CMP 403 SOFTWARE ENGINEERING
Group Assignment
GROUP MEMBERS:

1.	EJIGA ALEXANDER ACHILE.	BHU/20/04/05/0046
2.	OBENTA CHRISTOPHER OWOICHO.	BHU/20/04/05/0090
3.	IKEMKA ROMASON ROMANUS.	BHU/20/04/05/0016
4.	OGUH CHUBIYOJO FAVOUR.	BHU/20/04/05/0048
5.	KPONGO BEMSEN CHUKWUMA.	BHU/20/04/05/0115
6.	ADAMANI MIVANYI ENAN.	BHU/20/04/05/0044
7.	YOROM JOSHUA RIMAN	BHU/20/04/05/0014
8.	IDAMINABO CLEMENT-DAVIES	BHU/20/04/05/0054
9.	ADEJORI OLUWAFERANMI	BHU/20/04/05/0080

Summary and learning outcomes of chapter 6 & 7
Answers:
Chapter 6 – Architectural Design:
This chapter discusses: Architectural design decisions, Architectural views, Architectural patterns, and Application architectures.
Architectural design is the process for identifying the sub-systems making up a system and the framework for sub-system control and communication. It is an early stage of the system design process which represents the link between specification and design processes and it's often carried out in parallel with some specification activities. Architecture can be abstracted into two categories; Architecture in the small which is concerned with the architecture of individual programs, and architecture in the large which is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components.
Architectural representations are simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures. Architectural models can be used as a way of facilitating discussion about the system design and a way of documenting an architecture that has been designed. Architectural design is a creative process that involves, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.
Architectural views are perspectives which are useful when designing and documenting a system's architecture It shows how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network.
The architectural views involve:
A logical view, (key abstractions in the system as objects or object classes.),
A process view, (which shows how, at run-time, the system is composed of interacting processes.)
A development view, (shows how the software is decomposed for development.)
A physical view(shows the system hardware and how software components are distributed across the processors in the system.)

Architectural patterns are a means of representing, sharing and reusing knowledge, it is a stylized description of good design practice, which has been tried and tested in different environments.

The MVC (Model-View-Controller) pattern:

Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.

The Layered architecture pattern:

Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.

The Repository pattern:

All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.

The Client-server pattern:

In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server.

The Pipe and filter pattern:

he processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.

Application architecture is an architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements. It is used as a starting point for architectural design, a design checklist., away of organising the work of the development team and a means of assessing components for reuse.

Examples of Application types are :

Data processing applications:Data driven applications that process data in batches without explicit user intervention during the processing.

Transaction processing applications:Data-centred applications that process user requests and update information in a system database.

Event processing systems:Applications where system actions depend on interpreting events from the system’s environment.

Language processing systems: Applications where the users’ intentions are specified in a formal language that is processed and interpreted by the system.

Chapter 7 – Design and Implementation:

This chapter discusses: object-oriented design using the UML, design patterns implementation issues and open source development

Object-oriented design processes involve developing a number of different system models.Common activities in these processes include:
Define the context and modes of use of the system,
Design the system architecture,
Identify the principal system objects,
Develop design models,
Specify object interfaces.

Understanding the relationships between the software that is being designed is essential for deciding how to provide system functionality, structure the system to communicate with its environment and establish the boundaries of the system.
A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
An interaction model is a dynamic model that shows how the system interacts with its environment as it is used. Once interactions between the system and its environment have been understood, the information can be used for designing the system architecture.
The approach to identifying objects in a system include using a grammatical approach based on a natural language description of the system, using a behavioral approach and identify objects based on what participates in what behaviour, and using a scenario-based analysis where the objects, attributes and methods in each scenario are identified.
Design models show the objects and object classes and relationships between these entities. These models can either be static or dynamic. Static models describe the static structure of the system in terms of object classes and relationships. Dynamic models describe the dynamic interactions between objects. Examples of design models are subsystem models, sequence models, state machine models and other models such as use-case models, aggregation models, etc.
Subsystem models that show logical groupings of objects into coherent subsystems.
Sequence models that show the sequence of object interactions.
State machine models that show how individual objects change their state in response to events.
Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.

A Design pattern is a way of reusing abstract knowledge about a problem and its solution. Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism. Pattern elements include: Name of problem, problem description, solution description, consequences, the results and trade-offs of applying the pattern.

Observation patterns include:
Description: which Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description: which provides multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.
Solution description: This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.
Consequences: The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Implementation issues may arise from:
Reuse Most modern software is constructed by reusing existing components or systems.
Configuration management: keeping track of the many different versions of each software component in a configuration management system.
Host-target development: Production software does not usually execute on the same computer as the software development environment.

Reuse levels include:
The abstraction level: using knowledge of successful abstractions in the design of your software.
The object level: reuse objects from a library rather than writing the code yourself.
The component level: reuse of components are collections of objects and object classes in application systems.
The system level: reuse entire application systems.
The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system. Configuration management processes involve: version management, system integration,and problem tracking.
Host target development involves considering development platform usually have different installed software than execution platform; these platforms may have different architectures.

Development platform tools.
These include:
An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
A language debugging system.
Graphical editing tools, such as tools to edit UML models.
Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.
Project support tools that help you organize the code for different development projects.
Integrated development environment (IDE). An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.

Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process. Open source issues include questions such as Should the product that is being developed make use of open source components? Should an open source approach be used for the software's development?
More and more product companies are using an open source approach to development.
A fundamental principle of open-source development is that source code should be freely available, but this does not mean that anyone can do as they wish with that code.
Legally, the developer of the code still owns the code. They can place restrictions on how it is used.

License models include:
The GNU General Public License (GPL). This means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code.

License management are rules for maintaining information about open-source components that are downloaded and used. They include:
Being aware of the different types of licenses and understand how a component is licensed before it is used. Being aware of evolution pathways for components.