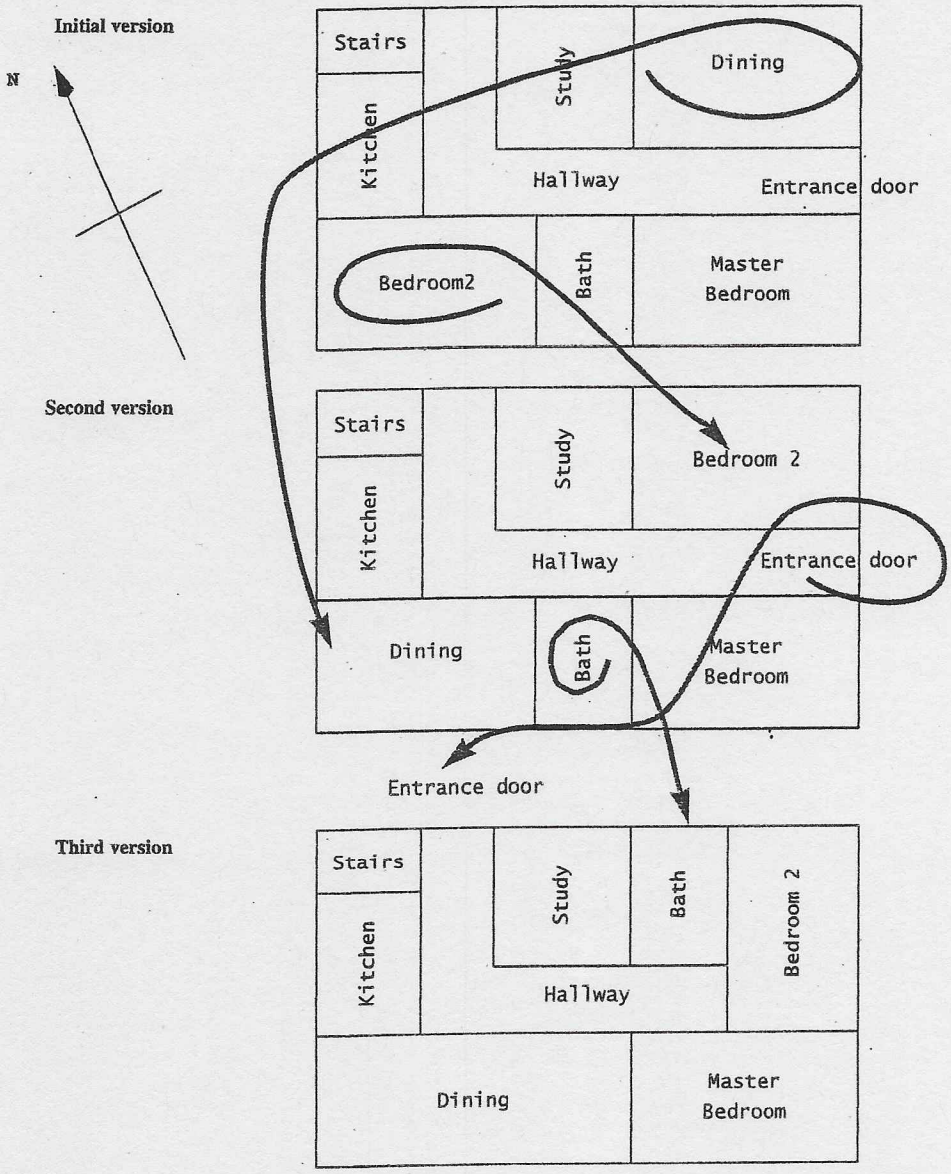


Figure 6-1 Example of floor plan design. Three successive versions show how we minimize walking distance and take advantage of sunlight.

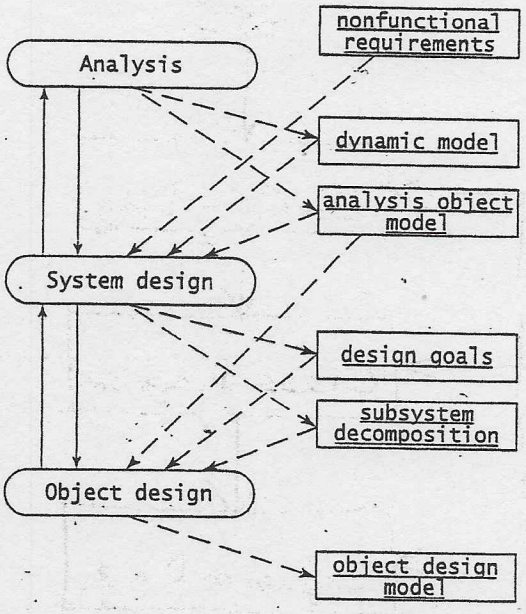


Figure 6-2 The activities of system design (UML activity diagram).

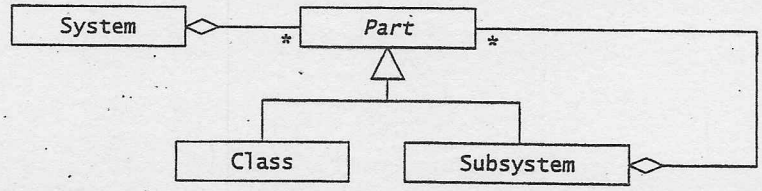


Figure 6-3 Subsystem decomposition (UML class diagram).

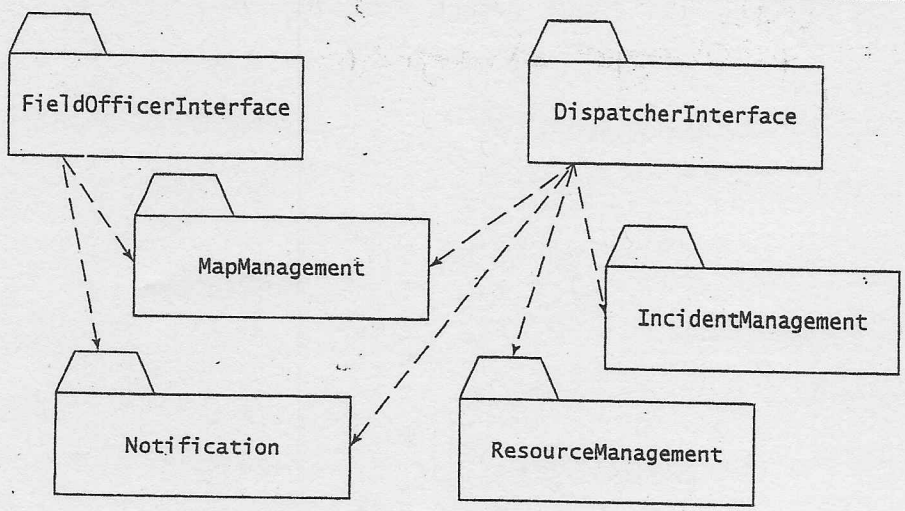
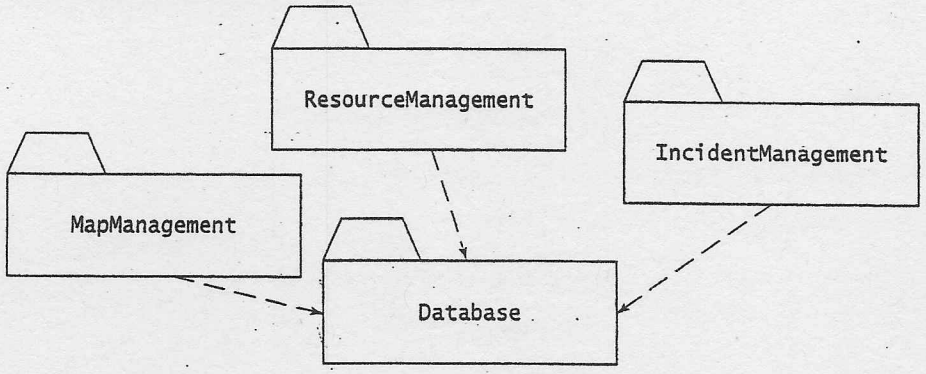


Figure 6-4 Subsystem decomposition for an accident management system (UML class diagram, collapsed view). Subsystems are shown as UML packages. Dashed arrows indicate dependencies between subsystems.

Alternative 1: Direct access to the Database subsystem



Alternative 2: Indirect access to the Database through a Storage subsystem

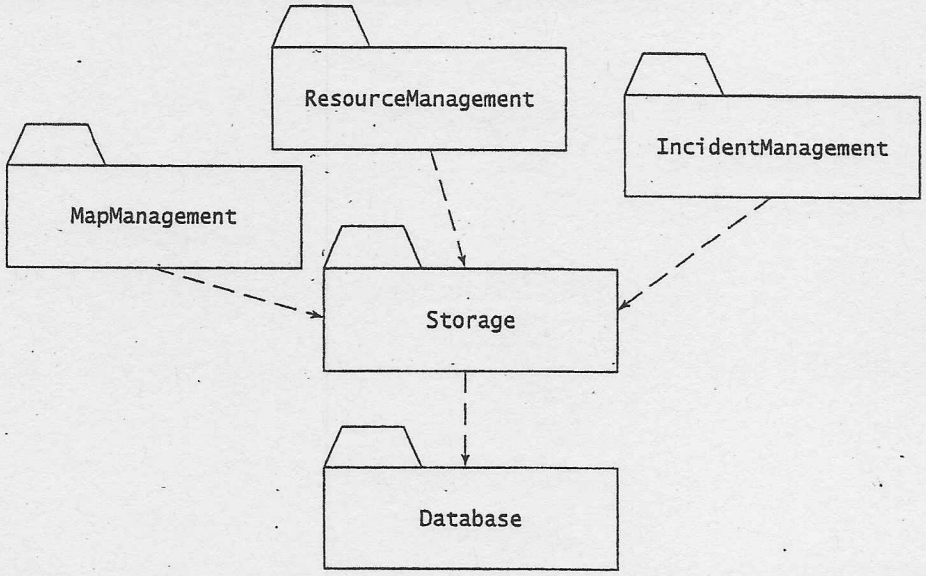
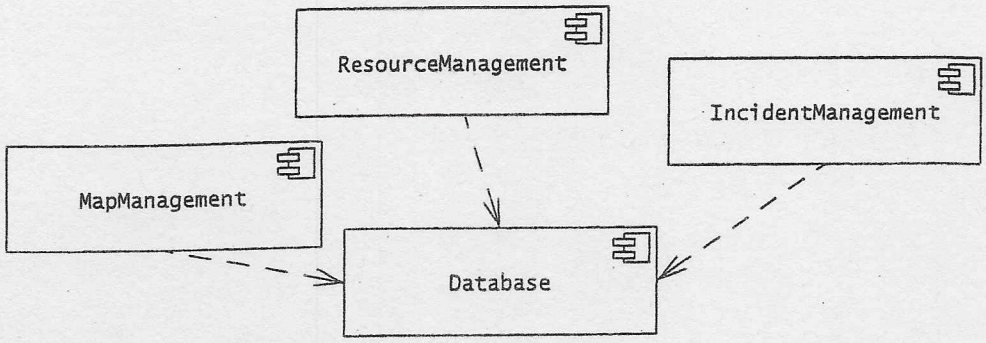


Figure 6-5 Example of reducing the coupling of subsystems (UML class diagram, subsystems `FieldOfficerInterface`, `DispatcherInterface`, and `Notification` omitted for clarity). Alternative 1 depicts a situation where all subsystems access the database directly, making them vulnerable to changes in the interface of the `Database` subsystem. Alternative 2 shields the database with an additional subsystem (`Storage`). In this situation, only one subsystem will need to change if there are changes in the interface of the Database subsystem. The assumption behind this design change is that the `Storage` subsystem has a more stable interface than the `Database` subsystem.

Alternative 1: Direct access to the Database subsystem



Alternative 2: Indirect access to the Database through a Storage subsystem

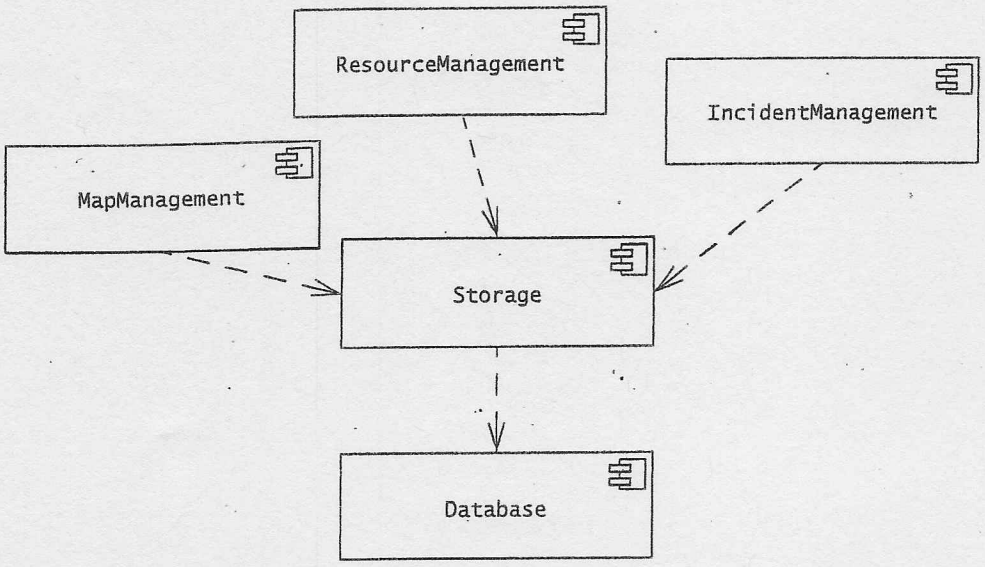


Figure 6-6 Example of reducing the coupling of subsystems (UML component diagram, subsystems `FieldOfficerInterface`, `DispatcherInterface`, and `Notification` omitted for clarity). Alternative 1 depicts a situation where all subsystems access the database directly, making them vulnerable to changes in the interface of the `Database` subsystem. Alternative 2 shields the database with an additional subsystem (`Storage`). In this situation, only one subsystem will need to change if there are changes in the interface of the `Database` subsystem. The assumption behind this design change is that the `Storage` subsystem has a more stable interface than the `Database` subsystem.

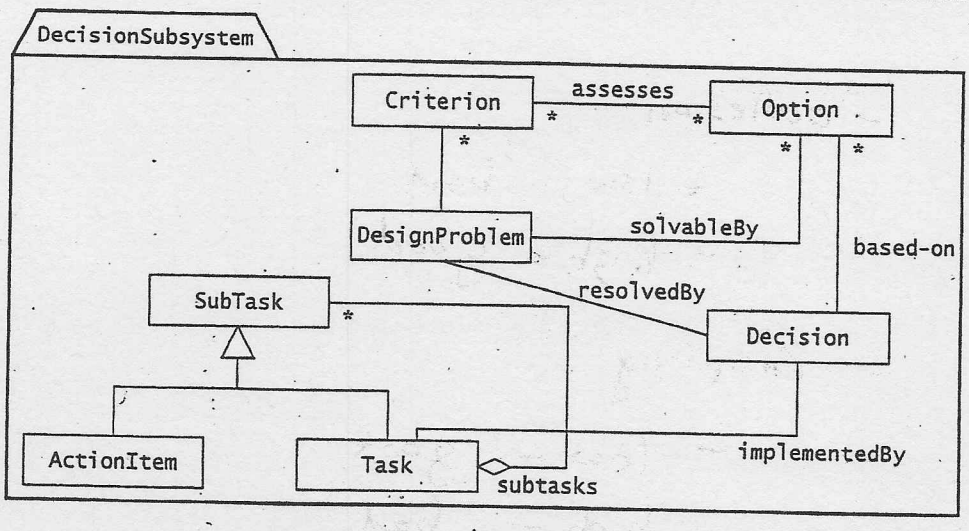


Figure 6-6 Decision tracking system (UML class diagram). The `DecisionSubsystem` has a low cohesion: The classes `Criterion`, `Option`, and `DesignProblem` have no relationships with `Subtask`, `ActionItem`, and `Task`.

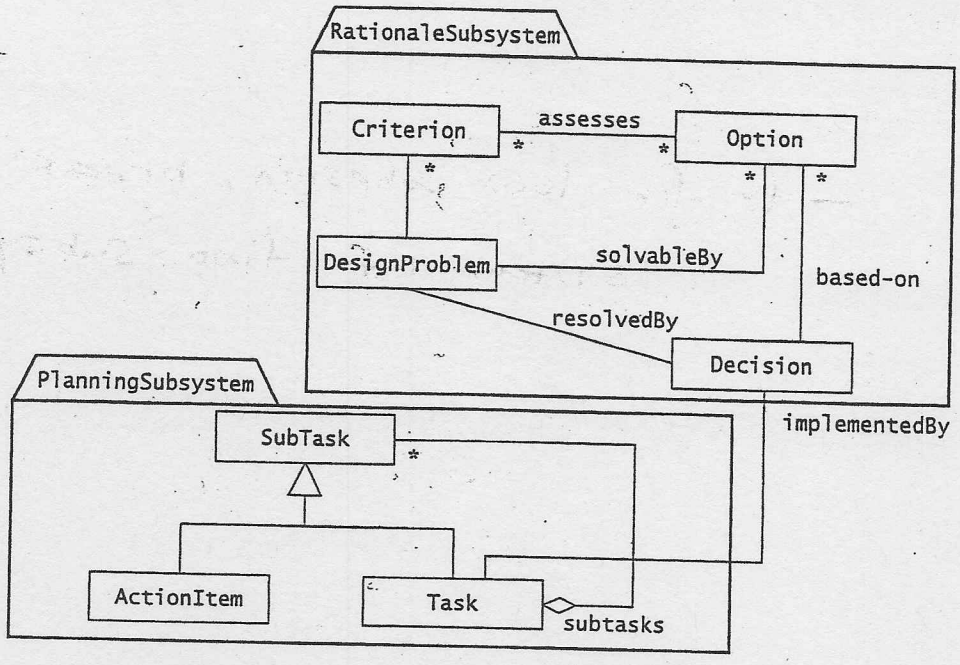


Figure 6-7 Alternative subsystem decomposition for the decision tracking system of Figure 6-6 (UML class diagram). The cohesion of the `RationaleSubsystem` and the `PlanningSubsystem` is higher than the cohesion of the original `DecisionSubsystem`. Note also that we also reduced the complexity by decomposing the system into smaller subsystems.

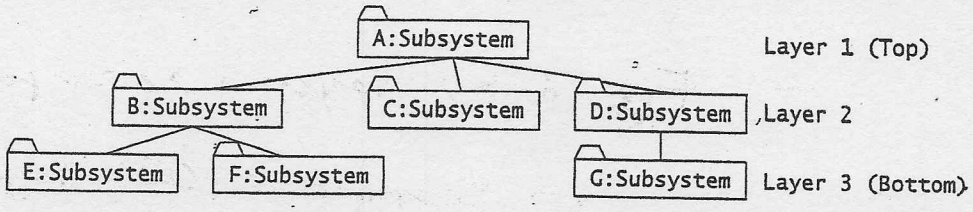


Figure 6-8 Subsystem decomposition of a system into three layers (UML object diagram). A subset from a layered decomposition that includes at least one subsystem from each layer is called a vertical slice. For example, the subsystems A, B, and E constitute a vertical slice, whereas the subsystems D and G do not.

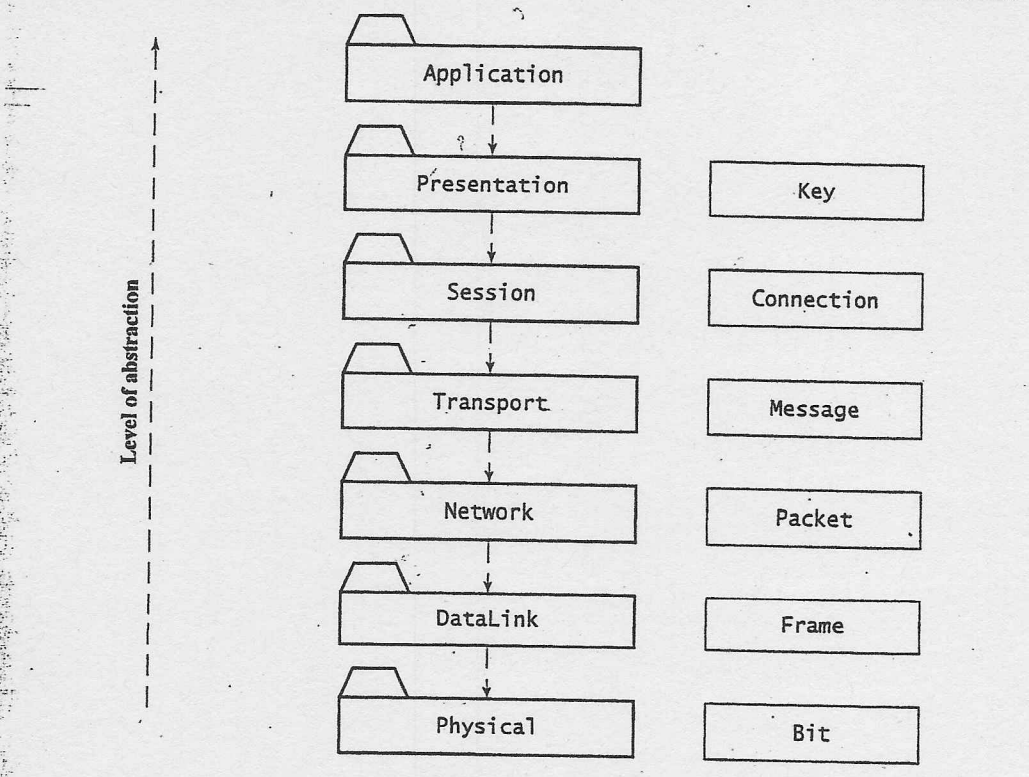


Figure 6-9 An example of closed architecture: the OSI model (UML class diagram). The OSI model decomposes network services into seven layers, each responsible for a different level of abstraction.

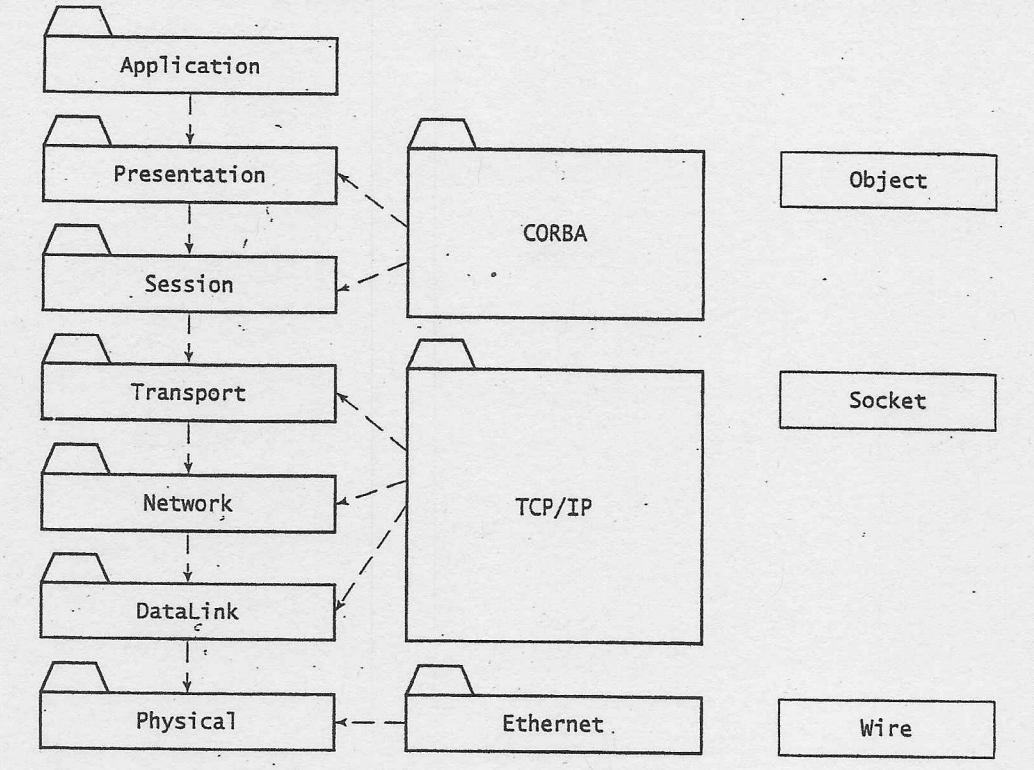


Figure 6-10 An example of closed architecture (UML class diagram). CORBA enables the access of objects implemented in different languages on different hosts. CORBA effectively implements the Presentation and Session layers of the OSI stack.

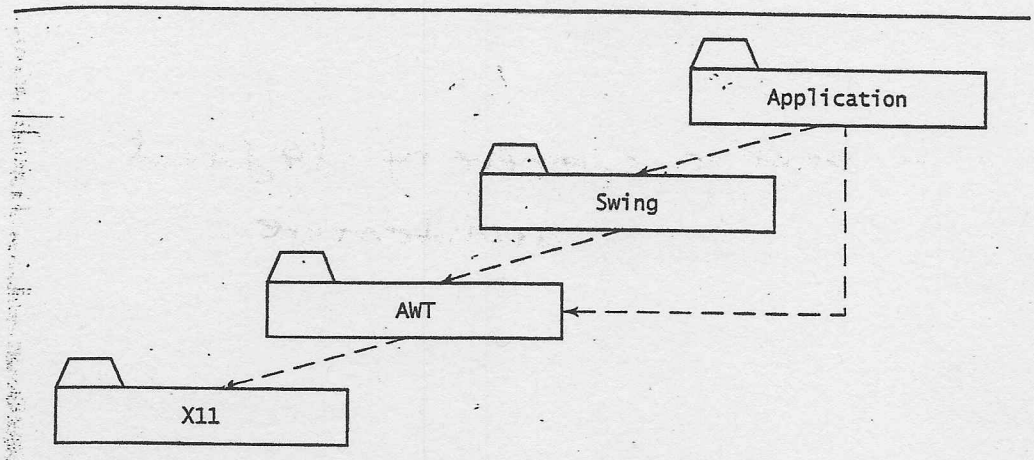


Figure 6-11 An example of open architecture: the `Swing` user interface library on an `X11` platform (UML class diagram, packages collapsed). `X11` provides low-level drawing facilities. `AWT` is the low-level interface provided by Java to shield programmers from the window system. `Swing` provides a large number of sophisticated user interface objects. Some `Applications` often bypass the `Swing` layer.

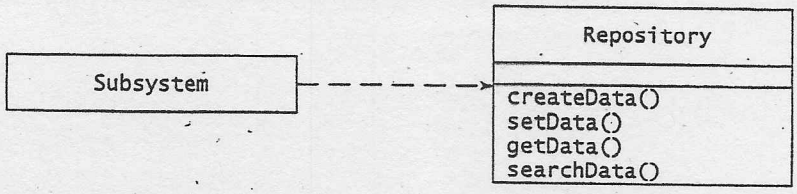


Figure 6-12 Repository architectural style (UML class diagram). Every Subsystem depends only on a central data structure called the Repository. The Repository has no knowledge of the other Subsystems.

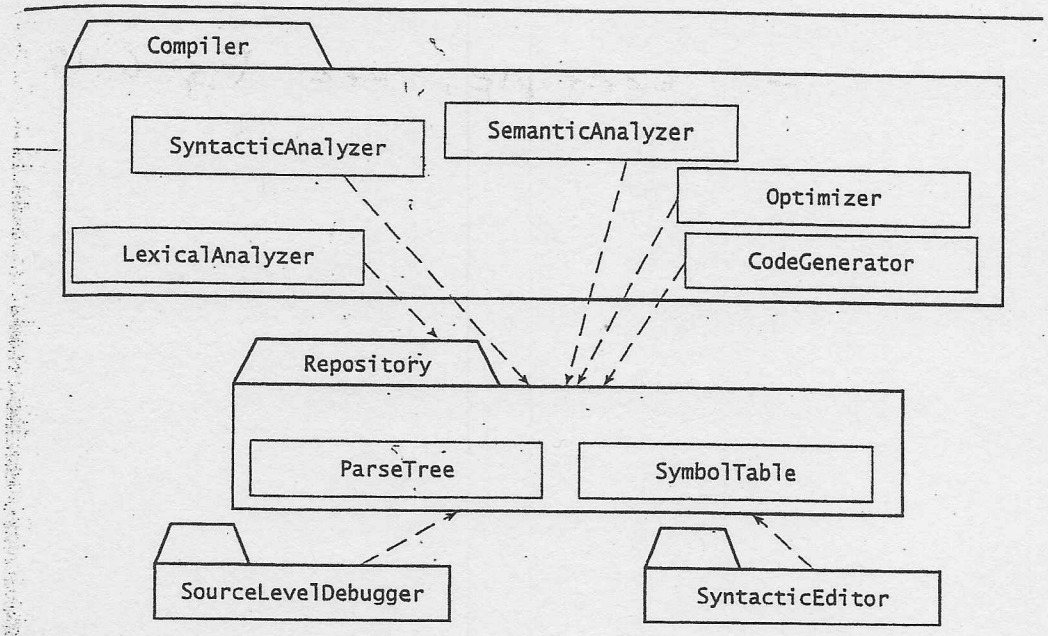


Figure 6-13 An instance of the repository architectural style (UML class diagram). A Compiler incrementally generates a ParseTree and a SymbolTable that can be used by SourceLevelDebuggers and SyntaxEditors.

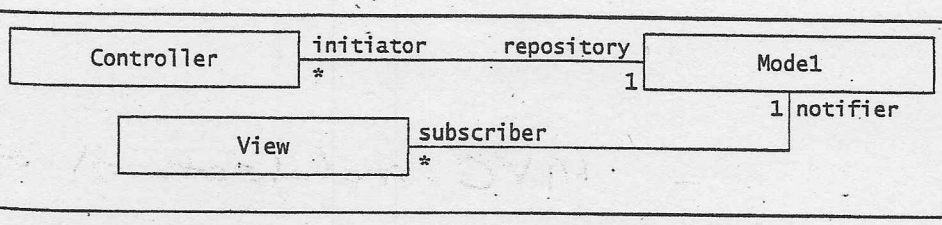


Figure 6-14 Model/View/Controller architectural style (UML class diagram). The `Controller` gathers input from the user and sends messages to the `Model`. The `Model` maintains the central data structure. The `Views` display the `Model` and are notified (via a subscribe/notify protocol) whenever the `Model` is changed.

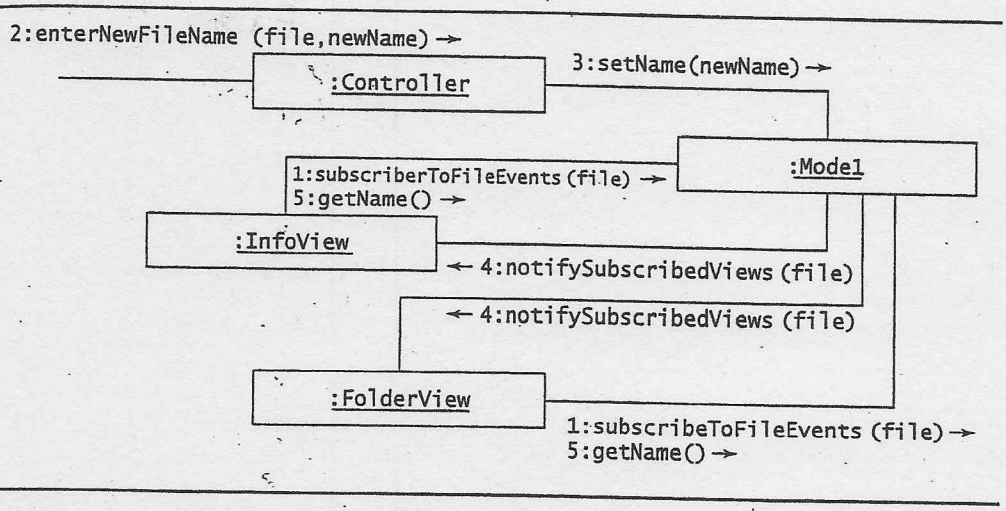


Figure 6-16 Sequence of events in the Model/View/Control architectural style (UML collaboration diagram).

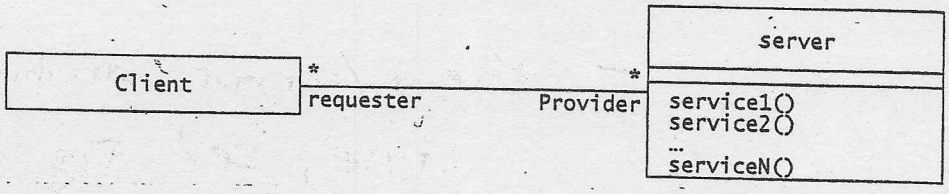


Figure 6-17 Client/server architectural style (UML class diagram). `Clients` request services from one or more `Servers`. The `Server` has no knowledge of the `Client`. The client/server architectural style is a specialization of the repository architectural style.

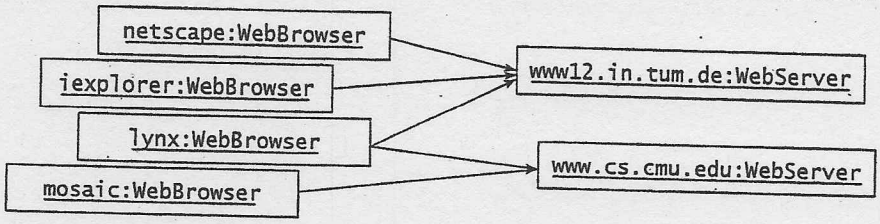


Figure 6-18 The Web as an instance of the client/server architectural style (UML object diagram).

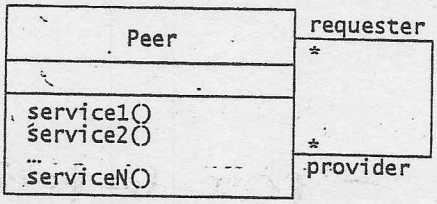


Figure 6-19 Peer-to-peer architectural style (UML class diagram). Peers can request services from and provide services to other peers.

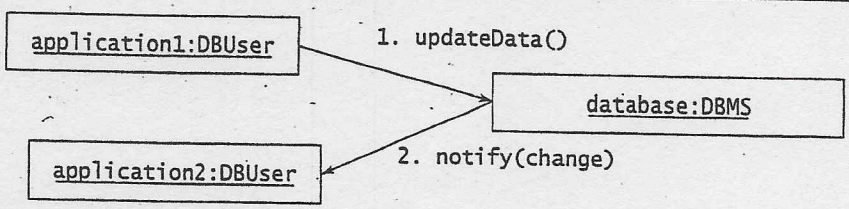


Figure 6-20 An example of peer-to-peer architectural style (UML collaboration diagram). The database server can both process requests from and send notifications to applications.

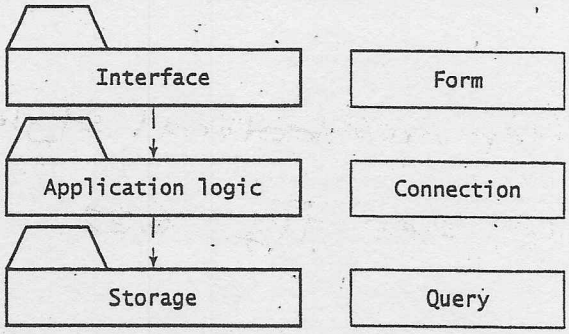


Figure 6-21 Three-tier architectural style (UML class diagram). Objects are organized into three layers realizing the user interface, the processing, and the storage.

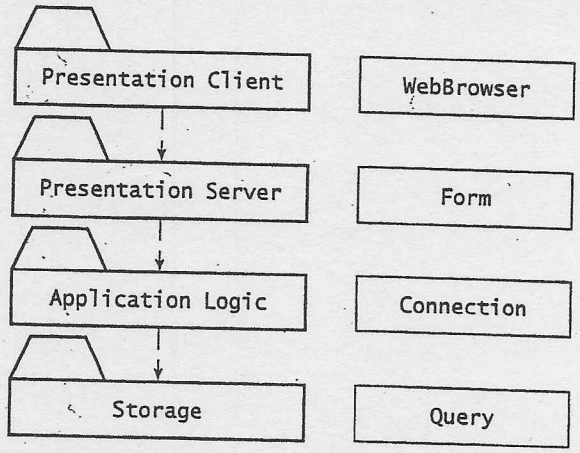


Figure 6-22 Four-tier architectural style (UML class diagram). The `Interface` layer of the three-tier style is split into two layers to enable more variability on the user interface style.

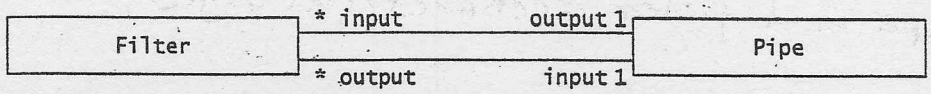


Figure 6-23 Pipe and filter architectural style (UML class diagram). A `Filter` can have many inputs and outputs. A `Pipe` connects one of the outputs of a `Filter` to one of the inputs of another `Filter`.