Génie logiciel pour la conception d'un Système d'Information

CSC4521

Voie d'Approfondissement
Intégration et Déploiement de Systèmes d'Information
(VAP DSI)

Architecture Analysis

paul.gibson@telecom-sudparis.eu

http://jpaulgibson.synology.me/~jpaulgibson/TSP/Teaching/CSC4521/

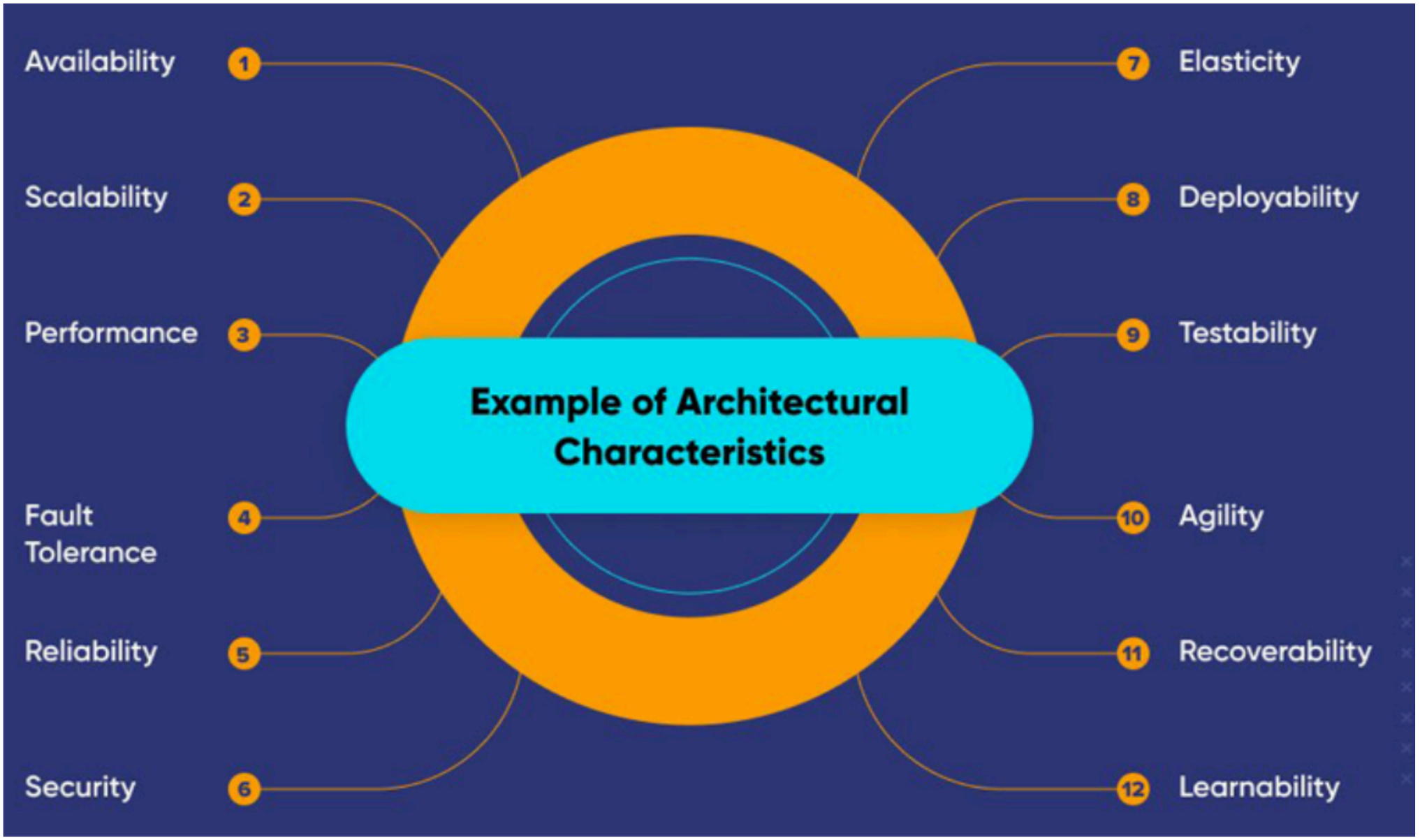
…/CSC4521/CSC4521-Architecture Analysis.pdf

Software architecture review: The state of practice

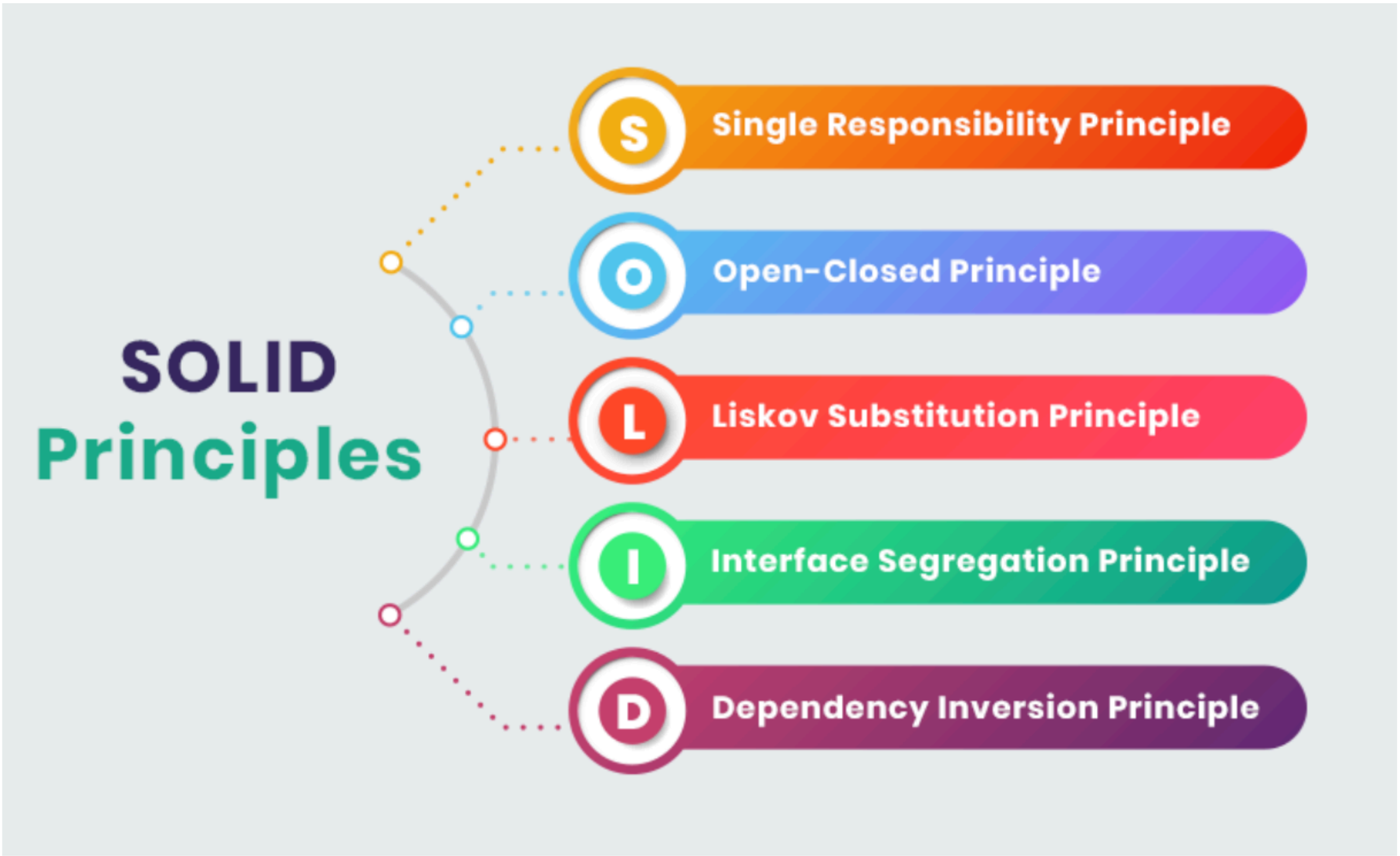
Table 1. Perceived benefits of architecture review.

Benefits/goals of conducting architecture review	Responses
a. Identifying potential risks in the proposed architecture	76 (88%)
b. Assessing quality attributes (for example, scalability, performance)	66 (77%)
c. Identifying opportunities for reuse of architectural artifacts and components	62 (72%)
d. Promoting good architecture design and evaluation practices	55 (64%)
e. Reducing project cost caused by undetected design problems	54 (63%)
f. Capturing the rationale for important design decisions	51 (59%)
g. Uncovering problems and conflicts in requirements	51 (59%)
h. Conforming to organization's quality assurance process	47 (55%)
i. Assisting stakeholders in negotiating conflicting requirements	37 (43%)
j. Partitioning architectural design responsibilities	34 (40%)
k. Identifying skills required to implement the proposed architecture	34 (40%)
l. Improving architecture documentation quality	34 (40%)
m. Facilitating clear articulation of nonfunctional requirements	27 (31%)
n. Opening new communication channels among stakeholders	27 (31%)

Babar, Muhammad Ali, and Ian Gorton. "Software architecture review: The state of practice." *Computer* 42.7 (2009): 26-32.



https://itechnolabs.ca/software-architecture-5-principles-you-should-know/



https://itechnolabs.ca/software-architecture-5-principles-you-should-know/

These are the underlying principles of **S**OLID:

- **S- Single Responsibility Principle:** Every module or class should only have responsibility for a single functionality. As outlined in the original principle, responsibility is a “reason to change,” and each class should have only one such reason.

These are the underlying principles of S**O**LID:

- **O- Open-Closed Principle:** Basically, the open-clo
  principle states that all software entities should be o
  to enhancements and extensions without having to
  modified or altered. The concept of polymorphism
  OOPS is similar to this.

These are the underlying principles of SO**L**ID:

- **L- Liskov Substitution Principle:** Defining each derived class as a replacement for its base class is stated in this principle. The following example is given to make this concept understandable. An object of type A may be substituted for an object of type B if A is a subclass of B. In other words, it means that the software program's objects can be replaced with examples of subtypes without interfering with the code.

These are the underlying principles of SOLID:

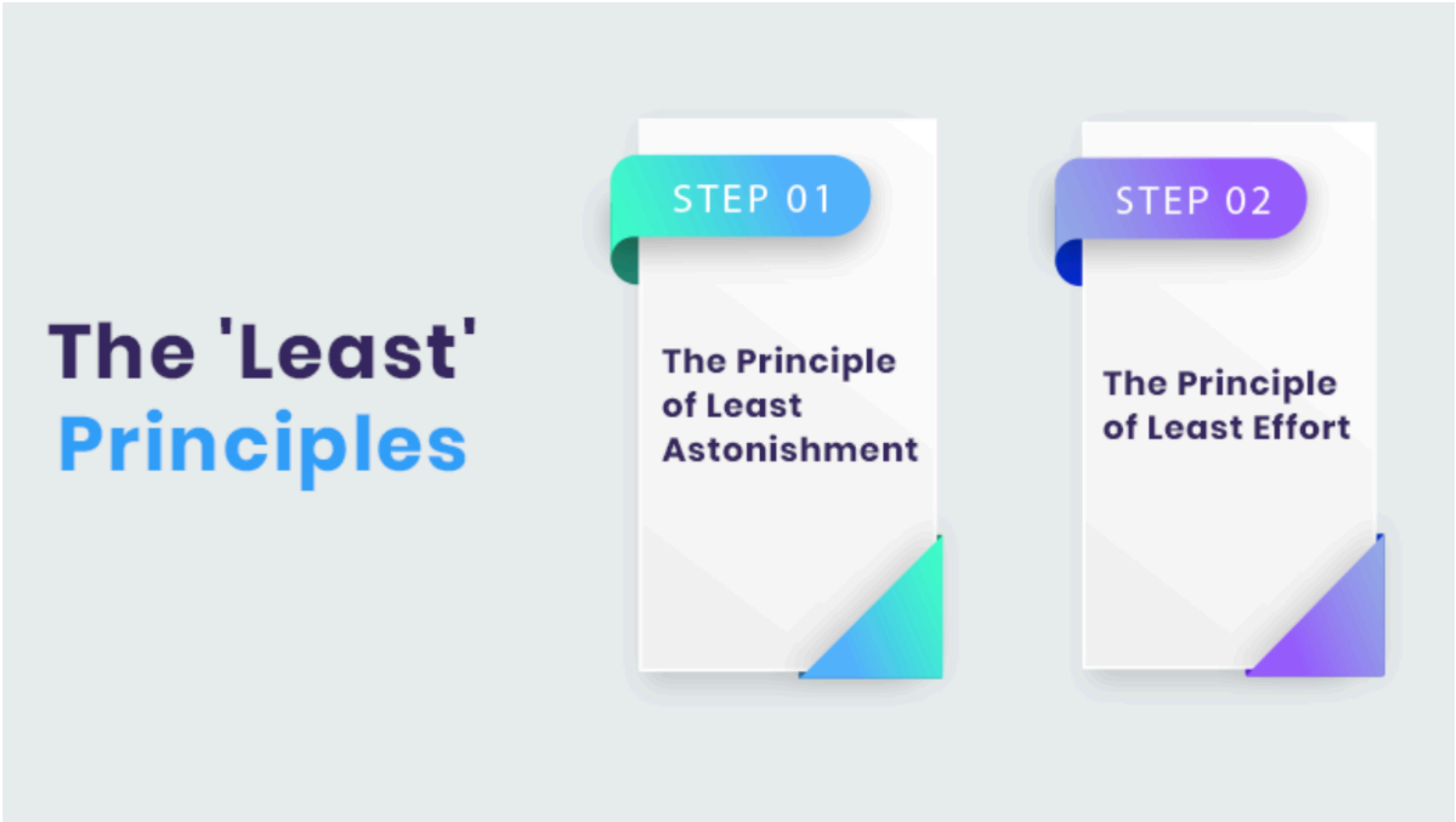
- **I- Interface Segregation Principle:** Keeping this principle in mind is one of the most important things you can do to increase efficiency. According to it, existing interfaces should not be augmented by the addition of new methods. Each interface should cater to a particular client, and each class must implement several interfaces.

These are the underlying principles of SOLID:

- **D- Dependency Inversion Principle:** According to this principle, high-level modules should never rely on low-level modules. Each module should be independent of the other. Also, abstractions should not rely on details, but details should be based on abstractions. Following this principle, one can introduce an interface abstraction to avoid dependencies between high-level and low-level modules.

TO DO - Evaluate your Wavestone Designs/ Architecture against the SOLID principles (where appropriate)

The 'Least' Principles



https://itechnolabs.ca/software-architecture-5-principles-you-should-know/

https://itechnolabs.ca/software-architecture-5-principles-you-should-know/

Software Architecture Metrics

Average Component Dependency

Measures component interdependencies.

Propagation Cost

Assesses impact of changes across components

Cyclomatic Complexity

Counts linearly independent paths through code.

01 02 03 04 05 06 07 08 09 10

Structural Debt Index

Indicates overall architectural debt.

Maintainability Index

Combines metrics like cyclomatic complexity and code size.

Depth of Inheritance

Measures inheritance levels in object-oriented designs.

Relative Cyclicity

Qualifies Degree of Cyclic Dependency

Biggest Cycle Group

Large Group of Independent Components

Component Coupling

Measures Independence between components

Nesting Depth

Deepest levels of nested Structures

https://medium.com/oolooroo/metrics-for-software-architects-and-designers-25b736a17a99

Software metrics warn us of lurking **architectural and technical debt.**

- **Average Component Dependency (ACD):** Measures how many other components a given component depends on. Lower values are preferred as they suggest a more modular architecture.
- **Propagation Cost (PC):** Quantifies how a change in one component might propagate to other components. Ideally, this should be minimized to ensure localized changes.
- **Structural Debt Index (SDI):** A composite metric that captures the overall architectural debt. Higher SDI values can hint at increased maintenance costs in the future.

Software metrics warn us of lurking **architectural and technical debt.**

- **Cyclomatic Complexity:** Measures the number of linearly independent paths through a program's source code, providi insight into the code's testability and maintainability.
- **Depth of Inheritance:** Reflects the inheritance levels in obje oriented programs. A deeper inheritance tree might lead to more complexity and reduced modularity.

TO DO - Estimate/Calculate the ACD and/or PC of your architecture

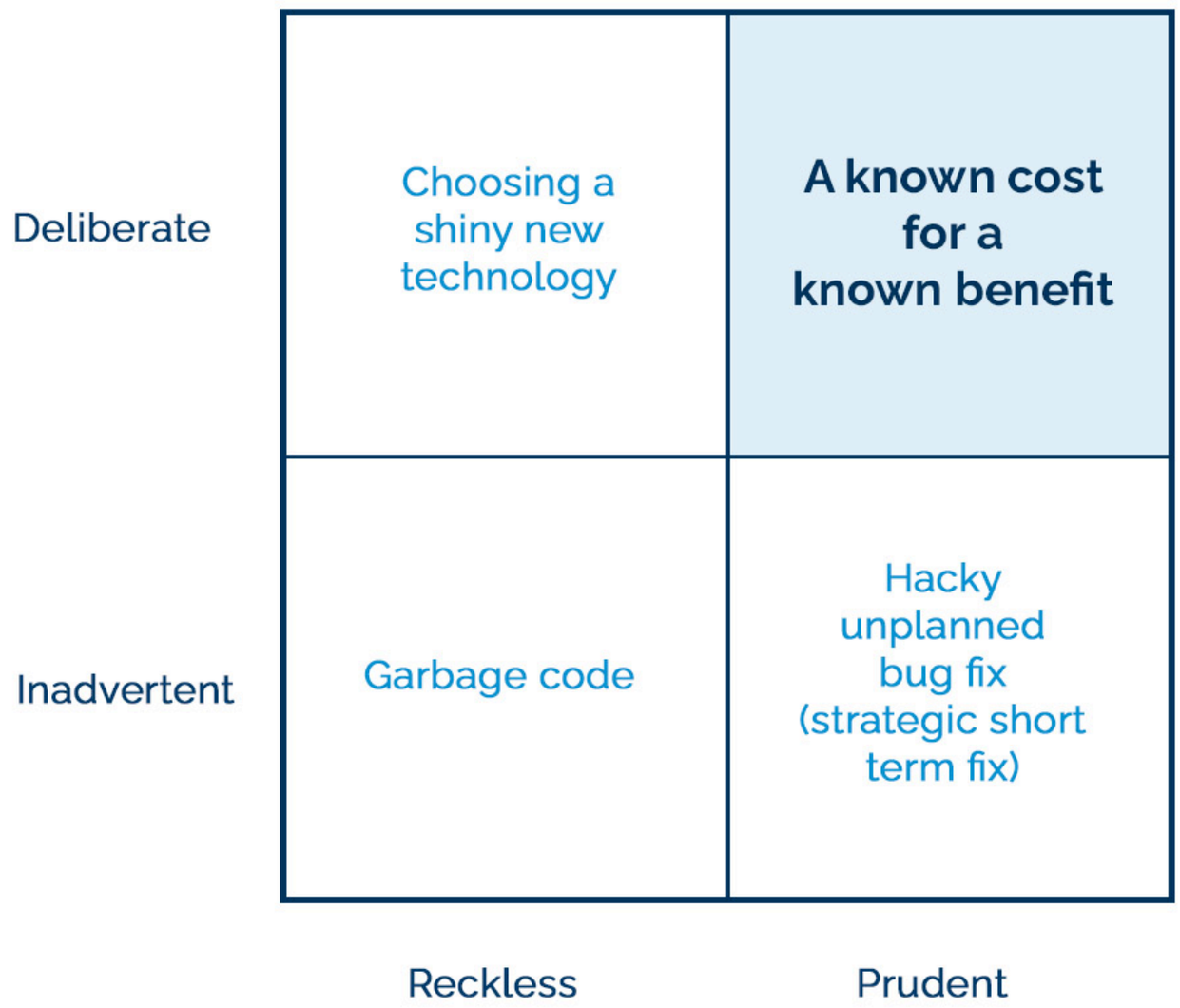
the need for certain actions, such as refactoring, and obtain the necessary

Software Architecture and Design Metrics – Comparison Matrix

Metric Name	Description	Purpose	Thresholds / Targets	Tools	Impact on Quality Attributes	Actionable Insights
Average Component Dependency	Measures average number of dependencies per component.	To reduce coupling and improve modularity.	Lower is typically better.	Static analysis tools	Maintainability, Modularity	Reduce dependencies where possible.
Propagation Cost	Indicates the impact of changes in one component on others.	To assess the impact of changes and improve system resilience.	Lower is typically better.	Impact analysis tools	Maintainability, Testability	Minimize component interdependencies.
Structural Debt Index	Assesses the cost to fix design shortcuts.	Encourage sound design decisions and reduce technical debt.	Lower is typically better.	Code quality tools	Maintainability	Prioritize refactoring where index is high.
Maintainability Index	Predicts the maintainability of the code.	To assess and improve code maintainability over time.	Higher is better (100 max).	Code analysis tools	Maintainability	Improve code readability and reduce complexity.
Cyclomatic Complexity	Counts the linearly independent paths through a program.	Minimize complexity for easier maintenance and testing.	Varies by component size; generally, lower is better.	Static analysis tools	Testability, Reliability	Simplify complex code.
Depth of Inheritance	Measures 'depth' of inheritance tree in OOP.	Prevent overly complex inheritance hierarchies.	Shallower is typically better.	UML tools, Code analysis tools	Reusability, Maintainability	Flatten deep inheritance trees where possible.
Relative Cyclicity	Quantifies degree of cyclic dependencies.	Identify areas where design can be improved to reduce complexity.	Zero cycles is ideal.	Architecture analysis tools	Maintainability	Eliminate cyclic dependencies.
Size of Biggest Cycle Group	Identifies largest group of interdependent components.	To highlight and address complex interdependencies.	Smaller groups are preferred.	Architecture analysis tools	Maintainability	Break down large cycle groups.
Maintainability Level	Qualitative measure of ease of maintaining software.	To guide improvements in software maintainability.	Higher levels are preferred.	Code quality metrics	Maintainability	Implement code quality improvements.
Max Nesting Depth	Indicates deepest level of nested structures in code.	To maintain code readability and manage complexity.	Shallower is better.	Code review tools	Maintainability, Readability	Refactor to reduce nesting.
Component Coupling	Measures interdependence between components.	Reduce coupling to enhance modularity and independence.	Lower indicates better modularity.	Static analysis tools	Modularity, Maintainability	Decouple components where feasible.
Component Cohesion	Assesses how related the functionalities of a component are.	Improve the single-responsibility principle within components.	Higher cohesion is desired.	Static analysis tools	Modularity, Maintainability	Increase cohesion by refactoring.
Component Size	Typically measured in lines of code or function points.	Keep components manageable and understandable.	Smaller, well-defined components are better.	Code analysis tools	Maintainability, Testability	Keep components small and focused.
Component Complexity	Often measured by cyclomatic complexity or similar metrics.	To signal when a component may be too complex to maintain or test effectively.	Lower complexity is better.	Static analysis tools	Maintainability, Testability	Simplify complex components.

https://medium.com/oolooroo/metrics-for-software-architects-and-designers-25b736a17a99

Technical Debt



https://akfpartners.com/growth-blog/what-is-tech-debt