

CHAPTER I

Introduction

The software crisis has been with us for quite some time [Pressman 87], and is not diminishing. A recent Software Engineering Institute (SEI) report identifies capacity to produce software over the near-term as a critical problem [Siegel 90]. The report states that post-deployment software support (PDSS or maintenance) is the most rapidly growing workload of the software process. Maintenance has long been known to be a large consumer of software budgets, with estimates ranging from 40 to 70 percent [Pressman 87]. Increases in demand by the maintenance phase can only lead to further reduction in new software development capacity because personnel are siphoned off.

Another aspect of the software crisis is the lack of quality. Although quality can be a subjective characteristic, overall system quality usually can be assessed in terms of providing the functionality expected by the customer, meeting customer performance requirements, and freedom from defects. The “internal” quality characteristics of a system can be assessed in terms of the components comprising the system. These features include working as advertised, having acceptable usage of time and space resources, being composable with other components, being understandable by clients and maintainers, and being usable in a possibly different context.

In an attempt to improve quality and productivity, many software engineers have focused on improving the software development process. This approach usually includes the use of computer-aided software engineering (CASE) tools. To help organizations assess their software process maturity, the SEI developed the software maturity questionnaire [Humphrey 87] and the Capability Maturity Model for Software [Paulk 91]. The hope is that improvements in how an organization goes about managing software development will lead to better productivity and to higher quality systems.

Another approach to the quality and productivity problem, similar to but crucially different from the software-process approach, is improving the design and implementation of individual components of a software system. Rather than working on a grand scale, this approach attempts to apply software engineering principles to component design in order to achieve improvements one component at a time. The idea is that small gains in quality and productivity at the component level will accrue to substantial gains over the entire software system.

There is a problem with the latter approach as it has been applied to date: component design is hard and requires skills and disciplines not generally taught to or practiced by software engineers. Software engineering was founded on the principles of procedural abstraction, data abstraction, and encapsulation, among others. But these principles applied alone or haphazardly to the design of a software component do not necessarily lead to the component having the desired properties (e.g., it works as advertised, or it has acceptable

usage of time and space resources). A very detailed discipline based on these principles in conjunction with component-specific and language-specific principles is needed when constructing software components. The purpose of this dissertation is to present such a detailed engineering discipline for constructing software components and to compare it to existing approaches.

1.1 The Thesis

This dissertation defends the following thesis:

There is a programming language-independent, component-type-independent engineering discipline for the construction of high-quality software components. Furthermore, the discipline can be refined to a specific programming language and component-type (such as Ada and abstract data types) and can be observed to lead to the development of locally certifiable high-quality components.

The approach taken in supporting this thesis is to illustrate the utility of our definition of a *software discipline* by using it as a tool for analyzing existing methodologies and guidelines found in the literature. Also, we develop a discipline for designing abstract data types in Ada by refining the component-type-independent and language-independent discipline defined in the next section (Section 1.2). Finally, we illustrate problems found with the most rigorously developed Ada components published in the literature and we compare these components to those constructed using our discipline.

1.2 Definition of a Software Discipline

This section provides our definition of a software discipline. It is based on software components because we view a software system as comprising components, which in turn comprise other components, etc., down to some primitive-level set of components (see Figure 1).

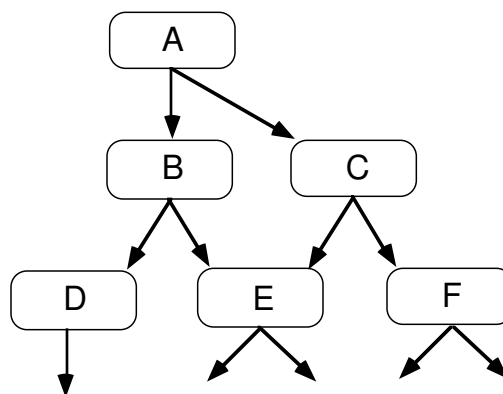


Figure 1 — A Software System as a Collection of Components

One interpretation of the diagram in Figure 1 is that the components are procedures and that an arrow represents a procedure call. Another interpretation might be that the components are modules and an arrow represents the use of one component by another for its implementation. Other interpretations are possible.

A definition for a component-based *software discipline* might be “a rule or system of rules governing the construction of software components.” This is an adaptation of the definition for “discipline” in *Webster’s Ninth New Collegiate Dictionary*. On the surface this definition of a software discipline may sound good, but it leaves too much “between the lines.” For example, it might allow for guidelines such as: “Exploit the features of the Ada language to write general purpose, adaptable code that has the maximum potential for future reuse” [SPC 89, page 174], or “A reusable component should exhibit low coupling and high cohesion” [SofTech 85, page 4-11], or “Packages should implement interfaces to reusable objects at a consistent abstract level” [Hooper 91, page 170]. Objectives such as these are nice, but do not really aid the component designer, nor do they help potential clients evaluate or differentiate among available components.

Our definition of a software discipline not only includes rules (below these are called principles) for governing the construction of software components; it also includes goals, properties, and certification techniques. The meaning of each of these is as follows:

- A discipline must be focused. It must define *goals* for the overall software process and provide good reasons for promoting them. Why? It is unlikely that anyone will follow a discipline for the sake of following a discipline. To obtain a voluntary following, there must be something to achieve and good reason to achieve it.
- It must provide milestones. A discipline must define *properties* that components are required to possess in order for the discipline’s goal to be achieved. Properties are analogous to milestones: they must be achieved in order to reach the ultimate goal.
- It must provide guidance. A discipline must define detailed *principles* that guide the engineer during component design and implementation. By adhering to the principles, the engineer (probably) will construct components that have the required properties. Without the principles, leaving only the goals and properties, we would know where we want to go, but would not know how to get there.
- It must be practical. A discipline must permit *feasible certification* that a component possesses the desired properties. To be feasible means that properties must be locally certifiable, i.e., that it is possible to determine component satisfaction of a property without knowledge of the component’s clients and without knowledge of implementation details of used components. If non-local certification were allowed, then to evaluate a component with respect to the discipline’s properties may very well require reasoning about the entire system. This would be a formidable task even for many small systems, and would undermine the utility of a discipline. Section 1.3.3 illustrates local and non-local certification with an example.

Since software design and implementation is a highly cognitive process, there is little hope at this time for its complete automation (although this does not preclude the use of tools to aid the process). In other words, we do not presume that the idea of a software discipline is going to answer all questions concerning the process of software design and implementation. For example, possible goals include: components that are easier to maintain, higher programmer productivity, etc. Since these goals are typically subjective in nature, choosing the “correct” properties for achieving them is also subjective. However, since a discipline must permit feasible certification, there is an objective and technical

foundation supporting the discipline: the connection between principles and properties. Thus, we introduce the idea of a software discipline as a means of organizing our ideas about software design and implementation by capturing, identifying and separating its subjective and objective aspects, and as a tool for analyzing proposed disciplines and existing guidelines and methodologies. We concentrate here on the objective aspects, to the extent possible.

1.3 A Discipline for Constructing High-Quality Components

This section outlines our discipline for software component construction. First we define an independent-level discipline, i.e., a language-independent and component-type-independent discipline. In Chapter 2, we fix the language to be Ada and the component-type to be abstract data types, and define the discipline more precisely in those terms.

1.3.1 *The Discipline's Goal and Properties*

The discipline's goal is improved programmer productivity and systems that are easier to maintain through the construction and use of high-quality software components. We define a *high-quality* software component as a component that is correct, composable, reusable and understandable. The properties of a high-quality component are:

- **Correctness** — A high-quality component should be correct. As a result, the component must support abstract reasoning about its behavior. Support for abstract reasoning is needed by both the component implementer (for reasoning about component correctness) and by the client programmer (for reasoning about client correctness).
- **Composability** — A high-quality component should be readily composable with other high-quality components. This property involves the mechanics of composing two or more components.
- **Reusability** — A high-quality component should be ready for reuse without source code modification. Adaptations to its behavior should be provided parametrically. Furthermore, a high-quality component should be relatively time-efficient and/or space-efficient in that, given any other component that correctly performs the same function, the high-quality component is not dominated in all dimensions by the performance of the other component.
- **Understandability** — A high-quality component should be understandable relative to the complexity of its intended behavior. Contributing to understandability are consistency and uniformity. High-quality components should exhibit a uniform appearance as a result of consistent application of specific engineering principles.

Building components that have these properties ought to result in components that work as advertised, can be composed with other components, can be understood by potential clients and maintainers, can be used again in a possibly different context given the opportunity, and are relatively efficient. If presented with such a component, we are more likely to believe in its integrity and will be inclined to use it when building a larger system. If the component fails to meet any of these properties, e.g., it does not work as advertised, we would be more inclined to build a new component from scratch. Furthermore, during the

maintenance phase of a high-quality system, programmer productivity can be improved because the programmer is less likely to be debugging incorrect components, and can understand the components comprising the system with less effort [Edwards 90, page 10].

1.3.2 *The Discipline's Principles*

The discipline's principles guide the engineer during component design and implementation. At the independent-level, little can be said concerning composability because it depends on aspects of the language and/or the component-type. However, we can introduce principles that support correctness, reusability and understandability.

Principle 0 — Formally specify each component's abstract behavior in mathematical terms and separate its specification from its implementation. Do not write the specification based on a possible representation of the component.

Support for the properties is as follows:

- **Correctness** — Formal specifications provide the means for the engineer to reason abstractly about the component. Furthermore, formal specifications provide a firm foundation for testing and formal verification of the component.
- **Reusability** — Potential users of the component are able to determine the behavior of the component without reading and/or understanding the component's source code. Additionally, separation of the specification from the implementation facilitates multiple implementations for the same specification, increasing the likelihood that a component exists with the right abstraction and an acceptable implementation. Furthermore, suppose an engineer finds the right *concept* (i.e., the abstraction the component embodies), but none of the existing implementations meet the space/time efficiency requirements. Reusability is still promoted through the reuse of the concept; the engineer has only to provide a new implementation which meets the behavioral requirements of the selected concept.
- **Understandability** — For non-trivial components it is usually easier to determine the behavior of a component by reading its abstract specification, than by reading the code that implements its behavior. Furthermore, separation of the specification from the implementation promotes information hiding. Therefore, the engineer of a client that uses a component is not bothered or confused by the component's irrelevant implementation details.

1.3.3 *Requiring the Discipline to Be Practical*

This section discusses the importance of requiring feasible certification of the properties. We will illustrate this through an example. Suppose we need an operation to reverse the contents of a (FIFO) queue. One possible algorithm might comprise two loops. The first

loop's body dequeues an item from the queue and then pushes it onto a stack until the queue becomes empty. The second loop's body pops an item from the stack and enqueues it onto the original queue until the stack becomes empty. Assuming the stack is initially empty, the contents of the queue should end up in reverse order. Figure 2 illustrates a possible set of components for constructing Reverse Queue:

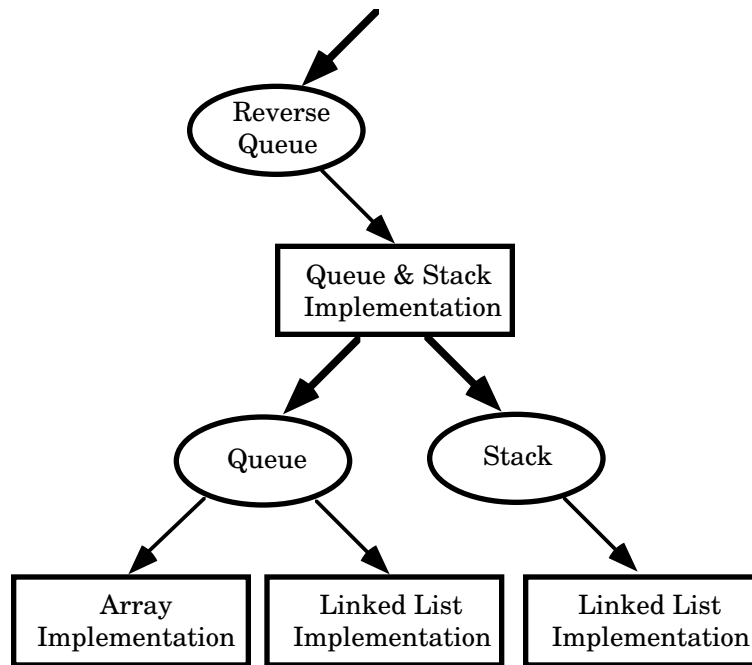


Figure 2 — Reverse Queue

In Figure 2, ovals denote concepts and rectangles denote implementations. A thin arrow from an oval to a rectangle indicates that the concept is realized by the implementation. A thick arrow from a rectangle to an oval indicates that the implementation uses the concept. In this situation notice that two arrows emanate from the Queue concept. This means that there are two separate implementations available for the Queue concept (one based on an array representation, the other based on a linked list representation), but only one of these implementations is needed and used in this case.

Suppose we want to certify that the correctness property holds for Reverse Queue, i.e., to certify that its implementation meets its specification. (We also would want to certify composability, reusability and understandability, but for this example, we concentrate on correctness.) The information required for the certification is contained within the area bounded by the dashed line in Figure 3.

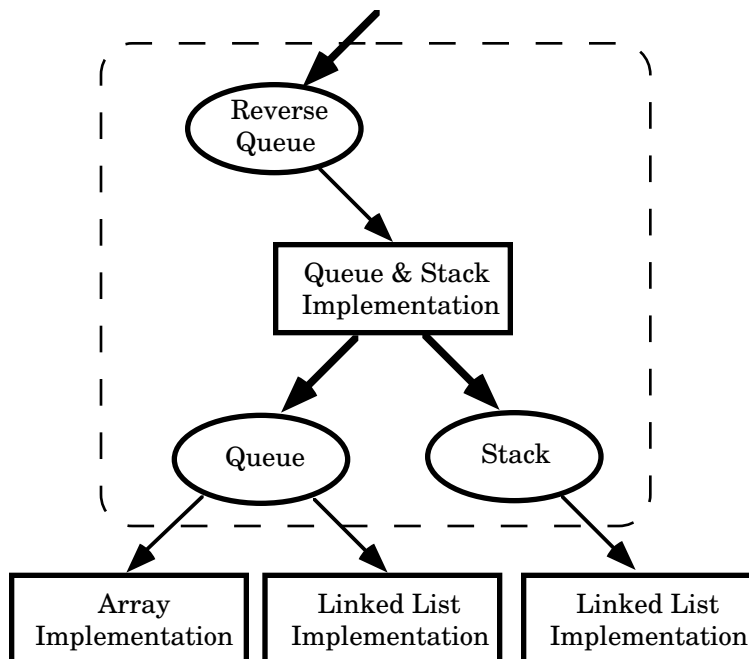


Figure 3 — Area of Reasoning for Certifying Reverse Queue

Notice that neither Stack’s implementation nor Queue’s implementation is included. This is a direct result of requiring local certification of correctness. That is, to certify correctness of an implementation (e.g., Queue & Stack Implementation) requires only the implementation’s specification (Reverse Queue) and the specification of the concepts it uses (Queue and Stack). It does not depend on any client of the higher-level concept (i.e., any client of Reverse Queue), nor on the implementation of any concept that it uses.

Local certification of correctness requires abstraction barriers with absolutely no implementation leaks. This can be achieved through the careful use of abstraction, information hiding, and encapsulation. The principles introduced in Chapter 2 show how this can be done in Ada.

But building components with leak-proof abstractions is not easy. On the contrary, building components with leaky abstractions turns out to be quite easy, as illustrated by examples in Section 2.1. The result of allowing implementation leaks is illustrated by the expansion of the bounded area, shown in Figure 4.

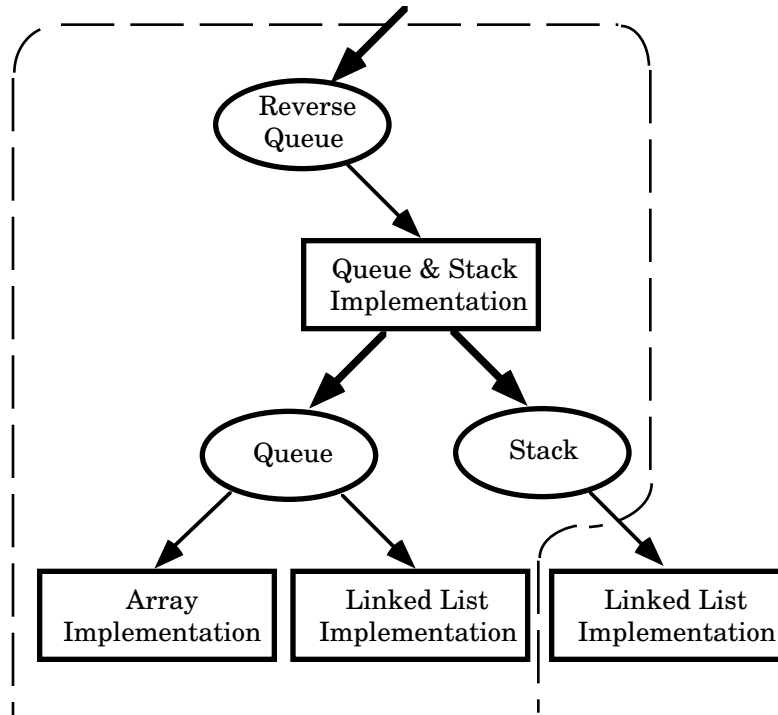


Figure 4 — Area of Reasoning Because of a Leaky Abstraction

Figure 4 shows that certification of correctness for Reverse Queue now includes reasoning about Queue’s implementation, in addition to all the items reasoned about before. Our job has become harder because we are reasoning about more things and because our reasoning is not limited to a used concept’s abstract specification, but now includes its implementation.

Allowing components to have leaky abstractions not only makes the certification job harder, it has potential to make it completely infeasible. To understand this, let’s assume that we allow leaky abstractions. In order to confidently reason about a component used by a client, the client programmer must first determine if there are any implementation leaks. The only way to do this is by examining and understanding the used component’s representation and its implementation. Actually, recognizing a leaky abstraction by looking only at its implementation is not possible; see the example in Section 2.1.6. But even if it were, since systems comprise components which in turn comprise components and so on, generally the implementation that the client programmer is examining uses other components. What is to stop their abstractions from being leaky? Nothing, if we allow it. Therefore, a client programmer who wants to confidently reason about the client may have to understand and reason about the entire hierarchy of components directly and indirectly used by the client program. This is not feasible for even modest sized systems because of the high level of effort required to find and understand all related components.

Moreover, from a practitioner’s standpoint, future maintainers of this type of system will be forced to reason about these *non-local* components in order to have confidence in their work. This is not feasible either, but it regularly happens with at least two possible results: 1) the maintainer does not fully understand the system under consideration, but makes changes anyway hoping that testing will point to the other parts of the system requiring

attention; or 2) the maintainer spends an inordinate amount of time grasping the interlocking details of the system. A worst case scenario is a combination of both!

The bottom line is this: In the short term, it is tempting to not require feasible certification because it is not easy to build components that permit it. However, in the long term we do ourselves a disservice if we do not require feasible certification of the properties, especially when our software systems reach the maintenance phase. The discipline proposed in this dissertation requires feasible certification, and Chapter 2 introduces principles that lead to the construction of Ada components that permit it.

1.3.4 Related Work

Examination of related work appears throughout the dissertation. In Chapter 2 we reference Ada components from the literature, paying attention to their specific design details. In Chapter 5 we examine published Ada component libraries and discuss their (lack of) adherence to the principles introduced in Chapter 2. In this section we examine related frameworks, methodologies and guidelines proposed by others in the literature, using our definition of a software discipline as a means of comparing these works.

We have chosen a variety of different works to illustrate different approaches to defining a framework, methodology or set of guidelines. We by no means can include all frameworks and methodologies that have been published. For example, some of the excluded works were primarily concerned with software metrics or testing, but not necessarily software design. These works often describe software properties but do not introduce specific guidelines for design (e.g., Boehm's *Characteristics of Software Quality* [Boehm 78]). In general, what we have found is that many methodologies have implicitly defined some parts of a discipline, e.g., some have goals and guidelines (i.e., principles), while others have goals and characteristics (i.e., properties). None has all four required parts to constitute a discipline.

We start with Booch's *Software Components With Ada: Structures, Tools and Subsystems* [Booch 87]. Here we discuss his approach to designing software components in Ada, and in Chapter 2 we make specific references to his components.

Although Booch does not explicitly state his goal, it is obvious that it is to build reusable software components. He outlines characteristics of software components, saying that they should be maintainable, efficient, reliable and understandable (page 34). He also outlines some general guidelines, noting that a software component should be built so that it is sufficient, complete and primitive (page 35). Booch also identifies some specific rules for constructing Ada components, and gives persuasive justification for these rules. However, he willingly violates some of his best rules. For example, on page 55 he suggests the use of Ada's limited private type, but on page 79 violates the rule with his linked-list package. Finally, there is no mention of how to certify that the resulting components are maintainable, efficient, reliable and understandable. In fact, by following his own rules, Booch constructs components whose satisfaction of his properties is questionable.

St. Dennis' *A Guidebook for Writing Reusable Source Code in Ada* [St. Dennis 86] comes quite close to implicitly defining a discipline. He identifies a goal — reusable Ada source code — and identifies 15 characteristics of reusable software. For example, he writes on page 14 that a “component can be used without change or with only minor modifications.” He explicitly identifies guidelines to be followed when constructing Ada components, e.g., on page 83, “G12-1: Use generic program units (i.e., packages and subprograms) to

effectively parameterize reusable software parts.” However, like Booch, St. Dennis does not mention how to certify that the resulting components actually satisfy the desirable characteristics. Finally, unlike Booch, St. Dennis provides very few complete examples for us to evaluate.

The Software Productivity Consortium’s (SPC’s) *Ada Quality and Style: Guidelines for Professional Programmers* [SPC 89] has a goal, properties and guidelines. The goal, found on page one, is “to help the computer professional produce better Ada programs.” The properties (e.g., code clarity, reliability, portability, etc.) are introduced throughout Chapter One. The rest of the book introduces guidelines which often are too general to be considered principles. For example, Section 5.4.3, beginning on page 76, discusses dynamic data structures, and offers the following guideline: “Differentiate between static and dynamic data. Use dynamically allocated objects with caution.” Additionally, there is no mention of certification techniques or the need for certification.

The apparent goal and property of SofTech’s *Ada Reusability Guidelines* [SofTech 85] are reusable Ada software and reusability, respectively. Most of the guidelines offered are too general to be considered as principles. There is no mention of certification techniques or the need for certification.

The last work that we examine is Meyer’s *Object-oriented Software Construction* [Meyer 88]. Meyer does not come as close as St. Dennis to implicitly defining a discipline. Meyer seems to have a goal, he identifies some properties and some general principles, but does not really define specific guidelines or principles for component construction. Meyer is similar to Booch, in that he identifies various rules for building components throughout the text. Finally, there is no mention of how to certify that the resulting components actually satisfy his properties.

In summary, from the literature that we have examined, others have not explicitly set out to define a discipline. However, the resulting “framework” or “methodology” has sometimes come close to implicitly doing so. The one ingredient that all lack is a requirement for feasible certification. Indeed, there is no mention of any kind of certification, feasible or not. This is unfortunate because, as Section 1.3.3 illustrates, feasible certification is a crucial requirement for an attack on software productivity and quality concerns in practical-size systems.

1.4 Outline of Dissertation

The dissertation is organized as follows. Chapter 2 introduces specific principles for constructing components in Ada. Chapter 3 introduces principles for adding additional capabilities to a component, for creating components that support testing and debugging, and for “partially instantiating” components. Chapter 4 shows how to bootstrap the discipline from “raw” Ada (i.e., how to encapsulate some of Ada’s built-in types and type constructors), and Chapter 5 discusses possible future work and conclusions drawn from the work so far.