Unifying Definitions for Modularity, Abstraction, and Encapsulation as a Step Toward Foundational Multi-Paradigm Software Engineering Principles

Stephen W. Clyde
Computer Science Department
Utah State University
Logan, Utah, USA
email: stephen.clyde@usu.edu

Jorge Edison Lascano
Departamento de Ciencias de la Computación
Universidad de las Fuerzas Armadas ESPE
Sangolquí, Ecuador
email: edison_lascano@yahoo.com

***Abstract*—The concepts of modularity, abstraction, and encapsulation have been an integral part of software engineering for over four decades. However, their definitions and application vary between software development paradigms. In some cases, conflicting definitions exist for a single paradigm. This paper first defines the concept of a principle for software-engineering, in general, and then provides a template for documenting principles so they can be easily referenced and taught. Next, it proposes initial unified definitions for modularity, abstraction, and encapsulation that are applicable to multiple programming paradigms. It then shows that these unified definitions for modularity, abstraction, and encapsulation are non-redundant but complimentary of each other. Finally, it discusses future work for refining and validating these unified definitions through a series of empirical studies.**

Keywords-software engineering principles; modularity; encapsulation; abstraction.

I. INTRODUCTION

Ideally, software engineers aim to build quality products on time and within budget [1, p. 8], where a quality product is one that supports the required functionality and has appropriate levels of understandability, testability, maintainability, efficiency, reliability, security, extensibility, openness, interoperability, reusability, and other desirable characteristics. On the surface, different programming paradigms appear to embrace different principles for helping developers achieve these characteristics. However, there are more commonalities than dissimilarities among these principles and developers would benefit from more general, unified definitions, especially as mixed-paradigm software development becomes more prevalent.

Object orientation (OO), which is currently the most common paradigm, places considerable importance on encapsulation and abstraction [2][3], but it also advocates modularity with low coupling and high cohesion [2][4]. Structural programming emphasizes modularization, but can be make use of control abstraction, certain kinds of data abstraction, and encapsulation. Functional programming (FP) emphasizes modularity and encapsulation using pure functions that have no side-effects [5][6], but also benefits from control abstraction. Logic programming (LP) emphasizes behavior (rule) and data (predicate) abstraction, but can leverage modularity and encapsulation. LP also takes advantage of control abstraction by hiding nearly all the underlying inference algorithm.

The *modularity*, *abstraction*, and *encapsulation* (MAE) principles are beneficial to virtually every programming paradigm. Unfortunately, there are no generally accepted definitions for the MAE principles or agreement on their application and potential benefits.

One problem is that software-engineering publications typically focus on a single paradigm, and if they define principles, do so using concepts and terms specific to that paradigm. Also, pressure to push the state-of-art forward and publish innovations encourages authors to reinvent or recast principles instead of adapting or generalizing existing work.

A lack of general, unifying definitions has led to overlapping and sometimes conflicting ideas about design principles. Consider for example, the SOLID principles [7]-[10], which are five design principles popular in object orientation (OO). Their definitions, which are specific to OO, have significant similarities with early work on the MAE principles, but differ in some subtle ways. Specifically, the first SOLID principle, called the *Single Responsibility Principle* (SRP), overlaps with the original notation of modularity for high cohesion but only deals with it at a class level [2, p. 54][11]. Similarly, the *Open/Closed Principle* overlaps with modularity for minimal coupling [2, p. 54], at least at a class-level. The five SOLID principles also overlap with themselves. For example, the *Interface Segregation Principle* can be re-cast as an application of SRP in the context of interface abstractions.

Literature about design principles is sparser for some paradigms than others. For example, there is relatively little written about design principles for FP and LP compared to OO and SP. This does not mean, that design principles are less important in these paradigms, but that developers are expected to carry them over from more mainstream paradigms, like OO and SP.

Problems caused by the lack of unified definitions for design principles is becoming more serious as new paradigms continue to emerge and programming languages evolve to support multiple paradigms. Java, C#, JavaScript, and C++, for example, now support mixed-paradigm approaches, where developers can use constructs from OO, FP, Aspect Orientation (AO), and Generic Programming (GP), and more, together within the same system [5][12].

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-590-6

This paper makes three initial contributions towards addressing this problem. First, Section II clarifies the purpose of software-engineering principles, in general, and distinguishes them from "best practices", idioms, and patterns. Section II also purposes a template for documenting principles that allows a principle's definition to go beyond just communicating the underlying concepts. Specifically, it provides a basis for assessing of adherence to the principle and a foundation for teaching the principle to programmers. Next, using this template, Sections III-V propose drafts of paradigm-independent definitions for the MAE principles. There are undoubtedly other paradigm-independent design principles besides the MAE, but these three are a good starting point because of their non-redundant yet complimentary relationships with each other. An explanation of these two relationships is given in Section VI and as another contribution of this paper.

The work presented here is not about inventing or reinventing the concepts of modularity, abstraction, or encapsulation. Instead, it aims to synthesize existing knowledge into a simple, accessible form for software developers and software-engineering education. Although this paper presents three contributions towards meeting this objective, it is just the first step that provides 1) a starting point for formulating research questions related to software quality across multiple paradigms, 2) a foundation for designing and conducting empirical studies, and 3) a basis for eventually defining metrics for systematically assessing quality in mixed-paradigm software systems. Section VII discusses these follow-on efforts in more detail, in addition to providing a summary of the contributions of this paper.

II. DESIGN PRINCIPLES

Before considering the MAE principles in detail and presenting unified definitions for them, it is necessary to first establish the meaning and purpose of software design principles and distinguish them from desirable characteristics, metrics, processes, best practices, patterns, idioms, and artifacts. This is important to reduce potential confusion, because existing literature uses a term, like "abstraction" to represent more than one of these ideas. For example, some authors define abstraction as the process or practice of isolating and distinguishing common features among objects [13]-[15]. Others define abstraction as software artifacts that specify conceptual boundaries between objects or types of objects [2, p. 38][16]. In this paper, we will define abstraction as a principle, and not as a process or artifact.

The Merriam-Webster and Oxford dictionaries define a *principle* as 1) a truth or proposition that supports reasoning, 2) a rule or code of conduct, or 3) an ingredient that imparts a characteristic quality (e.g., desirable characteristic) [17][18]. We specialize these definitions for software as follows: a *software design principle* is 1) a truth or proposition that supports reasoning about the desirable characteristics of a software system, 2) a rule for creating software with certain desirable characteristics, or 3) an aspect of software design that imparts certain desirable characteristics. In other words, a principle is a foundational concept (truth, proposition, rule, etc.) that leads to and supports reasoning about desirable characteristics, such as maintainability, efficiency, openness, reusability, etc.

If some concept, P, is a good principle for achieving a set of desirable characteristics Q, then the degree to which a software engineer adheres to P should predicate the degree to which Q is present in the software artifacts. In other words, the presence of Q is the goal or purpose of P. Ideally, the presence of Q in artifacts should be detectable or measurable through metrics based on the P [19][20]. However, creating valid and reliable metrics for measuring desirable qualities has proven to be challenging. We believe that one reason for this is that the principles upon which they are supposed to be based are not yet sufficiently defined and details about their relationships to desirable characteristics are still lacking.

Best practices are procedures or techniques that help developers adhere to principles without having to consider the details of a situation at a theoretical level. For example, consider the practices of "prefer aggregation over inheritance" and "program to an interface or abstract" [21][22]. By knowing and using these practices, a developer can improve modularity, abstraction, and encapsulation, without having to analyze in detail all the alternatives in terms of their resultant desirable characteristics. Unfortunately, best practices like these two tend to be specific to a programming paradigm or language.

Patterns also help developers achieve desirable characteristics; they exemplify principles by providing proven solutions to reoccurring problems in specific contexts [23]. Similarly, an idiom can help developers adhere to a principle by providing a solution for expressing a certain algorithm or data structure in a specific programming language [24].

Although software design principles are themselves not desirable characteristics, practices, patterns, idioms, or artifacts, they are at the heart of software engineering and their definitions should give developers the means to 1) reason about design decisions, 2) assess whether or how well a design either conforms to a principle, and 3) balance choices between conflicting objectives and design alternatives. The latter is important because software engineers must often make choices that weaken one desirable characteristic in favor of strengthening another. For example, a developer may have to sacrifice some extensibility in favor of efficiency.

Table I shows a template for capturing the definition of a software design principle in a way that accomplishes the three objectives listed above. As with practices, patterns, and idioms, a principle's *name* must accurately express the nature of the concept, because that name will become part of a vocabulary. The *essence* of a principle's definition is a short statement that aims to covey the fundamental concept at level that is understandable for most programmers and can be taught to beginning programmers. The essence should highlight the principles relationship to hoped-for desirable characteristics.

The next element of the template is a section that describes practices for following the principle and criteria that can be used to determine if a software system or component adheres to the principles. Like the essence, the practices and criteria need to be paradigm-agnostic and written a level that is understandable for most programmers.

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-590-6

TABLE I. PRINCIPLE-DEFINITION TEMPLATE

Name	*The name of the principle*
Essence	*A statement of the truth, proposition, or rule embodied in the principle and its relationship to hoped-for desirable characteristics*
Practices and Criteria	*Processes or criteria that, if followed, should help the developer adhere to the principle and lead to the hoped-for desirable characteristics*
Tradeoffs	*Factors that can help a developer decision when to go against a principle, in lieu of a conflicting objective. These factors may include the consequences of not meeting the criteria or common tradeoffs*
Paradigm Notes	*Notes about applying the principle in various paradigms*
Examples	*Good and poor examples in different paradigms*

The next element is a section that describes costs or other factors associated with following the principle that can help developers decide when to violate a principle, in lieu of some other conflicting objective. It may also include notes about the consequences of not meeting following the suggested practices or meeting the adherence criteria.

The last two elements of the template are optional, but serve to help developers apply a principle for a specific paradigm and to teach the principle to new programmers. Naturally, the knowledge captured in these two elements will be paradigm specific and could refer to a wide range of artifacts, like source code, build scripts, hyper-text, style sheets, and configuration files.

III. MODULARITY

Over the last 50 years, many respected authors have addressed the topic of modularity or modularization, which is the process of trying to achieve good modularity. One of the first was David Parnas, who, in 1972, outlined criteria for decomposing software into modules such that individual design decisions could be hidden in specific components [25]. His landmark paper set the stage for other research on using modularization to manage complexity [26]-[28].

These early works illuminated an important facet of good modularity, namely that a decision design, particularly one that is likely to change, should be isolated in one software component. We call this rule for modularity "localization of design decisions". By itself, this rule does not prescribe where the implementation of design decision should be placed, just that it should not be replicated or spread across multiple components. Failure to follow this rule leads to the "Duplicate Code" smell [29], which in turn can reduce maintainability.

Two other propositions or rules that are frequently associated with modularity are low coupling and high cohesion [4][30]-[32]. Low coupling exists when each component of a system is free of unnecessary dependencies (explicit or implied) on other components. Although coupling was first defined for SP, other definitions have been created for OO and AO [30][33]. It has even been applied to LP [34]. Cohesion is the degree to which the elements of one component relate to each other or the component's primary responsibility [31]. Ideally, each component should have a single responsibility, as advocated by SRP [7]. Like coupling, definitions for cohesion have been proposed in multiple paradigms [35]-[37].

Grady Booch said that the objective of modularization is "to build modules that are cohesive (by grouping logically related abstractions) and loosely coupled (by minimizing the dependencies among modules)" [2, p. 54]. It is widely believed that achieving low coupling and high cohesion results in software programs that are more understandable, testable, maintainable, reliable, secure, extensible, and reusable. It is also believed that they will avoid common code smells, like *Long Method*, *Large Class*, *Long Parameter List*, *Feature Envy*, and *Inappropriate Intimacy* [29][38].

Another facet of modularity deals with how far away from some component, *C*, a developer must look to reason about the functionality of *C*, particularly in preparation for making corrections or extensions. The component *C* has *modular reasoning* if a developer only needs to examine its implementation, public abstraction (e.g., its interface), and the public abstractions of referenced components [39]. *Extended modular reasoning* is a looser form of modularity, where developers also need to examine the internal details of referenced components [39]. *Global reasoning* is the loosest form of modularity, where developers may need to examine some other component in the system to reason about *C* [39]. A system comprised primarily of components with modular reasoning or extended modular reasoning is considered better than those with lots of components that require *global reasoning*. Some paradigms try to minimize global reasoning by introducing constructs that encourage the localization of certain design decisions, e.g., interfaces for responsibilities in OO and aspects for crosscutting concerns in AO.

Below is a definition for modularity, following the template given in Section II. This definition addresses the issues discussed above and is applicable to multiple programming paradigms. Due to the space limitations of this paper, the paradigm notes are limited and detailed examples are not show.

A. Essence

Modularity exists in a software system when it is comprised of loosely coupled and cohesive components that isolate each significant or changeable design decision in one component and ensure that related ideas are as close as possible. Modularity can improve understandability, testability maintainability, reliability, security, extensibility, and reuse. It can also help with collaboration during the software development process by outlining loosely coupled work units [2, p. 54].

B. Practices and Criteria

1) Localization of design decisions: Design decisions are identified and prioritized by significance and likelihood for future change. In a system with localization of design decisions, every significant or changeable decision is implemented in one component.

2) High cohesion: When making design decisions, a developer considers all the responsibilities of a given component and tries to ensure that there is just one or that responsibilities are all closely related. In a system with high

cohesion, every component has one primary responsibility or reason to change. Component's primary responsibility may be a high level, when the component is an aggregate or when it directs behaviors in other components.

3) Low coupling: When making design decisions, a developer aims to minimize the degree and number of dependencies (explicit or hidden) between components. In a system with low coupling, the components are free of hidden dependent and unnecessary explicit dependencies. Also, explicit dependencies are directly visible in the code, e.g., data-type references and function calls.

4) Modular reasoning: Developers should give preference to decompositions with modular reasoning over extended modular reasoning, and to extended modular reasoning over global reasoning. A system has modular reasoning if developers can understand the details of a component by examining only its implementation, public abstraction, and the public abstractions of referenced components.

C. Tradeoffs

Localization of design decisions and high cohesion can lead to many fine grain components. Although these help with testability, extensibility, and reuse, it can sometimes hinder readability. One common solution is to package small, related components into aggregation components.

Although modularity by itself will not guarantee understandability, testability maintainability, reliability, security, extensibility, and reuse, the lack modularity will compromise these desirable characteristics. Adherence or violation of the modularity principles typically affects multiple components. For example, if a design decision is not localized, then it can comprise the maintainability of every component that deals with that design decision.

D. Paradigm Notes

Only snippets of the paradigm notes from the full principle definition are shown here.

For OO, packages, classes, and methods are the primary types of components that developers need to work with when modularizing, but they may also consider other types of artifacts like build scripts, configuration files, and style sheets. Composite components, like packages, often have multiple responsibilities, but those responsibilities should be cohesive as described above in practices and criteria section.

For FP, the components are primarily functions, but could also include other artifacts like build scripts. By definition, a pure function in FP only depends on values that are passed in as input parameters, so developers can minimize coupling by ensuring that a function's parameters represent nothing than exactly what the function needs.

For LP, the components are primarily predicates, rules, and facts. Modularity is achieved by doing three things. First, developers must ensure every predicate represents a single idea or responsibility. In other words, every predicate should be highly cohesive. Second, every interesting or potentially changeable decision decisions need to be localized. This is done by defining a predicate and set of rules for each design decision.

IV. ABSTRACTION

From a process perspective, abstraction is the act of bringing certain details to the forefront while suppressing all others. John Guttag said that "the essence of abstractions is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context" [40]. This is something that most humans learn naturally as part of their cognitive and social development, in conjunction with learning to think conceptually and symbolically [41].

Nevertheless, it is interesting that a significant percentage of computer science students, and by extension software engineers, struggle with creating good software abstractions [15]. Perhaps, this is because creating good software abstractions are much harder to create than everyday abstractions. Software abstraction requires developers to sift through large and diverse collections of details about legacy systems, current and future requirements, existing and emerging technologies, tool-stack idiosyncrasies, work allocation nuances, and more, and then determine the most salient and distinguishing concepts. Fittingly, Abbott et al. described an abstraction as the "reification and conceptualization of a distinction" [13].

From an artifact perspective, every software component has an abstraction, independent of whether the developer thought about or declared it explicitly. A component's abstraction is everything about the component that is visible externally. Examples of common elements that contribute to component abstractions include descriptive or identifying labels like function names, class names, predicate names, and CSS style names; the public data members and methods of classes; the parameters of a function or method; meta-data annotations; and much more. None of these elements alone can represent a complete abstraction for the component to which they belong. A component's full abstraction consists of everything that other components might explicitly or implicitly depend on. Ideally, a full abstraction should be programmatically declared or documented, so it is readily accessible to developers and other components. However, this is rarely the case. Instead, components typically end up with leaky abstractions [42]. One reason for this is that developers often do not take the time to document all external discernable characteristics, like performance properties and side effects. Other sources of leaky abstractions are incomplete thinking, design errors, and implementation bugs that do not immediately contradict required functionality or exhibit damaging characteristics. Overtime, other components can come to depend on those erroneous characteristics. Then, when the original problems are corrected, all the components that depended on the erroneous characteristics fail.

Another potential problem is too much abstraction. This occurs when a component's abstraction does not expose all elements that others need to use or the abstraction does not provide sufficient parameterization for the elements that need to be customizable. These kinds of problem lead to lack of flexibility, which in turn leads to sloppy hacks that can compromise the overall quality of a system.

Below is a simple definition for the abstraction principle that can help developers capture and communicate meaningful

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-590-6

abstractions without leakage or over abstraction. The definition is sufficiently broad to cover control, function, behavior, and data abstraction. Note that the abstraction principle by itself does not address the placement or organization of design decisions, i.e., the decomposition of a system into components. The modularity principle guides decomposition and refactoring. Instead, the abstraction principle focuses on exposing and communicating the right aspects of a component, namely those that others will need to depend on.

A. Essence

For each component, there is an explicit and clear declaration of the component's accessible features or functionality. Depending on the paradigm and programming language, this declaration may be part of the source code, meta data, or documentation. The exposed features and functionality should be no more and no less than what other components may need or depend on.

Adherence to the abstraction principle can improve understandability, testability, maintainability, and reusability. It can also allow developers to follow modularity more effectively, because it will bring to light weakness with localization of design decisions, unnecessary coupling, and low cohesion.

B. Practices and Criteria

1) Meaningful labels and identifiers: A system has meaningful labels and identifiers, when each one expresses the essence and distinguishing aspect(s) of its associated component or element.

2) Context-aware labels and identifiers: This exists when the label or identifier for a component does not contain redundant information that can be inferred from the component's context or scope. For example, a method called *GetFirstName* in a Person class makes for better abstraction than *GetPersonFirstName*, because the context of person can be derived from the class name.

3) Abstraction completeness: Whenever possible, all externally visible characteristics for a component are explicitly declared as part of the component's definition, implementation, or meta data. When that is not possible, documentation must explain these characteristics clearly and concisely, and the closer document is to the component's implementation, the better.

4) Abstraction sufficiency: This exists when all the elements of a component that should be visible to outside components are exposed through the component's abstraction.

C. Tradeoffs

Not following the practices and criteria listed above can result in the loss of the hoped-for desirable characteristics in portion to the degree and amount of non-adherence.

D. Paradigm Notes

Only snippets from the notes for LP are shown here to give the reader a sense of what this part of the full definition contains.) As mentioned, the primary components in LP are predicates, rules, and facts. The abstractions for these components expose the "logic" of system while hiding most of its control aspects, namely the process for drawing conclusions or deductions. Predicates represent relations from the problem domain or design decision. Given a predicate, all the rules with the predicate in their heads and facts stated with that predicate form another kind of higher level abstraction in LP. One of these abstractions exposes all that is known about a relation or decision.

The abstraction for a predicate is comprised of a name and some number of parameters. A developer should choose a name that expresses the predicate's essence clearly, concisely, and accurately. Doing so not only improves understandability from an abstraction perspective, it helps with modular reasoning, which in turn contributes to better modularity. Facts and rules are typically given labels in LP, but can be grouped together into higher level packages to form higher level abstractions.

V. ENCAPSULATION

Encapsulation is typically equated with OO, but it is not unique to OO nor did it originate with OO. In fact, many old devices, like mechanical clocks from the middle ages, are good examples of encapsulation. All their implementation details, e.g., the time keeping mechanisms, are hidden behind a clock face in a sealed container.

In English, encapsulation means to enclose something inside a capsule or container [43]. In other words, it involves two things: the thing is being enclosed and the enclosure. In physical systems, like a train station that needs a clock, the choices for enclosures are relatively limited compared to those available in software systems, where developers have total control over the system's decomposition. As described in Section III, modularity can guide a developer in making good choices for the components, i.e., enclosures, such that each has a cohesive purpose and is loosely coupled to others. Abstraction, as discussed in Section IV, can help developers communicate the essence of component and expose only external characteristics. The principle of encapsulation can then help developers isolate or hide the internal characteristics of a component so others do not accidently become dependent on them.

Unfortunately, the close relationship among encapsulation, abstraction, and modularity has led to some ambiguity in use of these terms. This is particularly true for encapsulation. The various definitions for encapsulation in software engineering literature can be grouped into three categories. The first category includes definition that equate encapsulation to the bundling of data with operations on that data to create *Abstract Data Types* [44][45]. These definitions take either a process or artifact perspective, but typically lack a proposition, rule, or practice that qualify them as principle definitions. Also, these definitions sometimes overshadow or ignore modularity and its associated criteria like localization of design decisions, low coupling, high cohesion, and modular reasoning.

The second category includes definitions that represent encapsulation as a process or technique for hiding decisions behind logical barriers, preventing outsiders from modifying or even viewing the implementation details of components. This

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-590-6

category of encapsulation definitions stems from work on information hiding, which is the dual of abstraction. It has given rise to access-restricting language constructs, such as the *private* and *protected* modifiers in class-based languages. Although definitions of this category are valuable by themselves, they do not capture encapsulation's full potential.

In the third category, definitions explain encapsulation as a process for organizing components so the implementation details of one component can be modified without causing a ripple effect to other components [46]. These definitions focus on the minimization of inter-component dependencies, i.e., coupling. By themselves, these definitions miss other important aspects of encapsulation and blur it with modularization.

There are many documented "best practices" that experts believe can help programmers achieve good encapsulations. For example, in C#, experts believe that the use of auto-implemented properties is much better than public data members because they provide for stronger barrier between the abstraction and implementation details [47]. Unfortunately, such "best practices" tend to be language or paradigm specific.

As mentioned earlier, a design principle must be a truth, proposition, rule, or practice that yields desirable qualities. Below is a definition for encapsulation that aims to comply with this requirement for principles and can guide a developer achieve good encapsulation, independent developer's adherence to the principles of modularity and abstraction.

A. Essence

Ensure that the private implementation details (i.e., the internal characteristics) of a component are insulated so they cannot be accessed or modified by other components. Doing so will lead to better testability, maintainability, and reliability. It will also help with a clear separation of concerns and avoid accidental coupling.

B. Practices and Criteria

1) Conceptual barriers: For each component, a developer should identify the internal structures, behaviors, procedures, and definitions of that component and ensure that they are protected behind conceptual barriers. More specifically, developers should try to identify the minimum required scope for each internal element. For example, in OO, if a data member is only used within the scope invocations for a single method, then that data member should be refactored to a local variable of the method.

2) Programmatic barriers: Developers should ensure that the modifiability and visibility of every internal element is programmatically restricted to the desired scope. Developers should leverage the available features of the chosen programming language, whenever possible.

3) Usage barriers: If there are internal elements that cannot be isolated programmatically, then document appropriate rules for correct usage of the component, so developers can avoid accidental violations of the intended encapsulation.

C. Tradeoffs

Failure to protect a component's internal characteristics from other components opens the doors for abstraction leakage and hidden dependencies, which can damage testability, maintainability, and reliability in surreptitious ways. In cases where programmers are tempted to weaken the encapsulation of some element in a component, like make a data member in a class definition public, they could at least document the intended usage to minimize the formation of accidental hidden dependencies.

D. Paradigm Notes

Only a few snippets of the full paradigm notes are shown here. The mechanisms for achieving encapsulation vary greatly among paradigms and programming languages. For FP in compiled languages, functions can be strongly encapsulated behind their declarations. This is true even for anonymous function. But, for interpretative languages that support FP, functions are just other forms of data and the decisions they encapsulate may be externally visible and modifiable. In those cases, developers need to document what others may and may not access or change.

For OO and typed languages, developers can restrict the scope for each element to the smallest scope within which the element is used. Data members, for example, are typically private and made accessible only through methods. Some languages provide convenience short hands for getter and setter methods to help programming adhere to this best practice. Only methods that need to be used by other components should be public. Also, developers can use package-level scoping to restrict access to public classes or method that are only needed within a package.

VI. The Non-redundancy and Complimentary Nature of the MAE Principles

To be effective for multiple paradigms and for the long-term advancement of software engineering, it is important for general principles to be non-redundant with each other in two ways: 1) no general principle can be a special case of or subsumed by another principle or combination of principles and 2) developers should be able to choose to follow one principle but not the others.

The case for MAE principles meeting the first is condition is relatively straight forward. The essence of modularity deals with the decomposition of a system into components. Neither of the other two principles prescribe guidelines for organizing a system into modules. So, modularity cannot be subsumed by abstraction or encapsulation, and conversely. Abstraction and encapsulation both apply to individual components, which from an artifact perspective, can be thought of as "abstractions" and "encapsulations." However, abstraction and encapsulation from a principle perspective are fundamentally different from each other. In fact, from a principle perspective, they are approximate duals of each other. Abstraction guides a developer in exposing just the right elements of a component so it is easy to understand and use. Encapsulation guides a developer in hiding internal design decisions so other components cannot intentionally or accidentally depend on

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-590-6

them. Abstraction cannot be recast or explained as a special type, variation, or subpart of encapsulation. Similarly, encapsulation cannot be fully explained in terms of just abstraction.

The satisfaction of the second condition for non-redundancy can be shown using a simple example plus three variations, where each one illustrates adherence to one principle but not the others. The example, shown in Figure 1, consists of two classes: *Line* and *Point*, where the *Point* class represents movable points in a 2D coordinate space and the *Line* class represents lines comprised of two points and that know how to compute their own lengths. Fig. 1 shows an implementation that of this simple system that has good modularity, abstraction, and encapsulation according to the definition given in Sections III-V.

```
public class Line {
  private Point point1;
  private Point point2;

  public Line(Point point1, Point point2) {
      this.point1 = point1;
      this.point2 = point2;
  }

  public double ComputeLength() { /* .. */ }
}

public class Point {
  private double x, y;

  public Point(double x, double y) {
    this.x = x; this.y = y; }

  public double getX() { return x; }
  public void moveX(double deltaX) { x += deltaX;}
  public double getY() { return y; }
  public void moveY(double deltaY) { y += deltaY;}
}
```

Figure 1. A simple example implementation with good modularity, abstraction, and encapsulation.

In the first variation (see Fig. 2), the designs decisions are still localized, the classes still have low coupling and high cohesion, and they support modular reasoning. In other words, the system's decomposition has good modularity. However, the system has poor abstraction or encapsulation, caused unnecessary exposure of the internal design decisions and by poor identifiers, which are either do not communicate the true essence of the elements. Of course, there could be other ways that the abstraction and encapsulation could be degraded, but the purpose of this example is to simply show that modularity can exist without abstraction and encapsulation.

```
public class Line {
  public XY point1;
  public XY point2;
  public Line(XY point1, XY point2) { /* … / }
  public double Calc() { /* Compute length … */ }
}

public class XY {
  public double x, y;
  public XY(double x, double y) { /* … / }
}
```

Figure 2. A simple example implement with good modularity, but poor abstraction and encapsulation.

In the second variation (see Fig. 3), the *Line* and *Point* classes have good abstractions, but lack modularity and encapsulation. Specifically, the class definition expose the functionality that other components need to use, with sufficient flexibility. However, the decision for calculating the distance between two points is not localized, which goes against the modularity principle, and the data members in both classes are public, which violates the intended encapsulation.

```
public class Line {
  public Point point1;
  public Point point2;

  public Line(Point point1, Point point2) {/* … */}

  public double ComputeLength()
  {
    return Math.sqrt(Math.pow(point2.getX() –
                              point1.getX(), 2) +
                     Math.pow(point2.getY() –
                              point1.getY(), 2));
  }
}
```

```
public class Point {
  public double x, y;

  public Point(double x, double y) { /* … *. }

  public double getX() { return x; }
  public void moveX(double deltaX) { x += deltaX;}
  public double getY() { return y; }
  public void moveY(double deltaY) { y += deltaY;}

  public double ComputeDistance(Point otherPoint)
  {
    return Math.sqrt(Math.pow(otherPoint.x – x, 2) +
                     Math.pow(otherPoint.x – y, 2));
  }
}
```

Figure 3. A simple example implementation with good abstraction, but poor modularity and encapsulation.

A third variation with good encapsulation but poor modularity and abstraction would be an implementation that had the poor names, i.e., poor abstraction, from the first variation plus the lack of localization of decision designs, i.e., poor modularity, from the second variation.

These three variations show that it is possible for a developer to apply each of the three MAE principles, independently of each other. In fact, there may be some special circumstances where this is exactly what a developer needs to do to meet external requirements or adjust for limitations of a programming language or framework. However, such cases should be rare.

Although the MAE principles are non-redundant, they complement each other nicely. In other words, adherence to one encourages, but does not require, adherence to the others. Specifically, adhering to modularity will set the stage for adherence both abstraction and encapsulation, because modularity brings to light what the components need to know about each other and which the design decisions need to be

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-590-6

encapsulated. Likewise, abstraction can lead to better understanding of the inter-component couplings and therefore better modularization. Also, elements that are not part of a component's abstraction are candidates for encapsulation. Finally, doing encapsulation will act as a double check and balance for both modularization and abstraction.

VII. Summary and Future Work

This paper has explained the purpose of design principles, in general, and provided a template for establishing working definitions that can help with teaching the principles to software developers, assessing adherence, and pursuing research questions. This paper then provided an overview of existing ideas related to modularity, abstraction, and encapsulation and unifying them into initial paradigm-independent definitions. Finally, it showed that these principles are non-redundant and complimentary, in the sense that no combination of two could replace the third and that adherence to one encourage adherence to the others.

The work reported in this paper is just one step forward in a larger effort. Our next step is to formulate research questions related the application of MAE principles in mixed-paradigm environments and then setup concrete empirical studies to explore those questions. Below is a sampling of the research questions that we hope to pursue soon:

- When using OO together with LP, what kinds of functionality or responsibility are best handled using LP?
- When using OO with LP, FP, or GP, is it best to design the overall architecture using an OO approach and encapsulate specific responsibilities in LP-based, FP-based, or GP-based components? If so, why and what kinds of components are best suited for LP, FP, and GP?
- What kinds of responsibilities are best encapsulated in aspects when using AO with OO?
- When using FP with OO, how can developer know if high-order functions are following the MAE principles?

After exploring these and other research questions, we believe that another important step would be to explore metrics for systematically assessing quality in mixed-paradigm software systems. The details of any given metric may end up being platform dependent. However, if there were a suite of metrics based on a unified principle definition, then the metrics might yield measurements that are comparable across software components, even when those components are implemented with different languages or paradigms.

Finally, our plans for future work include investigations into other design principles beyond MAE. For example, we hope to unify definitions for classification and generalization/ specialization. Although these principles are heavily used in OO, they can apply to other paradigms as well.

Overall, the effort to unify principle definitions is important as programming languages continue to expand to support multi-paradigm software development. This effort will require input from many different sources and involve a wide range of subfields within software engineering. To this end, we welcome and encourage collaboration from all who are interested in creating a stronger foundation for software methods, teaching software engineering, improving development tools, or assessing software quality.

References

[1] I. Sommerville, Software Engineering, 10 edition. Boston: Pearson, 2015.

[2] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, and K. A. Houston, Object-Oriented Analysis and Design with Applications, 3 edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2007.

[3] "Abstraction (software engineering)," Wikipedia. 06-Jun-2017.

[4] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using Cohesion and Coupling for Software Remodularization: Is It Enough?," ACM Trans Softw Eng Methodol, vol. 25, no. 3, pp. 24:1–24:28, Jun. 2016.

[5] "Functional programming," Wikipedia. 03-Jun-2017.

[6] "Purely functional programming," Wikipedia. 10-Apr-2017.

[7] R. C. Martin, "The Principles of OOD," PrinciplesOfOod, 2005. [Online]. Available: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod. [Accessed: 07-Jun-2017].

[8] R. C. Martin, "Getting a SOLID start. - Clean Coder," 2009. [Online]. Available: https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start. [Accessed: 07-Jun-2017].

[9] S. Metz, "SOLID Object-Oriented Design - GORUCO 2009." [Online]. Available: http://confreaks.tv/videos/goruco2009-solid-object-oriented-design. [Accessed: 07-Jun-2017].

[10] R. C. Martin, Agile Software Development, Principles, Patterns, and Practices, International ed edition. Harlow: Pearson Education Limited, 2013.

[11] T. DeMarco and P. J. Plauger, Structured Analysis and System Specification, 1 edition. Englewood Cliffs, N.J: Prentice Hall, 1979.

[12] "Comparison of multi-paradigm programming languages," Wikipedia. 22-Mar-2017.

[13] R. Abbott and C. Sun, "Abstraction Abstracted," in Proceedings of the 2Nd International Workshop on The Role of Abstraction in Software Engineering, New York, NY, USA, 2008, pp. 23–30.

[14] "abstraction | cognitive process," Encyclopedia Britannica. [Online]. Available: https://www.britannica.com/topic/abstraction. [Accessed: 06-Sep-2017].

[15] J. Kramer, "Is Abstraction the Key to Computing?," Commun ACM, vol. 50, no. 4, pp. 36–42, Apr. 2007.

[16] W. R. Cook, "On Understanding Data Abstraction, Revisited," in Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, New York, NY, USA, 2009, pp. 557–572.

[17] "Definition of PRINCIPLE." [Online]. Available: https://www.merriam-webster.com/dictionary/principle. [Accessed: 07-Jun-2017].

[18] "principle - definition of principle in English | Oxford Dictionaries," Oxford Dictionaries | English. [Online]. Available: https://en.oxforddictionaries.com/definition/principle. [Accessed: 07-Jun-2017].

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-590-6

[19] V. R. Basili, G. Calderia, and H. D. Rombach, “The Goal Question Metric Approach,” in Encyclopedia of Software Engineering, vol. 2, John Wiley & Sons Ltd, 1994, pp. 528–532.
[20] C. N. Sant’Anna, A. F. Garcia, C. von F. G. Chavez, C. J. de L. Lucena, and A. von Staa, “On the reuse and maintenance of aspect-oriented software: An assessment framework,” in Proc. 17th Brazilian Symposium on Software Engineering, Manaus, Brazil, 2003.
[21] “GOF Advice: Favor Aggregation over Inheritance | Net Objectives.” [Online]. Available: http://www.netobjectives.com/competencies/favor-aggregation-over-inheritance. [Accessed: 06-Sep-2017].
[22] E. Freeman, B. Bates, K. Sierra, and E. Robson, Head First Design Patterns: A Brain-Friendly Guide, 1st edition. Sebastopol, CA: O’Reilly Media, 2004.
[23] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch, Design Patterns: Elements of Reusable Object-Oriented Software, 1 edition. Reading, Mass: Addison-Wesley Professional, 1994.
[24] A. J. Perlis and S. Rugaber, “Programming with Idioms in APL,” in Proceedings of the International Conference on APL: Part 1, New York, NY, USA, 1979, pp. 232–235.
[25] D. L. Parnas, “On the Criteria to Be Used in Decomposing Systems into Modules,” Commun ACM, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.
[26] G. J. Myers, Composite/Structured Design. New York: Van Nostrand Reinhold, 1978.
[27] B. H. Liskov, “A Design Methodology for Reliable Software Systems,” in Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I, New York, NY, USA, 1972, pp. 191–199.
[28] D. L. Parnas, P. C. Clements, and D. M. Weiss, “The Modular Structure of Complex Systems,” IEEE Trans. Softw. Eng., vol. SE-11, no. 3, pp. 259–266, Mar. 1985.
[29] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, and E. Gamma, Refactoring: Improving the Design of Existing Code, 1 edition. Reading, MA: Addison-Wesley Professional, 1999.
[30] Y. Press and L. L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, 1 edition. Englewood Cliffs, N.J: Prentice Hall, 1979.
[31] W. P. Stevens, G. J. Myers, and L. L. Constantine, “Structured Design,” IBM Syst J, vol. 13, no. 2, pp. 115–139, Jun. 1974.
[32] S. R. Chidamber and C. F. Kemerer, “A Metrics Suite for Object Oriented Design,” IEEE Trans Softw Eng, vol. 20, no. 6, pp. 476–493, Jun. 1994.
[33] M. H. Samadzadeh and S. J. Khan, “Stability, Coupling, and Cohesion of Object-oriented Software Systems,” in Proceedings of the 22Nd Annual ACM Computer Science Conference on Scaling Up : Meeting the Challenge of Complexity in Real-world Computing Applications: Meeting the Challenge of Complexity in Real-world Computing Applications, New York, NY, USA, 1994, pp. 312–319.
[34] S. Kramer and H. Kaindl, “Coupling and Cohesion Metrics for Knowledge-based Systems Using Frames and Rules,” ACM Trans Softw Eng Methodol, vol. 13, no. 3, pp. 332–358, Jul. 2004.
[35] G. Gui and P. D. Scott, “Coupling and Cohesion Measures for Evaluation of Component Reusability,” in Proceedings of the 2006 International Workshop on Mining Software Repositories, New York, NY, USA, 2006, pp. 18–21.
[36] A. Kumar, R. Kumar, and P. S. Grover, “Towards a Unified Framework for Cohesion Measurement in Aspect-Oriented Systems,” in Proceedings of 19th Australian Conference on Software Engineering, Australia, 2008, pp. 57–65.
[37] B. C. da Silva, C. Sant’Anna, and C. Chavez, “Concern-based Cohesion As Change Proneness Indicator: An Initial Empirical Study,” in Proceedings of the 2Nd International Workshop on Emerging Trends in Software Metrics, New York, NY, USA, 2011, pp. 52–58.
[38] “Code Smells,” Code Smells. [Online]. Available: https://sourcemaking.com/smells. [Accessed: 08-Jun-2017].
[39] G. Kiczales and M. Mezini, “Aspect-oriented Programming and Modular Reasoning,” in Proceedings of the 27th International Conference on Software Engineering, New York, NY, USA, 2005, pp. 49–58.
[40] J. Guttag, Introduction to Computation and Programming Using Python. The MIT Press, 2013.
[41] J. Piaget and B. Inhelder, The Psychology Of The Child, 2 edition. New York: Basic Books, 1969.
[42] “The Law of Leaky Abstractions,” Joel on Software, 11-Nov-2002. [Online]. Available: https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/. [Accessed: 18-May-2017].
[43] “Definition of ENCAPSULATE.” [Online]. Available: https://www.merriam-webster.com/dictionary/encapsulate. [Accessed: 10-Jun-2017].
[44] “Encapsulation (computer programming),” Wikipedia. 08-Jun-2017.
[45] “Abstract data type,” Wikipedia. 10-Jun-2017.
[46] “Encapsulation & Modularity.” [Online]. Available: https://atomicobject.com/resources/oo-programming/encapsulation-modularity. [Accessed: 10-Jun-2017].
[47] B. Wagner, “Auto-Implemented Properties (C# Programming Guide).” [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/auto-implemented-properties. [Accessed: 10-Jun-2017].

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-590-6