PAPER • OPEN ACCESS

Axiomatic Software Engineering

Axiomatic Software Engineering

J T Foley[1,2], **M Kyas**[1]

[1]Reykjavik University, Menntavegur 1, Reykjavik 102, Iceland
[2]Massachusetts Institute of Technology, 77 Massachusetts Ave, Cambridge MA 02139, USA

E-mail: `foley AT ru.is`

Abstract. Axiomatic Design and Complexity Theory have primarily focused on refining the process of an abstracted need into a physical solution even under challenging conditions. The concepts have high applicability as guidelines for non-physical design, such as those found in programming and its applied field of software engineering. Software engineering came out of the software crisis of the 1960s: its goal is to deliver safe and secure software that meets its specifications on time and within its budget. The product of software engineering is a program. The program is written in a programming language that looked suitable for the problem at hand. For some time, object-oriented technology (OOT) looked popular, because it promised to address information hiding, composability, and extensibility of programs. In hindsight, OOT failed to deliver on its promises. We analyze how Axiomatic Design leads to solutions that are proposed by current programming technologies. The authors believe that different paradigms such as cohesion also need to be included.

1. Introduction

Software used to be found primarily in highly technical areas: nuclear weapon design, CNC milling machines, and telephone billing systems. Today, software controls every aspect of our daily life: cars, washing machines, refrigerators, coffee machines, light-bulbs, etc. At the same time, the complexity of software has increased. The digital autopilot for the Apollo lunar lander required about 2,000 lines of assembly code [1], while it has been estimated that there are about 1.5 million lines of code in the onboard command and control computers on the International Space Station [2]. Greater needs have arisen for producing software in terms of quantity while maintaining quality. Our reliance on software has not necessarily resulted in the desired reliability evidenced by Microsoft Windows' "Blue Screen of Death" showing up in unexpected areas.

The increased size of software systems created by teams of hundreds of programmers gave rise to specializations in software companies. The teams consist of managers, secretaries, programmers, testers, infrastructure engineers, build engineers, domain experts, and designers. Consequently, the focus of programming has changed from designing and implementing algorithms to constructing software systems bottom-up using an ecosystem of software components. Today's programmers must be skilled in the elements of *software engineering* to fit into such large-scale projects.

The authors focus on the role of Axiomatic Design in modern software engineering. They illustrate the problem and history of software engineering. Then, they argue that object-oriented programming is not the best fit for Axiomatic Design. Instead, they propose modular programming methods which have been rediscovered for modern languages like Go and Rust.

1.1. A Very Brief History of Software Engineering

The term *software engineering* appeared in the 1960s. It was invented to distinguish the process of creating software-intensive systems from computer science. Most textbooks credit the 1968 NATO on Software Engineering conference in Garmisch-Partenkirchen as the origin of the term [3], or at last, it popularized it. However, the name was in use before the conference. Anthony Oettinger uses the term in a letter to the Communications of the ACM in 1966 [4]. Early written uses of the term appear in the June 1965 issue of Computers and Automation [5]. Folklore tells that Margaret Hamilton coined the term in 1963 or 1964 [6]. She was the lead developer for Skylab and Apollo while working at Draper Lab.

The 1968 NATO conference of software engineering addressed the *software crisis:* software projects were delivered late, software was not working to specification, and budgets were frequently overrun. Participants of the conference proposed pioneering solutions. M.D. McIlroy introduced the idea of “mass-produced software components”. He likened the production of software to mechanical engineering [7].

Software modules were proposed as one such unit of reuse. In software engineering, a module is a unit of software that encapsulates code and data to implement a particular functionality, provides an interface that lets clients uniformly access its functionality, can work in any context that interacts through the interface, and is usually deployed as a single unit. Examples of such modules are system libraries.

Parnas [8] argues that a module should not represent a part of the program execution, like a block in a flow chart. Instead, a module should encapsulate design decisions. If done correctly, the code that relies on a module is robust to changes in the module’s internals. This property, called *information hiding,* enables programmers to work on the functionality of their module without depending on the programmers implementing the surrounding modules. The interface serves as a contract on which each programmer relies when using other modules while ensuring that their implementation conforms to their module’s interface.

Barbara Liskov developed the programming language CLU, which supports information hiding as part of the language [9]. CLU defines objects as instances of data types that can only be manipulated through their interface. This language became a key component for teaching MIT’s 6.170 Software Engineering course until 1995 when it transitioned to C++.

Early approaches to software development focused on a top-down process. Herbert Benington describes what became known as the Waterfall model in 1956. The process goes through six phases in almost sequential order: Software Requirement Analysis, Preliminary or Abstract Design, Detailed Design, Coding and Unit Testing, Integration, and Testing. This process is then followed by Software Maintenance, in which defects found during use are corrected, and the system is changed to meet evolving requirements. Benington clarified in 1983 that this step-by-step process was defined to describe the specializations of programming tasks and is not necessarily followed in this order [10]. Winston Royce describes this process as a straw-man argument and rejects its as highly risky [11]. He suggests that such a process should always be executed at least twice. It still was standardized by the Department of Defense [12] until it was superseded in 1994 [13], which marks a departure from the strict Waterfall model.

Indeed, the attempt of clarifying all the functional requirements in advance and decomposing the system into modules and tasks resulted in one of software engineering most spectacular failures: Fred Brooks “The Mythical Man-Month”, originally published in 1975, attributed the failure of the OS/360 system to this waterfall model [14]. Understanding this failure is important enough that it is required reading as part of MIT’s 6.033 Operating Systems course. The manuals of OS/360 describe the user-visible, expected behavior of the system. Brooks describes its volume as a “six-foot shelf of manuals” [14, p. 134]. His lessons include: the functional requirements of modern software systems are complex and never fully understood. The functional requirements of system features may interact in undesirable ways. Such *feature interactions* occur when a feature interferes with the operation of another feature [15]. For example, phone subscriptions may support *call forwarding* and *call waiting*. If both features are enabled at the same time while the user is in a phone call, it is not clear whether a second call may wait or has to be forwarded.

The shortcoming of the Waterfall model for software development leads to the development of iterative processes. The main problem of the Waterfall model is that the later one is in the process, the more expensive it becomes to address mistakes in earlier phases. If a mistake in the requirements is detected during the testing phase, it may result in a complete redesign and a new implementation of the complete system.

Software engineers were commonly not successful in capturing all the requirements of a system correctly. Fred Brooks, for example, suggested developing one software prototype to learn the requirements and how to develop solutions. This prototype is then to be discarded, and a new system shall be developed from scratch. Of course, there is an inherent danger that this replacement may fall victim to the "Second System Effect" and yet another failure [14].

Most software development processes are using an incremental and iterative design (IID) model. According to Larman and Craig, Walter Shewhart laid the foundations for IID by developing "plan-do-study-act" (PDSA) cycles for quality improvement in the 1930s [16]. The first use in software engineering was Project Mercury in the 1960s. The method got popular during the 1980s after the 1970s observed the weaknesses of the strict waterfall model. Since then, IID has been adopted as the dominant design method.

The main idea of IID can be grouped into four phases [17]. *Inception* identifies project scope, requirements (functional and non-functional), and risks at a high level but in enough detail that work can be estimated. *Elaboration* delivers a functional architecture that mitigates the top risks and fulfills the non-functional requirements. *Construction* incrementally fills in the architecture with production-ready code produced from analysis, design, implementation, and testing of the functional requirements. *Transition* delivers the system into the production operating environment. Each phase can be iterated multiple times.

An iteration starts with planning. A feature of the system is selected, and its requirements are formulated. The developers analyze the requirements and design the feature. The feature is implemented and tested. The test results are analyzed, and plans are updated. Already implemented features may undergo a redesign as a consequence of what the team learned from the iteration.

Proponents of object-oriented technology hoped that their programming methods (see below) support encapsulation and modularization as part of the design and implementation process. Objects are supposed to be units of composition. They can be reused and modified by inheritance. Late binding realizes polymorphism by overriding parts of a super-class implementation. Finally, it allows the creation of large systems by composing classes in a bottom-up process. As the authors show below, these promises have not been fulfilled.

This realization is the starting point of Agile Software Development. It is defined by the "Manifesto for Agile Software Development" [18]:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> Individuals and interactions over processes and tools
> Working software over comprehensive documentation
> Customer collaboration over contract negotiation
> Responding to change over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

1.2. A History of Object-Oriented Technology

Sutherland pioneered initial ideas of object-oriented programming at MIT. Nygaard and Dahl designed the first object-oriented programming language SIMULA in 1962, which they had been developing since the 1950s [19]. They developed SIMULA to make the programming of discrete event systems easier. The name stands for SIMulation LAnguage.

Influenced by LISP and SIMULA, Alan Kay developed the language Smalltalk and created the name "object-oriented" in 1972 [20]. The concept of a class became a fashionable programming abstraction and led to the creation of C++ [21] and Objective-C [22].

Inspired by the work of Kay and Stroustrup, software engineers placed high hope in object orientation as a tool to address the programming crisis. Software engineering methods based on Object-Oriented programming were developed by Booch [23] and Meyer [24, 25]. This work resulted in the Unified Modeling Language (UML), an amalgamation of Booch [26], Jacobson [27] and Rumbaugh's [28] methods. UML is still a dominant software engineering tool. Agile development methods in current practice have displaced the accompanying Rational Unified Process.

1.3. Axiomatic Design

Nam Pyo Suh developed Axiomatic Design in the 1980s to provide a systematic approach to designing machines, particularly manufacturing equipment [29]. The AD process is relatively straightforward but requires a slight departure from many iterative-focused design processes. Designers develop lists of items in the four domains: Customer, Function, Physical, and Process. The items represent Customer Needs (CNs) mapping to Functional Requirements (FRs) to Design Parameters (DPs) to Process Variables (PVs). These mappings are analyzed in the form of design matrices (a Cartesian product of the elements in the two adjacent domains), or design decomposition trees [30, 31]. The goal of this representation is to show where coupling between elements in a given list are occurring in order to choose designs that fulfill AD's axioms.

Independence Axiom: Maintain the independence of the functional requirements[32, p. 16]

Information Axiom: Minimize the information content I of the design where $I = -\log_2 p_i$ and p_i is the probability of meeting FR$_i$[32, p. 39]

The Independence axiom indicates that in the matrix representation, there should only be diagonal elements with coupling i.e. a one-to-one mapping. The Information axiom tells the designer to choose implementations that minimize the chance of failure to meet a mapped FR.

The authors would initially adjust these concepts in the context of software to be:

Independence Axiom: Maintain the independence of the functional requirements

Information Axiom: Enforce encapsulation, maintain consistent internal state (representation invariants are often employed here), reduce coupling

Software engineers call an FR the definition of a function that the software system or one of its components has to implement. The function is typically specified by relating valid inputs to outputs.

2. What AD has previously said about software engineering

Suh's primary investigation into the topic of software design appears as chapter 5 in the 2001 book "Axiomatic Design: Advances and Applications"[32]. He details an Axiomatic Design process specifically for software [32, pp. 245–248] based upon [33]:

(i) Define FRs of the Software System
(ii) Mapping between the Domains and Independence of Software Functions
(iii) Information Content for Software Systems to Select the Best Design

(iv) Decomposition of FR, DP, PV, and Software Modules
(v) Object-Oriented Programming and Modules for Leaves
(vi) Design Matrix and Module-Junction Diagrams
(vii) Flow Chart Representation of Software System Architecture
(viii) System Control Command

Much of the details in this chapter focus on the implementation of software using this process that greatly resembles the "waterfall method" previously described where the requirements have been set at the beginning. Of note, AD's approach explicitly acknowledges that a designer will need to revisit previous phases in the form of a zig-zag between domains as part of the decomposition. The applications of this approach in the chapter are from related papers of that era. This includes: library index[33], calculator for the geometry of ribs in an injection molded part[34], and optimization software for CRT geometry optimization[35], Acclaro software for supporting AD development[36],

Acclaro is the only known software product designed using AD still in existence as of 2021. This product has shifted focus from AD to Design for Six Sigma, which is a popular production design methodology in its own right.

Since this publication, there have been additional AD-related publications on AD used in software, primarily focused on database design or development of an application. As an example, Pacifici et al.[37] discussed software deployment for remote patient monitoring software.

3. A critique of the AD approach

To some extent, the approach that Suh takes to the development of software is based on assumptions that software engineering considers to be faulty. He states, "Its framework is based on the idea that software must be designed to satisfy a set of functional requirements and constraints before commencing coding activities."[32, pp. 241] and "It also provides a methodology for design that includes the idea of mapping, decomposition, and the creation of leaves". (The leaves mentioned in this statement are the endpoints of decomposition in contrast to a "branch".) Empirical evidence demonstrates that it is not possible to capture the correct functional requirements and constraints before coding activities start. The risk is that if functional requirements and decomposition are performed before any coding activities, the requirements are not satisfiable, resulting in a redesign of the system.

A second observation is that the definition of software on [32, pp. 241] does not capture what modern software is doing. It is the definition of "algorithm," which is an integral part of software systems, but does not capture software. Instead, we propose to base the definition of software on Harel and Pnueli's definition of reactive systems [38]:

> Software is a set of instructions and data that tell a computer system what to do. The computer system is maintaining ongoing interaction with its environment, without necessarily producing a final result or terminating.

This lack of termination is important even in the simplest case of a small control loop like a PID controller which should *never* terminate in proper usage. Buttazzo, for example, shows that a real-time system can become infeasible if executed on a faster computer than it was designed for![39].

3.1. OOT increases coupling and decreases cohesion

One of the takeaways from Suh's chapter on software engineering [32, pp. 245–248] is that you should use object-oriented technology (OOT) to reduce coupling.

In software engineering, *coupling* is the measure of the degree of interdependence of modules. Tight coupling means that code in one component has intimate knowledge of the internals of other components. Changes in one component ripple through the system and change the behavior of other components. Loose coupling restricts knowledge to published interfaces. Changes in a component do not ripple through the system as long as the interface is maintained.

According to Stevens et al. [40], *cohesion* is a measure of how closely related the responsibilities of data and code in a component are to each other. High cohesion means that unrelated responsibilities are implemented in different components. The responsibilities have to be "glued" together explicitly. The mantra is, "increase cohesion within all components." Changes for one feature should not affect unrelated features. For example, a customer database maintains names and email addresses. Should a customer have a send-email method to send emails to the customer? To achieve high cohesion, we should implement sending emails in a separate component. The customer is glued into it; spammer.send_email(customer.get_email(), text()). AD's Independence Axiom follows the same mindset: changing a single Functional Requirement (send email) should only affect one implementation Design Parameter (customer's email field).

Object-oriented technologies defining mechanisms tend to increase coupling and decrease cohesion. Object-oriented programming languages can be characterized by the following features:

late binding: the system determines at run-time what code it executes.

self-reference: objects in the system can call their code.

open recursion: code in a class can execute code defined in a subclass.

Information hiding is violated and coupling is increased by these features. By looking at a piece of code, we cannot determine if it gets executed or what code it will execute.

Further investigation into such advice reveals a contradiction in the way OOT is often implemented using an inheritance model between classes. The problem is that inheritance introduces a strong dependency on the behavior of the super-classes and the sub-classes. AD and similar methods focus on decoupling the interactions between all the elements of the system. The designer wants to clearly understand the interactions and the interactions be as clear and as little as possible. In OOT inheritance, when you take a class and derive a subclass, whatever happens in the super-class affects the behavior of all the sub-classes. In short, to understand the bottom of the hierarchy of inheritance, you must understand everything above it. This can quickly lead to mental overload and frustration which strongly resembles the elements of Imaginary Complexity.

Another issue that speaks against reuse in OOT is that objects depend on an environment of other objects to do their work. Joe Armstrong describes it as follows [41, p. 213]:

> I think the lack of reusability comes in object-oriented languages, not in functional languages. Because the problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

Older programming languages such a C did not include an inheritance model, causing many programmers to create conceptual equivalents in terms of how functions were named to indicate their relationship. Modern languages wish to enforce a level of modularity, in many ways implementing AD's Independence Axiom, but without creating the coupling problem of inheritance. For example, the robustness-optimized language Rust doesn't have inheritance, instead choosing to only incorporate OOT's method calls. Rust eschews inheritance for "traits". Traits define constraints on types and group methods that can be applied to all types that satisfy the constraints. Static type safety is achieved through a template mechanism with trait bounds.

As we will discuss in the next section, constraints in another sense are an important element in design, especially in the ephemeral world of software.

```
class BoundedBuffer {
    constexpr static size_t N = 64;
    std::mutex lock;
    std::condition_variable not_full,
        not_empty;
    std::array<int, N> buffer;
    size_t capacity, front, rear;
public:
    BoundedBuffer() : lock(),
        not_full(), not_empty(),
        buffer(), capacity(N),
        front(0), rear(0) { }
    void put(const T&);
    int get(void);
};

void BoundedBuffer::put(const int& value)
{
    std::unique_lock<std::mutex>
        lck(this->lock);
    while (this->capacity == 0)
        this->not_full.wait(lck);
    this->buffer[this->front] = value;
    this->capacity--;
    this->front = (this->front + 1) % N;
    this->not_empty.notify_one();
}

int BoundedBuffer::get(void)
{
    std::unique_lock<std::mutex>
        lck(this->lock);
    while (this->capacity > N-1)
        this->not_empty.wait(lck);
    int value = this->buffer[this->rear];
    this->capacity++;
    this->rear = (this->rear + 1) % N;
    this->not_full.notify_one();
    return value;
}
```

Figure 1: A concurrent bounded buffer in C++

3.2. Multi-threaded Programming Challenges

An instructive example of the issues that are inherent to object-oriented technology is the fragile base-class problem [42]. The problem arises in multi-threaded programming. The fragile base class problem states that inheriting code from a super-class is often only possible if other, seemingly unrelated methods, need to be redefined. Consider the implementation of a bounded buffer in C++ below.

A programmer needs a bounded buffer with a method getTwo() that waits until at least two elements are available in the buffer and returns two consecutive ones. To implement this, they derive a subclass BoundedBufferTwo that inherits the buffer and the methods put and get from its super-class BoundedBuffer. Such an implementation may appear natural, but this particular implementation exhibits an anomaly. If a process waits in get() and another one waits in getTwo(), the system may block after a call to put(). Input, only one process is notified by the presence of a new element. It may be delivered to the process that waits in getTwo(). This process cannot proceed if only one element is in the buffer. The process that waits in get() never gets notified. To correct this issue, the programmers have to override the put() method to replace the statement **this**->not_empty.notify_one() with **this**->not_empty.notify_all().

To avoid this problem, the programmer needs to conceive all possible reuse of the code. This is at odds with efficiency constraints. In our example, calling **this**->not_empty.notify_one() is much more efficient than calling **this**->not_empty.notify_all().

4. Robustness in Programming Languages

Software Engineering is focused on the premise that it is possible to create reliable software that meets specified requirements on time and with the provided resources. For the sake of our study, it is important to understand that we look at the process of software construction. Software can become unreliable by introducing errors into the system by correcting defects in classes that result in new errors in other classes due to coupling. It can become unreliable by errors in memory management and unsafe memory access (dereferencing null pointers, use after

```
class BoundedBufferTwo : BoundedBuffer {
public:
    void getTwo(int*);
};

void BoundedBufferTwo::getTwo(int* values)
{
    std::unique_lock<std::mutex> lck(this->lock);
    while (this->capacity > N-2)
        this->not_empty.wait(lck);
    values[0] = this->buffer[this->rear+0];
    values[1] = this->buffer[(this->rear+1)%N];
    this->rear = (this->rear + 2) % N;
    this->capacity += 2;
    this->not_full.notify_all();
}
```

Figure 2: Subclass of a bounded buffer

free). The Information Axiom is relevant to this field because higher information content in a program is by definition unable to meet the previous statement.

Memory management in its various forms is a common challenge embedded software engineering. It consists of describing, constructing, transferring, and deleting the memory structures of interest. Doing this consistently and reliably is key to the software itself being reliable. Memory management can be done by a garbage collector that keeps track of when objects (and their associate memory) are no longer in use. This garbage collector incurs processing overhead and creates a non-deterministic execution-time behavior. For time-critical systems, failure to meet the timing requirements reliably would be considered Information Content.

For manually allocated memory, the problem can be even more structural. Careful choice of constructing and deleting can result in a system that runs reliably, but the moment any changes are made to that code in terms of sequence or function, Information Content can arrive in the form of double-allocations, free after use, memory leaks, etc. The program may also be vulnerable to malicious attacks in the form of buffer overflows. It is noted that due to these failures arising from memory issues that map to Information Content that Axiomatic Design would also provide insight into how one might correct it.

The paradigm employed to manage these issues begins with type-checking. Type checking verifies that no statement of the programming language operates on data leads to a run-time error or undefined behavior. Undefined behavior is behavior that is not defined by the programming language but that emerges from how the program is executed. It is not defined what the program does and it may do something undesired.

A type system assumes that all data has a type assigned to it and that every operation has declared input and output types. An operation can be executed safely if the input data conforms to the declared types. Then, the operation is guaranteed to produce a value of the output type.

In a static type system, the compiler verifies that the program never operates on data of the wrong type. If the programmer writes an operation in that the types do not conform, the compiler rejects the program. Dynamic type systems instead perform the checks at execution time, aborting execution if the check fails. Static type systems allow integrated development environments to inform the programmer about what types of data are going to be handed to operations and consequently by choosing an output type and provide information to the next function involved.

Static type systems are considered limiting in terms of expressiveness: programmers often cheat around the type system by casting data to types they want later. Many systems try to perform static type analysis to assist in debugging these type issues, but Rice's Theorem[43] indicates that you can only do a proper analysis of trivial things. Many desirable properties of type systems are not trivial. For example, one can decide the type of an array or if an array index is in its bounds, but never both at the same time. Pierce gives a comprehensive account of types and limitations of type checking [44].

Memory management in the reliability-focused Rust language uses a *linear type system* to track memory. Memory is allocated and initialized at the time of the variable declaration. It is de-allocated when it goes out of scope. This approach, in addition to having immutable variables and objects by default, allows for relatively easy analysis of memory and types. The Rust compiler employs a mechanism called a "borrow checker" to enforce proper memory management; programs that fail to follow the rules fail to compile. In the terms of the Information Axiom, the compiler is reducing the information content of the complete system by ensuring that only compliant code can be executed.

5. Lack of constraints is a problem

Software in its purest form is only constrained by theoretical limitations, unlike designs that are mechanical engineering. In many cases, Computer Science curricula focus on elegant and fast algorithms to solve abstract problems. If processing power or memory becomes something to consider, you can dump it into the cloud and spend more money! This concept is within the purview of the public due to crypto-currency correlating processing energy costs with an income stream. Conversely, edge computing, IoT, and low-power sensing have highlighted that money cannot always solve the processing problem.

Embedded programmers are constrained by the limitations of the target platform. In time-critical systems, the deadlines must be met. In mass-produced, systems, slow and memory-constrained computers are economically beneficial. Also, the domain in which the program executes imposes constraints on what the software does. Sensors are limited by their range. Actuators are limiting by the properties of the domain. These constraints make the design and testing of software systems much easier.

In the even more concrete world of mechatronics, reality quickly creates constraints. A drone software engineer might not know what the physical parameters represent are and is just asked to make it fly. The meaning of those values is from the domain expert: a pilot or avionics engineer. The mechatronics and mechanical engineer might do the AD decomposition, then hand the slots to be filled in by the software guy. This results in occasional mismatches between what the software commands and what is physically possible. One of the authors was tasked with improving the odometry on a larger land-based robot using a new sensor. In the testing process, he accidentally added a 0 to the end of the rotation rate speed, which resulted in the robot spinning at an extremely high rate, scraping his shin, and denting his filing cabinet in an embarrassing episode. In the internal pure model of the software interfacing with the hardware, there is no significant difference commanding a robot to turn at $0.1\,\mathrm{rad\,s^{-1}}$ or $100\,\mathrm{rad\,s^{-1}}$. In the physical world, it becomes painfully clear that this is not true. (In a similar incident that happened on the same day, the author's colleagues input the incorrect forward velocity on a similar robot which caused it to jump out of the test stand and destroy an expensive laptop.) A concrete example of where poorly design software resulted in injury and death is the case of the Therac-25 radiation therapy machine. In this case, a race condition in the user interface and the removal of hardware-based safety interlocks resulted in a very dangerous environment [45].

How can useful constraints be created? Gualtieri et al. [46] have applied AD to the design

```
type Vehicle interface {
  func Drive()
}

type Car struct {}

func (car *Car) Drive() {
    // drive the car
}
```

```
type Truck struct {
      car *Car
}

func (truck *Truck) Drive() {
      // delegate the call to the car
      truck.car.Drive()
}
```

Figure 3: Use of delegation to implement code reuse

of collaborative Human-Robot workspaces, particularly focusing on the relevant European workplace standards. This helps create a physically safe environment, but what about a safe environment on the software side? Pedagogically, instructors need to use a computing system that imposes constraints like a Raspberry Pi. This has limited processing power, memory, and IO. Such constraints force the designer to start thinking about resource management earlier and these constraints help reduce the solution space. This is particularly relevant because low-cost microcontrollers such as the Intel 8051, TI MSP430, and ARM7 are still extremely common in modern appliances. It must be noted that making changes later is extremely expensive, an observation that AD often reminds us of.

Another thought is that well-rounded multi-disciplinary education for programmers might provide a better understanding of real-world constraints. Students have trouble with inertia the first time they interface with software motors. They think it is sufficient to turn the motor to full then turn it off to stop. Once they have made this mistake a few times, possibly painfully or with a great deal of smoke, they are highly motivated to improve their software model of the motor system.

6. Modern programming language approaches

Liskov's previously mentioned CLU [9] avoids the complications of object-oriented programming. The language supports information hiding and encapsulation of data. It allows access to the data only through well-defined interfaces (MIT Students in the software engineering course quickly learned to dread the warning "interfaces changed" while compiling.) But the language does not support inheritance. It also does not support open recursion. Operations executed on the current object can be resolved at compile time.

More recent programming languages avoid object orientation for the same reason.

6.1. The Go Programming Language: interfaces and delegation

The Go programming language encapsulates groups of operations into packages. Data types inside packages can be specified through interfaces. An interface declares a set of operations that operate on the data type. Any object that supports all the operations specified in an interface can be used as an instance of this interface. This differs from C++ or Java, where a class needs to specify that it implements an interface.

The Go programming language does not support inheritance. Code reuse can be achieved through delegation as shown in Figure 3. In the code example below we define an interface for Vehicles. We define a data **type** Car that implements the function Drive(). We define a data **type** Truck that implements the function Drive() through delegating the calls to a car object.

6.2. Rust: traits
As previously mentioned, the Rust language deals with object organization differently from common OO languages such as Java and C++. The language incorporates structures and methods. Methods operate on objects of the structure types. Rust uses "traits" to define shared behavior and organize common methods across structures. Klabnik and Nichols [47, Chapter 10.2] describe it thusly:

> Different types share the same behavior if we can call the same methods on all of those types. Trait definitions are a way to group method signatures together to define a set of behaviors necessary to accomplish some purpose.

Traits are not types. Different structured can implement traits, and programmers can thus assume that they can use objects that share a trait the same way.

Let us consider the simple example of a window manager system requiring elements drawing to the screen. The programmer can specify a Draw trait to express the signature of the draw method. The elements implement the Draw trait.

```
pub trait Draw {
  fn draw(&self, scr: &Screen);
}
```

To ensure that the elements implement the Draw trait, they can specify a generic function that only accepts types that implement the Draw trait:

```
fn paint<Elem: Draw>(t: Elem) {
  t.draw(screen)
}
```

The compiler produces a compile error if t does not belong to a trait that implements the Draw trait. The programmers can rely on any such object to have a draw() method.

This mechanism is significantly safer than an inheritance hierarchy because it only needs to consider the structure's signature at the time of compilation to ensure that it meets the desired requirements. In contrast, an inheritance system needs to "walk the tree" to find the appropriate super-class that matches the signature, while implementations are in sub-classes.

7. Summary
Axiomatic Design software practices in 2001 advocated the use of OOT, emphasized inheritance and strict hierarchy following the development of the FR-DP mapping and decomposition. Many researchers have realized this creates complexity in the form of the obscurity of implementation and unreliability. More strikingly, multiprocessors are now standard, creating opportunities and a more challenging environment to program in. Modern languages employ information hiding in objects and methods as prescribed by the Independence Axiom. They also avoid inheritance in favor of a flatter organization modeled on interfaces, delegation, or traits. Software engineering recognizes imaginary complexity as accidental complexity and contrasts it to essential complexity. Elimination of the coupling created by inheritance reduces imaginary complexity from not understanding the class hierarchy involved in the implementation. The Independence Axiom indicates one should Increase coherence, even though it resembles coupling. In a highly coherent system, each component addresses one functional requirement. The authors believe that mapping Axiomatic Design's axioms to modern software engineering paradigms is key to gaining the benefits of an AD mindset's structured creativity while still following coding standards.

References
[1] Clark L 2004 *IEEE Control Systems Magazine* **24** 100–101
[2] Johnson C W 2005 *Formal Methods* pp 9–25

[3] Naur P and Randell B (eds) 1968 *Proc. Nato Software Engineering Conference* (Garmisch)
[4] Oettinger A A 1966 *Communications of the ACM* **9**
[5] 1965 computers and automation: The computer directory and buyer's guide, 1965 URL `https://ia601909.us.archive.org/26/items/bitsavers_computersA_13094490/196506.pdf`
[6] Cameron L 2021 *IEEE Computing Edge* URL `https://www.computer.org/publications/tech-news/events/what-to-know-about-the-scientist-who-invented-the-term-software-engineering`
[7] McIlroy M D 1968 in Naur and Randell [3] pp 138–150
[8] Parnas D L 1972 *Communications of the ACM* **15** 1053–1058
[9] Liskov B and Zilles S 1974 *Proceedings of the ACM SIGPLAN symposium on Very high level languages* (Santa Monica, CA, USA: ACM) pp 50–59
[10] Benington H D 1983 *Annals of the History of Computing* **5** 350–361
[11] Royce W W 1970 *Proceedings IEEE WESCON* (Los Angeles, CA, USA: IEEE) pp 328–338
[12] Department of Defense 1988 *DOD-STD-2167A: Defense Systems Software Development*
[13] Department of Defense 1994 *MIL-STD-498: Software Development and Documentation*
[14] Brooks Jr F P 1995 *The Mythical Man-Month* anniversery ed (Reading, Mass.: Addison-Wesley) ISBN 0-201-83595-9
[15] Bowen T, Dworack F, Chow C, Griffeth N, Herman G and Lin Y J 1989 *Seventh International Conference on Software Engineering for Telecommunication Switching Systems* (Bournemouth, UK: IET) pp 59–62
[16] Larman C and Basili V R 2003 *Computer* **37** 47–56
[17] Jacobson I, Booch G and Rumbaugh J 1999 *The Unified Software Development Process* Addison-Wesley Object Technology Series (Addison-Wesley) ISBN 978-0321822000
[18] Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, Kern J, Marick B, Martin R C, Mellor S, Schwaber K, Sutherland J and Thomas D 2001 Manifesto for agile software development Web page `https://agilemanifesto.org/`
[19] Nygaard K and Dahl O J 1978 *History of programming languages* (Los Angeles, CA, USA: ACM) pp 439–480
[20] Kay A C 1993 *ACM SIGPLAN Notices* **28** 69–95
[21] Stroustrup B 1980 Classes: An abstract data type facility for the C language Computer Science Technical Report CSTR-84 Bell Laboratories
[22] Cox B L 1983 *ACM SIGPLAN Notices* **18** 15–22
[23] Booch G 1982 *ACM SIGAda Ada Letters* **I** 64–76
[24] Meyer B 1985 *IEEE Software* **2** 6–20
[25] Meyer B 1988 *Object-Oriented Software Construction* (Prentice Hall)
[26] Booch G 1993 *Object-Oriented Analysis and Design with Applications* 2nd ed (Benjamin Cummings)
[27] Jacobson I, Christerson M and Jonsson P 1992 *Object-Oriented Software Engineering — A Use Case Driven Approach* (Reading, MA: Addison-Wesley)
[28] Rumbaugh J, Blaha M, Premerlani W, Eddy F and Lorensen W 1990 *Object-Oriented Modeling and Design* (Prentice Hall)
[29] Suh N P 1990 *The Principles of Design* (Oxford University Press)
[30] Cochran D S, Foley J T and Bi Z 2016 *International Journal of Production Research* **55**(3) 870–890
[31] Benavides E M 2012 *Advanced engineering design - An integrated approach* (Woodhead Publishing)
[32] Suh N P 2001 *Axiomatic Design - Advances and Applications* (Oxford University Press)
[33] Kim S J, Suh N P and Kim S G 1991 *Annals of the CIRP* **40** 165–170
[34] Kim S G and Suh N P 1987 *International Journal of Robotics and CIM* **3**
[35] Do S H and Park G J 1996 *CIRP Workshop on Design and Intelligent Manufacturing Systems* (Tokyo)
[36] Do S H and Suh N P 2000 *CIRP Annals* **49**
[37] Pacifici B, Parretti C, Girgenti A and Citti P 2017 *11th International Conference on Axiomatic Design (ICAD)* ed Dodoun O (Iasi, Romania: MATEC Web of Conferences) p 01010 sep. 15–18
[38] Harel D and Pnueli A 1985 *Logics and Models of Concurrent Systems* (*NATO ASI Series* vol 13) ed Apt K R (Springer Verlag) pp 477–498
[39] Buttazzo G C 2006 *IEEE Computer* **39** 54–59
[40] Stevens W P, Myers G J and Constantine L L 1974 *IBM Systems Journal* **13** 115–139
[41] Seibel P 2009 *Coders at Work: Reflections on the Craft of Programming* 1st ed (Apress) ISBN 978-1430219484
[42] Matsuoka S and Yonezawa A 1993 *Research Directions in Concurrent Object-Oriented Programming* ed Agha G, Wegner P and Yonezawa A (MIT Press) pp 107–150
[43] Rice H G 1953 *Transactions of the American Mathematical Society* **74** 358–366
[44] Pierce B C 2002 *Types and Programming Languages* (MIT Press)
[45] Leveson N G and Turner C S 1993 *IEEE Computer Magazine* 18–41
[46] Gualtieri L, Rauch E, Rojas R, Vidoni R and Matt D 2018 *12th International Conference on Axiomatic Design (ICAD)* ed Puik E, Foley J T, Cochran D and Betasolo M (Reykjavík, Iceland: MATEC Web of

Conferences) p 01003 october. 9–11
[47] Klabnik S and Nichols C 2019 *The Rust Programming Language* (No Starch Press) URL `https://doc.rust-lang.org/book`