



Milestone 4

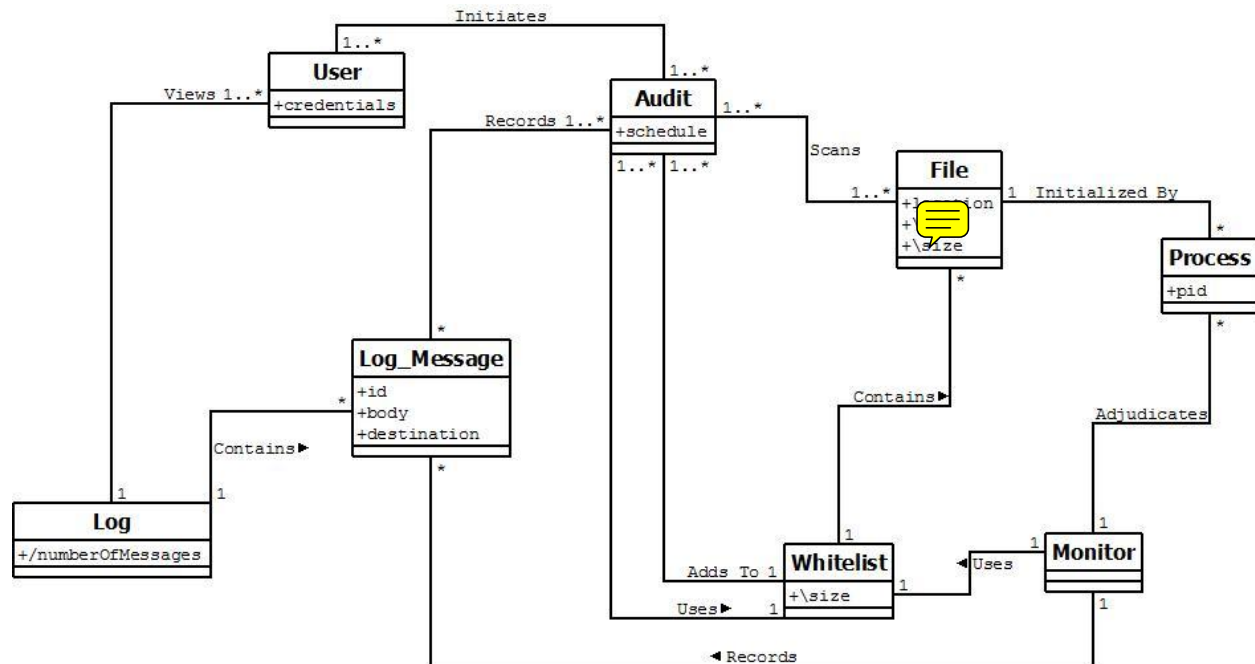
Software Security API

1-29-10

Eric Crockett
Josh Dash
David Pick

Domain Model

Our domain model contains domain objects relevant to our system.

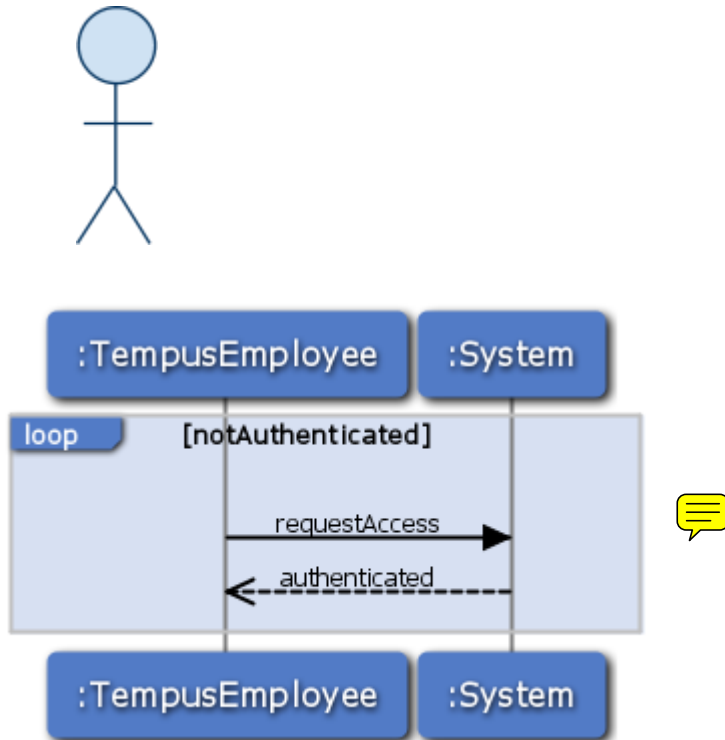


In this model, Users (Tempus employees) can initiate system Audits, or Audits may be scheduled to run regularly. Users must be authenticated to the Audit system before they are allowed to initiate a system Audit. A system Audit scans the entire directory tree. It compares the hash of each scanned file to the Whitelist to see if the file has been previously detected. The system Audit may add files to the Whitelist and will also send Log_Messages back to Tempus. Users can later view all of the Log_Messages. A separate part of the system is the Monitor. The Monitor detects when a new Process attempts to run, checks it against the Whitelist, and then either approves or denies the Process. Each Process is associated with the File which initiated it. The Monitor routine may also send Log_Messages which can be viewed later by Users. Although Files and Processes may seem synonymous, they are distinct in our model because of what they symbolize. Files represent files in the hard drive which have been whitelisted; they are created by the Audit and persist as long as they are whitelisted. Processes on the other hand exist only in memory and represent a running program. A new Process is created when some executable is run. The Monitor can either approve or deny the Process. Even though Processes and Files exist in a 1-1 relationship, they are distinct because Audit operates on Files, while Monitor operates on Processes.

System Sequence Diagrams

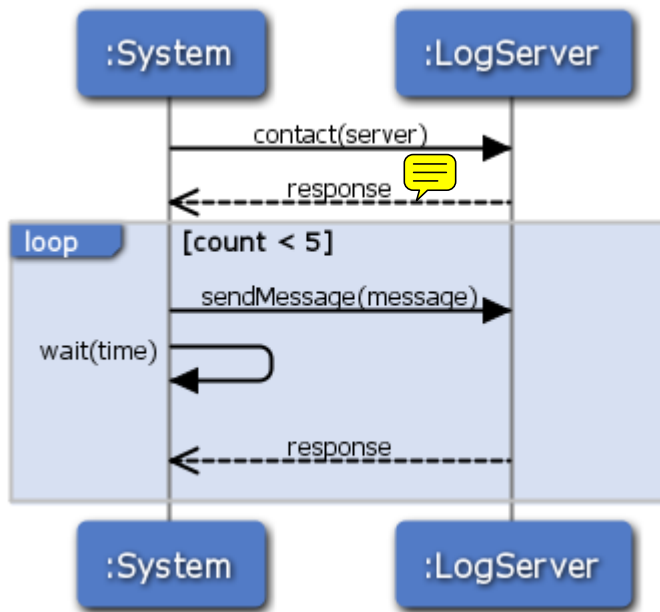
The following system sequence diagrams (SSD) show the interactions between our system and external actors, such as the computer user (represented as ":TempusClient"), a Tempus employee (represented as ":TempusEmployee"), and the Tempus log server. The Software Security API is represented as a black box by ":System" in the diagrams.

Login SSD:



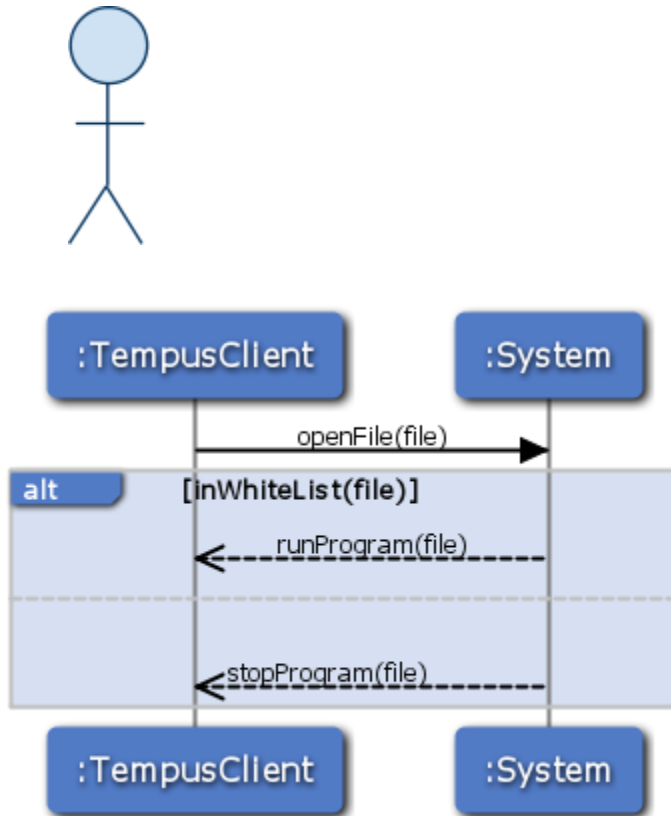
When a user attempts to log into the system the user sends a message to the system with their credentials, the system will then reply with whether their credentials were accepted or not.

SendLog Message SSD:



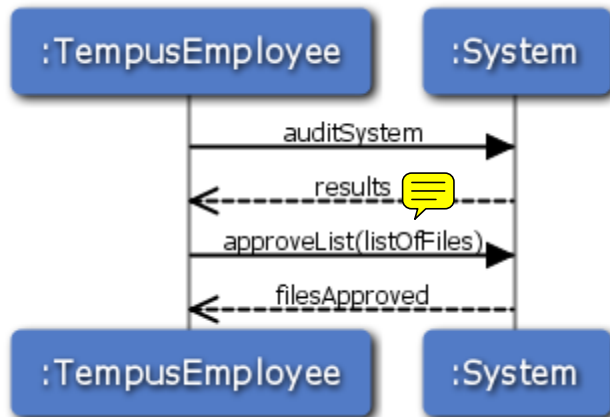
When the system needs to make a log message, it first tries to contact the remote sever, if it receives a response it then tries to send the message, if after a specific wait time no confirmation has been received from the server, the system will try again, up to five times.

Open File SSD:



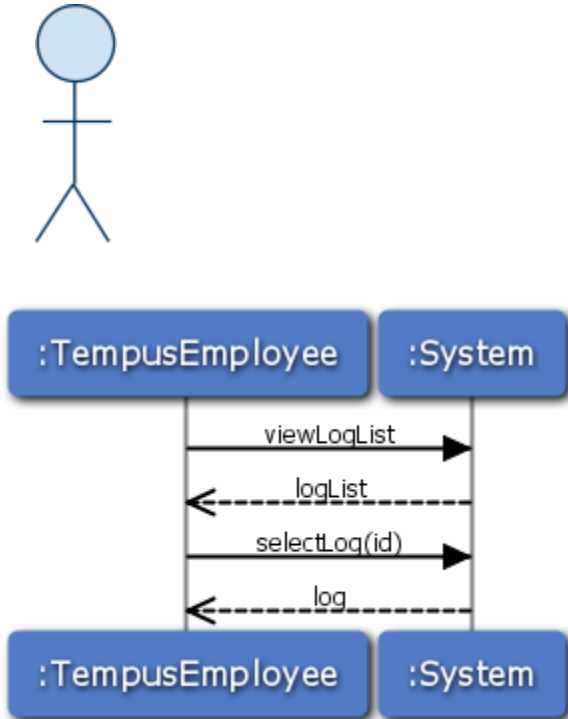
When a user tries to open a file it will first send an **openFile** request to the system through Windows. The system will then check to see if the file is in the whitelist, if it is, the system will run the program for the user, if it is blacklisted the program will be prevented from running. Note the **stopProgram()** message has an operation contract below containing more details.

Audit SSD:



When a user tries to perform an audit of the system, they first send an audit message to the system. The system then replies with the results of the audit, and a list of any new files that were found. This way the user can then tell the system which files should be whitelisted, by sending an `approveList` message. The user only needs to send files to be whitelisted, because the system already has a full list of files, and files that are not going to be whitelisted, are assumed to be blacklisted. Note the `approveList()` message has an operation contract below containing more details.

View Log SSD:



When a user would like to view a log, they first must query the system for a list of all of the logs. The system will reply with this list, when the user has decided which log message they would like to view, they send a message to the system with it's ID. The system will then reply with the correct log message.

Operation Contracts

The operation contracts below provide details for some of the more complex system operations described above in system sequence diagrams. Operation contracts describe the changes the system has undergone as a result of a system operation.

Operation Contract for approveList
Operation: approveList(listOfFiles:FilePath)



Cross References: Audit SSD

Preconditions: Initial scan has been performed.

Postconditions:

- For each approved file location in the list, a new File fn was created.
 - fn.location was set to listOfFiles[n].
 - fn.name was set to name of file.
 - fn.LastSolidified was set to current datetime.
 - fn.GUID was generated.
 - fn.LastAccessed was set to file's last access datetime.
 - fn.Instance was set to "local". fn.hash was generated from the file at fn.location.
 - fn.size was recorded from the file at fn.location.
 - fn.whitelisted was set to true.
- For each file location that was scanned but was not in the approved list, a new File fn was created.
 - fn.location was set to listOfFiles[n].
 - fn.name was set to name of file.
 - fn.LastSolidified was set to current datetime.
 - fn.GUID was generated.
 - fn.LastAccessed was set to file's last access datetime.
 - fn.Instance was set to "local". fn.hash was generated from the file at fn.location.
 - fn.size was recorded from the file at fn.location.
 - fn.whitelisted was set to false.

Operation Contract for stopProgram



Operation: stopProgram(file)

Cross References: OpenFile SSD

Preconditions: Monitor is running and user has opened a file that is not in the whitelist.

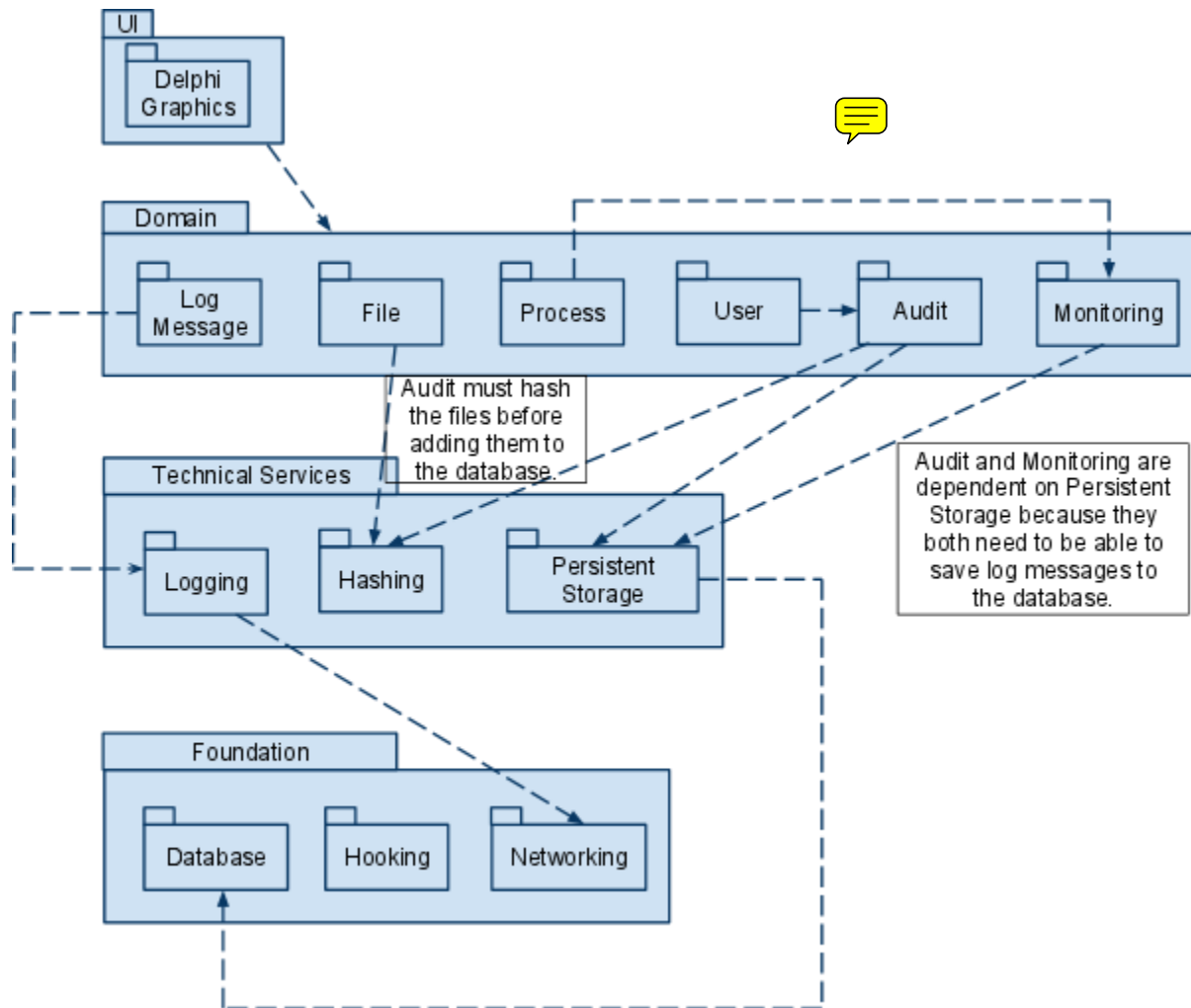
Postconditions:

- A new Log_Message m was created.
- m.GUID was generated.
- m.classification was set to "execution denied".

- m.severity was set to "important".
m.body was set to include the file name and location.
m.destination was set to external_server. m.timestamp was set to current datetime.
- m was added to the internal database.
- m was sent to external_server.

Package Diagram and Logical Architecture

This diagram groups classes into packages and layers. It shows how the overall system will fit together regardless of operating system, or hardware it is running on. Each layer shows a grouping of classes, packages, or subsystems, that are related for a major aspect of the system. The diagram is organized so that higher layers call upon services of lower layers.



The Foundation layer includes the basic technologies that the project relies on. Above that are the Technical Services which provides the upper layers with an interface for the low level Foundation packages. The Domain layer contains packages that run the software and holds current system information in memory. The UI layer rests above everything else and allows the user to interact with the Domain layer to control the system.

The Foundation includes the Database, Hooking, and Networking packages. The Database holds the Accuracer database classes. Hooking consists of Madshi madCodeHook and is used to hook onto the Windows API to monitor processes. Networking contains components to send log messages to Tempus' servers.

The Technical Services layer holds the Logging package to save, access, and send log messages and the Monitoring package that utilizes the Database and Hooking packages to intercept and validate files running on the system. Logging needs the database to save messages to the local database log, and the network to send the log to Tempus' remote servers. The Monitoring package reads file validation from the database and monitors the system through the hooking service. Hashing provides a format for recording files.

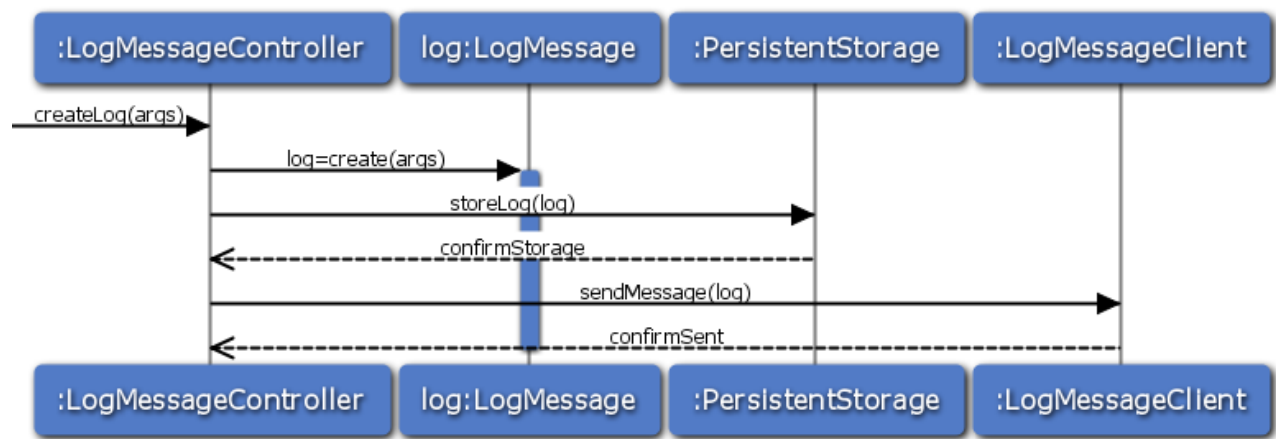
The Domain layer contains individual Log Messages that are recorded with the Logging package. An Audit package exists to periodically scan the entire system and allow new files to be allowed to run. The Audit system needs the Hashing service to store information in the Database. Files represent various files on the system and relies on Hashing in Technical Services to compute the hash. Likewise, Processes represent processes currently running on the system and need to be approved by the monitor. The User package exists to contain information about the current user of the system and allow them to update it as necessary. It needs to access Audit to start a periodic scan of the system.

The UI layer consists of various Delphi Graphics classes. It allows control of the system and needs access to various parts of the Domain layer to appropriately instruct the system.

Interaction Diagrams

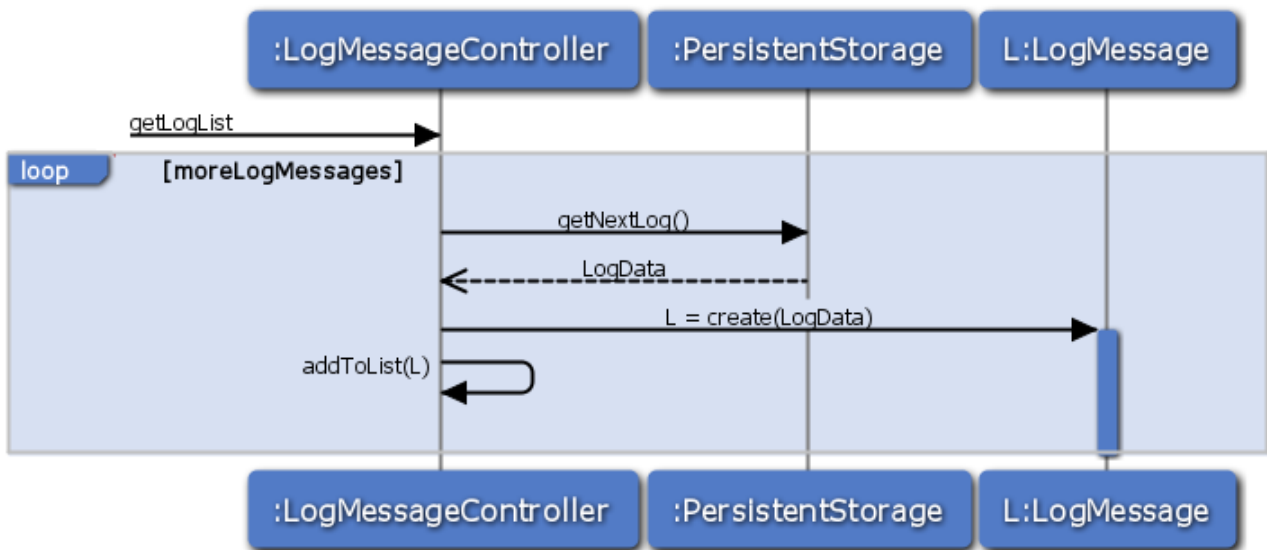
These diagrams detail the user interactions detail system operations in the SSDs above.


SendLogMessage Sequence Diagram



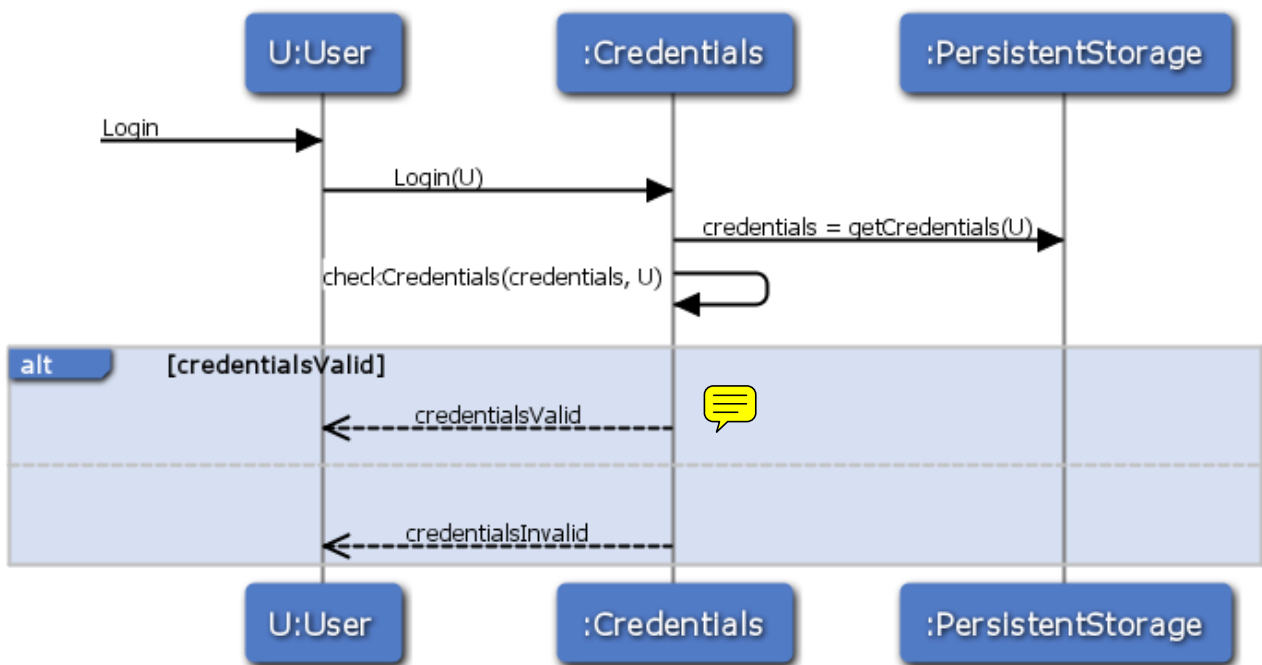
This sequence diagram shows how a log message is processed. Either Audit or Monitor can trigger a loggable event, which results in a 'createLog(args)' message being sent with the appropriate parameters. The LogMessageController creates a new LogMessage from these arguments, then sends it to PersistentStorage as well as the LogMessageClient, which it turn sends the message to an external server.

ViewLogList Sequence Diagram



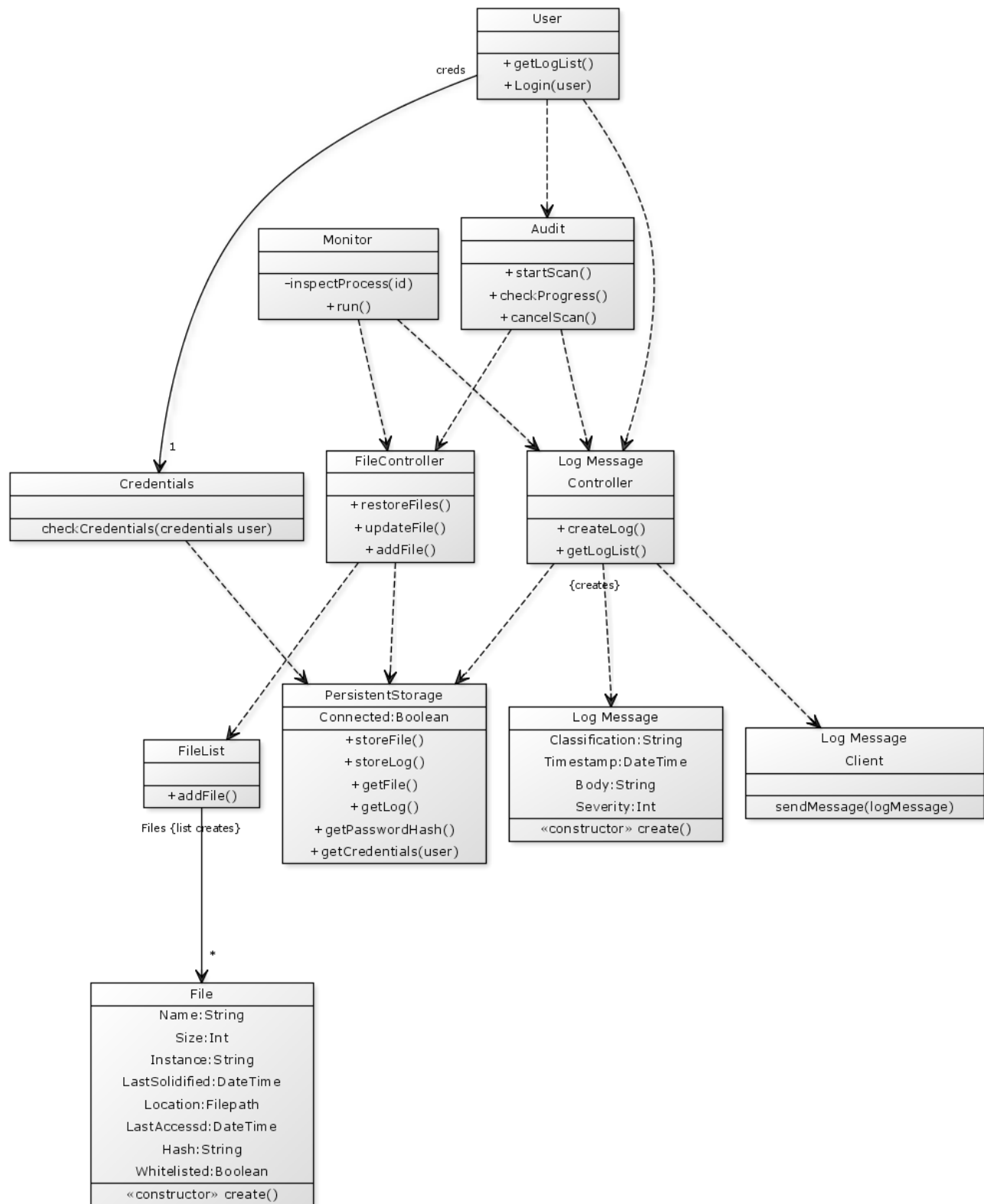
This sequence diagram shows how a list of log messages stored in the system is returned. This function will be called when a user would like to view all the log messages that have been stored by the system. When the found message is received `LogMessageController` asks `PersistentStorage` for each log. Since `PersistentStorage` does not store `LogMessage` objects, the `LogMessageController` must create a `LogMessage` object, each time `PersistentStorage` returns log data. When the `LogMessage` object is created it is added to a list, which is eventually what the function returns. 

Login Sequence Diagram



This sequence diagram shows what happens when a user tries to log into the system. First the user sends a message to the Credentials object. The credentials object, then looks up the actual credentials in the database. It then compares the two sets of credentials and either allows the user to login or informs them that it was an invalid username and password set.

Design Class Diagram



This design class diagram describes the software perspective of the design. Some classes are inspired by domain model concepts, such as User, Audit, Monitor, and File. Other classes do not appear in the domain model, but are part of the logical design, such as LogMessageClient and Persistent Storage.

User represents a logical authenticated entity, not a physical person. A User can start a system audit, and needs access to PersistentStorage to check for a valid password. A User gets a list of log messages from the LogMessageController.

Audit can trigger log messages as it performs a system scan, so it needs to have access to the LogMessageController. It also needs access the FileList via the FileController.

Similarly, Monitor can trigger log messages as it protects the system, and also requires access to the FileList. The Monitor is started by Delphi, it exists when the program starts.

FileController provides a cohesive and stable interface for interacting with files on the system. It hides the duplication of the file list in memory and in the database and handles all synchronization issues. FileController is tasked with repopulating FileList from PersistentStorage, but it does not <<create>> FileList; Delphi handles the creation of some objects.

Similarly, LogMessageController provides a cohesive interface for log messages, and hides the logic of sending log messages to a remote server as well as to the local database.


The LogMessageClient exports the logic of sending a log message to an external location from LogMessageController. This creates a cohesive design and protects the rest of the system from variation in the external server interface.

An explicit log message will be created because the information is needed for both internal storage and the external server.

PersistentStorage allows User, LogMessageController, and FileController to access and store data in the underlying embedded database.

FileList is a list of files, which may be either whitelisted or blacklisted. This list mirrors the File table in the PersistentStorage, but is kept as an object for performance purposes.

GRASP Patterns in Design

The General Responsibility Assignment Software Patterns (GRASP) are a collection of design patterns which aid in making a good object-oriented design. Below are examples of how we applied the GRASP patterns to our design. 

High Cohesion:

In order to keep our objects focused and manageable, we chose to create two "Controller" objects, which handle all log events and all file events. This design increases the cohesion of Audit and Monitor, which deal only with the two controller objects in our design. An alternative to this approach would be to let Monitor, for example, create its own LogMessages, and also be responsible for sending them to the database and to the external server. This alternative design makes Monitor have many responsibilities, which decreases the cohesion of the design. This is why we chose to create both the File and LogMessage Controllers. As a side effect of our decision, our design has lower coupling than the alternative would have had.

Low Coupling:

In order to reduce the impact of change on our design, we designed the system to have low coupling. This pattern is useful in evaluating design alternatives. For example, consider the user class and the operation to get a list of log messages. This corresponds to the "View Log" SSD, and the "ViewLogList" sequence diagram. We chose to couple User to the LogMessageController. PersistentStorage is already visible to the LogMessageController, and LogMessageController also already has the capability to create a LogMessage.

An alternative design that we considered was to let User control the operation by being coupled to both PersistentStorage and LogMessage. We felt that LogMessage could be a relatively unstable part of our design, since we might choose to change some fields at a later time. Thus we wanted to avoid coupling with LogMessage if possible. Further, our choice to use LogMessageController reduces overall coupling in the design and increases the cohesion of User.

Information Expert:

We choose not to use information expert and instead used controllers in order to lower coupling and increase cohesion. For example, while it might seem like Monitor should create LogMessages directly since it has the information to construct them, we choose to use a controller so that Monitor would not be forced to interact with LogMessageClient, LogMessage, and Persistent Storage. This reduces coupling with both Monitor and Audit and increases cohesion of the overall design.

Polymorphism:

When deciding how to implement our logging system we considered creating a separate class for each kind of log message we would have to create, which would follow the polymorphism pattern. However after several discussions we decided this would be a bad idea because there was not enough variation in the kinds of log messages that would be created in order to justify creating separate classes. Log messages have several fields which identify their attributes, and these need to be stored explicitly in the database anyway. Thus polymorphic message types are

not a part of our design. No other aspect of our design merited polymorphism because almost all of the other classes will be singletons.

Creator:

We applied the creator pattern by having the FileList class create File objects. This made sense because FileList is a composite of Files, and records every file that is stored in the database. FileList also closely uses individual files when a request for an individual file is made. Another alternative would be to have FileController create Files, and then insert them into the FileList. This would have increased coupling with FileController, and we should also note that FileController closely uses the File List, not individual Files. The alternative design has the advantage that FileController has the data to initialize a File when it obtains it from the database. However, we chose to let FileList create the files because FileController doesn't need to be concerned with individual Files.

Controller:

Our system contains several use case controllers which receive system operations from the UI. User receives the message to login and to retrieve log messages (see Login Sequence Diagram). Audit is responsible for scanning the system and returning results. Monitor receives messages from the operating system. These three classes correspond to our three main use cases. An alternative design would be to use a facade controller. We chose to use use case controllers because our use cases are disjoint. A facade controller would have been incohesive because it would have been responsible for many different system operations. Further, we needed Monitor to be its own controller because it will most likely need its own thread.

Pure Fabrication:

Although there is no concept of PersistentStorage in our domain model, we used Pure Fabrication to create a PersistentStorage class. The PersistentStorage class increases the cohesion of our design because all of the database communication logic is abstracted out to our Pure Fabrication rather than being duplicated in each object that needs the database. This means that other classes do not need to know how to talk to the database, they simply will have to make class to PersistentStorage, which will know how to talk to the database. If no Pure Fabrication was used in this case, our design would have low cohesion. Our Pure Fabrication also protects the rest of our system from a varying database interface and results in a reusable component that could be used in future systems. All three of our interaction diagrams use PersistentStorage, which shows how many other objects rely on database communication in our system.

Protected Variations and Indirection:

Our system uses Protected Variations and Indirection to introduce the LogMessageClient object (see SendLogMessage sequence diagram). This object allows us to avoid coupling LogMessageController directly to the external log message server, which is unstable relative to the system. LogMessageClient provides a stable interface to LogMessageController and increases cohesion in the controller as well. The indirection allows us to reuse the LogMessageClient code for future systems, and also reduces coupling with unstable components. The alternative to this approach would be to not have a LogMessageClient object. All of the remote server logic would be forced into LogMessageController, which would

decrease cohesion. Additionally, LogMessageController would be coupled with an unstable element of our design.