1

Lecture Information

- Name: Võ Văn Hải
- Email: VoVanHaiQN@gmail.com
- Scopus: 57211989550
- Orcid: 0000-0002-5427-1960
- Scholar: MvT6c8YAAAAJ
- Blog: https://vovanhai.wordpress.com/sa/

2

2

3

Objectives

1. Introduction
2. Architecture Thinking
3. Modularity
4. Architecture Characteristics
5. Identifying Architecture Characteristics
6. Measuring and Governing Architecture Characteristics
7. Scope of Architecture Characteristics
8. Component-Based Thinking

4

4

Introduction

5

Introduction

"In most successful software projects, the expert developers working on that project have a shared understanding of the system design. This shared understanding is called architecture."

— Martin Fowler

"Architecture is the ***fundamental organization*** of a system, embodied in its ***components***, their ***relationships*** to ***each other*** and the ***environment***, and the principles governing its ***design and evolution***."

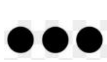
ANSI/IEEE Std 1471-2000

"The software architecture of a program or computing system is the ***structure or structures*** of the system, which comprise ***software elements***, the ***externally visible properties*** of those elements, and the ***relationships*** among them."

SEI- Software Engineering Institute

"[Software architecture goes] beyond the algorithms and data structures of the computation; designing and specifying the ***overall system structure*** emerges as a new kind of problem. ***Structural issues*** include gross organization and global control structure; ***protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance;*** and ***selection among design alternatives***."

Garlan and Shaw



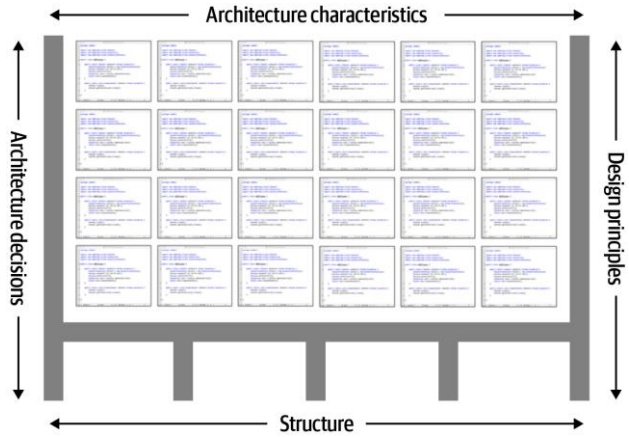
6

6

SA - Definition

Four dimensions that define software architecture

- Software architecture consists of the structure of the system, combined with architecture characteristics ("-ilities") the system must support, architecture decisions, and finally design principles.



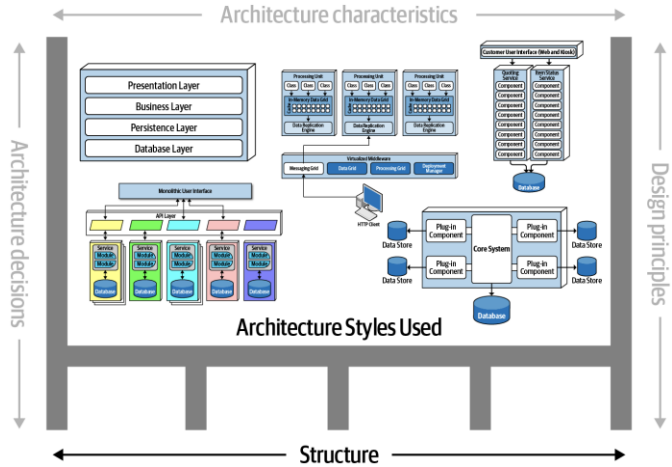
7

7

SA - Definition

Structure of the system

- Structure refers to the type of architecture styles used in the system such as microservices, layered, or microkernel



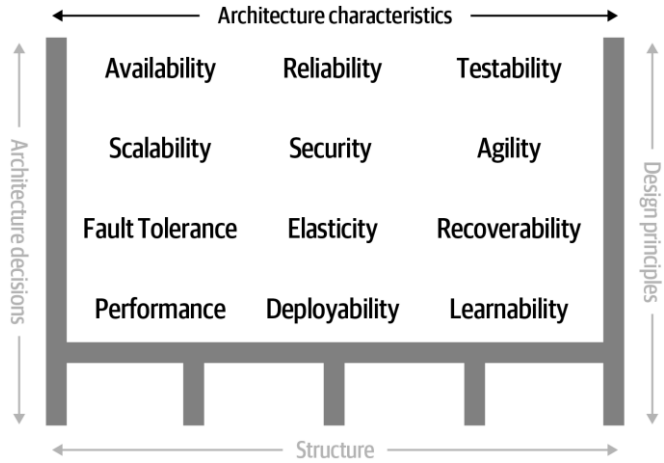
8

8

SA - Definition

Architecture characteristics

- The architecture characteristics define the success criteria of a system, which is generally orthogonal to the functionality of the system.



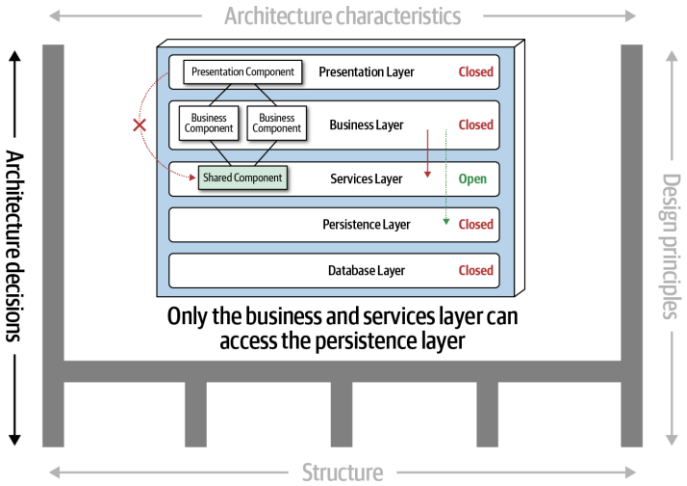
9

9

SA - Definition

Architecture decisions

- Architecture decisions define the rules for how a system should be constructed



For example, an architect might make an architecture decision that only the business and services layers within a layered architecture can access the database

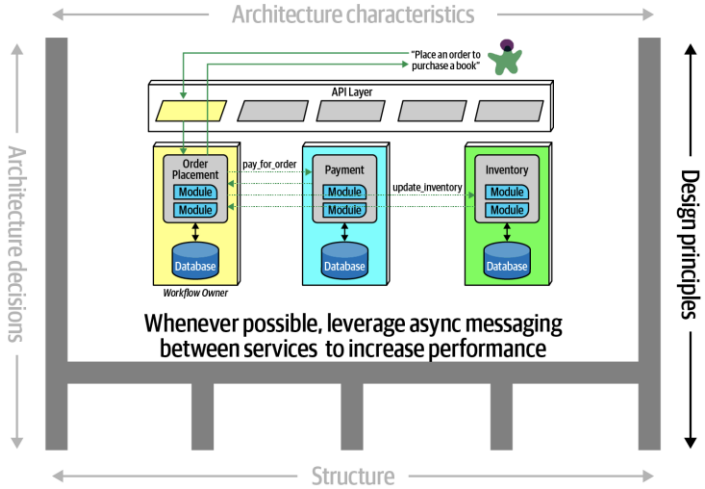
10

10

SA - Definition

Design principles

- Guidelines for constructing systems



11

11

SA - Definition (cont.)

Architecture Decision vs. Design Principle

- A design principle differs from an architecture decision in that a design principle is a guideline rather than a hard-and-fast rule.
- If a particular architecture decision cannot be implemented in one part of the system due to some condition or other constraint, that decision (or rule) can be broken through something called a variance. Most organizations have variance models that are used by an architecture review board (ARB) or chief architect.
- Example
  - An architecture decision (rule) could never cover every condition and option for communication between services, so a design principle can be used to provide guidance for the preferred method (in this case, asynchronous messaging) to allow the developer tochoose a more appropriate communication protocol (such as REST or gRPC) given a specific circumstance.

12

12

SA - Expectations

Core expectations of a software architect

1. Make architecture decisions :
   - Define the architecture decisions and design principles used to guide technology decisions within the team, the department, or across the enterprise
   - Instead of saying (use ReactJs) : an architect should instead instruct development teams to use a reactive-based framework for frontend web development
2. Continually analyze the architecture : continually analyze the architecture and current technology environment and then recommend solutions for improvement
3. Keep current with latest trends :
   - Keep current with the latest technology and industry trends
   - Understanding and following key trends helps the architect prepare for the future and make the correct decision
4. Ensure compliance with decisions : with architecture decisions and design principles
5. Diverse exposure and experience : have exposure to multiple and diverse technologies, frameworks, platforms, and environments
6. Have business domain knowledge : have a certain level of business domain expertise
7. Possess interpersonal skills : possess exceptional interpersonal skills, including teamwork, facilitation, and leadership
8. Understand and navigate politics : understand the political climate of the enterprise and be able to navigate the politics.

13

13

SA -

Laws of Software Architecture

First Law

- Everything in software architecture is a trade-off.
  - **Corollary 1:** If an architect thinks they have discovered something that isn't a trade-off, more likely they just haven't identified the trade-off yet.

Second Law

- WHY is more important than HOW.

14

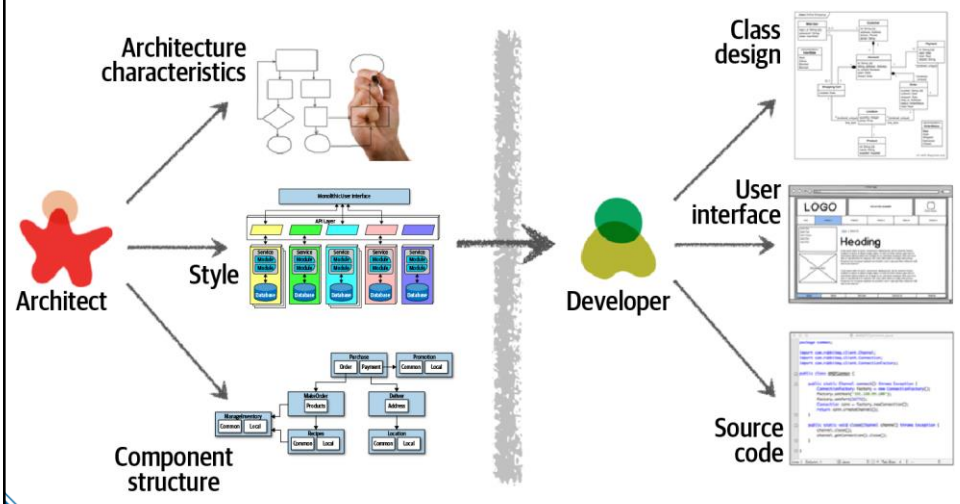
14

Architectural Thinking

15

SA - Architecture vs Design

Traditional View : Architecture vs Design



16

16

SA - Architecture vs Design

Traditional View : Architecture vs Design (cont.)

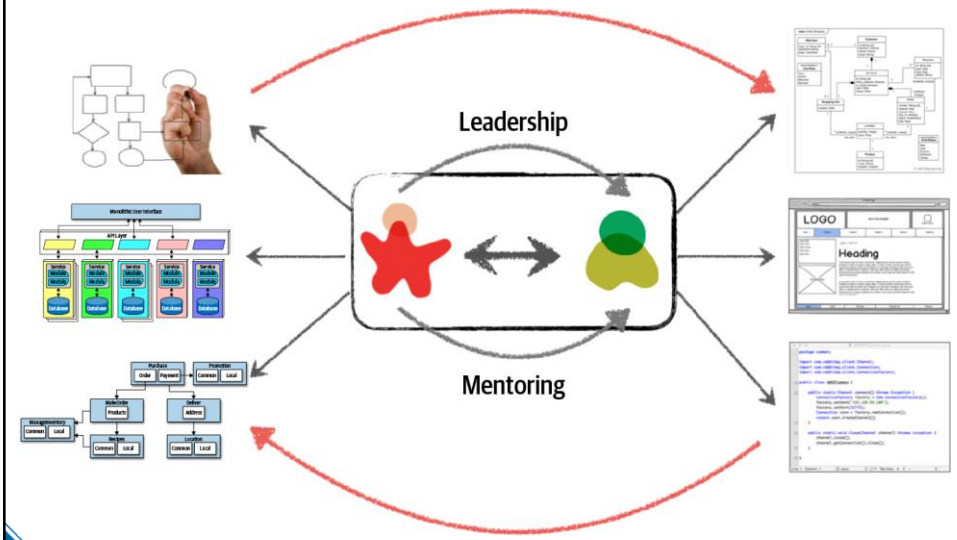
- It is the unidirectional arrow passing though the virtual and physical barriers separating the architect from the developer that causes all of the problems associated with architecture
- Decisions an architect makes sometimes never make it to the development teams, and decisions development teams make that change the architecture rarely get back to the architect
- The architect is disconnected from the development teams, and as such the architecture rarely provides what it was originally set out to do.

17

17

SA - Architecture vs Design

*Making architecture work through **collaboration***



18

18

SA - Architecture vs Design

*Making architecture work through **collaboration** (cont.)*

- To make architecture work, both the physical and virtual barriers that exist between architects and developers must be broken down
- Forming a strong bidirectional relationship between architects and development teams
- The architect and developer must be on the same virtual team to make this work
- Not only does this model facilitate strong bidirectional communication between architecture and development, but it also allows the architect to provide mentoring and coaching to developers on the team.

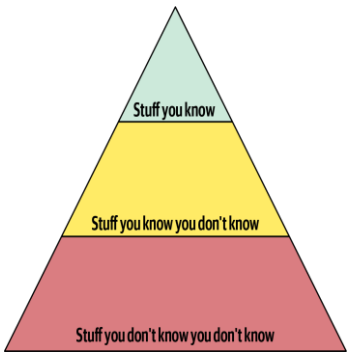
19

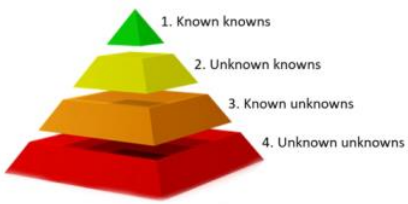
19

SA - Levels of knowledge

Knowledge triangle

- Stuff you know - includes the technologies, frameworks, languages, and tools a technologist uses on a daily basis to perform their job, such as knowing Java as a Java programmer
- Stuff you know you don't know - includes those things a technologist knows a little about or has heard of but has little or no expertise in
- Stuff you don't know you don't know - includes the entire host of technologies, tools, frameworks, and languages that would be the perfect solution to a problem a technologist is trying to solve, but the technologist doesn't even know those things exist





20

20

SA - Technical Breadth

Technical Depth and Technical Breadth

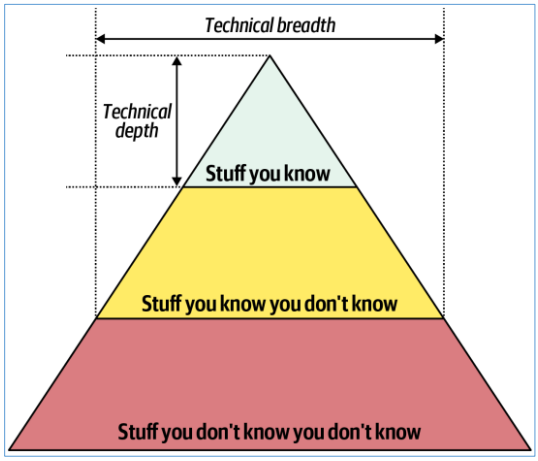


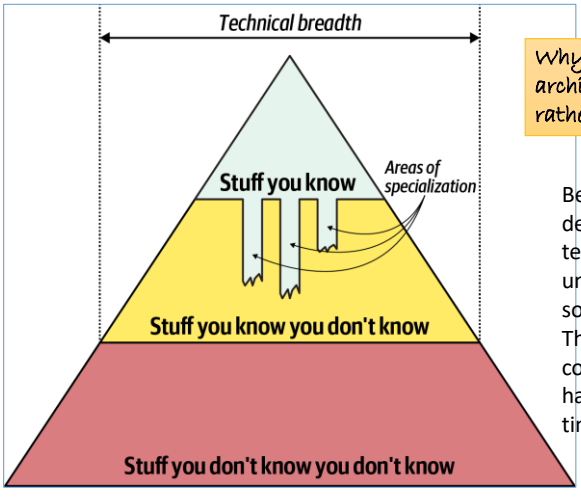
Figure 2.5. What someone knows is technical depth, and how much someone knows is technical breadth

21

21

SA - Technical Breadth

Developers will focus on Technical Depth and Architect on Technical Breadth



Why is it more important for an architect to focus on technical breadth rather than technical depth?

Because architects must make decisions that match capabilities to technical constraints, a broad understanding of a wide variety of solutions is valuable.
Thus, for an architect, the wise course of action is to sacrifice some hard-won expertise and use that time to broaden their portfolio.

Figure 2-6. Enhanced breadth and shrinking depth for the architect role

22

22

SA - Analyzing Trade-Offs

- *Everything* in architecture is a trade-off, which is why the famous answer to every architecture question in the universe is "it depends."
  - It depends on:
    - the deployment environment,
    - business drivers,
    - company culture, budgets,
    - timeframes,
    - developer skill set,
    - and dozens of other factors.
  - Everyone's environment, situation, and problem is different, hence why architecture is so hard.

Architecture is the stuff you can't Google. (Mark)

There are no right or wrong answers in architecture—only trade-offs. (Neal)

23

23

SA - Analyzing Trade-Offs

Example

The Bid Producer service generates a bid from the bidder and then sends that bid amount to the Bid Capture, Bid Tracking, and Bid Analytics services.

Can you Google the answer?

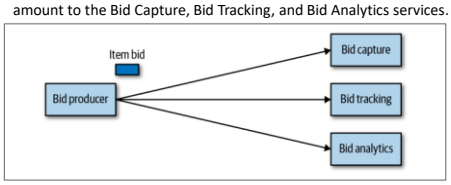


Figure 2-7. Auction system example of a trade-off—queues or topics?

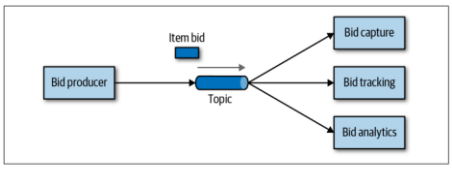


Figure 2-8. Use of a topic for communication between services

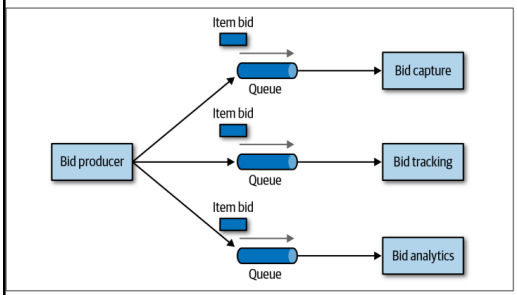


Figure 2-9. Use of queues for communication between services

Programmers know the benefits of everything and the trade-offs of nothing. Architects need to understand both.

Table 2-1. Trade-offs for topics

Topic advantages	Topic disadvantages
Architectural extensibility	Data access and data security concerns
Service decoupling	No heterogeneous contracts
	Monitoring and programmatic scalability

24

24

SA - Understanding Business Drivers

What are business drivers?

- Business drivers are the major inputs and actions that drive a company's operational and financial success. For example, this may include salespeople, the number of stores, online traffic, the number and price of products sold and production units.
  - Businesses may use these key drivers to determine how to improve their company and often aim to maximize any within their control.
- Thinking like an architect is understanding the business drivers required for the system's success and translating those requirements into architectural characteristics (such as scalability, performance, and availability).
- This is a challenging task that requires the architect to have some level of business domain knowledge and healthy, collaborative relationships with key business stakeholders.
  - Discuss later

25

25

SA - Balancing Architecture and Coding

Balancing Architecture and Hands-On Coding

- Striving for a balance between hands-on coding and being a software architect is avoiding the bottleneck trap.
  - The bottleneck trap occurs when the architect has taken ownership of code within the critical path of a project and becomes a bottleneck to the team.
  - To avoid the bottleneck trap as an effective software architect is to delegate the critical path and framework code to others on the development team and then focus on coding a piece of business functionality (a service or a screen) one to three iterations down the road.
- To do frequent proof-of-concepts (POCs)
- Tackle some of the technical debt stories or architecture stories, freeing the development team up to work on the critical functional user stories.
- Working on bug fixes within an iteration is another way of maintaining hands-on coding while helping the development team as well.
- Leveraging automation by creating simple command-line tools and analyzers to help the development team.
- To do frequent code reviews

26

26

Chapter 3
Modularity

27

SA - Modularity

Definition

> "95% of the words [about software architecture] are spent extolling the benefits of "modularity" and that little, if anything, is said about how to achieve it."
>
> —Glenford J. Myers (1978)

- The dictionary defines module as "*each of a set of standardized parts or independent units that can be used to construct a more complex structure.*"
- We use modularity to describe a logical grouping of related code, which could be a group of classes in an object-oriented language or functions in a structured or functional language.
  - Most languages provide mechanisms for modularity (package in Java, namespace in .NET, and so on).
  - Developers typically use modules as a way to group related code together.
    - For example, the $com.mycompany.customer$ package in Java should contain things related to customers.
- Architects must be aware of how developers package things because it has important implications in architecture.

28

28

SA - Measuring Modularity

Cohesion

- Cohesion refers to what extent the parts of a module should be contained within the same module.
  - In other words, it is a measure of how related the parts are to one another.
  - Ideally, a cohesive module is one where all the parts should be packaged together, because breaking them into smaller pieces would require coupling the parts together via calls between modules to achieve useful results.
- Cohesion metrics measure how well the methods of a class are related to each other. A cohesive class performs one function while a non-cohesive class performs two or more unrelated functions. A non-cohesive class may need to be restructured into two or more smaller classes.
  - High cohesion is desirable since it promotes encapsulation. As a drawback, a highly cohesive class has high coupling between the methods of the class, which in turn indicates high testing effort for that class.
  - Low cohesion indicates inappropriate design and high complexity. It has also been found to indicate a high likelihood of errors. The class should probably be split into two or more smaller classes.

> Attempting to divide a cohesive module would only result in increased coupling and decreased readability.
>
> —Larry Constantine

29

29

SA - Measuring Modularity

Cohesion listing

- Functional cohesion
  - Every part of the module is related to the other, and the module contains everything essential to function.
- Sequential cohesion
  - Two modules interact, where one outputs data that becomes the input for the other.
- Communicational cohesion
  - Two modules form a communication chain, where each operates on information and/or contributes to some output. For example, add a record to the database and generate an email based on that information.
- Procedural cohesion
  - Two modules must execute code in a particular order.

30

30

SA - Measuring Modularity

Cohesion listing (continue)

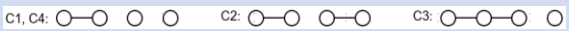
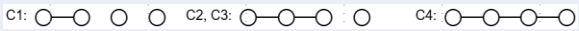
- Temporal cohesion
  - Modules are related based on timing dependencies. For example, many systems have a list of seemingly unrelated things that must be initialized at system startup; these different tasks are temporally cohesive.
- Logical cohesion
  - The data within modules is related logically but not functionally. For example, consider a module that converts information from text, serialized objects, or streams. Operations are related, but the functions are quite different. A common example of this type of cohesion exists in virtually every Java project in the form of the StringUtils package: a group of static methods that operate on String but are otherwise unrelated.
- Coincidental cohesion
  - Elements in a module are not related other than being in the same source file; this represents the most negative form of cohesion.

31

31

SA - Measuring Modularity

The Lack of Cohesion in Methods (LCOM)

Class Cohesion Metric	Definition/Formula
Lack of Cohesion of Methods (LCOM1) (Chidamber & Kemerer, 1991)	LCOM1 = Number of pairs of methods that do not share attributes
LCOM2 (Chidamber & Kemerer, 1991)	P = Number of pairs of methods that do not share attributes Q = Number of pairs of methods that share attributes $LCOM2 = \begin{cases} \lvert P\rvert - \lvert Q\rvert, if\ \lvert P\rvert > \lvert Q\rvert \\ 0, \quad otherwise \end{cases}$
LCOM3 (Li & Henry, 1993)	LCOM3 = Number of disjoint components in the graph that represents each method as a node and the sharing of at least one attribute as an edge 
LCOM4 (Hitz & Montazeri, 1995)	Similar to LCOM3 and additional edges are used to represent method invocations 
LCOM5 (Henderson-Sellers, 1996)	LCOM5 = (a – kℓ) / (ℓ – kℓ), where ℓ is the number of attributes, k is the number of methods, and a is the summation of the number of distinct attributes accessed by each method in a class.

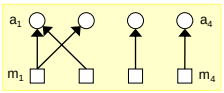
32

32

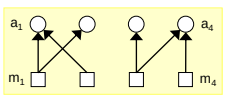
OSA - Measuring Modularity

The Lack of Cohesion in Methods (LCOM) - example

class C1

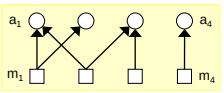


class C2

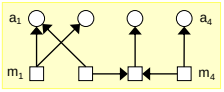


$$\#\text{ Method Pairs} = NP = \binom{M}{2} = \frac{M!}{2! \cdot (M-2)!}$$

class C3



class C4



$$NP(Ci) = \binom{4}{2} = 6$$

LCOM1:
LCOM1(C1) = P = NP – Q = 6 – 1 = 5
LCOM1(C2) = 6 – 2 = 4
LCOM1(C3) = 6 – 2 = 4
LCOM1(C4) = 6 – 1 = 5

LCOM2:
LCOM2(C1) = P – Q = 5 – 1 = 4
LCOM2(C2) = 4 – 2 = 2
LCOM2(C3) = 4 – 2 = 2
LCOM2(C4) = 5 – 1 = 4

LCOM3:
LCOM3(C1) = 3
LCOM3(C2) = 2
LCOM3(C3) = 2
LCOM3(C4) = 3

LCOM4:
LCOM4(C1) = 3
LCOM4(C2) = 2
LCOM4(C3) = 2
LCOM4(C4) = 1

LCOM5:
LCOM5(C1) = (5 – 4×4) / (4 – 4×4) = 11 / 12
LCOM5(C2) = 10 / 12 = 5 / 6
LCOM5(C3) = 5 / 6
LCOM5(C4) = 11 / 12

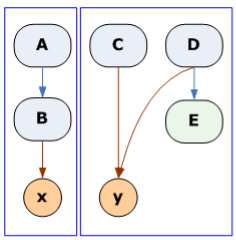
33

33

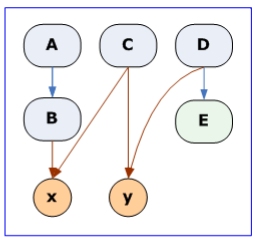
OSA - Measuring Modularity

The Lack of Cohesion in Methods (LCOM) - example

- Which methods are related? Methods A and B are related if:
  - they both access the same class-level variable, or
  - A calls B or vice versa.
- After determining the related methods, we draw a graph linking the related methods to each other. LCOM4 equals the number of connected groups of methods.
  - LCOM=1 indicates a cohesive class, which is the "good" class.
  - LCOM>=2 indicates a problem. The class should be split into so many smaller classes.
  - LCOM=0 happens when there are no methods in a class. This is also a "bad" class.



LCOM4 = 2



LCOM4 = 1

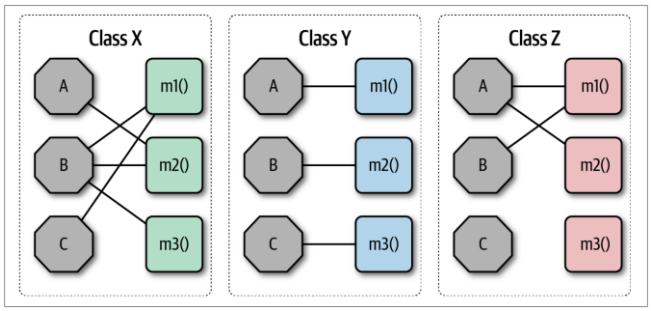
(1) shows a class consisting of methods A through E and variables x and y. A calls B and B accesses x. Both C and D access y. D calls E, but E doesn't access any variables. This class consists of 2 unrelated components (LCOM4=2) ➔You could split it as {A, B, x} and {C, D, E, y}.
In (2), we made C access x to increase cohesion. Now the class consists of a single component (LCOM4=1). It is a cohesive class.

34

34

SA - Measuring Modularity

The Lack of Cohesion in Methods - example



fields appear as single letters and methods appear as blocks.

Figure 3-1. Illustration of the LCOM metric, where fields are octagons and methods are squares

In Class X, the LCOM score is low, indicating good structural cohesion.

Class Y, however, lacks cohesion; each of the field/method pairs in Class Y could appear in its own class without affecting behavior.

Class Z shows mixed cohesion, where developers could refactor the last field/method combination into its own class.

36

36

SA - Measuring Modularity

Cohesion Tools

- JArchitect
- TAACO
- CK https://github.com/mauricioaniche/ck
- Metric Reloaded (IntelliJ IDEA plugin)
- Java Methods Reference Diagram (IntelliJ IDEA plugin)
- …

37

37

SA - Measuring Modularity

Coupling

- Afferent coupling measures the number of incoming connections to a code artifact (component, class, function, and so on).
- Efferent coupling measures the outgoing connections to other code artifacts.
- For virtually every platform tools exist that allow architects to analyze the coupling characteristics of code in order to assist in restructuring, migrating, or understanding a code base.

Why Such Similar Names for Coupling Metrics?

Why are two critical metrics in the architecture world that represent opposite concepts named virtually the same thing, differing in only the vowels that sound the most alike? These terms originate from Yourdon and Constantine's *Structured Design*. Borrowing concepts from mathematics, they coined the now-common afferent and efferent coupling terms, which should have been called incoming and outgoing coupling. However, because the original authors leaned toward mathematical symmetry rather than clarity, developers came up with several mnemonics to help out: *a* appears before *e* in the English alphabet, corresponding to *incoming* being before *outgoing*, or the observation that the letter *e* in efferent matches the initial letter in *exit*, corresponding to outgoing connections.

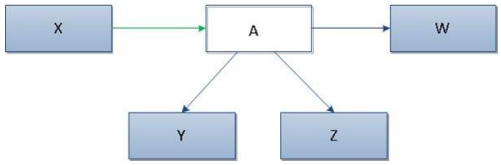
38

38

SA - Measuring Modularity

Coupling – efferent (C^e) & afferent (C^a)

- Efferent coupling (C^e): outgoing dependency
- Afferent coupling (C^a): incoming dependency
- When a C^e > 20 indicates instability of a component, changes in any external component may cause the need for changes within this component. The C^e value should be between 0 and 20, and higher values cause problems with development and maintenance.
- The higher the value of C^a means the stability of the component will be higher. This shows that there are many components depending on this component. The recommended value of C^a is around 0 to 500.
- Example:



$$C^e = 3$$
$$C^a = 1$$

39

39

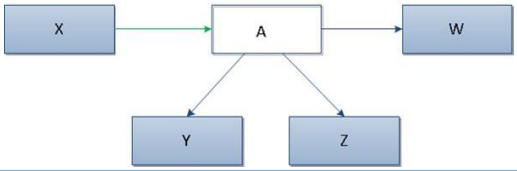
SA - Measuring Modularity

Coupling - Instability

- Instability is defined as the ratio of efferent coupling to the sum of both efferent and afferent coupling.

$$I = \frac{C^e}{C^e + C^a}$$

  - C^e represents *efferent* (or outgoing) coupling, and C^a represents *afferent* (or incoming) coupling.
- The instability metric determines the volatility of a code base. A code base that exhibits high degrees of instability breaks more easily when changed because of high coupling.
  - For example, if a class calls to many other classes to delegate work, the calling class shows high susceptibility to breakage if one or more of the called methods change.



$$I = \frac{C^e}{C^e + C^a} = \frac{3}{3+1} = 0.75$$

40

40

SA - Measuring Modularity

Coupling – Instability (cont.)

- Components have many incoming dependencies and not many outcoming dependencies (I approaching 1), which are unstable because these components are highly susceptible to change;
- Components have many outcoming dependencies and not many incoming dependencies (I value is close to 0), so they will be less likely to be changed. However, changes from them will have a considerable impact.
- Preferred values for the instability (I) should be in the range from 0 to 0.3 or 0.7 to 1 (Components should be very stable or unstable). The medium-stability components should be avoided.

41

41

SA - Measuring Modularity

Coupling - Abstractness

- Abstractness is the ratio of abstract artifacts (abstract classes, interfaces, and so on) to concrete artifacts (implementation).
  - It represents a measure of abstractness versus implementation.
  - For example, consider a code base with no abstractions, just a huge, single function of code (as in a single main() method). The flip side is a code base with too many abstractions, making it difficult for developers to understand how things wire together.

$$A = \frac{\sum m^a}{\sum m^c}$$

    - m^a represents abstract elements (interfaces or abstract classes) with the module,
    - m^c represents concrete elements (nonabstract classes).
  - Example: consider an application with 5,000 lines of code, all in one main() method. The abstractness numerator is 1, while the denominator is 5,000, yielding an abstractness of almost 0. Thus, this metric measures the ratio of abstractions in your code.
- Architects calculate abstractness by calculating the ratio of the sum of abstract artifacts to the sum of the concrete ones.

42

42

SA - Measuring Modularity

Coupling – Abstractness (cont.)

- The value of *A* ranges from 0 to 1.
  - A module with a high abstractness value (close to 1) indicates that it contains a significant number of abstract classes and interfaces, suggesting a more abstract and less concrete implementation.
  - Conversely, a low abstractness value (close to 0) indicates a more concrete implementation with fewer abstract classes and interfaces.
- The goal of abstractness is to strike a balance based on the design principles.
  - High abstractness can lead to more flexibility and extensibility, allowing for easier modifications and extensions.
  - However, excessive abstractness without concrete implementations may lead to a lack of functionality.
  - The optimal level of abstractness depends on the specific design goals and requirements of the software system.

43

43

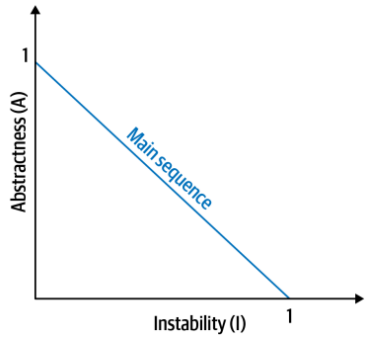
SA - Measuring Modularity

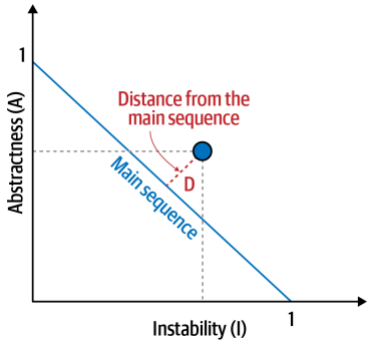
Coupling - Distance from the Main Sequence

- One of the few holistic metrics architects have for architectural structure is distance from the main sequence, a derived metric based on instability and abstractness.

$$D = |A + I - 1|$$

  - A = *abstractness* and I = *instability*.





44

44

SA - Measuring Modularity

Coupling - Distance from the Main Sequence

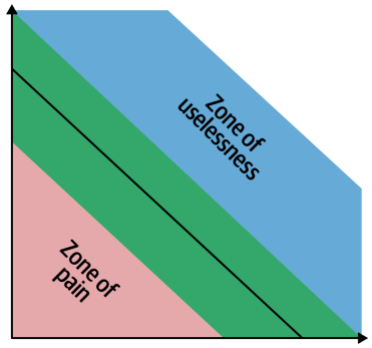


Figure 3-4. Zones of Uselessness and Pain

- The closer to the line, the better balanced the class.
  - Classes that fall too far into the upper-righthand corner enter into what architects call the zone of uselessness: code that is too abstract becomes difficult to use.
  - Conversely, code that falls into the lower-lefthand corner enter the zone of pain: code with too much implementation and not enough abstraction becomes brittle and hard to maintain.

45

45

SA - Measuring Modularity

Coupling - Distance from the Main Sequence

- Limitations of Metrics
  - While the industry has a few code-level metrics that provide valuable insight into code bases, our tools are extremely blunt compared to analysis tools from other engineering disciplines.
  - Even metrics derived directly from the structure of code require interpretation. For example, cyclomatic complexity measures complexity in code bases but cannot distinguish from essential complexity (because the underlying problem is complex) or accidental complexity (the code is more complex than it should be).
  - Virtually all code-level metrics require interpretation, but it is still useful to establish baselines for critical metrics such as cyclomatic complexity so that architects can assess which type they exhibit.

46

SA - Measuring Modularity

Connascence

- "Two components are connascent if a change in one would require the other to be modified in order to maintain the overall correctness of the system." - Meilir Page-Jones
- Types of connascence
  - Static: source-code-level coupling. It is a refinement of the afferent and efferent couplings defined by Structured Design. In other words, architects view the following types of static connascence as the degree to which something is coupled, either afferently or efferently
  - Dynamic: execution-time coupling. It analyzes calls at runtime

47

SA - Connascence

Static Connascence

- Connascence of Name (CoN)
  - Multiple components must agree on the name of an entity.
  - Names of methods represents the most common way that code bases are coupled and the most desirable, especially in light of modern refactoring tools that make system-wide name changes trivial.
- Connascence of Type (CoT)
  - Multiple components must agree on the type of an entity.
  - This type of connascence refers to the common facility in many statically typed languages to limit variables and parameters to specific types. However, this capability isn't purely a language feature—some dynamically typed languages offerselective typing, notably Clojure and Clojure Spec.
- Connascence of Meaning (CoM) or Connascence of Convention (CoC)
  - Multiple components must agree on the meaning of particular values. The most common obvious case for this type of connascence in code bases is hard-coded numbers rather than constants. For example, it is common in some languages to consider defining somewhere int TRUE = 1; int FALSE = 0. Imagine the problems if someone flips those values.

48

48

SA - Connascence

Static Connascence

- Connascence of Position (CoP)
  - Multiple components must agree on the order of values.
  - This is an issue with parameter values for method and function calls even in languages that feature static typing. For example, if a developer creates a method void updateSeat(String name, String seatLocation) and calls it with the values updateSeat("14D", "Ford, N"), the semantics aren't correct even if the types are.
- Connascence of Algorithm (CoA)
  - Multiple components must agree on a particular algorithm.
  - A common case for this type of connascence occurs when a developer defines a security hashing algorithm that must run on both the server and client and produce identical results to authenticate the user. Obviously, this represents a high form of coupling—if either algorithm changes any details, the handshake will no longer work.

49

49

SA - Connascence

Dynamic Connascence

- Connascence of Execution (CoE)
  - The order of execution of multiple components is important.
- Connascence of Timing (CoT)
  - The timing of the execution of multiple components is important.
  - The common case for this type of connascence is a race condition caused by two threads executing at the same time, affecting the outcome of the joint operation.
- Connascence of Values (CoV)
  - Occurs when several values relate on one another and must change together.
- Connascence of Identity (CoI)
  - Occurs when multiple components must reference the same entity

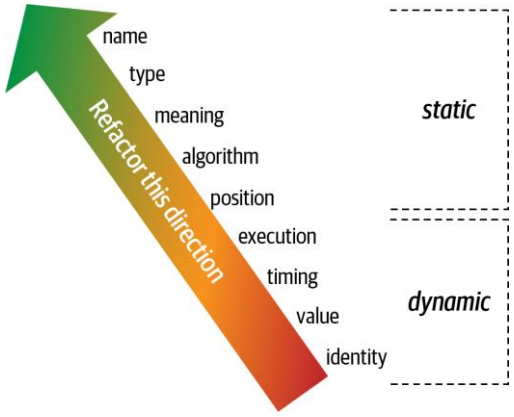
50

50

SA - Connascence

Connascence properties

- Strength
- Locality
- Degree



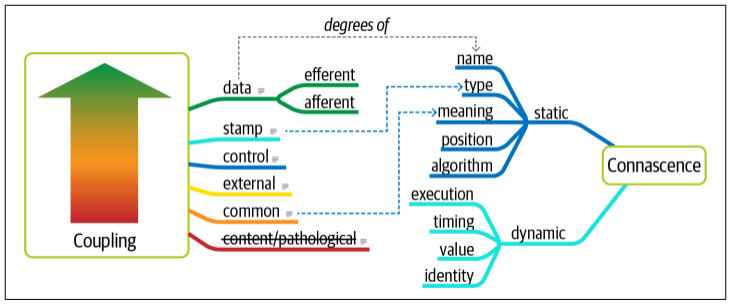
51

51

SA - Coupling & Connascence

Unifying Coupling and Connascence Metrics

- These two views overlap.
  - What Page-Jones identifies as static connascence represents degrees of either incoming or outgoing coupling. Structured programming only cares about in or out, whereas connascence cares about how things are coupled together



52

52

Chapter 4
Architecture Characteristics Defined

53

SA - Introduction

- Not just gathering the list of requirement of developing system, the architect must consider many other factors in designing a software solution (figure 4.1).
- Architects may collaborate on defining the domain or business requirements. Still, one key responsibility entails defining, discovering, and otherwise analyzing all the things the software must do that aren't directly related to the domain functionality: architectural characteristics.

Favor the term **Architecture Characteristics** instead of **nonfunctional requirements** (NFR)

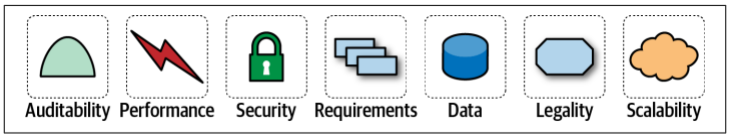


Figure 4-1. A software solution consists of both domain requirements and architectural characteristics

54

54

SA - Architecture Characteristics

Architecture characteristic criteria

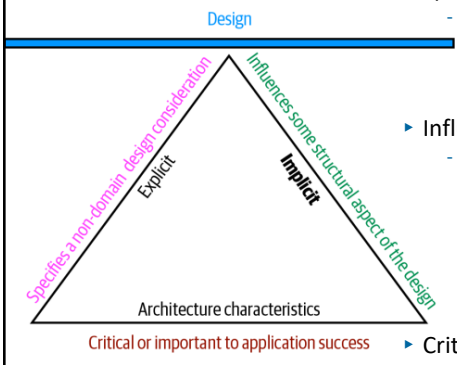
1. Specifies a non domain design consideration :
   - Requirements specify what the application should do
   - Architecture characteristics specify operational and design criteria for success
   - Concerning how to implement the requirements and why certain choices were made
   - Ex : Level of performance for the application
2. Influences some structural aspect of the design :
   - Does this architecture characteristic require special structural consideration to succeed
   - Ex : Security aspects
3. Is critical or important to application success :
   - Applications could support a huge number of architecture characteristics…but shouldn't

55

55

SA - Architecture Characteristics

Features of architecture characteristics



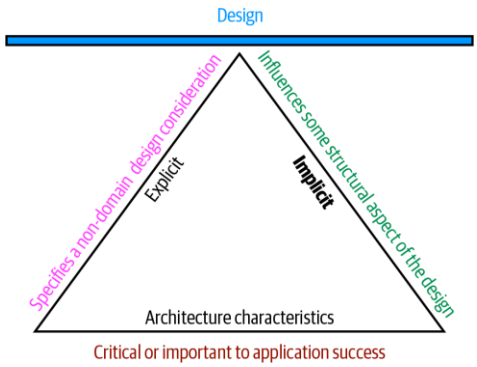
- Specifies a nondomain design consideration
  - When designing an application, the requirements specify what the application should do; architecture characteristics specify operational and design criteria for success, concerning how to implement the requirements and why certain choices were made.
- Influences some structural aspect of the design
  - The primary reason architects try to describe architecture characteristics on projects concerns design considerations: does this architecture characteristic require special structural consideration to succeed? For example, security is a concern in virtually every project, and all systems must take a baseline of precautions during design and coding. However, it rises to the level of architecture characteristics when the architect needs to design something special
- Critical or important to application success
  - Applications could support a huge number of architecture characteristics…but shouldn't. Support for each architecture characteristic adds complexity to the design. Thus, a critical job for architects lies in choosing the fewest architecture characteristics rather than the most possible.

56

56

SA - Architecture Characteristics

Features of architecture characteristics



- Implicit ones rarely appear in requirements, yet they're necessary for project success.
  - Examples : Availability, reliability, and security
  - Architects must use their knowledge of the problem domain to uncover these architecture characteristics during the analysis phase
- Explicit architecture characteristics appear in requirements documents

**The choice of a triangle is intentional: each definition element supports the others, which in turn helps the system's overall design. The triangle's fulcrum illustrates that these architectural characteristics often interact with one another, leading to the pervasive use of the term trade-off among architects.*

57

57

SA - Architectural Characteristics

Operational Architecture Characteristics

Term	Definition
Availability	How long the system will need to be available (if 24/7, steps need to be in place to allow the system to be up and running quickly in case of any failure).
Continuity	Disaster recovery capability.
Performance	Includes stress testing, peak analysis, analysis of the frequency of functions used, capacity required, and response times. Performance acceptance sometimes requires an exercise of its own, taking months to complete.
Recoverability	Business continuity requirements (e.g., in case of a disaster, how quickly is the system required to be online again?). This will affect the backup strategy and requirements for duplicated hardware.
Reliability/ safety	Assess if the system needs to be fail-safe, or if it is mission critical in a way that affects lives. If it fails, will it cost the company large sums of money?
Robustness	Ability to handle error and boundary conditions while running if the internet connection goes down or if there's a power outage or hardware failure.
Scalability	Ability for the system to perform and operate as the number of users or requests increases.

58

58

SA - Architectural Characteristics

Operational Architecture Characteristics

Term	Definition
Configurability	Ability for the end users to easily change aspects of the software's configuration (through usable interfaces).
Extensibility	How important it is to plug new pieces of functionality in.
Installability	Ease of system installation on all necessary platforms.
Leverageability/ reuse	Ability to leverage common components across multiple products.
Localization	Support for multiple languages on entry/query screens in data fields; on reports, multibyte character requirements and units of measure or currencies.
Maintainability	How easy it is to apply changes and enhance the system?
Portability	Does the system need to run on more than one platform? (For example, does the frontend need to run against Oracle as well as SAP DB?
Supportability	What level of technical support is needed by the application? What level of logging and other facilities are required to debug errors in the system?
Upgradeability	Ability to easily/quickly upgrade from a previous version of this application/solution to a newer version on servers and clients.

59

59

SA - Architectural Characteristics

Cross-Cutting Architecture Characteristics

Term	Definition
Privacy	Ability to hide transactions from internal company employees (encrypted transactions so even DBAs and network architects cannot see them).
Security	Does the data need to be encrypted in the database? Encrypted for network communication between internal systems? What type of authentication needs to be in place for remote user access?
Supportability	What level of technical support is needed by the application? What level of logging and other facilities are required to debug errors in the system?
Usability/ achievability	Level of training required for users to achieve their goals with the application/solution. Usability requirements need to be treated as seriously as any other architectural issue.

60

60

SA - Architectural Characteristics

ISO 's Architecture Characteristics

The quality mode ISO 25010 (iso25000.com)

- No complete list of standards exists. The International Organization for Standards (ISO) publishes a list organized by capabilities, overlapping many of the ones we've listed, but mainly establishing an incomplete category list.
- Read more: pages 81-82
- *** The many ambiguities in Software Architecture (page 83)

61

61

SA - Architectural Characteristics – Trade-offs

Trade-Offs and Least-Worst Architecture

- Applications can only support a few of the architecture characteristics for various reasons.
  - First, each supported characteristic requires design effort and perhaps structural support.
  - Second, the bigger problem lies with the fact that each architecture characteristic often has an impact on others.
  - For example, suppose an architect wants to improve security. In that case, it will almost certainly negatively impact performance.
- Thus, architects rarely encounter the situation where they can design a system and maximize every single architecture characteristic. More often, the decisions come down to trade-offs between several competing concerns.
- Too many architecture characteristics lead to generic solutions trying to solve every business problem, and those architectures rarely work because the design becomes unwieldy.

Never shoot for **the best** architecture, but rather the **least-worst** architecture.

62