Instances of refactoring

1. Using getters - Why are getters better than `public final`?
    a. This allows more flexibility if you want to change the `public final` variable. If you are unsure how the `public final` variable will be stored, just make it private. You can make as many changes as you want to your so-called `public final` variable and as long as you maintain the getter's functionality, the program will not break to other components.
    b. SOLID principles upheld: O/C (minimal changes needed per change needed)
2. Extracted methods in Controller Input switch statements
    a. This prevents the switch statement from getting too large. I'm a bit against this, as it makes the code harder to navigate - as it makes it harder to map from command as a string to what it does.
3. Timetable used to store its planned courses as a separate reference to the planned courses in students. Now it uses an alias from the student class, and students are now dependent on timetables.
    a. This reduces the need to constantly update timetables every time a change is done in students, or at least reduces what has to be constantly updated.

Bad design instances that we've decided to leave in

This is the list of things that violate good design that we may have left in our project.

1. Controllers accessing entities, particularly in ControllerInputTimetable
    a. Entities are only accessed for a split second. The controller does not call any methods from the entities; they only pass them to use cases, who have the full right to access those entities.
2. ~~Using print statements in Controllers~~
    a. ~~If any print statements are left in the controllers, they are placed there because it is faster to write print statements in controllers rather than put the burden of them on the presenter. Note that there is only one presenter… for now, as all controllers need access to the same presenter - I'm not sure how we can have multiple presenters without the need to add more frustrating lines of code.~~
3. Using println statements instead of the logging module inside use cases (StorageLoader)

    a. The logging module is too complicated to learn at the time when there are so many other aspects in the project to worry about at the time. I'll fix that maybe in phase 2, though these println statements will likely show up in edge cases.

4. Warnings about unused variables or unused return values
    a. We plan to use them later; not now. However, removing them limits our decisions, and we'd rather have more decisions in the future. This project is extremely large and I'd rather not spend hours removing something I'd add back later.
    b. A lot of the errors are Course-related - we intend to use these values later, so we'll keep them.
5. Many instances of the warning: Fields can be converted to a local variable
    a. That is extremely short-sighted.
6. The extreme number of classes in this project.
    a. This is what happens when presenters can't access entities.
7. Clear cases where dependency inversion is **not** used
    a. Some classes are just responsible for doing what they are supposed to do. At least somewhere, a new instance of a class has to be created.
8. Courses not having a builder
    a. Creating the builder for course will 100% cause more pain than just leaving it as it is. Course is 100% functional at the moment. Because the structure of Course is tree-like and is very complicated, refactoring how course is constructed is like doing something very expensive with little to no benefit at the time (considering all other UofT APIs aren't accessible).
9. Using the **static** keyword with the logger
    a. Loggers are development only. Treat them as if they never existed.
10. Entities such as Timetable doing way too much logic
    a. This may have been fixed; I'm not sure.
11. Course being present in timetable's Presenter
    a. This will be fixed in phase 2. Student storing Course objects was a terrible idea.

Dependency rules

This program tries to follow the dependency rules most of the time, though there are some instances where it does not (check the previous section). Here's an example of the program following the dependency rule:

Timetable presenter

- Uses timetable communicators (these are use cases) to narrow down entity data into built in types, in a way that it can be easily presented to the screen
- The

Design patterns

1. Builder
    a. Ex: RequisiteListBuilder in commit hash d55ae6aaa85e6af8838aefeef401f20f3fdda36d (sorry for bad commit name - that occurred early in the project)
2. Factory
    a. Ex: ControllerInputFactory (no commit hash, as this was implemented in the phase 0 code after it was submitted)

Design patterns that could've been implemented

1. Using DI for Timetable in Student's constructor

UML diagram

Feel free to zoom in this image using a PDF viewer.