Week 5

CS2212
Introduction to Software Engineering

Design Concepts & Modeling

Some images and content from Software Engineering: A Practitioner's Approach 9th Edition By Roger Pressman and Bruce Maxim

Software Design

- **Mitch Kapor**, the creator of Lotus 1-2-3, presented a “software design manifesto” in **Dr. Dobbs Journal** in the early 1990s. He said:
  - **Good software design should exhibit:**
    - **Firmness:** A program should not have any bugs that inhibit its function.
    - **Commodity:** A program should be suitable for the purposes for which it was intended.
    - **Delight:** The experience of using the program should be pleasurable one.

Software Design

- **Software Design** encompasses the set of **principles**, **concepts**, and **practices** that lead to the development of a **high-quality** system.

- **Design Principles:** overriding philosophy that guides the designer.

- **Design Concepts:** concepts that must be understood before **design practices** are applied.

- **Design Practices:** practices that lead to the creation of various representations (e.g. **design models**) of the software.

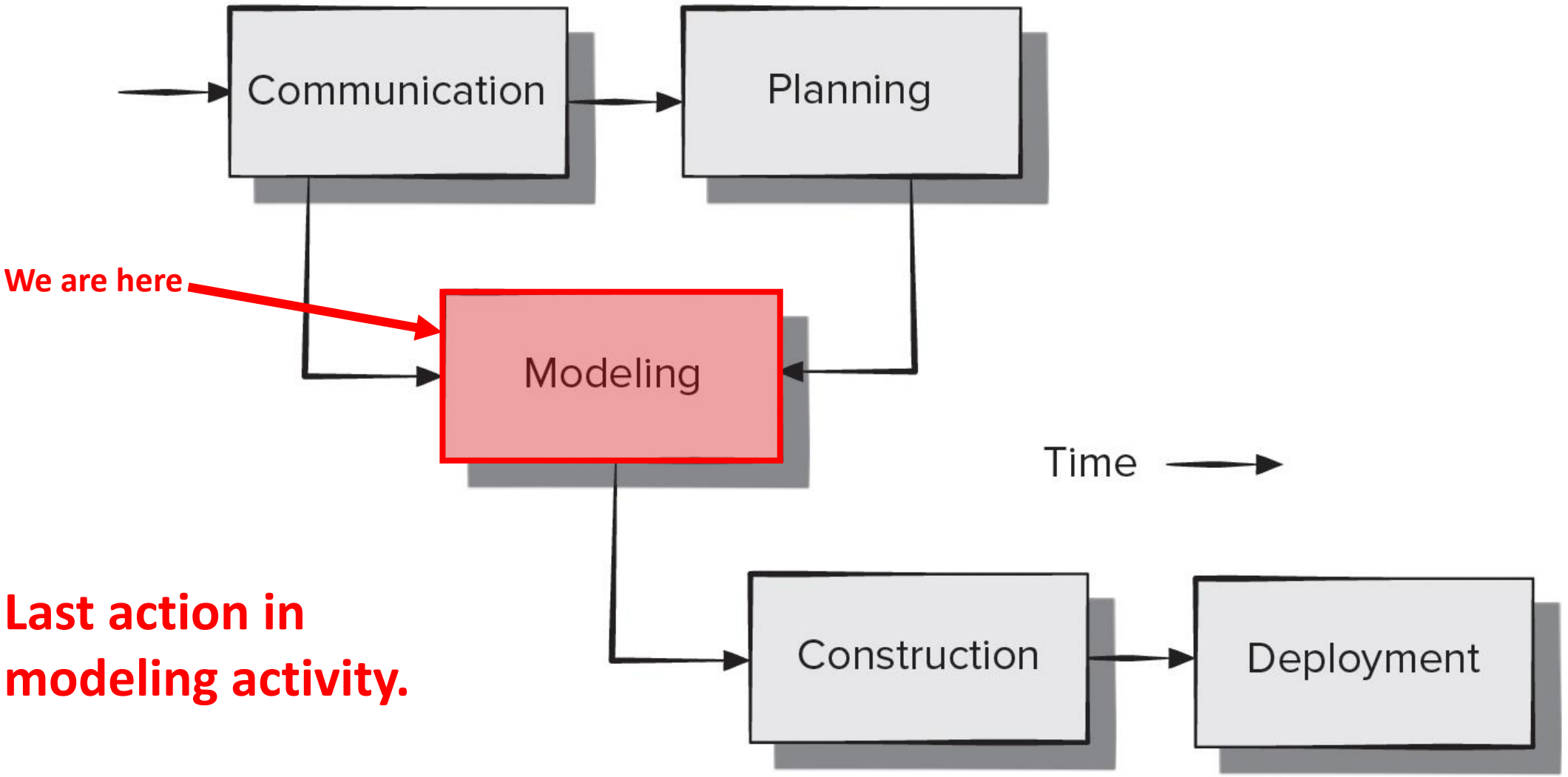
Why Care About Design?

- We have requirements and use cases, why not just start coding?

- **Technical Debt:** Costs associated with rework caused by choosing *"quick and dirty"* solution rather than better approach that takes more time.

- Pay down technical debt by **Refactoring**.

Software Design



Last action in modeling activity.

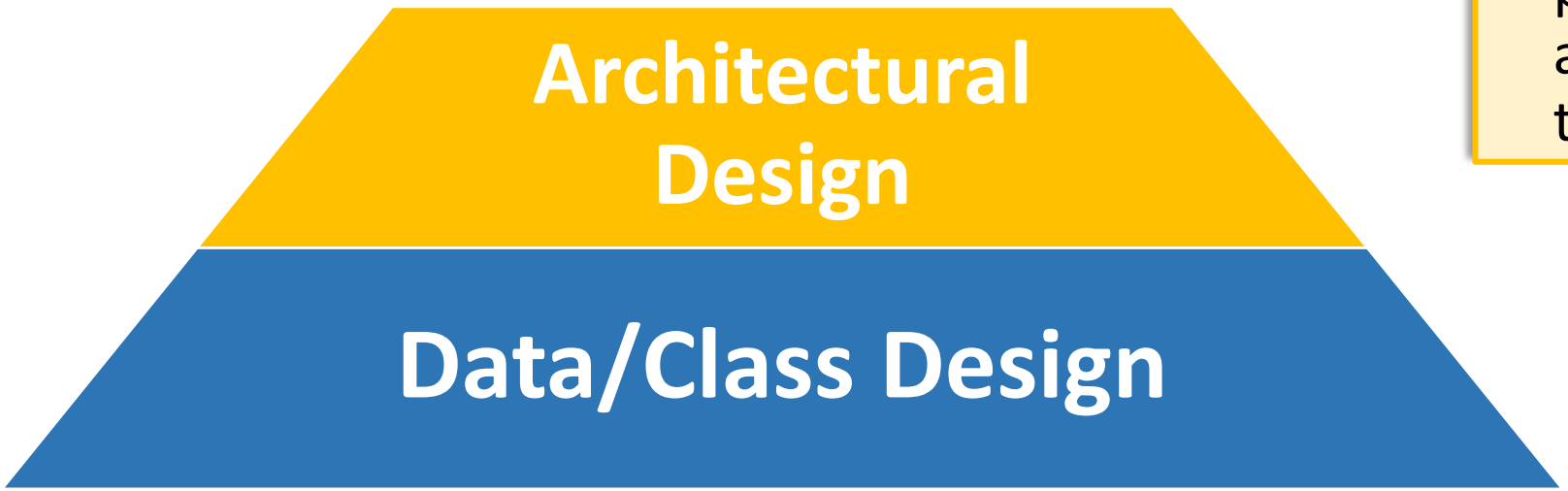
Software Engineering Design

Data/Class Design

- Transforms **analysis classes** into implementation classes and data structures (**design classes**).
- UML **Class Diagrams** and Entity-relationship diagrams (ERDs) often used (especially for databases).

Data/Class Design

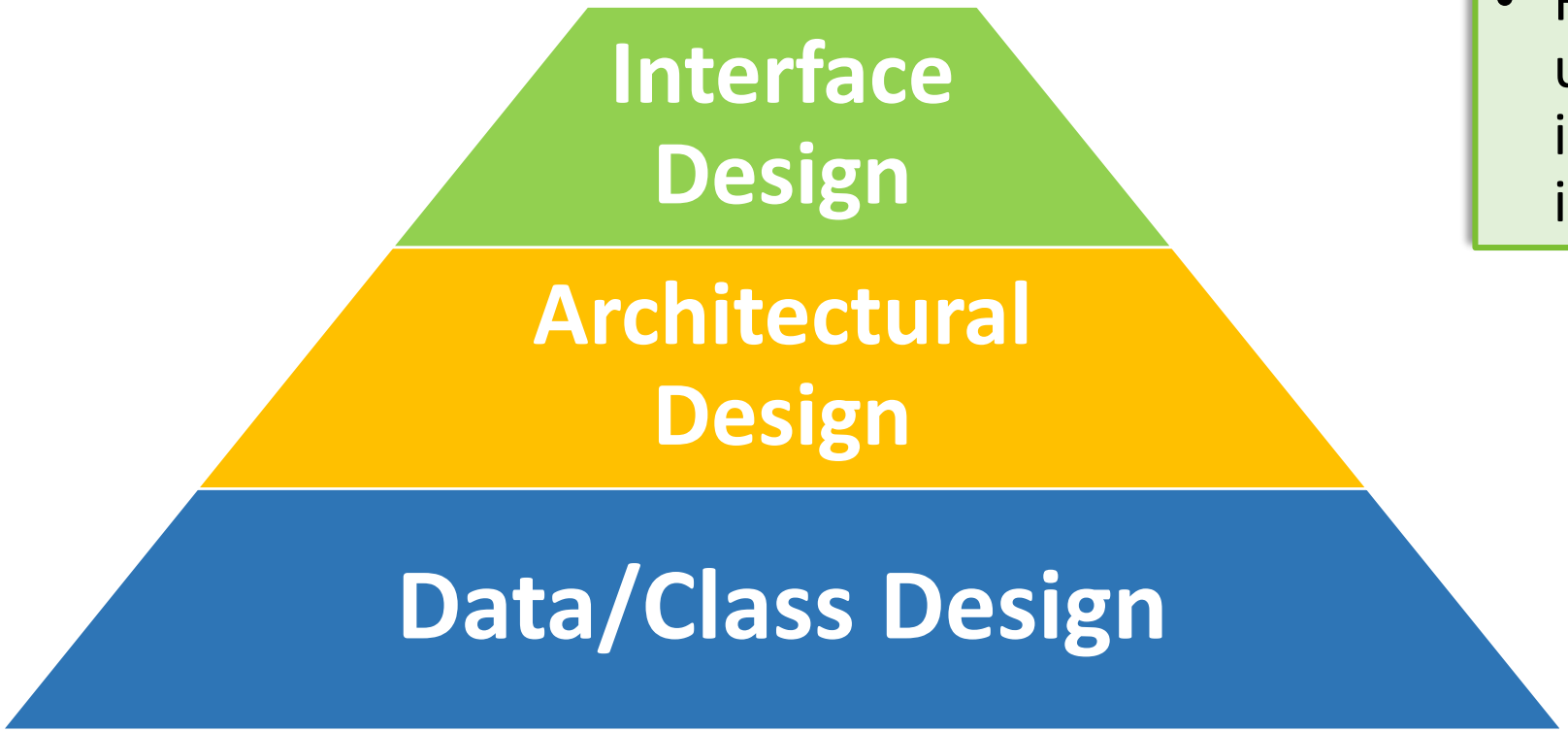
Software Engineering Design



Architectural Design

- Defines relationships among the major software structural elements.
- Decomposes the system into interacting components.
- Defines the structure and properties of the components and the relationship among these components.

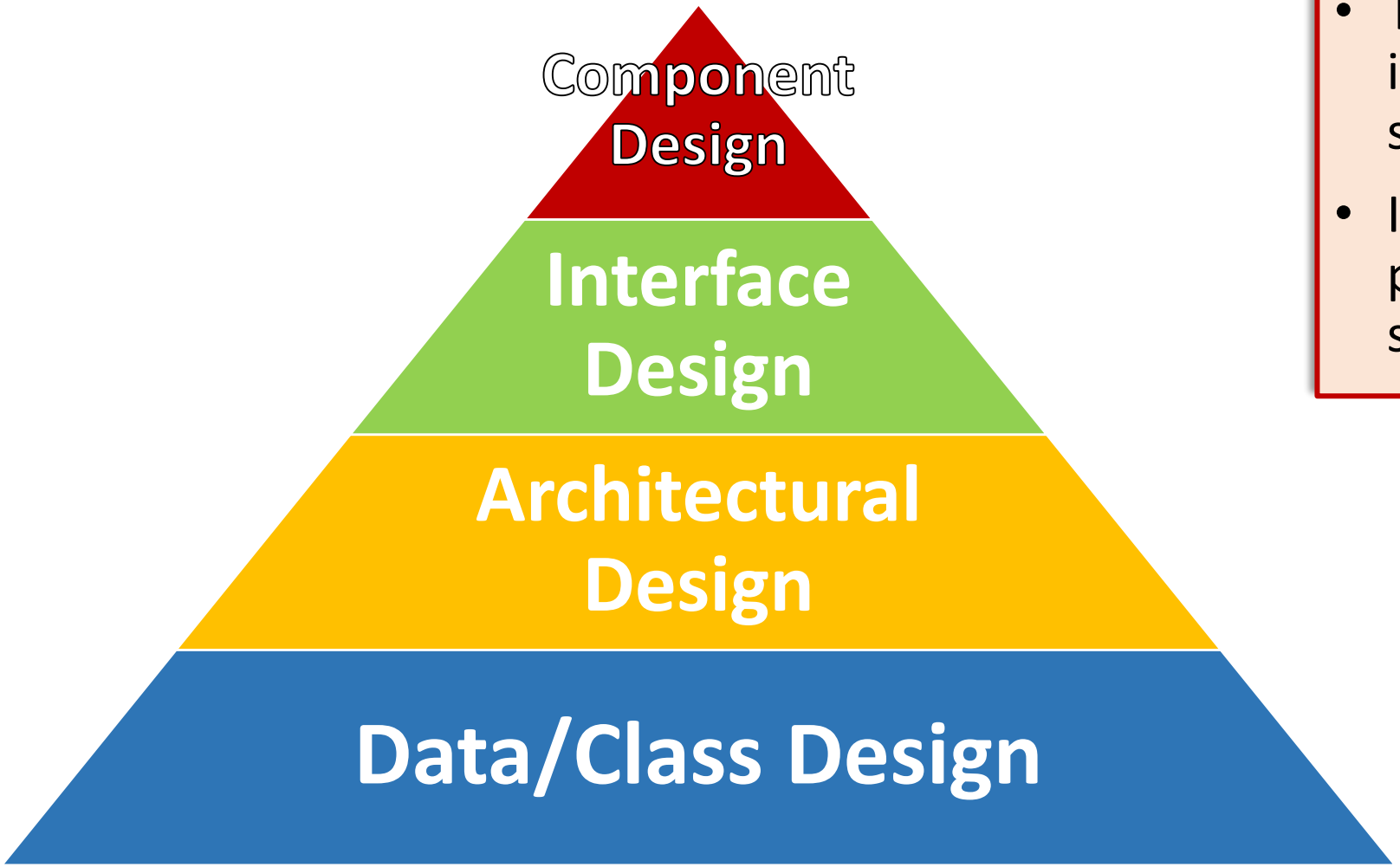
Software Engineering Design



Interface Design

- Defines how software elements, hardware elements, and end-users communicate.
- Focuses on user interfaces (UI), user experience (UX), internal interfaces, and external interfaces.

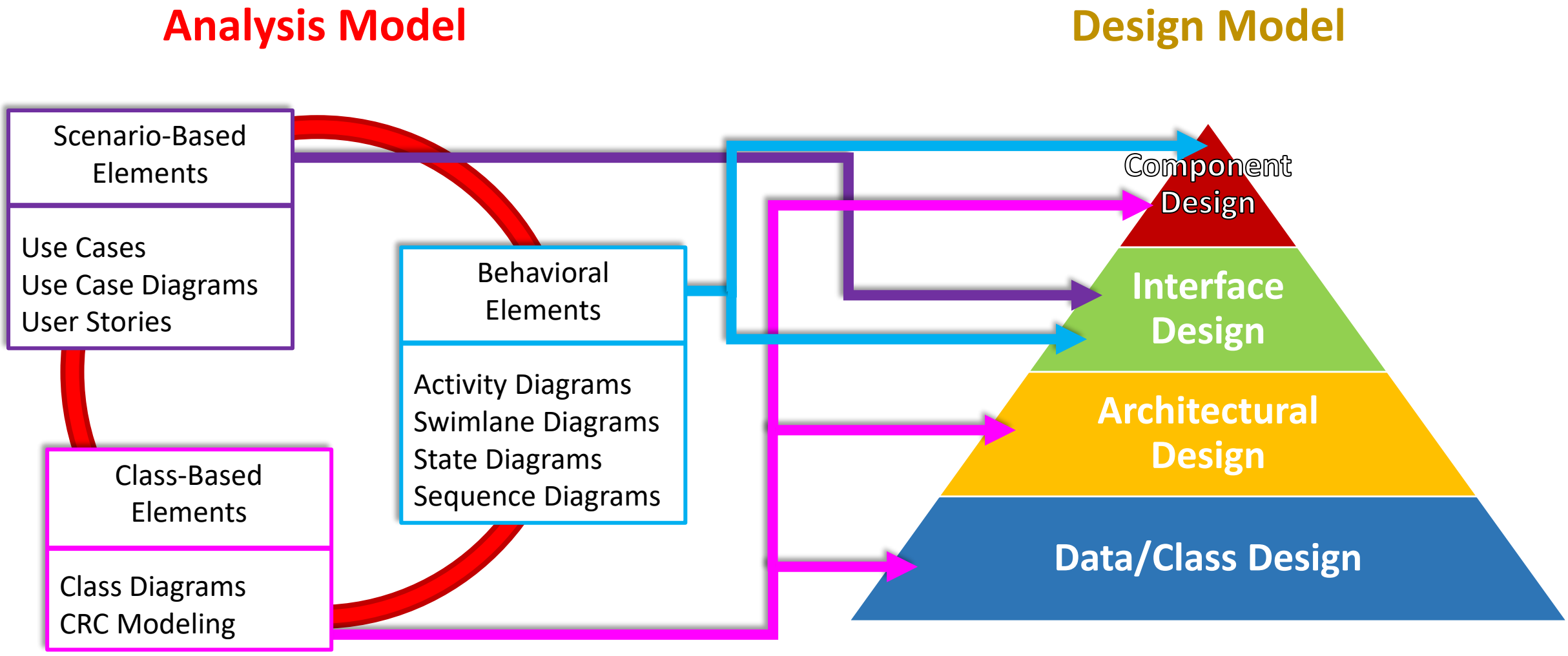
Software Engineering Design



Component-Level Design

- Transforms structural elements into procedural descriptions of software components.
- Internal data structures and processing details of all the software components.

Software Engineering Design



Software Engineering Design

- These models and designs become the **blueprint** for the **Construction Activity**.
- A good design leads to a **high-quality** software product.
- But how do we ensure our design is good?

Design and Quality

- Throughout the design process, the **quality** of the evolving design must be assessed.

- Three recommended characteristics that serve as a guide for the evaluation of a **good design:**

1. The design must implement **all explicit requirements** and must accommodate **all implicit requirements** desired by the customer.

2. The design must be a **readable**, **understandable** guide for those who **generate code** and for those who **test and support** the software.

3. The design should provide a **complete picture of the software**, addressing the data, functional, and behavioral domains from an implementation perspective.

Design and Quality

- These three characteristics are the **goals of the design process**.
- How do we achieve them?

Quality Guidelines

1. A design should **exhibit an architecture** (a) created using recognizable architectural styles or patterns, (b) composed of well designed components (c) implemented in an evolutionary fashion.
2. A design should be **modular**.
3. A design should contain **distinct representations** of data, architecture, interfaces, and components.
4. A design should lead to **data structures** that are drawn from recognizable data patterns.
5. A design should contain **functionally independent components**.
6. A design should lead to **interfaces that reduce the complexity of connections** between components and the external environment.
7. A design should be derived using a **repeatable method** that is driven by software requirements analysis.
8. A design should be represented using **meaningful notation**.

Quality Guidelines

These design guidelines are not achieved by chance; they are achieved through the **application of fundamental design principles**, **systematic methodology**, and **thorough review**.

Quality Guidelines

One method of conducting a **Thorough Review** is through **Technical Reviews (TR)**:

- **Technical Review**: Meeting conducted by members of the software team.
- 2 to 4 people, one **review leader** (facilitates the meeting), one **recorder** (takes notes), one **producer** (person whose work product is being reviewed).
- Review **work product** and note all **errors**, **omissions**, or **ambiguity**.
- Decided if further work is needed by **producer** before design work can be approved.

Design Concepts

Design Concepts: concepts that must be understood before **design practices** are applied.

Abstraction

- Conceptual process wherein **general rules and concepts** are derived from the usage and classification of specific examples, literal signifiers, first principles, or other methods.

Highest Level of Abstraction

Thing/Object

Animal

Bird

Waterfowl

Duck

Lowest Level of Abstraction

Mallard Duck

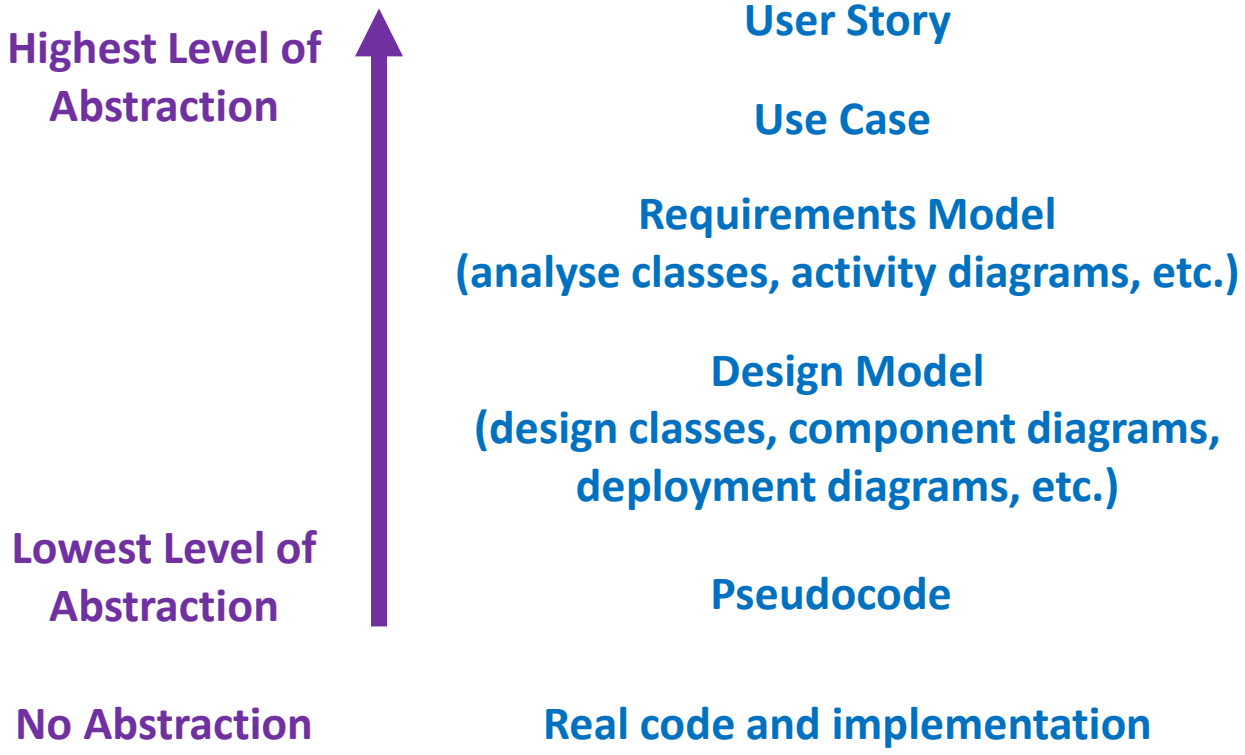
No Abstraction

Frank the Duck, Age 5, Sex Male

Design Concepts

Abstraction

- Conceptual process wherein **general rules and concepts** are derived from the usage and classification of specific examples, literal signifiers, first principles, or other methods.



Design Concepts

Abstraction

- Conceptual process wherein **general rules and concepts** are derived from the usage and classification of specific examples, literal signifiers, first principles, or other methods.

Procedural Abstraction: Sequence of instructions that have specific and limited function.

Data Abstraction: Collection of data that describes a data object.

Procedural Example

- Word **use** for **camera** in *SafeHome* system.
- **Use** implies long sequence of procedural steps.
- E.g., activate the *SafeHome* system, log on, select a camera, locate camera controls, etc.

Design Concepts

Abstraction

- Conceptual process wherein **general rules and concepts** are derived from the usage and classification of specific examples, liter
other methods.

Procedural Abstraction: Sequence of instructions that have specific and limited function.

Data Abstraction: Collection of data that describes a data object.

Data Example

- From last example, we can define data abstraction for **camera**.
- Abstraction for **camera** would encompass a set of attributes that describe a **camera**.
- Example Attributes:
  - Camera ID
  - Location
  - Field of view
  - Pan angle
  - Zoom

Design Concepts

Separation of Concerns

- **Separation of concerns** suggests that a **complex problem can be more easily handled if it is subdivided** into pieces that can each be solved independently.
- A **concern** is a feature or behaviour that is part of the requirements model.
- By separating **concerns** into smaller pieces, a problem takes less effort and time to solve using a **divide-and-conquer** strategy

Design Concepts

Modularity

- **Modularity** is the most common manifestation of **separation of concerns**.
- Software is **divided into separately named components**, also referred to as **modules**, that are **integrated** to satisfy requirements.
- *"Modularity is the single attribute of software that allows a program to be intellectually manageable".*

Design Concepts

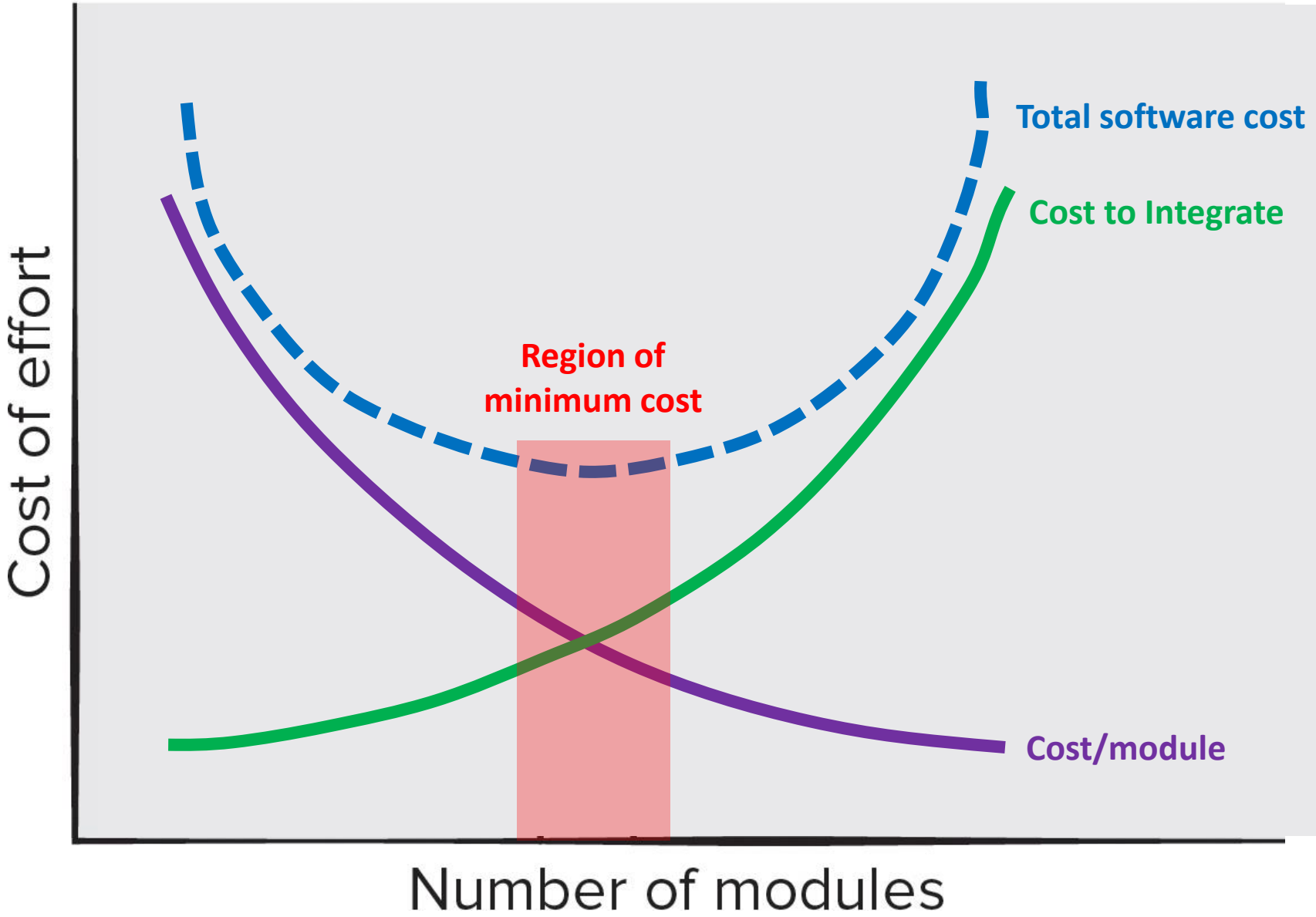
Modularity

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

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

- So can we reduce the cost to nothing by infinitely subdividing the problem?

Design Concepts



Design Concepts

Information Hiding

- The intent of **information hiding** is to **hide the details of data structures and procedural processing** behind a module interface.

- **Information hiding** implies that **effective modularity can be achieved by defining a set of independent modules** that communicate with one another **only when necessary**.

Design Concepts

Information Hiding

- **The benefits of information hiding include:**
  - Reducing the likelihood of ***"side effects".***
  - Limiting the global **impact of local design decisions**.
  - Emphasizing **communication through controlled interfaces**.
  - **Discouraging** the use of **global data**.
  - **Encapsulation**—an attribute of high quality design.
  - Generally **higher quality software**.

Design Concepts

Functional Independence

- **Functional Independence** is achieved by developing modules with ***"single-minded"*** **purpose** and an ***"aversion"*** **to excessive interaction** with other modules.
- Assessed on two characteristics:
  - **Cohesion:** is an indication of the relative **functional strength of a module.**
  - **Coupling:** is an indication of the relative **interdependence among modules.**

Design Concepts

Functional Independence

- **Functional Independence** is ach
  ***"single-minded"* purpose** and a
  interaction with other modules.

- Assessed on two characteristics:
  - **Cohesion:** is an indication of the relative **functional strength of a module.**
  - **Coupling:** is an indication of the relative **interdependence among modules.**

Cohesion

- A **cohesive module** performs a **single task**, requiring little interaction with other components in other parts of a program.
- Stated simply, a **cohesive module** should (ideally) **do just one thing.**
- Hard to fully achieve in real-life.

Design Concepts

Functional Independence

- **Functional Independence** is achieved by developing modules with ***"single-minded"*** **purpose** and an ***"aversion"*** **to excessive interaction** with other modules.
- Assessed on two characteristics:
  - **Cohesion:** is an indication of the relative **functional strength of a module.**
  - **Coupling:** is an indication of the relative **interdependence among modules.**

Design Concepts

Functional Independence

- **Functional Independence** is ach
  ***"single-minded"* purpose** and a
  interaction with other modules.

- Assessed on two characteristics:
  - **Cohesion:** is an indication of the relative **functional strength of a module.**
  - **Coupling:** is an indication of the relative **interdependence among modules.**

Coupling

- **Coupling** depends on the **interface complexity** between modules.
- Aim for **lowest possible coupling**.
- Simple connectivity results in software that is **easier to understand** and **less likely to propagate errors**.

Design Concepts

Stepwise Refinement

- **Stepwise Refinement** is a **top-down design strategy** originally proposed by **Niklaus Wirth**:
  - An application is developed by **successively refining levels of procedural detail.**
  - A hierarchy is developed by **decomposing** a macroscopic statement of a **function in a stepwise fashion** until programming language statements are reached.
- As such, **refinement** is a process of **elaboration**, successively adding more and more detail as it is applied.

Design Concepts

Stepwise Refinement

- **Abstraction** and **refinement** are complementary concepts.
- **Abstraction** enables you to specify data and procedure 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 **abstraction** and **refinement** allow you to create a complete design model as the design evolves over time.

Design Concepts

Stepwise Refinement

Example:

Apply a stepwise refinement approach to develop three different levels of procedural abstractions for the following problem:

Develop a check writer that given a numeric dollar amount, will print the amount in words normally required on a check.

Example Input: 1,456

Example Output: One Thousand Four Hundred Fifty-Six

Design Concepts

Stepwise Refinement

Refinement 1:

Write dollar amount in words.

Design Concepts

Stepwise Refinement

Refinement 2:

```
Procedure write_amount;
        Validate amount is within bounds;
        Parse to determine each dollar unit;
        Generate alpha representation;
end write_amount;
```

Refinement 3:

```
procedure write_amount;
        do while checks remain to be printed
                if dollar amount > upper amount bound then print "amount too large error”
                else set process flag true;
                endif;

                determine maximum significant digit;

                do while (process flag true and significant digits remain)
                        set for corresponded alpha phrase;
                        divide to determine whole number value;
                        concatenate partial alpha string;
                        reduce significant digit count by one;
                enddo;

                print alpha string;
        enddo;
end write_amount;
```

Group Work: Refine a Function

Apply a stepwise refinement approach to develop **three different levels** of procedural abstractions for the following problem:

Develop a function to return the first position of a given character in a string. This should be accomplished by looping through each character of the string.

Example Input: ‘D’, ‘Rubber Duck’

Example Output: 8

Example Input: ‘u’, ‘Rubber Duck’

Example Output: 2

Design Concepts

Refactoring

- **Refactoring** is a reorganization technique that **simplifies the design (or code)** of a component without changing its function or behaviour.

- **Martin Fowler** 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** can be an important design activity for many agile methods to ensure a sound design despite frequent additions, modifications, and evolutions.

Design Concepts

Refactoring

- 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.

Design Concepts

Design Classes

- Class-based **requirements modeling** defines a set of **analysis classes**.
- Each of these classes **describe some element of the problem domain**, focusing on **aspects of the problem that are user visible**.
- As the **design model** evolves, you will define a set of **design classes** that **refine** these **analysis classes** by providing **additional detail** that will enable the classes to be implemented.

Design Classes

Analysis Class

Pothole

idNumber
streetAddress
size
location
district
repairPriority

assign()
determineDistrict()
determinePriority()

Refinement

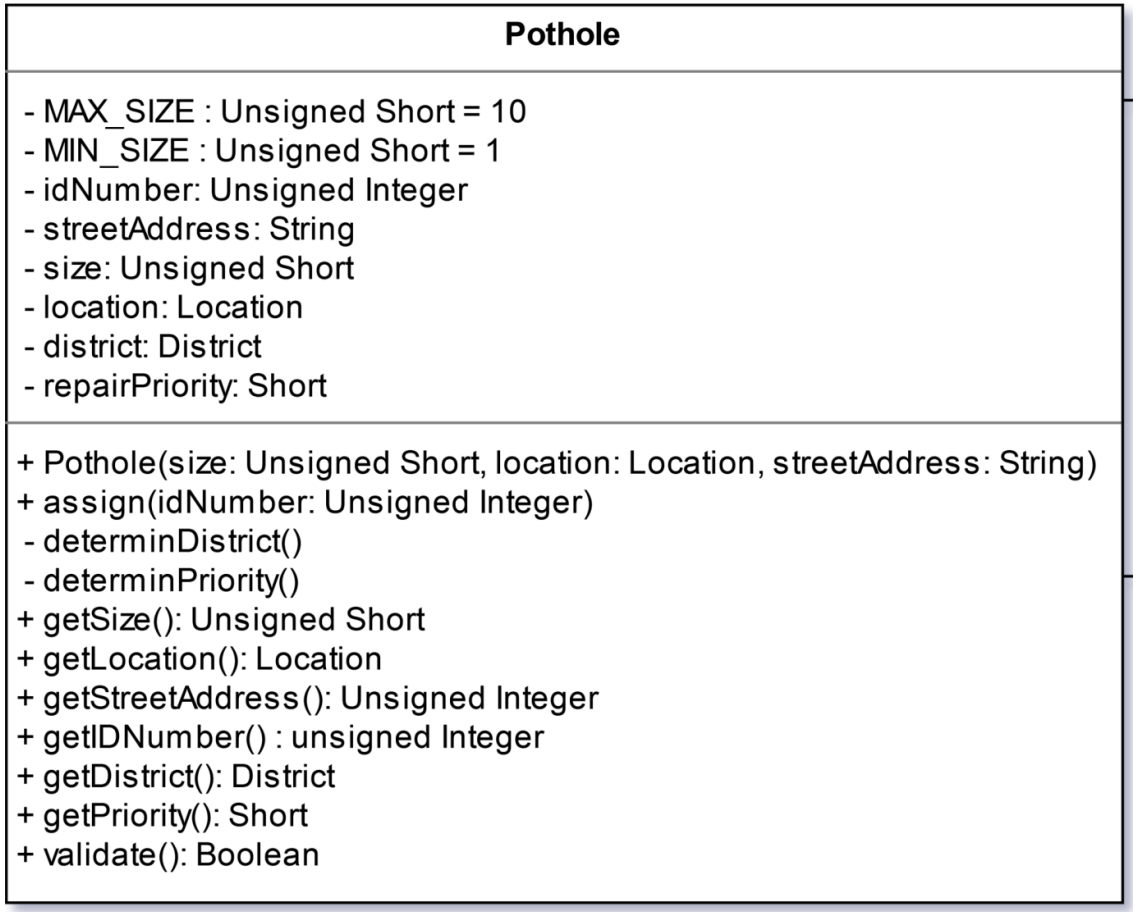
Design Class

Pothole

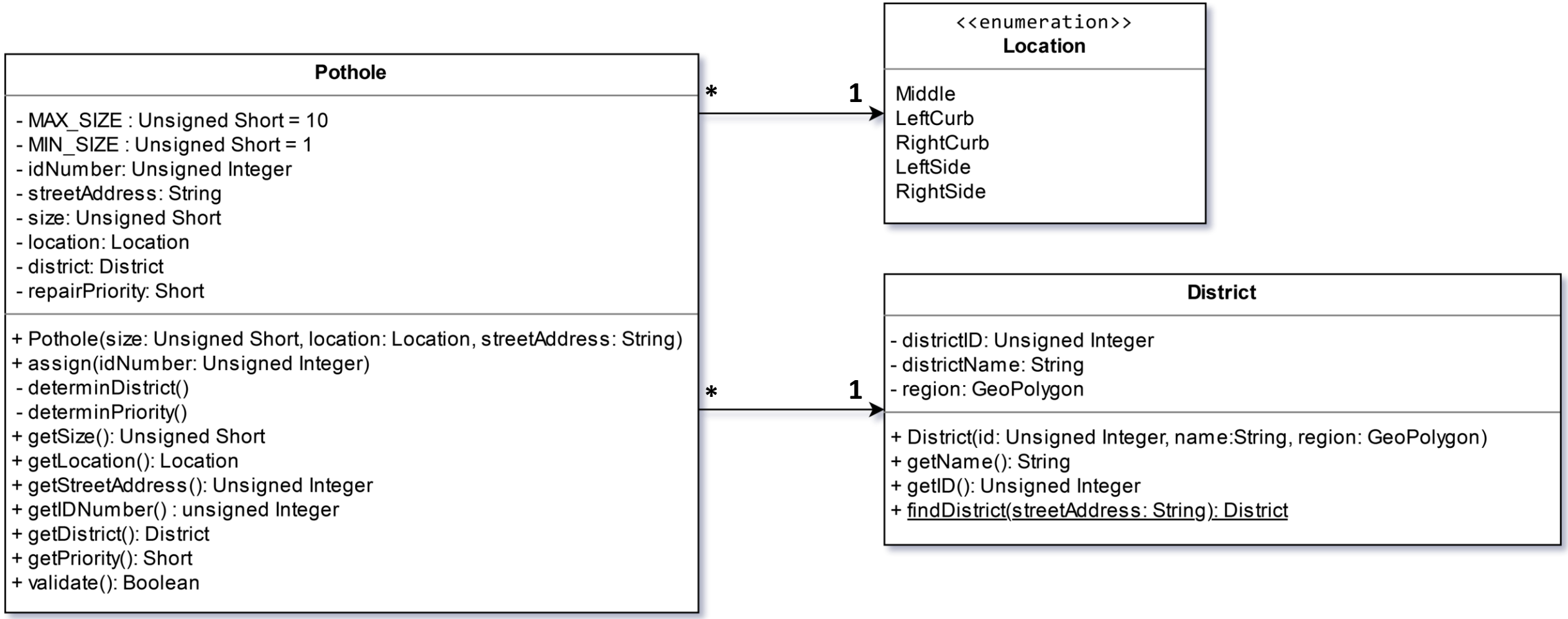
- MAX_SIZE : Unsigned Short = 10
- MIN_SIZE : Unsigned Short = 1
- idNumber: Unsigned Integer
- streetAddress: String
- size: Unsigned Short
- location: Location
- district: District
- repairPriority: Short

+ Pothole(size: Unsigned Short, location: Location, streetAddress: String)
+ assign(idNumber: Unsigned Integer)
- determinDistrict()
- determinPriority()
+ getSize(): Unsigned Short
+ getLocation(): Location
+ getStreetAddress(): Unsigned Integer
+ getIDNumber() : unsigned Integer
+ getDistrict(): District
+ getPriority(): Short
+ validate(): Boolean

Design Classes

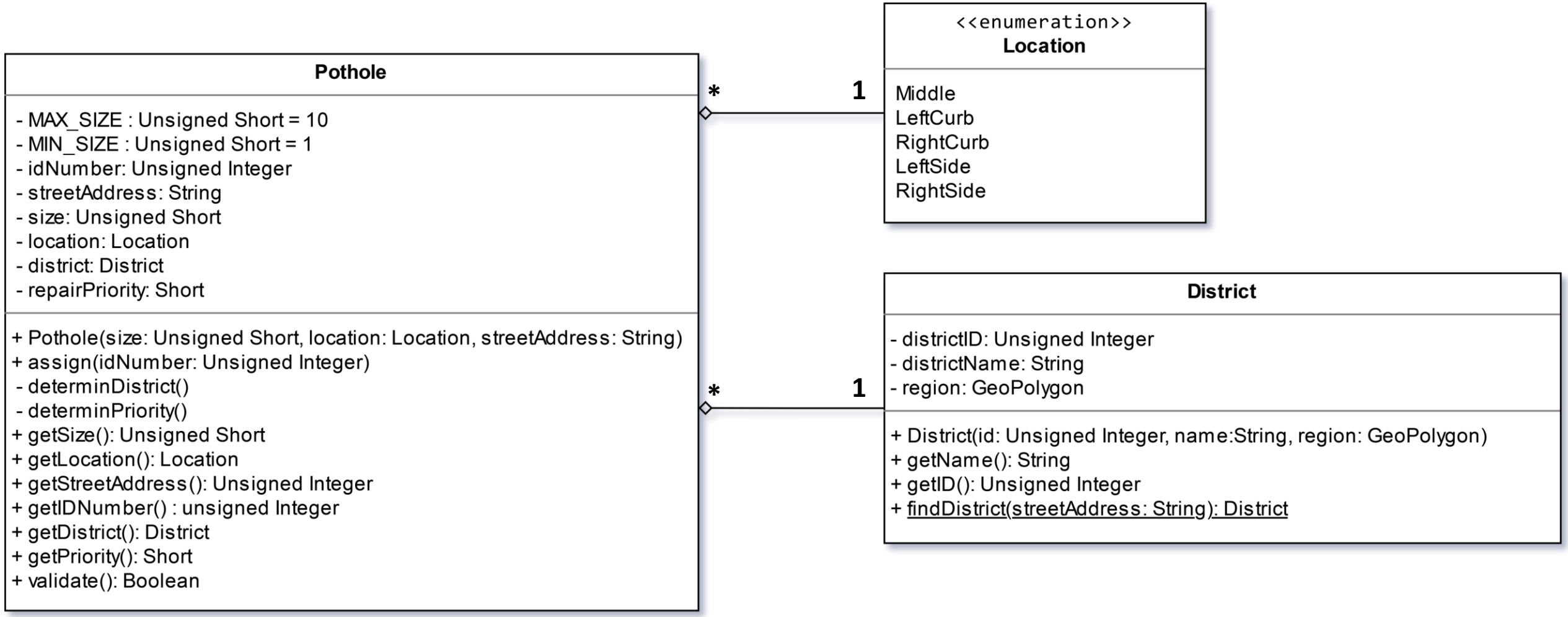


Design Classes



Design Classes

If we view the pothole as consisting of a Location and District (they are part of the pothole) we could use Aggregation. This solution is effectively the same as the last one.



Design Concepts

Design Classes

- Four characteristics of a **well-defined** **design class**:

1. **Complete and Sufficient:** A class is **complete** if it includes all necessary attributes and methods. It is **sufficient** if it contains only those methods needed to achieve class intent.

2. **Primitiveness:** Each class method focuses on providing **one service** for the class.

3. **High Cohesion:** Small, focused, **single-minded classes** with attributes and methods oriented around that **single focus**.

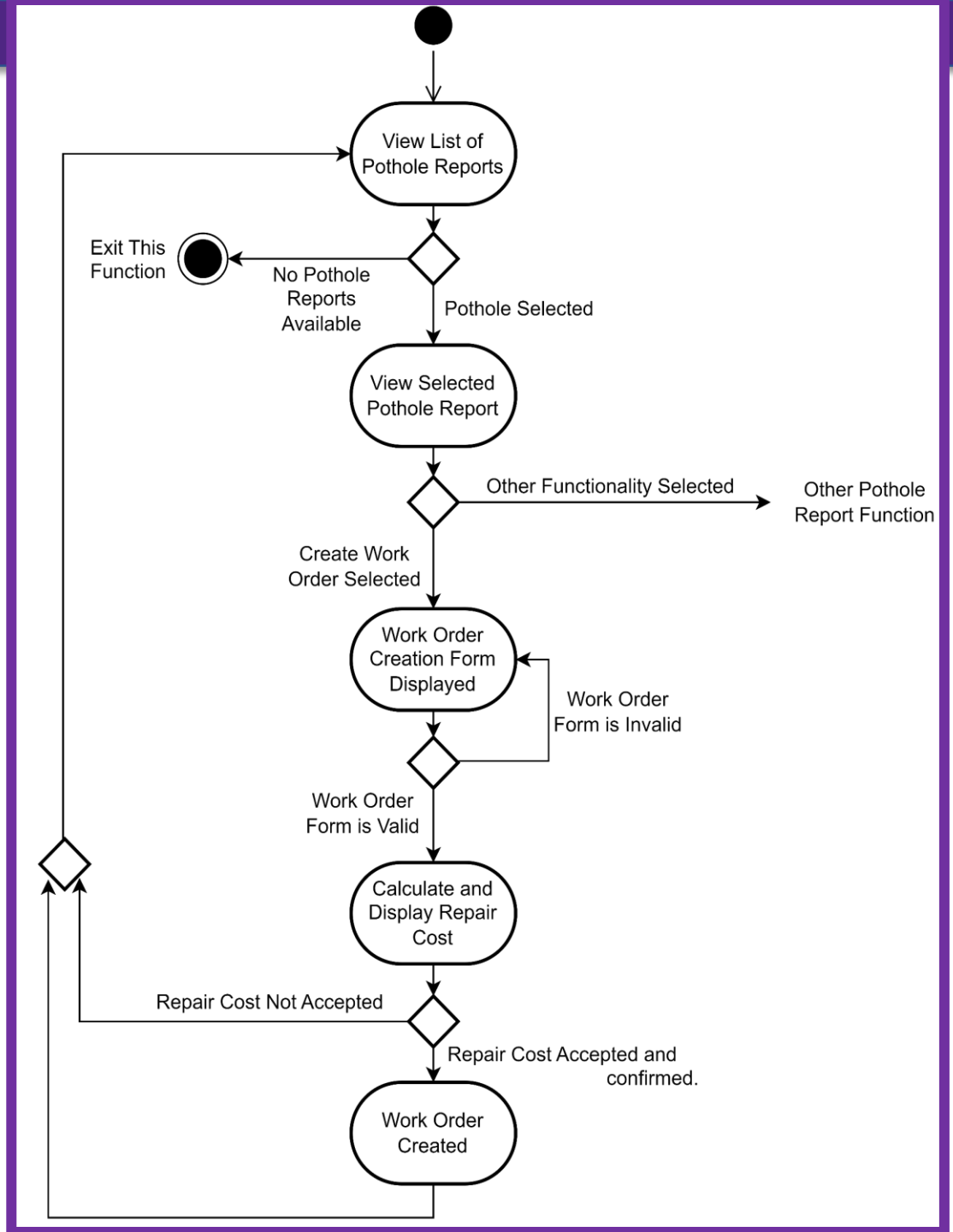
4. **Low Coupling:** Class **collaboration kept to minimum**, with classes only having **limited knowledge of other classes**.

Design Classes Activity

Example Scenario: Web-based Pothole Tracking and Repair System (PHTRS)

Citizens can log onto a website and report the location and severity of potholes. As potholes are reported they are logged within a "public works department repair system" and are assigned an identifying number, stored by street address, size (on a scale of 1 to 10), location (middle, curb, etc.), district (determined from street address), and repair priority (determined from the size of the pothole). **Work order data are associated with each pothole and include pothole location and size, repair crew identifying number, number of people on crew, equipment assigned, hours applied to repair, hole status (work in progress, repaired, temporary repair, not repaired), amount of filler material used, and cost of repair (computed from hours applied, number of people, material and equipment used).** Finally, a damage file is created to hold information about reported damage due to the pothole and includes citizen's name, address, phone number, type of damage, and dollar amount of damage. PHTRS is an online system; all queries are to be made interactively.

Activity Diagram for Work Order Use Case



Design Classes Activity

Analysis Class

WorkOrder
pothole repairCrewID numOnCrew equipmentAssigned hoursForRepair holeStatus amountMaterialUsed costOfRepair
calculateCost() assignCrew()

- **Refine this analysis class into a design class:**
  - Add parameters and types.
  - Set public/private on each property and method.
  - Add any needed methods/properties.
  - Split into multiple classes if needed.
  - Show relationships with Pothole and any classes you add.
  - Add multiplicity for relationships.
  - Don’t need to show the whole Pothole class, can just be a box with the word “Pothole” in it.
  - **Recommend using:** https://app.diagrams.net/

The Design Model

- The elements of the **design model** use many of the **same diagrams** that were used in the **requirements/analysis model**.
- In some cases, there is a clear distinction between the models, and in others, the **requirements/analysis model** **slowly blends into the design** and a clear distinction is less obvious.
- That said, the **design model** **is typically more refined** and detailed, with more attention given to the implementation that is to come.

The Design Model

Viewed in Two Dimensions

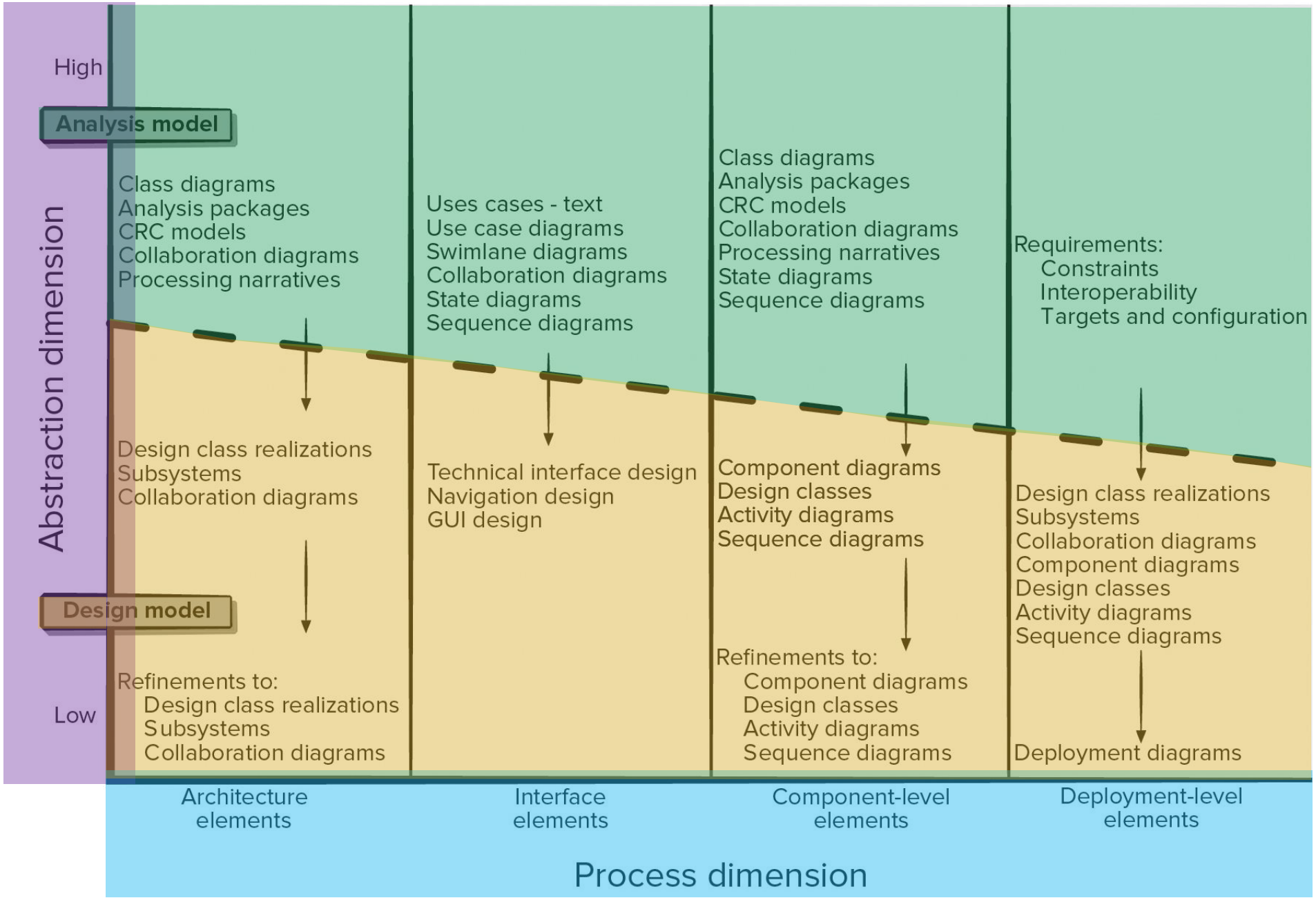


Figure 9.4 from your textbook.

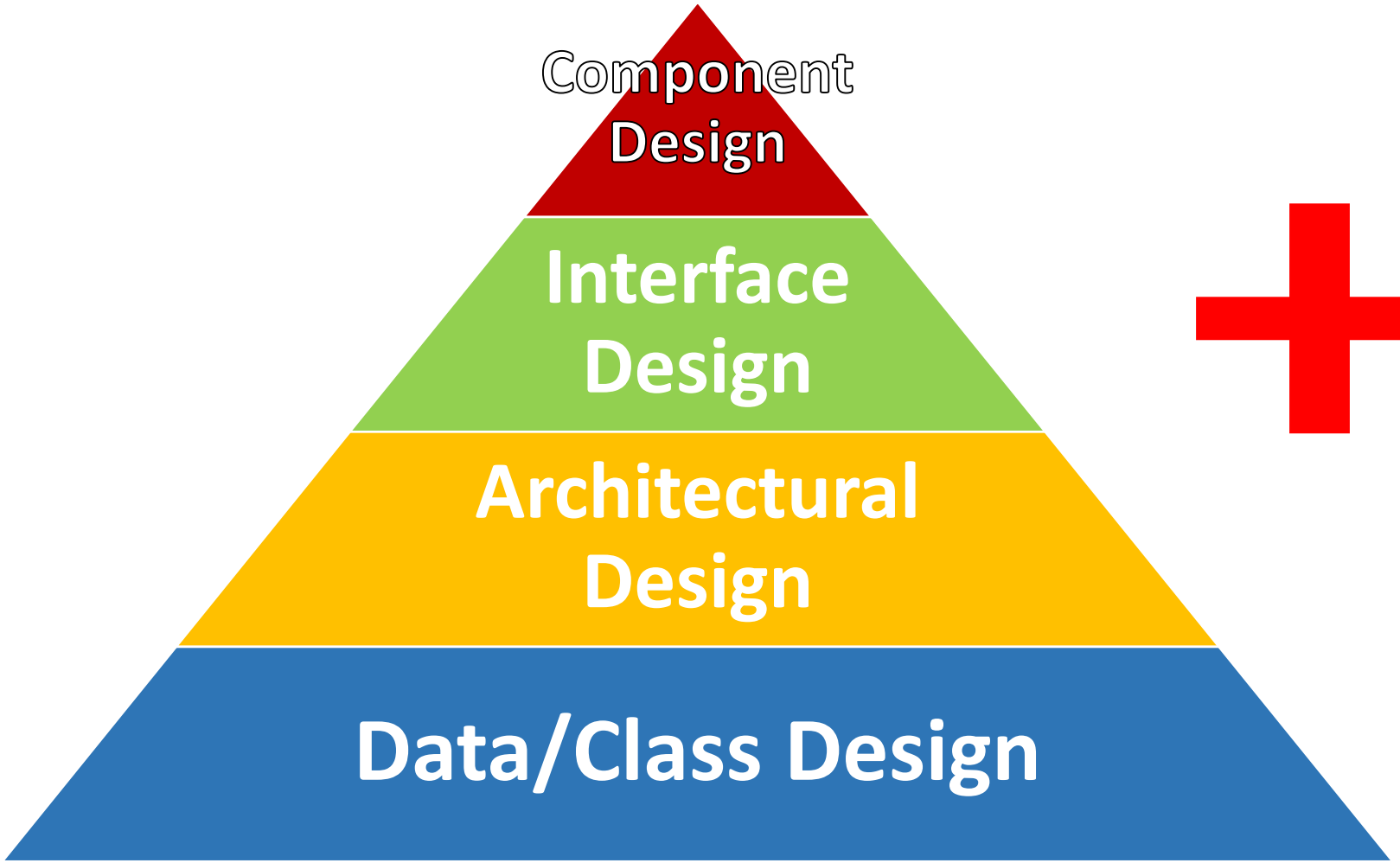
Design Modeling Principles

- **Principle #1:** Design should be **traceable to the requirements model**.
- **Principle #2:** Always **consider the architecture** of the system to be built.
- **Principle #3: Design of data is** as **important** as design of processing functions.
- **Principle #4: Interfaces** (both internal and external) must be **designed with care**.
- **Principle #5: User interface design** should be tuned to the **needs of the end-user** and stress ease of use.

Design Modeling Principles

- **Principle #6:** Component-level design should be **functionally independent**.
- **Principle #7:** Components should be **loosely coupled** to each other.
- **Principle #8:** Design representations (models) should be **easily understandable**.
- **Principle #9:** The design should be **developed iteratively**.
- **Principle #10:** Creation of a design model **does not preclude using an agile approach.**

Elements of the Design Model



+

Deployment Design Elements

Data Design Elements

- **Data design** creates a **model of data** and/or information that is represented **at a high level of abstraction** (the customer or user's view of data).
- This **data model** is then **refined into progressively more implementation-specific representations** that can be processed effectively and efficiently.
- In many software applications, the architecture and **design of the data will have a profound influence on the architecture of the software** that must process it.

Data Design Elements

- **Data model** is comprised of **data objects** and **database architectures**.
- **Data object** can be an **external entity**, a **thing**, an **event**, a **place**, a **role**, an **organizational unit**, or a **structure**.
- **Data objects** contain a **set of attributes** that act as a quality, characteristic, or descriptor of the object.
- **Data objects** may be **connected to one another** in many different ways.

Interface Design Elements

- The **interface design elements** for software **depict information flows into and out of a system** and how it is communicated among the components that are part of the architecture.

- There are three important **elements of interface design**:

  1. The **user interface (UI)**

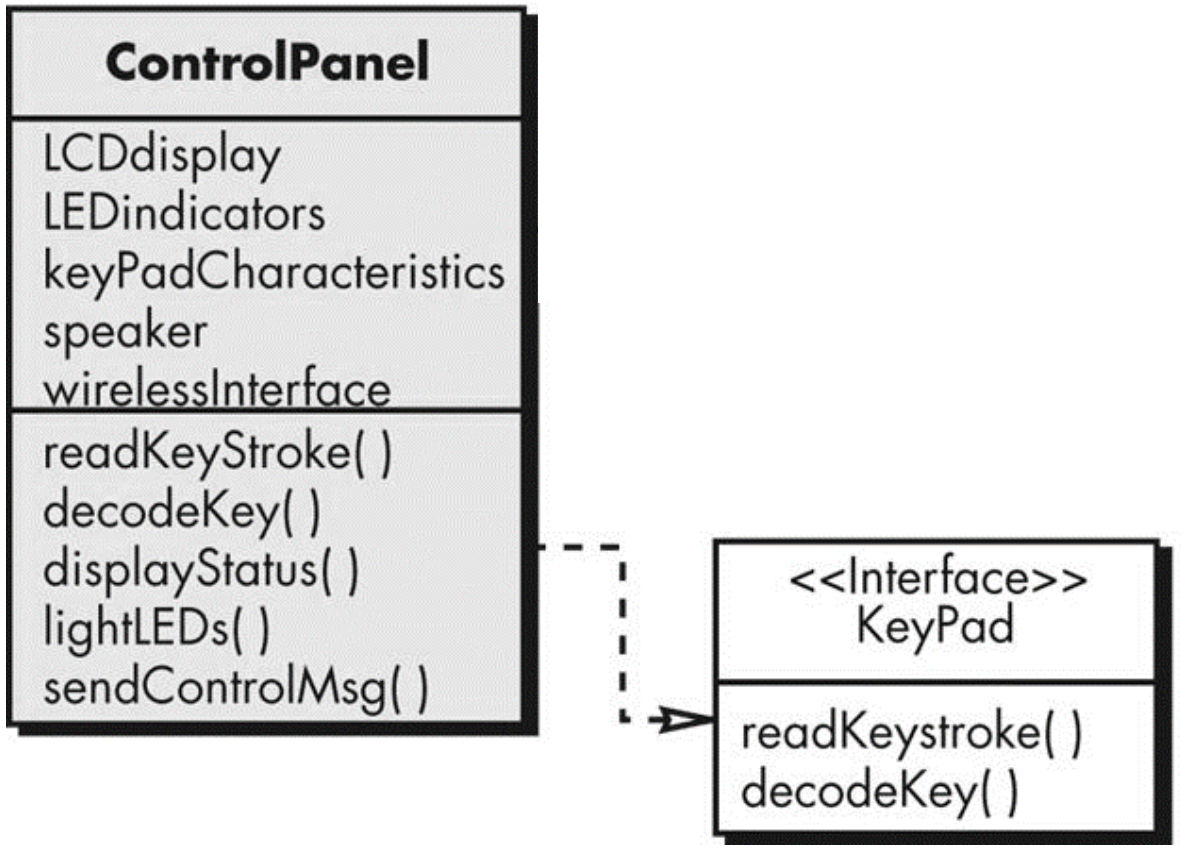
  2. **External interfaces** to other systems, devices, networks or other producers or consumers of information.

  3. **Internal interfaces** between various design components.

Interface Design Elements

Internal Interface Example

- Using an **UML class diagram** to model part of the *SafeHome* system using an interface.
- One can view an interface here as a set of operations that describes some part of the behaviour of a class, and something that provides access to these operations.



Based on Figure 9.5 from your textbook.

Architectural Design Elements

- **Architectural design** is intended to give an **overall view of the software**.
- As such, it is usually **depicted as a set of interconnected subsystems**, often derived from analysis packages and classes from the requirements model.
- The **architectural model** is typically derived from three sources:
  1. Information about the **application domain** for the software to be built.
  2. **Specific requirements** model elements such as data flow diagrams or **analysis classes**, their relationships and collaborations for the problem at hand.
  3. The availability of **architectural patterns** and styles.

Architectural Design Elements

- **Architectural design** is intended to give an **overall view of the software**.
- As such, it is usually **depicted as a set of interconnected subsystems**, often derived from analysis packages and classes from the requirements model.
- The **architectural model** is typically derived from three sources:
  1. Information about the **application domain** for the software to be built.
  2. **Specific requirements** model elements such as data flow diagrams or **analysis classes**, their relationships and collaborations for the problem at hand.
  3. The availability of **architectural patterns** and styles.

Far More on Architectural Design in Chapter 10

Component-Level Design Elements

- The **component-level design** for software fully **describes the internal detail of each software component.**
- To accomplish this, the **component-level design** defines:
  - **Data structures** for all local data objects.
  - **Algorithmic detail** for all component processing functions.
  - An **interface** that allows access to all component operations (behaviours).
- Modeled using **UML component diagrams** or **activity diagrams**, **pseudocode**, and/or **flowcharts.**

Component-Level Design Elements

- The **component-level design** for software fully describes **the internal detail of each software component**.
- To accomplish this, the **component-level design** defines:
  - **Data structures** for all local data objects.
  - **Algorithmic detail** for all component processing functions.
  - An **interface** that allows access to all component operations (behaviours).
- Modeled using **UML component diagrams** or **activity diagrams**, **pseudocode**, and/or **flowcharts.**

Far More on Component Design in Chapter 11

Deployment-Level Design Elements

- **Deployment-level design** indicates **how software functionality and subsystems will be allocated** within the physical computing environment.

- Modeled using **UML deployment diagrams**.

  - **Descriptor form deployment diagrams show the computing environment,** but dose not explicitly indicate configuration details.

  - **Instance form deployment diagrams identifying specific named hardware configurations,** are developed during the latter stages of design.