Todo list

Important: Lay abstract. iii
Important: Abstract. iv
Important: Acknowledgements. v
Important: Replace reading notes. xii
Important: Declaration of Academic Achievement. xiii
Easy: Such as this one, but check out Section 2.5 for more options. . . . 3
Important: "Important" notes. 6
Generic inlined notes. 6
Later: TODO notes for later! For finishing touches, etc. 6
Easy: Easier notes. 6
Needs time: Tedious notes. 6
Q #1: Questions I might have? 7

THE GENERATION OF TEST CASES IN DRASIL

THE GENERATION OF TEST CASES IN DRASIL

By SAMUEL CRAWFORD, B.Eng.

A Thesis
Submitted to the Department of Computing and Software
and the School of Graduate Studies
of McMaster University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Applied Science

Master of Applied Science (2023) McMaster University
(Department of Computing and Software) Hamilton, Ontario

TITLE: The Generation of Test Cases in Drasil
AUTHOR: Samuel Crawford, B.Eng.
SUPERVISOR: Dr. Carette and Dr. Smith
PAGES: xiii, 15

Lay Abstract

Important: Lay abstract.

Abstract

Important: Abstract.

Acknowledgements

Important: Acknowledgements.

Contents

Todo list i

Lay Abstract iii

Abstract iv

Acknowledgements v

Contents vi

List of Figures viii

List of Tables ix

List of Source Codes x

List of Abbreviations and Symbols xi

Reading Notes xii

Declaration of Academic Achievement xiii

1 Introduction 1
- 1.1 Template Organization . . . 1
- 1.2 Writing Tips . . . 2
- 1.3 Development Recommendations . . . 3
- 1.4 Troubleshooting . . . 4

2 Extras 5
- 2.1 Writing Directives . . . 5
- 2.2 Acronyms . . . 5
- 2.3 HREFs . . . 5
- 2.4 Code Snippets . . . 6
- 2.5 TODOs . . . 6

3 Notes 8
- 3.1 Software Metrics . . . 8
- 3.2 Software Testing . . . 8

3.2.1 General Testing Notes 8
3.2.2 Types of Testing 8

4 Before Submitting **13**

Bibliography **14**

Appendix **15**

List of Figures

List of Tables

1.1 Template Organization 1

List of Source Codes

2.1 Pseudocode: exWD 5
2.2 Pseudocode: exPHref 6

A.1 “MultiDefinitions” (MultiDefn) Definition 15
A.2 Pseudocode: Broken QuantityDict Chunk Retriever 15

List of Abbreviations and Symbols

HREF	Hypertext REFerence
HTML	HyperText Markup Language
IDE	Integrated Development Environment
PDF	Portable Document Format

Reading Notes

Before reading this thesis, I encourage you to read through these notes, keeping them in mind while reading.

- The source code of this thesis is publicly available.
- This thesis template is primarily intended for usage by the computer science community[1]. However, anyone is free to use it.
- I've tried my best to make this template conform to the thesis requirements as per those set forth in 2021 by McMaster University. However, you should double-check that your usage of this template is compliant with whatever the "current" rules are.

Important: Replace reading notes.

[1] Hence why there are some LaTeX macros for "code" snippets.

Declaration of Academic Achievement

Important: Declaration of Academic Achievement.

Chapter 1

Introduction

Congratulations! If you're seeing this, it means you've managed to compile the PDF, which also means you can get started on typesetting your thesis[1].

This template is adapted from my thesis. If you'd like to see an example of this template in practice, please feel free to use my thesis as an example.

1.1 Template Organization

I've broken up the template according to my preferred organization: chapters in separate files, various kinds of assets (images, tables, code snippets, macros, etc.) in separate files, etc. The split is approximately according to Table 1.1.

Table 1.1: Template Organization

File/Folder	Intended Usage & Description
`thesis.tex`	Focal LaTeX file that collects everything and is used to build your thesis/report document.
`Makefile`	A basic `Makefile` configuration. See `make help` for a list of helpful commands.
`build/`	When you build your PDF, this folder is used as the working directory of LuaLaTeX. Using this allows us to quickly get rid of LaTeX build files that can cause problems when we re-build documents.
`manifest.tex`	Basic options that you should certainly configure according to your needs.
`chapters.tex`	All chapters of your thesis should be included here.

[1] Or report or ...

`chapters/`	Enumeration of the chapters of your thesis. I prefer using a two-digit indexing pattern for the prefix of file names so that I can quickly open up by chapter number using VS Codium.
`assets.tex`	Enumeration of the various kinds of "assets" in the `assets/` folder. See the file for examples on how you can write your extra utility macros.
`assets/`	Enumeration of various kinds of "assets," with subdirectories for images and figures, tables, and code snippets.
`front.tex`	All front matter of your thesis should be included here.
`front/`	Enumeration of the front chapters of your thesis. These chapters should all be numbered using Roman numerals.
`back.tex`	All back matter of your thesis should be included here.
`back/`	Enumeration of the back matter content.
`acronyms.tex`	List of acronyms you intend to use in your thesis. This uses the "acro" LaTeX package.
`macros.tex`	Helpful macros!
`unicode_chars.tex`	At times, you might find issues with unicode characters, especially in verbatim environments, where you might need to manually define them using other font glyphs.
`mcmaster_colours.tex`	Macros for the McMaster colour palette.
`README.md`	Read it!
`.gitignore`	List of files in the working directory that should be ignored by git.
`latexmkrc`	Used for setting the timezone for latexmk, but can be used for other options.

1.2 Writing Tips

When drafting chapters, I:

1. wrote "writing directives" for each chapter to understand what I need to write about (see Section 2.1),

2. wrote "todo" notes for tedious things that I might want to do later (such as citations, figures, code snippets, etc., see Section 2.5), and

3. regularly built my thesis using `make debug` to make sure that whatever I wrote didn't break the LaTeX code.

For workflow recommendations, you should speak with your supervisor as they might prefer you work in a specific way with them.

1.3 Development Recommendations

Other than the basic tools I used for this template, I enjoyed using the following tools while writing my thesis:

1. VS Codium/VS Code[2] with the following extensions:
   (a) LaTeX Workshop, for LaTeX syntax highlighting, code formatting (this is highly recommended), and code completion,
   (b) LTeX - LanguageTool grammar/spell checking, for grammar checking using LangaugeTool, and
   (c) Todo Tree, for quickly listing all of my TODO notes in my IDE (in addition to the list at the top of the PDF).
2. texcount (which should come with your LaTeX installation) to quickly check the word count of individual LaTeX files, and
3. Zotero for collecting my references and quickly exporting bib entries that I could use.

In particular, when writing, I found it particularly helpful to use VS Code's "Zen Mode" (to see your keybind, press `CTRL+ALT+P` and search for "Zen"), which enters a stripped-down full-screen version of the current working file, keeping your eyes purely focused on the document in front of you. Being comfortable with the keybinds is particularly helpful for working effectively in this setup. For example, I found the following[3] to be helpful: `CTRL+TAB` and `CTRL+SHIFT+TAB` to scroll between open files, `CTRL+P` to quickly open up recent files, `CTRL+ALT+P` to run commands you forgot the keybind for, `CTRL+O` to open up files out of the current working directory.

While writing, I enjoyed:

1. using "TODO" notes

   Easy: Such as this one, but check out Section 2.5 for more options.

   to collect notes that I would want to do later,
2. formatting the LaTeX code to make it easier to read (the LaTeX Workshop plugin has functionality for this),

[2] I prefer VS Codium simply because I prefer libre software.

[3] If you're not using Linux, I cannot guarantee that these will be the same for you, so you should use `CTRL+ALT+P` to look for your appropriate bound keybinds.

3. breaking the non-textual content into separate files and "include"-ing them in the LaTeX code so that they didn't cause large visual interruptions,
4. using git to version control copies of my thesis, chapters, etc.,
5. using TikZ and draw.io/diagrams.net to build graphics and diagrams, and
6. building the thesis often using `make debug` to quickly debug issues in the written code.

1.4 Troubleshooting

"StackOverflow" is a great area to look for solutions to common LaTeX issues. Otherwise, feel free to use create a ticket or sending an email to me.

Chapter 2

Extras

Writing Directives

- What macros do I want the reader to know about?

2.1 Writing Directives

I enjoy writing directives (mostly questions) to navigate what I should be writing about in each chapter. You can do this using:

Source Code 2.1: Pseudocode: exWD

```
\begin{writingdirectives}
    \item What macros do I want the reader to know about?
\end{writingdirectives}
```

Personally, I put them at the top of chapter files, just after chapter declarations.

2.2 Acronyms

I used a lot of acronyms in my thesis, and I wanted to add a glossary to the front matter. This is compulsory for McMaster theses, but you may remove it if you don't need it. I used the `acro` package, feel free to read their documentation. For example, I might want to write HyperText Markup Language (HTML) (written using `\acf{html}`) or just HTML (written using `\acs{html}`). To define them, you can find examples in the `acronyms.tex` file.

2.3 HREFs

For PDFs, we have (at least) 2 ways of viewing them: on our computers, and printed out on paper. If you choose to view through your computer, reading links

(as they are linked in this example, inlined everywhere with "clickable" links) is fine. However, if you choose to read it on printed paper, you will find trouble clicking on those same links. To mitigate this issue, I built the "porthref" macro (see `macros.tex` for the definition) to build links that appear as clickable text when "compiling for computer-focused reading," and adds links to footnotes when "compiling for printing-focused reading." There is an option (`compilingforprinting`) in the `manifest.tex` file that controls whether PDF builds should be done for computers or for printers. For example, by default, McMaster is made with clickable functionaity, but if you change the `manifest.tex` option as mentioned, then you will see the link in a footnote (try it out!).

Source Code 2.2: Pseudocode: exPHref

```
\porthref{McMaster}{https://www.mcmaster.ca/}
```

2.4 Code Snippets

Since I did my Master's in computer science, I needed code snippet listings. For them, I chose to use the `minted` package (which lets you write colourized code snippets, supporting syntax highlighting for many languages). However, I also wanted source code listings to be linked to the main Drasil [1] repository. For example, to write Source Code A.1[1], I had to write `assets/code/example.tex` and create appropriate links in `assets.tex`. Additionally, for pseudocode, you can also use the pseudocode environment, such as that used in Source Code A.2 (similarly built).

2.5 TODOs

While writing, I plastered my thesis with notes for future work because, for whatever reason, I just didn't want to, or wasn't able to, do said work at that time. To help me sort out my notes, I used the `todonotes` package with a few extra macros (defined in `macros.tex`). For example,...

Important notes:

Important: "Important" notes.

Generic inlined notes:

Generic inlined notes.

Later: TODO notes for later! For finishing touches, etc.

Notes for later:

Some "easy" notes:

Easy: Easier notes.

Tedious work:

[1]Note: the header links use the portable HREFs (as per Section 2.3)!

Needs time: Tedious notes.

Q #1: Questions I might have?

Questions:

Chapter 3

Notes

3.1 Software Metrics

- The following branches of testing started as parts of quality testing:
  - Reliability testing [2, p. 18, ch. 10]
  - Performance testing [2, p. 18, ch. 7]
- Reliability and maintainability can start to be tested even without code by “measur[ing] structural attributes of representations of the software” [2, p. 18]
- The US Software Engineering Institute has a checklist for determining which types of lines of code are included when counting [2, pp. 30-31]
- Measurements should include an entity to be measured, a specific attribute to measure, and the actual measure (i.e., units, starting state, ending state, what to include) [2, p. 36]
  - These attributes must be defined before they can be measured [2, p. 38]

3.2 Software Testing

3.2.1 General Testing Notes

- Simple, normal test cases (test-to-pass) should always be developed and run before more complicated, unusual test cases (test-to-fail) [3, p. 66]

3.2.2 Types of Testing

Dynamic Black-Box (Behavioural) Testing [3, p. 64-65]

"Entering inputs, receiving outputs, and checking the results" [3, p. 64]

Requirements

- Requirements documentation (definition of what the software does) [3, p. 64]; relevant information could be:
  - Requirements: Input-Values and Output-Values
  - Input/output data constraints

Exploratory Testing [3, p. 65]

An alternative to dynamic black-box testing when a specification is not available [3, p. 65]. The software is explored to determine its features, and these features are then tested [3, p. 65]. Finding any bugs using this method is a positive thing [3, p. 65], since despite not knowing what the software *should* do, you were able to determine that something is wrong.

Requirements

- Source code [3, p. 65] (how can Drasil deduce functionality from this? Doxygen documentation? Is this necessary if we have the specification for dynamic black-box testing? Is this useful?)

Equivalence Partitioning/Classing [3, p. 67-69]

The process of dividing the infinite set of test cases into a finite set that is just as effective (i.e., by revealing the same bugs) [3, p. 67].

Requirements

- Ranges of possible values [3, p. 67]; could be obtained through:
  - Input/output data constraints
  - Case statements

Data Testing [3, p. 70-79]

The process of "checking that information the user inputs [and] results", both final and intermediate, "are handled correctly" [3, p. 70].

Boundary Conditions [3, p. 70-74] "[S]ituations at the edge of the planned operational limits of the software" [3, p. 72]. Often affects types of data (e.g., numeric, speed, character, location, position, size, quantity [3, p. 72]) each with its own set of (e.g., first/last, min/max, start/finish, over/under, empty/full, shortest/longest, slowest/fastest, soonest/latest, largest/smallest, highest/lowest, next-to/farthest-from [3, p. 72-73]). Data at these boundaries should be included in an equivalence partition, but so should data in between them [3, p. 73]. Boundary conditions should be tested using "the valid data just inside the boundary, ... the

last possible valid data, and ... the invalid data just outside the boundary" [3, p. 73]. *Buffer overruns* are "the number one cause of software security issues" [3, p. 75].

Requirements

- Ranges of possible values [3, p. 67, 73]; could be obtained through:
  - Case statements
  - Input/output data constraints (e.g., inputs that would lead to a boundary output)

Sub-Boundary Conditions [3, p. 75-77] Boundary conditions "that are internal to the software [but] aren't necessarily apparent to an end user" [3, p. 75]. These include powers of two [3, p. 75-76] and ASCII and Unicode tables [3, p. 76-77].

Requirements

- Knowledge of powers of two [3, p. 75-76] (stored alongside `Integer, Rational, Real, Natural, Matrix :: Space`)
- Knowledge of ASCII and Unicode tables [3, p. 76-77] (stored alongside `Char, String, DiscreteS :: Space`)

Default, Empty, Blank, Null, Zero, and None [3, p. 77-78] These should be their own equivalence class, since "the software usually handles them differently" than "the valid cases or ... invalid cases" [3, p. 78].

Requirements

- Knowledge of an "empty" value for each `Space` (stored alongside each type in `Space`?)
- Knowledge of how input data could be omitted from an input (e.g., a missing command line argument, an empty line in a file); could be obtained from:
  - User responsibilities
- Knowledge of how a programming language deals with `Null` values and how these can be passed as arguments

Invalid, Wrong, Incorrect, and Garbage Data [3, p. 78-79] This is testing-to-fail [3, p. 77].

Requirements This seems to be the most open-ended category of testing.

- Specification of correct inputs that can be ignored; could be obtained through:
  - Input/output data constraints (e.g., inputs that would lead to a violated output constraint)
  - Type information for each input (e.g., passing a string instead of a number)

State Testing [3, p. 79-87]

The process of testing "a program's states and the transitions between them" [3, p. 79].

Logic Flow Testing [3, p. 80-84] This is done by creating a state transition diagram that includes:

- Every possible unique state
- The condition(s) that take(s) the program between states
- The condition(s) and output(s) when a state is entered or exited

to map out the logic flow from the user's perspective [3, p. 81-82]. Next, these states should be partitioned using one (or more) of the following methods:

1. Test each state once
2. Test the most common state transitions
3. Test the least common state transitions
4. Test all error states and error return transitions
5. Test random state transitions [3, p. 82-83]

For all of these tests, the values of the state variables should be verified [3, p. 83].

Requirements

- Knowledge of the different states of the program [3, p. 82]; could be obtained through:
  - The program's modules and/or functions
  - The program's exceptions
- Knowledge about the different state transitions [3, p. 82]; could be obtained through:
  - Testing the state transitions near the beginning of a workflow more?

Testing States to Fail [3, p. 84-87] The goal here is to try and put the program in a fail state by doing things that are out of the ordinary. These include:

- Race Conditions and Bad Timing [3, p. 85-86] (Is this relevant to our examples?)
- Repetition Testing: "doing the same operation over and over", potentially up to "thousands of attempts" [3, p. 86]
- Stress Testing: "running the software under less-than-ideal conditions" [3, p. 86]
- Load testing: running the software with as large of a load as possible (e.g., large inputs, many peripherals) [3, p. 86]

Requirements

- Repetition Testing: The types of operations that are likely to lead to errors when repeated (e.g., overwriting files?)
- Stress testing: can these be automated with pytest or are they outside our scope?
- Load testing: Knowledge about the types of inputs that could overload the system (e.g., upper bounds on values of certain types)

Other Black-Box Testing [3, p. 87-89]

- Act like an inexperienced user (likely cannot be generated by Drasil)
- Look for bugs where they've already been found (keep track of previous failed test cases?)
- Think like a hacker (is this out of scope?)
- Follow experience (this will implicitly be done just by using Drasil)

Chapter 4

Before Submitting

Just before submitting your thesis, you should make sure you've appropriately configured this template using the `manifest.tex` file. Specifically, you should make sure that no errors appear in the `build/thesis.log` file, `make debug` runs without issue, and that your PDF is visually appealing.

Happy hacking!

Bibliography

[1] Jacques Carette, Spencer Smith, Jason Balaci, Ting-Yu Wu, Samuel Crawford, Dong Chen, Dan Szymczak, Brooks MacLachlan, Dan Scime, and Maryyam Niazi. *Drasil*. Version v0.1-alpha. 2021-02-09. DOI: 10.5281/ZENODO.4526445. URL: https://github.com/JacquesCarette/Drasil/tree/v0.1-alpha (cit. on p. 6).

[2] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. 2nd ed. Boston, MA, USA: PWS Publishing Company, 1997. ISBN: 0-534-95425-1 (cit. on p. 8).

[3] Ron Patton. *Software Testing*. 2nd ed. Indianapolis, IN, USA: Sams Publishing, 2006. ISBN: 0-672-32798-8 (cit. on pp. 8–12).

Appendix

Source Code A.1: "MultiDefinitions" (MultiDefn) Definition

```haskell
-- | 'MultiDefn's are QDefinition factories, used for showing one
 ↪  or more ways
--   we can define a QDefinition.
data MultiDefn e = MultiDefn{
  -- | UID
  _rUid :: UID,
  -- | Underlying quantity it defines.
  _qd :: QuantityDict,
  -- | Explanation of the different ways we can define a quantity.
  _rDesc :: Sentence,
  -- | All possible ways we can define the related quantity.
  _rvs :: NE.NonEmpty (DefiningExpr e)
}
```

Source Code A.2: Pseudocode: Broken QuantityDict Chunk Retriever

```haskell
retrieveQD :: UID -> ChunkDB -> Maybe QuantityDict
retrieveQD u cdb = do
    (Chunk expectedQd) <- lookup u cdb
    pure expectedQd
```