

CSCI-5448 SPRING 2016
OBJECT ORIENTED ANALYSIS AND DESIGN
PROJECT REPORT - PACKET SNIFFER

Team Members: Apoorva Bapat
Nehal Kamat
Sunil Baliganahalli Narayana Murthy

Description: A packet sniffer application using Java and having a GUI that can capture and analyze packets being transmitted and received over a network. The captured packets can be used to gain information about the type and content of messages being transmitted over the network.

1. What features were implemented?

The features of the packet analyzer are implemented from the user requirements are as follows:

User Requirements				
ID	Requirement	Topic Area	User	Priority
UR-001	User should be able to launch application.	Interaction	Any	High
UR-002	User should be able to close the application	Freedom	Any	Medium
UR-003	User should be able to start capturing packets by selecting network interface		Any	High
UR-004	User should be able to stop capturing packets		Any	High
UR-007	User should be able to import/export the saved Packets.		Any	High
UR-010	User should be able to filter packets according to detected protocols.		Any	Medium
UR-011	User should be able to inspect the packet information for a selected packet		Any	Medium
UR-012	User should be able to view only packet header.		Any	Medium

UR-013	User should have the option to view real time network statistics	Stats	Any	High
--------	------------------------------------------------------------------	-------	-----	------

2. Which features were not implemented from Part 2?

The features that did not make it to the prototype are:

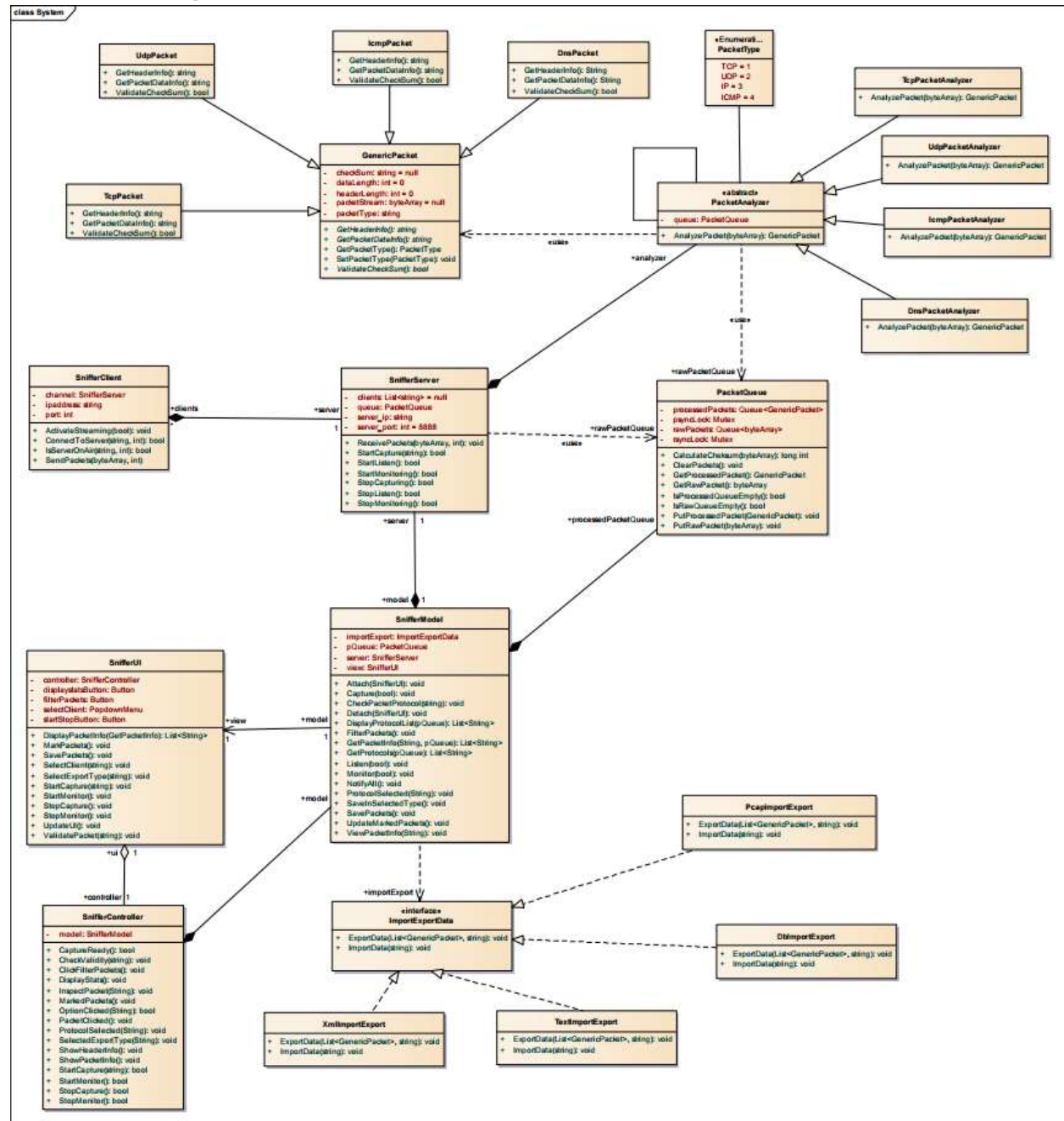
User Requirements				
ID	Requirement	Topic Area	User	Priority
UR-005	User should be able to mark packets for saving packet information.		Any	Medium
UR-006	User should be able to save either all the captured packets or marked captured packets		Any	High
UR-008	User should be able to view types of protocols used in captured packets.		Any	High
UR-009	Users should have the option of choosing the client machine to monitor packets from.	Freedom	Any	High
UR-014	User should be able to validate a selected packet for its integrity and authenticity	Validity	Any	High

The decision to leave the above features out of the prototype was based on our analysis of the complexity that would have been involved in the development process and the time constraint that we had to deal with.

Also, the above features are not *essential* to the functioning of a vanilla packet sniffer and the core functionality has been implemented.

3. Show your Part 2 class diagram and your final class diagram. What changed? Why? If it did not change much, then discuss how doing the design up front helped in the development.

Part 2 Class Diagram



```

classDiagram
    class TCPAnalyzer {
        + Analyze(byteArray): void
        + getProtocolName(): String<Array>
        + getValue(): Object
        + getValueNames(): String<Array>
        + IsAnalyzable(byteArray): bool
    }
    class HTTPAnalyzer {
        + Analyze(byteArray): void
        + getProtocolName(): String<Array>
        + getValue(): Object
        + getValueNames(): String<Array>
        + IsAnalyzable(byteArray): bool
    }
    class UDPAnalyzer {
        + Analyze(byteArray): void
        + getProtocolName(): String<Array>
        + getValue(): Object
        + getValueNames(): String<Array>
        + IsAnalyzable(byteArray): bool
    }
    class PacketAnalyzer {
        + Analyze(byteArray): void
        + getProtocolName(): String<Array>
        + getValue(): Object
        + getValueNames(): String<Array>
        + IsAnalyzable(byteArray): bool
    }
    class FrameView {
        + captor: PacketCaptor
        + captureButton: JButton
        + captureMenu: JMenu
        + exitMenu: JMenu
        + newWinMenu: JMenu
        + openButton: JButton
        + openMenu: JMenu
        + saveButton: JButton
        + saveMenu: JMenu
        + statMenu: JMenu
        + statusLabel: JLabel
        + stopButton: JButton
        + tablePane: TablePanelView
        + clear(): void
        + disableCapture(): void
        + enableCapture(): void
        + getImageIcon(): ImageIcon
        + loadProperty(): void
        + openNewWindow(PacketCaptor): FrameView
        + saveProperty(): void
        + startUpdating(): void
        + stopUpdating(): void
    }
    class PacketCaptor {
        - frame: FrameView
        - importExport: ImportExportData
        - queue: PacketQueue
        - server: SnifferServer
        + capturePacketsFromDevice(): void
        + clear(): void
        + closeAllWindows(): void
        + getPacketList(): List<Packet>
        + loadPacketsFromFile(): void
        + saveAll(): void
        + saveToFile(): void
        + setUIFrame(frame): void
        + startCaptureThread(): void
        + stopCapture(): void
        + stopCaptureThread(): void
    }
    class SnifferServer {
        - clients: List<string>
        - queue: PacketQueue
        - server_ip: string
        - server_port: int = 8888
        + ReceivePackets(byteArray, int): void
        + StartListen(): bool
        + StartMonitoring(): bool
        + StopCapturing(): bool
        + StopListen(): bool
        + StopMonitoring(): bool
    }
    class SnifferClient {
        - channel: SnifferServer
        - ipaddress: string
        - port: int
        + ActivateStreaming(bool): void
        + ConnectToServer(string, int): bool
        + IsServerOnAir(string, int): bool
        + SendPackets(byteArray, int)
    }
    class PacketQueue {
        + getPacket(): pkt
        + putPacket(): void
    }
    class ImportExportData {
        <interface>
        + ExportData(List<GenericPacket>, string): void
        + ImportData(string): void
    }
    class PcapImportExport {
        + ExportData(List<GenericPacket>, string): void
        + ImportData(string): void
    }
    class XmlImportExport {
        + ExportData(List<GenericPacket>, string): void
        + ImportData(string): void
    }
    class TextImportExport {
        + ExportData(List<GenericPacket>, string): void
        + ImportData(string): void
    }
    class FrameController {
        - captor: PacketCaptor
        - view: FrameView
        + actionPerformed(ActionEvent): void
        + saveProperty(): void
    }

    TCPAnalyzer --|> PacketAnalyzer
    HTTPAnalyzer --|> PacketAnalyzer
    UDPAnalyzer --|> PacketAnalyzer
    PacketAnalyzer --> FrameView : +view
    FrameView --> PacketCaptor : +model
    PacketCaptor --> SnifferServer : +server
    PacketCaptor --> PacketQueue : +rawPacketQueue
    PacketCaptor --> ImportExportData : +importExport
    ImportExportData <|.. PcapImportExport
    ImportExportData <|.. XmlImportExport
    ImportExportData <|.. TextImportExport
    FrameView --> FrameController : +controller
    SnifferServer --> SnifferClient : +clients
    
```

The class `GenericPacket` was removed and the functionality that it would have provided is now being handled by the `PacketAnalyzer` class. This is partly due to our discovery of the `JPCap` library that helps to fetch and handle packets that are being transmitted and processed on the network interface and partly due to us planning to use and apply the Chain of responsibility design pattern to deal with handling of packets more autonomously without extensive reliance on `JPCap`'s functionality. The `PacketAnalyzer` doesn't currently follow the Chain of responsibility

design pattern that we originally intended to code it around, but instead serves as a base class from which other specific packet analyzer classes are extended.

Our initial plan to include client-server functionality was also left out of the final version because it turned out that we would require a lot more time to integrate that feature into the system and given our time constraint, it did not seem to be feasible. Also, applying a design pattern to implement this feature might prove to be more efficient in the long run and hence would require some thorough analysis of the way we would need to structure it.

The PacketQueue class is a dummy class that will be used once the client server architecture is up and running. For now, all the incoming packets are handled by the jpcap library that processes the packet and extracts the packet information according to the packet type.

We, however, retained the use of the Model View Controller design pattern to implement the actual views that are displayed to the user and the model that works at the backend to provide the required functionality according to what actions the user performs. Designing the class diagram up-front helped us realize what sort of framework we'd want to follow and we could thus program our implementation around that framework. The functioning of the current prototype is built around the proposed MVC model and although the class names and certain attributes and operations were added and deleted, the design pattern itself was retained and plays the most important role in the functioning.

4. Did you make use of any design patterns in the implementation of your final prototype? If so, how? If not, where could you make use of design patterns in your system?

The following design patterns were used in implementation of the final prototype:

1. Model View Controller
2. Observer Pattern
3. Strategy Pattern

A. Model-View-Controller

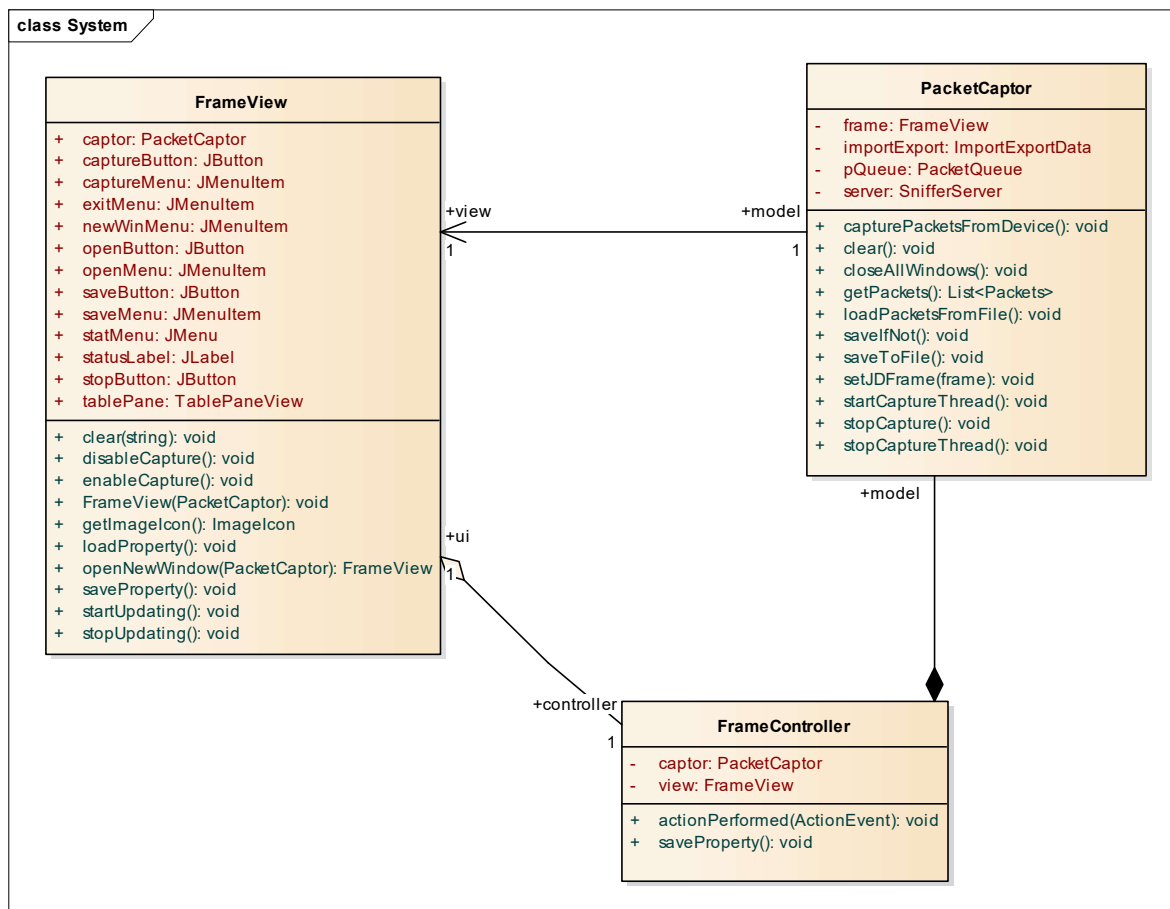
The PacketCaptor class, FrameView class and FrameController class form a model-view-controller architecture.

The PacketCaptor class is the model which contains the basic data obtained from the calling of function and has all the necessary functions that are required to be called according to a user's requirements.

The FrameView class serves as the View component that helps to render the window or frame of the window of the graphical user interface that the user will interact with (interaction such as inputting text, selecting options, etc.)

The FrameController class serves as the controller component of the MVC that helps to bind the data flow between the view and the model and to help keep the implementation of the 2 classes independent of each other.

The FrameController class relays a user action or input in the view to the model and the model calls a function on that action. The result is replicated in the view and any modifications made to the view are replicated in the model, through the controller class.



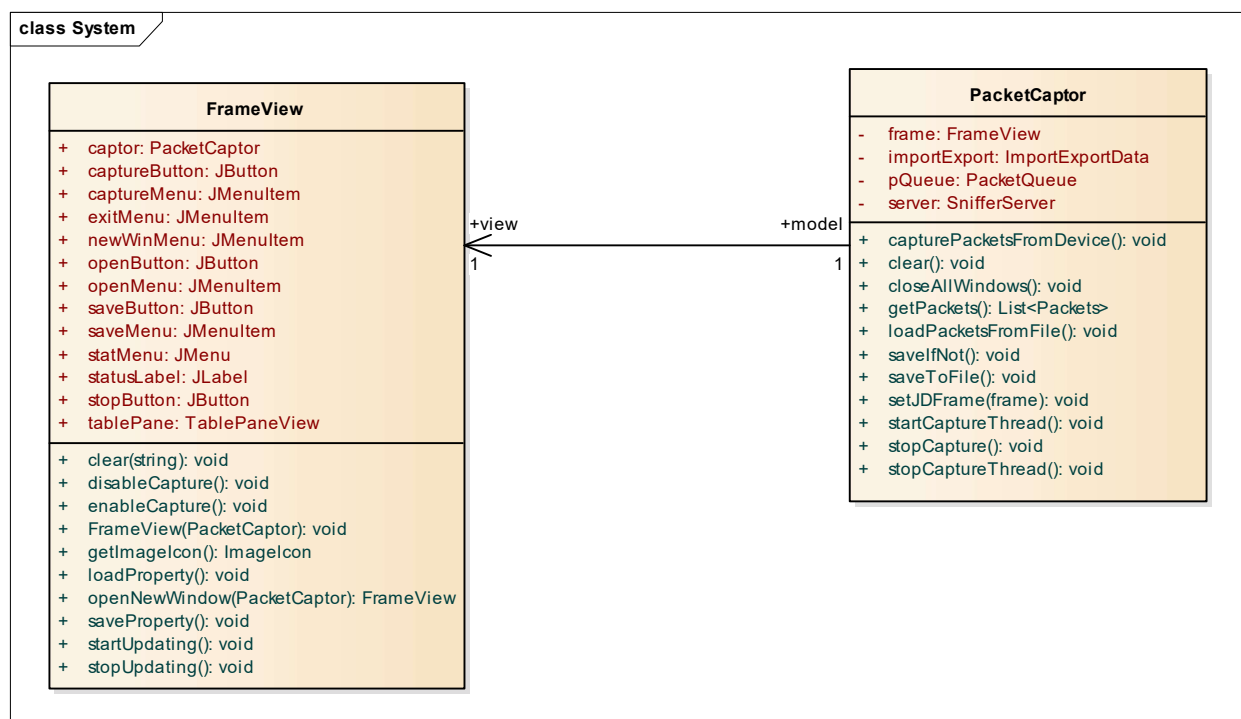
B. Observer Pattern

The Observer pattern is basically a pattern that signifies one class observing the behavior of another and reacting to some stimuli that the observed class has to offer.

The PacketCaptor class observes the FrameView class for any action that the user performs (interaction) with the view and takes corresponding action.

The FrameView class observes the value of the variable or return value of a function in the PacketCaptor class corresponding to the user action and takes on a value according to what the PacketCaptor has for that entity.

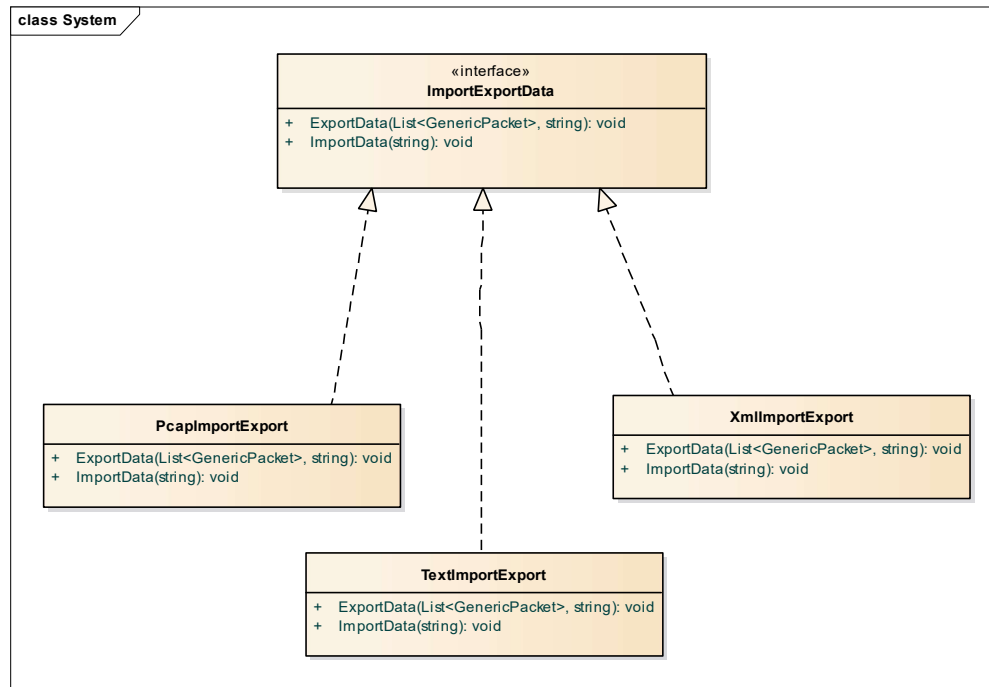
Essentially, both classes monitor each other and keep their connected data in sync (not through direct connection but with the help of a connector)



C. Strategy Pattern

The ImportExportData class helps to save information of the packets in the session to a log in the following 3 formats: Pcap, txt or XML.

The strategy pattern comes into play when loading a log of packet information having any of the 3 given extension choices into the program. The strategy pattern helps the function to resolve the extension properly and load the data in human-readable form.



What other design patterns can be used?

A Chain of responsibility design pattern can be used to handle the different packet analyzers. Each Analyzer could list the next in the list as its successor and so no packet would go unhandled as long as there exists a class to handle that type of packet.

5. What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system?

Analysis of a system in terms of its requirements and proposed design is extremely crucial at the beginning of any of any software design process as it helped us draft a rough sketch of what the system could look like and what features are important for the essential requirements of the system to function properly.

The nature and requirements of the system will change as the development progresses and so sufficient provision to accommodate this change dynamically should be programmed into the system's architecture.

It is important for all the team members to understand what each one is doing so as to work on his/her own part of the system keeping in mind the overall structure of how the system is being built, i.e., compatibility between modules

Implementation of a system is not a trivial task and implementing all of the use cases and requirements by a given deadline is an extremely difficult task keeping in mind the dynamic nature of the requirements and the process of development.

The architecture of the system reflects the designers' mindset and approach to the given system. Making use of Object-Oriented concepts in the design shows that the developers are interested in making a system keeping in mind the pitfalls of development but at the same time keeping it open for expansion or modification at later stages during the development.

The use of design patterns also simplifies the task at hand to a great extent and we think about the system in a more modular way than we would have otherwise. This helps to keep the code more extendable and avoids code smells.

6. Future Scope

- Implementing Client Server architecture
- Saving logs in more number of formats
- View more detailed analysis of packets
- Select which client to monitor packets from

---XXX---