Software Design Process:

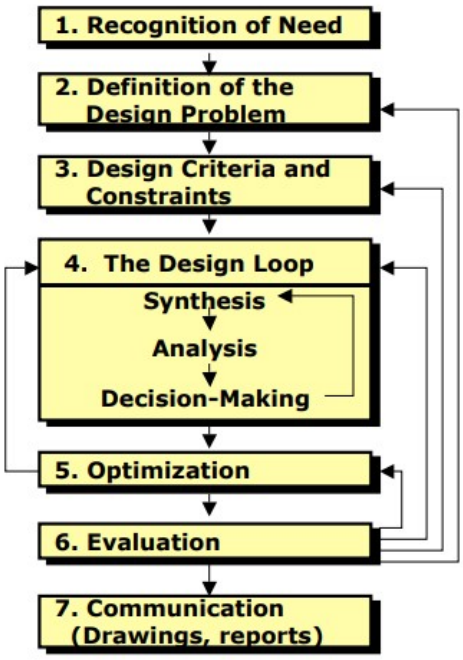
An engineering design is a model of the product or structure to be engineered. The model is used to

• Evaluate suitability of proposed product/system.

• Communicate proposed product to others.

An engineering design process describes a set of steps for constructing an engineering design.

Engineering Design Process:



Andrews, Aplevich, Fraswer, Ratz, *Introduction to Professional Engineering in Canada,* Pearson, 2002.

Software Design Principles:

Goals of Software Design:

Software design is primarily about managing complexity. Software systems are often very complex and have many moving parts. Most systems must support dozens of features simultaneously. Each feature by itself might not seem very complicated. However, when faced with the task of creating one coherent structure that supports all of the required functionality at once, things become complicated very quickly. Human capacity to deal with complexity is quite limited; people become overwhelmed and confused relatively quickly. Perhaps the primary objective of software design is to make and keep software systems well organized, thus enhancing our ability to understand, explain, modify, and fix them. Based on this view of software design, disorganization (or sloppiness) is the antithesis of good software design.

As the laws of physics teach us, the universe tends to become more disorganized over time unless we take active steps to make and keep it organized. Software systems are very much the same way. If created or modified without careful forethought, software systems quickly become incomprehensible, tangled messes that don't work right and are impossible to fix. This is especially true for systems that remain in use over extended periods of time, and are periodically upgraded to support new features. Even if a system starts out with a good design, we must consistently strive to preserve the integrity of its design throughout its lifetime by carefully considering all changes we make to it. Based on these principles, we can list several important goals of software design:

- Software that works
- Software that is easy to read and understand
- Software that is easy to debug and maintain
- Software that is easy to extend and holds up well under changes
- Software that is reusable in other projects

Design as an Iterative Process

Software design is a complex undertaking. Therefore, you will rarely get a design right the first time. Implementing a design provides new insights into its deficiencies: things you didn't think about, better ways of doing things, etc. Such insights should feed back into your design to make it better. For this reason, design and implementation activities are usually interleaved in short iterations: Design, code, test, debug, Design, code, test, debug.

The notion that a complex system can be completely designed in every detail before implementation begins is fallacious. Such an approach deprives designers of valuable knowledge and experience that come only from actually implementing the design. The opposite extreme is also dangerous, starting implementation having done little or no design at all. Those who start coding immediately and wing it as they go are even more prone to failure than those who try to design everything up front. The truth lies between these two extremes. You should do enough design to have a fairly detailed idea of how things will work, and then implement the design to discover its deficiencies. Then, go back and incorporate what you've learned into the design, and then implement some more. This process will eventually converge on a good design.

Abstraction

Abstraction is one of the software designer's primary tools for coping with complexity. Most programming languages and their associated libraries are meant to be general purpose. They can be used to implement solutions to problems in any application domain (finance, retail, biology, communications, etc.). Due to their general purpose nature, these languages provide only low-level abstractions such as bit, byte, character, string, integer, float, array, file, etc. that model the machines on which the software will run rather than the application domain of the problem being solved. Programs written solely in terms of these low-level abstractions are extremely difficult to understand. Effective software design requires the creation of new, higher-level abstractions that map directly to the

application domain rather than the underlying computer. In object-oriented design, application-specific abstractions are represented as classes. Classes encapsulate the state (or data) and operations (or algorithms) associated with a particular higher-level application concept. For example, the design for a word processor would contain classes such as Document, Font, Table, Figure, and Printer. Similarly, the design for a web browser would contain classes such as Favorites, URL, Viewer, and NetworkProtocol.

Naming

Abstraction involves taking something that is complicated, giving it a simple name, and then referring to it by its simple name. This way, complex ideas can be conveyed very concisely. With this in mind, one of the most important tools for achieving effective abstraction is the identifier. An identifier is a name that we assign to something. We choose names for classes, methods, variables, constants, source files, etc. While selecting a name might seem to be a relatively inconsequential thing, it is not. The names we choose for things go a long way toward determining how readable our code becomes. Even if I create the right class, if I name it poorly, much of the benefit to be gained from abstraction has been lost. For example, if I name the class that represents printers as Thingy instead of Printer, I have done significant harm to the readability of my design. The name assigned to a class, variable, or method should clearly and accurately reflect the function performed by that class, variable, or method. The name Printer implies that a class represents a printer; the name calculatePayrollTax implies that a method calculates payroll taxes; the name homeAddress implies that a variable stores a home address. In contrast, the names Thingy, doStuff, and info would convey no information whatsoever to the reader. Name selection makes a huge difference.

Classes and operations should be highly cohesive. Each class should represent one well-defined concept, and should be given a name that clearly reflects the concept it represents (e.g., URL). Cohesive classes are almost always easy to name. In fact, the name they should be given is often obvious, because they represent only one concept. The operations on a class should all be highly-related to the concept represented by the class. For example, URL operations should all be highly related to storing and manipulating URLs. Operations like getPath, getFileName, and resolveRelative would be appropriate. Operations that are loosely related or unrelated to the concept represented by the class should be placed on some other class. For example, a URL class should not have a display method that renders the document referenced by the URL on the screen. The rendering function is only loosely related to the concept of a URL, and so should be placed on a different class (e.g., FileViewer).

Class operations should also be highly cohesive. Each operation should perform one well-defined task, and should be given a name that clearly reflects the task it performs (e.g., rebootComputer). Cohesive operations are almost always easy to name, because they do only one thing. If a method does a bunch of loosely related or unrelated things, it will either be hard to find a good name that describes what the operation does, leading to inferior names like handleStuff, or the method's name will become too long (e.g., sweepFloorAndDoDishesAndPayBills).

Decomposition

In addition to abstraction, another fundamental technique for dealing with complexity is taking the original problem and dividing it into several smaller sub-problems. The subproblems are smaller and hence less complex than the original, thus making them more approachable. After solving each sub-problem individually, the solutions to the subproblems can be combined to create a solution to the original, larger problem. This approach is frequently called “divide and conquer”. After breaking the original problem into sub-problems, we may find that the subproblems themselves are still too complex to solve directly. In this case, we decompose the sub-problems yet again to create second-level sub-problems that are even simpler. Sub-problems are divided into smaller and smaller parts until the smallest sub-problems are simple enough to solve directly, and thus require no further subdivision. In effect, we create a tree of problems, where the original problem is at the root, and each successive level of subdivision adds another level of nodes to the tree. The solution to each subproblem makes use of the solutions to the sub-problems below it. This approach allows us to cope with the inherent complexity of the original problem in bite-size chunks. Decomposition is strongly related to abstraction. The solution to each sub-problem is abstracted as a class or method. The solution to the larger problem invokes the abstractions which encapsulate the sub-problem solutions. This results in a concise solution to the original problem, and allows the details of the sub-problem solutions to be temporarily ignored, thus reducing the cognitive burden of solving the original problem.

Software Modeling

Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of object-oriented software engineering. UML includes a set of graphic notation techniques to create visual models of object-oriented software systems. UML combines techniques from data modeling, business modeling, object modeling, and component modeling and can be used throughout the software development life-cycle and across different implementation technologies.

There is a difference between a UML model and the set of diagrams of a system. A diagram is a partial graphic representation of a system’s model. The model also contains documentation that drives the model elements and diagrams (such as written use cases). UML diagrams represent two different views of a system model:

Static (or structural) view

This view emphasizes the static structure of the system using objects, attributes, operations, and relationships. Ex: Class diagram, Composite Structure diagram.

Dynamic (or behavioral) view

This view emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. Ex: Sequence diagram, Activity diagram, State Machine diagram.

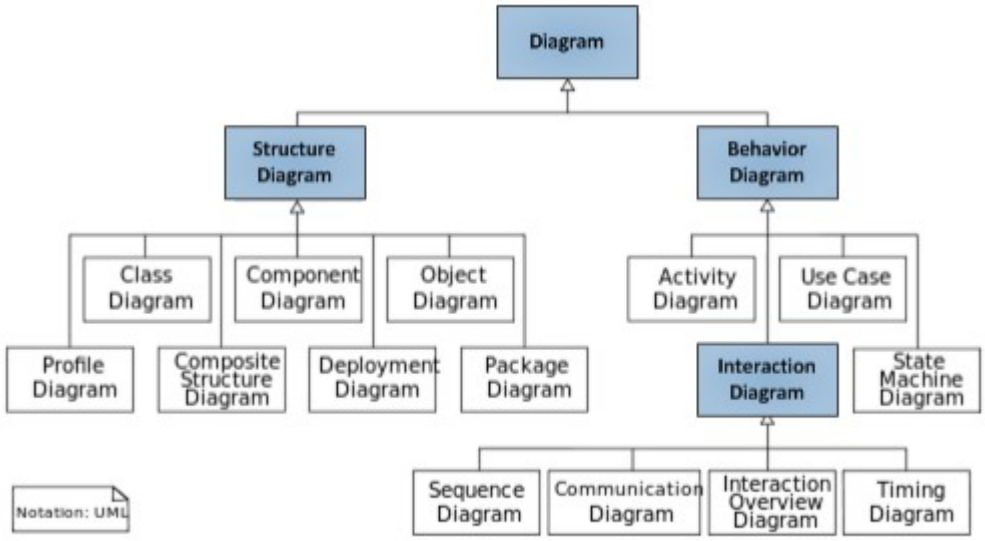


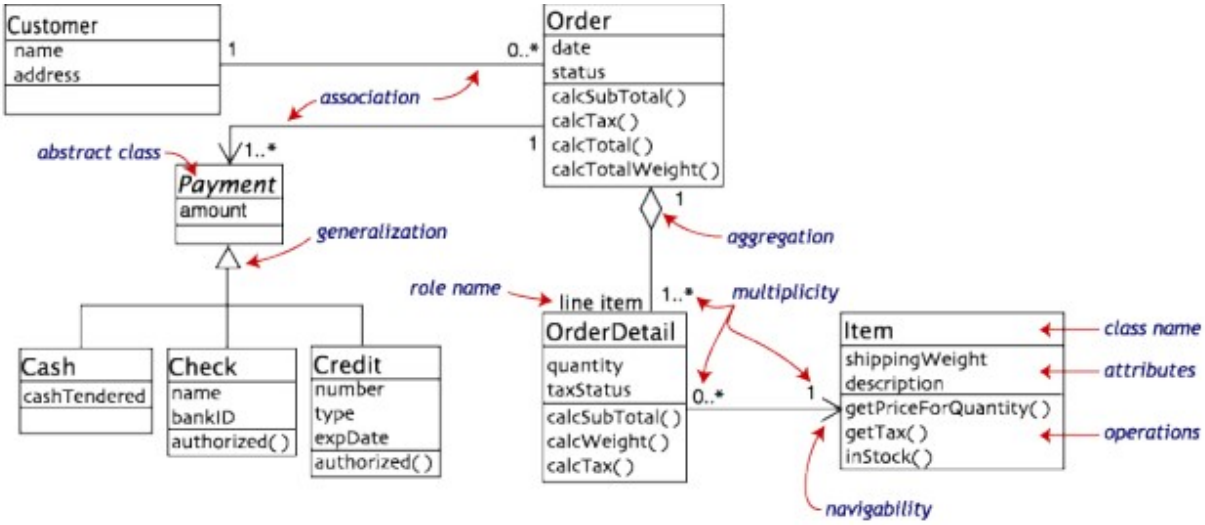
Figure 1Types of UML Diagram

Structure Diagrams

These diagrams emphasize the things that must be present in the system being modeled. Since they represent the structure, they are used extensively in documenting the software architecture of software systems.

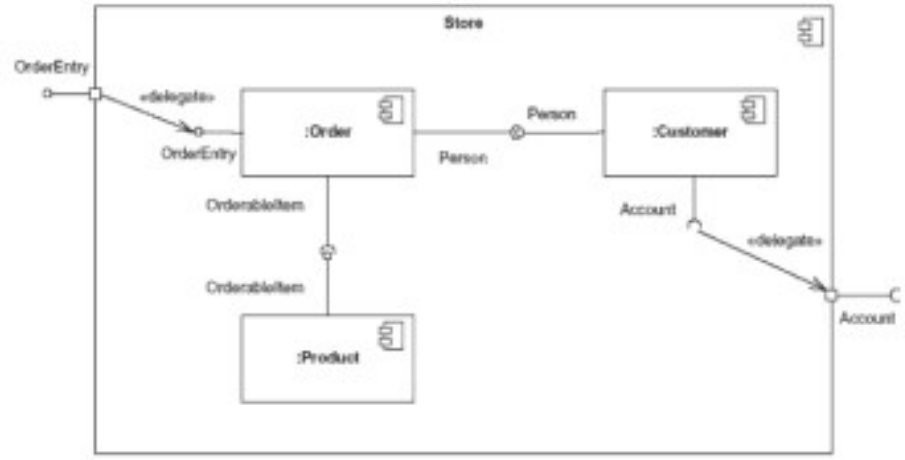
- Class Diagram

Describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.



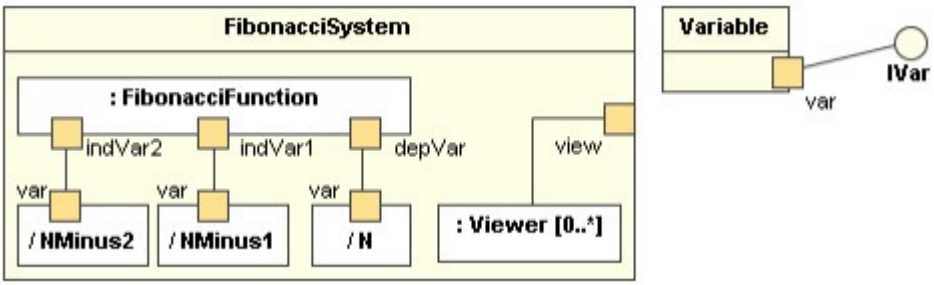
- Component Diagram

Describes how a software system is split-up into components and shows the dependencies among these components.



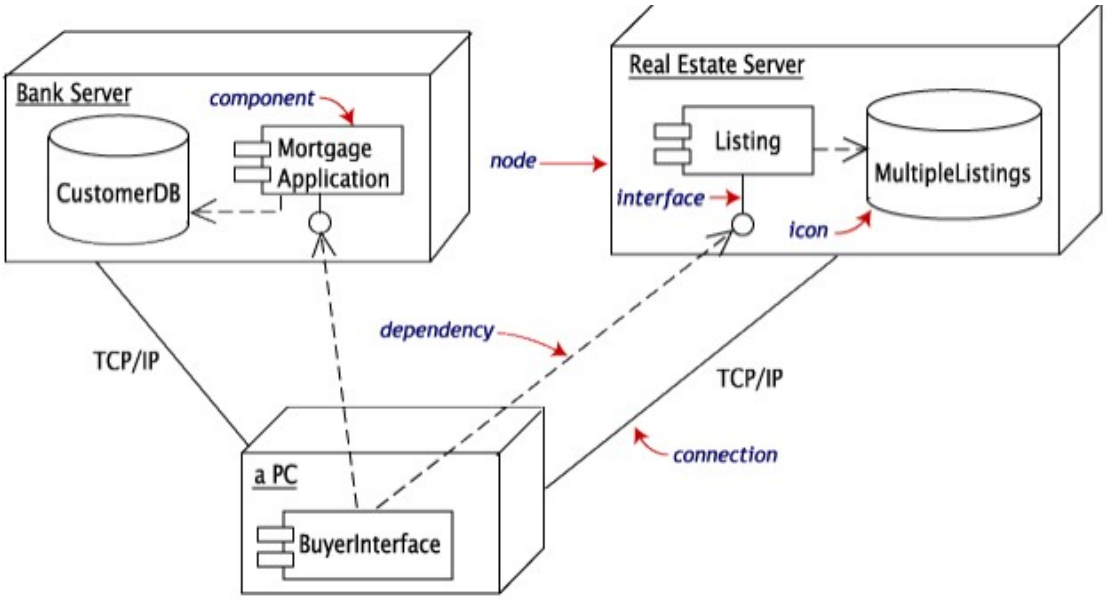
- Composite Structure Diagram

Describes the internal structure of a class and the collaborations that this structure makes possible.



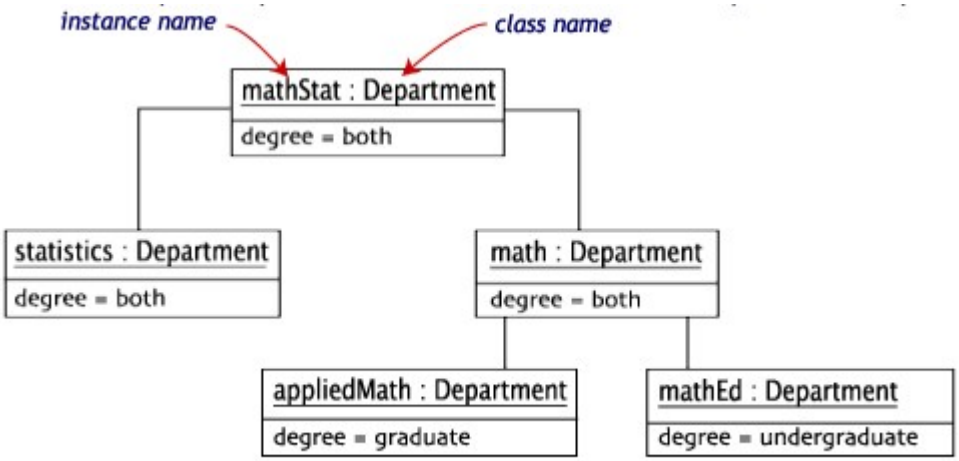
- Deployment Diagram

Describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.



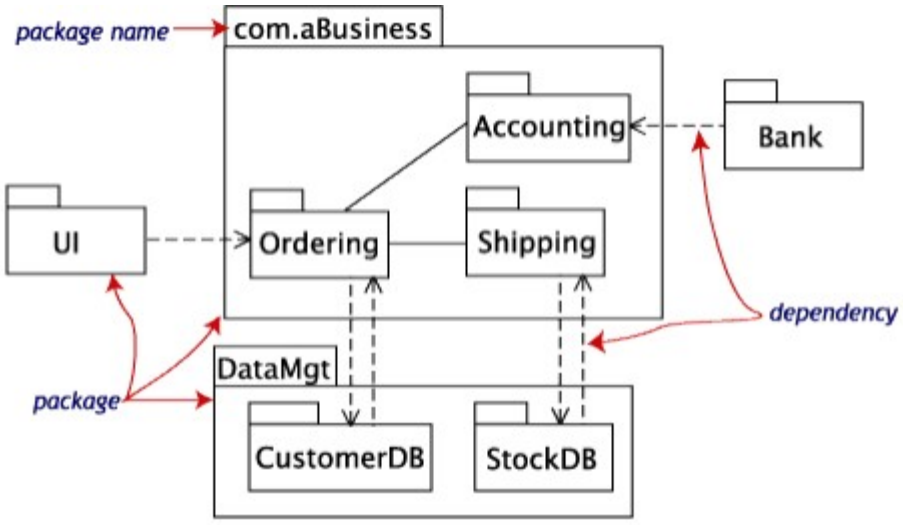
- Object Diagram

Shows a complete or partial view of the structure of an example modeled system at a specific time.



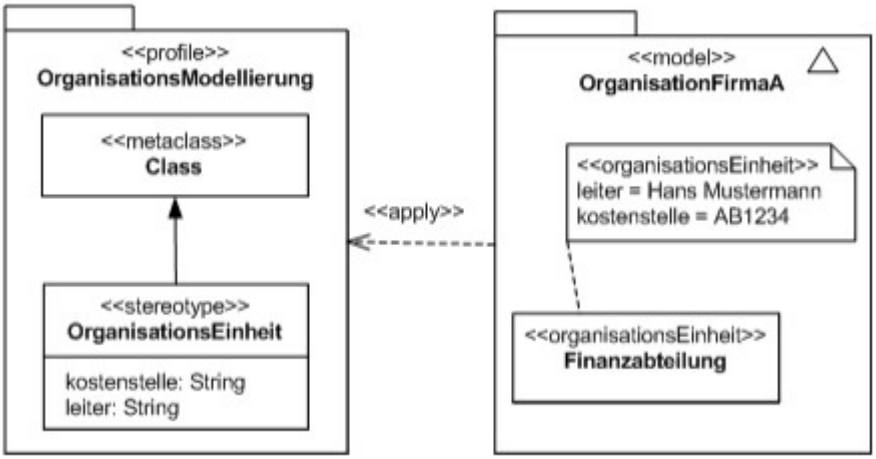
- Package Diagram

Describes how a system is split-up into logical groupings by showing the dependencies among these groupings.



- Profile Diagram

Operates at the metamodel level to show stereotypes as classes with the< <stereotype>> stereotype, and profiles as packages with the < <profile>> stereotype. The extension relation (solid line with closed, filled arrowhead) indicates what metamodel element a given stereotype is extending.



Behavior Diagrams

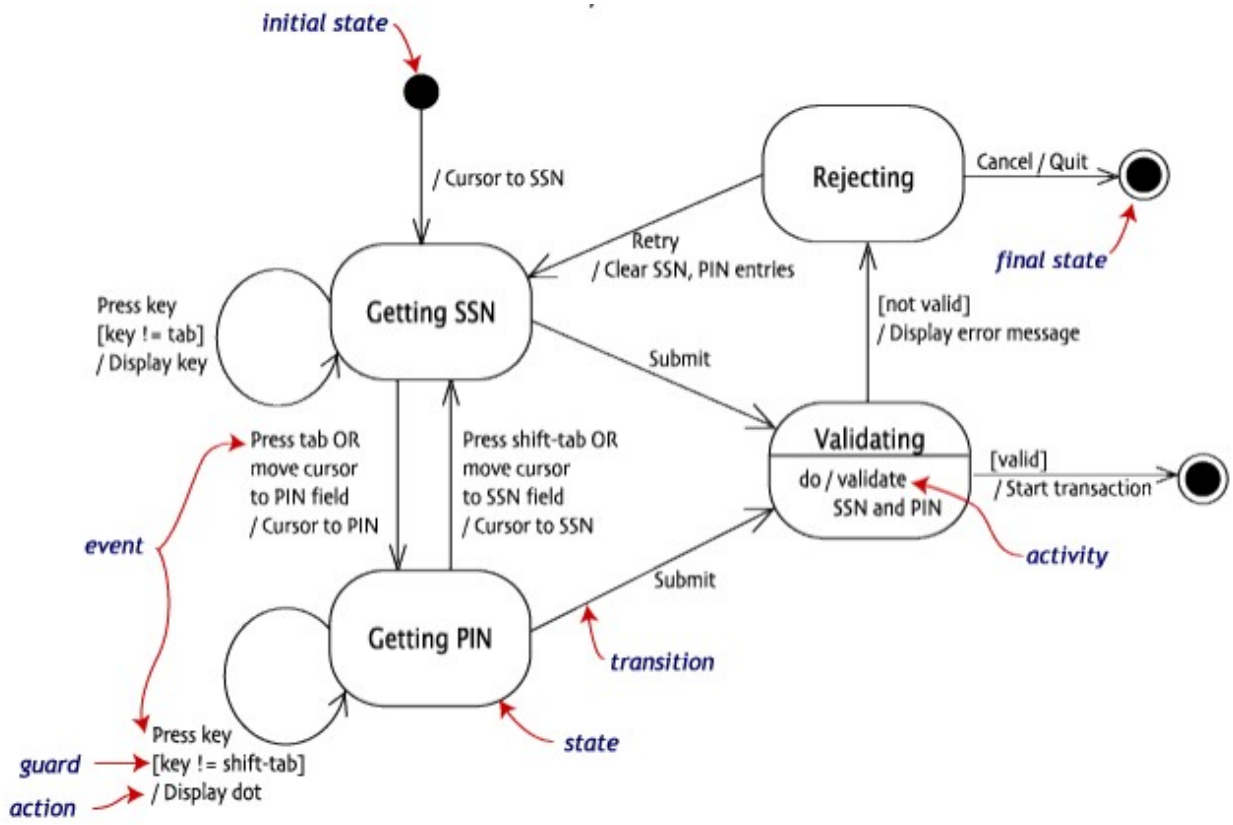
These diagrams emphasize what must happen in the system being modeled. Since they illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.

Activity Diagram

Describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.

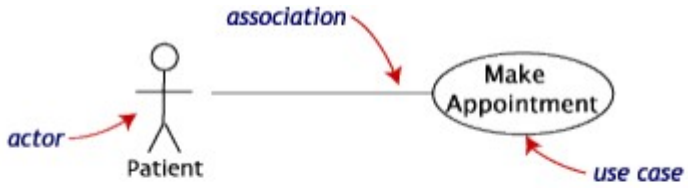
. State Machine Diagram

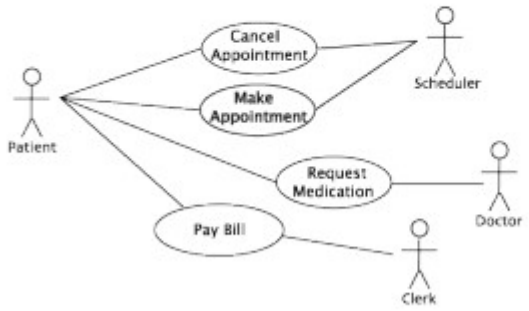
Describes the states and state transitions of the system.



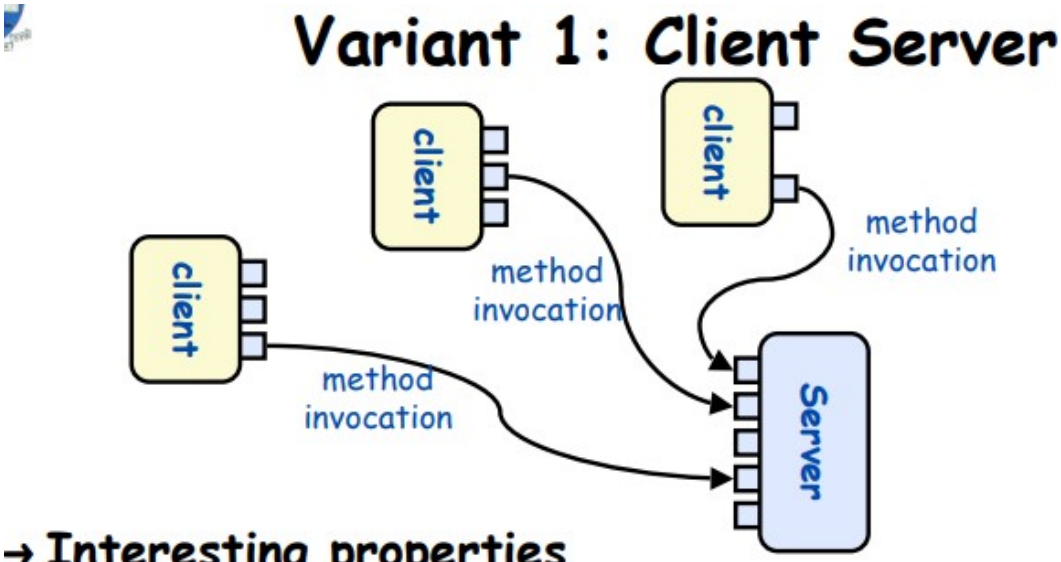
Use Case Diagram

Describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.





Software Architecture



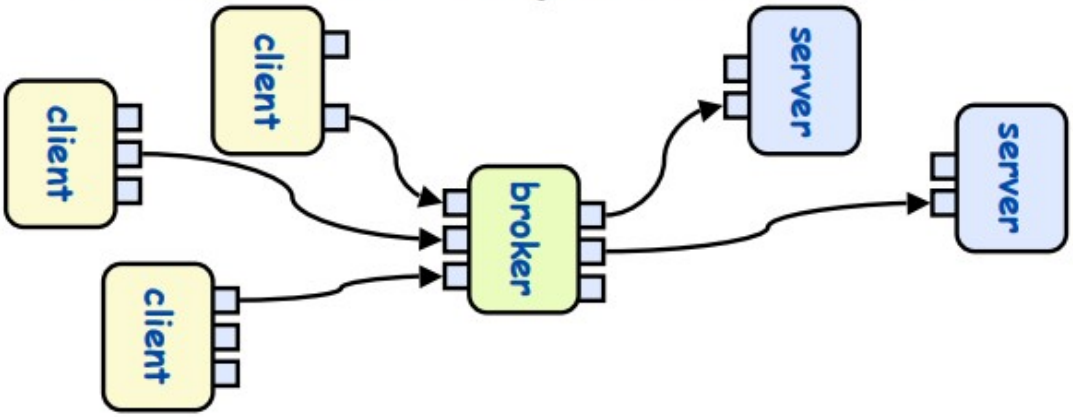
→ Interesting properties

- Is a special case of the previous pattern object oriented architecture
- Clients do not need to know about one another

→ Disadvantages

- Client objects must know the identity of the server

Variant 2: Object Brokers



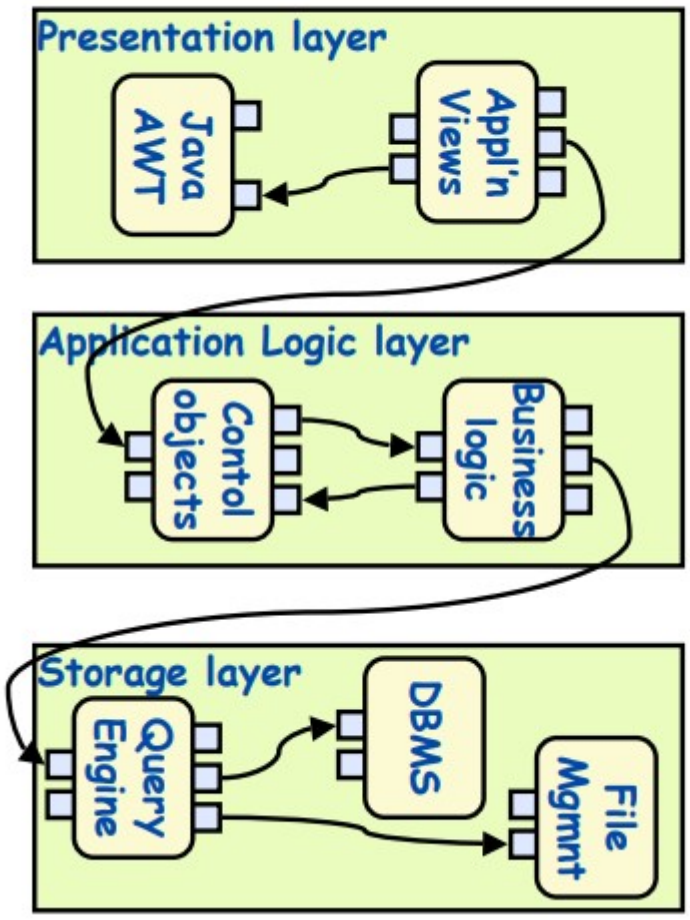
→ Interesting properties

- Adds a broker between the clients and servers
- Clients no longer need to know which server they are using
- Can have many brokers, many servers.

→ Disadvantages

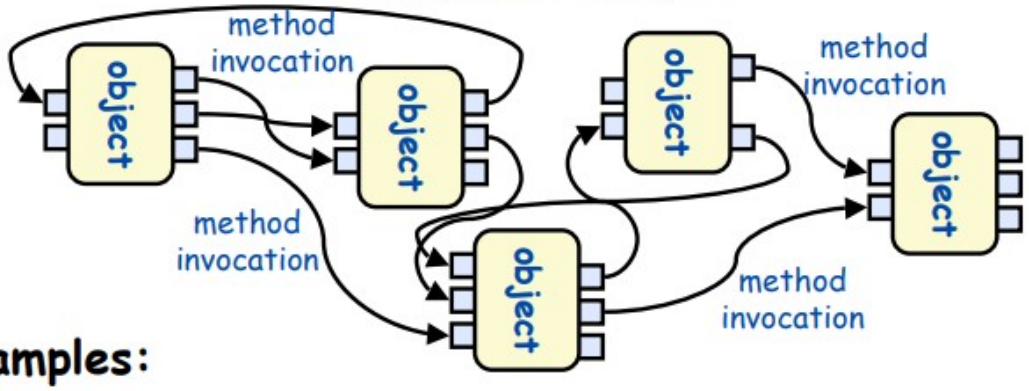
- Broker can become a bottleneck
- Degraded performance

Variant: 3-layer data access



Object Oriented Architectures

Source: Adapted from Shaw & Garlan 1996, p22-3.



→ Examples:

- abstract data types

→ Interesting properties

- data hiding (internal data representations are not visible to clients)
- can decompose problems into sets of interacting agents
- can be multi-threaded or single thread

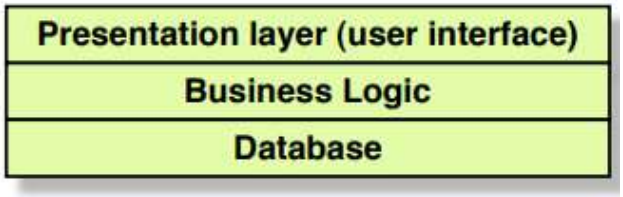
→ Disadvantages

- objects must know the identity of objects they wish to interact with

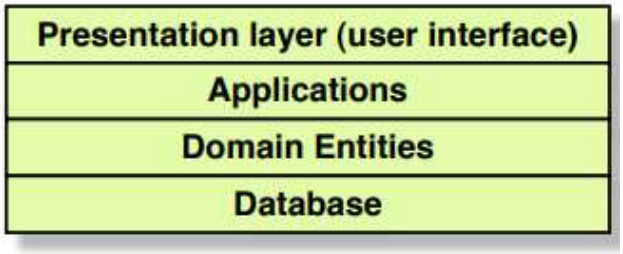
How many layers?

- **2-layers:**
  - application layer
  - database layer
  - e.g. simple client-server model

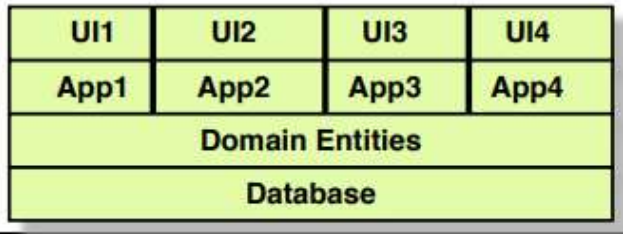
- **3-layers:**
  - separate out the business logic
    - helps to make both user interface and database layers modifiable

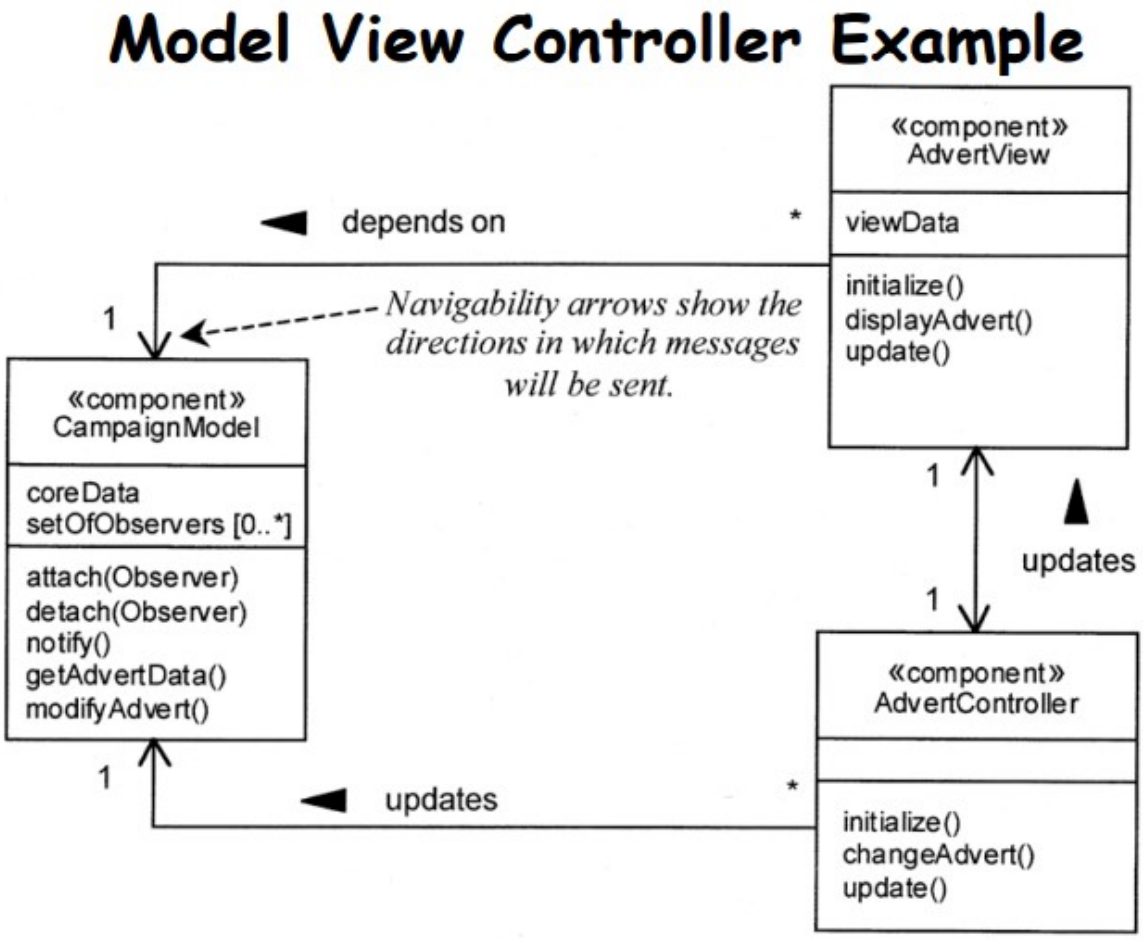


- **4-layers:**
  - Separates applications from the domain entities that they use:
    - boundary classes in presentation layer
    - control classes in application layer
    - entity classes in domain layer



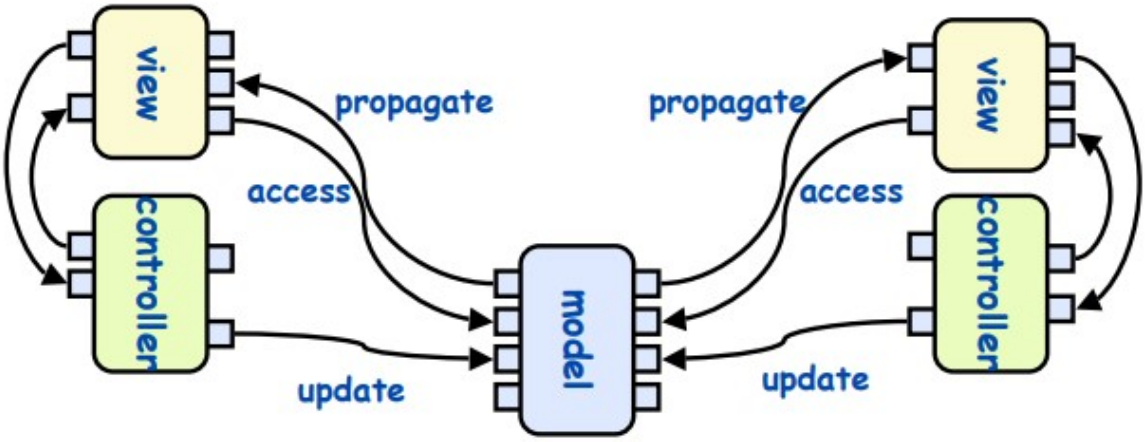
- **Partitioned 4-layers**
  - identify separate applications





User Interface Design

Variant: Model-View-Controller



Properties

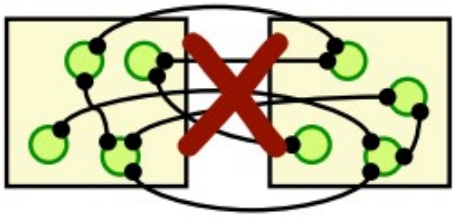
- One central model, many views (viewers)
- Each view has an associated controller
- The controller handles updates from the user of the view
- Changes to the model are propagated to all the views

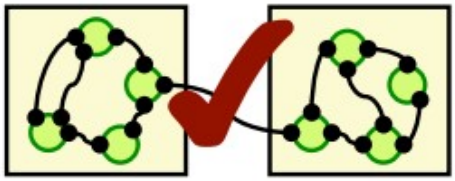
Coupling and Cohesion

→ Architectural Building blocks:

→ A good architecture:

- Minimizes coupling between modules:
  - Goal: modules don't need to know much about one another to interact
  - Low coupling makes future change easier
- Maximizes the cohesion of each module
  - Goal: the contents of each module are strongly inter-related
  - High cohesion makes a module easier to understand





User Interface Design:

Importance of user interface design can be considered from the following statements.

- "Today, user needs are recognized to be important in designing interactive computer systems, but as recently as 1980, they received little emphasis." J. Grudin
- "We can't worry about these user interface issues now. We haven't even gotten this thing to work yet!" Mulligan

Poor usability results in

– anger and frustration

– decreased productivity in the workplace

– higher error rates

– physical and emotional injury

– equipment damage

– loss of customer loyalty

– costs money

Usability: Usability is a measure of the effectiveness, efficiency and satisfaction with which specified users can achieve specified goals in a particular environment.

The User Interface today is often one of the most critical factors regarding the success or failure of a computer system

• Good UI design:

– Increases efficiency

– Improves productivity

– Reduces errors

– Reduces training

– Improves acceptance

• Approach: The UI is the system

– Design with the UI in mind

• Things to consider

– Technical issues in creating the UI

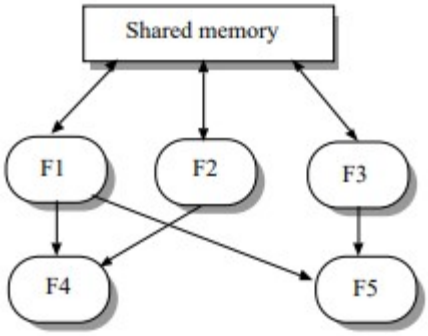
– User's mental model

– Conceptual mode

Function Oriented Design:

A function-oriented design strategy relies on decomposing the system into a set of interacting functions with a centralised system state shared by these functions (Figure 1). Functions may also maintain local state information but only for the duration of their execution. Function-oriented design has been practised informally since programming began. Programs were decomposed into subroutines which were functional in nature. In the late 1960s and early 1970s several books were published which described 'top-down' functional design. They specifically proposed this as a 'structured' design strategy (Myers, 1975; Wirth, 1976; Constantine and Yourdon, 1979). These led to the development of many design methods based on functional decomposition. Function-oriented design conceals the details of an

algorithm in a function but system state information is not hidden. This can cause problems because a function can change the state in a way which other functions do not expect. Changes to a function and the way in which it uses the system state may cause unanticipated changes in the behaviour of other functions. A functional approach to design is therefore most likely to be successful when the amount of system state information is minimised and information sharing is explicit. Systems whose responses depend on a single stimulus or input and which are not affected by input histories are naturally functionally-oriented. Many transaction-processing systems and business data-processing systems fall into this class. In essence, they are concerned with record processing where the processing of one record is not dependent on any previous processing. An example of such a transaction processing system is the software which controls automatic teller machines (ATMs) which are now installed outside many banks. The service provided to a user is independent of previous services provided so can be thought of as a single transaction. Figure 2 illustrates a simplified functional design of such a system. Notice that this design follows the centralised management control model introduced in Chapter 13. In this design, the system is implemented as a continuous loop and actions are triggered when a card is input. Functions such as Dispense_cash, Get_account_number, Order_statement, Order_checkbook, etc. can be identified which implement system actions. The system state maintained by the program is minimal. The user services operate independently and do not interact with each other. An object-oriented design would be similar to this and would probably not be significantly more maintainable.



```
loop
    loop
    Print_input_message (" Welcome - Please enter your card") ;
    exit when Card_input ;
    end loop ;
    Account_number := Read_card ;
    Get_account_details (PIN, Account_balance, Cash_available) ;
    if Validate_card (PIN) then
    loop
     Print_operation_select_message ;
    case Get_button is
              when Cash_only =>
                   Dispense_cash (Cash_available, Amount_dispensed) ;
              when Print_balance =>
                   Print_customer_balance (Account_balance) ;
              when Statement =>
                   Order_statement (Account_number) ;
              when Check_book =>
                   Order_checkbook (Account_number) ;
    end case ;
    Eject_card ;
    Print ("Please take your card or press CONTINUE") ;
    exit when Card_removed ;
    end loop ;
    Update_account_information (Account_number,  Amount_dispensed) ;
     else
          Retain_card ;
    end if ;
end loop ;
```

Figure:2