Architectural Design

SOFTWARE ARCHITECTURE

Architecture serves as a **blueprint for a system**. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components. It defines a **structured solution** to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.

What Is Architecture?

"**The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them**." The architecture is not the operational software. Rather, it is a representation that enables you to

(1) analyze the effectiveness of the design in meeting its stated requirements,

(2) consider architectural alternatives at a stage when making design changes is still relatively easy, and

(3) reduce the risks associated with the construction of the software.

Why Is Architecture Important?

Software architecture is important for the following reasons:

• Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.

• The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.

• Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together" The architectural design model and the architectural patterns contained within it are transferable.

Architectural Descriptions

An architectural description of a software-based system must exhibit characteristics that are analogous to those noted for the office building.

The IEEE Computer Society has proposed, *Recommended Practice for Architectural Description of Software-Intensive Systems,* with the following objectives:

(1) to establish a conceptual framework and vocabulary for use during the design of software architecture,

(2) to provide detailed guidelines for representing an architectural description,

and

(3) to encourage sound architectural design practices.

The IEEE standard defines an *architectural description* (AD) as "a collection of products to document an architecture." The description itself is represented using multiple views, where each *view* is "a representation of a whole system from the perspective of a related set of concerns."

Architectural Decisions

Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern. Therefore, architectural decisions themselves can be considered to be one view of the architecture. The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns.

ARCHITECTURAL GENRES

The architectural ***genre*** will often dictate the specific architectural approach to the structure that must be built. In the context of architectural design, *genre* implies a specific category within the overall software domain. **Grady Booch** suggests the following architectural genres for software-based systems:

- **Artificial intelligence**—Systems that simulate or augment human cognition, locomotion, or other organic processes.
- **Commercial and nonprofit**—Systems that are fundamental to the operation of a business enterprise.
- **Communications**—Systems that provide the infrastructure for transferring and managing data, for connecting users of that data, or for presenting data at the edge of an infrastructure.
- **Content authoring**—Systems that are used to create or manipulate textual or multimedia artifacts.
- **Devices**—Systems that interact with the physical world to provide some point service for an individual.
- **Entertainment and sports**—Systems that manage public events or that provide a large group entertainment experience.
- **Financial**—Systems that provide the infrastructure for transferring and managing money and other securities.
- **Games**—Systems that provide an entertainment experience for individuals or groups.
- **Government**—Systems that support the conduct and operations of a local, state, federal, global, or other political entity.
- **Industrial**—Systems that simulate or control physical processes.

• **Legal**—Systems that support the legal industry.

• **Medical**—Systems that diagnose or heal or that contribute to medical research.

• **Military**—Systems for consultation, communications, command, control, and intelligence as well as offensive and defensive weapons.

• **Operating systems**—Systems that sit just above hardware to provide basic software services.

• **Platforms**—Systems that sit just above operating systems to provide advanced services.

• **Scientific**—Systems that are used for scientific research and applications.

• **Tools**—Systems that are used to develop other systems.

• **Transportation**—Systems that control water, ground, air, or space vehicles.

• **Utilities**—Systems that interact with other software to provide some point service.

ARCHITECTURAL STYLES

An *architectural style* as a descriptive mechanism to differentiate the house from other styles. The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses (1) a set of components (e.g., a database, computational modules) that perform a function required by a system; (2) a set of connectors that enable "communication, coordination and cooperation" among components; (3) constraints that define how components can be integrated to form the system; and (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system.

Architectural Styles

Data-centered architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Client software accesses a central repository. In some cases the data repository is passive. Data-centered architectures promote ***integrability.***

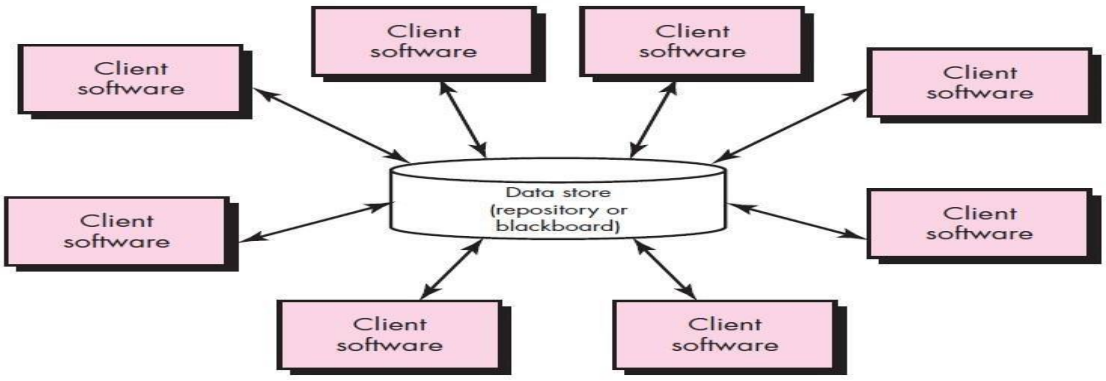


Fig : Data-centered architecture

Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. It has a set of components, called ***filters***, connected by ***pipes*** that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output of a specified form. However, the filter does not require knowledge of the Workings of its neighboring filters.

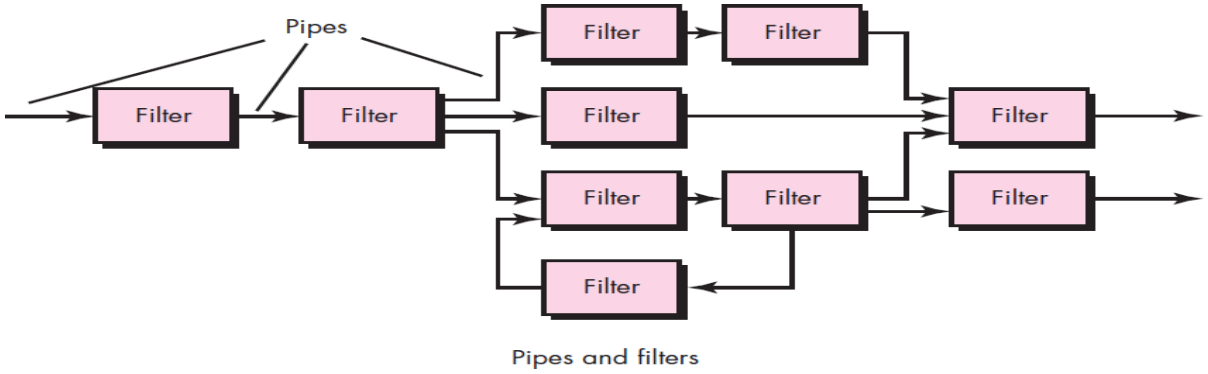
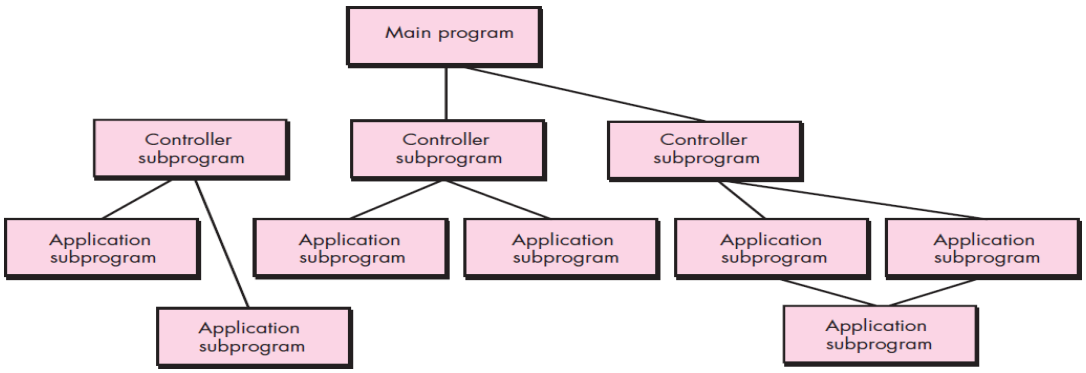


Fig : Data-flow architecture

Call and return architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of sub styles exist within this category:

• ***Main program/subprogram architectures***. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other components. The following figure illustrates an architecture of thistype.

• ***Remote procedure call architectures***. The components of a main program/subprogram architecture are distributed across multiple computers

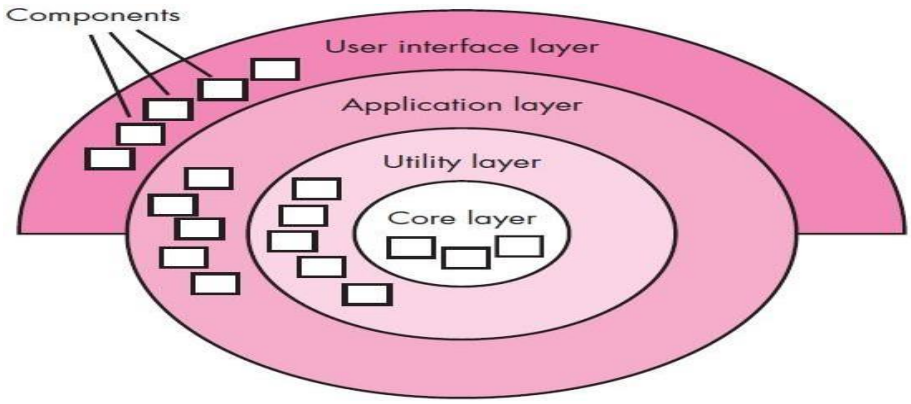


on anetwork.

Fig : Main program/subprogram architecture

Object-oriented architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via **message passing**.

Fig : Layered architecture



Layered architectures. The basic structure of a layered architecture is illustrated in following figure. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

Architectural Patterns

Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

Organization and Refinement

The following questions provide insight into an architectural style:

Control. How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system? How is control shared among components? What is the control topology? Is control synchronized or do components operate asynchronously?

Data. How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer? Do data components exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active? How do data and control interact within the system?

These questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

ARCHITECTURAL DESIGN

As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural **archetypes**.

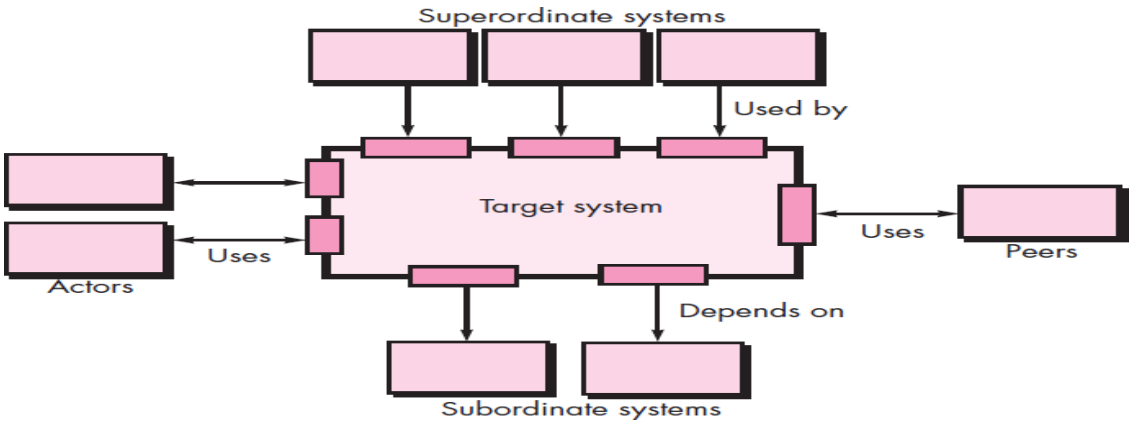
An ***archetype*** is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail.

Representing the System in Context

At the architectural design level, a software architect uses an ***architectural context diagram*(ACD)** to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in following figure. Referring to the figure, systems that interoperate with the *target system* (the system for which an architectural design is to be developed) are represented as

- ***Superordinate systems***—those systems that use the target system as part of some higher-level processing scheme.
- ***Subordinate systems***—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- ***Peer-level systems***—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.

• ***Actors***—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite



processing.

Fig : Architectural context diagram

Defining Archetypes

An ***archetype*** is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

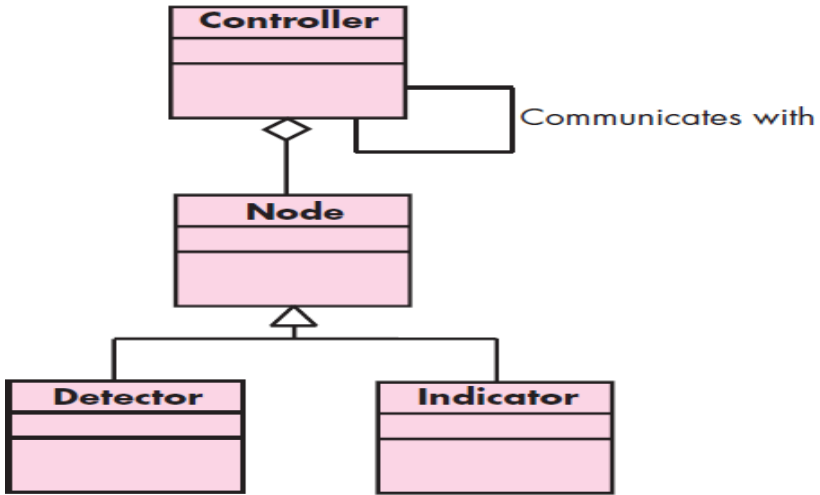
The following archetypes can be used :

• **Node.** Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors and (2) a variety of alarm (output)indicators.

• **Detector.** An abstraction that encompasses all sensing equipment that feeds information into the target system.

• **Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.

• **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.



Refining the Architecture into Components

As the software architecture is refined into components, the structure of the system begins to emerge. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. Set of top-level components that address the following functionality:

- ***External communication management***—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
- ***Control panel processing***—manages all control panel functionality.
- ***Detector management***—coordinates access to all detectors attached to the system.
- ***Alarm processing***—verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall architecture.

ASSESSING ALTERNATIVE ARCHITECTURAL DESIGNS

An Architecture Trade-Off Analysis Method

The Software Engineering Institute (SEI) has developed an ***architecture trade-off analysis method*** **(ATAM)** that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

1. ***Collect scenarios***. A set of use cases is developed to represent the system from the user's point of view.
2. ***Elicit requirements, constraints, and environment description***. This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have beenaddressed.
3. ***Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements.*** The architectural style(s) should be described using one of the following architectural views:
   - ***Module view*** for analysis of work assignments with components and the degree to which information hiding has been achieved.
   - ***Process view*** for analysis of system performance.
   - ***Data flow view*** for analysis of the degree to which the architecture meets functional requirements.
4. ***Evaluate quality attributes by considering each attribute in isolation***. The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability,

performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.

5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style. This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed *sensitivity points.*

6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step5.

Architectural Complexity

A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the system. Zhao suggests **three** types of dependencies:

Sharing dependencies represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers

Flow dependencies represent dependence relationships between producers and consumers of resources.

Constrained dependencies represent constraints on the relative flow of control among a set of activities.

Architectural Description Languages

Architectural description language (ADL) provides a semantics and syntax for describing a software architecture. Hofmann and his colleagues suggest that an ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components.

ARCHITECTURAL MAPPING USING DATA FLOW

A mapping technique, called *structured design* is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram to software architecture. The transition from information flow (represented as a DFD) to program structure is accomplished as part of a **six** step process:

(1) the type of information flow is established,

(2) flow boundaries are indicated,

(3) the DFD is mapped into the program structure,

(4) control hierarchy is defined,

(5) the resultant structure is refined using design measures and heuristics, and

(6) the architectural description is refined and elaborated.

In order to perform the mapping, the type of information flow must be determined. One type of information flow is called ***transform flow*** and exhibits a linear quality. Data flows into the system along an ***incoming flow*** *path* where it is transformed from an external world representation into internalized form. Once it has been internalized, it is processed at a

transform center. Finally, it flows out of the system along an ***outgoing flow path*** that transforms the data into external world.

Transform Mapping

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. To map data flow diagrams into a software architecture, you would initiate the following design steps:

Step 1. **Review the fundamental system model. The fundamental system model :**The fundamental system model or context diagram depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function. The following figure depicts a level 0 context model, and the next figure shows refined data flow for the security function.

Fig : Context-level DFD for the *SafeHome*security function

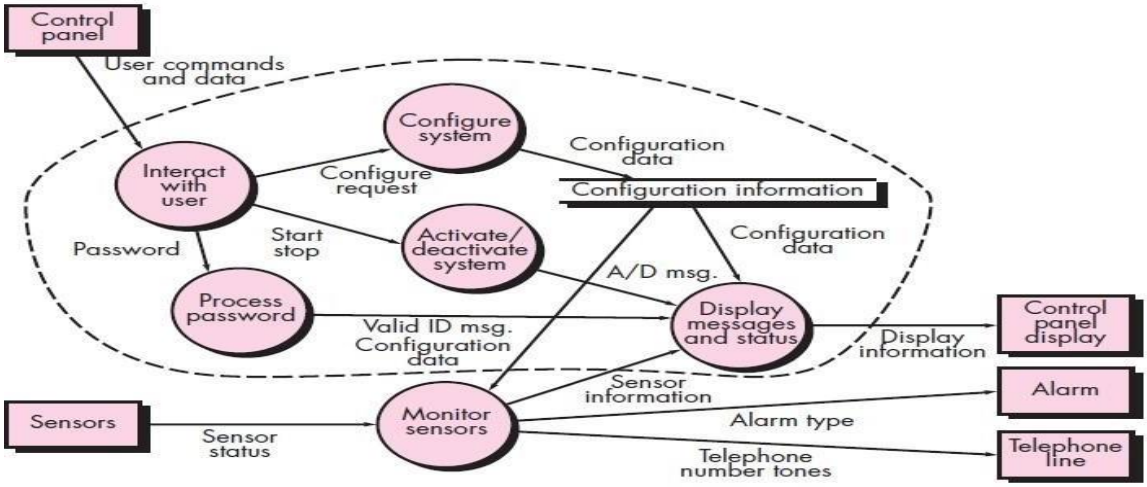


Fig : Level 1 DFD for the *SafeHome*security function

Step 2. Review and refine data flow diagrams for the software. Information obtained from the requirements model is refined to produce greater detail

Step 3. Determine whether the DFD has transform or transaction flow characteristics. Evaluating the DFD, we see data entering the software along one incoming path and exiting along three outgoing paths. Therefore, an overall transform characteristic will be assumed for informationflow.

Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries. Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations.

Step 5. Perform "first-level factoring." The program architecture derived using this mapping results in a top-down distribution of control. ***Factoring*** leads to a program structure in which top-level components perform decision making and low level components perform most input, computation, and output work. Middle-level components perform some control and do moderate amounts of work.

Step 6. Perform "second-level factoring." Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure. The general approach to second level is a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur. Two or even three bubbles can be combined and represented as one component, or a single bubble may be expanded to two or more components.

Step 7. Refine the first-iteration architecture using design heuristics for improved software quality. A first-iteration architecture can always be refined by applying concepts of functional independence. Components are exploded or imploded to produce sensible factoring, separation of concerns, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

Refining the Architectural Design

Refinement of software architecture during early stages of design is to be encouraged. Design refinement should strive for the smallest number of components that is consistent with effective modularity and the least complex data structure that adequately serves information requirements.

Component-level design

Component-level design occurs after the first iteration of architectural design has been completed. At this stage, the overall data and program structure of the software has been established. The intent is to translate the design model into operational software.

A ***component*** is a modular building block for computer software. More formally,

the *OMG Unified Modeling Language Specification* defines a component as "a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

An Object-Oriented View

In the context of object-oriented software engineering, a component contains a set of collaborating classes. Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation. To accomplish this, you begin with the requirements model and elaborate analysis classes and infrastructure classes.

The Traditional View

In the context of traditional software engineering, a component is a functional element of a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it. A traditional component, also called a **module**, resides within the software architecture and serves one of **three** important roles:

(1) A control component that coordinates the invocation of all other problem domain components,

(2) a problem domain component that implements a complete or partial function that is required by the customer, or

(3) an infrastructure component that is responsible for functions that support the processing required in the problem domain.

DESIGNING CLASS-BASED COMPONENTS

Basic Design Principles

Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied.

The Open-Closed Principle (OCP). *"**A module [component] should be open for extension but closed for modification**"* This statement seems to be a contradiction, but it represents one of the most important characteristics of a good component-level design. Stated simply, you should specify the component in a way that allows it to be extended without the need to make internal modifications to the component itself.

The Liskov Substitution Principle (LSP). *"**Subclasses should be substitutable for their base classes**".* This design principle, originally proposed by Barbara Liskov, suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead. LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it. In the context of this discussion, a "contract" is a ***precondition*** that must be true before the component uses a base class and a ***post condition*** that should be true after the component uses a base class.

Dependency Inversion Principle (DIP). *"**Depend on abstractions. Do not depend on***

concretions". The more a component depends on other concrete components, the more difficult it will be to extend.

The Interface Segregation Principle (ISP). *"**Many client-specific interfaces are better than one general purpose interface"***. ISP suggests that you should create a specialized interface to serve each major category of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface for that client. If multiple clients require the same operations, it should be specified in each of the specialized interfaces.

The Release Reuse Equivalency Principle (REP). *"**The granule of reuse is the granule of release"***. When classes or components are designed for reuse, there is an implicit contract that is established between the developer of the reusable entity and the people who will use it. The developer commits to establish a release control system that supports and maintains older versions of the entity while the users slowly upgrade to the most current version. Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve.

The Common Closure Principle (CCP). *"**Classes that change together belong together.**"* Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioral area. When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads to more effective change control and release management.

The Common Reuse Principle (CRP). *"**Classes that aren't reused together should not be grouped together"***. When one or more classes within a package changes, the release number of the package changes. All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operates without incident. If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed.

Component-Level Design Guidelines

Components. Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model.

Interfaces. Interfaces provide important information about communication and collaboration. Ambler recommends that (1) lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach, when diagrams grow complex; (2) for consistency, interfaces should flow from the left-hand side of the component box; (3) only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available.

Cohesion

cohesion is the "single-mindedness" of a component. Lethbridge and Laganiére define a number of different types of cohesion

Functional. Exhibited primarily by operations, this level of cohesion occurs when a component performs a targeted computation and then returns a result.

Layer. Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.

Communicational. All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

Coupling

Coupling is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep **coupling as low as is possible**.

Class coupling can manifest itself in a variety of ways. Lethbridge and Laganiére define the following coupling categories:

Content coupling. Occurs when one component "surreptitiously modifies data that is internal to another component".

Common coupling. Occurs when a number of components all make use of a global variable. Although this is sometimes necessary, common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.

Control coupling. Occurs when operation *A()* invokes operation *B()* and passes a control flag to *B*. The control flag then "directs" logical flow within *B*. The problem with this form of coupling is that an unrelated change in *B* can result in the necessity to change the meaning of the control flag that *A* passes. If this is overlooked, an error will result.

Stamp coupling. Occurs when **ClassB**is declared as a type for an argument of an operation of **ClassA**. Because **ClassB**is now a part of the definition of **ClassA**, modifying the system becomes more complex.

Data coupling. Occurs when operations pass long strings of data arguments. The "bandwidth" of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.

Routine call coupling. Occurs when one operation invokes another. This level of coupling is common and is often quite necessary. However, it does increase the connectedness of a system.

Type use coupling. Occurs when component **A** uses a data type defined in component **B.** If the type definition changes, every component that uses the definition must also change.

Inclusion or import coupling. Occurs when component **A** imports or includes a package or the content of component **B**.

External coupling. Occurs when a component communicates or collaborates with infrastructure components. Although this type of coupling is necessary, it should be limited

to a small number of components or classes within a system.

Software must communicate internally and externally. Therefore, coupling is a fact of life. However, the designer should work to **reduce coupling whenever possible**.

CONDUCTING COMPONENT-LEVEL DESIGN

The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1. Identify all design classes that correspond to the problem domain. Using the requirements and architectural model, each analysis class and architectural component is elaborated.

Step 2. Identify all design classes that correspond to the infrastructure domain. These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point.

Step 3. Elaborate all design classes that are not acquired as reusable components. Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

Step 3a. Specify message details when classes or components collaborate. The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. As component-level design proceeds, it is sometimes useful to show the details of these collaborations by specifying the structure of messages that are passed between objects within a system.

Step 3c. Elaborate attributes and define data types and data structures required to implement them. In general, data structures and types used to define attributes are defined within the context of the programming language that is to be.

Step 3d. Describe processing flow within each operation in detail. This may be accomplished using a programming language-based pseudocode or with a UML activity diagram.

Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them. Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design.

Step 5. Develop and elaborate behavioral representations for a class or component. UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes.

Step 6. Elaborate deployment diagrams to provide additional implementation detail. Deployment diagrams are used as part of architectural design and are represented in descriptor form. During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components.

Step 7. Refactor every component-level design representation and always consider alternatives. The first component-level model you create will not be as complete, consistent, or accurate as the *n*th iteration you apply to the model.

COMPONENT-LEVEL DESIGN FOR WEBAPPS

A WebApp component is (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end user or (2) a cohesive package of content and functionality that provides the end user with some required capability. Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

Content Design at the Component Level

Content design at the component level focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end user.

Functional Design at the Component Level

Modern Web applications deliver increasingly sophisticated processing functions that (1) perform localized processing to generate content and navigation capability in a dynamic fashion, (2) provide computation or data processing capability that is appropriate for the WebApp's business domain, (3) provide sophisticated database query and access, or (4) establish data interfaces with external corporate systems. To achieve capabilities, you will design and construct WebApp functional components that are similar in form to software components for conventional software.

WebApp functionality is delivered as a series of components developed in parallel with the information architecture to ensure that they are consistent.

During architectural design, WebApp content and functionality are combined to create a functional architecture. A ***functional architecture*** is a representation of the functional domain of the WebApp and describes the key functional components in the WebApp and how these components interact with each other.

DESIGNING TRADITIONAL COMPONENTS

The foundations of component-level design for traditional software components were formed in the early 1960s and were solidified with the work of Edsger Dijkstra and his colleagues. In the late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed. The constructs emphasized "maintenance of functional domain."

The constructs are **sequence, condition, and repetition**. ***Sequence*** implements processing steps that are essential in the specification of any algorithm. ***Condition*** provides the facility for selected processing based on some logical occurrence, and ***repetition*** allows for looping. These **three** constructs are fundamental to ***structured programming***—an

important component-level design technique.

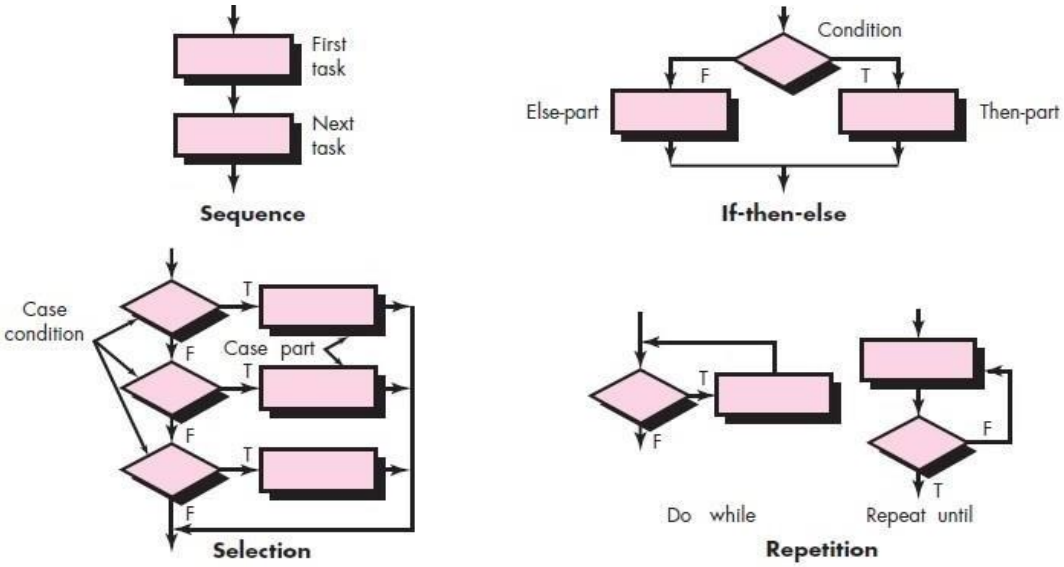
Graphical Design Notation

"A picture is worth a thousand words," but it's rather important to know which picture and which 1000 words. There is no question that graphical tools, such as the UML activity diagram or the flowchart, provide useful pictorial patterns that readily depict procedural detail.

The activity diagram allows you to represent **sequence, condition, and repetition** and all elements of **structured programming. A**nd is a descendent of an earlier pictorial design representation called a ***flowchart.*** A flowchart, like an activity diagram, is quite simple pictorially. A box is used to indicate a processing step. A diamond represents a logical condition, and arrows show the flow . The following figure illustrates **three** structured constructs.

The ***sequence*** is represented as two processing boxes connected by a line (arrow) of control. ***Condition***, also called ***if-then-else***, is depicted as a decision diamond that, if true, causes ***then-part*** processing to occur, and if false, invokes ***else-part*** processing. ***Repetition*** is represented using two slightly different forms. The ***do while*** tests a condition and executes a loop

Fig : Flowchart constructs



task repetitively as long as the condition holds true. A ***repeat until*** executes the loop task first and then tests a condition and repeats the task until the condition fails. The ***selection*** (or *select- case*) construct shown in the figure is actually an extension of the ***if-then-else***.

Tabular Design Notation

Decision tables provide a notation that translates actions and conditions into a tabular form. The table is difficult to misinterpret and may even be used as a machine-readable input to a table-driven algorithm. Decision table organization is illustrated in

following figure. Referring to the figure, the table is divided into **four** sections. The **upper left-hand quadrant** contains a list of all conditions. The **lower left-hand quadrant** contains a list of all actions that are possible based on combinations of conditions. The **right-hand quadrants** form a matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination. Therefore, each column of the matrix may be interpreted as a ***processing rule***. The following steps are applied to develop a decision table:

1. List all actions that can be associated with a specific procedure (or component).

2. List all conditions (or decisions made) during execution of the procedure.

3. Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.

4. Define rules by indicating what actions occur for a set of conditions.

	Rules					
Conditions	**1**	**2**	**3**	**4**	**5**	**6**
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional *x* percent discount		✓		✓		✓

Fig : Decision table

Program Design Language

Program design language **(PDL)**, also called *structured English* or *pseudocode*, incorporates the logical structure of a programming language with the free-form expressive ability of a natural language (e.g., English). Automated tools can be used to enhance the application of PDL.

A basic PDL syntax should include constructs for component definition, interface description, data declaration, block structuring, condition constructs, repetition constructs, and input-output (I/O) constructs. It should be noted that PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt handling, inter process synchronization, and many other features.

COMPONENT-BASED DEVELOPMENT

Component-based software engineering (CBSE) is a process that emphasizes the design

and construction of computer-based systems using reusable software“ components.”

Domain Engineering

The intent of ***domain engineering*** is to identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain. The overall goal is to establish mechanisms that enable software engineers to share these components to reuse them during work on new and existing systems. Domain engineering includes **three** major activities— **analysis, construction, and dissemination**. The overall approach to ***domain analysis*** is often characterized within the context of object-oriented software engineering. The steps in the process are definedas:

1. Define the domain to be investigated.
2. Categorize the items extracted from the domain.
3. Collect a representative sample of applications in the domain.
4. Analyze each application in the sample and define analysis classes.
5. Develop a requirements model for the classes.

Component Qualification, Adaptation, and Composition

Domain engineering provides the library of reusable components that are required for component-based software engineering. Some of these reusable components are developed in- house, others can be extracted from existing applications, and still others may be acquired from third parties.

Component Qualification. Component qualification ensures that a candidate component will perform the function required, will properly “fit” into the architectural style specified for the system, and will exhibit the quality characteristics (e.g., performance, reliability, usability) that are required for the application.

Among the many factors considered during component qualification are :

- **Application programming interface(API).**
- **Development and integration tools** required by the component.
- **Run-time requirements**, including resource usage (e.g., memory or storage), timing or speed, and network protocol.
- **Service requirements**, including operating system interfaces and support from other components.
- **Security features**, including access controls and authentication protocol. Embedded design assumptions, including the use of specific numerical or non numerical algorithms.
- **Exception handling**. Each of these factors is relatively easy to assess when reusable components that have been developed in-house are proposed.

Component Adaptation : An adaptation technique called ***component wrapping.*** When a software team has full access to the internal design and code for a component ***white-box wrapping*** is applied. **white-box** wrapping examines the internal processing details of the component and makes code-level modifications to remove any conflict. ***Gray-box wrapping*** is applied when the component library provides a component extension language or API that enables conflicts to be removed or masked. ***Black-box wrapping*** requires the introduction of pre- and post processing at the component interface to remove or mask conflicts.

Component Composition. The component composition task assembles qualified, adapted, and engineered components to populate the architecture established for an application. To accomplish this, an infrastructure must be established to bind the components into an operational system. The infrastructure provides a model for the coordination of components and specific services that enable components to coordinate with one another and perform common tasks.

Analysis and Design for Reuse

Binder suggests a number of key issues that form a basis for design for reuse:

Standard data. The application domain should be investigated and standard global data structures (e.g., file structures or a complete database) should be identified. All design components can then be characterized to make use of these standard data structures.

Standard interface protocols. Three levels of interface protocol should be established: the nature of intra modular interfaces, the design of external technical (nonhuman) interfaces, and the human-computer interface.

Program templates. An architectural style is chosen and can serve as a template for the architectural design of a new software.

Classifying and Retrieving Components

A reusable software component can be described in many ways, but an ideal description encompasses the ***3C model***—**concept, content, and context.**

The ***concept*** of a software component is "a description of what the component does". The interface to the component is fully described and the semantics represented within the context of pre- and post conditions is identified. The concept should communicate the intent of the component.

The ***content*** of a component describes how the concept is realized. In essence, the content is information that is hidden from casual users and need be known only to those who intend to modify or test the component.

The ***context*** places a reusable software component within its domain of applicability. That is, by specifying conceptual, operational, and implementation features, the context enables a software engineer to find the appropriate component to meet application requirements.

A reuse environment exhibits the following characteristics:

• A component database capable of storing software components and the classification information necessary to retrieve them.

• A library management system that provides access to the database.

• A software component retrieval system that enables a client application to retrieve components and services from the server.

• CBSE tools that support the integration of reused components into a new design or implementation.

****