two examples of mechanisms.
Persistence is data/information that is stored over time (1.0). Examples could include RAM, disk, databases but accept anything that seems reasonable (0.5 each).

(b) [2 MARKS] Name the five design principles of SOLID.
Single-Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, Dependency Inversion.

- 0.5 for naming two and 0.5 for each additional principle.

(c) [2 MARKS] Explain the "D" and one of its benefits in two sentences.
Higher level modules should depend on abstractions that are compatible with low-level modules (1.0). This allows modules to independently change while the abstractions in-between are invariant (1.0).

(d) [2 MARKS] Name two typical violations of CA.
Any (examples of) two of the following will do:

- Dependencies skipping layers.
- Backward dependencies.
- A class containing code from multiple layers.

Question 6. Design Processes I [8 MARKS]

Consider an airline management system that tracks passengers and flights. Passengers have personal information, customer records (including loyalty points), and booking information with respect to flights. For flights, the system records information about aircraft, schedules, prices, luggage destination/storage locations, pilots and staff. When a customer wants to purchase a ticket, the airline should be able to calculate the fare based on flight details and customer points. When delaying or cancelling flights, the airline should be able to look up flight details, make modifications, and notify customers.

(a) [3 MARKS] Identify six likely entity classes. 0.5 for each.

(b) [1 MARK] Identify the two obvious use-case classes. 0.5 for each.

(c) [4 MARKS] Draw a CRC card for each use-case class, including relationships with all applicable entity classes you identified. 2.0 for each CRC card. This question is somewhat open-ended and part of the design process. It doesn't have to be perfect, so accept any reasonable solutions! This is a sample:

TicketSale

- Retrieve customer information, including loyalty points.
- Retrieve flight details.
- Calculate and report fare cost.
- Record the new flight ticket.

Collaborators: customer, passenger, flight, destination, airline.

FlightModification

- Identify flight and retrieve details.
- Modify flight details.
- Search for all tickets linking to passengers.
- Notify passengers and customers.

Collaborators: customer, passenger, flight, airline.

Question 7. Design Processes II [10 MARKS]

Refer to `Kitchen.java` on the last page that models a restaurant kitchen and answer the following questions. Anywhere you see `/* implementation */` or `/* definition */` is code that is hidden to simplify this question; don't worry about it.

(a) [1 MARK] Why might some subclasses of `Worker` have constructors with different parameters?
This implies some of them have special/default values.

(b) [2 MARKS] According to code, does `Chef` cook? Explain why or why not.
Yes (1.0), because `Chef` still implicitly inherits `Cooking` through `Cook` (1.0).

(c) [7 MARKS] Draw a UML diagram to model `Kitchen.java`. Do not include constructors. You may tear off the last page containing `Kitchen.java` if that helps; just make sure to include it in your submission, especially if you need the blank page (backside) for your answers.

- 1.0 for a correct `Worker`.
- 1.0 for correct interfaces (0.5 each).
- 3.0 for `Worker` subclasses/descendents (0.5 each).
- 2.0 for correct edges between types.

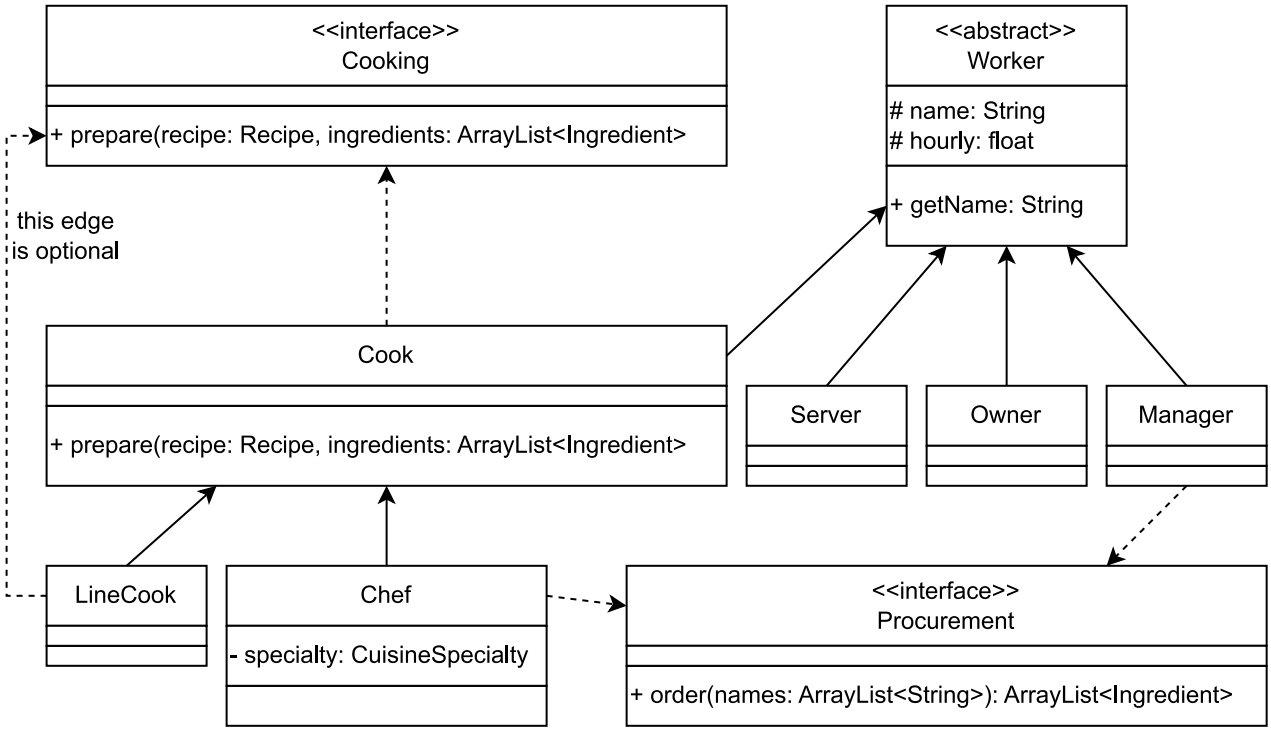

Figure 1: UML Diagram for `Kitchen.java`

Kitchen.java

```
/* exclude boxes for these classes in your UML; just indicate type dependencies */
class Recipe { /* definition */ }
class Ingredient { /* definition */ }
class Dish { /* definition */ }
enum CuisineSpecialty { /* definition */ } // treat this as a primitive type

/* include boxes for these classes in your UML */
abstract class Worker {
    protected String name;
    protected float hourly;
    public Worker(String name, float hourly) { /* implementation */ }
    public String getName() { /* implementation */ }
}
interface Cooking {
    public Dish prepare(Recipe recipe, ArrayList<Ingredient> ingredients);
}
interface Procurement {
    default public ArrayList<Ingredient> order(ArrayList<String> names) { /* implementation */ }
}
class Cook extends Worker implements Cooking {
    public Cook(String name, float hourly) { /* implementation */ }
    public Dish prepare(Recipe recipe, ArrayList<Ingredient> ingredients) { /* implementation */}
}
class Chef extends Cook implements Procurement {
    private CuisineSpecialty specialty;
    public Chef(String name, float hourly) { /* implementation */ }
}
class LineCook extends Cook implements Cooking {
    public LineCook(String name) { /* implementation */ }
}
class Server extends Worker {
    public Server(String name) { /* implementation */ }
}
class Manager extends Worker implements Procurement {
    public Manager(String name, float hourly) { /* implementation */ }
}
class Owner extends Worker {
    public Owner(float hourly) { /* implementation */ }
}
```

You may tear off this page to help you complete the UML question; just make sure to include it in your submission, especially if you need the blank page (other side) for your answers.