CA400 Final Year Project

comet | Technical Guide

Students	**Kevin McGonigle** 16318486 kevin.mcgonigle2@mail.dcu.ie **James Miles** 16349533 james.miles2@mail.dcu.ie
Supervisor	**Dr. Geoff Hamilton** geoffrey.hamilton@dcu.ie
Date	16/05/2020
Git Repo	https://gitlab.computing.dcu.ie/mcgonik2/2020-ca400-mcgonik2-milesj2

0. Abstract

Comet is a powerful, comprehensive and user-friendly code metrics and static analysis utility. The system takes the form of Django-based RESTful API providing functionality for uploading codebases and obtaining insightful quantitative measurements and helpful models. This information is calculated through the application of the visitor design pattern on parse trees produced using the ANTLR parser generator, and is displayed to the user on an interactive, intuitive and intelligible front-end user interface, built using the React JavaScript library.

1. Preface

1.1 Scope

This guide provides an in-depth overview of the technical aspects of the comet project, including details on how the project was conceived, managed, designed, developed and validated. It contains information regarding the technical outline of comet's design, extensive coverage of its implementation, and concluding remarks on challenges faced, solutions devised and potential future directions for its development.

1.2 Purpose

This guide seeks to provide a comprehensive and useful insight into comet's background and technical details to the reader, so that they might gain a greater understanding of how comet functions.

1.3 Audience

This document is intended for operators and users who wish to implement or use any of comet's subsystems. It assumes no prior knowledge or experience and serves as a good starting point for developers or users to become more familiar with the technical aspects of comet.

1.4 Glossary

comet

The name given to the developed code metrics and analysis tool.

Django

A high-level Python Web framework that encourages rapid development and clean, pragmatic design.

Representational State Transfer (REST)

An architectural style for creating web applications hinging on interoperability.

ANTLR

A parser generator for processing and translating languages by generating parse trees based on custom grammar files.

2. Background

2.1 Motivation

The motivation for comet stems from the ever-increasing demand on the software development industry to produce manageable, efficient and reliable code in as swift a manner as possible. With the emergence of agile software development frameworks and the boom in the role that automation plays in software development life-cycle, the domain software development is increasingly emphasising speed and efficiency.

The motivation behind comet therefore was to produce a tool that would dynamically provide users with valuable information about the structure, reliability and performance of their code. This information would provide both an opportunity for adhoc or even automated quality assessment, as well as a continuous educational opportunity to inform users about flaws in their codebase and help them to improve as programmers.

It was also imperative that comet would take the form of a universal API with simple requests and responses to ensure portability and flexibility. This would allow the fulfilment of all manner of different from a number of different contexts, including the implementation of quality gates in automated deployment pipelines, usage as an educational tool, or as a preemptive measure to reduce costs incurred by running inefficient code in the context of function as a service (FaaS).

2.2 Users

Comet is unique in that it provides a valuable service to programmers of all levels of ability and experience. In a professional context, comet can be employed by expert users to quickly and dynamically obtain useful insights into their codebase's structure and performance. This information can shine a light on areas of the

codebase that are in desperate need of refactoring or optimisation, potentially consequently saving the business money wasted on both man-hours wasted on code analysis and by reducing the resource usage of the company (e.g. FaaS).

Conversely, comet also serves as a useful resource to beginner programmers or at an educational level. Through its navigable, intuitive user interface, comet can offer novice or recreational users a novel perspective of their code and allow them to learn how their code works and also how it might be improved. Ultimately, comet is designed to be universally valuable to the area of software development as a whole.

2.3 Project Management

The design, development, validation and maintenance of comet was and is meticulously planned and documented, making use of a variety of tools, frameworks and technologies to ensure that the project process was planned and managed effectively.

2.3.1 Agile Development

The basis for comet's development process was the adoption of a scrum-like development framework, in which the two developers would regularly check-in with one another with updates on progress, issues and short discussion about solutions. This correspondence, in conjunction with scheduled, weekly review and planning meetings with the project supervisor, ensured that all team members were informed about the current state of the project and involved in the planning process.

2.3.2 Issue Tracking

To support the agile software development approach adopted throughout the project's duration, one of the most valuable resources for managing and planning comet's development was the utilisation of the YouTrack issue tracking solution by JetBrains.

YouTrack provided functionality for logging epics, user stories and tasks, assigning them to team members, planning sprints and logging time and comments against issues. This proved to be an invaluable resource, with an agile board and product backlog providing the team with a consistent, coherent idea of what was required on a week-by-week and on a long-term basis.

2.3.3 Codebase Management

With development on comet being split between two independent developers, it was imperative that a centralized version-control system was employed and maintained to facilitate simultaneous feature development. The codebase for comet is maintained on GitLab, with feature branches being created and merged as

functionality is continuously added to the system and verified. Hosting on GitLab also aids in the deployment of the codebase to production and testing servers, with facilities for continuous integration and deployment.

3. Design

3.1 Features

3.1.1 Multiple Language Support

The major advantage that comet holds over its competitors is its independence with respect to the source languages it accepts for analysis. While most other static analysis tools provide language-specific functionality, comet was designed to be abstract in its nature and is extensible to include all manners of language. The developers felt that this was an important feature and major selling point as software development companies increasingly seek to detangle monolithic, single-language applications and move in the direction of microservices and polyglot solutions. The ability to provide a single, common, familiar program that caters for several languages was a high priority requirement in comet's design.

3.1.2 Model Generation

Central to comet's design was the provision of a variety of different models for visualising uploaded codebases from different perspectives.

3.1.2.1 Abstract Syntax Tree (AST)

The model that is most important to comet's functionality is the generation of abstract syntax trees. AST's grant the language independence as discussed in the previous section by breaking language-specific parse trees down into a structure that represents equivalent code in different languages in an identical manner. This allows for additional models and structures to be calculated using the same logic, irrespective of language.

3.1.2.2 Control Flow Graph (CFG)

The control flow graph shows all the possible paths through a given program, depending on the presence of certain control-flow structures such as branching statements (e.g. "if", "if-else" and "switch" statements), loops and jumps. The control-flow graph is useful for visualising how complex a program is, showing clearly how many paths exist in the code. It can also be used for calculating certain metrics such as cyclomatic complexity and nesting depth.

3.1.2.3 Class Diagram

Based on the UML specification, comet's design includes functionality for deriving class diagrams from code, displaying the characteristics and relationships of classes in the codebase. Class diagrams are usually generated as part of the design phase of a project, but providing the ability to dynamically generate them from produced code allows a clear, familiar view of the code to ensure that the program has implemented the design properly,

3.1.2.4 Dependency Graph

A subset of the class diagram, the dependency graph provides an overview of all dependencies present between classes within the program. This sheds light on how tightly coupled a program may be or any potential circular dependencies that may exist.

3.1.2.5 Inheritance Tree

The inheritance tree is designed to display inheritance information for classes within a submitted program. This model therefore makes it clear which classes a given class inherits members from.

3.1.3 Metric Calculation

A primary feature of comet's design and functionality is its ability to calculate quantitative metrics that offer insight into the maintainability, efficiency and performance of a submitted program.

Metric	Description
Cyclomatic complexity	The number of linearly independent paths through a program.
Logical lines of code	The number statements in a program.
Afferent coupling	The number of classes that depend on a given class.
Efferent coupling	The number of classes that a given class depends on.
Instability	The ratio of efferent coupling to total coupling.
Maximum inheritance depth	The maximum path length from a base class to an inheriting class in an inheritance tree.
Maximum nesting depth	The maximum number of encapsulated scopes in a program.

3.1.4 User Interface

While the comet is primarily designed as an independent API capable of providing code quality measurements in a number of dynamic, automatic or ad hoc contexts, the design includes an accompanying graphical user interface to support manual file upload and interactive visualisation of the generated models and metrics. Such a user interface could be used in an integrated IDE extension or simply as a web page in a browser and provides a graphical means for interacting with the comet system.

3.2 Architecture

Comet is designed to conform to a multi-tier web-based API architecture. Multiple layers of separate subsystems aid in the separation of concerns and portability within comet's design.

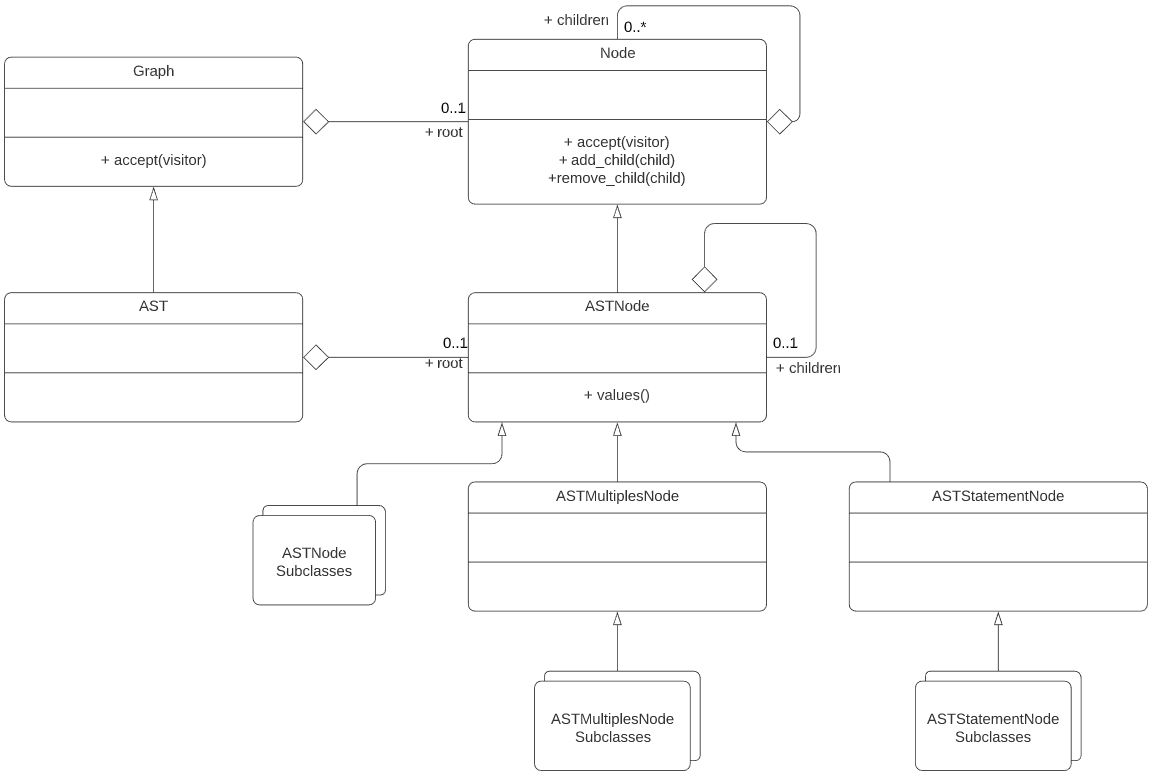
3.2.1 Overview

As documented, comet's architecture design utilises a multi-tier architecture with the system divided into distinctive layers of functionality that interact with adjacent layers to form the boundaries of the system. On the user's end, the presentation layer is responsible for any and all external interaction with the system. This encompasses both user's directly interacting with a graphical user interface and automated integrations with other programs and systems using comet as part of the deployment cycle.

3.2.2 Domain Logic Structure

The domain logic for comet is designed to use and extend the base classes provided by the ANTLR parser generator tool to produce the different metrics and structures that comet provides. Comet's domain logic layer relies on the concepts of inheritance, abstraction and polymorphism to conform to the principles of object-orientation and "Don't Repeat Yourself" (DRY). This is accomplished through a carefully designed inheritance structure with base classes that both aggregate common behaviour between its subclasses and provide abstract methods to dependent classes such that functionality is abstracted.

An example is provided for the abstract syntax tree structure. It can be seen that the majority of the AST's functionality is derived from base Graph and Node classes. This allows comet to interact with different graphs in a semi-polymorphic way and enables the creation of abstract and generalised visitors and calculators with coherent and uniform behaviours.



3.3 Languages

3.3.1 Python (version 3.7+)

Python is a high-level, object-oriented programming language. It is dynamically typed with extensive built-in libraries and a plethora of powerful third-party libraries easily available. Comet uses Python version 3.7 or higher as the basis for both the domain logic layer and the application layer.

The decision to use Python came as a result of its inherent flexibility, extensibility and readability, aspects that the team felt were crucial to comet, as an application that relies heavily on abstraction and requires relatively complex calculations and control flows.

Another factor was Python's powerful and sophisticated unit testing framework, PyUnit. PyUnit provided the developers the ability to effectively unit test every aspect of the back-end code, complete with dependency mocking to ensure purity of the tests and prevent unwanted side effects.

Additionally, Python's built-in documentation facilities also provided the team with a standardised, simple way of describing code requirements and behaviour; which, coupled with Python's highly readable syntax, means that comet's backend code is highly readable and easily understood.

3.3.2 JavaScript (ES6)

Javascript is a scripting language that allows the implementation of complex features on web pages through the creation of dynamically updating content, multimedia, animated images among many other things. Comet uses Javascript ES6 due to the multitude of new features that have been introduced through ES6, such as arrow functions and improved syntax.

Furthering this React was chosen to be used as the front-end's main framework. React enables a component-based approach with a well-defined lifecycle, which makes it extremely flexible, testable and performance friendly. As React is a Javascript library, this further cemented our use with the language.

3.3.3 HTML

Hypertext Markup Language is the standard markup language for documents designed to be displayed in a web browser. It defines the structure and content of web pages, and is used both statically as HTML files and dynamically in the form of React components in the delivery of the comet web user-interface.

3.3.4 CSS

Cascading Style Sheets (CSS) is used in tandem with HTML to describe the layout and design of a website. CSS is used extensively within comet to apply styles to each of the subcomponents of the web page to provide a visually appealing and well-designed layout.

3.4 Tools & Technologies

3.4.1 React

Enables the development of encapsulated, reusable components that manage their own state, and the amalgamation of these components into a complex, dynamic UI.

3.4.2 Redux

A state container which helps to ensure consistent behaviour by centralising the application's state.

3.4.3 Django

A high-level Python web framework that enables rapid development of secure and maintainable database-driven web applications[1]. Django is utilised as the basis for communication between the user and the comet domain logic layer.

[1] Django, https://www.djangoproject.com/

3.4.3.1 Django REST Framework

A powerful and flexible extension library for Django that provides the ability to develop web-browsable RESTful APIs with support for routing, serialization and viewset definition[2]. Comet uses the Django REST framework extensively throughout its application layer implementation to define the API endpoint structure and behaviour.

3.4.4 ANTLR

A powerful parser generator for reading, processing, executing or translating structured text or binary files[3]. Used for generating parse trees for program input.

4. Implementation

4.1 Domain Logic

4.1.1 Parse Tree Generation

The domain logic layer of comet is implemented using Python with the ANTLR runtime libraries. Using the ANTLR parser generator tool and the language grammars provided by ANTLR, base classes for lexing, parsing and visiting input are generated for a Python target. These classes provide the functionality needed to generate parse trees that represent the input provided to them, first by lexing tokens and then by applying parser rules. These parse trees, however, are highly language-specific, following the structure outlined in the parser rules for the given language. To derive each of the metrics and models based on these parse trees, therefore, would require a distinct definition of how to calculate each one for every supported language.

4.1.2 AST Generation

To remedy this and better adhere to the principles of “Don’t Repeat Yourself” (DRY), the visitor design pattern is employed to convert said parse trees into a language-independent AST. The visitor pattern is commonly applied throughout comet’s domain logic as a means of traversing the different generated structures so that behaviour can be easily customised and the complexities of recursion are abstracted. This pattern involves writing a visitor with a corresponding “visit” function for each structure “node” that it might visit. Consequently, each of these said nodes includes an “accept” method which simply calls its corresponding visit function in the visitor being accepted. Currently, support for AST generation is

[2] Django REST Framework, https://www.django-rest-framework.org/
[3] ANTLR, https://www.antlr.org/

implemented for Python 3 and C#, two of the most popular and rapidly growing object-oriented languages in industry.

4.1.3 Structure Generation

Once the AST has been generated, other structures (such as CFG, class diagram, etc.) can subsequently be derived, again using the visitor design pattern. from the AST. A base AST visitor is defined with a generation visitor or each structure inheriting from it and overriding its behaviour where appropriate to generate the corresponding structure. For example, when generating a CFG, upon visiting a loop statement node in the AST, a loop block structure is added to the CFG being generated.

4.1.4 Metric Calculation

Each available metric is also calculated using a corresponding visitor for the relevant structure. For example, cyclomatic complexity is calculated by visiting a CFG structure and counting the number of unique nodes and edges encountered during the process before applying the formula.

The reuse of the visitor pattern throughout and the production of base classes for structures with common attributes and/or behaviour allows for a consistent and coherent codebase with very little repetition.

4.2 Application

4.2.1 Models

Models are Python objects which Django accesses and manages data through. These define the structure of stored data and represent a table or collection within a database. This allows Django to handle the communication of data with the database and thus provides simplicity, consistency, version control and advanced metadata handling. For example comet contains a File model, which explicitly states the properties required, in this case the file hash, name, size, type, upload date, and file itself.

4.2.2 Views

A view is a Python function which handles Web requests consequently providing a Web response. Essential to the functionality of comet, enabling arbitrary logic for the request and response to be executed from within the function. As an example in comet's case, the upload view serves the specific functionality of uploading a file and returning a JSON response of the parsed metric information.

4.3 Presentation

4.3.1 Framework

Comet is developed upon the React framework. Due to its fast, scalable and simple design. This is achieved by flexibility and simplicity within the framework's internal design through its HTML-like interface syntax called JSX which generates HTML and CSS.

React enables the development of simple modular components which together build a complex user interface. Each component decides how it should be rendered and each component has its own internal logic. Modularity is the key to comet's components, with components being as decoupled from one another as possible - allowing for in depth testing and assurance that each component will work independently of each other - while also maintaining React's modular style and structure. As a result of this, the internals of the application remain consistent and code reuse enables easier maintenance and growth of the codebase.

4.3.2 Design

The objective of comet's front-end design is to provide an intuitive and succinct representation of a more complicated subject matter. From the minimalist layout to the colours used - all design choices have been rigidly thought out, tested and have received feedback from real world users.

4.3.3 Colour

Colour choice is arguably the most important aspect of engaging user interfaces and it is crucial that the colours used convey meaning, evoke emotion, contrast together and look good as a palette.

Combining a dark blue theme with light blue and white highlights achieves the above requirements. Each colour is chosen because of the emotional associations attached with them; dark blue is associated with depth, expertise and stability; light blue is associated with health and understanding; and white is associated with coolness and cleanliness. The small but contrasting colours create a consistent colour palette across the entire website and encompasses comet's overall goal - to provide in depth details about metrics which can be used to quantify the health of a system.

Red and green are used as action colours, drawing the attention of the user immediately to the point of action when required - such as in the event of a failed or successful upload.

4.3.4 Pages

It was essential to maintain the same level of simplicity and ease of use throughout all levels of the web application, including the website's structure. To ensure this the user requirements were determined and from this two webpages were deduced to be needed; homepage and a metrics page.

The homepage consists of two buttons; create and upload. Both of which upon clicking open a modal with information on how to create or upload their file/s. Upon a successful upload, the user will be redirected to the metrics page.

A metrics webpage consisting of multiple independent structures reduces structure complexity due to their highly decoupled nature. While utilising more complex components, the webpage's design, style and look follows the methodology of simple structuring. A file directory is shown to the left of the webpage, with the right hand side containing a tab toolbar with the corresponding tab's data displayed below. Changes between tabs are fluid with animations easing the required information onto the webpage. As a result the graphical representations and metric information can be enclosed within a singular page without appearing cluttered or becoming confusing.

5. Validation

5.1 Unit Testing

5.1.1 Front-End

Each React component was individually tested via the Jest framework. Each component was required to have 85% code coverage before being integrated into the system. Snapshots were utilised to generate a serializable value for the React component, these could then be compared to the expected object to ensure no unwarranted or unexpected changes had been made during the test.

5.1.2 Back-End

Using PyUnit and its powerful testing and mocking functionalities, comet's domain logic and application layers were extensively tested with test suites consisting of test cases derived for each class being prepared. Each test case contains a set of tests, designed for complete unit testing of each method within the class, that model and examine the expected behaviour and outputs of the corresponding method by mocking its dependencies and asserting correct behaviour.

5.2 Integration Testing (End-to-End)

End-to-end (E2E) tests have been implemented to ensure the correct application flow from start to end, utilising the simulation of real user events and scenarios to

validate the system's flow. The Puppeteer framework was used to execute the steps a real-world user would take when using the web application; from the clicking of a button to the uploading or creation of files and the viewing of uploaded metric data, among other things.

E2E tests are vital in assuring the correct behaviour and longevity of the system as they encapsulate the behaviour and communication between both the front and back end, asserting that there is no deviances in expected behaviour.

5.3 System Testing

The system as a whole was tested by regularly deploying stable iterations of the product to an Amazon Elastic Compute Cloud (EC2) instance that was configured to operate as an Apache web server. This provided the opportunity to analyse comet's performance in a production context and ensure that the system behaved cohesively and as expected in a "real-world" environment.

5.4 Acceptance Testing

The aforementioned Amazon EC2 instance was also configured with a domain name URL of its own, which allowed comet to be easily shared with potential users for feedback and criticism. The URL was shared with a number of trusted and valued peers and industry professionals and plenty of constructive criticism and insights were obtained.

The user opinions were ultimately very positive, with most users remarking that they loved comet's flexibility and clean user interface, in many cases stating that comet is a system that they would use again in both professional and personal capacities. There was a consensus that, with further development, including the support of additional languages and providing dedicated integrations with popular CI/CD tools, comet has a considerable scope for utility as a tool for improving the software development life cycle.

6. Conclusion

6.1 Encountered Problems and Resolutions

6.1.1 Unfamiliarity with Technologies

While both developers of comet are well acquainted with the language of Python and hold a relative familiarity with many of the front-end languages and tools used throughout comet's development, many of the technologies encountered were totally novel. The most prevalent example of this is ANTLR, a tool with which neither

team member had any prior experience. Furthermore, both developers had only limited prior knowledge of React, Redux and Django.

A commitment was made early on in the development process for each member to endeavour to research, experiment and familiarise themselves with all aspects of the project to ensure competence and that best practices were being followed to provide optimal performance and maintainability.

6.1.2 Abstract Syntax Tree Derivation

A primary challenge faced over the course of development was distinguishing how to derive an abstract syntax tree, that not only provided the base level of functionality required for the calculation of metrics and models, but did so in a manner that balanced representing each of the supported languages in an abstract manner while reducing omission of their respective nuances. This was achieved by careful analysis of the underlying grammars for each of these languages to identify both the commonalities between them and the features that were pertinent to metric and model calculation.

6.2 Future Directions & Development

6.2.1 CI/CD Integration

CI/CD is an integral part of efficient development and thus an extension of comet should be able to provide useful metrics during the CI/CD pipeline. Currently, a client could hit the comet API and receive metric information before parsing the response for the required values. However, this would ideally be configurable from the client side and be included within the request. A client would be able to state the acceptable metric values within the request, such as cyclomatic complexity per function being no more than 10, and receive a boolean value on whether the stated requirements were met or not. This functionality would enable developers to ensure required standards are met before changes are accepted to a branch.

6.2.2 IDE Extension

Ideally comet would be an extension of a users already existing workflow, rather than an addition in steps taken to get a result. Thus the integration of comet directly into a work environment, such as VSCode, would allow users to access the benefits of comet while maintaining their general workflow. Enabling the unpacking of comets powerful tools into an IDE.

6.2.3 Language Support

The driving motivation behind comet is to facilitate extensibility and to be language-inclusive. A natural future direction therefore would be to establish

support for more languages, perhaps branching to different paradigms to provide even more utility and flexibility.

6.2.4 Caching

The addition of a caching functionality to comet would prove beneficial for response times and overall system efficiency. Creating a temporary store of user files which have been uploaded to their existing calculated metric information would result in only requiring the generation of metrics for each unique file that was uploaded, rather than all files as a whole.