Solving Really Big Problems

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 11 — 10/02/2007

© University of Colorado, 2007

Why Start with a Song?

Note: I played Stan Kenton's version of Malaguena before lecture, making notes on the song's structure

- We are going to be learning about software architecture this week
  - Thinking about music can help in understanding architecture
- What is the architecture of a song?
  - Components: verses, refrain, solos, …
  - Sub-Components: Notes, Rests, Lyrics
  - Connectors: Arrangements, "the bridge", "swing section", …
  - Styles: Jazz, Classical, 80s alternative, indie, funk, goth, death metal, …
  - Common (or Stylistic) Elements: melody, counter melody, echoing, themes, muscial pyramids, etc.
  - Experience: same song can be vastly different based on the performers

Lecture Goals

- Review material from Chapter 6 of the OO A&D textbook
  - How do you solve big problems
    - That is, how do you design and build really large software systems?
  - Domain Analysis
  - Use Case Diagrams
  - Introduction to Software Architecture
  - Discuss the Chapter 6 Example: Gary’s Game Framework
  - Emphasize the OO concepts and techniques encountered in Chapter 6

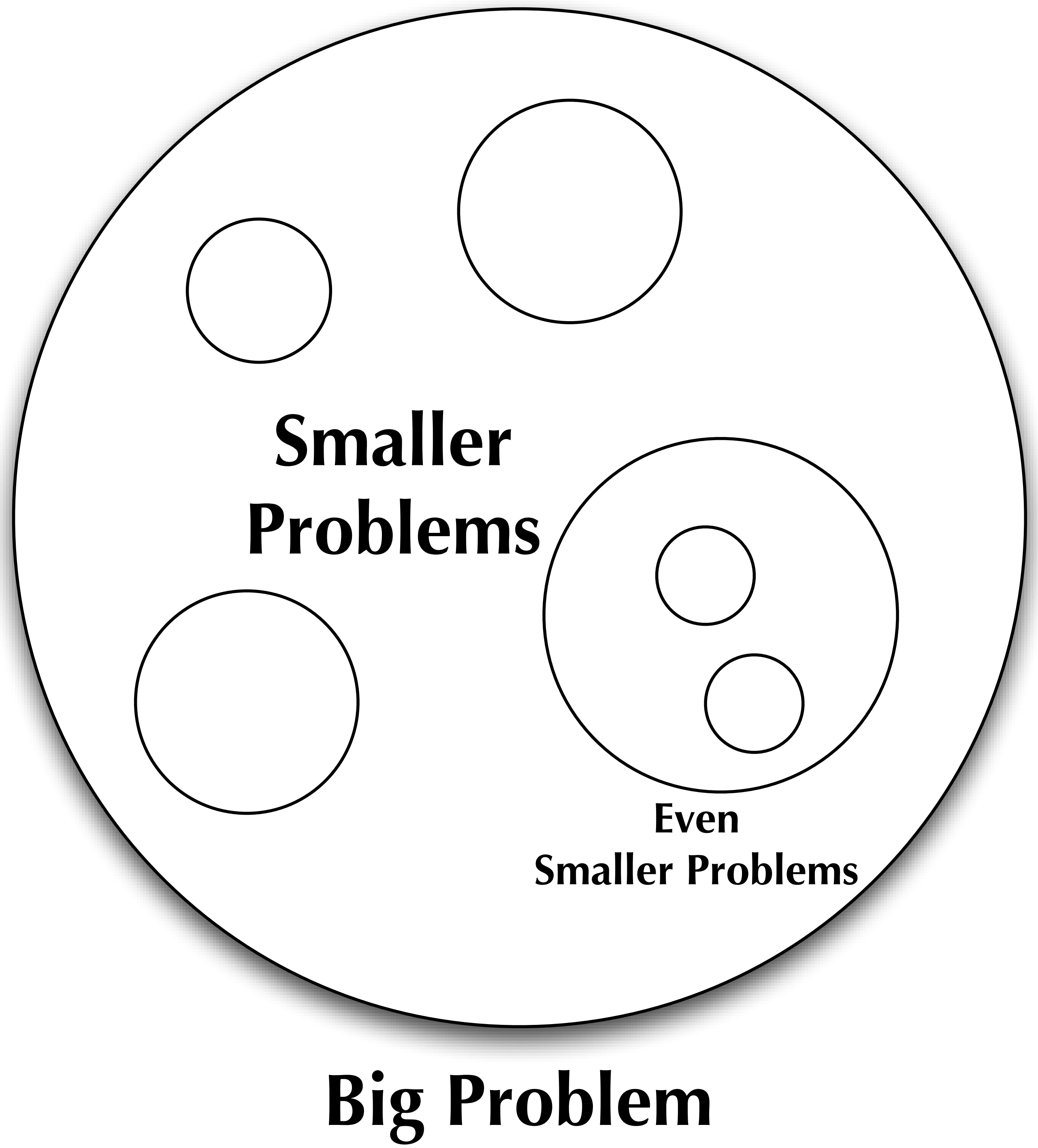
Living in Smallville?

- So far, we've been discussing OO A&D in the context of small applications
  - Rick's Guitars: Less than 15 classes (at its worst)
  - Doug's Dog Doors: Never more than 5 classes!
- Will the techniques that we've learned so far apply to real systems?
  - which tend to be big, complex, and consist of 100s to 1000s of classes
- The quick answer?
  - Yes
- Our three step life cycle (make your software work, apply OO principles, strive for a maintainable, reusable design) still applies to large situations
  - with the assistance of new techniques: software architecture, use case diagrams, domain analysis, design patterns, and more
- The long answer?

Its just a matter of perspective!

The (Sometimes Painful) Real World

- Dealing with the difficulties of large-scale, real-world software development can feel the same as if a bull is rapidly bearing down on you!
  - But if you view the problem in the right way (and get out of the bull's way **pronto**), the complexity of the real world can be handled
- The key is "divide and conquer"
  - You can solve a big problem by breaking it into lots of functional pieces, and then work on each piece individually
    - perhaps by applying "divide and conquer" again!



What have we learned so far?

- **Analysis helps ensure that your systems works in real-world contexts**
  - Analysis is even more important when working on large systems
- **Get good requirements by understanding what the system needs to do**
  - Apply this to the small problems, combine to address the big problem
- **Encapsulate what varies to achieve flexible, easy-to-change software**
  - In large systems, encapsulation breaks up big problems into small ones
- **Code to an interface to create software that is easy to extend**
  - In large systems, coding to an interface can reduce internal dependencies
- **Ensure that components have only one reason to change**
  - High cohesion is critical in large systems: individual pieces are independent of each other and can be worked on in isolation

Example: Gary’s Games

- The example in this chapter involves designing a **game framework** (note: not a *game* but a game *framework*)
  - The book presents you with a vision statement from your client
    - It has some details but doesn’t come close to a requirements spec
- However, when dealing with a large system, **avoid jumping straight into creating requirements and use cases**
  - You need a **detailed understanding of the application domain** (**problem domain**) before you can create a requirements spec and use cases
  - We need to know
    - What is the system like?
      - strategy board games, it turns out
    - What is the system not like?
      - Halo 3 (for instance)

Step 1: Listen to the Customer

- To gain this information, we need to meet with the customer and listen to their discussions about the system
    - The "customer" may be many different people playing different roles
        - Management
        - Marketing
        - Design
        - Sales
    - All will have important information to provide and the multiple perspectives will give you a more accurate picture of your system's real-world context

Step 2: Find the Features

- Using the information provided by the customer, identify the features that your system will have
  - A feature is a high-level description of something a system needs to do
- Features can then lead to requirements
  - Think of them as **compound requirements**
    - One feature may be decomposed into multiple requirements
    - These requirements then need to be implemented to satisfy the feature
- Because features are high-level, they are a useful for getting a project started when the customer has not yet provided you with a lot of details

Gary’s Features

Features for Gary’s Game System

1. Supports different time periods, including fictional periods like sci-fi and fantasy
2. Supports add-on modules for additional campaigns or battle scenarios
3. Supports different types of terrain
4. Supports multiple types of troops or units that are game-specific
5. Each game has a board, made up of square tiles, each with a terrian type.
6. The framework keeps up with whose turn it is an coordinates basic movement

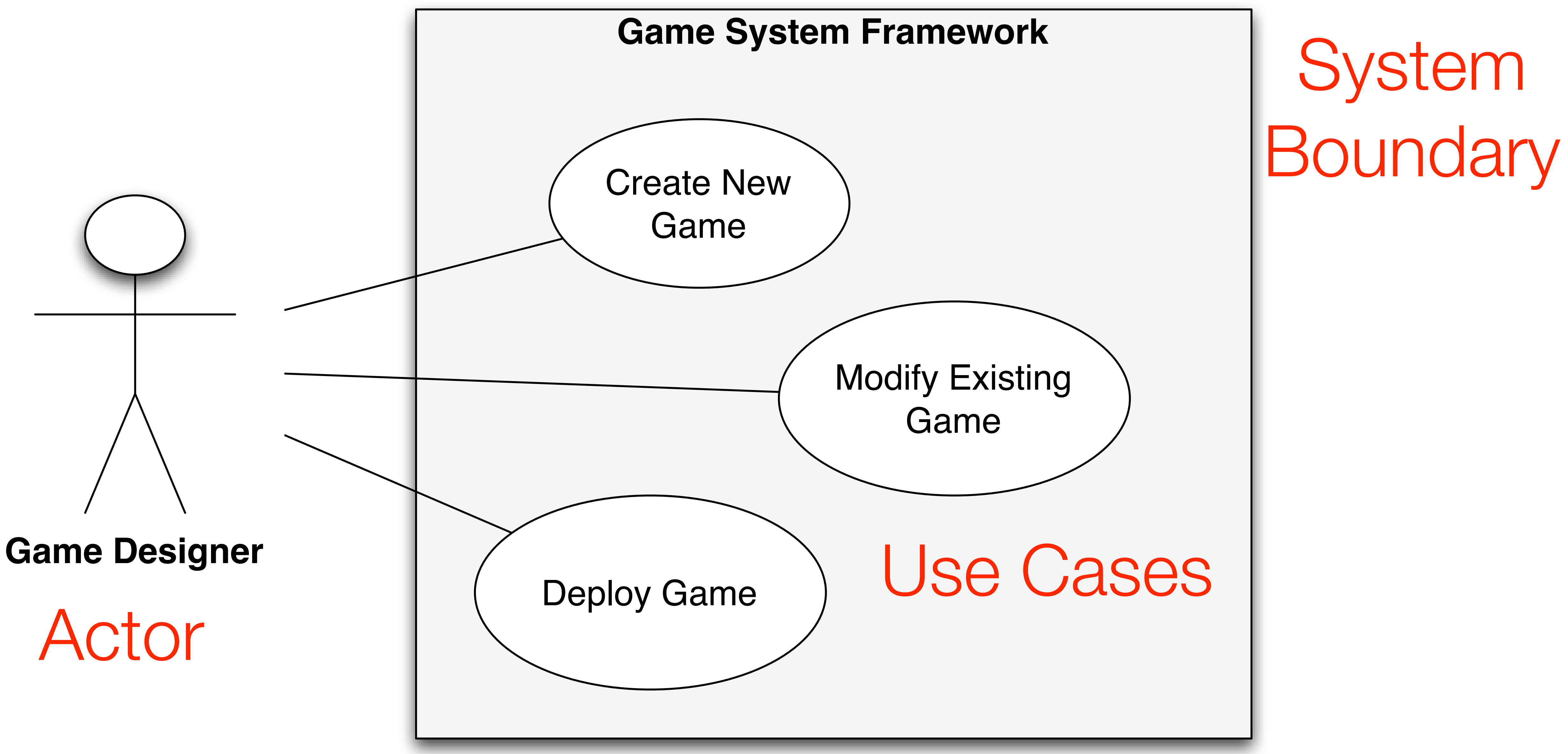
Feature vs. Requirement?

- The book warns against getting too caught up in the difference between features and requirements
  - Just think of features as **compound requirements**
- Since they can be decomposed into lots of smaller requirements, they cover **broad classes of functionality** that the system has to support
  - Thus making them **easier to find** than smaller requirements when a project is just getting started

Step 3: Big Picture View

- The next step is to acquire a broad view of the major activities your system engages in:
  - “What the system is supposed to do”
  - without resorting to writing specific use cases
    - use cases again require a lot of detail; detail that you might not have
- The solution?
  - Identify the types of users that interact with the system (aka Actors)
  - Identify the names of the use cases these actors interact with
    - In other words, what are the major tasks handled by this system?
- This view is called the **use case diagram**

Example



But how do features relate to this view of the system?

Getting back to features

- **Use your feature list to make sure your use case diagram is complete**
  - Try to assign features to use cases
    - If a feature can't be assigned, then add use cases until coverage is complete
  - The book assigns five of the features to the Create New Game use case
    - I thought this was a bit excessive: for instance, I felt that the "add-on modules" feature should have been assigned to the "Modify" use case
  - One feature "handle turns, coordinate movement" was left unassigned
    - They asked the question: what Actor would need this use case?
    - The answer: not a Game Designer but a Game itself
  - Since a Game is built using the framework, its an external actor that needs its own use cases!

Updated Use Case Diagram

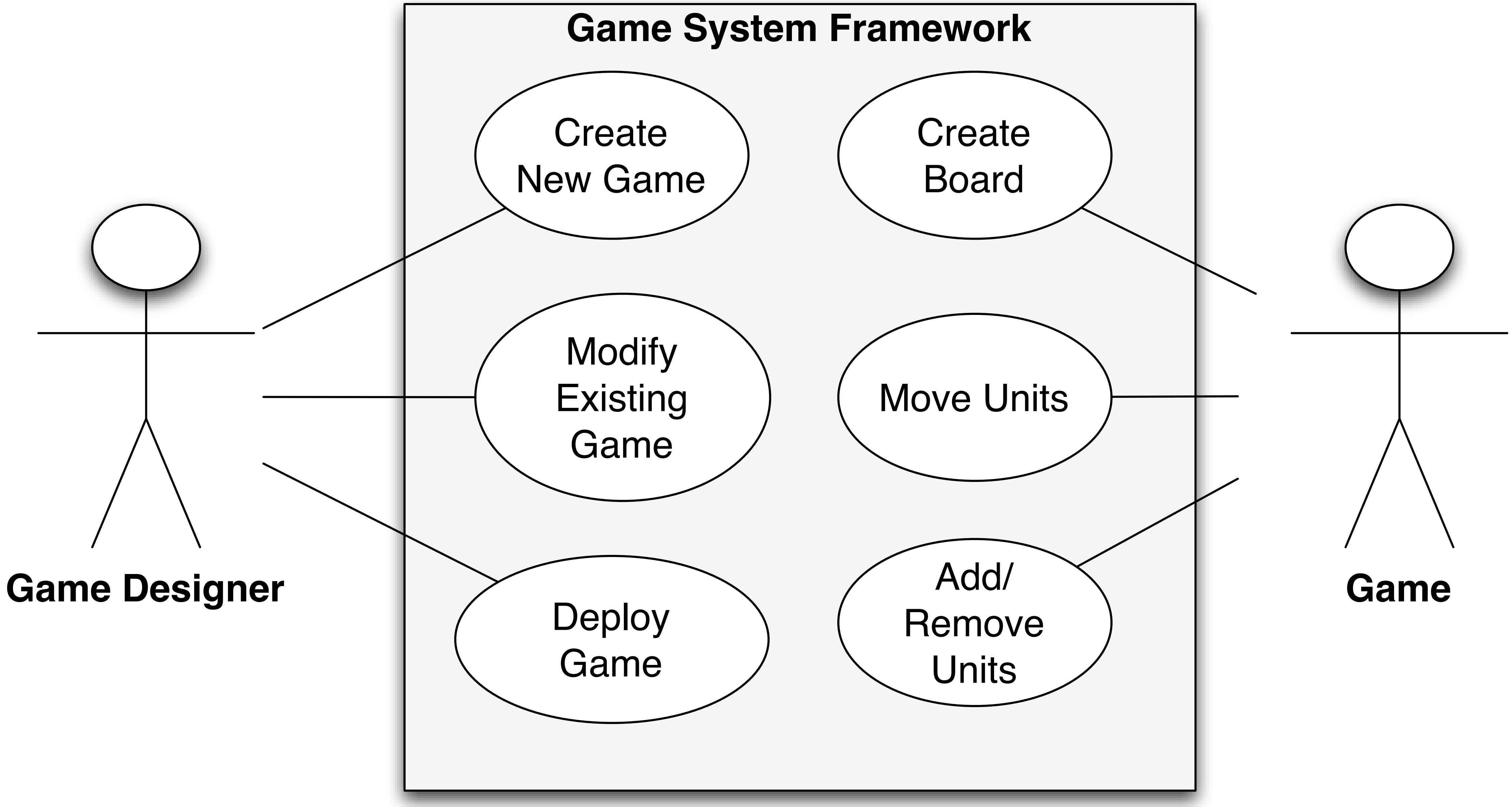


Diagram gives “big picture” view of framework

The Result?

- We have created a feature list to capture the **BIG THINGS** that your system needs to do
  - Features don't require as much detail as individual requirements
  - They allow us to capture broad categories of functionality
- We drew a use case diagram to identify important actors and use cases
  - without getting bogged down in the specifics of the use cases
    - which often require a lot of detail
- These two artifacts combine to give us a "big picture" view of the system
  - also called the **"system at 10,000 feet"** view
  - It shows us what the system IS without getting into too much detail
- But, we can now use this as a starting point for additional OO A&D work **once we break this information up into smaller pieces of functionality**

Domain Analysis

- These two artifacts by staying at a high level of abstraction allowed us to conduct **domain analysis** (without even knowing it!)
  - Domain Analysis (def): The process of identifying, collecting, organizing, and representing the relevant information of a domain
    - based upon the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology in the domain
- Feature lists capture requirements **using terms familiar to the customer**
  - Rather than giving our customers: packages, UML diagrams, and code
  - We give them: features and scenarios using familiar terminology
- This makes for very happy customers who can provide additional guidance now that “they know that you know” the core details of the domain

What’s Next?

- The Big Break-Up
  - That is, splitting our “big problem” into smaller problems
  - The book introduces the concept of “module” (aka “package”) as a means to partition our large system into something more manageable
  - Looking at the feature list, they created the following modules
    - Game, Board, Units, Controller, Utilities
    - (the last one was added just on general principle)
  - What about Graphics?
    - Not in Scope!
    - Very important lesson with respect to framework design:
      - Clearly define the functional boundaries between the framework and the applications that **USE** the framework

Design Patterns

- Its a bit premature but the book pauses to notice that in partitioning the framework the way we did, we have two of the pieces needed for a famous design pattern
  - The Model-View-Controller pattern
- The most important lesson at this point of the chapter is
  - Design patterns don't go into your code, they go into your BRAIN
  - Design patterns are SOLUTIONS to common design PROBLEMS
    - The more of these common solutions that you know, the better you'll be at avoiding the common design problems
  - Applying design patterns is one of the LAST steps of design
    - They are best applied during "step 3" of our simple OO A&D process
- We will turn our attention to learning design patterns once we have finished with the OO A&D textbook

Turning a Big Problem into Smaller Problems

- Summary
  - We listened to the customer: **vision statement, domain analysis**
  - We made sure we understood the system: **feature list**
  - We drew up blueprints for the system we're building: **use case diagram**
  - We broke the big problem up into smaller pieces of functionality: **modules**
  - We apply design patterns to help us solve smaller problems: stay tuned!
- Moving on…
  - Since we will be learning about the role software architecture plays in designing and implementing large software systems next lecture, lets end this lecture with a brief introduction

Introduction to Software Architecture (I)

- Any **complex system** is composed of **subsystems** that interact with one another to provide the overall system's intended functionality
- **Software architecture** is an area of **software engineering research** aimed at providing tools and techniques for specifying a system's subsystems and their interrelationships
- **WARNING**: This is the **TRADITIONAL** view of software architecture
    - Our textbook has an **alternate interpretation** that focuses on the **practicalities** of incorporating software architecture techniques into your day-to-day work practice
        - This is good! Its often difficult to understand how software architecture concerns impact day-to-day tasks and decision making

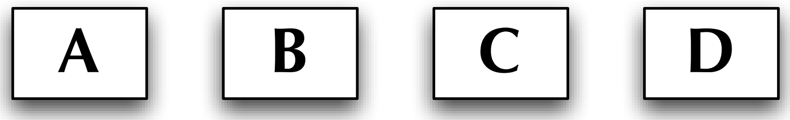
Introduction to Software Architecture (II)

- The level of granularity for software architecture design is at the **system level**, not the **package**, **module**, or **class** level.
  - For many complex systems, each individual subsystem may itself be a large software system that has its own internal architecture
- Software architecture is a **relatively recent** research area (mid-90s) with an active research community
  - Architecture Description Languages
  - Architecture Modeling Tools
  - Architecture Analysis Tools
- Definition: The principled study of software components, including their properties, relationships, and patterns of combination

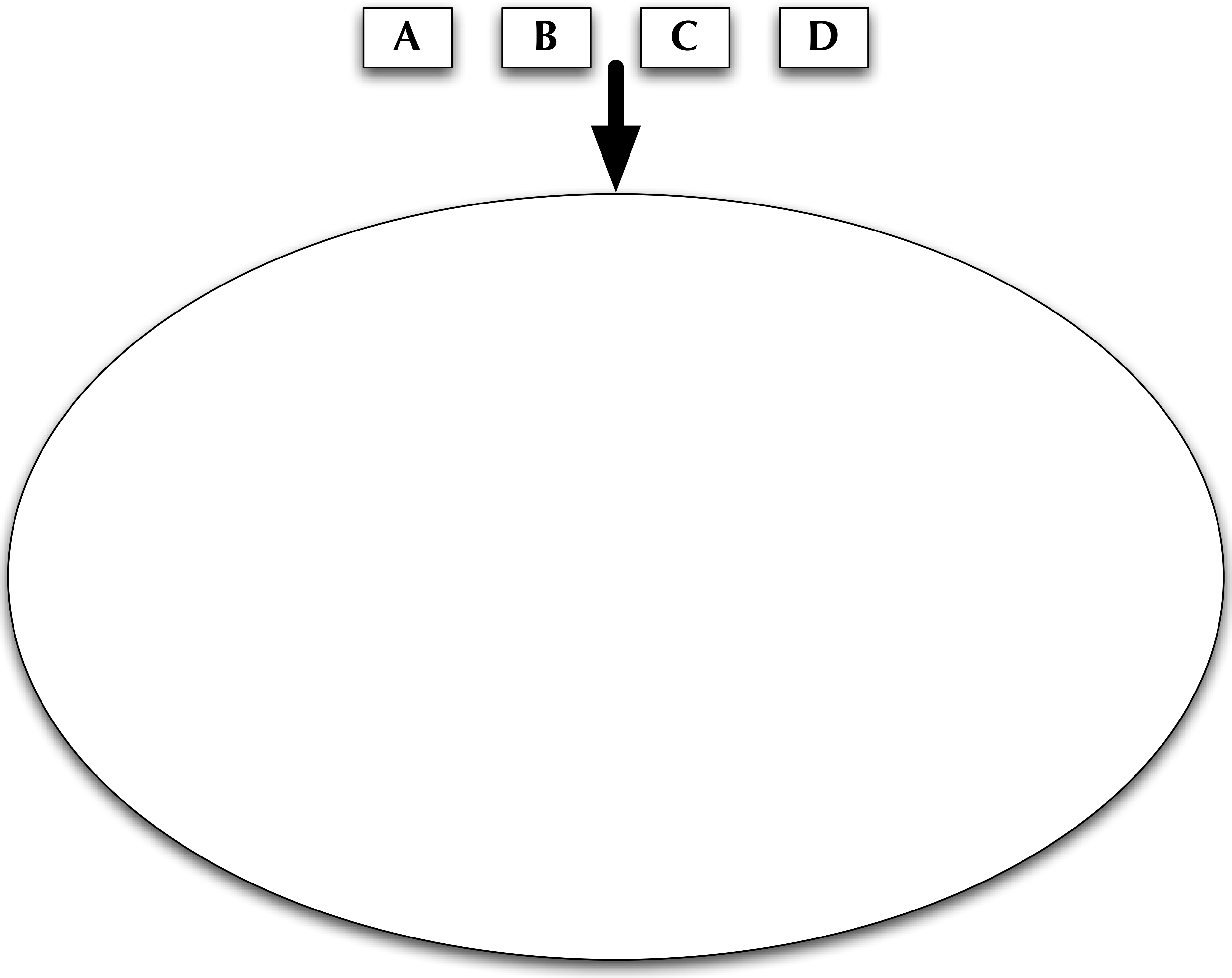
Introduction to Software Architecture (III)

- The design of a system's architecture is one of the first places in which decisions concerning technologies for implementing the system are made
    - For instance, consider the use of middleware technology or a large-scale relational database
        - As much as we would like to separate design and implementation, these types of technologies are **expensive**; if a company has invested in them, it may not be possible to choose an alternative technology
- The design of a system's architecture is also the **earliest phase** in which certain **non-functional requirements** such as security, performance, and reliability can be addressed
    - For instance, if a system's subsystems must share information using encrypted communication links, this can be specified in the system's architecture model

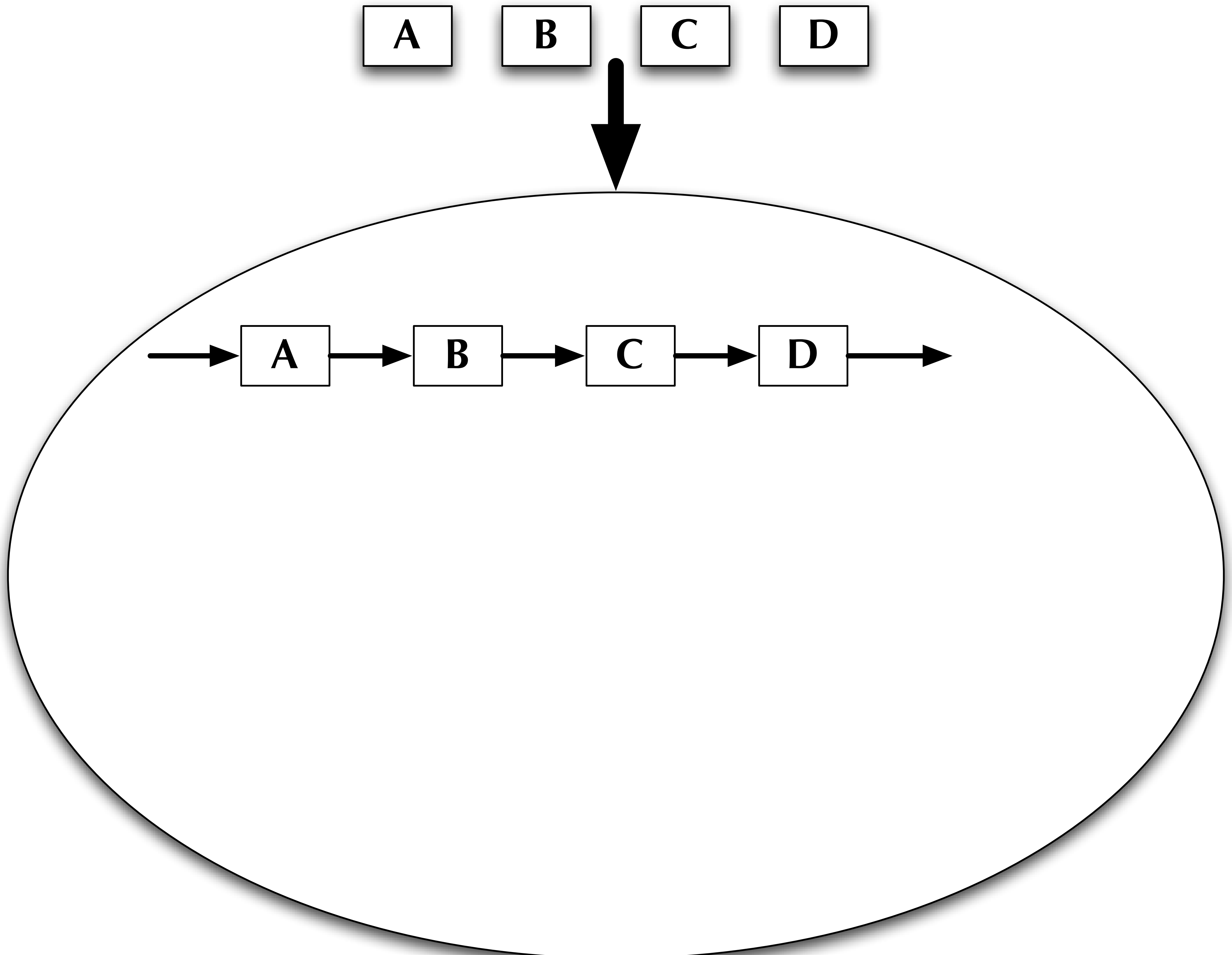
Software Architecture (I)



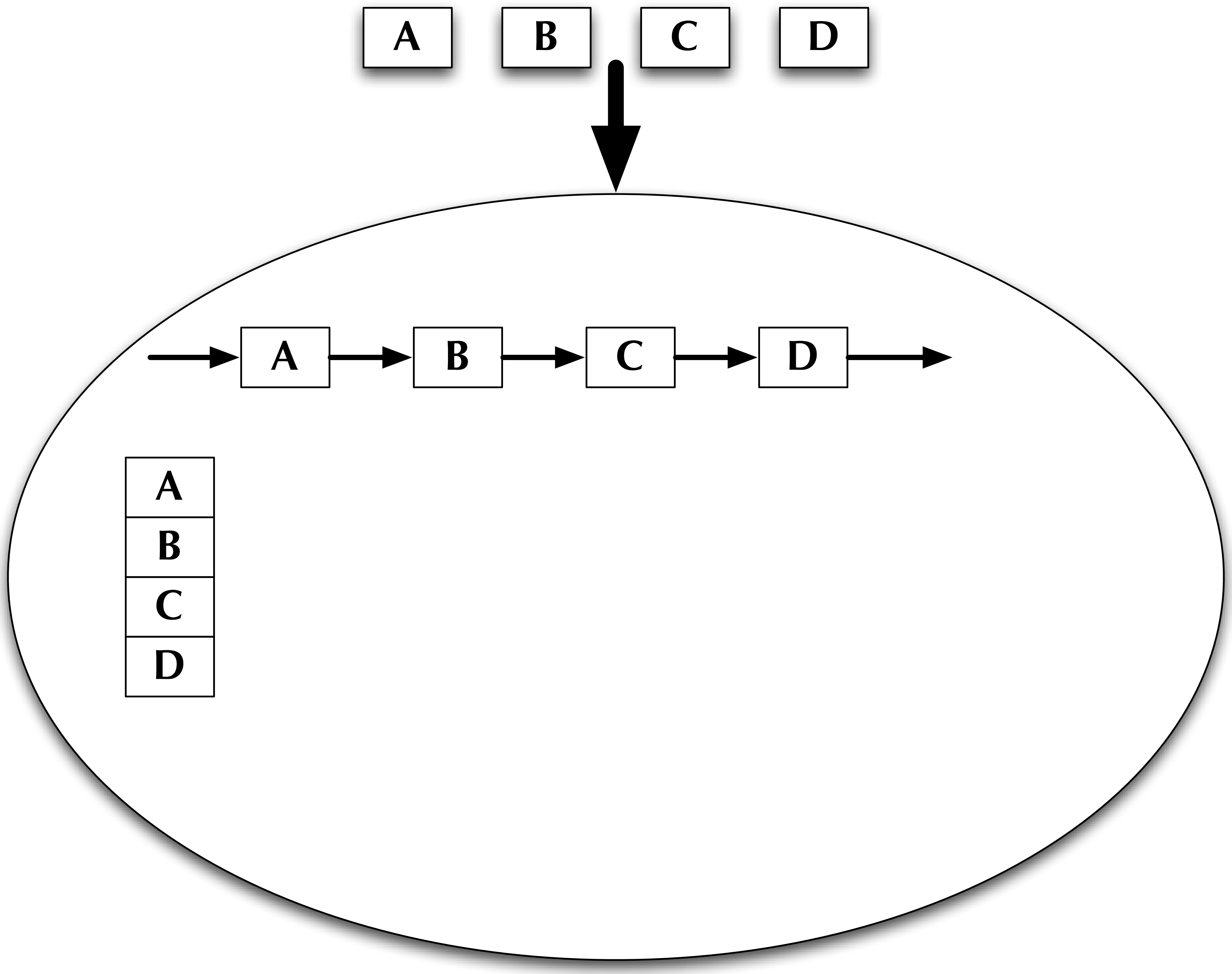
Software Architecture (II)



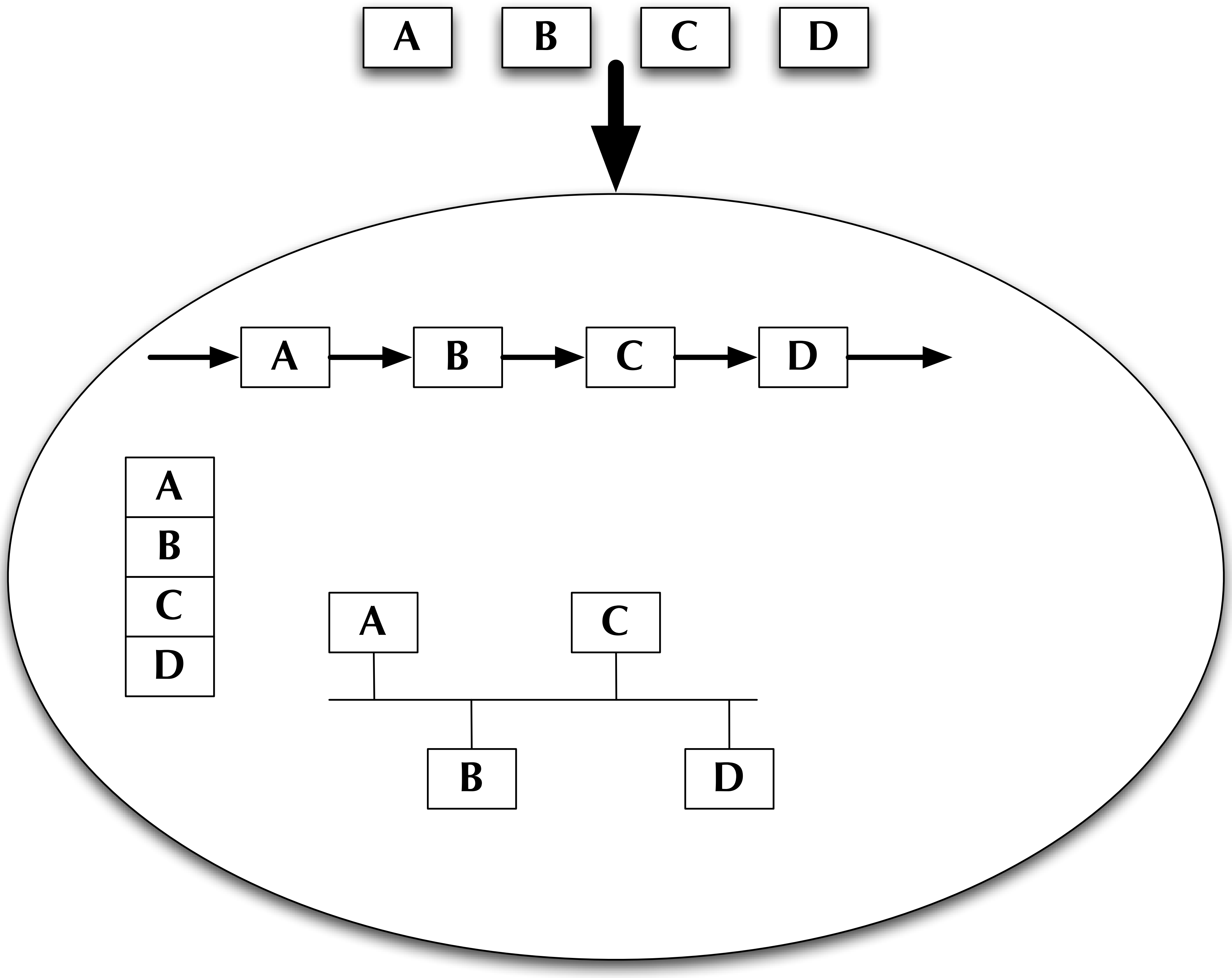
Software Architecture (III)



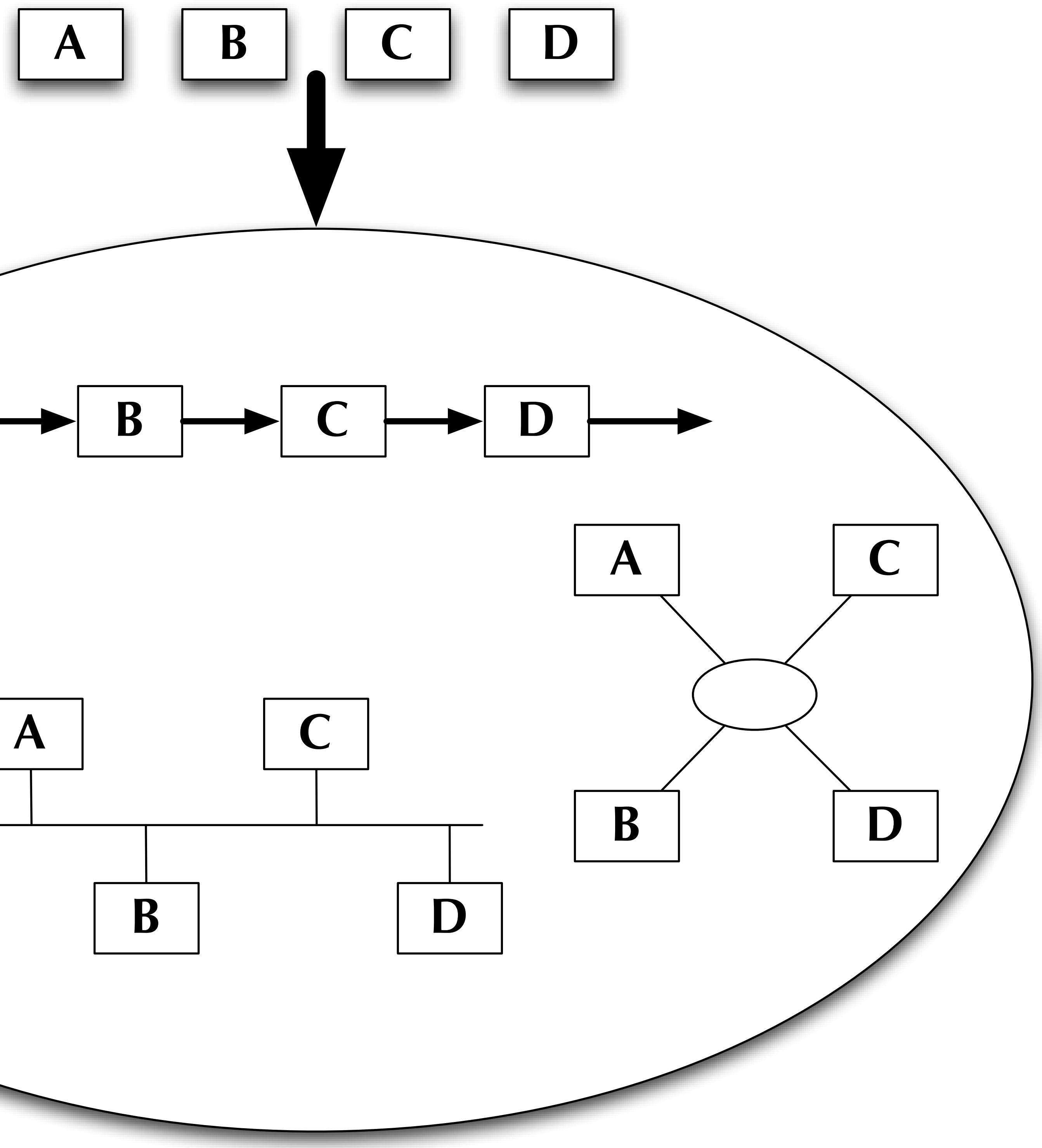
Software Architecture (IV)



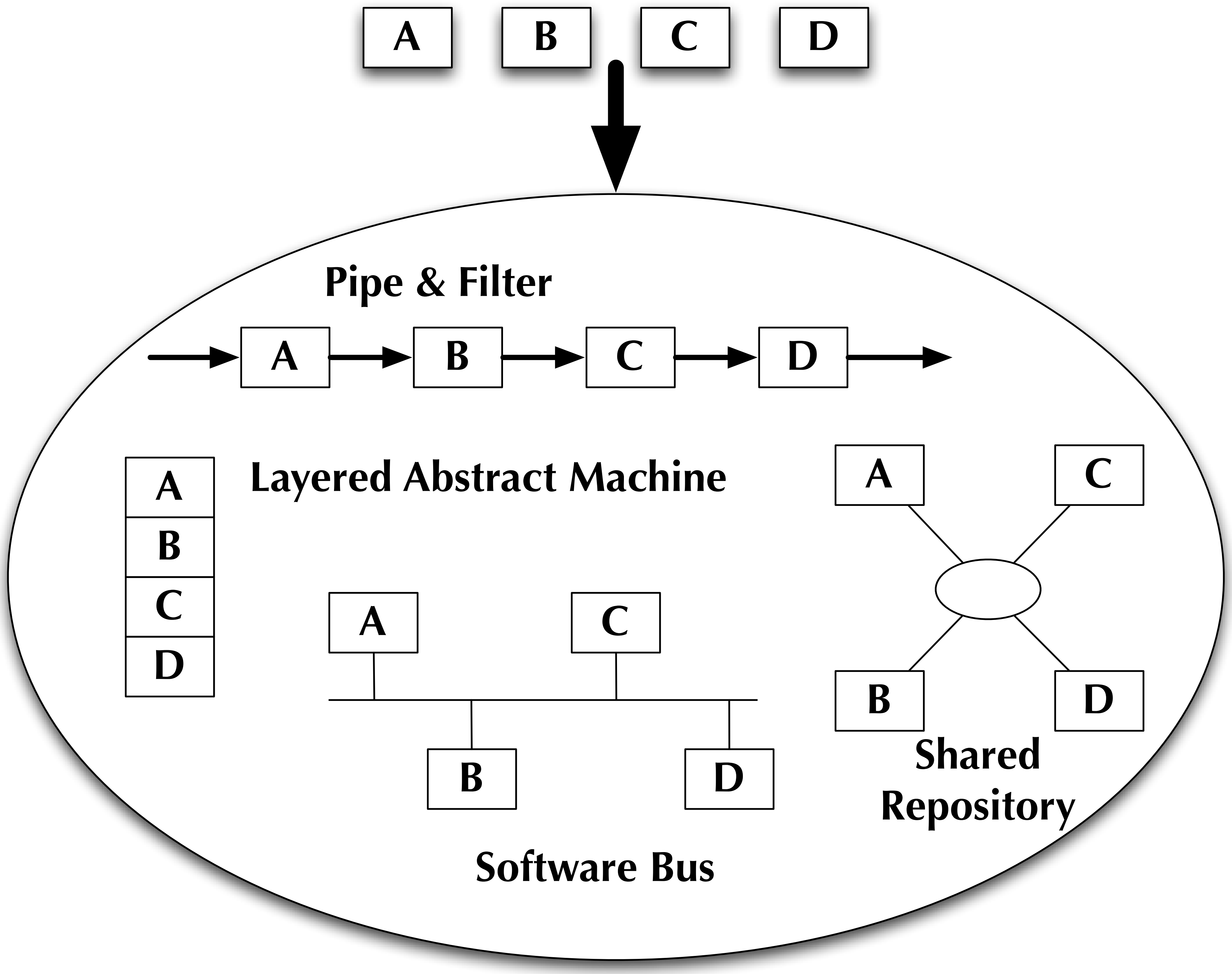
Software Architecture (V)



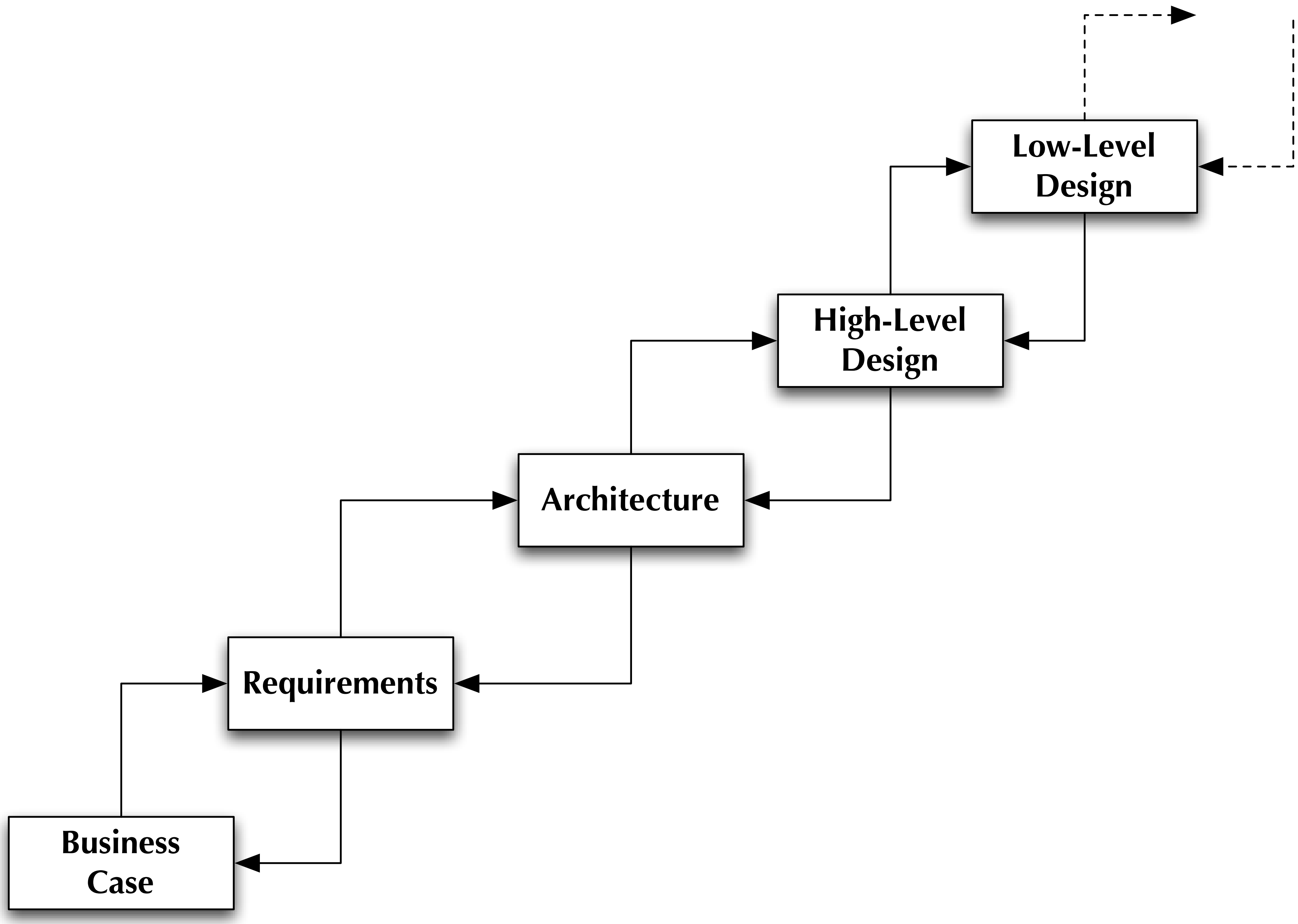
Software Architecture (VI)



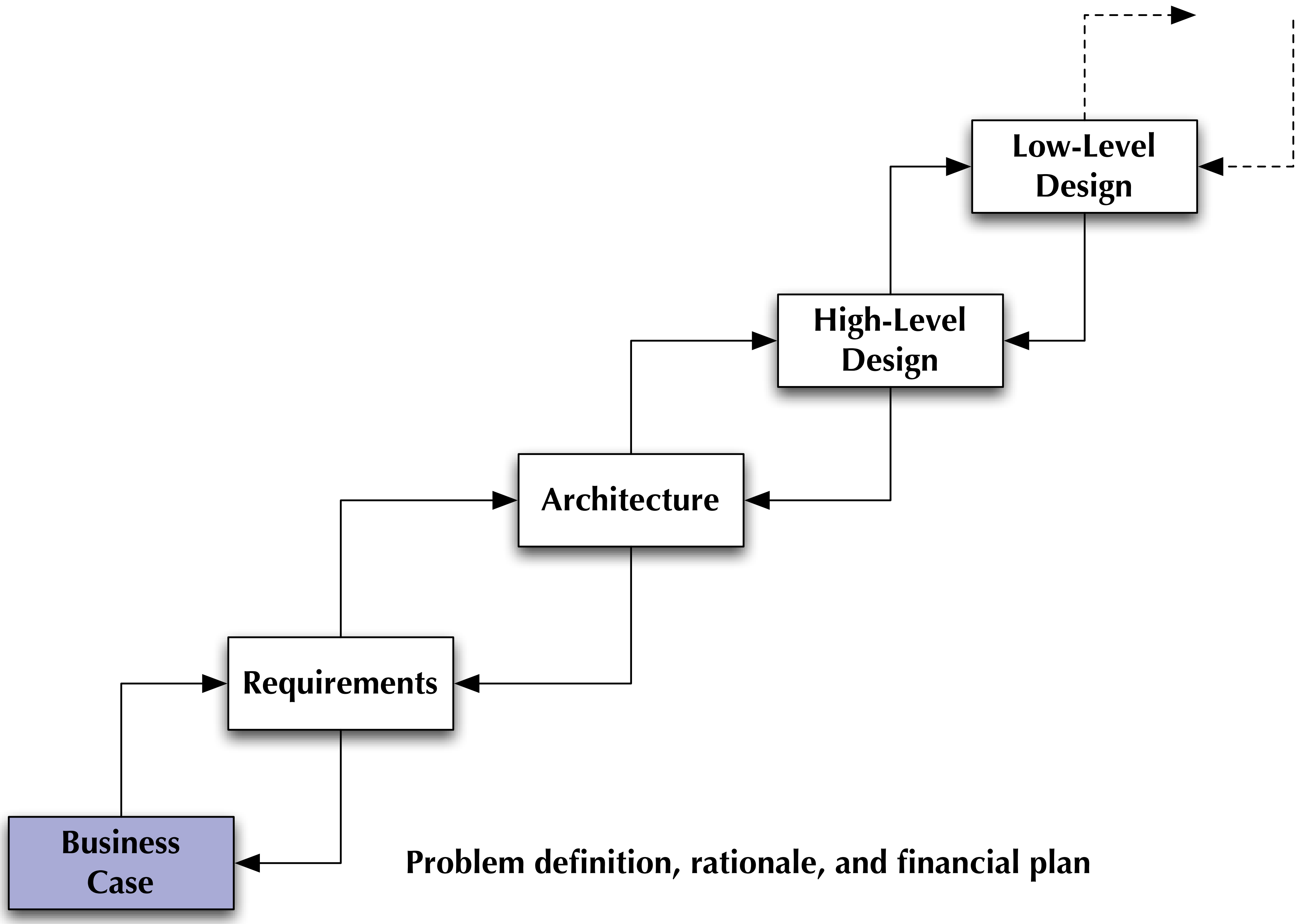
Software Architecture (VII)



The Role of Architecture (I)

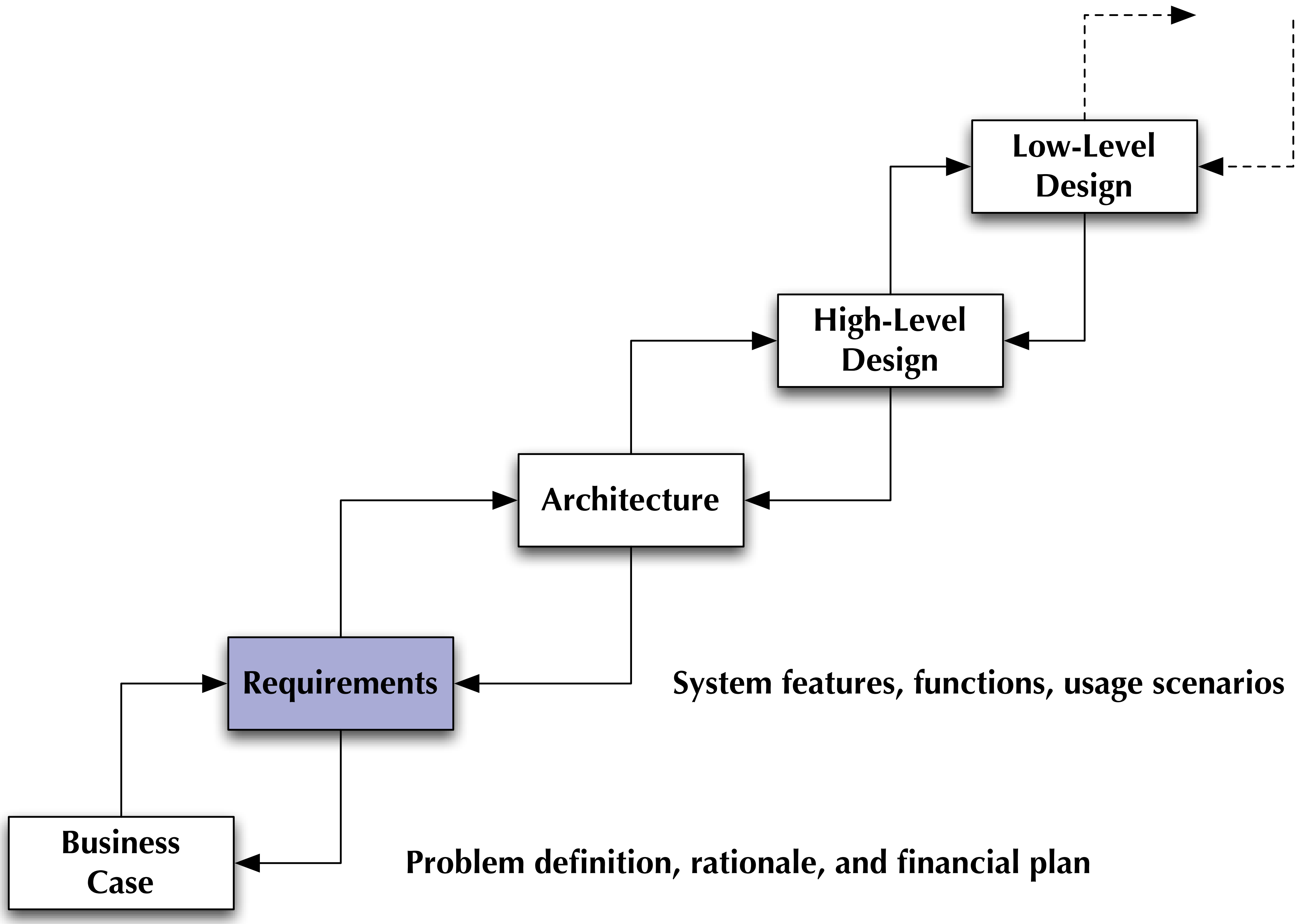


The Role of Architecture (II)



Problem definition, rationale, and financial plan

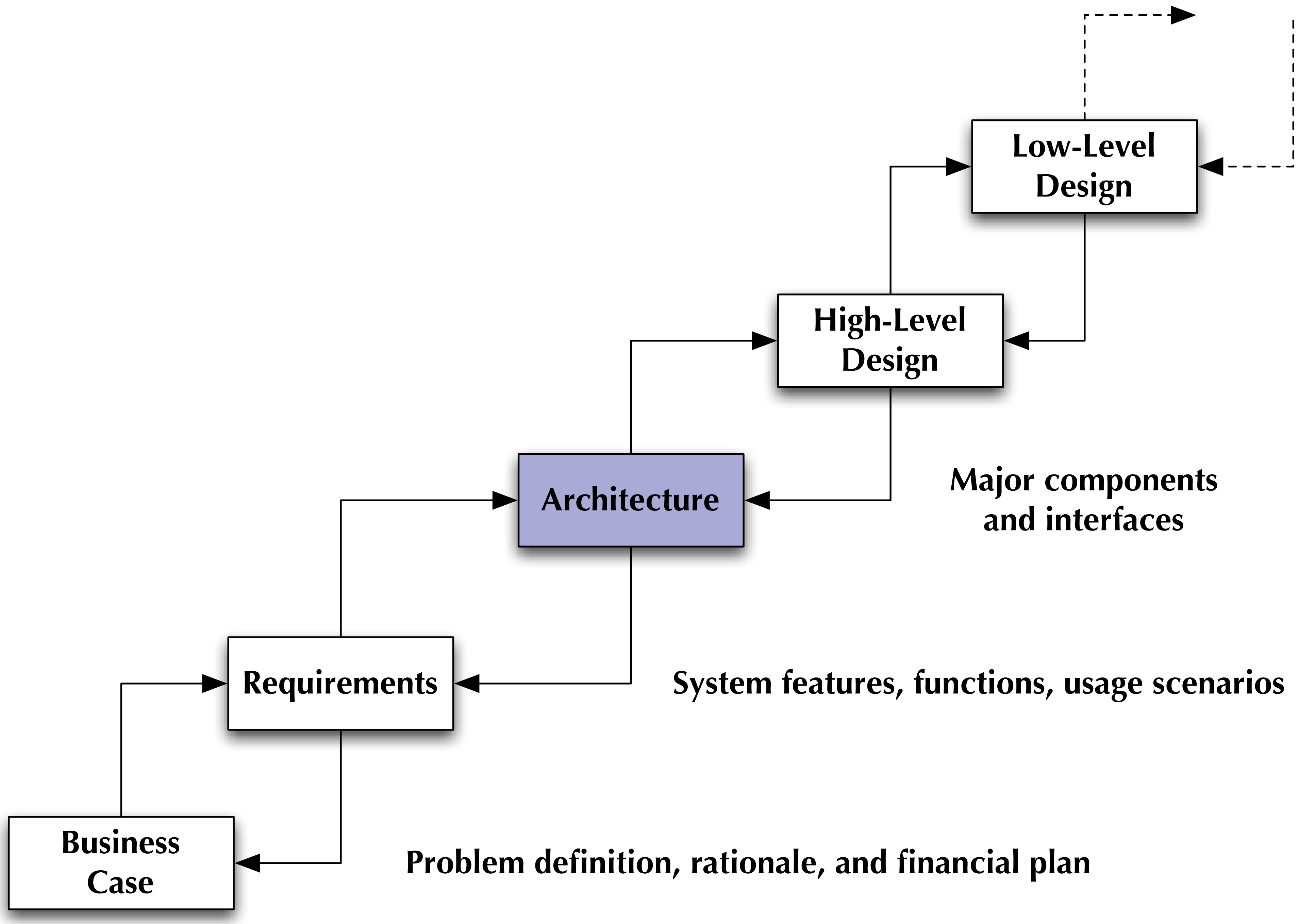
The Role of Architecture (III)



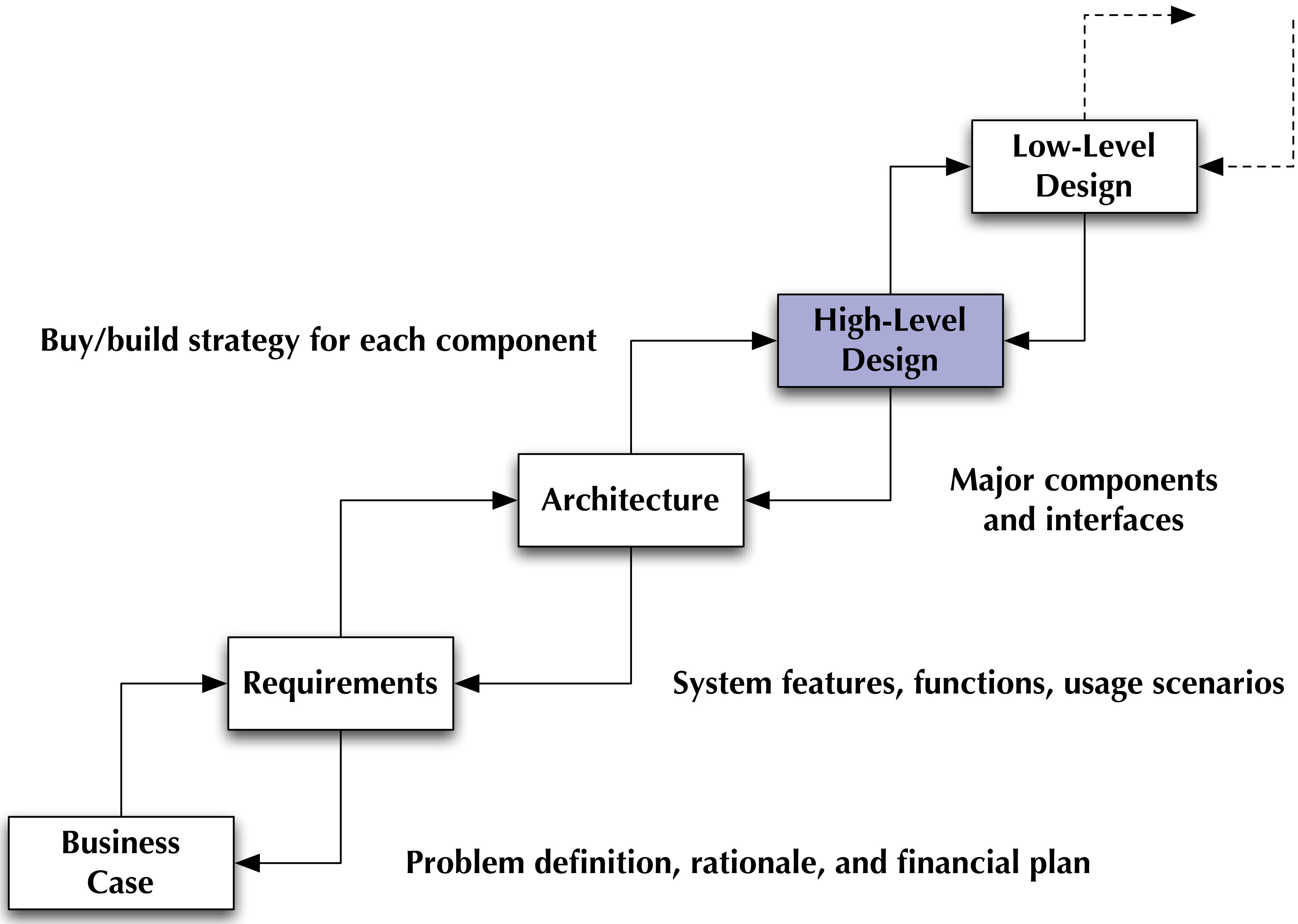
System features, functions, usage scenarios

Problem definition, rationale, and financial plan

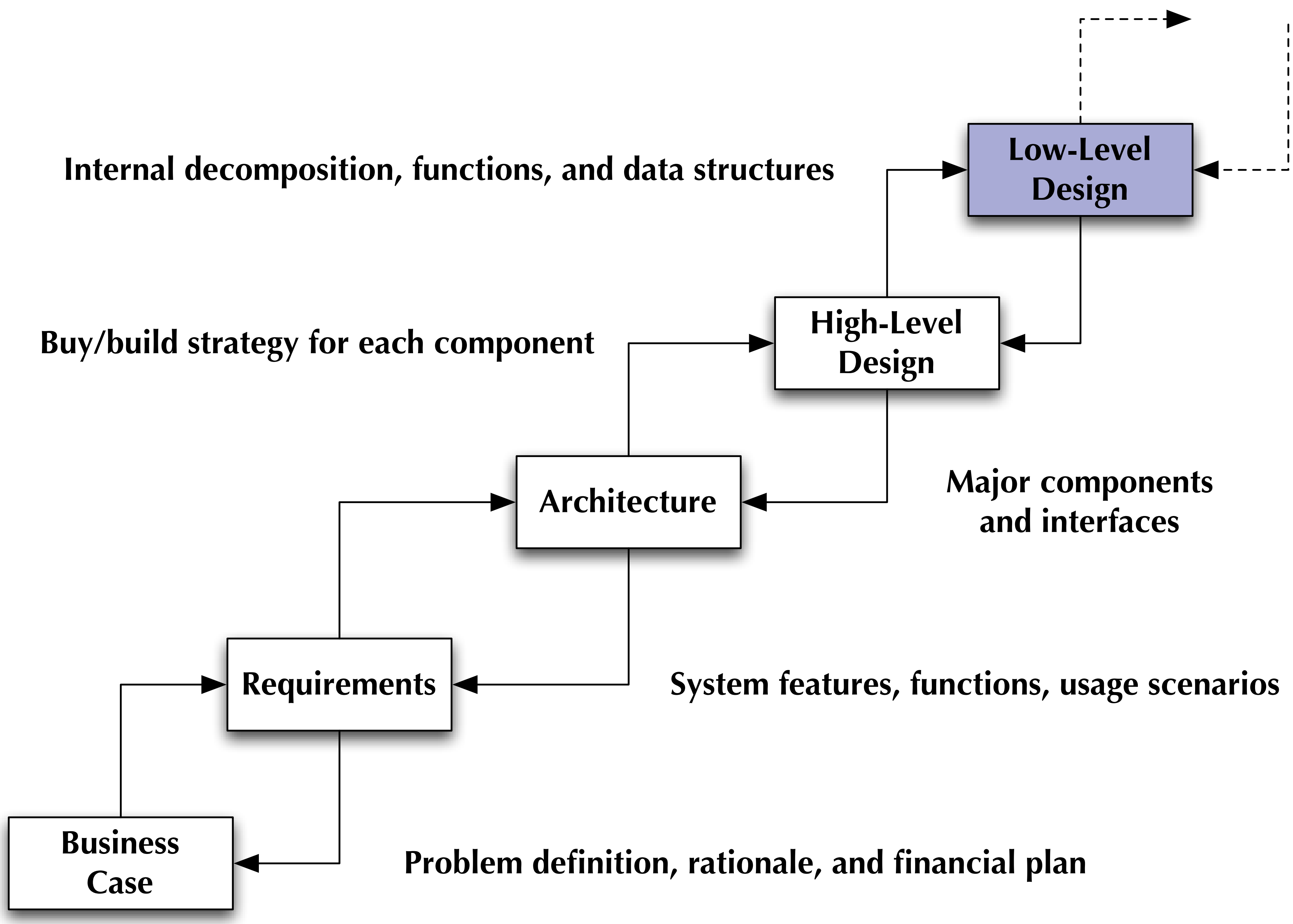
The Role of Architecture (IV)



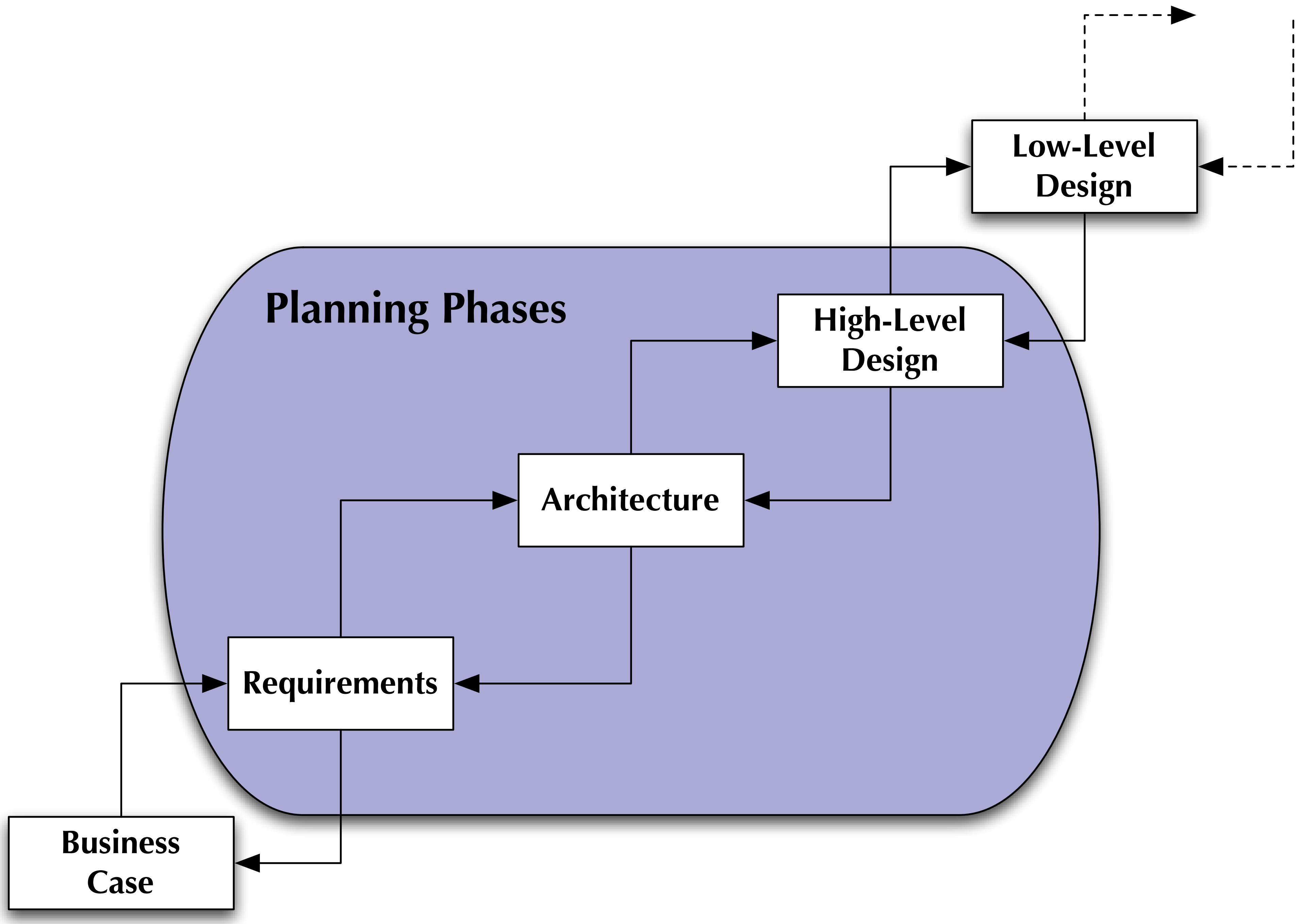
The Role of Architecture (V)



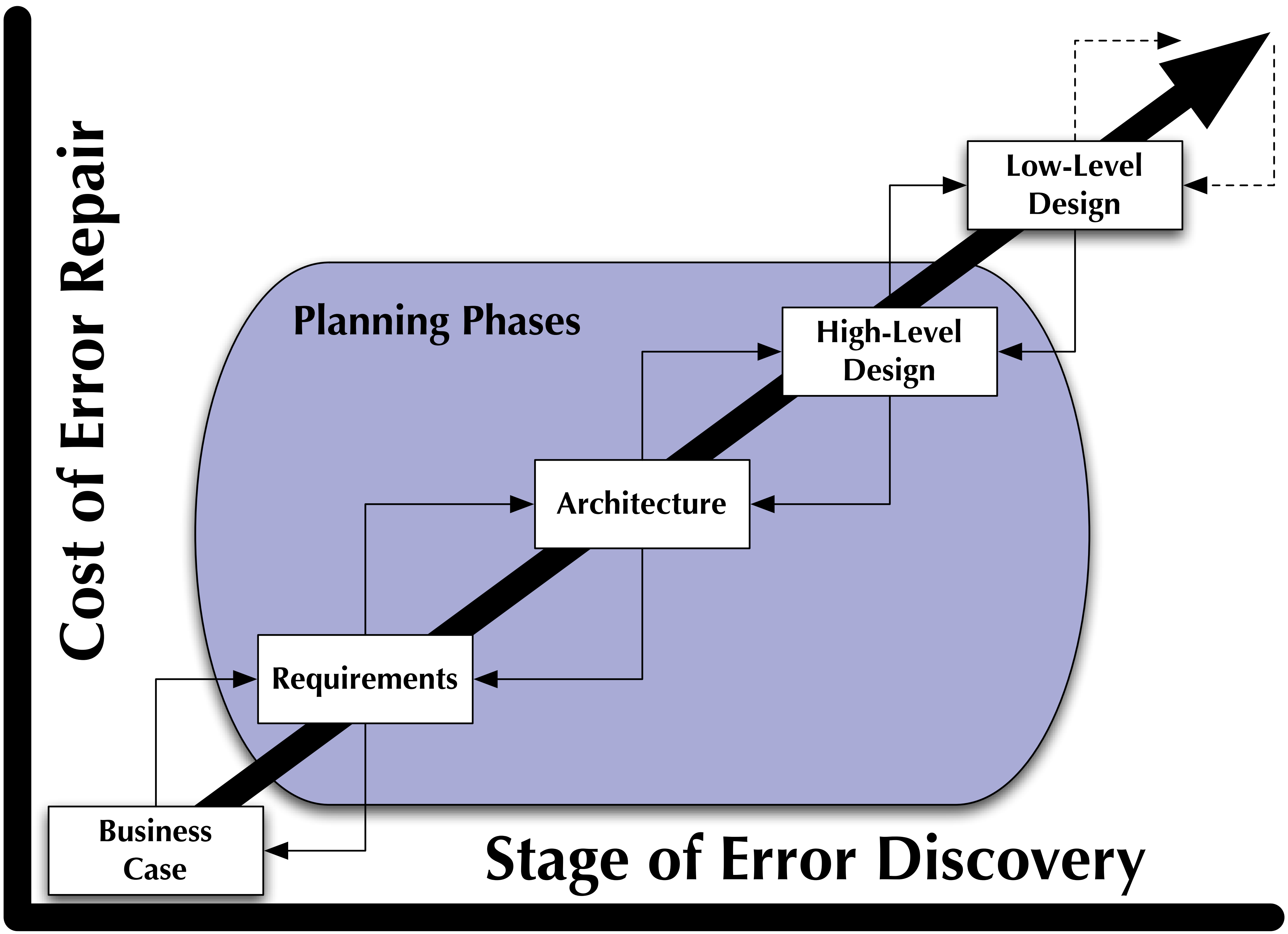
The Role of Architecture (VI)



The Role of Architecture (VII)



The Role of Architecture (VIII)



Component and Connector View

- **Components**: Computational elements or data stores
- **Connectors**: Means of interaction between components
- Useful Metaphor:
    - Polo, Water Polo, and Soccer (aka football in the rest of the world)
    - Similar in processors and data (components), but differ in connectors
- The C&C view describes a graph of components connected via connectors (often displayed as a boxes-and-arrows diagram)
    - It is mainly a runtime view of a system's architecture: what components exist at runtime and how do these components communicate with one another

Wrapping Up

- More Tools in your Toolbox: Solving Big Problems
    - Listen to the customer and figure out what they want you to build
    - Put together a feature list, in language the customer understands
    - Make sure your features are what the customer actually wants
    - Create blueprints of the system using use case diagrams
    - Break the big system up into lots of smaller pieces
    - Apply design patterns to the smaller sections of the system
    - Use basic OO A&D principles to design and code each smaller section
- Reviewed basic concepts of Software Architecture

Coming Up Next

- Lecture 12: Bringing Order to Chaos
  - Read Chapter 7 of the OO A&D book (warning: long chapter)
- Lecture 13: Originality is Overrated
  - Read Chapter 8 of the OO A&D book