Design and Critique of Fitchfork: a test-case based software automarker

Hannes Janse van Vuuren

October 2014

Abstract

This paper documents and criticizes the functionality and internal architecture of a test-case based automatic assessment system (automarker) called Fitchfork (FF). Fitchfork is currently (2014) used by the Department of Computer Science (CS) of the University of Pretoria (UP) to grade practical homework assignments and exams.

FF lacks certain functionalities and CS is having difficulty expanding the system. Replacing FF entirely has been considered, but no suitable alternatives could be identified to date. This document's analysis of FF indicates significant design erosion in the codebase. This erosion most likely developed as a consequence of years of incremental additions without code review, during which the clarity of the codebase gradually worsened.

This paper's primary purpose is to produce formal documentation for Fitchfork in order to support further maintenance as well as to inform CS department's decisions regarding the future of the system.

1 Introduction and problem statement

The University of Pretoria (UP) offers a number of courses on programming. For some of these courses an automatic assessment system called "Fitchfork" is used to grade students' submissions for programming assignments and examinations.

Fitchfork (or **FF** for short) was developed in-house by university students. The devolpment started in December 2005 and the system first went live in February 2006. Despite many difficulties with this system, it has since proven extremely useful to lecturers presenting programming courses in the CS department at UP.

Fitchfork is an old system that has never been thoroughly documented or reviewed, and although its operation seems simple in principle, the internal workings have proven difficult to understand and maintain because of severe design erosion:

Among other issues, the quality of FF's source code suffers from initially being written in a hurry before the user requirements were properly understood, and later being repeatedly modified without due care. The system also has no explicit documentation, and because of this the code can be

very difficult to grasp on its own, especially code and comments that have been neglected during updates to other sections of the program.

Software design erosion is a phenomenon studied in more detail by others such as van Gurp and Bosch [14]. It can occur whenever changes are repeatedly made to a working system in order to fit new requirements. It especially occurs when said changes only modify enough of the system to (partially) effect the desired outcome without a adequate rethinking of the design or updates to other related parts of a software system. For example: some functions of the system might be (partially) obsoleted by the new change, but since the system as a whole still works and the goal of a new feature implementation is reached, a programmer might not be inclined to "rip out and replace" the sections pertaining to that feature even though it would simplify the codebase.

Eroded designs often exhibit poor separation of concerns and unnecessary coupling between modules. This can increase the difficulty of understanding the way the system works, and therefore the effort of redesigning it in a more streamlined and logical way. At some point the effort of redesign becomes greater (at least in the short term) than the effort of simply "hacking" another change onto the system, so the erosion effect snowballs over time (similar to erosion in geography; there is also some similarity to the "broken window effect" observed in urban neighbourhoods [44] [41] – "if a window in a building is broken and is left unrepaired, all the rest of the windows will soon be broken").

FF is powerful as it stands, but lecturers have found that it sorely lacks certain features (Section 5). FF was not "designed for change", and features have been added and changed throughout the years. With each iteration the quality of the underlying design was not considered as important as implementing the features on a tight schedule. In general there isn't much structure to FF's maintenance/development cycle; as of writing features are still being added ad-lib without reviewing the decisions or applying test suites.

In short, a "minimal effort strategy" (to use van Gurp et al's term) has been the main approach to Fitchfork's development so far causing the software to "age" significantly. Detractors of the term "software ageing" argue that information cannot *age* – if a program is correct now it will remain correct forever. Parnas however explains that software ageing is a real and distinct phenomenon that does not explicitly occur over time but rather across changes to the software [27].

This "minimal effort strategy" and subsequent ageing has led to difficulty in implementing new features, modifying old ones, fixing bugs and security loopholes, and improving the design. This is evidenced by cases of the system breaking down during practical assignment sessions days after new modifications have been made. Some attempts have been made to remedy these underlying problems, but so far none have been particularly successful.

A decision has to be made regarding the future of Fitchfork. Although it isn't currently being superseded by another system Fitchfork can be consid-

ered an old or “legacy” system. The topic of dealing with legacy systems in a production environment is common in the software engineering industry [2].

Several different paths could be taken such as: replacing FF with a different existing package, developing a new automarker, making a single large refactor attempt, or redesigning Fitchfork over a longer period of time (progressive reengineering or “perfective maintenance” [11]).

Deciding on whether to reimplement or improve Fitchfork could be difficult [1]. Refactoring is an oft-recommended approach [22] but in some cases replacing software could be preferable to trying to improve the existing design, and some advise doing so on a periodic basis [4], although others caution against rewriting systems that could be refactored with less total effort [38].

This document makes no definite attempt at deciding the future of Fitchfork. Instead, it is aimed at providing sufficiently detailed information about Fitchfork’s design in order to help inform such a decision, and to support further maintenance or a rewrite.

1.1 Replacement

One solution might be to replace FF with a better, already existing system. No suitable candidate has been found and decided on so far. Fitchfork’s ability to grade assignments in a flexible manner using regular expressions makes it superior to many simpler automarking systems (Section 3.2).

The Department of Computer Science also prefer not to be constrained by the intellectual property rights and licensing fees of proprietary software, so any alternative to Fitchfork should be free and open source [18], or owned by CS department.

1.2 Reimplementation

Another solution may be to create a new and improved automarker system using the lessons that were learned from implementing, maintaining and using FF. This would be similar to treating the current implementation as a prototype. Prototyping is a software development approach where one or more prototype systems are created to help solidify understanding of user requirements before building the final system [8]. For the sake of cleaner architecture, such prototypes are intended to be discarded and not copied directly into the final system. Indeed, Fitchfork has helped CS Department gain a clearer understanding of what they need in an automarker.

The motivation for prototyping is often quoted from Brooks’ book “The mythical man-month” [3]: “Where a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time. Hence plan to throw one away; you will, anyhow”

It should be noted that Brooks has since reconsidered his stance that prototyping is virtually always applicable [21], however the approach may

be of merit in Fitchfork's case. Reimplementing FF is a solution that seems well within reach considering that FF's core is simple enough to understand: apply regular expression matches to some lines of output collected from a student's program and award marks accordingly.

However it would be no small endeavour as there is much more to the system than its core concept, and even simple "logistical" issues such as extracting and compiling a student's program are non-trivial (at least in the current implementation). Writing a new automarker to take FF's place could still be advantageous over improving the existing system if due care is taken: the new architecture could be designed without being bound to the flaws of the existing system [4].

2 Context of Fitchfork's use

As mentioned, FF is used to grade student submissions for programming tests. This description is vague and imprecise though: Fitchfork is designed for and used in a specific context.

FF is primarily used in first-year level imperative programming courses such as COS131, COS132 and COS110 in which students are taught to program in C or C++. These courses typically have hundreds of students registered, and they are expected to write and fix programs that are smaller than typical real-world applications and simple enough to grade automatically. Students may set up and use their own development environments to practice programming, but these courses teach and recommend using the Gnu C/C++ Compiler Toolchain in a Linux-based operating system environment.

Automarked tests (assignments, semester tests and exams) for these courses normally consist of two to five tasks, each with a description of the goal the student's program should accomplish and some example input and output. For example, students might be required to write a program which prompts the user for two numbers and then calculates and outputs the lowest common multiple of the two numbers.

Fitchfork can only assess programs based on what they read and write on the standard input, standard output and standard error streams. It is therefore well-suited to text-based "command-line" programs, and unable to assess event-driven programs with graphical user interfaces. This fact makes for a fairly simple interface between FF and the program being assessed.

Fitchfork receives student-submitted source code in compressed "tarball" archives from the web-based student interface (which is written in PHP and Python). FF extracts and compiles this code, and then invokes and takes control of the standard input and output streams of the resulting executable program. A set of memorandum files determine the input FF feeds to the student's program, and also a regular expression structure against which the program's output is compared for complete or partial correctness.

After determining the total score for a submission, FF returns this score to the web interface infrastructure that initially invoked it. The actual management and storage of student score data is not part of Fitchfork itself.

2.1 The name

The name “Fitchfork” is a bit unusual. According to one of its original development team’s members, the name started as an in-joke among the development team after one of them misspelled “pitchfork” in an IRC conversation with the others. The word stuck, and the project was eventually dubbed “Fitchfork”. According to another anecdote, the system originally ran on a server in UP’s network named HAL (after the fictional Heuristically-programmed ALgorithmic computer from Arthur C Clarke’s novel *2001: A Space Odyssey*), and was humorously referred to as “[p]itchfork from hell” perhaps in part due to the perceived rigidness in the way it awards marks (this is discussed, among other things, in section 6.4).

3 Related work

3.1 Automatic assessment of programming assignments

Automatic assessment (‘AA’ or ‘automarking’) tools for programming assignments can save a lot of time for those who teach programming, often to the point of making marking of otherwise intractably large assignments feasible. They can also otherwise augment the pedagogical value of programming courses [5] [9]. When tutors need to spend less time marking, they can spend more time interacting with students and explaining difficult sections. Enabling students to work on larger assignments with access to marks shortly after a submission can help them gain more experience than they would have working only on small projects and code snippets that get marked manually.

As noted by researchers such as Douce et al[9] and Pieterse[29] using an automarker certainly does not come without difficulties. Programming assignments, especially more complex ones, tend to be very open-ended and as such it can be challenging to set up adequately lenient automarking memoranda for them, assuming that it is even possible to assess the given assignment with an automarker.

In their assessment of then recent (2006-2010) automarking tools for programming assignments [17], Ihantola et al compare aspects of different tools and conclude that there are too many new, underdeveloped automarkers being ‘reinvented’ and too few well-tested, comprehensive ones available. Fitchfork arguably falls into the former category. It suffers from the same hurried initial creation and lack of serious maintenance effort that Ihantola identify as a common cause for the incomprehensiveness of many automarking tools. However, Fitchfork does have some features that distinguishes it from other similar tools (Section 3.2).

Pieterse discussed automatic assessment's role in a MOOC context and specifically analysed Fitchfork: [29] highlights some of the requirements for a good automarker, and issues that can occur with automarker use. Two significant issues experienced with Fitchfork in particular are: terse feedback to students and lecturers, and difficulty in setting up memos that are adequately lenient (especially with regard to otherwise insignificant output formatting alternatives) despite the fact that regular expressions can be used in the specification of memoranda.

3.2 Automarker requirements

Between 2013 and 2014, the possibility of replacing Fitchfork with a similar existing system was investigated by staff at UP Computer Science department. A list of criteria by which to compare systems was set up. Some factors that are more difficult to define explicitly also played a role in the decision making process, such as estimated effort required to install a system and the magnitude of uncertainty regarding how thoroughly that system implements certain features. Therefore, the fact that Fitchfork is well known by the department's staff and also known to work well with the university's network weighed in its favour.

Other systems were tested and analysed to various extents. These include: Sharif Judge [35], BOSS [19], peach[3] [28], Sphere Online Judge (spoj) [37], The SIO2 Project [36], Web Cat [39], Ceilidh [13], Codejudge [7], Kattis [26], and EasyAccept [10]. Further details regarding why each system was rejected could not be acquired at the time of writing though, and thorough investigation of the matter lies beyond the scope of this document.

"Non-negotiable" criteria determined by CS are:

1. Support for C and C++
2. Ability to integrate with CS department's publicly accessible website.
3. Students should be able to upload submissions from home (via the CS website).
4. Strict security measures to prevent a program submitted by a user to perform any malicious actions deliberately or by accident.
5. Specification of output-matching criteria using POSIX Extended Regular Expressions (or similar). [30]
6. Multiple submissions (limited to a number specified in the memorandum) must be possible.
7. Automatically storing backups of all student submissions.
8. High degree of availability (proprietary and externally-served systems might suffer from server downtime, or disappear from the market entirely whereas small open source projects risk stagnation)
9. Ability to specify time-out limits in memoranda (ie. limits on the amount of time a compiled submission is allowed to run before termination)

Fitchfork conforms to most of these requirements, except point number three to an extent. Security fixes are an ongoing matter, but several flaws were identified at the time of writing 5. One of the strongest points in FF's favour when compared to most other systems is number four. The last two points also disqualified several systems simply because they are provided only as subscription services which reside entirely outside the university network, or because they are too difficult to integrate with the CS website.

Other factors were also considered. Some points are followed by remarks regarding Fitchfork's performance in that aspect.

10. Effort required to specify test data and criteria for marking. (Fitchfork can be seen as intermediate in this regard; the XML memo format is generally not difficult to use, but breaking assignments into separate tasks [to provide finer-grained marking in cases where compile-time errors are likely] requires effort)
11. What happens with existing submissions when a memo is changed after submissions were made? Delete/keep/reassess? (Fitchfork plainly deletes all submissions when a memo is edited)
12. Is the system likely to be improved in the future, and is it possible for CS department to participate directly in its improvement?
13. What kinds of feedback (regarding errors in memoranda) are provided to instructors while setting up assignments? (Fitchfork's web front-end provides usable feedback in most cases. It breaks down completely in some cases, providing only a blank screen to hint that something went wrong)
14. What kinds of feedback to students can be specified in the memo? (Fitchfork's feedback is generally too sparse. Recent improvements allow limited feedback based on whether certain output was matched or not)
15. Can alternative answers with unequal number of output lines be specified? (with some modification, Fitchfork should be capable of this)
16. Can the system evaluate code in terms of static code analysis using metrics that correspond to complexity and style? (FF has very limited support in this regard: it can only scan and reject files for using taboo header inclusions and function calls)
17. Can the system identify the use of specific programming techniques such as the use of recursion, static variables, global variables, global constants, etc. (Fitchfork has some support for detecting recursion)
18. Can the system identify certain bad coding practices?
19. Can the system restrict access to specified libraries or functions? (FF supports this to some extent by restricting access to certain C/C++ header files)
20. Does the system support simple plagiarism detection such as MD5-sum or simple diff-comparison within a group? (Some support scripts provide these features to lecturers, but this is not part of FF itself)

21. Does the system support or integrate with another system for advanced plagiarism detection? (As with the previous point; CS dept. uses Stanford University's MOSS system [23])
22. What other programming languages are supported? (FF can in theory support any language that provides standard output since recent modifications)
23. Is it possible to specify a formula for calculating marks in cases where multiple submissions are allowed, eg. best / worst / first / last / average / weighted? (FF always grades according to last submission)
24. What statistics can be gathered for and provided to students re. their own performance and performance in relation to the group? (FF lacks such features)
25. What statistics are gathered for instructors' use? (FF has only rudimentary support)

3.3 Ruby scripting

Fitchfork is implemented entirely in Ruby. The 'runlimit' tool it uses to impose limits on student programs is written in C.

Ruby is a dynamic, reflective, object-oriented, general-purpose programming language. It was initially designed and developed in the mid-1990s by Yukihiro Matsumoto in Japan [43].

According to its authors, Ruby was influenced by Perl, Smalltalk, Eiffel, Ada, and Lisp. It supports multiple programming paradigms, including functional, object-oriented, and imperative. It also has a dynamic type system and automatic memory management [43].

Ruby is used in many successful projects ranging from scientific research to web applications [33]. Some of these projects are very large and popular eg. the Ruby on Rails web application framework (which is used by Twitter and Github, among others [32]).

I believe that an interpreted language such as Ruby which supports reflection, automatic memory management, and detailed crash traces is well-suited to an application like Fitchfork which isn't primarily performance-critical (as opposed to a less dynamic but faster compiled language such as C++)

3.4 Separation of concerns and the DRY principle

Two of the main issues that underlie many of Fitchfork's architectural problems as pointed out in this analysis are:

- Poorly separated concerns
- Duplication or near-duplication of functionality

"Separation of concerns" is a well-known software engineering principle which has been discussed at length by researches and practitioners alike [16] [12] [24]. The basic idea behind it is that a system should – as far as

possible and sensible – be factored into individually cohesive, orthogonal, individually configurable, reusable components. A 'component' in this context could be a class, function, module, utility program, etc. that may be composed of other sub-components. The point is that different components interact to form the whole, but each component specializes in performing its own job without being *concerned* with the inner details of other components.

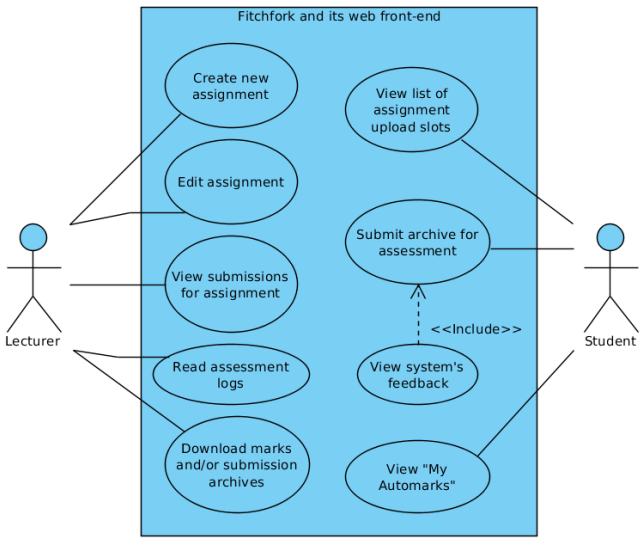
A related idea is that of minimizing repetition of functionality and information. This piece of common software engineering knowledge is presented more formally by Hunt and Thomas [15] as the Don't Repeat Yourself (DRY) principle. The principle not only concerns avoiding repetition of code, but of information in general, as mentioned by Thomas in an interview [42]. Unnecessary duplication in software has several disadvantages, the most obvious being an increase in maintenance complexity.

4 Design and critique

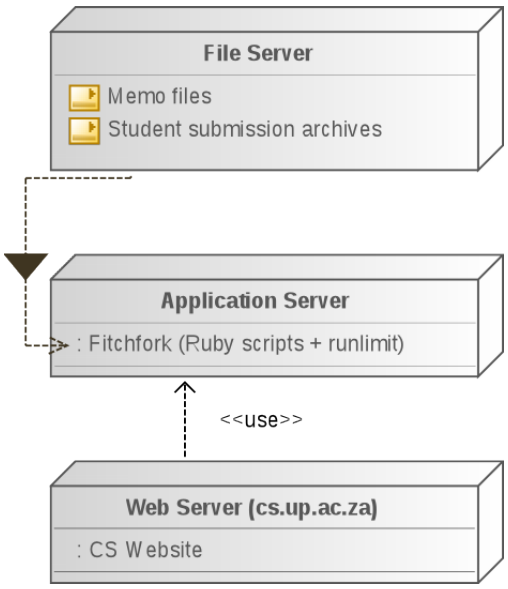
This section comprises the core of this document. It describes Fitchfork's internal architecture on a point-by-point basis. In contrast, section 6 explains Fitchfork's internals from an execution-flow perspective.

4.1 Overview

(a) Use Case diagram



(b) Deployment diagram



The web front-end that manages Fitchfork (hosted by the CS department's internal servers) distinguishes between two types of users: Lecturers and Students. Via the relevant course's web page to which they've been granted access, only lecturers may do the following:

- Create new assignments by posting an XML memo file with the assignment's name and deadline information.
- Edit the details of any assignment.
- View the marks and an "output trace" of the marking process of each student's marks.
- Download concatenated Comma Separated Value files that map student numbers to final marks for selected assignments.

Only students may do the following via the web page for a course that they are registered in:

- View the list of assignment upload slots available to them.
- Submit an assignment.
- View a page containing FF's feedback and marks awarded for all their assignments so far.

4.2 Fitchfork's memo XML file format

The technical analysis and critique that follows assumes working knowledge of FF's features the memo XML format.

Lecturers interact with most of FF's features by writing memorandum XML (Extensible Markup Language) files. Most sections of the codebase are at least indirectly related to one or more of the tags that can be used in these files.

Unfortunately there is no formal documentation and full knowledge of each element's behaviour and caveats can only be attained by actually using the system for some time. There is, however, a "reference example document" that was made some time ago (`ff-config.xml` in `fitchforkDocumentation.zip`) [6]. This document is somewhat out of date but still compatible with current Fitchfork, and demonstrates the most important features.

4.3 Fitchfork's modules

Below is a summary of the files that constitute Fitchfork. These files all occur in a single directory on the server on which FF runs.

Responsibilities often overlap between FF's modules and classes (poor cohesion and decoupling), therefore summarizing the responsibilities of each specific file is not necessarily straightforward. This overview should be enough to help a developer find their way with the system.

This section is followed by criticism on a per-module basis, followed by a more detailed look at certain aspects such as the file system and the most important modules.

The files are listed approximately in order of the flow of execution when FF marks an assignment submission.

marker.c A small C program that accepts three arguments and invokes and passes along these arguments to `marker.rb`. Besides that it sets `RUBYLIB` environment variable to a hardcoded constant (`RUBYLIB` counter-intuitively simply contains the directory in which all FF's Ruby scripts are placed, including `marker.c`). It also sets the effective and real user IDs of the process (ie. itself and `marker.rb`) to zero (ie. root).

make_marker_wrapper.sh Shell script. Compiles `marker.c` and ensures that `marker.rb` has a specific `#!` ('shebang') line at the beginning to invoke Ruby.

marker Executable produced by compiling `marker.c`

marker.rb Program entry point for marking an assignment; invokes the other modules. Requires three command line arguments: name of student upload archive, name of memo XML file, one-time-password used when transmitting calculated mark to mark server. Like other parts of FF it parses info about the course, student number, practical, upload number etc. directly from the names of the archive and XML files, so these file names have to be given in a specific format by whatever program invokes FF.

config.txt Fitchfork's configuration file. Contains `key=value` pairs. 'Debug mode' can be turned on from here. Configuration is accessible to the rest of the program via the global array variable `config`.

config.rb This file loads `config.txt` and populates the global variable `config` with its contents if it doesn't exist yet.

sourcefilter.rb **`class`** `SourceFilter` contains a function to check student-submitted C++ code for any `fork()` calls. It is one of the parts of FF that is bound to C++ specifically. Contrary to expectation, this module does *not* check for the inclusion of disallowed ('tabu') header files (which is done by **`class`** `Assignment`).

runlimit.c Source code for the `runlimit` sandboxing utility.

memo.rb **`class`** `Memo` is used for parsing (read, interpret, error-check) the assignment memo XML file. During this process it also sets up the temporary directories for sandboxing a submission. It contains a host of similar methods, one for each 'section' of the XML file. Interestingly however, these methods are almost entirely for error- and form-checking purposes. This class uses Ruby's REXML module to get a tree structure that holds the contents of the memo XML, but doesn't copy most of that data into appropriate data structures*; the rest of the program uses the REXML structure directly (code that makes use of memo data eg. in the sandbox module is written to manipulate the REXML object and its subobjects)

*The selection-matching part of the memo, (against which output is marked) is the exception to this: it is stored as a tree of objects specific to the purpose. The classes for these objects are defined in `memotree.rb`

memotree.rb Contains several classes that derive from an abstract base MemoNode. Each class corresponds with a memo XML element type, although some elements are represented by different classes for each parent/child context in which the element may occur. Each of these classes primarily cover two responsibilities: reading the data from the XML file and recursively building the corresponding object tree (`parse` method) and using this data to recursively evaluate and award marks to a given section of student program output (`evaluate` method).

assignment.rb **`class`** `Assignment` performs most of the duties concerning an uploaded assignment. Each Assignment uses a single Memo and single SourceFilter object. It splits everything into six tasks: archive extract, 'canonicalise', validate, filter, compile, execute. Although an Assignment object keeps track of whether it has been executed, this class doesn't actually perform that final step. The responsibility of execution is instead left to the Sandbox class. The other tasks roughly involve the following:

Extract: create the appropriate working directory ("`student_dir`") and extract the uploaded archive into it.

Canonicalise: First 'sanitize' `student_dir` by deleting files specified in the memo's `<delete-patterns>` tag. Then recursively uses `chmod` to ensure all files in `student_dir` are readable and writable. Then copy the files specified in `<extra-files>` into `student_dir`.

Validate: Scan student source files for tabu `#include` statements. Files are identified as source files through the regexes defined in `<source-patterns>`, so if a file contains source code but its name doesn't match one of those regexes, it isn't scanned (`assignment:321`; this seems to be the only use of `<source-patterns>`)

Filter: Uses a SourceFilter object to check student source files for `fork` calls. Suffers from the same 'weakness' as the previous step.

Compile: Invokes the first makefile found in `student_dir` and checks whether the executable was created correctly. A recent security fix made to FF ensures that this process occurs inside a chroot jail (student makefiles were run as root *outside* a chroot jail in the past!) but for the sake of consistency this investigation was kept mostly to a single version of FF from before this fix.

sandbox.rb **`class`** `Sandbox` contains functions for creating the 'sandbox' (chroot-jail environment) in which the student's executable will be run, and for actually running it via the runlimit utility program (`Sandbox::run`). Interestingly this class holds almost no state; its methods manipulate an Assignment object passed *as an argument* to every one of its methods.

The following source files support the ones listed above:

- **logging.rb** Small file that sets up a Ruby `Logger` object as the global object `log`. Used by `marker.rb` to write to the logfile specified in config.
- **tag.rb** Mixes methods directly into **class** `REXML::Element` that make it more convenient for the programmer to use that class in the rest of the code. Only gets included by `memo.rb`. [1]
- **metachars.rb** Adds convenience methods to Ruby's `String` class for escaping and unescaping certain characters with backslashes. Used for security purposes to 'sanitize' arguments in shell commands that FF executes, and strings sent to the web server.
- **filefinder.rb** Utility filesystem functions incl. recursively finding (by regex), deleting and changing permissions on files. Used by `assignment.rb` and `sandbox.rb`.
- **fitchfork_daemon.rb** Either a no longer working or abandoned script that was apparently intended to start Fitchfork's marker as a background process (daemon). Its code attempts to instantiate a non-existent **class** `Marker`.
- **parsememo.rb** Program entry point independent of `marker.rb`. This script uses `memo.rb` to check whether a given memo XML file is valid. It only outputs a 1 (invalid) or 0 (valid), besides the output generated by memo parsing. If an error in the XML results in an (REXML or other) exception, `parsememo.rb` doesn't output anything itself (it never catches the error and a stack trace is dumped instead). This script is possibly *not* the one used for verification upon memo upload by a lecturer – the web infrastructure is beyond the scope of this investigation.
- **manual_mark.rb** A simple script that invokes `marker.rb` for each assignment archive in a given directory.
- **treediagram.rb** No longer working script that generates Graphviz DOT markup for a diagram to visualize a given memo's matching/alternatives tree.
- **passwd.rb** Utility functions for mapping user and group names to uid's and gid's. Reads the `/etc/passwd` and `/etc/group` files.
- **recursivecheck.rb** **class** `RecursiveCheck` can be used to ascertain whether a given function in a piece of C/C++ source code is recursive. This file gets included by `memo.rb` but (as of writing) the class is never used; it only ever gets instantiated by test case scripts. It (redundantly) features code for eg. filtering comments from the code that is to be analysed, something that can rather be done with a correct invocation of a compiler's preprocessor.
- **leakcheck.rb** **class** `LeakCheck` has a `check` function that uses Valgrind's `memcheck` to ascertain whether a given executable leaks dynamic memory. It gets used in `Assignment::mark_leak`, a method that never gets

[1] Adding methods to already existing classes ('mixing in') is a feature of the Ruby programming language.

called. Note that checking leaks with `memcheck` requires executing the subject program, and `leakcheck.rb` does this (or rather: would do it) outside of any sandbox/chroot-jail. According to comments in its source, `runlimit` itself causes some memory leaks; this would make it hard to judge the presence of memory leaks in a student's program if `runlimit` (which invokes the student program) is invoked from `memcheck` .

- **`README.md`** Mandatory readme file for Bitbucket. Explains the git workflow to be used when working on FF. One of the maintainers wrote in it: *"Abandon all hope, ye who enter here."*
- **`sandbox_old.rb`** Apparently an older sandbox implementation kept around while the new one was being developed.

The following scripts are self-contained test cases for Fitchfork, using Ruby's unit test class (`require 'test/unit'`). They no longer seem to work.

- **`unittest.rb`** A unit test file, apparently not related to the others.
- **`test.rb`** Invokes each of the following three Ruby scripts.
- **`test_memo.rb`**
- **`test_sandbox.rb`**
- **`test_sourcefilter.rb`**

The following test scripts just invoke the marker with various no longer existing memo files.

- **`test_assignment2.rb`**
- **`test_prac2Monday.rb`**
- **`test_prac2Thursday.rb`**
- **`test_prac2Tuesday.rb`**
- **`test_recursive.rb`**

4.4 Criticism regarding each component

4.4.1 marker.c

It is not clear *why* `marker.c` exists at all: it merely sets `RUBYLIB` environment variable (l42), tries to set both effective user id and real user id to root (l43), and invokes `marker.rb`. `marker.c` doesn't check whether the `setreuid` call has failed, and performs minor but rather blatant out-of-bounds access on a static array (l39). For a simple task such as setting an environment variable, a shell or Ruby script would be more suitable (and require no recompilation).

The `RUBYLIB` env var is like a `$PATH` variable containing a list of directories Ruby will scan for .rb files when `require` is used. Since all of FF's files are in a single directory, `RUBYLIB` can be set to '.' (directory of current script) to avoid the need to set this env var when installing or moving FF.

4.4.2 Configuration and directory system

The presence of a plaintext config file in the system is definitely a positive thing. It even supports using `#` for writing comments in the file, which is helpful.

However, there is no reference config file commented with an explanation of each option. The only (vague) documentation of `config.txt` is in the comments in `config.rb` (l20) and those are vague (several comments read "relative to assignment dir" but no indication is given as to which other dir option is the "assignment dir").

4.4.3 assignment.rb and friends

The idea that an assignment (-submission) should be represented by a class is logical.

There are several architecturally-not-logical things that occur in and around this class. Notably:

- The existence of `Sandbox`, a **`class`** with no state.
- The distinction between 'validate' and 'filter'; both do a text-search on code to check for illegal tokens.

`class` `Sandbox` basically contains methods that should have been in **`class`** `Assignment`: every one of its methods take an Assignment object as the first parameter and uses that object extensively. Sandbox has virtually no state: it doesn't even store a reference to an Assignment object (it does however store a string which is passed to the shell to invoke a student program, but that doesn't even need to be stored). It is the C++ equivalent of a hollow helper class declared as a `friend` of Assignment, something that easily provokes eyebrows raised among modern C++ developers.

It is understandable that the code for sandbox/execution phase be maintained in a separate file, but it would be better to make **`class`** `Sandbox` a module (stateless construct that holds functions, constants, etc. in Ruby) instead of a class, or to make its functions part of **`class`** `Assignment` (it is still entirely possible to keep the functions in a separate file though).

As noted elsewhere, checking for 'tabu' file `#includes` ('validate' step) is not an entirely effective way to disallow certain library functionality from being accessed. Rather, those libraries should not be available during compilation or execution at all, or/and extra limits (such as a zero cap on TCP connections) should be imposed on student programs via operating system facilities (in that case everything could be simplified by discarding code-scanning code).

The 'filter' step might be an effective way to prevent students from submitting fork bombs, although as it stands it can likely be circumvented by constructing and calling a pointer to the `fork` function (code that assigns fork to a pointer wouldn't match the regular expressions used by source filter). Again, it might be better to employ operating system calls that disallow process forking instead.

If these two parts of FF aren't replaced entirely, they should at least be merged into one class or module because 'filter' and 'validate' consist of the same basic operation (although implemented differently): searching the code for certain disallowed strings.

Another thing is that Sourcefilter is very restricted in its functionality as its stands, for no good reason: it can only scan for fork calls, even though its code is written in a way that suggests having other options in mind. If the memo writer wished to disallow calls to a particular function other than fork, it should be possible. SourceFilter's code would allow this without much alteration.

However, it is unclear whether SourceFilter is a class at all, since it doesn't keep any significant state information. It would seem that SourceFilter is a failed or incomplete attempt at the Method Object pattern.

4.4.4 C++ dependencies

C++ is definitely not the only language that the University teaches, so it is reasonable to expect a tool like Fitchfork to be easy to adapt for assessing other languages. As it stands, it is in fact possible to automark Assembly assignments with it (and of course C too). Using makefiles for the compilation process and for initiating the execution process certainly lends a lot of flexibility to the system in this regard.

However, as noted some components are hard-coded to work with C++: **`class`** `SourceFilter` (l48) and **`class`** `Assignment`'s tabu-inclusion checker (l321) specifically invoke the GNU C++ compiler/preprocessor (g++). **`class`** `RecursiveCheck` doesn't, but relies on the assumption that the code it scans is C/C++. It is not strictly correct to use g++ for scanning C source (although the chance of trouble is low in this case), and it is definitely not correct to use g++ on assembly files or most other languages' source. Ideally these components should be implemented in a more dynamic way to make it easy to switch between languages and their compilers/tools on a per-memo basis.

A more general language-lock-in issue is that FF assumes that after the make process the product is in fact some executable file (the name of this file is specified in the memo with the `<executable>` tag). This may not be the case with interpreted languages, although in many cases a shebang line and a raised executable bit may be all that's needed – all perfectly doable from within a makefile.

A language like Java presented a bigger problem: running a Java app requires invoking `java` with the right flags instead of the `.jar` file itself. Recent modifications to FF allow this by requiring that the makefiles in question contain a `run` target. FF now invokes submitted programs via `make run` instead of `./executable`. This rather obsoletes the `<executable>` XML tag, but it is still present in the code *and required* for a memo to work even though it makes little sense for non-compiled programs.

4.4.5 Memo parser

`memo.rb` might be the least elegant part of Fitchfork's code, and the least interesting.

Much of its function consists of checking for semantic errors, and the code for it is quite repetitive: the module itself is 429 lines long, and does nothing else. Fortunately Ruby's REXML module employed here already does most of the work regarding parsing the XML and detecting syntactic errors.

REXML throws exceptions in such cases though, and `memo.rb` never catches these but instead lets them bubble up to the program entry point (the only occurrence in all of FF of the **`rescue`** keyword is in `marker.rb`).

Most of the semantic and form-validation code in `marker.rb` could be drastically shortened by using an XML Schema checking library such as RXSD [34] or Nokogiri [25]. Schema checking libraries and code generators were invented to solve this particular problem of coders having to write swathes of code just to validate the form and content of an XML file before using it.

It's a bit odd that **`class`** `Memo` contains the `configure_working_directories` method (l119). This method doesn't relate to parsing but to using the parsed data. It always gets called by `parse` after the memo's name is extracted (l377). This violates the principle of separating the concerns of extracting information (parsing) and acting on it (creating the relevant directories), making it impossible to use this memo parsing code for simply checking a memo's validity. It also violates this principle in an inconsistent manner, since not all of the file setup work is done during parsing (eg. `Sandbox::configure_extra_files` which is called from `memo.rb` via `Sandbox::setup`)

Another point of criticism is that **`class`** `Memo` doesn't entirely isolate the extracted REXML structure's type semantics (such as it is) from the rest of Fitchfork. Some information such as `@title` is stored as a simple string, as would be logical. But data such as `@extra_files` gets stored as a list of REXML document elements instead of eg. a list of strings or hashes. This extends Memo's dependency on REXML to other modules that use memo info (a text search for ".text" in `assignment.rb` and `sandbox.rb` demonstrates several cases).

4.4.6 memotree.rb

`memotree.rb` contains the classes that represent the actual regular expression tree (subelements of `<case>` besides `<arguments>`). This is the only part of the XML that expands to a tree of objects of variable size (although there can also be many `<output>` tags, the contents of only one is read during any invocation of FF). In a way, `memotree.rb` contains the heart of Fitchfork.

`memotree.rb` contains an abstract base class `MemoNode` and nine other classes, each representing a tag in a specific parent-child context. These classes contain functions for co-recursively constructing such a tree starting with a root REXML element (**`def`** `parse`) and functions for recursively evaluating student program output against the constructed tree (**`def`** `evaluate`).

Oddly these classes also contain functions that would apparently be used by `treediagram.rb` to generate GraphViz dot output (**`def`** `dot_opts`). While this feature could be very helpful for debugging FF, there aren't many convincing reasons for these trivial functions to be members of these classes. `treediagram.rb` could accomplish the same while keeping the relevant code in one place by simply using a hash that maps MemoTree classes to dot format strings.

This module performs its function fairly well. However it has some problems:

- There is a need for a mechanism for coping with line-misaligned program output.
- Some XML tags are represented by two different classes (for different parent-child contexts). This is not necessarily a big problem, but such redundancy doesn't indicate good architecture either.
- The classes in it are extremely similar anyway, leaving the module at about 730 lines long. A lot of this redundancy could potentially be factored out by re-thinking the class structure. That would make the module easier to understand and maintain in the future (although admittedly no bugs have been discovered in this module recently, so refactoring it is not priority). One suggestion described later is using only three classes to represent containment/grouping, alternative-selection and matching nodes, which are the three essential roles into which the relevant XML tags fall.

4.4.7 Dubious sandbox security

`class` `Sandbox` uses a third-party `runlimit` tool to run the student program if the `no_chroot` option isn't set in FF's config file. There are several issues with this tool and how it is used:

- It has not been stress-tested and verified to be a secure tool.
- It has not been updated since it was first used in Fitchfork.
- It does not limit network connections or access to existing processes on the system, which makes it very inadequate from a security perspective. [20]
- Some of the parameters passed to the tool are hard-coded in Fitchfork and are likely not optimal (`sandbox.rb:116`).

4.5 File system

During the entire process of upload to output marking, Fitchfork works with a certain directory structure. The paths for the directories used can be specified in `config.txt`

Nine directories are specified: `tmpdir`, `memodir`, `jaildir`, `outputdir`, `syc_dir`, `archive_dir`, `extra_dir`, `done_dir`. There is also `student_dir`, but it is never defined in the config file, only in Assignment (`assignment.rb:37`).

The config file and `config.rb` contain no documentation or comment that explicitly states the purpose of each directory.

4.5.1 Upload file names

When Fitchfork is invoked, `marker.rb` accepts only three arguments: name of the upload archive, name of the memo XML file , and a one-time password to be used when sending the mark information to the relevant server.

Note that the student's ID, course code, memo name etc. are not passed to it explicitly. Instead, FF makes several assumptions (`marker.rb:38`):

- The archive's file name has to be in a specific format the indicates the student number, course, etc.
- The memo XML file name is in a similar format.
- That the above and related files (such as those specified by the `<extra-files>` tag) are located in specific subdirectories whose names depend on the course code, assignment number, etc.

4.5.2 Directory structure

- `memodir`: FF assumes that all memo XML can be found in this directory. It is specified as an absolute directory in `config.txt`.
- `tmpdir`: Absolute directory. Also called the 'assignment dir' in source comments. Serves as the root directory for most of the processing. Most other directories are specified relative to this one.
- `student_dir`: relative to `tmpdir`. Its value is specific to each submission: `class Asssignment` initializes it to `tmpdir + @student`, where @student is a string that contains the student number (as passed to `Assignment::initialize`, which is in turn derived from the filename of the submitted archive elsewhere in FF; see criticism below)
- `jaildir`: relative to `student_dir`. This is the directory into which student programs are 'chroot jailed' during execution.
- `outputdir`: relative to `jaildir`. Student programs' output is collected into files in this directory during execution.
- `archive_dir`: relative to `tmpdir`. FF assumes that submission archives are uploaded to this directory. The names of the uploaded archives have to be in a specific format.
- `extra_dir`: relative to `tmpdir`. FF assumes that extra files that accompany each memo are located in this directory.
- `done_dir`: relative to `tmpdir`. `Assignment::cleanup` moves the uploaded archive to this directory (`assignment.rb:157`). Note that `cleanup` never gets called, so FF currently doesn't use this directory.
- `syc_dir`: relative to `tmpdir`. `Assignment::cleanup` archives the contents of `outputdir` and places it in an archive in `syc_dir` (`assignment.rb:137`).

4.5.3 Criticism of the directory structure setup

When first investigating the system (for development or just to get it running), it can be difficult to figure out what goes where. `config.rb` can definitely do with some comments that quickly help the reader in the right direction, as this is one of the first files someone is likely to read for this information. An even better idea might be a heavily commented sample configuration file that explains each directory's purpose (including such a config file that helps an admin set things up is quite common in large systems)

It seems that whether the directories are relative or absolute is chosen somewhat arbitrarily. The use of underscores in the names is also inconsistent.

Another annoyance is that concatenating relative directories with eg. `tmp_dir` isn't done in one central place in the code (and is always done with a slash in between – not all operating systems use / as a directory separator). Instead, concatenation is done and re-done all through `assignment.rb` and `memo.rb`. This makes it difficult to tell at a glance how each directory is used.

A simple solution to this might be to require specifying all directories with absolute paths in the config file.

It is a little peculiar that default values for `jaildir` and `outputdir` are prefixed with a dot and therefore considered "hidden by default" (`config.rb :24`). On a normal *nix-based filesystem, there is no such thing as a 'hidden file' (some files can be made inaccessible via permissions, or they could be 'hidden' inside directories not accessible to all users, but that is different). It is merely convention that files whose names start with a dot are not, by default, listed by utilities like `ls` or included in shell expansions such as `*.txt`. It is possible that whoever designed the directory system of FF was under the impression that prefixing these directory names with a dot would somehow (?) restrict access to their contents.

Details such as the student number, course number and assignment part number are never passed directly to Fitchfork upon invocation, but rather encoded in the file paths of the archive and memo. Fitchfork therefore depends on the web front-end that invokes it to place and name these files according to this implicit (and therefore fragile) naming scheme. This somewhat complicates test runs of Fitchfork.

This format is also difficult to parse at a glance as it involves numeric codes for courses different from those used by the rest of the university. For example, this is the name of an archive from the COS110 course:
`u13059926.1.120.673.3.tar.gz`. Everything before the first dot is clearly a student number, but the rest is hard to tell.

Code that parses these file names occurs in more than one place, violating cohesion and the DRY principle.

4.6 class Sandbox (sandbox.rb)

class Sandbox essentially contains the sandbox and execution related methods of Assignment. The class has four methods besides initialize and a accessor/mutator pair. All four accept an Assignment object as the first argument.

The entire module performs slightly different actions depending on whether $config['no_chroot'] is true or false. If it is false, then it is required that FF be run as root. It is unclear whether setting it to true actually lifts this restriction, as runlimit gets told to chroot into jaildir anyway (l115).

def setup(ass) "Sandboxing" is the process of creating a directory in which the student executable will be run. Note that this is done after compilation (inside Assignment::compile). Like other methods, setup takes an Assignment object as an argument and operates on it.

setup starts on sandbox.rb:70 and first does some sanity checks. At this point ass.initialize should already have executed. Note that line 74 is not simply a sanity check: it is (most likely) the first invocation of Assignment::executable, a read-accessor that invokes the other five steps in **class** Assignment ie. extraction to compilation!

setup stores the shell command ./executable-name in attribute @exec, stripping shell special characters from it (it is unclear whether the program will therefore be executed at all if its name contains such characters).

setup then creates jaildir inside student_dir and subdirectory outputdir inside it (see assignment.rb:38-39), and copies the student executable into jaildir. Next it invokes configure_extra_files.

def configure_extra_files(ass) Copies each file that should be in jaildir (<extra-files> in XML) from student_dir. The comment on l20 can be a bit confusing: all it actually indicates is that this code simply copies from ass.student_dir because any files that should have been overwritten there at this point already have been because of Assignment::canonicalise

It also sets each copied file to read-only or writeable. Note however that student programs can do the same once they execute.

def configure_input_files(ass, kase) Returns the base name of the <extra-files> file that should be used as the student program's input for a given <case> (kase). It silently returns **nil** if no input file is specified, and silently returns the first one if more than one is specified in the XML.

def run(ass, kase) Executes student executable for the specific case. Calls configure_input_files to get the name of the input file. Execution is performed chroot-jailed inside jaildir (and with other limitations) via the runlimit utility program (path to it specified by $config['runlimit']). The parameters passed to runlimit specified on l116 could be made configurable.

This method forks FF's execution on line 119 with an IO.popen call. The purpose of this is not entirely clear: forking here doesn't do anything that couldn't be done otherwise, and uses more resources as the entire Ruby interpreter state is duplicated upon this kind of fork.

The parent process simply reads the contents of the relevant inputfile and sends it down the pipe to the child, and reads runlimit's output from the pipe. The child invokes runlimit with Ruby's backtick syntax and prints its

output to the parent. Within the backtick-syntax invocation it uses the `2>&1 >` syntax to redirect all output of the student program to `outputdir + kase`. Note that this causes a subshell to be invoked, causing additional overhead.

The fork ends on l149, after which the output of execution is read from the relevant file and stored in `ass.output[kase]`. Information about the run (number of bytes produced etc) is logged and a simple regular expression match is used to determine whether everything executed with or without warnings (the 'Execute SUCCESS' line). It is probably possible for the executable to not execute at all, but in that case it will simply be detected as if it ran with warnings.

4.7 class Assignment (assignment.rb)

This is Fitchfork's largest module byte-wise, and a very central one architecture-wise.

An instance of **`class`** `Assignment` represents a single student submission for a specific assignment that needs grading. The Assignment class acts as a central point where the other classes' actions meet (Memo, MemoTree, SourceFilter and Sandbox) and is therefore somewhat 'faceted': some of its attributes are important to Sandbox, some to Memo, and some to the main program.

Assignment's main duties are the five steps: extract, canonicalise, validate, filter, compile. These steps are explained in the summary of modules above (4.3) and to some extent in the program flow investigation (6.2). Each of these five steps are represented by a member function. The final step before marking is execution (during which the student executable's output is captured), but it is performed by **`class`** `Sandbox` even though the results of execution are stored inside Assignment.

4.7.1 Triggering the process

These methods are all declared private to the class. The only way to access them from the outside is by invoking *the read accessor* `executable`. This can be very perplexing when one first tries to trace the flow of execution. Read accessors are not commonly used to trigger large, core amounts of processing, but `executable` invokes the entire cascade of steps from extraction to compilation when invoked for the first time. Upon subsequent calls it simply acts as a regular read accessor: it returns the path to the compiled student executable relative to `student_dir`.

This is confusing because a clear, explicit call to trigger Assignment's five steps never occurs throughout the source; it all happens implicitly. Things become simpler once the coder realizes where `Assignment::executable` is used for the first time: it all starts at `sandbox.rb:74`, a line that seems to simply double-check whether `executable` has been generated yet. There is an activity diagram of this in section 6.2.

4.7.2 Attributes

Although there are several attributes defined in `initialize`, a few of them are centrally important.

The most important to Assignment itself is `@status`. This variable is a simple bit-mask that keeps track of what has been done so far for the relevant student submitted archive. The 'constants' used to fill this bitmask can be seen at the top of `assignment.rb:8`, and are self-explanatory: each one corresponds to one of the five steps. The last two (`executed` and `executed_success` correspond to the execution step, but that is performed on an Assignment object by **`class`** `Sandbox`.

It is somewhat nonsensical for this attribute to be a bitmask instead of eg. a simple integer: each step follows upon the another; it would make no sense to attempt compilation before canonicalising, so it makes no sense to keep track of each step in a separate bit. This, of course, is not a major design flaw by any measure, however given the typical semantics of a bitmask it can be a confusing when inspecting the code.

4.7.3 Public methods

Each attribute has its own accessors (l59-99). (this code could be made more compact by using `attr_accessor` for R/W accessors)

As mentioned `executable` is an unusual read accessor that triggers the five steps if they haven't been performed yet, and updates `@status` accordingly.

`mark_recursive` not-currently-working function that is supposed to award marks based on whether or not certain C++ functions in the uploaded source (as specified in memo XML) are recursive. Apparently `Memo` should instantiate **`class`** `RecursiveCheck` during its parse process (`Memo::parse_recursive` , circa l240), but it doesn't.

`mark_leak` Instantiates a LeakCheck object and uses it to check `@executable` for any memory leaks. Note: this is unfinished functionality that doesn't work: the implementation of LeakCheck is flawed (no limits/jailing imposed or input fed), and `Assignment::mark_leak` never gets called anywhere else in Fitchfork.

`cleanup` As its name suggests this function is intended to be called when marking is done. It deletes `student_dir` and archives the generated output in `syc_dir`. This function also never gets called (a call is commented out on `marker.rb` l100)

4.7.4 Doer-knower anti-pattern

In the main program (`marker.rb`), after being used by a Sandbox object, the Assignment instance is only treated as a data-holder (little is used besides accessors). Sandbox doesn't hold state of its own, only modifies an Assignment object. In this way, Sandbox and Assignment form a "doer-knower" anti-pattern duo.

Not much has been formally written about this anti-pattern, likely because identifying it with certainty is a little difficult since there are many practical examples wherein a class should indeed act only as a passive data container (such as “plain old data structs” in C). However, writing classes whose contents are primarily acted on by external classes generally goes against the principle of encapsulation in object orientation. This anti-pattern has therefore also been described (informally but rather accurately) as “not doing object-oriented programming” [40].

5 Unfulfilled functional requirements

Although the most of the analysis concerns the “architectural” aspect of Fitchfork, there are several functional features that need improvement or are lacking entirely, besides those highlighted in section 3.2.

- Sparse or hard-to-read feedback in many cases. This goes for the feedback given to students, memo writers, and developers that debug the system.
- Error handling incomplete or errors handled in obtuse ways. This ties in with the previous point, and goes for errors (or blank screens) encountered when uploading a memo to the system, and for the ‘hints’ (or rather lack thereof) given to students when they make small but critical mistakes such as typos in file names.
- Hard-to-read assessment log. Also related to the first point; the log that FF writes about the marking process for a particular assignment (that is later read by lecturers when testing their memos or double-checking student marks) is an unformatted chunk of plain text. At the least, coloring the output according to the structure of the text would improve the readability.
- There is no mechanism for coping with misaligned lines in the output of student programs. Using regular expressions to ’absorb’ extra characters **within** a given line of output is easy, but a single additional blank line at the beginning of output could cause an otherwise perfect program to score zero. This problem is harder to work around; solving the problem properly might require implementing some form of “inter-line” backtracking mechanism in the output marker (inter-line as opposed to the “intra-line” backtracking already performed by regular expressions).
- There is no way to set the maximum amount of memory or storage an uploaded program may use for a particular assignment.
- Fitchfork is bound to the C++ language, and to the GNU C++ Compiler (C is close enough that this doesn’t cause problems, but internally FF always uses g++ to analyze source files).
- FF isn’t particularly flexible about handling the archives uploaded to it. It can only handle gzipped, bzipped or uncompressed tarball and

ZIP archives. Although these types are very common, it would be nice if FF can handle more archive types. To avoid reinventing the wheel it might be a good idea if FF delegates the task of extraction to an utility like `dtrx`. If a student uploads an uncompressed archive, FF should automatically compress it before storing it away.

- A less important point: FF doesn't reject or filter out particular kinds of files upon receiving an upload, notably compiled binaries. In COS132 it was often a problem that students uploaded executable files along with their archives. This was not a big problem as these files get ignored or overwritten in most cases, but often times such uploads would be rejected from the start for being 'too big' (there is a size-limit to uploads to UP's servers) with no hint as to what the student should do. FF should have the option to automatically erase these files, give a warning, proceed with marking, and then to store the filtered student archive instead of the original. Note that this requirement is not fulfilled by the existing `<delete-patterns>` feature of the memo XML, as those files are only deleted inside the controlled jail environment, after uploading was otherwise successful. (and currently, if a student upload is allowed through but still contains unnecessary binary files, it still gets stored as-is after marking)

The following issues are also functionality-related but a bit more concerned with security. Note that these are somewhat fundamental security concerns based on "regular" sysadmin/software developer knowledge; a more in-depth analysis by a security expert might be required at some point in the future.

- Fitchfork is run as root all the time (both euid and ruid). This is very bad practice, even if FF is run on a seperate server inside its own virtual machine. A program that doesn't absolutely need root priveleges shouldn't have it all the time no matter how secure said program is considered. Instead, the system should only gain root access for the sections of code that need it (eg. by making the runlimit utility suid root). Root priveleges include access to all hardware on the system, aborting and starting any process or invoking any library call, controlling storage mount points, modifying what is done at boot, and doing any sort of network operation.
  - The uid and gid used when running student programs inside their sandboxes can be set in FF's configuration file. The default setting for both is still root though.
- FF requires that submissions be linked statically (ie. all external functions such as the C++ standard library are copied into the executable itself), but there is no real reason for this. This was likely done initially because compiling submissions happened outside the chroot-jail environment, and when the compiled program gets run, no libraries were included in the chroot-jail, so student executables had to 'bring along' the entire standard library in order to function. There was no

point to this in the first place – why not just include essentials such as the standard library in the jail environment? This significant security vulnerability (running student-submitted makefiles as root outside a jail) existed in FF for an unknown span of time before being patched in 2014 (in turn causing and/or prompting the discovery of other bugs that were later patched). Makefiles are now run as root inside the chroot-jail, but the static linkage requirement still exists. If requiring static linkage is really desired, it should be an option that can be toggled on a per-assignment basis.

The following functional issues are relevant to developers and systems administrators:

- It is difficult to set up a test instance Fitchfork to run locally (ie. apart from the university network). It has no mode for running with just a memo file and an 'upload' archive as its input. Fortunately, there is a `debug` option in the configuration file, but any developer trying to get FF running on their own computer is left in the dark as to what directory structure FF expects, and there is no utility to set it up. The config file isn't very helpful either: some of the specifiable directory names are absolute and some or relative to the others, but figuring out how they relate to each other takes some trail and error, and even then the required file name format isn't clear without inspecting the code. This calls for a technical reference manual that includes installation instructions.
- `marker.rb` also doesn't provide any option to get more (or less) verbose log output on the screen or in the log file (fortunately FF does write to a single fixed log file, but it doesn't split this file up according to date). These logs aren't always easy to read and don't always contain the info you are looking for. Of 'always giving all the info a developer is looking for' is impossible. But like the log output given to students and lecturers, debug output facilities can also be improved.
- Installing FF for local testing requires compiling and installing the `runlimit` utility. Its source is 'bundled' with FF since it's not a standard part of any OS or easily available as a package. This program has not been updated in a long time, and has (apparently) never been formally reviewed for security issues. The alternative is to hack the source so that FF doesn't use `runlimit` when testing it. This leads to an architectural question: should FF use `runlimit` or another, better-known utility? Or should it perform chroot-jailing and resource limiting right inside the Ruby source?
  - A different utility with the exact same name exists as an open source project on the Internet. Its command line parameters differ completely from that of FF's `runlimit`. According to correspondence with FF's original dev team, the `runlimit` used in FF is a modified version of a jailing utility used in ACM International Collegiate Programming Contests. This other `runlimit` is likely a better choice for use in Fitchfork.

5.1 A single point of entry

For systems administrators' and developers' ease of use, Fitchfork could do with a single coherent facade program entry point called `fitchfork`. Such a facade should provide options for accessing FF's various modes of functioning: checking a memo's validity, marking an assignment, generating a faux assignment directory structure, dumping configuration defaults, or running regression tests.

This is somewhat similar to the Facade Pattern common in software engineering. There are many examples of large programs (or rather suites of programs) that use this "invocation facade pattern" very effectively. A notable example is the GNU Compiler Collection: the C preprocessor, linker and compiler can all be invoked conveniently through the `gcc` command by using various command-line options.

6 Investigating the program flow

This section contains a more loosely structured and somewhat conversational exposition of Fitchfork's internals. It was written progressively (and later revised) as I investigated Fitchfork's source code. The discussion roughly follows the flow of execution through the codebase as FF marks a student submission. It contains some additional details that aren't discussed in the main documentation above.

My approach was to trace the flow of execution through Fitchfork's codebase in order to begin to understand its structure. I initially skipped over sections regarding memo XML parsing for several reasons, so it is discussed last.

First, extracting the information in memo XML is not Fitchfork's most important feature. Second, I already understood the XML format in fair amount of detail. Ironically though, the "most important" code (line matching and scoring) that I was looking for resides in `memotree.rb` amidst sections of less important XML parsing code (both are contained in the same classes).

Fitchfork's overall architecture is not apparently complex, as can be gleaned from `marker.rb`. `marker.rb` is invoked in FF's most important use case: student submission of a source code archive for grading.

1. marker.rb is invoked by the marker executable with arguments: student number, part and other details of a student submission that has to be automarked. It is also given a certain one-time-password (used in the last step)

2. The relevant memo XML file is loaded and parsed (line 47, `Memo.new`)

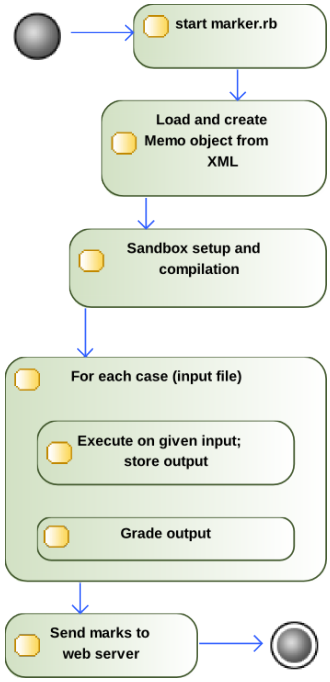
3. The student's submission and memo's pre-specified files (`<extra-files>`) are copied into a sandbox (chroot jail) and compiled according to a makefile (line 53, `sb.setup`)

4. If compilation succeeds, then for each `<case>` specified in the memo:

(a) Run the student's executable via `runlimit` and feed it the file containing standard input for the current case. Its output is piped into a file which is retained (line 55, `sb.run`).

(b) If the executable terminated normally[2], its output is evaluated against the memo's specifications and the calculated mark is added to a total (line 61, `tree.evaluate`).

5. Transmit the calculated total to the relevant server over HTTP, authenticating with the one-time password.



A rough overview of marker.rb

In every case an exception might be thrown if something goes wrong. Exceptions in steps three and four (the only steps which *should* be able to throw during student use cases) are caught and logged (line 68). It is therefore assumed that nothing will go wrong during memo parsing.

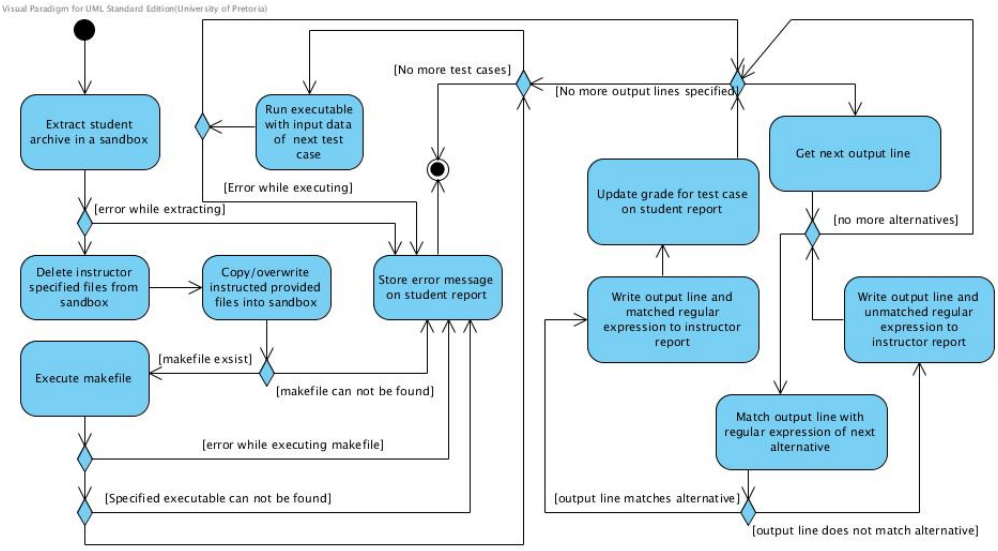
`marker.rb` is fairly simple overall, but the details of each of these steps are more involved.

In the use case of a lecturer uploading a memo file, the poorly-named `parsememo.rb` is invoked instead. This program entry point is simpler than `marker.rb`: it merely uses `class Memo` to 'parse' a given memo file, and then

[2] Up until recently (September 2014) Fitchfork would award no marks for a case if a student's executable crashed at all during that case, even after producing fully correct output. A particular memory management assignment prompted the implementation of a fix: if a student's program crashes, FF will now yield error code 2 but still evaluate the output up to the point of the crash.

sends success or error information about this operation to the web server specified in FF's config file. Since `Memo` gets used by `marker.rb` anyway, `parsememo.rb` is not discussed further.

Pieterse [29] used the following activity diagram to describe Fitchfork's marking process. It leaves out the memo XML parsing process and describes the alternatives-matching process only in the abstract, but compared to the above diagram it provides a more detailed overall idea of the tasks Fitchfork performs during marking.



FF activity diagram as seen in Pieterse 2013

6.1 Diving in

I started by manually tracing through the code, beginning at the entry point `marker.rb`. Skipping over what seems to be boilerplate code, the program starts on line 46.

Before a submission is sandboxed and tested, the relevant assignment's memo (rubric) file is loaded into memory. A Memo object is created and it's `valid?` property is set to true only once the parse() method runs to completion without any errors.

A SourceFilter object is created. All this object seems to do is to check for and reject any submitted files that contain the `fork()` system call (a reasonable safety check; a fork bomb could bring down the automarking system). The first peculiarity is that this object is created before the Assignment object that uses it (next).

The second peculiarity is that this object does NOT check for and reject specific 'tabu' `#include` directives in student code, something that is done elsewhere in the program (cf `Assignment::validate`). One could speculate that the original coders aimed for a cohesive solution in which either SourceFilter or Assignment would perform tasks related to scanning source

code for taboo segments. Apparently, this plan was progressively derailed as the system was built, leaving a fairly vestigal SourceFilter class. SourceFilter even seems to accept other options in an array `@check`, but the only entry in this array that actually triggers behaviour later on is `@check['fork']`.

If the memo file is valid then the program proceeds.

6.2 The Sandbox

An Assignment object (named `ass`) is initialized. It accepts many parameters: archive file name, student number, assignment part, course code, practical number, upload number, the memo object, and the SourceFilter object `sf`.

Next, a SandBox object is initialized, which will be used to perform chroot jail setup and compilation on the Asssignment's behalf. Its `new` method accepts no parameters. In fact, SandBox's only link to Assignment is via parameters to its methods – not its internal state – which is not a very "OO way" to do things (also discussed in section 4.7.4). As it stands, SandBox could be written as a Ruby module (code container) instead of a class.

At any rate, `sb.setup` is called and the chroot jail directory is set up. One might intuit this process as follows: `sb.setup` sets up the jail, copies the relevant student- and `<extra-files>` files into it, and later the program will chroot into this directory and compile everything before testing it.

This is not entirely the case, and this is where things begin to seem obscure: `sb.setup` is called before any method that is clearly involved with compiling the source. However, at no point in the main program (`marker.rb`) is `Assignment::compile` invoked, although `sb.run` is called almost immediately after `sb.setup`. Logically then `SandBox::setup` invokes this compilation. However, that method clearly creates "the jaildir and the outputdir" (lines 77 and 86) then promptly copies `ass.executable` – which should contain the path of the already compiled executable – into the jaildir. This line (95) of code is even accompanied by comment that states this obvious fact, but it doesn't explain when `Assignment::compile` was invoked. My immediate conclusion was that this seemingly premature copy was a remnant of a previous implementation geared towards students uploading their executables directly.

That does not turn out to be the case: `ass.executable` is a property with an associated read-accessor method. Inside this accessor the entire "extract, canonicalise, validate, filter, compile" process is invoked – if it wasn't performed yet. Invoking a large amount of important processing implicitly via a read-accessor violates the so-called principle of least astonishment.

During my initial investigation there was something much more worrisome at this point in the code: `make` is not executed inside a chrooted environment, despite the fact that many memos require students to submit their own makefiles. What's more, FF "[m]ust run this with euid=root to use chroot" (an error message that might be returned from line 72), in other words if FF is set up to execute student programs inside a chroot jail at all, then it

must be invoked as the root user, ergo student-submitted makefiles will be *executed with full system privileges*.

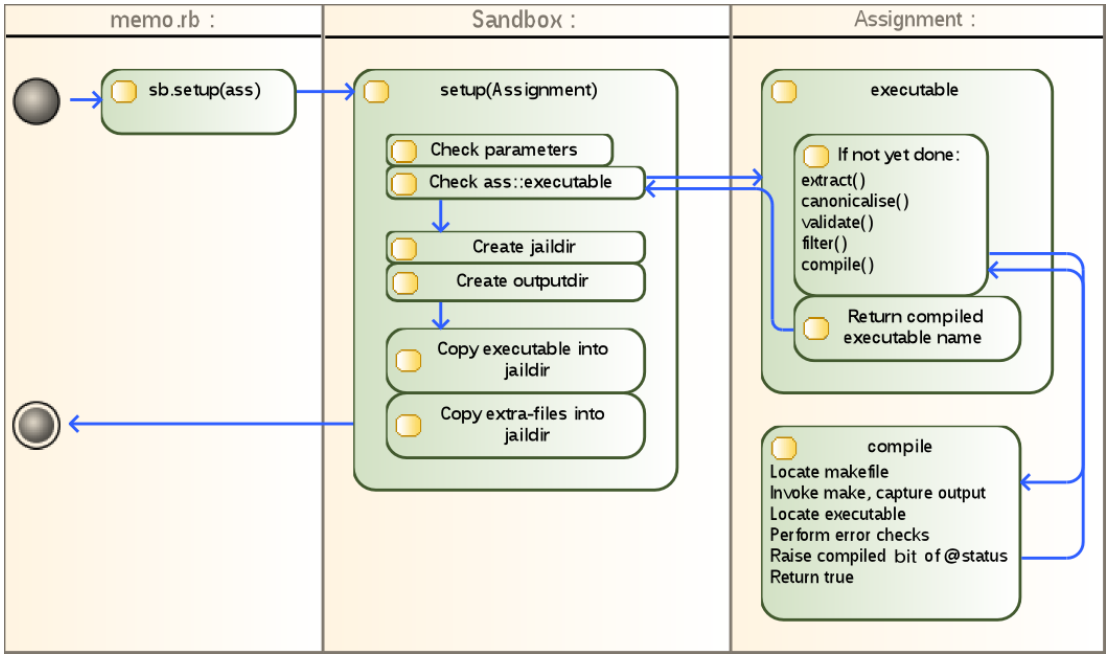
This is no subtle security loophole. It is a profound hazard: including arbitrary shell commands into a makefile is not only possible but common.

The security risk is likely mitigated somewhat by the fact that Fitchfork is served from a virtual machine, but up until recently there was no mechanism in the code or surrounding setup that explicitly compensated for the fact that `make` is not executed in a chroot jail.

It seems that this loophole was the result of design erosion, but one can only speculate as to how the loophole came to be. It is possible that support for makefiles was not originally part of FF's workflow, and that the feature was later added on without due care.

After the student executable has been compiled and copied into the jail directory, the relevant "extra files" (specified in the `<extra-files>` block in the memo) are also copied into the jail directory (performed by `Sandbox::configure_extra_files`).

Therefore once `sb.setup(ass)` has been executed, the student's submission has been compiled and placed in the configured sandbox. Next comes executing the student program with specific inputs and collecting the output, and then marking this output against the alternatives-regex structure specified for each `<case>` in the memo.



6.3 Execution

A given program uploaded for marking has to be compiled once and then executed for each case of input data it will be tested against (<case> tags). This process begins on line 54 of `marker.rb`:

```
sb.setup(ass)
memo.memotree.each_pair do |kase, tree|
        sb.run(ass, kase)
```

Note the spelling of 'case' as 'kase'. This is likely to avoid clashing with Ruby's **case** keyword without resorting to capitalization which is conventionally reserved for class names.

Inside `sb.run(ass, kase)` the relevant input file for the specific case is selected.

Execution is forked at this point using `IO.popen`, duplicating the Ruby interpreter process (l119 of `sandbox.rb`).

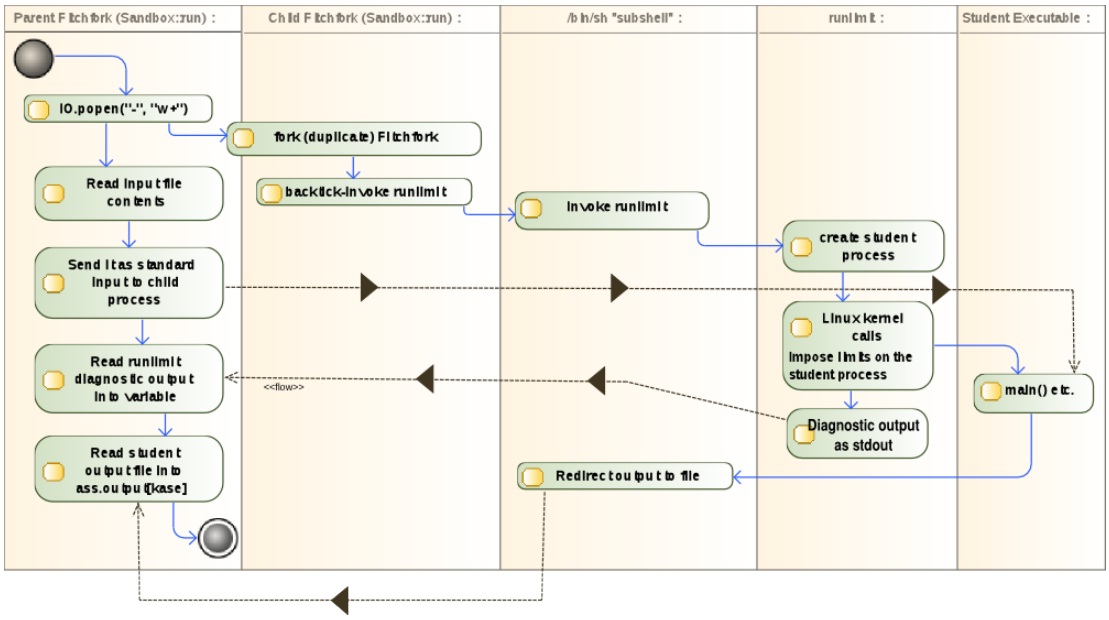
The 'parent' process of this fork reads the relevant input lines from a file and sends them down the pipe to the child process. It then reads all available output that the child process yields, and stores it in a variable. This output is *not* written to a file for reasons that will presently become clear: this is `runlimit`'s diagnostic output and not the output of the student program itself.

The 'child' process of the fork invokes `runlimit` with specific parameters using Ruby's shell-style backtick notation (with output redirection, below). This passes control to `runlimit`. The arguments passed to `runlimit` instruct it to invoke the student's executable with certain restrictions (the eponymous 'limits') namely jailing it in jaildir, restricting its maximum subprocesses to 25 (odd, considering the tabu on `fork()` calls), imposing a 1MiB limit on the amount of disk space it may use, and limiting how long the process may run in terms of wall-clock time (`sandbox.rb` l115 and l141).

The time limit can be adjusted inside an assignment's memo file. The other limits cannot, and should ideally be adjustable on a per-memo basis instead of being hardcoded as they are.

Since the child process uses Ruby's backtick syntax to invoke and redirect the output of `runlimit` – ie. in the form
`‘runlimit [arguments] student_executable 2>&1 > output_file_name‘`
– `runlimit` itself is not invoked directly. Instead it is invoked via a subshell, ie. an instance of `/bin/sh` [31]. Using this subshell adds yet another layer of overhead to the execution process.

`sh` redirects the student executable's output into a file in `Assignment.outputdir`. Directly after, this file's contents is read into a string which is stored in the array `Assignment.output` (line 151).



An activity diagram visualization of spawned processes, activities, and the flow of data (output/input streams and file contents) during Sandbox::run. runlimit produces diagnostic output as the student program executes.

6.4 Marking the output against memo elements

The structure of matching expressions defined in the memo file are now contained in the object `memo`. The different chunks of output (generated from different test cases' inputs) produced by the student's program are usually compared line-by-line to corresponding alternatives in this hierarchy, and each line of output is graded based on the score assigned to the first alternative it matches against. Multiple lines may also be matched at a time by placing them in a `<group>` block, but in practice this use is somewhat uncommon.

Output lines may be matched against exact strings (defined by `<exact></exact>` tags in the memo XML file) or against regular expressions (defined by `<regexp></regexp>`). As mentioned, every line (or even blocks of lines) may be compared against multiple alternatives (defined by the `<alt></alt>` tags). Together these two structures (matching and alternative selection) form the basis of Fitchfork's marking system.

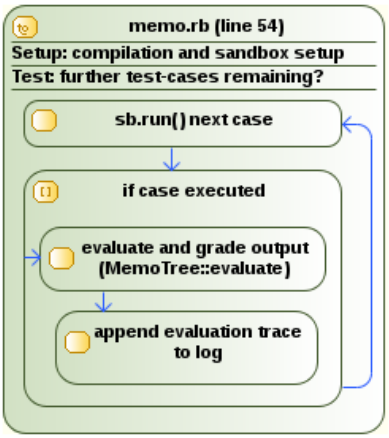
The program iterates through every element in `memo.memotree`. `Memo.memotree` is not a MemoTree object (cf. `memotree.rb`) but rather a collection of MemoTree objects, each mapping to one test-`<case>`.

The idea is straightforward: each assignment consists of different parts (or tasks). Each requires the student to write a different program having a specific set of source files, makefile and executable name. And of course, each part has a different total number of marks achievable. For each part a specific student program is tested in one or more test cases, and in each of

these cases this program is tested with different input data.

According to the comments above `Memo` and `MemoTree` in `memo.rb` and `memotree.rb`, each instance corresponds to a set of `<output></output>` tags in the XML file, each representing a different "part" of the assignment. However, this is not true: in `Memo::parse_cases` a new MemoTree is created for every set of `<case></case>` tags, which must occur inside an `<output>` block. `<output>` blocks are supposed to correspond to parts of an assignment, yet the program proceeds straight to iterating through the `<case>` tags (objects in `memo.memotree`).

The `ff-config.xml` "master XML memo reference" (4.2) specifies that "at least one `<output>` tag [is] required", but itself doesn't show an example of specifying a second output tag. In practice this use is uncommon. Lecturers write a different XML memo for every task in an assignment, each requiring a different upload instead of using the "parts" system in a single XML memo.



It turns out this functionality was designed with sequential assignment tasks in mind: when multiple assignment parts (`<output>` blocks) are defined, students have to complete and pass a given part before being allowed to attempt the parts after it. This feature is not bad to have: in some cases lecturers would indeed like to set up assignments to work in this way. However it is not very common in COS132, COS131 and COS110: most of the time assignment tasks are independent of each other, and can be attempted in any order. The result is that most actual assignments are represented by multiple XML memoranda (one per task), instead of multiple tasks inside a larger memorandum.

FF's developers likely originally intended for only one memo per assignment, and thought that the sequential `<output>` parts system would be sufficient. Lecturers on the other hand typically split assignments into multiple memos that students resulting in tasks that can be done independently. There is more than one reason for this. The most obvious is that each task of the assignment is usually a cohesive unit (one task may involve writing a word counter, and another writing a program to find the median of a set of values). Thus it seems logical to place each assignment section in its own file.

Another reason why sequential tasks might not be used as often is that students could find passing earlier tasks difficult, preventing them from obtaining marks in subsequent sections. In some courses students have to complete practical assignments in a single sitting of a few hours, and unnecessarily forcing a sequence on these tasks can only worsen an already frantic experience.

At this point it should be mentioned that FF has a reputation among students for “marking too rigidly” (ie. it rejects certain correct answers because they don’t conform exactly to the memorandum). This is not simply an inherent misfeature of FF, as is often perpetuated. The difficulty of constructing automarker memoranda that are correct, fair *and* adequately lenient has been observed on a widespread basis by other users and designers of such systems.

In contrast to several other automarking systems which only support exact matching or limited forms of substitution matching, FF offers much more potential for lenient marking through regular expression matching. The (perceived and actual) rigid nature of grading can often be attributed to inadequate competency in the design of proper memorandum test suites, as well as to the fact that students too often believe their programs “work perfectly” after passing a single poorly-designed test case.

Much of FF’s bad reputation can be attributed to memos that apply poorly designed test cases or fail to use the power offered by regular expressions. This can be remedied by better training and a user manual for lecturers to improve their competency in automarker memo writing, since it is quite different from setting up regular memoranda.

That is not to say that the problem doesn’t lie with Fitchfork as well. More flexibility needs to be added to the ways that it can parse the output of the programs it assesses. An example of a feature that is sorely lacking is the ability to specify alternative sections of expected output that *don’t* consist of an equal number of lines.

Another factor that contributes to the ‘rigidity’ of an assignment is the extent to which it has been divided into subtasks. Arguably, the smaller the section of code that Fitchfork marks at a time, the less likely it is to mark in an incorrect or unfair way.

This is especially true for C++ programming assignments for which students have to write a only a specific class or function. Such assignments are intentionally presented in a way that leaves students in some degree of doubt as to how exactly their code will be tested – similar to the experience of library writers in everyday software engineering. However a common problem is that the memo writer would decide the test all the functions in a class with a single test harness (“main file”). If the signature of even one of these functions aren’t written as expected by the main file, a compile-time error occurs and the student receives a mark of zero despite possibly having wrote a near-perfect solution, since no output can be produced after compilation failed, leading to the the dreaded “marks awarded: -1; error code: 1” message returned by FF’s web front end. (of course there are also many students who receive this message simply because they submit their solutions without testing them at all).

The fact that assignments aren’t always adequately split into sub-parts can in part be attributed to the fact that Fitchfork does provide an easy way to run an already compiled executable multiple times against various inputs (the `<case>` tag) but *doesn’t* provide an easy way to specify different compilations of a set of submitted source files against various “main files”.

The memo writer is left with the surprisingly time-consuming and error-prone task of manually creating a new memo file with a different name, copy-paste-modifying a similar memo, uploading it to Fitchfork and then testing it against a model solution.

The memo writer should have the freedom to choose whether and how an assignment's memorandum is split in terms of compilations ("builds"), test cases ("runs") and memo XML files. Even when sequential tasks are used, the memo writer should have the option of putting each `<output>` block in a separate file and linking to those files from the main memo file in some way, if only for the sake of aiding the readability of larger memos (software engineers split program components into smaller modules and files to ease maintenance of the program and to facilitate cohesion and decoupling – these concepts can be applied to the organization of memo files as well).

In this kind of system, implementing a specific class can be represented as a single task with multiple builds (one for each function that is used, so that compile time errors in one function are less likely to affect the compilation of others), with each build having a number of runs against various inputs.

Resuming our investigation of the source code on `marker.rb` line 54:

The reason why the program proceeds straight to iterating through the cases instead of parts is now clear: for a given upload (ie. a given invocation of `marker.rb`) only one part of the assignment is evaluated. When the `memo` object is created the part number is passed to it, and it only parses the XML for that specific part; *cases in other parts are ignored*.

This corresponds with the way multi-part assignments are handled from the student's perspective: write and archive code for the first part, select the upload for first part, upload the archive, view the marks awarded, do the second part which is based on the first, upload the second part, view marks awarded, etc.

This leads one to wonder why assignment parts are designated by a tag called "output" and not "part". It is possible that the name of this tag was chosen very early in development when there was a simple division between the `<config>` and the `<output>` sections of a memo, and that the meaning of the `<output>` tag changed to 'part' when support for multiple parts was later added although the tag itself was never renamed to reflect the slightly altered meaning. The comment on `sandbox.rb` line 69 might be evidence of this particular suspected case of design erosion.

Another point of criticism: `Memo` isn't an entirely ideal name for this class since it doesn't hold information for everything in a memo (only for a single part). `MemoPart` might be a more intuitive name.

6.5 memotree.rb: The matching-selection tree

From here on the discussion proceeds from the marking (`evaluate`) process 'back' to the XML parsing process. The former is implemented mostly in `memotree.rb` and the latter in `memo.rb`. During parsing the flow of execution passes first through `memo.rb` and 'visits' `memotree.rb` during that pro-

cess, building a structure of memo element objects. The evaluation process consists mainly of calling the co-recursive methods of the objects in this structure.

For each case the student program's output is collected (`Sandbox.run`) and then marked:

```
ass.mark += tree.evaluate(ass.output[kase])
```

As mentioned, at the heart of the marking process lies the combination of two operations: string matching (using the `<regexp>` and `<exact>` tags) and alternative selection (`<alt>` and `<alternative>` blocks). The default 'unit' of output that is used throughout is a single line of text (designated by `<line>`). Most assignment memos don't need anything besides the `<line>`, `<alt>` and `<regexp>` tags.

Sequential lines can be grouped with a `<group>` block (multiple such line groups may again be grouped inside an `<alt>` block). Lines may also be split into different fields (using a delimiter string) with the `split` attribute; each field is then designated by a `<field>` block inside the line block.

The first 'flaw' that is apparent in `memo.rb` and `memotree.rb` is that output grading logic and XML parsing (deserialization) logic are mixed (there is also some sandbox setup logic and even some graphviz `dot` diagram generation code in there). Separating the concerns of operations on data from the concerns of (de)serialization of said data generally yields better program architecture – memo parsing and marking are conceptually different operations that are performed during separate stages of the program. This guideline especially applies when those tasks are non-trivial: understanding and maintaining the code for one aspect becomes harder as its presence becomes 'drowned out' by the other code in between.

There are ten classes defined inside `memotree.rb`. A loaded memo is stored as a hierarchy of instances of these classes ('nodes'). Some classes correspond to tags with the same name but used in different contexts (ie. as children of different parent tags); certain XML elements behave a little differently depending on context, and actually translate to different classes in the code. This can be attributed to poor design: ideally a class should be designed to be flexible enough to handle similar cases like these; a different class shouldn't be required for slightly altered behaviour (or if this definitely required, the classes should relate somehow through inheritance and not simply be siblings as is the case here).

Every class in `memotree.rb` has an `evaluate` method that performs the marking of (a part of) the collected student program output. In all cases `evaluate` accepts a part of the output to be evaluated as a string argument (or the entire program output in the case of the root `<case>` node) and returns the mark awarded for that section as an integer.

The root node's `evaluate` function splits the output into lines and then invokes the `evaluate` method of each of its direct children, passing each a part of the input. Each of those children may have child nodes of their own. Those that do in turn call their children's `evaluate` method. Line nodes may split the line of output they receive into smaller sections handled as fields (`<field>`).

The most fundamental task being carried out is text matching, so the leaf nodes of the memo hierarchy are all instances of either of these two classes:

- **MemoExact** Exact string matcher specified by an `<exact>` tag.
- **MemoRegexp** Regular expression matcher specified by `<regexp>`.

Nodes of these classes are henceforth referred to as "match nodes". Of course, their `evaluate` methods only compare their input to a certain exact string or regular expression and return the mark assigned to them. Nodes of the other classes defined in `memotree.rb` essentially only structure the use of these two.

Keep in mind that this hierarchy structure is quite flexible: the nodes can be arranged in different ways and still yield the same behaviour.

For example, this fragment of memo XML:

```
<alt>
    <line><regexp mark='3'>[Ii]nput\s*20.06</regexp></line>
    <line><regexp mark='2'>\D+20.05\d*</regexp></line>
</alt>
```

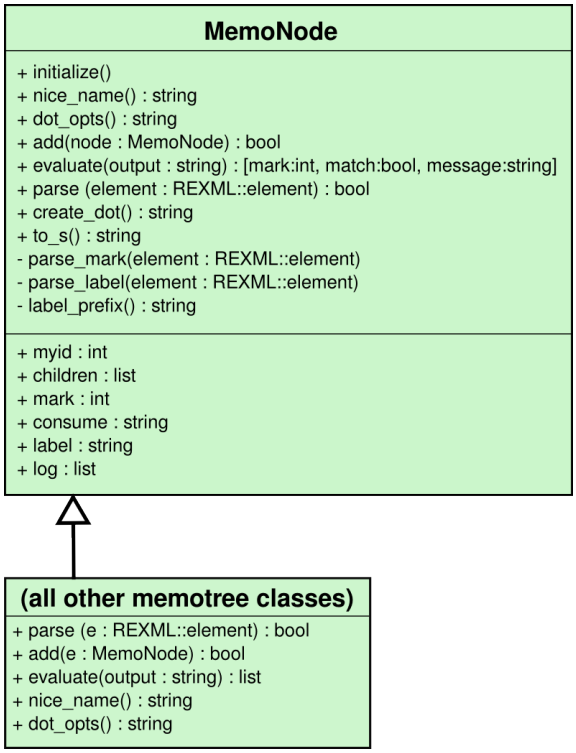
Has the same behaviour as the following fragment:

```
<line>
    <alt>
        <regexp mark='3'>[Ii]nput\s*20.06</regexp>
        <regexp mark='2'>\D+20.05\d*</regexp>
    </alt>
</line>
```

The other classes in `memotree.rb` are:

- **MemoNode** Abstract base class for all the other node classes.
- **MemoTree** The root node for a test-`<case>` block. The array `memo.memotree` contains objects of this class.
- **MemoGroupLine** A `<group>` block grouping MemoLine and/or MemoAltLine instances.
- **MemoAltLine** Alternatives block designated by either `<alt>` or `<alternative>`. Contains either MemoGroupLine or MemoLine instances. Of course, in the case of MemoGroupLines the alternatives must have equal length (ie. ultimately contain the same number of MemoLines). When marking, all alternatives are evaluated against the output and the highest mark obtained among them is returned.
- **MemoLine** Consumes a single line of output. May contain a single match node or MemoAlt (not MemoAltLine). If the line is split into fields by the delimiter specified in the `split` XML attribute, it may contain multiple MemoFields and/or MemoGroups (not MemoGroupLines).
- **MemoAlt** Alternative block containing only match nodes. When marking it evaluates output against every alternative and returns the highest mark obtained among them.
- **MemoGroup** A `<group>` tag inside a `<line>`. May only contain MemoField instances.

- **MemoField** A field inside a `<line>` or `<group>`. Similar to MemoLine but instead of a line of output it consumes a single field, which is one of the parts of a line after it has been split by a delimiter. May contain a single match or MemoAlt node.

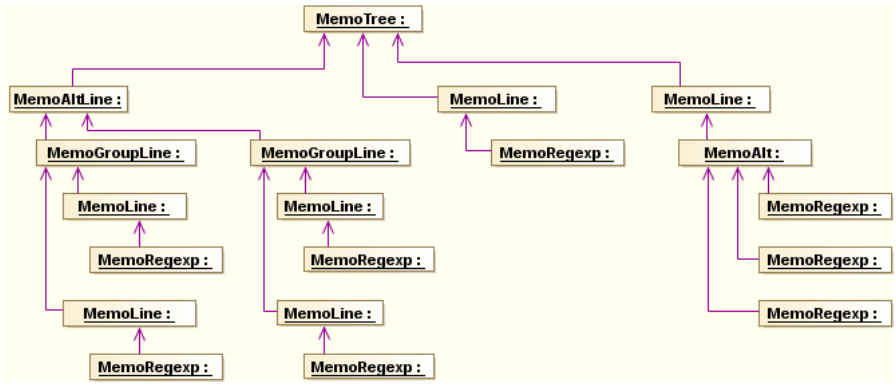


This pseudo-UML diagram depicts the hierarchy in memotree.rb. All the other classes mentioned above override the same methods. In addition, match nodes also override initialize()

The following example roughly visualizes the hierarchy of objects that a fragment of memo XML translates to. In the figure, the purple arrows indicate ownership (composition).

An example memo XML test case

```
<case id="1">
<alternative>
<group mark="2">
        <line><regexp>^[Ww]elcome.+[Cc]alculator</regexp></line>
        <line><regexp>Please enter your name</regexp></line>
</group>
<group mark="1">
        <line><regexp>elcome</regexp></line>
        <line><regexp>.*</regexp></line>
</group>
</alternative>
<line mark="1"><regexp>[Ee]nter.+expression.+</regexp></line>
<line>
        <alt>
                <regexp mark="3">[Rr]esult:?\s*8.49232$</regexp>
                <!-- Correct result and number of decimal places -->
                <regexp mark="2">[Rr]esult:?\s*8.49\d+</regexp>
                <!-- Near-correct result (more or fewer decimal places) -->
                <regexp mark="1">[Rr]Result:?\s*\d([.]\d)*</regexp>
                <!-- Correct format -->
        </alt>
</line>
</case>
```



The resultant tree of objects

Every non-leaf node is considered to be a **consume structure**[3]: each 'consumes' and returns the mark for a certain amount of output, measured in lines or fields. This amount is determined for each node as the XML for it is interpreted. This value is then stored in each node's member variable `@consume` *as a string* in number-unit format: eg. `'12l'` indicates twelve lines and `'1f'` indicates one field. When a node calls `evaluate` on its children, this attribute tells it how much of the input it should pass to each child.

The `<case>` root node's `evaluate` 'consumes' all lines of output and returns the mark awarded for it (although it only actually consumes that portion of the output for which its children define markable lines). MemoLine

[3]The term "consume structure" occurs in the comments and some error messages in the code, but is otherwise only implicitly used to describe this concept.

consumes a single line, and may produce 'fields' that are consumed by MemoField or MemoGroup. MemoGroupLine naturally consumes the number of MemoLines grouped within it. MemoAltLine consumes a certain number of lines, and this number has to be the same for all its children (ie. one if its children are MemoLines or some greater number if its children are MemoGroupLines). The same applies to MemoAlt which consumes a single line or field (the implementation does not allow groups inside MemoAlt). MemoField consumes one field. MemoGroup consumes as many fields as the number of MemoFields in it.

Besides the mark, `evaluate` also returns a 'pass' flag and a log message (which ultimately gets displayed to lecturers via Fitchfork's web interface when they view student submissions made so far). Therefore `evaluate` actually returns an ordered list: `[mark_awarded, pass_flag, message]`. For brevity the `message` part of the return value is left out of the descriptions below.

Every node has an associated mark defined by the `mark` attribute in XML and stored in its `@mark` instance variable as an integer. If this number is not defined in the XML for a given node, then zero is assumed during the parsing process (`MemoNode::parse_mark`). The `evaluate` method for each kind of node works as follows:

Match nodes return `[@mark, true]` if the string passed to them matches their pattern defined in the XML (which is stored in member variable `@match`). They return `[0, false]` if it doesn't match. Note that match nodes don't care exactly how much they are consuming (line or field), only that they receive a string to be compared. Here the `pass` flag is redundant. Also note that a match node's `@mark` may equal zero, so it is possible to return `[0, true]`.

A MemoGroupLine node returns its own `@mark` *plus* the sum of the marks returned by its child nodes and **`true`** if and only if all children returned **`true`** for their pass flags. That is to say, if a single line in a MemoGroupLine doesn't match student output, the output scores zero against that MemoGroupLine.

MemoGroup works in the same way, except it can only contain MemoFields.

This indirectly causes flexible behaviour: if a memo writer wishes to match three consecutive lines and award X marks if they all pass, then she can simply define the mark attribute for the `<group>` and omit it in the match nodes. If she instead wishes to define the mark for each match node and have the group sum them if all pass, then that is possible. If marks are specified for each match node as well as for the `<group>`, each line is awarded individually and the group's mark is added to the total if and only if all the lines in the group pass.

MemoLine behaves in exactly the same way on its MemoField children if it has a `split` attribute defined. If a MemoLine doesn't have `split` defined it only makes sense for it to have one child. However it is possible to put more tags inside a `<line>` that isn't split: the MemoLine class will not complain during parsing. It will then call `evaluate` on only its first child when marking

and otherwise silently ignore its other children.

Interestingly, `MemoLine::evaluate` always `strips` leading and trailing spaces on the output given to it before doing anything else. At a glance this seems reasonable, but it is really arbitrary and can be extremely annoying when leading (or trailing) spaces actually matter to the memo writer. An example is when the student's program is expected to output an 'ASCII-art' shape and leading spaces may be needed to position other characters in the output. When matching against regular expressions that don't explicitly contain ^ and $, leading and trailing spaces don't matter in the first place, and it is otherwise easy to use an `\s*` clause to absorb optional white space if the memo writer wishes to do so.

MemoAltLine may have either MemoGroupLine or MemoLine objects as its children. Its `evaluate` takes either a single string (for MemoLine children) or an array of strings. It passes this output to each of its children's `evaluate`, and returns the highest mark obtained from those children that have matched. If none match, it returns `[0, false]`.

`MemoAlt::evaluate` works in the same way, except it doesn't accept an array of strings, and doesn't have a `consume` attribute as it gets encapsulated by either MemoLine or MemoField.

Like MemoLine, `MemoField::evaluate` only invokes its first (and presumably only) child's `evaluate` and returns its results.

6.5.1 Room for simplification

Some of these classes are near-identical copies of each other. It should in fact be possible to distil the contents of `memotree.rb` into a much smaller number of classes, one for each concept employed: matcher, alternative selector, container (or grouper).

The current implementation conforms to a kind of interpreter pattern: XML elements and structure translate into a hierarchy of objects à la composite pattern. However, the presence of (or rather apparent conformation to) a formal design pattern itself does not imply that a design is optimal. As pointed out, the codebase maps certain XML elements to more than one class even though their implementations consist of highly similar chunks of code, and despite the similarity in what these elements themselves do. Dependencies on context that otherwise don't affect the functionality of the individual elements should, as far as possible, be dealt with during parsing as opposed to during marking.

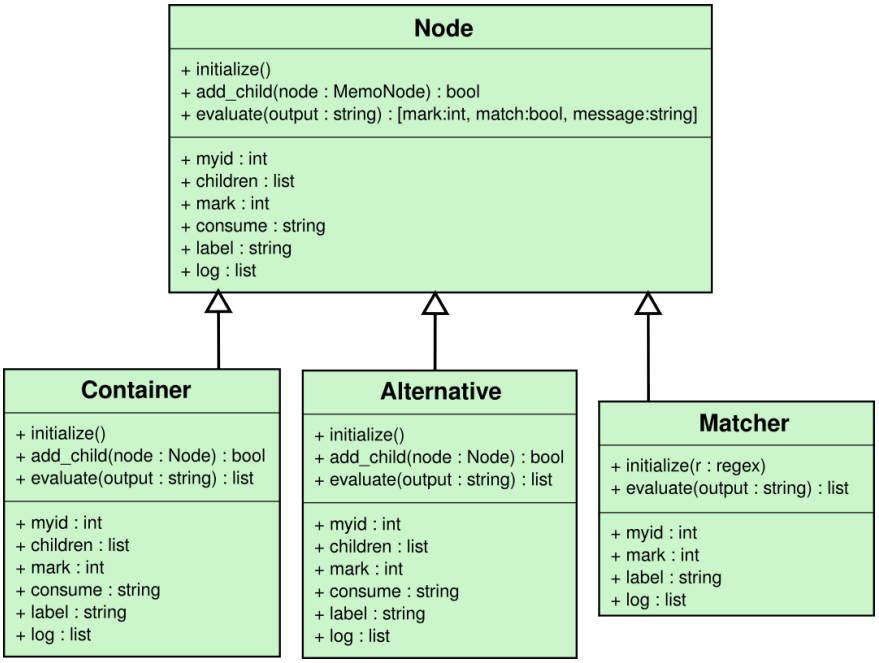
It is easy to imagine how the two match node classes can be rewritten as a single *matcher* class. It might also be possible to represent groups, fields, lines and the overarching `MemoTree` using a single *container* class with different consume values. Keeping with current functionality, the *alternative selector* can then be made to operate on any set of container children that consume the same amount of output.

The alternative selector concept can even be merged with the container concept: `<group>` is a container for which all items have to match (analogous to an AND gate) and `<alt>` is a container for which one of the items has to

match (OR gate). Similarly, an `<output>` section is a container that tallies the mark-totals of the matches so far (an accumulating container).

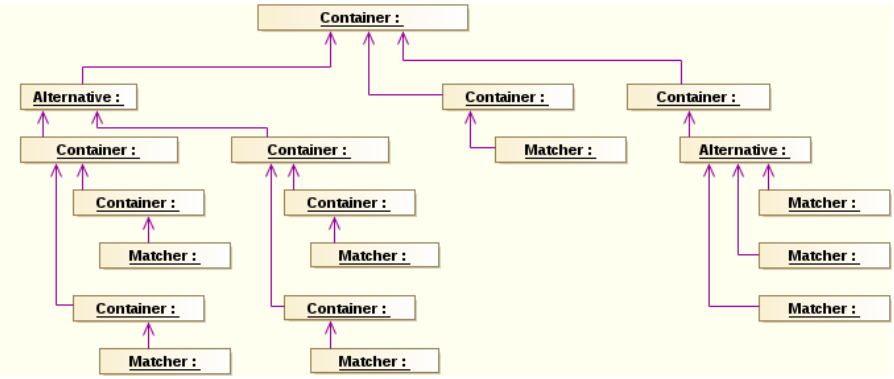
Such a design would retain the overall composite-interpreter pattern, but it would simplify the grading code by specifying ways to handle each of the three proposed classes instead of having `*::evaluate` methods in eight different sibling classes. The complexity would be reduced but won't disappear entirely; some of it would simply be rearranged between the parser and evaluator. The parser, for example would handle the correct instantiation of a `Container` for either `<group>` or `<alt>` eg. `Container.new(:MATCH_ALL)` vs. `Container.new(:MATCH_FIRST)` whereas the `evaluate` methods for this class would alter its behaviour based on the argument with which the object was instantiated.

Note that such a simplification of the internal representation of memo elements does not have to (and should not) affect the XML representation at all.



Class diagram for proposed new memotree hierarchy. For improved separation of concerns, parsing routines should be implemented as a set of factory methods that instantiate these classes.

The object diagram example given earlier would translate as follows under the suggested changes:



The memo element tree still contains the same amount of complexity, but is composed of only three different concrete object types instead of six.

6.6 memo.rb

The rest of the memo XML's information is parsed and stored by **`class`** `Memo` in `memo.rb`. Interestingly, it never parses the `<config>` block of the root `<memo>` tag – this information is probably used only by the web infrastructure that invokes FF and stores student marks. `Memo` is therefore mostly concerned with the subtags of `<output>` besides the actual alternatives matching tags which are handled by `memotree.rb`.

The majority of the functions in this class are for parsing, interpreting and storing the memo information, but it also contains some code for setting up directories for the compilation and execution process. The parse-and-interpret process is initiated with the `parse` method which uses Ruby's REXML module to facilitate the whole process.

Many of the functions in Memo are very similar – a lot of code has been nearly duplicated. The code is about 430 lines long, and the majority of it does a very simple task: read and interpret XML tags that contain simple string an numeric values that are later used in sandbox setup and execution.

6.7 Criticism concerning the parsing architecture

The main issues with FF's memo XML parser are:

- Much of the parsing code consists of repeated segments with minor variations. This violates the "Don't Repeat Yourself" (DRY) principle and increases the effort needed to modify anything about the parsing process.
- It only does a partial job of finding errors when a memo writer uploads their memo to the system. Sometimes the interface correctly points out errors such as missing closing tags, and sometimes it simply crashes with no feedback, and at other times it appears to succeed but doesn't actually upload the memo. The concern of providing feedback upon memo upload is consistent with Fitchfork's overall concerns, but

this functionality is contained in the web front-end's code, not in FF's codebase. It would be better if the web interface delegates this task of memo verification to some part of Fitchfork itself, reducing the amount of duplication in the system (again, the DRY principle and cohesion).

- FF parses and does its own sanity checks on the memo data every time a student upload is evaluated. Although it can be good to have safeguards at every turn, this is ineffective when the safeguards aren't thorough, and can become inefficient when the system is required to handle a high volume of uploads. Under normal (non-debugging) circumstances, error checking should arguably be done only once before the memo gets used repeatedly.

At this point it become necessary to discern more clearly between syntax and form checking, parsing, verification, storing, and actual use of data. As in the case of FF's code, the word "parsing" is generally often used to mean any one or all of these things. Dividing these concepts more clearly from one-another is not important in every context, but it is helpful to do so in this case.

The term "parse" can be very ambiguous. "Parsing" is the process of reading the XML string data, recognizing individual XML tags (tokenizing), and then performing some operation based on this information Usually this operation consists simply of building a structure that represents this data in memory, or checking it for errors, but it could also mean acting on the data in a certain way according to the semantics of each tag. The REXML library is described as an XML parsing library, ie. it is used to traverse an XML tree. But Fitchfork's Memo class is *also* described as a parser, ie. it uses REXML to traverse the XML tree and extract the data from it, checks some of the data for errors, and builds a data structure of `MemoNode` instances accordingly.

The stages of reading an XML data file – basic syntax checking, schema verification, interpretation of data (ie. acting on its meaning), verification and storing – are conceptually different operations but in practice often mixed or confused because the concepts are inherently related to each other. Failing to draw sufficiently clear lines between these operations can lead to poorly separated concerns in code.

When creating a MemoLine object and storing it in a MemoTree's list of children, the following is done:

1. A `<line>` tag and its mark attribute has to be read from the XML (and implicitly gets verified for syntax – it can't be read otherwise)

2. A MemoLine object is created, and its `parse` method is called.

3. The mark attribute is initially stored as a string by the REXML parser, which is then converted to an integer by `MemoLine::parse` (verification – if it can't be converted an exception is generated at this point)

4. This mark attribute is stored as member variable `@mark` of the new MemoLine object... (storing)

5. and then checked to be non-negative; FF generates an error otherwise (verification – arguably very unnecessary in this case since memo writers may want the option of 'awarding' negative marks for output that is obviously wrong)

6. Later, the MemoLine object is used to mark a student program's output. Evaluate returns `@mark` if the relevant line matches, and this value is in turn added to a total (use of previously stored and verified data).

Obviously verifying or using data implies understanding (interpreting) it somehow; interpretation is a concept on its own but not an isolated stage.

FF's XML reading code mingles form checking with parsing and interpretation and verification of the data. In the case of the marking hierarchy it comes close to mixing all that with the actual use of the data by including `parse` and `evaluate` in the same classes. This is a gross violation of the known best practice of separation of concerns.

Any XML document has to adhere to XML's syntax rules, but beyond that also to a certain schema. A schema determines what combinations of tags make sense for a certain type of document. During FF's initial development, no XML schema file was written. It is possible that one was later written for the memo upload verifier of the CS website.

There are libraries that simplify the process of verifying an XML document against a given schema document, such as RXSD [34] or Nokogiri [25].

Once it is determined (with the help of such a library) that a file has no syntax errors and adheres to the overall structure of eg. a memo XML file, the data's semantic content can be validated and then stored in memory. Separating the concerns of syntax and form checking from interpretation of the data in this way could simplify Fitchfork's codebase significantly by eliminating most of `memo.rb`. Restructuring Fitchfork to use a schema file and relevant library could also make the code safer and faster, as publicly-available libraries are usually developed and maintained with speed and security in mind.

7 Conclusion and future work

Fitchfork has been used and incrementally developed over the past eight years. As an automarker, it has proven to be an indispensable tool in the presentation of early courses in computer programming.

Over the years Fitchfork has built up a bad reputation among CS students for its 'overly rigid marking'. This can be attributed to a variety of factors including a lack of features, lack of competency regarding regular expressions among lecturers, as well as students' lack of ability to realize that their programs are indeed not perfect without a detailed explanation (which an automarker cannot give the way a lecturer can).

FF has also proven challenging to maintain for the Computer Science department tech team. Its lack of certain features often restricts lecturers'

ability to write quality assignment memoranda, or otherwise significantly increases the amount of time and effort required to do so.

Many of these problems are related to poor internal design. Throughout FF's development, it would seem that attention was more often focussed on smaller sections of the codebase at a time, and the system as a whole eroded as a result. No thorough code reviews have been done so far, and the effort of understanding the system once one dives into the code has only grown over time. FF has no working test suites for regression testing, even though new tweaks are slowly but constantly being implemented (as of writing in 2014).

The documentation and critical discussion of the codebase contained in this document are intended to assist further development, as well as to help inform CS department's decisions on Fitchfork's future. This documentation should be able help developers who haven't studied Fitchfork to understand its internal architecture and flaws.

Further research is needed regarding alternative available automarking systems. It could be that the alternative that CS dept is looking for already exists. As outlined though, there are criteria that have disqualified a myriad of other available automarkers when the matter was investigated by CS staff earlier in this year.

However important it is to look for existing alternatives, it is very unlikely that Fitchfork will be entirely discarded, at least in the near future. As such, it is important that the missing functionality and issues with its architecture be addressed. Once this is done it might even be possible for CS to make Fitchfork's code available to the public or other Universities.

Older versions of Fitchfork's architecture evidently remain as fragments or peculiarities in the codebase. Much work is still required to streamline Fitchfork's architecture by rethinking the memo parser, evaluator and sandbox/compilation modules while considering the architecture as a whole. The reasons for conditions and requirements that were carried over from old versions (such as requiring static linkage of submissions) must be questioned in this process.

To ensure that these aren't built upon the existing flaws in the foundation, this should ideally be done before further bug fixes and feature implementations. Refactoring any program's "foundational architecture" is liable to break existing features and force some extent of redesigning them.

I recommend an extensive once-off effort to improve Fitchfork's architecture – a rewrite based on the current codebase (as opposed to a rewrite from scratch). If this is not feasible, the same end could be achieved through gradual, progressive refactoring of the system ("perfective maintenance").

Improving FF's architecture should open the way for creating thorough regression testing suites to support further development. Previous regression tests have fallen out of use in part because of the way the architecture has shifted over time to accommodate user requirements.

Besides overall improvement of the architecture, further work is needed to assess the feasibility of and eventually implement the highlighted missing functionalities that Fitchfork (and by extent lecturers and students) suffers

from.

References

[1] Anitesh Barua and Tridas Mukhopadhyay. “A cost analysis of the software dilemma: to maintain or to replace”. In: *System Sciences, 1989. Vol. III: Decision Support and Knowledge Based Systems Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*. Vol. 3. IEEE. 1989, pp. 89–98.

[2] Keith H Bennett, Magnus Ramage, and Malcolm Munro. “Decision model for legacy systems”. In: *IEE Proceedings-Software* 146.3 (1999), pp. 153–159.

[3] Frederick P Brooks. *The mythical man-month*. Vol. 1995. Addison-Wesley Reading, MA, 1975.

[4] Taizan Chan, Siu Leung Chung, and Teck Hua Ho. “An economic model to estimate software rewriting and replacement times”. In: *Software Engineering, IEEE Transactions on* 22.8 (1996), pp. 580–598.

[5] Brenda Cheang et al. “On automated grading of programming assignments in an academic institution”. In: *Computers & Education* 41.2 (2003), pp. 121–131.

[6] Theuns Cloete and Francois Geldenhuys. *fitchforkDocumentation.zip*. Unpublished. can be requested from techteam@cs.up.ac.za. 2006.

[7] *Codejudge: host coding competitions anywhere, anytime*. URL: `http://sankhs.com/codejudge/`.

[8] Alan M Davis. “Software prototyping”. In: *Advances in computers* 40 (1995), pp. 39–63.

[9] Christopher Douce, David Livingstone, and James Orwell. “Automatic test-based assessment of programming: A review”. In: *Journal on Educational Resources in Computing (JERIC)* 5.3 (2005), p. 4.

[10] *EasyAccept Website*. URL: `http://easyaccept.sourceforge.net/`.

[11] Stephen G Eick et al. “Does code decay? assessing the evidence from change management data”. In: *Software Engineering, IEEE Transactions on* 27.1 (2001), pp. 1–12.

[12] Erik Ernst. “Separation of concerns”. In: *Proceedings of the AOSD 2003 Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT), Boston, MA, USA*. 2003.

[13] Sandra P Foubister, GJ Michaelson, and Nils Tomes. “Automatic assessment of elementary Standard ML programs using Ceilidh”. In: *Journal of Computer Assisted Learning* 13.2 (1997), pp. 99–108.

[14] Jilles van Gurp and Jan Bosch. “Design erosion: problems and causes”. In: *Journal of Systems and Software* 61.2 (2002), pp. 105 –119. ISSN: 0164-1212. DOI: http://dx.doi.org/10.1016/S0164-1212(01)00152-2. URL: http://www.sciencedirect.com/science/article/pii/S0164121201001522.

[15] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

[16] Walter L. Hürsch and Cristina Videira Lopes. *Separation of Concerns*. Tech. rep. 1995.

[17] Petri Ihantola et al. “Review of recent systems for automatic assessment of programming assignments”. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. ACM. 2010, pp. 86–93.

[18] Open Source Initiative et al. *The open source definition*. URL: http://opensource.org/osd.

[19] Mike Joy, Nathan Griffiths, and Russell Boyatt. “The boss online submission and assessment system”. In: *Journal on Educational Resources in Computing (JERIC)* 5.3 (2005), p. 2.

[20] Poul-Henning Kamp and Robert NM Watson. “Jails: Confining the omnipotent root”. In: *Proceedings of the 2nd International SANE Conference*. Vol. 43. 2000, p. 116.

[21] Kevin Kelly. *Master Planner: Fred Brooks Shows How to Design Anything*. 2010. URL: http://www.wired.com/2010/07/ff_fred_brooks/.

[22] Raimund Moser et al. “A case study on the impact of refactoring on quality and productivity in an agile team”. In: *Balancing Agility and Formalism in Software Engineering*. Springer, 2008, pp. 252–266.

[23] *MOSS: A System for Detecting Software Plagiarism*. URL: http://theory.stanford.edu/~aiken/moss/.

[24] Microsoft Developer Network. *Microsoft Application Architecture Guide, 2nd Edition: Chapter 2: Key Principles of Software Architecture*. 2009. URL: http://msdn.microsoft.com/en-us/library/ee658124.aspx.

[25] *Nokogiri: HTML, SAX, XML and Reader parser*. URL: http://www.nokogiri.org/.

[26] *OpenKattis Website*. URL: https://open.kattis.com/.

[27] David Lorge Parnas. “Software aging”. In: *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press. 1994, pp. 279–287.

[28] *peach3 website*. URL: http://peach3.nl/trac/.

[29] Vreda Pieterse. “Automated Assessment of Programming Assignments”. In: *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research*. Open Universiteit, Heerlen. 2013, pp. 45–56.

[30] *Regular Expressions, The Open Group Base Specifications Issue 6 IEEE Std 1003.1*. The IEEE and The Open Group. 2004. URL: http://pubs.opengroup.org/onlinepubs/007904975/basedefs/xbd_chap09.html.

[31] *Ruby forum topic: backtick subshell*. 2006. URL: https://www.ruby-forum.com/topic/88182.

[32] *Ruby on Rails official website*. URL: http://rubyonrails.org/.

[33] *Ruby Website — Success Stories*. URL: https://www.ruby-lang.org/en/documentation/success-stories/.

[34] *RXSD: XML Schema Definition to/from Ruby Translator*. URL: http://projects.morsi.org/wiki/RXSD.

[35] *Sharif Judge: An online judge for programming courses*. URL: http://docs.sharifjudge.ir.

[36] *SIO2 Project on Github*. URL: https://github.com/sio2project/oioioi.

[37] *Sphere Online Judge website*. URL: http://www.spoj.com/.

[38] Joel Spolsky. *Joel on Software: Things You Should Never Do*. URL: http://www.joelonsoftware.com/articles/fog0000000069.html.

[39] *The Web-CAT Community: Resources for automated grading and testing*. URL: http://web-cat.org/group/web-cat.

[40] Various. *C2 Wiki: Doer And Knower*. 2011. URL: http://c2.com/cgi/wiki?DoerAndKnower.

[41] Bill Venners. *Don't Live with Broken Windows, A Conversation with Andy Hunt and Dave Thomas, Part I*. 2003. URL: http://www.artima.com/intv/fixit2.html.

[42] Bill Venners. *Orthogonality and the DRY Principle, A Conversation with Andy Hunt and Dave Thomas, Part II*. 2003. URL: http://www.artima.com/intv/dry.html.

[43] Wikipedia. *Ruby (programming language) — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/w/index.php?title=Ruby_(programming_language)&oldid=622335096.

[44] James Q Wilson and George L Kelling. "Broken windows". In: *Atlantic monthly* 249.3 (1982), pp. 29–38.