P1. Sprint 1

ASSIGNED: 25 January 2024
DUE: 11:59PM, Tuesday, 27 February 2024

100 points

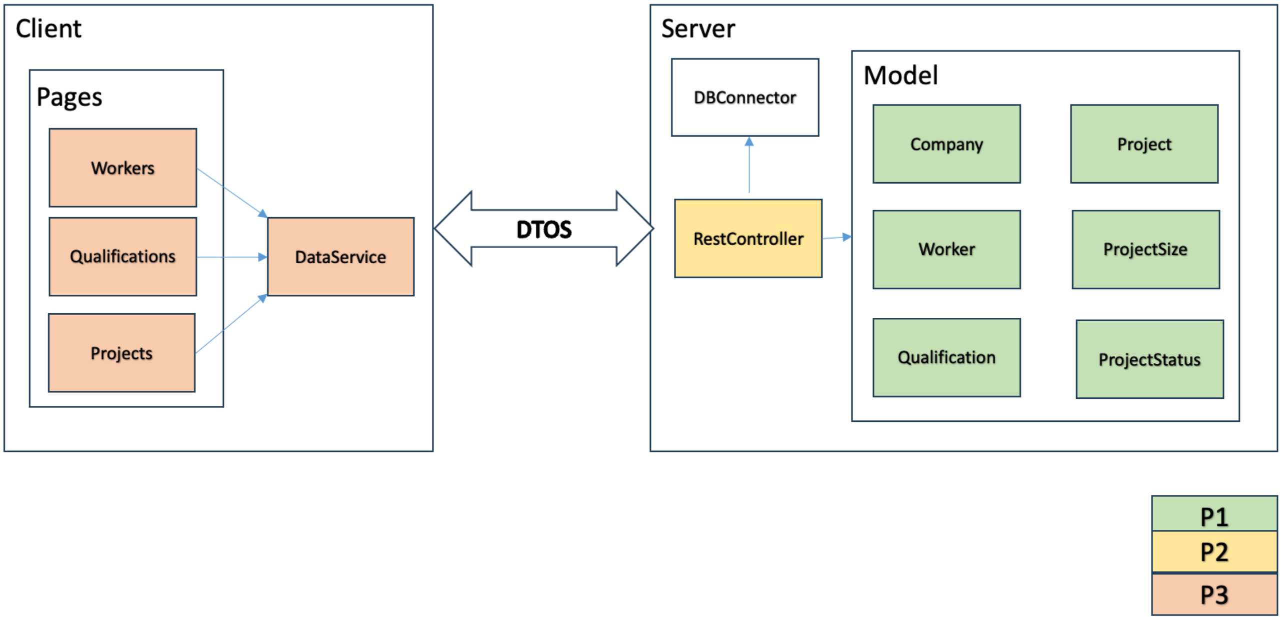
1. Objectives

- Practice test-first development to implement core business logic of an application.
- Review programming in Java given a design specification.
- Develop JUnit 4 test cases using input space partitioning technique.
- Assess coverage using JaCoCo.

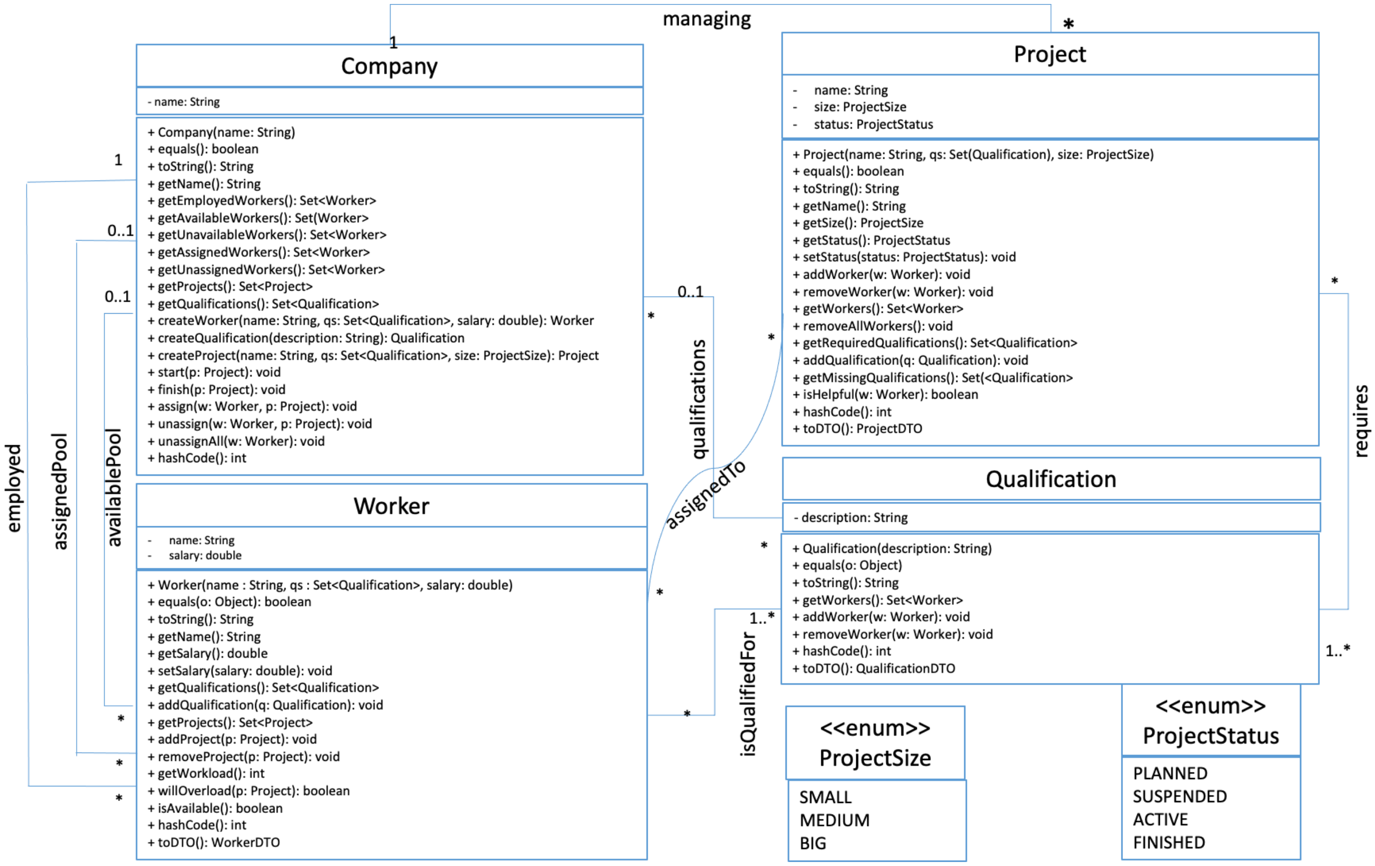
Read the entire assignment before starting to code to get a full picture of what is needed. Some parts may be intentionally left unspecified. This will illustrate how TDD can help you identify what to do in such situations.

2. Description of Semester Project

The overall architecture is shown in the next figure along with colored annotations depicting the sprint that will develop the corresponding components. Note that DTOs and DBConnector components will be provided to you.



The next figure shows a partial design of a project management system as a class diagram containing the necessary classes and enumeration types in the core business logic. The classes are `Company`, `Worker`, `Project`, and `Qualification`. The enumeration types are `ProjectStatus` and `ProjectSize`, each with their shown values.


managing
1
*
Company
- name: String
+ Company(name: String)
+ equals(): boolean
+ toString(): String
+ getName(): String
+ getEmployedWorkers(): Set<Worker>
+ getAvailableWorkers(): Set(Worker>
+ getUnavailableWorkers(): Set<Worker>
+ getAssignedWorkers(): Set<Worker>
+ getUnassignedWorkers(): Set<Worker>
+ getProjects(): Set<Project>
+ getQualifications(): Set<Qualification>
+ createWorker(name: String, qs: Set<Qualification>, salary: double): Worker
+ createQualification(description: String): Qualification
+ createProject(name: String, qs: Set<Qualification>, size: ProjectSize): Project
+ start(p: Project): void
+ finish(p: Project): void
+ assign(w: Worker, p: Project): void
+ unassign(w: Worker, p: Project): void
+ unassignAll(w: Worker): void
+ hashCode(): int
Project
- name: String
- size: ProjectSize
- status: ProjectStatus
+ Project(name: String, qs: Set(Qualification), size: ProjectSize)
+ equals(): boolean
+ toString(): String
+ getName(): String
+ getSize(): ProjectSize
+ getStatus(): ProjectStatus
+ setStatus(status: ProjectStatus): void
+ addWorker(w: Worker): void
+ removeWorker(w: Worker): void
+ getWorkers(): Set<Worker>
+ removeAllWorkers(): void
+ getRequiredQualifications(): Set<Qualification>
+ addQualification(q: Qualification): void
+ getMissingQualifications(): Set(<Qualification>
+ isHelpful(w: Worker): boolean
+ hashCode(): int
+ toDTO(): ProjectDTO
1
0..1
0..1
employed
assignedPool
availablePool
0..1
*
qualifications
*
assignedTo
*
requires
Worker
- name: String
- salary: double
+ Worker(name : String, qs : Set<Qualification>, salary: double)
+ equals(o: Object): boolean
+ toString(): String
+ getName(): String
+ getSalary(): double
+ setSalary(salary: double): void
+ getQualifications(): Set<Qualification>
+ addQualification(q: Qualification): void
+ getProjects(): Set<Project>
+ addProject(p: Project): void
+ removeProject(p: Project): void
+ getWorkload(): int
+ willOverload(p: Project): boolean
+ isAvailable(): boolean
+ hashCode(): int
+ toDTO(): WorkerDTO
*
*
*
*
Qualification
- description: String
*
+ Qualification(description: String)
+ equals(o: Object)
+ toString(): String
+ getWorkers(): Set<Worker>
+ addWorker(w: Worker): void
+ removeWorker(w: Worker): void
+ hashCode(): int
+ toDTO(): QualificationDTO
1..*
1..*
*
isQualifiedFor
<<enum>>
ProjectSize
SMALL
MEDIUM
BIG
<<enum>>
ProjectStatus
PLANNED
SUSPENDED
ACTIVE
FINISHED

2.1. How to read and implement the UML class diagram

Classes are shown with their names (e.g., `Company`), attributes (e.g., `name` of type `String` inside `Company`), and operations (e.g., `createProject`, which takes a project name, a set of `Qualification` instances, and project size, and returns a `Project` instance.

In the above diagram, associations are shown between two classes using multiplicity information at the association ends. For example, the association `Requires` between `Project` and `Qualification` indicates that (1) a project requires a certain set of qualifications (at least one qualification denoted by 1..*) and (2) the same qualification may be required in zero or more projects (denoted by *). A worker must have at least one qualification to be employed in the company.

Sometimes two classes may be related by multiple associations (e.g., Worker and Company) and all these associations need to be implemented/enforced by your code. For example, a worker may be employed in a company. A worker may be available or not depending on their workload. A worker may be assigned to a project or not. A worker may be in the assigned set but no longer available for other projects because of their workload.

Associations must be implemented using appropriate data structures as additional fields in the appropriate class. For example, a set should be implemented using an implementation of the `Set` interface). The method parameters and return values in such cases should be `Set`, not the name of the implementation class (e.g., `HashSet` and certainly not `ArrayList` -- don't use `ArrayList` in the implementation anyway!).

2.2. Assumptions

Assume that worker names and company names are unique. So are project names and qualification descriptions. You don't need to implement checks for uniqueness. Test data will contain only unique names.

A constraint for the entire system is that no worker should ever be overloaded. To determine overloading, consider all the projects the worker is involved in (except FINISHED projects) and calculate the workload. The workload is computed as (3*number_of_big_projects + 2*number_of_medium projects + number_of_small_projects) for the unfinished projects the worker is involved with. This should never exceed 12. If the workload is less than 12, then the worker is considered to be available, otherwise not.

2.3. Specification of Business Logic Classes and Operations

The operation specifications are listed for each class as follows. If during the development of test cases you feel that some specifications are missing, feel free to make up your own and note them in an overview document that you will submit as `overview.txt`.

2.3.1. Class `Qualification` to be implemented in P1

1. `Qualification(description: String)`: Constructor
   Creates a new instance of qualification using the description.

2. `equals(o: Object): boolean`
   This operation will be used by JUnit. Note that the argument to this operation must be of type `Object`, i.e. not `equals(q : Qualification)`, etc. You will override the `equals(o : Object)` method inherited by the class. Two `Qualification` instances are equal if and only if their descriptions match.

3. `hashCode(): int`
   This operation is used when storing the object into a hashset or hashtable. It returns the hashcode of the attribute `description`.

4. `toString(): String`
   Returns the `description`.

5. `getWorkers(): Set<Worker>`
   Returns the set of workers that have this qualification. The order in the set is not under your control, so your tests shouldn't assume any specific ordering in the returned object. Return an empty set if there are no workers.

6. `addWorker(w: Worker): void`
   Adds worker to the set of workers with this qualification. It is not the responsibility of this method to ensure that the worker actually has the qualification. This will be ensured by the method that calls `addWorker` (e.g., `addQualification` in `Worker`). Otherwise, the system will be in an inconsistent state.

7. `removeWorker(w: Worker): void`
   Removes the worker from the set of workers with this qualification. It is not the responsibility of this method to ensure that the worker is actually in the set of workers for this qualification, or is actually removed from the company or the projects. All of this will be ensured by the method that calls `removeWorker`. Otherwise, the system will be in an inconsistent state.

8. `toDTO(): QualificationDTO`
   Returns a data-transfer-object representing the qualification. The object contains the names of the workers. Do not use the toString() method.

2.3.2. Class `Worker` to be implemented in P1

1. `Worker(name: String, qs: Set<Qualification>, salary: double): Constructor`
   Creates a new worker with the given `name`, the set of qualifications and salary. These qualifications must already be present in the company's set of qualifications.

2. `equals(o: Object): boolean`
   This operation will be used by JUnit. Note that the argument to this operation must be of type `Object`, i.e. not `equals (w : Worker)`, etc. You will override the `equals(o: Object)` method inherited by the class. Two `Worker` instances are equal if and only if their names match.

3. `hashCode(): int`
   This operation is used when storing the object into a hashset or hashtable. It returns the hashcode of the attribute `name`.

4. `toString(): String`
   Returns a `String` that includes the name, colon, #projects, colon, #qualifications, colon, salary. For example, a worker named "Nick", working on 2 projects, and having 10 qualifications and a salary of 10000.20 will result in the string `Nick:2:10:10000`. Note that the salary has no decimals but is always rounded down (truncated.

5. `getName(): String`
   Returns the name field.

6. `getSalary(): double`
   Returns the salary field.

7. `setSalary(salary: double) void`
   Sets the salary field.

8. `getQualifications(): Set<Qualification>`
   Returns the qualifications of the worker as a `Set`. The order in the set is not under your control, so your tests shouldn't assume any specific ordering in the returned object. Return an empty set if there are no qualifications.

9. `addQualification(q: Qualification): void`

Addd the qualification q to the set of qualifications of the worker. The set of qualifications for the worker should have no duplicates. It's the caller's responsibility to ensure that this qualification is from the company's set of qualifications.

10. `getProjects(): Set(Project)`
    Returns a set of projects that this worker is in. The order in the set is not under your control, so your tests shouldn't assume any specific ordering in the returned object. Return an empty set if there are no projects.

11. `addProject(p : Project): void`
    The project gets added to this worker. It's the responsibility of the caller (not `addProject`) to check if the project can be added to the worker and also to ensure that the worker is added to the project.

12. `removeProject(p: Project): void`
    Removes the project from the worker. It's the responsibility of the caller (not `removeProject`) to check if the project can be removed from the worker, and also to ensure that the worker is removed from the project.

13. `getWorkload(): int`
    Returns the workload of the worker. The workload is computed as (3*number_of_big_projects + 2*number_of_medium projects + number_of_small_projects) for the projects the worker is involved with (but not FINISHED projects).

14. `willOverload(p: Project): boolean`
    Returns true if a worker will be overloaded if the worker gets assigned to the project "`p`", false otherwise. If adding the new project (irrespective of its current status) to the existing unfinished projects of the worker makes the total workload greater than 12, then the worker will be overloaded. It's the caller's responsibility to ensure that the project is in the set of the company's projects, but think carefully about what gets passed as a parameter (e.g., if the worker is already part of the project, etc).

15. `isAvailable(): boolean`
    Returns true if the workload of the worker is less than 12. False otherwise.

16. `toDTO(): WorkerDTO`
    Returns a data-transfer-object representing the worker. The object contains the names of the projects. Do not use the toString() method.

2.3.3. Class `Project` to be implemented in P1

1. `Project(name: String, qs: Set(Qualification), size: ProjectSize): constructor`
   Creates an instance of a project with the name, qualifications, and size. A project always starts in the `PLANNED` state. Check for boundary conditions on the qualification set as well as the requirements on the String reference (not null). These qualifications must be from the company's set of qualifications.

2. `equals(o: Object): boolean`
   This operation will be used by JUnit. Note that the argument to this operation must be of type `Object`, i.e. not `equals (p : Project)`, etc. You will override the `equals(o : Object)` method inherited by the class. Two `Project` instances are equal if and only if their names match. Note that it is good practice to override the `hashCode` method when `equals` is overridden but we won't be testing it in this assignment.

3. `hashCode(): int`
   This operation is used when storing the object into a hashset or hashtable. It returns the hashcode of the attribute `name`.

4. `toString(): String`
   Returns a `String` that includes the name, colon, number of assigned workers, colon, status. For example, a project named "CS5Anniv" using 10 assigned workers and status PLANNED will result in `CS5Anniv:10:PLANNED`. In the string, status is in upper case (as shown in the UML class diagram).

5. `getName(): String`
   Returns the name of the project.

6. `getSize(): ProjectSize`
   Returns the size of the project.

7. `getStatus(): ProjectStatus`
   Returns the status of the project.

8. `setStatus(s: ProjectStatus): void`
   Sets the status of the project.

9. `addWorker(w: Worker): void`
   Adds a worker to the project. It is up to the caller to determine whether this worker can be added to the project and all the project and company constraints are still enforced. For example, it is called by the `assign` method in the `Company` class.

10. `removeWorker(w: Worker): void`
    Removes a worker from the project. It is up to the caller to determine whether this worker can be removed from the project and if removed, making sure all the other project and company constraints are still enforced. For example, it is called by the `unassign` in the `Company` class.

11. `getWorkers(): Set<Worker>`
    Returns the set of workers assigned to the project. The order in the set is not under your control, so your tests shouldn't assume any specific ordering in the returned object. Return an empty set if there are no workers.

12. `removeAllWorkers(): void`
    Removes all the workers assigned to the project. It's the responsibility of the calling method to make sure this can be done, and if it is done, the caller ensures that the state of the project is consistent. This part is not the responsibility of the `removeAllWorkers` method.

13. `getRequiredQualifications(): Set<Qualification>`
    Returns the set of all the qualifications required for the project. The order in the set is not under your control, so your tests shouldn't assume any specific ordering in the returned object. Return an empty set if there are no qualifications.

14. `addQualification(q: Qualification): void`
    Add q to the set of qualifications required for the project. The set of qualifications for the project should have no duplicates. It's the caller's responsibility to ensure that this qualification is from the company's set of qualifications.

15. `getMissingQualifications(): Set<Qualification>`
    Compare the qualifications required by the project and those that are met by the workers currently assigned to the project. Return the set of qualifications that are not met. An empty set (not null set) is returned when all the qualification requirements are met. The order in the set is not under your control, so your tests shouldn't assume any specific ordering in the returned object. Return an empty set if there are no missing qualifications.

16. `isHelpful(w: Worker): boolean`
    If at least one of the missing qualification requirements of a project is satisfied by the worker, then return true, else return false.

17. `toDTO(): ProjectDTO`
    Returns a data-transfer-object representing the project. The object contains the names of the workers. Do not use the toString() method.

2.3.4. Class `Company` to be implemented in P1

1. `Company(name: String)` Constructor
   Creates a company instance, and sets the name. There are no workers, projects, or qualifications to begin work.

2. `equals(o: Object) : boolean`
   This operation is needed by JUnit. Note that the argument to this operation must be of type `Object`, i.e. not `equals (c : Company)`, etc. You will override the `equals(o: Object)` method inherited by the class. Two `Company` instances are equal if an only if their names match. Note that it is good practice to override the `hashCode` method when `equals` is overridden but we won't be testing it in this assignment.

3. `toString() : String`
   Returns a `String` that includes the company name, colon, number of available workers, colon, and number of projects carried out. For example, a company called ABC that has 20 available workers and 10 projects will result in the string `ABC:20:10`.

4. `getName(): String`
   Returns the name of the company.

5. `getEmployedWorkers(): Set<Worker>`
   Returns the set of all workers employed by the company. The order in the set is not under your control, so your tests shouldn't assume any specific ordering in the returned object. Return an empty set if there are no employed workers.

6. `getAvailableWorkers(): Set<Worker>`
   Returns the set of all workers available in the company. The order in the set is not under your control, so your tests shouldn't assume any specific ordering in the returned object. Return an empty set if there are no available workers.

7. `getUnavailableWorkers(): Set<Worker>`
   Returns the set of all workers who are employed but not available. The order in the set is not under your control, so your tests shouldn't assume any specific ordering in the returned object. Return an empty set if there are no unavailable workers.

8. `getAssignedWorkers(): Set<Worker>`
   Returns the set of all workers assigned to some project in the company. The order in the set is not under your control, so your tests

shouldn't assume any specific ordering in the returned object. Return an empty set if there are no assigned workers.

9. `getUnassignedWorkers(): Set<Worker>`
   Returns the set of all workers who are employed but not assigned to any projects in the company. The order in the set is not under your control, so your tests shouldn't assume any specific ordering in the returned object. Return an empty set if there are no unassigned workers.

10. `getProjects(): Set<Project>`
    Returns the set of projects in the company. The order in the set is not under your control, so your tests shouldn't assume any specific ordering in the returned object. Return an empty set if there are no projects.

11. `getQualifications(): Set<Qualification>`
    Returns the set of qualifications in the company. The order in the set is not under your control, so your tests shouldn't assume any specific ordering in the returned object. Return an empty set if there are no qualifications.

12. `createWorker(name: String, qs: Set<Qualification>, salary: double): Worker`
    AKA hire a worker. Creates a new worker with the given name, set of qualifications, and salary. The set of qualifications must be a subset of the company's set of qualifications. For programming safety, check for null references, and the presence of at least one qualification and return null if a worker instance can't be created. A newly created worker has no assigned projects. Be sure to update the list of employed workers and available workers. This worker must have at least one qualification. You must add the worker to the set of workers for each of the qualifications instances possessed by the worker.

13. `createQualification(description: String): Qualification`
    Creates a new qualification with the given description and add it to the set of qualifications of this company. This qualification has no workers associated with it yet. Beware of null strings.

14. `createProject(name: String, qs: Set(Qualification), size: ProjectSize): Project`
    A new project is created and is entered in the set of projects carried out by the company. The name and size of the project are set. For each qualification in `qs`, add the qualification in the qualification requirements set of the project. The caller ensures that these qualifications must already be present in the company's set of qualifications.

15. `start(p : Project): void`
    A PLANNED or SUSPENDED project may be started as long as the project's qualification requirements are all satisfied. This project is now in ACTIVE status. Otherwise, the project remains PLANNED or SUSPENDED (i.e., as it was before the method was called).

16. `finish(p : Project): void`
    An ACTIVE project is marked FINISHED. The project no longer has any assigned workers. A SUSPENDED or PLANNED project remains as it was. Think of side-effects on all of the sets to make the appropriate changes..

17. `assign(w : Worker, p: Project): void`
    Only workers from the pool of available workers can be assigned as long as they are not already assiged to the same project. The project must not be in the ACTIVE or FINISHED state. The worker should not get overloaded by adding to this project. The worker can be added only if the worker is helpful to the project (i.e., meets at least one missing qualifications). If the conditions are satisfied, (1) the assigned worker is added to the pool of assiged workers of the company unless they were already present in that pool, and (2) the worker is also added to the project. Check if the worker should be moved out of the available pool.

    Note that the same worker can be in both the available pool and assigned pool of wokers at the same time. The worker can also be only in the assigned pool but not in the available pool if they are at their load limit (i.e., adding any project will make them overloaded).

18. `unassign(w: Worker, p: Project): void`
    The worker must have been assiged to the project to be unassigned. If this was the only project for the worker, then delete this worker from the pool of assigned workers of the company. Also think about other situations for the available and assigned pools. If the qualification requirements of an ACTIVE project are no longer met, that project is marked SUSPENDED. A PLANNED OR SUSPENDED project remains in that state.

19. `unassignAll(w: Worker): void`
    Remove the worker from all the projects that were assigned to the worker. Also remove the worker from the pool of assigned workers of the company. Change the state of the affected projects as needed.

3. P1 Tasks

Always use good programming style and Github etiquette. Never commit to the main branch, always work on other branches. Create issues, pull requests, and merge pull requests. You are not allowed to merge your own pull requests. All of these factors will be considered during grading.

3.1. Implementation

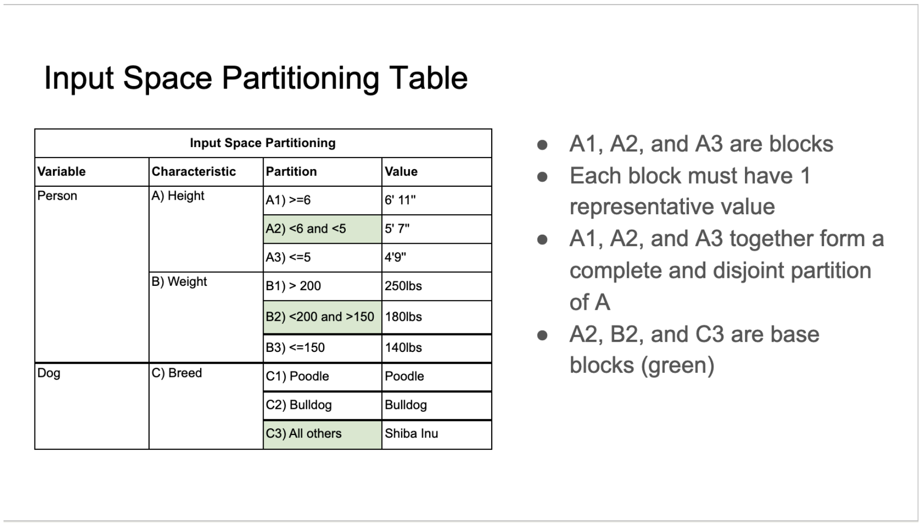
Each team has a repo assigned to them in the CSU-CS-CS415-Spring2024 organization. The repo contains some base code that you will work on. Your task is to use test first development to implement the provided UML class diagram in the **core business logic**, including all the classes, all the associations, and all the enumerated types.

- Don't include print statements anywhere in your submitted code.
- Don't change the names/spellings/capitalization of methods, classes, enum types and values, and packages, and the types of parameters and associations.
- Don't add any public or protected methods that are not listed. Feel free to add private methods if you need helper methods. The reason for disallowing public and protected methods is that using them will prevent your code from compiling with our code.
- Don't add new classes/interfaces/enums. All that you need are the provided classes.

3.2. Testing

You must apply a systematic technique for identifying test input values. The technique is called input space partitioning and you must show your work to explain how you derived the input values by first identifying the domains, then the partitions, and showing that the partitions are complete and non-overlapping. You will also use Base Choice Coverage.

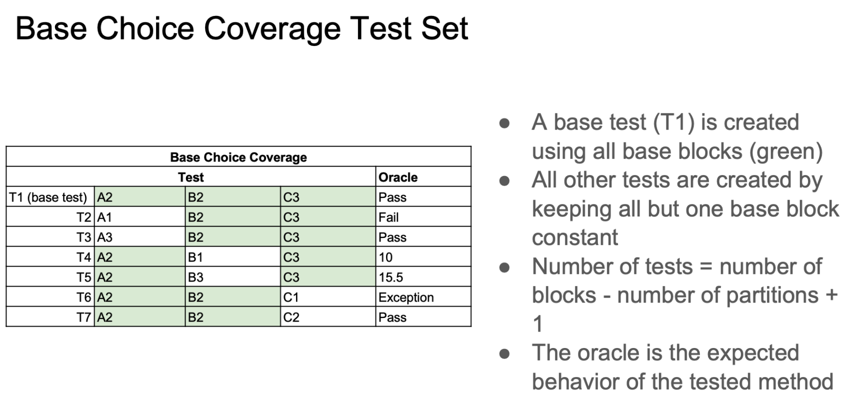
Create a file `reports/P1_ISP.md` (markdown file) in your repository. It should describe your strategies for creating the tests for the methods in the four classes you implemented in this iteration. The following sample tables show how you can display the partitions and BCC.



Input Space Partitioning Table

Input Space Partitioning			
Variable	**Characteristic**	**Partition**	**Value**
Person	A) Height	A1) >=6	6' 11"
		A2) <6 and <5	5' 7"
		A3) <=5	4'9"
	B) Weight	B1) > 200	250lbs
		B2) <200 and >150	180lbs
		B3) <=150	140lbs
Dog	C) Breed	C1) Poodle	Poodle
		C2) Bulldog	Bulldog
		C3) All others	Shiba Inu

- A1, A2, and A3 are blocks
- Each block must have 1 representative value
- A1, A2, and A3 together form a complete and disjoint partition of A
- A2, B2, and C3 are base blocks (green)



Base Choice Coverage Test Set

Base Choice Coverage				
Test				Oracle
T1 (base test)	A2	B2	C3	Pass
T2	A1	B2	C3	Fail
T3	A3	B2	C3	Pass
T4	A2	B1	C3	10
T5	A2	B3	C3	15.5
T6	A2	B2	C1	Exception
T7	A2	B2	C2	Pass

- A base test (T1) is created using all base blocks (green)
- All other tests are created by keeping all but one base block constant
- Number of tests = number of blocks - number of partitions + 1
- The oracle is the expected behavior of the tested method

Implement your JUnit test cases such that the test cases for each class in the class diagram appear in a separate file. For example, the test cases for `Project` must be written in `ProjectTest.java`, `Worker` in `WorkerTest`, and so on. Thus, you will have four test files in P1 called `CompanyTest.java`, `CompanyTest.java`, `WorkerTest.java`, `ProjectTest.java`, and `QualificationTest.java`.

Create a table showing the names of the JUnit test methods and the corresponding combination of input values that you used in the ISP table. This will help us confirm that you actually used ISP in your JUnit tests. You may write additional JUnit tests that are created using some other strategy.

3.3. Coverage Report

The workflow CI will produce coverage information, so the graders will get the report from there. However, you should also collect the coverage information and **reflect** on it in your coverage report.

Create a file `reports/P1_coverage.md` in your repository. It should contain a coverage report showing in tabular format, the method, statement, and branch coverages for each method in the classes `Company`, `Project`, `Worker`, and `Qualification`. Remember to reflect on this table.

3.4. Reflection Report

Create a file `reports/P1_reflection.md` in your repository. It should contain a reflection on what went well and did not go well, and what would be done differently next time.

3.5. Peer evaluation survey on Canvas

Each team member does a survey and gets a separate grade. Note that this grade is separate from the grades mentioned in the next section.

---

4. Grading criteria

Team scores will be based on both the product and the process. Individual adjustments will be made based on quality of contributions.

4.1. Team scores

- Process component (25 points)
  1. Pull requests
  2. Issues
  3. Branches
  4. Commits
  5. Commits to main (Should be none)
  6. How the work is distributed across the sprint (waiting to work until the end is bad)

- Product component (75 points)
  1. Quality of your implementation (20 points): We will test your implementation using our test cases.
  2. Quality of your test cases (20 points): We will use faulty versions with known faults and assess the ability of your test cases to detect these faults.
  3. Quality of your implementation and tests with respect to each other (5 points): We will run your code against your tests and we expect all your tests to pass.
  4. Use of input space partitioning (20 points): We will grade your `P1_ISP.md` report on how you applied input space partitioning.
  5. Submission of coverage report (5 points): We will grade your code coverage report (`P1_coverage.md`).
  6. Reflection report (5 points): We will grade your reflection (`P1_reflection.md`)

Note that we are going to evaluate your implementation and tests VERY, VERY thoroughly. We have hundreds of test cases and faults to inject. You must think about all possible corner cases, inputs, and sequence of method calls.

4.2. Individual Adjustments

Adjustments can be positive or negative up to 20 points and for reasons such as:

- low individual process score,
- bad peer reviews from teammates,
- gaming the system (e.g., purposefully increasing the number of commits without actually imlementing much functionality).

4.3. GitHub Actions

We use automated Github Actions to compile and tests the code that you push code to Github. You must never push broken code to the main branch, in other words you must never "break main". Github Actions are one of the tools that we use to calculate your grade. You are not allowed to add/modify/delete any GitHub action, workflow, or workflow run.

- If you modify a workflow file, we are going to see it in the repository git history, and your team is going to lose points.
- If you delete a workflow run, we are going to see that there is a missing workflow run for a specific 'git push', and your team is going to lose points.