Solving Really Big Problems
&
Bringing Order to Chaos

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 11 — 09/30/2008

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 Chapters 6 & 7 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
    - How do you use software architecture to guide the development process?
    - The three Q's of architecture
    - Risk and how to reduce it
    - Commonality Analysis
- Discuss the Chapters 6 & 7 Example: Gary's Game Framework
- Emphasize the OO concepts/techniques encountered in Chapters 6 & 7

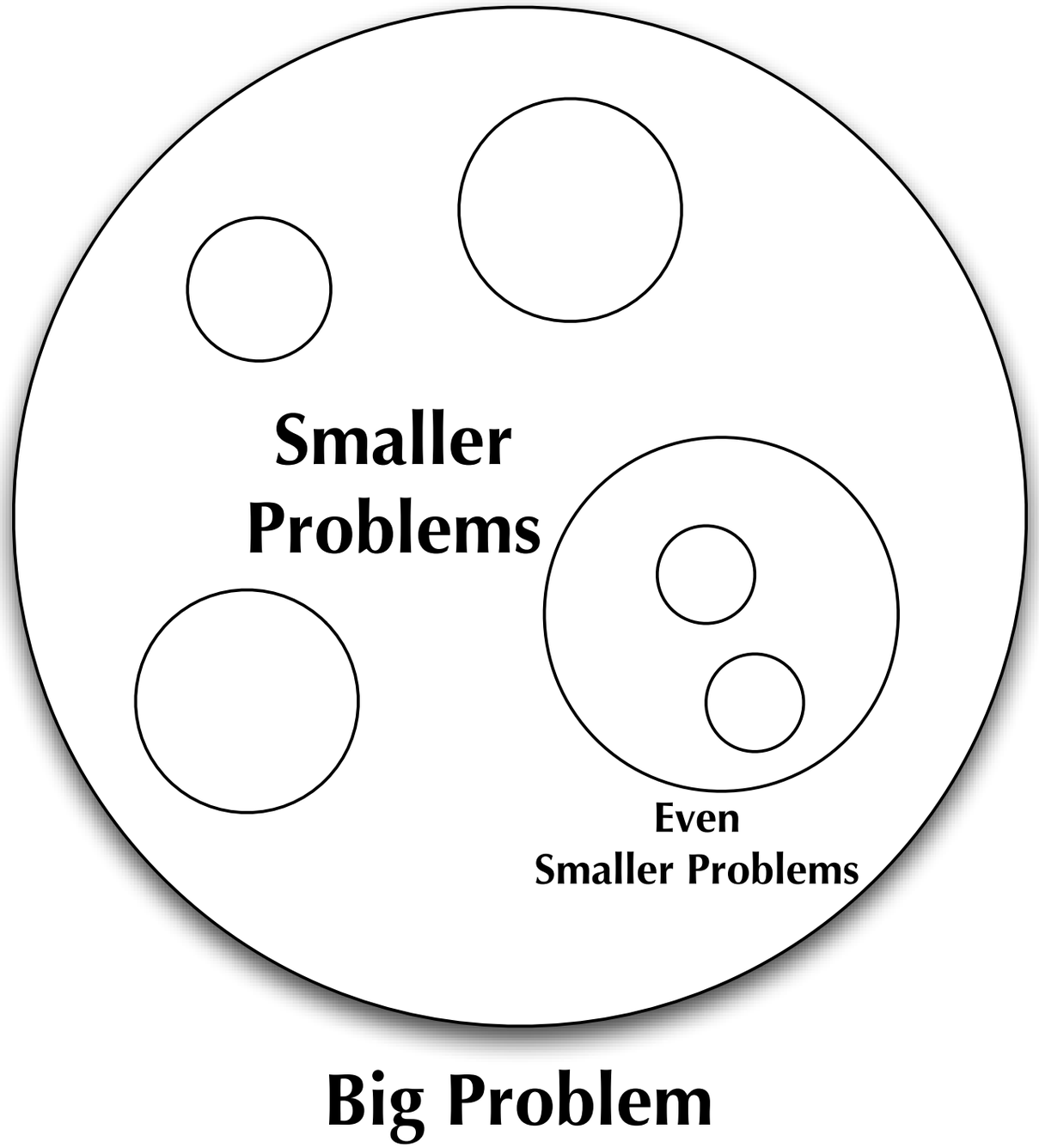
Living in Smallville?

- 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 terrain type.

6. The framework keeps up with whose turn it is an coordinates basic movement

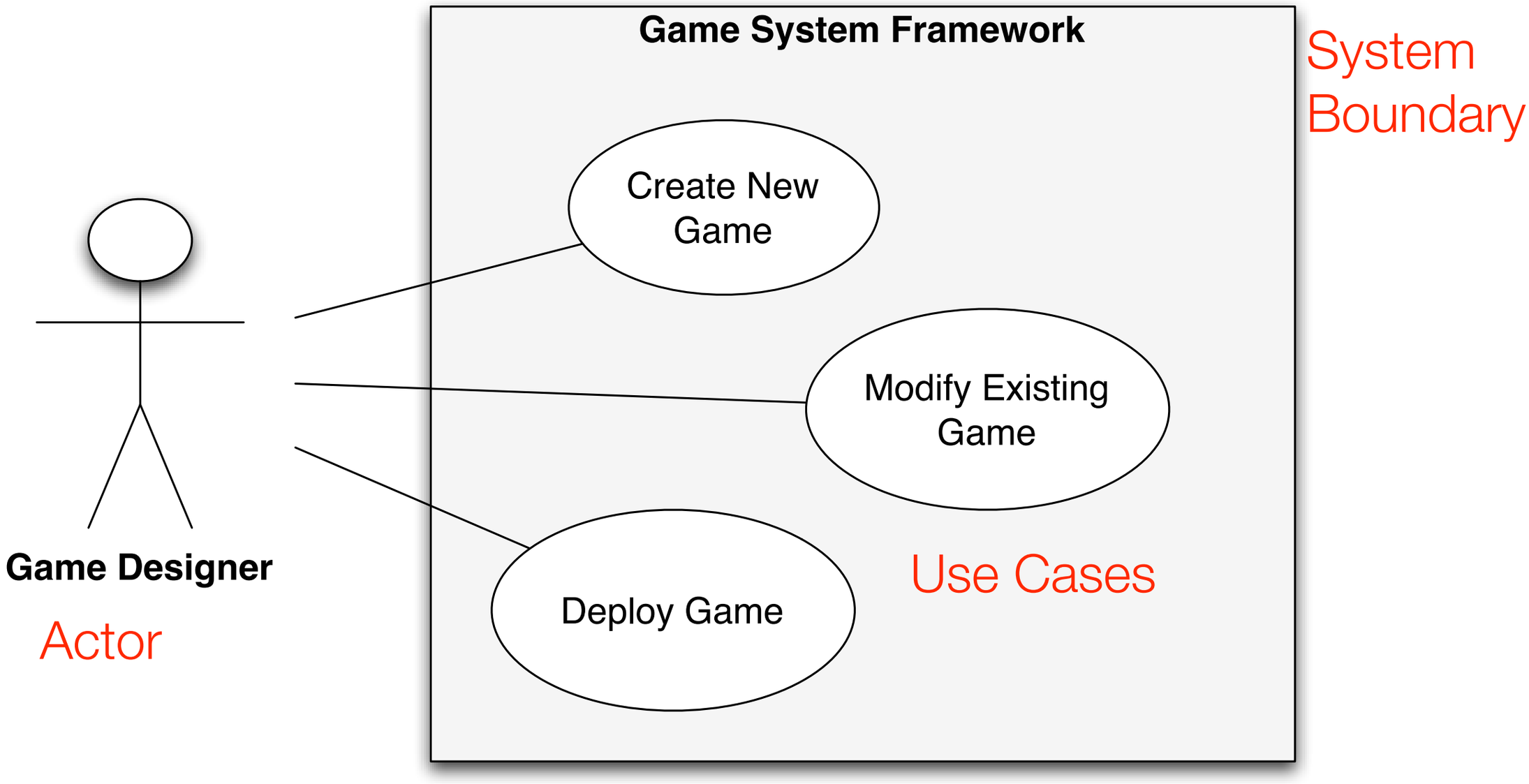
Feature vs. Requirement?

- 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

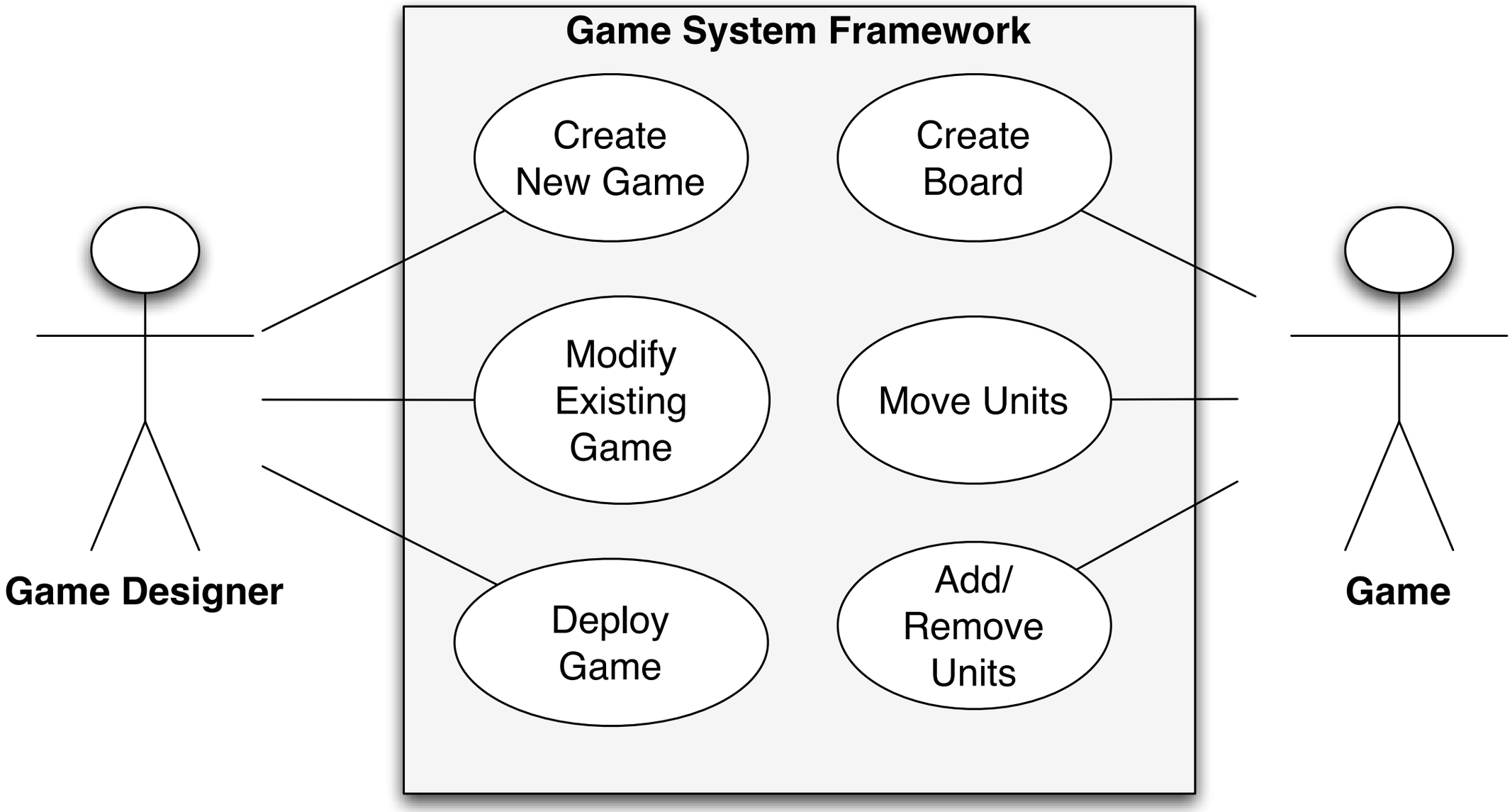


Diagram gives “big picture” view of framework

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

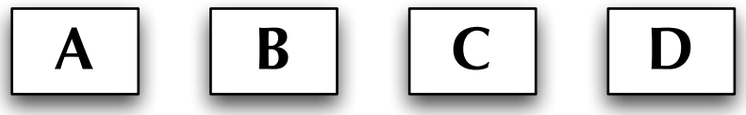
Introduction to Software Architecture (II)

- 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

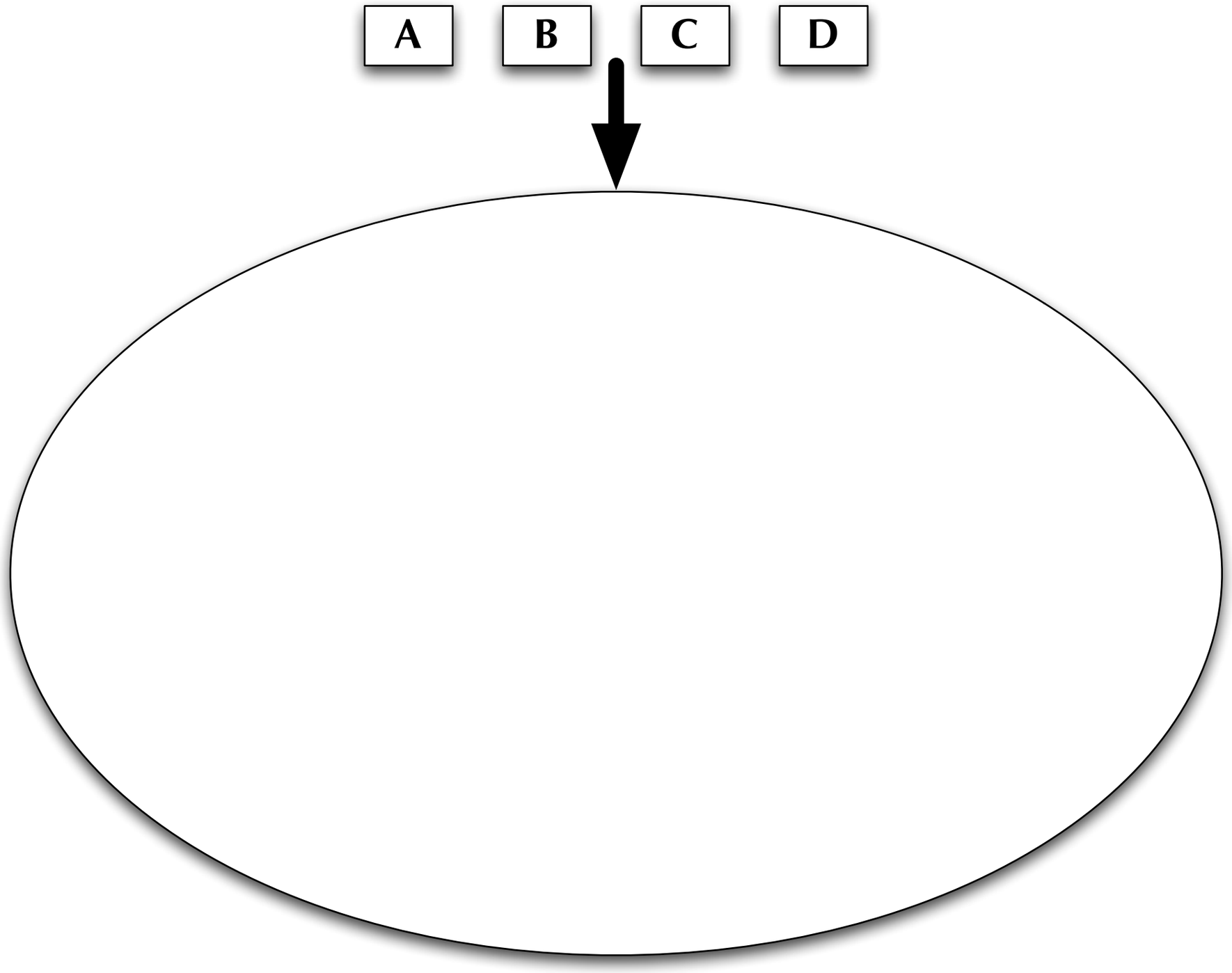
Introduction to Software Architecture (III)

- 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

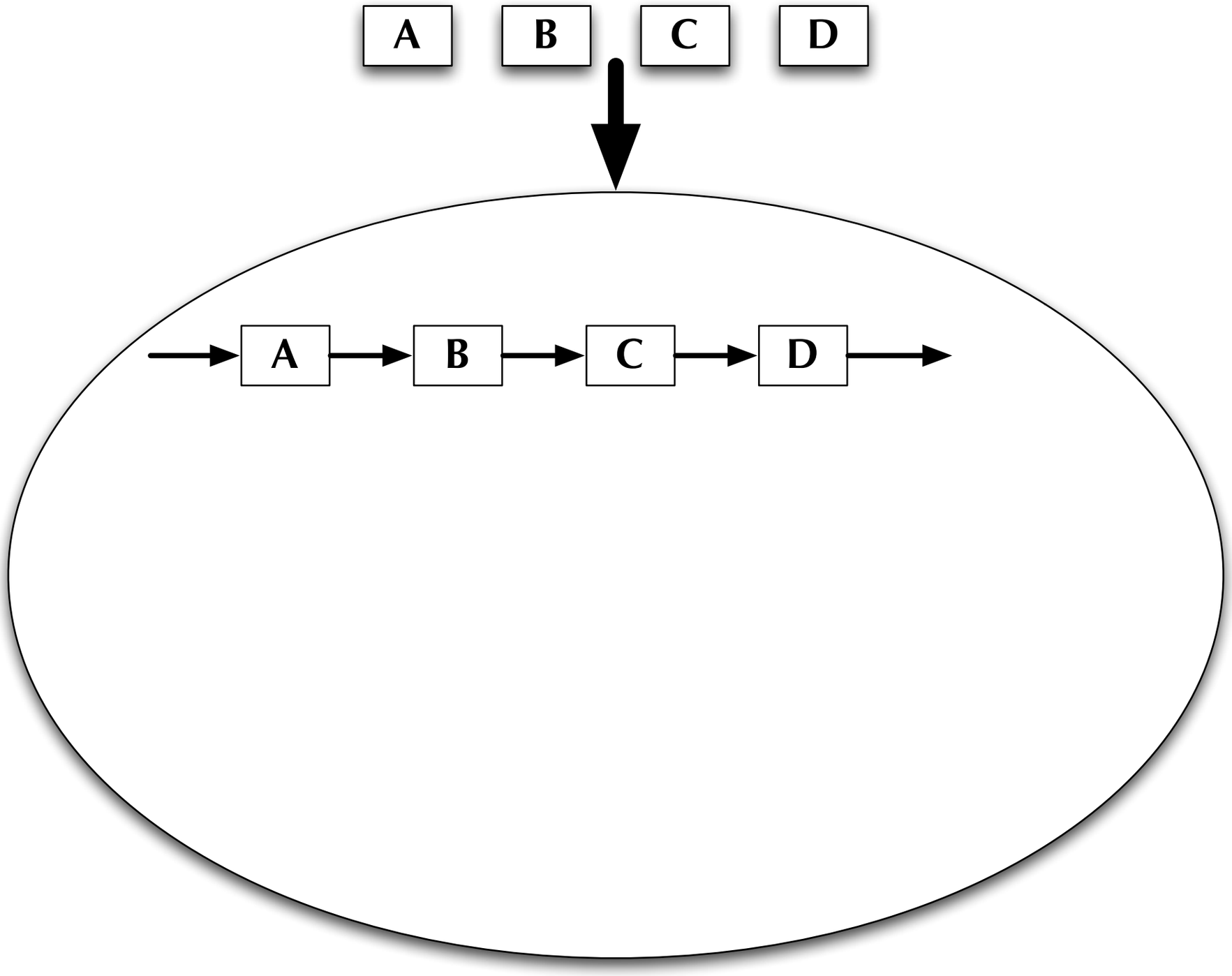
Software Architecture (I)



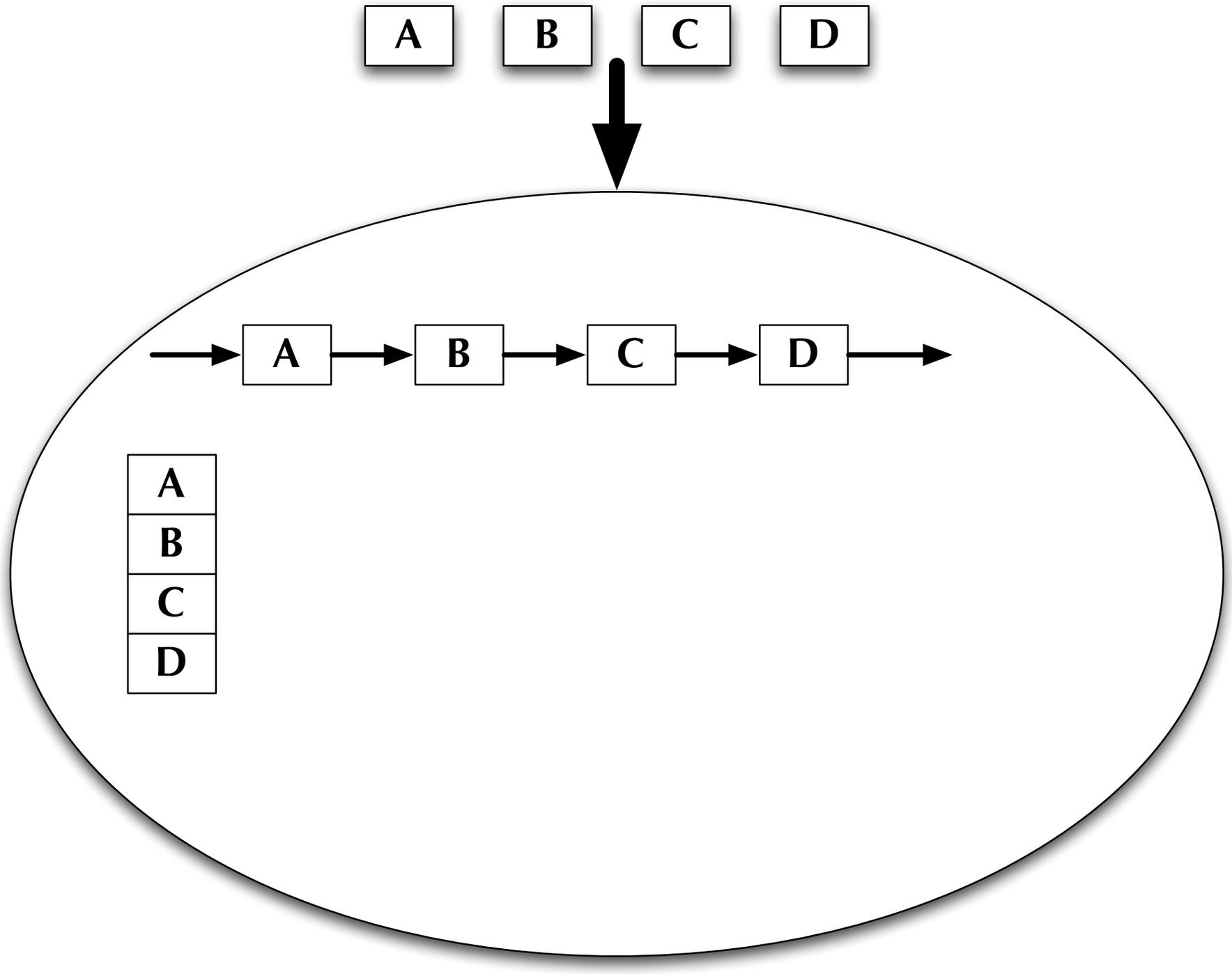
Software Architecture (II)



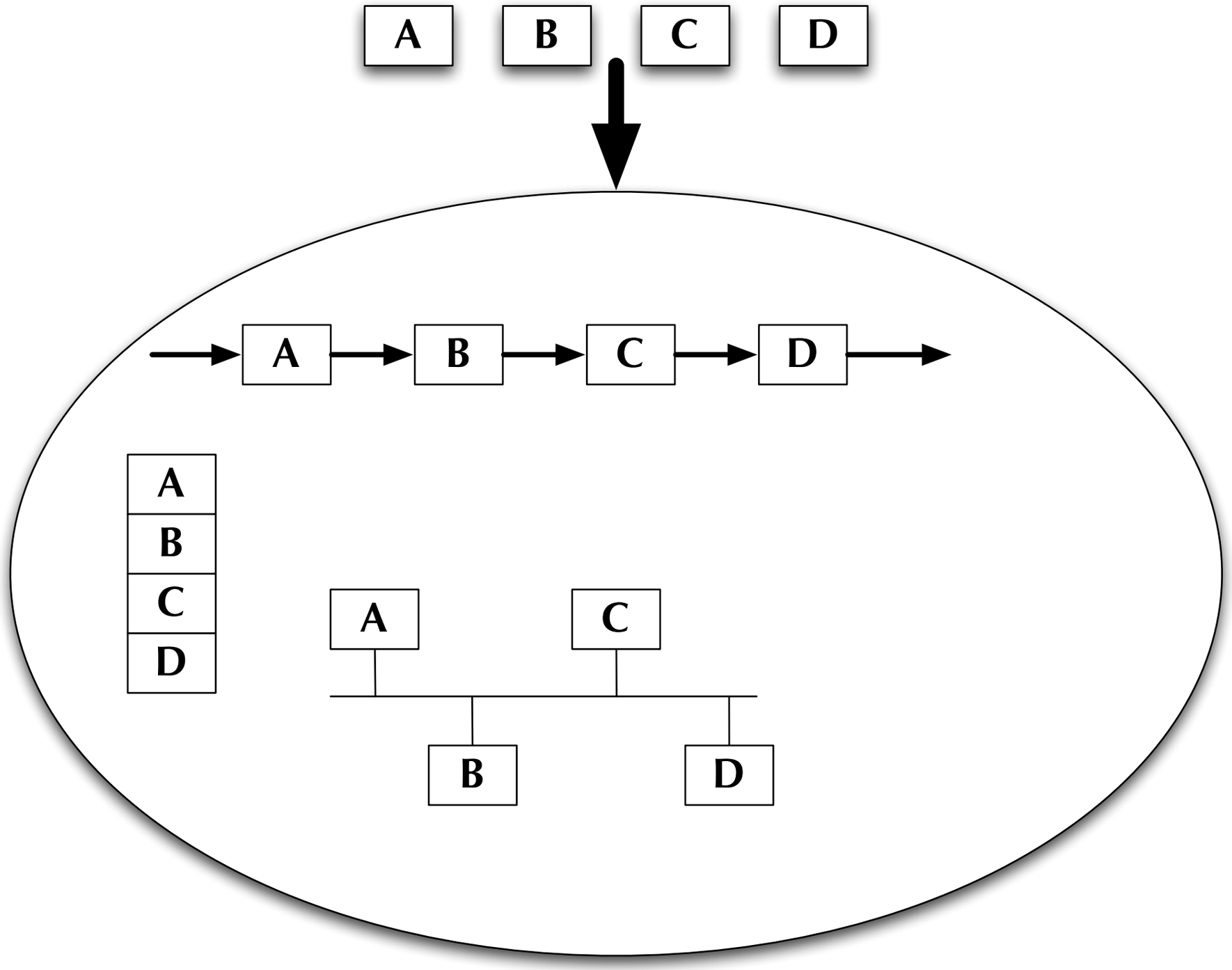
Software Architecture (III)



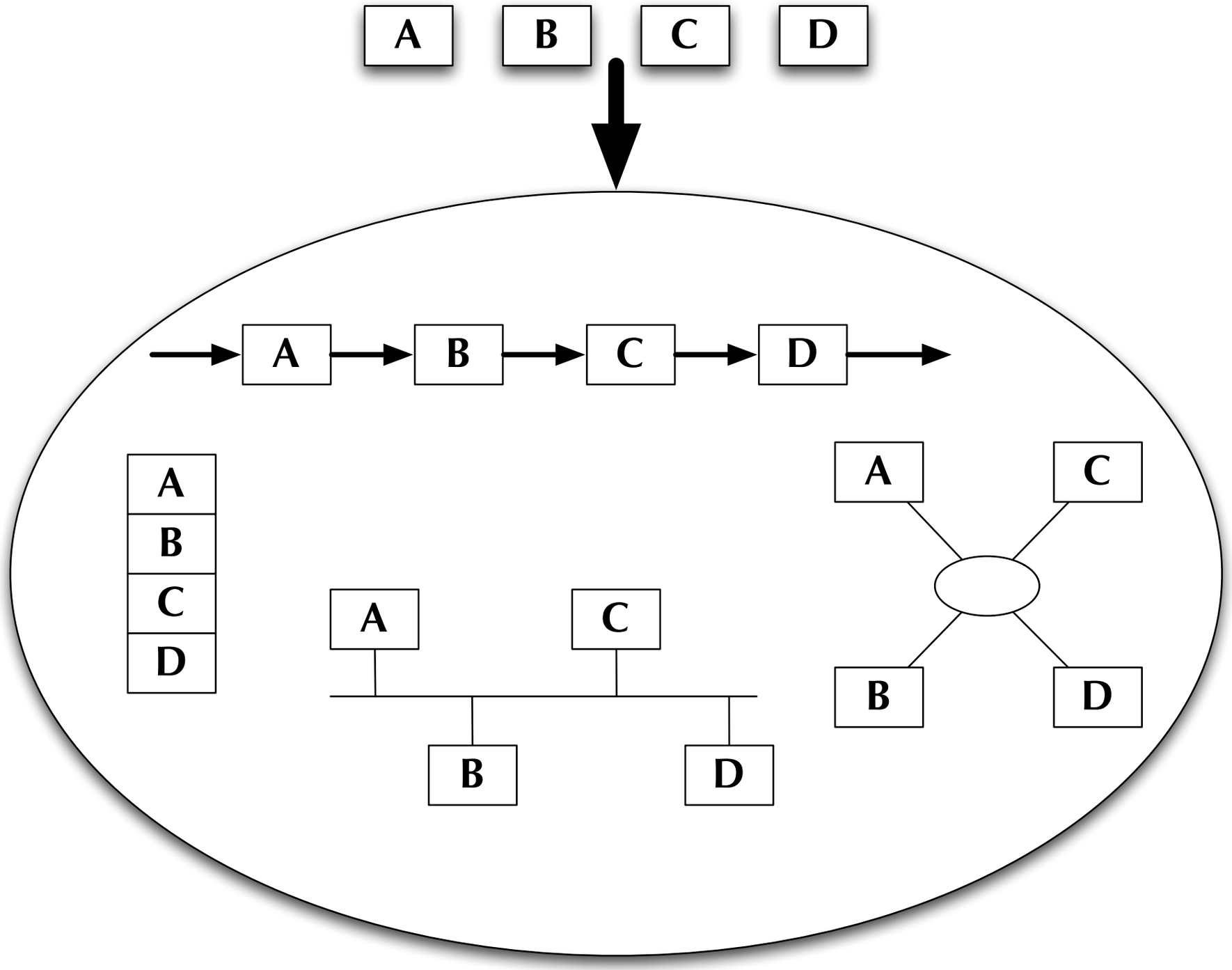
Software Architecture (IV)



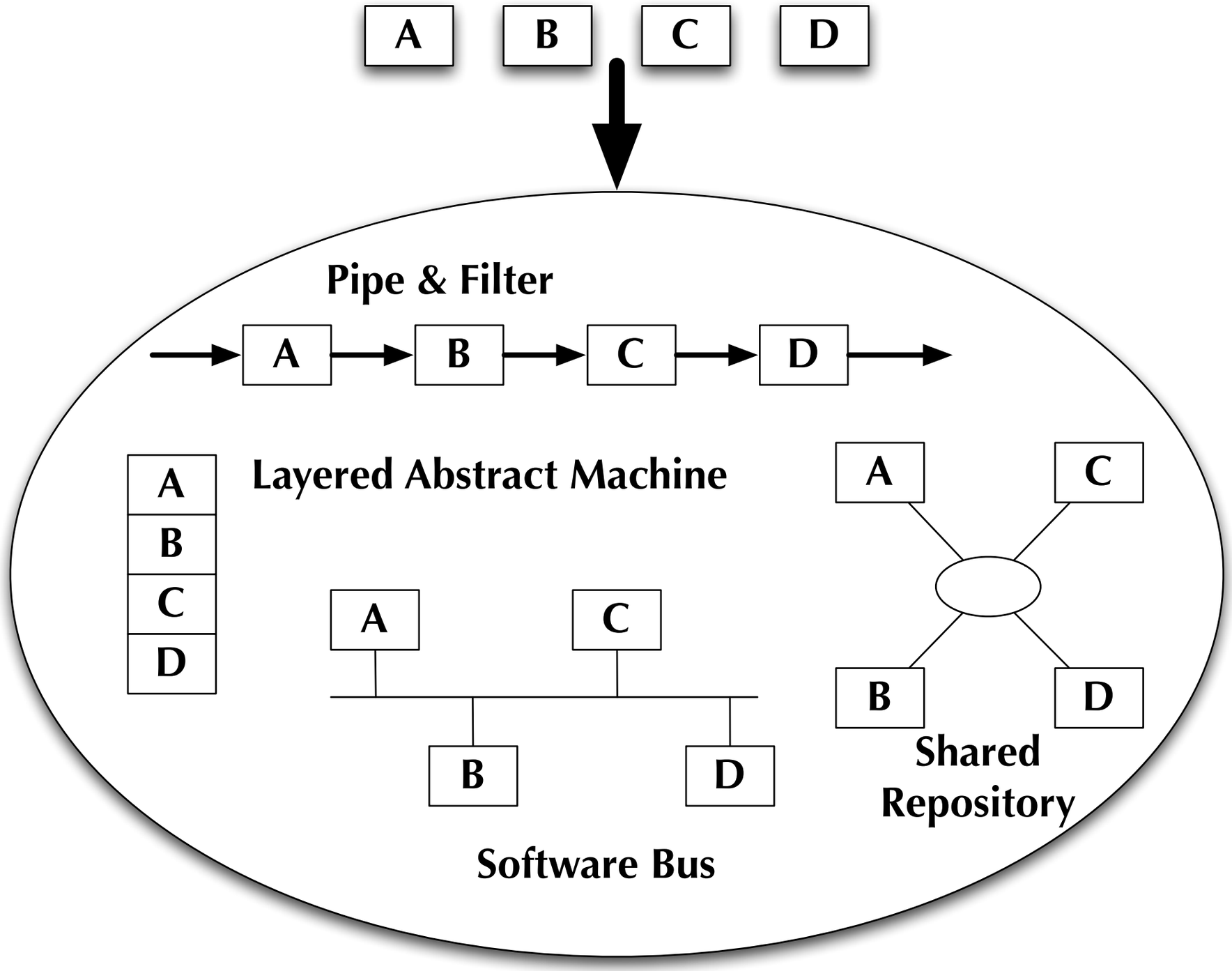
Software Architecture (V)



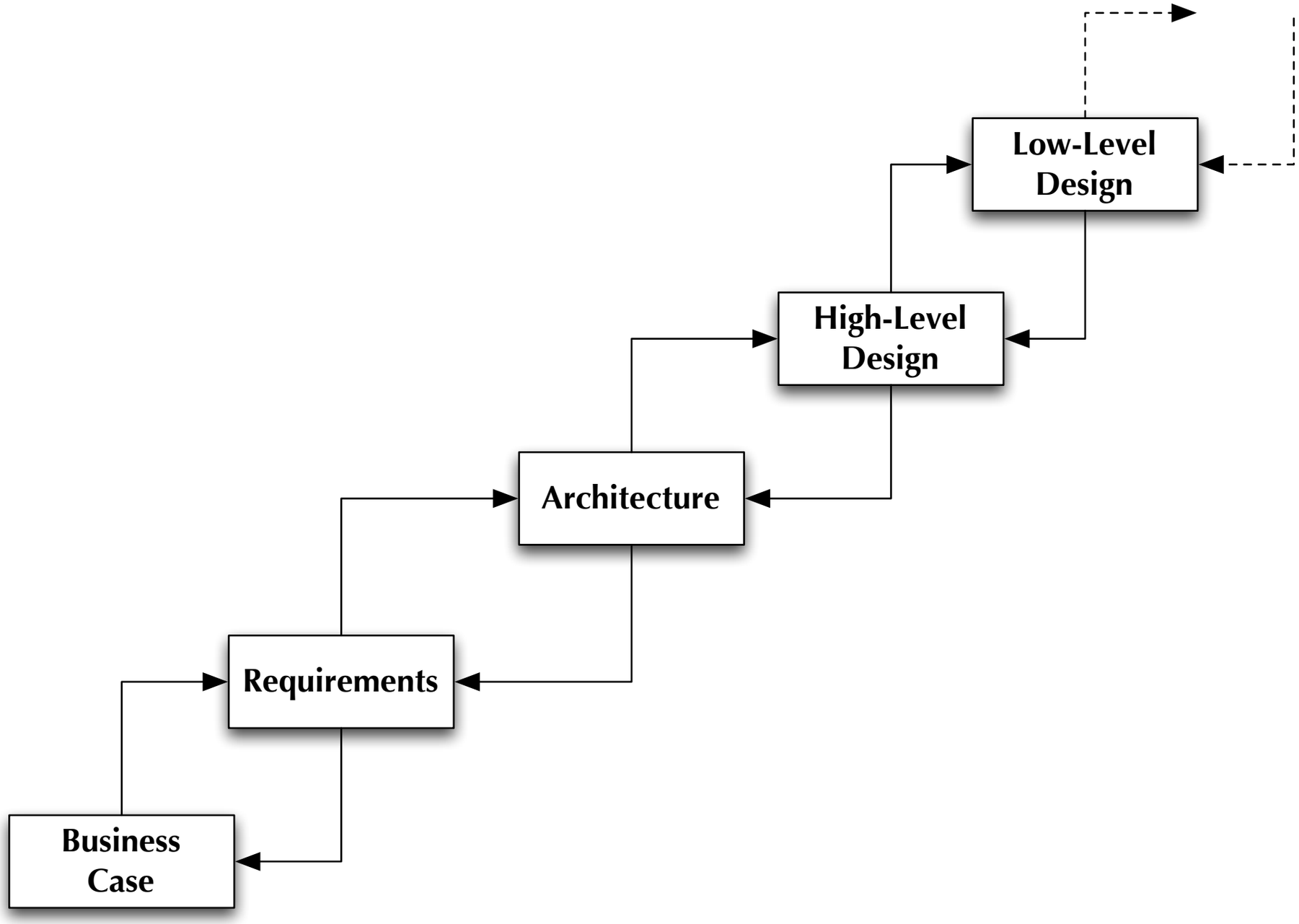
Software Architecture (VI)



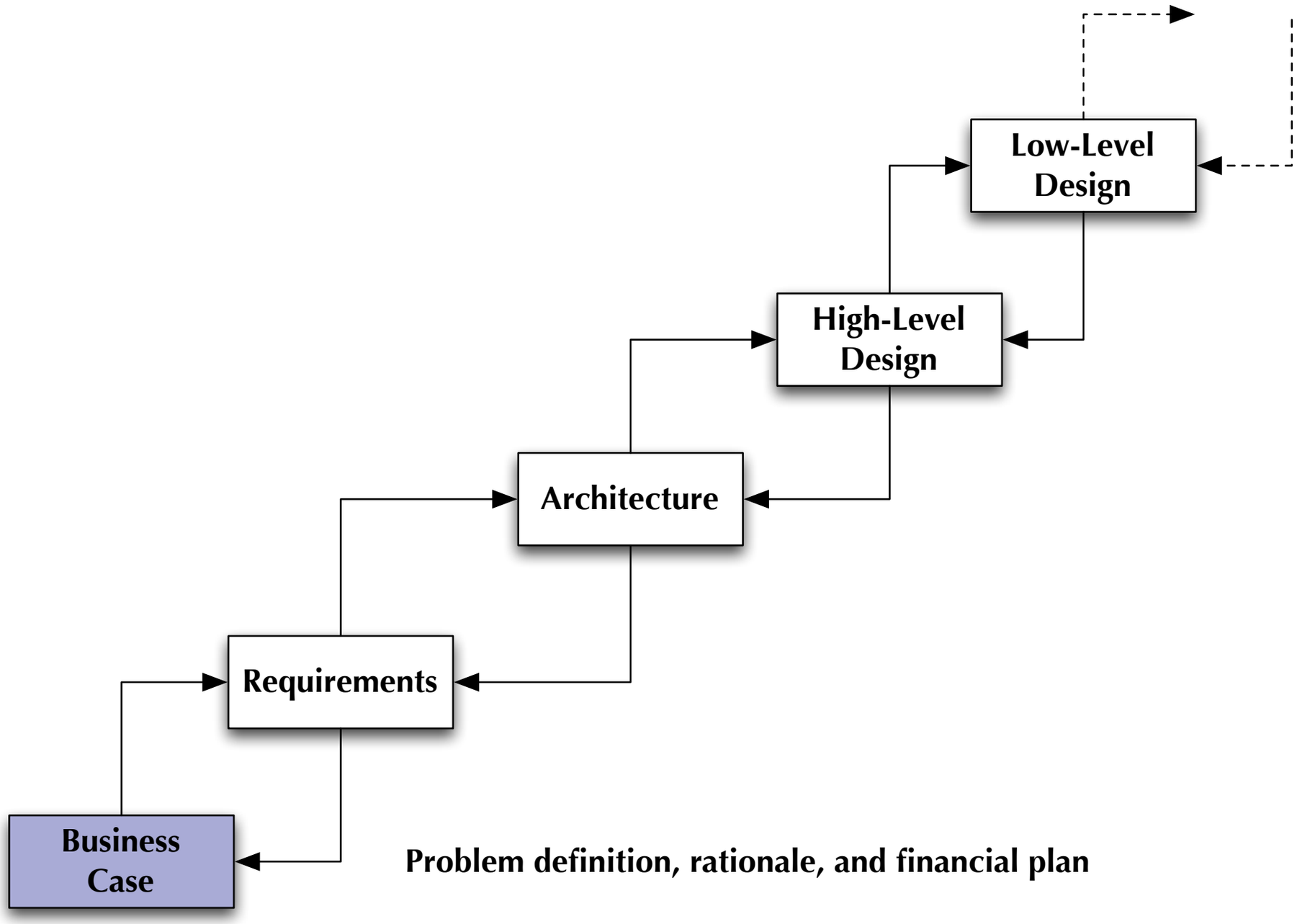
Software Architecture (VII)



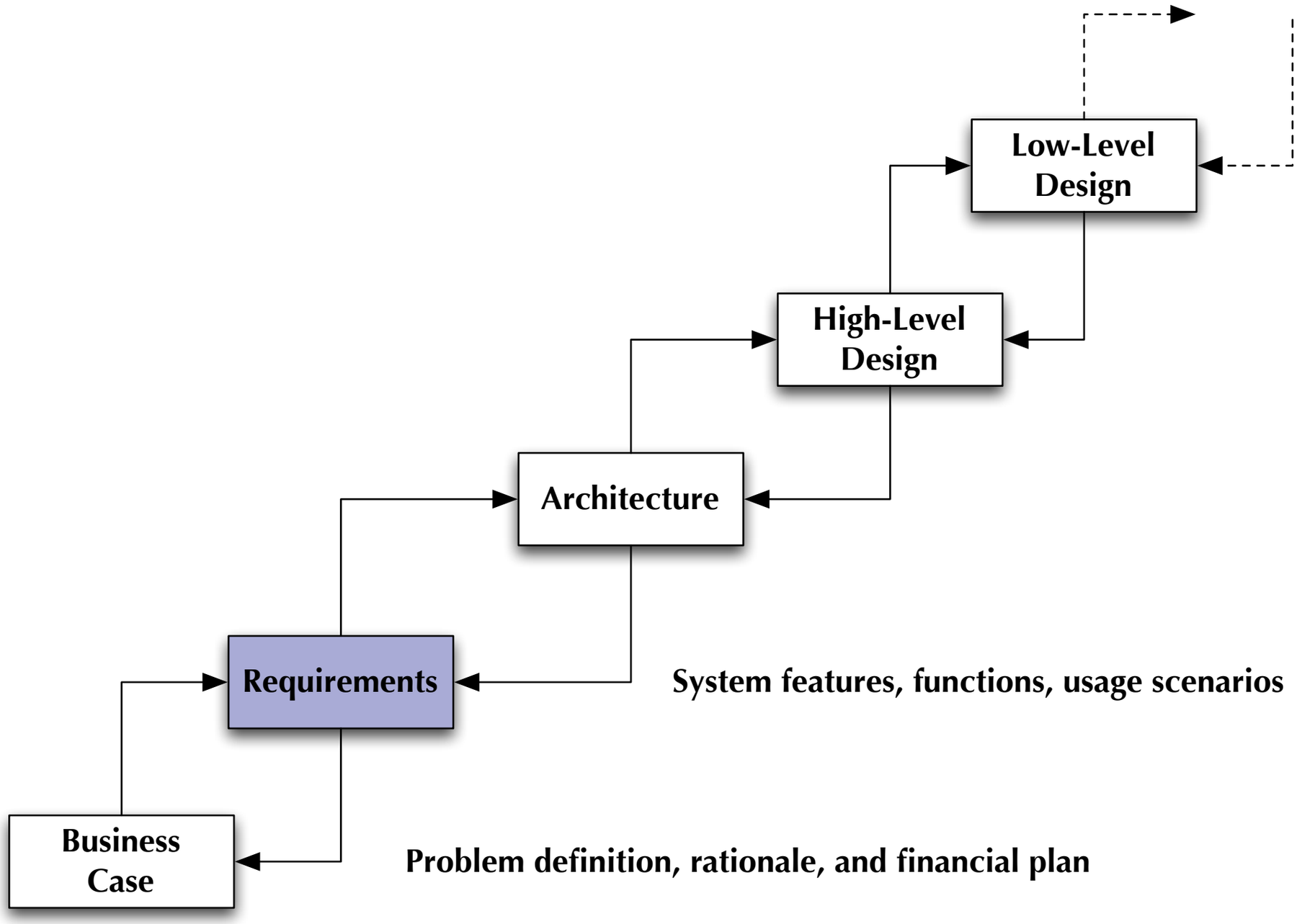
The Role of Architecture (I)



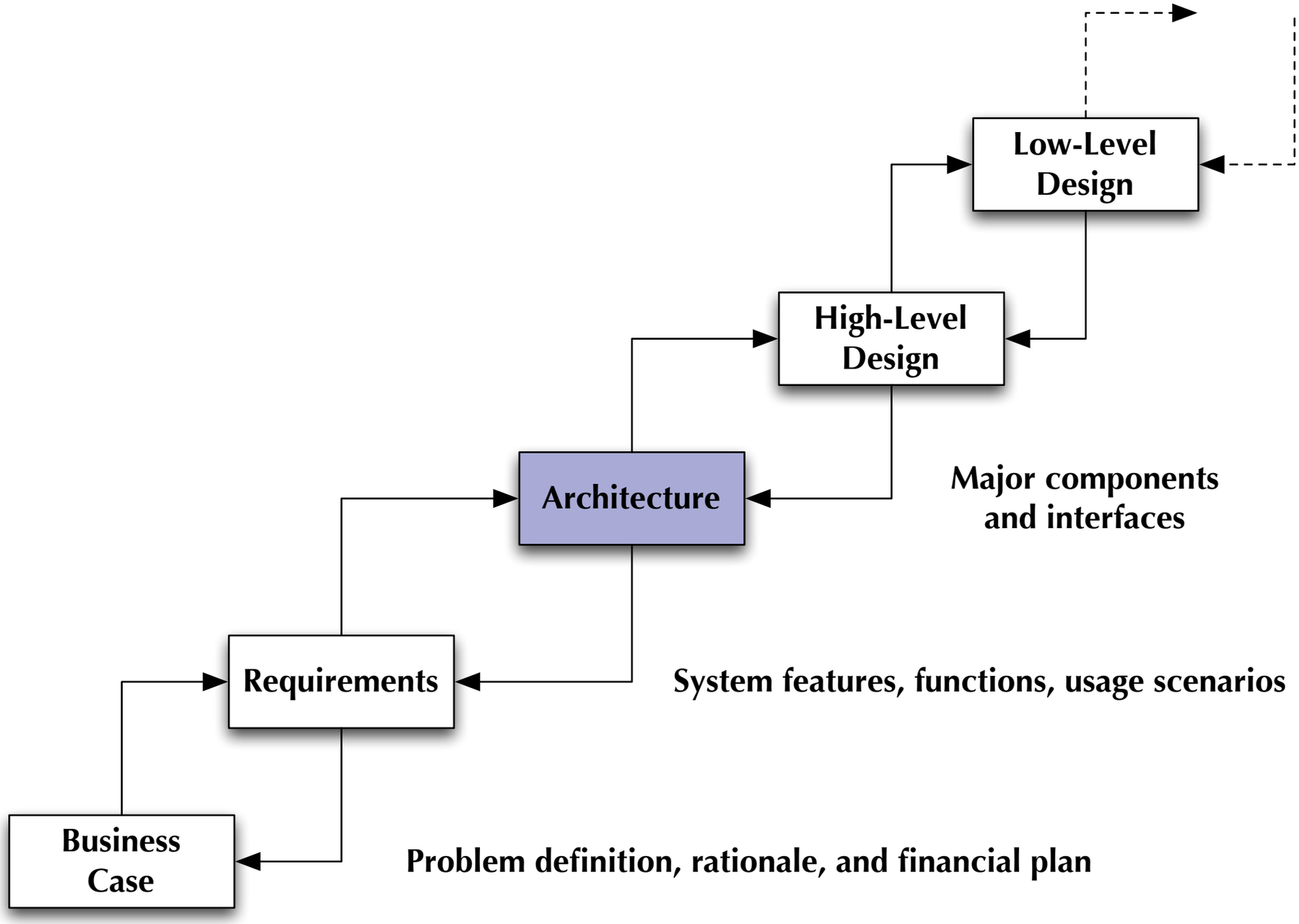
The Role of Architecture (II)



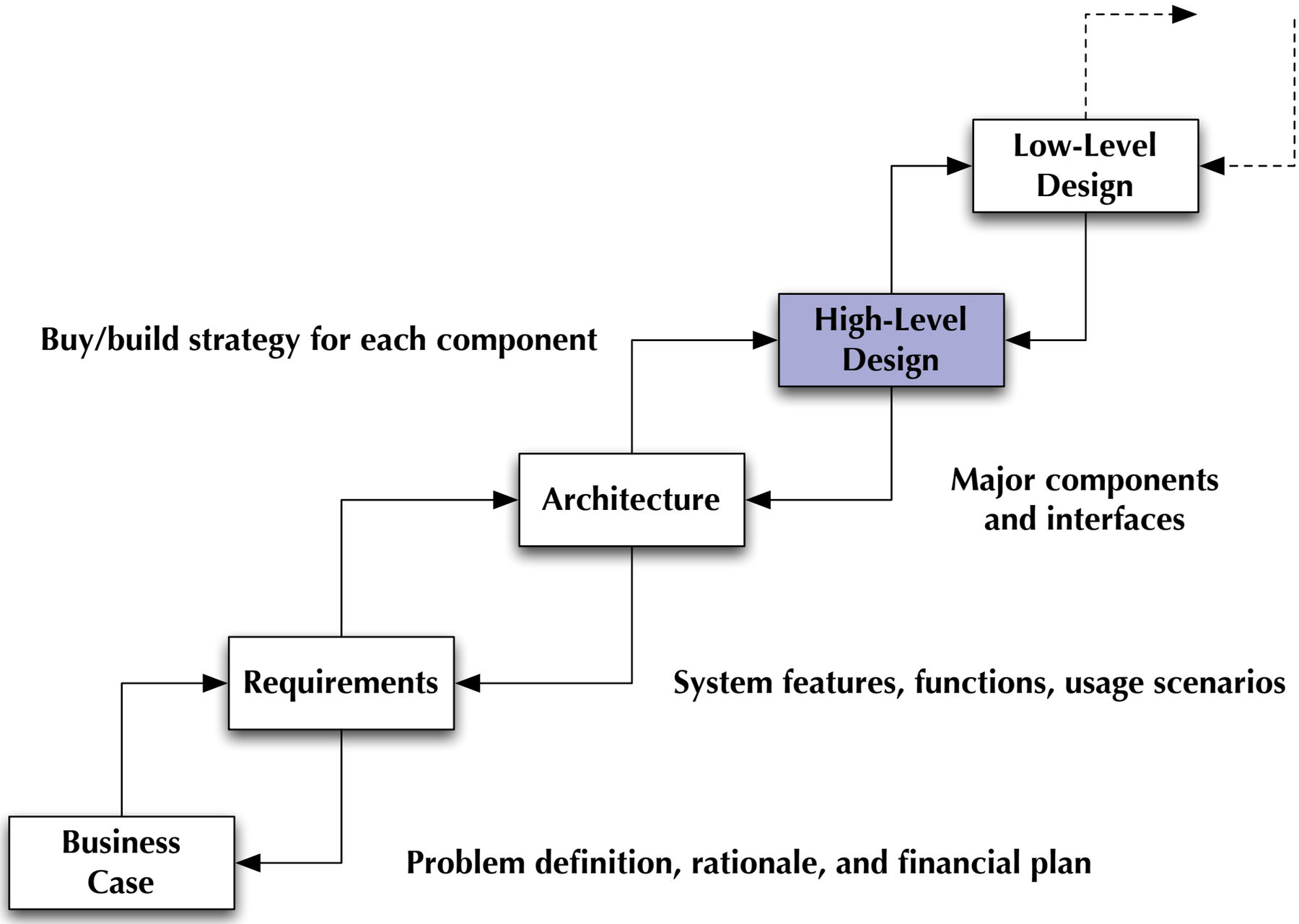
The Role of Architecture (III)



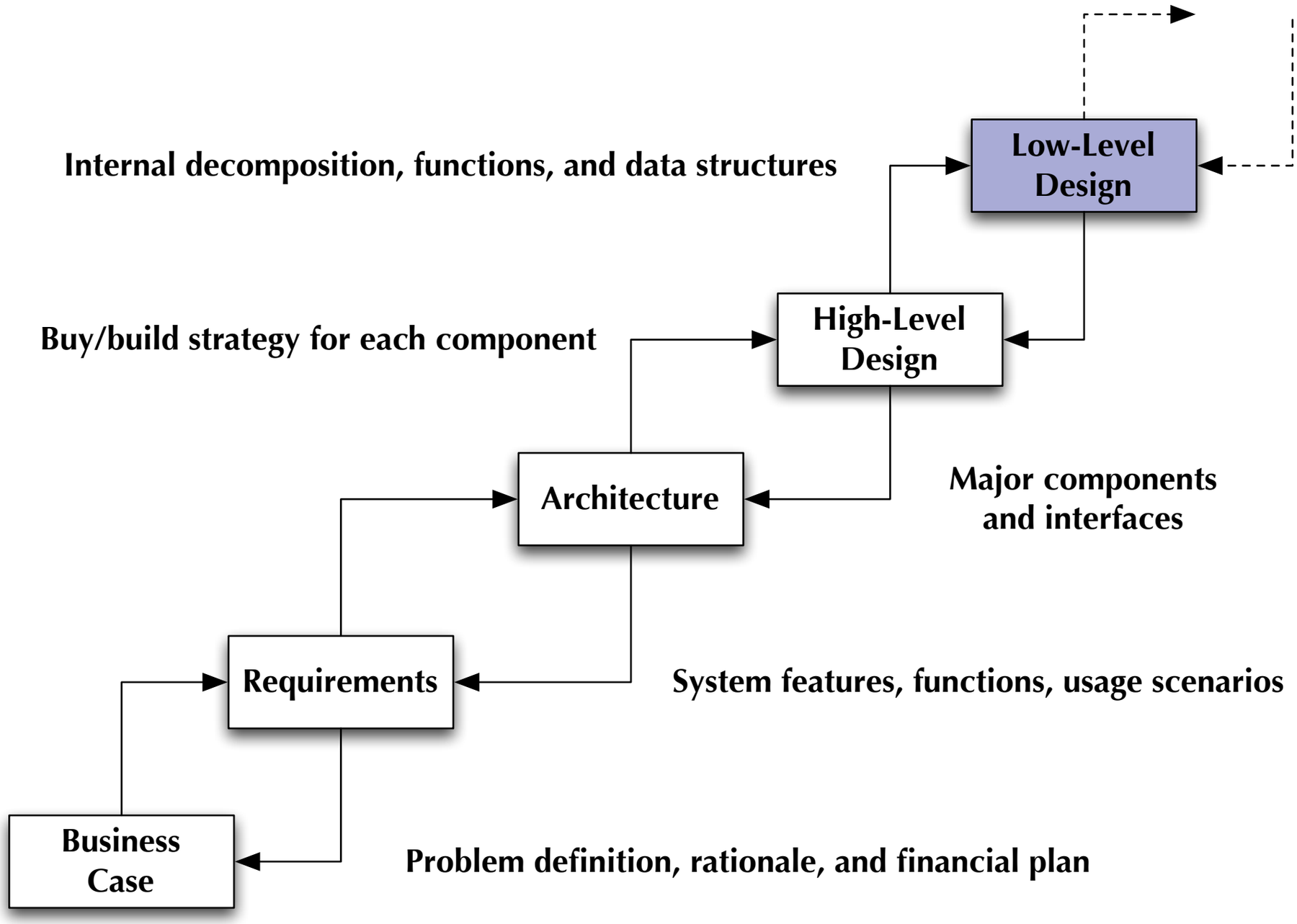
The Role of Architecture (IV)



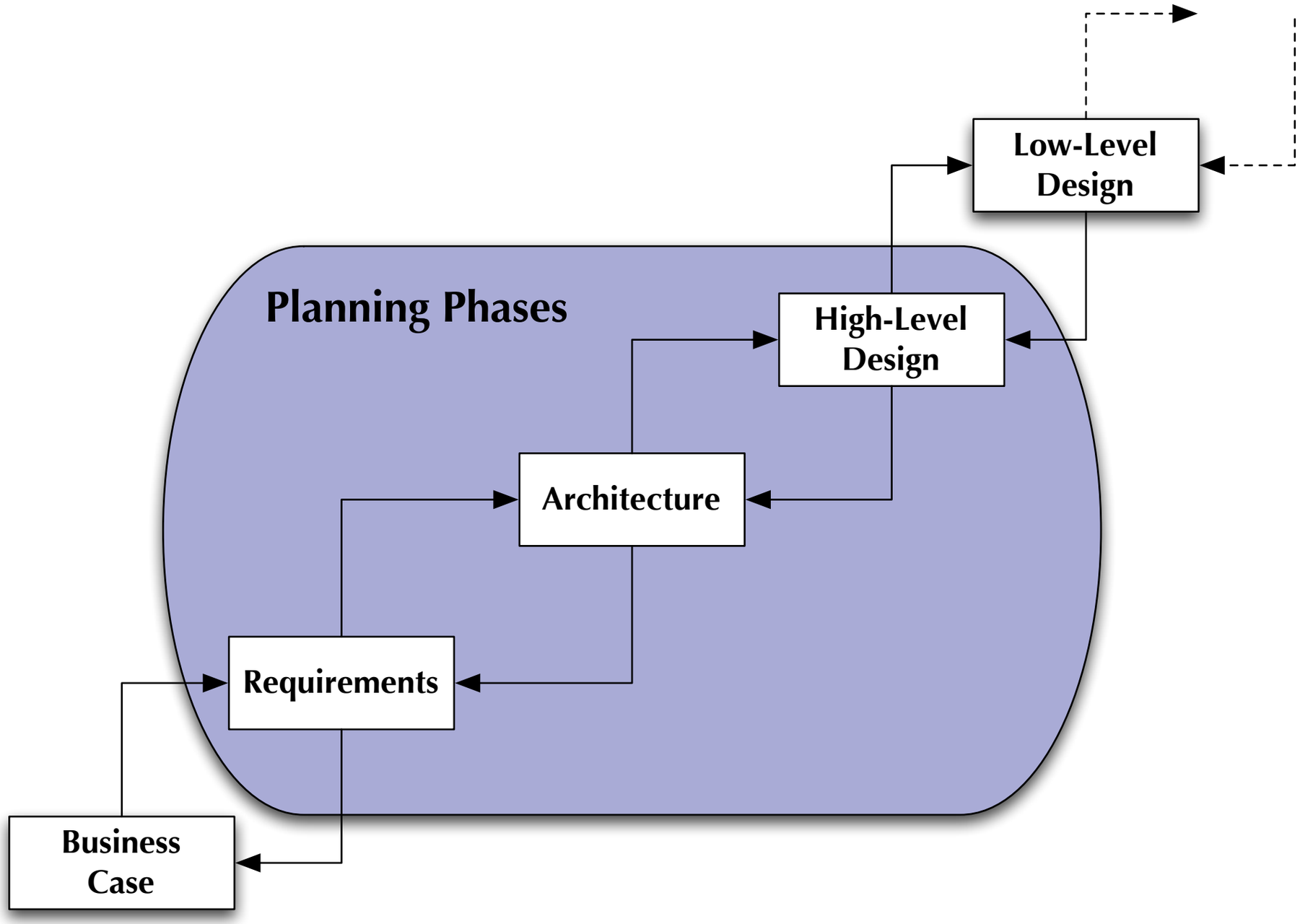
The Role of Architecture (V)



The Role of Architecture (VI)



The Role of Architecture (VII)



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

Bringing Order to Chaos

- At the start of a development project for a large, complex software system
  - You don’t have a lot of information
  - You don’t know a lot about your domain
- And yet…
  - You are expected to make
    - implementation decisions (concerning expensive pieces of middleware)
    - decisions regarding non-functional requirements
    - progress on decomposing the problem domain into manageable pieces
    - progress on identifying the features of the system under design

How Does One Start?

- Indeed after doing the work that we just described…
  - domain analysis
    - talking to customer
    - developing feature list
    - developing use case diagram
      - which requires hard work like identifying all of the different types of users and their major tasks
  - decomposing “big problem” into “smaller modules”
- … you may be feeling **overwhelmed** with the information you have so far
  - or have **doubts about its accuracy** (which then leads to **paralysis**)
- So, how does one identify what to do next?

Answer: Software Architecture

- It's not enough to identify the individual pieces of a big problem
  - You also need to know how those pieces fit together
  - You need to have a mechanism for prioritizing your work on those pieces
- The missing piece is the software architecture for your system
  - A software architecture **provides a structure to the design** of a software system, **highlights the most important parts** of your system, and the **relationships between those parts**
- A second definition
  - A software architecture is the **organizational structure** of a system, including its **decomposition into parts**, **their connectivity**, **interaction mechanisms**, and the **guiding principles and decisions** you use in the design of a system
- An architecture can take a chaotic mess and turn it into a well-ordered app!

Back to the Three Step Process

- Our three step OO A&D process
  - Make sure your software does what the customer wants it to do
  - Apply basic OO principles to add flexibility
  - Strive for a maintainable, reusable design
- points the way towards what to do first: **Focus on Functionality**
  - As such, we will start with what the customer wants to do
  - And for large systems that means looking at the feature list

The Feature List

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 terrain type.
6. The framework keeps up with whose turn it is an coordinates basic movement

But now the question is which of these are the most important?

The Three Qs of Architecture

- We are trying to identify the important functionality of our system
  - the features that are **architecturally significant**
- To identify important pieces of functionality, **ask 3 questions of each feature**
  - Is the feature part of the **essence** of the system?
    - The essence of a system is what that system is at its most basic level
    - “If I don’t implement this feature, would the system meet its goals?”
  - What does the feature **mean**?
    - Not having a clear idea might indicate that the feature can take a lot of time to get right; best to start on it early
  - How will I implement the feature?
    - Focus on features that **seem really hard to implement** so they don’t bog you down later in the development cycle

The Essence of Gary's Game Framework

- After applying the 3 Qs of architecture, the book focuses on these features
  1. The board for the game (the essence)
  2. Game-Specific Units (the essence and what does this mean?)
  3. Framework coordinates basic movement (Can we implement this?)
- Having identified these features, we can now examine them in depth
  - and this will serve to finally get the project moving… after these features get fleshed out, it will be easier to move on to the remaining features/ modules and flesh them out as well

Why are these features important? Risk

- The importance of the three Qs of architecture is that these questions help you to identify the major risks associated with your development process
  - The reason that the three features on the previous slide are architecturally significant is that they all introduce **RISK** to your project (i.e. the risk that the project will fail due to this particular feature/requirement)
    - If we don't know what some feature means, we have a risk of not meeting schedules and deadlines as we perform a lot of work to discover its meaning
    - If a feature seems hard to implement, there is a risk we won't figure it out or it will take a really long time to accomplish
    - If the core features are not in place, there's a risk that the customer won't like the finished product
- The key task during this phase of transitioning from large, complex problems to smaller, more manageable, problems is **reducing risk**

Feature 1: The board

- We want to consider the design of the board
  - an essential part of the framework
- Since we don’t have a lot of detail from the user, we can’t create a use case
  - but we may have enough information to develop a single scenario
    - recall that a single use case will have one or more paths (aka scenarios)
- A scenario for the board is developed
  - create the board
  - move player 1’s and player 2’s units such that a battle takes place
    - take into consideration the terrain that they encounter
  - perform the battle
  - remove units that lost the battle from the board

From Scenario to Code

```
1  public class Board {
2
3      private int width, height;
4      private List tiles;
5
6      public Board(int width, int height) {
7          this.width = width;
8          this.height = height;
9          initialize();
10     }
11
12     private void initialize() {
13         tiles = new ArrayList(width);
14         for (int i=0; i<width; i++) {
15             tiles.add(i, new ArrayList(height));
16             for (int j=0; j<height; j++) {
17                 ((ArrayList)tiles.get(i)).add(j, new Tile());
18             }
19         }
20     }
21
22     public Tile getTile(int x, int y) {
23         return (Tile)((ArrayList)tiles.get(x-1)).get(y-1);
24     }
25
26     public void addUnit(Unit unit, int x, int y) {
27         getTile(x, y).addUnit(unit);
28     }
29
30     public void removeUnit(Unit unit, int x, int y) {
31         getTile(x, y).removeUnit(unit);
32     }
33
34     ...
35 }
36
```

Board is placed in a Java package called board (not shown)

The creation of the board is moved out of the constructor and into its own method

Place a Tile() at each coordinate and offer a means for retrieving tiles

Delegate the handling of Units to the Tile class (not all Unit methods shown)

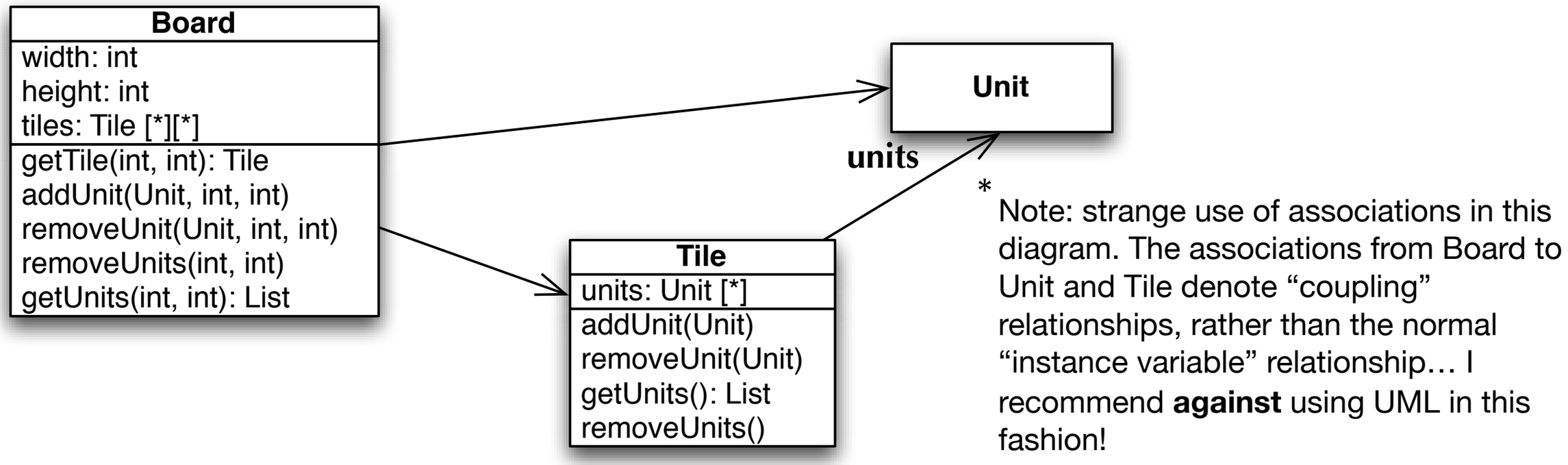
Create skeleton classes for Tile and Unit… they are outside our focus for now

Focus on one feature at a time!

Don’t get distracted with features that won’t reduce the risks of your project

Reflection: Rapid Prototyping

- Writing code for the board class at this stage of the game is an example of rapid prototyping
  - It helps answer the question
    - do I really understand what the board has to do?
  - We could have also chosen to just create a class diagram and outline Board's attributes and methods and associations, like this:



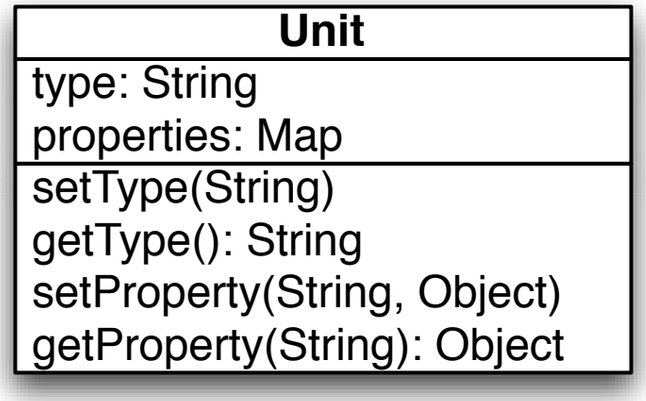
Feature 2: Game Specific Units

- We focus on the “game specific units” feature next for several reasons
  - Its a key feature but currently way underdeveloped (see previous diagram)
  - It is related to the Board concept and since architecture includes specifying the relationships between major system components, we need to define Units further such that we can begin to consider the relationship between Board and Unit
- Game Specific Units
  - Different game designers have different ideas about Units
    - some want attack, defense, and experience properties
    - some want lots of weapons
    - some want units to have names and track relationships with other units
    - all want different types of units: land, air, and space

Commonality Analysis

- For framework design, we want to identify the shared properties of all the different requests received from the game designers
  - Shared aspects go in the framework
  - Variable aspects go outside the framework but should still be accessible to the framework via inheritance, polymorphism, interfaces, etc.
- The common need in this instance is a bit abstract but can be stated succinctly
  - They need a Unit class with a variable number of atts of different types
    - Sound familiar?
  - This is exactly the same requirement that our InstrumentSpec class had
    - When the number and type of properties is the “thing that varies”, make the properties dynamic via a collection class
  - We want to avoid, therefore, an approach in which Unit is sub-classed

Solution: Just like Instrument Spec



Note: no need for code this time. The implementation of this class is obvious!

Provides Scalability

Number of Unit Types	Separate Subclasses	Dynamic Properties
3	4	1
25	26	1
100	101	1

To Code or Not to Code

- The book brings up an excellent point on page 364
  - They have a discussion about why they wrote code for the Board class but skipped writing code for the Unit class, noting that **reducing risk is key**
    - Do some initial design work to verify that a feature is **essential**, **well understood**, and **feasible**
      - If you need code, then write it but if you do not, then avoid it!
- They then ask the question “So, there’s not a lot of code involved in OO A&D, is there?”. Their response is excellent.
  - OO A&D is **ALL** about code. Its about getting your design right, having it be clean, well structured, flexible and extensible… because if you get that right then the code will be EASY to write and maintain
  - Sometimes the **best way** to **write great code** is to **hold off** on writing code **as long as you can**!

Feature 3: Coordinating Movement

- They repeat the process used for feature 2 for feature 3
  - Ask the Customer
  - Perform Commonality Analysis
  - Develop a Design for the feature (gain confidence you can implement it)
- After doing these two steps, they acquire this information:

What's common?	What's variable?
Check if move is legal	**The algorithm to check the legality of a move is different for every game**
Unit's attributes determine how far it can move	**The number and specific properties used for this are different for every game**
Other factors (such as terrain) will affect movement	**The other factors that affect movement are different for every game**

What do they decide? They Punt!

- In the book, they decide to punt on this feature and declare it out-of-scope
  - Lame! (In my not so humble opinion…)
- They say: “When you find more things that are different about a feature than things that are the same, there may not be a good generic solution”
- But I think they missed out on an opportunity to further define the Board and Unit classes (note: the book specifically discounts my approach)
  - In particular, if they had made Unit an abstract base class, they could have game designers add subclasses that represent movement rules for different types of Units (losing some of the previous scalability)
    - The abstract interface: canMove(Tile), determinePath(), etc.
    - The Board class could employ a Strategy pattern to allow Units to retrieve the other factors that might affect movement, etc.
- The framework’s complexity goes up, but in exchange it does more for you!

Classic Programmer Reaction

- “Great. So we’ve got a little sheet of paper with some check marks, a few classes that we know aren’t finished, and lots of UML diagrams. And I’m supposed to believe **this** is how you write great software.”
- Absolutely!
  - Remember this is just the start of the **first** iteration of **many**
  - When you start a project, you don’t have a lot of information and you can waste a lot of time if you don’t start by acquiring more detail in a systematic fashion
    - **Domain Analysis**, **The Three Qs of Architecture**, and **Commonality Analysis** can all help you to systematically whittle a large problem down to smaller more manageable chunks
  - The key point is to **reduce risk with each step you take**, so you can get to a point where you have a handle on the “big picture” and can start working on its constituent parts

Wrapping Up

- Architecture helps you turn all your diagrams, plans, and feature lists into a well-ordered application
- Focus on features that are the essence of your system, that have unclear meanings, or that don't seem feasible
- If you don't need all of the detail of a use case, just write simple scenarios to help you identify the characteristics of a particular feature
    - These simple scenarios can be turned into the "main path" of a use case at a later stage in the software life cycle
- Be open to getting lots of feedback from your customer… if something is unclear, go back to them and get additional feedback so that you can move forward with your design

Coming Up Next

- Lecture 12: Originality is Overrated
    - Read Chapter 8 of the OO A&D book
- Lecture 13: Testing And Iterating
    - Read Chapter 9 of the OO A&D book