UNIVERSIDAD DE LAS FUERZAS ARMADAS

Name: **Andy Josue Calderón Merchan**

Date: **1/2/2022**

UNIFYING DEFINITIONS FOR MODULARITY, ABSTRACTION, AND ENCAPSULATION AS A STEP TOWARD FOUNDATIONAL MULTI-PARADIGM SOFTWARE ENGINEERING PRINCIPLES

SOFTWARE ENGINEERING GOALS

The goal of software engineers is to deliver high-quality products on time and on budget.

Desirable Qualities:

- understandability

- security

- testability

THREE CORE PRINCIPLES

- Modularity
- Abstraction
- Encapsulation

COMMON PARADIGMS

Multi-paradigm software development is now supported by several languages and development environments (MPSD).

PROBLEM BACKGROUND

All of these common software development approaches prioritize modularity, abstraction, and encapsulation to varying degrees. Different paradigms have different notions and definitions for these principles.

CORE PROBLEM

Particularly for multiparadigm software development, there are no common, unifying definitions.

- Programmers, especially in multi-paradigm software development, frequently do not understand the essential ideas and hence do not profit from their guidance. The lack of unified definitions makes tool support difficult.

CONTRIBUTIONS OF THIS INITIAL PAPER

- The goal of this study is not to reintroduce modularity, abstraction, or encapsulation.

Allows the definition of a principle to go beyond simply expressing the underlying concepts also provides a foundation for evaluating adherence to the principle.

Propose drafts of paradigm-independent MAE principles definitions.

PURPOSE OF PRINCIPLES

1. a truth or proposition that can be used to back up a claim

2. a set of rules or guidelines
3. an element that gives a product a distinct quality (e.g., a desired quality)

SOFTWARE ENGINEERING PRINCIPLES

A principle is a fundamental concept (truth, statement, rule, etc.) that leads to and supports reasoning about desirable qualities like maintainability, efficiency, openness, reusability, and so on.

BEST PRACTICES, PATTERNS, AND IDIOMS

Best practices are procedures or strategies that enable developers to follow principles without having to think about the specifics of a situation on a theoretical level. Idioms are ways to express a certain method or data structure in a programming language.

PRINCIPLES VS. BEST PRACTICES, PATTERNS, AND IDIOMS VS. DESIRABLE CHARACTERISTICS

Although practices, patterns, and idioms are not principles in and of themselves, they aid developers in adhering to the principles.

Principles aren't desirable qualities, but sticking to them is.

PRINCIPLE-DEFINITION TEMPLATE

OBSERVATIONS RELATIVE TO MODULARITY

Design decisions must be made and applied in one location, users must be "hidden" from design decisions.

When software evolves in expected (and some unexpected) ways, good modularity should reduce ripple effects.

• Coupling: the degree to which components depend on each other.

• Cohesion: the degree to which the properties of a component relate to the component's primary responsibility.

- "To construct modules that are coherent (by grouping conceptually related abstractions) and loosely connected (by minimizing dependencies among modules)," according to Grady Booch.

PARADIGM-INDEPENDENT DEFINITION FOR MODULARITY

When a software system is made up of loosely linked and cohesive components, each critical or changeable design decision is isolated in one component, and related ideas are kept as near together as possible. Understanding, testability, maintainability, dependability, security, extensibility, and reuse can all benefit from modularity.

PRACTICES AND CRITERIA:

• Localization of design decisions
• Low Coupling
• High Cohesion
• Modular Reasoning

TRADEOFFS:

Localization of design decisions and strong cohesion can result in a large number of fine-grained components, which is beneficial for testability, extensibility, and reuse but can make it difficult to understand.

PARADIGM NOTES:

Packages and composite components sometimes have many tasks, but those responsibilities should be consistent, as indicated in the practices and criteria section.

Every predicate must reflect a single notion or obligation, according to the developers. To put it another way, each predicate should be really cohesive.

Every design decision that is interesting or potentially adjustable must be localized. For each design decision, this is accomplished by specifying a predicate and a set of rules.

OBSERVATIONS RELATIVE TO ABSTRACTION

"The core of abstractions," according to John Guttag, "is preserving knowledge that is important in a given context while forgetting information that is unnecessary in that context."

Software abstraction necessitates sifting through enormous and diverse collections of details to identify the most important and distinct notions. It's difficult to make good software abstractions.

TWO COMMON PROBLEMS WITH ABSTRACTION

Other components end up relying on details that aren't explicitly specified in the abstraction because the characteristics aren't established as such.

Too much is "forgotten" or deemed irrelevant due to abstraction.

Users of the abstraction are given insufficient control.

PARADIGM-INDEPENDENT DEFINITION FOR ABSTRACTION

The available capabilities and functionality should be limited to what other components may require or rely on.

Understanding, testability, maintainability, and reusability can all be improved by adhering to the abstraction principle. It may also make it easier for developers to follow modularity because it will highlight flaws such as design decision localization, excessive dependency, and low cohesion.

PARADIGM-INDEPENDENT DEFINITION FOR ABSTRACTION

PRACTICES AND CRITERIA:

• Meaningful labels and identifiers
• Context-aware labels and identifiers
• Abstraction completeness
• Abstraction sufficiency

OBSERVATIONS RELATIVE TO ENCAPSULATION

The absence of a statement, rule, or practice qualifies these definitions as principle definitions. Furthermore, these notions frequently overlap when it comes to modularity and its associated criteria.

- The bundling of data with operations

  In class-based languages, definitions have given rise to access-restricting language constructs such as the private and protected modifiers. Although definitions of this category are useful, they fall short of capturing the entire potential of encapsulation.

- The hiding decisions behind logical barriers.

PARADIGM-INDEPENDENT DEFINITION FOR ENCAPSULATION

Ensure that a component's secret implementation details are separated from other components so that they cannot be viewed or modified. As a result, testability, maintainability, and dependability will improve. It will also aid in the separation of concerns and the avoidance of unintentional coupling.

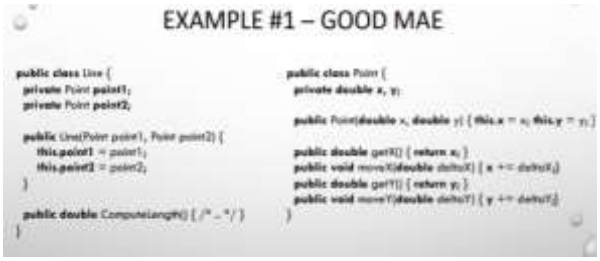
PRACTICES AND CRITERIA:

• Conceptual barriers
• Programmatic barriers
• Usage barriers

THE NON-REDUNDANCY AND COMPLIMENTARY NATURE OF THE MAE PRINCIPLES

CRITERIA:

1. There can be no particular case of or subsumption of a general principle by another principle or combination of principles.
2. Developers should have the option of adhering to one principle but not the others.



SUMMARY

The aim of software-engineering principles has been clarified

- A framework for documenting principles was proposed.
- These definitions have been shown to be non-redundant and complementary.

FUTURE WORK

Form research questions about the use of MAE principles in mixed-paradigm environments.

- Construct concrete empirical studies to investigate those issues.
- Investigate metrics for evaluating quality in a mixed-paradigm software system in a systematic manner.
- Extensive research into different design principles than MAE.