

Marcus Craske

Registration number 6195601

PALS – Programming Assessment and Learning System

Supervised by Dr Pierre Chardaire



University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

The automatic assessment of programming skills has been an emerging area since the 1960s, linked with disciplines such as computer aided assessment (CCA) software and software metrics, with many existing systems abandoned or non-existent. This project aims to create a new cross-platform system for assisting self-guided formative learning and summative assessment, called PALS – *Programming Assessment and Learning System*.

The Java programming language is the focus of assessment, with extensibility for new languages in the future through a plugin architecture, which is conducted through dynamic and static analysis within a secure and sandboxed environment. The system also provides administration tools to facilitate the easy and quick creation of material, collection of basic statistical information and management of users and modules.

There is also support for clustering multiple instances across machines to handle large workloads and for additional security.

Acknowledgements

A great amount of gratitude is due towards the CMP faculty for their mentoring over the last three years, especially Dr Pierre Chardaire, my project supervisor, for his invaluable feedback and support.

Contents

1. Introduction	1
2. Existing Work	2
2.1. Evaluation	8
3. Project Planning	12
3.1. Requirements Analysis	12
3.2. Time Management	13
3.3. Software Engineering	14
4. Design and Implementation	15
4.1. Architecture	15
4.1.1. RMI Communication	15
4.1.2. Storage	16
4.1.3. Settings	16
4.1.4. Core	16
4.1.5. Website	16
4.1.6. Database	17
4.1.7. Plugins	19
4.1.8. Hooks	20
4.2. Plugins	20
4.2.1. Assignment Marker	20
4.2.2. Default Auth	21
4.2.3. Miscellaneous	21
4.3. Applications	22
4.3.1. Java Sandbox	22
4.3.2. Windows User Tool	23
4.3.3. Node	24
4.4. Administration	24
4.4.1. Questions	24
4.4.2. Modules & Assignments	25

4.4.3. Users and Groups	25
4.4.4. Mass-Enrolment	26
4.4.5. Stats	26
4.4.6. System	27
4.5. Security	27
5. Design and Implementation – Assessment	28
5.1. Question Types	29
5.1.1. Code Java	29
5.2. Criteria Types	30
5.2.1. Matching	30
5.2.2. Java – Code Metrics	31
5.2.3. Java – Testing Inputs	31
5.2.4. Java – Testing Standard Input and Output	32
5.2.5. Java – Testing Existence	33
5.2.6. Java – Enum Constants	34
5.2.7. Java – Inheritance & Interfaces	34
5.2.8. Java – Custom Code	34
6. Evaluation	35
6.1. Completion	35
6.2. Previous Systems	35
6.3. Problems Encountered	36
6.3.1. Java Sandbox	36
6.3.2. Assignment Marker	36
6.4. Throughput Testing	37
6.5. Limitations & Future Work	38
6.5.1. Method Invoked	38
6.5.2. Multiple Programming Languages	38
6.5.3. Assignment Marker	39
6.5.4. Resits	39
6.5.5. Marking	39

6.5.6. Randomised Snippets	40
7. Conclusion	40
References	41
A. Organisation of the accompanying submission disc	44
B. Entity Relationship Diagram	44

1. Introduction

The automated assessment of programming skills has been around since the 1960s, with the first system, *Automatic Grader*, running a student's program from a punch-card and comparing the results to the correct answers stored elsewhere in memory (Hollingsworth, 1960). Three generations of systems, as defined by Douce et al. (2005), have since emerged. The first generation required modifications to operating systems and compilers, as seen with Hollingsworth (1960). The second generation consisted of graphical-interfaces, scripts and tools; with systems such as ASSYST (Jackson and Usher, 1997) and Ceilidh (Benford et al., 1995) using UNIX shell scripts and tools, assessing work with techniques such as dynamic profiling, software metrics and testing code-correctness. The third, and present, generation of systems use more sophisticated assessment techniques and web-based interfaces, with BOSS (Joy et al., 2005) allowing for unit testing with the Java programming language, and CourseMarker (Higgins et al., 2003) *capable* of being segmented over a network and assessing diagrams.

With some institutions enrolling *hundreds* of students each year, each producing multiple programs, the marking of programming skills is *difficult* and *time-consuming*, which can result in fewer exercises and thus less experience for students (Blumenstein et al., 2004). Human marking can also be subjective.

Many programming assessment systems lack features and are no longer under development; almost all of them, with the exception of BOSS, are not open source and thus cannot form the basis of new systems, as found by Ihantola et al. (2010b) and myself. CourseMarker, originally called Ceilidh, being most notable, and also obtained by 43 academic institutions (including the University of East Anglia) (Symeonidis, 2006); it also required licensing. The only pre-existing active system found was BOSS, with its main purpose being a place to upload and store assignments with some automated testing (such as using JUnit) (Joy et al., 2005). Most systems would only provide certain features, with CourseMarker not natively supporting multiple choice questions without external tools. Style++ only assessed typography (indenting, comments) and analysis of variable usage (incorrect data-types, uninitialized variables) for C++ (Ala-Mutka et al., 2004). QuizPACK would only partially randomise snippets of provided code for questions in a timed environment (Brusilovsky and Sosnovsky, 2005).

This project has created a new third-generation system called PALS, *Programming Assessment and Learning System*, which allows the assessment and, through formative exercises, learning of programming skills. This is achieved by combining and building-upon the pre-existing work in areas such as dynamic and static analysis with uploaded programs and the ability to create different types of questions (multiple choice, single response, code).

This report first begins by looking at *existing work*, through a literature review, with an evaluation to highlight the important points learned and how it can be applied to the context of my project. This is followed by *project planning*, which covers: requirements analysis, planning and software engineering. Then the design and implementation phase of the project, which discusses the details of how PALS has been created. This section is broken-up into parts, discussing the general architecture, the initial plugins used to form the overall system's functionality, the assessment of students, administration, for lecturers and system administrators, and security. A reflection is then made on the progress and issues experienced with the project, through an *evaluation*, with the report ending on a *conclusion*.

2. Existing Work

The requirement for a system capable of automatic assessment of programming skills predominantly comes from an increase in the number of students enrolled on programming courses/classes and the lack of resources to promptly mark and provide feedback (Joy et al., 2005; Higgins et al., 2003) from manual assessment. It has also been mentioned that the task of manual assessment is *laborious and error-prone* (Jackson and Usher, 1997), with marking hard to keep *fair* (Cheang et al., 2003) because of the possibility of inconsistency between markers (Thorburn and Rowe, 1997). The automating of assessment of programming skills allows for objective marking and more exercises to be created, because of less time and resources required for marking, giving students the potential to gain more experience and feedback (Saikkonen et al., 2001; Cheang et al., 2003).

An automated assessment system can also become a learning environment, whereby

students can learn from their mistakes with sufficient feedback; this is useful for students learning remotely (Higgins et al., 2003). This is also one of the intentions of Style++ (Ala-Mutka et al., 2004), a tool for typographic analysis of C++, where students run the tool on their code to produce an analysis on their coding style. Higgins et al. (2005) states too much feedback can have a reverse effect, whereby students do not think about how to improve their work.

Relying on a system to entirely conduct assessment has not been the case, with ASSYST (Jackson and Usher, 1997) being used as a tool for assisting with spotting errors missed from human marking; this is due to trust issues with both the lecturers and students. This is not surprising, with systems and tools such as Style++ able to produce false negatives from consistency issues with a programming language's specification (Ala-Mutka et al., 2004). Joy et al. (2005) argue elements such as structure, commenting and the correct usage of language constructs are better to be marked by a human, who use automated techniques to assist and speed-up the process of marking; this has been employed with BOSS, which is a system intended for the storage and submission of assignments. However, SchemeRobo (Saikkonen et al., 2001) is completely automated because of a lack of human resources to provide feedback instantaneously; this may also be because the system is aimed at having a greater number of smaller exercises, whereas BOSS is marking entire assignments.

The methods of assessment used for code in most systems include white box and black box testing and typography analysis; some systems are able to determine authenticity (such as plagiarism detection) and provide questions such as multiple choice / response.

White box testing looks at the internal structure of code, which is commonly achieved using software metrics, such as: lines of code (LOC), average length of identifiers, number of blank lines (Thorburn and Rowe, 1997); without any program execution – a process called static (code) analysis. McCabe's cyclomatic complexity metric (McCabe, 1976), as seen in ASSYST, calculates the minimum number of linear independent paths of a piece of code, $V(G)$. This is used as an indicator of maintainability because of the amount of testing required. This is achieved by converting code into a control flow graph and applying the following formula, for a graph which is not strongly-connected

– meaning the connected components, or exit nodes (in this context), do not link back to the root node:

$$V(G) = e - n + 2p$$

Where e is the number of edges, n is the number of nodes and p is the number of connected components. This metric has been criticised by Shepperd (1988) as lacking *theoretical foundations*, with the value of $V(G)$ remaining as one for a linear sequence of statements for any length, and lacking the depth and context of a decision.

Another white box technique is testing for the existence of features, such as: methods, classes and even constructs, as seen in CourseMarker (Higgins et al., 2003). This can also be used to blacklist the usage of certain libraries or constructs; this was used by Saikkonen et al. (2001) for when students had to reimplement a feature, of the Scheme language, for reversing a list. Their system could also convert code into an abstract syntax tree, which could be compared against a model skeleton.

Black box testing is generally used to determine *code correctness* by ensuring a program meets its required specification, by testing the inputs and checking for the correct output. A common technique, used since the first generation of systems, is dynamic analysis, where a program is executed and the output is compared against a model (Hollingsworth, 1960; Higgins et al., 2003; Cheang et al., 2003). Ceilidh achieves dynamic analysis by using a separate tool (creating an additional operating-system process), which takes the output of a program and uses regular expressions to determine if it is the same as a model output provided for an exercise (Benford et al., 1995). Online Judge (Cheang et al., 2003) and ASSYST (Jackson and Usher, 1997) also measure the efficiency of a student's program based on its execution time.

Instead, SchemeRobo (Saikkonen et al., 2001) and PASS (Thorburn and Rowe, 1997) compile an individual function and test a range of values, with PASS using a subset of randomly picked numbers between a range (stating it would be *impractical* to test the full range of possible inputs). BOSS uses a third-party tool called JUnit, where classes are created to test individual methods by passing test data and asserting conditions on the output.

The assessment of coding style is achieved using typographic analysis, such as: naming conventions, commenting and indentation. Style++ was created for students to self-

analyse their code and fix common issues, as well as to aid teachers in the marking of work. The article states that students wrote more *reliable and understandable* code, and it also provides a table displaying the frequency of style issues, in programming assignments, before the usage of Style++. However, this table lacks the frequency of issues after the usage of Style++, thus this claim is questionable because there is no comparative group.

Testing just code alone may be insufficient to test the programming skills of a student, since they may not entirely understand what they are writing. Ceilidh allowed multiple choice questions and written response exercises (Benford et al., 1995). CourseMarker, created as the successor to Ceilidh, does not natively allow multiple choice questions and requires students to either run a separate program or for the system to use an external tool to mark submitted code as a multiple choice question (Higgins et al., 2003).

Such testing, with static questions, presents plagiarism issues, especially in non-controlled scenarios such as away from a classroom; one approach used in CCA systems, in areas such as maths and physics, is parametrized questions. Due to a lack of such a system for programming, Brusilovsky and Sosnovsky (2005) created QuizPack. The system allows the same fragments of code to be used with parameters for generating random, as opposed to having static, values; these fragments of code are then compiled and executed, with the value compared against the student's answer. An issue with non-controlled scenarios is that students could compile some questions for an answer. Also, most programming assessment systems reviewed only allow the assessment of skills through code and lack other methods.

Mark distribution varies between systems. BOSS suggests the mark to be given to assignments for different criteria, with humans able to adjust and finalize the final mark. Ceilidh uses a configuration file, with different criteria assigned a weight. It also has the ability to scale marks, for metrics, using the technique established by Rees (1982), with marks scaled between four thresholds (in-order of value, low to high): *lo*, *lotol*, *hitol* and *hi*. A metric value between the thresholds of *lotol* and *hitol*, typically the lower-upper and upper-lower bounds of a metric's value range, receive full marks. A value between *lo* (the minimum value for the metric's range) to *lotol* or *hitol* to *hi* (the maximum value for the metric's range) is scaled between full marks to zero marks. The

idea behind this technique is to penalize code with e.g. too many or too few comments. This technique is also used by ASSYST and Style++ (Jackson and Usher, 1997; Alami-Mutka et al., 2004). PASS and Online Judge do not go into detail about how marks are rewarded. CourseMarker requires a separate Java library to be programmed and dynamically loaded at runtime for each exercise, which instructs the system on marking.

Architecture is possibly another concern due to the potential for a high volume of work, with CourseMarker having been used to mark more than 3,000 assignments per week at the National University in Singapore (Higgins et al., 2003). It uses an architecture where different parts of the system are split-up into *virtual* servers; these servers can use Java RMI (Remote Method Invocation) for communication, allowing them to be physically segmented across a network on different machines. A web client can be used to perform administration tasks, such as the creation of exercises, and a Java Swing (GUI (Graphical User Interface) library) client is used by students to submit work and for teachers to monitor student progress – which also uses RMI for communication. A remote server tool also allows debugging, statistics and control of the *virtual* servers. These servers consisted of: *submission* for handling the submission of work and invoking other servers, *course* for managing course materials, *Ceilidh* for course information and *login* for student authentication. The code of the servers has to be modified and recompiled if the system requires a new external tool/process to mark work; this also requires the system to be restarted.

Its predecessor, Ceilidh, used a monolithic architecture running within a single process (Benford et al., 1995); this process would invoke external tools as separate processes to mark work. Higgins et al. (2003) state that Ceilidh was also hard to configure and its text-based interface was difficult to use, even though they also state a web interface was later created; something which CourseMarker did not have. It is not clear if the web interface was only accessible on the same local machine, as was the Ceilidh process, and if the database layer (Higgins et al., 2003) was networked.

BOSS is similar to CourseMarker, with segmented virtual servers and a Java Swing client for students (Joy et al., 2005) using RMI with Secure Sockets Layer (SSL) security. The *virtual* servers (separate processes) include: an automated test server, student server (for authentication, submission of assignments and logging), staff server (for

marking, moderation and testing), Sherlock (a plagiarism detector) and a secure web server (an alternative to the Java client). Since BOSS and CourseMarker use Java, they are also cross-platform and able to operate on Windows and UNIX.

ASSYST is similar to Ceilidh, and also restricted to UNIX; however, students interact with the system using a GUI (Jackson and Usher, 1997). Style++ runs as an independent UNIX tool, outputting the analysis of a file to console (Ala-Mutka et al., 2004). Online Judge runs as a process on a UNIX server, opening-up a port to receive submissions sent using a *wrapper program*; information about the system can be viewed using a web interface, which uses CGI scripts (Cheang et al., 2003). QuizPack's platform support is not discussed, but students interact with the system through a web-interface (Brusilovsky and Sosnovsky, 2005). PASS is similar to Style++ in architecture, but restricted to Windows computers using IBM-PC machines; the type of interface (command-line or graphical) is not specified (Thorburn and Rowe, 1997). SchemeRobo is quite different from any other system, with students interacting with the system using e-mail. New exercises are sent by e-mail, with students sending their work back and receiving feedback automatically. The platform of the server is not stated (Saikkonen et al., 2001).

A part of authenticity is determining the identity of a user, such as a student or teacher, through authentication. Most systems have an isolated list of usernames and passwords with user-groups or permissions. However, CourseMarker offers an alternate option, for students, by using a POP3 mail server (Higgins et al., 2003).

Hollingsworth (1960) discussed a limitation, whereby student programs could modify the assessment system. This issue has been resolved in systems such as Ceilidh and CourseMarker (Higgins et al., 2003) by performing white box feature checking and executing student programs under an account with limited permissions.

Users, besides using malicious code, can abuse a system by brute forcing a solution or using the system as a compiler; this was noticed by Benford et al. (1995) with Ceilidh, where students changed their code to see how their marks would change, and Cheang et al. (2003) with Online Judge. The system, Online Judge, used a queue for processing assignments based on first in, first out (FIFO) priority. This resulted in students poorly checking their programs and even writing programs to flood the queue – inflicting a

denial of service attack. Their solution was to limit the number of resubmissions and to only allow one item in the queue, per user, at one time. Most systems, like SchemeRobo (Saikkonen et al., 2001), also enforce an execution time-limit; this protects against malicious or buggy programs from indefinitely running.

Benford et al. (1995) also discuss the benefits of administration duties, with the ability for their system to inform teachers of students not submitting work and collect *160 paper reports on a given date* without the need for physical boxes and manual processing, eliminating the possibility of lost work and significantly speeding up the process. BOSS also allows teachers to attach feedback to submitted work, with a minimal number of keystrokes for marking by automatically filling out marks from automated tests (Joy et al., 2005). Higgins et al. (2005) also mention using a modified Java compiler, which aggregated compilation errors. The most common errors would then be used to change lecture and exercise material.

2.1. Evaluation

This section evaluates the literature review, on existing work, by looking at the problems, techniques, features and areas for improvement, and what they mean for the project.

There is an evident need for automated assessment systems and tools, primarily due to a high number of students enrolling – although it is possible this factor has or could change. However, automated assessment has other motives: requires less resources, provides feedback faster (and instant in some cases), less error prone and a reduction of administrative duties.

Some systems and tools such as SchemeRobo and CourseMarker allow entirely automated marking with instant feedback, whereas BOSS provides the storage of assignments, automated processing of work and tests and finalized marks from teachers. Style++ and ASSYST assist with manual marking by automatically assessing assignments and providing an analysis and spotting issues. This raises the question: how automated should a new system be? This would appear to be based on the usage of an assessment system in terms of formative/summative assessments and self-learning exercises; thus PALS should be open to allowing a mix of automated and manual marking.

Something such as false negatives, as with Style++, would not be as much of an issue during formative, as opposed to summative, assessments.

The methodology for assessing programming skills should also be open, with many systems assessing just code alone. A student may be able to produce code, perhaps through trial and error as seen by Cheang et al. (2003), but they may not entirely understand what they are doing. QuizPack is a solution, but it does not allow for other methods of assessment and it is open to being cheated by compiling parametrized code. Joy et al. (2005) mention questions are suitable for testing comprehension and knowledge, but not enough for testing skills alone. Therefore, PALS should assess students through both questions and code.

Testing code should use white box and black box testing with typographic analysis. Black box testing is important to ensure a program has code correctness, meeting its specification. However, using a subset of random values from a range of possible values for numeric inputs, as used by PASS, is not sufficient because there is the possibility for an incorrect output. This could result in an incorrect program receiving marks. Therefore, PALS should, where possible, test all of the possible inputs or use both random values and human-provided cases where errors are likely to occur; with test inputs defined by a human, when creating exercises.

Measuring the execution time of a program to evaluate efficiency (as seen with ASYST), should be avoided. Different systems could have different execution times and the operating system's process scheduling may interfere. A time restriction will need to be added to stop buggy or malicious programs from indefinitely running, as mentioned with SchemeRobo.

Black box testing also presents many security issues, especially when looking at flaws in previous systems. A previous version of CourseMarker compiled student programs on client machines (Higgins et al., 2003, 2005), which seems dangerous because the compiler on the client machine may be compromised. Thus the program sent from the client may consist of different, and possibly malicious, code. If the server executes the program with incorrect privilege restrictions, the entire system could become compromised. The platform/architecture of the client and server may also be incompatible with the compiled program, and unable to execute on the server. Therefore, compiling should

completely take place on the server.

With code being compiled on the server, it could also undergo white box testing before dynamic analysis, or any form of executing code, for security. SchemeRobo allows for the blacklisting of keywords. Therefore, a new system could expand this feature by implementing white (allowed) and black (disallowed) listing for constructs and libraries. Since students can also upload assignments, unnecessary files should also be removed – something which has not been mentioned.

PASS also discovered an issue whereby students could define functions and never call them, but write the code within their program's entry-point function. Therefore, if static analysis checked for the function and dynamic analysis produced the correct output, the student's program would be correct. Thus feature checking should also check a function is invoked or a class has been used. This limitation of PASS has not been mentioned elsewhere in literature and may be an issue in assessment systems with feature checking such as CourseMarker.

As found with Online Judge, it should be possible to place a limit on resubmissions to avoid students brute forcing solutions. To avoid flooding the queue, it should be made mandatory for only one of each assignment/exercise per student to be in the marking queue at one time. Online Judge restricts one submission per student instead. However, a student may want to move onto the next exercise; therefore, using this type of restriction could impede a student from working.

Jackson and Usher (1997) acknowledges the criticisms of McCabes complexity metric, implementing it in ASSYST to measure complexity because of being *well-known* and *easy to automate*; a future system may implement the metric, but it should be optional since it could be irrelevant – this could be the same for all metrics. Therefore, when creating exercises with PALS, the specific metrics used on code should be specified.

The new system should also use a web interface as a client for interfacing with the system, for both teachers and students, and Java for the server-side. BOSS uses both a web interface and client. However, this seems pointless because additional features would require two separate projects to be extended – which is wasted effort. A web interface is more suitable because it does not require a client to be installed and future

updates deployed. The serverside should use Java because it supports multiple platforms; it would also allow any code to be migrated between PALS, CourseMarker or BOSS.

Java also allows for dynamic loading at runtime of external libraries, as seen with CourseMarker for dynamically loading marking programs. A similar approach could be used for reloading certain functions of the system, a limitation discussed by (Higgins et al., 2003) where CourseMarker had to be restarted. This would also allow a system to load functions required for marking new languages, overcoming the limitation seen in Style++ and SchemeRobo of supporting only a single language. Different functions could also exist for allowing different ways of authenticating users. Therefore, institutions could integrate PALS into existing infrastructure, without the need for an isolated list – which could otherwise become inconsistent.

These functions could also be separated across physical machines, as seen with BOSS and CourseMarker; this would allow the system to continue functioning in the event a function failed. Support for clustering could also be added, which has been stated by Higgins et al. (2003) as being possible in CourseMarker (although the authors do not make a distinction between the feature being implemented or simply possible as a future extension). Using RMI, as used with BOSS and CourseMarker, may be unsuitable; a defined protocol would allow platforms, unable to use RMI, interface with the system.

Marking should also not require an external library to be dynamically loaded by the system. This would require a lecturer to create a mark program/library for each exercise, which could take a lot of time. A simpler option would be to evolve the idea of using a configuration file, as seen in some of the systems and tools reviewed such as ASSYST and Style++. Feedback from marking also needs to be appropriate; too much can have a negative effect on learning, as mentioned, and too little could be insufficient for a student to self-learn. The level or type of feedback provided could also be set when configuring an exercise. The collection of errors would most likely be useful as feedback for teachers; rather than use a modified Java compiler, errors could be parsed from compilers using standard-output.

CourseMarker, and its predecessor Ceilidh, have been referenced by a lot of literature (Joy et al., 2005; Jackson and Usher, 1997; Ala-Mutka et al., 2004; Brusilovsky and

Sosnovsky, 2005; Saikkonen et al., 2001; Thorburn and Rowe, 1997). Therefore, an observation can be made that it is one of the most important automated assessment systems in this area; and yet their system has disappeared. It was also found that a lot of the other systems and tools mentioned no longer exist, or are no longer developed, with the exception of BOSS; this was also found by Ihantola et al. (2010a). They also mention that a lot of systems share common features, as we have seen, but many systems are written for specific courses, not suitable for distribution, or closed source due to being licensed, as with CourseMarker (Higgins et al., 2003). A solution Ihantola et al. (2010a) propose is for new systems to be made open source; this would allow a new system to be continually developed by people besides the original authors. It would also allow new features to be added, which may have not been originally considered. Therefore, PALS should be made open source at the end of this project.

3. Project Planning

3.1. Requirements Analysis

MoSCoW was used to determine the scope and priority of work to be implemented and excluded, which is also useful for assessing the success of the project and to clearly define a schedule of work to be completed.

Must have:

- Ability to create and manage: users, assignments and questions; without the need for external libraries to be created.
- Ability to assess code using dynamic and static analysis.
- Security for dynamic analysis, to restrict malicious or accidental damaging or/and disruptive behaviour.
- Code compilation taking place on the server-side.
- FIFO processing of assignments with limits on resubmissions to avoid brute-forcing and indefinite postponement.

Should have:

- A web interface for accessing the system.

- Assessment able to be automatic or/and manual.
- Multiple methods of assessment, besides just code.
- Assignments and users organized into modules.
- Inputs, used to test code, defined by a human.
- Appropriate feedback from automated assessment.
- A cross-platform system.

Could have:

- A system able to be extended with multiple languages.
- Ability to test code with random inputs.
- A system capable of being segmented or distributed across multiple nodes.
- Checking of methods being invoked to fix the issue mentioned by PASS.
- Uploading of student projects, with removal of unnecessary uploaded files.
- Assignment deadlines.
- Libraries reloaded during runtime.
- Collection of compiler errors for stats.
- Testing of both standard output and input.

Wont have:

- McCabe's cyclomatic complexity metric.
- Multiple languages – a single language should be sufficient to demonstrate the potential of the system.
- Testing of graphical user interfaces.

Use case diagrams and descriptions were also created, which can be found in the appendix (section A).

3.2. Time Management

A GANTT chart was used to keep an overview of the time schedule for items, with a to-do list to track sub-items. The to-do list was also useful for tracking bugs and for logging some feedback from project advisor meetings. Refer to appendix for chart and to-do items (section A).

3.3. Software Engineering

The code base has an underlying framework/library called the *base*, which is used for starting instances of nodes, and by plugins (explained later). The base was developed with a software development cycle, which consisted of the following phases: analysis of requirements, design, implementation and testing; these phases were repeated in multiple short iterations. The code-base was initially continuously integrated with my virtual private server (VPS) to store backups, which evolved to automated testing using JUnit and Jenkins, running each time changes were committed. The unit tests would be ran on my Windows machine, as well as the VPS, which was running Linux; therefore testing was done on multiple platforms.

Design patterns have also been used to help reusability, with much of this paragraph explained in the next section. A singleton pattern is used on *NodeCore* (*pals.base*), allowing for instances of nodes to be created once during a runtime, to avoiding conflicting classes from dynamic plugins. This same class also acts as a facade with sub-system components, which can be added, developed and possibly removed over time. The same class also acts as a factory for *Connectors* (*pals.base.database*), which uses the template method pattern to allow custom wrappers for database connections. A factory, *SSL_Factory*, is also used to create client/server SSL sockets for RMI. Template-hook pattern is also used for plugins, which also uses the observer pattern for a hook mechanism. The *WebManager* (*pals.base*) uses the chain-of-responsibility pattern for serving web requests, where multiple plugins may be able to handle a request or page-not-found request. *ThreadPool* (*pals.base.utils*) uses the thread-pool pattern. *RMI_DefaultServer* uses the visitor pattern to allow remote operations on sub-components.

All code has been documented with Javadoc, which allows future developers to extend and support the system, and a class diagram has been created for the base. Both can be found in the appendix (section A).

Since the system needed to be open-source, the MIT license was selected, under which the code would be distributed. This allows for both commercial and non-commercial usage, whilst forcing future distributions to retain the same license for the original code. The aim is to give a body of work which can be either extended or used for a newer system in the future, to avoid problems found by Ihantola et al. (2010b).

4. Design and Implementation

4.1. Architecture

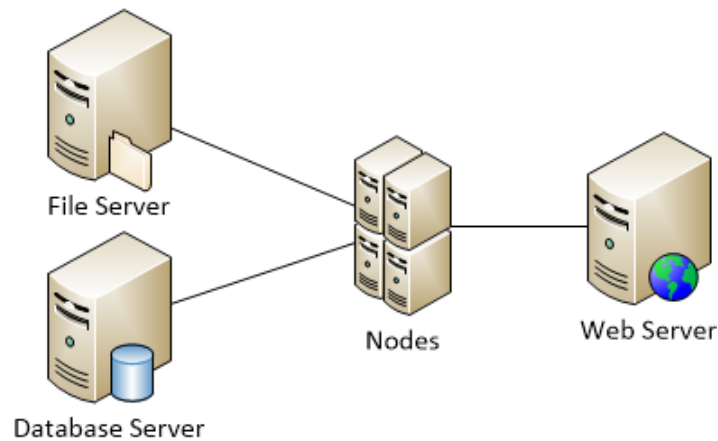


Figure 4.1: Diagram of a PALS system.

Unlike previous systems in the literature review, PALS can be distributed across multiple host machines, as multiple processes. These instances, of processes, are referred to as *nodes* (figure 4.1). This section discusses the design and implementation phases, by looking at the architecture, plugins and administration of PALS.

4.1.1. RMI Communication

Each node communicates using RMI, as seen in literature with BOSS and Course-Marker, which can use a custom socket factory to create SSL connections. When used, this protects against man-in-the-middle attacks, on a network, between a node and web server, since traffic is encrypted. This also prevents a student from maliciously invoking available methods on a node. SSL private and public certificates are loaded from a Java Key Store (JKS), which can be easily generated using *keytool* in the Java Development Kit (JDK).

4.1.2. Storage

All of the nodes require a shared directory, such as Samba or a Windows file-share, for sharing: temporary web uploads (uploaded files to a website are transferred here), templates for rendering a web-page and assessment files – such as compiled byte-code and test files.

4.1.3. Settings

Each node has a *node.config* XML file, with: database settings, the node's universally unique identifier (UUID) for identification, mail-server settings, path of the shared storage, account credentials for sandboxing (discussed later) and embedded web-server settings.

4.1.4. Core

Each node contains a core, called *NodeCore*, which uses a singleton pattern – so only a single instance can be created. It is responsible for starting and stopping a node, as well as acting as a facade between system components and plugins.

4.1.5. Website

The website (figure 4.1) only serializes web requests, which are delivered to a node using RMI, for processing and rendering; this means their traditional role of handling an entire request is not the case with PALS. This is because PALS allows the system to be extended with plugins (discussed later), which allows plugins to handle requests and remain within, and have access to, a node environment. This also avoids issues with the web application pool stopping after a period of inactivity and plugins being loaded at both the website and node ends.

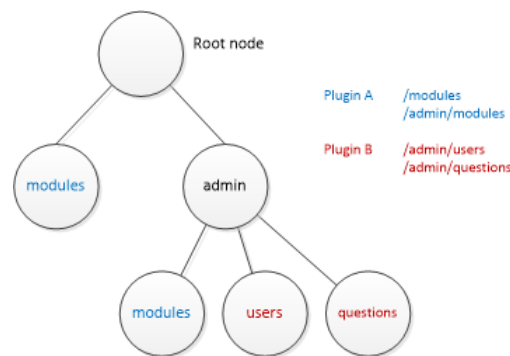


Figure 4.2: A URL tree for storing relative paths handled by plugins.

When a plugin is loaded during runtime, it registers the relative paths of uniform resource locators (URL) it handles, with a component called the *web manager*. These paths are split-up by each directory and placed inside a data structure called a *URL tree* (figure 4.2), where each node from the root forms the full relative path, allowing multiple plugins to handle different sub-directories of a path. When a web request is received, the tree is searched and the plugins at the deepest levels of the tree are invoked, until the request is served – else a 404 event occurs. If the bottom-most directory of a relative URL is not present in the tree, the plugins of the parent nodes are invoked; this means each possible relative path does not need to be registered, only the roots, which is useful for dynamic paths.

Another component called the *template manager* is used for rendering content, using an open-source library called FreeMarker. A custom handler can then load templates from shared storage and from the Java Archive's (JAR) of plugins, which FreeMarker caches. The benefits of this methodology, over traditional JSPs (JavaServer Pages), is that rendering is used for both web and e-mail content, which can be extended for anything else, and it can still feature dynamic content like JSPs. This also allows web-content to be rendered on nodes, outside of a traditional web application environment.

4.1.6. Database

The database currently only supports PostgreSQL, which was initially chosen due to its powerful PL/SQL. MySQL was also supported in the initial implementation. Due to

a limited amount of time for testing, support has not been added back, but the system could be extended in the future.

Session data is stored in the database (figure B.2), which allows multiple nodes to process web-requests, with each session holding a row in *pals_http_sessions*; this table contains a flag to indicate if the session is private and a timestamp of when the session was last active. Each session can have serialized data stored under a key-name, which is stored in *pals_http_session_data*, which is currently used for storing cross-site request forgery (CSRF) and captcha strings, and the identifier of the user tied to the session. This can be extended by plugins, by simply specifying a key and a serializable object.

The assessment part, of PALS, first starts with a pool of questions (*pals_questions*), which derive from a type of question (*pals_question_types*). Each question can consist of multiple criteria (*pals_question_criteria*), which each derive from a criteria-type (*pals_criteria_type*). Questions can then be included in multiple assignments (*pals_assignments*), multiple times (*pals_assignment_questions*). Each assignment belongs to a module (*pals_modules*), where each can have multiple users enrolled (*pals_modules_enrollment*).

When a user takes an assignment, an instance is created (*pals_assignment_instance*), which allows an assignment to be taken multiple times. Each question is processed and rendered by a plugin, which is determined by the question-type; the plugin can then create an instance of the question for the instance of the assignment (*pals_assignment_instance_question*), which can store serialized data – such as a user’s answer.

On submission, an instance of a criteria (*pals_assignment_instance_question_criteria*) is created for the instance of assignment, for each criteria in every question. These criteria are then processed by the cluster of nodes, which each lock the *pals_node_locking* table (as a semaphore) in *ACCESS EXCLUSIVE* mode, to avoid repeating similar work, and fetch unmarked instances of criterias.

Once all of the instance of criterias have been processed and marked by plugins, on nodes, the mark for each instance of an assignment question is computed (formula A in figure 4.3). Each criteria can have different weights (*criteria weight*), and assignment questions themselves can have different weights (*question weight*). These weights allow

more marks to be allocated towards more important criterias and questions. Once the mark has been computed for each assignment question, the overall mark for the instance of assignment is computed (formula B in figure 4.3).

$$question\ mark = \left[\frac{\sum [criteria\ mark \cdot criteria\ weight]}{\sum criteria\ weight} \right]$$

(a) For an instance of an assignment question.

$$mark = \left[\frac{\sum [question\ mark \cdot question\ weight]}{\sum question\ weight} \right]$$

(b) For an instance of an assignment.

Figure 4.3: Computing grades for assignments.

4.1.7. Plugins

Plugins allow PALS to be extended dynamically, without the entire code base requiring recompiling each time a modification is required. Plugins can also be loaded and unloaded during runtime, meaning an instance of PALS does not need to restart for changes to the runtime to occur, unless changes are required to the base; this is handled by a component inside the core called the *plugin manager*. Plugins are also reloaded, automatically, when the main Java Archive (JAR) file is modified, and unloaded when deleted.

Plugins are distributed as a JAR file, which have to be placed in a directory specified in a node's configuration file. They also have their own configuration file, which specifies:

- **UUID** – a universally unique identifier (UUID) is used for identifying a plugin. Rather than give an auto-incrementing identifier from a database, plugins have a unique identifier so they can specify the UUIDs of the other plugins they are dependent upon – which can be optionally defined in this configuration file. Since a UUID is 16 bytes (2^{128} combinations), a collision is unlikely due to a very high amount of entropy.
- **Versioning** - the *major*, *minor* and *build* version information. This is checked against the database to ensure all nodes are running the correct version of the

plugin.

- **Class-path** – each plugin contains a class, which extends a class in the base called *Plugin*. This class must implement abstract methods to handle the loading, install and uninstall events of a plugin – these are invoked by the plugin manager.

Plugins can also override methods to: handle web-requests, handle hooks, register URLs with the web manager (which invokes the plugin when the URLs are requested), register hooks, register templates with the template manager and much more. In the event the web manager or template manager fail, they can invoke all of the plugins to re-register URLs or templates.

4.1.8. Hooks

Registering and handling hooks is an event mechanism, where plugins subscribe to an event by a name, through the plugin manager. Other plugins can invoke an event by a name, with registered plugins invoked to handle the hook. The name can be anything a programmer desires.

Hooks can also be invoked through RMI, over the network; this is useful for threads, of plugins, which need to poll the database for work. Rather than threads polling the database periodically, this methodology allows threads to efficiently sleep until work is due. This is currently used for sending e-mails, marking assignments and cleaning session data.

4.2. Plugins

4.2.1. Assignment Marker

This plugin is responsible for marking and computing the overall marks of instances of: criteria, assignment questions and assignments; based on first-in, first-out queueing. When loaded into PALS, it creates a thread for fetching work, instance of criterias (IAC) to be marked, which are then placed into a queue. It then creates a thread pool of workers, which block on the queue, until an IAC is available. This is then processed and marked. Since each IAC belongs to a criteria, which belongs to a criteria-type, the

plugin which owns the criteria-type is invoked, through the hook mechanism, to mark the work. The instance of assignment (IA), of the IAC, is then placed into a buffer. The main thread checks the IAs in the buffer to see if all the IACs have been marked; if this is true, the overall grade for the instances of questions and IA is computed, as discussed in section 4.1.6.

If an IAC is unable to be marked, it is set to a *manual marking* state, where a lecturer has to mark it or set the IAC to be reprocessed automatically. This is a fallback method in case a plugin encounters an exception, since a student should not receive zero marks when the system is at fault. If a plugin is not available on the node, to mark the instance of criteria, a timestamp is set, with the instance of criteria not marked for a configured (plugin settings) timeout period. As mentioned, in section 4.1.6, a table is locked when retrieving instances of criteria, as a semaphore.

4.2.2. Default Auth

The default authentication manager for PALS, which uses the *pals_users* table (figure B.2) for storing login information. This is a separate plugin to allow an alternate authentication manager to be used in the future. This is also responsible for the web interfaces for logging in and out, resetting an account's password by e-mail, updating an account's settings and the management of users and user groups.

4.2.3. Miscellaneous

- **Assignments** – responsible for the web interface for manual marking, taking instances of assignments and viewing instances of assignments (as a lecturer or student). This plugin uses the question and criteria types, which point to plugins responsible for them, for rendering both questions and marking feedback for criterias – using the hook-mechanism.
- **Captcha** – used for human verification, discussed in section 4.5.
- **Default Question Criteria Handlers** – provides the default question and criteria types, for this project, which includes the web interfaces to edit and render them, as well as marking criteria types.

- **E-mail** – e-mail is persisted into a table in the database, which is sent by this plugin. This allows e-mails to be sent by a different thread during web requests, which could otherwise cause a time out. It also allows for e-mails to be re-attempted, after failing to be sent, and for other nodes to send e-mails. Persisting the data also carries it across reboots.
- **Example** – an example plugin which demonstrates how to implement 404 pages; it is also responsible for displaying web pages with static content, such as the home-page.
- **Jetty** – runs an embedded instance of the Jetty (Eclipse Foundation) web server, which avoids having to setup a web server. It also allows for PALS to be portable.
- **Modules** – responsible for the web interfaces for displaying and managing modules.
- **Node Active** – updates the database periodically to state the node is still alive. This is important for retrieving and caching a list of RMI nodes for remote invocations, such as waking up threads on other nodes.
- **Questions** – responsible for the web interfaces for modifying questions.
- **Runtime Plugin Reloader** – creates a file monitor on the plugins directory and loads/unloads plugins.
- **Session Cleaner** – removes old data from the system. This checks for expired session in the database, as well as file uploads. It also removes old data from deleted instances of assignments, which is registered in the table *pals_cleanup* (figure B.2).
- **Stats** – responsible for logging exceptions from compilation and dynamic analysis, for code questions. This also provides web interfaces for reviewing the data. Refer to section 4.4.5.

4.3. Applications

4.3.1. Java Sandbox

Student's, and possibly question or criteria, code is compiled by PALS into class-files. When the code needs to be executed, PALS launches a new JVM (Java Virtual Machine), as a separate process, of the Java Sandbox – an application for securely executing Java

within a restricted environment, often referred to as *sandboxing*. It is launched with arguments specifying: a white-list of allowed classes, the path of the class files, the fully-qualified entry-point class name and method and method parameter types – only primitives are currently supported as parameters, which also allows arrays.

A custom *SecurityManager* (Java API) implementation is first applied to the Java Virtual Machine (JVM), which is used to restrict certain features of Java, by throwing a *SecurityException* for restricted features. At present, file I/O is restricted to the same directory as the class files and all other major features, such as sockets, which could be used to communicate with the node's physical machine (a security risk), are disallowed.

A custom implementation of a *URLClassLoader* is then used to locate the entry-point class and method, which is then used to load required class files, and any classes required by the class files. If a white list, of allowed classes, has been specified, any class not listed will result in a custom exception being thrown, which PALS can then use to inform a student that their code uses a restricted class.

A thread is next launched, just before the entry-point method is invoked, which kills the sandbox after a time period, to protect against indefinite or long periods of execution. PALS also does something similar, where a marking/worker thread waits for the process to exit before a time period, or forcibly kills the process. The timeout thread, in the Java Sandbox, acts as a fail-safe in the event PALS is prematurely terminated or unable to stop the sandbox.

A problem was also experienced with the Java Sandbox, where, on occasion, no data would be written to standard output. To resolve this issue, the Java Sandbox outputs the line *javasandbox-end-of-program* and waits for a character to be written, to standard input, before terminating.

4.3.2. Windows User Tool

PALS also launches processes, such as this sandbox, under another user, for additional security, using a process wrapper called *PalsProcess* in the base library. On Windows, another application had to be created, called *WindowsUserTool*, which uses the Microsoft .NET Framework to launch a new process under new credentials, which then forwards any standard input, output and error data between PALS and the new process.

This overcame the problem of using the Windows *runas* tool, which required credentials to be manually entered and saved, as seen with CourseMarker.

The credentials may be insecurely stored, but this does not matter, since any student code will execute under a new user, which should be restricted from reading any PALS configuration files, with the Java Sandbox able to also restrict I/O. This trade-off allows for a quicker, less manual, setup of new nodes.

With Linux, the process is executed over SSH on localhost, to change the user of the process. However, the working directory could not be changed. This was overcome by having the sandbox check the working directory is the same as the class files path, or the sandbox creates an identical process of its self, but with the correct working directory.

4.3.3. Node

The *Node* is a simple application, which uses the base library to fetch, or otherwise create (singleton pattern – mentioned in section 4.1.4), an instance of a *NodeCore*. The method *start* is then invoked to start a new instance of a fully-functioning node. This means another, pre-existing, application could include the base library and, with very few lines of code and little effort, integrate with PALS.

4.4. Administration

Web pages for administration are handled between multiple plugins, meaning a future plugin could implement its own administration section(s). In this section, the individual parts of the administration system are discussed.

4.4.1. Questions

A pool of questions is shared between all the assignments of modules (4.1.6), which allows questions to be created, modified and deleted without involving any configuration files. Creating a question is as simple as specifying a title, a description (optional) and selecting the question type (section 5.1). Next, the user is presented with a page to edit the question. The user then goes to the question overview to manage the criterias, which can be combined, and weighted, to form the overall mark for the question. Adding a

new criteria is similar to creating a question, where the user enters a title, weight and selects the type of criteria (section 5.2); the user is then presented with a page to edit the criteria. Since many questions can exist in the system, the questions overview page has a filter for question title, with each question able to have a description.

4.4.2. Modules & Assignments

Modules are used to create groups of users assigned to assignments. The user is able to create, modify, delete assignments; view active assignments, view and print off overall marks for students, view attempts by students and manage enrolment.

Creating an assignment requires a title, weight and the number of maximum attempts. Maximum attempts can be -1 for unlimited retries, which also allows students to reopen an assignment after submission, with the same answers submitted. A due date can also be optionally set, with the *assignments marker* plugin auto-submitting active assignments, once surpassed. Assignments are also weighted, which is used to compute the overall module mark for an enrolled student. The user is then presented with a page to add, change the page and order of questions. This means multiple questions can be presented on a page, with multiple pages to divide up different sets of questions. Marks for an assignment can also be viewed, printed and downloaded as a comma-separated values (CSV) file; for an assignment with multiple attempts, the highest mark is used.

The user is also able to view an attempt, by a student, and modify the marks for questions, reprocess the criteria for automated marking, recompute the overall grade and reopen the attempt (with the same answers submitted).

4.4.3. Users and Groups

Users can be created, modified and deleted. Each user belongs to a user group, which each specify the permissions granted to the belonging users. These permissions include:

- Login – a group can be disabled from logging-in; useful for retaining data, of old users.
- General Marking – allows the user to perform manual marking; useful for teaching/marking assistants of a module.

- Modules, Questions, Users, System – allows access to specified elements of administration. Users should only require the specific access they require to perform their role. Example: it makes no sense allowing module organizers to modify user-accounts, since the area of attack, i.e. a lecturer's account becoming compromised, is wider.

4.4.4. Mass-Enrolment

Users can be added, enrolled, disenrolled and removed from the system and modules en masse using CSV and tab-separated files. The same formats are also supported for downloading users belonging to either the system, a user-group or a module. This is useful for sharing information between third-party information systems.

4.4.5. Stats

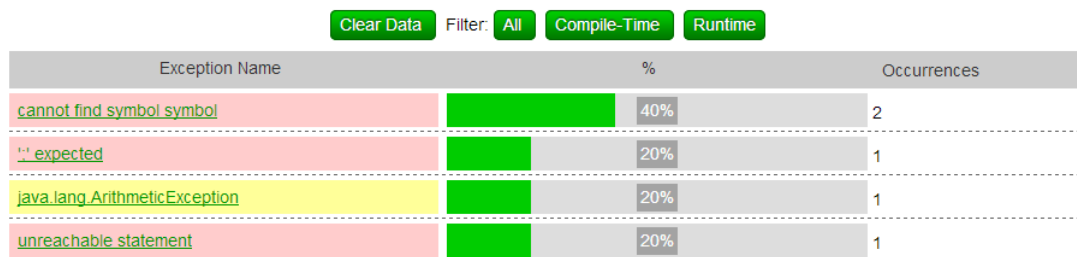


Figure 4.4: Stats Overview with Runtime Errors in Yellow and Compilation Errors in Red.

Compilation and run-time errors are collected, which is useful feedback for lecturers on areas where students may be encountering issues. Errors can be viewed by type in an overview (figure 4.4), or filtered by: question, module or assignment – by visiting the overview of the respective items. The user can also view individual exceptions, along with their messages and the instance of the assignment of where the error occurred. Hints can also be added to exceptions, which are shown as feedback after submission of an assignment (figure 4.5).

			at com.example.Test.sum(Test.java:7)	
5	262,318	580	Exception: java.lang.ArithmeticException - cause: / by zero Stack Trace: at com.example.Test.sum(Test.java:7)	0%
6	-621,-679	-1300	Exception: java.lang.ArithmeticException - cause: / by zero Stack Trace:	0%
				Sum: 0%

Check your code does not divide by zero!

No inputs produced correct output.

Figure 4.5: An Example of an Exception Hint from the Stats System.

4.4.6. System

All, and individual, nodes can be rebooted and shut down, useful for general maintenance on a cluster. The e-mail queue can also be viewed, which could serve as a useful diagnostic for a system encountering problems. Plugins can also be uninstalled, deleted and unloaded from all nodes; the same page also displays the state and information of plugins, useful for diagnostics.

4.5. Security

The system uses security guidelines set by OWASP, a *free and open software security community*. Even though it is not an academic resource, it has been praised by academia: *OWASP contributed significantly in developing a secured Web application* (Sedek et al., 2009). Since this system is handling sensitive data, such as grades, it is crucial to protect against malicious activity. PALS has protection against/for:

- **XSS**: cross-site scripting (XSS) attacks, where users inject malicious content into web-pages; this is avoided using HTML escaping for user input.
- **CSRF**: cross-site request forgery (CSRF), where requests are made without the user's knowledge. This is usually by embedding a malicious URL, to commit an action, in an image or running a script to make requests on a third-party web-page. This is avoided by asking users to confirm actions and adding a randomly-generated 32-character alpha-numeric phrase into HTML forms, which is saved on the server-side in the user's session data.

- **Session management:** each user session has an identifier, stored in a web-browser as a cookie, which is tied to an IP address. Each identifier is generated by using the SHA-256 algorithm to hash 512 pseudo-randomly generated bytes. The random number generator (RNG) seed is based on the current date and time, as well as a random number from another time-seeded RNG, created when the node starts. Since PALS is being used within a school environment, multiple users could be tied to a single IP. Therefore, hashing prevents brute forcing and determining new session identifiers. Sessions can also be set to private or public when logging in, which expire after 60 or 10 minutes of inactivity.
- **Passwords:** the default authentication plugin irreversibly hashes user passwords with SHA-512, using a unique 32-alpha-numeric salt for each password. Salting further protects against rainbow table attacks to guess passwords, in the event such data was leaked.
- **SQL injections:** all query inputs are parametrized, which are then sanitized by the connection provider/library.
- **Automated attacks:** captcha verification is used on high-value pages, such as logging-in and deleting modules. Bots can get around randomly-generated verification techniques, such as CSRF, by scraping for text data. Instead, this renders alpha-numeric text, surrounded by noise, in an image, which the user has to enter.

5. Design and Implementation – Assessment

This section is a continuation of the previous section, but with more detail regarding the assessment part of the system.

Question and criteria types, as previously mentioned, are used to form questions, which can be added to assignments. These types are handled by plugins, so they can be replaced and extended over-time. They both use UUIDs as identifiers; this is partly for plugins to interact, but also for plugins to be deleted and old data retained. Even though old data, in the event of a plugin being uninstalled, would fail to be rendered and generate error messages, it could prevent the loss of data from human error.

5.1. Question Types

Question types are responsible for capturing and displaying different types of languages. If a language was to be supported in the future, it would be supported through a new question type.

Written text was one of the first types implemented, which simply takes user input. This type is aimed at essay writing, which would be manually marked.

Multiple choice/response allows a set of possible answers to be specified, with the option to allow, or disallow, multiple answers to be selected. When the options are presented to the user, the order is picked randomly and persisted for the instance of the question. The intention is to make memorizing the order of answers, for assignments with multiple attempts, more difficult.

5.1.1. Code Java

The type *Code Java* is the primary reason for the system, which allows the assessment of Java code. Code can be provided in two different ways. The first allows for students to provide a single class, which uses a third-party library, Code Mirror, for syntax highlighting. This requires JavaScript to run on the browser, however, normal textbox capture is available as a fallback. The second allows for files to be uploaded, including ZIP archives, which are then extracted. Any previously-compiled *class* objects are ignored as a security precaution, since the compiled objects may not be from the same code.

A white list, of allowed classes, can be specified, which is used to restrict the classes allowed during dynamic analysis/execution of code. A skeleton can also be specified, which the student can use as a base for answering the question.

Questions can also be provided a set of files, which are merged into the temporary directories of instances of questions. This allows hidden code to be added to questions, which can be tested in order to perform tests of its own. It may be useful in a scenario where students write classes for a skeleton, in order to solve a problem, such as managing a collection of data-types or sorting an array. If a file already exists in the instance of a question, it is replaced with the file for the question. This prevents tampering with provided files, which also prevents accident modifications and cheating.

Compilation of code currently uses the *javax.tools* library of the Java library, which allows for compilation errors to be retrieved and used by the stats part of PALS. Compiling code in memory was initially tested, but this presented many issues. The first issue was security.

The fields *in* and *out* in the *System* class are required for printing to standard input and output. The same class also contains a method for exiting the current runtime, which could shut down a node. Therefore, this class could not be disallowed, but the exit method can be restricted by a *SecurityManager* (mentioned in section 4.3.1), but it makes shutting down a node difficult, which would require an overcomplicated mechanism.

Also, since only a single instance of a node can be created, due to a singleton pattern, student code could actually fetch an instance of the current core and, without restriction, modify the system. Another major issue was efficiency. Since multiple nodes can mark criteria, the code would have to be compiled on each node. The solution to all of these problems was to compile the code to the shared storage, with the code executed within the Java Sandbox as a separate process, isolated from the runtime of the node.

5.2. Criteria Types

The criteria types are used to mark the captured input from questions; they can also serve multiple question-types and can be in separate plugins. This means a new plugin could be created to mark a code question-type differently.

5.2.1. Matching

The first two criterias created allow for text to be matched using plain text or regular expressions, which support every question type.

Regular expressions are useful for matching phases in text responses. If a match occurs, full marks are rewarded; the inverse can also occur, where if no match occurs, full marks are also rewarded. This could also be useful for providing marks for the usage, or lack of usage, of certain control flow statements.

Another matching type was added for multiple choice/response questions, where if

the correct answers are selected, full marks are awarded.

5.2.2. Java – Code Metrics

This criteria uses the *lo*, *lotol*, *hitol* and *hi* thresholds previously mentioned (section 2), which can be applied to a literal or ratio (between 0 to 1) number. This supports the static analysis of lines of code, blank lines, comment lines and average identifier length (classes, methods or fields).

5.2.3. Java – Testing Inputs

This type performs dynamic analysis on static methods, through the Java Sandbox, which supports the passing of primitive parameters and return types. The criteria first requires the entry-point, which is the full class name and method name. Next, it requires input/parameter types, separated by commas and in the same order as the parameters of the method. Then the input is specified, with a different set of parameter values on each line. These values are separated by semi-colons for each parameter, with multiple values, for arrays, separated by commas.

Then a piece of test code is specified, which must contain the method located at the entry-point. Since this criteria is used with code questions, it attempts to fetch the entry-point method and class of student code, using reflection. If they exist, the student's code is invoked with each set of values, the test code is also invoked with the same values, with the output compared. If both outputs match, the answer is correct. The overall mark for the question is then computed based on the number of tests correct over the total number of tests.

Correct inputs

Marking Criteria Results

[100] 100%

#	Input	Output		Mark
		Expected	Your Code	
1	1,1	2	2	16.667%
2	2,2	4	4	16.667%
3	5,5	10	10	16.667%
4	-1,-1	-2	-2	16.667%
5	78,180	258	258	16.667%
6	-853,-596	-1449	-1449	16.667%
Sum:				100%

All inputs produced correct output.

Figure 5.1: Example Feedback for Testing Inputs Criteria

Once marked, the student is shown a table with the expected output, unless the solution is set to be hidden, and the output produced from their method (figure 5.1). One drawback could be that only static methods can be tested, but this is not true, since hidden code, as previously mentioned, could be invoked to create instances of objects and produce output based on the results.

5.2.4. Java – Testing Standard Input and Output

This type performs dynamic analysis in a similar way to *Testing Inputs*, where it requires an entry-point full class name and method. However, it only supports a single test and requires arguments, instead of inputs. These arguments consist of each parameter, in-order, on each line, with the data type name and value separated by an equals symbol. This also only supports primitives, including arrays.

Instead of test code, an input/output script is specified. Each line starts with either *in*, for standard input, or *out*, for standard output. This is preceded by an equals symbol, along with the text data to either be matched on standard output or to be written on standard input. The student's code is then executed and the script is followed. This is useful for exercises requiring the student to read, validate and sanitize real input data for real-world scenarios.

Marking Criteria Results			
Correct Name Construction			[100] 100%
#	Type	Line	Mark
1	Input	John	--
2	Input	Smith	--
3	Correct	John Smith	33.333%
Solution Script			
<pre>in=John in=Smith out=John Smith</pre>			

Figure 5.2: Example Feedback for Testing Standard Input and Output Criteria

Feedback at the end shows the script, unless the solution is set to be hidden, with the output of the student's program, indicating which lines are correct/incorrect (figure 5.2). Since the output is logged and the program could execute for a few seconds, an indefinite loop could produce a huge amount of garbage. This could be due to an accidental mistake, or malicious intent to cause harm to the system, by using excessive resources. Therefore, an error threshold is specified, which is the maximum number of incorrect/differing lines of output, before the program is terminated. Standard error can also be merged with standard output.

5.2.5. Java – Testing Existence

A criteria type exists for checking the existence of either a method, class or field. This can also be used to check the modifiers, with a different mark, as specified, awarded if the modifiers are incorrect.

For a class, only the full class name needs to be specified. For a method, the full class name is specified, along with the method name, parameter types and the return type – with non-primitives and primitives both supported. Fields require the full class name of where they reside, along with the type. Optionally, a generic type can be specified and checked, which is useful in exercises involving data-structures such as *ArrayList* (Java API).

5.2.6. Java – Enum Constants

Checking the existence of an enum can be achieved through checking the existence of a class, since an enum is considered a class in Java. However, it does not allow for checking the defined constants specified.

This requires the full class name of where the enum resides, along with a list of names of constants to be defined – with the options to ignore case and allow extra constants to be defined. The overall mark is computed based on the number of constants over the total number of constants required. If extra constants are not allowed, the total percent of a single correct constant is deducted until the mark is zero. Therefore, if two correct constants are defined and four are required, with two incorrect constants, the overall mark would be zero.

5.2.7. Java – Inheritance & Interfaces

A powerful feature of modern programming languages, such as Java, is interfaces and inheritance. This criteria allows for a class to be inspected, by providing a full class name. If this class exists, it can be checked to see if it extends a class and/or implements a set of interfaces. An optional field allows for an inherited class to use a generic type. This is useful in exercises where a student has to write a class to extend a binary tree node or the operations of a data-structure.

The mark computed can vary on the scenario. If only inheritance is being checked, full marks are given if the target class exists and the class is correctly extending a specified class. If only interfaces are being checked, the mark is based on the interfaces, required, implemented over the total number of interfaces required. If both the interfaces and inheritance are being checked, both are marked the same, but the overall mark is split between the two areas. Therefore, if the extended class was correct but only half of the required interfaces were implemented, the overall mark would be 75%.

5.2.8. Java – Custom Code

Hidden code, as previously mentioned, is useful for indicating certain outputs, but this feature is made even more powerful by allowing code to perform its own marking and

return a value between 0 to 100 as the mark for an instance of a criteria.

This works by the question having hidden code uploaded, which contains a method with *public static* modifiers, an integer return type for the mark and no parameters. This is then invoked to perform marking. Feedback can then be specified by outputting a line to standard output, starting with either: *error*, *warning* or *success*; to indicate the respective type of feedback message, with the message following after. Any standard output starting without either of those keywords is logged as an *info* feedback message.

As seen with previous criteria, accidents or malicious intent could use the feedback mechanism against the system. Therefore, a (feedback) message threshold is specified, which once surpassed, the program is terminated.

6. Evaluation

This section evaluates the completion of the project, a comparison to previous systems in the literature review, discusses problems encountered, throughput testing and finishes on limitations and potential for future work.

6.1. Completion

The completion of the project has been determined by looking at the original requirements, as outlined by MoSCoW in project planning (section 3.1). All of the *must have* and *should have* criterias have been fulfilled. Almost all of the *could have* criterias have been fulfilled, with the exception of checking defined methods have been invoked. The system's base library has also been unit tested, with 96 tests, and documented for future extensions. However, the code base is of a significant size, consisting of 24,883 lines of code and 209 classes, which may present maintainability issues in the future.

6.2. Previous Systems

PALS provides a system capable of both automated and manual feedback. It features static analysis, with metrics similar to those seen in Style++. And checking the existence, properties and values of structure of code, and dynamic analysis through testing

inputs, as seen with PASS. As well as standard input/output, as seen with CourseMarker. Security includes placing limits on resubmissions, as mentioned in Online Judge, but the system also features sandboxing, something not seen during the literature review. Like BOSS, and unlike CourseMarker, the RMI implementation has SSL encryption.

Compiler error collection, as seen with CourseMarker to assist lecturers, has also been improved, by also collecting dynamic exceptions and allowing custom hints to be specified as feedback. The system has also been tested as a distributed system, rather than a segmented system as with BOSS and CourseMarker, with up to eight nodes. Assessment can also be in the form of small formative exercises, as seen with SchemeRobo, and large summative assignments, with the option to place questions on different pages.

6.3. Problems Encountered

6.3.1. Java Sandbox

A modification was required to the Java Sandbox. Since the return value of an invoked method (section 4.3.1) is outputted to standard output at the end, the user could output their own value and terminate the environment. This could be used to spoof output, and even possibly the mark with the *Java custom code* criteria.

The *SecurityManager* of the Java Sandbox will now catch the exit call and output a message to indicate if the environment has been terminated, which appears as the last message, instead of a potential spoofed line. This could be a general flaw with the way PALS and the Java Sandbox communicate. Serialization, of the return type, was considered as a fix, but nothing from the sandbox environment should be loaded into the PALS environment, for security. Writing text output to a file was also considered, but this could present concurrency issues, with multiple nodes, and such file writes could potentially be spoofed too.

6.3.2. Assignment Marker

When testing a cluster of eight machines, the throughput of work marked became as low as five per second, with the database becoming incredibly slow and eventually non-responsive. A single node could mark around 10-15 instance of criteria (IAC) per sec-

ond. The problem was due to the way worked was being fetched. Each node would have eight worker threads, which would each lock a table to fetch and mark work, as well as compute the overall mark of assignments and submit any due assignments. With eight nodes, this means a total of 64 threads were created and locking a table. Each thread would only pull a single criteria to be marked, resulting in a lot of overhead.

Something called the *fetch-rate* has been introduced, which is the amount of work placed into a shared pool, with each worker thread blocking until an IAC is available in the pool, which is then processed. A main thread was then introduced to fetch the work, compute the overall mark of assignments and handle due assignments. This resulted in a throughput as high as a few hundred IACs being marked per second, with only one thread locking a table per a node and 16 IACs being fetched for 8 threads – a significant improvement.

6.4. Throughput Testing

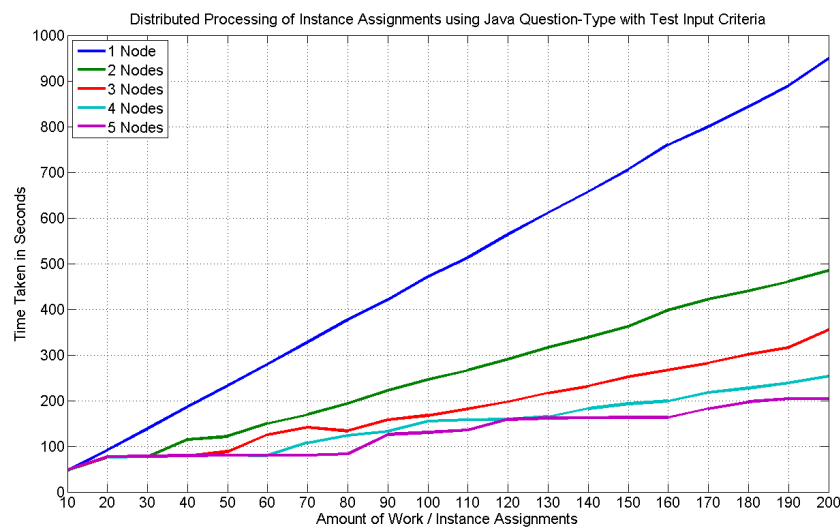


Figure 6.1: Graph showing distributed processing of Instance Assignments.

To test the performance of multiple nodes, a cluster of five nodes was created. Each node ran, within an Arch Linux virtual machine, with a dedicated CPU. A database and file-share was setup on the host machine. The throughput of instance assignments (IA) was

then tested, which contained the same Java question-type and *test inputs* criteria, with the same answer for each instance. The *pals_node* table was locked, to prevent nodes from fetching work, whilst test data was inserted. The table was then unlocked, with the IA table polled every five milliseconds to check if all the work had been marked.

The results (figure 6.1) show a significant decrease in the time taken to process IAs, as the number of nodes increase. The largest amount of work, 200 assignments, falls from 15.83 to 3.41 minutes by using five nodes, instead of one – a 78.48% decrease in time. However, the overall performance increase from each new node decreases. This is most likely due to the overhead from each node locking the table to fetch work – a problem mentioned earlier, when each thread was locking the table.

6.5. Limitations & Future Work

6.5.1. Method Invoked

The testing of a method being invoked was not completable due to time constraints. This could be first solved by generating a tree to determine classes referencing the method. Each class in the tree would then have to trace to the entry-point of the program. But this would still not guarantee execution, since an invocation inside of a conditional statement may never have the condition satisfied. Therefore, guaranteed scenarios could be tested to solve this issue. However, a new issue arises where the method may be invoked, but the code does not do the intended feature.

This issue is partially addressed by PALS with the *Testing Inputs* criteria, which is testing the functional code-correctness. But even when a method is producing correct output for a given set of inputs, the code may still not meet the correct specifications. An example may be sorting an array and making a call to a utility library, which would produce the correct output. This has, again, been addressed with white-listing, but the student could still solve a solution with e.g. poor complexity.

6.5.2. Multiple Programming Languages

One of the major drawbacks of the system is that only one programming language is currently supported. Due to the extensibility of the plugin architecture, additional pro-

programming languages can be added in the future. PALS could even be extended to support non-programming assessments, which is already the case with written-response and multiple choice/response question types (MCQ/MRQ).

6.5.3. Assignment Marker

The current algorithm used for fetching work, by the *Assignment Marker* plugin, is very primitive and could be improved. Some criteria are very lightweight and take very little time to mark, such as testing for the existence of a field. Criteria, such as testing for inputs, can, depending on the number of tests, take a long period of time. Therefore, criteria types could compute some form of weight, based on e.g. the number of tests, with each node configured to collect as many instances of criteria to mark, up to a certain weight threshold.

Programs may also only work on certain platforms or environment. Therefore, nodes could be put into groups, with the ability to set the group used to mark a criteria.

6.5.4. Resits

A real world issue, overlooked in the initial design, was the resitting of assignments, since a student may be delayed from taking an assignment. This could be addressed by placing students within groups for when the assignment is available, with the option to add all students to an initial group, with students resitting an assignment added to a different group, created in the future.

6.5.5. Marking

There may be scenarios where questions may be mandatory for a page/section, zero marks are awarded where one or more instances of criteria receive zero marks for a question or a student fails a module in the event they fail an assignment. Therefore, marking could be improved by allowing custom code, or providing more options, to compute the overall mark for assignments and questions.

6.5.6. Randomised Snippets

As seen with QuizPACK, randomised snippets of code could be generated, with questions regarding the structure and flow of execution. This was not in the original requirements due to time constraints.

7. Conclusion

This project set out to produce a new third-generation programming assessment system for programming languages, which also allows for self-guided learning through automated feedback, to increase objective marking and to reduce the workload of lecturers, allowing for more exercises to be created, and greater experience for students.

PALS successfully fulfills this requirement, and allows for the assessment of written-response, MCQ/MRQ and Java programming. Due to the open source licensing and plugin architecture, further programming languages can be supported, as well as other forms of assessment for areas such as mathematics.

Also, if student numbers are increasing, the system is scalable, with the option to add and remove nodes over time. Shared storage can also be used with plugins, with the ability for updates to be sent to all the nodes within a few seconds, since nodes can automatically detect changes and reload plugins during runtime.

This project first started with a literature review, which was very important, since it provided many existing areas to build upon for the design phase, which followed after. Due to the unique architecture of PALS, it was critical to test the design ideas to avoid problems later during the project, which would waste time. Initially this used basic test classes, which evolved in the implementation phase, to using continuous integration and automated unit testing of the base.

Even if this system is not adopted by many, it may contribute new ideas and help form the foundations of a new assessment system. Overall this project has been successful and achieved its original aims, delivering a new system and a completed report, on time.

References

- Ala-Mutka, K., Uimonen, T., and JÄärvinen, H.-M. (2004). Supporting students in c++ programming courses with automatic program style assessment. *Journal of Information Technology Education*, 3.
- Benford, S. D., Burke, E. K., Foxley, E., and Higgins, C. A. (1995). The ceilidh system for the automatic grading of students on programming courses. *ACM-SE 33 Proceedings of the 33rd annual on Southeast regional conference*, pages 176–182.
- Blumenstein, M., Green, S., Nguyen, A., and Muthukkumarasamy, V. (2004). Game: a generic automated marking environment for programming assessment. *Proceedings of the International Conference on Information Technology: Coding and Computing*, pages 212–216.
- Brusilovsky, P. and Sosnovsky, S. (2005). Individualized exercises for self-assessment of programming knowledge: An evaluation of quizpack. *ACM Journal on Educational Resources in Computing*, 5(3).
- Cheang, B., Kurnia, A., Lim, A., and Oon, W.-C. (2003). Automated grading of programming assignments. *Computers & Education*, 41(2):121–131.
- Douce, C., Livingstone, D., and Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3).
- Higgins, C., Gray, G., Symeonidis, P., and Tsintsifas, A. (2005). Automated assessment and experiences of teaching programming. *Journal on Educational Resources in Computing*, 5(3).
- Higgins, C., Hegazy, T., Symeonidis, P., and Tsintsifas, A. (2003). The coursemarker cba system: Improvements over ceilidh. *Education and Information Technologies*, 8(3):287–304.
- Hollingsworth, J. (1960). Automatic graders for programming classes. *Communications of the ACM*, 3(10):528–529.

- Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. (2010a). Review of recent systems for automatic assessment of programming assignments. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 86–93.
- Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. (2010b). Review of recent systems for automatic assessment of programming assignments. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 86–93.
- Jackson, D. and Usher, M. (1997). Grading student programs using assyst. *SIGCSE '97 Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, 29(1):335–339.
- Joy, M., Griffiths, N., and Boyatt, R. (2005). The boss online submission and assessment system. *Journal on Educational Resources in Computing*, 5(3).
- McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering Journal*, SE2(4):308–320.
- Rees, M. (1982). Automatic assessment aids for pascal programs. *ACM SIGPLAN Notices*, 17(10):33–42.
- Saikkonen, R., Malmi, L., and Korhonen, A. (2001). Fully automatic assessment of programming exercises. *ITiCSE '01 Proceedings of the 6th annual conference on Innovation and technology in computer science education*, 33(3):113–136.
- Sedek, K. A., Osman, N., Osman, M. N., and Jusoff, H. K. (2009). Developing a secure web application using owasp guidelines. *CCSECIS – Computer and Information Science*, pages 137–143.
- Shepperd, M. (1988). A critique of cyclomatic complexity as a software metric. *Software Engineering*, 3(2):30–36.
- Symeonidis, P. (2006). Automated assessment of java programming coursework for computer science education. *University of Nottingham*.

Thorburn, G. and Rowe, G. (1997). Pass: An automated system for program assessment.
Computers & Education, 29(4):195–206.

A. Organisation of the accompanying submission disc

The following appendices can be found on the disc submitted with this report, which also contains code.

Code This can be found in the directory */Codebase*.

Documentation JavaDoc from the code-base can be found in the directory */JavaDoc*; each sub-directory is the name of each corresponding directory in */Codebase*. The class diagram can be found in */Project Planning*.

GANTT Chart and To-Do Items The GANTT chart and to-do items can be found in the directory */Project Planning*.

Usecase Diagrams and Descriptions These can be found in the directory */Project Planning/Usecase*.

B. Entity Relationship Diagram

Color	Meaning	Color	Meaning
	User System		Modules
	Questions		Assignments
	Statistics		Miscellaneous

Figure B.1: key for figure B.2

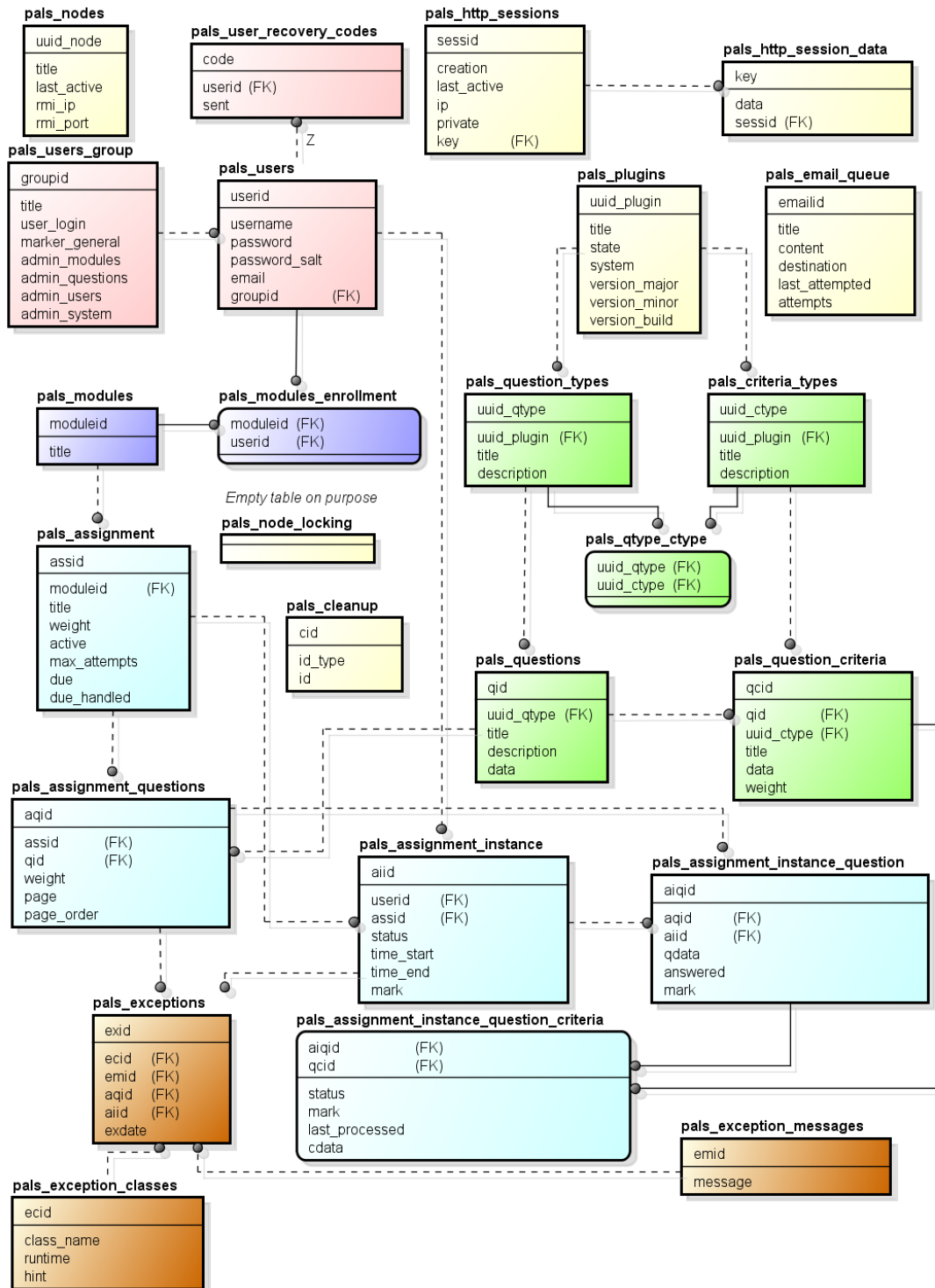


Figure B.2: The Entity Relationship Diagram for PALS.