COURSEWORK CLARIFICATIONS & TIPS

Parsing – you may be still developing this, but remember:

- If encountering issues with the input file, **do not** simply output something like this

```
Error: invalid input file.
```

- or like this

```
Error: invalid token found in input file.
```

- **Do** tell the user what the problem encountered was, e.g.:

```
Error: Invalid input file provided. Parameter 'stopTime'
missing. The simulation will terminate.
```

COURSEWORK CLARIFICATIONS & TIPS

Parsing – you may be still developing this, but remember:

- If you did encounter an error, **interrupt execution**.
- Otherwise, if you did parse all the parameters required, but the values of some do not make much sense, you may continue execution.
- However, **issue a warning first**, e.g.

```
Warning: 'stopTime' parameter smaller than 'warmUpTime'.
The simulation will continue.
```

COURSEWORK CLARIFICATIONS & TIPS

Parsing – you may be still developing this, but remember:

- **Do not** prompt the user for input during execution! That is, something like the following is not acceptable:

```
Press any key to start simulation...
```

- If you do need to stop execution for debugging purposes, while allowing us to perform **automated testing** you can work with an optional flag which enables debugging, e.g.

```
$ ./simulate.sh
Usage: ./simulate.sh <input_filename> [debug=on|off]
        Default option: debug=off
        </input_filename>
```

COURSEWORK CLARIFICATIONS & TIPS

- Alternatively create a development branch, leaving the master always 'testable', e.g.

```
$ git branch dev
```

- will create a branch named 'dev'. You will need to select it (checkout) first, to make commits to it.
- You can then make changes to the development branch and merge with master when new feature completed.
- I will come back to branching later.

COURSEWORK CLARIFICATIONS & TIPS

Parsing – you may be still developing this, but remember:

- Distinguishing between errors/warning can be at times debatable. For instance

```
...
disposalDistrShape 1 2 3
...
```

- Since you have a list of values after a valid parameter keyword, but the 'experiment' keyword is missing, you may regard this as invalid input and issue an error.
- But you could also ignore the tokens following the first value ('1') and issue a warning.

COURSEWORK CLARIFICATIONS & TIPS

- If the input file is indeed invalid (or missing) and you are coding in Java, do not simply throw an exception.
- **Handle it by returning a short message explaining the problem to the user**.
- The 'binServiceTime' is expressed in seconds. No need to expect a float value. Working with a 16-bit integer is fine.
- Some parameters may be given in different order, but do not expect this for area descriptions. This is valid

```
areaIdx 0 serviceFreq 0.0625 thresholdVal 0.7 noBins 5
```

- This is not:

```
areaIdx 0 noBins 5 serviceFreq 0.0625 thresholdVal 0.7
```

COURSEWORK CLARIFICATIONS & TIPS

Events generation:

- Bin disposal events are independent at different bins.
- The average disposal rate is **wrt. a bin** not per area.
- Do you need to store all the delays between disposal events at each bin upfront?
- Or can you extract new delays from the given distribution once a disposal event at the target bin was executed?
- It may be quite inefficient to store all the events happening throughout a days long simulation.

COURSEWORK CLARIFICATIONS & TIPS

Bin overflows:

- 'Exceeded' refers to something strictly greater than.
- Threshold exceed `!=` bin overflowed, unless `thresholdVal==1` or
- new bag disposal caused
  content volume > threshold * bin capacity, and
- content volume > bin capacity (think large bag).
- Overflow can happen at most once between two services.
- If bin not overflowed, add new bag irrespective of volume.
- **Do not** 'partially' service a bin.

THE SCOREBOARD

- This is meant only to give you an indication of where you are with the development of the simulator.
- Only functionality required for Part 2 is tested at the moment.
- There is no 1-to-1 correspondence between the tests for which you see results on the scoreboard and the Part 2 evaluation.
- Though they will be closely related.
- What the marker is testing for at the moment is existing fork, error free compilation, correct parsing of valid files, generation of valid output, invalid input detection.

MULTIPLE FILES

- Question: Should you spread your implementation across multiple source code files?
- There may be *some* good reasons to do so:
  - Increase code reusability
  - Reduces compilation time
  - Could help navigating source code faster

MULTIPLE FILES

- Not suggesting you should not, but do so for a good reason.
- Given the size of this project, you could try to use as few files as possible.
- Move type definitions, functions, etc. to separate files when that seems necessary.

SHOULD I DEVELOP CODE WITH OR WITHOUT AN IDE?

- This shouldn't make a difference, but you may have good reasons for choosing one of the two approaches.
- Coding using a plain text editor (e.g. vi, nano)
  - You can easily code remotely (over ssh) on e.g. a DiCE machine
  - In some cases you may need to write a **makefile** yourself (especially if working with multiple files).

SHOULD I DEVELOP CODE WITH OR WITHOUT AN IDE?

- Using IDEs
  - Nicer keyword highlighting;
  - Some auto complete braces/brackets/parenthesis;
  - Some may have integrated help for functions;
  - Some warn about certain syntax errors as you type;
  - Perhaps easier if you are not a very experienced programmer;
- If you decide to code using an IDE, it's entirely up to you which one you choose (NetBeans, Eclipse, CodeLite, etc.)

CODE OPTIMISATION

OPTIMISATION

- Re-usability can conflict heavily with readability.
- Similarly optimised or fast code can conflict with readability.
- You are writing a simulator which may have to simulate millions of events.
- In order to obtain statistics, it may then have to repeat the simulation thousands of times.
- Optimised code is generally the opposite of reusable code.
- It is optimised for its particular assumptions which cannot be violated.

PREMATURE OPTIMISATION

- The notion of optimising code before it is required.
- The downside is that code becomes less adaptable.
- Because the requirements on your optimised piece of code may change, you may have to throw away your specialised code and all its optimisations.
- Note: This does not refer to the requirements of the **CSLP**.
  - In a realistic setting they may, but not here.
  - It is the requirements of a particular portion of your code which may change.

TIMELY OPTIMISATION

- So when is the correct time to optimise?
- Refactoring is done in between development of new functionality
  - Recall this makes it easier to test that this process has not changed the behaviour of your code.
- This is also a good time to do **some** optimisation
  - You should be in a good position to test that your optimisations have not negatively impacted correctness.

WHEN TO OPTIMISE?

- When you discover that your code is not running fast enough, it's probably wise to optimise it.
- Often this will come towards the end of the project.
- It should certainly come after you have something deployable.
- Preferably after you have developed and tested some major portion of functionality.

A PLAUSIBLE STRATEGY

- Perform no optimisation until the end of the project once **all** functionality is complete and tested.
- This is a reasonable approach; however:
- During development, you may find that your test suite takes a long time to run.
- Even one simple run to test the functionality you are currently developing may take minutes/hours.
- This can slow down development significantly, so it may be appropriate to do some optimisation at that point.

HOW TO OPTIMISE

- The very first thing you need **before** you could possibly optimise code is a *benchmark*.
- This can be as simple as timing how long it takes to run your test suite.
- $O(n^2)$ solutions will beat O(n log n) solutions on sufficiently small inputs, so your benchmarks must not be too small.

HOW TO OPTIMISE

Once you have a suitable benchmark then you can:

1. Save a local copy of your current code, or branch (I will come back to this option);
2. Run your benchmark and record the run time;
3. Perform what you think is an optimisation on your source code;
4. Re-run your benchmark & compare the run times;
5. If you successfully improved the performance of your code keep the new version, otherwise revert changes;
6. Do one optimisation at a time.

HOW TO OPTIMISE

- However, bear in mind that you are writing a **stochastic** simulator
  - This means each run is different and hence may take a different time to run,
  - Even if the code has not changed or has changed in a way that does not affect the run time significantly.
  - Simply using the same input several times should be enough to reduce or nullify the effect of this.

PROFILING

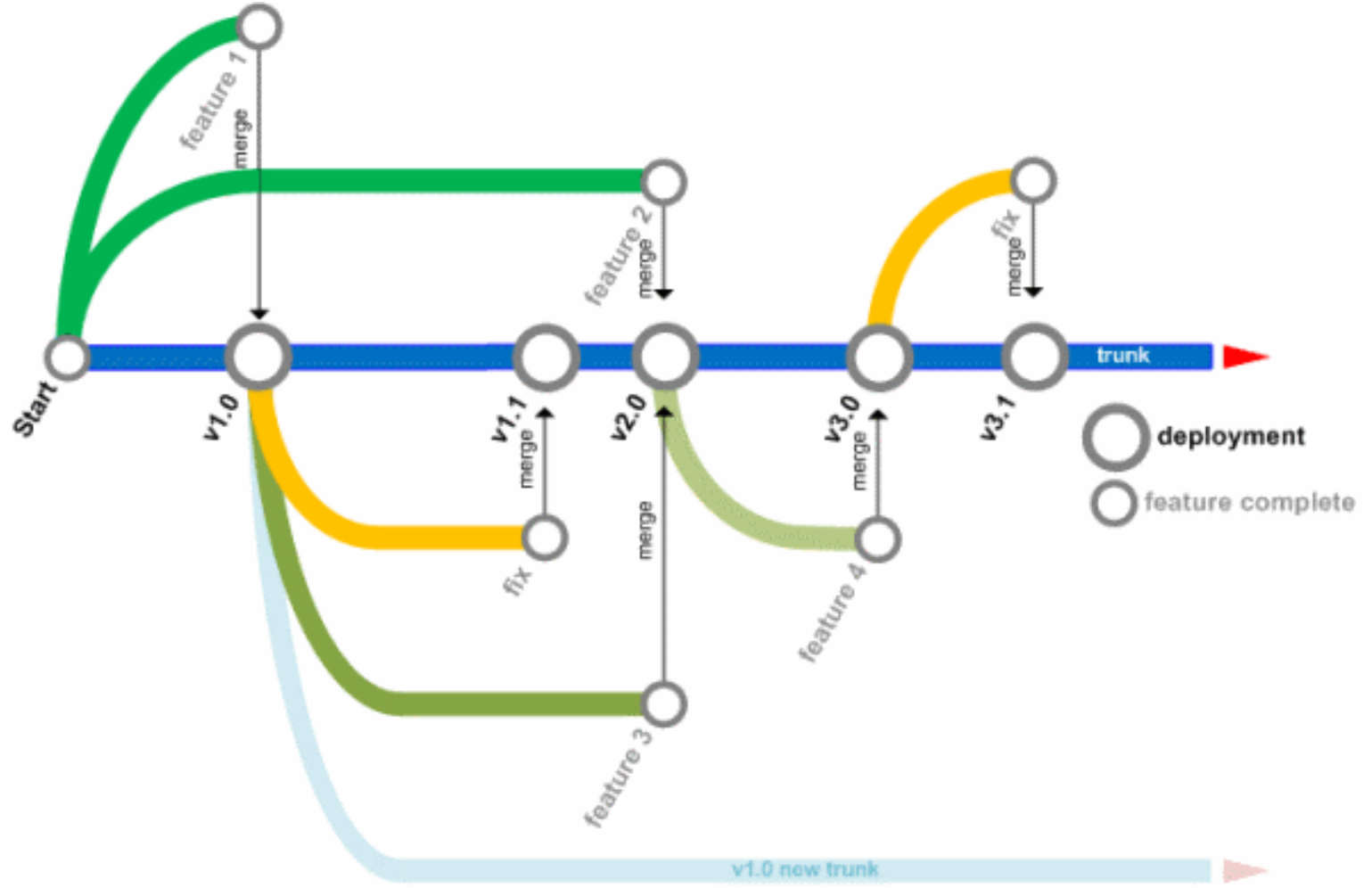
- *Profiling* is **not the same** as *benchmarking*.
- *Benchmarking*:
  - determines how quickly your program runs;
  - is to performance what testing is to correctness.
- *Profiling*:
  - is used after benchmarking has determined that your program is running too slowly;
  - is used to determine which parts of your program are causing it to run slowly;
  - is to performance what debugging is to correctness.

BENCHMARKING & PROFILING

- Without benchmarking you risk making changes to your program that will lead to poorer performance.
- Without profiling you risk wasting effort optimising a part of code which is either already fast or rarely executed.

Documenting: Source code comments are a good place to explain *why* the code is the way it is.

BRANCHING



source: activegrade.com

BRANCHING

- This occurs in software development frequently.
- In particular, you aim to add a new feature only to discover that the supporting code does not support your enhancement.
- Hence you need to first improve the supporting code, which may itself require modification.
- Branching is the software solution to this problem, that most other projects do not have available.
- It is easy to copy the current state of a project, work on the copy and then merge back if the work is successful.

BRANCHING - THE BASIC IDEA

When commencing a unit of work:

1. Begin a branch, this *logically* copies the current state of the project.
2. The original branch might be called ‘master’ and the new branch ‘feature’.
3. Complete your work on the ‘feature’ branch.
4. When you are happy merge the results back into the ‘master’ branch.

BRANCHING - REASONS

- Mid-way through, should you discover that your new feature is ill-conceived,
- or, your approach is unsuitable,
- You can simply revert back to the master branch and try again.
- Of course you can revert commits anyway, but this means you're not entirely deleting the failed attempt.
- You can also concurrently work on several branches and only throw away the changes you do not want to keep.

BRANCHING

- Stay organised!
- One approach is to have a new branch for each feature
  - This has the advantage that multiple features can be worked upon concurrently.
  - Usually each feature branch is deleted as soon as it is merged back into ‘master’.
- A more lightweight solution is to develop everything on a branch named ‘dev’.
- After each commit, merge it back to ‘master’ you then always have a way of creating a new branch from the previous commit.

BRANCHING

- After you created a branch, be sure you selected it, otherwise you are still making commits to the master.
- Example:

```
$ git branch dev
$ git checkout dev
```

- This effectively tells git to navigate to the 'dev' branch.
- Once the new feature/optimisation is ready, merge back to master.

```
$ git checkout master
$ git merge dev
```

- You will find a more detailed discussion in **this tutorial**.

FINAL NOTE

- There are still a few people who did not fork the CSLP repository and/or given us read permissions.
- There are only two weeks left until the deadline for Part 2.
- Remember this carries 50% of the marks – act now!