

Name: Ben Anderson

Candidate Number: [REDACTED]

Centre Name: Greenhead College

Centre Number: [REDACTED]

Computer Science NEA OCR H446/03 Computer Science A Level
Summer Series 2023

Security Camera System

Table of Contents

Analysis.....	6
The Problem.....	6
Computational Methods.....	6
What Makes The Problem Computationally Solvable.....	10
Target Audience & Stakeholder.....	10
Similar Existing Solutions.....	12
Ring.....	12
Tapo C100.....	16
Bluecherry.....	18
Storage.....	22
Security.....	23
DOS & DDOS.....	23
Wi-Fi Deauthentication.....	24
Radio Jamming.....	24
Brute Force & Dictionary Attacks.....	24
Packet Sniffing.....	25
Zero-Days.....	25
Containerization.....	25
Features.....	26
Essential.....	26
Future Iterations.....	28
Not To Include.....	29
Development Methodologies.....	30
The Waterfall Method.....	31
The Spiral Model.....	31
Rapid Application Development.....	32
Agile Methodologies & Extreme Programming.....	33
Most Suitable Methodology.....	33
Solution Limitations.....	34
Solution Requirements.....	35
Software.....	35
Hardware.....	35
Success Criteria.....	36
Design.....	37
Programming Language.....	37
Decomposition.....	38
Brief Overview.....	39
Server Software.....	39
Client Software.....	40
Recognising Shared Functionality.....	41
System Design.....	42
Server Software.....	42
Account System.....	42
Considering System Separation.....	42
Considering Account Storage Solutions.....	43
Considering Account Sessions.....	44
Class Designs.....	44
Pseudocode.....	46

Camera Viewer.....	51
First Iteration.....	51
Second Iteration.....	52
Third Iteration.....	55
Networking Server.....	58
Implications Of Language Choice.....	60
Client Software.....	61
GUI.....	61
Main Window.....	61
Camera Connection Window.....	62
Settings Window.....	62
Control System.....	63
Networking Client.....	64
Common Networking Module.....	64
Packet Abstraction.....	64
Connection Abstraction.....	66
Test Design.....	67
Standard Test Data Sets.....	68
Integer.....	68
String.....	69
Real.....	70
Character.....	70
Boolean.....	70
Standard Erroneous Test Data Set.....	71
User Inputs.....	71
Configuration Inputs.....	71
Camera Connection Inputs.....	72
IP Address.....	72
TCP Port.....	73
Post Development Testing.....	74
Black Box Tests.....	74
White Box Tests.....	75
Development.....	78
Pre-Development Choices.....	78
Integrated Development Environment.....	78
Build System.....	78
Project Structure.....	78
Version Control System.....	79
Documentation.....	79
General Third Party Dependencies.....	79
Automatic Code Generation [Lombok].....	79
Unit Testing [Mockito, JUnit].....	79
Annotations [JetBrains Annotations].....	80
Logging [Log4j].....	80
First Prototype.....	80
Standard Test Suite.....	82
Common Networking Module.....	85
Packet Abstraction.....	85
Connection Abstraction.....	90
Connection Class.....	91
PacketController Class.....	94
PacketSender Class.....	97
PacketListener Class.....	102

Testing Module Prototype.....	107
PacketController Class.....	107
PacketListener Class.....	118
PacketSender Class.....	123
Server Software.....	127
Configuration.....	128
Testing Submodule Prototype.....	134
Networking Server.....	137
Testing Submodule Prototype.....	143
Camera Viewer.....	165
CameraViewer Class.....	165
Testing The CameraViewer Class.....	167
Video Classes.....	168
Testing The Video Classes.....	172
Combining Server Submodules.....	175
Client Software.....	177
End-To-End Testing First Prototype.....	180
Stakeholder Black Box Testing.....	183
Second Prototype.....	184
Account System.....	184
Hashing.....	186
UserManager.....	189
Authentication Flow.....	194
Graphical User Interface.....	205
Implementation.....	205
Testing.....	211
Main Window.....	212
Camera Connection Window.....	213
Invalid Port Pop-ups.....	214
Connection Error Pop-up.....	215
Successful Login Correct Credentials Pop-up.....	215
Unsuccessful Login Incorrect Credentials Pop-up.....	215
Connected Camera View.....	216
Reconsidering The Settings Window.....	216
Configurable Accounts.....	216
Stakeholder Black Box Testing.....	217
Third Prototype.....	219
VideoEncoder Class.....	220
VideoEncoderTest Class.....	223
Video Integration.....	227
Developer Black Box Testing.....	228
Image Compression.....	228
Stakeholder Black Box Testing.....	229
Evaluation.....	230
Final Testing.....	230
Evidence.....	230
White Box Testing.....	230
Black Box Testing.....	244
Success Criteria.....	249
Fully Achieving Point 10.....	250
Achieving Point 15.....	251
Usability.....	251
Client Software.....	251

Server Software.....	260
Project Successes.....	261
Solution Limitations & Maintenance.....	262
Solution Improvements.....	263
Final Program Code.....	265
Bibliography.....	353

Analysis

The Problem

People across the globe own possessions which either have sentimental or economical value and need protecting, however there cannot always be a security guard watching over said belongings. There are other methods than just security guards though, using a security camera is a proven and effective way to negate the risk of theft whilst also capturing evidence regarding any illegal activities that it may witness.

Security cameras are widely used by police in most countries to identify criminals, prevent crimes from happening and aid in quickly solving criminal cases with material evidence. These benefits also apply to the average consumers of security cameras, as they can work closely with law enforcement and provide any necessary footage captured by their cameras to them. Furthermore, one of the most positive effects of an individual installing their own security camera is simply the peace-of-mind it can provide – especially if they previously felt at risk. It is recognised that security cameras are not a complete solution to home security, however research shows they help a lot.

Large companies often have security teams and expensive security solutions, but small businesses and individuals do not have the ability or finances required to implement such solutions. Many people see security cameras as a more cost-effective solution however the majority of them still come at a price that makes the average family uncomfortable to purchase, despite the features being mainly basic. Businesses and individuals are willing to spend as much money as it takes to protect what is valuable to them and it is arguably immoral that the ridiculously high pricing of quality security camera systems reflect this.

I will use computational methods when creating the security camera system to ensure I create the type of product that I envision. For example, I will use abstraction to ensure I don't include irrelevant features, use decomposition to make the development process easier, use performance modelling to ensure the camera will have the necessary performance to be usable, use concurrent programming to increase efficiency of the system, use logical thinking to make the system non-linear, and think ahead in order to create adaptable, reusable code.

Computational Methods

Method	Justification of use	How it can be incorporated
Abstraction	To best suit the product of a security camera, unnecessary features or details that don't... <ul style="list-style-type: none"> • enhance the security of the user, • enhance the security of the user's data, • enhance the user experience, ...should not be implemented.	A clear example of unnecessary information would be how exactly the security camera transmits data over the network to the user's device and each piece of data that is received by the user. I will use information hiding by only showing the end user the images captured by the security camera instead of informing them of each individual packet transferred across the network. This helps the user focus on the what the camera is seeing instead of distracting them with the de-

		<p>tails surrounding it, as that's not what they're interested in.</p> <p>Another way abstraction will be used is when establishing network connections between the user and the camera in order to transmit the video to them as they don't need to be aware of anything like a protocol handshake – they only care about the result. Therefore, I will only show the user the result of successfully connecting or the reason why the connection may have failed – this is a form of information hiding. Also, the functionality of connecting to the camera could also be described as functional abstraction as the user doesn't need to know how it works, just the basic use of it.</p>
Decomposition	<p>Problems can be complex, especially those which contains multiple large sub-problems. Decomposition breaks down a complex problem into several simple modules, which each accomplish an identifiable task whilst being more approachable and manageable. This makes things easier for the developer.</p>	<p>Instead of just picturing a 'security camera system', I will break the system down into multiple modules which can be easily manageable and adaptable by analysing the inputs and outputs of the system. These de-coupled components will combine to create a functioning camera security system. This is beneficial as each module can be developed independently and makes the system much more flexible and adaptable to change in the future if design decisions change during the development process.</p>
Performance Modelling	<p>It is imperative that the security camera feed is smooth and constant. The person viewing the camera needs to be seeing the footage in almost real-time otherwise they might not see an important event that occurred. Performance modelling can be used to analyse what may cause a delay in camera streaming before implementing it into the final prototype.</p>	<p>Mathematical approximations for the runtime of algorithms can be used to predict potential performance issues and optimise wherever possible. I will use Big O notation to measure the runtime of my algorithms. I will model the time required for my algorithms to run on expected data sets by taking into account their time complexity.</p> <p>For example, a binary search has a time complexity of $O(\log n)$, which is logarithmic. Therefore, as the number of elements in the dataset being searched increases, the time for the algorithm to run would increase proportionally to the logarithmic scale of the number of elements. Choosing algorithms with a low time & space complexity, and then modelling their impact on application performance using performance modelling, can help</p>

		<p>achieve a more efficient application and smoother user experience.</p> <p>This should also be applied to data structures, as there are many different data structures specialised for different operations. For example, adding an item to the end of a Queue has a time complexity of O(1) (constant time), whereas adding an item to the end of a Linked List has a time complexity of O(n) (linear time) as every element must be traversed in order to locate the end of the list – a Queue stores and maintains a tail pointer to avoid this pitfall.</p> <p>Performance modelling can be applied iteratively by making several prototypes and identifying what is causing performance issues at each one, then optimising it for the next prototype. This is considered to varying degrees, depending on the development methodology used for the project.</p> <p>Due to the image/video processing that may be needed when working with the camera, I also need to be careful about which algorithms I use in order to have the least amount of time between an image being taken from the camera and being streamed to the user to ensure an elegant, seamless user experience. I will evaluate multiple methods to ensure I choose the most optimised (and achievable) solution.</p>
Thinking Concurrently	<p>Thinking concurrently refers to a system being able to manage multiple tasks at the same time. The user shouldn't have to wait for one task (or line of code) to complete, before the next one starts – especially if the task takes a long time.</p> <p>Thinking concurrently includes both concurrent and parallel programming and can increase efficiency of code when implemented correctly – especially when the problem contains multiple tasks that can be run concurrently or in parallel to reach the end result faster than otherwise.</p>	<p>In order to allow the user to access the camera from multiple devices, the camera could make use of multithreading to concurrently manage the connected users and communicate with them simultaneously – using concurrent and parallel processing.</p> <p>It would be ideal to access the camera on one thread and then use concurrency and multithreading to send that data to user devices to view instead of doing it all synchronously as that would cause a large delay until the next image was read from the camera due to the delay caused by sending packets across a network us-</p>

		<p>ing protocols such as TCP. An asynchronous approach would allow user devices to get a camera feed closer to real-time and more constantly updated than previously possible – creating a much more elegant solution.</p> <p>Data structures such as linked lists should be used cautiously in a concurrent context. Acting as a shared mutable resource, different threads could interact concurrently with the shared linked list whilst believing it is in a different state than it really is. For example, a data race could occur if both threads tried to remove an element from the linked list at the same time, resulting in one thread attempting to remove empty memory from the data structure, resulting in undefined and unpredictable behaviour. This is yet another reason why it is important I am careful when implementing concurrency.</p>
Thinking Logically	<p>Thinking logically refers to the use of branching/selection and loops/iteration. This adds interactivity, interest and complexity to the program, creating a non-linear experience. Without these logical operators the program would be completely linear, and the user would feel insignificant to its result and execution. However, with logical thinking, the program can produce different outcomes based on conditions, or repeat instructions several times.</p>	<p>One module to the problem that I will implement is a login system to restrict access to the camera, hence increasing the security of the camera and user's data. Selection will be used when authenticating the user to check their login credentials and authorise them access to view the camera.</p> <p>Another example of thinking logically in the system will be the software that the client's devices use to connect to the camera. There are many edge-cases that must be handled when implementing networked software solutions and the software will need to be able to account for that. Branching will be used judiciously to accommodate for as many possibilities as possible.</p>
Thinking Ahead	<p>This refers to careful planning to make sure the project is completed efficiently. Thinking ahead also includes determining preconditions to avoid potential bugs in the code, identifying inputs and outputs of the program to ensure I am creating the program to the required specification, and caching data when necessary to reduce load times.</p>	<p>I will reuse code in order to more efficiently write the modules of the problem and save time during the development process. For example, I will make a shared networking module to create connections between clients and the camera and vice versa. This lets me reuse the code to send and receive packets – as both connections need to work in the same way in order to be compatible with</p>

		<p>each other anyway. The shared networking module will contain code to increase the convenience of development and reduce duplicate code.</p> <p>Another example of how thinking ahead may be used, is in the database query language SQL. This could be used as it promotes a consistent database structure and can often lead to more organised databases.</p>
--	--	---

What Makes The Problem Computationally Solvable

Cameras are underpinned by theory and mathematics as bytes of data are what determine the RGB (Red-Green-Blue) values of each pixel in a captured picture. Computer networking can also be used to send bytes over a network in packets. Combining these two pieces of information give us a theoretical way to capture images using a camera and send the data of said images across a computer network to be able to remotely view what the security camera can see.

The problem of a security camera can also be abstracted as many systems already exist which make up the security camera. For example, cameras will already capture images and give us access to them therefore most of what I need to do is manipulating the data to send it across the network. And from this seemingly large task, I can derive sub-tasks by decomposing the problem. These sub-tasks/modules include a streaming server to send image data over the network, viewing software for the user to receive image data over the network and display it and potentially even a login if I choose to add authentication after initial prototypes.

One of the largest parts of a security camera that make it computationally solvable is the fact that after initial setup has been completed, the functioning of the security camera is effectively automatic as it will just keep streaming the image data to connected clients until stopped. Automation is an ideal trait for a problem in order for it to be computationally solvable.

Target Audience & Stakeholder

The general target audience for this project is small-medium sized businesses, the specific stakeholder for this project is the company [REDACTED]. They are a small company providing [REDACTED].

However, there have been instances of theft in the alleyway outside the company's office. A security camera would help de-incentivize criminals from operating near the office, hence increasing its security – this is something [REDACTED] have expressed interest in. It is worth noting that the stakeholder is fairly technically literate and therefore will be competent when it comes to carrying out the installation process.

I asked the company to complete a questionnaire regarding their use case of the camera and any specific requirements they had. Here are all the questions (and their respective responses) from the questionnaire:

- Would a security camera make you feel more comfortable that thefts are unlikely outside the office?
“Yes, it’d be nice to know what shady activities are going on outside the office.”
- What environment do you plan to deploy the security camera in?
“We plan to deploy it on the wall, in the alleyway outside our office.”
- What features do you see as a requirement for the camera?
“It needs to be easy to install, not needing much data and user-friendly.”
- How important is the efficiency of the system to you?
“I want it to be quick enough that I can easily access it and not spend ages waiting for things to load.”
- What type of design would you like the user interface to take?
“I would like it to be easy to use and not too complicated.”
- What kinds of devices do you plan to view the security camera from?
“We plan to view it from computers in the office.”
- How long would you like to be able to view events after they happen with the camera?
“It would be nice if I could view it a few days later in case I’m away.”

Their response to the questionnaire provided a valuable insight into how I could develop the system to suit their requirements. To effectively meet the company's needs the security camera will need to be wall-mountable, therefore the system needs to be able to run on small, portable devices such as the Raspberry Pi which have very limited system resources. Due to these restrictions, I will program the system with efficiency and memory usage central to many design decisions – including the algorithms I implement.

Furthermore, I will ensure the portability of the software by providing an easy way to distribute the developed system onto other devices as well as making the program function as expected independently of the operating system it is running on. This will make it easier for the stakeholder to move the system onto an embedded device such as a Raspberry Pi and mount it on the wall in aforementioned alley outside their office.

The stakeholder also mentions that they would like to view camera footage after the events occur. This means that the stakeholder requires storage of video footage somewhere, I'll touch more on storage solutions and the approach that I'll take with it as I move through the analysis section.

Finally, the devices they planned to use to view the security camera were desktop computers, therefore desktop applications will be sufficient for them to use to view the camera. This will also likely provide a more enjoyable user experience when interfacing with the camera as I have prior experience developing graphical user interface (GUI) desktop applications in comparison to web applications.

In conclusion, the stakeholder questionnaire has revealed some interesting needs which may not have been considered prior to the stakeholder completing the questionnaire. I will delve further into each of the requirements when researching alternate solutions as I believe it will give a very nice comparison between products which implement their needs and ones that don't. Using this information, I can try spot a trend and adapt my product to suit their needs the most effectively.

Similar Existing Solutions

Ring

Ring are a subsidiary of Amazon, providing home security solutions. I will reference both their security camera and smart doorbell solutions due to the large overlap between the products and the fact they are most known for their smart doorbells. I believe I should analyse Ring's security cameras due to their large market share of 17.9% (with respect to their smart doorbells) in 2020[41] - more than any other single company.

The security camera section of Ring's website as of 9th March 2022 can be seen below:

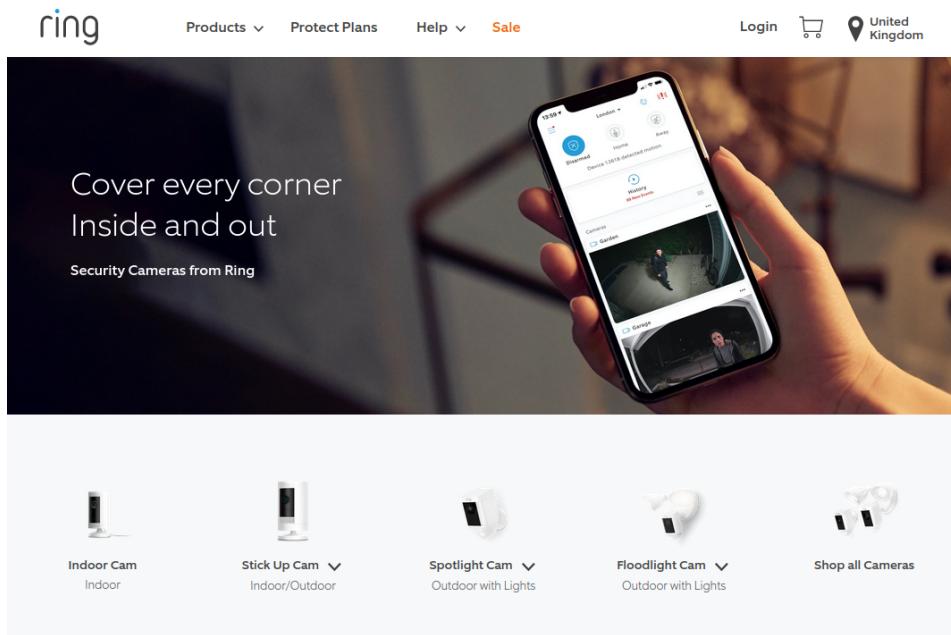
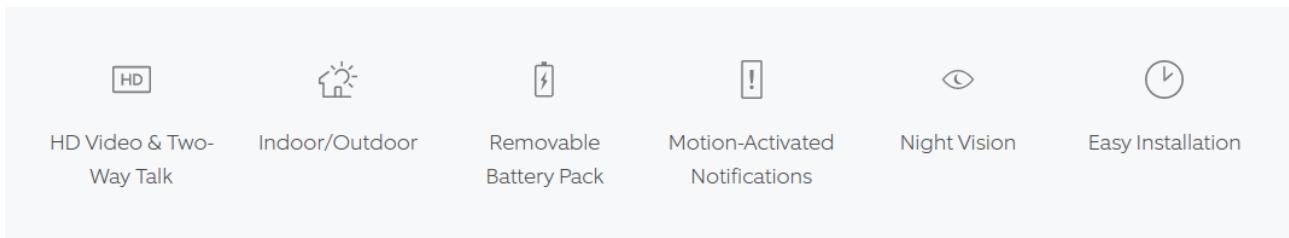


Figure 1: Ring website homepage

Here they display their different security cameras as well as the software that accompanies them. Focusing in on a specific camera (the 'Stick Up Cam')[35] displays the camera's features:



Goes everywhere. Sees everything.

Add eyes and ears inside or out with the versatile camera that goes almost anywhere. With countless placement options and customisable motion settings, you can find your perfect setup for you and your home.

Figure 2: general features of Ring products on their website

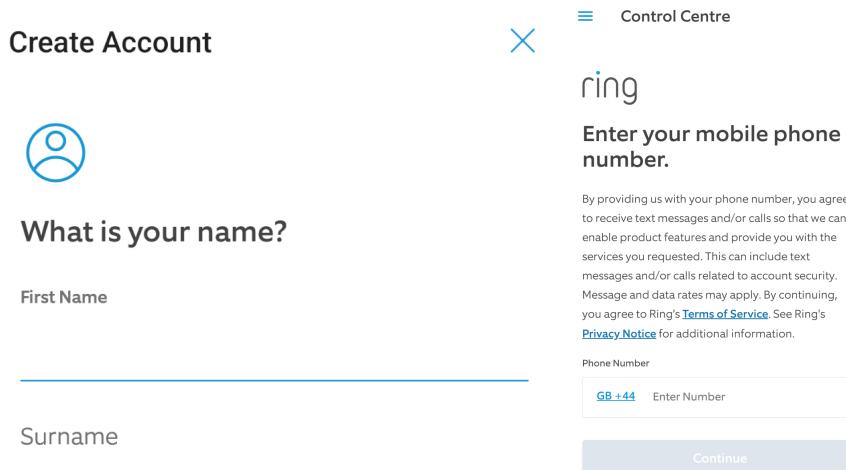
Many Ring products feature two-way ‘talk’/audio (a speaker and microphone) or at the very least one-way audio (a microphone) – and the ‘Stick Up Cam’ is no exception. From a security perspective, there aren’t many cases when two-way audio would be particularly useful; hearing the sounds that the security camera hears is very unlikely to provide any benefit to the user remotely viewing the camera. However, an alarm sound ringing out from the camera when it detects motion under predetermined conditions is something that I think is worth considering as part of a future design iteration due to the fact it could consistently scare off criminals, especially opportunists.

The security camera also advertises features such as night-vision which improves visibility in low light conditions. Many of Ring’s security cameras that support the night vision feature, achieve it by using an infrared light sensor on the camera and combining depth information from it with any available ambient colours[36]. However, due to the fact that the scope of this project doesn’t include the actual hardware, night-vision support cannot be achieved in the same way (using infrared light sensors). The stakeholder and other users who wish to adopt the project should be free to choose their own hardware, adapting to the price of their choice – this means that I cannot guarantee any advanced sensors will be available on the camera. One way to still support a night-vision feature would be to implement a software-level solution using image manipulation algorithms.

The stakeholder for the project wishes to use the camera in a side-alley which has limited lighting and visibility. Therefore, I aim to implement a software-level ‘night-vision’ solution in order to make the camera more effective in the evening (lower light levels), especially since this is when the majority of thefts occur.

Ring’s security camera boasts ‘easy installation’, this is something which I believe is vital to user satisfaction when it comes to self-setup security camera solutions. I aim to achieve a similar level of installation ease as the Ring security cameras, if not easier.

There are some aspects to the Ring security camera installation and setup procedure which I feel should be completely removed or at least adapted, to help improve user experience and privacy. Firstly, the use of any Ring device requires a Ring account, which asks for an excessive amount of the user's information including full name, residing country, email address and phone number. Shown below is the Ring application prompting for some of this information:



*Figure 4: Ring account creation screen - Figure 3: Ring account creation screen - name
- phone number*

All of this information is mandatory for the user to begin using the Ring app and setup their device. For the average user, this level of information is unnecessary and there are very little justifications that can be made for needing the information. This strongly brings into question the extent to which the user's privacy is considered; more privacy conscious users may feel uncomfortable giving this information Ring and linking it all directly with their camera.

With this information in mind, I further questioned the stakeholder as to how much control they would like over their data, to which they responded that they didn't want it to be unnecessarily sent around. This answer was what I was expecting, as I have recognised that many people have gradually become more conscious about how their data is used and where it is sent. For this reason, I plan to design the camera setup in such a way that it is non-intrusive for the user (account only requiring username and password) and only stores account information locally on the user's device or on the security camera – both of which are strictly under the user's control and on their local network.

Another aspect of the Ring security cameras that I wish to change, is that they often communicate with external servers. In reality, a home security camera that operates on the local network doesn't need to reach out to external servers, apart from for an occasional automatic software/firmware update. Ring's dependency on external services means that it is necessary for the device to have internet access. However many individuals or small businesses such as [REDACTED] recognise the potential security implications of having a security network connected to the internet – whether it be through a conventional router or other means. Combining this with the fact it is mostly unnecessary for the camera to access the internet, it seems clear to suggest that the camera should be fully

functional without internet access – the only requirement is for it to be connected to a local area network (LAN). This will be another design difference between the camera I will make and the Ring security camera.

One of the most attractive features of Ring to the average consumer, is the user interface and experience provided by the software that is used to access the camera. The application features a very intuitive user interface making it easy for the user to access the camera or change its settings at any time. This provides a nice element of convenience when interacting with the camera which many solutions fail to properly address. Not only is it carefully laid out, it uses high-quality, appealing components like buttons and toggle switches; all of which following a familiar colour palette, creating a sense of ease for the user. Here I have annotated some key features of the main screen[43] a user with a Ring smart doorbell interacts with:

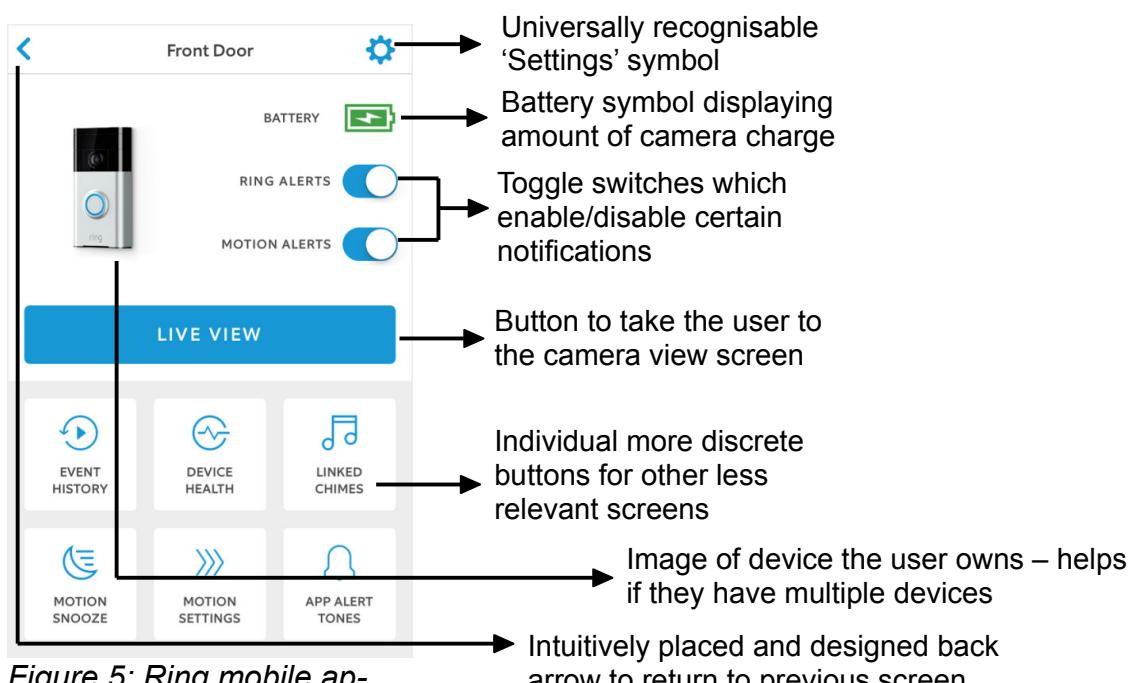


Figure 5: Ring mobile application controls screen-shot

This is a feature of the Ring application I would like to mimic as I feel it is vital to ensuring the user enjoys interacting with the camera system. I recognise that whilst I may wish to create a user interface of similar quality to the Ring application, I don't have the same level of visual-design skills that the front-end developers at Ring do. Therefore, I will do my best to ensure all user interface elements are placed intuitively and don't produce unexpected behaviour, whilst trying to make the software somewhat visually appealing, to provide a good user experience.

One feature that is paramount for a security camera system, is the ability to view previously recorded events by the camera using the software. Here is an example taken from the marketing page of the 'Stick Up Cam'[35] on Ring's website:

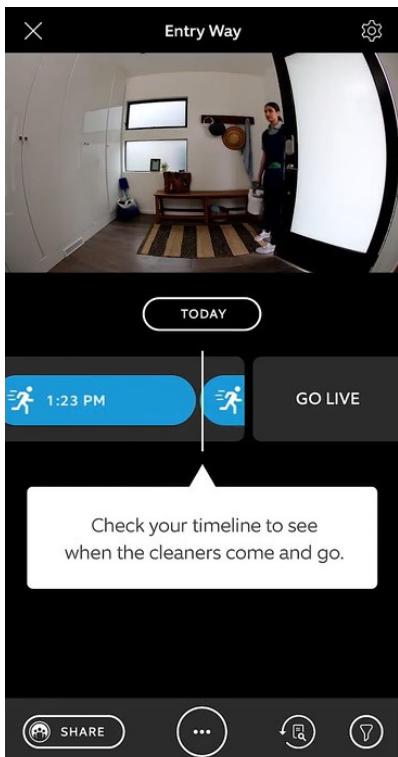


Figure 6: Ring mobile application video playback screenshot

Ring's video playback feature behaves like an embedded video player as on mobile phones there is no standard way to playback video to the user, external to the application. However, having the video player embedded into the software also makes the experience more seamless for the user when reviewing the footage – assuming the video player is implemented in a way which enables a nice user experience.

In terms of my implementation of video playback, I want the user to be able to download video files recorded by the camera onto their computer so that they can use the footage however they wish. Therefore, I may also add support for video download. I see video playback as a necessary feature to allow the user to review footage which they have recorded, this becomes particularly useful when the user needs to use video evidence to prove that something happened.

One of least attractive features of the Ring 'Stick Up Cam' security camera, is the high price of £89.99[35]. This is an incredibly high price to say it is just a simple security camera with accommodating viewing software. As I noted in my problem analysis, high prices of security cameras are a common occurrence, because companies know that consumers are willing to pay high prices for peace-of-mind, despite the fact the camera could easily be priced significantly lower. This further reinforces the reason why I am creating my own solution – as an example that retail security cameras can contain valuable functionality at only a fraction of the price of other security cameras.

Tapo C100

Much like the Ring 'Stick Up Cam', the Tapo C100[42] boasts features such as 'advanced night vision'. I couldn't find how they have implemented the feature from research, however its presence

further indicates that it is a necessary feature consumer security cameras. Combined with the fact that my stakeholder expressed interest in the feature, I still believe this is a vital feature to implement on a software level in my final product; although for the minimum viable product, it is not instantly necessary, and can therefore be left for a future design iteration.

The Tapo C100 also has notification features which will alert the owner's phone when motion is detected by the camera. The effectiveness of this feature is debatable. On the one hand, it can usefully inform the owner of the camera as to activity it detects which may be useful at some points, however it is very likely that by the time the user has received the notification and checked their phone, the possible criminal will have been long gone. Another problem with this feature, is that it is impossible to determine whether motion detected by the camera indicates criminal behaviour or not – the owner could be notified of their own presence when they are returning home.

For my stakeholder, the alleyway they wish to position the security camera is also the entrance into their facility, therefore having a motion detection algorithm to send them mobile notifications would not be useful as there would be a large number of notifications with very little criminal threat. Although, it is worth considering the implementation of motion detection without a notification system – perhaps just a text-based log of detected motion events

As I discussed when analysing Ring's security camera, an alarm feature on the camera is an interesting idea. The Tapo C100 actually implements such functionality by having a sound alarm toggleable from the camera management application. It seems like the best way to implement an alarm feature if it was going to be added, because many of the negative points that come with motion detection alarms are removed if it has to be manually activated by the user.

However, I don't believe this feature would be often used, especially in my stakeholder's situation. The fact that they wish to place the camera in an alley outside their office to prevent thefts in the alleyway shows that their priority is not actually to protect the office itself – as they have had no issues with the office's security due to the alarm systems they have in place. Therefore, any thefts caught by the camera would have a very short duration, the stakeholder merely wishes to be able to record and report these events with evidence, not try to catch the criminals in the act and scare them – because there is very little chance they will catch them in the act when the event is so short and fast. It is for this reason, that a manually toggleable alarm feature may not actually be a useful feature for my stakeholder – even if most consumer-ready cameras have it.

The Tapo C100 has two-way audio, much like the Ring camera previously discussed. However, I retain my belief that it is not necessary or useful for my stakeholder's scenario. With the realisation that they will rarely actually catch a criminal in the act, my belief is further cemented and I don't see the need for any audio functionality in the camera software, due to the lack of benefit it would provide the stakeholder.

One of the limiting factors of consumer-ready security cameras (such as the Tapo C100) sold to mass markets, is that consumers are limited to the hardware and software provided by the camera. For example, the Tapo C100 can only store up to 128 GB of video footage on its local storage. Owners of the camera have little options if they wish to extend this storage. However, my approach to a more open system design, the fact the user can choose their own hardware, means that the amount of video footage that can be stored locally is arbitrary – the user is free to expand the storage infinitely (theoretically).

The question is whether the camera should record constantly or only when there is motion detected. It would be nice to allow the user to choose by allowing them to configure the setting in the application, however I feel this is a feature that should be considered more during the design phase, as it is closely related to how the system will perform and how much storage it will actually consume when tested.

The Tapo C100 has a feature I've not previously seen on security cameras – privacy mode. This feature allows the owner to shut down the camera to protect their privacy whenever they want – basically a remote power on/off button. This is not a bad idea as a feature, however it could raise security concerns. If a vulnerability was found in the camera allowing other people to access this 'privacy mode' feature, then anyone has the ability to disable the camera at all – rendering it useless. However, that is not a very likely situation to arise assuming there are no major vulnerabilities in the camera software.

But whilst it's not a bad feature idea, it equally wouldn't be useful for my stakeholder as their security camera will be placed outside in the alley, this is not a place I expect them to be spending much of their time or requiring privacy. Therefore I won't implement the feature on the security camera I am making. It may be an incomparable feature due to the fact that the Tapo C100 is often marketed as an indoor security camera, however I still think the feature should be addressed as it is still applicable for many outdoor scenarios – such as somebody requiring privacy in a garden, which would make sense.

Tapo's camera management software, much like Ring's, is a mobile application. However, as previously discovered from the questionnaire I had the stakeholder fill out, they will be accessing the security camera from their computer. Therefore, there is no need for a mobile application of any kind, contrary to the camera management applications available for other security cameras which are aimed at the general consumer. Instead, I will develop a desktop application for the stakeholder to access their camera from.

Tapo's camera management software also boasts easy camera setup much like many other apps. One element of the camera setup which I believe is noteworthy is how the mobile application discovers the camera in the first place. The application utilises the mobile phone's built in Bluetooth technology to discover and communicate with any Tapo security cameras in its immediate vicinity. This provides a seamless user experience as it results in much of the setup process being automated, which is definitely something that I aim to achieve – a refined user experience. However, due to the fact that I am allowing the user/stakeholder to use their own hardware and simply implement my solution on a software level, I cannot guarantee that the hardware systems my software will interact with will have Bluetooth capabilities. Therefore, I will analyse alternate solutions for automatically discovering the security camera upon setup during the design phase.

Bluecherry

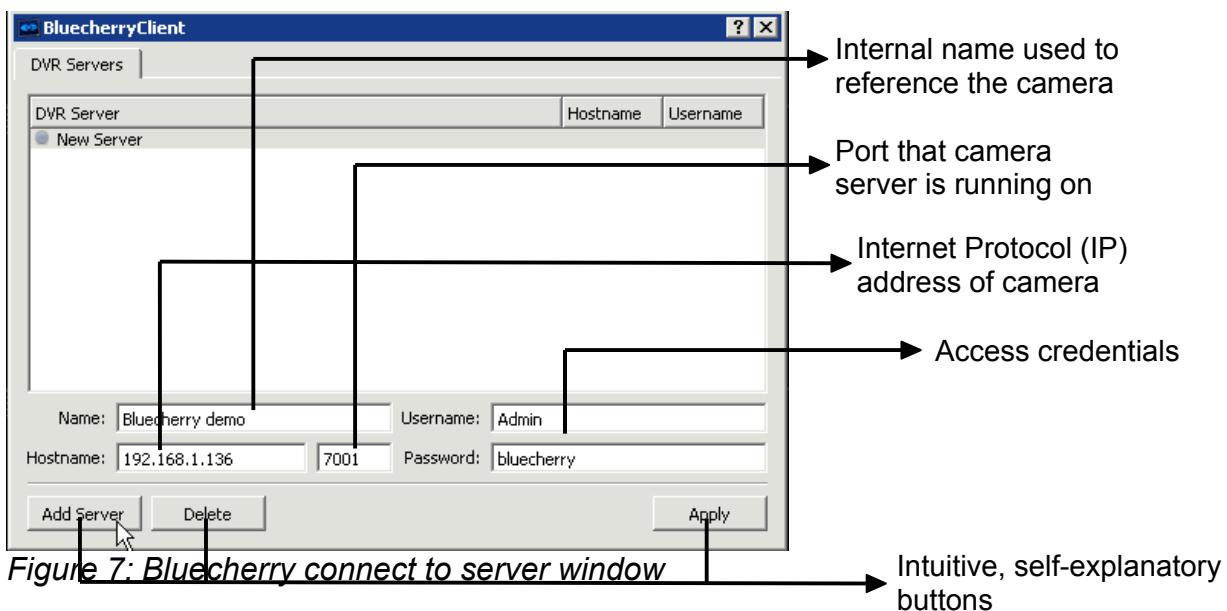
Bluecherry[2] is a piece of open source video surveillance software. It is not actually a security camera itself, instead Bluecherry connects to other security cameras on the local network (that follow the ONVIF specification[22]) and provides an elegant way to interact with them all from one application. Both server and client software are provided, however the server software only supports the Linux operating system.

The fact that the server software only supports the Linux operating can be seen as a weakness, as it means it doesn't support major operating systems such as Microsoft Windows. Now whilst Linux is often more computationally efficient than Microsoft Windows, and much more commonly used for servers, they are still quite niche in terms of the average user with only a 1.3% share in the desktop and laptop market (as of August 2020)[7]. Despite the fact that server software is often specialised to Linux, I would prefer an approach that favoured the use of all operating systems in order to accommodate many everyday users – especially because the average user has very little experience with the Linux operating system.

This operating system requirement is partly due to how Bluecherry structure their project and how their systems communicate – using a server to receive communications from cameras and then a client to communicate with the server. I also plan on a similar approach in the sense of using a client-server architecture, however the components will not serve the same purposes and will instead be non-dependant on operating system – making for a highly usable final product which almost anyone is capable of installing and using.

Bluecherry's server and client applications are both written using the C++ programming language, which is a compiled language, best known for its high efficiency, high speeds and low level control over memory. I believe this design decision was made to increase the throughput of the software, allowing for higher performance during streaming of cameras to clients. I will evaluate which language to use closer in the design phase, this will include the consideration of efficiency and my own existing knowledge.

The Bluecherry client has support for adding multiple camera servers and viewing multiple cameras at the same time. The following GUI can be used to register a Bluecherry server with the application:



The clear buttons and text fields in the user interface create a streamlined intuitive process for adding a camera server to the application. The fact that it also asks for an internal name for how to refer to the camera, makes it very easy for the user to identify different cameras in the application. I

would like to use a similar menu to add cameras and save them in my client software so that users can simply connect to an IP address and port using given credentials with the click of a button – instead of typing information in each time they launch the application.

Cameras added to the application through this menu will appear listed next to the main viewing area, giving the user the ability to change/access different cameras easily:

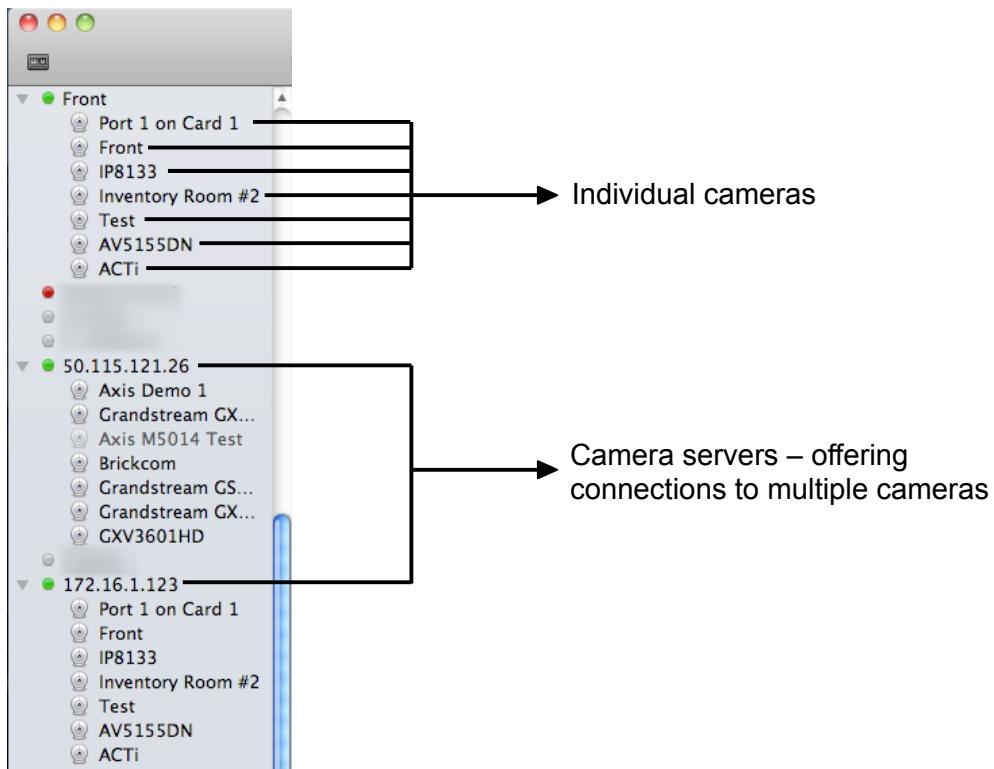


Figure 8: Bluecherry list of cameras

As noted in the Bluecherry documentation[6], the user can click on one of these cameras and literally drag and drop the camera into the main viewing area to view the camera and make it a part of the scene. Now whilst this is an incredibly powerful feature which creates a much cleaner experience for the user, it is something that is quite difficult to implement. When designing my own solution, I will try to find a balance between the implementation here and not having the list be interactive at all. Likely a solution which involves a purely clickable camera list, but not supporting drag and drop.

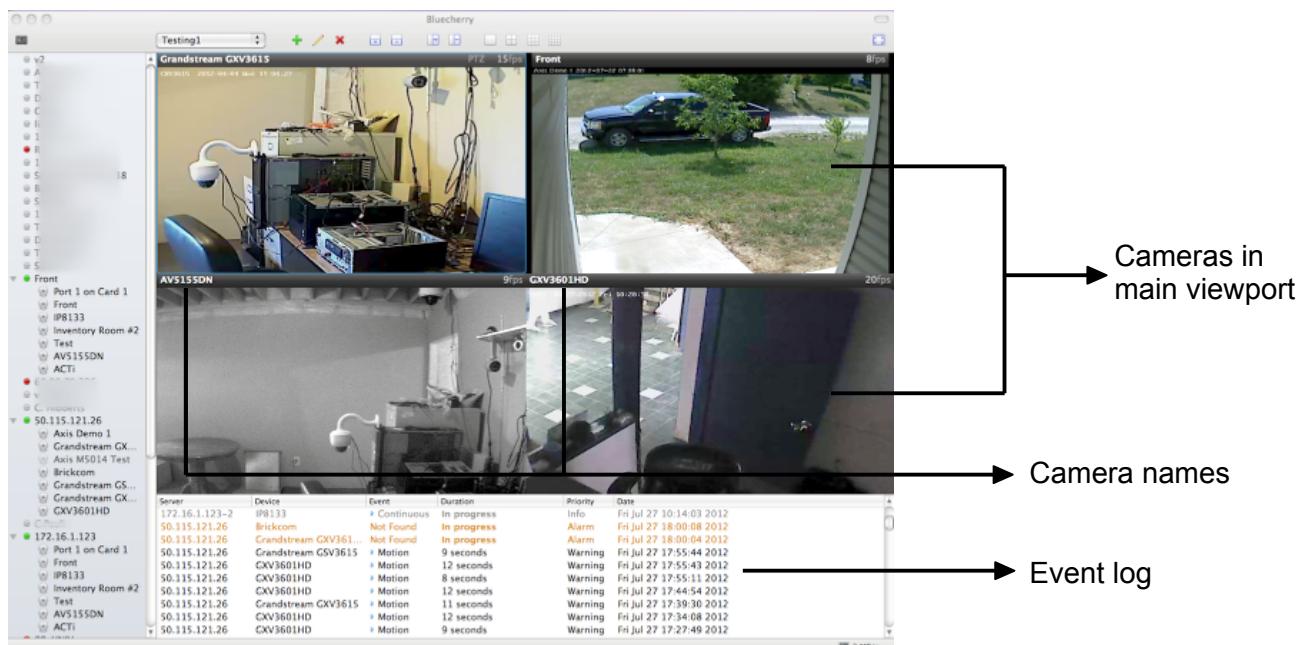


Figure 9: Bluecherry main application window

Shown above is what the main viewport looks like when it is filled with multiple cameras. Bluecherry supports adding/removing cameras to/from this main viewport in real-time, in response it will dynamically re-organise the scene in real-time to ensure camera views are covering as much of the main viewport as possible.

This is ideal for business settings, where there will be allocated security employees monitoring the cameras constantly as they can view as many cameras as possible whilst using as little screen space as possible – one screen per camera would be quite inefficient. Furthermore, large businesses are likely to have many more cameras than an average, security-aware family. Therefore I do not see this advanced, real-time organisation of camera views on the main viewport as an essential feature and will likely not include such functionality in my product. Mainly due to the fact that my stakeholder isn't a large business with more than 10 cameras to monitor, it is just a small office requiring one camera. If they wanted to add and view a second camera at the same time, they could always just open another instance of the client viewing software on their computer and place them side-by-side – one connected to the first camera and another connected to the second camera.

Also, if I'm not going to add support for multiple cameras in the main viewport, then the name of the camera in the view is not necessary either, although I will consider adding the name to the title of the window – it just doesn't need to be as intrusive on the main viewport as it is in Bluecherry, all-be-it still useful.

Finally, the way that Bluecherry incorporate the event log as a central feature to the application is noteworthy. I believe the primary reason for this is the fact that the software is designed to support many cameras at once, giving a much higher probability of events being triggered. However, as previously stated in my analysis of the Tapo C100, motion alerts and other types of events are not central to the focus of this project and are therefore out of scope when it comes to development of the client application. Therefore, I will have no 'event log' type panel in the user interface.

Storage

Security cameras need to store video footage that they capture so that the user can access it at a later date. The question is, how much storage will be required to store the recorded footage? I did some research into calculating the file size of video files, and found that it is actually quite difficult to calculate/predict file size of videos in advance. This is due to the fact that video file size depends on many factors, such as bitrate, compression ratio and encoding.

I found an article[5] providing averages and estimates for video file sizes of different resolutions, the below table is taken from the article:

Resolution	Bitrate	1 minute	Recording Duration per GB
4K (UHD)	20 Mbps	84MB	12 minutes
1080p (FHD)	5 Mbps	20MB	50 minutes
720p (HD)	1 Mbps	5MB	3.5 hours
480p (SD)	500 Kbps	2MB	8 Hours

The user/stakeholder may need to view footage for an arbitrary amount of time after it has been captured, as they may become aware of a crime being committed a few days after the event, or they may be unable to access the camera as they are on away.

I used the data provided to calculated the average video storage requirements (to three significant figures) for the following resolutions and durations:

Resolution	Duration (days)					
	1	3	5	7	14	21
480p	2.88 GB	8.64 GB	14.4 GB	20.2 GB	40.3 GB	60.5 GB
720p	7.20 GB	21.6 GB	36.0 GB	50.4 GB	101 GB	151 GB
1080p	28.8 GB	86.4 GB	144 GB	202 GB	403 GB	605 GB
2160p	121 GB	363 GB	605 GB	847 GB	1690 GB	2540 GB

Although 2160p (4K) cameras are not particularly common at the time of writing, they are becoming increasingly available and growing in popularity. Therefore, I believe it is worth considering the implications that 4K footage will have on the storage requirements of the system – especially due to the fact that the user can choose any webcam camera they want when setting up the system.

The data reveals that for 480p, 720p and 1080p, you can comfortably store 14 days of video footage on approximately half a terabyte of storage. A magnetic hard disk drive (HDD) with a capacity of 500 GB (half a terabyte) is quite reasonably priced at the time of writing, and is likely to become cheaper over time. Hence, if the system is not using a 4K camera, it can happily store two weeks of video footage, with even more on resolutions below 1080p. In my opinion, two weeks is a reasonable length of time to store the data for, because the user is very likely to have found a reason to save the footage within that period of time or managed to find time to access the camera and download it.

If the user wished to utilise a 4K camera with the security system, then I'd recommend they only store 7 days of video footage, as the increment in storage requirements between 7 and 14 days is

a whole additional terabyte – which is fairly drastic and can be quite expensive. Only storing 4K quality footage for 7 days seems like a fair compromise between storage time and file size.

It should be noted that the calculations carried out above are based on data which came from a small sample of Youtube videos as mentioned in the article where the table originates. There is likely to be moderate difference in the actual file size of video footage for a given resolution and duration in comparison to its predicted file size. Margin of error increases as the duration of the footage increases due to the nature of the calculation. Henceforth, the calculated data should be viewed sceptically (taken with ‘a pinch of salt’); the user should not expect to be able store video footage for an amount of time at a resolution which is predicted to produce a file size very close to their storage drive capacity. If the user is concerned that they may be saving a volume of footage close to the drive capacity, it is recommended that they store shorter or lower quality video footage.

A potentially useful feature, would be for the system to automatically detect and adjust camera quality, and suggest to the user how long to store data, all depending on the capacity of the storage device.

Due to the do-it-yourself (DIY) nature of the system hardware in this project, the storage can have an arbitrary capacity, meaning the volume and quality of footage that can be stored entirely depends upon the user’s hardware decisions.

Security

The involvement of a security camera system in capturing criminal activity means it is vital that the system itself cannot be compromised by any criminals. The system must retain its integrity to remain credible in a court of law and for the user to feel safe having it connected to their home network.

It is not completely unheard of, although it is rare, for criminals to perform cyber-attacks against security cameras before committing a crime in order to avoid being recorded. For this reason, I believe it is highly important to identify the security risks facing a camera system, and explore the mitigation techniques I can utilise to defend against them.

DOS & DDOS

DOS stands for Denial Of Service. A DOS attack is an attack whereby a malicious actor sends lots of packets over the network to a target device in attempt to slow down the target device. This happens because the target device attempts to decode and understand all the inbound network packets, despite the fact that they contain no useful information, as it is often difficult to distinguish between legitimate network traffic and network traffic that only intends to create noise and spam the target with meaningless data.

DDOS stands for Distributed Denial Of Service. DDOS attacks are an extension of DOS attacks, where instead of there being just one attacker spamming the target device with meaningless data, there are many attackers spamming the target. DDOS attacks are much more effective and therefore more common than DOS attacks; they are responsible for the majority of malicious server outages in history.

In order for a DOS or DDOS attack to be carried out on the security camera as a singular device, the attacker would need to be inside the local network or the network would need to be specially

configured to expose the security camera to the internet. If they did succeed in the attack, then the device running the security camera could crash or at least experience a severe decrease in performance, making the camera fairly ineffective. There is little that can be done to mitigate this attack other than ensuring that the malicious actor doesn't get into the local network in the first place – it's recommended that the network use the latest and most secure Wi-Fi standard, with a strong password.

If the DOS or DDOS attack was carried out against the router which the security camera was connected to (assuming the router was reachable via the internet), then it is likely that the camera would not be reachable for other devices on the network. In this case, one mitigation technique is to record and store video footage locally on the camera in order to retrieve it at a later time (once the cyber attack has stopped).

Wi-Fi Deauthentication

A deauthentication attack is a type of Denial of Service attack which exploits a vulnerability of the Wi-Fi protocol in order to remotely disconnect a device from a network without even being connected to said network.

The IEEE 802.11 Wi-Fi standard[52] contains a 'deauthentication frame' which is sent by a router to a device in order to disconnect it. However, the vulnerability comes in the fact that this frame/packet is not encrypted and therefore any device within wireless range of the target can send fraudulent deauthentication frames to disconnect a device from a network.

This attack is not as rare as DOS and DDOS, because there are many tool suites such as 'air-crack-ng' which provide the functionality to carry out this attack with just one command.

However, this attack was addressed in the IEEE 802.11w Wi-Fi standard amendment[53], which focused on encrypting management frames; it was introduced in September 2009 but is still not used by all wireless devices, therefore it cannot be assumed that the hardware which the stakeholder uses with the security camera system will be compliant with the specification amendment.

The impact of this attack on the security camera is similar to that of DOS and DDOS in the sense that the camera will not be accessible to other devices on the network. Therefore, this vulnerability further reinforces the idea that saving videos locally on the camera is a necessary feature.

Radio Jamming

Radio jamming is the deliberate transmission of lots of noisy radio signals in order to cause interference and disrupt radio communications – which Wi-Fi depends upon. Jammers are not that difficult to obtain and can be purchased online, although their usage is highly illegal in most countries.

The impact of this attack on the security camera is similar to that of DOS, DDOS and Wi-Fi Deauthentication in the sense that the camera will not be accessible to other devices on the network. Once again, the clear mitigation technique is to store recordings locally on the camera to be accessed once the radio jamming has ceased.

Brute Force & Dictionary Attacks

Brute force and dictionary attacks are methods of effectively 'guessing' the credentials to access a system. Brute forcing involves literally trying every combination of characters and digits as the

password to a system, while dictionary attacks use a predefined data set and combine entries together to generate potential passwords.

If the password to the security camera system is guessed using these methods, then the integrity of the camera system and the data on it, will be lost. Hence, it is important that I enforce strict password requirements in order to ensure that the stakeholder chooses an un-guessable password when setting up the system. This will effectively mitigate any form of password guessing as a problematic attack for the system.

Packet Sniffing

Packet sniffing is the action of listening for network packets that are not associated with you, and monitoring them. The only requirement to carry out packet sniffing is to use a Wi-Fi adapter which supports monitor mode, a fair amount do and they can be acquired online for fairly cheap. The ease of carrying out the attack means that its implications must be evaluated.

By packet sniffing, a malicious actor can view the content of packets on a network. If the user was connected to their camera, then the camera would be sending network packets of the video footage to the user – this could be captured by the attacker, who could then watch the camera feed themselves. As previously mentioned, nobody without proper permission should be able to view the camera footage.

One effective mitigation of packet sniffing, is to encrypt data that we are sending across the network. This attack is one of the main reasons why protocols such as HTTP (hyper-text transfer protocol) had to evolve and implement encryption, becoming HTTPS (hyper-text transfer protocol secure).

Therefore, I shall encrypt data using either symmetric or asymmetric encryption before sending said data over the network, in order to mitigate the threat posed by packet sniffing.

Zero-Days

The last threat I wish to discuss, is that posed by ‘zero-days’. A zero-day is a vulnerability which is not known about by the developer yet still exists in the solution. As soon as the vulnerability is discovered, it is referred to as a zero-day, it won’t be patched on any systems.

Attempting to stop a zero-day from existing is like trying to expect the unexpected. You can test software lots, but it is not always possible to discover every single vulnerability in the code before distributing it.

Therefore, it is ideal to put security measures in place as a form of damage control if a vulnerability is discovered in the system, i.e. an attacker finds a way to remotely access the security camera without proper permission.

Containerization

One way I can increase security of the system is to containerize any software applications I develop as part of the project. Containerization is the packaging of all required files for an application into a single isolated environment.

A container cannot access data outside its isolated environment, similar to a virtual machine. However, a virtual machine uses completely different system resources in comparison to the rest of the computer, whereas a container shares system resources with the rest of the computer.

Modern containerization technology such as Docker are very useful, not just for security but usability too. Docker produces containers which can run on any operating system and behave exactly the same across each. So I thought that Docker would be a perfect candidate for this project as I want all software I develop to be cross-platform, allowing a user to turn any computer into a security camera.

However, whilst researching Docker and thinking more about its implementation into this project specifically. I wanted to check that the container would actually be able to access the camera hardware that the system was going to use. I was already aware of the '--device <file>' flag that can be used with Docker to pass through devices to containers (when on Linux), however MacOS and Microsoft Windows do not treat cameras and external devices as files like Linux does, and therefore cannot achieve the same functionality.

I researched further into this topic: reading different articles[14] and Q&A posts. Ultimately, I reached the conclusion that the only ways of achieving what I needed, were incredibly convoluted and not at all plausible. Sadly, this meant that I had to abandon the idea of using containerization with my software and would hence forfeit some security in the event that the system was breached.

I came to the conclusion that the best way I could defend against zero-day vulnerabilities was to require as few permissions as possible in the software, so an attacker could only do minimal damage if the system was infiltrated.

Features

Having analysed Ring, the Tapo C100 and Bluecherry DVR, I have established a good foundation of essential features for the security camera software which will improve the user experience for the stakeholder and create a more useful product. I have also considered which features could potentially be introduced in a much later iteration of the project and are not part of the minimum viable product (MVP). Finally, I have analysed which features I believe would actively make the system less user-friendly and less suitable for its purpose, in order to ensure I avoid implementing them.

Essential

Here are the features I have deemed as essential for the project, they will be added as the system progresses through iterations:

Feature	Explanation & Justification
Easy to install	It is essential that I create a seamless experience for the stakeholder when they install and setup any software made as part of the project, otherwise the user could become frustrated.
Function without internet	For security and reliability purposes, the software should all be able to function without access to the internet, as outages have previously occurred at the stakeholder's office. If the internet stops working then the camera should continue to function normally, this will also protect against previously men-

	tioned deauthentication and denial of service (DOS) attacks (see the security analysis).
Don't send data outside the local network	For privacy and security reasons, the software should never attempt to send data outside the local network (such as accessing the internet). From a privacy perspective, the stakeholder can be sure that data is never being shared without their permission; from a security perspective, there will be a decreased risk of data being stolen through use of packet sniffing (see the security analysis).
Working network communication	The server software needs to be able to communicate with the client software, therefore both pieces of software will need to successfully implement networking communication to achieve this.
Correct serialization and deserialization of network packets	Data integrity needs to be maintained when data is transferred across the network. For example, if the image data of what the camera sees is corrupted when it reaches the client application, then the image will be meaningless and the camera would effectively be non-functional.
Minimal yet intuitive user interface	The client software should have a minimal graphical user interface in order to be very easy to understand and not overwhelm the user. User interface components shall be placed intuitively, in order to provide the best possible user experience.
Can login to the camera server using credentials	The server software will make use of an account/user management system to ensure that nobody can access the stakeholder's security camera without proper permissions. Whilst this wasn't something explicitly specified as a requirement by the stakeholder, my previous experiences with client-server software indicated that account systems are often necessary for security, whilst add more flexibility to the system (in terms of if multiple people would like to access it) as user permissions can be controlled with different access levels.
Users can be added and deleted to the camera	
Downloadable video footage	Video footage must be downloadable from the camera onto the user's computer. This allows the user to easily access media recorded by the camera and utilise it in an effective way, for example they may download some footage from the camera, place it on a flash storage device and share it with law enforcement.
Cross-platform	All software solutions developed as a part of this project should be cross-platform. One of the main reasons for this is the fact that I want the software to function on different setups, for example a normal desktop computer running Microsoft Windows and a Raspberry Pi running Raspberry Pi OS (based on Linux). A cross-platform approach will provide software which allows both of these devices to be used as a security camera with very little difference from a usability perspective. This essential feature will also impact which programming language I choose to develop the software for this project.
Expandable storage	Whilst I have labelled this as a necessary feature, it is also a feature that comes automatically with the project approach I have chosen. The fact that the user chooses the hardware for the project means that the storage capacity of the system is variable and depends upon the choices made by the user. Therefore, if the stakeholder wishes to prioritise storing data for a long period of time at a high resolution, they will need to use a storage device with high capacity (see the storage analysis section for more information).
Server software	The server software needs to be able to capture image frames from the cam-

can access the camera hardware	era hardware connected to the device running the server software, as these image are what are sent to client software instances.
Functioning video creation from camera frames	Creating videos from the camera frames means that videos would be downloadable by clients even if a client wasn't connected to the camera when the event happened. As noted in prior analysis, background creation of videos from the camera feed is integral to the security of the stakeholder.
Save video locally to client computer	Video footage will be stored locally on the camera as the stakeholder requested to be able to view security footage multiple days after events occurred. Furthermore, storing video footage locally also helps mitigate multiple threats that face the security camera – as discussed in the security analysis.
Using multithreading for networking	The client should not have to wait for network packets to be received until the program responds to their next input, especially since there is a lot of data being sent and received. Therefore, a concurrent approach is required so that the user has a much more seamless experience.

Future Iterations

Here are the features I have deemed as appropriate candidates for future iterations of the project, it is not ideal for them to be implemented in early prototypes as they provide little-to-no insight regarding the effectiveness of the solution; they should only be added as refining features in the later stages of development:

Feature	Explanation & Justification
Software-level night-vision	Night vision would be a useful feature for the stakeholder due to reasons previously discussed. I will try to implement it on a software level using some image manipulation at a later stage in the project; it is not vital for the camera to do its job, but a night vision feature would improve user experience and usability of the camera.
Automatic discovery of cameras on network	A feature that would highly improve the user experience during the setup process would be automatic discovery of cameras on the network. It wouldn't be particularly difficult to implement, and would yield an incredibly smooth setup experience. The only reason why this is not considered an essential feature is because it is really just a refinement to the system, and should not be implemented on early prototypes.
Ability to add multiple cameras	Whilst this feature is considered essential in software such as Bluecherry, I do not believe it particularly applies here, this is because the stakeholder is only initially planning to use one camera. The reason why I consider adding this in future iterations is that the stakeholder could easily decide to setup another camera. Whilst they won't have an unfriendly user-experience interacting with both cameras, it won't be cleanly integrated into the application and a 'hacky' work-around will be required (opening multiple instances of the application).
List of added cameras (using internal names)	As an extension to the ability to add multiple cameras, the list of added cameras would provide a nice hub for users to select the camera to view. As with the ability to add multiple cameras, it is not yet an essential feature and will not be implemented in early iterations of the client software at all.
Aesthetically pleasing user inter-	A nice feature would be refined user interface components such as aesthetically pleasing buttons. Once again, this is not at all essential and only serves

Face components	to refine the user experience, hence it will only be considered for future iterations of the software.
Store user credentials locally	Storing user credentials locally would significantly improve the workflow of the stakeholder everyday as when they launched the client software, it would use the saved credentials to try and connect to the security camera server. This means that the user wouldn't have to input their credentials for the camera every time they open the application.
Control video quality and storage duration depending on storage drive capacity	This is a potentially complex feature to add and would require lots of testing to fine-tune the recommended quality and storage duration. Furthermore, it is not essential to the function of the camera at all and realistically has minimal impact on the user experience if they have correctly predicted or overestimated the required storage capacity for their setup. I am only keeping it as a potential feature for future iterations due to the fact that it could have a very positive impact on the user experience if the user under-estimated the capacity of storage required for their desired setting.
Enforce password requirements	Enforcing strict password requirements to increase the strength and resilience of passwords to cyber attacks such as brute force and dictionary attacks would greatly benefit the system and increase its security. Due to the fact that it is not an essential function, the feature will be postponed until a future iteration, but it do believe the feature could help consolidate the integrity of the camera system.
Encrypt data sent across network	This feature would create much more security in the camera system. I may start planning its implementation and ensuring the project is laid out correctly to support the feature, in early iterations (in terms of how the software is designed and abstracted), but it won't be concretely supported until future iterations.

Not To Include

These are the features that I have decided are out of scope of the project and will not be included.

Feature	Explanation & Justification
Motion detection	Motion detection was a feature that I was strongly considering adding to the list of features for future iterations, it wouldn't send the user notifications, but instead just write a log of times and dates that the camera detected motion. However, due to the time constraints of this project, I believe it would be more plausible to not try and implement the feature, and instead ensure higher quality implementation of previous features instead of rushing the code – especially because the stakeholder hasn't expressed a direct interest in this feature despite the fact it may be useful.
Alarms	As mentioned in my analysis of alternate solutions, I do not believe that alarms would be useful for the stakeholder and are therefore out of scope for this project. Also, if I was going to implement audio transmission, I'd need to be sure that the camera hardware actual could output audio, but because the camera will be made up of the user's chosen hardware components, I cannot verify that this is the case
Audio transmission	Implementing audio transmission means I would require that either the computer the stakeholder is viewing the camera from to have a microphone or

	the camera hardware include a microphone, neither of which I want to have as requirements. I also concluded in the analysis of similar solutions that audio transmission would not benefit my stakeholder or the average user of my system.
Remote enable/disable	Allowing for the camera to be remotely enabled or disabled increases the attack surface of the camera more than necessary and would be a central point of attack for any malicious actors; the stakeholder has expressed no interest in the feature and it will not be implemented.
View of multiple cameras at once	As previously discussed in my analysis of the Bluecherry software, adding the ability to view multiple cameras in the main viewport at the same time will require complex algorithms to determine where each camera view should be placed and the size of the view. This feature would be quite difficult to implement and require lots of development time, which I do not have that much of. Combine this with the fact that the stakeholder won't likely need the feature as they only plan on initially using one camera, means that there is little-to-no benefit of implementing it.
Event log	The event log comes alongside features such as motion detection, if I don't implement motion detection then there's not reason to implement an event log as I'll not be able to distinguish what the events the camera sees actually are anyway - at least not without the use of artificial intelligence, but that is far outside the scope of this project.

Development Methodologies

The software development lifecycle defines splitting up the software development process into various stages:

- Feasibility – whether the problem is solvable.
- Requirements – working out what the solution needs to do.
- Analysis and Design – working out how the solution needs to fulfil its requirements.
- Implementation – programming the solution.
- Testing – checking the solution works.
- Deploying – installed the solution in the target environment.
- Evaluation – checking that the delivered solution effectively meets the needs of the user.
- Maintenance – ensuring the solution continues to function after deployment, by way of improvements, patches and updates.

Software development methodologies are ways of approaching the software development lifecycle. I'll consider four software development methodologies for use in this project:

- The Waterfall Method
- The Spiral Model
- Rapid Application Development

- Agile Methodologies & Extreme Programming

The Waterfall Method

The waterfall methodology has concrete phases with well defined start and end points. Each phase has identifiable deliverables and feeds into the following phase, hence the ‘waterfall’ concept.

Here is a diagram showing a visualisation of the methodology:

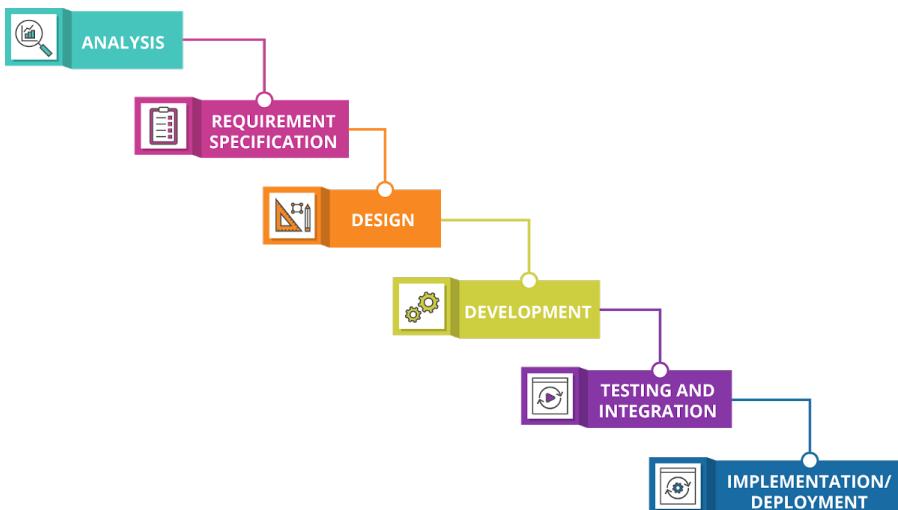


Figure 10: waterfall methodology diagram

Advantages	Disadvantages	Ideal Use Case
<ul style="list-style-type: none"> - Easy to implement - Easy to manage 	<ul style="list-style-type: none"> - Requirements need to be concretely defined at the start of the project. - User has very little influence on the project during the process – end product could be very different from how the stakeholder initially envisioned the solution. - If a developer wishes to amend work done in a previous section, they must adjust work done in all the following stages too – can take significant time. 	A situation where the project requirements are incredibly clear and concrete.

The Spiral Model

The spiral model is a very risk-focused development methodology, and involves hiring risk assessors to evaluate risks involved in the project. The model has four stages:

- Identify
- Design
- Construct (Develop)
- Evaluate

Here is a diagram showing a visualisation of the methodology:

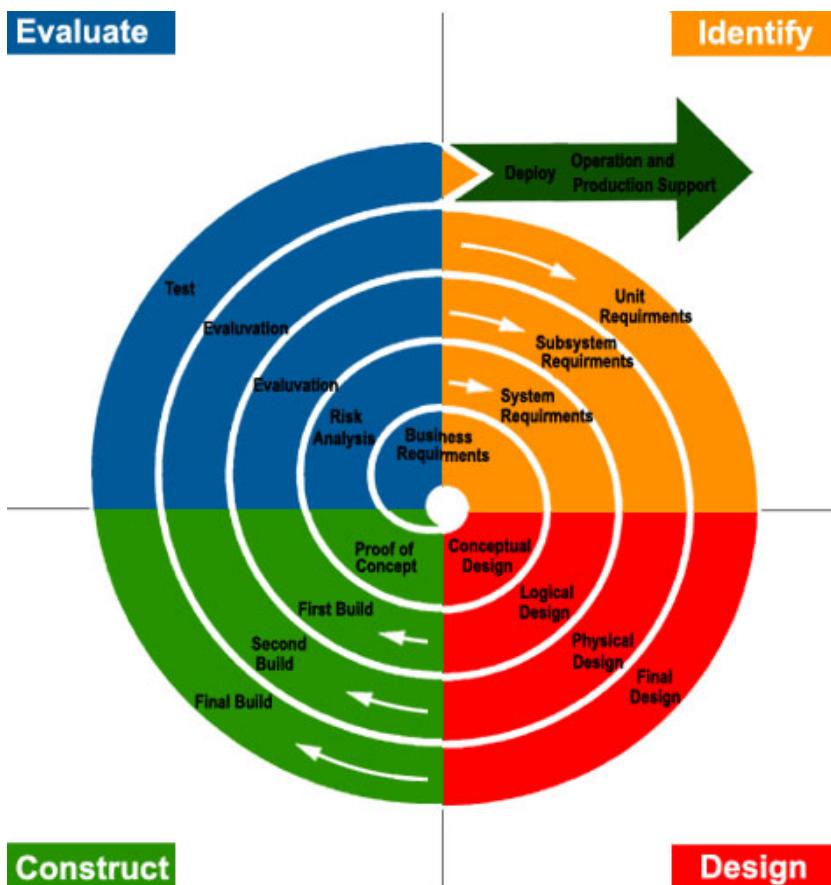


Figure 11: spiral methodology diagram

Advantages	Disadvantages	Ideal Use Case
<ul style="list-style-type: none"> - Recognises that risks are at the heart of many large-scale projects. - Deals with risks as they arise, before they become major. 	<ul style="list-style-type: none"> - Risk assessment is a very specialised job and hiring risk assessors can be very expensive. 	Large company working on a large project with lots of risk involved.

Rapid Application Development

Rapid application development is a user-focused methodology, which rapidly builds iterative prototypes and relies on constant user-evaluation to improve them.

Here is a diagram showing a visualisation of the methodology:

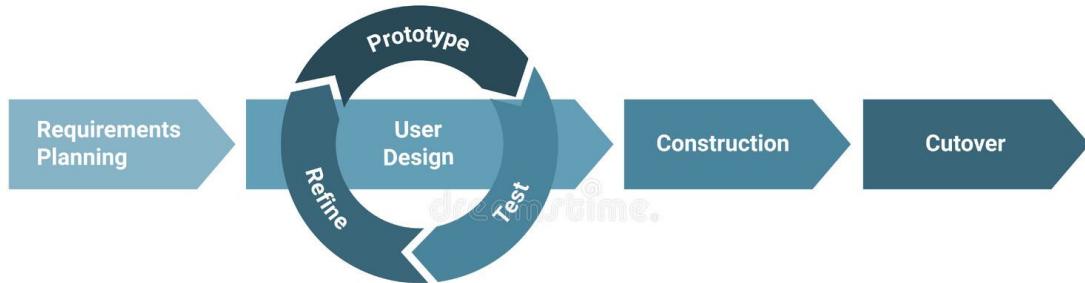


Figure 12: rapid application development methodology diagram

Advantages	Disadvantages	Ideal Use Case
<ul style="list-style-type: none"> - Flexible – allows for easy changes between prototypes. - Rapid – focuses on delivering prototypes quickly. 	<ul style="list-style-type: none"> - Does not scale well – code can become inefficient due to frequent edits and changes. - Requires the stakeholder/user to be frequently available to evaluate prototypes. 	A situation where user requirements aren't concretely defined at the start of the process.

Agile Methodologies & Extreme Programming

Extreme programming in combination with agile methodologies, can be considered a framework for developing software. It defines many programming techniques to produce high code quality and promote developers' quality of life. Its common values are:

- Simplicity
- Communication
- Feedback
- Courage
- Respect

The framework encourages regular, small, iterative software releases.

Advantages	Disadvantages	Ideal Use Case
<ul style="list-style-type: none"> - Produces high quality of code. - Improves developers' quality of life. 	<ul style="list-style-type: none"> - Requires team to work in close collaboration. 	A situation involving a team which is collaborating on a software product at a company.

Most Suitable Methodology

Methodology	Suitability
Waterfall	In my project, the requirements are not concretely defined, and development with the waterfall methodology could easily lead to a solution which doesn't meet the needs to the stakeholder. Furthermore, the additional time required to adjust previous sections, combined with the time con-

	straints of this project, make it an unsuitable candidate.
Spiral	This project is not at all large scale and has no financial backing, therefore hiring risk assessors to evaluate the risks of the project is both unnecessary and unfeasible. The spiral model is not suitable for this project.
Rapid Application Development	Flexibility is quite important to this project as the stakeholder's requirements are not concrete or thorough. The project will not have to scale massively and therefore isn't prone to much inefficiency between editions. I also have frequent access to the stakeholder. Due to the previously mentioned reasons, rapid application development is quite a suitable development methodology for this project.
Agile & Extreme Programming	I am the only person working on this project and therefore many of the benefits yielded from extreme programming techniques do not apply to my use case. Furthermore, I do not have the time to carry out many of the quality of life tasks promoted by the framework (such as code refactoring). For these reasons, I do not believe that agile & extreme programming are a suitable combination for this project.

As a result of my analysis on the suitability of four software development methodologies, I have determined that rapid application development is the most suitable and effective methodology for this project. I believe it will help me build a solution which pleases the stakeholder, and doesn't take too long to make.

Solution Limitations

Whilst I have spent lots of time analysing alternate solutions and ensuring I make minimal mistakes in the design of my own solution, it is important to consider the limitations and negative implications of my decisions thus far.

I discovered throughout the problem analysis, that security cameras have to be incredibly reliable. Therefore, one of the limitations of trying to solve the problem of security, is the fact that I am attempting to solve it computationally. It's not uncommon for computers to falter; software can crash if not programmed carefully. If the security camera software had unexpected errors occur, this would be detrimental for the security of the user; I will program the software with extreme care and test everything thoroughly to ensure no code could cause the software to crash or falter.

Another problem which may occur due to the spontaneous nature of electronics, is the connection between the computer system and physical camera device may fail. This could be due to a loose wire or other input-output (IO) issues. There is nothing that can be done on a software level to recover the system at this stage, hence I consider this issue an unsolvable limitation of the system.

Due to a combination of the stakeholder's requirements and my aim to make a better product than those I analysed, I decided that the camera shouldn't connect to the internet at all. However, one of the downsides of this is the fact that the camera will not be able to download automatic updates from the internet, it will need to be updated manually by the user when there are updates. Updates may include security patches, and therefore could be integral to the integrity of the camera system – although automatic updates are not necessarily the 'silver bullet' they seem. This is a fairly seri-

ous limitation to the system. The best thing a user can do to ensure the impact is minimal, is to check for updates and download them themselves when they are available.

I would not consider programming skills a limiting factor for this project due to my extensive experience with the programming language I wish to use, and strong general knowledge in the field. I also won't consider financial aspects a limiting factor of this project, as I have specifically chosen to only make the software for the security camera system and not directly involve hardware in the project.

Time constraints can definitely be considered a limitation of this project. My limited time available to make this project, could easily result in some sections being slightly rushed and will perhaps detract from time that could be spent improving efficiency & organisation of the code base.

Finally, a problem that may arise is an error could occur when trying to save camera footage to the camera system storage (for no particular reason). There's very little that the camera software can do if this occurs, its best hope is to just try again – but that's not something I'll be looking to implement in early iterations of the software, if at all.

Solution Requirements

Software

The software requirements for this project are going to be very minimal as I will attempt to package as many software dependencies as possible into the application installer.

However, for operating systems which I do not supply an installer (although I plan to supply one for Microsoft Windows and Linux), the Java Runtime Environment (JRE) will be a requirement due to my choice of programming language (discussed in the design section). For operating systems where an installer has been provided, I will attempt to package the JRE into the installer.

Hardware

The hardware requirements for this project are as follows:

- Network Switch or Network Router (creates the local area network (LAN))
- Network Interface Card (NIC)
- 1 gigahertz (GHz) (or faster) clock speed
- 2 gigabytes (GB) of RAM
- 30 gigabytes (GB) of storage
- Graphics card with DirectX 9 or later with WDDM 1.0 driver
- Display with at least 800x600 resolution

The requirements relating to networking exist so that the user can connect to the camera and view it remotely, using the client software. The more specific requirements for the computer hardware are to ensure that the application can run smoothly in conjunction with an operating system.

Success Criteria

Criteria	Functional- ity or Usab- ility	Why is it required?	How will it be tested?
1. Easy to install	Usability	To improve user experience during setup.	Tested by the user to get feedback on the installation & setup workflow.
2. Function without internet connection	Functionality	Allows the user to use the software without having to expose it to the internet. Was requested by stakeholder.	The code could be ran in an environment isolated from internet.
3. Don't send data outside the local network	Functionality	The camera may only be connected to one local network without access to others, therefore it should not rely on third party services.	The code could be ran in an environment isolated from internet.
4. Working network communication	Functionality	So that the client software can interact with the security camera remotely.	Unit tests will be used to test functionality on client and server software. They will be manually integration tested and demonstrated working together.
5. Correct serialization and deserialization of network packets	Functionality	So that the data being sent from one computer to another does not get corrupted in any way.	Integration tests will be used with the client and server, as well as unit tests of both applications. A basic packet type will be sent across the network and the result checked for equality.
6. Minimal yet intuitive user interface	Usability	So that the user has a nice experience when using the application.	I will show client application GUI designs to the stakeholder to get feedback on them. I will also get the stakeholder's feedback on the GUI once implemented.
7. Streaming of camera frames over the network	Functionality	So that the client software can view the camera without a physical connection.	I will send a predefined image across the network and check it is the same at the receiving end.
8. Can login to the camera server using credentials	Functionality	So that the user can log in and view a camera.	Unit tests will be used on the server to ensure the account system accepts correct credentials and rejects incorrect credentials.
9. Users can be added and deleted to/ from the camera	Usability	To allow the user/stakeholder flexibility with different accounts.	Unit tests will test the account system on the security camera, the tests will include adding and deleting an account – checking the necessary changes have occurred for each operation.

10. Downloadable video footage	Usability	Allows the stakeholder to view footage of the camera from when they weren't connected to it.	Unit tests will be carried out to ensure that files data is streamed successfully.
11. Cross-platform	Usability	So that the stakeholder can use the software on computers which run on different operating systems.	I will manually test the software on both Microsoft Windows 10 and Linux Mint (based on Ubuntu Linux).
12. Expandable storage	Usability	So that the stakeholder can store lots of video footage on the camera device.	Test the video saving functionality on a variable size hard drive.
13. Server software can access the camera on the machine	Functionality	So that the server will be able to record and stream the camera.	Unit tests on the server software will ensure it can access the camera.
14. Functioning video creation from camera frames	Functionality	So that the user can view recorded camera footage from the past.	Unit tests will involve a collection of images and attempt to encode them into a video on the server software. The test will then attempt to read and decode the video, ensuring it is in the correct format.
15. Save video locally to client computer	Usability	Ensures footage is accessible if camera disconnects from the network.	Unit test to ensure video files are successfully created.
16. Using multithreading for networking	Functionality	Will allow receiving of packets and continuation of application concurrently for both client and server software.	Test application doesn't 'hang' (block execution) when waiting for a packet to be received.

Design

Programming Language

I will be developing the software for this project using the Java programming language. This decision was made because I have approximately three years experience writing software with the language and I also have a professional certification in the language (Oracle Certified Professional Java SE 8 Programmer).

The Java language works by compiling the raw Java source code into intermediary bytecode, this bytecode is then interpreted by the Java Virtual Machine (JVM) which is a part of the Java Runtime Environment (JRE).

Java has several strong features which make it ideal for this project:

- The well-established ecosystem surrounding the language means there is a plethora of tools and libraries that I can use to aid in development.
- Java often has a higher throughput of data and is more efficient than languages such as python which is directly interpreted. This is due to the fact that Java compiles the source code into bytecode, which is then interpreted by the JVM – allowing the compiler to perform many optimisations.
- Java has quite impressive efficiency when it comes to server software. This means that the server software running on the security camera will be more efficient than it would be with many other languages.
- Java software is cross-platform and runs on any system which has the Java Runtime Environment (JRE) installed – the JRE is available for most operating systems. I want all software I make for this project to be cross-platform so Java definitely helps in this regard.

However, Java has a few limitations which make it less ideal for this project:

- Java source code is only compiled into intermediary bytecode, not machine code, therefore it is slower than languages which compile directly to machine code – such as C++ (used by Bluecherry DVR). However, languages such as C++ are more low level than Java (which is considered more high-level), and hence more difficult to learn and work with.
- It is quite an old language and doesn't boast as many modern features as other, newer programming languages. Languages such as Kotlin have been created to be fully interoperable with Java and provide more modern features, however it is syntactically very different to Java.
- The most widely-used Java library for graphical user interfaces (which is built into the standard library) is Java Swing. Swing uses an outdated threading model which restricts GUI operations to only being executed from a single thread and can just generally be quite clunky to develop visual applications with.

Its successor (Java FX) fixed many of the issues with Java Swing however is not part of the standard library and therefore has a lot less support in the Java ecosystem. Furthermore, it is considered more difficult to use than Java Swing. Due to the fact that there aren't that many limitations of Swing that are likely to directly affect this project, I will use Swing as it is a part of the standard library and I already have experience using it – although I acknowledge that an approach using Java FX would likely yield a more elegant user interface and cleaner application code.

Decomposition

Decomposition is the process of breaking down a large problem into a series of smaller sub-problems. These sub-problems are then easier to solve and approach than the big problem they make up. I am going to decompose the problem of a security camera system, of which I have already laid out a success criteria in the analysis section, in order to create an easily-solvable and manageable series of sub-problems.

Brief Overview

The project consists of two major pieces of software, the client software and the server software. The client software will be used on the stakeholder's computer to view the camera, and the server software will be on the same computer as the camera.

The modules in my top-down decomposition diagrams are heavily influenced by the success criteria I laid out at the end of the Analysis section. This ensures that I meet the needs and requirements of the stakeholder.

I have structured all software in such a way which allows for me to de-couple components/modules and make developments or enhancements to them without negatively impacting the overall system. This also allows me to not have to develop each system side-by-side, instead I can develop one system at a time and seamlessly integrate them together when they are finished. Testing the system also becomes easier as each module can be subject to unit tests without other modules required to be functional or complete.

Server Software

The server software will be made up of three main components:

- Networking Server – accepts client connections, processes packets and constantly streams the camera feed (taken from the Camera Viewer) to clients.
- Account System – manages and authenticates users.
- Camera Viewer – accesses the camera, captures camera frames and creates videos from the feed to store locally.

Here is a top-down diagram representing the components of the server software.

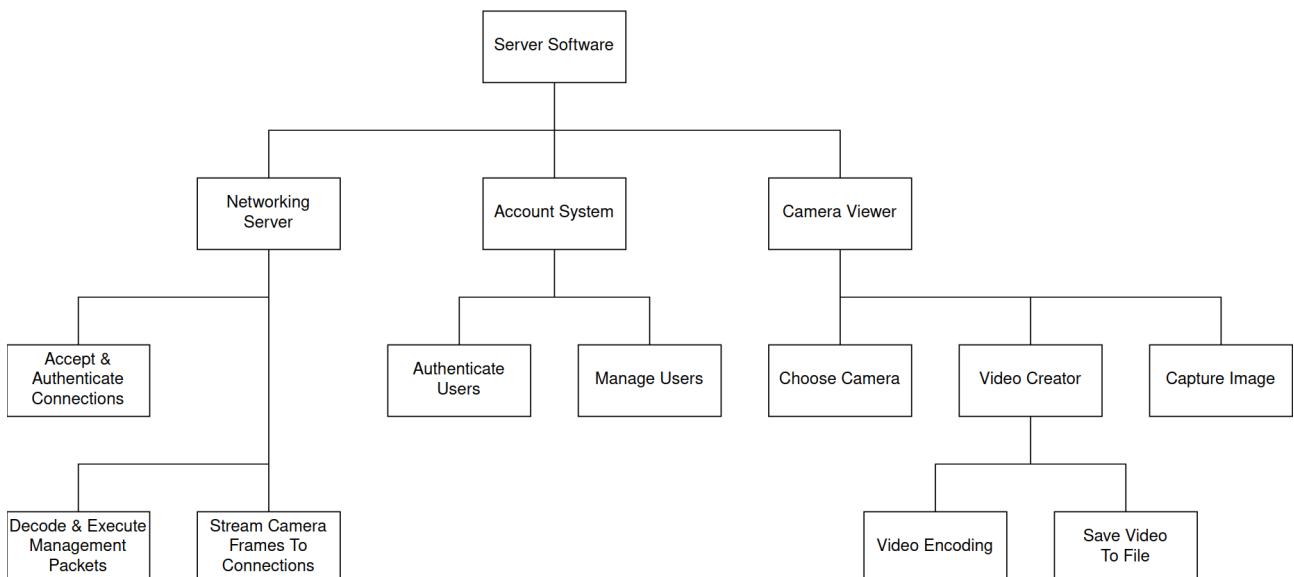


Figure 13: server software top-down decomposition diagram

One module of the server software is the networking server module. I have decided that this is a module because the client software will need to connect to a hosted server using sockets, and the networking server module is a perfect candidate for this. There are multiple operations and man-

agement systems that need to take place under the networking server module, hence I have added three submodules. Accepting and authenticating connections is the first main submodule of the server: the server must continually accept connections from different client software instances and check the client's transferred credentials using the Account System to determine if they are a valid user. From there, all authenticated clients can continuously receive the camera feed.

I decided to abstract the authentication system away from the networking server module as this would give the user more flexibility in how they used the system, it also creates an individual reusable module which can be employed in other programming projects. The account system is responsible for checking login validating login credentials and managing users, therefore I consider these elements the submodules of the authentication system module.

The media content to stream to the clients will come from the camera, and if I implement the camera access and viewing in a separate, de-coupled module, the code becomes more adaptable without affecting other modules of the server software. The main functions of the camera access & viewing module are to choose the correct camera device to stream, capture images of the chosen camera stream (for the networking server to send), and encode & save video content. I expanded these functions into submodules of the camera viewing module as there was a clear logical separation in the functionality of the code.

Client Software

The client software will also be made up of three major components:

- Networking Client – connects to security camera server software.
- GUI – the display for the user to interact with.
- Control system – handles inputs and requests from the GUI and communicates with the networking client.

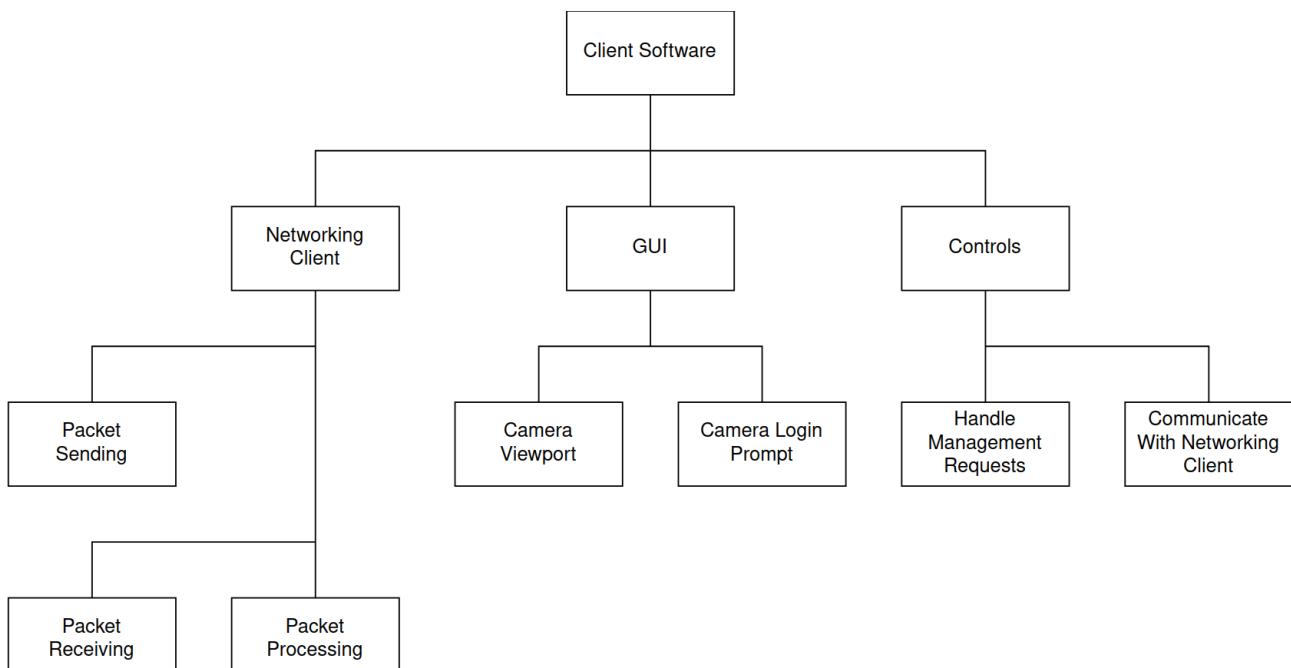


Figure 14: client software top-down decomposition diagram

The networking client module of the client software is responsible for managing the connection between the client software and server software. It provides an application programming interface (API) to the control system, in the form of a packet sending submodule – this will allow the control system module to send data through the network client to the server software. Internally, the networking client is also responsible for continually receiving packets/data from the server software, this will require its own concurrent thread and algorithm, making it a perfect candidate for a submodule. Finally, all the packets received by the networking client need to be processed and acted on, this is where the packet processing submodule comes in. The packet processing submodule will decode the packet/data it receives, determine its type, and how to correctly act upon it.

I am to keep the graphical user interface as de-coupled as possible from the controlling code of the client software, this is due to the fact that it is very common for UI elements to change and the layout of the application to evolve over development iterations. If I de-couple other components of the client software from the GUI, I ensure that no functionality is lost between design iterations due to UI changes. For this reason, I consider GUI to be a module of the client software. The GUI will contain multiple components such as the main camera viewport and the camera login prompt. I consider these to be the most significant components of the GUI and have therefore allocated them as separate submodules of the GUI module in order to emphasise the focus that needs to be placed on them.

The control module is intended to provide a nice layer of abstraction, between the graphical user interface of the client application, and the code which understands the user's intentions and transmits them to the camera server. I talked about the justification behind de-coupling these components in the previous paragraph on the GUI module. The control module's purpose is to communicate with other modules in the program and use abstraction techniques such as information hiding to provide an intuitive application programming interface (API) for any developers to interact with. The reason why I called it the control module, is because it literally controls communication between other modules in the client application and coordinates the flow of execution – creating a non-linear user experience. The submodule 'handle management requests' refers to the API it creates for the GUI, as it will receive requests to manage the camera from the GUI's input events. Also, the submodule 'communicate with networking client', simply refers to how the control module will use data it has gathered (or more specifically requests made by the GUI) to coordinate information and packet sending across the network (to the security camera server) using the networking client module.

Recognising Shared Functionality

When decomposing the client and server software, I noticed that both contained a networking module. Whilst they both have very different functionality, I am quite certain that I will end up writing duplicate code when programming each of the networking modules.

Whilst the decomposition diagram of the server software doesn't explicitly state it, the networking server will contain code to receive, send and process packets – much like the client software's networking client. Instead of rewriting the same code, I will create a common module (shared by both the client software & server software) which aims to provide an abstraction over the standard networking libraries available – implementing functionality to receive, send and process packets in a standardised and consistent manner. I am doing this as it will save time during the development process, and a successfully abstracted networking module can be used with future projects.

Throughout the design of this shared module, I may refer to it as the ‘common networking module’. ‘Common’ because it is common/shared in both the client software and server software, ‘networking’ because it aims remove duplicate networking code, and ‘module’ as it is a re-usable, de-coupled, abstracted component of the project. I may also refer to it as the ‘connection framework’ or ‘connection library’ because it will be based around the idea that all communication channels can be abstracted to a ‘connection’.

System Design

Server Software

Account System

The account system is required to store user credentials and validate login attempts using the stored data. It also needs to be able to manage this stored account data, including deleting it and creating more – these are requirements which were noted in the server software decomposition diagram and project success criteria.

Considering System Separation

Firstly, I need to consider what kind of relationship the account system should have with the code in the server software. Should it be closely attached and built as a part of the code in the server software, or should it be built as a completely de-coupled piece of code running its own public API? I first considered how it would function in both of these situations.

If the code was completely detached from the server software, I would likely achieve it in the form of a standard web-based API as that would be the quickest and easiest to create, whilst leaving lots of flexibility. The system would be interacted with in the form of Hyper-Text Transfer-Protocol (HTTP) requests, and response payloads in a JavaScript Object Notation (JSON) format would be returned by the same protocol. This approach to API development is quite common and many libraries/tools exist to create this API in a very efficient manner. This approach is often referred to as a REST API or RESTful API – REST standing for representational state transfer.

If I chose the other approach, and had the code for account management entirely in the server software, I would just use typical object-oriented programming paradigms for the system to be interacted with – such as the calling of methods and creation of objects.

There are different advantages & disadvantages of each approach to developing the account system. For the web-based API, one major benefit is that the system gains lots of flexibility in the sense that other applications can use the developed account system, and it can be adapted easily without affecting the public-facing interface of the application at all. However, it also poses the drawback that you face extra computational load on the system, as another server will have to run in order to provide the REST API. It is worth remembering that this project is designed to be able to run on hardware with minimal computation capabilities, therefore this extra computational requirement paints the REST API in an unfavourable light.

The less abstracted approach of having the account system built into the server software does boast better computational performance, which is more ideal for my use case. Furthermore, it may take even less time to develop than the HTTP REST API. In reality, there is not going to be that much code making up either system, so it's actually quite hard to justify creating an entirely separ-

ate program just to manage the user accounts – especially when I don't expect it to come under much computational load (in the form of user requests). Furthermore, the flexibility gained by the HTTP REST API isn't actually particularly useful for my project, as other applications using the account system is very much out of the scope of this project and I don't expect the stakeholder will every need this capability in the future.

After evaluating the advantages & disadvantages of both systems, a clear narrative has been established which heavily favours writing all the account system code as part of the server software, and not separating it into a separate program that could be used by multiple pieces of software.

Considering Account Storage Solutions

The two most popular and applicable account storage solutions, are files and databases. Both have potential to be used in this project for storing accounts and therefore I ought to consider the advantages & disadvantages of both, as well as how well suited they are to the problem I'm solving.

Firstly however, I would just like to take a moment to justify why I'm not just going to be storing the account data in the memory of the server software using a list or an array: if data was not stored on any non-volatile physical storage device, it would not persist. This means that if you restarted the security camera or the server software, then all account data would be lost if it was only stored in the memory of the server software. Therefore, files and databases, which both make use of persist storage mediums, are the only viable options for account storage.

Storing data in files, is much easier and more convenient than storing data in databases. Files are ideal for data storage in initial prototypes of applications, as they allow for a demonstration of data-persistence whilst creating little overhead, and a simplified development process. However, they become less easy to manage as the dataset grows, and may not always meet the stakeholder's needs – for example they can make it difficult to manage access rights for the stored data.

Databases are generally used when the amount of data has grown significantly. They become increasingly easy to manage as they grow due to the rigidly enforced structure of the database, through its use of tables which each have columns that accept certain data types and can have different data requirements. However, their application to large datasets doesn't really help this project as I'm only planning to store account data, and the stakeholder will only require a few accounts at most. Designing the system to use a fully-fledged database straight away is not justifiable and will only be considered later into the development process for sake of benefits provided by a database management system (DBMS) such as user access rights.

However, there is a middle-ground in the form of a flat-file database. A flat-file database is a database that is stored in a single file and is interacted with by the use of Structured Query Language (SQL), just like a normal database. However, with a flat file database there is none of the computational overhead of using a fully-fledged DBMS.

Another approach I could use, is to distribute account data between multiple files by storing the data for one user account in each file. This could help ensure that no dangerous bugs in the program code leak information of other users to the currently logged-in user, as that data would be in a completely separate file. Furthermore, minimal data loss will be suffered if the file becomes corrupted as only the user account who's file was corrupted is lost; if I used a flat-file database and the file became corrupted, then all user accounts data would be lost.

After evaluating my options, I believe using multiple files (one for each user account) is the safest and most effective way to store user accounts. It is very plausible that, further into the development process, I may decide to change the account storage solution to a flat-file database for sake of its rigidity and ease-of-interaction through the use of SQL. But for now, I'll design the storage of user accounts to be distributed across multiple files.

Considering Account Sessions

Usually, servers using protocols such as HTTP are so high-level and abstracted, that a constant connection cannot be guaranteed between the client and the server, resulting in a request-response architecture where the client must prove their identity with every request they make to the server – through the use of session tokens or JSON Web Tokens (JWTs).

Session tokens and JWTs are sent from the client as additional arguments to any request they make to the server, and the server will verify that this token is valid and associated with a user account/session. What session tokens and JWTs actually are, is fairly irrelevant for the point I'm making – if the reader is interested I would direct them to my blog post[1] on the topic. All that needs to be understood, is that on a high level, this creates the illusion of a user session.

However, due to the fact that I'm writing the networking code for the software myself (in the form of the shared networking module), I have much more granular control over how a session is stored and represented in a connection. The protocol I am going to be using for networking is TCP, which is a protocol that keeps the connection alive between the client and the server for as long as they are communicating. Therefore, I can assume that the TCP connection represents the connection session and I will associate user accounts with their corresponding TCP connection. When the user disconnects from the server, they will have to authenticate again. This approach and control removes the need for session tokens or JWTs.

Class Designs

The UML diagrams for the account system submodule represent a useful abstraction in the sense that they don't include the complexity of storing the accounts in files. I will address the complexity in the pseudocode, but the UML diagrams will only represent the necessary classes, attributes and methods of the account system in order to avoid representing implementation specifics.

Firstly, I will use a `User` class to represent user accounts on the system. This way, I can effectively utilise objects throughout the code to communicate information about users without having to list lots of parameters in method definitions; for example, I can return a `User` object from a method to encapsulate multiple pieces of data.

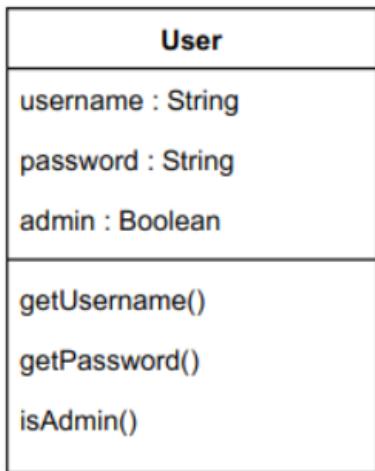


Figure 15: 'User' class UML diagram

As can be clearly seen, the 'User' class is quite simple and exists purely as a class to provide convenience to the developer in the form of an immutable class. The only part of the class which may come as a surprise, is the 'admin' attribute and 'isAdmin()' method. These exists because the stakeholder will need to be able to manage accounts (including creating & deleting), but accounts created by the stakeholder should not have this same ability; hence the 'admin' attribute represents the permission level of the user accounts and the 'isAdmin()' method is just the getter method for the 'admin' attribute. If the attribute's value is 'false' then the user is not an administrator, if it is 'true' then the user is an administrator and can manage other accounts.

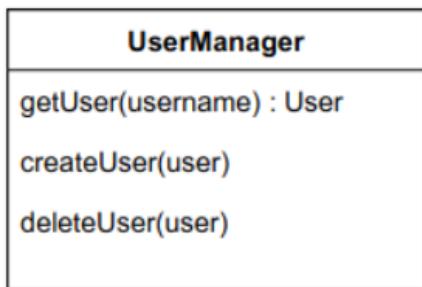


Figure 16: 'UserManager' class UML diagram

Shown above is the UML diagram for the 'UserManager' class; it details the basic operations of user retrieval by username, user creation and user deletion. One method which I actively chose not to include in the class was the 'isValidCredentials(username, password)' method which I was initially considering adding. The reason I didn't add the method is that it would promote poor quality code.

Consider the typical implementation for the 'isValidCredentials' method:

```

FUNCTION isValidCredentials(username, password)
    //get user object from username
  
```

```

user = getUser(username)
//return Boolean value denoting whether the given password is the same as
the password of the retrieved user object
RETURN password == user.getPassword()
END FUNCTION

```

Now consider how the `isValidCredentials` function will typically be used (in some login code):

```

username = connection.readUsername()
password = connection.readPassword()
//check valid credentials
IF userManager.isValidCredentials(username, password) THEN
    //correct login credentials
    //get user after credentials confirmed
    user = userManager.getUser(username)
    connection.setUser(user)
ELSE
    //invalid login credentials
END IF

```

As you can see, the method `UserManager#getUser(username)` is called twice by the code – even though it really only needs to be called once. To prove this, let's consider that the method `UserManager#isValidCredentials(username, password)` didn't exist, then the typical login code would look like the following:

```

username = connection.readUsername()
//get user object from username
user = userManager.getUser(username)
//check if received password is equal to password of user object
IF connection.readPassword() == user.getPassword() THEN
    //correct login credentials
    connection.setUser(user)
ELSE
    //invalid login credentials
END IF

```

As you can see, without the `UserManager#isValidCredentials` method, I can successfully discourage the developer from creating duplicate code in the scenario when the method would have been most commonly called; `UserManager#getUser` is now only called once, and this code has the same desired affect as the code that involved duplicate method calls.

This improvement becomes especially noticeable when the `UserManager#getUser` function has a non-constant time complexity according to the Big O notation, or consistently takes a long time to run (relative to the other code).

Pseudocode

Below is the pseudocode for the `User` class, which is an exact interpretation of the UML diagram for the class:

```

//User class definition
CLASS User
    //private attributes
    PRIVATE username
    PRIVATE password
    PRIVATE admin

    //constructor with parameters matching to private attributes
    PUBLIC PROCEDURE new(givenUsername, givenPassword, adminStatus)
        username = givenUsername
        password = givenPassword
        admin = adminStatus
    END PROCEDURE

    //username attribute getter
    PUBLIC FUNCTION getUsername()
        RETURN username
    END FUNCTION

    //password attribute getter
    PUBLIC FUNCTION getPassword()
        RETURN password
    END FUNCTION

    //admin attribute getter
    PUBLIC FUNCTION isAdmin()
        RETURN admin
    END FUNCTION
END CLASS

```

And here is the default implementation of the `UserManager` class in pseudocode form (using one file for each account):

```

//UserManager class definition
CLASS UserManager
    PUBLIC PROCEDURE deleteUser(user)
        //get list of accounts
        accountFiles = LIST_FILES("accounts")
        //sort list of accounts by the account username
        sortedAccountFiles = sortByUsername(accountFiles)
        //return user found by searching sorted files with binary search
        userFile = searchSortedUserFiles(sortedAccountFiles, user.getUsername())
        //if user file found then delete it
        IF userFile != NULL THEN
            userFile.delete()
        END IF
    END PROCEDURE

    //procedure to create the file for a user in the 'accounts' directory
    PUBLIC PROCEDURE createUser(user)
        //create file with name being user's username in 'accounts' directory
        file = OPENWRITE("accounts/" + user.getUsername())

```

```

//write user attributes to file
file.writeLine(user.getUsername())
file.writeLine(user.getPassword())
file.writeLine(user.isAdmin())
//ensure close file to avoid causing IO errors
file.close()
END PROCEDURE

//function to retrieve user object from username
PUBLIC FUNCTION getUser(username)
    //get list of accounts
    accountFiles = LIST_FILES("accounts")
    //sort list of accounts by the account username
    sortedAccountFiles = SORT_BY_USERNAME(accountFiles)
    //return user found by searching sorted files with binary search
    userFile = searchSortedUserFiles(sortedAccountFiles, username)
    IF userFile == NULL THEN
        RETURN NULL
    END IF
    RETURN fileToUser(userFile)
END FUNCTION

//function to sort the account files by username (the name of the file)
//using a merge sort
PRIVATE FUNCTION sortByUsername(accountFiles)
    //recursion base case to stop splitting array when it has only one
    element left
    IF accountFiles.length <= 1 THEN
        RETURN accountFiles
    END IF

    //split accountFiles array down middle, into left and right array
    midpoint = accountFiles.length // 2
    array leftAccountFiles[midpoint]
    FOR index = 0 TO midpoint - 1
        leftAccountFiles[index] = accountFiles[index]
    END FOR
    array rightAccountFiles[accountFiles.length - midpoint]
    FOR index = midpoint TO accountFiles.length - 1
        rightAccountFiles[index] = accountFiles[index]
    END FOR

    //recursively sort arrays
    leftAccountFiles = sortByUsername(leftAccountFiles)
    rightAccountFiles = sortByUsername(rightAccountFiles)

    //declare variables to use with iteration
    leftIndex = 0
    rightIndex = 0
    sortedIndex = 0
    //create array for sorted files
    array sortedAccountFiles[accountFiles.length]
    //iterate whilst there are elements in both arrays
    WHILE leftIndex < leftAccountFiles.length AND rightIndex < rightAc-
    countFiles.length
        //put the username which is lower alphabetically first

```

```

        IF leftAccountFiles[leftIndex] < rightAccountFiles[rightIndex]
THEN
    //if username in left array is lower alphabetically, add it
to sortedAccountFiles and increment leftIndex
    sortedAccountFiles[sortedIndex] = leftAccountFiles[leftIn-
dex]
    leftIndex = leftIndex + 1
ELSE
    //if username in right array is lower alphabetically, add
it to sortedAccountFiles and increment rightIndex
    sortedAccountFiles[sortedIndex] =
rightAccountFiles[rightIndex]
    rightIndex = rightIndex + 1
END IF
//increment current index to insert elements in sorted array
sortedIndex = sortedIndex + 1
END WHILE

//add any remaining elements from left array into sorted array
//will only run when the right array is empty
WHILE leftIndex < leftAccountFiles.length
    sortedAccountFiles[sortedIndex] = leftAccountFiles[leftIndex]
    leftIndex = leftIndex + 1
    sortedIndex = sortedIndex + 1
END FOR

//add any remaining elements from right array into sorted array
//will only run when the left array is empty
WHILE rightIndex < rightAccountFiles.length
    sortedAccountFiles[sortedIndex] = rightAccountFiles[rightIndex]
    rightIndex = rightIndex + 1
    sortedIndex = sortedIndex + 1
END FOR

//return sorted files
RETURN sortedAccountFiles

END FUNCTION

//function to search the sorted account files using a binary search
PRIVATE FUNCTION searchSortedUserFiles(sortedAccountFiles, username)
    //assign right and left pointers which allow for divide and conquer
search with binary search
    left = 0
    right = sortedAccountFiles.length
    //loop until User object found or not in account list
    WHILE left < right
        //calculate point in middle of left and right pointer
        midpoint = (left + right) // 2
        //convert file at midpoint to User object
        user = fileToUser(sortedAccountFiles[midpoint])
        //if User object is the one at midpoint, return file
        IF user.getUsername() == username THEN
            RETURN sortedAccountFiles[midpoint]
        //move right pointer to below midpoint if the User object at
midpoint has username which is higher alphabetically than the username that
we are searching for
    
```

```

        ELSE IF user.getUsername() > username THEN
            right = midpoint - 1
            //move left pointer to above midpoint if the User object at mid-
            point has username which is lower alphabetically than the username that we
            are searching for
        ELSE
            left = midpoint + 1
        END IF
    END WHILE
    RETURN NULL
END FUNCTION

//private function to convert account file into a User object
PRIVATE FUNCTION fileToUser(fileName)
    //open account file
    file = OPENREAD("accounts/" + fileName)
    //read attributes from file
    username = file.readLine()
    password = file.readLine()
    adminStatus = file.readLine()
    //ensure close file to avoid causing IO errors
    file.close()
    //create User object from attributes read from file
    RETURN new User(username, password, adminStatus)
END FUNCTION
END CLASS

```

The `UserManager` class loosely follows the singleton design pattern, in the sense that its function does not revolve around the use of objects. The methods are intended to be accessed and function the same regardless of any object they are acting on – many programming languages would use the `static` keyword when describing the signatures of these methods.

The `UserManager` class also has two additional private methods which weren't included in the UML diagram of the class, these exist to remove duplicate code and improve the overall readability of the code.

I have also implemented the binary search algorithm in the pseudocode to locate the account file for a user through the use of divide and conquer. This is implemented in the `UserManager#searchSortedUserFiles` method, which searches through the provided list of user files to find one with the same username as the one being searched for. A precondition of a binary search, is that the data must be sorted before it can be searched; therefore the `sortedAccountFiles` parameter is assumed to be sorted – hence the naming of it to indicate to developers the requirement.

A binary search was implemented due to its logarithmic time complexity according to the Big O notation: $O(\log n)$. This means that as the list of users grows, not that it is expected to grow massively, the algorithm continues to run very fast, with very few elements of the dataset needing to be checked in order to determine the user's account file in the provided dataset of files. This is much better than the alternative (a linear search), which would have a linear time complexity, meaning that the runtime of the algorithm finding the account file for a user would be directly pro-

portional to the number of user's on the system – which would result in a noticeable decrease in the efficiency of the 'UserManager' class as the number of users increases.

The sorting precondition for the 'UserManager#searchSortedUserFiles` method resulted in my implementation of the sorting method 'UserManager#sortByUsername'. This method implements a merge sort, which has a linear-logarithmic time complexity according to the Big O notation: $O(n \log n)$. I decided to use the merge sort because its time complexity is much more ideal than a linear one. The logarithmic part of the time complexity comes from the fact that the algorithm uses divide-and-conquer to split up the dataset – in my implementation recursion is used to achieve this. The linear element of the time complexity comes from the part of the algorithm which rejoins all the data together by compares the values.

When translating the pseudocode for the 'UserManager#sortByUsername` method, the developer should consider the fact that I used the less-than comparison operator ('<') to determine alphabetic order. Not all languages support this, therefore the developer should use an appropriate comparison operation/symbol which supports alphabetic ordering.

Camera Viewer

The camera viewer is going to be implemented directly into the server software as a module of code because other code in the server software needs to be able to access the camera and capture images of it, which can then be sent to connected client software instances.

First Iteration

The code which views the camera and captures images of it, should be in a class which successfully describes its purposes. As hinted by my previous top-down decomposition diagram, I believe the functions of creating a video and capturing the camera frames are very different, and should be separated through the use of abstraction and de-coupling of code modules. Using object oriented programming, I will separate these functionalities into two different classes, 'CameraViewer' and 'VideoEncoder'.

Below is a UML diagram, showing my first design iteration of the 'CameraViewer' and 'VideoEncoder` classes:

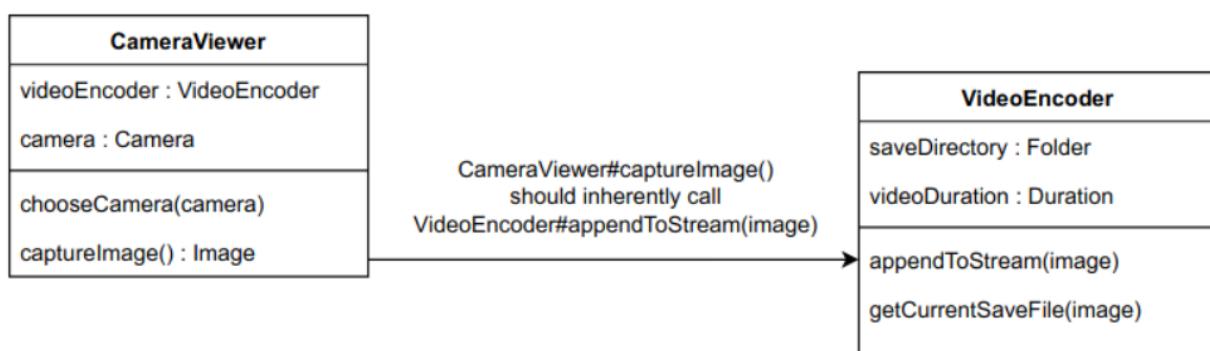


Figure 17: 'CameraViewer' & 'VideoEncoder' classes UML diagram - first iteration

I initially envisioned the 'CameraViewer` class following the singleton design pattern, which is where there is only ever one object created from the 'CameraViewer` class. The singleton object's

state would mutable and the camera that the object uses could be changed through the use of ‘setter’ methods – in this case the `chooseCamera(camera)` method.

However, after finishing this UML diagram I considered the implications of mutability, on the reliability of the `CameraViewer` class. The fact that the `camera` attribute of the singleton object would be mutable (as required to have the `chooseCamera(camera)` method), would make the program much harder to debug as the `camera` attribute of the object could change at any moment, making it more difficult to trace the value of the attribute and discover logical errors in the code. This design would also make it easier for me to accidentally introduce bugs into the final solution; for example, the value of the `camera` attribute may indirectly change due to the execution of other code, leading to unexpected errors.

Another reason why I was sceptical of the effectiveness of my initial design of the camera viewing system, was because `CameraViewer#captureImage()` should inherently call `VideoEncoder#appendToStream(image)`. This means that if I ever desired to write another `CameraViewer` class in the future (which derived functionality from the default implementation), I would have to remember to follow this same principle. Furthermore, any other developers who could add to the code in the future may not be aware of this requirement – even if it was properly documented. Also, the method name `captureImage` describes a very simple task, not related to `VideoEncoder#appendToStream(image)` in any way. Therefore, it would be a more logical design choice to leave calling that to a separate part of the program, and instead ensure the `captureImage()` method did only what is described by its name.

Following these considerations, I sought to redesign the `CameraViewer` class and its relationship with the `VideoEncoder` class as a second design iteration. I required the design to make the `CameraViewer` class immutable, remove the singleton design pattern, and remove the required inherent method call in the `CameraViewer#captureImage()` method implementation.

Second Iteration

I came up with the following UML diagram for my second design iteration of the camera system classes:

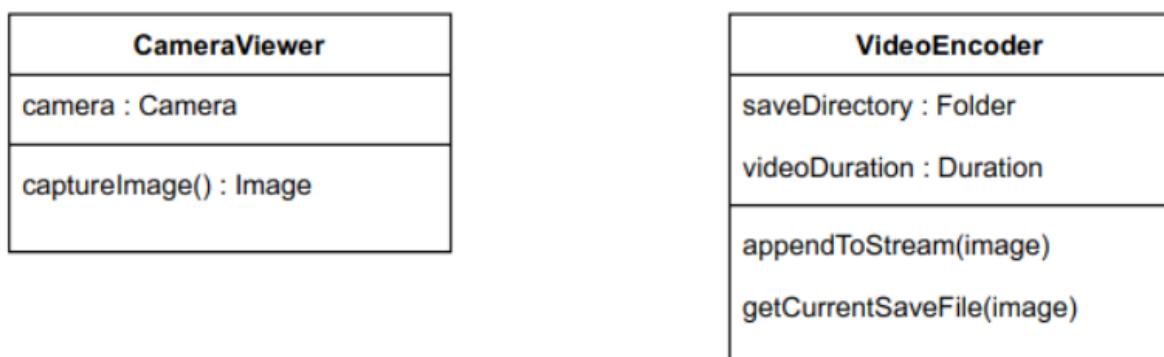


Figure 18: 'CameraViewer' & 'VideoEncoder' classes UML diagram - second iteration

I removed some features from the `CameraViewer` class so that it only carries out the operations which the class describes. The camera will be chosen upon instantiation of the `CameraViewer` class in the constructor (instantiation is the process of creating an object from a class). If the user

wishes to change the camera used whilst the program is running, they will have change the active `CameraViewer` in another object to be a new instance of `CameraViewer`. This decision was made to support the immutability of the `CameraViewer` class and to ensure the class doesn't try to achieve more than it should (or carry out operations out of the scope of the class).

I have made no changes to the `VideoEncoder` class design as I believe it still effectively accomplishes the goals the class is designed to. It is also an immutable class as the `saveDirectory` and `videoDuration` attributes will be initialised during the instantiation of `VideoEncoder` objects – through the constructor. This is ideal for debugging and writing clean, manageable code.

Other code in the program will now be responsible for appending the captured image from the `CameraViewer` to the video stream in the `VideoEncoder` class. The removal of the inherent method call requirement allows for effective, reusable program modules – ideal to expand upon and create variations of.

After confirming I was happy with the new and improved UML diagram, I created pseudocode to demonstrate how the CameraViewer and VideoEncoder classes may be implemented. This also helps people reading the design to understand its implementation in a non-language-specific way.

```
//CameraViewer class definition
CLASS CameraViewer
    //encapsulated camera attribute
    PRIVATE camera

    //constructor taking cameraToView as a parameter
    PUBLIC PROCEDURE new(cameraToView)
        //set the camera attribute to have the value of the constructor
        //parameter
        camera = cameraToView
    END PROCEDURE

    //function to capture image from the camera
    PUBLIC FUNCTION captureImage()
        //open camera so that image can be captured
        camera.open()
        //save captured image in local variable
        image = camera.capture()
        //ensure that camera is closed after to avoid causing errors
        camera.close()
        //return captured image from the function
        RETURN image
    END FUNCTION
END CLASS
```

```
//VideoEncoder class definition
CLASS VideoEncoder
    //encapsulated attributes
    PRIVATE saveDirectory
    PRIVATE videoDuration

    //constructor taking givenSaveDirectory and givenVideoDuration as para-
    meters
```

```

PUBLIC PROCEDURE new(givenSaveDirectory, givenVideoDuration)
    //set the encapsulated attributes to have the values of the constructor parameters
    saveDirectory = givenSaveDirectory
    videoDuration = givenVideoDuration
END PROCEDURE

//procedure to append an image to the current video save stream
PUBLIC PROCEDURE appendToStream(image)
    //encode the image so that it can be written to the video file in the correct format
    encodedImage = image.encode()
    //open the current save file as a writable video stream
    currentSaveFile = getCurrentSaveFile()
    videoStream = OPENWRITE(currentSaveFile)
    //write the encoded image to the video stream and ensure file is closed to avoid causing IO errors
    videoStream.write(encodedImage)
    videoStream.close()
END PROCEDURE

//function to get the current file to save video to
PUBLIC FUNCTION getCurrentSaveFile()
    //initialise variable with null value so that it can be overriden
    mostRecentSaveFile = NULL
    //iterate over files in the saveDirectory
    saveFiles = saveDirectory.files()
    FOR fileIndex = 0 TO saveFiles.length - 1
        //set the mostRecentSaveFile variable to be the save file with the newest creationDateTime
        IF mostRecentSaveFile == NULL OR saveFiles[fileIndex].creationDateTime() > mostRecentSaveFile.creationDateTime() THEN
            mostRecentSaveFile = saveFiles[fileIndex]
        END IF
    END FOR

    //save the current datetime in a local variable
    currentDateTime = CURRENT_DATETIME()
    //return a new save file if there isn't one already, or most recent one has expired due to long duration. Otherwise just return most recent save file.
    IF mostRecentSaveFile == NULL OR mostRecentSaveFile.creationDateTime() < (currentDateTime - videoDuration) THEN
        RETURN CREATE_NEW_FILE(saveDirectory, currentDateTime)
    ELSE
        RETURN mostRecentSaveFile
    END IF
END FUNCTION
END CLASS

```

It should be noted that I have written fairly abstract and interpretative pseudocode for the `CameraViewer#captureImage()` method, this is due to the fact that I discourage any developer implementing this design, from implementing the functionality to capture an image from the connected camera themselves due to the significant complexity that the task entails. When I implement this design, I will use an existing library to capture an image from the camera (where

`camera.capture()` is called), and I encourage any other developer implementing my design to do the same.

In the pseudocode for the `VideoEncoder` class, I have removed the image attribute from the `getCurrentSaveFile` method as I realised it wasn't actually necessary for the function to be able to resolve the current save file. The pseudocode for the `getCurrentSaveFile` method is also fairly complicated due to its lack of linearity, use of selection and use of counter controlled iteration. Due to this complexity, I have commented the code quite thoroughly so that I can understand it when I look at it on a later date, and anyone else reading it can also understand the code.

One element of the code which I think needs to be addressed, is the fact that the function `VideoEncoder#getCurrentSaveFile` iterates over all files in the `saveDirectory` folder. Referencing the Big O notation, this results in a time complexity of $O(n)$ where 'n' is the number of save files in the save directory. Considering this linear time complexity: the longer the camera runs, the more save files it will create, and therefore the longer it takes for the `getCurrentSaveFile` function to return a value. After a few save files have been created, there will be a noticeable difference in application performance due to this linear time complexity – much more noticeable than if the algorithm has a logarithmic time complexity, which would run much faster for a larger number of save files.

Usually, linear time complexity is not something which would be detrimental to an application, however, in this instance there are multiple reasons why it would be incredibly destructive to the performance of the camera module and overall server software:

- IO Operations – input-output operations that interact with the file system are incredibly slow in relation to the running of a normal piece of code. This is mainly due to the fact that storage devices are very slow and even solid state drives (SSDs), which are the fastest type of storage, are significantly slower than the main memory (RAM) of a computer.
- Frequent Use – if I recorded the save video at a frame-rate of 30 frames per second (which is quite standard), then 30 times a second I would need to call `VideoEncoder#appendToStream(image)` as every frame needs to be encoded and appended to the video stream. However, `VideoEncoder#appendToStream(image)` inherently calls `VideoEncoder#getCurrentSaveFile()` in order to determine which video stream to save the image parameter to. This means that each save file in the save directory will be iterated on and checked 30 times per second. The time that this would take to run becomes absolutely unacceptable and detrimental to the performance of the application.

Even if the application managed to run for a short amount of time without the algorithm's inefficiency having a noticeable performance affect, the fact that the time complexity of the algorithm is linear means that after a few more save files were created (due to the passing of time), the impact would become very noticeable. This would create a terrible user experience for the stakeholder if the software stopped working effectively after they had only been running it for a few days.

Third Iteration

Because I was happy with the `CameraViewer` class designed in the previous iteration, I decided I would only redesign the `VideoEncoder` class, for reasons already mentioned.

For this attempt, I decided to favour mutability in the 'VideoEncoder' class, thinking that if the class functioned in a stateful manner, I would be able to store the current save file in a variable and change it after it had been 'videoDuration' time.

Here's the initial UML diagram I drew up for the third iteration of the camera viewer module:

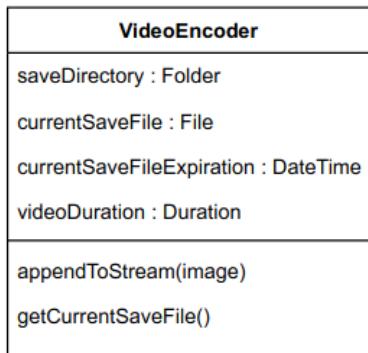


Figure 19: 'VideoEncoder' class UML diagram - third iteration

However, the majority of the functionality of this class does not fit with the class' purpose. Therefore, I felt it necessary to move the file operations into a separate class, and have the 'VideoEncoder' class use that class. This resulted in the following UML diagram:

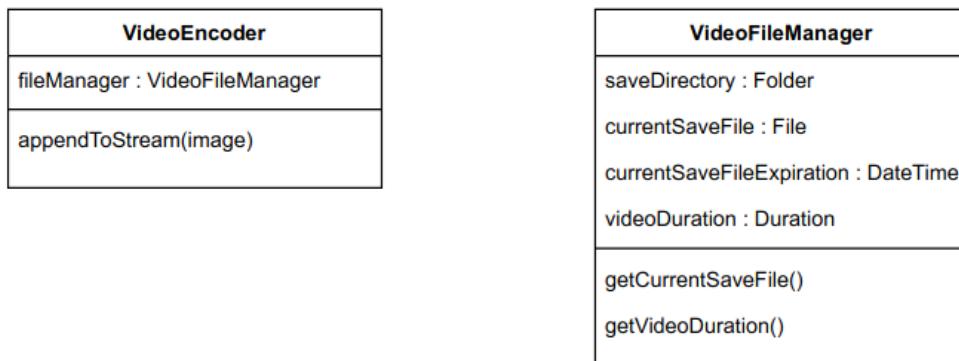


Figure 20: 'VideoEncoder' & 'VideoFileManager' classes UML diagram - third iteration

I am finally happy with this iteration of the camera viewer submodule; I feel the elements of it are successfully de-coupled, abstracted and decomposed. I concluded my design of the camera viewer submodule by writing the pseudocode for the updated 'VideoEncoder' class and the new 'VideoFileManager' class.

```
//VideoEncoder class definition
CLASS VideoEncoder
    //encapsulated attributes
    PRIVATE fileManager

    //constructor taking givenFileManager as a parameter
```

```

PUBLIC PROCEDURE new(givenFileManager)
    //set the encapsulated fileManager attribute to have the value of
the givenFileManager constructor parameter
    fileManager = givenFileManager
END PROCEDURE

//procedure to append an image to the current video save stream
PUBLIC PROCEDURE appendToStream(image)
    //encode the image so that it can be written to the video file in
the correct format
    encodedImage = image.encode()
    //open the current save file as a writable video stream
    videoStream = OPENWRITE(fileManager.getCurrentSaveFile())
    //write the encoded image to the video stream and ensure file is
closed to avoid causing IO errors
    videoStream.write(encodedImage)
    videoStream.close()
END PROCEDURE

END CLASS

```

```

//VideoFileManager class definition
CLASS VideoFileManager
    //encapsulated attributes
    PRIVATE saveDirectory
    PRIVATE currentSaveFile
    PRIVATE currentSaveFileExpiration
    PRIVATE videoDuration

    //constructor taking givenSaveDirectory and givenVideoDuration as para-
parameters
    PUBLIC PROCEDURE new(givenSaveDirectory, givenVideoDuration)
        //set the encapsulated attributes to have the values of the con-
structor parameters
        saveDirectory = givenSaveDirectory
        videoDuration = givenVideoDuration
        //set the save file attributes using `nextSaveFile()` method
        nextSaveFile()
    END PROCEDURE

    //private method to change the `currentSaveFile` variable to a new save
file and set the expiration - only used internally by this class hence
private
    PRIVATE PROCEDURE nextSaveFile()
        currentDateTime = CURRENT_DATETIME()
        currentSaveFile = CREATE_NEW_FILE(saveDirectory, currentDateTime)
        currentSaveFileExpiration = currentDateTime + videoDuration
    END PROCEDURE

    //getter method for the `videoDuration` attribute
    PUBLIC FUNCTION getVideoDuration()
        RETURN videoDuration
    END FUNCTION

```

```

//public getter method to get the currentSaveFile, will return a new file
if the previous one has expired
PUBLIC FUNCTION getCurrentSaveFile()
    //checks if the active save file expired, if it has then generate a
new one
    IF currentSaveFileExpiration < CURRENT_DATETIME() THEN
        nextSaveFile()
    END IF
    RETURN currentSaveFile
END FUNCTION
END CLASS

```

Example usage of the public-facing methods and constructors of the designed classes is shown below:

```

//create `VideoFileManager` object
saveDirectory = new Directory("videos")
videoDuration = new Duration(5, "minutes")
videoFileManager = new VideoFileManager(saveDirectory, videoDuration)

//create `VideoEncoder` object
videoEncoder = new VideoEncoder(videoFileManager)

//create `CameraViewer` object
cameraViewer = new CameraViewer(MAIN_CAMERA)

//capture camera image
image = cameraViewer.captureImage()
//write camera image to the save file video stream
videoEncoder.appendToStream(image)

```

I am quite happy with how the public-facing API is interacted with, and how the camera viewer submodule has been designed. However, I did spend multiple iterations, intricately designing this submodule; for other submodules I think that time may be more effectively spent on only one design iteration, and then adaptations can be made based on user feedback received from developed prototypes – this would also fall more in line with my chosen development methodology (rapid application development).

Networking Server

Firstly, it is worth noting that part of the core functionality of the networking server submodule, is the use of sockets and networking to allow clients to connect to the server and to facilitate the exchange of data. As previously planned, networking will be handled by the common networking module, therefore my design of the networking server submodule will heavily involve the use of the common networking module – which was designed prior to the networking server.

In the networking server, I will constantly listen for connections using the common networking module and store these connections in a data structure so that I can send media frames from the camera to these connections. This then opens the question of how I should store the connections.

Due to the fact that connections work concurrently and a user could disconnect from the server at any time, I can't be confident that the list of connections will be accurate 100% of the time without using additional synchronisation. This results in the requirement for me to utilise a concept known as 'locking'. Locking is a method of regulating access to a variable in a multithreaded context by using another variable as a 'lock', resulting in only one thread being able to access the 'locked' variable at any one time.

I must also consider which data structure used to store connections. I have no requirement to be able to access the connections by a certain index or key as I am going to be sending media frames to all authenticated clients and not really using it apart from that. This means that I will only need to iterate over the dataset of connections when I access them. Also, the size of the dataset will change dynamically because clients could connect to and disconnect from the media server at any time. Taking these requirements into account, it is clear that I need to use a list (due to the requirement for the data structure to be dynamic): either a linked list, or one backed by an array. Because of the fact that I don't need to be accessing connections by any form of index or key, this means that there's no real benefit in using a list backed by an array, and I would likely be wasting memory as I'll have to allocate a chunk of memory for the array which is not always proportional to the size of the dataset: it will usually be much bigger than necessary in order to minimize array reallocations.

Now allow me to consider how a linked list functions. Each node in a linked list is commonly stored as an object, and one property of said object is a pointer to the next node in the linked list – a linked list is therefore non-contiguous. This is known as a singly linked list and allows for an easy traversal of the dataset with an $O(n)$ time complexity. The minimal overhead of using a linked list and seemingly ideal functionality makes it an ideal candidate for my use case.

Here's a graph I have drawn to help corroborate the difference in space complexity between a linked list and a list backed by an array:

Space Complexity Analysis

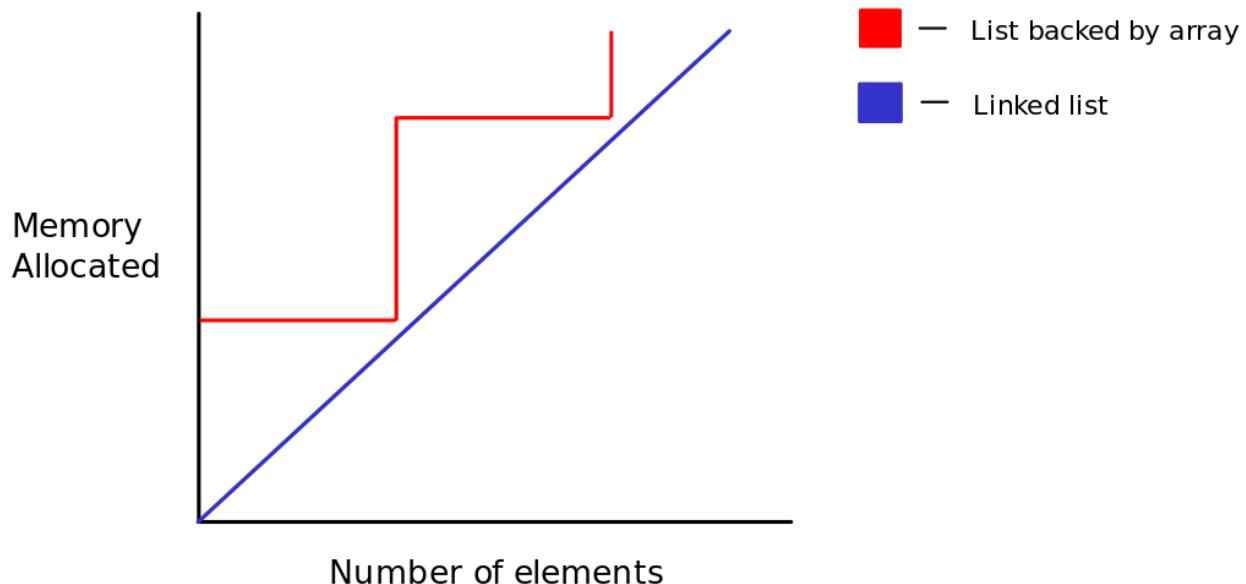


Figure 21: array-list vs linked-list space complexity graph

All prior considerations bring me to the conclusion that it would be most beneficial for the networking server to use a linked list data structure to store connections, and then use additional synchronisation (locking) to regulate access in a multithreaded context.

Implications Of Language Choice

In Java, there are two types of IO (input-output):

- Standard IO
- NIO

Standard IO is ‘blocking’ (blocks the flow of execution), therefore accepting new connections to the server requires a whole separate thread which runs concurrently. This is due to the fact that the `ServerSocket#accept` method blocks until a client connects to the server; combine this with the fact that the code will need to constantly be listening for new connections, and must be able to support multiple clients, makes it so that a separate thread is required for the Networking Server.

NIO was introduced in Java 4, and although it is very mature, it is infamously difficult to use and overly-complicated. NIO stands for Non-blocking (or ‘New’) IO, and does what the name suggests, it does not block the flow of execution – all NIO code will return instantly, describing whether there was any data read/written and how much. The event-driven architecture and minimal overhead that this allows is very appealing as there is no need for the use of multithreading, however I have decided not to use it in this project due to its previously mentioned complexity.

As a result of my decision to use Standard IO over NIO, the Networking Server will require its own thread, and each connected client will too. Whilst this may seem like a foolish decision due to the desire to minimize use of system resources with this project, I am confident it will lead to higher

maintainability – the overhead of multithreading is just something that I am going to have to accept, and attempt to make up for by increasing the efficiency of other algorithms in my code.

Client Software

GUI

As previously discussed, my client software will have a graphical user interface to show the user the camera view; I detailed in my ‘essential features’ section that the user interface must be ‘minimal yet intuitive’ with instinctively placed components.

Main Window

I will design the user interface with the main focus being on the camera viewport; the main priority of the user when using the application is to view the camera feed. To meet this user/stakeholder requirement, the camera viewport will be large and in the centre of the application viewport – covering the majority of the application.

I drew up the following annotated concept diagram for the main GUI of the software:

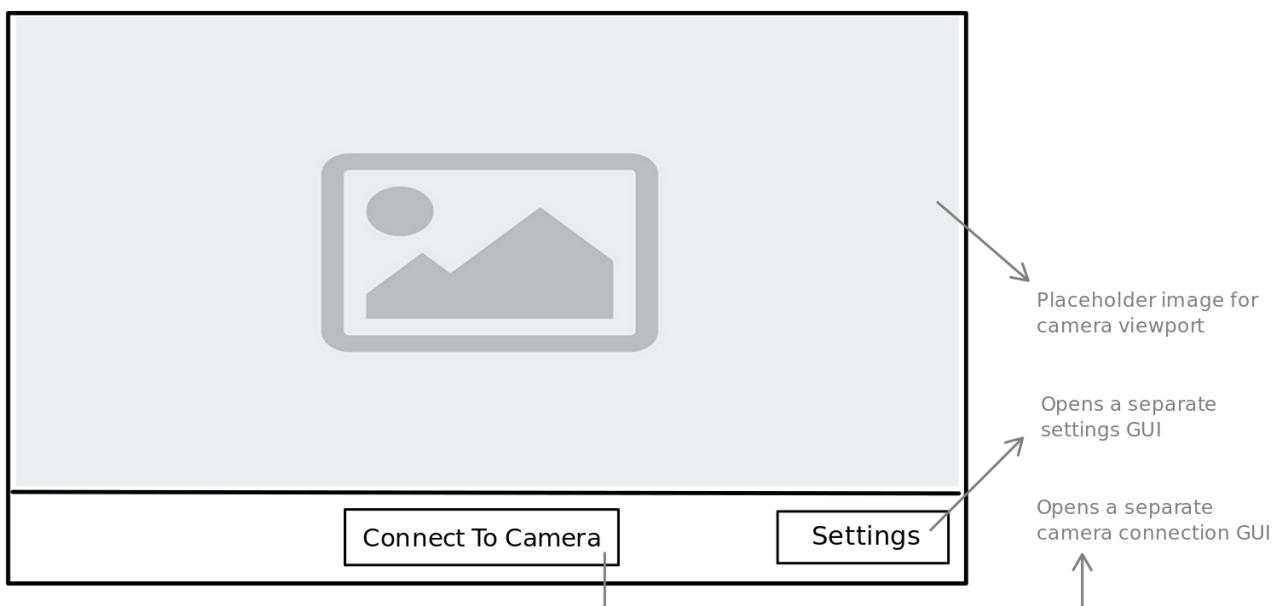


Figure 22: main window GUI design

I decided to centre the ‘Connect To Camera’ button at the bottom of the GUI to focus the user’s attention on it when they launch the application; this creates a streamlined experience for the user as they are subtly directed around the application in an intuitive manner.

The idea of using separate GUIs for connecting to a camera and the camera/application settings, was to use up as little of the application viewport as possible, allowing for more of it to be covered by the camera feed. The ‘Settings’ button seen in the bottom right is also typically placed in corners on the right-hand side of the screen, therefore the user should feel familiar and comfortable with its placement.

Camera Connection Window

The camera connection window needs to provide the user with the ability to input the internet protocol (IP) address of the camera it is connecting to, and the transmission control protocol (TCP) port on which the camera server is running. I will use a text input for each of these required data fields and validate that the user input meets the required format – more on this in the section ‘Camera Connection Inputs’.

Here’s a mock-up of what the camera connection window will look like:

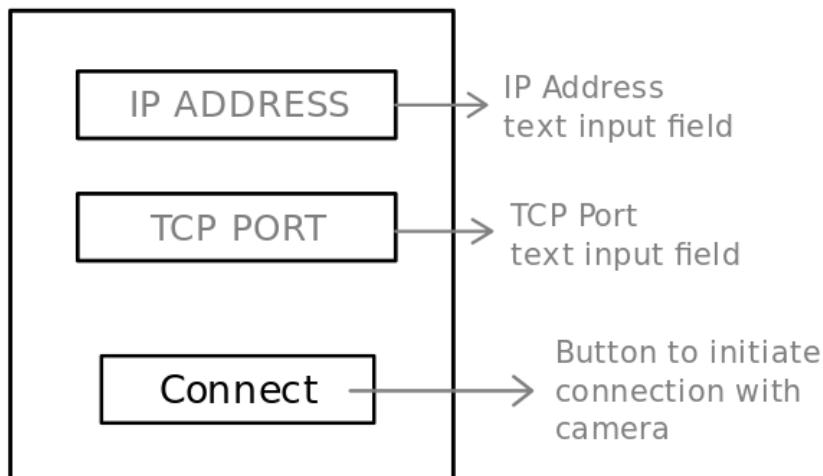


Figure 23: camera connection window GUI design

As before, I have kept everything very minimal, with only the essentials fields being asked for. In a future iteration, if I decide to implement encryption, I may also add a drop-down selection menu to allow the user to choose the type of encryption.

Pressing the ‘Connect’ button will initiate an internal communication with the control system, which in turn will direct the networking client to instantiate a TCP network connection with the IP address & TCP port input to the camera connection window.

Settings Window

The settings window will display a variety of configuration options, which can affect how both the client and server applications function. Whilst developing this project, the configurable elements of the camera software and how they are configured may change, to account for this I will need to design the settings window in such a way that allows configuration options to be added and removed with minimal hassle.

In order to achieve this, I will use a table-style window, which displays all the options in a table. This decision allows dynamic addition and removal of configuration options, suiting the adaptable nature of the settings window.

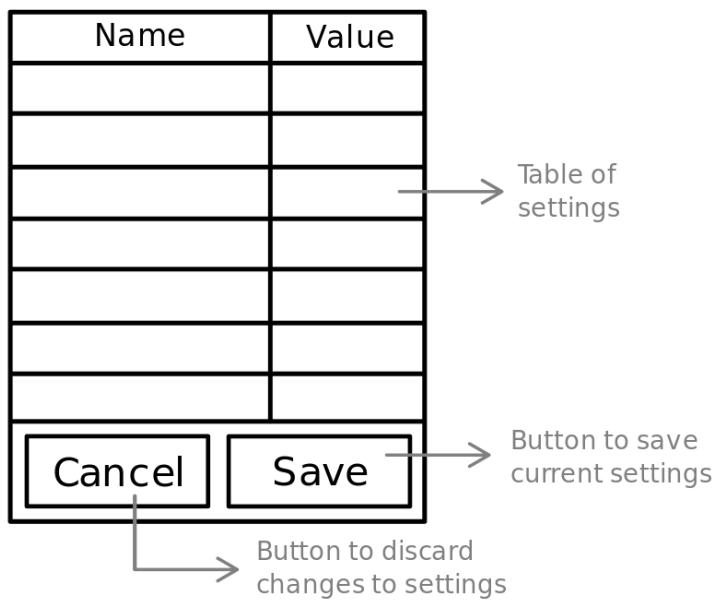


Figure 24: settings window GUI design

As seen in the above concept diagram, I have chosen not to have the 'Name' and 'Value' columns of the table taking up equal horizontal space. This decision was made because I expect the name of a configuration option to be longer than the value of it, for example, a configuration option of 'max connected clients' could have a value of '5'.

I have also included a 'Cancel' and 'Save' button which should make it much easier for the user to change the camera's settings without feeling anxious of changing the wrong setting. If they accidentally delete a setting they could just click the 'Cancel' button and no changes would be applied. Equally, they could also click the 'Save' button and all changes made to the settings table would be applied to either the settings of the connected camera or the client application – depending on the nature of the configuration option.

When the 'Save' button is pressed, the GUI submodule will interact with the control system which will decide what changes need to be made to the local client application and the remote server application. It will then execute these actions, which may involve the use of the networking client submodule to configure the remote camera server.

To be convenient for the user and developer, settings related to the server will be stored in a CSV (comma separated values) file on the server computer and settings related to the client will be stored in a CSV file on the client computer. This allows settings to persist between application restarts so that the user doesn't have to apply their settings every time they launch the client application or every time the server restarts.

The discussed design decisions should maximise usability and create a less stressful experience for the user when configuring the camera.

Control System

The control system submodule exists to provide a centralised section of code designated to controlling the flow of the application. This allows other submodules to only try and achieve what is relevant to them, and not try to control other submodules. The control system will be used by all other

submodules to enact a change in the way the application functions, or to achieve a behaviour that isn't solely relevant to the submodule using it.

An example of a procedure included in the control module, is one which changes a setting on the camera server. This would be called by the GUI submodule, the control system will handle propagating this action and its implications to other submodules such as the networking submodule, which will send the packet requesting that the remote camera server to change the setting.

Due to the fact that I have already documented other submodules fairly extensively, I have chosen not to document the control system further, as many of its procedures just involve directing the actions of other submodules and the majority of those actions are just bridging the gap between the GUI & networking client submodules.

Networking Client

The networking client, as shown by my original decomposition of the client software, involves connecting to a server, sending packets, receiving packets and processing packets – where 'packets' is a term describe data sent across the network. In my design of the common networking module, I have given a more concrete, object-oriented definition of packets: explaining how I will implement the concept in my software.

As expected, the common networking module, more specifically the connection abstraction, makes up a large majority (almost all) of the functionality required by the networking client. Therefore, there is actually very little design to do for the networking client. The only functionality to implement is the creation of a socket pointing towards the networking server, then a 'Connection' object needs to be created overlaying this socket.

Common Networking Module

This submodule forms the backbone of the other two networking-intensive submodules, therefore it is imperative that the module be carefully and accurately abstracted in order to suit the needs of the other networking submodules.

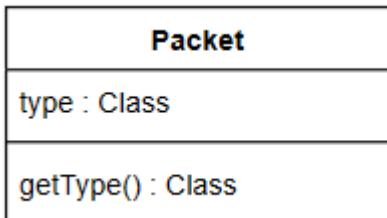
Packet Abstraction

To effectively make use of object oriented programming (OOP), I would like the data that I transmit across the network to be objects – this would make the code quite seamless due to the fact that I use OOP everywhere else in the software.

Serialization describes the process of taking an object and turning it into data which can be later converted (deserialized) to create the original object. Suitable abstractions already exist in programming languages that create the illusion of an object being written to and read from data streams; in actual fact, the object is being serialized into its data before being written to the stream, and deserialized into an object after data is read from the stream.

Because I would like to send objects over the network, I believe it would be suitable to use polymorphism with immutable classes. Polymorphism describes the ability for an object to take many forms. For example, if I have an object which has the type of a superclass, then it can be cast to (take the form of) the appropriate subclass. An appropriate name for the data superclass would be 'Packet', as it suitably describes the fact that the class' instances are to be transmitted across the network as 'packets' of information.

In order to be able to effectively make use of polymorphism, the abstract 'Packet' type must define an attribute describing the appropriate subclass for it to be cast to. Here is the UML diagram for the 'Packet' class:



*Figure 25: 'Packet' class
UML diagram*

Each specific packet type, will extend and implement the functionality provided by the 'Packet' class, by being defined as a subclass of it. Below I have designed the UML diagrams for three useful packet types:

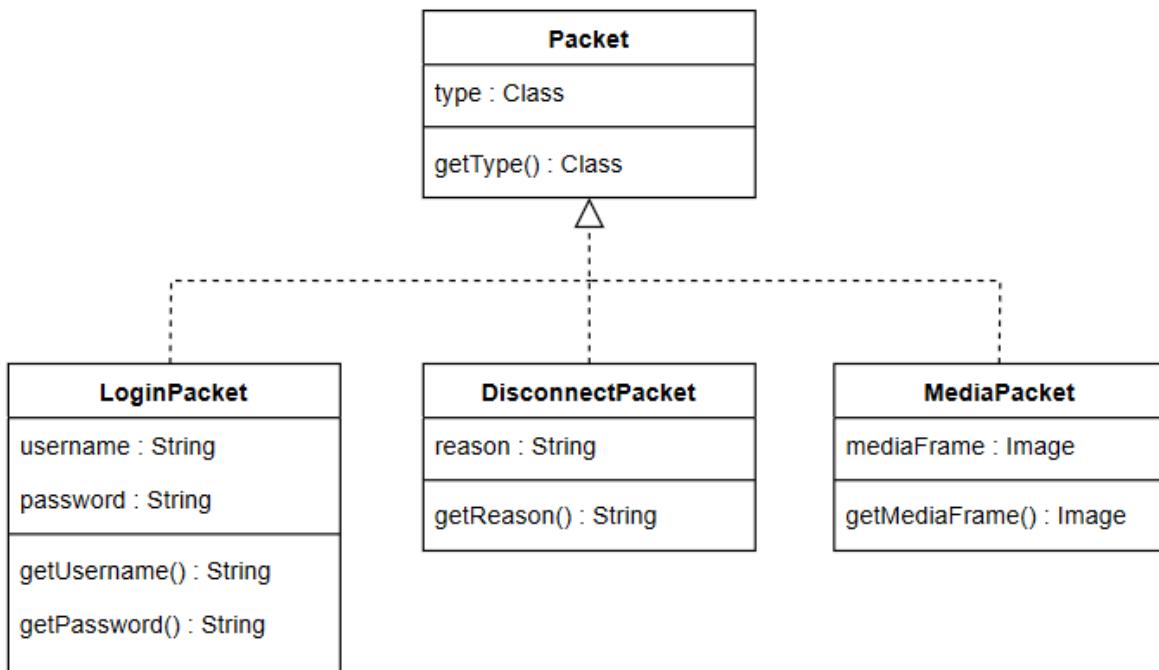


Figure 26: UML diagram showing 'Packet' class inheritance model

The packet implementations shown share practically no functionality, but they all require the functionality of the 'Packet' superclass, as the clients and server will have to interpret and determine the type of the packet.

Pseudocode is dynamically typed meaning it doesn't have a type system; the 'Packet#getType()' method would not be needed if the program was written in pseudocode. However, due to the fact I will be later programming this in a language with a static typing system, and the fact that many languages are statically typed, I am planning ahead and adding the awareness of a type casting requirement to my module design.

Here is the pseudocode for the 'Packet' class:

```
//`Packet` class definition
CLASS Packet
    //private `type` attribute declaration
    PRIVATE type

    //getter method for the `type` attribute
    PUBLIC FUNCTION getType()
        RETURN type
    END FUNCTION
END CLASS
```

And here is an example of how it may be used to receive data from a connection (methods in capitals are assumed to be already existing):

```
//read raw data from connection
dataReceived = READ_FROM_INPUT_STREAM()
//deserialize raw data into `Packet` superclass
rawPacketReceived = DESERIALIZE_TO_PACKET(dataReceived)
//cast object from the `Packet` superclass to its appropriate subclass
packetReceived = CAST_TYPE(rawPacketReceived, rawPacketReceived.getType())
```

Connection Abstraction

A socket is one endpoint of a two-way communication link between two programs running on the network[33]. Most programming languages provide the functionality of a socket through their standard library.

I will be using a `Connection` class as an abstraction overlaying the conventional `Socket` classes provided by programming languages. This is mostly due to the fact that typical `Socket` classes provided by libraries are very generic and low-level, because they need to be written such that developers can use them for different types of data – which can always be represented as bytes at the lowest level.

My `Connection` implementation will be built around the `Packet` type I defined previously. It will contain components which receive and send packets, implementing support for encryption of said packets in a future iteration.

Firstly, because my networking code will be built upon normal, blocking input-output, I will need to be constantly listening for data to receive using multithreading. This naturally leads to the decomposition of the `Connection` class into the following components:

- `PacketController` - class that receives and sends `Packet`s by directly interfacing with the raw network socket.
- `PacketListener` - class that uses multithreading to listen for `Packet`s, fires an event when it receives a `Packet`.
- `PacketSender` - class that provides functionality to send `Packet`s.

The `PacketListener` class will not expose a public API to the developer, however it will take a list of event callbacks as a constructor parameter, which will be ran when `Packet`s of an appropriate type are received.

The `PacketSender` class will expose the method `PacketSender#sendPacket(Packet)` as part of its public API, however all other operations will remain encapsulated. The `sendPacket` method will use a `Queue` data structure internally to manage the order of `Packet`s being sent through the `Connection`, this helps protect against misuse of the class through multithreading.

The `PacketController` class will not expose a public API to the developer. This class will be used by both `PacketListener` and `PacketSender` to do the low-level receipt and sending of data.

Ostensibly, the use of object-oriented programming seems unnecessary here, however by creating a subclass of the `PacketController` component and using it in a `Connection`, a developer could quickly and easily implement encryption of `Packet`s.

Due to the language-specific nature of networking code, I will not write pseudocode for the components of the `Connection` class, nor will I document the internals of these components. However, I believe knowledge of how these components relate to the enclosing `Connection` class is noteworthy.

Test Design

Testing software is incredibly important for the following reasons:

- It helps identify functionality issues with the solution
- It helps understand how close to completion the solution is
- It helps the developer locate bugs in the code
- It clearly shows when changes made during a new software iteration have broken features of previous iterations.

Due to the fact that I have chosen an iterative software development methodology (RAD), the last reason is particularly significant.

There are different ways to test software, see the below table for the meaning of each testing method, whether it is applicable to this project and why it is/isn't applicable:

Method of testing	Definition	Evaluation of applicability
White box	Where every possible path of execution of the program code is tested by someone with knowledge of how the program is structured and written.	I will use white box testing as I have many modules with components that can be unit tested with white box testing – ensuring my code functions correctly.
Black box	Where the inputs to a program module are tested against the outputs to ensure they are as expected. Consequently, the internal structure of the program code is not considered, and not every possible path of execution may be followed.	I will use black box testing in the form of integration tests when piecing together components of my solution.

Alpha	Where a small group of individually chosen testers use the software and report any issues they find.	I will use a form of alpha testing, although very minimally, as I will give a prototype solution to my stakeholder after each software iteration – ensuring they are getting what they want.
Beta	Where a small-medium group of users are selected to use the software and report any issues they find.	I will not use beta testing as my solution is for an individual stakeholder and not made for a small-medium sized group of people.

In the ‘Success Criteria’ section, I detailed features that make up the solution and how they should be tested. The column ‘How will it be tested?’ outlined white box tests (unit tests) and black box tests (integration tests) for each of the solution components.

A unit test is a piece of code which tests the logic of a subroutine by running it with various inputs and checking that the output is as expected. I will write them in such a way that they follow every path of execution in a subroutine, hence making them very effective white box tests. Since unit tests are automated, a developer can run thousands of them in very little time, which allows the code to be tested thoroughly, swiftly and autonomously.

An integration test is what occurs when you piece together two individually tested software components and check that they interact correctly together. This can be done on a small scale (two subroutines working together) or a large scale (testing the client software with the server software).

There are different types of test data that can be used when testing a piece of program code:

- Valid – within the range of expected inputs
- Boundary – on the edge of the range of expected inputs
- Erroneous – outside the range of expected inputs

I will use all these types of test data in my testing to ensure my program code is robust and tested as thoroughly as possible – making it very unlikely for the stakeholder to accidentally break the software.

To reduce the number of times I write out test data, I will define a standard test data set for each expected data type and a standard erroneous test data set that spans across multiple data types, hence can be used to test erroneous inputs on every expected data type.

Standard Test Data Sets

In the following sections: headers denote the data type expected by the input and their corresponding tables denote the actual data fed in as the input.

I would like to once again stress that the data does not have an expected output attached to it in the table as it exists purely to test whether the application crashes or faults severely when the data below is entered.

Integer

Data	Description
------	-------------

-1	Negatives
0	Zero
1	Positives
2147483647	Boundary maximum value of signed 32 bit integer
-2147483647	Boundary minimum value of signed 32 bit integer
1000000	Large positive
-1000000	Large negative
4294967295	Boundary maximum value of unsigned 32 bit integer
2147483648	Integer overflow for signed 32 bit integer
-2147483648	Integer overflow for signed 32 bit integer
4294967296	Integer overflow for unsigned 32 bit integer
9999999999999999	Exceedingly large integer

String

Data	Description
abc	Basic string
a b c	Basic string with spaces
1234567890	Basic string with numbers
¡™£¢∞§¶•º—≠	Unicode symbols
(╯°□°)╯︵ ┻━┻	Japanese style emoticons
emoji	Emoji
,./;'\`-=	Special characters
<>?:{} _+	Special characters
!@#\$%^&*()~	Special characters
\${{<%[%"}}%.	Template injection vulnerability test string[34]
	Zero-width character
abcdefghijklmnopqrstuvwxyz	Basic string
aaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaa	Long string

Real

Data	Description
1.23	Basic decimal
1.0	Basic decimal with zero after decimal place
9999999999999999.0	Left-heavy decimal
0.9999999999999999	Right-heavy decimal
0.0	Zero decimal
1	Integer
0.0000000000000001	Small decimal
123456789.123456789	Balanced decimal

Character

Data	Description
c	Basic character
1	Basic character with number
	Zero-width character
¶	Unicode character
	Emoji
	New-line character

Boolean

Data	Description
True	True mixed case
TRUE	True upper case
true	True lower case
False	False mixed case
FALSE	False upper case
false	False lower case
1	True alternate integer (depends on programming language)
0	False alternate integer (depends on programming language)

Standard Erroneous Test Data Set

The following table defines a collection of test data that should be used with all user input fields in attempt to break the application by using data of multiple types that do not match the expected input data type.

Data (Raw)	Description
“abc”	String
‘a’	Character
123	Integer
1.23	Real
True	Boolean
False	Boolean
	Nothing

User Inputs

Due to the fact that a large majority of my program code will be automatic and not require any user interaction, I believe it is useful to identify the few places where user input will be used. Doing so means that I can test those parts extensively with many different types of erroneous data, and trace where the data is used in other parts of the program.

The GUI submodule for the client application contains all the inputs that a user will make. These can be seen in the form of the settings window and the camera connection window. It could be argued that the user makes inputs on the main window, however there are only two buttons there, therefore it can't be tested for a variety of input types – it is only sensitive to a mouse click within the screen region of the button.

Configuration Inputs

Configuration inputs are what I will call the inputs on the settings window as they are used to configure the client and server software. The types of these inputs will vary depending on the setting, therefore they will utilise my previously defined standard test data sets.

Some configuration inputs will only have a few options, therefore I will use drop-down selection menus where possible to limit the user's options and confiscate their ability to input erroneous data. This also helps direct the user as to their available choices without having to tell them explicitly.

Currently, I cannot accurately state all the settings I will have in the settings windows as many of them will likely only exist because of the programming language I have chosen to write the software in. These quirks will make themselves obvious when I actually begin writing the program code, furthermore I may realise that adding a setting would yield a great improvement to the user's workflow after trying the application; these are a few of the reasons that I decided to make the settings window use a table and be as adaptable as possible.

For now, I will predict a few of the settings that my solution will contain and design their tests. All settings are presumed to be subject to their data type's corresponding standard test data set and the standard erroneous test data set.

Name	Data type	Client or server	Meaning	Input validation considerations
max connected clients	Integer	Server	How many clients should be allowed to be connected to the camera server simultaneously.	Cannot be less than the current number of connections.
connect on startup	Boolean	Client	Whether the client should connect to its last connected camera on application startup.	Would be ideal to make it a checkbox, but Boolean would be fine if it is too difficult to implement.
video save folder	String	Server	Where video recordings of the camera should be stored on the server file system.	Path entered must exist & be write-permissive on the server file system.
video duration minutes	Integer	Server	How long (in minutes) recordings of the camera should be before being stored.	Must be greater than 0 and less than 300.

Camera Connection Inputs

The two input parameters entered by the user when connecting to a camera will be the IP address of the camera and the TCP port that the camera server is running on. Each of these parameters has a few considerations to take into account.

IP Address

Initially, I believed that the IP address of the camera could be validated using a regular expression[56], or perhaps a hand-made parser. But as well as IPv4 addresses, the parser or regex would need to also be able to parse IPv6 addresses[54].

However, after a little more consideration, I realised that this would not be feasible as domain names can also be used to represent IP addresses and will be resolved by the DNS gateway used by the machine.

For example, at the time of writing my computer resolves the domain ‘google.com’ to the IPv4 address ‘142.250.178.14’ and the IPv6 address ‘2a00:1450:4009:815::200e’ – it should be noted that it is highly unlikely for these results to be reproducible as Google frequently rotate the IP address that their domain resolves to and use different addresses depending on location, in order to balance load across their servers. This was checked using the ‘nslookup’ command[19] on windows.

The final ‘nail in the coffin’ is the fact that a user may also have a custom entry in the ‘hosts’ file[51] on their computer, mapping any possible string of characters to an IP address – much like a local DNS that doesn’t restrict hostnames to be valid domains.

After all the above considerations, I decided the best way to test whether the IP address/domain entered by the user was valid, was to actually attempt to connect to it using the TCP port they provided – therefore I will not produce any specific format or type validation code.

TCP Port

Transmission Control Protocol (TCP) has a port range of 0-65535 (because in TCP, ports are 16-bit unsigned integers[57]; $2^{16} = 65536$). Furthermore, the first 1024 ports of TCP (1-1023) are privileged ports which require permission from the operating system to listen on – this will affect the allowed ports on the camera server.

The above limitations have a few consequences:

- if the user tries to start the camera server with a TCP port number between 0 and 1024 (exclusive), the operating system will reject the request to start a TCP server on said port and throw an exception in the application code (in the case of Java). This should be handled by the program code.
- both the client and server should not attempt to use a port outside of the acceptable port range for TCP (0-65535) otherwise errors will occur.
- both the client and server should input the port as an integer.

Below is a test table to account for the possible consequences of the user's input:

Client or server or both	Test input data	Expected outcome
Both	the port is five five five five	Error message to the user – “Port number must be an integer”.
Both	70000	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.
Both	65536	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.
Both	0	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.
Both	-1	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.
Server	80	Either an error message due to lack of privileges, or a successful server setup depending on the privileges with which the camera server was ran.
Server	9001	Successful server setup assuming that port 9001 is not being used by another application.
Client	8192	Successful connection assuming there is a camera server on port 8192. Otherwise, error message to user denoting that no camera server could be found on that port.

There are two final considerations I would like to make for the TCP port:

- two services cannot listen on the same port, therefore the camera server should display an error to the user when they try to start the TCP server on a port which is already being listened on by another service.
- if the client connects to an open TCP port on a computer but that TCP server is not actually the camera server, the connection should be torn down and an error message displayed to the user denoting the fact that the server that the client connected to was not a camera server. This can either be achieved by detecting if any errors occur during the handshake, or by the camera server first sending a ‘header packet’ to identify itself as a camera server upon connection instantiation.

Post Development Testing

Black Box Tests

Here are black box tests that I have designed for the user, considering the client software GUI design:

Test input field	Test input data	Expected outcome	Was expected outcome achieved? (if no, what was the actual outcome?)
Camera Port	the port is five five five five	Error message to the user – “Port number must be an integer”.	
Camera Port	70000	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.	
Camera Port	65536	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.	
Camera Port	0	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.	
Camera Port	-1	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.	
Camera Port	80	Either an error message due to lack of privileges, or a successful server setup depending on the privileges with which the camera server was ran.	
Camera Port	9001	Successful server setup assuming that port 9001 is not being used by another application.	
Camera Port	8192	Successful connection assuming there is a camera server on port 8192. Otherwise, error message to user denoting that no camera server could be found	

		on that port.	
Camera Address	Ncu28*("6!"£\$%^&*()_+=	Error message to the user specifying that a connection error occurred and that the camera address was invalid.	
Camera Address	google.com	Error message to the user specifying that a connection refused occurred because the target address isn't running a camera server.	
Camera Address	127.0.0.1	Error message to the user specifying that a connection refused occurred because the target address isn't running a camera server.	
Username	!\"£\$%^&*()`¬ 1a-=_+[]{};:@#~,./<>?	Disconnected from camera due to incorrect credentials and message shown to user to notify of this.	
Password	!\"£\$%^&*()`¬ 1-=_+[]{};:@#~,./<>?	Disconnected from camera due to incorrect credentials and message shown to user to notify of this.	
Cancel Button	Mouse Click	The camera connection window closes and no further action occurs.	
Connect Button	Mouse Click	The client software attempts to connect to the server software at the specified address and closes the camera connection window.	
Connect To Camera Button (on main GUI)	Mouse Click	Opens the camera connection window.	

I don't currently have a designed black box testing table for the server software as there is still not complete certainty regarding which settings will actually be implemented in the server configuration file.

White Box Tests

I have designed the following white box tests for the server module and connection framework:

Class(es) being tested	How is the class being tested	Expected outcome from the class being tested	Was expected outcome achieved? (if no, what was the actual outcome?)

PacketSender	A PacketSender instance is created and a test packet sent to it. The internal packet queue is then inspected.	The internal packet queue contains the test packet.	
PacketSender	A PacketSender instance is created and attached to a Connection. A test packet is sent. Sent/received data is checked at the attached Connection.	The attached connection receives the test packet.	
PacketListener	A PacketListener instance is created and callbacks are ran with a test packet when there are no callbacks registered.	Nothing happens, including no errors.	
PacketListener	A PacketListener instance is created and callbacks are ran with a test packet when there is a single callback registered. The status of the single callback will be checked.	The single callback runs and consumes the test packet.	
PacketListener	A PacketListener instance is created and callbacks are ran with a test packet when there is a single callback registered but for a different packet type. The status of the single callback will be checked.	The single callback doesn't run.	
PacketListener	A PacketListener instance is created and callbacks are ran with a test packet when there is one callback registered for the correct type but also one registered for a different packet type. The status of both callbacks will be checked.	The callback of the correct type runs but the other one doesn't run.	
PacketListener	A PacketListener instance is created and callbacks are ran with a test packet when there are two callbacks registered for the correct type. The status of both callbacks will be checked.	Both callbacks run.	
PacketListener	A PacketListener instance is created with a callback registered for the 'TestPacket' type and attached to a Connection. A test packet is then sent to the Connection and status of the registered callback is checked.	The callback runs.	
PacketController	A PacketController instance is created and attached to a Connection. The PacketController writes a test packet to the Connection and the Connection is checked to see if the packet has been received and is equal to the initially written test packet.	The test packet is received by the Connection and is equal to what was written to it.	
PacketController	A PacketController instance is created and attached to a Connection. The Connection writes a test packet to the PacketController and the PacketController attaches to the Connection.	The test packet is received by the PacketController and is equal to what was written to it.	

	tempts to read it. The received and read packet from the PacketController is checked for equality with the initially sent test packet.	what was written to it.	
UserManager	A UserManager instance is created and a user that does not exist is queried, the result is checked.	An empty result is returned.	
UserManager	A UserManager instance is created and a user that does exist is queried, the result is checked.	A result containing the user data is returned.	
UserManager	A UserManager instance is created and a user is created, the user file is then checked.	The file contains the correct user data.	
UserManager	A UserManager instance is created and the existing user is deleted. The user file is then checked if it is present.	The file containing the user data has been deleted and is no longer present.	
VideoEncoder	A VideoEncoder instance is created and two images appended to the raw media save stream using VideoEncoder#appendToStream. The media stream file is then read and tested for the correct values.	The file contains the correct data about the media frames.	
Main	A standard Server instance is constructed with authentication and callbacks added for disconnect and info packets. A Connection is then attached to the Server and an incorrect password is sent as part of the authentication process. The statuses of the callbacks are then checked.	The disconnect callback will be triggered with a reason citing incorrect credentials. The info callback will not be triggered.	
Main	A standard Server instance is constructed with authentication and callbacks added for disconnect and info packets. A Connection is then attached to the Server and an incorrect username is sent as part of the authentication process (a username that is not associated with any active account). The statuses of the callbacks are then checked.	The disconnect callback will be triggered with a reason citing incorrect credentials. The info callback will not be triggered.	
Main	A standard Server instance is constructed with authentication and callbacks added for disconnect and info packets. A Connection is then attached to the Server and correct, valid credentials are sent as part of the authentication process. The statuses of the callbacks are then checked.	The info callback will be triggered with a message informing the Connection of correct, valid credentials. The disconnect callback will not be triggered.	

		triggered.	
--	--	------------	--

It is worth noting that there are no client software white box tests here, that is because the client software is heavily GUI based, hence cannot be easily white box tested.

Development

Pre-Development Choices

Integrated Development Environment

The first thing I decided, was that I was going to use an integrated development environment (IDE) to assist me in developing the application. This is because IDEs have features such as automatic error checking, syntax highlighting and code completion – all of which will speed up the development process, whether that be assisting in code writing or debugging.

The IDE I chose was IntelliJ IDEA[11] by JetBrains due to it's wide-ranging, high-quality support for many essential features when developing Java software. It is in the industry standard when it comes to development of Java software. I have lots of experience with this IDE so it will not take any time to learn how to use it effectively.

Build System

I am programming the solution in Java, which is known to have quite a clunky workflow if you wish to run or compile your code without a build system – it can also be incredibly difficult to include third part libraries in your solution. There are two main build systems used in the Java ecosystem, Apache Maven[44] and Gradle[8]. I have lots of experience with Maven and therefore decided it would be best for me to use that; Gradle also has slightly less support in third-party libraries in comparison to Maven.

Support for Maven is built directly into the IDE that I will use (IntelliJ IDEA), therefore, it will provide me with a seamless development experience.

Project Structure

I will develop the different modules of my project together under the same parent module, but will split them up into submodules using Maven's module system. This will allow me to maintain and develop multiple separate systems at once under the same parent directory, but have them considered as different Java project whilst being effectively decoupled. However, I can integrate them together as dependencies whenever I wish, seamlessly.

The directory structure of the project will, as a result, be similar to the below:

```
SecurityCameraSystem    <-- parent project (manages child modules)
  ├── Server           <-- camera server module
  ├── Client           <-- client application module
  └── Networking       <-- common networking module
```

Version Control System

In order to track changes I have made, store & version backups and test new features without breaking the existing code-base (plus many other reasons), I have decided to use a version control system (VCS[4]) during the development of my program. The VCS I will use is Git[38] as it is incredibly well established, feature-packed and I already have lots of experience with it.

An additional consideration that must be made is that I would lose all my code if it was just stored and developed on a singular laptop, this is where another benefit of Git comes in. Git is ‘distributed’, in the sense that I can store code on a remote server using Git and it is already fully integrated into the Git workflow. This provides a type of ‘off-site’ backup which makes my project more resilient. I already run my own Gitea[45] instance on a Raspberry Pi 4B which I have made accessible from anywhere – therefore I can develop code anywhere in the world whilst ensuring the integrity of my backups.

Documentation

I will document my program code using a healthy mix of traditional comments and Javadoc comments[25]. Javadoc comments are used to explain different things about properties, methods and classes and can be used with the Javadoc tool to automatically generate documentation documents. Javadoc comments are fully integrated into many Java IDEs (such as the one I am using) and can quickly and efficiently provide insight into the function of a method, field, class etc. without the developer having to navigate to the source code of the program unit. Therefore, any other developer working on/looking at this project’s code (now and in the future) can swiftly understand the code without even having to look at it. Javadoc comments will have a green colour in my code screenshots.

Traditional comments may be used sparsely between lines of code in methods to convey meaning of the code and provide the developer with deeper insight into how it works or why it exists. Traditional comments will have a gray colour in my code screenshots.

General Third Party Dependencies

Here I outline some Maven dependencies which will likely be used in all modules of my project, and therefore have been added to the Maven parent module.

Automatic Code Generation [Lombok]

From my prior experience with Java, I know it can be tedious to write out getter methods for each variable and constructors for each class. Instead, I can use Java annotations and a third-party dependency to automatically generate these chunks of code for me at compile time. For this I will use the Maven dependency Lombok[47] – which provides this functionality through annotations such as ‘@Getter’[48], ‘@AllArgsConstructor’[49], etc.

Unit Testing [Mockito, JUnit]

To achieve the unit testing I outline in my success criteria, I will need to use some addition dependencies which make testing Java code significantly easier. I’ll use JUnit[46] to regulate and run my unit tests. I’ll also use Mockito[21] to create mock & stub objects[40] that I can use in my testing.

Annotations [JetBrains Annotations]

To keep my code organised and label any preconditions, I shall use the JetBrains Annotations[12] library. This will allow me to do things such as labelling a constructor parameter as '@Nullable'[13] to indicate that 'null'[3] may be passed in as an argument and the code will still behave in a documented manner.

Logging [Log4j]

Logging error, warning & debug messages make debugging software much easier for both the developer and end user. To log data in an effective manner, I will use the well-known Apache Log4j[55] library.

First Prototype

After all prior considerations, the configuration file for my Maven parent projects is as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>xyz.benanderson</groupId>
  <artifactId>scs_parent</artifactId>
  <packaging>pom</packaging>
  <version>0.0.1</version>

  <properties>
    <maven.compiler.source>16</maven.compiler.source>
    <maven.compiler.target>16</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.24</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.9.0</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-core</artifactId>
      <version>4.7.0</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.jetbrains</groupId>
      <artifactId>annotations</artifactId>
      <version>23.0.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-core</artifactId>
      <version>2.18.0</version>
    </dependency>
  </dependencies>

</project>

```

Figure 27: parent pom.xml with dependencies

It is worth noting here that under the ‘version’ tag, it says the version is ‘0.0.1’. This is because I intend to use semantic versioning[50] during the development of my software, as it helps keep track

of changes in an iterative environment and indicates more precise versions of the software to users.

The ‘maven.compiler.source’ & ‘maven.compiler.target’ properties indicate to Maven that the project should be built and ran with Java 16. The ‘project.build.sourceEncoding’ property indicates the character set used in the source code of the solution.

Standard Test Suite

I implemented a standard test suite class as defined in my ‘standard test data set’ part of the design section. Some raw values that I previously defined were not accepted by the Java language/my IDE, therefore I created two classes: `StandardTestSuite` and `StandardInputTestSuite`. `StandardTestSuite` treats all the values as raw values and can be used to test Java methods if required, whilst `StandardInputTestSuite` treats all the values as strings as if the user input them into an input field in the GUI.

The code for the classes is below:

Figure 28: StandardTestSuite class

Figure 29: StandardInputTestSuite class

Common Networking Module

I decided to begin the coding portion of my first prototype by programming the common networking module, as it is crucial to my solution – used by both the client and server software. This would bring me closer to achieving success criteria 4 and 5.

I created a Maven child project for the ‘Networking’ module, here is the configuration file for it:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <artifactId>scs_parent</artifactId>
        <groupId>xyz.benanderson</groupId>
        <version>0.0.1</version>
    </parent>

    <artifactId>scs_networking</artifactId>
    <version>0.0.1</version>

    <properties>
        <maven.compiler.source>16</maven.compiler.source>
        <maven.compiler.target>16</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
</project>
```

Figure 30: initial networking pom.xml

I then referenced this in my parent project with the following lines:

```
<modules>
    <module>Networking</module>
</modules>
```

Figure 31: networking module reference in parent pom.xml

Packet Abstraction

The simplest and most thoroughly designed piece of the common networking module is the Packet abstraction, therefore I decided to implement that code first. Below is the resulting code:

```
package xyz.benanderson.scs.networking;

import lombok.AllArgsConstructor;
import lombok.Getter;

@AllArgsConstructor
public abstract class Packet<T> {

    @Getter
    private final Class<T> type;

}
```

Figure 32: Packet class

This code really demonstrates the power of using Lombok[47]; comparing it to the alternative code without using Lombok:

```
package xyz.benanderson.scs.networking;

public abstract class Packet<T> {

    private final Class<T> type;

    protected Packet(Class<T> type) {
        this.type = type;
    }

    public Class<T> getType() {
        return type;
    }

}
```

Figure 33: Packet class without Lombok annotations

This code is arguably much more cluttered whereas the example with Lombok is much more readable and uses common, understandable terminology in annotations to explain what code should exist and how the class behaves. This is why I will be using the code in figure 32 (not figure 33) and will not be showing many more examples of code without Lombok unless the difference is stark and significantly affects development.

I then began adding the subclass packet types referenced in figure 26 and quickly realised that the Java generics which I decided to use when writing the `Packet` class would only hinder the writing of other packet types and not provide any benefit – consequently I removed generics from my `Packet` class:

```
package xyz.benanderson.scs.networking;

import lombok.AllArgsConstructor;
import lombok.Getter;

@AllArgsConstructor
public abstract class Packet {

    @Getter
    private final Class<?> type;

}
```

Figure 34: Packet class without generics

and added some basic packet types, designed in figure 26, to the `packets` Java package:

```
package xyz.benanderson.scs.networking.packets;

import lombok.AccessLevel;
import lombok.Getter;
import xyz.benanderson.scs.networking.Packet;

/**
 * Packet send from a client to a server when attempting to authenticate.
 */
public class LoginPacket extends Packet {

    /**
     * Username that the connection is attempting to authenticate with
     */
    @Getter(AccessLevel.PUBLIC)
    private final String username;

    /**
     * Password that the connection is attempting to authenticate with
     */
    @Getter(AccessLevel.PUBLIC)
    private final String password;

    /**
     * Constructor for {@code LoginPacket} class
     *
     * @param username username of user to authenticate as
     * @param password password to use when authenticating
     */
    public LoginPacket(String username, String password) {
        //set `type` property in the superclass to `LoginPacket.class`
        super(LoginPacket.class);
        //assign object properties to constructor parameters
        this.username = username;
        this.password = password;
    }

}
```

Figure 35: LoginPacket class

```
package xyz.benanderson.scs.networking.packets;

import lombok.Getter;
import xyz.benanderson.scs.networking.Packet;

/**
 * Packet sent by a connection when the connection wishes to disconnect from its peer.
 */
public class DisconnectPacket extends Packet {

    /**
     * Reason for the connection disconnecting
     */
    @Getter
    private final String reason;

    /**
     * Constructor for {@code DisconnectPacket} class
     *
     * @param reason reason why the connection wishes to disconnect
     */
    public DisconnectPacket(String reason) {
        //set `type` property in the superclass to `DisconnectPacket.class`
        super(DisconnectPacket.class);
        //assign instance property to constructor parameter
        this.reason = reason;
    }

}
```

Figure 36: *DisconnectPacket* class

```

package xyz.benanderson.scs.networking.packets;

import lombok.Getter;
import xyz.benanderson.scs.networking.Packet;

import java.awt.image.BufferedImage;

/**
 * Packet sent from a server to a client. This packet contains an image
 * which is the media frame/image of the security camera at a point in time.
 */
public class MediaPacket extends Packet {

    /**
     * Media frame/image that makes up the main content packet
     */
    @Getter
    private final BufferedImage mediaFrame;

    /**
     * Constructor for {@code MediaPacket} class
     *
     * @param mediaFrame BufferedImage representing the media frame/image
     */
    public MediaPacket(BufferedImage mediaFrame) {
        //set `type` property in the superclass to `MediaPacket.class`
        super(MediaPacket.class);
        //assign instance property to constructor parameter
        this.mediaFrame = mediaFrame;
    }

}

```

Figure 37: *MediaPacket* class

All the above packet types were kept in line with their original UML diagrams designs.

Connection Abstraction

I then began building out the connection abstraction; this would effectively involve implementing the components (PacketController, PacketReceiver & PacketSender) I conceptualised (but didn't actually design) when discussing the connection abstraction in the design section.

Connection Class

A further consideration that I had as I was implementing this class in Java was that, on a high level, a connection is something that has two states: open and closed. It may be useful for me to be able to check the state of a connection object at some point in the future when writing code, therefore I decided to implement an `isConnected()` method which returns a boolean value denoting whether the connection is open or closed.

Java also has some built in classes/interfaces which can help more effectively insinuate semantics of the class to other developers: one such interface is `AutoCloseable`. It is an interface which when implemented by a type, allows that type to be used in a try-with-resources statement[32] and further insinuates the existence of an opened/closed state on the object. For clarity: a try-with-resources statement allows a developer to run code where an object is treated as a 'resource', if this code throws an exception (or terminates successfully) the 'resource' is automatically closed – allowing for effective management of object state and the object's mutability.

I implemented all prior considerations into the program code and it resulted with the following Java class:

```
package xyz.benanderson.scs.networking.connection;

import lombok.AccessLevel;
import lombok.Getter;

import java.io.IOException;
import java.net.Socket;

/**
 * Connection interface which provides high level access to the connection between
 * two services. This component contains the key elements for a peer-to-peer connection
 * including the Socket, PacketController, PacketSender and PacketListener.
 */
public class Connection implements AutoCloseable {

    /**
     * Socket attribute with connection-package level getter visibility
     */
    @Getter(AccessLevel.PACKAGE)
    private final Socket socket;

    /**
     * PacketController attribute with connection-package level getter visibility
     */
    @Getter(AccessLevel.PACKAGE)
    private final PacketController packetController;

    /**
     * PacketSender attribute with getter
     */
    @Getter(AccessLevel.PUBLIC)
    private final PacketSender packetSender;

    /**
     * PacketListener attribute with getter
     */
    @Getter(AccessLevel.PUBLIC)
    private final PacketListener packetListener;
```

Figure 38: Connection class part 1/2

```

/**
 * Constructor for {@code Connection} class.
 *
 * @param socket the lower-level Java socket which the {@code Connection}
 *                object will be built on top of.
 * @throws IOException thrown if an I/O errors when preparing the I/O streams.
 */
public Connection(Socket socket) throws IOException {
    this.socket = socket;
    this.packetController = new PacketController(this);
    this.packetSender = new PacketSender(this);
    this.packetListener = new PacketListener(this);
}

/**
 * Method to override default implementation in {@link AutoCloseable} interface.
 * Requests {@code PacketController} to close input & output streams of the socket,
 * the closes the socket directly.
 *
 * @throws Exception thrown if an I/O errors when closing the I/O streams or socket.
 */
@Override
public void close() throws Exception {
    getPacketController().close();
    socket.close();
}

/**
 * Method to check the open/close state of the connection.
 *
 * @return true if this connection is connected to a peer, false if it isn't.
 */
public boolean isConnected() {
    return getSocket().isConnected() && !getSocket().isClosed();
}
}

```

Figure 39: Connection class part 2/2

I would like to bring particular attention to the `isConnected()` method for a moment. When implementing the method, I saw the documentation[31] for `Socket#isConnected` said the following:

Note: Closing a socket doesn't clear its connection state, which means this method will return true for a closed socket (see `isClosed()`) if it was successfully connected prior to being closed.

After read the documentation, I realised I also needed to check if the socket had been closed by using `Socket#isClosed` - and I couldn't just use that alone because the `isClosed` method doesn't know when a socket has been opened, only when it has been closed.

Therefore, by combining the result of `Socket#isConnected` with the negated result of `Socket#isClosed` (through the use of a Boolean AND operation), I could determine whether the socket was actively connected to (and involved in communication with) a peer.

PacketController Class

When creating the Connection class, I created almost-empty PacketController, PacketSender & PacketListener classes in order for the Connection class to not have any compilation (unknown symbol) errors. Now it was time to actually implement these classes. I started with `PacketController`:

```
package xyz.benanderson.scs.networking.connection;

import xyz.benanderson.scs.networking.Packet;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

/**
 * The {@code PacketController} class provides an encapsulation around the low-level
 * reading/writing operations from/to the {@code Socket} which underpins the
 * {@link Connection}.
 *
 * No API is publicly exposed and the class has no visibility modifier set, therefore
 * defaulting to package-private (only accessible by classes in the same package -
 * components of the connection abstraction). A {@code PacketController} is only
 * expected to be used by a {@link PacketListener} and a {@link PacketSender}.
 */
class PacketController implements AutoCloseable {

    /**
     * ObjectOutputStream wrapping & encapsulating the low-level
     * output stream of the {@code Socket}
     */
    private final ObjectOutputStream objectOutputStream;
    /**
     * ObjectInputStream wrapping & encapsulating the low-level
     * input stream of the {@code Socket}
     */
    private final ObjectInputStream objectInputStream;
```

Figure 40: PacketController class part 1/4

```
/**  
 * Constructor for {@code PacketController} class  
 *  
 * @param connection {@code Connection} object that this {@code PacketController} is  
 *                   controlling the packets for.  
 * @throws IOException thrown if an I/O error occurs when accessing the  
 * input or output streams  
 */  
public PacketController(Connection connection) throws IOException {  
    //create ObjectInputStream from socket's abstract InputStream  
    this.objectInputStream = new ObjectInputStream(  
        //get InputStream from the Socket in the Connection  
        //and pass it as the argument to the ObjectInputStream constructor  
        connection.getSocket().getInputStream()  
    );  
    //create ObjectOutputStream from socket's abstract OutputStream  
    this.objectOutputStream = new ObjectOutputStream(  
        //get OutputStream from the Socket in the Connection  
        //and pass it as the argument to the ObjectOutputStream constructor  
        connection.getSocket().getOutputStream()  
    );  
}
```

Figure 41: PacketController class part 2/4

```

/**
 * Method to write a {@code Packet} object to the output stream of the socket.
 * Visibility is 'protected' in order to encapsulate the method and make it only
 * accessible to classes in the same package (components of the connection abstraction).
 *
 * @param packet {@code Packet} object to write to the socket
 * @throws IOException thrown if an I/O error occurs
 */
protected void writePacketToSocket(Packet packet) throws IOException {
    //synchronize access to the output stream across threads to avoid race conditions
    //in a multithreaded environment - because this method is not guaranteed to
    //be run by one thread at a time (atomically)
    synchronized (objectOutputStream) {
        //write the packet object to the output stream
        objectOutputStream.writeObject(packet);
        //flush the output stream to ensure it is pushed out of memory and
        //across the network to the receiving Socket
        objectOutputStream.flush();
        //reset the output stream to clear the cache and reset the state
        //of the output stream
        objectOutputStream.reset();
    }
}

```

Figure 42: PacketController class part 3/4

Due to my prior knowledge of implementing network communication, I know a particular peculiarity of the `ObjectOutputStream` class, is that in order to reliably send packet information, methods should be called in the order as follows:

1. writeObject[29] - writes the object to the internal output stream
2. flush[27] - flushes the output stream (sending data across the network)
3. reset[28] - resets the internal state of the ObjectOutputStream object

Whilst methods 1 & 2 are (in my opinion) intuitive, method 3 is not. The ObjectOutputStream class works in a strange way in the sense that it attempts to be too smart. It caches objects sent down the stream, and when it detects an object with the same hash-code as one stored in the cache, it just uses the cached object. This can result in stale versions of data that are stored in the cache being sent instead of objects with new data.

Calling `ObjectOutputStream#reset` will clear the internal state of the object (including its cache) and avoid this error from occurring[39]. Furthermore, it seems to cache objects nearly infinitely, and may run out of memory if many different, large objects are sent through it.

These are the reasons why I am calling the relevant methods and why I am calling them in the order I am. If I didn't program it this way, runtime exceptions would occur.

```

/**
 * Method to read a {@code Packet} object from the input stream of the socket.
 * Visibility is 'protected' in order to encapsulate the method and make it only
 * accessible to classes in the same package (components of the connection abstraction).
 *
 * @throws IOException thrown if an I/O error occurs
 * @throws ClassNotFoundException thrown if the class of the object received does
 * not exist in the source code of the receiving application.
 */
protected Packet readPacketFromSocket() throws IOException, ClassNotFoundException {
    //synchronize access to the input stream across threads to avoid race conditions
    //in a multithreaded environment - because this method is not guaranteed to
    //be run by one thread at a time (atomically)
    synchronized (objectInputStream) {
        //read an object from the objectInputStream and then cast it to a
        //`Packet` object (changes the type to be the `Packet` class)
        return (Packet) objectInputStream.readObject();
    }
}

/**
 * Method overriding the default `close()` implementation from {@link AutoCloseable}.
 * This method closes the input and output streams of the Socket.
 *
 * @throws Exception thrown if an exception occurs when closing the I/O streams.
 */
@Override
public void close() throws Exception {
    //close object input & output streams - will also close the underlying
    //abstract InputStream & OutputStream on the Socket.
    objectInputStream.close();
    objectOutputStream.close();
}
}

```

Figure 43: PacketController class part 4/4

PacketSender Class

Now that I had a class with limited accessibility to control the low-level components of my connection, I could begin implementing the higher-level API that a developer would use. I began by implementing the `PacketSender` class:

```
package xyz.benanderson.scs.networking.connection;

import xyz.benanderson.scs.networking.Packet;

import java.io.IOException;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.atomic.AtomicBoolean;

/**
 * The {@code PacketSender} class exposes a high-level API to developers,
 * allowing them to send packets across the parent {@code Connection}.
 * This class internally uses a {@code Queue} for {@code Packet}s waiting to be sent.
 * This packet queue is then accessed internally to send packets asynchronously.
 */
3 usages  Ben Anderson
public class PacketSender implements AutoCloseable {

    //encapsulated packet queue which stores packets to be sent asynchronously
    4 usages
    private final Queue<Packet> packetQueue;
    //encapsulated asynchronous thread which packets are sent from
    3 usages
    private final Thread packetSendingThread;
    //private, thread-safe, atomic boolean variable to control the condition-controlled
    //loop in the packet sending thread.
    2 usages
    private final AtomicBoolean sendingPackets = new AtomicBoolean(initialValue: true);
```

Figure 44: PacketSender class part 1/4

```
/**  
 * Constructor for {@code PacketSender} class  
 *  
 * @param connection {@code Connection} object that this {@code PacketSender} is  
 *                   sending the packets for.  
 */  
1 usage  Ben Anderson  
public PacketSender(Connection connection) {  
    //initialise queue attribute with a ConcurrentLinkedQueue object,  
    //this allows asynchronous access to the queue without any race  
    //conditions or unexpected behaviour.  
    this.packetQueue = new ConcurrentLinkedQueue<>();  
  
    //create thread  
    this.packetSendingThread = new Thread(() -> {  
        //code to run in the thread  
        while (sendingPackets.get() && connection.isConnected()) {  
            //using peek instead of poll so that if a packet failed to send  
            //it can be tried again  
            Packet packetToSend = packetQueue.peek();  
            //if there are no packets in the queue, go to the start of the while loop  
            //and check again for packets to send  
            if (packetToSend == null) continue;  
    }  
}
```

Figure 45: *PacketSender* class part 2/4

```
//try to write the packet to the underlying PacketController
try {
    connection.getPacketController().writePacketToSocket(packetToSend);
    //if the packet was successfully sent across the connection,
    //remove it from the queue of packets to send
    packetQueue.remove();
} catch (IOException e) {
    //exception will be thrown if the connection was closed, in which
    //case ignore it and return
    if (!connection.isConnected()) return;

    //if the packet was NOT successfully sent across the connection,
    //keep it in the queue of packets to send and instead log the error
    //to the standard error stream - this results in trying to
    //send the packet again later.
    System.err.println("[ERROR] An error occurred whilst sending a packet" +
        " across a connection:");
    e.printStackTrace();
}
},
"Packet Sending Thread" /* name of the thread */);

//start the asynchronous packet sending thread
this.packetSendingThread.start();
}
```

Figure 46: *PacketSender class part 3/4*

```

/**
 * Public API method to be used when a developer wishes to send a {@code Packet}
 * across the parent {@code Connection}. This method adds the provided {@code Packet} object
 * to the packet queue. The packet will then be sent across the {@code Connection} asynchronously.
 *
 * @param packet {@code Packet} to send across the connection
 */
↳ Ben Anderson
public void sendPacket(Packet packet) {
    //add packet to queue
    this.packetQueue.add(packet);
}

/**
 * Method overriding the default `close()` implementation from {@link AutoCloseable}.
 * This method disables the constant sending of packets on the asynchronous
 * 'Packet Sending Thread' and waits for the thread to die.
 *
 * @throws InterruptedException thrown if an exception occurs when waiting for the thread to die.
 */
↳ Ben Anderson
@Override
public void close() throws InterruptedException {
    //set loop-controlling variable to false in order to disable the while loop
    //in the packet sending thread
    this.sendingPackets.set(false);
    //wait for the packet sending thread to die
    this.packetSendingThread.join();
}

}

```

Figure 47: PacketSender class part 4/4

In order to increase efficiency of the connection framework and to take advantage of the multicore processors that modern computers have. I have designed the `PacketSender` class to use multithreading to send packets in the packet queue.

In Java, threads will be executed either concurrently or in parallel, depending on the availability of hardware resources. Concurrent execution is when the CPU manages multiple tasks at the same time and will allocate small sections of CPU time to each. Parallel execution is when the two tasks are ran on separate CPU cores simultaneously.

One fact that makes multithreading complicated, is the fact that it can lead to race conditions which often result in unexpected behaviour. This usually occurs due to multiple threads accessing a shared stateful, mutable resource. Therefore, if one thread accesses the shared resource and updates it, other threads accessing the shared resource will be working with an out-of-sync version of the resource and will act on it with the incorrect knowledge they have about the resource.

One solution to race conditions is to use locking. Locking synchronises access to the shared mutable resource and ensures that only one thread accesses the resource at a time. However, developers should be careful when implementing locking as it can lead to situations such as deadlock (both threads are waiting for action from each other) and starvation (one thread is constantly locking access to the shared resource).

My solution has been carefully designed to avoid these potential pitfalls. I've made use of existing, thread-safe types provided by the Java standard library such as `ConcurrentLinkedQueue`[24] and `AtomicBoolean`[23]. According to the documentation of these provided classes, they use locking-free solutions. `AtomicBoolean` achieves this through the use of 'volatile' variables which are re-read from memory every time they are accessed, allowing them to avoid creating race conditions. `ConcurrentLinkedQueue` achieves it through the use of an advanced theoretical algorithm proposed in an academic paper[16] by Maged M. Michael and Michael L. Scott.

The `AtomicBoolean` variable used in the code allows the execution of the thread to be stopped from a different thread as changing its value to `false` will stop the condition-controlled loop in the packet sending thread from sending packets across the connection.

PacketListener Class

The final core element of the connection framework to implement is the `PacketListener` class:

```
package xyz.benanderson.scs.networking.connection;

import xyz.benanderson.scs.networking.Packet;

import java.io.IOException;
import java.util.*;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.function.Consumer;

/**
 * The {@code PacketSender} class exposes a high-level API to developers,
 * allowing them to create callbacks for received packets from the parent {@code Connection}.
 * This class internally uses a map to correlate packet types to a list of packet callbacks
 * for that type. These packet callbacks are executed asynchronously from packet listening
 * thread and therefore should not block the flow of execution with blocking callouts.
 */
3 usages  Ben Anderson
public class PacketListener implements AutoCloseable {

    //encapsulated map data structure storing a list of callback code blocks to run for each
    //packet type.
    6 usages
    private final Map<Class<? extends Packet>, List<Consumer<Packet>>> callbacks;
    //encapsulated asynchronous thread, on which, packets are listened for.
    3 usages
    private final Thread packetListeningThread;
    //private, thread-safe, atomic boolean variable to control the condition-controlled
    //loop in the packet listening thread.
    2 usages
    private final AtomicBoolean listeningForPackets = new AtomicBoolean( initialValue: true);
```

Figure 48: *PacketListener* class part 1/4

```
/**  
 * Constructor for {@code PacketListener} class  
 *  
 * @param connection {@code Connection} object that this {@code PacketListener} is  
 *                   listening for packets from.  
 */  
1 usage  ↗ Ben Anderson  
public PacketListener(Connection connection) {  
    //initialise the callback map with an empty hashmap  
    this.callbacks = new HashMap<>();  
  
    //create thread  
    this.packetListeningThread = new Thread(() -> {  
        //code to run in the thread  
        while (listeningForPackets.get() && connection.isConnected()) {  
            //try to read the packet from the underlying PacketController  
            try {  
                Packet packet = connection.getPacketController().readPacketFromSocket();  
                //if the packet was successfully read from the connection,  
                //run callbacks associated with the packet  
                runCallbacks(packet);  
            } catch (IOException e) {  
                //exception will be thrown if the connection was closed, in which  
                //case ignore it and return  
                if (!connection.isConnected()) return;  
    }
```

Figure 49: *PacketListener* class part 2/4

```

        //if the packet was NOT successfully read from the connection,
        //log the error to the standard error stream.
        System.err.println("[ERROR] An error occurred whilst reading a packet" +
            " from a connection:");
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        //if the packet received was not a valid packet type, log the error
        //to the standard error stream.
        System.err.println("[WARNING] An unknown packet type was received" +
            " from a connection: ");
        e.printStackTrace();
    }
}
}

}, "Packet Listening Thread" /* name of the thread */);

//start the asynchronous packet listening thread
this.packetListeningThread.start();
}

/**
 * Method to add a callback to the parent connection. The callback will be run when a packet
 * is received with the correct type for that callback.
 *
 * @param packetClass type of the packet that the callback will be triggered by
 * @param callback the code to run with the packet
 */

```

✉ Ben Anderson

```

public <T extends Packet> void addCallback(Class<T> packetClass, Consumer<T> callback) {
    //if a callback of the packet class type has not already been registered,
    //then add it to the map with a new, empty linked list.
    if (!callbacks.containsKey(packetClass))
        callbacks.put(packetClass, new LinkedList<>());

    //add the callback to the list corresponding to the packet class
    //key in the callbacks map
    callbacks.get(packetClass).add((Consumer<Packet>) callback);
}

```

Figure 50: *PacketListener* class part 3/4

```

/**
 * Method with private visibility to run all callbacks associated with the type
 * of the packet provided.
 *
 * @param packet packet to run callbacks on
 */
1 usage  ↗ Ben Anderson
private void runCallbacks(Packet packet) {
    //check if any callbacks are registered for the packet type
    if (callbacks.containsKey(packet.getType()))
        //if callbacks are registered run all callbacks for the packet type
        //using the packet as the argument
        callbacks.get(packet.getType()).forEach(callback -> callback.accept(packet));
}

/**
 * Method overriding the default `close()` implementation from {@link AutoCloseable}.
 * This method disables the constant listening for packets on the asynchronous
 * 'Packet Listening Thread' and waits for the thread to die.
 *
 * @throws InterruptedException thrown if an exception occurs when waiting for the thread to die.
 */
👤 Ben Anderson
@Override
public void close() throws InterruptedException {
    //set loop-controlling variable to false in order to disable the while loop
    //in the packet listening thread
    this.listeningForPackets.set(false);
    //wait for the packet listening thread to die
    this.packetListeningThread.join();
}

}

```

Figure 51: PacketListener class part 4/4

There are many intricate design decisions that I made during the implementation of this class. One such decision was to use a `HashMap` to implement the association of packet types with a list of the callbacks for that packet type. The class allows developers to register callbacks for packets of certain types by using the public-visibility `addCallback` method which takes the packet type and the callback code as parameters.

Due to the fact that there are an arbitrary number of packet types and the `addCallback` method needs to accept all of them, I have used Java generics to allow the behaviour of the method to change dynamically depending on the type that is used with the method. The generic type `'T'` is defined as extending `'Packet'` and therefore the method only accepts types which are subclasses of the abstract `'Packet'` type.

Using a HashMap was a good choice for my above use case in comparison with other data structures as I need to be able to filter callbacks by type, and access them quickly. These initial requirements already suggested the use of a key-value data structure such as an abstract Map. However, the decision was made to use a HashMap, instead of another structure such as a TreeMap, due to the fact that it has a time complexity of O(1) when retrieving the value associated with a key from the data structure (Big O notation constant time). This consistent access time for the data structure is very important for my use case as callbacks are going to be ran for every received packet, and packets will be received very often therefore the code that handles them needs to be as efficient as possible. Due to the constant time complexity, if callbacks are registered for many more packet types (which is definitely a possibility depending on how much the application and its features grow), there will be no impact on performance.

The value for each key in the HashMap, is a LinkedList. This is a non-contiguous data structure with an O(n) space complexity and O(n) time complexity for both retrieving elements from the list and adding elements to the list. A linked list works by creating individual nodes to encapsulate data and storing one node as the head of the linked list. Each node then stores an attribute denoting the memory address of the next node object in the list.

The combination of a HashMap and LinkedLists to store packet callbacks results in an O(1) time complexity for locating the list of callbacks and an O(n) time complexity for iterating over each callback for the packet type. The time it will take to execute each callback is completely arbitrary and depends on the algorithm implemented by the callback.

As I reflect upon my use of multithreading in the connection framework, I realise and acknowledge that my calls to `Connection#isConnected` in the packet sending thread and packet listening thread are likely ineffective and may not detect the change of state of the Socket due to the fact that it is on a different thread from where the value is changed. Using functionality such as locking would be ineffective as that just regulates access to the variable and will not force the thread to acknowledge a change of state. I cannot implement the use of the `volatile` keyword as values are read from the `Socket` class which is included in the Java standard library meaning that I cannot change it.

The ultimate implication of this problem is that the `Connection#isConnected` method may return an incorrect value (true) for a few calls after the socket has been actually closed. This results in a few extra iterations of the loops in the threads when it is not necessary. However, thankfully that is the full extent of the implication of this fact and the code should still all function correctly with an unnoticeable impact on performance and application behaviour.

Testing Module Prototype

In order to be sure that the connection framework worked correctly, and to be able to tick off success criteria points 4, 5 & 16, I wrote unit tests.

PacketController Class

I started by testing the lowest-level component (the `PacketController` class), my code for the unit tests are shown below:

```
package xyz.benanderson.scs.networking.connection;

import lombok.AccessLevel;
import lombok.Getter;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import xyz.benanderson.scs.networking.Packet;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Random;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

↳ Ben Anderson *
public class PacketControllerTest {

    4 usages
    private PacketController packetController;
    3 usages
    private Socket connectionSocket, peerSocket;

    //method runs before each test method in this class
↳ Ben Anderson
@BeforeEach
void setupPacketController() throws IOException {
    //create server on randomly assigned available port
    try (ServerSocket embeddedServer = new ServerSocket(port: 0)) {
        //create socket connections from both sides
        connectionSocket = new Socket(embeddedServer.getInetAddress(), embeddedServer.getLocalPort());
        peerSocket = embeddedServer.accept();
    }
    //create a mock Connection object to return connectionSocket when Connection#getSocket is called
    Connection connection = mock(Connection.class);
    doReturn(connectionSocket).when(connection).getSocket();

    //instantiate PacketController with connection
    packetController = new PacketController(connection);
}
}
```

Figure 52: *PacketControllerTest* class part 1

```

//method runs after each test method in this class
▲ Ben Anderson
@AfterEach
void destroyPacketController() {
    try {
        packetController.close();
        connectionSocket.close();
        peerSocket.close();
    } catch (IOException ignored) {}
}

//test packet type only used in testing to confirm
//data is correctly transmitted and received
7 usages new*
static class TestPacket extends Packet {
    1 usage
    @Getter(AccessLevel.PUBLIC)
    private final int testData;

    2 usages new*
    public TestPacket(int testData) {
        super(TestPacket.class);
        this.testData = testData;
    }
}

▲ Ben Anderson *
@Test
void testWritePacket() throws IOException, ClassNotFoundException {
    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    packetController.writePacketToSocket(testPacket);

    Packet receivedPacket;
    try (ObjectInputStream peerInputStream = new ObjectInputStream(peerSocket.getInputStream())) {
        receivedPacket = (Packet) peerInputStream.readObject();
    }
    assertEquals(testPacket.getType(), receivedPacket.getType());
    assertEquals(testPacket.getTestData(), ((TestPacket) receivedPacket).getTestData());
}

new*
@Test
void testReadPacket() throws IOException, ClassNotFoundException {
    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    try (ObjectOutputStream peerOutputStream = new ObjectOutputStream(peerSocket.getOutputStream())) {
        peerOutputStream.writeObject(testPacket);
    }

    Packet receivedPacket = packetController.readPacketFromSocket();
    assertEquals(testPacket.getType(), receivedPacket.getType());
    assertEquals(testPacket.getTestData(), ((TestPacket) receivedPacket).getTestData());
}
}

```

Figure 53: *PacketControllerTest* class part 2

However, when running the unit tests, I encountered an error as I found that the tests did not finish executing and instead the code hung indefinitely.

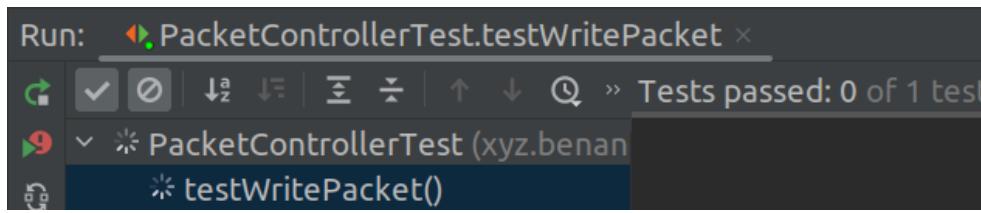


Figure 54: *PacketControllerTest#testWritePacket hanging indefinitely*

To try get to the root cause of the problem, I utilised breakpoints (a debugging feature in my chosen IDE) to pause the execution of the unit test.

```

    @BeforeEach
    void setupPacketController() throws IOException {
        //create server on randomly assigned available port
        try (ServerSocket embeddedServer = new ServerSocket( port: 0 )) {
            //create socket connections from both sides

```

Figure 55: *PacketControllerTest breakpoint example 1*

```

    @Test
    void testWritePacket() throws IOException, ClassNotFoundException {
        int testData = new Random().nextInt();
        TestPacket testPacket = new TestPacket(testData);

```

Figure 56: *PacketControllerTest breakpoint example 2*

```

    @Test
    void testReadPacket() throws IOException, ClassNotFoundException {
        int testData = new Random().nextInt();
        TestPacket testPacket = new TestPacket(testData);

```

Figure 57: *PacketControllerTest breakpoint example 3*

I ran the tests in debug mode and used the stepping through capabilities of the IDE to run the test code line-by-line – this would allow me to see where the execution flow paused.

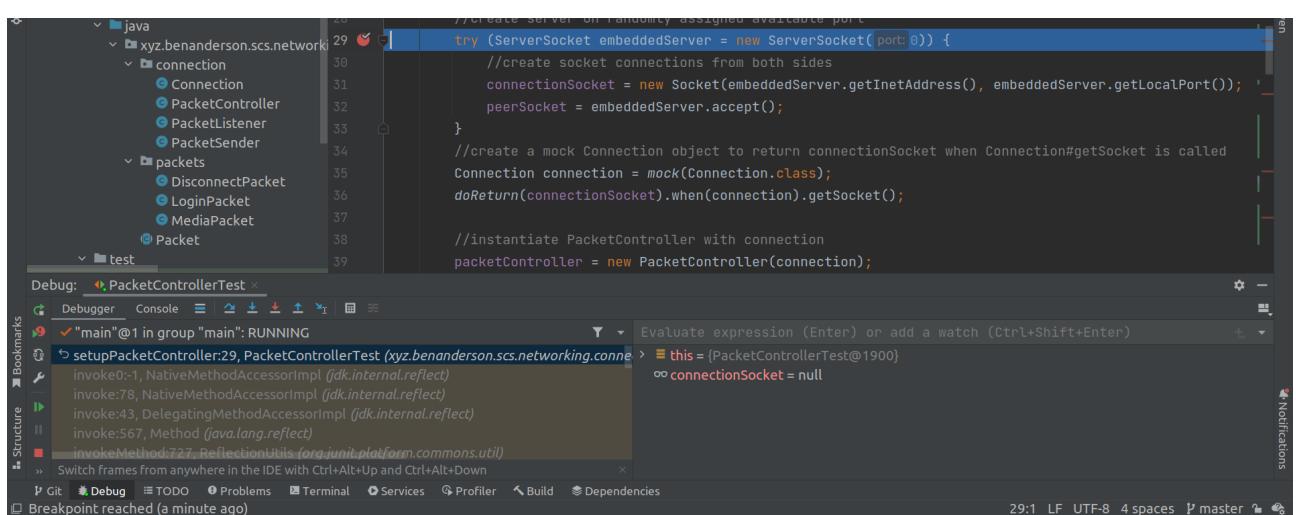


Figure 58: *PacketControllerTest stepping through start*

```

    ...
    39     packetController = new PacketController(connection); connection: "Mock for Connection, hashCode: 1674
    40 }
    41
    42 //method runs after each test method in this class
  
```

Breakpoint reached (3 minutes ago)

Figure 59: *PacketControllerTest* stepping through, before running suspect line

Frames are not available

The application is running

Breakpoint reached (4 minutes ago)

Figure 60: *PacketControllerTest* stepping through, waiting for suspect line to complete

After doing the above, I found that the code paused during the instantiation of my 'PacketController' object. I then inspected the code closer by stepping into the 'PacketController' constructor instead of stepping over it.

2 usages Ben Anderson

```

  ...
  39 @ public PacketController(Connection connection) throws IOException {
  40     //create ObjectInputStream from socket's abstract InputStream
  41     this.objectInputStream = new ObjectInputStream(
  42         //get InputStream from the Socket in the Connection
  43         //and pass it as the argument to the ObjectInputStream constructor
  44         connection.getSocket().getInputStream()
  45     );
  46     //create ObjectOutputStream from socket's abstract OutputStream
  47     this.objectOutputStream = new ObjectOutputStream(
  48         //get OutputStream from the Socket in the Connection
  49         //and pass it as the argument to the ObjectOutputStream constructor
  50         connection.getSocket().getOutputStream()
  51     );
  52 }
  
```

Breakpoint reached (moments ago)

Figure 61: *PacketControllerTest* stepping through, start of *PacketController* constructor

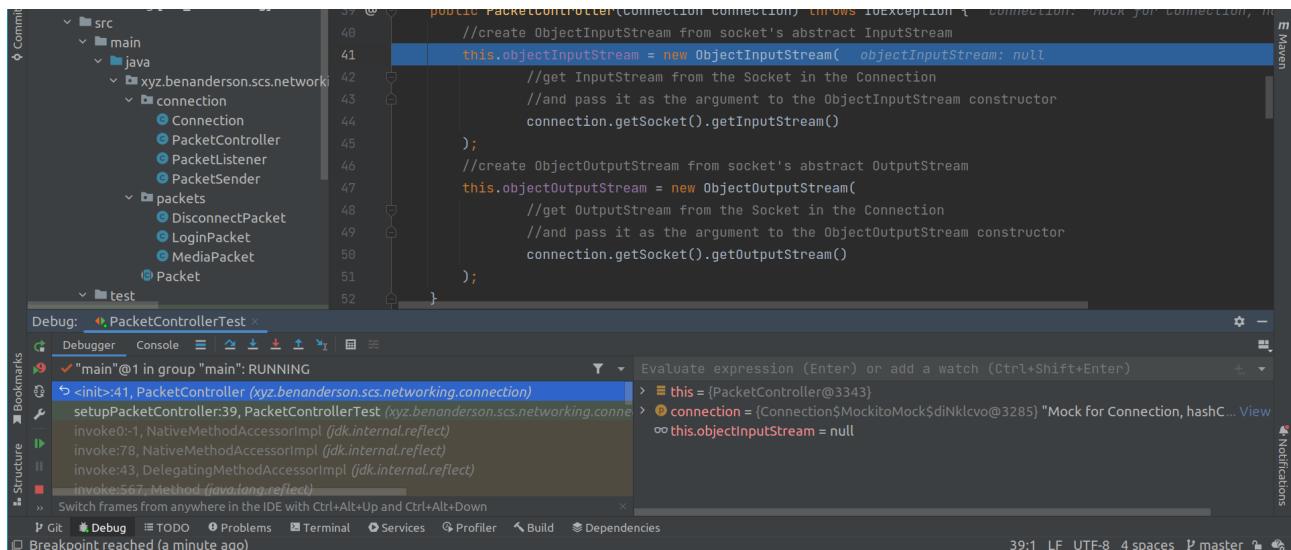


Figure 62: *PacketControllerTest* stepping through, before running suspect line in *PacketController* constructor

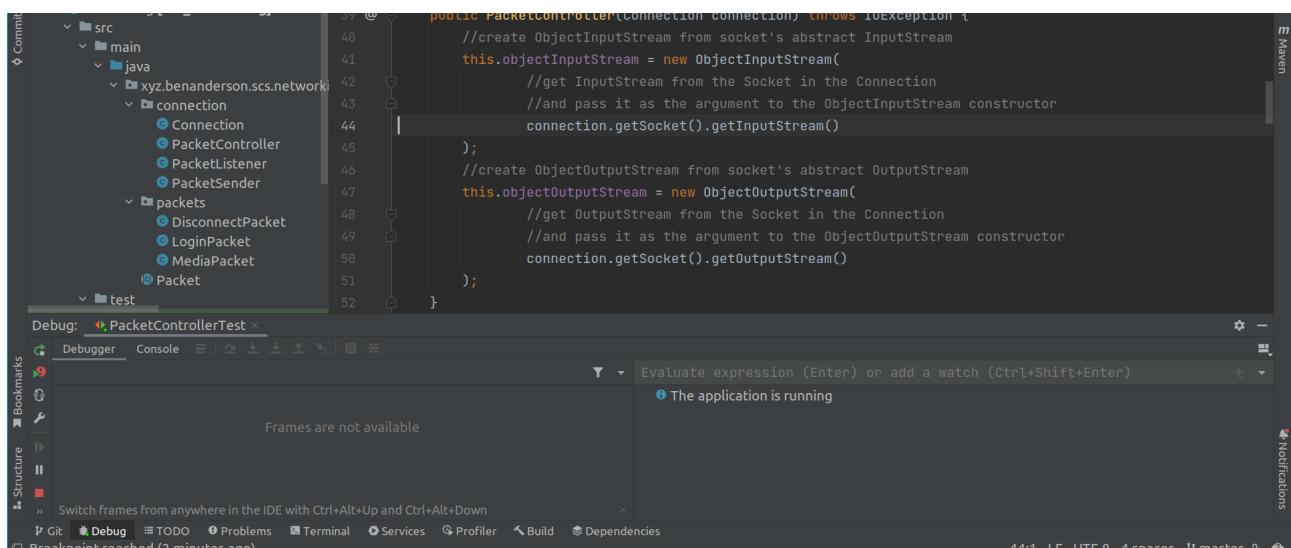


Figure 63: *PacketControllerTest* stepping through, waiting for suspect line in *PacketController* constructor to complete

I found that the code was stopping on the following line:

```
this.objectInputStream = new ObjectInputStream(
    //get InputStream from the Socket in the Connection
    //and pass it as the argument to the ObjectInputStream constructor
    connection.getInputStream()
);
```

Figure 64: *PacketController* constructor line responsible for infinite hanging bug

Now that I had located the bug in my code, I needed to understand the cause and find the solution. I started by taking a quick look at the documentation[26] of the `ObjectInputStream` constructor, and noticed it said the following:

“This constructor will block until the corresponding ObjectOutputStream has written and flushed the header.”

This means that if I instantiate an `ObjectInputStream` in a `PacketController`, an `ObjectOutputStream` must be instantiated by the connection peer in order for the code to continue. This can either happen before or after I attempt the `ObjectInputStream` instantiation.

Therefore, if both peers were using a `PacketController` to communicate with each other, they would both need to instantiate `ObjectOutputStream` first in order to write and flush the necessary headers to their peer so that the peer can instantiate an `ObjectInputStream` without blocking the flow of execution.

With this in mind, I changed the `PacketController` class’ constructor to reflect this requirement. It changed from this:

```
public PacketController(Connection connection) throws IOException {
    //create ObjectInputStream from socket's abstract InputStream
    this.objectInputStream = new ObjectInputStream(
        //get InputStream from the Socket in the Connection
        //and pass it as the argument to the ObjectInputStream constructor
        connection.getSocket().getInputStream()
    );
    //create ObjectOutputStream from socket's abstract OutputStream
    this.objectOutputStream = new ObjectOutputStream(
        //get OutputStream from the Socket in the Connection
        //and pass it as the argument to the ObjectOutputStream constructor
        connection.getSocket().getOutputStream()
    );
}
```

Figure 65: *PacketController class constructor before the bug fix*

to this:

```
public PacketController(Connection connection) throws IOException {
    //create ObjectOutputStream from socket's abstract OutputStream
    this.objectOutputStream = new ObjectOutputStream(
        //get OutputStream from the Socket in the Connection
        //and pass it as the argument to the ObjectOutputStream constructor
        connection.getSocket().getOutputStream()
    );
    //create ObjectInputStream from socket's abstract InputStream
    this.objectInputStream = new ObjectInputStream(
        //get InputStream from the Socket in the Connection
        //and pass it as the argument to the ObjectInputStream constructor
        connection.getSocket().getInputStream()
    );
}
```

Figure 66: *PacketController class constructor after the bug fix*

I simply changed the order that I instantiate `ObjectInputStream` and `ObjectOutputStream` in.

The connection framework should now work when each peer uses a `PacketController`, assuming that each of the two `PacketController`s are not instantiated on the same thread.

I proceeded to adapt the unit tests to work with my new found knowledge of instantiation order significance. This meant that I would instantiate the `ObjectOutputStream` before creating the `PacketController` in the unit tests. I would just like to once again re-iterate that if the `PacketController`s were instantiated on separate threads or separate machines, then both `PacketController`s could be instantiated at the same time (or around about); instantiation would block the thread, but the block would be released when the other was instantiated on a separate thread/machine.

My unit tests for the `PacketController` class now were as follows:

```
package xyz.benanderson.scs.networking.connection;

import lombok.AccessLevel;
import lombok.Getter;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import xyz.benanderson.scs.networking.Packet;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Random;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

public class PacketControllerTest {

    private Connection localConnection, peerConnection;

    //method runs before each test method in this class
    @BeforeEach
    void setupPacketController() throws IOException {
        Socket localSocket, peerSocket;
        //create server on randomly assigned available port
        try (ServerSocket embeddedServer = new ServerSocket(0)) {
            //create socket connections from both sides
            localSocket = new Socket(embeddedServer.getInetAddress(), embeddedServer.getLocalPort());
            peerSocket = embeddedServer.accept();
        }
        //create a mock Connection object to return localSocket when Connection#getSocket is called
        Connection localConnection = mock(Connection.class);
        doReturn(localSocket).when(localConnection).getSocket();
        this.localConnection = localConnection;

        //create a mock Connection object to return peerSocket when Connection#getSocket is called
        Connection peerConnection = mock(Connection.class);
        doReturn(peerSocket).when(peerConnection).getSocket();
        this.peerConnection = peerConnection;
    }

    //method runs after each test method in this class
    @AfterEach
    void destroyPacketController() {
        try {
            localConnection.getSocket().close();
            peerConnection.getSocket().close();
        } catch (IOException ignored) {}
    }
}
```

Figure 67: *PacketControllerTest* class version 2 part 1

```

//test packet type only used in testing to confirm
//data is correctly transmitted and received
static class TestPacket extends Packet {
    @Getter(AccessLevel.PUBLIC)
    private final int testData;

    public TestPacket(int testData) {
        super(TestPacket.class);
        this.testData = testData;
    }
}

@Test
void testWritePacket() throws IOException, ClassNotFoundException {
    //instantiated to stop the PacketController from hanging
    ObjectOutputStream peerOutputStream = new ObjectOutputStream(peerConnection.getSocket().getOutputStream());
    PacketController localPacketController = new PacketController(localConnection);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    localPacketController.writePacketToSocket(testPacket);

    Packet receivedPacket;
    try (ObjectInputStream peerInputStream = new ObjectInputStream(peerConnection.getSocket().getInputStream())) {
        receivedPacket = (Packet) peerInputStream.readObject();
    }
    peerOutputStream.close();
    localPacketController.close();

    assertNotNull(receivedPacket);
    assertEquals(testPacket.getType(), receivedPacket.getType());
    assertEquals(testPacket.getTestData(), ((TestPacket) receivedPacket).getTestData());
}

@Test
void testReadPacket() throws IOException, ClassNotFoundException {
    //instantiated to stop the PacketController from hanging
    ObjectOutputStream peerOutputStream = new ObjectOutputStream(peerConnection.getSocket().getOutputStream());
    PacketController localPacketController = new PacketController(localConnection);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    peerOutputStream.writeObject(testPacket);

    Packet receivedPacket = localPacketController.readPacketFromSocket();
    peerOutputStream.close();
    localPacketController.close();

    assertEquals(testPacket.getType(), receivedPacket.getType());
    assertEquals(testPacket.getTestData(), ((TestPacket) receivedPacket).getTestData());
}
}

```

Figure 68: PacketControllerTest class version 2 part 2

Now when running the unit tests, they did not infinitely hang, however an error was displayed in the console:

```
java.io.NotSerializableException: xyz.benanderson.scs.networking.connection.PacketControllerTest$TestPacket
    at java.base/java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1192)
    at java.base/java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:352)
    at xyz.benanderson.scs.networking.connection.PacketControllerTest.testReadPacket(PacketControllerTest.java:98) <31 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <28 internal lines>
```

Figure 69: *PacketControllerTest* - error displayed reading/writing Packet

The error is saying that a `NotSerializableException` was thrown when I tried to write the `TestPacket` object to the `ObjectOutputStream`. I solved this issue by making the abstract `Packet` superclass implement the Serializable interface, allowing for packet objects to be serialized into bytes and sent across the network.

```
package xyz.benanderson.scs.networking;

import lombok.AllArgsConstructor;
import lombok.Getter;

import java.io.Serializable;

@AllArgsConstructor
public abstract class Packet implements Serializable {

    @Getter
    private final Class<? extends Packet> type;

}
```

Figure 70: *Packet* class implementing *Serializable*

Now running the unit tests resulted in them successfully executing and passing. This means that the `PacketController` class is now fully functional.

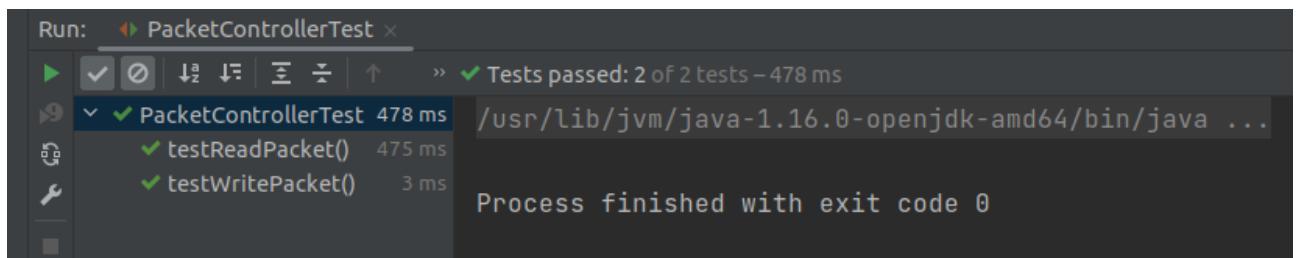


Figure 71: *PacketControllerTest* class tests passing

It's worth noting that as I began writing unit tests for other classes, I decided to move the `Test-Packet` class into its own file and outside the `PacketControllerTest` class so that it could be used in multiple unit tests. Doing this produced strange behaviour when running my unit tests; I believed this was due to JUnit not detecting changes in my code files correctly. Therefore I added the `maven-surefire-plugin` to the build configuration of my parent module.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
      <executions>
        <execution>
          <phase>test</phase>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Figure 72: Maven parent pom with surefire test configuration

This would allow me to run JUnit through Maven (using the command `mvn test`), thus ensuring all code is tested at its most recent version – the previously observed strange behaviour disappeared after this, indicating that I fixed the issue.

PacketListener Class

I implemented the unit tests for the `PacketListener` class using techniques discovered during the development of the `PacketListenerTest` class. One additional consideration that I would have to tackle whilst developing the unit tests for this class was that it used multithreading, this makes it much harder to test as I can only access the central thread of execution in my testing framework.

After some research, I realised I could solve this problem by using the `CompletableFuture` class provided by the standard library, as suggested by a StackOverflow answer[17]. This would allow me to wait an arbitrary amount of time until the listening thread executed the callback.

Here are the contents of the `PacketListenerTest` class (5 unit tests for different callback management scenarios and 1 unit test for the integration with multithreaded listening):

```
package xyz.benanderson.scs.networking.connection;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import xyz.benanderson.scs.networking.Packet;
import xyz.benanderson.scs.networking.packets.TestPacket;

import java.io.IOException;
import java.io.ObjectOutputStream;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Random;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.function.Consumer;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

↳ Ben Anderson
public class PacketListenerTest {

    24 usages
    private PacketListener packetListener;
    12 usages
    private Connection connectionMock;

↳ Ben Anderson
@BeforeEach
void setupPacketListener() {
    PacketController packetControllerMock = mock(PacketController.class);
    connectionMock = mock(Connection.class);
    doReturn(packetControllerMock).when(connectionMock).getPacketController();
}

↳ Ben Anderson
@AfterEach
void destroyPacketListener() {
    try {
        packetListener.close();
    } catch (Exception ignored) {}
}
```

Figure 73: *PacketListenerTest* class part 1

```
↳ Ben Anderson
@Test
void testNoCallbacks() throws NoSuchMethodException {
    //so that the listening thread doesn't start - it isn't relevant to this test
    doReturn(false).when(connectionMock).isConnected();
    packetListener = new PacketListener(connectionMock);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    Method runCallbacksMethod = packetListener.getClass()
        .getDeclaredMethod(name: "runCallbacks", Packet.class);
    runCallbacksMethod.setAccessible(true);

    assertDoesNotThrow(() -> runCallbacksMethod.invoke(packetListener, testPacket));
}

↳ Ben Anderson
@Test
void testSingleCallbackRun() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException {
    //so that the listening thread doesn't start - it isn't relevant to this test
    doReturn(false).when(connectionMock).isConnected();
    packetListener = new PacketListener(connectionMock);

    AtomicBoolean callbackRan = new AtomicBoolean(initialValue: false);
    Consumer<TestPacket> callback = testPacket -> callbackRan.set(true);
    packetListener.addCallback(TestPacket.class, callback);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    Method runCallbacksMethod = packetListener.getClass()
        .getDeclaredMethod(name: "runCallbacks", Packet.class);
    runCallbacksMethod.setAccessible(true);
    runCallbacksMethod.invoke(packetListener, testPacket);

    assertTrue(callbackRan.get());
}
```

Figure 74: *PacketListenerTest* class part 2

```
↳ Ben Anderson
@Test
void testSingleCallbackNoRun() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException {
    //so that the listening thread doesn't start - it isn't relevant to this test
    doReturn(false).when(connectionMock).isConnected();
    packetListener = new PacketListener(connectionMock);

    AtomicBoolean callbackRan = new AtomicBoolean(initialValue: false);
    Consumer<Packet> callback = testPacket -> callbackRan.set(true);
    packetListener.addCallback(Packet.class, callback);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    Method runCallbacksMethod = packetListener.getClass()
        .getDeclaredMethod(name: "runCallbacks", Packet.class);
    runCallbacksMethod.setAccessible(true);
    runCallbacksMethod.invoke(packetListener, testPacket);

    assertFalse(callbackRan.get());
}

↳ Ben Anderson
@Test
void testMultipleCallbacksRunOnlyOne() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException {
    //so that the listening thread doesn't start - it isn't relevant to this test
    doReturn(false).when(connectionMock).isConnected();
    packetListener = new PacketListener(connectionMock);

    AtomicBoolean callbackOneRan = new AtomicBoolean(initialValue: false);
    Consumer<TestPacket> callbackOne = testPacket -> callbackOneRan.set(true);
    AtomicBoolean callbackTwoRan = new AtomicBoolean(initialValue: false);
    Consumer<Packet> callbackTwo = testPacket -> callbackTwoRan.set(true);
    packetListener.addCallback(TestPacket.class, callbackOne);
    packetListener.addCallback(Packet.class, callbackTwo);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    Method runCallbacksMethod = packetListener.getClass()
        .getDeclaredMethod(name: "runCallbacks", Packet.class);
    runCallbacksMethod.setAccessible(true);
    runCallbacksMethod.invoke(packetListener, testPacket);

    assertTrue(callbackOneRan.get());
    assertFalse(callbackTwoRan.get());
}
```

Figure 75: *PacketListenerTest* class part 3

```

    ↳ Ben Anderson
    @Test
    void testMultipleCallbacksRunBoth() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException {
        //so that the listening thread doesn't start - it isn't relevant to this test
        doReturn(false).when(connectionMock).isConnected();
        packetListener = new PacketListener(connectionMock);

        AtomicBoolean callbackOneRan = new AtomicBoolean(initialValue: false);
        Consumer<TestPacket> callbackOne = testPacket -> callbackOneRan.set(true);
        AtomicBoolean callbackTwoRan = new AtomicBoolean(initialValue: false);
        Consumer<TestPacket> callbackTwo = testPacket -> callbackTwoRan.set(true);
        packetListener.addCallback(TestPacket.class, callbackOne);
        packetListener.addCallback(TestPacket.class, callbackTwo);

        int testData = new Random().nextInt();
        TestPacket testPacket = new TestPacket(testData);
        Method runCallbacksMethod = packetListener.getClass()
            .getDeclaredMethod(name: "runCallbacks", Packet.class);
        runCallbacksMethod.setAccessible(true);
        runCallbacksMethod.invoke(packetListener, testPacket);

        assertTrue(callbackOneRan.get());
        assertTrue(callbackTwoRan.get());
    }

    ↳ Ben Anderson
    @Test
    void testPacketListening() throws IOException, ExecutionException, InterruptedException {
        Connection localConnectionMock = mock(Connection.class);
        Socket localSocket, peerSocket;
        //create server on randomly assigned available port
        try (ServerSocket embeddedServer = new ServerSocket(port: 0)) {
            //create socket connections from both sides
            localSocket = new Socket(embeddedServer.getInetAddress(), embeddedServer.getLocalPort());
            peerSocket = embeddedServer.accept();
        }
        doReturn(localSocket).when(localConnectionMock).getSocket();
        doCallRealMethod().when(localConnectionMock).isConnected();
        ObjectOutputStream peerObjectOutputStream = new ObjectOutputStream(peerSocket.getOutputStream());
        PacketController localPacketController = new PacketController(localConnectionMock);
        doReturn(localPacketController).when(localConnectionMock).getPacketController();
        packetListener = new PacketListener(localConnectionMock);

        CompletableFuture<String> completableFuture = new CompletableFuture<>();
        Consumer<TestPacket> callback = testPacket -> {
            //deactivate the listening thread
            completableFuture.complete(value: "Callback Ran Successfully");
            try {
                localSocket.close();
                peerSocket.close();
            } catch (IOException ignored) {}
        };
        packetListener.addCallback(TestPacket.class, callback);

        int testData = new Random().nextInt();
        TestPacket testPacket = new TestPacket(testData);
        peerObjectOutputStream.writeObject(testPacket);

        assertEquals(expected: "Callback Ran Successfully", completableFuture.get());
    }
}

```

Figure 76: *PacketListenerTest* class part 4

Another point worth mentioning regarding these unit tests, is that I invoke the method `PacketListener#runCallbacks` using the following code:

```
Method runCallbacksMethod = packetListener.getClass()
    .getDeclaredMethod( name: "runCallbacks", Packet.class);
runCallbacksMethod.setAccessible(true);
runCallbacksMethod.invoke(packetListener, testPacket);
```

Figure 77: *PacketListenerTest* invoking *PacketListener#runCallbacks* using reflection

This is despite the fact that `PacketListener#runCallbacks` is defined with a private visibility. I have achieved this using an advanced Java technique called reflection, which allows me to bypass typical Java access patterns. I did this so that I wouldn't have to weaken the visibility on the `PacketListener#runCallbacks` method for the sake of my unit test.

Running all the above unit tests, yields the following result (all successful):

```
[INFO] Running xyz.benanderson.scs.networking.connection.PacketListenerTest
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.025 s
```

Figure 78: *PacketListenerTest* class unit tests successfully running

PacketSender Class

Below are the unit tests for the `PacketSender` class:

```
package xyz.benanderson.scs.networking.connection;

import org.junit.jupiter.api.Test;
import xyz.benanderson.scs.networking.Packet;
import xyz.benanderson.scs.networking.packets.TestPacket;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Field;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Queue;
import java.util.Random;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.mockito.Mockito.*;

public class PacketSenderTest {

    @Test
    void testAddPacketToQueue() throws NoSuchFieldException, IllegalAccessException {
        PacketController packetControllerMock = mock(PacketController.class);
        Connection connectionMock = mock(Connection.class);
        doReturn(packetControllerMock).when(connectionMock).getPacketController();
        //so that the listening thread doesn't start - it isn't relevant to this test
        doReturn(false).when(connectionMock).isConnected();
        PacketSender packetSender = new PacketSender(connectionMock);

        int testData = new Random().nextInt();
        TestPacket testPacket = new TestPacket(testData);
        Field packetQueueField = packetSender.getClass()
            .getDeclaredField( name: "packetQueue");
        packetQueueField.setAccessible(true);

        Queue<Packet> packetQueue = (Queue<Packet>) packetQueueField.get(packetSender);
        assertEquals(expected: 0, packetQueue.size());
        packetSender.sendPacket(testPacket);
        assertEquals(expected: 1, packetQueue.size());
        assertEquals(testPacket, packetQueue.peek());

        try {
            packetSender.close();
        } catch (InterruptedException ignored) {}
    }
}
```

Figure 79: PacketSenderTest class part 1

```

@Test
void testPacketSending() throws IOException, ClassNotFoundException {
    Connection localConnectionMock = mock(Connection.class);
    Socket localSocket, peerSocket;
    //create server on randomly assigned available port
    try (ServerSocket embeddedServer = new ServerSocket( port: 0)) {
        //create socket connections from both sides
        localSocket = new Socket(embeddedServer.getInetAddress(), embeddedServer.getLocalPort());
        peerSocket = embeddedServer.accept();
    }
    doReturn(localSocket).when(localConnectionMock).getSocket();
    doCallRealMethod().when(localConnectionMock).isConnected();
    new ObjectOutputStream(peerSocket.getOutputStream());
    PacketController localPacketController = new PacketController(localConnectionMock);
    ObjectInputStream peerObjectInputStream = new ObjectInputStream(peerSocket.getInputStream());
    doReturn(localPacketController).when(localConnectionMock).getPacketController();
    PacketSender packetSender = new PacketSender(localConnectionMock);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    packetSender.sendPacket(testPacket);

    TestPacket receivedPacket = (TestPacket) peerObjectInputStream.readObject();
    doReturn( toBeReturned: false).when(localConnectionMock).isConnected();
    try {
        localSocket.close();
        peerSocket.close();
    } catch (IOException ignored) {}

    assertNotNull(receivedPacket);
    assertEquals(testPacket.getTestData(), receivedPacket.getTestData());
}
}

```

Figure 80: *PacketSenderTest class part 2*

Running the unit tests yielded a successful result:

```
[INFO] Running xyz.benanderson.scs.networking.connection.PacketSenderTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 s
```

Figure 81: *PacketSenderTest class unit tests successfully running*

I have chosen not to utilise the standard test suite in the testing of the connection framework as I see no obvious place where unpredictable, primitive data is handled.

Here is the relevant section of the white box test table I made for the connection framework in the design section:

Class(es) being tested	How is the class being tested	Expected outcome from the class being tested	Was expected outcome achieved? (if no, what was the actual outcome?)
Packet-Sender	A PacketSender instance is created and a test packet sent to it using PacketSender#sendPacket. The internal packet queue is then inspected.	The internal packet queue contains the test packet.	Yes
Packet-Sender	A PacketSender instance is created and attached to a Connection. A test packet is sent using PacketSender#sendPacket. Sent/received data is checked at the attached Connection.	The attached connection receives the test packet.	Yes
PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there are no callbacks registered.	Nothing happens, including no errors.	Yes
PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there is a single callback registered. The status of the single callback will be checked.	The single callback runs and consumes the test packet.	Yes
PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there is a single callback registered but for a different packet type. The status of the single callback will be checked.	The single callback doesn't run.	Yes
PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there is one callback registered for the correct type but also one registered for a different packet type. The status of both callbacks will be checked.	The callback of the correct type runs but the other one doesn't run.	Yes
PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there are two callbacks registered for the correct type. The status of both callbacks will be checked.	Both callbacks run.	Yes
PacketListener	A PacketListener instance is created with a callback registered for the 'TestPacket' type and attached to a Connection. A test packet is then sent to the Connection and status of the registered callback is checked.	The callback runs.	Yes
Packet-Controller	A PacketController instance is created and attached to a Connection. The PacketController writes a test packet to the Connection	The test packet is received by the Connection and is equal	Yes

	and the Connection is checked to see if the packet has been received and is equal to the initially written test packet.	to what was written to it.	
Packet-Controller	A PacketController instance is created and attached to a Connection. The Connection writes a test packet to the PacketController and the PacketController attempts to read it. The received and read packet from the PacketController is checked for equality with the initially sent test packet.	The test packet is received by the PacketController and is equal to what was written to it.	Yes

This concludes the development and testing of the connection framework. I can now confidently say that success criteria objectives 4, 5 and 16 are achieved.

Server Software

In my parent Maven module I created a new submodule called ‘Server’ with the following configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <artifactId>scs_parent</artifactId>
        <groupId>xyz.benanderson</groupId>
        <version>0.0.1</version>
    </parent>

    <artifactId>scs_server</artifactId>
    <version>0.0.1</version>

    <properties>
        <maven.compiler.source>16</maven.compiler.source>
        <maven.compiler.target>16</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>xyz.benanderson</groupId>
            <artifactId>scs_networking</artifactId>
            <version>0.0.1</version>
        </dependency>
    </dependencies>
</project>
```

Figure 82: Server Maven module pom.xml

It's also worth pointing out that there is a dependencies section here with the 'Networking' module being referenced. These lines in the configuration file will allow me to use the connection framework in the 'Server' module.

Configuration

In my design section, I said that "settings related to the server will be stored in a CSV (comma separated values) file on the server computer". However, during the implementation of the 'Configuration` class, I decided that it would make much more sense to use the 'properties' file format which is officially supported by the Java standard library for exactly this use case.

In a '.properties' file, configuration entries are stores in the format "KEY = VALUE", where "KEY" is they entry's key and "VALUE" is the entry's value. Each configuration entry is stored on a new line. This makes it very easy for the user to understand which setting is on which line (as the key will be descriptive); the file format also supports comments, which I can use to explain different setting options to the user. This makes for a much more intuitive user experience.

I implemented a 'Configuration` class and a 'ConfigurationWrapper` class – both organised into the package `xyz.benanderson.scs.server.configuration` ('configuration` being a subpackage of the root `server` package used by the 'Server` maven submodule). Both classes were commented and documented appropriately.

The 'Configuration` class provides abstract, arbitrary access to the configuration file, whilst managing the creation & reading of it, and loading of defaults.

```
package xyz.benanderson.scs.server.configuration;

import lombok.Getter;

import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.NoSuchElementException;
import java.util.Objects;
import java.util.Optional;
import java.util.Properties;

@Getter
public class Configuration {

    private final Properties properties;
    private final String configFileName = "server.properties";
    private final Path configFile;

    public Configuration(Path configFolder) {
        //get path to config file
        configFile = configFolder.resolve(configFileName);
        //load internal default config
        Properties internalProperties = new Properties();
        loadInternalProperties(internalProperties);
        //create properties with defaults
        this.properties = new Properties(internalProperties);
        //load properties from file if it exists, otherwise create file with defaults
        if (createFileIfNotExists()) {
            loadFromFile();
        }
    }

    //loads properties into the given properties object from the
    //included default config file embedded in the application
    private void loadInternalProperties(Properties internalProperties) {
        try {
            internalProperties.load(getClass().getClassLoader().getResourceAsStream(configFileName));
        } catch (IOException e) {
            System.err.println("[WARNING] Failed to read internal configuration file.");
            e.printStackTrace();
        }
    }
}
```

Figure 83: Configuration class part 1

```

//loads properties from the configFile into the properties attribute of this object
private void loadFromFile() {
    try (InputStream inputStream = Files.newInputStream(configFile)) {
        this.properties.load(inputStream);
    } catch (IOException e) {
        System.err.println("[WARNING] Failed to read external configuration file - resorting to internal configuration.");
        e.printStackTrace();
    }
}

//copies the default config file that is embedded in the application to the location where the config file is stored
//on disk. this only occurs if the file does not exist on disk already.
private boolean createFileIfNotExists() {
    if (!Files.exists(configFile)) {
        try (InputStream inputStream = this.getClass().getClassLoader().getResourceAsStream(configFileName)) {
            Files.copy(Objects.requireNonNull(inputStream), configFile);
        } catch (Exception e) {
            System.err.println("[WARNING] Failed to write default config to configuration file '" + configFile + "'.");
            e.printStackTrace();
            return false;
        }
    }
    return true;
}

/**
 * Get an {@code Optional} denoting a {@code String} value from the config.
 *
 * @param key configuration entry key
 * @return configuration entry value as an {@code Optional}
 */
public Optional<String> getString(String key) {
    //first check system environment variables to allow for dynamic entry inclusion
    String env = System.getenv(key);
    //if key was not a system environment variable, return the Optional wrapped result from the properties table
    if (env == null || env.strip().length() == 0)
        return Optional.ofNullable(this.properties.getProperty(key));
    return Optional.of(env);
}

/**
 * Convenience method which throws a {@code NoSuchElementException} if the {@code Optional} returned by
 * {@link Configuration#getString(String)} is empty.
 *
 * @param key configuration entry key
 * @return configuration entry value (not null)
 */
public String getRequiredString(String key) throws NoSuchElementException {
    return getString(key).orElseThrow(() -> new NoSuchElementException("'" + key + "' cannot be empty"));
}

```

Figure 84: Configuration class part 2

```

/**
 * Get an {@code Optional} denoting an {@code Integer} value from the config.
 *
 * @param key configuration entry key
 * @return configuration entry value as an {@code Optional}
 */
public Optional<Integer> getInt(String key) {
    return getString(key).map(Integer::parseInt);
}

/**
 * Convenience method which throws a {@code NoSuchElementException} if the {@code Optional} returned by
 * {@link Configuration#getInt(String)} is empty.
 *
 * @param key configuration entry key
 * @return configuration entry value (not null)
 */
public int getRequiredInt(String key) throws NoSuchElementException {
    return getInt(key).orElseThrow(() -> new NoSuchElementException("'" + key + "' must be an integer"));
}

/**
 * Get an {@code Optional} denoting a {@code Double} value from the config.
 *
 * @param key configuration entry key
 * @return configuration entry value as an {@code Optional}
 */
public Optional<Double> getDouble(String key) {
    return getString(key).map(Double::parseDouble);
}

/**
 * Convenience method which throws a {@code NoSuchElementException} if the {@code Optional} returned by
 * {@link Configuration#getDouble(String)} is empty.
 *
 * @param key configuration entry key
 * @return configuration entry value (not null)
 */
public double getRequiredDouble(String key) throws NoSuchElementException {
    return getDouble(key).orElseThrow(() -> new NoSuchElementException("'" + key + "' must be a double"));
}

/**
 * Get a {@code Boolean} value from the config, or return the provided default value if the key is not present
 * in the config or they entry's value is not a boolean.
 *
 * @param key configuration entry key
 * @return configuration entry value or the default value as a {@code boolean}
 */
public boolean getBoolean(String key, boolean defaultValue) {
    return getString(key).map(Boolean::parseBoolean).orElse(defaultValue);
}
}

```

Figure 85: Configuration class part 3

The `ConfigurationWrapper` class manages the active instance of the `Configuration` class and provides getter methods for each of the settings in the configuration file to help reduce inconsistencies in code. The `ConfigurationWrapper` class follows the singleton design pattern.

```
package xyz.benanderson.scs.server.configuration;

import java.nio.file.Path;
import java.nio.file.Paths;
import java.time.Duration;
import java.time.temporal.ChronoUnit;
import java.time.temporal.TemporalUnit;

public class ConfigurationWrapper {

    private static ConfigurationWrapper instance;
    private final Configuration configuration;

    //only access provided to object through public static getInstance() method
    public static ConfigurationWrapper getInstance() {
        if (instance == null) instance = new ConfigurationWrapper();
        return instance;
    }

    //private constructor to restrict access to public static getInstance() method
    private ConfigurationWrapper() {
        //System.getProperty("user.dir") returns current working directory
        //therefore creates configuration files in current working directory
        this.configuration = new Configuration(Paths.get(System.getProperty("user.dir")));
    }

    /**
     * return Max allowed concurrent connections to the server
     */
    public int getMaxConnections() {
        return configuration.getInt("server.max-connections").orElse(Integer.MAX_VALUE);
    }
}
```

Figure 86: ConfigurationWrapper class part 1

```

/**
 * @return TCP port for the server to run on, or 0 if no port was specified in the config
 */
public int getServerPort() {
    return configuration.getInt("server.port").orElse(0);
}

/**
 * @return TCP address for the server to run on, or 127.0.0.1 (localhost) if no address was specified in the config
 */
public String getServerAddress() {
    return configuration.getString("server.address").orElse("127.0.0.1");
}

/**
 * @return {@code Path} denoting directory to save recorded videos to
 */
public Path getVideoSaveDirectory() {
    return Paths.get(configuration.getRequiredString("video.save-directory"));
}

/**
 * @return {@code Duration} denoting preferred length of recorded videos
 */
public Duration getVideoDuration() {
    int durationNumber = configuration.getInt("video.duration.number").orElse(30);
    TemporalUnit durationUnit = ChronoUnit.valueOf(configuration.getString("video.duration.unit")
        .orElse("minutes").toUpperCase());
    return Duration.of(durationNumber, durationUnit);
}
}

```

Figure 87: ConfigurationWrapper class part 2

As evident in the `Configuration` class, I have embedded the file `server.properties` into the packaged application. Whilst developing the `Configuration` and `ConfigurationWrapper` classes, I thought of some settings to put into the configuration file, the `server.properties` file is shown below:

```

server.address = 127.0.0.1
server.port = 8192
server.max-connections = 5

video.save-directory = /home/ben/Videos
video.duration.number = 30
video.duration.unit = minutes

```

Figure 88: server.properties file (first version)

These settings can be seen being used in the `ConfigurationWrapper` class, as functions such as `ConfigurationWrapper#getVideoDuration` have been implemented to parse data in the `server.-properties` file such as the preferred video duration.

Testing Submodule Prototype

I wrote a test class (`ConfigurationTest`) in order to execute an end-to-end integration test for the `Configuration` and `ConfigurationWrapper` class.

```
package xyz.benanderson.scs.server.configuration;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.time.Duration;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;

//end-to-end integration test of the `Configuration` submodule
public class ConfigurationTest {

    private Path configFile;

    @BeforeEach
    public void setupConfigFile() throws IOException {
        this.configFile = Paths.get(System.getProperty("user.dir")).resolve("server.properties");
        if (Files.exists(this.configFile)) {
            Files.delete(this.configFile);
        }
    }

    @Test
    public void testConfigurationAndWrapper() {
        Path configFile = Paths.get(System.getProperty("user.dir")).resolve("server.properties");
        assertFalse(Files.exists(configFile));

        assertEquals(ConfigurationWrapper.getInstance().getServerAddress(), "127.0.0.1");
        assertEquals(ConfigurationWrapper.getInstance().getServerPort(), 8192);
        assertEquals(ConfigurationWrapper.getInstance().getMaxConnections(), 5);
        assertEquals(ConfigurationWrapper.getInstance().getVideoDuration(), Duration.ofMinutes(30));
    }

    @AfterEach
    public void deleteConfigFile() throws IOException {
        if (Files.exists(this.configFile)) {
            Files.delete(this.configFile);
        }
    }
}
```

Figure 89: ConfigurationTest class

The class tests that the default configuration file is created and successfully loaded with the default settings. Running the test yielded a successful result with no errors, passing assertions that the `ConfigurationWrapper` was successfully retrieving the correct default values.

```
[INFO] Running xyz.benanderson.scs.server.configuration.ConfigurationTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.036 s
```

Figure 90: Configuration Test class tests passing

Here is a white box test table for the configuration submodule:

Class(es) being tested	How is the class being tested	Expected outcome from the class being tested	Was expected outcome achieved? (if no, what was the actual outcome?)
ConfigurationWrapper	The default configuration is loaded by the ConfigurationWrapper and its values checked against hard-coded defaults.	All values will match.	Yes

Here is a black box test table for values I currently have in the configuration:

Test input field	Test input data	Expected outcome	Was expected outcome achieved? (if no, what was the actual outcome?)
server.address	fwehc8sd9	Exception is thrown informing the user that it is not a valid host to run the server on.	Yes
server.address	1.2.3.4	Exception is thrown informing the user that they cannot use the requested address.	Yes
server.address	127.0.0.1	Server starts successfully and is only accessible from the current computer (because 127.0.0.1 is the localhost interface).	Yes
server.address	0.0.0.0	Server starts successfully and is accessible by any machine on the local network.	Yes
server.port	the port is five five five five	Warning is displayed stating the acceptable port range and that the server has been started on a random port. The server is started on a random port.	Yes
server.port	70000	Warning is displayed stating the acceptable port range and that the server has been started on a random port. The server is started on a random port.	Yes
server.port	65536	Warning is displayed stating the acceptable port range and that the server has been started on a random port. The server is started on a random port.	Yes
server.port	0	Server starts on a random port (chosen by OS).	Yes
server.port	-1	Warning is displayed stating the acceptable port range and that the server has been	Yes

		started on a random port. The server is started on a random port.	
server.port	80	If they aren't running with elevated privileges, an exception is thrown informing the user that they don't have the required permissions to run the camera server on that port. But if they are running with elevated privileges, the server starts on port 80.	Yes
server.max-connections	-1	Camera server starts successfully but no clients can connect to it.	Yes
server.max-connections	0	Camera server starts successfully but no clients can connect to it.	Yes
server.max-connections	1	Camera server starts successfully but only one client can connect to it.	Yes

Networking Server

The networking server allows clients to connect to the server software. As planned, it was built on top of the connection framework in order to re-use code and save time.

I implemented two generic (not in reference to Java generics) classes, `ServerBuilder` and `Server`. The `ServerBuilder` class allows a developer to create a server and have arbitrary code execute when a client connects/disconnects to/from the server. I have developed the solution such that the `Server` class is not directly instantiable by the developer, instead the `ServerBuilder` class will handle the construction of `Server` objects – strictly following the ‘builder class’ design pattern.

```
package xyz.benanderson.scs.server.networking;

import lombok.Getter;
import xyz.benanderson.scs.networking.connection.Connection;

import java.net.InetAddress;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

/**
 * The {@code ServerBuilder} class allows a developer to construct a {@code Server} instance
 * and configure it prior to instantiation. Use {@link ServerBuilder#build()} to build a {@code Server}
 * from this {@code ServerBuilder}.
 */
@Getter
public class ServerBuilder {

    //attributes that will be used by the server during construction
    private final int port;
    private final InetAddress bindAddress;
    private BiConsumer<Connection, Server> clientConnectListener;
    private BiConsumer<Connection, Server> clientDisconnectListener;
    private Consumer<Server> serverShutdownListener;

    /**
     * @param port TCP port to run the networking server on
     * @param bindAddress TCP address to run the networking server on
     */
    public ServerBuilder(int port, InetAddress bindAddress) {
        this.port = port;
        this.bindAddress = bindAddress;
        this.clientConnectListener = (con, serv) -> {};
        this.clientDisconnectListener = (con, serv) -> {};
        this.serverShutdownListener = serv -> {};
    }
}
```

Figure 91: *ServerBuilder* class part 1

```
/**  
 * Runs after a client connects and the {@code Connection} is established  
 */  
public ServerBuilder onClientConnect(BiConsumer<Connection, Server> clientConnectListener) {  
    this.clientConnectListener = clientConnectListener;  
    return this;  
}  
  
/**  
 * Runs as the client disconnects, prior to the {@code Connection} closing  
 */  
public ServerBuilder onClientDisconnect(BiConsumer<Connection, Server> clientDisconnectListener) {  
    this.clientDisconnectListener = clientDisconnectListener;  
    return this;  
}  
  
/**  
 * Runs as the server shuts down, prior to the {@code Connection}s closing  
 */  
public ServerBuilder onServerShutdown(Consumer<Server> serverShutdownListener) {  
    this.serverShutdownListener = serverShutdownListener;  
    return this;  
}  
  
/**  
 * Construct a {@code Server} from this {@code ServerBuilder}  
 */  
public Server build() { return new Server(this); }  
}
```

Figure 92: *ServerBuilder class part 2*

```
package xyz.benanderson.scs.server.networking;

import lombok.Getter;
import xyz.benanderson.scs.networking.connection.Connection;
import xyz.benanderson.scs.server.configuration.ConfigurationWrapper;

import java.io.IOException;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Collections;
import java.util.SortedMap;
import java.util.TreeMap;
import java.util.UUID;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

public class Server implements AutoCloseable {

    //attributes of the server
    @Getter
    private ServerSocket serverSocket;
    private Thread receiveConnectionsThread;
    @Getter
    private final int port;
    private final InetAddress bindAddress;
    private final SortedMap<UUID, Connection> connections;
    private final AtomicBoolean running;
    private final BiConsumer<Connection, Server> clientConnectListener;
    private final BiConsumer<Connection, Server> clientDisconnectListener;
    private final Consumer<Server> serverShutdownListener;
```

Figure 93: Server class part 1

```
/*
 * Constructor with package-private (default) visibility to allow only the {@code ServerBuilder}
 * class to instantiate {@code Server} instances.
 *
 * @param serverBuilder builder to construct this {@code Server} from
 */
Server(ServerBuilder serverBuilder) {
    //create a synchronized map to avoid race conditions in a multithreaded environment
    //uses a TreeMap as a key-value store for connections, where the key is the
    //Connection's identifier and the value is the Connection object.
    //A TreeMap was used instead of a HashMap as O(1) retrieval is not essential as
    //operations against this map will not occur often.
    this.connections = Collections.synchronizedSortedMap(new TreeMap<>());
    //mark the server as running
    this.running = new AtomicBoolean(true);
    //assign attributes from serverBuilder...
    this.port = serverBuilder.getPort();
    this.bindAddress = serverBuilder.getBindAddress();
    this.clientConnectListener = serverBuilder.getClientConnectListener();
    //append code to remove the Connection from the Server's internal connections map
    //to the end of the client disconnect listener
    BiConsumer<Connection, Server> closeListener = (con, server) -> {
        try { connections.remove(con.getId()).close(); } catch (Exception ignored) {}
    };
    this.clientDisconnectListener = serverBuilder.getClientDisconnectListener().andThen(closeListener);
    this.serverShutdownListener = serverBuilder.getServerShutdownListener();
}

/**
 * @return boolean denoting if the {@code Server} is open and running.
 */
public boolean isOpen() { return serverSocket != null && !serverSocket.isClosed() && running.get(); }

/**
 * @return An unmodifiable SortedMap representing the {@code Connection}(s) to the server
 */
public SortedMap<UUID, Connection> getConnections() { return Collections.unmodifiableSortedMap(connections); }
```

Figure 94: Server class part 2

```
/**  
 * Starts the {@code Server}'s receive-connections thread and opens a {@code ServerSocket}  
 * to allow connections.  
 */  
public void start() throws IOException {  
    start(new ServerSocket(this.port, 5, this.bindAddress));  
}  
  
//private-visibility method used by the public visibility start() method  
private void start(ServerSocket serverSocket) {  
    this.serverSocket = serverSocket;  
    this.receiveConnectionsThread = new Thread(this::receiveConnections);  
    this.receiveConnectionsThread.start();  
}  
  
/**  
 * Closes the {@code Server} and stops listening for new connections. The server shutdown  
 * listener will be triggered by the receive-connections thread. All connections will be  
 * closed and cleared from the {@code Server}.  
 */  
@Override  
public void close() {  
    //attempt to stop listening for connections on the receive-connections thread  
    running.set(false);  
    try {  
        serverSocket.close();  
    } catch (Exception ignored) {}  
    //wait for the receive-connections thread to terminate (also waits for the  
    //server shutdown listener to execute).  
    try {  
        receiveConnectionsThread.join();  
    } catch (InterruptedException ignored) {}  
    //closes all connections  
    connections.values().forEach(con -> {  
        try {  
            con.close();  
        } catch (Exception ignored) {}  
    });  
    //clear connections from memory of the server  
    connections.clear();  
}
```

Figure 95: Server class part 3

```

//private-visibility method ran only by the receive-connections thread
private void receiveConnections() {
    //condition-controlled loop to keep listening thread active as long as the
    //server is running and open
    while (isOpen()) {
        //don't accept new connections if already at max connections as specified in config
        if (connections.size() >= ConfigurationWrapper.getInstance().getMaxConnections())
            continue;
        try {
            //block until connection is accepted from the server's socket
            Socket socket = serverSocket.accept();
            //instantiate Connection object (from connection framework) using
            //the connection's socket
            Connection connection = new Connection(socket);
            //add connection to server's map of connections
            connections.put(connection.getId(), connection);
            //run the client connection listener with the newly accepted connection
            clientConnectListener.accept(connection, this);
            //add disconnect listener to the connection
            connection.setDisconnectListener(con -> clientDisconnectListener.accept(con, this));
        } catch (IOException e) {
            //print error if an exception occurs
            e.printStackTrace();
        }
    }
    //run the server shutdown listener as isOpen() now returns false
    serverShutdownListener.accept(this);
}
}

```

Figure 96: Server class part 4

There are quite a few interesting design decisions made in the implementation of the `Server` class, however I won't discuss them here as I have justified the majority of my decisions in the code of the class using comments.

Testing Submodule Prototype

When writing the `ServerTest` class, I decided to slightly re-arrange some aspects of the `Server` class and some minor elements of the connection framework in order to be more intuitive, developer friendly and atomic (through the use of synchronisation).

For example, building a server can now throw an `IOException` because the server is started immediately upon being built – the `Server#start` method has been removed. The mutability caused by the method led to uncertainty as to whether the server was running or not at different points of execution in the program. Removing this method has created a certainty regarding the state of the object at all times.

One error I encountered when developing the `ServerTest` class was with the `Server#getPort` method. Due to the fact I was creating a `ServerBuilder` object with the port value of `0`, the oper-

ating system would randomly assign the server an available port. However, the `Server#getPort` method returned the integer used in the `ServerBuilder` instance, not the port that the `ServerSocket` was actually using.

Therefore, when attempting to connect to the `ServerSocket` using the stored port, I was facing connection errors. I debugged the error using variable watch features in my IDE and found the port number that the socket was describing was `0`, hence it would not succeed in connecting to the server on the random port.

To solve this, I changed the code in `Server#getPort` to request the port from the `ServerSocket` that was running, instead of the port number that the `ServerBuilder` provided.

```
public int getPort() {  
    return getServerSocket().getLocalPort();  
}
```

Figure 97: Server#*getPort* method after bug fix

The implementation of the `ServerTest` class is shown below:

```
package xyz.benanderson.scs.server.networking;

import org.junit.jupiter.api.Test;
import xyz.benanderson.scs.networking.connection.Connection;

import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;

▲ Ben Anderson
public class ServerTest {

    ▲ Ben Anderson
    @Test
    public void testServer() throws UnknownHostException {
        CompletableFuture<String> shutdownCompletableFuture = new CompletableFuture<>();
        Consumer<Server> serverShutdownListener =
            serv -> shutdownCompletableFuture.complete(value: "Shutdown");
        CompletableFuture<String> connectCompletableFuture = new CompletableFuture<>();
        BiConsumer<Connection, Server> connectListener =
            (con, serv) -> connectCompletableFuture.complete(value: "Connected");
        CompletableFuture<String> disconnectCompletableFuture = new CompletableFuture<>();
        BiConsumer<Connection, Server> disconnectListener =
            (con, serv) -> disconnectCompletableFuture.complete(value: "Disconnected");
```

Figure 98: ServerTest class part 1

```

    ServerBuilder serverBuilder = new ServerBuilder( port: 0, InetAddress.getLocalHost())
        .onServerShutdown(serverShutdownListener)
        .onClientConnect(connectListener)
        .onClientDisconnect(disconnectListener);

    try (Server server = serverBuilder.build()) {
        Socket socket = new Socket(InetAddress.getLocalHost(), server.getPort());
        Connection connection = new Connection(socket);

        assertEquals(expected: "Connected", connectCompletableFuture.get());
        assertFalse(disconnectCompletableFuture.isDone());
        assertFalse(shutdownCompletableFuture.isDone());

        connection.close();
        assertEquals(expected: "Connected", connectCompletableFuture.get());
        assertEquals(expected: "Disconnected", disconnectCompletableFuture.get());
        assertFalse(shutdownCompletableFuture.isDone());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
    try {
        assertEquals(expected: "Connected", connectCompletableFuture.get());
        assertEquals(expected: "Disconnected", disconnectCompletableFuture.get());
        assertEquals(expected: "Shutdown", shutdownCompletableFuture.get());
    } catch (InterruptedException | ExecutionException ignored) {}
}
}

```

Figure 99: ServerTest class part 2

When running the unit test, I experienced an error in the output:

```

[ERROR] An error occurred whilst reading a packet from a connection:
java.io.EOFException Create breakpoint
    at java.base/java.io.ObjectInputStream$BlockDataInputStream.peekByte(ObjectInputStream.java:3159)
    at java.base/java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1640)
    at java.base/java.io.ObjectInputStream.readObject(ObjectInputStream.java:495)
    at java.base/java.io.ObjectInputStream.readObject(ObjectInputStream.java:453)
    at xyz.benanderson.scs.networking.connection.PacketController.readPacketFromSocket(PacketController.java:94)
    at xyz.benanderson.scs.networking.connection.PacketListener.lambda$new$0(PacketListener.java:44) <1 internal

```

Figure 100: EOF error when running initial ServerTest implementation

An EOF (end of file) error occurs when the application tries to read from a socket connection that has already been closed by the peer. In order to fix the error, I added a `catch` clause in the `PacketListener` class to stop trying to read from the socket when an `EOFException` occurred. In collaboration with this, I moved the call to `socket.close()` from the bottom to the top of the `Connection` class.

tion#close` method in order to ensure the socket is closed as the first point of call when tearing down a connection.

This solved the error and I no longer saw any exceptions in the test output, however, the test never terminated. When stepping through the code, I found that it was hanging when tearing down the `PacketSender` thread in a `Connection`. The call to `Thread#join` was infinitely hanging as it waited for the thread to terminate.

I checked the variable watch on my IDE when running the program and saw the state of the thread that I was attempted to `join` was 'BLOCKED'. This meant that a method in the thread had blocked the flow of execution, either due to a locking issue or a blocking network I/O method call.

Upon further inspection, I found that the problem was caused by the call to `Connection#isConnected`, I believed this was due to multiple threads accessing the method and the fact that the method was marked as `synchronized`. This resulted in a race condition in which one thread was starving other threads on the resource and not giving up access to the `Connection#isConnected` method.

To solve this, I removed the `synchronized` keyword from the method and re-ran the test.

```
[INFO] Running xyz.benanderson.scs.server.networking.ServerTest  
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.052 s
```

Figure 101: ServerTest test passing successfully

The test was now passing successfully, indicating that the `Server` class and connection framework were now 100% functional.

Due to the fact that some changes were made to the `Server` class and the connection framework during testing, I have included updated versions of the edited classes below:

```
package xyz.benanderson.scs.networking.connection;

import lombok.AccessLevel;
import lombok.Getter;
import lombok.Setter;

import java.io.IOException;
import java.net.Socket;
import java.util.UUID;
import java.util.function.Consumer;

/**
 * Connection interface which provides high level access to the connection between
 * two services. This component contains the key elements for a peer-to-peer connection
 * including the Socket, PacketController, PacketSender and PacketListener.
 */
👤 Ben Anderson
public class Connection implements AutoCloseable {

    /**
     * Socket attribute with connection-package level getter visibility
     */
    2 usages
    @Getter(AccessLevel.PACKAGE)
    private final Socket socket;

    /**
     * PacketController attribute with connection-package level getter visibility
     */
    1 usage
    @Getter(AccessLevel.PACKAGE)
    private final PacketController packetController;
```

Figure 102: Connection class (after fixes & edits) part 1

```
/**  
 * Unique ID attribute to help tell difference between connections  
 */  
1 usage  
@Getter(AccessLevel.PUBLIC)  
private final UUID id;  
  
/**  
 * PacketSender attribute with getter  
 */  
1 usage  
@Getter(AccessLevel.PUBLIC)  
private final PacketSender packetSender;  
  
/**  
 * PacketListener attribute with getter  
 */  
1 usage  
@Getter(AccessLevel.PUBLIC)  
private final PacketListener packetListener;  
  
/**  
 * Constructor for {@code Connection} class.  
 *  
 * @param socket the lower-level Java socket which the {@code Connection}  
 * object will be built on top of.  
 * @throws IOException thrown if an I/O errors when preparing the I/O streams.  
 */  
👤 Ben Anderson  
public Connection(Socket socket) throws IOException {  
    this.id = UUID.randomUUID();  
    this.socket = socket;  
    this.packetController = new PacketController(connection: this);  
    this.packetSender = new PacketSender(connection: this);  
    this.packetListener = new PacketListener(connection: this);  
    this.disconnectListener = con -> {};  
}
```

Figure 103: Connection class (after fixes & edits) part 2

```
1 usage
@Getter
@Setter
private Consumer<Connection> disconnectListener;

/**
 * Method to override default implementation in {@link AutoCloseable} interface.
 * Requests {@code PacketController} to close input & output streams of the socket,
 * then closes the socket directly.
 *
 * @throws Exception thrown if an I/O errors when closing the I/O streams or socket.
 */
↳ Ben Anderson
@Override
public synchronized void close() throws Exception {
    socket.close();
    getDisconnectListener().accept(this);
    getPacketController().close();
    getPacketSender().close();
    getPacketListener().close();
}

/**
 * Method to check the open/close state of the connection.
 *
 * @return true if this connection is connected to a peer, false if it isn't.
 */
↳ Ben Anderson
public boolean isConnected() { return getSocket().isConnected() && !getSocket().isClosed(); }

}
```

Figure 104: Connection class (after fixes & edits) part 3

```
package xyz.benanderson.scs.networking.connection;

import xyz.benanderson.scs.networking.Packet;

import java.io.EOFException;
import java.io.IOException;
import java.util.*;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.function.Consumer;

/**
 * The {@code PacketSender} class exposes a high-level API to developers,
 * allowing them to create callbacks for received packets from the parent {@code Connection}.
 * This class internally uses a map to correlate packet types to a list of packet callbacks
 * for that type. These packet callbacks are executed asynchronously from packet listening
 * thread and therefore should not block the flow of execution with blocking callouts.
 */
10 usages  Ben Anderson
public class PacketListener implements AutoCloseable {

    //encapsulated map data structure storing a list of callback code blocks to run for each
    //packet type.
    6 usages
    private final Map<Class<? extends Packet>, List<Consumer<Packet>>> callbacks;
    //encapsulated asynchronous thread, on which, packets are listened for.
    3 usages
    private final Thread packetListeningThread;
    //private, thread-safe, atomic boolean variable to control the condition-controlled
    //loop in the packet listening thread.
    2 usages
    private final AtomicBoolean listeningForPackets = new AtomicBoolean(initialValue: true);
```

Figure 105: *PacketListener* class (after fixes & edits) part 1

```

/**
 * Constructor for {@code PacketListener} class
 *
 * @param connection {@code Connection} object that this {@code PacketListener} is
 *                   listening for packets from.
 */
7 usages  • Ben Anderson
public PacketListener(Connection connection) {
    //initialise the callback map with an empty hashmap
    this.callbacks = new HashMap<>();

    //create thread
    this.packetListeningThread = new Thread(() -> {
        //code to run in the thread
        while (listeningForPackets.get() && connection.isConnected()) {
            //try to read the packet from the underlying PacketController
            try {
                Packet packet = connection.getPacketController().readPacketFromSocket();
                //if the packet was successfully read from the connection,
                //run callbacks associated with the packet
                runCallbacks(packet);
            } catch (EOFException disconnected) {
                break;
            } catch (IOException e) {
                //exception will be thrown if the connection was closed, in which
                //case ignore it and return
                if (!connection.isConnected()) return;

                //if the packet was NOT successfully read from the connection,
                //log the error to the standard error stream.
                System.err.println("[ERROR] An error occurred whilst reading a packet" +
                    " from a connection:");
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                //if the packet received was not a valid packet type, log the error
                //to the standard error stream.
                System.err.println("[WARNING] An unknown packet type was received" +
                    " from a connection: ");
                e.printStackTrace();
            }
        }
        try {
            if (connection.isConnected())
                connection.close();
        } catch (Exception ignored) {}
    }, "Packet Listening Thread" /* name of the thread */);

    //start the asynchronous packet listening thread
    this.packetListeningThread.start();
}

```

Figure 106: *PacketListener* class (after fixes & edits) part 2

```
/*
 * Method to add a callback to the parent connection. The callback will be run when a packet
 * is received with the correct type for that callback.
 *
 * @param packetClass type of the packet that the callback will be triggered by
 * @param callback the code to run with the packet
 */
7 usages  Ben Anderson
public <T extends Packet> void addCallback(Class<T> packetClass, Consumer<T> callback) {
    //if a callback of the packet class type has not already been registered,
    //then add it to the map with a new, empty linked list.
    if (!callbacks.containsKey(packetClass))
        callbacks.put(packetClass, new LinkedList<>());

    //add the callback to the list corresponding to the packet class
    //key in the callbacks map
    callbacks.get(packetClass).add((Consumer<Packet>) callback);
}

/*
 * Method with private visibility to run all callbacks associated with the type
 * of the packet provided.
 *
 * @param packet packet to run callbacks on
 */
1 usage  Ben Anderson
private void runCallbacks(Packet packet) {
    //check if any callbacks are registered for the packet type
    if (callbacks.containsKey(packet.getType()))
        //if callbacks are registered run all callbacks for the packet type
        //using the packet as the argument
        callbacks.get(packet.getType()).forEach(callback -> callback.accept(packet));
}
```

Figure 107: *PacketListener* class (after fixes & edits) part 3

```
/*
 * Method overriding the default `close()` implementation from {@link AutoCloseable}.
 * This method disables the constant listening for packets on the asynchronous
 * 'Packet Listening Thread' and waits for the thread to die.
 */
• Ben Anderson
@Override
public void close() throws InterruptedException {
    //set loop-controlling variable to false in order to disable the while loop
    //in the packet listening thread
    this.listeningForPackets.set(false);
    //wait for the packet listening thread to die
    this.packetListeningThread.join();
}

}
```

Figure 108: *PacketListener* class (after fixes & edits) part 4

```
package xyz.benanderson.scs.networking.connection;

import xyz.benanderson.scs.networking.Packet;

import java.io.IOException;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.atomic.AtomicBoolean;

/**
 * The {@code PacketSender} class exposes a high-level API to developers,
 * allowing them to send packets across the parent {@code Connection}.
 * This class internally uses a {@code Queue} for {@code Packet}s waiting to be sent.
 * This packet queue is then accessed internally to send packets asynchronously.
 */
7 usages  Ben Anderson
public class PacketSender implements AutoCloseable {

    //encapsulated packet queue which stores packets to be sent asynchronously
    4 usages
    private final Queue<Packet> packetQueue;
    //encapsulated asynchronous thread which packets are sent from
    3 usages
    private final Thread packetSendingThread;
    //private, thread-safe, atomic boolean variable to control the condition-controlled
    //loop in the packet sending thread.
    2 usages
    private final AtomicBoolean sendingPackets = new AtomicBoolean(initialValue: true);
```

Figure 109: *PacketSender* class (after fixes & edits) part 1

```
/*
 * Public API method to be used when a developer wishes to send a {@code Packet}
 * across the parent {@code Connection}. This method adds the provided {@code Packet} object
 * to the packet queue. The packet will then be sent across the {@code Connection} asynchronously.
 *
 * @param packet {@code Packet} to send across the connection
 */
2 usages  Ben Anderson
public void sendPacket(Packet packet) {
    //add packet to queue
    this.packetQueue.add(packet);
}

/*
 * Method overriding the default `close()` implementation from {@link AutoCloseable}.
 * This method disables the constant sending of packets on the asynchronous
 * 'Packet Sending Thread' and waits for the thread to die.
 *
 * @throws InterruptedException thrown if an exception occurs when waiting for the thread to die.
 */
Ben Anderson
@Override
public void close() throws InterruptedException {
    //set loop-controlling variable to false in order to disable the while loop
    //in the packet sending thread
    this.sendingPackets.set(false);
    //wait for the packet sending thread to die
    this.packetSendingThread.join();
}

}
```

Figure 111: *PacketSender class (after fixes & edits) part 3*

```
package xyz.benanderson.scs.server.networking;

import xyz.benanderson.scs.networking.connection.Connection;
import xyz.benanderson.scs.server.configuration.ConfigurationWrapper;

import java.io.IOException;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Collections;
import java.util.SortedMap;
import java.util.TreeMap;
import java.util.UUID;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

16 usages  ↳ Ben Anderson *
public class Server implements AutoCloseable {

    //attributes of the server
    3 usages
    private final ServerSocket serverSocket;
    3 usages
    private final Thread receiveConnectionsThread;
    7 usages
    private final SortedMap<UUID, Connection> connections;
    3 usages
    private final AtomicBoolean running;
    2 usages
    private final BiConsumer<Connection, Server> clientConnectListener;
    2 usages
    private final BiConsumer<Connection, Server> clientDisconnectListener;
    2 usages
    private final Consumer<Server> serverShutdownListener;
```

Figure 112: Server class (after fixes & edits) part 1

```
/**  
 * Constructor with package-private (default) visibility to allow only the {@code ServerBuilder}  
 * class to instantiate {@code Server} instances.  
 *  
 * @param serverBuilder builder to construct this {@code Server} from  
 */  
1 usage  ↗ Ben Anderson *  
Server(ServerBuilder serverBuilder) throws IOException {  
    //create a synchronized map to avoid race conditions in a multithreaded environment  
    //uses a TreeMap as a key-value store for connections, where the key is the  
    //Connection's identifier and the value is the Connection object.  
    //A TreeMap was used instead of a HashMap as O(1) retrieval is not essential as  
    //operations against this map will not occur often.  
    this.connections = Collections.synchronizedSortedMap(new TreeMap<>());  
    //mark the server as running  
    this.running = new AtomicBoolean(initialValue: true);  
    //assign attributes from serverBuilder...  
    int port = serverBuilder.getPort();  
    InetSocketAddress bindAddress = serverBuilder.getBindAddress();  
    this.clientConnectListener = serverBuilder.getClientConnectListener();  
    //append code to remove the Connection from the Server's internal connections map  
    //to the end of the client disconnect listener  
    BiConsumer<Connection, Server> closeListener = (con, server) -> {  
        try { connections.remove(con.getId()).close(); } catch (Exception ignored) {}  
    };  
    this.clientDisconnectListener = serverBuilder.getClientDisconnectListener().andThen(closeListener);  
    this.serverShutdownListener = serverBuilder.getServerShutdownListener();  
  
    this.serverSocket = new ServerSocket(port, backlog: 5, bindAddress);  
    this.receiveConnectionsThread = new Thread(this::receiveConnections);  
    this.receiveConnectionsThread.start();  
}  
  
1 usage  ↗ Ben Anderson  
public int getPort() {  
    return getServerSocket().getLocalPort();  
}
```

Figure 113: Server class (after fixes & edits) part 2

```

public ServerSocket getServerSocket() {
    synchronized (serverSocket) {
        return serverSocket;
    }
}

/**
 * @return boolean denoting if the {@code Server} is open and running.
 */
1 usage  ± Ben Anderson
public boolean isOpen() { return getServerSocket() != null && !getServerSocket().isClosed() && running.get(); }

/**
 * @return An unmodifiable SortedMap representing the {@code Connection}(s) to the server
 */
± Ben Anderson
public SortedMap<UUID, Connection> getConnections() { return Collections.unmodifiableSortedMap(connections); }

/**
 * Closes the {@code Server} and stops listening for new connections. The server shutdown
 * listener will be triggered by the receive-connections thread. All connections will be
 * closed and cleared from the {@code Server}.
 */
± Ben Anderson
@Override
public void close() {
    //attempt to stop listening for connections on the receive-connections thread
    running.set(false);
    try {
        getServerSocket().close();
    } catch (Exception ignored) {}
    //wait for the receive-connections thread to terminate (also waits for the
    //server shutdown listener to execute).
    try {
        receiveConnectionsThread.join();
    } catch (InterruptedException ignored) {}
    //closes all connections
    connections.values().forEach(con -> {
        try {
            con.close();
        } catch (Exception ignored) {}
    });
    //clear connections from memory of the server
    connections.clear();
}

```

Figure 114: Server class (after fixes & edits) part 3

```
//private-visibility method ran only by the receive-connections thread
1 usage  ▲ Ben Anderson
private void receiveConnections() {
    //condition-controlled loop to keep listening thread active as long as the
    //server is running and open
    while (isOpen()) {
        //don't accept new connections if already at max connections as specified in config
        if (connections.size() >= ConfigurationWrapper.getInstance().getMaxConnections())
            continue;
        try {
            //block until connection is accepted from the server's socket
            Socket socket = getServerSocket().accept();
            //instantiate Connection object (from connection framework) using
            //the connection's socket
            Connection connection = new Connection(socket);
            //add disconnect listener to the connection
            connection.setDisconnectListener(con -> clientDisconnectListener.accept(con, u: this));
            //add connection to server's map of connections
            connections.put(connection.getId(), connection);
            //run the client connection listener with the newly accepted connection
            clientConnectListener.accept(connection, u: this);
        } catch (IOException e) {
            //print error if an exception occurs
            e.printStackTrace();
        }
    }
    //run the server shutdown listener as isOpen() now returns false
    serverShutdownListener.accept(t: this);
}
```

Figure 115: Server class (after fixes & edits) part 4

```
package xyz.benanderson.scs.server.networking;

import lombok.Getter;
import xyz.benanderson.scs.networking.connection.Connection;

import java.io.IOException;
import java.net.InetAddress;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

/**
 * The {@code ServerBuilder} class allows a developer to construct a {@code Server} instance
 * and configure it prior to instantiation. Use {@link ServerBuilder#build()} to build a {@code Server}
 * from this {@code ServerBuilder}.
 */
7 usages  ↳ Ben Anderson
@Getter
public class ServerBuilder {

    //attributes that will be used by the server during construction
    1 usage
    private final int port;
    1 usage
    private final InetAddress bindAddress;
    2 usages
    private BiConsumer<Connection, Server> clientConnectListener;
    2 usages
    private BiConsumer<Connection, Server> clientDisconnectListener;
    2 usages
    private Consumer<Server> serverShutdownListener;
```

Figure 116: ServerBuilder class (after fixes & edits) part 1

```
/**  
 * @param port TCP port to run the networking server on  
 * @param bindAddress TCP address to run the networking server on  
 */  
1 usage  ↗ Ben Anderson  
public ServerBuilder(int port, InetAddress bindAddress) {  
    this.port = port;  
    this.bindAddress = bindAddress;  
    this.clientConnectListener = (con, serv) -> {};  
    this.clientDisconnectListener = (con, serv) -> {};  
    this.serverShutdownListener = serv -> {};  
}  
  
/**  
 * Runs after a client connects and the {@code Connection} is established  
 */  
1 usage  ↗ Ben Anderson  
public ServerBuilder onClientConnect(BiConsumer<Connection, Server> clientConnectListener) {  
    this.clientConnectListener = clientConnectListener;  
    return this;  
}  
  
/**  
 * Runs as the client disconnects, prior to the {@code Connection} closing  
 */  
1 usage  ↗ Ben Anderson  
public ServerBuilder onClientDisconnect(BiConsumer<Connection, Server> clientDisconnectListener) {  
    this.clientDisconnectListener = clientDisconnectListener;  
    return this;  
}
```

Figure 117: ServerBuilder class (after fixes & edits) part 2

```

/**
 * Runs as the server shuts down, prior to the {@code Connection}s closing
 */
1 usage  Ben Anderson
public ServerBuilder onServerShutdown(Consumer<Server> serverShutdownListener) {
    this.serverShutdownListener = serverShutdownListener;
    return this;
}

/**
 * Construct a {@code Server} from this {@code ServerBuilder}
 */
2 usages  Ben Anderson
public Server build() throws IOException {
    return new Server(serverBuilder: this);
}

}

```

Figure 118: ServerBuilder class (after fixes & edits) part 3

Here is a white box test table for the networking server submodule:

Class(es) being tested	How is the class being tested	Expected outcome from the class being tested	Was expected outcome achieved? (if no, what was the actual outcome?)
Server & Server-Builder	A ServerBuilder instance is created with callbacks attached, the Server instance is then built and a Connection attached to it. At each stage of the Connection lifecycle the status of the callbacks is checked.	After the Connection has connected, only the connect callback will have triggered. After the Connection disconnects, only the connect and disconnect callbacks will have triggered. After the server shuts down, all callbacks will have triggered.	Yes

Camera Viewer

CameraViewer Class

In order to access the webcam / video camera attached to the computer, I will use existing third-party libraries as they will already have built in support for different operating systems and camera drivers.

After some research, I came across an open source library called ‘webcam-capture’. This would allow me to do exactly what I needed, providing an intuitive API to open and capture images connected cameras.

I added the dependency to my server software’s `pom.xml` project file:

```
<dependency>
    <groupId>com.github.sarxos</groupId>
    <artifactId>webcam-capture</artifactId>
    <version>0.3.12</version>
</dependency>
```

Figure 119: Maven server software pom.xml file with webcam-capture dependency

I configured the dependency to be compiled into my final built executable so that the user wouldn’t need to install additional dependencies in order to run the server software.

I proceeded to develop a ‘CameraViewer’ class according to the second iteration design from the ‘Camera Viewer’ system design section.

```
package xyz.benanderson.scs.server.video;

import com.github.sarxos.webcam.Webcam;

import java.awt.image.BufferedImage;
import java.util.Optional;

4 usages  ↳ Ben Anderson *
public class CameraViewer implements AutoCloseable {

    4 usages
    private final Webcam camera;

    2 usages  ↳ Ben Anderson
    public CameraViewer(Webcam camera) {
        this.camera = camera;
        this.camera.open();
    }

    2 usages  ↳ Ben Anderson
    public Optional<BufferedImage> captureImage() {
        return Optional.ofNullable(camera.getImage());
    }

    ↳ Ben Anderson
    @Override
    public void close() {
        camera.close();
    }

}
```

Figure 120: CameraViewer class

The only difference between my design of the class and the implementation, is that I have chosen to only open the camera when the class is instantiated and close it when the class is no longer

used in my implementation, in comparison to the design where I open and close the camera every time I capture an image. I did this as it will decrease the processing time required to capture an image because the camera IO device will not need to be opened and closed for every image to be captured. Instead I only open it once and capture images until it is closed – this increases efficiency of the algorithm and provides a more optimised solution.

Testing The CameraViewer Class

I wrote and ran a quick test for the `CameraViewer` class:

```
package xyz.benanderson.scs.server.video;

import com.github.sarxos.webcam.Webcam;

import javax.swing.*;
import java.awt.image.BufferedImage;
import java.util.Optional;

/*
 * Ben Anderson *
 */

public class CameraViewerTest {

    /*
     * Ben Anderson *
     */

    public static void main(String[] args) {
        JFrame f = new JFrame();
        f.setVisible(true);
        f.setSize( width: 800, height: 480 );
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel label = new JLabel();
        f.getContentPane().add(label);

        try (CameraViewer cameraViewer = new CameraViewer(Webcam.getDefault())) {
            while (true) {
                Optional<BufferedImage> imageOptional = cameraViewer.captureImage();
                if (imageOptional.isEmpty()) continue;
                BufferedImage image = imageOptional.get();
                if (f.getWidth() != image.getWidth() || f.getHeight() != image.getHeight())
                    f.setSize(image.getWidth(), image.getHeight());
                label.setIcon(new ImageIcon(image));
            }
        }
    }
}
```

Figure 121: CameraViewerTest class

The expected outcome of this test was for a window to appear containing a constantly updating video feed from the webcam connected to the computer it was ran on.

When running the test, the expected outcome was achieved, a screenshot of the result is shown below:

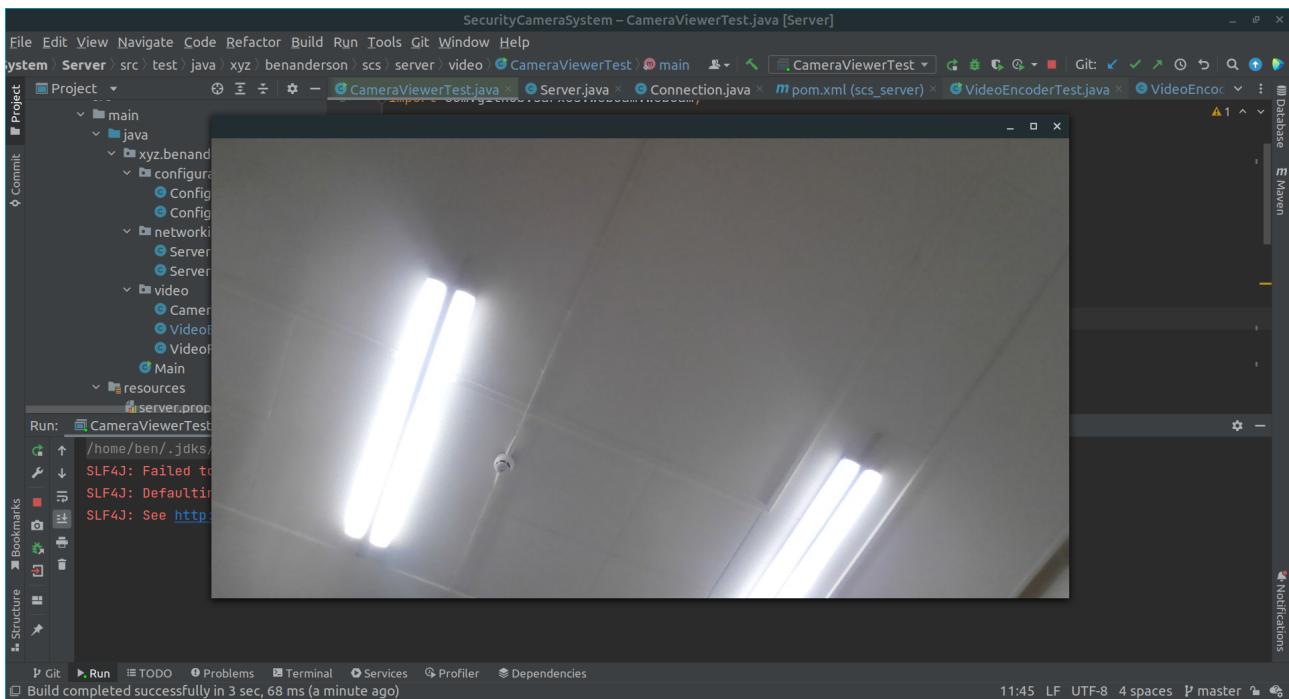


Figure 122: Outcome of running CameraViewerTest class

This confirmed that success criteria entry 13 had been achieved successfully. An additional note for this class is to bring attention to the return type of `CameraViewer#captureImage`. The function returns an `Optional` type[30], which is an object that may or may not contain a value – it is effectively a safe wrapper around a value that could be 'null', ensuring `NullPointerException`s aren't be thrown (assuming the class is used correctly).

Video Classes

Next, I implemented the `VideoFileManager` class, exactly following the design I had specified for it earlier.

```
package xyz.benanderson.scs.server.video;

import lombok.Getter;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Optional;

3 usages  ↳ Ben Anderson
public class VideoFileManager {

    2 usages
    private final Path saveDirectory;
    2 usages
    @Getter
    private final Duration videoDuration;
    4 usages
    private Path currentSaveFile;
    2 usages
    private LocalDateTime currentSaveFileExpiration;

    1 usage  ↳ Ben Anderson
    public VideoFileManager(Path saveDirectory, Duration videoDuration) {
        this.saveDirectory = saveDirectory;
        this.videoDuration = videoDuration;
        nextSaveFile();
    }
}
```

Figure 123: *VideoFileManager* class part 1

```
2 usages  Ben Anderson
private void nextSaveFile() {
    LocalDateTime currentDateTime = LocalDateTime.now();
    currentSaveFile = saveDirectory.resolve(currentDateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
    try {
        Files.createFile(currentSaveFile);
    } catch (IOException e) {
        System.err.println("[ERROR] Failed to create video save file.");
        e.printStackTrace();
    }
    currentSaveFileExpiration = currentDateTime.plus(videoDuration);
}

2 usages  Ben Anderson
public Optional<Path> getCurrentSaveFile() {
    if (currentSaveFileExpiration.isBefore(LocalDateTime.now())) {
        nextSaveFile();
    }
    return Files.exists(currentSaveFile) ? Optional.of(currentSaveFile) : Optional.empty();
}

}
```

Figure 124: VideoFileManager class part 2

This class does not have comments as they were already provided extensively in the design specification of the class, which this is the exact same as.

The only remaining component of the camera viewer system to be implemented was the `VideoEncoder` class, which would turn the captured images into a recorded video and save them to disk.

After some research, I came across an answer[18] on ‘Stackoverflow’ which suggested the use of a library called JCodec[10]. This seemed appealing as JCodec was one of the few video libraries for Java which was still being maintained and updated, furthermore, the code to use it seemed incredibly simple and intuitive.

I added the dependencies I would need for JCodec into my server module’s pom.xml file:

```

<dependency>
    <groupId>org.jcodec</groupId>
    <artifactId>jcodec</artifactId>
    <version>0.2.5</version>
</dependency>
<dependency>
    <groupId>org.jcodec</groupId>
    <artifactId>jcodec-javase</artifactId>
    <version>0.2.5</version>
</dependency>

```

Figure 125: Server Maven module pom.xml with JCodec dependencies

Below is my initial implementation of the `VideoEncoder` class using JCodec:

```

package xyz.benanderson.scs.server.video;

import lombok.AllArgsConstructor;
import org.jcodec.api.awt.AWTSequenceEncoder;

import java.awt.image.BufferedImage;
import java.io.IOException;
import java.nio.file.Path;
import java.util.Optional;

2 usages  ↳ Ben Anderson *
@AllArgsConstructor
public class VideoEncoder {

    1 usage
    private final VideoFileManager videoFileManager;

    1 usage  ↳ Ben Anderson *
    public void appendToStream(BufferedImage image) throws IOException {
        Optional<Path> currentSaveFileOptional = videoFileManager.getCurrentSaveFile();
        if (currentSaveFileOptional.isEmpty()) {
            System.err.println("[ERROR] Failed to fetch video save file.");
            return;
        }
        Path currentSaveFile = currentSaveFileOptional.get();
        AWTSequenceEncoder encoder = AWTSequenceEncoder.createSequenceEncoder(currentSaveFile.toFile(), fps: 30);
        encoder.encodeImage(image);
    }
}

```

Figure 126: VideoEncoder class

Testing The Video Classes

I also produced the `VideoEncoderTest` class, to test the `VideoEncoder` class.

```
package xyz.benanderson.scs.server.video;

import com.github.sarxos.webcam.Webcam;

import java.io.IOException;
import java.nio.file.Paths;
import java.time.Duration;

/*
 * Ben Anderson *
 */

public class VideoEncoderTest {

    /*
     * Ben Anderson *
     */
    public static void main(String[] args) {
        //instantiate CameraViewer instance
        try (CameraViewer cameraViewer = new CameraViewer(Webcam.getDefault())) {
            //instantiate VideoFileManager instance to define
            //the video save directory and video duration
            VideoFileManager videoFileManager = new VideoFileManager(Paths.get(first: "/home/ben/Videos/"),
                Duration.ofMinutes(1));
            //instantiate VideoEncoder instance to be tested
            VideoEncoder videoEncoder = new VideoEncoder(videoFileManager);
            //capture 100 images from the CameraViewer and write each of them
            //to the VideoEncoder instance straight after capturing them
            //(order is 'capture', 'write', 'capture', 'write', etc.)
            for (int i = 0; i < 100; i++) {
                cameraViewer.captureImage().ifPresent(image -> {
                    try {
                        videoEncoder.appendToStream(image);
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                });
            }
        }
    }
}
```

Figure 127: *VideoEncoderTest* class

Running the test lasted approximately 8 seconds. Afterwards, I saw an output file generated (although it seemed abnormally small to be a video).

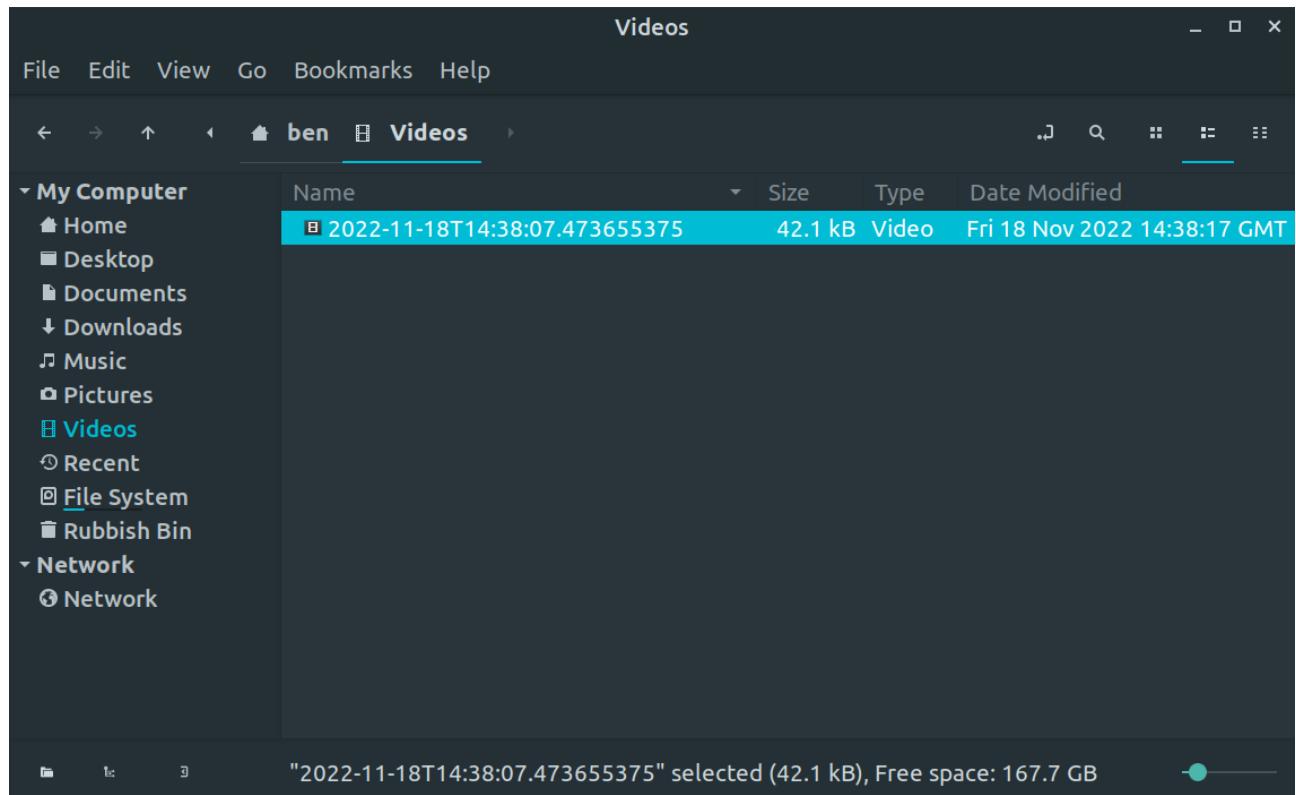


Figure 128: *VideoEncoderTest* initial test file saved

However, when trying to play the video, an error occurred:

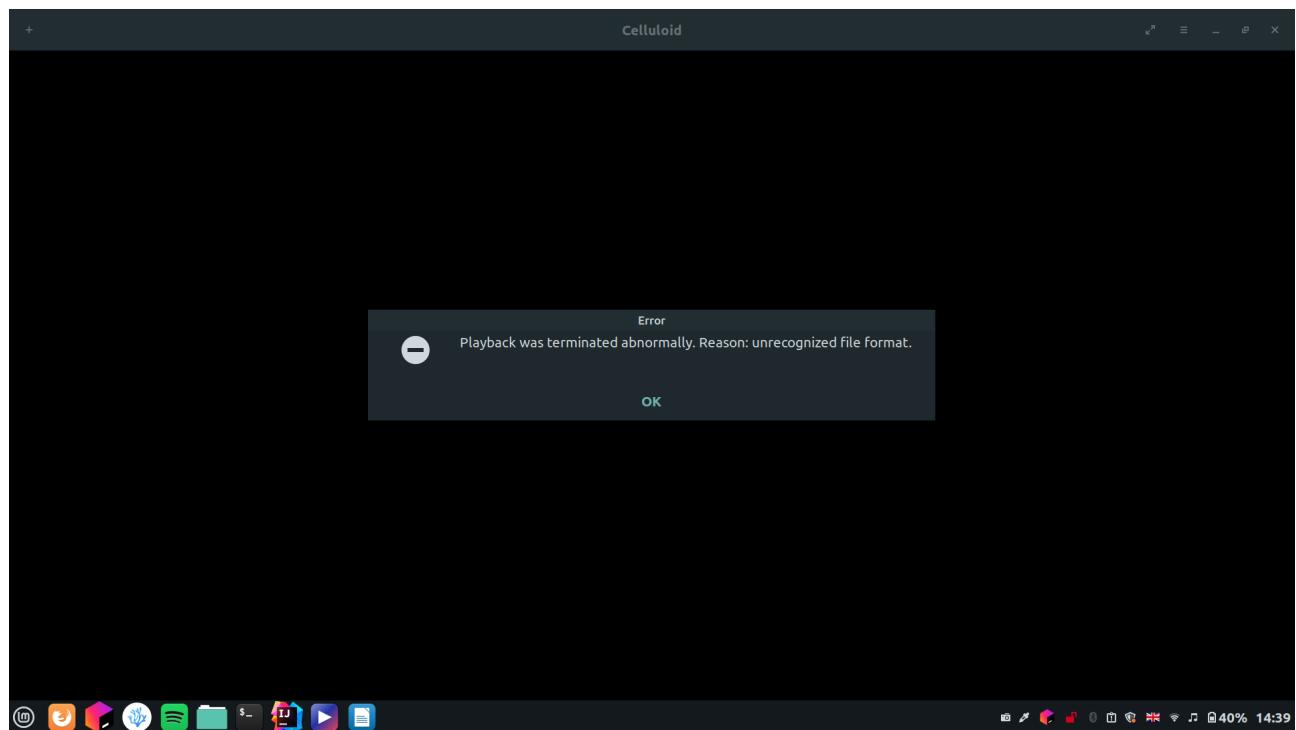


Figure 129: *VideoEncoderTest* initial test file playback error

Due to the fact that the video file was so small, I had the suspicion that the file may be being overwritten with each camera frame, thus deleting all the previous data. To test this theory, I added some code to the test to output the file size in kilobytes after each camera frame was written to it.

```
package xyz.benanderson.scs.server.video;

import com.github.sarxos.webcam.Webcam;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.time.Duration;

▀ Ben Anderson *
public class VideoEncoderTest {

    ▀ Ben Anderson *
    public static void main(String[] args) {
        //instantiate CameraViewer instance
        try (CameraViewer cameraViewer = new CameraViewer(Webcam.getDefault())) {
            //instantiate VideoFileManager instance to define
            //the video save directory and video duration
            VideoFileManager videoFileManager = new VideoFileManager(Paths.get(first: "/home/ben/Videos/"),
                Duration.ofMinutes(1));
            //instantiate VideoEncoder instance to be tested
            VideoEncoder videoEncoder = new VideoEncoder(videoFileManager);
            //capture 100 images from the CameraViewer and write each of them
            //to the VideoEncoder instance straight after capturing them
            //((order is 'capture', 'write', 'capture', 'write', etc.)
            for (int i = 0; i < 100; i++) {
                cameraViewer.captureImage().ifPresent(image -> {
                    try {
                        videoEncoder.appendToStream(image);
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                });
                //output current video file size in KB (Kilobytes)
                videoFileManager.getCurrentSaveFile().ifPresent(file -> {
                    try {
                        System.out.println(Files.size(file) / 1_000d);
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                });
            }
        }
    }
}
```

Figure 130: *VideoEncoderTest* class

Running this code resulted in the following output:

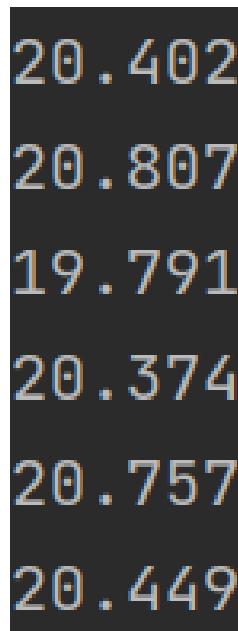


Figure 131: *VideoEncoderTest* file size output
(truncated)

The output clearly showed that the file size stayed effectively the same after each frame was written to it, and in some cases even decreased. This led me to the conclusion that the file was being overwritten with each frame.

At this point, I decided that I had spent long enough developing the `VideoEncoder` class for this development iteration as it wasn't an essential core feature and I was having problems with my current approach. Therefore, I decided to re-approach this feature in my second development iteration and leave it for now.

Now the only remaining task for the server software, is to combine all the submodules and write some code to run the server and serve camera frames to the user. As you may notice, I have not yet developed the account system submodule, this is because I don't feel it is essential for the first prototype, hence will implement it later.

Combining Server Submodules

I wrote a `Main` class for the server software containing the entry-point of the application. It is responsible for starting the networking server, using the camera viewer to capture images, and sending these images to all connected clients.

```

package xyz.benanderson.scs.server;

import com.github.sarxos.webcam.Webcam;
import xyz.benanderson.scs.networking.Packet;
import xyz.benanderson.scs.networking.packets.MediaPacket;
import xyz.benanderson.scs.server.configuration.ConfigurationWrapper;
import xyz.benanderson.scs.server.networking.Server;
import xyz.benanderson.scs.server.networking.ServerBuilder;
import xyz.benanderson.scs.server.video.CameraViewer;

import java.io.IOException;
import java.net.InetAddress;

1 usage  ↳ Ben Anderson *
public class Main {

    ↳ Ben Anderson *
    public static void main(String[] args) throws IOException {
        //create basic ServerBuilder without any additional configuration
        ServerBuilder serverBuilder = new ServerBuilder(ConfigurationWrapper.getInstance().getServerPort(),
            InetAddress.getByName(ConfigurationWrapper.getInstance().getServerAddress()));
        //build server and open camera
        try (Server server = serverBuilder.build();
            CameraViewer cameraViewer = new CameraViewer(Webcam.getDefault())) {
            while (true) {
                //attempt to capture camera image
                cameraViewer.captureImage().ifPresent(img -> {
                    //if successful in capturing an image, create a packet from the image
                    //and send it to all active connections
                    Packet packet = new MediaPacket(img);
                    server.getConnections().values().forEach(conn -> {
                        conn.getPacketSender().sendPacket(packet);
                    });
                });
            }
        }
    }
}

```

Figure 132: Main class server software entry-point

The yellow highlighting over the token ‘while’ is done by the IDE to indicate that the while loop will run indefinitely and not gracefully terminate, however for this program it is suitable and is not of concern.

I ran the above code and profiled it using a built-in profiling tool of my IDE. The below diagram shows that the code had fairly consistent and reasonable CPU & RAM usage, hence I do not believe there are any performance bottlenecks or memory leaks in the application.

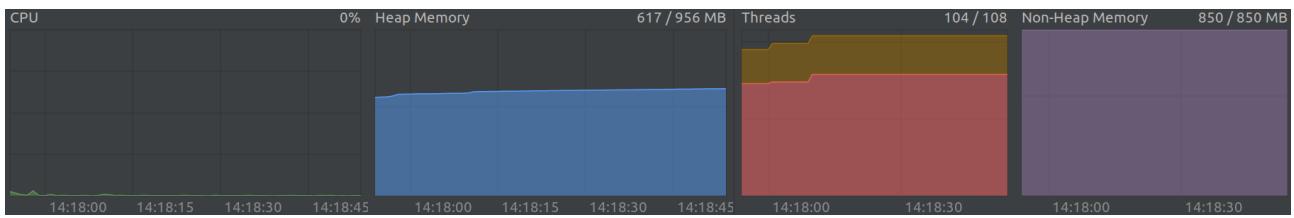


Figure 133: Server software profiling first prototype

Client Software

Due to the fact that the first development iteration mainly focused on building a concrete, durable server and networking library, the client software in this version is going to be very minimal - only hoping to act as a functional receiver and displayer for media frames.

Here is the code for the implemented entry-point of the client software:

```
package xyz.benanderson.scs.client;

import xyz.benanderson.scs.networking.connection.Connection;
import xyz.benanderson.scs.networking.packets.MediaPacket;

import javax.swing.*;
import java.io.IOException;
import java.net.Socket;

public class Main {

    public static void main(String[] args) throws IOException {
        JFrame f = new JFrame();
        f.setVisible(true);
        f.setSize( width: 800, height: 480 );
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel label = new JLabel();
        f.getContentPane().add(label);

        Socket socket = new Socket( host: "127.0.0.1", port: 8192 );
        Connection connection = new Connection(socket);
        connection.getPacketListener().addCallback(MediaPacket.class, mediaPacket -> {
            label.setIcon(new ImageIcon(mediaPacket.getMediaFrame()));
        });
    }
}
```

Figure 134: Main class client software entry-point

However, when running the code I encountered the following error:

```
[ERROR] An error occurred whilst sending a packet across a connection:  
java.io.NotSerializableException Create breakpoint : java.awt.image.BufferedImage  
    at java.base/java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1192)  
    at java.base/java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1577)  
    at java.base/java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1534)  
    at java.base/java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1443)  
    at java.base/java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1186)  
    at java.base/java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:352)  
    at xyz.benanderson.scs.networking.connection.PacketController.writePacketToSocket(PacketController.java:68)  
    at xyz.benanderson.scs.networking.connection.PacketSender.lambda$new$0(PacketSender.java:51) <1 internal line>
```

Figure 135: Initial integration test error - first prototype

If we read the error message, we can clearly see that the error is caused by the fact that the `BufferedImage`'s from the `MediaPacket` objects aren't serializable. After some research, I came across a solution[37] on StackOverflow and implemented it into the `MediaPacket` class in the networking module:

```
package xyz.benanderson.scs.networking.packets;

import lombok.Getter;
import xyz.benanderson.scs.networking.Packet;

import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

/**
 * Packet sent from a server to a client. This packet contains an image
 * which is the media frame/image of the security camera at a point in time.
 */
5 usages  Ben Anderson *
public class MediaPacket extends Packet {

    /**
     * Media frame/image that makes up the main content packet
     */
    3 usages
    @Getter
    private transient BufferedImage mediaFrame;
```

Figure 136: MediaPacket class (after fix) part 1

```
/*
 * Constructor for {@code MediaPacket} class
 *
 * @param mediaFrame BufferedImage representing the media frame/image
 */
1 usage  Ben Anderson
public MediaPacket(BufferedImage mediaFrame) {
    //set `type` property in the superclass to `MediaPacket.class`
    super(MediaPacket.class);
    //assign instance property to constructor parameter
    this.mediaFrame = mediaFrame;
}

new *
private void writeObject(ObjectOutputStream out) throws IOException {
    out.defaultWriteObject();
    ImageIO.write(mediaFrame, formatName: "jpg", out);
}

new *
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    mediaFrame = ImageIO.read(in);
}

}
```

Figure 137: *MediaPacket* class (after fix) part 2

It's worth bringing attention to the fact that when serializing the image using `ImageIO` (a class from the Java standard library), I used the 'jpg' format (JPEG). This image format utilises lossy compression whilst maintaining a reasonable image quality. The reason I chose to use lossy compression and not lossless, is that a slightly worse image quality is suitable for a security camera. Furthermore, JPEG compression achieves a higher compression ratio than lossless compression techniques such as PNG, resulting in more efficient use of network bandwidth as I'll be able to send more media frame in the same number of bytes across the network. This provides a user experience which is closer to a 'real-time' security camera.

End-To-End Testing First Prototype

Running the first server software prototype, followed by the first client software prototype, resulted in a successful test with the camera being streamed to the client software application.

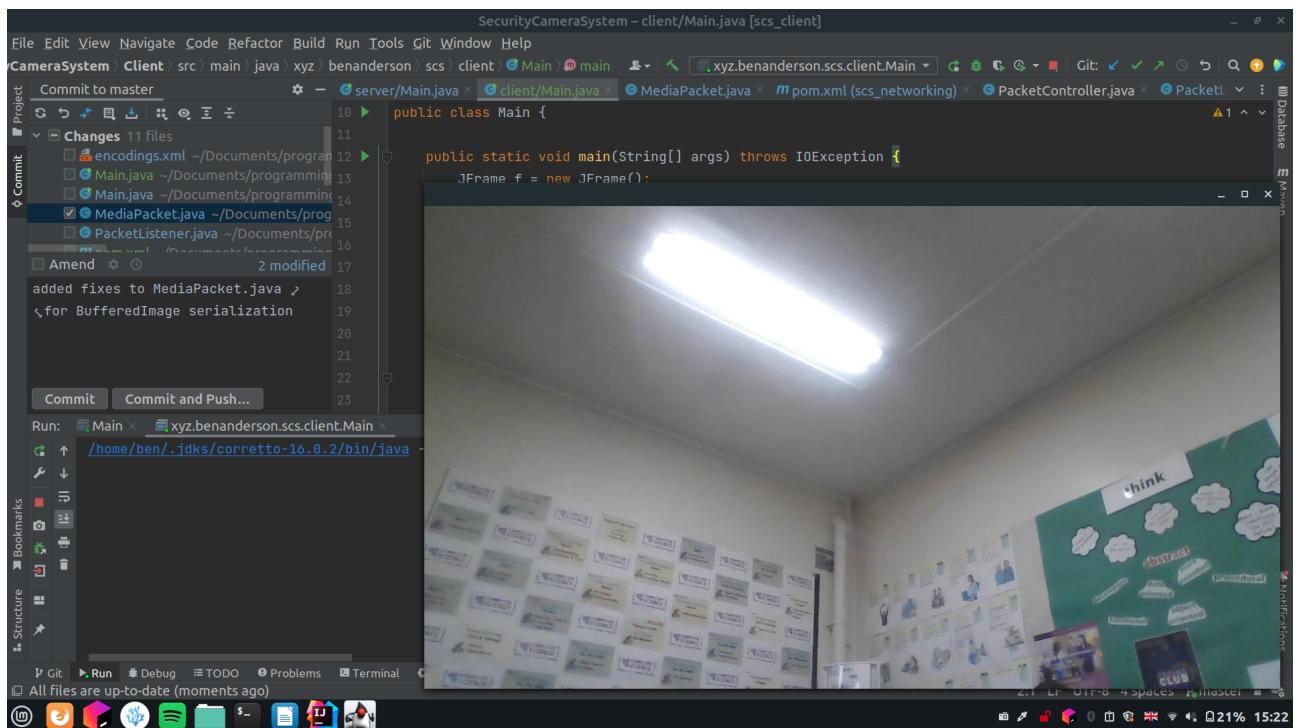


Figure 138: First prototype successful integration test

Class(es) being tested	How is the class being tested	Expected outcome from the class being tested	Was expected outcome achieved? (if no, what was the actual outcome?)
Packet-Sender	A PacketSender instance is created and a test packet sent to it using PacketSender#sendPacket. The internal packet queue is then inspected.	The internal packet queue contains the test packet.	Yes
Packet-Sender	A PacketSender instance is created and attached to a Connection. A test packet is sent using PacketSender#sendPacket. Sent/received data is checked at the attached Connection.	The attached connection receives the test packet.	Yes
PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there are no callbacks registered.	Nothing happens, including no errors.	Yes
PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there is a single callback registered. The status of the single callback will be checked.	The single callback runs and consumes the test packet.	Yes

PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there is a single callback registered but for a different packet type. The status of the single callback will be checked.	The single callback doesn't run.	Yes
PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there is one callback registered for the correct type but also one registered for a different packet type. The status of both callbacks will be checked.	The callback of the correct type runs but the other one doesn't run.	Yes
PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there are two callbacks registered for the correct type. The status of both callbacks will be checked.	Both callbacks run.	Yes
PacketListener	A PacketListener instance is created with a callback registered for the 'TestPacket' type and attached to a Connection. A test packet is then sent to the Connection and status of the registered callback is checked.	The callback runs.	Yes
PacketController	A PacketController instance is created and attached to a Connection. The PacketController writes a test packet to the Connection and the Connection is checked to see if the packet has been received and is equal to the initially written test packet.	The test packet is received by the Connection and is equal to what was written to it.	Yes
PacketController	A PacketController instance is created and attached to a Connection. The Connection writes a test packet to the PacketController and the PacketController attempts to read it. The received and read packet from the PacketController is checked for equality with the initially sent test packet.	The test packet is received by the PacketController and is equal to what was written to it.	Yes
ConfigurationWrapper	The default configuration is loaded by the ConfigurationWrapper and its values checked against hard-coded defaults.	All values will match.	Yes
Server & Server-Builder	A ServerBuilder instance is created with callbacks attached, the Server instance is then built and a Connection attached to it. At each stage of the Connection lifecycle the status of the callbacks is checked.	After the Connection has connected, only the connect callback will have triggered. After the Connection disconnects, only the connect and disconnect callbacks will have triggered. After the server shuts down, all callbacks will have triggered.	Yes

Stakeholder Black Box Testing

After finishing the first prototype, I gave it to my stakeholder so that they could perform black-box testing on the system and provide feedback.

Test input field	Test input data	Expected outcome	Was expected outcome achieved? (if no, what was the actual outcome?)
server.address	fwehc8sd9	Exception is thrown informing the user that it is not a valid host to run the server on.	Yes
server.address	1.2.3.4	Exception is thrown informing the user that they cannot use the requested address.	Yes
server.address	127.0.0.1	Server starts successfully and is only accessible from the current computer (because 127.0.0.1 is the localhost interface).	Yes
server.address	0.0.0.0	Server starts successfully and is accessible by any machine on the local network.	Yes
server.port	the port is five five five five	Warning is displayed stating the acceptable port range and that the server has been started on a random port. The server is started on a random port.	Yes
server.port	70000	Warning is displayed stating the acceptable port range and that the server has been started on a random port. The server is started on a random port.	Yes
server.port	65536	Warning is displayed stating the acceptable port range and that the server has been started on a random port. The server is started on a random port.	Yes
server.port	0	Server starts on a random port (chosen by OS).	Yes
server.port	-1	Warning is displayed stating the acceptable port range and that the server has been started on a random port. The server is started on a random port.	Yes
server.port	80	If they aren't running with elevated privileges, an exception is thrown informing the user that they don't have the required permissions to run the camera server on that port. But if they are running with elevated privileges, the server starts on port 80.	Yes
server.max-connections	-1	Camera server starts successfully but no clients can connect to it.	Yes
server.max-	0	Camera server starts successfully but no cli-	Yes

connections		ents can connect to it.	
server.max-connections	1	Camera server starts successfully but only one client can connect to it.	Yes

After testing the system, they came back to me with the following points:

1. *"The system worked well to say it is a first prototype, everything functioned as expected. The application streamed the camera in real time across my network."*
2. *"Installation of the system was fairly simple as it was just one file with no additional dependencies."*
3. *"I would like to increase security by requiring a login to view the camera."*
4. *"I want to be able to interact with the client application, currently it is just a streaming window. I want to be able to choose the camera to view and login – all through the GUI."*

Feedback point 1 and my own testing verifies that success criteria points 2, 3, 4, 5, 7, 11, 13 and 16 have all been achieved (11 can be verified because I developed and ran the software on the Linux Mint operating system whereas my stakeholder ran the software on Microsoft Windows).

Feedback point 2 shows strong initial ease-of-installation, verifying success criteria point 1. Whilst I haven't provided a proper installer to the user yet, it may appear that I don't need to – at least not yet. This is mainly due to the fact that I don't use any external assets or unpackaged dependencies in either my client or server software, hence the software is very easy to install.

Feedback point 3 notes a specific addition that the user would like to the current prototype. Luckily, the account system is a submodule which I have already designed and was aware the stakeholder would want, however I just chose not to include it in the first prototype. Therefore, a clear development goal for the second prototype would be to implement the account system – this would complete success criteria points 8 and 9 when verified working.

Feedback point 4 shows a clear desire for more user interface options on the client software. This is also something I plan on adding and have already designed. I plan on adding this in either the second or third iteration as it will be a big task.

In response to feedback point 4, I needed to be sure that the user interface I had designed was going to meet the stakeholder's requirements. Therefore, I showed them the design diagrams for the graphical user interface in attempt to understand if they would be happy with this design. They expressed happiness towards the graphical user interface design and sound they would be happy for the GUI to function as detailed in the design documentation.

Second Prototype

After establishing the account system as a clear requirement for the second prototype, I began to implement my already-designed solution for the submodule.

Account System

I first added the classes to my code according to the UML class diagrams defined in the design section, with the only difference being that 'UserManager' was implemented as an interface in or-

der to be able to define common behaviour among an array of different `UserManager` implementations (which I would place in the `managers` subpackage). This would mean that I could swap out `UserManager`'s interchangeably at a later date without causing any errors elsewhere in the code-base.

```
package xyz.benanderson.scs.server.account;

import lombok.Data;
import lombok.EqualsAndHashCode;
import xyz.benanderson.scs.networking.packets.LoginPacket;

11 usages  ↳ Ben Anderson *
@Data
@EqualsAndHashCode
public class User {

    1 usage
    private final String username, hashedPassword;

    1 usage
    private final boolean admin;

    1 usage  ↳ Ben Anderson
    public static User fromHashedPassword(String username, String hashedPassword, boolean admin) {
        return new User(username, hashedPassword, admin);
    }

    no usages  ↳ Ben Anderson *
    public static User fromPlainTextPassword(String username, String plainTextPassword, boolean admin) {
        return new User(username, LoginPacket.hashPassword(plainTextPassword), admin);
    }

    2 usages  ↳ Ben Anderson
    private User(String username, String hashedPassword, boolean admin) {
        this.username = username;
        this.hashedPassword = hashedPassword;
        this.admin = admin;
    }
}
```

Figure 139: User class

```
package xyz.benanderson.scs.server.account;

import java.util.Optional;

2 usages 1 implementation Ben Anderson *
public interface UserManager {

    no usages 1 implementation Ben Anderson
    Optional<User> getUser(String username);

    no usages 1 implementation Ben Anderson
    void createUser(User user) throws Exception;

    no usages 1 implementation Ben Anderson
    void deleteUser(String username) throws Exception;

}
```

Figure 140: *UserManager interface*

Hashing

As well as the above, given that the user was wanting to increase security with the account system, I decided it would also be necessary to hash the password of the user. Hashing is a one-way algorithm that produces a hash digest which cannot be reverted into the original data; given the same input data it will always give the same output data. Hashing is commonly used when storing passwords and helps companies to comply with the Data Protection Act – as to not expose a user's plain-text password if the company's database is hacked.

The hashing algorithm I chose was SHA-256 as it is considered quite security and is the industry standard.

Hashing was implemented by providing two public, static, factory methods in the 'User' class ('User#fromPlainTextPassword' and 'User#fromHashedPassword') which create 'User' objects, and by restricting access to the constructor by marking it as 'private'. This allowed me to clearly differentiate when creating 'User' objects, whether the given password was already hashed or not.

Furthermore, I added the publicly available static method 'LoginPacket.hashPasssword' to the 'LoginPacket' class in order to provide a common method for hashing a plain-text password into a hexadecimal string. I placed this code in the 'LoginPacket' class as it would be accessible to both the client and server software, as they would both need to access this code.

The code found in the `LoginPacket.hashPassword` method is a combination of code found on a StackOverflow post[15] about hashing in Java and a blog post[20] about converting a byte array to a hexadecimal string.

```
package xyz.benanderson.scs.networking.packets;

import lombok.AccessLevel;
import lombok.Getter;
import xyz.benanderson.scs.networking.Packet;

import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;

/**
 * Packet send from a client to a server when attempting to authenticate.
 */
7 usages  ↳ Ben Anderson *
public class LoginPacket extends Packet {

    /**
     * Username that the connection is attempting to authenticate with
     */
    1 usage
    @Getter(AccessLevel.PUBLIC)
    private final String username;

    /**
     * Password that the connection is attempting to authenticate with
     */
    1 usage
    @Getter(AccessLevel.PUBLIC)
    private final String hashedPassword;
```

Figure 141: LoginPacket class with hashing considerations part 1

```
/**  
 * Constructor for {@code LoginPacket} class  
 *  
 * @param username username of user to authenticate as  
 * @param hashedPassword password to use when authenticating  
 */  
2 usages  ↳ Ben Anderson *  
private LoginPacket(String username, String hashedPassword) {  
    //set `type` property in the superclass to `LoginPacket.class`  
    super(LoginPacket.class);  
    //assign object properties to constructor parameters  
    this.username = username;  
    this.hashedPassword = hashedPassword;  
}  
  
no usages  new *  
public static LoginPacket fromPlainTextPassword(String username, String plainTextPassword) {  
    return new LoginPacket(username, hashPassword(plainTextPassword));  
}  
  
no usages  new *  
public static LoginPacket fromHashedPassword(String username, String hashedPassword) {  
    return new LoginPacket(username, hashedPassword);  
}  
  
2 usages  new *  
public static String hashPassword(String plainTextPassword) {  
    try{  
        final MessageDigest digest = MessageDigest.getInstance(algorithm: "SHA-256");  
        final byte[] hash = digest.digest(plainTextPassword.getBytes(StandardCharsets.UTF_8));  
        final StringBuilder hexString = new StringBuilder();  
        for (byte b : hash) {  
            final String hex = Integer.toHexString(~0xff & b);  
            if (hex.length() == 1)  
                hexString.append('0');  
            hexString.append(hex);  
        }  
        return hexString.toString();  
    } catch(Exception ex){  
        throw new RuntimeException(ex);  
    }  
}  
}  
}
```

Figure 142: LoginPacket class with hashing considerations part 2

UserManager

I then proceeded to develop a `MultiFileUserManager` implementation based off the design of the `UserManager` class. Whilst developing the solution, I realised that using a binary search added unnecessary complexity to the code and increased the time taken to carry out `UserManager` operations in most cases. Due to the fact that I had already designed the class to save user accounts as files where the file name was the user's username, I realised that I had effectively emulated an O(1) time complexity `Map` data structure on the secondary storage – where the username was the key and the user data was the value. This allowed me to optimise methods such as `UserManager#getUser` to provide O(1) time complexity as it simply attempted to retrieve the user data file using the username, if it didn't exist it would return an empty optional.

The final implementation of the `UserManager` interface according to the design specification and taking into account the above adaptations was created in the class `MultiFileUserManager`, stored in the `managers` subpackage under the `account` package.

```
package xyz.benanderson.scs.server.account.managers;

import xyz.benanderson.scs.server.account.User;
import xyz.benanderson.scs.server.account.UserManager;
import xyz.benanderson.scs.server.configuration.ConfigurationWrapper;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
import java.util.Optional;

no usages  ↳ Ben Anderson
public class MultiFileUserManager implements UserManager {

    no usages  ↳ Ben Anderson
    @Override
    public Optional<User> getUser(String username) {
        Path userSaveFile = ConfigurationWrapper.getInstance().getUsersSaveDirectory()
            .resolve(username);
        if (!Files.exists(userSaveFile)) {
            return Optional.empty();
        }
        try {
            List<String> lines = Files.readAllLines(userSaveFile);
            User parsedUser = User.fromHashedPassword(lines.get(0),
                lines.get(1),
                Boolean.parseBoolean(lines.get(2)));
            return Optional.of(parsedUser);
        } catch (Exception e) {
            return Optional.empty();
        }
    }
}
```

Figure 143: MultiFileUserManager class part 1

```
no usages  ↳ Ben Anderson
@Override
public void createUser(User user) throws IOException {
    //create directory if it does not exist
    Path usersSaveDirectory = ConfigurationWrapper.getInstance().getUsersSaveDirectory();
    Files.createDirectories(usersSaveDirectory);
    //creates and writes user data to file (automatically closes it)
    Path userSaveFile = usersSaveDirectory.resolve(user.getUsername());
    Files.writeString(userSaveFile, csq: user.getUsername() + System.lineSeparator()
        + user.getHashedPassword() + System.lineSeparator()
        + user.isAdmin());
}

no usages  ↳ Ben Anderson
@Override
public void deleteUser(String username) throws IOException {
    Path userSaveFile = ConfigurationWrapper.getInstance().getUsersSaveDirectory()
        .resolve(username);
    Files.deleteIfExists(userSaveFile);
}

}
```

Figure 144: MultiFileManager class part 2

To test this class, I created a `MultiFileManagerTest` class:

```
package xyz.benanderson.scs.server.account;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import xyz.benanderson.scs.server.account.managers.MultiFileUserManager;
import xyz.benanderson.scs.server.configuration.ConfigurationWrapper;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
import java.util.Optional;

import static org.junit.jupiter.api.Assertions.*;

no usages  ↳ Ben Anderson
public class MultiFileManagerTest {

    4 usages
    static UserManager userManager = new MultiFileManager();
    2 usages
    static Path usersFolder = ConfigurationWrapper.getInstance().getUsersSaveDirectory();
    10 usages
    static Path userFile = ConfigurationWrapper.getInstance().getUsersSaveDirectory()
        .resolve("testUser".toLowerCase());

    no usages  ↳ Ben Anderson
    @BeforeAll
    static void createUsersFolder() throws IOException {
        Files.createDirectories(usersFolder);
    }

    no usages  ↳ Ben Anderson
    @AfterEach
    void deleteUserFile() throws IOException {
        Files.deleteIfExists(userFile);
    }
}
```

Figure 145: MultiFileManagerTest class part 1

```

@AfterAll
static void deleteUsersFolder() throws IOException {
    Files.deleteIfExists(userFile);
    Files.deleteIfExists(usersFolder);
}

no usages  ↳ Ben Anderson
@Test
public void test GetUserNotExists() { assertTrue(userManager.getUser( username: "noUser").isEmpty()); }

no usages  ↳ Ben Anderson
@Test
public void test GetUserExists() throws IOException {
    Files.writeString(userFile, cs: "testUser\nhashedPassword\nfalse");
    Optional<User> userOptional = userManager.getUser( username: "testUser");
    assertTrue(userOptional.isPresent());
    assertEquals( expected: "testUser", userOptional.get().getUsername());
    assertEquals( expected: "hashedPassword", userOptional.get().getHashedPassword());
    assertFalse(userOptional.get().isAdmin());
}

no usages  ↳ Ben Anderson
@Test
public void test CreateUser() throws Exception {
    assertFalse(Files.exists(userFile));
    User user = User.fromHashedPassword( username: "testUser", hashedPassword: "hashedPassword", admin: false);
    userManager.createUser(user);
    assertTrue(Files.exists(userFile));
    List<String> lines = Files.readAllLines(userFile);
    assertEquals( expected: "testUser", lines.get(0));
    assertEquals( expected: "hashedPassword", lines.get(1));
    assertFalse(Boolean.parseBoolean(lines.get(2)));
}

no usages  ↳ Ben Anderson
@Test
public void deleteUser() throws Exception {
    assertFalse(Files.exists(userFile));
    Files.writeString(userFile, cs: "testUser\nhashedPassword\nfalse");
    assertTrue(Files.exists(userFile));
    userManager.deleteUser( username: "testUser");
    assertFalse(Files.exists(userFile));
}

}

```

Figure 146: MultiFileManagerTest class part 2

Running the test yielded a successful result and indicated that the `MultiFileManager` class was working as expected.

```
[INFO] Running xyz.benanderson.scs.server.account.MultiFileManagerTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.024 s
```

Figure 147: MultiFileManagerTest class passing tests

Class(es) being tested	How is the class being tested	Expected outcome from the class being tested	Was expected outcome achieved? (if no, what was the actual outcome?)
Multi-FileManager	The users folder is created. A MultiFileManager instance is created and a user that does not exist is queried, the result is checked.	An empty result is returned.	Yes
Multi-FileManager	The users folder is created and a file for a user is added. A MultiFileManager instance is created and a user that does exist is queried, the result is checked.	A result containing the user data is returned.	Yes
Multi-FileManager	The users folder is created. A MultiFileManager instance is created and a user is created, the user file is then checked.	The file contains the correct user data.	Yes
Multi-FileManager	The users folder is created and a file for a user is added. A MultiFileManager instance is created and the existing user is deleted. The user file is then checked if it is present.	The file containing the user data has been deleted and is no longer present.	Yes

Authentication Flow

The final part of the authentication code to implement, was the actual authorization flow that the user would experience. The majority of this was implemented in the 'Main' class of the server module, with a method being added to configure the authentication flow on a given 'ServerBuilder'. In addition to this, I used a global, thread-safe 'Set' (data structure similar to a list but backed by a 'HashTable') to store a list of currently authenticated users, which is checked every time a camera frame is being broadcast.

I used a 'Set' here as there is an O(1) time complexity for checking if an element exists in the 'Set', which would provide the most optimal solution as I need to check if each 'Connection' is authenticated every time I send a packet.

Below is the new version of the server module's entry-point:

```
package xyz.benanderson.scs.server;

import com.github.sarxos.webcam.Webcam;
import xyz.benanderson.scs.networking.Packet;
import xyz.benanderson.scs.networking.packets.DisconnectPacket;
import xyz.benanderson.scs.networking.packets.InfoPacket;
import xyz.benanderson.scs.networking.packets.LoginPacket;
import xyz.benanderson.scs.networking.packets.MediaPacket;
import xyz.benanderson.scs.server.account.User;
import xyz.benanderson.scs.server.account.UserManager;
import xyz.benanderson.scs.server.account.managers.MultiFileUserManager;
import xyz.benanderson.scs.server.configuration.ConfigurationWrapper;
import xyz.benanderson.scs.server.networking.Server;
import xyz.benanderson.scs.server.networking.ServerBuilder;
import xyz.benanderson.scs.server.video.CameraViewer;

import java.io.IOException;
import java.net.InetAddress;
import java.util.*;

4 usages  ↳ Ben Anderson *
public class Main {

    no usages  ↳ Ben Anderson *
    public static void main(String[] args) throws IOException {
        ServerBuilder serverBuilder = new ServerBuilder(ConfigurationWrapper.getInstance().getServerPort(),
            InetAddress.getByName(ConfigurationWrapper.getInstance().getServerAddress()));
        UserManager userManager = new MultiFileUserManager();
        addAuthenticationToServerBuilder(serverBuilder, userManager);

        //todo demo test then remove / or ship with preset creds / or allow set creds in config
        try {
            userManager.createUser(User.fromPlainTextPassword(username: "testUsername",
                LoginPacket.hashPassword(plainTextPassword: "testPassword"), admin: false));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Figure 148: Server module Main class part 1

```

//build server and open camera
try (Server server = serverBuilder.build();
    CameraViewer cameraViewer = new CameraViewer(Webcam.getDefault())) {
    while (true) {
        //attempt to capture camera image
        cameraViewer.captureImage().ifPresent(img -> {
            //if successful in capturing an image, create a packet from the image
            //and send it to all active connections
            Packet packet = new MediaPacket(img);
            server.getConnections().values().forEach(conn -> {
                if (LoggedInUsers.contains(conn.getId()))
                    conn.getPacketSender().sendPacket(packet);
            });
        });
    }
} catch (Exception e) {
    e.printStackTrace();
}
}

3 usages
private final static Set<UUID> loggedInUsers = Collections.synchronizedSet(new HashSet<>());

4 usages Ben Anderson *
static void addAuthenticationToServerBuilder(ServerBuilder serverBuilder, UserManager userManager) {
    //add `LoginPacket` listener on connect
    serverBuilder.onClientConnect((connection, server) -> {
        connection.getPacketListener().addCallback(LoginPacket.class, loginPacket -> {
            Optional<User> userOptional = userManager.getUser(
                //remove all '..' to protect against directory traversal vulnerability
                loginPacket.getUsername().replace(target: "..", replacement: ""))
            ;
            //check if user exists with given username
            if (userOptional.isEmpty()) {
                //username not found in users
                connection.getPacketSender().sendPacket(new DisconnectPacket(reason: "Incorrect Credentials"));
                try {
                    Thread.sleep(millis: 200);
                    connection.close();
                } catch (Exception ignored) {}
                return;
            }
            User expectedUser = userOptional.get();
        });
    });
}

```

Figure 149: Server module Main class part 2

```
User expectedUser = userOptional.get();
//check if password is correct
if (!expectedUser.getHashedPassword().equals(
    LoginPacket.hashPassword(loginPacket.getHashedPassword())))
{
    //password is wrong
    connection.getPacketSender().sendPacket(new DisconnectPacket(reason: "Incorrect Credentials"));
    try {
        Thread.sleep( millis: 200);
        connection.close();
    } catch (Exception ignored) {}
    return;
}
//login successful
loggedInUsers.add(connection.getId());
connection.getPacketSender().sendPacket(new InfoPacket("Correct Credentials"));
};

};

//remove connection from loggedInUsers on disconnect - prevents infinite memory usage
serverBuilder.onClientDisconnect((connection, server) -> {
    loggedInUsers.remove(connection.getId());
});
}

}
```

Figure 150: Server module Main class part 3

You'll notice that there is a `todo` comment in the code, this is just to remind me to remove or change the fact that I am adding a test user with hard-coded credentials in the future. This user is currently intended for testing purposes.

During the development of this code, I decided it would be useful for the client to be made aware when they successfully logged in, hence I added an `InfoPacket` class to the connection framework which just transmits generic information. In this case, it transmits the message “Correct Credentials”.

```
package xyz.benanderson.scs.networking.packets;

import lombok.Getter;
import xyz.benanderson.scs.networking.Packet;

9 usages  ↳ Ben Anderson
public class InfoPacket extends Packet {

    1 usage
    @Getter
    private final String info;

    1 usage  ↳ Ben Anderson
    public InfoPacket(String info) {
        super(InfoPacket.class);
        this.info = info;
    }

}
```

Figure 151: *InfoPacket* class

To ensure that the client could no longer view the camera feed on the server without authenticating, I attempted to connect to the new server with the unchanged client:

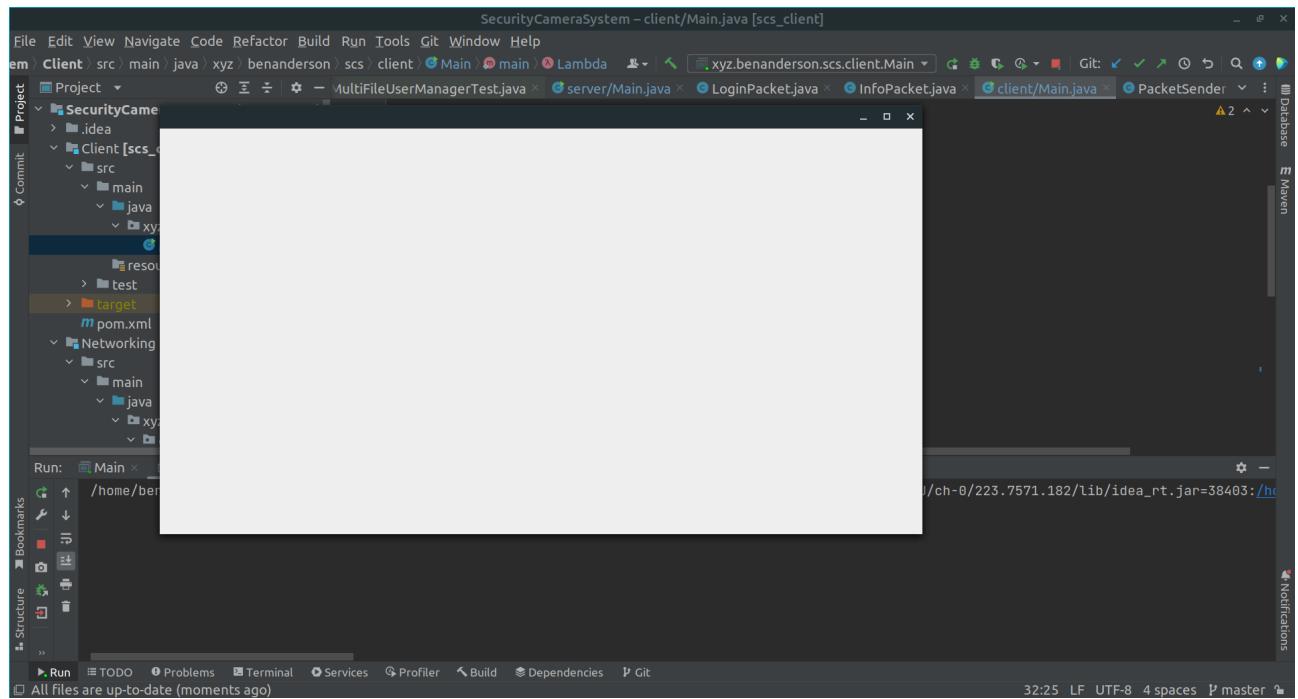


Figure 152: Connection to server without using authentication test

As can be observed in the figure above, no image was displayed on the client, indicating that clients couldn't view the camera feed without being unauthenticated.

Prior to implementing authentication on the client, I wanted to be sure that all the authentication code on the server was working as intended, hence I implemented the `ServerLoginTest` class. The test code creates a `ServerBuilder` with a fake `UserManager`, adds the standard authentication flow offered by the `Main` class in the server module, and tests it against all possible login scenarios to ensure the code logic of the authentication flow is correct.

```
package xyz.benanderson.scs.server;

import org.junit.jupiter.api.Test;
import xyz.benanderson.scs.networking.connection.Connection;
import xyz.benanderson.scs.networking.packets.DisconnectPacket;
import xyz.benanderson.scs.networking.packets.InfoPacket;
import xyz.benanderson.scs.networking.packets.LoginPacket;
import xyz.benanderson.scs.server.account.User;
import xyz.benanderson.scs.server.account.UserManager;
import xyz.benanderson.scs.server.networking.Server;
import xyz.benanderson.scs.server.networking.ServerBuilder;

import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.*;

no usages  ↳ Ben Anderson *
public class ServerLoginTest {

    3 usages  ↳ Ben Anderson
    private UserManager mockUserManager() {
        User user = User.fromPlainTextPassword(username: "testUsername",
            LoginPacket.hashPassword(plainTextPassword: "testPassword"),
            admin: false);
        UserManager userManager = mock(UserManager.class);
        doReturn(Optional.of(user)).when(userManager).getUser(username: "testUsername");
        try {
            doNothing().when(userManager).createUser(any(User.class));
            doNothing().when(userManager).deleteUser(anyString());
        } catch (Exception ignored) {}
        return userManager;
    }
}
```

Figure 153: *ServerLoginTest* class part 1

```

@Test
public void testIncorrectPassword() throws UnknownHostException {
    CompletableFuture<String> result = new CompletableFuture<>();
    ServerBuilder serverBuilder = new ServerBuilder( port: 0, InetAddress.getLocalHost());
    Main.addAuthenticationToServerBuilder(serverBuilder, mockUserManager());
    try (Server server = serverBuilder.build()) {
        Socket socket = new Socket(InetAddress.getLocalHost(), server.getPort());
        Connection connection = new Connection(socket);
        connection.getPacketListener().addCallback(InfoPacket.class, infoPacket -> {
            result.complete(infoPacket.getInfo());
        });
        connection.getPacketListener().addCallback(DisconnectPacket.class, disconnectPacket -> {
            result.complete(disconnectPacket.getReason());
        });
        connection.getPacketSender().sendPacket(
            LoginPacket.fromPlainTextPassword(
                username: "testUsername", plainTextPassword: "incorrectPassword"
            )
        );
        assertEquals(expected: "Incorrect Credentials", result.get());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

no usages  Ben Anderson *

@Test
public void testIncorrectUsername() throws UnknownHostException {
    CompletableFuture<String> result = new CompletableFuture<>();
    ServerBuilder serverBuilder = new ServerBuilder( port: 0, InetAddress.getLocalHost());
    Main.addAuthenticationToServerBuilder(serverBuilder, mockUserManager());
    try (Server server = serverBuilder.build()) {
        Socket socket = new Socket(InetAddress.getLocalHost(), server.getPort());
        Connection connection = new Connection(socket);
        connection.getPacketListener().addCallback(InfoPacket.class, infoPacket -> {
            result.complete(infoPacket.getInfo());
        });
        connection.getPacketListener().addCallback(DisconnectPacket.class, disconnectPacket -> {
            result.complete(disconnectPacket.getReason());
        });
        connection.getPacketSender().sendPacket(
            LoginPacket.fromPlainTextPassword(
                username: "testUsername", plainTextPassword: "incorrectPassword"
            )
        );
        assertEquals(expected: "Incorrect Credentials", result.get());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
}

```

Figure 154: ServerLoginTest class part 2

```
@Test
public void testSuccessfulLogin() throws UnknownHostException {
    CompletableFuture<String> result = new CompletableFuture<>();
    ServerBuilder serverBuilder = new ServerBuilder( port: 0, InetAddress.getLocalHost());
    Main.addAuthenticationToServerBuilder(serverBuilder, mockUserManager());
    try (Server server = serverBuilder.build()) {
        Socket socket = new Socket(InetAddress.getLocalHost(), server.getPort());
        Connection connection = new Connection(socket);
        connection.getPacketListener().addCallback(InfoPacket.class, infoPacket -> {
            result.complete(infoPacket.getInfo());
        });
        connection.getPacketListener().addCallback(DisconnectPacket.class, disconnectPacket -> {
            result.complete(disconnectPacket.getReason());
        });
        connection.getPacketSender().sendPacket(
            LoginPacket.fromPlainTextPassword(
                username: "testUsername", plainTextPassword: "testPassword"
            )
        );
        assertEquals( expected: "Correct Credentials", result.get());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Figure 155: ServerLoginTest class part 3

Running the `ServerLoginTest` class yielded the following successful test results:

```
[INFO] Running xyz.benanderson.scs.server.ServerLoginTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.357 s
```

Figure 156: ServerLoginTest class successful tests

I then added code to login on the client:

```

package xyz.benanderson.scs.client;

import xyz.benanderson.scs.networking.connection.Connection;
import xyz.benanderson.scs.networking.packets.DisconnectPacket;
import xyz.benanderson.scs.networking.packets.InfoPacket;
import xyz.benanderson.scs.networking.packets.LoginPacket;
import xyz.benanderson.scs.networking.packets.MediaPacket;

import javax.swing.*;
import java.net.Socket;

no usages ▲ Ben Anderson
public class Main {

    no usages ▲ Ben Anderson
    public static void main(String[] args) throws Exception {
        JFrame f = new JFrame();
        f.setVisible(true);
        f.setSize( width: 800, height: 480 );
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel label = new JLabel();
        f.getContentPane().add(label);

        Socket socket = new Socket( host: "127.0.0.1", port: 8192 );
        Connection connection = new Connection(socket);
        connection.getPacketListener().addCallback(MediaPacket.class, mediaPacket -> {
            label.setIcon(new ImageIcon(mediaPacket.getMediaFrame()));
        });
        connection.getPacketListener().addCallback(InfoPacket.class, infoPacket -> {
            System.out.println("[INFO] " + infoPacket.getInfo());
        });
        connection.getPacketListener().addCallback(DisconnectPacket.class, disconnectPacket -> {
            System.out.println("[DISCONNECTED] Reason: " + disconnectPacket.getReason());
            f.dispose();
        });
        Thread.sleep( millis: 5000 );
        LoginPacket loginPacket = LoginPacket.fromPlainTextPassword( username: "testUsername", plainTextPassword: "testPassword" );
        connection.getPacketSender().sendPacket(loginPacket);
    }
}

```

Figure 157: Main class from client software (with login)

Here the credentials are hard-coded to authenticate with the username “testUsername” and password “testPassword”. With the account existing on the server software, I tested the new version of the client software.

For the first five seconds of the client software running (before the ‘LoginPacket’ is sent), there was no visible camera output (as expected): For the first five seconds of the client software running (before the ‘LoginPacket’ is sent), there was no visible camera output (as expected):

```

    ...
    void main(String[] args) throws Exception {
        new JFrame();
        e(true);
        width: 800, height: 480);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel label = new JLabel();
        f.getContentPane().add(label);

        Socket socket = new Socket(host: "127.0.0.1", port: 8192);
        Connection connection = new Connection(socket);
    }
}

```

The screenshot shows the IntelliJ IDEA interface with the code editor open to the Main.java file. The code initializes a JFrame and connects to a socket. The project structure on the left shows a package named 'Networking [scs_networking]' containing a 'main' directory with a 'java' subdirectory containing 'xyz.benanderson.scs.networking'. The code editor has syntax highlighting for Java and some annotations. The bottom status bar indicates a successful build.

Figure 158: Client software before authentication

After five seconds of the client software running (after the 'LoginPacket' was sent), there was camera output displayed to the client (as expected):

```

    ...
    void main(String[] args) throws Exception {
        new JFrame();
        e(true);
        width: 800, height: 480);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel label = new JLabel();
        f.getContentPane().add(label);

        Socket socket = new Socket(host: "127.0.0.1", port: 8192);
        Connection connection = new Connection(socket);
    }
}

```

The screenshot shows the IntelliJ IDEA interface with the code editor open to the Main.java file. The code is identical to Figure 158. The terminal window at the bottom shows the message '[INFO] Correct Credentials'. The status bar indicates a successful build.

Figure 159: Client software after authentication

The below white box test table illustrates testing against the authentication flow implementation:

Class(es) being	How is the class being tested	Expected outcome from the class being	Was expected outcome
-----------------	-------------------------------	---------------------------------------	----------------------

tested		ing tested	achieved? (if no, what was the actual outcome?)
Main	A standard Server instance is constructed with authentication and callbacks added for disconnect and info packets. A Connection is then attached to the Server and an incorrect password is sent as part of the authentication process. The statuses of the callbacks are then checked.	The disconnect callback will be triggered with a reason citing incorrect credentials. The info callback will not be triggered.	Yes
Main	A standard Server instance is constructed with authentication and callbacks added for disconnect and info packets. A Connection is then attached to the Server and an incorrect username is sent as part of the authentication process (a username that is not associated with any active account). The statuses of the callbacks are then checked.	The disconnect callback will be triggered with a reason citing incorrect credentials. The info callback will not be triggered.	Yes
Main	A standard Server instance is constructed with authentication and callbacks added for disconnect and info packets. A Connection is then attached to the Server and correct, valid credentials are sent as part of the authentication process. The statuses of the callbacks are then checked.	The info callback will be triggered with a message informing the Connection of correct, valid credentials. The disconnect callback will not be triggered.	Yes

Graphical User Interface

Implementation

The only remaining aspect of the second prototype to implement was a more comprehensive GUI, this would allow the user to choose the credentials they used to connect to the camera server with.

Using my IDEs built-in tools, I designed and implemented the Main Window Client GUI and the Camera Connection Window (both very close to the original design specification) using Java Swing GUI components.

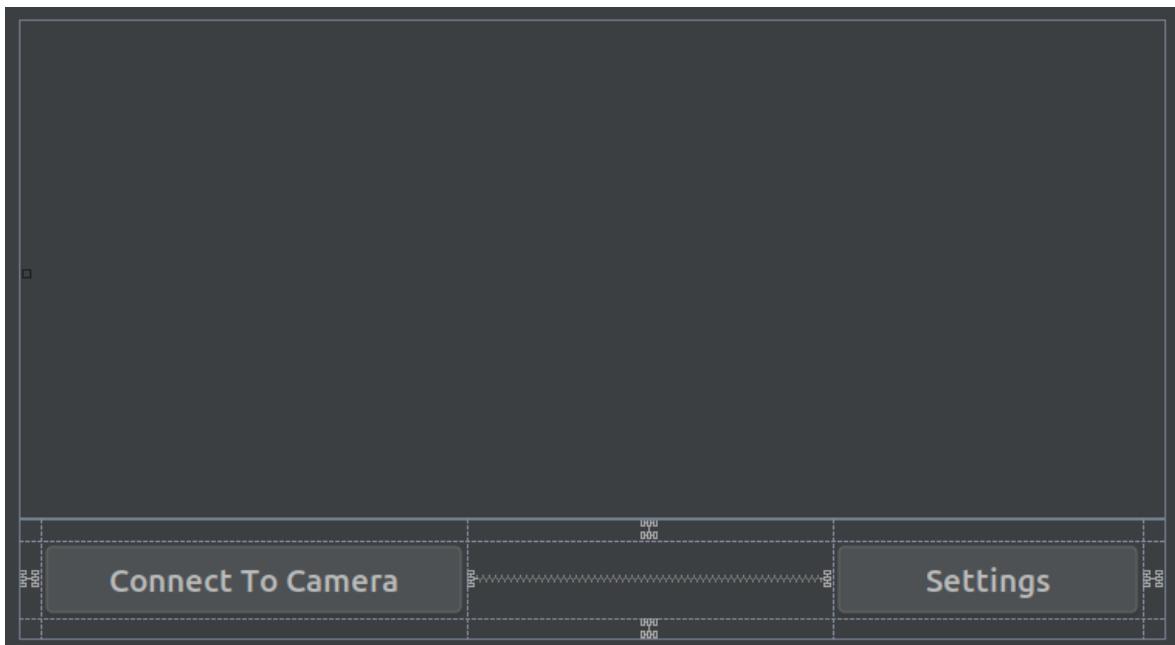


Figure 160: Main Window Client GUI Java Swing Design

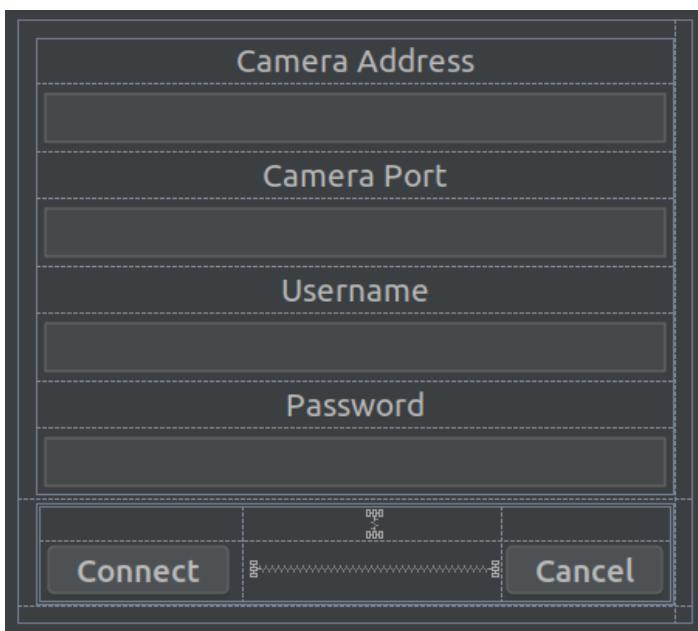


Figure 161: Camera Connection Window Client GUI Java Swing Design

I also implemented pop-ups which would notify the user upon an input validation error, such as the input camera port not being in the acceptable range. I also used pop-ups to indicate if an error occurred when attempting to connect to the camera at the specified address, or if the access credentials were incorrect.

See below for the code that powers the logic of the camera connection window:

```
package xyz.benanderson.scs.client;

import lombok.Getter;
import xyz.benanderson.scs.networking.Validation;

import javax.swing.*;
import java.awt.event.*;
import java.io.IOException;

2 usages  ↳ Ben Anderson
public class ConnectionDialog extends JDialog {
    3 usages
    private JPanel contentPane;
    3 usages
    private JButton buttonConnect;
    2 usages
    private JButton buttonCancel;
    1 usage
    @Getter
    private JTextField addressInput;
    1 usage
    @Getter
    private JTextField portInput;
    1 usage
    @Getter
    private JTextField usernameInput;
    1 usage
    @Getter
    private JPasswordField passwordInput;

    1 usage  ↳ Ben Anderson
    public ConnectionDialog() {
        setContentPane(contentPane);
        setModal(true);
        setLocationRelativeTo(null);
        getRootPane().setDefaultButton(buttonConnect);
        buttonConnect.addActionListener(e -> onClickConnect());
        buttonCancel.addActionListener(e -> onClickCancel());
    }
}
```

Figure 162: *ConnectionDialog* class part 1

```
// call onCancel() when cross is clicked
setDefaultCloseOperation(DO NOTHING ON CLOSE);
addWindowListener((WindowAdapter) windowClosing(e) -> { onClickCancel(); });

// call onCancel() on ESCAPE
contentPane.registerKeyboardAction(e -> onClickCancel(),
    KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, modifiers: 0), JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);

pack();
setVisible(true);
}

1 usage  Ben Anderson
private void onClickConnect() {
    int validatedPort;
    try {
        validatedPort = Validation.parsePort(getPortInput().getText());
    } catch (Validation.ValidationException e) {
        JOptionPane.showMessageDialog(parentComponent: this, e.getMessage(), title: "Input Validation Error", JOptionPane.ERROR_MESSAGE);
        return;
    }
    try {
        Main.attemptConnect(getAddressInput().getText(), validatedPort,
            getUsernameInput().getText(), new String(getPasswordInput().getPassword()));
    } catch (IOException e) {
        JOptionPane.showMessageDialog(parentComponent: this, e.getMessage(), title: "Connection Error", JOptionPane.ERROR_MESSAGE);
        return;
    }
    dispose();
}

3 usages  Ben Anderson
private void onClickCancel() {
    dispose();
}
}
```

Figure 163: ConnectionDialog class part 2

As can be seen in the `ConnectionDialog#onClickConnect` method, I used the `parsePort` method in the newly implemented `Validation` class, this class was implemented to handle validation of ports for both the client and server software.

```
package xyz.benanderson.scs.networking;

6 usages  ↳ Ben Anderson
public class Validation {

    1 usage  ↳ Ben Anderson
    public static int parsePort(String portInputStr) throws ValidationException {
        try {
            int port = Integer.parseInt(portInputStr);
            return parsePort(port);
        } catch (NumberFormatException numberFormatException) {
            throw new ValidationException("Port number must be an integer");
        }
    }

    2 usages  ↳ Ben Anderson
    public static int parsePort(int portInput) throws ValidationException {
        if (portInput < 0 || portInput > 65535) {
            throw new ValidationException("Port number must be between 1 and 65535 (inclusive)");
        }
        return portInput;
    }

    6 usages  ↳ Ben Anderson
    public static class ValidationException extends Exception {
        2 usages  ↳ Ben Anderson
        public ValidationException(String s) { super(s); }
    }
}
```

Figure 164: Validation class

Incorporating all the above code and GUIs required a rework of the `Main` class in the client software, including abstracting the concept of connecting to a camera into its own method, which would be called by `ConnectionDialog#onClickConnect`.

```
package xyz.benanderson.scs.client;

import xyz.benanderson.scs.networking.connection.Connection;
import xyz.benanderson.scs.networking.packets.DisconnectPacket;
import xyz.benanderson.scs.networking.packets.InfoPacket;
import xyz.benanderson.scs.networking.packets.LoginPacket;
import xyz.benanderson.scs.networking.packets.MediaPacket;

import javax.swing.*;
import java.awt.*;
import java.io.IOException;
import java.net.Socket;

1 usage  Ben Anderson
public class Main {

    10 usages
    private static ClientGUI clientGUI;
    8 usages
    private static Connection connection;

    no usages  Ben Anderson
    public static void main(String[] args) throws Exception {
        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());

        JFrame frame = new JFrame();
        frame.setVisible(true);
        frame.setTitle("Security Camera Client");
        frame.setSize( width: 1280, height: 800 );
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLocationRelativeTo(null);

        clientGUI = new ClientGUI();
        clientGUI.getSettingsButton().setFont(new Font(Font.MONOSPACED, Font.BOLD, size: 20));
        clientGUI.getSettingsButton().setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
        clientGUI.getConnectButton().setFont(new Font(Font.MONOSPACED, Font.BOLD, size: 20));
        clientGUI.getConnectButton().setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
        clientGUI.getConnectButton().addActionListener(event -> new ConnectionDialog());

        frame.setContentPane(clientGUI.getContentPane());
    }
}
```

Figure 165: Main class from client software with GUI (part 1)

```

public static void attemptConnect(String address, int port, String username, String password) throws IOException {
    if (connection != null && connection.isConnected()) {
        try {
            connection.close();
        } catch (Exception ignored) {}
    }

    Socket socket = new Socket(address, port);
    connection = new Connection(socket);
    connection.getPacketListener().addCallback(MediaPacket.class, mediaPacket -> {
        EventQueue.invokeLater(() -> clientGUI.getVideoComponent()
            .setIcon(new ImageIcon(mediaPacket.getMediaFrame())));
    });
    connection.getPacketListener().addCallback(InfoPacket.class, infoPacket -> {
        JOptionPane.showMessageDialog(clientGUI.getContentPane(), infoPacket.getInfo(),
            title: "Information", JOptionPane.INFORMATION_MESSAGE);
    });
    connection.getPacketListener().addCallback(DisconnectPacket.class, disconnectPacket -> {
        JOptionPane.showMessageDialog(clientGUI.getContentPane(),
            message: "Disconnected From Camera." + System.lineSeparator() + "Reason: " + disconnectPacket.getReason(),
            title: "Disconnected", JOptionPane.WARNING_MESSAGE);
    });
    LoginPacket loginPacket = LoginPacket.fromPlainTextPassword(username, password);
    connection.getPacketSender().sendPacket(loginPacket);
}
}

```

Figure 166: Main class from client software with GUI (part 2)

Testing

Due to the fact that it is very difficult to write unit tests for graphical user interfaces, as their appearance differs depending on the system software, it is best to test the appearance and behaviour of the graphical user interface manually.

For all of the above GUI screenshots, it is essential to remember that my operating system is configured to display GUI components following a certain theme and colour palette. Running the software of different operating systems may produce incredibly varied results; for example, the application is almost guaranteed to use a lighter colour palette when ran on a standard Microsoft Windows installation. The GUI tailors itself to the theme of the system it is running on due to the first line of code in the `main` method entry-point of the client software:

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

Below are the produced user interfaces when running and interacting with the program.

Main Window

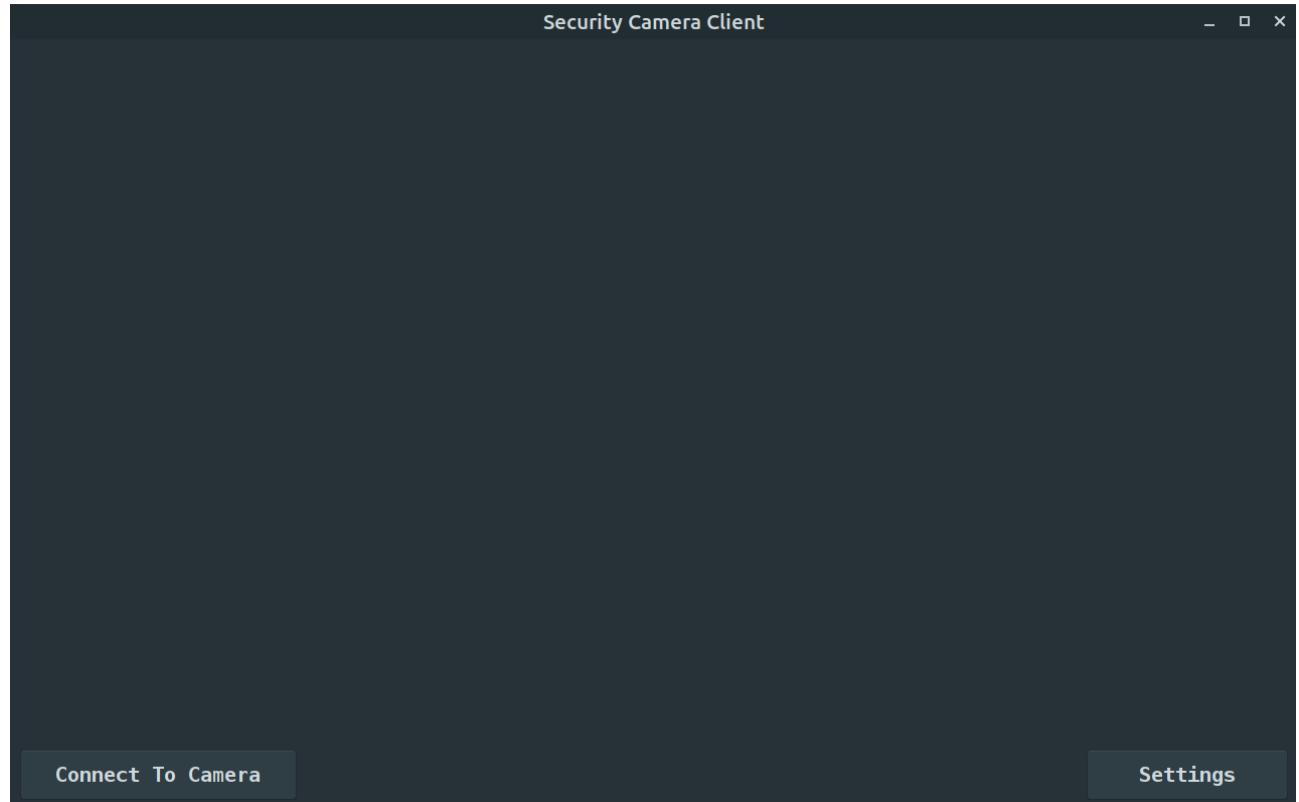


Figure 167: Client software main window

Camera Connection Window

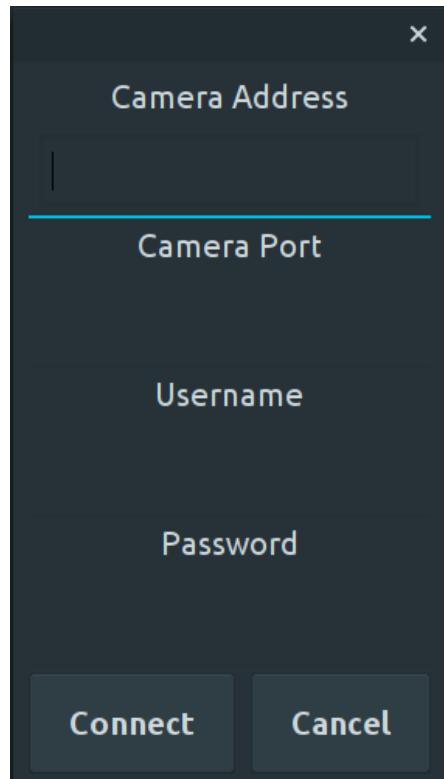


Figure 168: Client software camera connection window with empty inputs

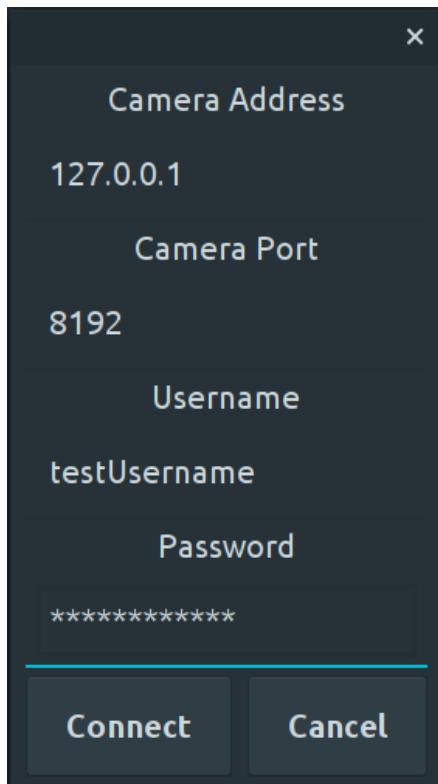


Figure 169: Client software camera connection window with filled inputs

Invalid Port Pop-ups

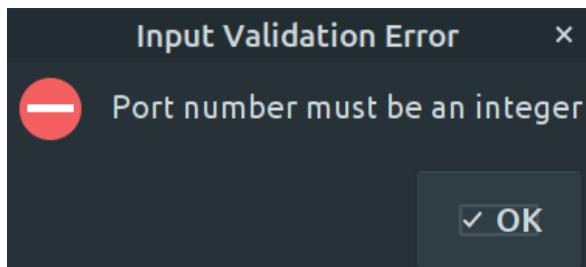


Figure 170: Client software camera connection window invalid port pop-up

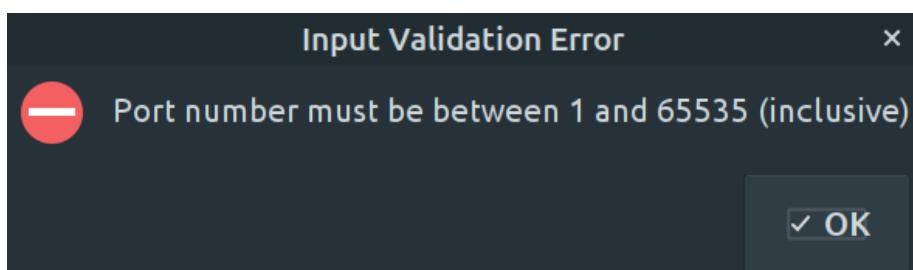


Figure 171: Client software camera connection window invalid port range pop-up

Connection Error Pop-up

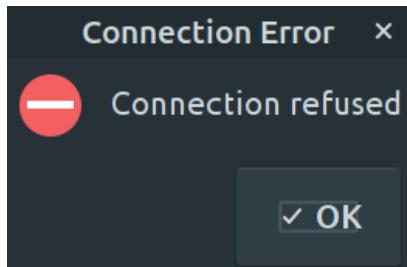


Figure 172: Client software camera connection window connection error pop-up

Successful Login Correct Credentials Pop-up

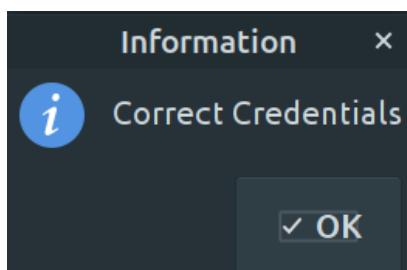


Figure 173: Client software camera connection window correct credentials pop-up

Unsuccessful Login Incorrect Credentials Pop-up

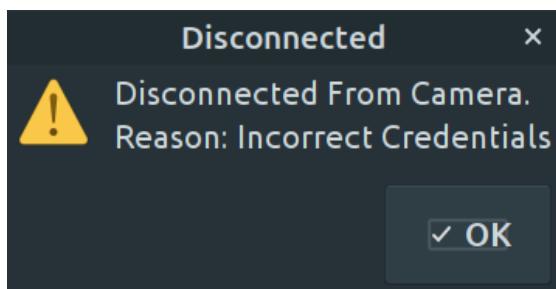


Figure 174: Client software camera connection window incorrect credentials pop-up

Connected Camera View

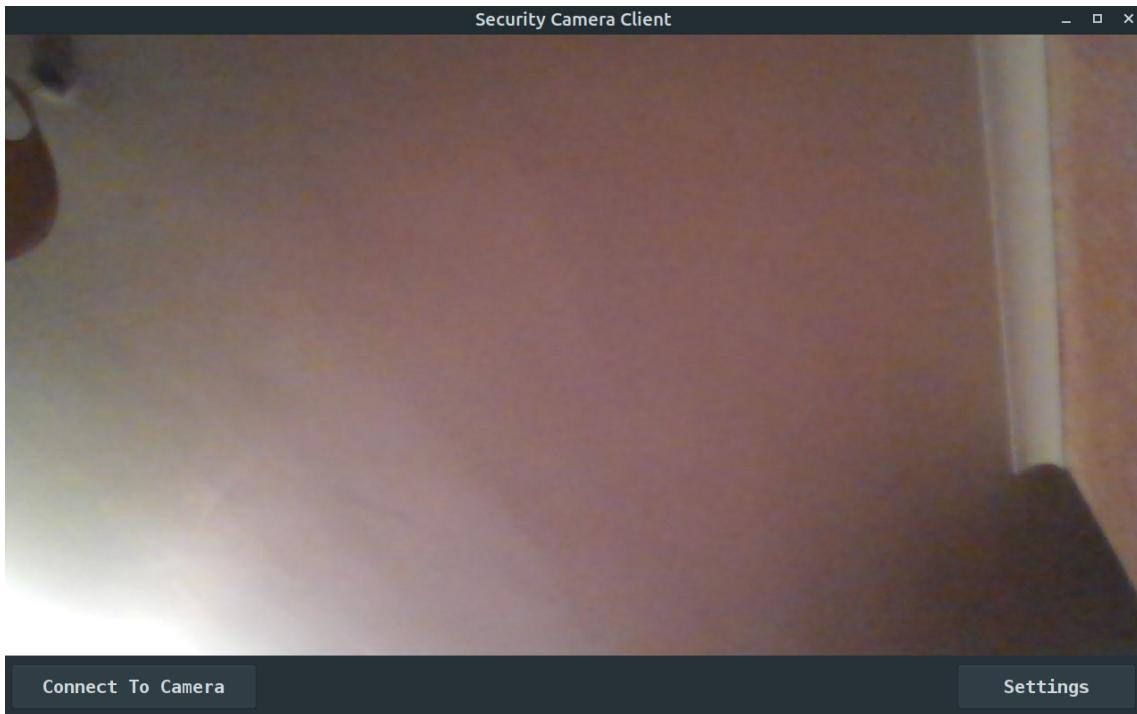


Figure 175: Client software successfully streaming camera into GUI

Reconsidering The Settings Window

In my original GUI design, I included a 'Settings Window' which could be used to remotely change the settings on the camera server. During the development of the second prototype, I have been consistently considering whether this feature would be beneficial for the stakeholder, and whether its implementation is worth the complications that come along with it – especially considering the time restraints of this project.

To resolve this conflict, I asked the stakeholder how important the feature was for them, they responded as follows:

"It's not an important feature for me, I'll just configure the camera once when setting it up."

This input from my stakeholder concluded that the feature was not necessary and therefore wouldn't be implemented. Hence, I proceeded to remove the 'Settings' button from the main window.

Configurable Accounts

Up to this point, the code for the camera server has been automatically creating an account with username "testUsername" and password "testPassword". However, because the account system has now been fully implemented using files, the stakeholder can create a file on the camera using the following format and it will create an account on the server:

File Name: *USERNAME*

File Contents:

```
USERNAME
PASSWORD_HASHED_TWICE
IS_ADMIN
```

- `USERNAME` is the username of the user.
- `PASSWORD_HASHED_TWICE` is the result of processing the password of the user through the SHA-256 hashing algorithm twice and converting it to a hex digest.
- `IS_ADMIN` is a boolean value denoting if the user is an administrator (options: ‘true’, ‘false’).

In order to assist the stakeholder, I will still create a default account that they can use, but just with much more secure credentials that can be configured in the camera server configuration file.

The default credentials for this account will be as follows:

Username: admin

Password: 5J6*9XodH&&mAUBz

The password is 16 characters long, consisting of numbers, lower-case letters, upper-case letters and symbols in order to ensure maximum security. The username and password can both be customised and changed in the camera server configuration file to meet the stakeholder’s needs.

Due to the fact that the camera server will create the default account if it does not exist every time the server is restarted, this default account feature in the config can also be used as a method of creating standard accounts.

Stakeholder Black Box Testing

I built the project and provided the new executable files for the client and server software to the stakeholder.

They filled out the following black box test table for the client software:

Test input field	Test input data	Expected outcome	Was expected outcome achieved? (if no, what was the actual outcome?)
Camera Port	the port is five five five five	Error message to the user – “Port number must be an integer”.	Yes
Camera Port	70000	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.	Yes
Camera Port	65536	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.	Yes
Camera Port	0	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.	Yes
Camera Port	-1	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.	Yes
Camera	80	Either an error message due to lack of priv-	Yes

Port		ileges, or a successful server setup depending on the privileges with which the camera server was ran.	
Camera Port	9001	Successful server setup assuming that port 9001 is not being used by another application.	Yes
Camera Port	8192	Successful connection assuming there is a camera server on port 8192. Otherwise, error message to user denoting that no camera server could be found on that port.	Yes
Camera Address	Ncu28*("6!"£\$%^&*()_+-=	Error message to the user specifying that a connection error occurred and that the camera address was invalid.	Yes
Camera Address	127.0.0.1	Error message to the user specifying that a connection refused occurred because the target address isn't running a camera server.	Yes
Username	!"£\$%^&*()`¬ 1a=_+[]{};:@#~,./<>?	Disconnected from camera due to incorrect credentials and message shown to user to notify of this.	Yes
Password	!"£\$%^&*()`¬ 1=_+[]{};:@#~,./<>?	Disconnected from camera due to incorrect credentials and message shown to user to notify of this.	Yes
Cancel Button	Mouse Click	The camera connection window closes and no further action occurs.	Yes
Connect Button	Mouse Click	The client software attempts to connect to the server software at the specified address and closes the camera connection window.	Yes
Connect To Camera Button (on main GUI)	Mouse Click	Opens the camera connection window.	Yes

They filled out the black box test table for the server software:

Test input field	Test input data	Expected outcome	Was expected outcome achieved? (if no, what was the actual outcome?)
users.save-directory	56347890	The directory '56347890' is created in the current working directory and used to store user data.	Yes
users.save-directory	users	The directory 'users' is created in the current working directory and used to store user data.	Yes
users.save-	./users/	The directory 'users' is created in the current	Yes

directory		working directory and used to store user data.	
users.default.username	x703-N*(CDJ(9FK;	The user 'x703-N*(CDJ(9FK;' is successfully created and can be used to authenticate.	Yes
users.default.username	vcx/dedg	An exception is thrown informing the user that '/' is a reserved character that cannot be used in username due to the operating systems file implementation.	Yes
users.default.username	admin	The user 'admin' is successfully created and can be used to authenticate.	Yes
users.default.password	5J6*9XodH&&mAUBz	The user is successfully created with password '5J6*9XodH&&mAUBz' and can be used to authenticate.	Yes
users.default.is_admin	Hfwehui289	The value defaults to 'true'.	Yes
users.default.is_admin	Hfwehui289	The value defaults to 'false' – the user is not considered an admin.	Yes
users.default.is_admin	true	The value is 'true' – the user is considered an admin.	Yes

They came back with the following feedback:

"I'm really happy with how the system is working now and I think the GUI is much better than before. The only other feature I would like is the ability to record and download videos of what the camera captures."

The stakeholder's approval of the GUI means that success criteria point 6 is now completed. Their successful interaction with the account system also indicates that success criteria point 8 is completed. Success criteria point 9 can also be said to be completed as users are managed using files or the configuration file on the camera server.

However, the feedback expresses a clear desire for me to continue to implement the camera recording and downloading feature which I attempted to implement in the first prototype but eventually delayed it for a future prototype. This feature would tick off success criteria points 10, 12, 14 and 15.

Third Prototype

After establishing camera recording and video downloading as a clear requirement for the third prototype, I began to design and implement an improved solution.

VideoEncoder Class

When attempting to implement the `VideoEncoder` class in the first prototype, I ran into issues which were documented and explained. However, I have a plan for a new implementation approach.

The idea I have is to append the images (camera frames) as serialized bytes to a large file which effectively stores a collection of the images, not formatted as a video file. This file can be considered to follow my own custom file format and may also contain a metadata header depending on if I end up needing it during file parsing. This file will then later be converted to a video by a scheduled process on the camera server.

The video submodule of the camera server now consists of a few key components:

- a subroutine to append a media frame to the raw media save file
- a subroutine to process a raw media save files and encode them into videos
- a scheduled, batch job to run the processing and encoding subroutine on raw media save files

A 'camera raw media save file' is the custom file format that I mentioned previously. The subroutines listed above both function on a very low level – writing/ready specific bytes directly to the raw media save file and jumping around in it between bytes.

As I implemented the `VideoEncoder` class, the format of the custom file changed slightly and became more concretely defined:

- bytes 1-8: `long` data type denoting the number of media frames in the file
- bytes 9-16: `long` data type denoting the start timestamp of the video in milliseconds
- bytes 17-24: `long` data type denoting the end timestamp of the video in milliseconds
- remaining bytes: JPEG encoded image data for all the media frames in sequential order

`long` is an 8 byte long data type in Java that can be used to represent large numbers without decimals.

The file format is designed in such a way that adding a new frame to the raw media save file is easy and only requires the following:

- increment the 'number of media frames' (bytes 1-8)
- set the 'end timestamp of the video' (bytes 17-24) to be the current time in milliseconds
- append the bytes of the JPEG encoded image to the end of the file

Below is the implementation of the `VideoEncoder` class:

```
package xyz.benanderson.scs.server.video;

import lombok.AllArgsConstructor;
import org.jcodec.api.awt.AWTSequenceEncoder;

import javax.imageio.ImageIO;
import javax.imageio.ImageReader;
import javax.imageio.stream.FileImageInputStream;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.OutputStream;
import java.io.RandomAccessFile;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;
import java.util.Optional;

4 usages Ben Anderson
@AllArgsConstructor
public class VideoEncoder {

    1 usage
    private final VideoFileManager videoFileManager;

    //file format of raw media save file is as follows:
    //- long (number of media frames in the file)
    //- long (start timestamp in milliseconds)
    //- long (end timestamp in milliseconds)
    //- list of serialized media frames

    2 usages Ben Anderson
    public void appendToStream(BufferedImage image, long currentTimeMillis) {
        //output error if can't access a save file
        Optional<Path> currentSaveFileOptional = videoFileManager.getCurrentSaveFile();
        if (currentSaveFileOptional.isEmpty()) {
            System.err.println("[ERROR] Failed to fetch video save file.");
            return;
        }
        //open current save file as `RandomAccessFile`, therefore being able to jump ar
```

Figure 176: *VideoEncoder class in third prototype part 1*

```

//open current save file as `RandomAccessFile`, therefore being able to jump around the file
Path currentSaveFile = currentSaveFileOptional.get();
try (RandomAccessFile randomAccessFile = new RandomAccessFile(currentSaveFile.toFile(), mode: "rw")) {
    //if file size is zero, it is new and needs some header metadata at the start of the file
    if (Files.size(currentSaveFile) == 0L) {
        randomAccessFile.writeLong(v: 1L);
        randomAccessFile.seek(pos: 8);
        randomAccessFile.writeLong(currentTimeMillis);
    } else {
        //if file size is not zero, increase the number of media frames in the header
        long currentNumberFrames = randomAccessFile.readLong();
        randomAccessFile.seek(pos: 0);
        randomAccessFile.writeLong(v: currentNumberFrames + 1);
    }
    //write the time of the last media frame
    randomAccessFile.seek(pos: 16);
    randomAccessFile.writeLong(currentTimeMillis);
} catch (Exception e) {
    //output error if one is encountered
    System.err.println("[ERROR] An error occurred when writing timestamp metadata to a save file.");
    e.printStackTrace();
}
//append the latest media frame to the file serialized as bytes
try (OutputStream outputStream = Files.newOutputStream(currentSaveFile, StandardOpenOption.APPEND)) {
    ImageIO.write(image, formatName: "jpg", outputStream);
} catch (Exception e) {
    //output error if one is encountered
    System.err.println("[ERROR] An error occurred when writing a media frame to a save file.");
    e.printStackTrace();
}
}

1 usage  Ben Anderson
public void processRawMediaSave(Path rawMediaSaveFile) {
    //open file using `RandomAccessFile` to be able to skip around in the file
    try (RandomAccessFile randomAccessFile = new RandomAccessFile(rawMediaSaveFile.toFile(), mode: "r");
        FileInputStream fileInputStream = new FileInputStream(randomAccessFile)) {
        //read header metadata
        long numberOffFrames = randomAccessFile.readLong();
        long startTimestamp = randomAccessFile.readLong();
        long endTimestamp = randomAccessFile.readLong();
        //calculate fps (frames per second) using header metadata
        long videoDurationInMillis = endTimestamp - startTimestamp;
        int fps = (int) (numberOffFrames / (videoDurationInMillis / 1000));
    }
}

```

Figure 177: VideoEncoder class in third prototype part 2

```
//setup image (media frame) reading
fileImageInputStream.seek(pos: 24);
ImageReader reader = ImageIO.getImageReadersBySuffix(fileSuffix: "jpg").next();
reader.setInput(fileImageInputStream);

//setup video encoding
File videoOutputFile = new File(rawMediaSaveFile.toString().replace(target: ".crms", replacement: ".mp4"));
AWTSequenceEncoder encoder = AWTSequenceEncoder.createSequenceEncoder(videoOutputFile, fps);

//read frames from the raw media save file and encode them into the output file
for (int frameNumber = 0; frameNumber < numberOfFrames; frameNumber++) {
    BufferedImage image = reader.read(frameNumber);
    encoder.encodeImage(image);
}
encoder.finish();
} catch (Exception e) {
    //output error if encountered
    System.err.println("[ERROR] An error occurred when encoding a video from a raw media save file.");
    e.printStackTrace();
}
}
```

Figure 178: VideoEncoder class in third prototype part 3

There aren't many detailed comments in the class, instead I have opted for general comments which describe how each block of the code functions in an abstract manner. This is due to the high complexity of the implementation as there is lots of advanced image and byte manipulation.

VideoEncoderTest Class

Instead of the manual testing that was carried out during the previous attempted implementation of the `VideoEncoder` class, I have also implemented unit tests in the `VideoEncoderTest` class to automate the testing of raw media save file creation and video processing and encoding:

```
package xyz.benanderson.scs.server.video;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.io.TempDir;

import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.Duration;

import static org.junit.jupiter.api.Assertions.assertArrayEquals;
import static org.junit.jupiter.api.Assertions.assertEquals;

no usages  ↳ Ben Anderson *
public class VideoEncoderTest {

    no usages  ↳ Ben Anderson
    @Test
    public void testAppendToStream(@TempDir Path tempDir) throws IOException {
        //test data
        long numberOffFrames = 2;
        long startTimestamp = System.currentTimeMillis() - 1000;
        long endTimestamp = startTimestamp + 1000;
        BufferedImage[] testImages = getTestImages();

        //serializing images so that they can be compared to the output
        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
        ImageIO.write(testImages[0], "jpg", byteArrayOutputStream);
        ImageIO.write(testImages[1], "jpg", byteArrayOutputStream);

        //running the method to be tested with the test data
        VideoFileManager videoFileManager = new VideoFileManager(tempDir, Duration.ofMinutes(5));
        VideoEncoder videoEncoder = new VideoEncoder(videoFileManager);
        videoEncoder.appendToStream(testImages[0], startTimestamp);
        videoEncoder.appendToStream(testImages[1], endTimestamp);
    }
}
```

Figure 179: *VideoEncoderTest* class in third prototype part 1

```

Path rawMediaSaveFile = videoFileManager.getCurrentSaveFile().orElseThrow(FileNotFoundException::new);
//checking that the data outputted by the method being tested is correct
try (InputStream inputStream = Files.newInputStream(rawMediaSaveFile);
     DataInputStream dataInputStream = new DataInputStream(inputStream)) {
    assertEquals(numberOfFrames, dataInputStream.readLong());
    assertEquals(startTimestamp, dataInputStream.readLong());
    assertEquals(endTimestamp, dataInputStream.readLong());
    assertEquals(byteArrayOutputStream.toByteArray(), dataInputStream.readAllBytes());
}
}

no usages ▲ Ben Anderson *
@Test
public void testProcessRawMediaSave(@TempDir Path tempDir) throws IOException {
    //test data
    long numberOfFrames = 2;
    long startTimestamp = System.currentTimeMillis() - 1000;
    long endTimestamp = startTimestamp + 1000;
    BufferedImage[] testImages = getTestImages();

    //file setup
    VideoFileManager videoFileManager = new VideoFileManager(tempDir, Duration.ofMinutes(5));
    VideoEncoder videoEncoder = new VideoEncoder(videoFileManager);

    Path rawMediaSaveFile = videoFileManager.getCurrentSaveFile().orElseThrow(FileNotFoundException::new);
    //writing test data to file
    try (OutputStream outputStream = Files.newOutputStream(rawMediaSaveFile);
         DataOutputStream dataOutputStream = new DataOutputStream(outputStream)) {
        dataOutputStream.writeLong(numberOfFrames);
        dataOutputStream.writeLong(startTimestamp);
        dataOutputStream.writeLong(endTimestamp);
        ImageIO.write(testImages[0], "jpg", dataOutputStream);
        ImageIO.write(testImages[1], "jpg", dataOutputStream);
    }
    System.out.println("Size: " + Files.size(rawMediaSaveFile));

    //running the method to be tested
    videoEncoder.processRawMediaSave(rawMediaSaveFile);

    File videoOutputFile = new File(rawMediaSaveFile.toString().replace(target: ".crms", replacement: ".mp4"));
    //todo parse output file to check length, fps etc. (maybe even frames)
}
}

```

Figure 180: VideoEncoderTest class in third prototype part 2

```
//private method to generate test images used as test data by other methods
2 usages  Ben Anderson
private BufferedImage[] getTestImages() {
    //create two images
    BufferedImage frameOne = new BufferedImage( width: 1280, height: 720, BufferedImage.TYPE_INT_RGB);
    BufferedImage frameTwo = new BufferedImage( width: 1280, height: 720, BufferedImage.TYPE_INT_RGB);
    //fill first image with red pixels
    for (int y = 0; y < frameOne.getHeight(); y++) {
        for (int x = 0; x < frameOne.getWidth(); x++) {
            frameOne.setRGB(x, y, rgb: 0xff0000);
        }
    }
    //fill second image with blue pixels
    for (int y = 0; y < frameTwo.getHeight(); y++) {
        for (int x = 0; x < frameTwo.getWidth(); x++) {
            frameTwo.setRGB(x, y, rgb: 0x0000ff);
        }
    }
    //return images
    return new BufferedImage[] {frameOne, frameTwo};
}
```

Figure 181: VideoEncoderTest class in third prototype part 3

The `todo` comment at the end of `VideoEncoderTest#testProcessRawMediaSave` discusses a possible improvement to the unit test method to ensure the encoded video is exactly correct.

```
[INFO] Running xyz.benanderson.scs.server.video.VideoEncoderTest
[INFO] Finished encoding video 2023-01-06T14:54:25.374156642.mp4
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.944 s
```

Figure 182: VideoEncoderTest class tests successfully passing

Here is a white box test table for the `VideoEncoder` class:

Class(es) being tested	How is the class being tested	Expected outcome from the class being tested	Was expected outcome achieved? (if no, what was the actual outcome?)
VideoEncoder	A VideoEncoder instance is created and two images appended to the raw media save stream using VideoEncoder#appendToStream. The raw media save file is then read and its header tested for the correct values, the file body is tested for equality against the raw byte data of the two test images.	The file header contains the correct data about the file body. The file body bytes are equal to the raw byte data of the two test images.	Yes
VideoEncoder	A raw media save file is written. A VideoEncoder instance is created and processes the	No errors will be produced during the en-	Yes

	raw media save file with VideoEncoder#processRawMediaSave. The encoded video output file will be checked.	coding of the video and it will encode the video correctly such that it can be played by standard video viewing tools.	
--	---	--	--

Video Integration

I then integrated the video system into the server software's 'Main' class. Firstly, I made the following changes to the main 'try' block of the server software's 'main' method:

```
try (Server server = serverBuilder.build();
    CameraViewer cameraViewer = new CameraViewer(Webcam.getDefault());
    VideoFileManager videoFileManager = new VideoFileManager(
        ConfigurationWrapper.getInstance().getVideoSaveDirectory(),
        ConfigurationWrapper.getInstance().getVideoDuration()
    );
    VideoEncoder videoEncoder = new VideoEncoder(videoFileManager);
    startVideoEncodingScheduledJob(videoFileManager, videoEncoder);
    System.out.println("[INFO] Server Started Successfully");
    while (true) {
        //attempt to capture camera image
        cameraViewer.captureImage().ifPresent(img -> {
            videoEncoder.appendToStream(img, System.currentTimeMillis());
            //if successful in capturing an image, create a packet from the image
            //and send it to all active connections
            Packet packet = new MediaPacket(img);
            server.getConnections().values().forEach(conn -> {
                if (loggedInUsers.contains(conn.getId()))
                    conn.getPacketSender().sendPacket(packet);
            });
        });
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

Figure 183: Server software Main class try-block with integrated video system

I also implemented the method 'startVideoEncodingScheduledJob' which schedules a batch processing job to run asynchronously. The processing job searches for raw media save files in the 'video' directory and runs the 'VideoEncoder#processRawMediaSave' method against them to encode them into videos.

```

private final static Set<Path> videosBeingEncoded = Collections.synchronizedSet(new HashSet<>());

private static void startVideoEncodingScheduledJob(VideoFileManager videoFileManager, VideoEncoder videoEncoder) {
    ScheduledExecutorService scheduledExecutorService = Executors.newSingleThreadScheduledExecutor();
    scheduledExecutorService.scheduleWithFixedDelay(() -> {
        Optional<Path> currentSaveFile = videoFileManager.getCurrentSaveFile();
        if (currentSaveFile.isEmpty()) {
            System.err.println("[ERROR] An error occurred when accessing the current save file.");
            return;
        }
        try (Stream<Path> pathStream = Files.list(videoFileManager.getSaveDirectory())) {
            pathStream.filter(path -> path.toString().endsWith(".crms"))
                .filter(path -> !path.equals(currentSaveFile.get()))
                .filter(path -> !videosBeingEncoded.contains(path))
                .forEach(path -> {
                    videosBeingEncoded.add(path);
                    videoEncoder.processRawMediaSave(path);
                });
        } catch (IOException e) {
            System.err.println("[ERROR] An error occurred when searching for raw media save files to encode into videos.");
        }
    }, 1, 1, TimeUnit.SECONDS);
}

```

Figure 184: Server software Main class startVideoEncodingScheduledJob method

The `videosBeingEncoded` Set is used to ensure raw media save files aren't attempted to be processed multiple times at once.

Developer Black Box Testing

I ran the camera server with the newly integrated to camera recording submodule and kept one minute worth of generated video footage, in addition to this I let it encode the video. The produced files can be seen below:

Name	Size	Date Modified
2023-01-09T21:45:01.590486564.crms	27.9 MB	Mon 09 Jan 2023 21:46:01 GMT
2023-01-09T21:45:01.590486564.mp4	34.5 MB	Mon 09 Jan 2023 21:50:28 GMT

Figure 185: Video files generated by server software after one minute of recording

As can be observed by checking the file size of the output files, they are quite large, especially to say that the video is only one minute in length. Just by multiplying up the file size, the size of a one hour video would be atleast 2.07 GB which is very large, especially when maintaining high image quality is not a paramount requirement.

Image Compression

As a result of my concern over file size of the generated video files, I decided to implement image compression of the media frames that are written to the raw media save file, and consequently encoded into the video.

Inspired by a code snippet[9] from StackOverflow, I implemented image compression into the `VideoEncoder` class when writing to raw media save files:

```

public byte[] compressImage(BufferedImage bufferedImage) {
    ByteArrayOutputStream compressed = new ByteArrayOutputStream();
    try (ImageOutputStream outputStream = new MemoryCacheImageOutputStream(compressed)) {
        ImageWriter jpgWriter = ImageIO.getImageWritersByFormatName("jpg").next();

        // Configure JPEG compression: 20% quality
        ImageWriteParam jpgWriteParam = jpgWriter.getDefaultWriteParam();
        jpgWriteParam.setCompressionMode(ImageWriteParam.MODE_EXPLICIT);
        jpgWriteParam.setCompressionQuality(0.2f);

        jpgWriter.setOutput(outputStream);
        jpgWriter.write(null, new IIOImage(bufferedImage, null, null), jpgWriteParam);
        jpgWriter.dispose();
    } catch (IOException e) {
        System.err.println("[ERROR] An error occurred when compressing a media frame for a raw media save file.");
        e.printStackTrace();
    }
    return compressed.toByteArray();
}

```

Figure 186: VideoEncoder class with compressImage method

The data returned by this method is then written to the raw media save file instead of using the `ImageIO` library which was my previous method.

I then ran the black box test again to see how the file sizes would change, with the same camera view:

Name	Size	Date Modified
2023-01-09T22:17:34.890834071.crms	27.1 MB	Mon 09 Jan 2023 22:18:34 GMT
2023-01-09T22:17:34.890834071.mp4	37.2 MB	Mon 09 Jan 2023 22:22:50 GMT

Figure 187: Video files generated by server software after one minute of recording with compression

The file size of the raw media save file came out slightly smaller (0.8 MB), however somehow the encoded video produced by the video encoding library I am using ended up being bigger (2.5 MB). This may indicate some unpredictability in the video encoding library I am using and is not something I believe I can do anything about at the current time.

Stakeholder Black Box Testing

I built and provided the most recent prototype's executable file to the stakeholder to allow them to test it and provide feedback.

They filled out the following black box test table for the server software:

Test input field	Test input data	Expected outcome	Was expected outcome achieved? (if no, what was the actual outcome?)
video.save-directory	56347890	The directory '56347890' is created in the current working directory and used to store video data.	Yes

video.save-directory	videos	The directory 'videos' is created in the current working directory and used to store video data.	Yes
video.save-directory	./videos/	The directory 'videos' is created in the current working directory and used to store video data.	Yes
video.duration.number	60	The duration number of 60 is used.	Yes
video.duration.unit	seconds	The duration unit of 'seconds' is used.	Yes

They also said the following:

"The third prototype works well and is recording videos properly. The video files are only stored on the server and no method of downloading them is provided in the software, this isn't ideal but is satisfactory enough as I can download video files from my server using other normal file transfer tools. I'm quite happy with the software in its current state."

The stakeholder's acknowledgement of the functioning video recording and encoding system means that success criteria points 12 and 14 have been successfully completed.

As mentioned by the stakeholder, I haven't implemented downloading video files to the client or recording live footage on the client software. Consequently, success criteria points 10 and 15 cannot be said to be completely finished. However, the stakeholder expressed satisfaction with their current method of downloading files and therefore I would not consider success criteria point 10 to be failed – instead partly functional, as the stakeholder still is able to download the recorded videos from the server software using their own methods.

Live recording of videos on the client software hasn't been implemented however it doesn't feel completely necessary and is not something that the stakeholder has specifically expressed dissatisfaction with. Considering this along with the time constraints provided by this project and complexity involved with implementing this feature, I won't be implementing the feature.

Evaluation

Final Testing

Evidence

Included with this documentation is a video demonstrating the system successfully operating and streaming media frames from a webcam through the camera server to the client software after having authenticated. There are also screenshots of the software in the usability section of the evaluation.

White Box Testing

The following table is a collation of all unit tests carried out during the development process, as well as a few extras - the code for the tests can be found in a table below it.

Test number	Class(es) being tested	How is the class being tested	Expected outcome from the class being tested	Was expected outcome achieved? (if no, what was the actual outcome?)
1	Packet-Sender	A PacketSender instance is created and a test packet sent to it using Packet-Sender#sendPacket. The internal packet queue is then inspected.	The internal packet queue contains the test packet.	Yes
2	Packet-Sender	A PacketSender instance is created and attached to a Connection. A test packet is sent using PacketSender#sendPacket. Sent/received data is checked at the attached Connection.	The attached connection receives the test packet.	Yes
3	PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there are no callbacks registered.	Nothing happens, including no errors.	Yes
4	PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there is a single callback registered. The status of the single callback will be checked.	The single callback runs and consumes the test packet.	Yes
5	PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there is a single callback registered but for a different packet type. The status of the single callback will be checked.	The single callback doesn't run.	Yes
6	PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there is one callback registered for the correct type but also one registered for a different packet type. The status of both callbacks will be checked.	The callback of the correct type runs but the other one doesn't run.	Yes
7	PacketListener	A PacketListener instance is created and PacketListener#runCallbacks is ran with a test packet when there are two callbacks registered for the correct type. The status of both callbacks will be checked.	Both callbacks run.	Yes
8	PacketListener	A PacketListener instance is created with a callback registered for the 'TestPacket' type and attached to a Connection. A test packet is then sent to the Connection and status of the registered callback is checked.	The callback runs.	Yes

9	Packet-Controller	A PacketController instance is created and attached to a Connection. The PacketController writes a test packet to the Connection and the Connection is checked to see if the packet has been received and is equal to the initially written test packet.	The test packet is received by the Connection and is equal to what was written to it.	Yes
10	Packet-Controller	A PacketController instance is created and attached to a Connection. The Connection writes a test packet to the PacketController and the PacketController attempts to read it. The received and read packet from the PacketController is checked for equality with the initially sent test packet.	The test packet is received by the PacketController and is equal to what was written to it.	Yes
11	Multi-FileUser-Manager	The users folder is created. A Multi-FileUserManager instance is created and a user that does not exist is queried, the result is checked.	An empty result is returned.	Yes
12	Multi-FileUser-Manager	The users folder is created and a file for a user is added. A MultiFileUserManager instance is created and a user that does exist is queried, the result is checked.	A result containing the user data is returned.	Yes
13	Multi-FileUser-Manager	The users folder is created. A Multi-FileUserManager instance is created and a user is created, the user file is then checked.	The file contains the correct user data.	Yes
14	Multi-FileUser-Manager	The users folder is created and a file for a user is added. A MultiFileUserManager instance is created and the existing user is deleted. The user file is then checked if it is present.	The file containing the user data has been deleted and is no longer present.	Yes
15	Configuration-Wrapper	The default configuration is loaded by the ConfigurationWrapper and its values checked against hard-coded defaults.	All values will match.	Yes
16	Server & Server-Builder	A ServerBuilder instance is created with callbacks attached, the Server instance is then built and a Connection attached to it. At each stage of the Connection lifecycle the status of the callbacks is checked.	After the Connection has connected, only the connect callback will have triggered. After the Connection disconnects, only the connect and disconnect callbacks will have triggered. After the server shuts down, all callbacks will have triggered.	Yes

17	VideoEncoder	A VideoEncoder instance is created and two images appended to the raw media save stream using VideoEncoder#appendToStream. The raw media save file is then read and its header tested for the correct values, the file body is tested for equality against the raw byte data of the two test images.	The file header contains the correct data about the file body. The file body bytes are equal to the raw byte data of the two test images.	Yes
18	VideoEncoder	A raw media save file is written. A VideoEncoder instance is created and processes the raw media save file with VideoEncoder#processRawMediaSave. The encoded video output file will be checked.	No errors will be produced during the encoding of the video and it will encode the video correctly such that it can be played by standard video viewing tools.	Yes
19	Main	A standard Server instance is constructed with authentication and callbacks added for disconnect and info packets. A Connection is then attached to the Server and an incorrect password is sent as part of the authentication process. The statuses of the callbacks are then checked.	The disconnect callback will be triggered with a reason citing incorrect credentials. The info callback will not be triggered.	Yes
20	Main	A standard Server instance is constructed with authentication and callbacks added for disconnect and info packets. A Connection is then attached to the Server and an incorrect username is sent as part of the authentication process (a username that is not associated with any active account). The statuses of the callbacks are then checked.	The disconnect callback will be triggered with a reason citing incorrect credentials. The info callback will not be triggered.	Yes
21	Main	A standard Server instance is constructed with authentication and callbacks added for disconnect and info packets. A Connection is then attached to the Server and correct, valid credentials are sent as part of the authentication process. The statuses of the callbacks are then checked.	The info callback will be triggered with a message informing the Connection of correct, valid credentials. The disconnect callback will not be triggered.	Yes

Here are the code sections for each of the above tests:

Test number	Test code screenshots

1

```
@Test
void testAddPacketToQueue() throws NoSuchFieldException, IllegalAccessException {
    PacketController packetControllerMock = mock(PacketController.class);
    Connection connectionMock = mock(Connection.class);
    doReturn(packetControllerMock).when(connectionMock).getPacketController();
    //so that the listening thread doesn't start - it isn't relevant to this test
    doReturn(false).when(connectionMock).isConnected();
    PacketSender packetSender = new PacketSender(connectionMock);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    Field packetQueueField = packetSender.getClass()
        .getDeclaredField("packetQueue");
    packetQueueField.setAccessible(true);

    Queue<Packet> packetQueue = (Queue<Packet>) packetQueueField.get(packetSender);
    assertEquals(0, packetQueue.size());
    packetSender.sendPacket(testPacket);
    assertEquals(1, packetQueue.size());
    assertEquals(testPacket, packetQueue.peek());

    try {
        packetSender.close();
    } catch (InterruptedException ignored) {}
}
```

2	<pre> @Test void testPacketSending() throws IOException, ClassNotFoundException { Connection localConnectionMock = mock(Connection.class); Socket localSocket, peerSocket; //create server on randomly assigned available port try (ServerSocket embeddedServer = new ServerSocket(0)) { //create socket connections from both sides localSocket = new Socket(embeddedServer.getInetAddress(), embeddedServer.getLocalPort()); peerSocket = embeddedServer.accept(); } doReturn(localSocket).when(localConnectionMock).getSocket(); doCallRealMethod().when(localConnectionMock).isConnected(); new ObjectOutputStream(peerSocket.getOutputStream()); PacketController localPacketController = new PacketController(localConnectionMock); ObjectInputStream peerObjectInputStream = new ObjectInputStream(peerSocket.getInputStream()); doReturn(localPacketController).when(localConnectionMock).getPacketController(); PacketSender packetSender = new PacketSender(localConnectionMock); int testData = new Random().nextInt(); TestPacket testPacket = new TestPacket(testData); packetSender.sendPacket(testPacket); TestPacket receivedPacket = (TestPacket) peerObjectInputStream.readObject(); try { localSocket.close(); peerSocket.close(); } catch (IOException ignored) {} assertNotNull(receivedPacket); assertEquals(testPacket.getTestData(), receivedPacket.getTestData()); } </pre>
3	<pre> @Test void testNoCallbacks() throws NoSuchMethodException { //so that the listening thread doesn't start - it isn't relevant to this test doReturn(false).when(connectionMock).isConnected(); packetListener = new PacketListener(connectionMock); int testData = new Random().nextInt(); TestPacket testPacket = new TestPacket(testData); Method runCallbacksMethod = packetListener.getClass() .getDeclaredMethod("runCallbacks", Packet.class); runCallbacksMethod.setAccessible(true); assertDoesNotThrow(() -> runCallbacksMethod.invoke(packetListener, testPacket)); } </pre>

4	<pre> @Test void testSingleCallbackRun() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException { //so that the listening thread doesn't start - it isn't relevant to this test doReturn(false).when(connectionMock).isConnected(); packetListener = new PacketListener(connectionMock); AtomicBoolean callbackRan = new AtomicBoolean(false); Consumer<TestPacket> callback = testPacket -> callbackRan.set(true); packetListener.addCallback(TestPacket.class, callback); int testData = new Random().nextInt(); TestPacket testPacket = new TestPacket(testData); Method runCallbacksMethod = packetListener.getClass() .getDeclaredMethod("runCallbacks", Packet.class); runCallbacksMethod.setAccessible(true); runCallbacksMethod.invoke(packetListener, testPacket); assertTrue(callbackRan.get()); } </pre>
5	<pre> @Test void testSingleCallbackNoRun() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException { //so that the listening thread doesn't start - it isn't relevant to this test doReturn(false).when(connectionMock).isConnected(); packetListener = new PacketListener(connectionMock); AtomicBoolean callbackRan = new AtomicBoolean(false); Consumer<Packet> callback = testPacket -> callbackRan.set(true); packetListener.addCallback(Packet.class, callback); int testData = new Random().nextInt(); TestPacket testPacket = new TestPacket(testData); Method runCallbacksMethod = packetListener.getClass() .getDeclaredMethod("runCallbacks", Packet.class); runCallbacksMethod.setAccessible(true); runCallbacksMethod.invoke(packetListener, testPacket); assertFalse(callbackRan.get()); } </pre>
6	<pre> @Test void testMultipleCallbacksRunOnlyOne() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException { //so that the listening thread doesn't start - it isn't relevant to this test doReturn(false).when(connectionMock).isConnected(); packetListener = new PacketListener(connectionMock); AtomicBoolean callbackOneRan = new AtomicBoolean(false); Consumer<TestPacket> callbackOne = testPacket -> callbackOneRan.set(true); AtomicBoolean callbackTwoRan = new AtomicBoolean(false); Consumer<Packet> callbackTwo = testPacket -> callbackTwoRan.set(true); packetListener.addCallback(TestPacket.class, callbackOne); packetListener.addCallback(Packet.class, callbackTwo); int testData = new Random().nextInt(); TestPacket testPacket = new TestPacket(testData); Method runCallbacksMethod = packetListener.getClass() .getDeclaredMethod("runCallbacks", Packet.class); runCallbacksMethod.setAccessible(true); runCallbacksMethod.invoke(packetListener, testPacket); assertTrue(callbackOneRan.get()); assertFalse(callbackTwoRan.get()); } </pre>

7	<pre> @Test void testMultipleCallbacksRunBoth() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException { //so that the listening thread doesn't start - it isn't relevant to this test doReturn(false).when(connectionMock).isConnected(); packetListener = new PacketListener(connectionMock); AtomicBoolean callbackOneRan = new AtomicBoolean(false); Consumer<TestPacket> callbackOne = testPacket -> callbackOneRan.set(true); AtomicBoolean callbackTwoRan = new AtomicBoolean(false); Consumer<TestPacket> callbackTwo = testPacket -> callbackTwoRan.set(true); packetListener.addCallback(TestPacket.class, callbackOne); packetListener.addCallback(TestPacket.class, callbackTwo); int testData = new Random().nextInt(); TestPacket testPacket = new TestPacket(testData); Method runCallbacksMethod = packetListener.getClass() .getDeclaredMethod("runCallbacks", Packet.class); runCallbacksMethod.setAccessible(true); runCallbacksMethod.invoke(packetListener, testPacket); assertTrue(callbackOneRan.get()); assertTrue(callbackTwoRan.get()); } </pre>
8	<pre> @Test void testPacketListening() throws IOException, ExecutionException, InterruptedException { Connection localConnectionMock = mock(Connection.class); Socket localSocket, peerSocket; //create server on randomly assigned available port try (ServerSocket embeddedServer = new ServerSocket(0)) { //create socket connections from both sides localSocket = new Socket(embeddedServer.getInetAddress(), embeddedServer.getLocalPort()); peerSocket = embeddedServer.accept(); } doReturn(localSocket).when(localConnectionMock).getSocket(); doCallRealMethod().when(localConnectionMock).isConnected(); ObjectOutputStream peerObjectOutputStream = new ObjectOutputStream(peerSocket.getOutputStream()); PacketController localPacketController = new PacketController(localConnectionMock); doReturn(localPacketController).when(localConnectionMock).getPacketController(); packetListener = new PacketListener(localConnectionMock); CompletableFuture<String> completableFuture = new CompletableFuture<>(); Consumer<TestPacket> callback = testPacket -> { //deactivate the listening thread completableFuture.complete("Callback Ran Successfully"); try { localSocket.close(); peerSocket.close(); } catch (IOException ignored) {} }; packetListener.addCallback(TestPacket.class, callback); int testData = new Random().nextInt(); TestPacket testPacket = new TestPacket(testData); peerObjectOutputStream.writeObject(testPacket); assertEquals("Callback Ran Successfully", completableFuture.get()); } </pre>

9	<pre> @Test void testWritePacket() throws IOException, ClassNotFoundException { //instantiated to stop the PacketController from hanging ObjectOutputStream peerOutputStream = new ObjectOutputStream(peerConnection.getSocket().getOutputStream()); PacketController localPacketController = new PacketController(localConnection); int testData = new Random().nextInt(); TestPacket testPacket = new TestPacket(testData); localPacketController.writePacketToSocket(testPacket); Packet receivedPacket; try (ObjectInputStream peerInputStream = new ObjectInputStream(peerConnection.getSocket().getInputStream())) { receivedPacket = (Packet) peerInputStream.readObject(); } peerOutputStream.close(); localPacketController.close(); assertNotNull(receivedPacket); assertEquals(testPacket.getType(), receivedPacket.getType()); assertEquals(testPacket.getTestData(), ((TestPacket) receivedPacket).getTestData()); } </pre>
10	<pre> @Test void testReadPacket() throws IOException, ClassNotFoundException { //instantiated to stop the PacketController from hanging ObjectOutputStream peerOutputStream = new ObjectOutputStream(peerConnection.getSocket().getOutputStream()); PacketController localPacketController = new PacketController(localConnection); int testData = new Random().nextInt(); TestPacket testPacket = new TestPacket(testData); peerOutputStream.writeObject(testPacket); Packet receivedPacket = localPacketController.readPacketFromSocket(); peerOutputStream.close(); localPacketController.close(); assertEquals(testPacket.getType(), receivedPacket.getType()); assertEquals(testPacket.getTestData(), ((TestPacket) receivedPacket).getTestData()); } </pre>
11	<pre> @Test public void test GetUser Not Exists() { assertTrue(userManager.getUser("noUser").isEmpty()); } </pre>
12	<pre> @Test public void test GetUser Exists() throws IOException { Files.writeString(userFile, "testUser\nhashedPassword\nfalse"); Optional<User> userOptional = userManager.getUser("testUser"); assertTrue(userOptional.isPresent()); assertEquals("testUser", userOptional.get().getUsername()); assertEquals("hashedPassword", userOptional.get().getHashedPassword()); assertFalse(userOptional.get().isAdmin()); } </pre>

13	<pre> @Test public void testCreateUser() throws Exception { assertFalse(Files.exists(userFile)); User user = User.fromHashedPassword("testUser", "hashedPassword", false); userManager.createUser(user); assertTrue(Files.exists(userFile)); List<String> lines = Files.readAllLines(userFile); assertEquals("testUser", lines.get(0)); assertEquals("hashedPassword", lines.get(1)); assertFalse(Boolean.parseBoolean(lines.get(2))); } </pre>
14	<pre> @Test public void deleteUser() throws Exception { assertFalse(Files.exists(userFile)); Files.writeString(userFile, "testUser\nhashedPassword\nfalse"); assertTrue(Files.exists(userFile)); userManager.deleteUser("testUser"); assertFalse(Files.exists(userFile)); } </pre>
15	<pre> @Test public void testConfigurationAndWrapper() { Path configFile = Paths.get(System.getProperty("user.dir")).resolve("server.properties"); assertFalse(Files.exists(configFile)); assertEquals(ConfigurationWrapper.getInstance().getServerAddress(), "127.0.0.1"); assertEquals(ConfigurationWrapper.getInstance().getServerPort(), 8192); assertEquals(ConfigurationWrapper.getInstance().getMaxConnections(), 5); assertEquals(ConfigurationWrapper.getInstance().getVideoDuration(), Duration.ofMinutes(1)); assertEquals(ConfigurationWrapper.getInstance().getVideoSaveDirectory(), Paths.get("./videos/")); assertEquals(ConfigurationWrapper.getInstance().getUsersSaveDirectory(), Paths.get("./users/")); assertEquals(ConfigurationWrapper.getInstance().getDefaultUser(), User.fromPlainTextPassword("admin", LoginPacket.hashPassword("5J6*9XodH&&mAUBz"), true)); } </pre>

16

```
@Test
public void testServer() throws UnknownHostException {
    CompletableFuture<String> shutdownCompletableFuture = new CompletableFuture<>();
    Consumer<Server> serverShutdownListener =
        serv -> shutdownCompletableFuture.complete("Shutdown");
    CompletableFuture<String> connectCompletableFuture = new CompletableFuture<>();
    BiConsumer<Connection, Server> connectListener =
        (con, serv) -> connectCompletableFuture.complete("Connected");
    CompletableFuture<String> disconnectCompletableFuture = new CompletableFuture<>();
    BiConsumer<Connection, Server> disconnectListener =
        (con, serv) -> disconnectCompletableFuture.complete("Disconnected");

    ServerBuilder serverBuilder = new ServerBuilder(0, InetAddress.getLocalHost())
        .onServerShutdown(serverShutdownListener)
        .onClientConnect(connectListener)
        .onClientDisconnect(disconnectListener);

    try (Server server = serverBuilder.build()) {
        Socket socket = new Socket(InetAddress.getLocalHost(), server.getPort());
        Connection connection = new Connection(socket);

        assertEquals("Connected", connectCompletableFuture.get());
        assertFalse(disconnectCompletableFuture.isDone());
        assertFalse(shutdownCompletableFuture.isDone());

        connection.close();
        assertEquals("Connected", connectCompletableFuture.get());
        assertEquals("Disconnected", disconnectCompletableFuture.get());
        assertFalse(shutdownCompletableFuture.isDone());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
    try {
        assertEquals("Connected", connectCompletableFuture.get());
        assertEquals("Disconnected", disconnectCompletableFuture.get());
        assertEquals("Shutdown", shutdownCompletableFuture.get());
    } catch (InterruptedException | ExecutionException ignored) {}
}
```

17

```

@Test
public void testAppendToStream(@TempDir Path tempDir) throws IOException {
    //test data
    long numberofFrames = 2;
    long startTimestamp = System.currentTimeMillis() - 1000;
    long endTimestamp = startTimestamp + 1000;
    BufferedImage[] testImages = getTestImages();

    //running the method to be tested with the test data
    VideoFileManager videoFileManager = new VideoFileManager(tempDir, Duration.ofMinutes(5));
    VideoEncoder videoEncoder = new VideoEncoder(videoFileManager);
    videoEncoder.appendToStream(testImages[0], startTimestamp);
    videoEncoder.appendToStream(testImages[1], endTimestamp);

    //serializing images so that they can be compared to the output
    ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
    byteArrayOutputStream.writeBytes(videoEncoder.compressImage(testImages[0]));
    byteArrayOutputStream.writeBytes(videoEncoder.compressImage(testImages[1]));

    Path rawMediaSaveFile = videoFileManager.getCurrentSaveFile().orElseThrow(FileNotFoundException::new);
    //checking that the data outputted by the method being tested is correct
    try (InputStream inputStream = Files.newInputStream(rawMediaSaveFile)) {
        DataInputStream dataInputStream = new DataInputStream(inputStream) {
            assertEquals(numberofFrames, dataInputStream.readLong());
            assertEquals(startTimestamp, dataInputStream.readLong());
            assertEquals(endTimestamp, dataInputStream.readLong());
            assertArrayEquals(byteArrayOutputStream.toByteArray(), dataInputStream.readAllBytes());
        };
    }
}

```

18

```

@Test
public void testProcessRawMediaSave(@TempDir Path tempDir) throws IOException {
    //test data
    long numberofFrames = 2;
    long startTimestamp = System.currentTimeMillis() - 1000;
    long endTimestamp = startTimestamp + 1000;
    BufferedImage[] testImages = getTestImages();

    //file setup
    VideoFileManager videoFileManager = new VideoFileManager(tempDir, Duration.ofMinutes(5));
    VideoEncoder videoEncoder = new VideoEncoder(videoFileManager);

    Path rawMediaSaveFile = videoFileManager.getCurrentSaveFile().orElseThrow(FileNotFoundException::new);
    //writing test data to file
    try (OutputStream outputStream = Files.newOutputStream(rawMediaSaveFile)) {
        DataOutputStream dataOutputStream = new DataOutputStream(outputStream) {
            dataOutputStream.writeLong(numberofFrames);
            dataOutputStream.writeLong(startTimestamp);
            dataOutputStream.writeLong(endTimestamp);
            ImageIO.write(testImages[0], "jpg", dataOutputStream);
            ImageIO.write(testImages[1], "jpg", dataOutputStream);
        };
    }

    //running the method to be tested
    videoEncoder.processRawMediaSave(rawMediaSaveFile);

    File videoOutputFile = new File(rawMediaSaveFile.toString().replace(".crms", ".mp4"));
    //todo parse output file to check length, fps etc. (maybe even frames
}

```

19	<pre>@Test public void testIncorrectPassword() throws UnknownHostException { CompletableFuture<String> result = new CompletableFuture<>(); ServerBuilder serverBuilder = new ServerBuilder(0, InetAddress.getLocalHost()); Main.addAuthenticationToServerBuilder(serverBuilder, mockUserManager()); try (Server server = serverBuilder.build()) { Socket socket = new Socket(InetAddress.getLocalHost(), server.getPort()); Connection connection = new Connection(socket); connection.getPacketListener().addCallback(InfoPacket.class, infoPacket -> { result.complete(infoPacket.getInfo()); }); connection.getPacketListener().addCallback(DisconnectPacket.class, disconnectPacket -> { result.complete(disconnectPacket.getReason()); }); connection.getPacketSender().sendPacket(LoginPacket.fromPlainTextPassword("testUsername", "incorrectPassword")); assertEquals("Incorrect Credentials", result.get()); } catch (Exception e) { throw new RuntimeException(e); } }</pre>
20	<pre>@Test public void testIncorrectUsername() throws UnknownHostException { CompletableFuture<String> result = new CompletableFuture<>(); ServerBuilder serverBuilder = new ServerBuilder(0, InetAddress.getLocalHost()); Main.addAuthenticationToServerBuilder(serverBuilder, mockUserManager()); try (Server server = serverBuilder.build()) { Socket socket = new Socket(InetAddress.getLocalHost(), server.getPort()); Connection connection = new Connection(socket); connection.getPacketListener().addCallback(InfoPacket.class, infoPacket -> { result.complete(infoPacket.getInfo()); }); connection.getPacketListener().addCallback(DisconnectPacket.class, disconnectPacket -> { result.complete(disconnectPacket.getReason()); }); connection.getPacketSender().sendPacket(LoginPacket.fromPlainTextPassword("testUsername", "incorrectPassword")); assertEquals("Incorrect Credentials", result.get()); } catch (Exception e) { throw new RuntimeException(e); } }</pre>

21

```

@Test
public void testSuccessfulLogin() throws UnknownHostException {
    CompletableFuture<String> result = new CompletableFuture<>();
    ServerBuilder serverBuilder = new ServerBuilder(0, InetAddress.getLocalHost());
    Main.addAuthenticationToServerBuilder(serverBuilder, mockUserManager());
    try (Server server = serverBuilder.build()) {
        Socket socket = new Socket(InetAddress.getLocalHost(), server.getPort());
        Connection connection = new Connection(socket);
        connection.getPacketListener().addCallback(InfoPacket.class, infoPacket -> {
            result.complete(infoPacket.getInfo());
        });
        connection.getPacketListener().addCallback(DisconnectPacket.class, disconnectPacket -> {
            result.complete(disconnectPacket.getReason());
        });
        connection.getPacketSender().sendPacket(
            LoginPacket.fromPlainTextPassword(
                "testUsername", "testPassword"
            )
        );
        assertEquals("Correct Credentials", result.get());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

```

Below are the successfully running test outputs for the connection framework module:

```

[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running xyz.benanderson.scs.networking.connection.PacketControllerTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.44 s
[INFO] Running xyz.benanderson.scs.networking.connection.PacketListenerTest
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.02 s
[INFO] Running xyz.benanderson.scs.networking.connection.PacketSenderTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.006 s
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0

```

Figure 188: Connection framework module all tests passing

Below are the successfully running test outputs for the server module:

```
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running xyz.benanderson.scs.server.account.MultiFileUserManagerTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.062 s
[INFO] Running xyz.benanderson.scs.server.ServerLoginTest
[INFO] User 'testUsername' logged in successfully.
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.439 s
[INFO] Running xyz.benanderson.scs.server.configuration.ConfigurationTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s
[INFO] Running xyz.benanderson.scs.server.video.VideoEncoderTest
[INFO] Finished encoding video 2023-01-26T21:37:18.170241219.mp4
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.886 s
[INFO] Running xyz.benanderson.scs.server.networking.ServerTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 11, Failures: 0, Errors: 0, Skipped: 0
```

Figure 189: Server module all tests passing

There aren't any white box unit tests for the client module as it is almost entirely GUI code and therefore very difficult to automatically unit test. Testing for the client module has been carried out manually and documented thoroughly in the development section. The black box testing section below contains further relevant testing information for the client GUI.

Black Box Testing

The following black box testing table is for the client software as defined in my design section, mainly the camera connection window as that contains the majority of user inputs:

Test number	Test input field	Test input data	Expected outcome	Was expected outcome achieved? (if no, what was the actual outcome?)
1	Camera Port	the port is five five five five	Error message to the user – “Port number must be an integer”.	Yes
2	Camera Port	70000	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.	Yes
3	Camera Port	65536	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.	Yes

4	Camera Port	0	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.	Yes
5	Camera Port	-1	Error message to the user – “Port number must be between 1 and 65535 (inclusive)”.	Yes
6	Camera Port	80	Either an error message due to lack of privileges, or a successful server setup depending on the privileges with which the camera server was ran.	Yes
7	Camera Port	9001	Successful server setup assuming that port 9001 is not being used by another application.	Yes
8	Camera Port	8192	Successful connection assuming there is a camera server on port 8192. Otherwise, error message to user denoting that no camera server could be found on that port.	Yes
9	Camera Address	Ncu28*("6!"£ \$%^&*()_-=	Error message to the user specifying that a connection error occurred and that the camera address was invalid.	Yes
10	Camera Address	google.com	Error message to the user specifying that a connection refused occurred because the target address isn't running a camera server.	No. The application hangs as it connects to the server and waits for data.
11	Camera Address	127.0.0.1	Error message to the user specifying that a connection refused occurred because the target address isn't running a camera server.	Yes
12	Username	!"£\$ %^&*()`¬ 1a-=_+[] {}';:@#~,./<>?	Disconnected from camera due to incorrect credentials and message shown to user to notify of this.	Yes
13	Password	!"£\$ %^&*()`¬ 1-=_+[] {}';:@#~,./<>?	Disconnected from camera due to incorrect credentials and message shown to user to notify of this.	Yes
14	Cancel Button	Mouse Click	The camera connection window closes and no further action occurs.	Yes
15	Connect Button	Mouse Click	The client software attempts to connect to the server software at the specified address and closes the camera connection window.	Yes
16	Connect To Camera	Mouse Click	Opens the camera connection window.	Yes

	Button (on main GUI)			
--	----------------------	--	--	--

All black box tests on the client software apart from number 10 have resulted in the expected behaviour.

I believe the error experienced on test 10 was caused by the fact that Google's domain name configuration doesn't directly reject the network connection despite the fact there is no active network socket listening on the target port. I think this because attempting to connect to different IP addresses and domain names which don't have any camera server running just results in a 'Connection Refused' error. Ultimately, this error is the result of a user attempting to connect to a foreign host which will react in a completely unpredictable manner, hence there is little that can be done about this error and it will only occur with certain hosts. As long as the user restricts their inputs to servers which they know are valid and safe to connect to, no problems will be experienced with the camera address input field.

The following black box testing table is for the configuration inputs in the server software, as these are the only user inputs in the server software.

Test number	Test input field	Test input data	Expected outcome	Was expected outcome achieved? (if no, what was the actual outcome?)
1	server.address	fwehc8sd9	Exception is thrown informing the user that it is not a valid host to run the server on.	Yes
2	server.address	1.2.3.4	Exception is thrown informing the user that they cannot use the requested address.	Yes
3	server.address	127.0.0.1	Server starts successfully and is only accessible from the current computer (because 127.0.0.1 is the localhost interface).	Yes
4	server.address	0.0.0.0	Server starts successfully and is accessible by any machine on the local network.	Yes
5	server.port	the port is five five five five	Warning is displayed stating the acceptable port range and that the server has been started on a random port. The server is started on a random port.	Yes
6	server.port	70000	Warning is displayed stating the acceptable port range and that the server has been started on a random port. The server is started on a random port.	Yes
7	server.port	65536	Warning is displayed stating the acceptable port range and that the server has been started on a random port. The server is started on a random port.	Yes

8	server.port	0	Server starts on a random port (chosen by OS).	Yes
9	server.port	-1	Warning is displayed stating the acceptable port range and that the server has been started on a random port. The server is started on a random port.	Yes
10	server.port	80	If they aren't running with elevated privileges, an exception is thrown informing the user that they don't have the required permissions to run the camera server on that port. But if they are running with elevated privileges, the server starts on port 80.	Yes
11	server-.max-connections	-1	Camera server starts successfully but no clients can connect to it.	Yes
12	server-.max-connections	0	Camera server starts successfully but no clients can connect to it.	Yes
13	server-.max-connections	1	Camera server starts successfully but only one client can connect to it.	Yes
14	video.save-directory	56347890	The directory '56347890' is created in the current working directory and used to store video data.	Yes
15	video.save-directory	videos	The directory 'videos' is created in the current working directory and used to store video data.	Yes
16	video.save-directory	./videos/	The directory 'videos' is created in the current working directory and used to store video data.	Yes
17	video.duration.number	dfxasw	The default duration number of 60 is used.	Yes
18	video.duration.number	-1	The default duration number of 60 is used.	No. The video recording & encoding thread throws an exception and video recording fails. The application still runs fine and allows client connections.
19	video.duration.unit	century	The default duration unit of 'seconds' is used.	No. The application throws an exception informing the user that 'century' is not a valid constant in

				the 'ChronoUnit' class. The application terminates.
20	video.duration.unit	seconds	The duration unit of 'seconds' is used.	Yes
21	user-s.save-directory	56347890	The directory '56347890' is created in the current working directory and used to store user data.	Yes
22	user-s.save-directory	users	The directory 'users' is created in the current working directory and used to store user data.	Yes
23	user-s.save-directory	./users/	The directory 'users' is created in the current working directory and used to store user data.	Yes
24	users.default.username	x703-N*(CDJ(9FK;	The user 'x703-N*(CDJ(9FK;' is successfully created and can be used to authenticate.	Yes
25	users.default.username	vcx/dedg	An exception is thrown informing the user that '/' is a reserved character that cannot be used in username due to the operating systems file implementation.	Yes
26	users.default.username	admin	The user 'admin' is successfully created and can be used to authenticate.	Yes
27	users.default.password	5J6*9XodH&&mAUBz	The user is successfully created with password '5J6*9XodH&&mAUBz' and can be used to authenticate.	Yes
28	users.default.is_admin	Hfwehui289	The value defaults to 'true'.	Yes
29	users.default.is_admin	Hfwehui289	The value defaults to 'false' – the user is not considered an admin.	Yes
30	users.default.is_admin	true	The value is 'true' – the user is considered an admin.	Yes

All black box tests on the server software apart from numbers 18 and 19 have resulted in the expected behaviour.

The error experienced on test 18 occurs when the user enters a video duration number which is less than or equal to zero; I'd consider this a purposeful erroneous input as it is clear to the user that the duration of videos should be greater than zero – hence, this error does not need to be considered severe. This error could be resolved by implementing the standard integer input validation as designed in the design section – defaulting to a different value if it is outside of the acceptable range (value must be greater than zero).

The error experienced on test 19 occurs when the user enters an invalid video duration unit. In this test specifically, the input was 'century', however this is not considered a valid time unit by the java standard library class 'ChronoUnit', resulting in an error. This error could definitely have been handled more gracefully instead of letting the program terminate with an exception. As long as the user sticks to the list of valid time units, this error won't occur. This error could be resolved by implementing input validation on the user input by checking if it is in the list of valid time units; if it isn't, then either halting program execution or defaulting to a the default value could be done.

Success Criteria

After developing and testing each prototype of the software, I evaluated which success criteria was completed as part of that prototype. Below, I have provided a summary as to the achievement status of each success criteria point:

Criteria	Has it been achieved? (yes/no/partially)	How do I know it's been achieved?	Additional Comments
1. Easy to install	Yes	User feedback has voiced happiness with the easy installation process. The program is just one file.	N/A
2. Function without internet connection	Yes	No code makes use of any internet services, only arbitrary addresses on the local network are accessed.	N/A
3. Don't send data outside the local network	Yes	No code makes network connections with any IP addresses other than the one specified by the user.	N/A
4. Working network communication	Yes	Unit and integration tested to be successfully working.	N/A
5. Correct serialization and deserialization of network packets	Yes	Unit and integration tested to be successfully working.	N/A
6. Minimal yet intuitive user interface	Yes	User has approved the GUI design and given positive feedback about using the user interface.	N/A
7. Streaming of camera frames over the network	Yes	Integration tested to be successfully working.	N/A
8. Can login to the camera server using credentials	Yes	Unit and integration tested to be successfully working.	N/A
9. Users can be added and deleted to the camera	Yes	Unit and integration tested to be successfully working.	N/A
10. Downloadable video footage	Partially	Footage that is stored on the camera server can be downloaded manually.	Video footage can be downloaded from the camera server's file system

			manually, however support for the feature is not built into the software.
11. Cross-platform	Yes	The software has been written in Java which is a cross-platform programming language, the software was also end-to-end tested on both Linux and Windows.	N/A
12. Expandable storage	Yes	The software was run on three systems which all had different storage volumes: 256GB, 2TB and 1TB.	N/A
13. Server software can access the camera on the machine	Yes	Unit and integration tested to be successfully working.	N/A
14. Functioning video creation from camera frames	Yes	Unit and integration tested to be successfully working.	N/A
15. Save video locally to client computer	No	N/A	Videos are not encoded and saved on the client software as footage is received. Instead, videos are only stored on the server.

Fully Achieving Point 10

To fully achieve success criteria point 10, I would implement some new packet types: `MediaDownloadRequestPacket`, `MediaDownloadPacket`, `MediaDownloadListRequestPacket` and `MediaDownloadListPacket`. The client software would have an element added to the user interface which, when pressed, would send a `MediaDownloadListRequestPacket` to the server software. When the server software receives this packet type, it would respond with a `MediaDownloadListPacket` containing a list of all the saved and encoded videos on the camera server.

The client software would then display this list of downloadable videos to the user and allow them to select one, once they did, it would send a `MediaDownloadRequestPacket` containing the video name to the camera server. The camera server would then asynchronously read the raw byte data stored in the encoded video file and put it into `MediaDownloadPacket` objects, which would be transmitted back to the client software and used to reconstruct the encoded video file.

All automated network responses would be programmed by adding callbacks to the `Connection`'s which would respond to the relevant `...Packet` types.

Achieving Point 15

To fully achieve success criteria point 15, I would re-use the video storage and encoding classes I programmed for the server software, in the client software. This would require the video recording submodule to be moved to a common/shared Maven module such as the networking module – however, I would probably end up creating a separate Maven module for all code related to videos.

I would then adjust the `MediaPacket` callback on the client software to write the received media frames to raw media save files using the `appendToStream` method in the `VideoEncoder` class. I would also move the `startVideoEncodingScheduledJob` method from the `Main` class of the server software into the separate Maven module for video, and use this on both the client and server software to process and encode the raw media save files into videos in a batch, scheduled job.

Usability

Client Software

As the client software has gone through multiple iterations, I decided to ask the stakeholder for a final piece of feedback about the client software as part of the project evaluation. They said the following:

"I think the client program works really well and it's easy for me to use. It all looks nice and the user interface elements are where they should be. I don't feel overwhelmed using the software either."

The above feedback clearly illustrates that the client software is very usable and stands out in the following ways:

- Easy to use and navigate
- Looks visually appealing
- Intuitively placed user interface components
- Minimalist and not overwhelming

Here are all the success criteria features which were categorised as usability features and an analysis on the extent to which each of them were reached:

Success Criteria For Usability	How do I know it has been achieved?
1. Easy to install	The stakeholder has expressed clear happiness with the installation process and how the software is to set up. <i>"Installation of the system was fairly simple as it was just one file with no additional dependencies."</i>
6. Minimal yet intuitive user interface	The stakeholder has clearly expressed how they " <i>don't feel overwhelmed using the software</i> ", indicating that the minimal design has had the desired affect. The fact that the stakeholder said " <i>the user interface elements are where they should be</i> " also shows that the elements are placed intuitively.

9. Users can be added and deleted to/from the camera	This feature has been extensively unit tested, end-to-end integration tested, and used by the stakeholder with no problems.
10. Downloadable video footage	This feature has been implemented to an extent and confirmed to function exactly as suspected by the user. It is fairly usable for the stakeholder as they " <i>can download video files from my server using other normal file transfer tools</i> ".
11. Cross-platform	The software has been ran on multiple computer systems with different operating systems (Microsoft Windows 10 and Linux Mint) with no problems. It is designed to be completely cross-platform, using the system API offered by the JRE JVM present on the machine.
12. Expandable storage	The software has been ran on multiple computer systems with different storage volumes and drive types (solid state drive and magnetic disk drive) with no problems. It is designed to utilise the cross-platform, adaptable storage medium API offered by the JRE JVM for the relevant operating system.
15. Save video locally to client computer	This feature has not been implemented but does not majorly affect the usability of the solution. See above discussion on 'Achieving Point 15'.

I have annotated some key features of all GUI windows and pop-ups below:

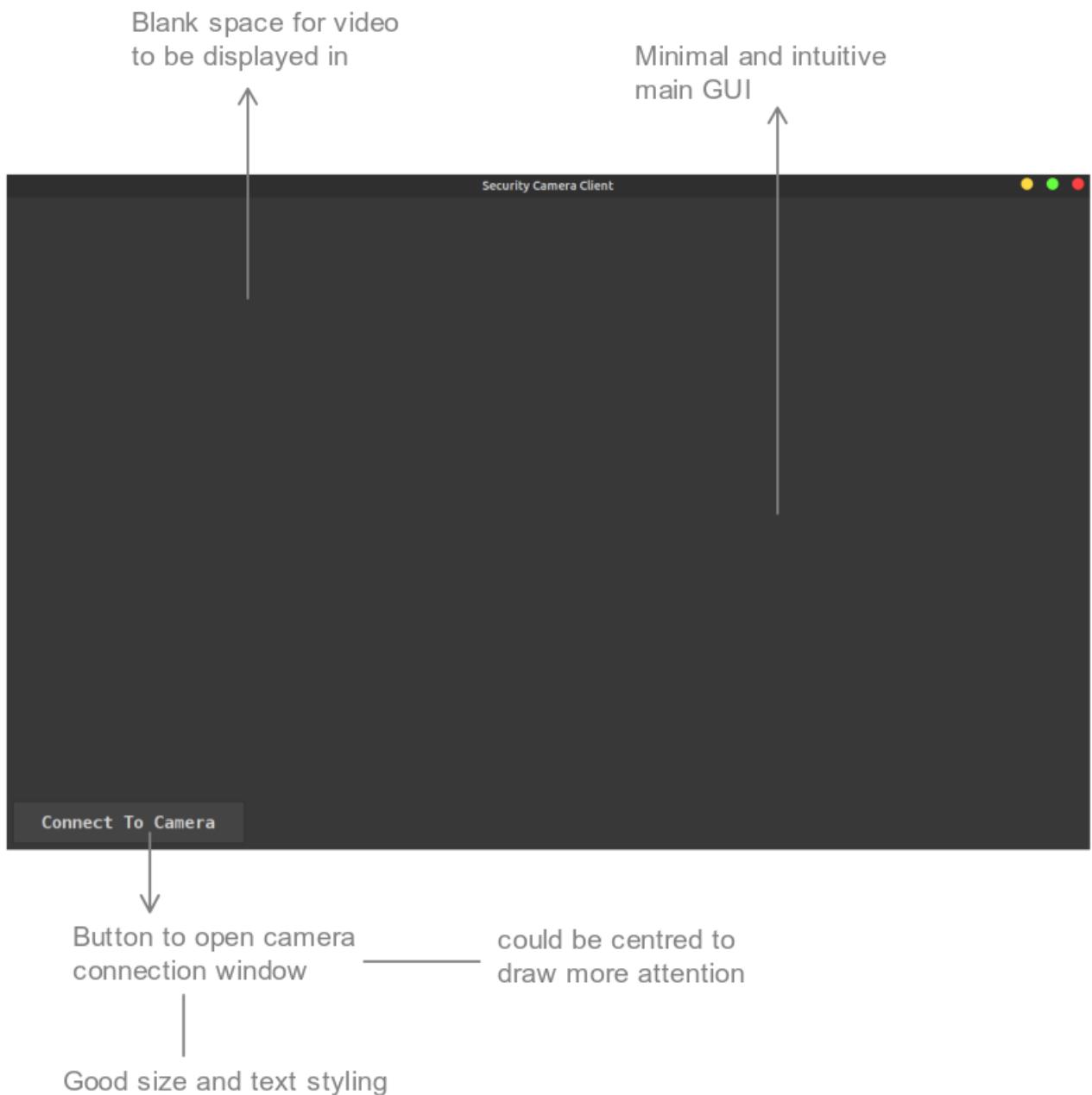


Figure 190: Final annotated main GUI

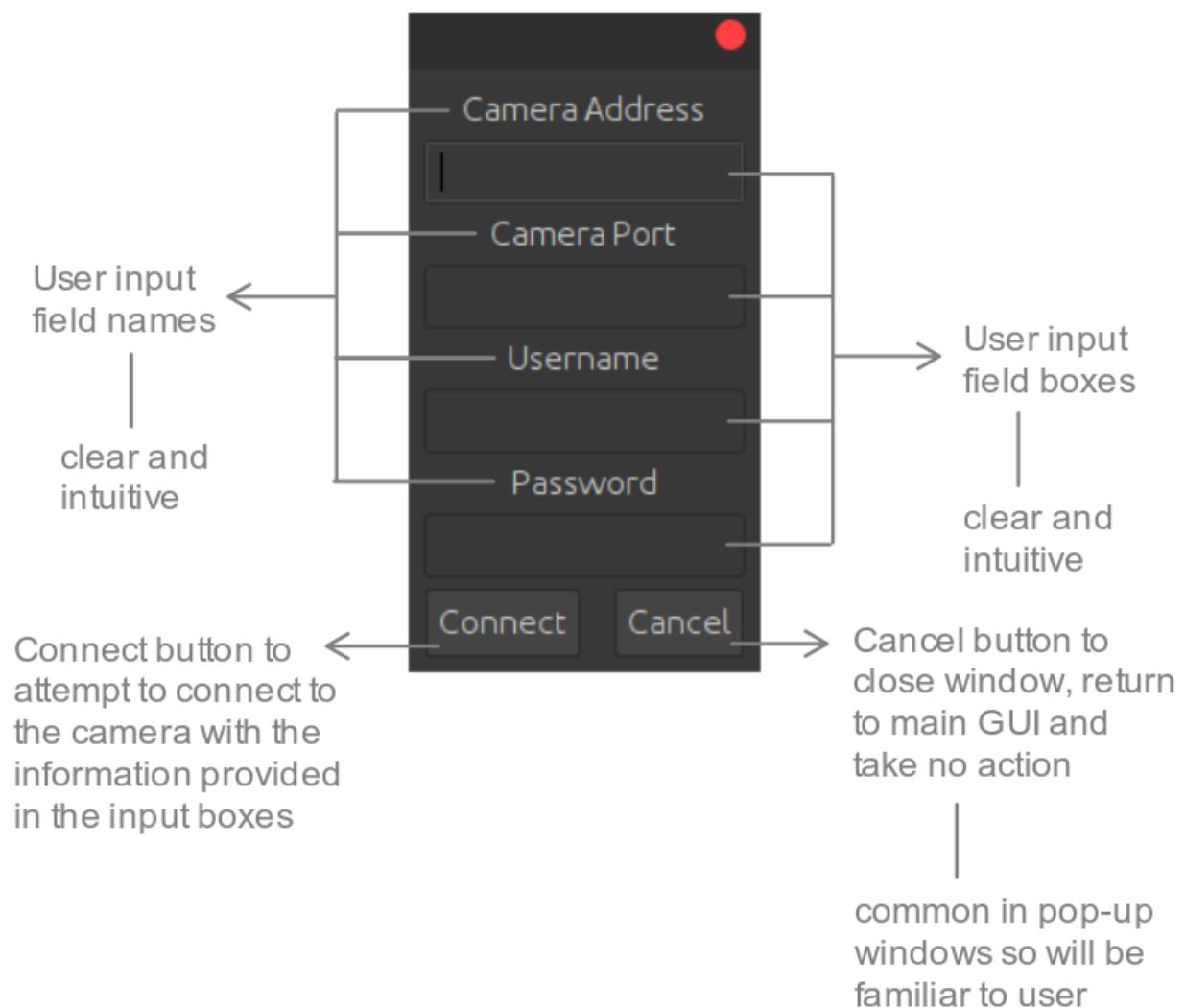


Figure 191: Final annotated camera connection window

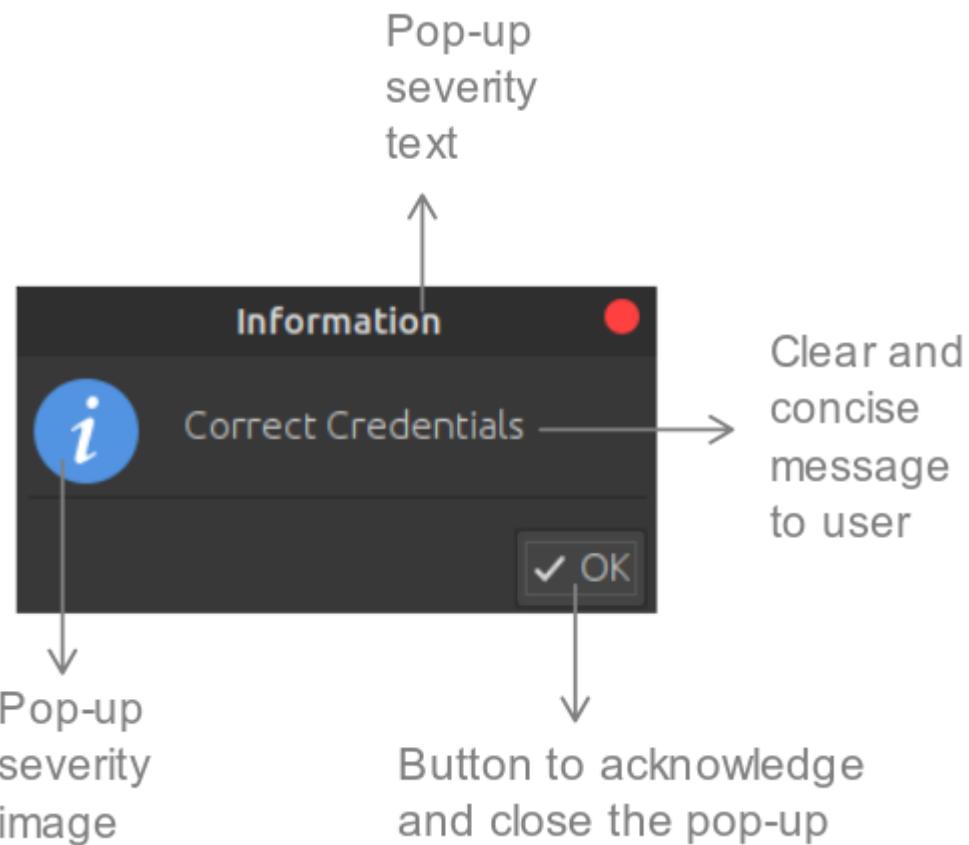


Figure 192: Final annotated correct credentials pop-up

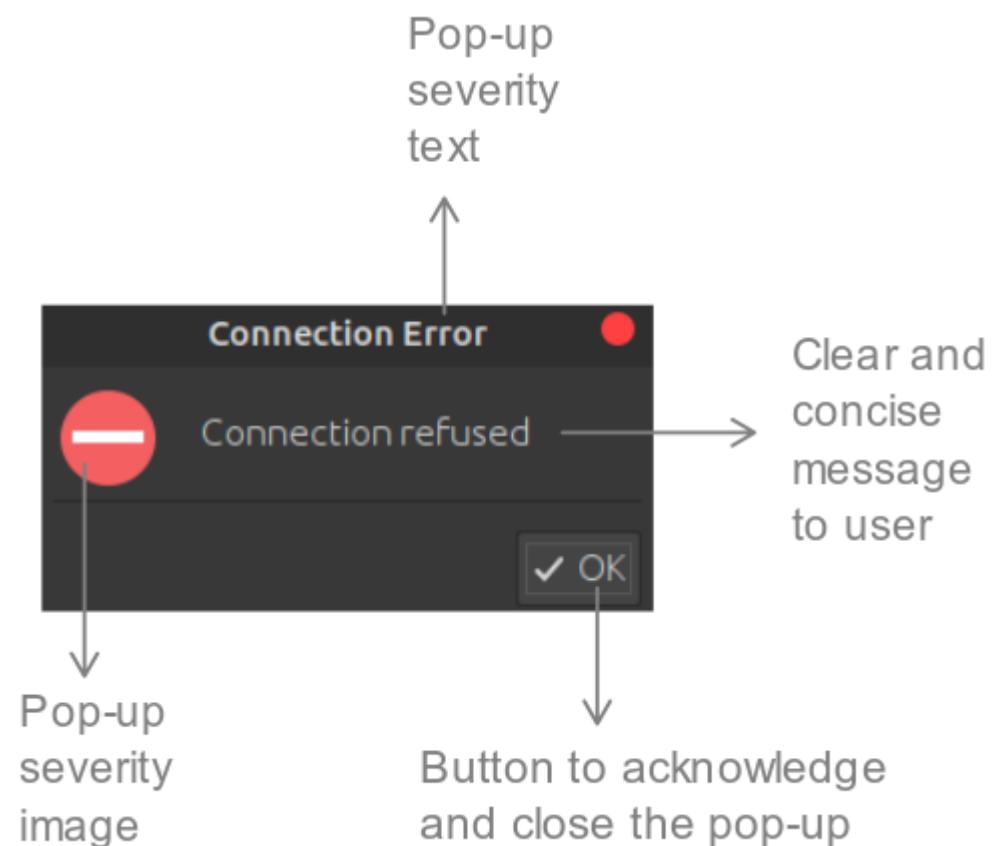


Figure 193: Final annotated connection error pop-up

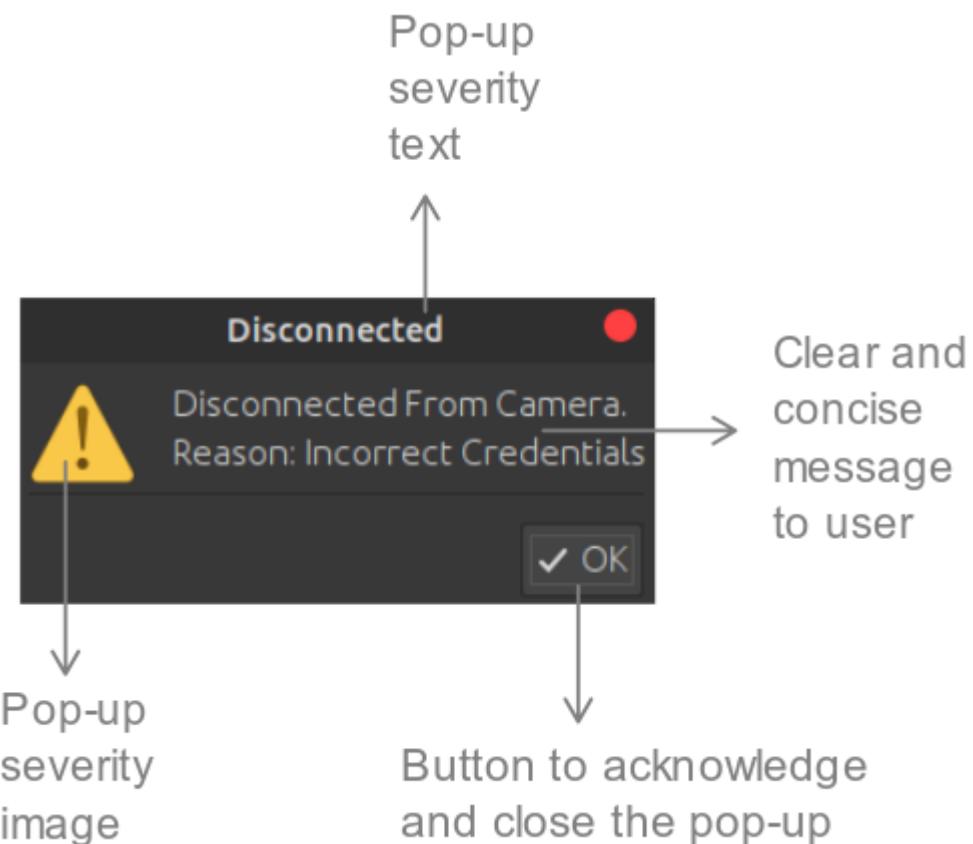


Figure 194: Final annotated incorrect credentials pop-up

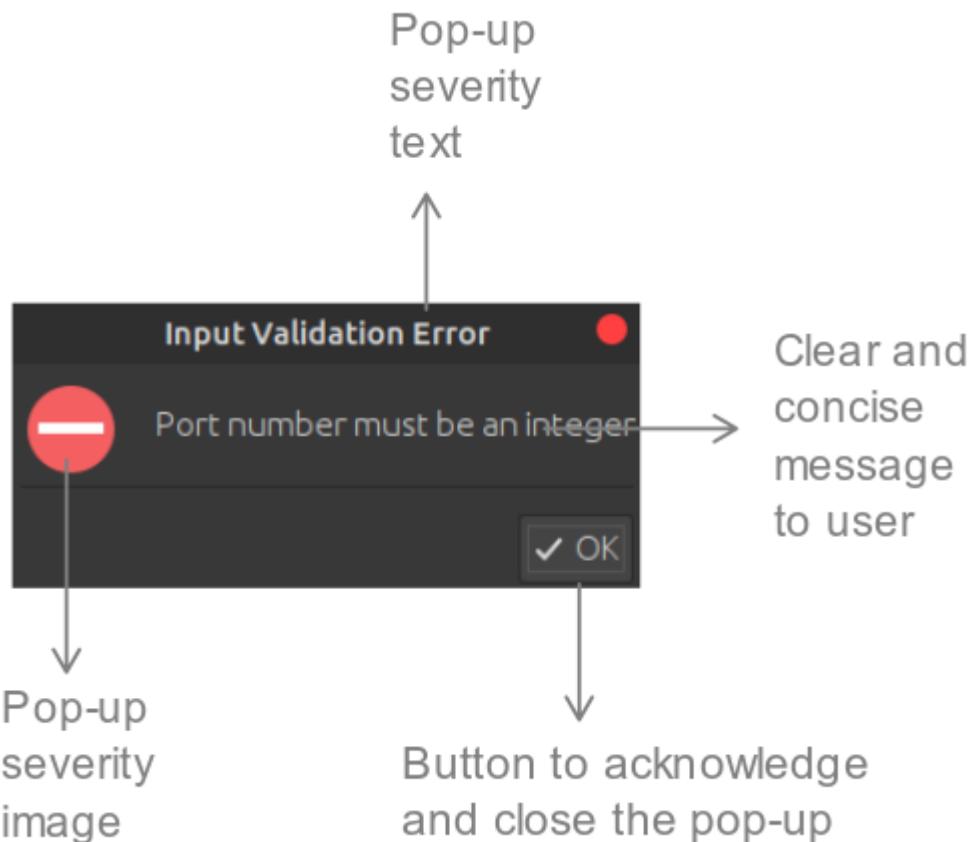


Figure 195: Final annotated port must be integer pop-up

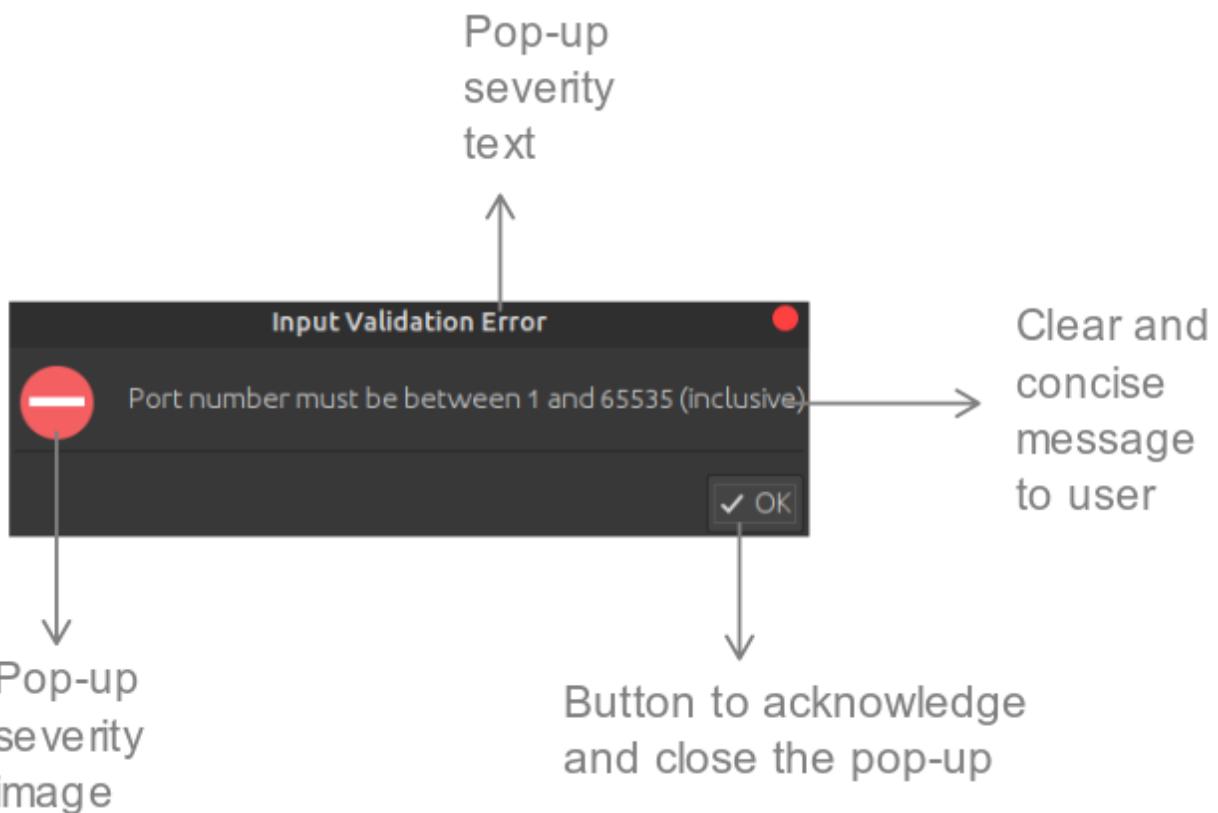


Figure 196: Final annotated port invalid range pop-up

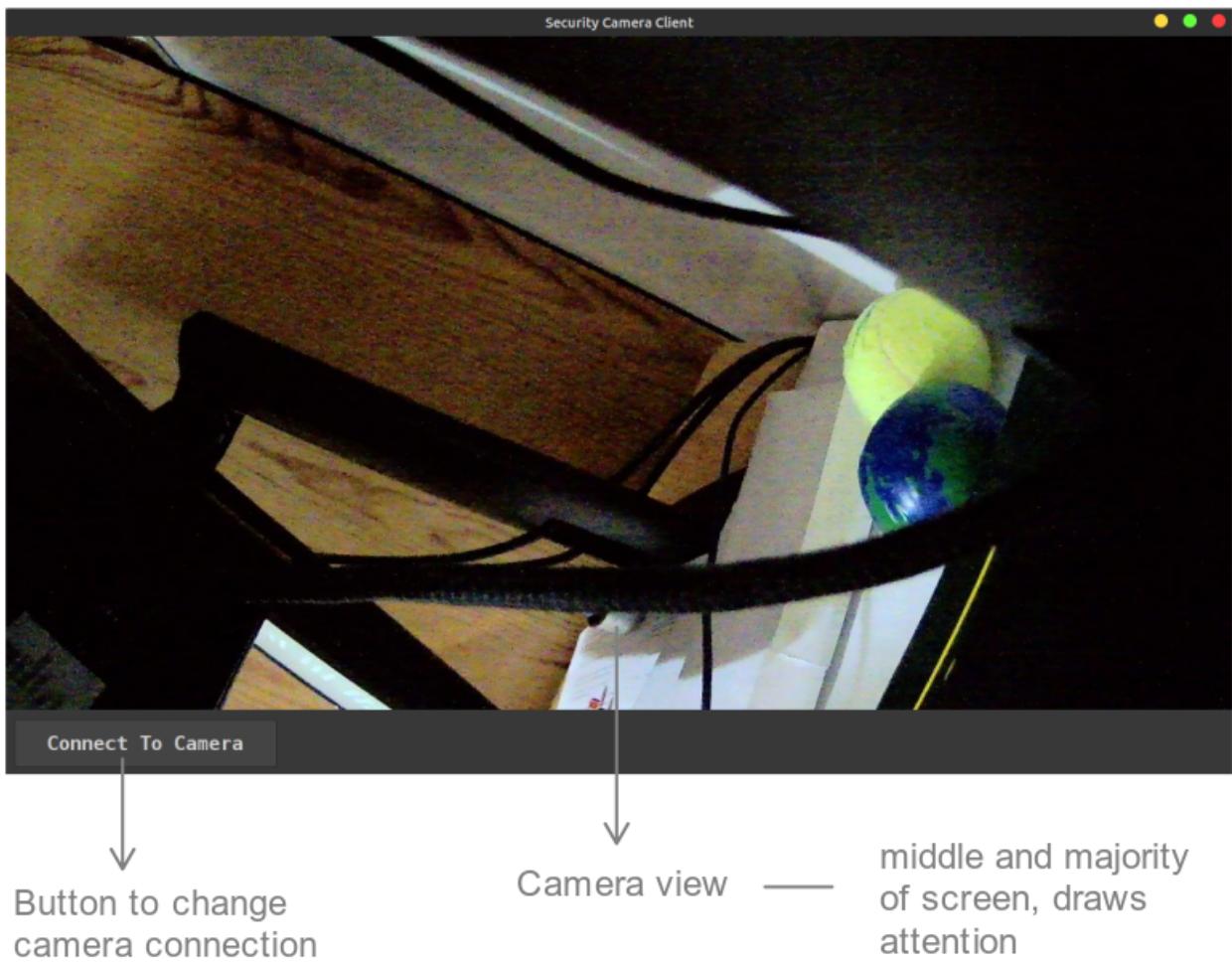


Figure 197: Final annotated main GUI with camera view

In all the above screenshots, the GUI looks slightly different to what it did in the second prototype documentation. This is because the program is running on a different computer with a different operating system theme, hence the software has adapted its GUI automatically to align with the system theme as planned.

Server Software

As the server software has gone through multiple iterations, I decided to ask the stakeholder for a final piece of feedback about the server software as part of the project evaluation. They said the following:

"The server software does the job well and gives me informative output messages when required. The configuration file is incredibly intuitive and easy to use - everything works as expected."

The above feedback clearly illustrates that the server software is very usable and stands out in the following ways:

- All features work well
- Provides informative outputs to user
- Easily configurable

I have annotated some key features of the server software output and configuration file below:



Figure 198: Final annotated server output

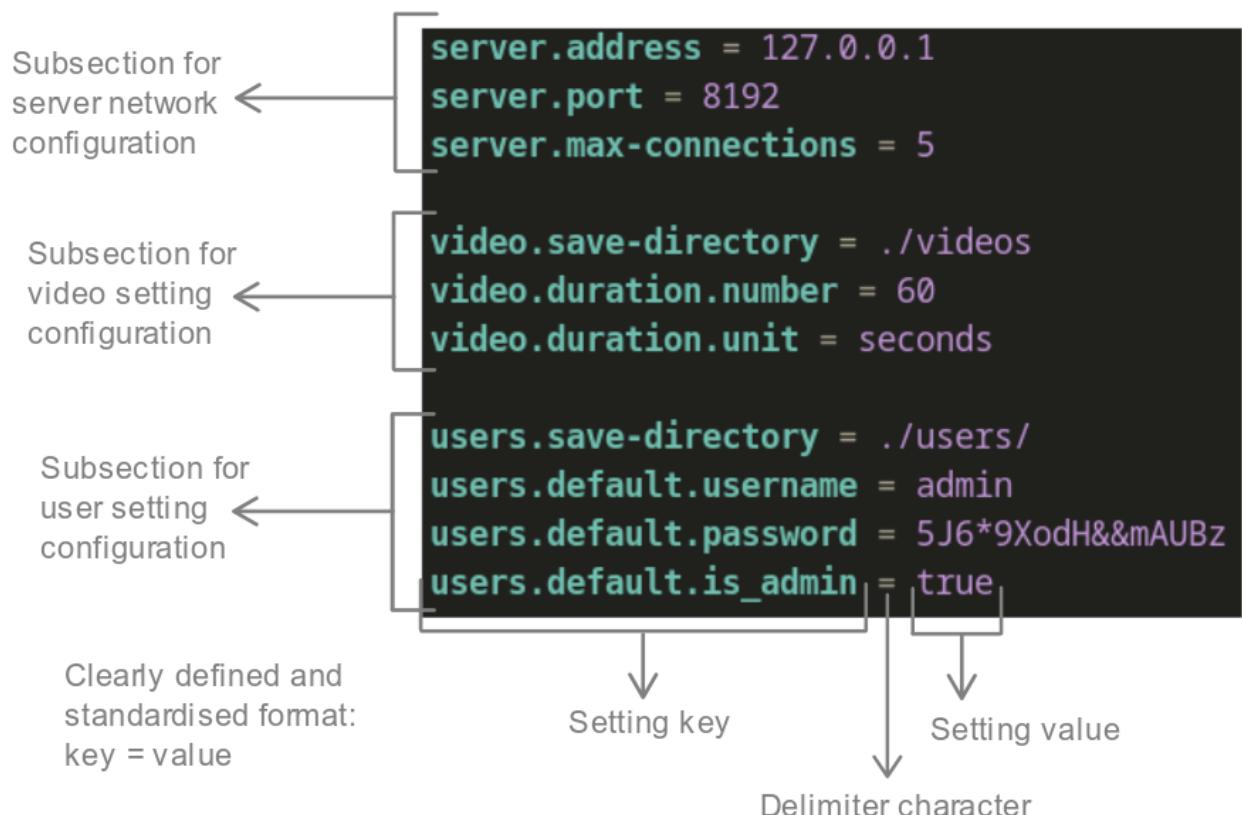


Figure 199: Final annotated server configuration file

Project Successes

I successfully delivered the majority of the success criteria to the fullest extent possible, providing most of what I planned in my final solution. Some aspects of the solution changed throughout the design and development process to better streamline the application for the user and make it easier for me to develop according to the project window.

Throughout the analysis phase of the project, I believe I successfully identified all key features and requirements for the security camera system and built a clear understanding of what I needed to achieve through my success criteria.

In the design phase of the project, I structured my project to be incredibly organised and easily maintainable – this certainly made my life easier when it came to development. I also designed the system to be very efficient and use multithreading to execute tasks concurrently. I believe the high-level abstractions I used for each module/submodule (and within each submodule) created a solution which is very easy to understand on both a high and low level, as well as being incredibly modular and de-coupled – components can be replaced or switched out very easily to customise and adapt the solution.

During the development of the solution, I carried out further optimisations on the design of the software such as the `UserFileManager` class, transforming its methods from having an $O(n \log n)$ time complexity to having an $O(1)$ time complexity. I used an IDE to significantly improve the speed of development as it provided key features such as syntax highlighting and code completion. I also used my own private Git server as a version control system for the project, this allowed me to be able to navigate between code iterations, clearly document code changes and safely store an off-site backup of the project code. The Maven build tool was essential in minimising repeated code as I used the `Networking` Maven module that I built as a dependency in both the server and client software modules with just a few lines of configuration. Unit testing with JUnit5 ended up being incredibly vital to the development of the solution, as writing and running unit tests helped me uncover lots of bugs which I thoroughly documented and fixed – these unit tests also give me confidence in the functionality of the software and its future maintainability.

Solution Limitations & Maintenance

One problem with the video files recorded and encoded by the server software, is that they are ever-so-slightly sped up. I predict that the encoded videos are approximately 1.2 times the speed of the real events. I believe that this problem is due to the encoding library I am using, and slight delays in recording could also contribute to the issues. The issue is mostly not noticeable, and also quite technically complex, therefore it is not really worth spending time to try and fix it.

Another limitation of the video encoding library that I am using is that it takes a long time to encode raw media save files into their video form. After a while, this will result in a backlog of videos to encode, but there won't be any impact on application performance. I'd like to use another video encoding library for this project to see if there would be reductions in video encoding time, however due to the fact that I'm programming in Java, it means that it will be very difficult to use an encoding library other than one that is written in Java. Sadly, Java is not a popular language for video encoding as it doesn't have low-level hardware access – such as the fact that it can't encode videos using the GPU (graphical processing unit), which would significantly reduce encoding time. Hence, the library that I am currently using was the only one which was simple to use and actually worked.

One aspect of my solution which is both a benefit and limitation, is my use of multithreading. It allows for tasks to run concurrently and not disrupt the flow of execution, however I fear that I may have over-used it, with each connection to the server creating atleast two extra threads. This puts a strain on the CPU and could severly impact machine performance after enough clients connect to the server.

When it comes to maintenance of the solution, the system was built in a very modular way such that components can easily be replaced by different implementations of the abstract class – this makes it easy to change elements of the software in future iterations. However, one downside of

my implementation and the fact that the system does not access the internet unprompted, is that the system does not support automatic updates for new versions of the solution that are released. New solutions versions must be downloaded manually from the software vendor (myself) and the old executable version replaced with the new one. As discussed in the analysis section, this does pose a slight security risk as patches for any security vulnerabilities must be manually downloaded after being implemented, instead of the patches being deployed automatically to running instances of the software.

Another great decision I made regarding maintenance of the solution, was that all libraries used are versions in the Maven public repository and therefore will not stop being available. The libraries have also been compiled into the final solution executable at a specific version, meaning no external changes will affect the software.

Solution Improvements

Images that are appended to raw media save files are currently compressed, however the images sent across the network from the camera server to the connected clients are not compressed. This results in a less efficient use of bandwidth as more data is sent than necessary (a full quality image has already been deemed unnecessary for the stakeholder). This could result in higher latency and slower user response times – affecting the software usability and decreasing the efficiency that success criteria point 7 has been implemented at. Therefore, an improvement for a future iteration of the server software is to use the previously implemented image compression to compress images before sending them across the network to all clients. This would result in more efficient use of bandwidth and a slight improvement in networking latency; there would be less data to send so it would take less time. It could also be implemented in such a way that there would be zero extra computational overhead, because image compression is already done as part of the video encoding submodule, hence the output of this compression would just be re-used and sent to the connected clients. This would enhance the usability of the software and increase the efficiency of success criteria point 7.

Another limitation of my current server software, is that raw media save files have quite a large file size. This was found during white box testing. This relates to success criteria point 12, as unnecessary extra storage volume may be being used. This is because for every media frame, the entire compressed image is appended to the raw media save file. Another contributing factor to the file size, is the fact that the image compression algorithm provided by the Java standard library is not very good – the size of high-quality images and compressed images aren't that far apart. An idea to significantly decrease the file size of raw media save files is to only store information of pixels that have changed between one media frame and the next, instead of storing the whole media frame. This change would require a bit more code when it comes to video encoding, however would result in a much smaller raw media save file size as security cameras don't often see lots of pixels constantly changing. Smaller raw media save files would result in more versatile, usable software as it could be ran on systems with lower storage volume (below the current recommendation for the video recording feature).

Success criteria points 10 and 15 should also be considered limitations of the solution as they are not fully complete and slightly limit the usability/functionality of the software. These points have been discussed more in above sections.

A limitation of my approach to this project, is that in the white box testing I didn't use the standard test suite that I designed and implemented earlier in the process. This could have potentially resulted in faulty code and less standardised or maintainable tests. However, for the most part, the effect on the project of missing out the use of the standard test suite has been negligible.

Another limitation of my approach to this project is that I didn't use the Log4J logging library which I initially discussed in my pre-development choices section. Instead, I ended up using the built-in 'System' output streams (standard-out, standard-err) and adding relevant messages to inform the user of the type of output (info, warning, error). This approach is much less standardised and configurable for the user, however I still believe it will meet the stakeholder's needs.

One potential way that networking latency of the media frames could be reduced is by utilising user datagram protocol (UDP) to send media frame image data faster and less precisely (impacting the way success criteria point 7 is implemented). Due to the nature of videos not requiring 100% of their data, transmission control protocol (TCP) is actually less suitable for sending media frames as TCP ensures that all frames are delivered in their entirety, and in order. But this is not required for the media frames, therefore some small performance improvements could be gained by utilising the more reckless but speedy user datagram protocol. However, currently there is only a negligible amount of network latency, therefore changing protocols is not necessarily an essential improvement.

Final Program Code

```
package xyz.benanderson.scs.networking.connection;

import lombok.AccessLevel;
import lombok.Getter;
import lombok.Setter;

import java.io.IOException;
import java.net.Socket;
import java.util.UUID;
import java.util.function.Consumer;

/**
 * Connection interface which provides high level access to the connection between
 * two services. This component contains the key elements for a peer-to-peer connection
 * including the Socket, PacketController, PacketSender and PacketListener.
 */
👤 Ben Anderson
public class Connection implements AutoCloseable {

    /**
     * Socket attribute with connection-package level getter visibility
     */
    2 usages
    @Getter(AccessLevel.PACKAGE)
    private final Socket socket;

    /**
     * PacketController attribute with connection-package level getter visibility
     */
    1 usage
    @Getter(AccessLevel.PACKAGE)
    private final PacketController packetController;

    /**
     * Unique ID attribute to help tell difference between connections
     */
    1 usage
    @Getter(AccessLevel.PUBLIC)
    private final UUID id;
```

Figure 200: Final Connection class part 1

```
/**  
 * PacketSender attribute with getter  
 */  
1 usage  
@Getter(AccessLevel.PUBLIC)  
private final PacketSender packetSender;  
  
/**  
 * PacketListener attribute with getter  
 */  
1 usage  
@Getter(AccessLevel.PUBLIC)  
private final PacketListener packetListener;  
  
/**  
 * Constructor for {@code Connection} class.  
 *  
 * @param socket the lower-level Java socket which the {@code Connection}  
 * object will be built on top of.  
 * @throws IOException thrown if an I/O errors when preparing the I/O streams.  
 */  
👤 Ben Anderson  
public Connection(Socket socket) throws IOException {  
    this.id = UUID.randomUUID();  
    this.socket = socket;  
    this.packetController = new PacketController(connection: this);  
    this.packetSender = new PacketSender(connection: this);  
    this.packetListener = new PacketListener(connection: this);  
    this.disconnectListener = con -> {};  
}  
  
1 usage  
@Getter  
@Setter  
private Consumer<Connection> disconnectListener;
```

Figure 201: Final Connection class part 2

```
/**  
 * Method to override default implementation in {@link AutoCloseable} interface.  
 * Requests {@code PacketController} to close input & output streams of the socket,  
 * the closes the socket directly.  
 *  
 * @throws Exception thrown if an I/O errors when closing the I/O streams or socket.  
 */  
➊ Ben Anderson  
@Override  
public synchronized void close() throws Exception {  
    socket.close();  
    getDisconnectListener().accept(⠁ this);  
    getPacketController().close();  
    getPacketSender().close();  
    getPacketListener().close();  
}  
  
/**  
 * Method to check the open/close state of the connection.  
 *  
 * @return true if this connection is connected to a peer, false if it isn't.  
 */  
15 usages ➋ Ben Anderson  
public boolean isConnected() { return getSocket().isConnected() && !getSocket().isClosed(); }  
}
```

Figure 202: Final Connection class part 3

```
package xyz.benanderson.scs.networking.connection;

import xyz.benanderson.scs.networking.Packet;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

/**
 * The {@code PacketController} class provides an encapsulation around the low-level
 * reading/writing operations from/to the {@code Socket} which underpins the
 * {@link Connection}.
 * No API is publicly exposed and the class has no visibility modifier set, therefore
 * defaulting to package-private (only accessible by classes in the same package -
 * components of the connection abstraction). A {@code PacketController} is only
 * expected to be used by a {@link PacketListener} and a {@link PacketSender}.
 */
14 usages  Ben Anderson
class PacketController implements AutoCloseable {

    /**
     * ObjectOutputStream wrapping & encapsulating the low-level
     * output stream of the {@code Socket}
     */
    6 usages
    private final ObjectOutputStream objectOutputStream;
    /**
     * ObjectInputStream wrapping & encapsulating the low-level
     * input stream of the {@code Socket}
     */
    4 usages
    private final ObjectInputStream objectInputStream;
```

Figure 203: Final PacketController class part 1

```

/**
 * Constructor for {@code PacketController} class
 *
 * @param connection {@code Connection} object that this {@code PacketController} is
 *                   controlling the packets for.
 * @throws IOException thrown if an I/O error occurs when accessing the
 * input or output streams
 */
5 usages  Ben Anderson
public PacketController(Connection connection) throws IOException {
    //create ObjectOutputStream from socket's abstract OutputStream
    this.objectOutputStream = new ObjectOutputStream(
        //get OutputStream from the Socket in the Connection
        //and pass it as the argument to the ObjectOutputStream constructor
        connection.getSocket().getOutputStream()
    );
    //create ObjectInputStream from socket's abstract InputStream
    this.objectInputStream = new ObjectInputStream(
        //get InputStream from the Socket in the Connection
        //and pass it as the argument to the ObjectInputStream constructor
        connection.getSocket().getInputStream()
    );
}

/**
 * Method to write a {@code Packet} object to the output stream of the socket.
 * Visibility is 'protected' in order to encapsulate the method and make it only
 * accessible to classes in the same package (components of the connection abstraction).
 *
 * @param packet {@code Packet} object to write to the socket
 * @throws IOException thrown if an I/O error occurs
 */
2 usages  Ben Anderson
protected void writePacketToSocket(Packet packet) throws IOException {
    //synchronize access to the output stream across threads to avoid race conditions
    //in a multithreaded environment - because this method is not guaranteed to
    //be run by one thread at a time (atomically)
    synchronized (objectOutputStream) {
        //write the packet object to the output stream
        objectOutputStream.writeObject(packet);
        //flush the output stream to ensure it is pushed out of memory and
        //across the network to the receiving Socket
        objectOutputStream.flush();
        //reset the output stream to clear the cache and reset the state
        //of the output stream
        objectOutputStream.reset();
    }
}

```

Figure 204: Final PacketController class part 2

```

/**
 * Method to read a {@code Packet} object from the input stream of the socket.
 * Visibility is 'protected' in order to encapsulate the method and make it only
 * accessible to classes in the same package (components of the connection abstraction).
 *
 * @throws IOException thrown if an I/O error occurs
 * @throws ClassNotFoundException thrown if the class of the object received does
 * not exist in the source code of the receiving application.
 */
2 usages  Ben Anderson
protected Packet readPacketFromSocket() throws IOException, ClassNotFoundException {
    //synchronize access to the input stream across threads to avoid race conditions
    //in a multithreaded environment - because this method is not guaranteed to
    //be run by one thread at a time (atomically)
    synchronized (objectInputStream) {
        //read an object from the objectInputStream and then cast it to a
        //`Packet` object (changes the type to be the `Packet` class)
        return (Packet) objectInputStream.readObject();
    }
}

/**
 * Method overriding the default `close()` implementation from {@link AutoCloseable}.
 * This method closes the input and output streams of the Socket.
 *
 * @throws IOException thrown if an exception occurs when closing the I/O streams.
 */
Ben Anderson
@Override
public void close() throws IOException {
    //close object input & output streams - will also close the underlying
    //abstract InputStream & OutputStream on the Socket.
    try {
        objectInputStream.close();
        objectOutputStream.close();
    } catch (Exception ignored) {}
}
}

```

Figure 205: Final PacketController class part 3

```
package xyz.benanderson.scs.networking.connection;

import xyz.benanderson.scs.networking.Packet;

import java.io.EOFException;
import java.io.IOException;
import java.net.SocketException;
import java.util.*;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.function.Consumer;

/**
 * The {@code PacketListener} class exposes a high-level API to developers,
 * allowing them to create callbacks for received packets from the parent {@code Connection}.
 * This class internally uses a map to correlate packet types to a list of packet callbacks
 * for that type. These packet callbacks are executed asynchronously from packet listening
 * thread and therefore should not block the flow of execution with blocking callouts.
 */
10 usages  Ben Anderson
public class PacketListener implements AutoCloseable {

    //encapsulated map data structure storing a list of callback code blocks to run for each
    //packet type.
    6 usages
    private final Map<Class<? extends Packet>, List<Consumer<Packet>>> callbacks;
    //encapsulated asynchronous thread, on which, packets are listened for.
    2 usages
    private final Thread packetListeningThread;
    //private, thread-safe, atomic boolean variable to control the condition-controlled
    //loop in the packet listening thread.
    2 usages
    private final AtomicBoolean listeningForPackets = new AtomicBoolean(initialValue: true);

    /**
     * Constructor for {@code PacketListener} class
     *
     * @param connection {@code Connection} object that this {@code PacketListener} is
     *                   listening for packets from.
     */
    7 usages  Ben Anderson
    public PacketListener(Connection connection) {
```

Figure 206: Final PacketListener class part 1

```
public PacketListener(Connection connection) {
    //initialise the callback map with an empty hashmap
    this.callbacks = new HashMap<>();

    //create thread
    this.packetListeningThread = new Thread(() -> {
        //code to run in the thread
        while (listeningForPackets.get() && connection.isConnected()) {
            //try to read the packet from the underlying PacketController
            try {
                Packet packet = connection.getPacketController().readPacketFromSocket();
                //if the packet was successfully read from the connection,
                //run callbacks associated with the packet
                runCallbacks(packet);
            } catch (EOFException | SocketException disconnected) {
                break;
            } catch (IOException e) {
                //exception will be thrown if the connection was closed, in which
                //case ignore it and return
                if (!connection.isConnected()) return;

                //if the packet was NOT successfully read from the connection,
                //log the error to the standard error stream.
                System.err.println("[ERROR] An error occurred whilst reading a packet" +
                    " from a connection:");
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                //if the packet received was not a valid packet type, log the error
                //to the standard error stream.
                System.err.println("[WARNING] An unknown packet type was received" +
                    " from a connection: ");
                e.printStackTrace();
            }
        }
        try {
            if (connection.isConnected())
                connection.close();
        } catch (Exception ignored) {}
    }, "Packet Listening Thread" /* name of the thread */);

    //start the asynchronous packet listening thread
    this.packetListeningThread.start();
}
```

Figure 207: Final *PacketListener* class part 2

```
/**  
 * Method to add a callback to the parent connection. The callback will be run when a packet  
 * is received with the correct type for that callback.  
 *  
 * @param packetClass type of the packet that the callback will be triggered by  
 * @param callback the code to run with the packet  
 */  
17 usages  Ben Anderson  
public <T extends Packet> void addCallback(Class<T> packetClass, Consumer<T> callback) {  
    //if a callback of the packet class type has not already been registered,  
    //then add it to the map with a new, empty linked list.  
    if (!callbacks.containsKey(packetClass))  
        callbacks.put(packetClass, new LinkedList<>());  
  
    //add the callback to the list corresponding to the packet class  
    //key in the callbacks map  
    callbacks.get(packetClass).add((Consumer<Packet>) callback);  
}  
  
/**  
 * Method with private visibility to run all callbacks associated with the type  
 * of the packet provided.  
 *  
 * @param packet packet to run callbacks on  
 */  
1 usage  Ben Anderson  
private void runCallbacks(Packet packet) {  
    //check if any callbacks are registered for the packet type  
    if (callbacks.containsKey(packet.getType()))  
        //if callbacks are registered run all callbacks for the packet type  
        //using the packet as the argument  
        callbacks.get(packet.getType()).forEach(callback -> callback.accept(packet));  
}
```

Figure 208: Final PacketListener class part 3

```
/**  
 * Method overriding the default `close()` implementation from {@link AutoCloseable}.  
 * This method disables the constant listening for packets on the asynchronous  
 * 'Packet Listening Thread' and waits for the thread to die.  
 *  
 * @throws InterruptedException thrown if an exception occurs when waiting for the thread to die.  
 */  
↳ Ben Anderson  
@Override  
public void close() throws InterruptedException {  
    //set loop-controlling variable to false in order to disable the while loop  
    //in the packet listening thread  
    this.listeningForPackets.set(false);  
    //wait for the packet listening thread to die  
    //    this.packetListeningThread.join();  
}  
}
```

Figure 209: Final PacketListener class part 4

```
package xyz.benanderson.scs.networking.connection;

import xyz.benanderson.scs.networking.Packet;

import java.io.IOException;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.atomic.AtomicBoolean;

/**
 * The {@code PacketSender} class exposes a high-level API to developers,
 * allowing them to send packets across the parent {@code Connection}.
 * This class internally uses a {@code Queue} for {@code Packet}s waiting to be sent.
 * This packet queue is then accessed internally to send packets asynchronously.
 */
7 usages  Ben Anderson
public class PacketSender implements AutoCloseable {

    //encapsulated packet queue which stores packets to be sent asynchronously
    4 usages
    private final Queue<Packet> packetQueue;
    //encapsulated asynchronous thread which packets are sent from
    3 usages
    private final Thread packetSendingThread;
    //private, thread-safe, atomic boolean variable to control the condition-controlled
    //loop in the packet sending thread.
    2 usages
    private final AtomicBoolean sendingPackets = new AtomicBoolean(initialValue: true);

    /**
     * Constructor for {@code PacketSender} class
     *
     * @param connection {@code Connection} object that this {@code PacketSender} is
     *                   sending the packets for.
     */
    3 usages  Ben Anderson
    public PacketSender(Connection connection) {
        //initialise queue attribute with a ConcurrentLinkedQueue object,
        //this allows asynchronous access to the queue without any race
        //conditions or unexpected behaviour.
        this.packetQueue = new ConcurrentLinkedQueue<>();
```

Figure 210: Final PacketSender class part 1

```
//create thread
this.packetSendingThread = new Thread(() -> {
    //code to run in the thread
    while (sendingPackets.get() && connection.isConnected()) {
        //using peek instead of poll so that if a packet failed to send
        //it can be tried again
        Packet packetToSend = packetQueue.peek();
        //if there are no packets in the queue, go to the start of the while loop
        //and check again for packets to send
        if (packetToSend == null) continue;

        //try to write the packet to the underlying PacketController
        try {
            connection.getPacketController().writePacketToSocket(packetToSend);
            //if the packet was successfully sent across the connection,
            //remove it from the queue of packets to send
            packetQueue.remove();
        } catch (IOException e) {
            //exception will be thrown if the connection was closed, in which
            //case ignore it and return
            if (!connection.isConnected()) return;

            //if the packet was NOT successfully sent across the connection,
            //keep it in the queue of packets to send and instead log the error
            //to the standard error stream - this results in trying to
            //send the packet again later.
            System.err.println("[ERROR] An error occurred whilst sending a packet" +
                " across a connection:");
            e.printStackTrace();
        }
    }
    try {
        if (connection.isConnected())
            connection.close();
    } catch (Exception ignored) {}
}, "Packet Sending Thread" /* name of the thread */);

//start the asynchronous packet sending thread
this.packetSendingThread.start();
}
```

Figure 211: Final PacketSender class part 2

```
/*
 * Public API method to be used when a developer wishes to send a {@code Packet}
 * across the parent {@code Connection}. This method adds the provided {@code Packet} object
 * to the packet queue. The packet will then be sent across the {@code Connection} asynchronously.
 *
 * @param packet {@code Packet} to send across the connection
 */
10 usages  Ben Anderson
public void sendPacket(Packet packet) {
    //add packet to queue
    this.packetQueue.add(packet);
}

/*
 * Method overriding the default `close()` implementation from {@link AutoCloseable}.
 * This method disables the constant sending of packets on the asynchronous
 * 'Packet Sending Thread' and waits for the thread to die.
 *
 * @throws InterruptedException thrown if an exception occurs when waiting for the thread to die.
 */
Ben Anderson
@Override
public void close() throws InterruptedException {
    //set loop-controlling variable to false in order to disable the while loop
    //in the packet sending thread
    this.sendingPackets.set(false);
    //wait for the packet sending thread to die
    this.packetSendingThread.join();
}

}
```

Figure 212: Final PacketSender class part 3

```
package xyz.benanderson.scs.networking.packets;

import lombok.Getter;
import xyz.benanderson.scs.networking.Packet;

/**
 * Packet sent by a connection when the connection wishes to disconnect from its peer.
 */
10 usages  ↳ Ben Anderson
public class DisconnectPacket extends Packet {

    /**
     * Reason for the connection disconnecting
     */
    1 usage
    @Getter
    private final String reason;

    /**
     * Constructor for {@code DisconnectPacket} class
     *
     * @param reason reason why the connection wishes to disconnect
     */
    2 usages  ↳ Ben Anderson
    public DisconnectPacket(String reason) {
        //set `type` property in the superclass to `DisconnectPacket.class`
        super(DisconnectPacket.class);
        //assign instance property to constructor parameter
        this.reason = reason;
    }

}
```

Figure 213: Final *DisconnectPacket* class

```
package xyz.benanderson.scs.networking.packets;

import lombok.AccessLevel;
import lombok.Getter;
import xyz.benanderson.scs.networking.Packet;

import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;

/**
 * Packet send from a client to a server when attempting to authenticate.
 */
22 usages  Ben Anderson
public class LoginPacket extends Packet {

    /**
     * Username that the connection is attempting to authenticate with
     */
    1 usage
    @Getter(AccessLevel.PUBLIC)
    private final String username;

    /**
     * Password that the connection is attempting to authenticate with
     */
    1 usage
    @Getter(AccessLevel.PUBLIC)
    private final String hashedPassword;

    /**
     * Constructor for {@code LoginPacket} class
     *
     * @param username username of user to authenticate as
     * @param hashedPassword password to use when authenticating
     */
    2 usages  Ben Anderson
    private LoginPacket(String username, String hashedPassword) {
        //set `type` property in the superclass to `LoginPacket.class`
        super(LoginPacket.class);
        //assign object properties to constructor parameters
        this.username = username;
        this.hashedPassword = hashedPassword;
    }
}
```

Figure 214: Final LoginPacket class part 1

```
public static LoginPacket fromPlainTextPassword(String username, String plainTextPassword) {
    return new LoginPacket(username, hashPassword(plainTextPassword));
}

no usages  ↳ Ben Anderson
public static LoginPacket fromHashedPassword(String username, String hashedPassword) {
    return new LoginPacket(username, hashedPassword);
}

6 usages  ↳ Ben Anderson
public static String hashPassword(String plainTextPassword) {
    try{
        final MessageDigest digest = MessageDigest.getInstance(algorithm: "SHA-256");
        final byte[] hash = digest.digest(plainTextPassword.getBytes(StandardCharsets.UTF_8));
        final StringBuilder hexString = new StringBuilder();
        for (byte b : hash) {
            final String hex = Integer.toHexString(0xff & b);
            if (hex.length() == 1)
                hexString.append('0');
            hexString.append(hex);
        }
        return hexString.toString();
    } catch(Exception ex){
        throw new RuntimeException(ex);
    }
}
```

Figure 215: Final LoginPacket class part 2

```
package xyz.benanderson.scs.networking.packets;

import lombok.Getter;
import xyz.benanderson.scs.networking.Packet;

9 usages  ↳ Ben Anderson
public class InfoPacket extends Packet {

    1 usage
    @Getter
    private final String info;

    1 usage  ↳ Ben Anderson
    public InfoPacket(String info) {
        super(InfoPacket.class);
        this.info = info;
    }

}
```

Figure 216: Final InfoPacket class

```

package xyz.benanderson.scs.networking.packets;

import lombok.Getter;
import xyz.benanderson.scs.networking.Packet;

import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

/**
 * Packet sent from a server to a client. This packet contains an image
 * which is the media frame/image of the security camera at a point in time.
 */
5 usages  ↗ Ben Anderson
public class MediaPacket extends Packet {

    /**
     * Media frame/image that makes up the main content packet
     */
    3 usages
    @Getter
    private transient BufferedImage mediaFrame;

    /**
     * Constructor for {@code MediaPacket} class
     *
     * @param mediaFrame BufferedImage representing the media frame/image
     */
    1 usage  ↗ Ben Anderson
    public MediaPacket(BufferedImage mediaFrame) {
        //set `type` property in the superclass to `MediaPacket.class`
        super(MediaPacket.class);
        //assign instance property to constructor parameter
        this.mediaFrame = mediaFrame;
    }

    no usages  ↗ Ben Anderson
    private void writeObject(ObjectOutputStream out) throws IOException {
        out.defaultWriteObject();
        ImageIO.write(mediaFrame, "jpg", out);
    }

    no usages  ↗ Ben Anderson
    private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
        in.defaultReadObject();
        mediaFrame = ImageIO.read(in);
    }
}

```

Figure 217: Final MediaPacket class

```
package xyz.benanderson.scs.networking;

import lombok.AllArgsConstructor;
import lombok.Getter;

import java.io.Serializable;

5 inheritors  Ben Anderson
@AllArgsConstructor
public abstract class Packet implements Serializable {

    no usages
    @Getter
    private final Class<? extends Packet> type;

}
```

Figure 218: Final Packet class

```
package xyz.benanderson.scs.networking;

6 usages  ↳ Ben Anderson
public class Validation {

    1 usage  ↳ Ben Anderson
    public static int parsePort(String portInputStr) throws ValidationException {
        try {
            int port = Integer.parseInt(portInputStr);
            return parsePort(port);
        } catch (NumberFormatException numberFormatException) {
            throw new ValidationException("Port number must be an integer");
        }
    }

    2 usages  ↳ Ben Anderson
    public static int parsePort(int portInput) throws ValidationException {
        if (portInput < 0 || portInput > 65535) {
            throw new ValidationException("Port number must be between 1 and 65535 (inclusive)");
        }
        return portInput;
    }

    6 usages  ↳ Ben Anderson
    public static class ValidationException extends Exception {
        2 usages  ↳ Ben Anderson
        public ValidationException(String s) { super(s); }
    }
}
```

Figure 219: Final Validation class

```
package xyz.benanderson.scs.networking.packets;

import lombok.AccessLevel;
import lombok.Getter;
import xyz.benanderson.scs.networking.Packet;

//test packet type only used in testing to confirm
//data is correctly transmitted and received
38 usages  ↳ Ben Anderson
public class TestPacket extends Packet {

    1 usage
    @Getter(AccessLevel.PUBLIC)
    private final int testData;

    10 usages  ↳ Ben Anderson
    public TestPacket(int testData) {
        super(TestPacket.class);
        this.testData = testData;
    }
}
```

Figure 220: Final TestPacket class

```

//method runs after each test method in this class
no usages  ▲ Ben Anderson
@AfterEach
void destroyPacketController() {
    try {
        localConnection.getSocket().close();
        peerConnection.getSocket().close();
    } catch (IOException ignored) {}
}

no usages  ▲ Ben Anderson
@Test
void testWritePacket() throws IOException, ClassNotFoundException {
    //instantiated to stop the PacketController from hanging
    ObjectOutputStream peerOutputStream = new ObjectOutputStream(peerConnection.getSocket().getOutputStream());
    PacketController localPacketController = new PacketController(localConnection);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    localPacketController.writePacketToSocket(testPacket);

    Packet receivedPacket;
    try (ObjectInputStream peerInputStream = new ObjectInputStream(peerConnection.getSocket().getInputStream())) {
        receivedPacket = (Packet) peerInputStream.readObject();
    }
    peerOutputStream.close();
    localPacketController.close();

    assertNotNull(receivedPacket);
    assertEquals(testPacket.getType(), receivedPacket.getType());
    assertEquals(testPacket.getTestData(), ((TestPacket) receivedPacket).getTestData());
}

no usages  ▲ Ben Anderson
@Test
void testReadPacket() throws IOException, ClassNotFoundException {
    //instantiated to stop the PacketController from hanging
    ObjectOutputStream peerOutputStream = new ObjectOutputStream(peerConnection.getSocket().getOutputStream());
    PacketController localPacketController = new PacketController(localConnection);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    peerOutputStream.writeObject(testPacket);

    Packet receivedPacket = localPacketController.readPacketFromSocket();
    peerOutputStream.close();
    localPacketController.close();

    assertEquals(testPacket.getType(), receivedPacket.getType());
    assertEquals(testPacket.getTestData(), ((TestPacket) receivedPacket).getTestData());
}
}

```

Figure 222: Final PacketControllerTest class part 2

```
package xyz.benanderson.scs.networking.connection;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import xyz.benanderson.scs.networking.Packet;
import xyz.benanderson.scs.networking.packets.TestPacket;

import java.io.IOException;
import java.io.ObjectOutputStream;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Random;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.function.Consumer;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

no usages  ↳ Ben Anderson
public class PacketListenerTest {

    24 usages
    private PacketListener packetListener;
    12 usages
    private Connection connectionMock;

    no usages  ↳ Ben Anderson
    @BeforeEach
    void setupPacketListener() {
        PacketController packetControllerMock = mock(PacketController.class);
        connectionMock = mock(Connection.class);
        doReturn(packetControllerMock).when(connectionMock).getPacketController();
    }

    no usages  ↳ Ben Anderson
    @AfterEach
    void destroyPacketListener() {
        try {
            packetListener.close();
        } catch (Exception ignored) {}
    }
}
```

Figure 223: Final *PacketListenerTest* class part 1

```
@Test
void testNoCallbacks() throws NoSuchMethodException {
    //so that the listening thread doesn't start - it isn't relevant to this test
    doReturn(toBeReturned: false).when(connectionMock).isConnected();
    packetListener = new PacketListener(connectionMock);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    Method runCallbacksMethod = packetListener.getClass()
        .getDeclaredMethod(name: "runCallbacks", Packet.class);
    runCallbacksMethod.setAccessible(true);

    assertDoesNotThrow(() -> runCallbacksMethod.invoke(packetListener, testPacket));
}

no usages  Ben Anderson
@Test
void testSingleCallbackRun() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException {
    //so that the listening thread doesn't start - it isn't relevant to this test
    doReturn(toBeReturned: false).when(connectionMock).isConnected();
    packetListener = new PacketListener(connectionMock);

    AtomicBoolean callbackRan = new AtomicBoolean(initialValue: false);
    Consumer<TestPacket> callback = testPacket -> callbackRan.set(true);
    packetListener.addCallback(TestPacket.class, callback);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    Method runCallbacksMethod = packetListener.getClass()
        .getDeclaredMethod(name: "runCallbacks", Packet.class);
    runCallbacksMethod.setAccessible(true);
    runCallbacksMethod.invoke(packetListener, testPacket);

    assertTrue(callbackRan.get());
}
```

Figure 224: Final PacketListenerTest class part 2

```

@Test
void testSingleCallbackNoRun() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException {
    //so that the listening thread doesn't start - it isn't relevant to this test
    doReturn(false).when(connectionMock).isConnected();
    packetListener = new PacketListener(connectionMock);

    AtomicBoolean callbackRan = new AtomicBoolean(initialValue: false);
    Consumer<Packet> callback = testPacket -> callbackRan.set(true);
    packetListener.addCallback(Packet.class, callback);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    Method runCallbacksMethod = packetListener.getClass()
        .getDeclaredMethod( name: "runCallbacks", Packet.class );
    runCallbacksMethod.setAccessible(true);
    runCallbacksMethod.invoke(packetListener, testPacket);

    assertFalse(callbackRan.get());
}

no usages  Ben Anderson
@Test
void testMultipleCallbacksRunOnlyOne() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException {
    //so that the listening thread doesn't start - it isn't relevant to this test
    doReturn(false).when(connectionMock).isConnected();
    packetListener = new PacketListener(connectionMock);

    AtomicBoolean callbackOneRan = new AtomicBoolean(initialValue: false);
    Consumer<TestPacket> callbackOne = testPacket -> callbackOneRan.set(true);
    AtomicBoolean callbackTwoRan = new AtomicBoolean(initialValue: false);
    Consumer<Packet> callbackTwo = testPacket -> callbackTwoRan.set(true);
    packetListener.addCallback(TestPacket.class, callbackOne);
    packetListener.addCallback(Packet.class, callbackTwo);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    Method runCallbacksMethod = packetListener.getClass()
        .getDeclaredMethod( name: "runCallbacks", Packet.class );
    runCallbacksMethod.setAccessible(true);
    runCallbacksMethod.invoke(packetListener, testPacket);

    assertTrue(callbackOneRan.get());
    assertFalse(callbackTwoRan.get());
}

```

Figure 225: Final *PacketListenerTest* class part 3

```

@Test
void testMultipleCallbacksRunBoth() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException {
    //so that the listening thread doesn't start - it isn't relevant to this test
    doReturn(false).when(connectionMock).isConnected();
    packetListener = new PacketListener(connectionMock);

    AtomicBoolean callbackOneRan = new AtomicBoolean(initialValue: false);
    Consumer<TestPacket> callbackOne = testPacket -> callbackOneRan.set(true);
    AtomicBoolean callbackTwoRan = new AtomicBoolean(initialValue: false);
    Consumer<TestPacket> callbackTwo = testPacket -> callbackTwoRan.set(true);
    packetListener.addCallback(TestPacket.class, callbackOne);
    packetListener.addCallback(TestPacket.class, callbackTwo);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    Method runCallbacksMethod = packetListener.getClass()
        .getDeclaredMethod(name: "runCallbacks", Packet.class);
    runCallbacksMethod.setAccessible(true);
    runCallbacksMethod.invoke(packetListener, testPacket);

    assertTrue(callbackOneRan.get());
    assertTrue(callbackTwoRan.get());
}

no usages ↗ Ben Anderson
@Test
void testPacketListening() throws IOException, ExecutionException, InterruptedException {
    Connection localConnectionMock = mock(Connection.class);
    Socket localSocket, peerSocket;
    //create server on randomly assigned available port
    try (ServerSocket embeddedServer = new ServerSocket(port: 0)) {
        //create socket connections from both sides
        localSocket = new Socket(embeddedServer.getInetAddress(), embeddedServer.getLocalPort());
        peerSocket = embeddedServer.accept();
    }
    doReturn(localSocket).when(localConnectionMock).getSocket();
    doCallRealMethod().when(localConnectionMock).isConnected();
    ObjectOutputStream peerObjectOutputStream = new ObjectOutputStream(peerSocket.getOutputStream());
    PacketController localPacketController = new PacketController(localConnectionMock);
    doReturn(localPacketController).when(localConnectionMock).getPacketController();
    packetListener = new PacketListener(localConnectionMock);

    CompletableFuture<String> completableFuture = new CompletableFuture<>();
    Consumer<TestPacket> callback = testPacket -> {
        //deactivate the listening thread
        completableFuture.complete(value: "Callback Ran Successfully");
        try {
            localSocket.close();
            peerSocket.close();
        } catch (IOException ignored) {}
    };
    packetListener.addCallback(TestPacket.class, callback);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    peerObjectOutputStream.writeObject(testPacket);

    assertEquals(expected: "Callback Ran Successfully", completableFuture.get());
}
}

```

Figure 226: Final *PacketListenerTest* class part 4

```
package xyz.benanderson.scs.networking.connection;

import org.junit.jupiter.api.Test;
import xyz.benanderson.scs.networking.Packet;
import xyz.benanderson.scs.networking.packets.TestPacket;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Field;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Queue;
import java.util.Random;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.mockito.Mockito.*;

no usages  ↳ Ben Anderson
public class PacketSenderTest {

    no usages  ↳ Ben Anderson
    @Test
    void testAddPacketToQueue() throws NoSuchFieldException, IllegalAccessException {
        PacketController packetControllerMock = mock(PacketController.class);
        Connection connectionMock = mock(Connection.class);
        doReturn(packetControllerMock).when(connectionMock).getPacketController();
        //so that the listening thread doesn't start - it isn't relevant to this test
        doReturn(toBeReturned: false).when(connectionMock).isConnected();
        PacketSender packetSender = new PacketSender(connectionMock);

        int testData = new Random().nextInt();
        TestPacket testPacket = new TestPacket(testData);
        Field packetQueueField = packetSender.getClass()
            .getDeclaredField(name: "packetQueue");
        packetQueueField.setAccessible(true);

        Queue<Packet> packetQueue = (Queue<Packet>) packetQueueField.get(packetSender);
        assertEquals(expected: 0, packetQueue.size());
        packetSender.sendPacket(testPacket);
        assertEquals(expected: 1, packetQueue.size());
        assertEquals(testPacket, packetQueue.peek());

        try {
            packetSender.close();
        } catch (InterruptedException ignored) {}
    }
}
```

Figure 227: Final PacketSenderTest class part 1

```
@Test
void testPacketSending() throws IOException, ClassNotFoundException {
    Connection localConnectionMock = mock(Connection.class);
    Socket localSocket, peerSocket;
    //create server on randomly assigned available port
    try (ServerSocket embeddedServer = new ServerSocket( port: 0)) {
        //create socket connections from both sides
        localSocket = new Socket(embeddedServer.getInetAddress(), embeddedServer.getLocalPort());
        peerSocket = embeddedServer.accept();
    }
    doReturn(localSocket).when(localConnectionMock).getSocket();
    doCallRealMethod().when(localConnectionMock).isConnected();
    new ObjectOutputStream(peerSocket.getOutputStream());
    PacketController localPacketController = new PacketController(localConnectionMock);
    ObjectInputStream peerObjectInputStream = new ObjectInputStream(peerSocket.getInputStream());
    doReturn(localPacketController).when(localConnectionMock).getPacketController();
    PacketSender packetSender = new PacketSender(localConnectionMock);

    int testData = new Random().nextInt();
    TestPacket testPacket = new TestPacket(testData);
    packetSender.sendPacket(testPacket);

    TestPacket receivedPacket = (TestPacket) peerObjectInputStream.readObject();
    try {
        localSocket.close();
        peerSocket.close();
    } catch (IOException ignored) {}

    assertNotNull(receivedPacket);
    assertEquals(testPacket.getTestData(), receivedPacket.getTestData());
}
}
```

Figure 228: Final *PacketSenderTest* class part 2

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <artifactId>scs_parent</artifactId>
        <groupId>xyz.benanderson</groupId>
        <version>0.0.2</version>
    </parent>

    <artifactId>scs_networking</artifactId>
    <version>0.0.4</version>

    <properties>
        <maven.compiler.source>16</maven.compiler.source>
        <maven.compiler.target>16</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
</project>
```

Figure 229: Final Networking pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <artifactId>scs_parent</artifactId>
        <groupId>xyz.benanderson</groupId>
        <version>0.0.2</version>
    </parent>

    <artifactId>scs_client</artifactId>
    <version>0.1.0</version>

    <properties>
        <maven.compiler.source>16</maven.compiler.source>
        <maven.compiler.target>16</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>xyz.benanderson</groupId>
            <artifactId>scs_networking</artifactId>
            <version>0.0.4</version>
        </dependency>
        <dependency>
            <groupId>com.intellij</groupId>
            <artifactId>forms_rt</artifactId>
            <version>7.0.3</version>
        </dependency>
    </dependencies>
```

Figure 230: Final Client pom.xml part 1

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.4</version>
            <configuration>
                <archive>
                    <manifest>
                        <addClasspath>true</addClasspath>
                        <mainClass>xyz.benanderson.scs.client.Main</mainClass>
                    </manifest>
                </archive>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.2.4</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <createDependencyReducedPom>false</createDependencyReducedPom>
                        <filters>
                            <filter>
                                <includes>
                                    <include>xyz.benanderson.scs.networking.*</include>
                                </includes>
                            </filter>
                        </filters>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

</project>
```

Figure 231: Final Client pom.xml part 2

```
package xyz.benanderson.scs.client;

import xyz.benanderson.scs.networking.connection.Connection;
import xyz.benanderson.scs.networking.packets.DisconnectPacket;
import xyz.benanderson.scs.networking.packets.InfoPacket;
import xyz.benanderson.scs.networking.packets.LoginPacket;
import xyz.benanderson.scs.networking.packets.MediaPacket;

import javax.swing.*;
import java.awt.*;
import java.io.IOException;
import java.net.Socket;

2 usages  ↲ Ben Anderson
public class Main {

    8 usages
    private static ClientGUI clientGUI;
    8 usages
    private static Connection connection;

    no usages  ↲ Ben Anderson
    public static void main(String[] args) throws Exception {
        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());

        JFrame frame = new JFrame();
        frame.setVisible(true);
        frame.setTitle("Security Camera Client");
        frame.setSize(width: 1280, height: 800);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLocationRelativeTo(null);

        clientGUI = new ClientGUI();
        clientGUI.getConnectButton().setFont(new Font(Font.MONOSPACED, Font.BOLD, size: 20));
        clientGUI.getConnectButton().setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
        clientGUI.getConnectButton().addActionListener(event -> new ConnectionDialog());

        frame.setContentPane(clientGUI.getContentPane());
    }
}
```

Figure 232: Final Client Main class part 1

```
public static void attemptConnect(String address, int port, String username, String password) throws IOException {
    if (connection != null && connection.isConnected()) {
        try {
            connection.close();
        } catch (Exception ignored) {}
    }

    Socket socket = new Socket(address, port);
    connection = new Connection(socket);
    connection.getPacketListener().addCallback(MediaPacket.class, mediaPacket -> {
        EventQueue.invokeLater(() -> clientGUI.getVideoComponent()
            .setIcon(new ImageIcon(mediaPacket.getMediaFrame())));
    });
    connection.getPacketListener().addCallback(InfoPacket.class, infoPacket -> {
        EventQueue.invokeLater(() -> JOptionPane.showMessageDialog(clientGUI.getContentPane(), infoPacket.getInfo(),
            title: "Information", JOptionPane.INFORMATION_MESSAGE));
    });
    connection.getPacketListener().addCallback(DisconnectPacket.class, disconnectPacket -> {
        EventQueue.invokeLater(() -> JOptionPane.showMessageDialog(clientGUI.getContentPane(),
            message: "Disconnected From Camera." + System.lineSeparator() + "Reason: " + disconnectPacket.getReason(),
            title: "Disconnected", JOptionPane.WARNING_MESSAGE));
    });
    LoginPacket loginPacket = LoginPacket.fromPlainTextPassword(username, password);
    connection.getPacketSender().sendPacket(loginPacket);
}
}
```

Figure 233: Final Client Main class part 2

```
package xyz.benanderson.scs.client;

import com.intellij.uiDesigner.core.GridConstraints;
import com.intellij.uiDesigner.core.GridLayoutManager;
import lombok.Getter;

import javax.swing.*;
import java.awt.*;

3 usages  Ben Anderson
public class ClientGUI {

    6 usages
    @Getter
    private JPanel contentPane;
    5 usages
    @Getter
    private JPanel cameraPane;
    6 usages
    @Getter
    private JButton connectButton;
    4 usages
    @Getter
    private JLabel videoComponent;

    {
        $$$setupUI$$$();
    }

    /**
     * @noinspection ALL
     */
}
```

Figure 234: Final ClientGUI class part 1

```
private void $$$setupUI$$$() {
    contentPane = new JPanel();
    contentPane.setLayout(new BorderLayout( hgap: 0, vgap: 0));
    final JPanel panel1 = new JPanel();
    panel1.setLayout(new GridBagLayout());
    contentPane.add(panel1, BorderLayout.SOUTH);
    final JPanel spacer1 = new JPanel();
    GridBagConstraints gbc;
    gbc = new GridBagConstraints();
    gbc.gridx = 2;
    gbc.gridy = 2;
    gbc.fill = GridBagConstraints.VERTICAL;
    panel1.add(spacer1, gbc);
    connectButton = new JButton();
    connectButton.setFocusPainted(false);
    connectButton.setPreferredSize(new Dimension( width: 275, height: 50));
    connectButton.setText("Connect To Camera");
    gbc = new GridBagConstraints();
    gbc.gridx = 1;
    gbc.gridy = 1;
    gbc.weighty = 1.0;
    panel1.add(connectButton, gbc);
    final JPanel spacer2 = new JPanel();
    gbc = new GridBagConstraints();
    gbc.gridx = 0;
    gbc.gridy = 1;
    gbc.fill = GridBagConstraints.HORIZONTAL;
    panel1.add(spacer2, gbc);
    final JPanel spacer3 = new JPanel();
```

Figure 235: Final ClientGUI class part 2

```
gbc = new GridBagConstraints();
gbc.gridx = 2;
gbc.gridy = 1;
gbc.weightx = 0.3;
gbc.fill = GridBagConstraints.HORIZONTAL;
panel1.add(spacer3, gbc);
final JPanel spacer4 = new JPanel();
gbc = new GridBagConstraints();
gbc.gridx = 2;
gbc.gridy = 0;
gbc.fill = GridBagConstraints.VERTICAL;
panel1.add(spacer4, gbc);
cameraPane = new JPanel();
cameraPane.setLayout(new GridLayoutManager( rowCount: 1, columnCount: 1,
    new Insets( top: 0, left: 0, bottom: 0, right: 0), hGap: -1, vGap: -1));
contentPane.add(cameraPane, BorderLayout.CENTER);
videoComponent = new JLabel();
videoComponent.setText("");
cameraPane.add(videoComponent, new GridConstraints( row: 0, column: 0, rowspan: 1,
    colSpan: 1, GridConstraints.ANCHOR_WEST, GridConstraints.FILL_NONE,
    GridConstraints.SIZEPOLICY_FIXED, GridConstraints.SIZEPOLICY_FIXED,
    minimumSize: null, preferredSize: null, maximumSize: null, indent: 0, useParentLayout: false));
}

/**
 * @noinspection ALL
 */
no usages  Ben Anderson
public JComponent $$$getRootComponent$$$() { return contentPane; }
}
```

Figure 236: Final ClientGUI class part 3

```
package xyz.benanderson.scs.client;

import com.intellij.uiDesigner.core.GridConstraints;
import com.intellij.uiDesigner.core.GridLayoutManager;
import com.intellij.uiDesigner.core.Spacer;
import lombok.Getter;
import xyz.benanderson.scs.networking.Validation;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.IOException;

2 usages ▲ Ben Anderson *
public class ConnectionDialog extends JDialog {
    8 usages
    private JPanel contentPane;
    6 usages
    private JButton buttonConnect;
    5 usages
    private JButton buttonCancel;
    3 usages
    @Getter
    private JTextField addressInput;
    3 usages
    @Getter
    private JTextField portInput;
    3 usages
    @Getter
    private JTextField usernameInput;
    3 usages
    @Getter
    private JPasswordField passwordInput;

    1 usage ▲ Ben Anderson *
    public ConnectionDialog() {
        setContentPane(contentPane);
        setModal(true);
        getRootPane().setDefaultButton(buttonConnect);
        buttonConnect.addActionListener(e -> onClickConnect());
        buttonCancel.addActionListener(e -> onClickCancel());

        // call onCancel() when cross is clicked
        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
        addWindowListener((WindowAdapter) windowClosing(e) -> { onClickCancel(); });

        // call onCancel() on ESCAPE
        contentPane.registerKeyboardAction(e -> onClickCancel(),
            KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, modifiers: 0),
            JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);

        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}
```

Figure 237: Final ConnectionDialog class part 1

```

private void onClickConnect() {
    int validatedPort;
    try {
        validatedPort = Validation.parsePort(getPortInput().getText());
    } catch (Validation.ValidationException e) {
        JOptionPane.showMessageDialog(parentComponent: this, e.getMessage(),
            title: "Input Validation Error", JOptionPane.ERROR_MESSAGE);
    }
    try {
        Main.attemptConnect(getAddressInput().getText(), validatedPort,
            getUsernameInput().getText(), new String(getPasswordInput().getPassword()));
    } catch (IOException e) {
        JOptionPane.showMessageDialog(parentComponent: this, e.getMessage(),
            title: "Connection Error", JOptionPane.ERROR_MESSAGE);
    }
    dispose();
}

3 usages  ↳ Ben Anderson
private void onClickCancel() { dispose(); }

{
    $$$setupUI$$$();
}

/**
 * @noinspection ALL
 */
1 usage  ↳ Ben Anderson *
private void $$$setupUI$$$() {
    contentPane = new JPanel();
    contentPane.setLayout(new GridLayoutManager( rowCount: 2, columnCount: 1,
        new Insets( top: 10, left: 10, bottom: 10, right: 10), hGap: -1, vGap: -1));
    final JPanel panel1 = new JPanel();
    panel1.setLayout(new GridLayoutManager( rowCount: 1, columnCount: 1,
        new Insets( top: 0, left: 0, bottom: 0, right: 0), hGap: -1, vGap: -1));
    contentPane.add(panel1, new GridConstraints( row: 1, column: 0, rowspan: 1, colSpan: 1,
        GridConstraints.ANCHOR_CENTER, GridConstraints.FILL_BOTH,
        HSizePolicy: GridConstraints.SIZEPOLICY_CAN_SHRINK |
            GridConstraints.SIZEPOLICY_CAN_GROW, VSizePolicy: 1,
        minimumSize: null, preferredSize: null, maximumSize: null, indent: 0, useParentLayout: false));
    final JPanel panel2 = new JPanel();
    panel2.setLayout(new GridLayoutManager( rowCount: 2, columnCount: 3,
        new Insets( top: 0, left: 0, bottom: 0, right: 0), hGap: -1, vGap: -1));
    panel1.add(panel2, new GridConstraints( row: 0, column: 0, rowspan: 1,
        colSpan: 1, GridConstraints.ANCHOR_CENTER, GridConstraints.FILL_BOTH,
        HSizePolicy: GridConstraints.SIZEPOLICY_CAN_SHRINK |
            GridConstraints.SIZEPOLICY_CAN_GROW,
        VSizePolicy: GridConstraints.SIZEPOLICY_CAN_SHRINK |
            GridConstraints.SIZEPOLICY_CAN_GROW, minimumSize: null,
        preferredSize: null, maximumSize: null, indent: 0, useParentLayout: false));
    buttonConnect = new JButton();
}

```

Figure 238: Final ConnectionDialog class part 2

```

buttonConnect = new JButton();
buttonConnect.setText("Connect");
panel2.add(buttonConnect, new GridConstraints( row: 1, column: 0,
    rowspan: 1, colSpan: 1, GridBagConstraints.ANCHOR_CENTER,
    GridBagConstraints.FILL_HORIZONTAL, HSizePolicy: GridBagConstraints.SIZEPOLICY_CAN_SHRINK |
    GridBagConstraints.SIZEPOLICY_CAN_GROW, GridBagConstraints.SIZEPOLICY_FIXED,
    minimumSize: null, preferredSize: null, maximumSize: null, indent: 0, useParentLayout: false));
buttonCancel = new JButton();
buttonCancel.setText("Cancel");
panel2.add(buttonCancel, new GridConstraints( row: 1, column: 2, rowspan: 1,
    colSpan: 1, GridBagConstraints.ANCHOR_CENTER, GridBagConstraints.FILL_HORIZONTAL,
    HSizePolicy: GridBagConstraints.SIZEPOLICY_CAN_SHRINK |
    GridBagConstraints.SIZEPOLICY_CAN_GROW, GridBagConstraints.SIZEPOLICY_FIXED,
    minimumSize: null, preferredSize: null, maximumSize: null, indent: 0, useParentLayout: false));
final Spacer spacer1 = new Spacer();
panel2.add(spacer1, new GridConstraints( row: 1, column: 1, rowspan: 1,
    colSpan: 1, GridBagConstraints.ANCHOR_CENTER, GridBagConstraints.FILL_HORIZONTAL,
    GridBagConstraints.SIZEPOLICY_WANT_GROW, VSizePolicy: 1, minimumSize: null,
    preferredSize: null, maximumSize: null, indent: 0, useParentLayout: false));
final Spacer spacer2 = new Spacer();
panel2.add(spacer2, new GridConstraints( row: 0, column: 1, rowspan: 1, colSpan: 1,
    GridBagConstraints.ANCHOR_CENTER, GridBagConstraints.FILL_VERTICAL, HSizePolicy: 1,
    GridBagConstraints.SIZEPOLICY_WANT_GROW, minimumSize: null, preferredSize: null,
    maximumSize: null, indent: 0, useParentLayout: false));
final JPanel panel3 = new JPanel();
panel3.setLayout(new GridLayoutManager( rowCount: 8, columnCount: 1,
    new Insets( top: 0, left: 0, bottom: 0, right: 0), hGap: -1, vGap: -1));
contentPane.add(panel3, new GridConstraints( row: 0, column: 0, rowspan: 1,
    colSpan: 1, GridBagConstraints.ANCHOR_CENTER, GridBagConstraints.FILL_BOTH,
    HSizePolicy: GridBagConstraints.SIZEPOLICY_CAN_SHRINK |
    GridBagConstraints.SIZEPOLICY_CAN_GROW,
    VSizePolicy: GridBagConstraints.SIZEPOLICY_CAN_SHRINK |
    GridBagConstraints.SIZEPOLICY_CAN_GROW, minimumSize: null,
    preferredSize: null, maximumSize: null, indent: 0, useParentLayout: false));
addressInput = new JTextField();
panel3.add(addressInput, new GridConstraints( row: 1, column: 0, rowspan: 1,
    colSpan: 1, GridBagConstraints.ANCHOR_WEST, GridBagConstraints.FILL_HORIZONTAL,
    GridBagConstraints.SIZEPOLICY_WANT_GROW, GridBagConstraints.SIZEPOLICY_FIXED,
    minimumSize: null, new Dimension( width: 150, height: -1), maximumSize: null,
    indent: 0, useParentLayout: false));
final JLabel label1 = new JLabel();
label1.setText("Camera Address");
panel3.add(label1, new GridConstraints( row: 0, column: 0, rowspan: 1, colSpan: 1,
    GridBagConstraints.ANCHOR_CENTER, GridBagConstraints.FILL_NONE,
    GridBagConstraints.SIZEPOLICY_FIXED, GridBagConstraints.SIZEPOLICY_FIXED,
    minimumSize: null, preferredSize: null, maximumSize: null, indent: 0, useParentLayout: false));
final JLabel label2 = new JLabel();

```

Figure 239: Final ConnectionDialog class part 3

```

final JLabel label2 = new JLabel();
label2.setText("Camera Port");
panel3.add(label2, new GridConstraints( row: 2, column: 0, rowspan: 1,
    colSpan: 1, GridBagConstraints.ANCHOR_CENTER, GridBagConstraints.FILL_NONE,
    GridBagConstraints.SIZEPOLICY_FIXED, GridBagConstraints.SIZEPOLICY_FIXED,
    minimumSize: null, preferredSize: null, maximumSize: null, indent: 0, useParentLayout: false));
portInput = new JTextField();
panel3.add(portInput, new GridConstraints( row: 3, column: 0, rowspan: 1,
    colSpan: 1, GridBagConstraints.ANCHOR_WEST, GridBagConstraints.FILL_HORIZONTAL,
    GridBagConstraints.SIZEPOLICY_WANT_GROW, GridBagConstraints.SIZEPOLICY_FIXED,
    minimumSize: null, new Dimension( width: 150, height: -1), maximumSize: null,
    indent: 0, useParentLayout: false));
final JLabel label3 = new JLabel();
label3.setText("Username");
panel3.add(label3, new GridConstraints( row: 4, column: 0, rowspan: 1,
    colSpan: 1, GridBagConstraints.ANCHOR_CENTER, GridBagConstraints.FILL_NONE,
    GridBagConstraints.SIZEPOLICY_FIXED, GridBagConstraints.SIZEPOLICY_FIXED,
    minimumSize: null, preferredSize: null, maximumSize: null, indent: 0, useParentLayout: false));
usernameInput = new JTextField();
panel3.add(usernameInput, new GridConstraints( row: 5, column: 0, rowspan: 1,
    colSpan: 1, GridBagConstraints.ANCHOR_WEST, GridBagConstraints.FILL_HORIZONTAL,
    GridBagConstraints.SIZEPOLICY_WANT_GROW, GridBagConstraints.SIZEPOLICY_FIXED,
    minimumSize: null, new Dimension( width: 150, height: -1), maximumSize: null,
    indent: 0, useParentLayout: false));
final JLabel label4 = new JLabel();
label4.setText("Password");
panel3.add(label4, new GridConstraints( row: 6, column: 0, rowspan: 1, colSpan: 1,
    GridBagConstraints.ANCHOR_CENTER, GridBagConstraints.FILL_NONE,
    GridBagConstraints.SIZEPOLICY_FIXED, GridBagConstraints.SIZEPOLICY_FIXED,
    minimumSize: null, preferredSize: null, maximumSize: null, indent: 0, useParentLayout: false));
passwordInput = new JPasswordField();
panel3.add(passwordInput, new GridConstraints( row: 7, column: 0, rowspan: 1,
    colSpan: 1, GridBagConstraints.ANCHOR_WEST, GridBagConstraints.FILL_HORIZONTAL,
    GridBagConstraints.SIZEPOLICY_WANT_GROW, GridBagConstraints.SIZEPOLICY_FIXED,
    minimumSize: null, new Dimension( width: 150, height: -1), maximumSize: null,
    indent: 0, useParentLayout: false));
}

/**
 * @noinspection ALL
 */
no usages Ben Anderson
public JComponent $$$getRootComponent$$$() { return contentPane; }
}

```

Figure 240: Final ConnectionDialog class part 4

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>xyz.benanderson</groupId>
    <artifactId>scs_parent</artifactId>
    <packaging>pom</packaging>
    <version>0.0.2</version>

    <modules>
        <module>Networking</module>
        <module>Server</module>
        <module>Client</module>
    </modules>

    <properties>
        <maven.compiler.source>16</maven.compiler.source>
        <maven.compiler.target>16</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.24</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.9.0</version>
            <scope>test</scope>
        </dependency>
        <dependency>
```

Figure 241: Final Parent pom.xml part 1

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>4.10.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.jetbrains</groupId>
    <artifactId>annotations</artifactId>
    <version>23.1.0</version>
    <scope>provided</scope>
</dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.22.2</version>
            <executions>
                <execution>
                    <phase>test</phase>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

</project>
```

Figure 242: Final Parent pom.xml part 2

```
package xyz.benanderson.scs.server.account.managers;

import xyz.benanderson.scs.server.account.User;
import xyz.benanderson.scs.server.account.UserManager;
import xyz.benanderson.scs.server.configuration.ConfigurationWrapper;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
import java.util.Optional;

4 usages  ↳ Ben Anderson
public class MultiFileUserManager implements UserManager {

    4 usages  ↳ Ben Anderson
    @Override
    public Optional<User> getUser(String username) {
        Path userSaveFile = ConfigurationWrapper.getInstance().getUsersSaveDirectory()
            .resolve(username.toLowerCase());
        if (!Files.exists(userSaveFile)) {
            return Optional.empty();
        }
        try {
            List<String> lines = Files.readAllLines(userSaveFile);
            User parsedUser = User.fromHashedPassword(lines.get(0),
                lines.get(1),
                Boolean.parseBoolean(lines.get(2)));
            return Optional.of(parsedUser);
        } catch (Exception e) {
            return Optional.empty();
        }
    }
}
```

Figure 243: Final MultiFileUserManager class part 1

```
@Override
public void createUser(User user) throws IOException {
    //create directory if it does not exist
    Path usersSaveDirectory = ConfigurationWrapper.getInstance().getUsersSaveDirectory();
    Files.createDirectories(usersSaveDirectory);
    //creates and writes user data to file (automatically closes it)
    Path userSaveFile = usersSaveDirectory.resolve(user.getUsername().toLowerCase());
    Files.writeString(userSaveFile, csq: user.getUsername() + System.lineSeparator()
        + user.getHashedPassword() + System.lineSeparator()
        + user.isAdmin());
}

2 usages  Ben Anderson
@Override
public void deleteUser(String username) throws IOException {
    Path userSaveFile = ConfigurationWrapper.getInstance().getUsersSaveDirectory()
        .resolve(username.toLowerCase());
    Files.deleteIfExists(userSaveFile);
}

}
```

Figure 244: Final MultiFileUserManager class part 2

```
package xyz.benanderson.scs.server.account;

import lombok.Data;
import lombok.EqualsAndHashCode;
import xyz.benanderson.scs.networking.packets.LoginPacket;

26 usages  ↳ Ben Anderson
@Data
@EqualsAndHashCode
public class User {

    1 usage
    private final String username, hashedPassword;
    1 usage
    private final boolean admin;

    2 usages  ↳ Ben Anderson
    public static User fromHashedPassword(String username, String hashedPassword, boolean admin) {
        return new User(username, hashedPassword, admin);
    }

    3 usages  ↳ Ben Anderson
    public static User fromPlainTextPassword(String username, String plainTextPassword, boolean admin) {
        return new User(username, LoginPacket.hashPassword(plainTextPassword), admin);
    }

    2 usages  ↳ Ben Anderson
    private User(String username, String hashedPassword, boolean admin) {
        this.username = username;
        this.hashedPassword = hashedPassword;
        this.admin = admin;
    }
}
```

Figure 245: Final User class

```
package xyz.benanderson.scs.server.account;

import java.util.Optional;

10 usages 1 implementation Ben Anderson
public interface UserManager {

    4 usages 1 implementation Ben Anderson
    Optional<User> getUser(String username);

    3 usages 1 implementation Ben Anderson
    void createUser(User user) throws Exception;

    2 usages 1 implementation Ben Anderson
    void deleteUser(String username) throws Exception;
}
```

Figure 246: Final UserManager interface

```
package xyz.benanderson.scs.server.configuration;

import lombok.Getter;

import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.NoSuchElementException;
import java.util.Objects;
import java.util.Optional;
import java.util.Properties;

5 usages  ↳ Ben Anderson
@Getter
public class Configuration {

    3 usages
    private final Properties properties;
    3 usages
    private final String configFileName = "server.properties";
    5 usages
    private final Path configFile;

    1 usage  ↳ Ben Anderson
    public Configuration(Path configFolder) {
        //get path to config file
        configFile = configFolder.resolve(configFileName);
        //load internal default config
        Properties internalProperties = new Properties();
        loadInternalProperties(internalProperties);
        //create properties with defaults
        this.properties = new Properties(internalProperties);
        //load properties from file if it exists, otherwise create file with defaults
        if (createFileIfNotExists()) {
            loadFromFile();
        }
    }
}
```

Figure 247: Final Configuration class part 1

```
//loads properties into the given properties object from the
//included default config file embedded in the application
1 usage  ± Ben Anderson
private void loadInternalProperties(Properties internalProperties) {
    try {
        internalProperties.load(getClass().getClassLoader().getResourceAsStream(configFileName));
    } catch (IOException e) {
        System.err.println("[WARNING] Failed to read internal configuration file.");
        e.printStackTrace();
    }
}

//loads properties from the configFile into the properties attribute of this object
1 usage  ± Ben Anderson
private void loadFromFile() {
    try (InputStream inputStream = Files.newInputStream(configFile)) {
        this.properties.load(inputStream);
    } catch (IOException e) {
        System.err.println("[WARNING] Failed to read external configuration file - resorting to internal configuration.");
        e.printStackTrace();
    }
}

//copies the default config file that is embedded in the application to the location where the config file is stored
//on disk. this only occurs if the file does not exist on disk already.
1 usage  ± Ben Anderson
private boolean createFileIfNotExists() {
    if (!Files.exists(configFile)) {
        try (InputStream inputStream = this.getClass().getClassLoader().getResourceAsStream(configFileName)) {
            Files.copy(Objects.requireNonNull(inputStream), configFile);
        } catch (Exception e) {
            System.err.println("[WARNING] Failed to write default config to configuration file '" + configFile + "'.");
            e.printStackTrace();
            return false;
        }
    }
    return true;
}
```

Figure 248: Final Configuration class part 2

```

/**
 * Get an {@code Optional} denoting a {@code String} value from the config.
 *
 * @param key configuration entry key
 * @return configuration entry value as an {@code Optional}
 */
7 usages  ▲ Ben Anderson
public Optional<String> getString(String key) {
    //first check system environment variables to allow for dynamic entry inclusion
    String env = System.getenv(key);
    //if key was not a system environment variable, return the Optional wrapped result from the properties table
    if (env == null || env.strip().length() == 0)
        return Optional.ofNullable(this.properties.getProperty(key));
    return Optional.of(env);
}

/**
 * Convenience method which throws a {@code NoSuchElementException} if the {@code Optional} returned by
 * {@link Configuration#getString(String)} is empty.
 *
 * @param key configuration entry key
 * @return configuration entry value (not null)
 */
4 usages  ▲ Ben Anderson
public String getRequiredString(String key) throws NoSuchElementException {
    return getString(key).orElseThrow(() -> new NoSuchElementException("'" + key + "' cannot be empty"));
}

/**
 * Get an {@code Optional} denoting an {@code Integer} value from the config.
 *
 * @param key configuration entry key
 * @return configuration entry value as an {@code Optional}
 */
5 usages  ▲ Ben Anderson
public Optional<Integer> getInt(String key) {
    try {
        return getString(key).map(Integer::parseInt);
    } catch (NumberFormatException e) {
        return Optional.empty();
    }
}

/**
 * Convenience method which throws a {@code NoSuchElementException} if the {@code Optional} returned by
 * {@link Configuration#getInt(String)} is empty.
 *
 * @param key configuration entry key
 * @return configuration entry value (not null)
 */
no usages  ▲ Ben Anderson
public int getRequiredInt(String key) throws NoSuchElementException {
    return getInt(key).orElseThrow(() -> new NoSuchElementException("'" + key + "' must be an integer"));
}

```

Figure 249: Final Configuration class part 3

```
/**  
 * Get an {@code Optional} denoting a {@code Double} value from the config.  
 *  
 * @param key configuration entry key  
 * @return configuration entry value as an {@code Optional}  
 */  
2 usages  ↗ Ben Anderson  
public Optional<Double> getDouble(String key) { return getString(key).map(Double::parseDouble); }  
  
/**  
 * Convenience method which throws a {@code NoSuchElementException} if the {@code Optional} returned by  
 * {@link Configuration#getDouble(String)} is empty.  
 *  
 * @param key configuration entry key  
 * @return configuration entry value (not null)  
 */  
no usages  ↗ Ben Anderson  
public double getRequiredDouble(String key) throws NoSuchElementException {  
    return getDouble(key).orElseThrow(() -> new NoSuchElementException("'" + key + "' must be a double"));  
}  
  
/**  
 * Get a {@code Boolean} value from the config, or return the provided default value if the key is not present  
 * in the config or the entry's value is not a boolean.  
 *  
 * @param key configuration entry key  
 * @return configuration entry value or the default value as a {@code boolean}  
 */  
1 usage  ↗ Ben Anderson  
public boolean getBoolean(String key, boolean defaultValue) {  
    return getString(key).map(Boolean::parseBoolean).orElse(defaultValue);  
}  
}
```

Figure 250: Final Configuration class part 4

```
package xyz.benanderson.scs.server.configuration;

import xyz.benanderson.scs.networking.Validation;
import xyz.benanderson.scs.networking.packets.LoginPacket;
import xyz.benanderson.scs.server.account.User;

import java.nio.file.Path;
import java.nio.file.Paths;
import java.time.Duration;
import java.time.temporal.ChronoUnit;
import java.time.temporal.TemporalUnit;

25 usages  ▲ Ben Anderson
public class ConfigurationWrapper {

    3 usages
    private static ConfigurationWrapper instance;
    11 usages
    private final Configuration configuration;

    //only access provided to object through public static getInstance() method
    18 usages  ▲ Ben Anderson
    public static ConfigurationWrapper getInstance() {
        if (instance == null) instance = new ConfigurationWrapper();
        return instance;
    }

    //private constructor to restrict access to public static getInstance() method
    1 usage  ▲ Ben Anderson
    private ConfigurationWrapper() {
        //System.getProperty("user.dir") returns current working directory
        //therefore creates configuration files in current working directory
        this.configuration = new Configuration(Paths.get(System.getProperty("user.dir")));
    }

    /**
     * @return Max allowed concurrent connections to the server
     */
    2 usages  ▲ Ben Anderson
    public int getMaxConnections() { return configuration.getInt(key: "server.max-connections").orElse(Integer.MAX_VALUE); }
}
```

Figure 251: Final ConfigurationWrapper class part 1

```

/**
 * @return TCP port for the server to run on, or 0 if no port was specified in the config
 */
2 usages ▲ Ben Anderson
public int getServerPort() {
    try {
        return Validation.parsePort(configuration.getInt(key: "server.port").orElse(-1));
    } catch (Validation.ValidationException e) {
        System.err.println("[WARNING] " + e.getMessage() + ". Resorting to random port number.");
        return 0;
    }
}

/**
 * @return TCP address for the server to run on, or 127.0.0.1 (localhost) if no address was specified in the config
 */
2 usages ▲ Ben Anderson
public String getServerAddress() { return configuration.getString(key: "server.address").orElse("127.0.0.1"); }

/**
 * @return {@code Path} denoting directory to save recorded videos to
 */
2 usages ▲ Ben Anderson
public Path getVideoSaveDirectory() { return Paths.get(configuration.getRequiredString(key: "video.save-directory")); }

6 usages ▲ Ben Anderson
public Path getUsersSaveDirectory() { return Paths.get(configuration.getRequiredString(key: "users.save-directory")); }

/**
 * @return {@code Duration} denoting preferred length of recorded videos
 */
2 usages ▲ Ben Anderson
public Duration getVideoDuration() {
    int durationNumber = configuration.getInt(key: "video.duration.number").orElse(60);
    TemporalUnit durationUnit = ChronoUnit.valueOf(configuration.getString(key: "video.duration.unit")
        .orElse("seconds").toUpperCase());
    return Duration.of(durationNumber, durationUnit);
}

2 usages ▲ Ben Anderson
public User getDefaultUser() {
    return User.fromPlainTextPassword(configuration.getRequiredString(key: "users.default.username"),
        LoginPacket.hashPassword(configuration.getRequiredString(key: "users.default.password")),
        configuration.getBoolean(key: "users.default.is_admin", defaultValue: false));
}

}

```

Figure 252: Final ConfigurationWrapper class part 2

```
package xyz.benanderson.scs.server.networking;

import xyz.benanderson.scs.networking.connection.Connection;
import xyz.benanderson.scs.server.configuration.ConfigurationWrapper;

import java.io.IOException;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Collections;
import java.util.SortedMap;
import java.util.TreeMap;
import java.util.UUID;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

22 usages ▲ Ben Anderson
public class Server implements AutoCloseable {

    //attributes of the server
    3 usages
    private final ServerSocket serverSocket;
    3 usages
    private final Thread receiveConnectionsThread;
    7 usages
    private final SortedMap<UUID, Connection> connections;
    3 usages
    private final AtomicBoolean running;
    2 usages
    private final BiConsumer<Connection, Server> clientConnectListener;
    2 usages
    private final BiConsumer<Connection, Server> clientDisconnectListener;
    2 usages
    private final Consumer<Server> serverShutdownListener;

    /**
     * Constructor with package-private (default) visibility to allow only the {@code ServerBuilder}
     * class to instantiate {@code Server} instances.
     *
     * @param serverBuilder builder to construct this {@code Server} from
     */
}
```

Figure 253: Final Server class part 1

```

Server(ServerBuilder serverBuilder) throws IOException {
    //create a synchronized map to avoid race conditions in a multithreaded environment
    //uses a TreeMap as a key-value store for connections, where the key is the
    //Connection's identifier and the value is the Connection object.
    //A TreeMap was used instead of a HashMap as O(1) retrieval is not essential as
    //operations against this map will not occur often.
    this.connections = Collections.synchronizedSortedMap(new TreeMap<>());
    //mark the server as running
    this.running = new AtomicBoolean( initialValue: true);
    //assign attributes from serverBuilder...
    int port = serverBuilder.getPort();
    InetSocketAddress bindAddress = serverBuilder.getBindAddress();
    this.clientConnectListener = serverBuilder.getClientConnectListener();
    //append code to remove the Connection from the Server's internal connections map
    //to the end of the client disconnect listener
    BiConsumer<Connection, Server> closeListener = (con, server) -> {
        try { connections.remove(con.getId()).close(); } catch (Exception ignored) {}
    };
    this.clientDisconnectListener = serverBuilder.getClientDisconnectListener().andThen(closeListener);
    this.serverShutdownListener = serverBuilder.getServerShutdownListener();

    this.serverSocket = new ServerSocket(port, backlog: 5, bindAddress);
    this.receiveConnectionsThread = new Thread(this::receiveConnections);
    this.receiveConnectionsThread.start();
}

5 usages  Ben Anderson
public int getPort() { return getServerSocket().getLocalPort(); }

5 usages  Ben Anderson
public ServerSocket getServerSocket() {
    synchronized (serverSocket) {
        return serverSocket;
    }
}

/**
 * @return boolean denoting if the {@code Server} is open and running.
 */
1 usage  Ben Anderson
public boolean isOpen() { return getServerSocket() != null && !getServerSocket().isClosed() && running.get(); }

```

Figure 254: Final Server class part 2

```
/**  
 * @return An unmodifiable SortedMap representing the {@code Connection}(s) to the server  
 */  
1 usage  Ben Anderson  
public SortedMap<UUID, Connection> getConnections() {  
    return Collections.unmodifiableSortedMap(connections);  
}  
  
/**  
 * Closes the {@code Server} and stops listening for new connections. The server shutdown  
 * listener will be triggered by the receive-connections thread. All connections will be  
 * closed and cleared from the {@code Server}.  
 */  
Ben Anderson  
@Override  
public void close() {  
    //attempt to stop listening for connections on the receive-connections thread  
    running.set(false);  
    try {  
        getServerSocket().close();  
    } catch (Exception ignored) {}  
    //wait for the receive-connections thread to terminate (also waits for the  
    //server shutdown listener to execute).  
    try {  
        receiveConnectionsThread.join();  
    } catch (InterruptedException ignored) {}  
    //closes all connections  
    connections.values().forEach(con -> {  
        try {  
            con.close();  
        } catch (Exception ignored) {}  
    });  
    //clear connections from memory of the server  
    connections.clear();  
}
```

Figure 255: Final Server class part 3

```
//private-visibility method ran only by the receive-connections thread
1 usage  • Ben Anderson
private void receiveConnections() {
    //condition-controlled loop to keep listening thread active as long as the
    //server is running and open
    while (isOpen()) {
        //don't accept new connections if already at max connections as specified in config
        if (connections.size() >= ConfigurationWrapper.getInstance().getMaxConnections())
            continue;
        try {
            //block until connection is accepted from the server's socket
            Socket socket = getServerSocket().accept();
            //instantiate Connection object (from connection framework) using
            //the connection's socket
            Connection connection = new Connection(socket);
            //add disconnect listener to the connection
            connection.setDisconnectListener(con -> clientDisconnectListener.accept(con, u: this));
            //add connection to server's map of connections
            connections.put(connection.getId(), connection);
            //run the client connection listener with the newly accepted connection
            clientConnectListener.accept(connection, u: this);
        } catch (IOException e) {
            //print error if an exception occurs
            e.printStackTrace();
        }
    }
    //run the server shutdown listener as isOpen() now returns false
    serverShutdownListener.accept(t: this);
}
}
```

Figure 256: Final Server class part 4

```
package xyz.benanderson.scs.server.networking;

import lombok.Getter;
import xyz.benanderson.scs.networking.connection.Connection;

import java.io.IOException;
import java.net.InetAddress;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

/**
 * The {@code ServerBuilder} class allows a developer to construct a {@code Server} instance
 * and configure it prior to instantiation. Use {@link ServerBuilder#build()} to build a {@code Server}
 * from this {@code ServerBuilder}.
 */
18 usages  Ben Anderson
@Getter
public class ServerBuilder {

    //attributes that will be used by the server during construction
    1 usage
    private final int port;
    1 usage
    private final InetAddress bindAddress;
    2 usages
    private BiConsumer<Connection, Server> clientConnectListener;
    2 usages
    private BiConsumer<Connection, Server> clientDisconnectListener;
    2 usages
    private Consumer<Server> serverShutdownListener;

    /**
     * @param port TCP port to run the networking server on
     * @param bindAddress TCP address to run the networking server on
     */
    5 usages  Ben Anderson
    public ServerBuilder(int port, InetAddress bindAddress) {
        this.port = port;
        this.bindAddress = bindAddress;
        this.clientConnectListener = (con, serv) -> {};
        this.clientDisconnectListener = (con, serv) -> {};
        this.serverShutdownListener = serv -> {};
    }
}
```

Figure 257: Final ServerBuilder class part 1

```
/**  
 * Runs after a client connects and the {@code Connection} is established  
 */  
2 usages  Ben Anderson  
public ServerBuilder onClientConnect(BiConsumer<Connection, Server> clientConnectListener) {  
    this.clientConnectListener = clientConnectListener;  
    return this;  
}  
  
/**  
 * Runs as the client disconnects, prior to the {@code Connection} closing  
 */  
2 usages  Ben Anderson  
public ServerBuilder onClientDisconnect(BiConsumer<Connection, Server> clientDisconnectListener) {  
    this.clientDisconnectListener = clientDisconnectListener;  
    return this;  
}  
  
/**  
 * Runs as the server shuts down, prior to the {@code Connection}s closing  
 */  
1 usage  Ben Anderson  
public ServerBuilder onServerShutdown(Consumer<Server> serverShutdownListener) {  
    this.serverShutdownListener = serverShutdownListener;  
    return this;  
}  
  
/**  
 * Construct a {@code Server} from this {@code ServerBuilder}  
 */  
6 usages  Ben Anderson  
public Server build() throws IOException {  
    return new Server(serverBuilder: this);  
}  
}
```

Figure 258: Final ServerBuilder class part 2

```
package xyz.benanderson.scs.server.video;

import com.github.sarxos.webcam.Webcam;

import java.awt.image.BufferedImage;
import java.util.Optional;

3 usages  ↳ Ben Anderson
public class CameraViewer implements AutoCloseable {

    4 usages
    private final Webcam camera;

    1 usage  ↳ Ben Anderson
    public CameraViewer(Webcam camera) {
        this.camera = camera;
        this.camera.open();
    }

    1 usage  ↳ Ben Anderson
    public Optional<BufferedImage> captureImage() {
        return Optional.ofNullable(camera.getImage());
    }

    ↳ Ben Anderson
    @Override
    public void close() {
        camera.close();
    }
}
```

Figure 259: Final CameraViewer class

```
package xyz.benanderson.scs.server.video;

import lombok.AllArgsConstructor;
import org.jcodec.api.awt.AWTSequenceEncoder;

import javax.imageio.*;
import javax.imageio.stream.FileImageInputStream;
import javax.imageio.stream.ImageOutputStream;
import javax.imageio.stream.MemoryCacheImageOutputStream;
import java.awt.image.BufferedImage;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Optional;

8 usages  Ben Anderson
@AllArgsConstructor
public class VideoEncoder {

    1 usage
    private final VideoFileManager videoFileManager;

    //file format of raw media save file is as follows:
    // - long (number of media frames in the file)
    // - long (start timestamp in milliseconds)
    // - long (end timestamp in milliseconds)
    // - list of serialized media frames
}
```

Figure 260: Final VideoEncoder class part 1

```
public void appendToStream(BufferedImage image, long currentTimeMillis) {  
    //output error if can't access a save file  
    Optional<Path> currentSaveFileOptional = videoFileManager.getCurrentSaveFile();  
    if (currentSaveFileOptional.isEmpty()) {  
        System.err.println("[ERROR] Failed to fetch video save file.");  
        return;  
    }  
    //open current save file as `RandomAccessFile`, therefore being able to jump around the file  
    Path currentSaveFile = currentSaveFileOptional.get();  
    try (RandomAccessFile randomAccessFile = new RandomAccessFile(currentSaveFile.toFile(), "rw")) {  
        //if file size is zero, it is new and needs some header metadata at the start of the file  
        if (Files.size(currentSaveFile) == 0L) {  
            randomAccessFile.writeLong(1L);  
            randomAccessFile.seek(8);  
            randomAccessFile.writeLong(currentTimeMillis);  
        } else {  
            //if file size is not zero, increase the number of media frames in the header  
            long currentNumberFrames = randomAccessFile.readLong();  
            randomAccessFile.seek(0);  
            randomAccessFile.writeLong(currentNumberFrames + 1);  
        }  
        //write the time of the last media frame  
        randomAccessFile.seek(16);  
        randomAccessFile.writeLong(currentTimeMillis);  
        randomAccessFile.seek(randomAccessFile.length());  
        //write compressed media frame  
        randomAccessFile.write(compressImage(image));  
    } catch (Exception e) {  
        //output error if one is encountered  
        System.err.println("[ERROR] An error occurred when writing timestamp metadata to a save file.");  
        e.printStackTrace();  
    }  
}
```

Figure 261: Final VideoEncoder class part 2

```

public void processRawMediaSave(Path rawMediaSaveFile) {
    //open file using `RandomAccessFile` to be able to skip around in the file
    try (RandomAccessFile randomAccessFile = new RandomAccessFile(rawMediaSaveFile.toFile(), "r");
        FileInputStream fileInputStream = new FileInputStream(randomAccessFile)) {
        //read header metadata
        long numberofFrames = randomAccessFile.readLong();
        long startTimestamp = randomAccessFile.readLong();
        long endTimestamp = randomAccessFile.readLong();
        //calculate fps (frames per second) using header metadata
        long videoDurationInMillis = endTimestamp - startTimestamp;
        if (videoDurationInMillis == 0) videoDurationInMillis = 1;
        int fps = (int) (numberofFrames / (videoDurationInMillis / 1000d));

        //setup image (media frame) reading
        fileInputStream.seek( pos: 24);
        ImageReader reader = ImageIO.getImageReadersBySuffix( fileSuffix: "jpg").next();
        reader.setInput(fileInputStream);

        //setup video encoding
        File videoOutputFile = new File(rawMediaSaveFile.toString().replace( target: ".crms", replacement: ".mp4"));
        AWTSequenceEncoder encoder = AWTSequenceEncoder.createSequenceEncoder(videoOutputFile, fps);
        //read frames from the raw media save file and encode them into the output file
        for (int frameNumber = 0; frameNumber < numberofFrames; frameNumber++) {
            BufferedImage image;
            try {
                image = reader.read(frameNumber);
            } catch (IndexOutOfBoundsException e) {
                break;
            }
            encoder.encodeImage(image);
        }
        encoder.finish();
        System.out.println("[INFO] Finished encoding video " + videoOutputFile.getName());
        //delete raw media save file after video constructed from it
        Files.delete(rawMediaSaveFile);
    } catch (Exception e) {
        //output error if encountered
        System.err.println("[ERROR] An error occurred when encoding a video from a raw media save file.");
        e.printStackTrace();
    }
}
}

```

Figure 262: Final VideoEncoder class part 3

```
public byte[] compressImage(BufferedImage bufferedImage) {  
    ByteArrayOutputStream compressed = new ByteArrayOutputStream();  
    try (ImageOutputStream outputStream = new MemoryCacheImageOutputStream(compressed)) {  
        ImageWriter jpgWriter = ImageIO.getImageWritersByFormatName("jpg").next();  
  
        // Configure JPEG compression: 20% quality  
        ImageWriteParam jpgWriteParam = jpgWriter.getDefaultWriteParam();  
        jpgWriteParam.setCompressionMode(ImageWriteParam.MODE_EXPLICIT);  
        jpgWriteParam.setCompressionQuality(0.2f);  
  
        jpgWriter.setOutput(outputStream);  
        jpgWriter.write(streamMetadata: null, new IIOMImage(bufferedImage, thumbnails: null, metadata: null), jpgWriteParam);  
        jpgWriter.dispose();  
    } catch (IOException e) {  
        System.err.println("[ERROR] An error occurred when compressing a media frame for a raw media save file.");  
        e.printStackTrace();  
    }  
    return compressed.toByteArray();  
}  
}
```

Figure 263: Final VideoEncoder class part 4

```
package xyz.benanderson.scs.server.video;

import lombok.Getter;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.Duration;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Optional;

9 usages  ↳ Ben Anderson
public class VideoFileManager {

    4 usages
    @Getter
    private final Path saveDirectory;
    2 usages
    @Getter
    private final Duration videoDuration;
    4 usages
    private Path currentSaveFile;
    2 usages
    private LocalDateTime currentSaveFileExpiration;

    3 usages  ↳ Ben Anderson
    public VideoFileManager(Path saveDirectory, Duration videoDuration) {
        this.saveDirectory = saveDirectory;
        this.videoDuration = videoDuration;
        nextSaveFile();
    }
}
```

Figure 264: Final VideoFileManager class part 1

```
private void nextSaveFile() {
    if (!Files.exists(saveDirectory)) {
        try {
            Files.createDirectories(saveDirectory);
        } catch (IOException e) {
            System.err.println("[ERROR] Failed to create videos directory.");
            e.printStackTrace();
        }
    }
    LocalDateTime currentDateTime = LocalDateTime.now();
    // "crms" stands for camera raw media save
    currentSaveFile = saveDirectory.resolve( other: currentDateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME) + ".crms");
    try {
        Files.createFile(currentSaveFile);
    } catch (IOException e) {
        System.err.println("[ERROR] Failed to create video save file.");
        e.printStackTrace();
    }
    currentSaveFileExpiration = currentDateTime.plus(videoDuration);
}

4 usages  Ben Anderson
public Optional<Path> getCurrentSaveFile() {
    if (currentSaveFileExpiration.isBefore(LocalDateTime.now())) {
        nextSaveFile();
    }
    return Files.exists(currentSaveFile) ? Optional.of(currentSaveFile) : Optional.empty();
}
```

Figure 265: Final VideoFileManager class part 2

```
package xyz.benanderson.scs.server;

import com.github.sarxos.webcam.Webcam;
import xyz.benanderson.scs.networking.Packet;
import xyz.benanderson.scs.networking.packets.DisconnectPacket;
import xyz.benanderson.scs.networking.packets.InfoPacket;
import xyz.benanderson.scs.networking.packets.LoginPacket;
import xyz.benanderson.scs.networking.packets.MediaPacket;
import xyz.benanderson.scs.server.account.User;
import xyz.benanderson.scs.server.account.UserManager;
import xyz.benanderson.scs.server.account.managers.MultiFileUserManager;
import xyz.benanderson.scs.server.configuration.ConfigurationWrapper;
import xyz.benanderson.scs.server.networking.Server;
import xyz.benanderson.scs.server.networking.ServerBuilder;
import xyz.benanderson.scs.server.video.CameraViewer;
import xyz.benanderson.scs.server.video.VideoEncoder;
import xyz.benanderson.scs.server.video.VideoFileManager;

import javax.imageio.ImageIO;
import java.io.IOException;
import java.net.InetAddress;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.*;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
import java.util.stream.Stream;

4 usages  ▲ Ben Anderson
public class Main {

    no usages  ▲ Ben Anderson
    public static void main(String[] args) throws IOException {
        ServerBuilder serverBuilder = new ServerBuilder(ConfigurationWrapper.getInstance().getServerPort(),
            InetAddress.getByName(ConfigurationWrapper.getInstance().getServerAddress()));
        UserManager userManager = new MultiFileUserManager();
        addAuthenticationToServerBuilder(serverBuilder, userManager);

        try {
            userManager.createUser(ConfigurationWrapper.getInstance().getDefaultUser());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Figure 266: Final Server Main class part 1

```
//build server and open camera
try {
    if (Webcam.getDefault() == null) {
        System.err.println("[ERROR] No Camera Detected");
        System.exit(status: 1);
    }
} catch (Exception e) {
    System.err.println("[ERROR] Error Occurred Accessing Camera");
    e.printStackTrace();
    System.exit(status: 1);
}

//setup server components
ImageIO.setUseCache(false);
try (Server server = serverBuilder.build();
    CameraViewer cameraViewer = new CameraViewer(Webcam.getDefault());
    VideoFileManager videoFileManager = new VideoFileManager(
        ConfigurationWrapper.getInstance().getVideoSaveDirectory(),
        ConfigurationWrapper.getInstance().getVideoDuration()
    );
    VideoEncoder videoEncoder = new VideoEncoder(videoFileManager);
    startVideoEncodingScheduledJob(videoFileManager, videoEncoder);
    System.out.println("[INFO] Server Started Successfully On Port " + server.getPort());
    while (true) {
        //attempt to capture camera image
        cameraViewer.captureImage().ifPresent(img -> {
            videoEncoder.appendToStream(img, System.currentTimeMillis());
            //if successful in capturing an image, create a packet from the image
            //and send it to all active connections
            Packet packet = new MediaPacket(img);
            server.getConnections().values().forEach(conn -> {
                if (loggedInUsers.contains(conn.getId()))
                    conn.getPacketSender().sendPacket(packet);
            });
        });
    }
} catch (Exception e) {
    e.printStackTrace();
}
}
```

Figure 267: Final Server Main class part 2

```
private final static Set<Path> videosBeingEncoded = Collections.synchronizedSet(new HashSet<>());  
  
1 usage  ↗ Ben Anderson  
private static void startVideoEncodingScheduledJob(VideoFileManager videoFileManager, VideoEncoder videoEncoder) {  
    ScheduledExecutorService scheduledExecutorService = Executors.newSingleThreadScheduledExecutor();  
    scheduledExecutorService.scheduleWithFixedDelay(() -> {  
        Optional<Path> currentSaveFile = videoFileManager.getCurrentSaveFile();  
        if (currentSaveFile.isEmpty()) {  
            System.err.println("[ERROR] An error occurred when accessing the current save file.");  
            return;  
        }  
        try (Stream<Path> pathStream = Files.list(videoFileManager.getSaveDirectory())) {  
            pathStream.filter(path -> path.toString().endsWith(".crms"))  
                .filter(path -> !path.equals(currentSaveFile.get()))  
                .filter(path -> !videosBeingEncoded.contains(path))  
                .forEach(path -> {  
                    videosBeingEncoded.add(path);  
                    videoEncoder.processRawMediaSave(path);  
                });  
        } catch (IOException e) {  
            System.err.println("[ERROR] An error occurred when searching for raw media save files to encode into videos.");  
        }  
    }, initialDelay: 1, delay: 1, TimeUnit.SECONDS);  
}  
  
3 usages  
private final static Set<UUID> loggedInUsers = Collections.synchronizedSet(new HashSet<>());
```

Figure 268: Final Server Main class part 3

```

static void addAuthenticationToServerBuilder(ServerBuilder serverBuilder, UserManager userManager) {
    //add `LoginPacket` listener on connect
    serverBuilder.onClientConnect((connection, server) -> {
        connection.getPacketListener().addCallback(LoginPacket.class, loginPacket -> {
            Optional<User> userOptional = userManager.getUser(
                //remove all '...' to protect against directory traversal vulnerability
                loginPacket.getUsername().replace(target: "...", replacement: ""))
            );
            //check if user exists with given username
            if (userOptional.isEmpty()) {
                //username not found in users
                connection.getPacketSender().sendPacket(new DisconnectPacket(reason: "Incorrect Credentials"));
                try {
                    Thread.sleep(millis: 200);
                    connection.close();
                } catch (Exception ignored) {}
                return;
            }
            User expectedUser = userOptional.get();
            //check if password is correct
            if (!expectedUser.getHashedPassword().equals(
                LoginPacket.hashPassword(loginPacket.getHashedPassword())))
                //password is wrong
                connection.getPacketSender().sendPacket(new DisconnectPacket(reason: "Incorrect Credentials"));
            try {
                Thread.sleep(millis: 200);
                connection.close();
            } catch (Exception ignored) {}
            return;
        }
        //login successful
        loggedInUsers.add(connection.getId());
        connection.getPacketSender().sendPacket(new InfoPacket("Correct Credentials"));
        System.out.println("[INFO] User '" + loginPacket.getUsername() + "' logged in successfully.");
    });
});
//remove connection from loggedInUsers on disconnect - prevents infinite memory usage
serverBuilder.onClientDisconnect((connection, server) -> {
    loggedInUsers.remove(connection.getId());
});
}
}

```

Figure 269: Final Server Main class part 4

```
server.address = 127.0.0.1
server.port = 8192
server.max-connections = 5

video.save-directory = ./videos/
video.duration.number = 60
video.duration.unit = seconds

users.save-directory = ./users/
users.default.username = admin
users.default.password = 5J6*9XodH&&mAUBz
users.default.is_admin = true
```

Figure 270: Final server.properties configuration file

```
package xyz.benanderson.scs.server.account;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import xyz.benanderson.scs.server.account.managers.MultiFileManager;
import xyz.benanderson.scs.server.configuration.ConfigurationWrapper;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
import java.util.Optional;

import static org.junit.jupiter.api.Assertions.*;

no usages  ↳ Ben Anderson
public class MultiFileManagerTest {

    4 usages
    static UserManager userManager = new MultiFileManager();
    2 usages
    static Path usersFolder = ConfigurationWrapper.getInstance().getUsersSaveDirectory();
    10 usages
    static Path userFile = ConfigurationWrapper.getInstance().getUsersSaveDirectory()
        .resolve("testUser".toLowerCase());
    no usages  ↳ Ben Anderson
    @BeforeAll
    static void createUsersFolder() throws IOException {
        Files.createDirectories(usersFolder);
    }

    no usages  ↳ Ben Anderson
    @AfterEach
    void deleteUserFile() throws IOException {
        Files.deleteIfExists(userFile);
    }

    no usages  ↳ Ben Anderson
    @AfterAll
    static void deleteUsersFolder() throws IOException {
        Files.deleteIfExists(userFile);
        Files.deleteIfExists(usersFolder);
    }

    no usages  ↳ Ben Anderson
    @Test
    public void testGetUserNotExists() { assertTrue(userManager.getUser(username: "noUser").isEmpty()); }
```

Figure 271: Final MultiFileManagerTest class part 1

```
@Test
public void test GetUserExists() throws IOException {
    Files.writeString(userFile, csq: "testUser\nhashedPassword\nfalse");
    Optional<User> userOptional = userManager.getUser( username: "testUser");
    assertTrue(userOptional.isPresent());
    assertEquals( expected: "testUser", userOptional.get().getUsername());
    assertEquals( expected: "hashedPassword", userOptional.get().getHashedPassword());
    assertFalse(userOptional.get().isAdmin());
}

no usages ▲ Ben Anderson
@Test
public void testCreateUser() throws Exception {
    assertFalse(Files.exists(userFile));
    User user = User.fromHashedPassword( username: "testUser", hashedPassword: "hashedPassword", admin: false);
    userManager.createUser(user);
    assertTrue(Files.exists(userFile));
    List<String> lines = Files.readAllLines(userFile);
    assertEquals( expected: "testUser", lines.get(0));
    assertEquals( expected: "hashedPassword", lines.get(1));
    assertFalse(Boolean.parseBoolean(lines.get(2)));
}

no usages ▲ Ben Anderson
@Test
public void deleteUser() throws Exception {
    assertFalse(Files.exists(userFile));
    Files.writeString(userFile, csq: "testUser\nhashedPassword\nfalse");
    assertTrue(Files.exists(userFile));
    userManager.deleteUser( username: "testUser");
    assertFalse(Files.exists(userFile));
}

}
```

Figure 272: Final MultiFileUserManagerTest class part 2

```
package xyz.benanderson.scs.server.configuration;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import xyz.benanderson.scs.networking.packets.LoginPacket;
import xyz.benanderson.scs.server.account.User;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.time.Duration;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;

//end-to-end integration test of the `Configuration` submodule
no usages  ▲ Ben Anderson
public class ConfigurationTest {

    5 usages
    private Path configFile;

    no usages  ▲ Ben Anderson
    @BeforeEach
    public void setupConfigFile() throws IOException {
        this.configFile = Paths.get(System.getProperty("user.dir")).resolve("server.properties");
        if (Files.exists(this.configFile)) {
            Files.delete(this.configFile);
        }
    }
}
```

Figure 273: Final ConfigurationTest class part 1

```
@Test
public void testConfigurationAndWrapper() {
    Path configFile = Paths.get(System.getProperty("user.dir")).resolve("server.properties");
    assertFalse(files.exists(configFile));

    assertEquals(ConfigurationWrapper.getInstance().getServerAddress(), actual: "127.0.0.1");
    assertEquals(ConfigurationWrapper.getInstance().getServerPort(), actual: 8192);
    assertEquals(ConfigurationWrapper.getInstance().getMaxConnections(), actual: 5);
    assertEquals(ConfigurationWrapper.getInstance().getVideoDuration(), Duration.ofMinutes(1));
    assertEquals(ConfigurationWrapper.getInstance().getVideoSaveDirectory(), Paths.get(first: "./videos/"));
    assertEquals(ConfigurationWrapper.getInstance().getUsersSaveDirectory(), Paths.get(first: "./users/"));
    assertEquals(ConfigurationWrapper.getInstance().getDefaultUser(),
        User.fromPlainTextPassword(username: "admin", LoginPacket.hashPassword(plainTextPassword: "5J6*9XodH&&mAUBz"), admin: true));
}

no usages ▾ Ben Anderson
@AfterEach
public void deleteConfigFile() throws IOException {
    if (Files.exists(this.configFile)) {
        Files.delete(this.configFile);
    }
}
```

Figure 274: Final Configuration Test class part 2

```
package xyz.benanderson.scs.server.networking;

import org.junit.jupiter.api.Test;
import xyz.benanderson.scs.networking.connection.Connection;

import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;

no usages  ↳ Ben Anderson
public class ServerTest {

    no usages  ↳ Ben Anderson
    @Test
    public void testServer() throws UnknownHostException {
        CompletableFuture<String> shutdownCompletableFuture = new CompletableFuture<>();
        Consumer<Server> serverShutdownListener =
            serv -> shutdownCompletableFuture.complete(value: "Shutdown");
        CompletableFuture<String> connectCompletableFuture = new CompletableFuture<>();
        BiConsumer<Connection, Server> connectListener =
            (con, serv) -> connectCompletableFuture.complete(value: "Connected");
        CompletableFuture<String> disconnectCompletableFuture = new CompletableFuture<>();
        BiConsumer<Connection, Server> disconnectListener =
            (con, serv) -> disconnectCompletableFuture.complete(value: "Disconnected");

        ServerBuilder serverBuilder = new ServerBuilder(port: 0, InetAddress.getLocalHost())
            .onServerShutdown(serverShutdownListener)
            .onClientConnect(connectListener)
            .onClientDisconnect(disconnectListener);
    }
}
```

Figure 275: Final ServerTest class part 1

```
try (Server server = serverBuilder.build()) {
    Socket socket = new Socket(InetAddress.getLocalHost(), server.getPort());
    Connection connection = new Connection(socket);

    assertEquals(expected: "Connected", connectCompletableFuture.get());
    assertFalse(disconnectCompletableFuture.isDone());
    assertFalse(shutdownCompletableFuture.isDone());

    connection.close();
    assertEquals(expected: "Connected", connectCompletableFuture.get());
    assertEquals(expected: "Disconnected", disconnectCompletableFuture.get());
    assertFalse(shutdownCompletableFuture.isDone());
} catch (Exception e) {
    throw new RuntimeException(e);
}
try {
    assertEquals(expected: "Connected", connectCompletableFuture.get());
    assertEquals(expected: "Disconnected", disconnectCompletableFuture.get());
    assertEquals(expected: "Shutdown", shutdownCompletableFuture.get());
} catch (InterruptedException | ExecutionException ignored) {}
}

}
```

Figure 276: Final ServerTest class part 2

```
package xyz.benanderson.scs.server.video;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.io.TempDir;

import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.Duration;

import static org.junit.jupiter.api.Assertions.assertArrayEquals;
import static org.junit.jupiter.api.Assertions.assertEquals;

no usages  ↳ Ben Anderson
public class VideoEncoderTest {

    no usages  ↳ Ben Anderson
    @Test
    public void testAppendToStream(@TempDir Path tempDir) throws IOException {
        //test data
        long numberOfFrames = 2;
        long startTimestamp = System.currentTimeMillis() - 1000;
        long endTimestamp = startTimestamp + 1000;
        BufferedImage[] testImages = getTestImages();

        //running the method to be tested with the test data
        VideoFileManager videoFileManager = new VideoFileManager(tempDir, Duration.ofMinutes(5));
        VideoEncoder videoEncoder = new VideoEncoder(videoFileManager);
        videoEncoder.appendToStream(testImages[0], startTimestamp);
        videoEncoder.appendToStream(testImages[1], endTimestamp);

        //serializing images so that they can be compared to the output
        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
        byteArrayOutputStream.writeBytes(videoEncoder.compressImage(testImages[0]));
        byteArrayOutputStream.writeBytes(videoEncoder.compressImage(testImages[1]));

        Path rawMediaSaveFile = videoFileManager.getCurrentSaveFile().orElseThrow(FileNotFoundException::new);
        //checking that the data outputted by the method being tested is correct
        try (InputStream inputStream = Files.newInputStream(rawMediaSaveFile)) {
            DataInputStream dataInputStream = new DataInputStream(inputStream) {
                assertEquals(numberOfFrames, dataInputStream.readLong());
                assertEquals(startTimestamp, dataInputStream.readLong());
                assertEquals(endTimestamp, dataInputStream.readLong());
                assertArrayEquals(byteArrayOutputStream.toByteArray(), dataInputStream.readAllBytes());
            }
        }
    }
}
```

Figure 277: Final `VideoEncoderTest` class part 1

```

@Test
public void testProcessRawMediaSave(@TempDir Path tempDir) throws IOException {
    //test data
    long numberofFrames = 2;
    long startTimestamp = System.currentTimeMillis() - 1000;
    long endTimestamp = startTimestamp + 1000;
    BufferedImage[] testImages = getTestImages();

    //file setup
    VideoFileManager videoFileManager = new VideoFileManager(tempDir, Duration.ofMinutes(5));
    VideoEncoder videoEncoder = new VideoEncoder(videoFileManager);

    Path rawMediaSaveFile = videoFileManager.getCurrentSaveFile().orElseThrow(FileNotFoundException::new);
    //writing test data to file
    try (OutputStream outputStream = Files.newOutputStream(rawMediaSaveFile)) {
        DataOutputStream dataOutputStream = new DataOutputStream(outputStream) {
            dataOutputStream.writeLong(numberofFrames);
            dataOutputStream.writeLong(startTimestamp);
            dataOutputStream.writeLong(endTimestamp);
            ImageIO.write(testImages[0], "jpg", dataOutputStream);
            ImageIO.write(testImages[1], "jpg", dataOutputStream);
        }
    }

    //running the method to be tested
    videoEncoder.processRawMediaSave(rawMediaSaveFile);

    File videoOutputFile = new File(rawMediaSaveFile.toString().replace(target: ".crms", replacement: ".mp4"));
    //todo parse output file to check length, fps etc. (maybe even frames
}

//private method to generate test images used as test data by other methods
2 usages Ben Anderson
private BufferedImage[] getTestImages() {
    //create two images
    BufferedImage frameOne = new BufferedImage(width: 1280, height: 720, BufferedImage.TYPE_INT_RGB);
    BufferedImage frameTwo = new BufferedImage(width: 1280, height: 720, BufferedImage.TYPE_INT_RGB);
    //fill first image with red pixels
    for (int y = 0; y < frameOne.getHeight(); y++) {
        for (int x = 0; x < frameOne.getWidth(); x++) {
            frameOne.setRGB(x, y, 0xff0000);
        }
    }
    //fill second image with blue pixels
    for (int y = 0; y < frameTwo.getHeight(); y++) {
        for (int x = 0; x < frameTwo.getWidth(); x++) {
            frameTwo.setRGB(x, y, 0x0000ff);
        }
    }
    //return images
    return new BufferedImage[] {frameOne, frameTwo};
}
}

```

Figure 278: Final VideoEncoderTest class part 2

```
package xyz.benanderson.scs.server;

import org.junit.jupiter.api.Test;
import xyz.benanderson.scs.networking.connection.Connection;
import xyz.benanderson.scs.networking.packets.DisconnectPacket;
import xyz.benanderson.scs.networking.packets.InfoPacket;
import xyz.benanderson.scs.networking.packets.LoginPacket;
import xyz.benanderson.scs.server.account.User;
import xyz.benanderson.scs.server.account.UserManager;
import xyz.benanderson.scs.server.networking.Server;
import xyz.benanderson.scs.server.networking.ServerBuilder;

import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.*;

no usages  ↳ Ben Anderson
public class ServerLoginTest {

    3 usages  ↳ Ben Anderson
    private UserManager mockUserManager() {
        User user = User.fromPlainTextPassword(username: "testUsername",
            LoginPacket.hashPassword(plainTextPassword: "testPassword"),
            admin: false);
        UserManager userManager = mock(UserManager.class);
        doReturn(Optional.of(user)).when(userManager).getUser(username: "testUsername");
        try {
            doNothing().when(userManager).createUser(any(User.class));
            doNothing().when(userManager).deleteUser(anyString());
        } catch (Exception ignored) {}
        return userManager;
    }
}
```

Figure 279: Final ServerLoginTest class part 1

```
@Test
public void testIncorrectPassword() throws UnknownHostException {
    CompletableFuture<String> result = new CompletableFuture<>();
    ServerBuilder serverBuilder = new ServerBuilder( port: 0, InetAddress.getLocalHost());
    Main.addAuthenticationToServerBuilder(serverBuilder, mockUserManager());
    try (Server server = serverBuilder.build()) {
        Socket socket = new Socket(InetAddress.getLocalHost(), server.getPort());
        Connection connection = new Connection(socket);
        connection.getPacketListener().addCallback(InfoPacket.class, infoPacket -> {
            result.complete(infoPacket.getInfo());
        });
        connection.getPacketListener().addCallback(DisconnectPacket.class, disconnectPacket -> {
            result.complete(disconnectPacket.getReason());
        });
        connection.getPacketSender().sendPacket(
            LoginPacket.fromPlainTextPassword(
                username: "testUsername", plainTextPassword: "incorrectPassword"
            )
        );
        assertEquals( expected: "Incorrect Credentials", result.get());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

no usages  ↳ Ben Anderson
@Test
public void testIncorrectUsername() throws UnknownHostException {
    CompletableFuture<String> result = new CompletableFuture<>();
    ServerBuilder serverBuilder = new ServerBuilder( port: 0, InetAddress.getLocalHost());
    Main.addAuthenticationToServerBuilder(serverBuilder, mockUserManager());
```

Figure 280: Final ServerLoginTest class part 2

```

try (Server server = serverBuilder.build()) {
    Socket socket = new Socket(InetAddress.getLocalHost(), server.getPort());
    Connection connection = new Connection(socket);
    connection.getPacketListener().addCallback(InfoPacket.class, infoPacket -> {
        result.complete(infoPacket.getInfo());
    });
    connection.getPacketListener().addCallback(DisconnectPacket.class, disconnectPacket -> {
        result.complete(disconnectPacket.getReason());
    });
    connection.getPacketSender().sendPacket(
        LoginPacket.fromPlainTextPassword(
            username: "testUsername", plainTextPassword: "incorrectPassword"
        )
    );
    assertEquals(expected: "Incorrect Credentials", result.get());
} catch (Exception e) {
    throw new RuntimeException(e);
}
}

no usages  ↳ Ben Anderson
@Test
public void testSuccessfulLogin() throws UnknownHostException {
    CompletableFuture<String> result = new CompletableFuture<>();
    ServerBuilder serverBuilder = new ServerBuilder(port: 0, InetAddress.getLocalHost());
    Main.addAuthenticationToServerBuilder(serverBuilder, mockUserManager());
    try (Server server = serverBuilder.build()) {
        Socket socket = new Socket(InetAddress.getLocalHost(), server.getPort());
        Connection connection = new Connection(socket);
        connection.getPacketListener().addCallback(InfoPacket.class, infoPacket -> {
            result.complete(infoPacket.getInfo());
        });
        connection.getPacketListener().addCallback(DisconnectPacket.class, disconnectPacket -> {
            result.complete(disconnectPacket.getReason());
        });
        connection.getPacketSender().sendPacket(
            LoginPacket.fromPlainTextPassword(
                username: "testUsername", plainTextPassword: "testPassword"
            )
        );
        assertEquals(expected: "Correct Credentials", result.get());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
}

```

Figure 281: Final ServerLoginTest class part 3

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <artifactId>scs_parent</artifactId>
        <groupId>xyz.benanderson</groupId>
        <version>0.0.2</version>
    </parent>

    <artifactId>scs_server</artifactId>
    <version>0.0.2</version>

    <properties>
        <maven.compiler.source>16</maven.compiler.source>
        <maven.compiler.target>16</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>xyz.benanderson</groupId>
            <artifactId>scs_networking</artifactId>
            <version>0.0.4</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-nop</artifactId>
            <version>2.0.5</version>
        </dependency>
        <dependency>
            <groupId>com.github.sarxos</groupId>
            <artifactId>webcam-capture</artifactId>
            <version>0.3.12</version>
        </dependency>
        <dependency>
```

Figure 282: Final Server pom.xml part 1

```
<dependency>
    <groupId>org.jcodec</groupId>
    <artifactId>jcodec</artifactId>
    <version>0.2.5</version>
</dependency>
<dependency>
    <groupId>org.jcodec</groupId>
    <artifactId>jcodec-javase</artifactId>
    <version>0.2.5</version>
</dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.4</version>
            <configuration>
                <archive>
                    <manifest>
                        <addClasspath>true</addClasspath>
                        <mainClass>xyz.benanderson.scs.server.Main</mainClass>
                    </manifest>
                </archive>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.2.4</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <createDependencyReducedPom>false</createDependencyReducedPom>
                        <filters>
                            <filter>
                                <includes>
                                    <include>xyz.benanderson.scs.networking.*</include>
                                </includes>
                            </filter>
                        </filters>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

</project>
```

Figure 283: Final Server pom.xml part 2

```

package xyz.benanderson.scs;

import java.util.function.Consumer;

no usages ▲ Ben Anderson
public class StandardInputTestSuite {

    1 usage
    private static final String[] integers = {
        "-1", "0", "1", "2147483647", "-2147483647", "1000000", "-1000000",
        "4294967295", "2147483648", "-2147483648", "4294967296", "9999999999999999"
    };
    1 usage
    private static final String[] strings = {
        "abc", "a b c", "1234567890", "ï™£¢¤¤§¶•°°-≠", "(J°□°) J ↴ █",
        "♥", ".;/'[ ]\\-=\n", "<>?:\\"{}|_+", "!@#$%^&*( )`~", "${{<%[%"}}%\\.", "\u200F\u200E\u200E",
        "abcdefghijklmnopqrstuvwxyz", "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" +
            "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" +
            "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" +
            "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
    };
    1 usage
    private static final String[] reals = {
        "1.23", "1.0", "99999999999999.0", "0.99999999999999", "0.0", "1", "0.0000000000000001", "123456789.123456789"
    };
    1 usage
    private static final String[] characters = {
        "c", "1", "\u200E", "¶", "♥", "\n"
    };
    1 usage
    private static final String[] booleans = {
        "True", "TRUE", "true", "False", "FALSE", "false", "1", "0"
    };
    1 usage
    private static final String[] erroneous = {
        "abc", "a", "123", "1.23", "True", "False", "", null
    };
}

```

Figure 284: Final StandardInputTestSuite class part 1

```
public static void testInteger(Consumer<String> integerConsumer) {
    for (String testInput : integers) {
        integerConsumer.accept(testInput);
    }
}

no usages  ↳ Ben Anderson
public static void testString(Consumer<String> stringConsumer) {
    for (String testInput : strings) {
        stringConsumer.accept(testInput);
    }
}

no usages  ↳ Ben Anderson
public static void testReal(Consumer<String> doubleConsumer) {
    for (String testInput : reals) {
        doubleConsumer.accept(testInput);
    }
}

no usages  ↳ Ben Anderson
public static void testCharacter(Consumer<String> characterConsumer) {
    for (String testInput : characters) {
        characterConsumer.accept(testInput);
    }
}

no usages  ↳ Ben Anderson
public static void testBoolean(Consumer<String> booleanConsumer) {
    for (String testInput : booleans) {
        booleanConsumer.accept(testInput);
    }
}

no usages  ↳ Ben Anderson
public static void testErroneous(Consumer<String> stringConsumer) {
    for (String erroneousInput : erroneous) {
        stringConsumer.accept(erroneousInput);
    }
}

}
```

Figure 285: Final StandardInputTestSuite class part 2

Figure 286: Final StandardTestSuite class part 1

```
public static void testInteger(Consumer<Integer> integerConsumer) {
    for (int testInput : integers) {
        integerConsumer.accept(testInput);
    }
}

no usages  ↳ Ben Anderson
public static void testString(Consumer<String> stringConsumer) {
    for (String testInput : strings) {
        stringConsumer.accept(testInput);
    }
}

no usages  ↳ Ben Anderson
public static void testReal(Consumer<Double> doubleConsumer) {
    for (double testInput : reals) {
        doubleConsumer.accept(testInput);
    }
}

no usages  ↳ Ben Anderson
public static void testCharacter(Consumer<Character> characterConsumer) {
    for (char testInput : characters) {
        characterConsumer.accept(testInput);
    }
}

no usages  ↳ Ben Anderson
public static void testBoolean(Consumer<Boolean> booleanConsumer) {
    for (boolean testInput : booleans) {
        booleanConsumer.accept(testInput);
    }
}

}
```

Figure 287: Final StandardTestSuite class part 2

Bibliography

- [1] Ben Anderson. (2022). *Web Security Basics - Cookies & Authenticated Sessions*. <https://www.benanderson.xyz/2022/04/05/web-cookies-auth-sessions/> : Ben Anderson
- [2] BluecherryDVR. (2022). *Linux video surveillance software*. <https://www.bluecherrydvr.com/> : BluecherryDVR
- [3] Christian Neumanns. (2018). *A quick and thorough guide to ‘null’: what it is, and how you should use it*. <https://www.freecodecamp.org/news/a-quick-and-thorough-guide-to-null-what-it-is-and-how-you-should-use-it-d170cea62840/> : freeCodeCamp
- [4] Chuck Gehman. (2019). *What Is Version Control Software (VCS)?*. <https://www.perforce.com/blog/vcs/what-is-version-control> : Perforce
- [5] CircleHD. (2020). *How to Accurately Calculate Video File Size*. <https://www.circlehd.com/blog/how-to-calculate-video-file-size> : CircleHD
- [6] Curtis Hall. (2019). *Bluecherry Client Setup*. <https://bluecherry-apps.readthedocs.io/en/latest/client.html> : Bluecherry
- [7] Elma Mrkonjić. (2020). *Linux Statistics: Market Share, Usage, and Fun Facts*. <https://writersblocklive.com/blog/linux-statistics/#stat12> : Writer's Block Live
- [8] Gradle. (2022). *Gradle Build Tool*. <https://gradle.org/> : Gradle
- [9] Harald K. (2016). *Java BufferedImage JPG compression without writing to file*. <https://stackoverflow.com/q/37713773/14844306> : StackOverflow
- [10] jcodec. (2022). *jcodec*. <https://github.com/jcodec/jcodec> : jcodec
- [11] JetBrains. (2022). *IntelliJ IDEA: The Capable & Ergonomic Java IDE by JetBrains*. <https://www.jetbrains.com/idea/> : JetBrains
- [12] JetBrains. (2022). *JetBrains Annotations*. <https://github.com/JetBrains/java-annotations> : JetBrains
- [13] JetBrains. (2021). *@Nullable and @NotNull*. <https://www.jetbrains.com/help/idea/nullability-and-notnull-annotations.html> : JetBrains
- [14] Jongmin Park. (2019). *Connect the webcam to Docker on Mac or Windows*. <https://medium.com/@jjupax/connect-the-webcam-to-docker-on-mac-or-windows-51d894c44468> : Medium
- [15] Leif Andersen. (2010). *Good Hash Function for Strings*. <https://stackoverflow.com/questions/2624192/good-hash-function-for-strings> : StackOverflow
- [16] Maged M. Michael & Michael L. Scott. (1996). *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*. https://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf : University of Rochester
- [17] Matt Sgarlata. (2014). *How to use JUnit to test asynchronous processes*. <https://stackoverflow.com/a/23809378/14844306> : Stack Exchange Inc
- [18] Max Vollmer. (2018). *How do I make a video file out of images in Java?*. <https://stackoverflow.com/a/48237935/14844306> : Stackoverflow
- [19] Microsoft. (2021). *nslookup*. <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/nslookup> : Microsoft

- [20] mkyong. (2020). *Java – How to convert byte arrays to Hex.* <https://mkyong.com/java/java-how-to-convert-bytes-to-hex/> : mkyong
- [21] Mockito. (2022). *Mockito framework site.* <https://site.mockito.org/> : Mockito
- [22] ONVIF. (2019). *Profile S Specification.* https://www.onvif.org/wp-content/uploads/2019/12/ONVIF_Profile_-S_Specification_v1-3.pdf : ONVIF
- [23] Oracle. (2014). *AtomicBoolean Javadoc.* <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicBoolean.html> : Oracle
- [24] Oracle. (2014). *ConcurrentLinkedQueue Javadoc.* <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html> : Oracle
- [25] Oracle. (2021). *How to Write Doc Comments for the Javadoc Tool.* <https://www.oracle.com/uk/technical-resources/articles/java/javadoc-tool.html> : Oracle
- [26] Oracle. (2014). *ObjectInputStream Constructor Documentation.* <https://docs.oracle.com/javase/8/docs/api/java/io/ObjectInputStream.html#ObjectInputStream-java.io.InputStream-> : Oracle
- [27] Oracle. (2014). *ObjectOutputStream#flush Documentation.* <https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html#flush--> : Oracle
- [28] Oracle. (2014). *ObjectOutputStream#reset Documentation.* <https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html#reset--> : Oracle
- [29] Oracle. (2014). *ObjectOutputStream#writeObject Documentation.* <https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html#writeObject-java.lang.Object-> : Oracle
- [30] Oracle. (2014). *Optional Documentation.* <https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html> : Oracle
- [31] Oracle. (2014). *Socket (Java Platform SE 8).* <https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html#isConnected--> : Oracle
- [32] Oracle. (2011). *The try-with-resources Statement.* <https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html> : Oracle
- [33] Oracle. (2011). *What Is a Socket?.* <https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html> : Oracle
- [34] PortSwigger. (2020). *Server-side template injection.* <https://portswigger.net/web-security/server-side-template-injection> : PortSwigger
- [35] Ring. (2020). *Stick Up Cam Battery .* <https://en-uk.ring.com/products/stick-up-security-camera-battery> : Ring
- [36] Ring. (2021). *Understanding Color Night Vision and Troubleshooting Night Vision .* <https://support.ring.com/hc/en-us/articles/360038559351-Understanding-Color-Night-Vision-and-Troubleshooting-Night-Vision> : Ring
- [37] Sam Barnum. (2013). *How to serialize an object that includes BufferedImages.* <https://stackoverflow.com/a/15127955/14844306> : StackOverflow
- [38] Scott Chacon. (2008). *Git.* <https://git-scm.com/> : Git
- [39] Stack Exchange Inc. (2018). *Difference Between flush() vs reset() in JAVA.* <https://stackoverflow.com/questions/51115157/difference-between-flush-vs-reset-in-java> : Stack Exchange Inc

- [40] Stack Exchange Inc. (2022). *What's the difference between a mock & stub?*. <https://stackoverflow.com/questions/3459287/whats-the-difference-between-a-mock-stub> : Stack Exchange Inc
- [41] Strategy Analytics. (2021). *Strategy Analytics: Amazon's Ring Remained atop the Video Doorbell Market in 2020*. <https://www.businesswire.com/news/home/20210512005336/en/> : Business Wire
- [42] Tapo. (2020). *Home Security Wi-Fi Camera | Tapo C100*. <https://www.tapo.com/uk/product/smart-camera/tapo-c100/> :
- [43] Tech Pro Team. (2021). *How to Use the Ring App*. <https://www.techsolutions.support.com/how-to/how-to-use-the-ring-app-11500> : Support.com
- [44] The Apache Software Foundation. (2022). *Apache Maven*. <https://maven.apache.org/> : The Apache Software Foundation
- [45] The Gitea Authors. (2022). *Gitea*. <https://gitea.io/en-us/> : Gitea
- [46] The JUnit Team . (2022). *JUnit 5*. <https://junit.org/junit5/> : The JUnit Team
- [47] The Project Lombok Authors. (2022). *Project Lombok*. <https://projectlombok.org/> : The Project Lombok Authors
- [48] The Project Lombok Authors. (2022). *@Getter and @Setter*. <https://projectlombok.org/features/GetterSetter> : The Project Lombok Authors
- [49] The Project Lombok Authors. (2022). *@NoArgsConstructor, @RequiredArgsConstructor, @AllArgsConstructor*. <https://projectlombok.org/features/constructor> : The Project Lombok Authors
- [50] Tom Preston-Werner. (2022). *Semantic Versioning 2.0.0*. <https://semver.org/> : semver.org
- [51] Wikipedia. (2004). *Hosts file*. [https://en.wikipedia.org/wiki/Hosts_\(file\)](https://en.wikipedia.org/wiki/Hosts_(file)) : Wikipedia
- [52] Wikipedia. (2022). *IEEE 802.11*. https://en.wikipedia.org/wiki/IEEE_802.11 : Wikipedia
- [53] Wikipedia. (2021). *IEEE 802.11w-2009*. https://en.wikipedia.org/wiki/IEEE_802.11w-2009 : Wikipedia
- [54] Wikipedia. (2009). *IPv6*. <https://en.wikipedia.org/wiki/IPv6> : Wikipedia
- [55] Wikipedia. (2022). *Log4j*. <https://en.wikipedia.org/wiki/Log4j> : Wikipedia
- [56] Wikipedia. (2001). *Regular Expression*. https://en.wikipedia.org/wiki/Regular_expression : Wikipedia
- [57] Wikipedia. (2003). *Transmission Control Protocol*. https://en.wikipedia.org/wiki/Transmission_Control_Protocol#TCP_ports : Wikipedia

