Software Engineering-BSCE-301L

Module 4:

Software Design

Dr . Saurabh Agrawal

Faculty Id: 20165

School of Computer Science and Engineering

VIT, Vellore-632014

Tamil Nadu, India

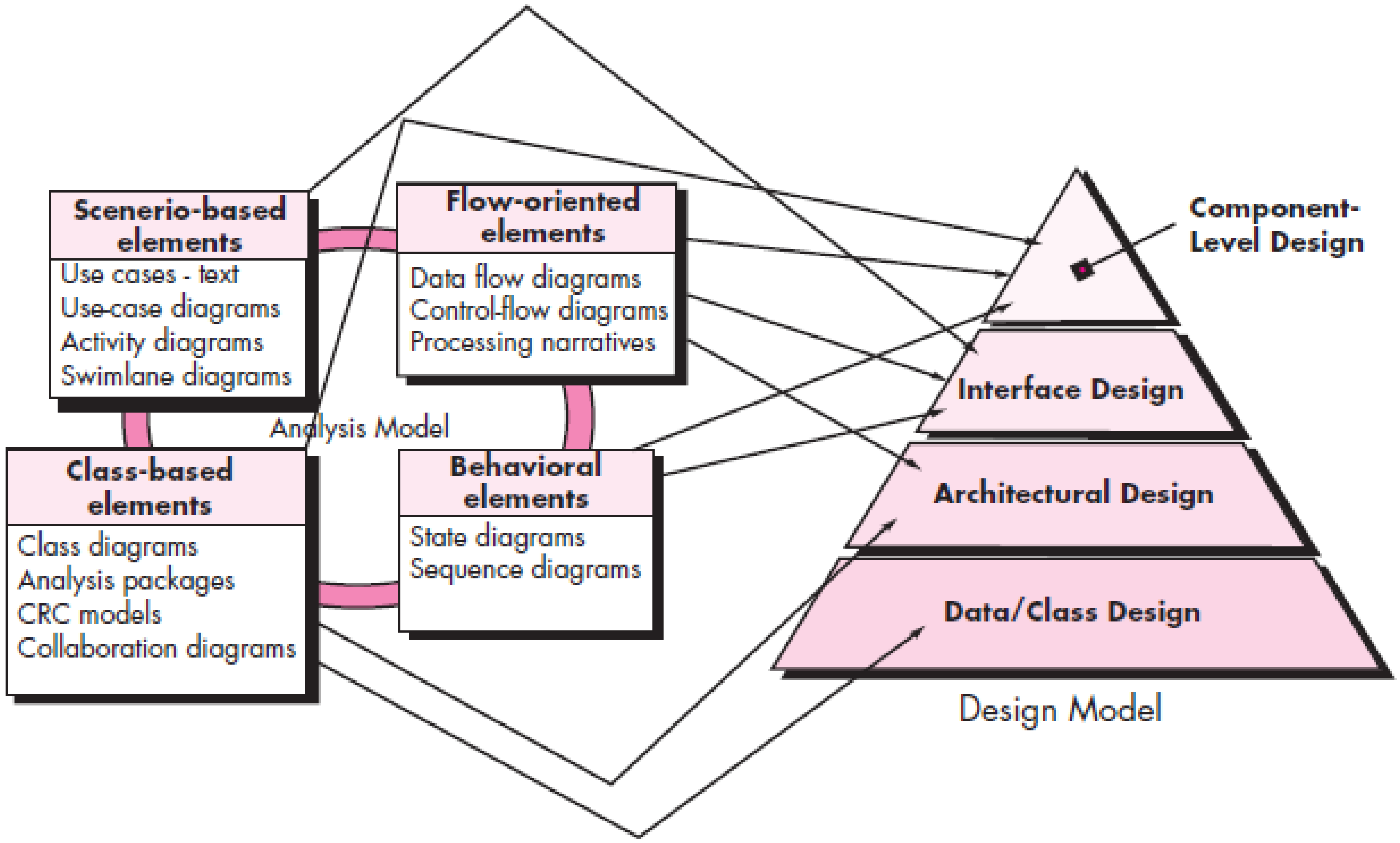
Outline

- ❑Design Concepts and Principles
- ❑Abstraction
- ❑Architecture
- ❑Modularity
- ❑Refinement
- ❑Cohesion
- ❑Coupling
- ❑Architectural Design
- ❑Detailed Design Transaction Transformation
- ❑Refactoring of Designs
- ❑Object oriented Design
- ❑User-Interface Design

Design Concepts and Principles

- ❑Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used.
- ❑Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for **construction.**
- ❑Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design.
- ❑The flow of information during software design is illustrated in Figure.
- ❑The requirements model, manifested by scenario-based, class-based, flow-oriented, and behavioral elements, feed the design task.
- ❑Using design notation and design methods discussed in later chapters, design produces a data/class design, an architectural design, an interface design, and a component design.

Design Concepts and Principles



Design Concepts and Principles

- The **data/class design** transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action. Part of class design may occur in conjunction with the design of software architecture. More detailed class design occurs as each software component is designed.
- The **architectural design** defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

Design Concepts and Principles

- The **interface design** describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.
- The **component-level** design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

Design Concepts and Principles

❑**Quality Attributes:** Hewlett-Packard [Gra87] developed a set of software quality attributes that has been given the acronym FURPS—functionality, usability, reliability, performance, and supportability. The FURPS quality attributes represent a target for all software design:

1. **Functionality** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
2. **Usability** is assessed by considering human factors, overall aesthetics, consistency, and documentation.
3. **Reliability** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
4. **Performance** is measured by considering processing speed, response time, resource, consumption, throughput, and efficiency.
5. **Supportability** combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, maintainability—and in addition, testability, compatibility, configurability , the ease with which a system can be installed, and the ease with which problems can be localized.

Design Concepts and Principles

❑A set of fundamental software design concepts has evolved over the history of software engineering.

❑Although the degree of interest in each concept has varied over the years, each has stood the test of time.

❑Each provides the software designer with a foundation from which more sophisticated design methods can be applied.

❑Each helps you answer the following questions:

▪What criteria can be used to partition software into individual components?

▪How is function or data structure detail separated from a conceptual representation of the software?

▪What uniform criteria define the technical quality of a software design?

❑M. A. Jackson [Jac75] once said: “The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right.” Fundamental software design concepts provide the necessary framework for “getting it right.”

Design Concepts and Principles

❑Following are the important software design concepts:

1. **Abstraction:** When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.
2. **Architecture:** Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system" [Sha95a]. In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

Design Concepts and Principles

❑Following are the important software design concepts:

3. **Patterns:** Brad Appleton defines a design pattern in the following manner: “A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns” [App00].
4. **Separation of Concerns:** Separation of concerns is a design concept [Dij82] that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A concern is a feature or behavior that is specified as part of the requirements model for the software.
5. **Modularity:** Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.

Design Concepts and Principles

❑Following are the important software design concepts:

6. **Information Hiding:** The concept of modularity leads you to a fundamental question: "How do I decompose a software solution to obtain the best set of modules?" The principle of information hiding [Par72] suggests that modules be "characterized by design decisions that (each) hides from all others."
7. **Functional Independence:** Functional independence is achieved by developing modules with "singleminded" function and an "aversion" to excessive interaction with other modules. Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure. It is fair to ask why independence is important.

Design Concepts and Principles

❑Following are the important software design concepts:

8. **Refinement:** Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth [Wir71]. A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.
9. **Aspects:** As requirements analysis occurs, a set of “concerns” is uncovered. These concerns “include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts” [AOS07]. Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently.

Design Concepts and Principles

❑Following are the important software design concepts:

10. **Refactoring:** An important design activity suggested for many agile methods, refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler [Fow00] defines refactoring in the following manner: “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”
11. **Object-Oriented Design Concepts:** The object-oriented (OO) paradigm is widely used in modern software engineering through OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others.
12. **Design Classes:** The requirements model defines a set of analysis classes. Each describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high.

Abstraction

- When you consider a modular solution to any problem, many levels of abstraction can be posed.
- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- At lower levels of abstraction, a more detailed description of the solution is provided.
- Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution.
- Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

Abstraction

❑As different levels of abstraction are developed, you work to create both:

1. **Procedural (Functional)**
2. **Data abstractions**

- A **procedural abstraction** refers to a sequence of instructions that have a specific and limited function.
- The name of a procedural abstraction implies these functions, but specific details are suppressed.
- An example of a procedural abstraction would be the word open for a door.
- Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

Abstraction

- A **data abstraction** is a named collection of data that describes a data object.
- In the context of the procedural abstraction open, we can define a data abstraction called door.
- Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).
- It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction door.

Abstraction

- As different levels of abstraction are developed, you work to create both:
    - A **procedural abstraction** refers to a sequence of instructions that have a specific and limited function.
    - The name of a procedural abstraction implies these functions, but specific details are suppressed.
    - An example of a procedural abstraction would be the word open for a door.
    - Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

Architecture

- ❑Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system".
- ❑In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.
- ❑In a broader sense, however, components can be generalized to represent major system elements and their interactions.
- ❑One goal of software design is to derive an architectural rendering of a system.
- ❑This rendering serves as a framework from which more detailed design activities are conducted.
- ❑A set of architectural patterns enables a software engineer to solve common design problems.

Architecture

- ❑Shaw and Garlan [Sha95a] describe a set of properties that should be specified as part of an architectural design:

1. **Structural properties.** This aspect of the architectural design representation defines: the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.
2. **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
3. **Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Architecture

- ❑Given the specification of these properties, the architectural design can be represented using one or more of a number of different models.
- ❑Structural models represent architecture as an organized collection of program components.
- ❑Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.
- ❑Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
- ❑Process models focus on the design of the business or technical process that the system must accommodate. Finally, functional models can be used to represent the functional hierarchy of a system.
- ❑A number of different *architectural description languages (ADLs) have been developed* to represent these models.
- ❑Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

Modularity

❑Modularity is the most common manifestation of separation of concerns.

❑Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.

❑It has been stated that “modularity is the single attribute of software that allows a program to be intellectually manageable”.

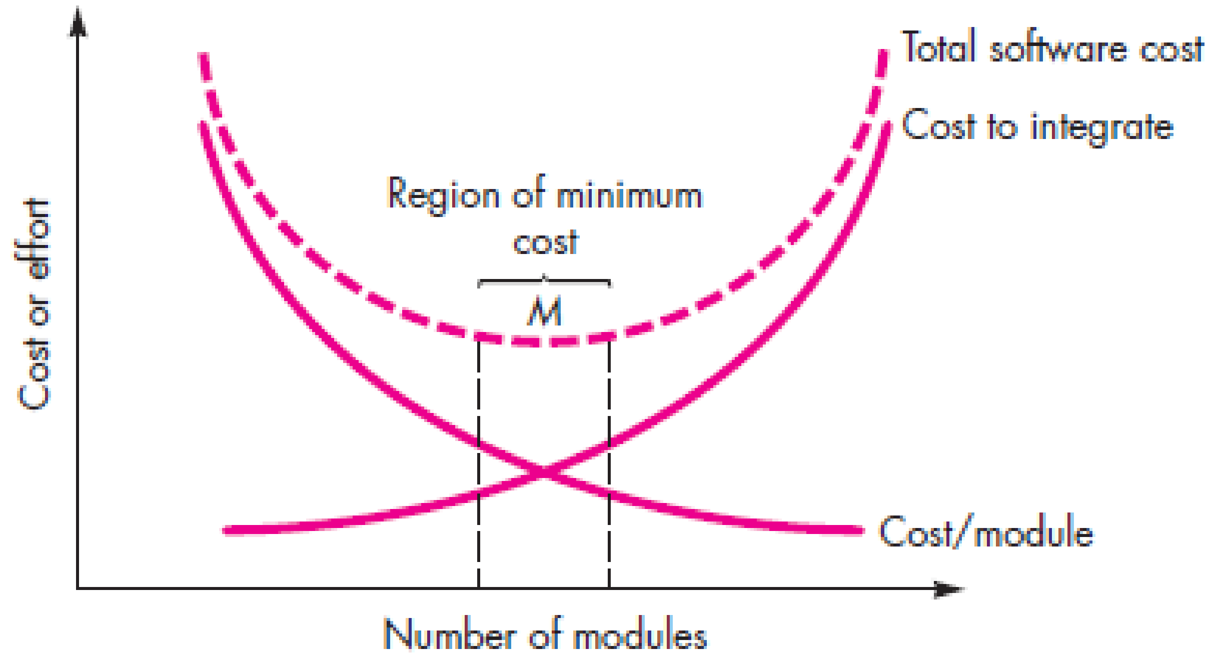
❑Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.

❑The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

❑In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

Modularity

- Referring to Figure, the effort (cost) to develop an individual software module does decrease as the total number of modules increases.



- Given the same set of requirements, more modules means smaller individual size.
- However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows.

Modularity

❑These characteristics lead to a total cost or effort curve shown in the figure.

❑There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

❑The curves shown in Figure do provide useful qualitative guidance when modularity is considered.

❑You should modularize, but care should be taken to stay in the vicinity of M.

❑Undermodularity or overmodularity should be avoided.

❑But how do you know the vicinity of M?

❑You modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

Modularity

- These characteristics lead to a total cost or effort curve shown in the figure.
- There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.
- The curves shown in Figure do provide useful qualitative guidance when modularity is considered.
- You should modularize, but care should be taken to stay in the vicinity of M.
- Undermodularity or overmodularity should be avoided.
- But how do you know the vicinity of M?
- You modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

Refinement

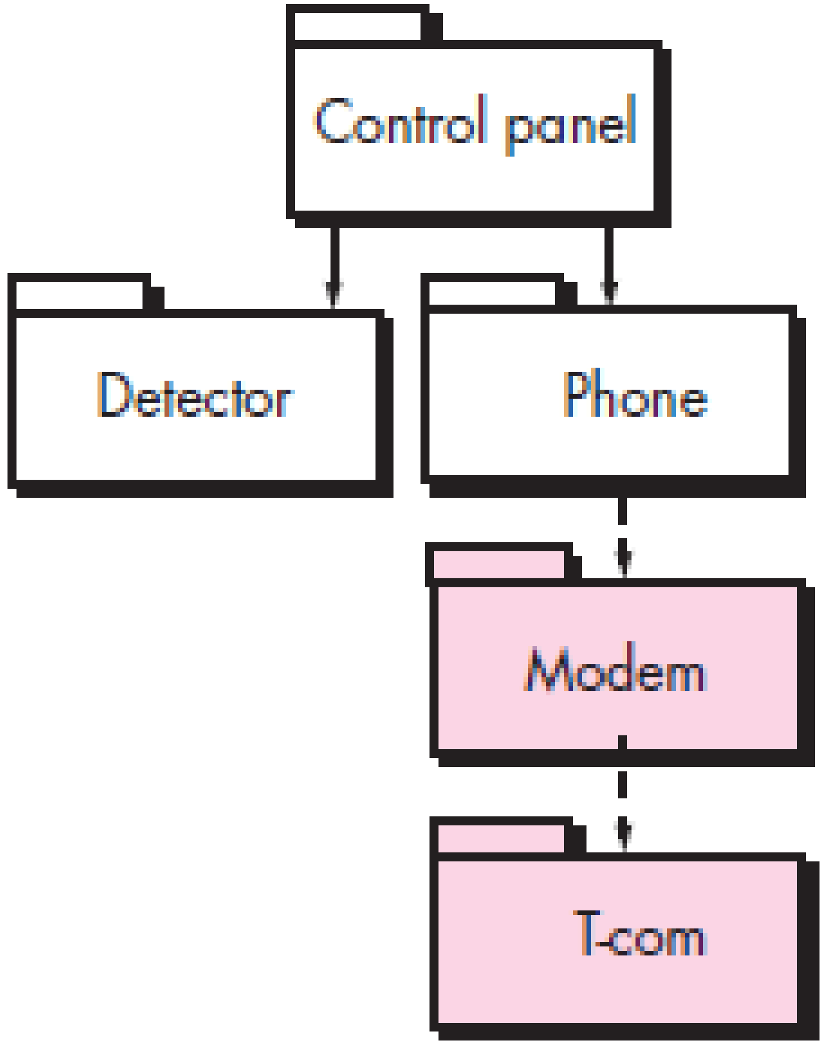
- ❑Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth [Wir71].
- ❑A program is developed by successively refining levels of procedural detail.
- ❑A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached. Refinement is actually a process of elaboration.
- ❑You begin with a statement of function (or description of information) that is defined at a high level of abstraction.
- ❑That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information.
- ❑You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Refinement

- ❑Abstraction and refinement are complementary concepts.
- ❑Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details.
- ❑Refinement helps you to reveal low-level details as design progresses.
- ❑Both concepts allow you to create a complete design model as the design evolves.

Cohesion

- ❑Cohesion is the "single-mindedness" of a component.
- ❑Within the context of component-level design for object-oriented systems, *cohesion* implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.



Cohesion

- Lethbridge and Laganiére [Let01] define a number of different types of cohesion (listed in order of the level of the cohesion):

1. **Functional.** Exhibited primarily by operations, this level of cohesion occurs when a component performs a targeted computation and then returns a result.
2. **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. Consider, for example, the SafeHome security function requirement to make an outgoing phone call if an alarm is sensed. It might be possible to define a set of layered packages as shown in Figure. The shaded packages contain infrastructure components. Access is from the control panel package downward.
3. **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.

Cohesion

- ❑Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain.
- ❑You should strive to achieve these levels of cohesion whenever possible.
- ❑It is important to note, however, that pragmatic design and implementation issues sometimes force you to opt for lower levels of cohesion.

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 [Let01] define the following coupling categories:

1. **Content coupling.** Occurs when one component “surreptitiously modifies data that is internal to another component” [Let01]. This violates information hiding—a basic design concept.
2. **Common coupling.** Occurs when a number of components all make use of a global variable. Although this is sometimes necessary (e.g., for establishing default values that are applicable throughout an application), common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.

Coupling

3. **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.
4. **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.
5. **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.
6. **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.

Coupling

7. **Type use coupling.** Occurs when component A uses a data type defined in component B (e.g., this occurs whenever “a class declares an instance variable or a local variable as having another class for its type” [Let01]). If the type definition changes, every component that uses the definition must also change.
8. **Inclusion or import coupling.** Occurs when component A imports or includes a package or the content of component B.
9. **External coupling.** Occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

Architectual Design

❑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
3. Reduce the risks associated with the construction of the software.

Architectural Design

- ❑This definition emphasizes the role of "software components" in any architectural representation.
- ❑In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers.
- ❑The properties of components are those characteristics that are necessary for an understanding of how the components interact with other components.
- ❑At the architectural level, internal properties (e.g., details of an algorithm) are not specified.
- ❑The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

Architectural Design

❑**Why Is Architecture Important?**

❑Bass and his colleagues [Bas03] identify three key reasons that software architecture is important:

1. Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
2. 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.
3. Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together”.

Architectural Design

❑Architectural Design Styles

❑The software that is built for computer-based systems 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.
4. Semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

Architectural Design

❑**Architectural Design Styles**

❑Although millions of computer-based systems have been created over the past 60 years, the vast majority can be categorized into one of a relatively small number of architectural styles:

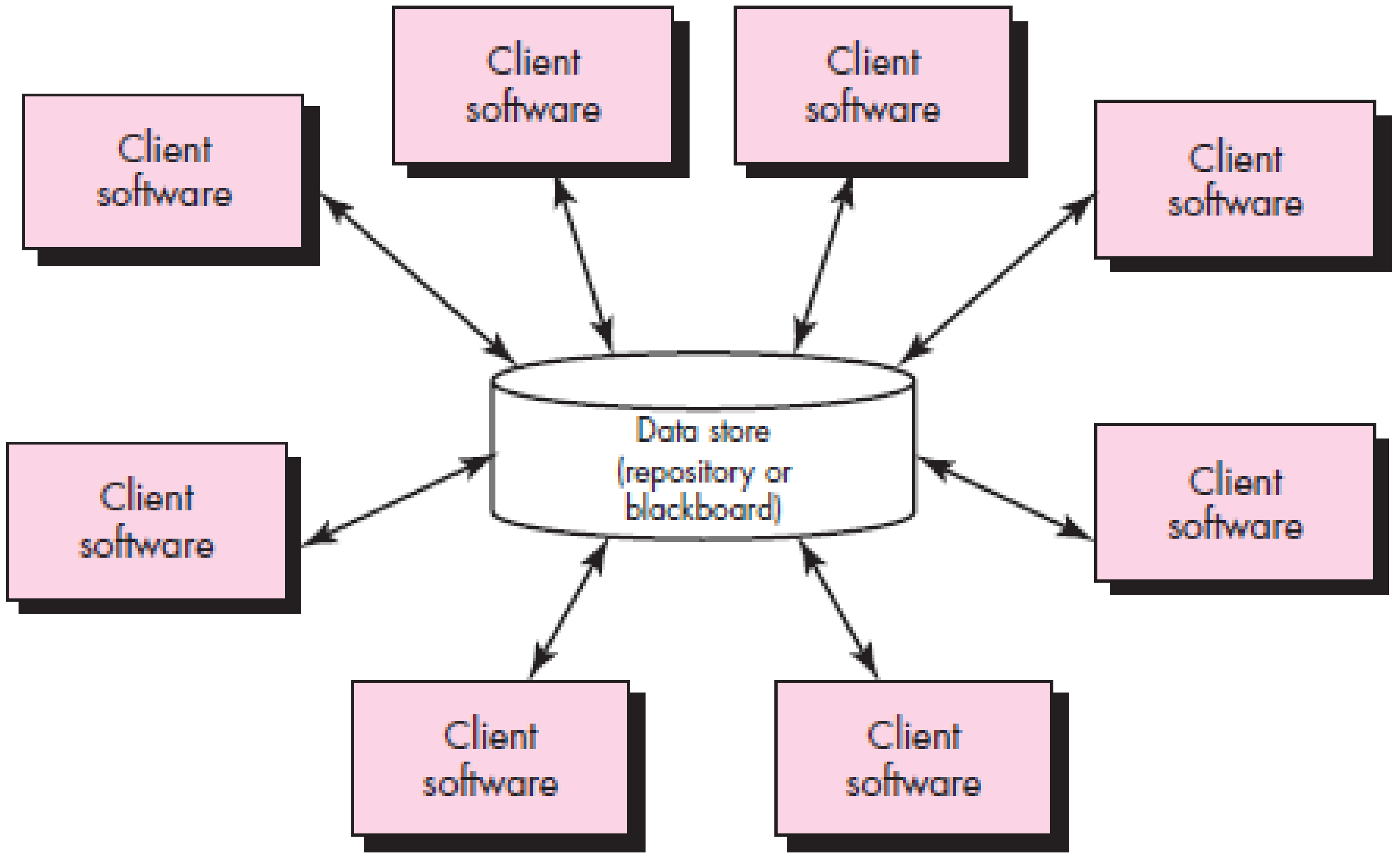
1. **Data-centered architectures**
2. **Data-flow architectures**
3. **Call and return architectures**
4. **Object-oriented architectures**
5. **Layered architectures**

Architectural Design

- ❑**Architectural Design 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.
- ❑Figure illustrates a typical data-centered style.
- ❑Client software accesses a central repository. In some cases the data repository is passive.
- ❑That is, client software accesses the data independent of any changes to the data or the actions of other client software.
- ❑A variation on this approach transforms the repository into a “blackboard”

Architectural Design

❑Architectural Design Styles: Data-centered architectures:

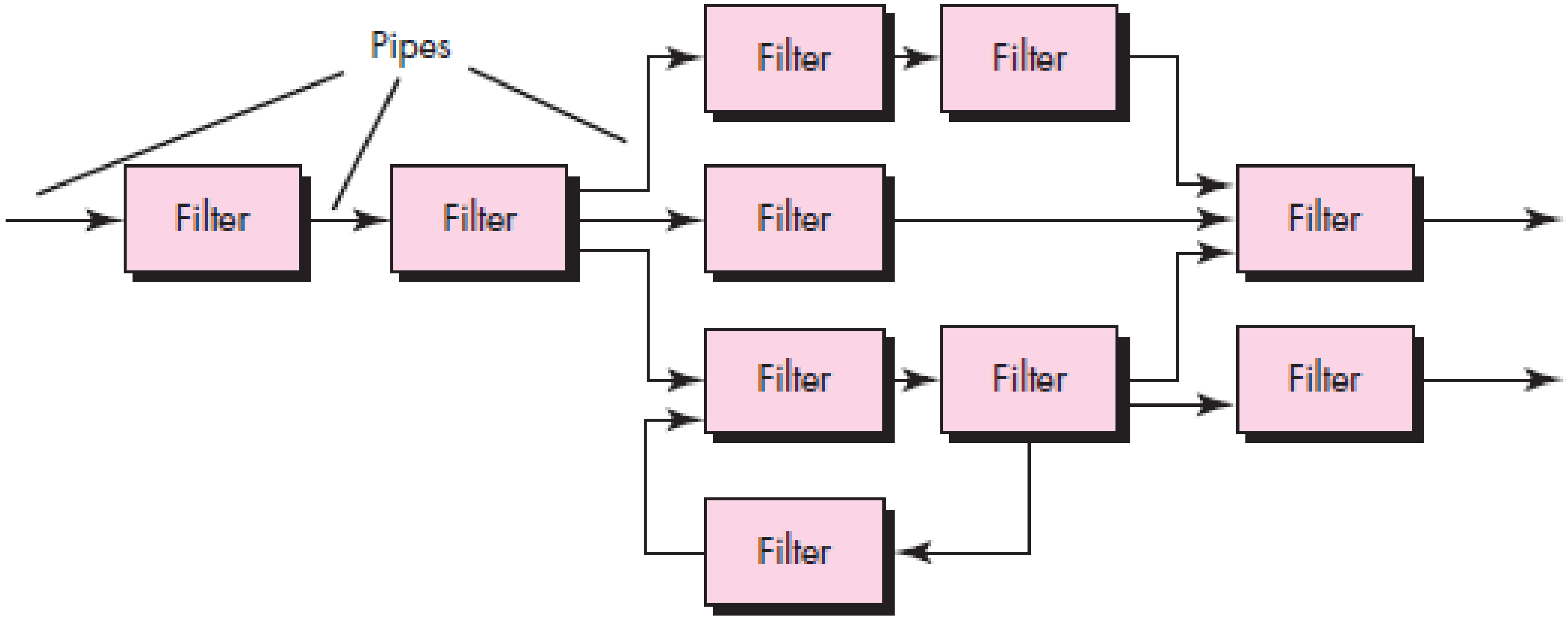


Architectural Design

- **Architectural Design Styles: 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.
- A pipe-and-filter pattern (Figure) 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 (to the next filter) of a specified form.
- However, the filter does not require knowledge of the workings of its neighboring filters.

Architectural Design

❑Architectural Design Styles: Data-flow architectures



Pipes and filters

Architectural Design

❑**Architectural Design Styles : Call and return architectures**

❑This architectural style enables you to achieve a program structure that is relatively easy to modify and scale.

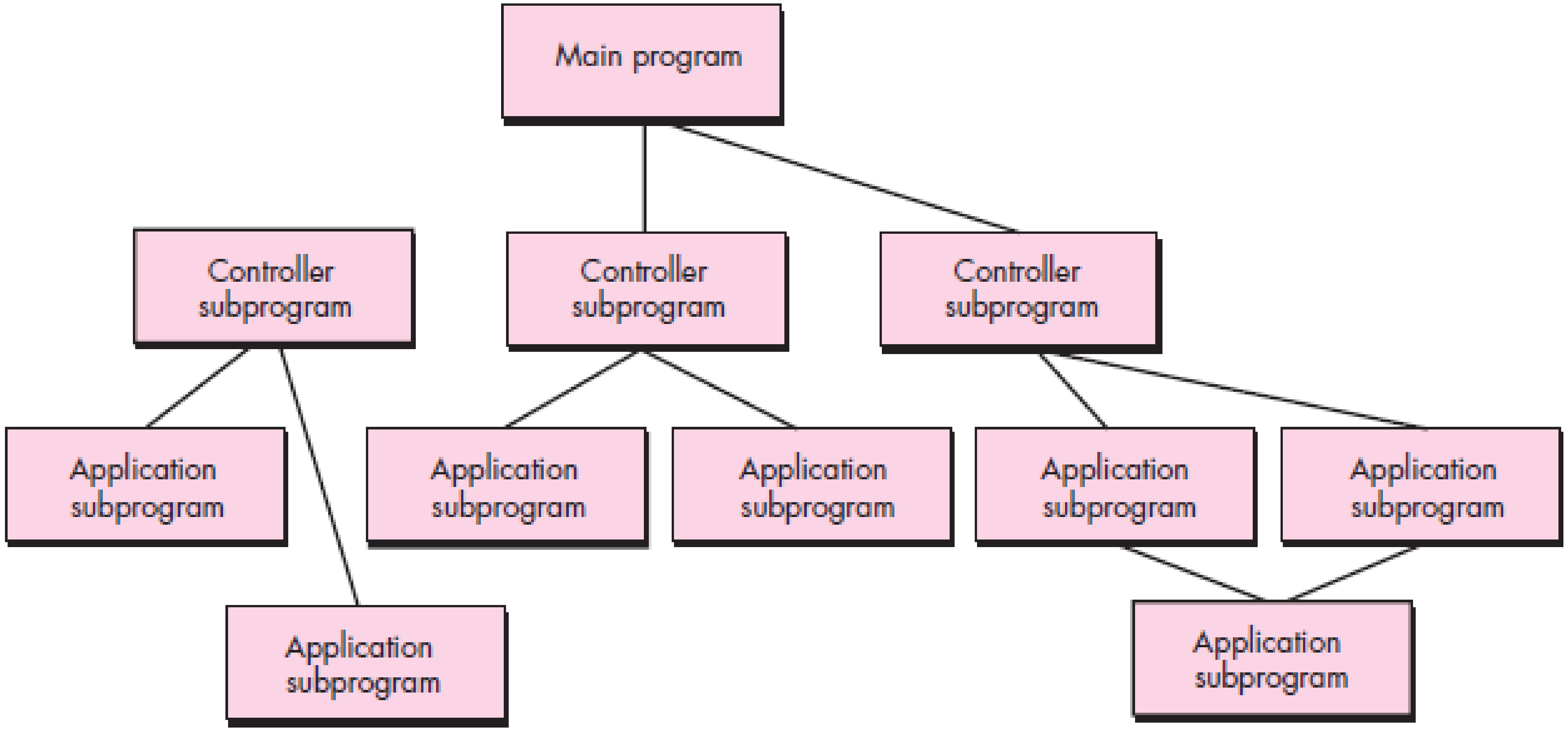
❑A number of substyles exist within this category:

▪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, figure illustrates an architecture of this type.

▪Remote procedure call architectures. The components of a main program/subprogram architecture are distributed across multiple computers on a network.

Architectural Design

❑Architectural Design Styles : Call and return architectures



Architectural Design

❑**Architectural Design Styles :Object-oriented architectures**

❑The components of a system encapsulate data and the operations that must be applied to manipulate the data.

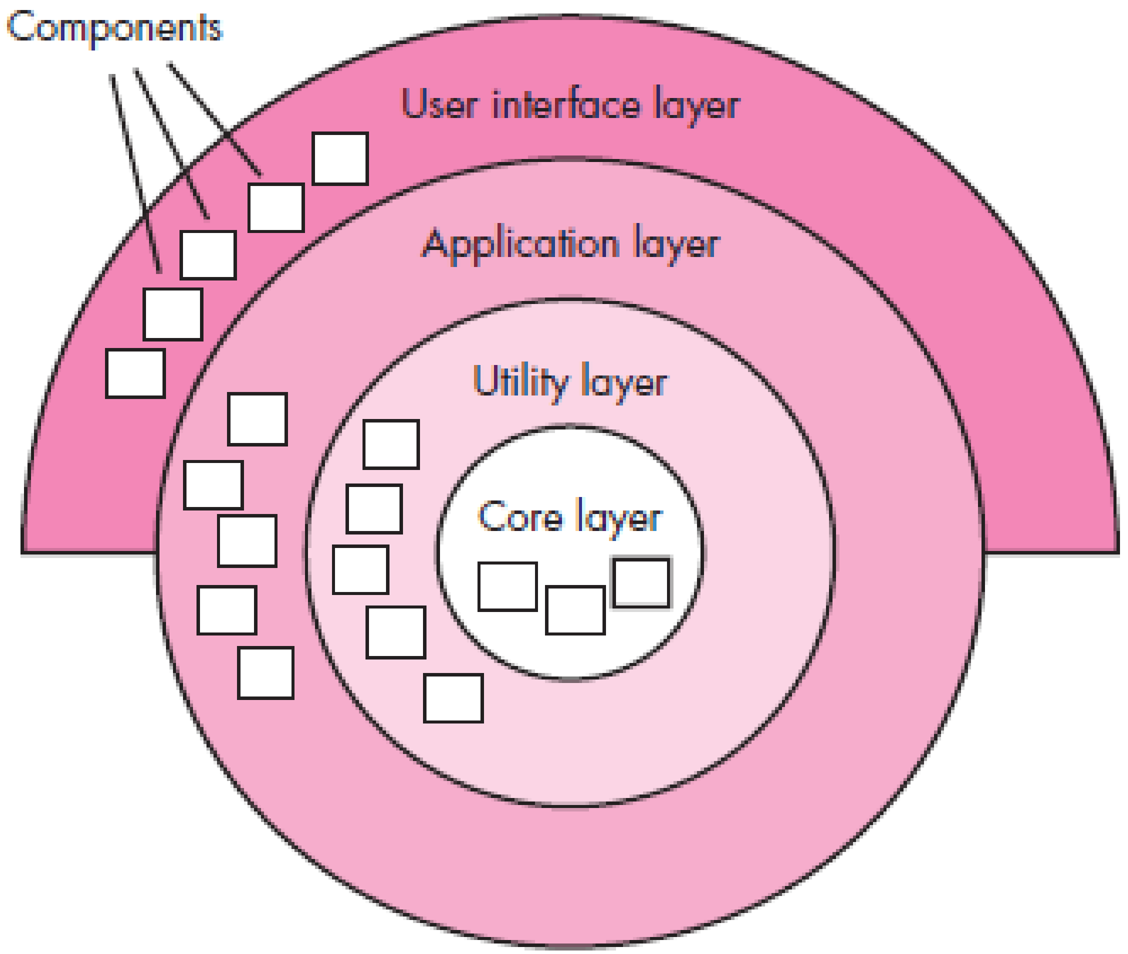
❑Communication and coordination between components are accomplished via message passing.

Architectural Design

- **Architectural Design Styles: Layered architectures**
- The basic structure of a layered architecture is illustrated in 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 Design

❑Architectural Design Styles: Layered architectures



Architectural Design

❑Factors to Consider in Architectural Design:

1. **Scalability:** Modern web applications need to be scalable, to handle the growing traffic and ensure business needs are not impacted by a huge customer base. Scalability techniques should be incorporated into the architecture during the design, code, and infrastructure phases.
2. **Security:** The world is going digitized, so there is a need to protect and secure user’s sensitive information and overall data. Security procedures and practices via configuration should be incorporated into architectural design.
3. **Maintainability:** The architecture of the software product must be modified with ease in order to correct defects, adapt to new changes, and make future maintenance easier.
4. **Performance:** The architecture should be designed in such a way that the system can perform under a particular workload. Prior to the beginning of coding engineering practices strive to build performance into the design and architecture of a system.

Detailed Design Transaction Transformation (Transaction Control Transformation)

- ❑A Transaction Control transformation is an active and connected transformation.
- ❑It allows us to commit and rollback transactions based on a set of rows that pass through a Transaction Control transformation.
- ❑Commit and rollback operations are of significant importance as it guarantees the availability of data.
- ❑A transaction is the set of rows bound by commit or rollback rows.
- ❑We can define a transaction based on the varying number of input rows.
- ❑We can also identify transactions based on a group of rows ordered on a common key, such as employee ID or order entry date.
- ❑When processing a high volume of data, there can be a situation to commit the data to the target.
- ❑If a commit is performed too quickly, then it will be an overhead to the system.

Detailed Design Transaction Transformation (Transaction Control Transformation)

❑If a commit is performed too late, then in the case of failure, there are chances of losing the data.

❑So the Transaction control transformation provides flexibility.

❑In PowerCenter, the transaction control transformation is defined in the following levels, such as:

1. **Within a mapping:** Within a mapping, we use the Transaction Control transformation to determine a transaction. We define transactions using an expression in a Transaction Control transformation. We can choose to commit, rollback, or continue on the basis of the return value of the expression without any transaction change.
2. **Within a session:** We configure a session for the user-defined commit. If the Integration Service fails to transform or write any row to the target, then We can choose to commit or rollback a transaction.

Detailed Design Transaction Transformation (Transaction Control Transformation)

- ❑When we run the session, then the Integration Service evaluates the expression for each row that enters the transformation.
- ❑When it evaluates a committed row, then it commits all rows in the transaction to the target or targets.
- ❑When the Integration Service evaluates a rollback row, then it rolls back all rows in the transaction from the target or targets.
- ❑If the mapping has a flat-file as the target, then the integration service can generate an output file for a new transaction each time.
- ❑We can dynamically name the target flat files.
- ❑Here is the example of creating flat files dynamically - Dynamic flat-file creation.

Detailed Design Transaction Transformation (Transaction Control Transformation)

- TCL COMMIT & ROLLBACK Commands
- There are five in-built variables available in the transaction control transformation to handle the operation.

1. **TC_CONTINUE_TRANSACTION:** The Integration Service does not perform any transaction change for the row. This is the default value of the expression.
2. **TC_COMMIT_BEFORE:**The Integration Service commits the transaction, begins a new transaction, and writes the current row to the target. The current row is in the new transaction. In tc_commit_before, when this flag is found set, then a commit is performed before the processing of the current row.
3. **TC_COMMIT_AFTER:** The Integration Service writes the current row to the target, commits the transaction, and begins a new transaction. The current row is in the committed transaction. In tc_commit_after, the current row is processed then a commit is performed.

Detailed Design Transaction Transformation (Transaction Control Transformation)

❑TCL COMMIT & ROLLBACK Commands

❑There are five in-built variables available in the transaction control transformation to handle the operation.

4. **TC_ROLLBACK_BEFORE:** The Integration Service rolls back the current transaction, begins a new transaction, and writes the current row to the target. The current row is in the new transaction. In tc_rollback_before, rollback is performed first, and then data is processed to write.
5. **TC_ROLLBACK_AFTER:** The Integration Service writes the current row to the target, rollback the transaction, and begins a new transaction. The current row is in the rolled-back transaction. In tc_rollback_after data is processed, then the rollback is performed.

Detailed Design Transaction Transformation (Transaction Control Transformation)

❑Steps to Create Transaction Control Transformation

- **Step 1**: Go to the mapping designer.
- **Step 2**: Click on transformation in the toolbar, and click on the **Create** button.
- **Step 3**: Select the transaction control transformation.
- **Step 4**: Then, enter the name and click on the **Create** button.
- **Step 5**: Now click on the **Done** button.
- **Step 6**: We can drag the ports into the transaction control transformation, or we can create the ports manually in the ports tab.
- **Step 7**: Go to the properties tab.
- **Step 8**: And enter the transaction control expression in the Transaction Control Condition.

Detailed Design Transaction Transformation (Transaction Control Transformation)

- Configuring Transaction Control Transformation: Here are the following components which can be configuring in the transaction control transformation, such as:

I. **Transformation Tab**: It can rename the transformation and add a description.

II. **Ports Tab**: It can create input or output ports.

III. **Properties Tab**: It can define the transaction control expression and tracing level.

IV. **Metadata Extensions Tab**: It can add metadata information.

Refactoring of Design

- ❑An important design activity suggested for many agile methods, refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
- ❑Fowler [Fow00] defines refactoring in the following manner: “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”

Refactoring of Design

❑When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

❑For example, a first design iteration might yield a component that exhibits low cohesion (i.e., it performs three functions that have only limited relationship to one another).

❑After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion.

❑The result will be software that is easier to integrate, easier to test, and easier to maintain.

Object Oriented Design

❑In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities).

❑The state is distributed among the objects, and each object handles its state data.

❑For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data.

❑The tasks defined for one purpose cannot refer or change data of other objects.

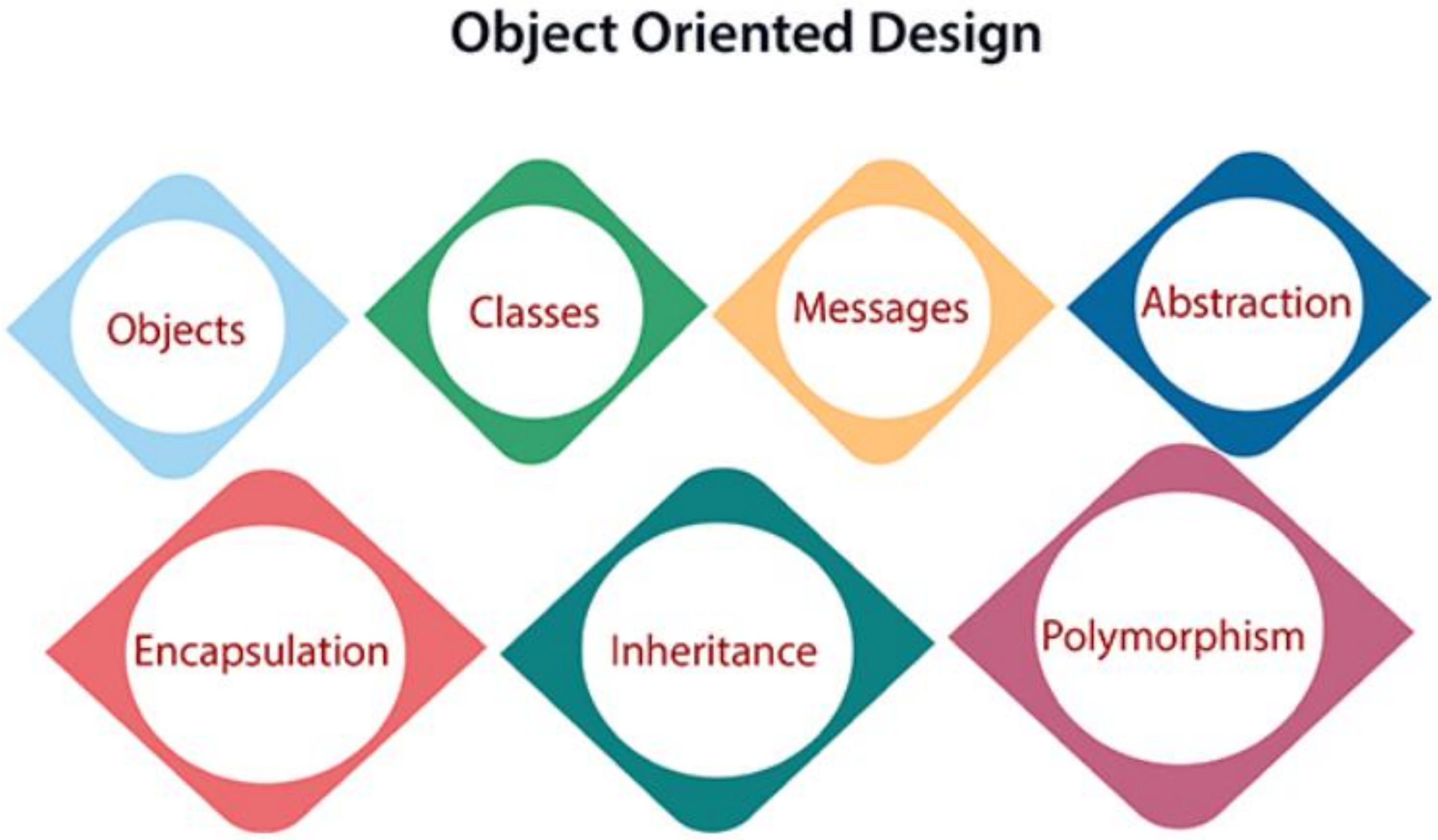
❑Objects have their internal data which represent their state. Similar objects create a class.

❑In other words, each object is a member of some class.

❑Classes may inherit features from the superclass.

Object Oriented Design

❑The different terms related to object design are:



Object Oriented Design

❑**The different terms related to object design are:**

1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.
2. **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.
3. **Messages:** Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.

Object Oriented Design

❑**The different terms related to object design are:**

4. **Abstraction:** In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.
5. **Encapsulation:** Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.
6. **Inheritance:** OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses.This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.

Object Oriented Design

❑**The different terms related to object design are:**

7. **Polymorphism:** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types. Depending upon how the service is invoked, the respective portion of the code gets executed.

Note for Students

- **This power point presentation is for lecture, therefore it is suggested that also utilize the text books and lecture notes.**