SD&D Main Exam

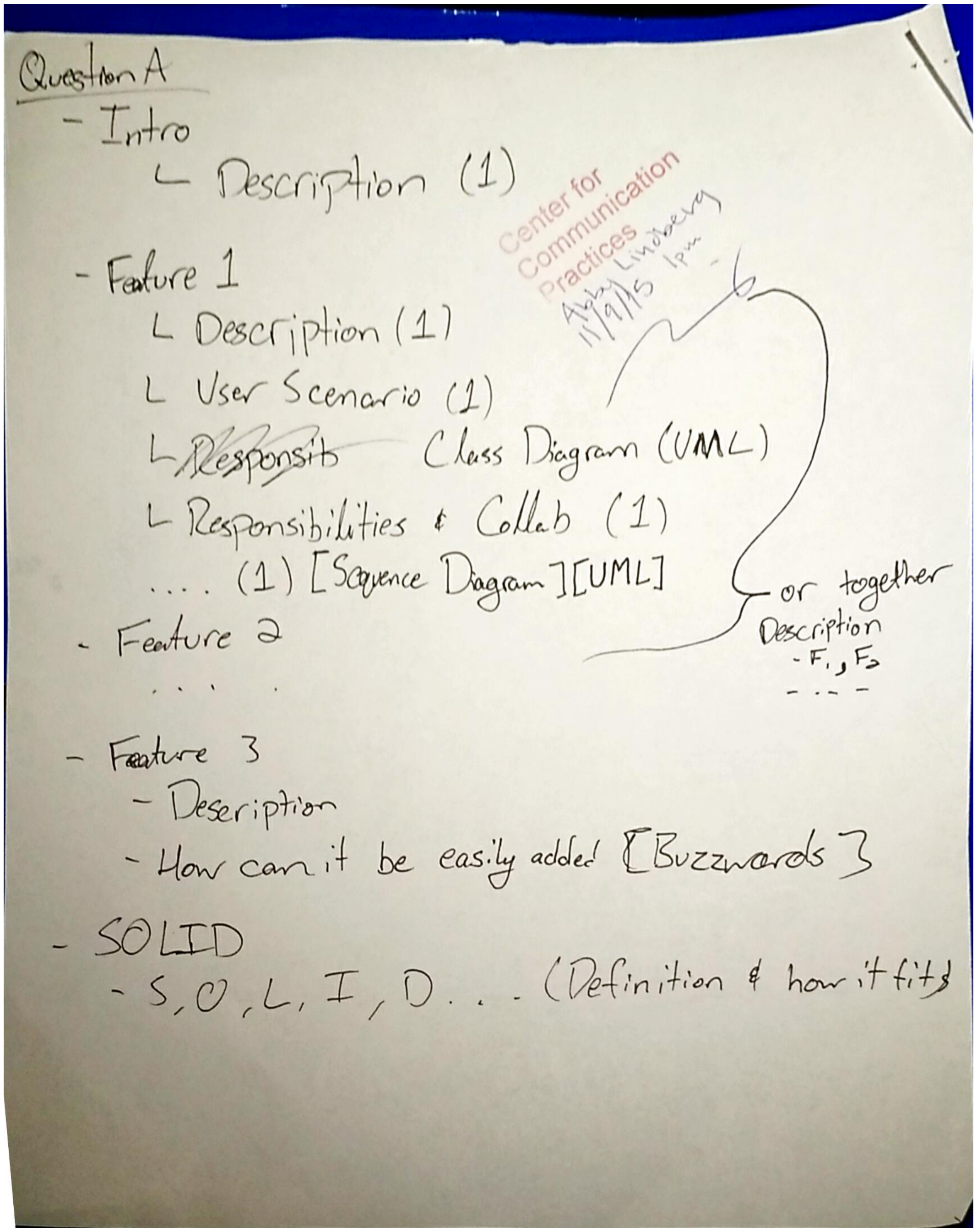
Ryan Delaney

November 2015

Project: Jadelint
`https://github.com/rrdelaney/jadelint`

This document is my own work: [illegible]

Proof of CCP Meeting



Question A

- Intro
  - └ Description (1)
- Feature 1
  - └ Description (1)
  - └ User Scenario (1)
  - └ ~~Responsib~~ Class Diagram (UML)
  - └ Responsibilities & Collab (1)
  - .... (1) [Sequence Diagram][UML]
- Feature 2
  - . . . .
- Feature 3
  - Description
  - How can it be easily added [Buzzwords]
- SOLID
  - S, O, L, I, D . . . (Definition & how it fits)

or together
Description
- F₁, F₂
- . . . -

Center for Communication Practices
Abby Lindberg
11/9/15 1pm

Contents

1 Question A

1.1 Jadelint

The project I chose to add features to is Jadelint. Jadelint is a program that statically analyzes, or lints, an HTML document written in the Jade templating language. Jadelint checks for both HTML5 and Jade best practices, and reports to the user what is wrong and where it's wrong. It's primary use is as a command line tool, but it had an easy to use API allowing for developers to incorporate it into their programs or plugins.

1.2 Feature 1

The first feature I want to add is **remote linting of files**. This will allow the user of Jadelint to provide a URL, and jadelint should look at the source at the specified URL, lint the source, and report all errors.

1.2.1 User Scenario

Guy Developer is a web developer that works on the front-end of applications. He works for a hot, young start-up centered in San Francisco, Guyify, that creates marketing websites for some of the top retailers in the area. Guy is not a time waster, and likes to make the most out of each moment. Despite this, he also strives for the best product he can make in a reasonable amount of time. For this reason, Guy likes to use the Jade templating language to spend less time writing HTML, and more time designing web pages and doing what he needs to do. Guy doesn't have as much time as he would like to thoroughly check his code every day, so Guy uses Jadelint to make sure his code is the best it can be.

Guy's co-workers aren't the most familiar with Jade though, and not with Jadelint! They often commit code to their repositories that isn't always up to snuff. There's often syntax errors, bad code, and HTML2 practices included in their templates.

Guy would like to be able to check his co-workers' code without cloning or downloading their entire repository. He wants to see the results on the command line along in the standard format.

To do this, Guy opens his console and begins by entering the command `jadelint` as he normally would. Instead of pointing jadelint to a local file, he uses the URL `http://github.com/GuyDeveloper/CodeProject/blob/master/myFile.jade`. Guy is then given the file's lint report right on the command line. Guy proceeds to do this with other files that the other team has written.

1.2.2 Responsibilities

The design adds two new components, a RemoteFile class and the DownloadManager singleton. The RemoteFile class is responsible for creating a compatible interface between a Javascript stream and the File interface, and dispatching a download request to the DownloadManager. The RemoteFile's stream is used extensively by the Linter class, so it can see no change between the RemoteFile class and File class.

The DownloadManager is only currently used by the RemoteFile class, but by having low coupling and using the Singleton pattern, it can easily be used by other parts of Jadelint. It is responsible for using the Node.js API to create a stream that can be passed to the RemoteFile class and then sending downloaded chunks to stream object.

1.2.3 Diagrams

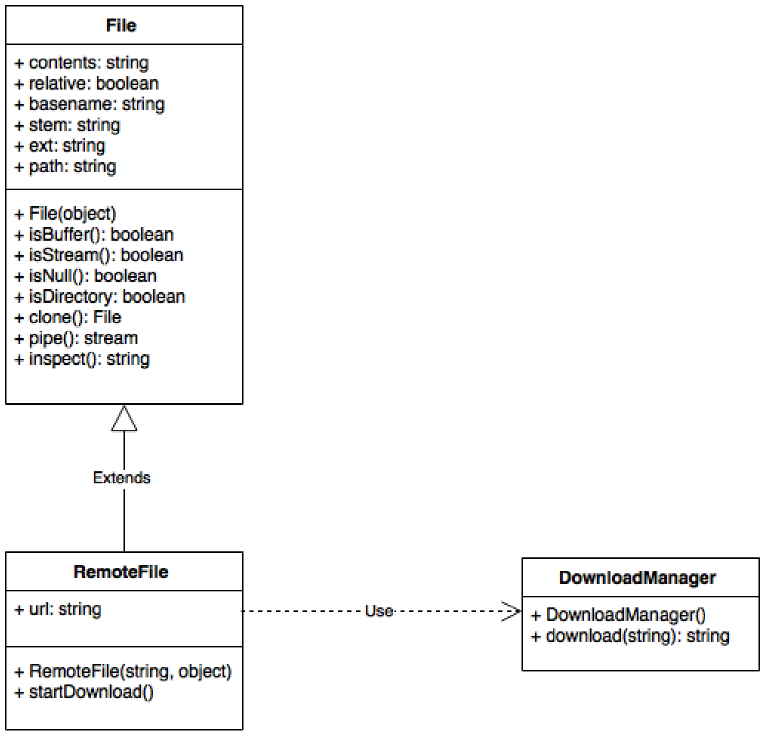


Figure 1: UML Diagram for Feature 1 [4]

The RemoteFile class implements a File interface provided by Jadelint. It uses a Download-Manager singleton to download all files, and accepts a streaming response from it to provide the file contents.

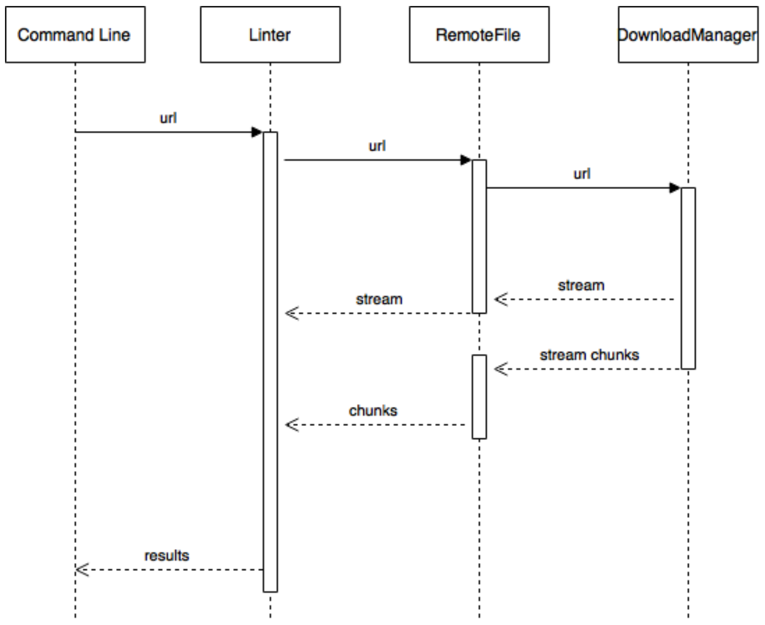


Figure 2: Sequence Diagram for Feature 1

The command-line application asks for the results from a given URL, which passes the URL directly to the Linter object. The Linter then creates a RemoteFile object, and begins the download. The RemoteFile requests the contents from the DownloadManager, which returns a stream object. The RemoteFile, which implements File, is then used in the Linter. The DownloadManager streams the file objects as they come in from the internet. When the file is complete, the Linter processes the contents and returns the results from the command line. Internally, the DownloadManager uses the Node.js API to download the file.

1.2.4 Design Choices

To incorporate remote files, with as little coupling as possible, it's best to leave alone the core linting aspects and touch just the File interface. The File interface in Jadelint follows the file interface also found in the Vinyl file project, which made it easy to adapt. By using the streaming interface rather than the buffer, a remote file didn't need a redesign to use callbacks.

The RemoteFile class uses a singleton DownloadManager to download files and produce a stream for the File interface. The DownloadManager abstracts away from the File how to request contents from the internet, as well as offer its services to other features in Jadelint. This reduces any coupling between the File interface and the DownloadManager.

The DownloadManager encapsulates any internet related data, and hides it from the class using it. This way a service or class can request a file, and have a stream given to them. Because streams are native in Javascript, this is essentially the same as returning a string. Encapsulating this information is good because changing something in the transport system (i.e. HTTP to HTTPS) becomes the problem of the DownloadManager, rather than the classes that use it. The same goes for new features to reduce latency in the network by batching requests.

By not abstracting away the requesting to the DownloadManager, there is a high cohesion between it and the RemoteFile. The RemoteFile should be able to request when it needs contents, and putting a layer between it and the DowwnloadManager would separate two close components.

This design enforces low coupling, encourages high cohesion, and encapsulates data between components.

1.2.5 SOLID Principles

The five pieces of S.O.L.I.D. are principles that relate to object-oriented software design.

The **Single Responsibility Principle** states that a class should have only one responsibility in regards to the entire program [1]. It can also be said as a component or class should only have one reason to change the state of the program. By following this principle, the developer reduces coupling between components. For example in Jadelint, if the Reporter needs to only report the results of the analysis, and prints the results in a static way, only changing state for the results. However, if the Reporter needed to also change the styling of the reports or the location, it is another reason to change and that reason, or responsibility, should be placed in another component or class.

The design to implement this feature sticks to the Single Responsibiltiy Principle well. Upon creation, the RemoteFile starts a download, the only time it will cause a change in the program. The download is handed off to the DownloadManager, which interacts with the Node.js API to return a stream object. This initial kick is the only change that the DownloadManager initiates. The File interface encapsulates all of this functionality to expose only the contents to the Linter.

The second of the SOLID principles is the **Open/Closed Principle**. This states that a class or component should be "...open for extension, but closed for modification". To follow this principle, a class should be able to have its behavior easily modified without changing the class's source code [1]. One way which to do this is by using inheritance in an object-oriented design. This allows a sub-class to act the same way under certain conditions, but implement different behavior for others. The benefits to this principles come from the ease of testing subclasses, rather than constantly re-writing both the main class as well as its tests.

The design closely sticks to this pattern because it add new behaviors to an existing class by extending it in certain situations, this case when a URL is specified instead of a local file. The Jadelint project does receive the mentioned benefits, because it has the ability to switch between the use of File and RemoteFile during its unit tests.

The **Liskov Substitution Principle** states that a subtype S of type T should be able to be swapped in without altering the correctness of the program [1]. This means that using S in place of T should not alter the intent of the program. This goes further than the formal definition of types by modifying the contract that subtypes need to keep. One key part of this is that no new exceptions should be thrown by S. This is strictly enforced in languages like Java, but developers must be careful with languages such as Javascript and Scala. By sticking to this principle, developers can design code without needing to think of careless internal uses of the API.

The design of the feature introduced adheres to this principle because RemoteFile serves the same purpose as File. They deliver the same intent, which is to lint a file, but RemoteFile allows it to be specified by a URL rather than a file location. The benefits are also aparent in Jadelint, as the Linter class doesn't need to worry if it is given a File or RemoteFile.

Another principle of SOLID is the **Interface Segregation Principle**. This principle states that an component or class should only touch what it needs to use [1]. By following this principle a component won't break if something unrelated to it breaks. The encourages and enforces low coupling between classes, while keeping cohesion tight. The benefits from doing this are having a system that can be easily modified and debugged due to isolation of components.

The feature added follows this because it offers a simple interface to the Linter, and only depending on the DownloadManager. The DownloadManager itself is an isolated component that only uses the Node.js API. Because the feature follows this principle, it is easy to test both the RemoteFile class and the DownloadManager singleton in both unit tests and integration tests.

The final principle of SOLID if the **Dependency Inversion Principle**. This principle states that "High-level modules should not depend on low-level modules. Both should depend on abstractions." and "Abstractions should not depend on details. Details should depend on abstractions." [1]. The idea behind this principle is that high-level components should interface with an abstraction from the low-level module, rather than the low-level itself. For example, if there was a component that used three functions to create a file, those three functions should be abstracted away behind an interface that exposes a single function. This reduces coupling throughout the entire application.

The RemoteFile class follows this principle because it doesn't call the Node.js network API itself. Instead that functionality is given to an abstraction away, the DownloadManager. The DownloadManager abstracts the low-level away from RemoteFile, which reduced coupling and encourages re-usability among components.

1.3 Feature 2

The second feature I want to implement in Jadelint is a **performance monitor** for the compilation and rendering of files. This will allow the user to find hot-spots and performance issues on template compile and render times. This is useful, because Jade is often used on the client-side.

1.3.1 User Scenario

Jose Developer, Guy's husband, is a template designer at Guyify. Jose doesn't code, but has tons of experience in design tools such as Photoshop, Lightroom, and GIMP. He obsesses over fonts and colors, and is a perfectionist. Unlike Guy, Jose will take his sweet time on projects to create the perfect design. To create templates for the front-end, Jose likes to use Jade.

Because Jose is so bad at coding, his templates' compile time affect the front-end performance a lot. Guy is not a fan of slow load-times, and will often yell at Jose, and bring even bring it into the bedroom. The constant tension between them at home due to work is really getting to Jose, and making him reconsider his marriage to Guy.

Jose needs to be able to profile the compilation of his templates and have the results reported in an easy to read manner. The results should show exactly where the hot-spots are in the compilation and render process, as well as show the severity of the slow down.

To do this, Jose opens up his terminal and changes the directory to his project. He lints and profiles his template, joseMain.jade, by running the command `jadelint -p joseMain.jade`. After the program runs, he is presented with the results in a clear table format detailing the bad spots of his template. Jose asks Guy to help refactor, and their marriage is saved.

1.3.2 Responsibilities

This feature will introduce four new classes. They are

1. Abstract Profiler
2. DOMProfiler
3. CodeProfiler
4. Timer

The Profiler class acts as a generic interface for different profilers. It adds a generic constructor as well as a timer to its subclasses. It interfaces with the Timer class directly, and is the interface provided to the Linter class. By giving the Linter a generic Profiler interface, it allows the program to specify behavior at run-time.

The DOMProfiler class implements the abstract Profiler. Its job is to measure how fast a provided JadeNode will render to the DOM. It is responsible for starting and ending its Timer as well as returning the results when it is finished profiling. It is useful to optionally enable this class because not all Jade is meant to be rendered to the DOM. It is used by the Linter class as a Profiler. The rendering to the DOM is handled by an external library.

The CodeProfiler class implements the abstract Profiler. It measures how quickly the Jade itself compiles. The CodeProfiler starts and ends a Timer to profile code, and then returns the results. It is used by the Linter class as a Profiler. The rendering of the Jade is done by the Jade compiler.

The Timer class abstracts away the platform's native timer from a Profiler. The responsibilities of this class are using the Node.js process API to accurately start and stop a timer for profiling purposes. This is used exclusively by Profilers, but is not meant to be exclusively used by them.

1.3.3 Diagrams

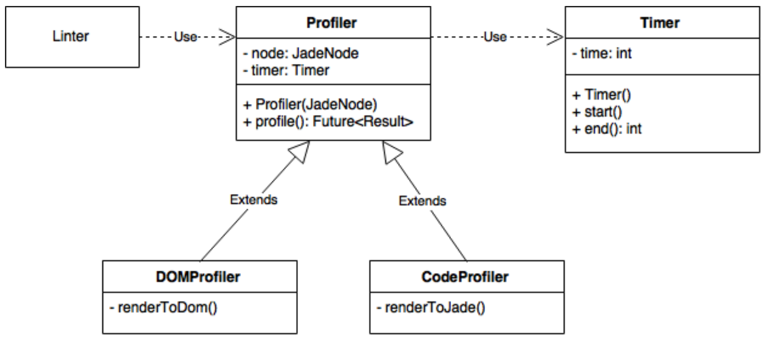


Figure 3: UML Diagram for Feature 2 [4]

The Linter class interfaces with multiple Profilers, depending on what the user wants profiled. The two options are the DOMProfiler and the CodeProfiler. Both can be used at the same time. Both of the Profilers use the Timer class to time their operations.

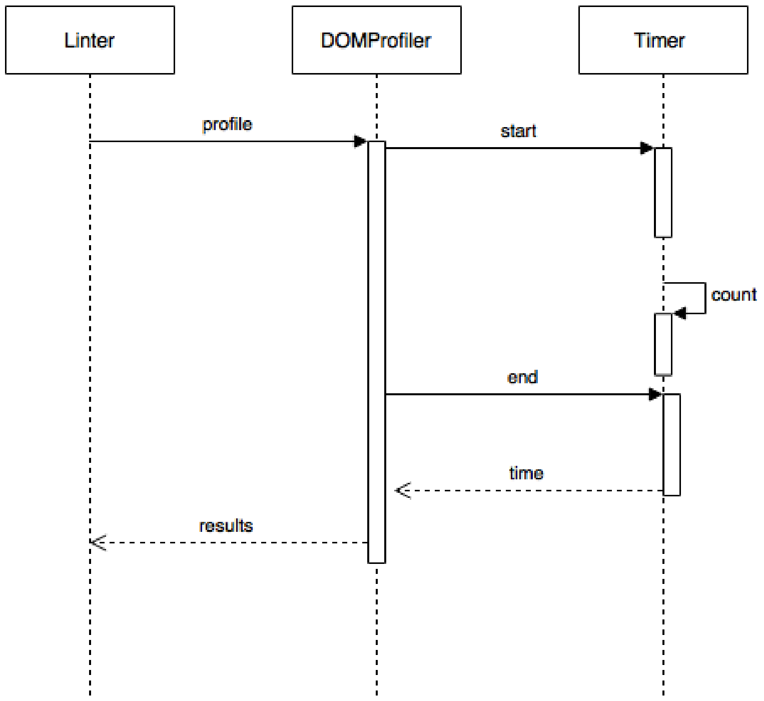


Figure 4: Sequence Diagram for Feature 2

1.3.4 Design Choices

For the most possible extensibility, a new type of profiling should be easy to add, and not require any modification to the Linter class. To properly encapsulate data, profilers should be independent of the native API, and should only worry about profiling a template.

By using the abstract Profiler class and the Timer class, both of these goals are met. The Linter becomes unaware of how, or what the Profiler is doing, which reduces coupling between the two classes. In addition, the Profiler encapsulates all data from the Linter, and only returns the results. This is good because the Linter can refer to any future profiler with the same abstraction.

The Timer class reduces the coupling between the Profiler sub-classes and the native Node.js API for timing. By having the Profiler only interact with the Timer, it is a much easier system to debug and add features to. For example, if the Node.js timer was always off by 500ms, it is easily corrected in the singular Timer class, rather than adjusting all of the sub-classes of profiler.

1.3.5 SOLID Principles

The design chosen for the profiling feature closely follows SOLID principles. The Single Responsibility Principle is show by the division of the Profiler class and the Timer class. The Profiler sub-classes should only change the program upon profiling a JadeNode and returning the results. If either of the Profilers implemented in this feature didn't use external libraries to test the code, it would be best to implement a render method in separate classes. The Timer class's sole responsibility is to interface with the Node.js API to keep time for the Profilers. It only changes state upon request from it.

The Open/Closed Principle is observed in the design with the introduction of the abstract Profiler class. By exposing a simple interface to the Linter class, it is independent of how the Profiler works and what it profiles. This means that new functionality and behaviors can be added by further sub-classing the Profiler class without modifying the original Linter class.

The design observes the Liskov Substitution Principle in the Profiler class. The Linter class can make use of any Profiler given to it, all using the same interface. Both of the Profilers detailed in this design give way to that by semantically doing the same thing, which is timing how long a piece of code takes to perform a certain task. If any new Profilers were introduced, they should also follow those semantics and not produce any unwanted side-effects.

The Interface Segregation Principle is shown in the feature design by the separation of the Timer and Linter classes. Although they are related, the two do not touch at all, and interfacing with the Timer is left to a Profiler. This means that if, for example, the Timer class breaks, the Linter won't have any direct effect from it.

The Timer class shows the Dependency Inversion Principle. By using the Timer class to interface with the low level timing API in Node.js, the Profiler is forced to rely on an abstraction, rather then the low-level component itself.

1.4 Feature 3

The third feature that would be useful to Jadelint is live profiling on an existing webpage. This would involve loading the webpage into memory, executing the javascript and look for any Jade that is rendered. This fits in well to the existing design because it extends several components from Feature 1 and Feature 2.

In Feature 1 a DownloadManager class was introduced to manage downloading all files needed by Jadelint. Because DownloadManager is re-usable, no modification to existing classes in Feature 1 is needed to implement the new feature.

Feature 2 introduced an abstract Profiler class which is used by the Linter to profile different Jade features. By extending the Profiler class, the new RemoteProfiler can interact with a RemoteFile and monitor the page performance. The Linter class would not need modification because it does not involve itself with the inner workings of the Profiler. Becasue the RemoteProfiler subclasses Profiler with the same intentions, it can also seamlessly use the Timer class to measure performance.

1.5 Ease of Change

The first question in the Ease of Change test is **How many classes did you have to add?**. For the combination of Feature 1 and Feature 2, a total of six new classes were added. Although adding less classes is often better, the design chosen follows the SOLID principles best.

The next question is **How many classes did you have to change?**. To implement these features, only the Linter had to be changed! The Linter class needed to be able to call a Profiler, so some extra logic was needed.

The third question is **How many classes would you need to change to store extra information?**. For this design you only need to add a property to the class you are adding information to.

The last question is **How many classes would you need to change to add an extra property?**. Because the entire design is based around SOLID principles and the idea of low coupling, the only class that would need modification is the one you are adding a property to.

2 Question B

To implement these features, a team of four developers will work over the course of a month to write the necessary code and documentation. The team will use an Agile methodology, specifically Scrum [2].

The team will be further divided into two groups, the “Feature 1 Team” and the “Feature 2 Team”. Each will consist of two members. Each team is responsible for the feature assigned to them. This doesn’t mean that people from one team can’t work with another, that is encouraged. It means that a team will be *responsible* for deliverables for that feature.

Each sprint will be a week long in length. Sprints will begin on Mondays and end on Fridays. This gives the team weekends off to relax and enjoy life with their families or cats.

2.1 Team Activities

The team will participate in a daily standup meeting where they will go over the work they have done the previous day. These standups will happen at 10:30am EST. The purpose of the standups will to keep everyone informed for the entire development process.

The team will also participate in a weekly retrospective, where the team will discuss what worked for them and what didn't, and how to improve upon it the next week. This meeting will allow to team to see its needs and self-correct if they are going off-path.

Following the retrospective, the team will host sprint planning where they decide what will be included in the next sprint so they can meet their deliverables on time. Although it is hard to plan for certain things, the sprint planning will give teams a goal to work to in the next week.

On the third Friday, there will be a team Super Smash Bros. tournament, to let the team relieve stress accumulated from the previous weeks. Attendance isn't mandatory, but is highly recommended.

2.2 Artifacts

The team will use a Scrum Board to organize all of their work throughout the sprints. The chart is an integral part of Scrum [2], and helps the team visualize where they are, where they were, and where they're going. It also helps someone in a management role evaluate the progress of the team over the entire process. The Scrum Board will be manged by JIRA.

Each day the team will produce a burn-down chart detailing their progress in the current sprint. Just as the Scrum Board allows the team to see the bigger picture, the burndown chart will provide insights into how the team operates on a more day to day basis. The burndown chart will be managed by JIRA.

Both of the artifacts will be useful in both the sprint retrospective as well as the sprint planning. By observing what they have done, they can more accurately predict what they can do.

2.3 Communication Tools

The team will use several Altassian tools for communication. As previously mentioned, project management will be done in JIRA. JIRA was chosen because it provides the best industry support for Scrum development environments. In addition, it also provides many hooks to other Altassian products.

For the best integration, the team will manage all code and other assets through BitBucket. Although similar to GitHub, BitBucket allows for tighter integration with other products, such as JIRA. BitBucket shares a very similar feature set with GithHub, and can do many of the same things, but it primarily acts as a central place to manage Git and Mercurial code repositories.

HipChat will be used as an instant messaging client for the team. Although Slack has gained traction over the fast few months, HipChat still has better integrations with the Altassian world of products. Similar to the BitBucket vs. GitHub situation, compared to Slack HipChat has a practically identical feature set. The benefits of choosing it though are very good support within the tooling ecosystem, although some features are lacking, such as the HipChat mobile apps.

2.4 Schedule

Week	**Team**	**Goal**	**Deliverable**
1	Feature 1 & Feature 2	Unit Tests Written	Outline of Testing Methods
2	Feature 1	Tested and Documented DownloadManager	Test Report on DownloadManager
2	Feature 2	Tested and Documented Timer	Test Report on Timer
3	Feature 1	Tested and Documented RemoteFile	Test Report on RemoteFile
3	Feature 2	Tested and Documented DOMProfiler & CodeProfiler	Test Report on both
4	Feature 1	Integrate Linter and RemoteFile	100% test coverage and documentation
4	Feature 2	Integrate Profiler and RemoteFile	100% test coverage and documentation

Table 1: Schedule for Team

This schedule works with the two teams created of two people. Each week has a goal and a given deliverable related to each goal. At the end, each team will be responsible for 100% complete unit tests and total code documentation.

3 Question C

There are five crucial misconceptions about development as described by Brooks in *The Mythical Man-Month* [3]. These misconceptions slow development down to a halt. Using Agile methodologies, specifically Scrum [2] can alleviate many of these pains, and help speed up development.

3.1 First Misconception

The first misconception of development is that everything will go well. This arises from the fact that programmers are optimists. The optimistic point of view often comes from the fact that developers have no physical limitations, what they create is purely theirs, and developers don't doubt their ideas. This is a cancerous thought process, because often when our ideas are wrong we blame what we know to be correct, and create bugs.

Being optimistic isn't the worst thing in the world, but it can lead to overworked developers. When optimistic programmers enthusiastically tell their manager that they can take on N tasks for the week, but only complete $N/10$, it forces the developer to take bad measures to ensure the software is out the door by the deadline. Fast, reckless programming creates endless bugs and only piles on to the technical debt of the team and project.

Scrum attempts to fix this by placing realistic limits on what each developer can do in a sprint, by the mechanism of a Story Point limit. Each User Story, or task, is assigned a point value with increasing weight with difficulty. When these are weighted realistically (see the fourth misconception), the Point Limit sets realistic expectations for the team. If the point limit is hit, more tasks are taken and the team adjusts by increasing their point limit for the next sprint.

Scrum has a negative effect if the tasks aren't properly weighted. By assuming that a low weighted task will be easier when the team underestimated it, the schedule for the sprint will be off from the gates, and it can be hard to recover from that.

3.2 Second Misconception

The second misconception about software development is the title of Brooks' piece, the Mythical Man-Month. The idea of the Man-Month is that what one man can do in 10 months, 10 men can do in one month. This idea holds true for certain industries and jobs, such as farming or manufacturing, but it crumbles in the software world.

This boils down to the fact that problem solving isn't a brute force approach, and neither is good programming. By forcing developers to speed through work or having uncoordinated efforts, the result will be exactly what you expect from such conditions.

Developing takes time and skill, but this doesn't mean more people don't help. In fact, having others available as a resource is one of the best things about working in a team, and solo developers miss out on that. Teamwork is beneficial to the team, however cramming developers on the same problem in hopes of brute-forcing a problem will never work.

3.3 Third Misconception

The time required for systems testing is the third misconception about programming. Due to the nature of testing, it must be left to the end of the development cycle to be fully implemented in a traditional sense (how can you test something that hasn't been built?). Due to this and the fact that programmers are optimistic create a lack of planning for testing when things go wrong. Unfortunately, things often go wrong.

Not leaving enough time to test the system at the end can have drastic side effects. If a manager promised a deadline for the software and a large bug pop ups at the end halting the release, it creates financial problems on top of the development issues that the team is having. These factors create a hugely stressful environment to work in, meaning the bugs aren't fixed as quickly. Add the second misconception in, the mythical man-month, and then development slows to a halt as the team grows larger and stress builds.

Agile development offers a way out of this called Test-Driven Development where the systems tests are part of the development process. The process is:

1. Write a failing unit test
2. Write code to make the unit test pass

It's really that simple! By ensuring your tests are always accompanied by code instead of the other way around, developers are guaranteed a certain safety whenever they refactor old code or add new features. And because the development time takes into account testing, there is no money-sucking black hole at the end of the development process.

3.4 Fourth Misconception

The fourth misconception of development as described by Brooks is that developers are bad at estimating how long it will take to complete a task. This problem has many causes, but even a few can cause massive problems. Some causes are developers not understanding the bug, large unrealized technical debt, and even the pressure of managers to complete a job faster. This all skews the viewpoint of the manager, and causes them to “turn up the heat” on the project.

If the time required to complete a task is not estimated properly, it can set the entire team spiraling off schedule and into despair. One, long, exacerbated bug can cause system melt downs, just because no one properly estimated what the bug fix involved.

Scrum attempts to solve this problem by using a more collective estimating method, often called Planning Poker. The general concept is each member of the Scrum team gets a set of cards, with each card weighted a different amount. The User Stories, or tasks for that sprint, are then laid out on a table, and each team member distributes their cards face down among each task. A higher weight given to a task means it is more important. Most importantly, this is all done without communicating with each other.

By using Planning Poker, Scrum teams avoid a huge problem: bandwagoning. By letting each individual speak their mind, no one’s voice is unheard and tasks are weighted by what the entire team thinks, rather than just the loudest or most senior developer.

3.5 Fifth Misconception

The last misconception is that adding people to a problem will solve it faster. Writing good software is just as much problem-solving as it is programming. Unfortunately, throwing more people at a problem to solve it only works up until a point until saturation is hit. Two programmers may be able to solve what one could in a longer amount of time, but ten is too much. Although the math is there, this unit of a man-month simply doesn’t exist in the world of programming.

When managers throw more and more developers on a single problem, other effects start to take play. Much like how gravity is only present in large groups of atoms, much of the bureaucracy of teams only starts to play in with large numbers. The advantage of small teams is that everyone is rather independent and can be free to solve a problem in the most efficient way, whether it is will liked or not. By adding more developers, managers just add more process to the team.

Scrum encourages small teams of about six or seven developers for this reason. It is small enough that work can get done well and resources are available, but it doesn’t destroy developers time with process. Another popular technique in the Agile world is pair programming, where two developers work on the same problem, and this can get great results! But the key is that there is no substitution for small group work when in a development environment.

References

[1] Robert C. Martin *Agile Software Development, Principles, Patterns, and Practices* 1st edition, 2002.

[2] N.S. Janoff; L. Rising *The Scrum Software Development Process for Small Teams* 2000.

[3] Frederick P. Brooks *Mythical Man Month*

[4] Craig Larman *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* 3rd edition, 2004.