Building Maintainable Software Using Abstraction Layering

John Spray, Roopak Sinha, *Senior Member, IEEE*, Arnab Sen, and Xingbin Cheng

***Abstract*—Increased software maintainability can help improve a company's profitability by directly reducing ongoing software development costs. Abstraction Layered Architecture (ALA) is a reference architecture for building maintainable applications, but its effectiveness in commercial projects has remained unexplored. This research, carried out as a 16-month joint industry-academic project, explores developing commercial code bases using ALA and the extent to which ALA improves maintainability. An existing application from Datamars, New Zealand, was re-developed by using ALA and compared with the original application. In order to carry out these comparisons, we developed suitable measures by adapting maintainability characteristics from the ISO 25010 family of standards. Specifically, we determined metrics to capture the five sub-characteristics of maintainability: modularity, reusability, analysability, modifiability, and testability; and used them to test our hypothesis that the use of ALA improved maintainability of the application. During the evaluation, we found that the modularity, reusability, analysability, and testability of the re-developed ALA application were higher than for the original application. The modifiability of the ALA-based application was lower in the short-term, but shown to trend upwards in the longer term. Our findings led to proposing a generalised ALA-based development method that promises a significant reduction in maintenance costs.**

***Index Terms*—maintainability, ALA, modularity, reusability, analysability, modifiability, testability**

I. Introduction

Software maintainability is directly correlated with profitability. More maintainable software is easier to update and extend, which helps reduce software development costs. It is said that 90% of commercial software is under maintenance [1], so even minor improvements in maintainability can provide high returns.

A recently proposed reference architecture, called Abstraction Layered Architecture (ALA) [2], claims to significantly improve maintainability. ALA was shown to qualitatively improve various maintainability sub-characteristics [3] in a small student-driven case study developed in a commercial setting. This article evaluates the impact of ALA on maintainability through the re-development of a commercial application. This experience also leads to exploring a generalised approach to using ALA in commercial software development projects, which was missing in the previous work.

J. Spray is with Datamars New Zealand. e-mail: johnspray74@gmail.com

R. Sinha is with the Department of Computer Science & Software Engineering, Auckland University of Technology, New Zealand. e-mail: roopak.sinha@aut.ac.nz

A. Sen is with the Department of Computer Science & Software Engineering, Auckland University of Technology, New Zealand. e-mail: arnab.sen210@gmail.com

X. Cheng is with Datamars New Zealand. e-mail: rosman.cheng@datamars.com

Datamars (New Zealand) manufactures various hardware and software solutions for livestock management. The Datamars application (DMA) is a key desktop software application, which allows Datamars' devices to interface with other devices and online databases for several purposes, including livestock data management, settings, device updates and reporting.

As the number of user stories, devices, radio protocols, and database APIs has grown, DMA has undergone continual maintenance, which has become increasingly costly over 20 years. Even small changes typically require understanding many parts of the code. As the code is highly coupled, it requires many 'all-files finds' to analyse. Changes continually compound the situation by requiring ever-widening interfaces, further increasing coupling and complexity. Other new major requirements, such as automatic data transfers, cannot even be contemplated because of their difficulty.

Due to its rising complexity, Datamars discontinued major feature addition on *Legacy-DMA* in 2018-19 and commissioned a "from-scratch" re-development of the application. As maintainability was key, we decided to use ALA for this re-development to evaluate its effectiveness. In order to do that systematically, we wanted to ascertain the best way to measure maintainability in the two versions of the application objectively. Furthermore, as this project also incorporated a development phase, it was considered valuable to document the process used. These motivations led to the formulation of the following research questions:

RQ1. How can the process of ALA-based development be generalised?
RQ2. What are the most relevant measures for assessing maintainability of a commercial object-oriented code base?
RQ3. To what extent does ALA improve maintainability, as measured on a commercial code base using the measures identified from **RQ2**?

RQ1 was answered via the case study methodology [4]. Building on our team's experience in ALA-based development in small, laboratory-based examples, we developed an ALA-based version of the DMA, called *ALA-DMA*. ALA is described in Sec. II. This experience led to identifying patterns of maintenance in ALA code bases and formulating a generalised ALA development process, which is presented in Sec. III. For **RQ2**, a systematic literature review was used to identify commercially available metrics for measuring the maintainability of code bases. In contrast to existing literature, such as [5], we provide the first categorisation of available metrics based on the sub-characteristics of maintainability defined by ISO 25010 [3] and the measures for each of these

sub-characteristics defined by ISO 25023 [6]. The findings of this categorisation are summarised in Sec. IV. **RQ3** was answered in two steps. We first conducted an initial, qualitative assessment of ALA's impact on maintainability, presented in Sec. V-A. Subsequently, in Sec. V-B, we describe a deeper and more comprehensive quantitative analysis of ALA-DMA using the metrics identified in Sec. IV.

This study makes four contributions. Firstly, we provide a commercial case study (ALA-DMA) to show how an ALA program can be developed in practice. Secondly, we propose a generalised process to reliably develop and maintain ALA applications, based on the experience gained from the ALA-DMA case study. Thirdly, we provide a categorisation of concrete and commercially available maintainability metrics based on the five sub-characteristics of maintainability defined by ISO 25010 [3]. These metrics are supported by commercial tools and can be readily used by practitioners. Finally, we present qualitative and quantitative (based on the ALA-DMA case study) analysis of ALA based on the metrics we propose. Our analysis shows that ALA can significantly improve four sub-characteristics of maintainability: modularity, reusability, analysability and testability. Modifiability is lower in the initial stages but is shown to trend upwards.

II. Abstraction Layered Architecture (ALA)

This section introduces the ALA reference architecture and illustrates its use through the development of ALA-DMA. We use Krutchen's 4+1 viewset model [7] to describe ALA in the following subsections.

At the fundamental level, ALA enforces two constraints: 1) Only one type of relationship is allowed - a dependency on an abstraction that is more abstract than the one using it. 2) All abstractions should be small, usually in the range of 100 to 500 LOC. From these basic rules, properties and patterns emerge as described in the following views. We define dependencies and abstractions clearly in subsequent subsections.

A. Development View

The development view describes both the process of developing ALA applications, as well as the artefacts that are produced during the development process [7].

1) Abstractions, Dependencies and Layers: Abstractions are the only package of code in ALA. An abstraction is more than a module in that it must also be a 'conceptual idea'. That conceptual idea gives it stability.

We define *dependency* as one artefact using or requiring another artefact to function. We define *coupling* as code in different artefacts having some form of knowledge of each other; some form of implicit or explicit collaboration is occurring at design-time. Changes in one can potentially cause changes in the other. A dependency on a more abstract abstraction is essentially a dependency on a conceptual idea. There is *zero coupling* between the code that is dependent on the abstraction and the code that implements the abstraction. For example, consider the abstraction, square-root. As a conceptual idea, it is stable. Therefore there is zero coupling between code that uses square-root and code that implements square-root. Since the only legal dependency in ALA is on a more abstract abstraction, ALA exhibits zero coupling. Abstractions are also internally highly cohesive, so ALA is said to have *zero coupling* and *high cohesion*. ALA shifts *information hiding* from *compile-time* (encapsulation) to *design-time* (abstraction).

ALA distinguishes between *good* and *bad* dependencies, whereas conventional software tends to view all dependencies the same way. The ALA view is that a dependency on a more abstract abstraction is good, not only because it has zero coupling, but because a high number of such dependencies means the abstraction has greater reuse. There are at least two types of dependencies found in conventional code that are illegal in ALA. One is used for communication between different parts of an application. The other is when a module is broken up arbitrarily into parts that are specific to that module. Both these types of dependencies cause coupling. ALA replaces coupling or collaboration between modules with cohesive code completely contained inside a new abstraction in a higher layer. ALA replaces hierarchical decomposition with composition of instances of abstractions from lower layers.

The ALA constraints cause the emergence of abstraction *layers*. Each layer has its own sub-folder containing multiple abstractions. Lower layers are more abstract than higher layers. Dependencies only go down the layers.

Abstractions in each layer tend to have certain characteristics. A reasonably sized application, such as ALA-DMA contains four layers: *Application* (top), *Domain abstractions*, *Programming paradigms*, *ALA foundation* (bottom). Fig. 1 shows a common pattern of dependencies through three layers.

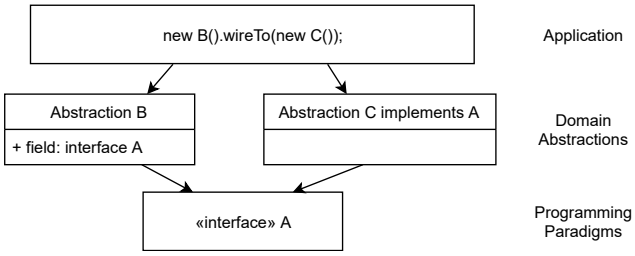

Fig. 1: A sample wiring pattern that conforms to ALA

The top-most application layer (covered in Sec. II-B) contains compositions of instances of *domain abstractions*. A domain abstraction is typically implemented as a class. Domain abstractions have *ports* that allow us to compose their instances in the application layer to describe user stories. Ports are instances of *programming paradigms*. Programming paradigms typically define the way instances of domain abstractions can be composed together. Each defines a different meaning for composition. They are typically implemented as an interface. Each has an execution model (covered in Sec. II-C). The ALA foundation layer has an abstraction, called the `wireTo` operator, that supports the whole pattern.

Fig. 1 shows a programming paradigm **`A`** implemented as an interface. Domain abstractions **`B`** and **`C`** use the programming paradigm **`A`** as its ports. **`B`** has a private field of the type of interface **`A`**, while **`C`** implements the interface. Instances of **`B`**

and `C` can therefore be *wired* together in the application layer using the `wireTo` method.

2) Development Process: The ALA development process shown in Fig. 2 features *up-front design* for a maximum of one agile iteration, similar to the waterfall model [8], where we identify a starting set of abstractions based on an initial set of functional and quality requirements.

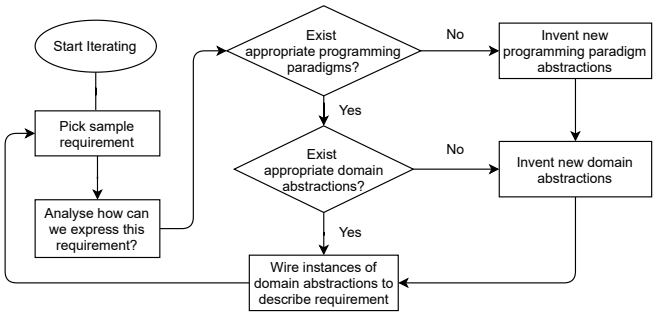

Fig. 2: ALA Up-front Design Process

The first step of the up-front design is to identify a representative sample of requirements. For ALA-DMA, the initial requirement types included connection with different Datamars livestock management (F1), downloading data from connected devices (F2) and saving it in different file formats (F3), importing and uploading files from/to other devices and cloud servers (F4, F5), as well as quality requirements related to interoperability between all Datamars products (Q1) and usability (Q2).

Next, we produce an initial top-level logical structure for the software called an *application diagram* (discussed in detail in Sec. II-B). This involves very rapid iterations where we process the representative sample of requirements, and "express" them in terms of a composition of instances of invented domain abstractions. In this up-front phase, the team does not build any implementation code for domain abstractions or programming paradigms; their job is to find an initial set of domain abstractions and programming paradigms that can express requirements in the domain, through rapid iterations of the loop in Fig. 2.

For ALA-DMA, around 50 user stories were expressed by the up-front design diagram. Each user story started with adding UI because that has the easiest identifiable abstractions, then handling any events from the UI or other external sources, then any data-flow sources, destinations, and transformations needed to complete the user story. Tab. I shows the initial abstractions created.

The up-front design team would typically include an "ALA architect" role - someone who understands and has some experience with creating domain abstractions and, more importantly programming paradigms. The programming paradigms should generally be conceived from common relationships found in the wording of requirements. If this is difficult, the requirements may be incomplete or vague. One approach is the architect re-writes the requirements as well-formed user stories or use cases, and validates them with stakeholders first. It is important that no longer than one agile iteration

Layer	Abstractions
Application	User stories
Domain Abstractions	Button, Grid, MainWindow, Menu, MenuItem, Horizontal, OpenFileBrowser, OpenWindowsExplorer, OptionBox, OptionBoxItem, Panel, Picture, PopupWindow, ProgressBar, RightJustify, RowButton, SaveFileBrowser, Text, ToolItem, Vertical, Wizard, WizardItem, ToDataFlow, ToEvent, Count, Equals, Filter, Iterator, LiteralString, Map, Not, Select, Sort, StringFormat, Transfer, Value, SCPProtocol, SCPSense, SCPSessions, SCPData, SCPLifeData, CSVFileReaderWriter
Programming Paradigms	UI-layout, Data-flow, Event-driven, Table-data-flow
Foundation	WireTo operator
Language	Underlying programming language

TABLE I: Initial sets of abstractions for ALA-DMA

is spent on up-front design; the domain abstractions and programming paradigms must be validated by building them and getting the user stories executing before continuing with more functionality in subsequent iterations.

After the up-front design iteration has concluded, the rest of the development adopts an agile process [9] in which domain abstractions and programming paradigms are allocated to team members or teams for implementation and user stories continue to be developed by other team members or teams. For ALA-DMA, Scrum-based development [10], which is standard practice at Datamars, was adopted to develop the application in two-week sprints carried out over 16 months.

As domain abstractions are zero coupled, and depend only on an understanding of programming paradigms, their implementation can be carried out by less-experienced developers. For ALA-DMA, the development was carried out by a master's student for the first 8 months, and three intern students for the remaining 8 months. Finally, maintenance activities involve continued addition of requirements by adding compositions of instances of existing domain abstractions to the application diagrams. New domain abstractions are required with decreasing frequency. Further details of the maintenance phase are given in Sec. III and Fig. 4.

B. Logical View

In ALA the *logical view* [7] is the content of the application layer. This *application diagram*, essentially a static UML object diagram, is the highest and most concrete layer in an ALA code base. All objects are instances of domain abstractions and are connected via their ports. During up-front design, the initial application diagram for ALA-DMA expressed roughly half of the known requirements and contained approximately 1000 instances of the 42 domain abstractions shown in Tab. I.

Fig. 3 shows a part of the ALA-DMA application diagram. All code artefacts for this diagram are available to download and execute at https://github.com/johnspray74/ALAExample. Note that only a subset of the ALA-DMA source code is published due to confidentiality agreements with Datamars. However, the published code shows all the mechanisms needed in an executing ALA application.

We have colour coded the diagram such that the yellow, green, blue, and brown sections represent four distinct user sto-

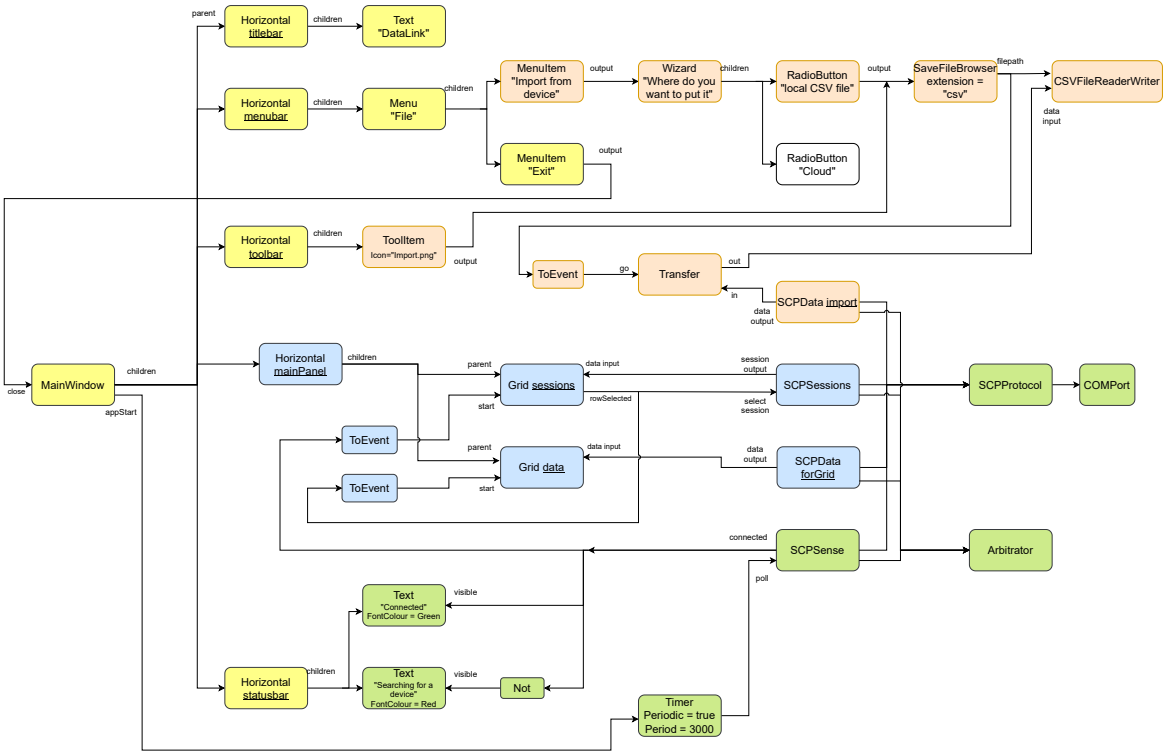

Fig. 3: Partial application diagram expressing user stories of ALA-DMA

ries. Each box represents an instance of a domain abstraction and the lines represent *wirings* between abstraction instances via ports that conform to specific programming paradigms. Instances are anonymous by default, but are sometimes given names (underlined) to describe their function in the context of the application. Some ports are not named where their function is obvious. For example, "input" and "output" for some Data-flow programming paradigm ports, or "children" and "parent" for UI layout programming paradigm ports. The application diagram uses multiple programming paradigms, as described in the following paragraphs.

The user story implemented in the yellow section of Fig. 3 primarily uses the UI-layout programming paradigm to build a UI framework for the application. The wiring from `MenuItem` `Exit` to the `MainWindow` uses the event-driven programming paradigm to signal closing the main window of the application via an event generated when the "Exit" menu item is clicked.

In the green user story, the application senses when a device is connected and indicates its status. A start event comes from the `MainWindow`'s `appStart` port when the application starts running. It starts a periodic timer that emits an event every 3 seconds, which causes the `SCPSense` instance to poll the serial COM port for a device (SCP stands for Serial Command Protocol). When `SCPSense` senses a device, it outputs a true value on its "connected" port, which is using the Data-flow Programming Paradigm. This is wired to the "visible" ports of two text messages that reside on the status bar.

In the blue user story, on connection of a device, the device's session files are displayed. When the user clicks on one of the sessions, it displays the data. The user can scroll through either the sessions or the data. The `Grid` requests rows from its data input as needed over a Table-data-flow programming paradigm.

In the brown user story, the user can choose to import a session from the menu. A wizard is used to select destination options, one of which is to write the data to a CSV file. The `Wizard` appears when it receives an event on its "start" port. When the user selects one of the items in the `Wizard` and clicks next, it emits an event on the output port of the `WizardItem` instance, which is wired in turn to an instance of `SaveFileBrowser`, which outputs a filepath. The filepath is wired both to the instance of a `CSVFileReaderWriter` and to an instance of `ToEvent` to convert it into an event, wired to an instance of `Transfer`. `Transfer` knows how to pull all the rows and columns of data on its "in" port and push them to its "out" port (which it does in batches). These ports are also of the Table-data-flow programming paradigm.

The application diagram, which was the output from the up-front design phase did not need to change when the domain abstractions and programming paradigms were actually implemented and, the diagram was able to execute.

C. Process View

The ALA *process view* is effectively governed by the programming paradigms. These are more general than the domain abstractions, and each defines a different execution model. They can be a simple interface, as is the case for a synchronous event-driven programming paradigm, or an execution engine of some kind, such as would be needed for asynchronous event-driven programming or hierarchical state

machines. Documentation of each programming paradigm describes the execution model, together with its properties such as performance, concurrency and distribution. Given a general understanding of the process view of the programming paradigms, the specific process of a user story appears as the relevant connections in the logical view.

1) Executing the Application Diagram: The programming paradigms, being quite abstract, are often relatively simple interfaces. For example, the data-flow paradigm was implemented as:

```
namespace ProgrammingParadigms {
    interface IDataFlow<T> {T Data { set; } };
}
```

In our current C# implementation of ALA-DMA, the domain abstractions just implement or accept these interfaces. One of the simplest domain abstraction implementations is listed below to show how the ports work. The input port is an implemented interface, and the output port is written as a private interface field.

```
using ProgrammingParadigms;

namespace DomainAbstractions
{
    // Emits an event whenever data is received.
    public class ToEvent<T> : IDataFlow<T> // input
    {
        public string InstanceName {get; set;} = "";
        private IEvent output;
        T IDataFlow<T>.Data {set => output?.Execute();}
    }
}
```

Instances of the domain abstractions are wired together by compatible ports according to the diagram. The wiring is achieved by a `WireTo` operator. `WireTo` is an extension method that uses reflection to achieve dependency injection. In other words, through the operation `a.WireTo(b)`, where `a` and `b` are instances of domain abstractions, `b` is assigned to one of `a`'s private interface fields if `b` is found to implement that interface. `WireTo` returns its first operand to support the fluent style. The code below shows how the wirings for the yellow user story from Fig. 3 can be hand-coded. Note that in practice, it is possible (and for the sake of maintainability, preferable) to automate the generation of the wiring code.

```
var mainWindow = new MainWindow();

mainWindow
    .WireTo(new Horizontal() {InstanceName="titlebar"}
        .WireTo(new Text("DMA")))
    .WireTo(new Horizontal() {InstanceName="menubar"}
        .WireTo(new MenuItem("Exit")
            .WireTo(mainWindow)))
    .WireTo(new Horizontal() {InstanceName="toolbar"})
    .WireTo(new Horizontal() {InstanceName="mainPanel"})
    .WireTo(new Horizontal() {InstanceName="statusbar"});
```

The diagram executes in three stages after the program starts up. First, all the wiring code shown above executes to instantiate domain abstractions and inject their wirings. Next, the `MainWindow` makes a call on its `children` port which is a list of IUI interfaces. This causes a tree of IUI calls, which constructs the static part of the entire UI tree from whatever underlying UI widget library the UI domain abstractions are implemented. Once this is done, the IUI interfaces are not used further. Finally, to start the application running, an event is emitted from the `appStart` port of the `MainWindow` to start any active domain abstraction instances. Events can also arise from UI domain abstractions as the user interacts with the application.

D. Physical View

In many architecture designs, the separation of the system starts with the physical view. Logical views are then separately developed for each physical location, often with different technologies. This leads to coupling between the modules at different locations. ALA's approach is to ignore the physical view entirely at first, and get the application layer abstractions that express whole user stories designed in a cohesive manner first. Parts of the application diagram can be annotated and subsequently distributed over multiple physical locations or processes as long as wirings between these parts use programming paradigms that support distribution, such as asynchronous (queued) events, or data-flow using async/await or future types of patterns. The distribution of ALA applications requires further work and was not a part of ALA-DMA which is a monolithic application.

E. Comparing ALA with other architectural styles

When the two fundamental ALA constraints are followed, a number of recognisable patterns emerge, which is not surprising. For example, dependency injection, domain-specific languages, frameworks, components and connectors, programming paradigms such as data-flow, UI-layouts and state-machines, composability, diagrams in general, monad-like two-stage wiring and execution, and composite and decorator patterns can all appear in ALA applications. Using them without the two constraints does not produce an ALA architecture, however. Each has similarities and differences, for example, layering in ALA becomes ”knowledge dependency” layering rather than ”communication dependency” layering. Some traditional wisdom is relatively at odds with ALA. For example, UML class diagrams encourage relationships between classes, preventing them from being good abstractions, which is explicitly disallowed in ALA; the class diagram of an ALA application would be drawn with no lines at all.

Frameworks [11]: ALA domain abstractions are typically closer to the domain of an application than a framework's [11]. The application diagram generally only assembles and configures instances of domain abstractions that then use ports to inter-communicate at run-time, without adding much code. Like frameworks, the ”execution engines” of the programming paradigms go in a lower layer, so the application layer does not control execution flow. However, ALA is also applicable to functional programming, where the application layer does control execution flow. Unlike frameworks, ALA forbids using an abstraction in a lower layer through inheritance. It uses composition instead. If up-calling is required, it passes in a lambda expression, an anonymous function, or uses the observer (publish/subscribe) pattern. The observer pattern cannot be used sideways within a layer.

Domain-driven design (DDD) [12]: Despite ALA's prominent use of the term “domain”, ALA and DDD are different.

DDD first creates a *domain model* outside the code based on gaining a deep understanding of the domain. This technique is complementary to ALA. Then it uses prescribed methods to decompose into modules based on functional partitioning. In contrast, ALA focuses on restating requirements directly and succinctly by composing instances of *domain abstractions*. If the logical diagram is considered to be the domain model, then the domain model is directly executable in ALA.

Domain specific language (DSL): Domain abstractions provide the nouns and verbs, and the programming paradigms provide the grammar of an internal DSL. However, ALA provides layering constraints on how the "language" is implemented. As with an external DSL, ALA generally puts the execution models into a lower layer.

Software product lines (SPL) [13]: SPLs are an enterprise-level architecture that focuses on predictive reuse of artifacts within multiple products with sufficient commonality. In comparison, ALA focuses on abstraction for its ability to organise and zero-couple code even within a single application. ALA is just a lightweight way to organise code

Microservices [14]: Another enterprise-level architecture, microservices effectively encourage better abstractions by using physical hard-boundaries for which external communications requires more effort in the design of APIs. However, peer services can still have design-time coupling because aspects of user stories can still be embedded into them. ALA brings the benefits of abstractions to monolithic applications, and does it with zero coupling. If ALA is applied to microservices, you would add a service to represent a cohesive user story in the top layer, and it would instantiate, configure and connect instances of abstract services. The microservices would use generic interfaces to communicate at run-time, and only provide an API to the upper layer for configuration.

Component based software engineering: Components are often touted as *reusable*, and to the extent that they are reusable they are also abstractions. So the UML component diagram with lollipop interfaces appears to be the same as the ALA application diagram. To conform to ALA the following additional constraints are required. Firstly, the component diagram must control the instantiation and wiring of components. Often a component diagram is just documentation and the actual topology is embedded into the components themselves in terms of the interfaces they provide and use, and implicitly in their collaboration. A component may be substitutable by another that implements the same interfaces, but that is not enough to conform with ALA, even if it is configured by container-based dependency injection. Secondly,, the lollipop interfaces in the component diagram must be more abstract than the components. The domain abstractions know about these interfaces, but the interfaces do not know about the domain abstractions. This allows the property of *composability* (a finite number of components can be assembled in an infinite number of ways). In ALA, we think of interfaces as being programming paradigms to help get them at the right level of abstraction. In terms of the principle of compositionality (the principle that the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them), we think of domain abstractions as the constituent expressions, and programming paradigms as the rules used to combine them.

III. Long-term Maintenance of ALA Code bases

Maintenance activities take three forms [15]. *Corrective* maintenance involves debugging and fixing defects. *Adaptive* maintenance happens in response to technology updates. Finally, *perfective* maintenance involves adding new features.

ALA-DMA underwent ongoing corrective maintenance after the initial up-front design phase concluded. The following stable pattern of carrying out corrective maintenance in ALA was observed. Debugging always begins in the application diagram by following data flows, event sequences or UI layouts in the diagram. There are two possibilities: A logical or wiring error may be found in the application, or an abstraction may not operate as expected. In the latter case, the relevant abstraction can be inspected and corrected. Due to zero coupling, other abstractions remain unaffected.

In adaptive maintenance, changes to the underlying programming language or its libraries can potentially affect all the other layers. In ALA-DMA, no substantial adaptive maintenance was carried out. However, updates in the .NET framework in which the application was written required only slight idiomatic changes to the implementation of the domain abstractions, and the application diagram remained unaffected. The domain abstractions layer can be thought of as providing a DSL-like or framework-like API for the direct implementation of requirements in the application layer. Even major technology updates requiring a re-implementation of certain domain abstractions will not affect this API, ensuring that the application layer stays unaffected during adaptive maintenance.

Regular perfective maintenance was carried out to continuously implement additional user stories, or modify the application for evolving requirements emerging from user feedback. The following pattern emerged:

1) The application diagram is updated to implement new or modified requirements in the form of updated wirings or domain abstraction instances.
2) If a suitable abstraction is not available when implementing a new or modified requirement, new abstractions may be created, in a similar fashion to ALA's up-front design phase.

Over the 16 months of development, ALA-DMA used over 100 domain abstractions and six programming paradigms. Fig. 4 shows the average commits per month to the application layer, the domain abstractions and the programming paradigms since their creation. The graphs give an indication of increasing stability of domain abstractions and programming paradigms. The application commits are applicable to approximately 10% of the code residing in the top-most application layer. The "hump" at the 8-9 month mark is caused by the arrival of the three student interns whose first task was to add comments to all existing abstractions as they learnt what the abstractions did.

ALA-based development and perfective maintenance are identical in nature, and lead to a generalised iterative and

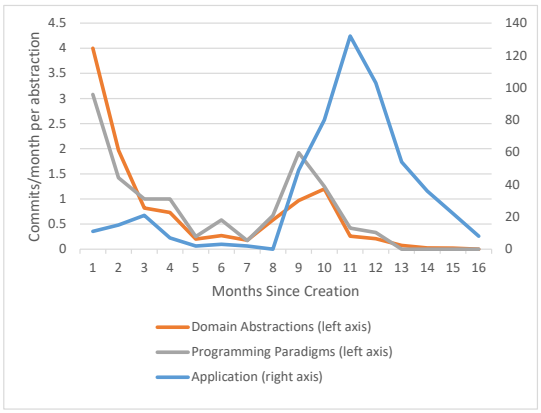

Fig. 4: Average Monthly Commits during ALA-DMA development by abstraction type since creation

agile [9] development process to answer **RQ1**, as shown in Fig. 5. The shaded boxes highlight the differences with the up-front design process of Fig. 2. Fig. 2 has an additional analysis step to emphasise that it is about the design of a set of abstractions to enable the expression of requirements. Fig. 5 has a validation step to emphasis that each user story is executed and tested. Also, any new abstractions that are introduced are built immediately. The primary artefact being incremented is the application diagram. The starting set of domain abstractions (the core output of the up-front design) is more stable because they are independent of specific requirements. Domain abstractions can continue to be added or refined, but with decreasing frequency as the iterations progress. This happens despite that these abstractions are continuously being reused to express new requirements.

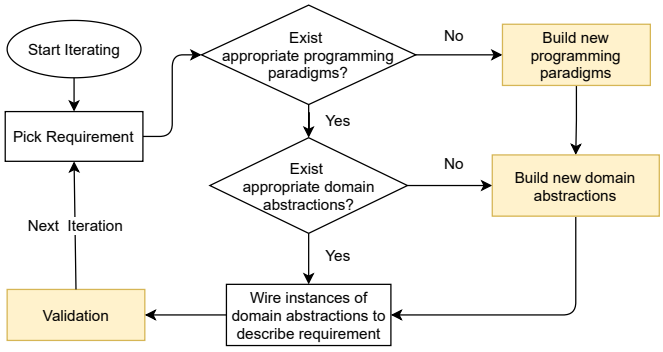

Fig. 5: A generalised ALA development process

IV. Maintainability Metrics and ALA's Impact

A systematic literature review (SLR) [23] of 102 primary studies was conducted to identify suitable metrics for measuring maintainability. Due to space restrictions, details of the SLR protocol, as well as extended findings can be found in [24] (pages 19-25).

In the 1970s, procedural programming related metrics like McCabe's Cyclomatic Complexity [17] and Halstead Volume [25] were proposed. As object-oriented programming grew in popularity in later decades, corresponding maintainability metrics, like CK metrics [16], the suites proposed by Li and Henry [26], Chen and Lum [27], Lorenz and Kidd [28] and the MOOD metrics [29], gained popularity. In the 2000s, focus shifted to the use of machine learning (ML) techniques to measure or predict maintainability, using techniques like Artificial Neural Networks [30] and Multivariate Regression Models [31]. More recently, ML techniques link maintainability measurements with code smells [32].

We classified maintainability metrics into four categories, where each category relates to measuring maintainability at specific phases in the Software Development Life Cycle (SDLC). *Architecture-level metrics* measure maintainability in early or high-level design phases and include seminal works like SAAM [33] and ALMA [34]. *Design-level metrics* assess maintainability in low-level design comprising diagrams and documentation linking classes, interfaces and their relationships in object-oriented programs. CK [16] and MOOD [29] metric suites fall in this category. *Code-level metrics* are used over detailed code implementations and include metrics such as Information Flow [35] and Nesting Level [36]. *Process-level metrics* focus on the subsequent maintenance of software that has already been deployed and includes metrics like Lines of Code (LOC) changed [37]. Progressively applying these metrics results in increasing and more accurate overall assessment of maintainability.

The ISO 25010 [3] family of standards and more specifically the ISO 25023 [6] standard provides several quantitative measures for the assessment of the five sub-characteristics of maintainability: *modularity, reusability, analysability, modifiability* and *testability*. However, some of the ISO 25023 measures are very abstract and do not apply directly to modern object-oriented code bases. For instance, in measuring modularity, ISO 25023 does not define how modular code should be arranged, and so we can use the CBO measure which explicitly defines the coupling between objects and classes.

Tab. II shows a summary of our answer to **RQ2.** It contains the refinement of the ISO 25023 measures for each maintainability sub-characteristic into more widely-used and better-known metrics that are easier to use due to the availability of tools. For some measures, such as the coupling of components for modularity and coding rules conformity for reusability, we identified specific metrics like CBO and layer violation, respectively, which provide equivalent and more specific measurements. For some measures, such as reusability of assets, we identify a combination of metrics such as CBO, LCOM, WMC and NOC for a more detailed analysis. For sub-characteristics like analysability and testability, no one-to-one or one-to-many relationships can be identified between ISO 25023 measures and available metrics. Therefore, a set of refined metrics was used to replace the set of ISO 25023 measures for these sub-characteristics. Finally, for modifiability, ISO 25023 measures are directly applicable to any code base, so no refinement was required.

V. Maintainability Evaluation of ALA

To answer **RQ3**, we first carried out an initial qualitative analysis of ALA using the maintainability metrics identified

Sub-characteristic	ISO 25023 measure	Refined measures
Modularity	Coupling of components	CBO (Coupling Between Object Classes) [16]. CBO explicitly shows the relations between classes and interfaces, which gives an intuitive reflection of whether a component has coupling with others.
	Cyclomatic complexity adequacy	CC (Cyclomatic Complexity) [17]. This metric reflects the complexity of the control flow in a function or a class.
Reusability	Reusability of assets	CBO (Coupling Between Objects Classes). Excessive coupling weaken the encapsulation of a class and inhibits reuse [18].
		LCOM (Lack of Cohesion Methods). If a method refers to more external variables, it is more specific to the application and less reusable [16].
		WMC (Weighted Methods per Class). A class with a larger number of methods demonstrates more specific functionality, limiting potential reuse [19]. A class with a single and simple responsibility has a higher chance to be reused.
		NOC (Number of Children). More children means more reuse [16]. In ALA, this indicates interfaces reusability, as there is no class inheritance.
		IT (Instantiated Times). The actual reused times of a class.
	Coding rules conformity	LV (Layer Violation) [20]. The layered architecture creates unidirectional downwards dependencies in ALA. Higher layer violations means decreased coding rules conformity.
Analysability*	System log completeness Diagnosis function effectiveness Diagnosis function sufficiency	CBO (Coupling Between Objects Classes). This metric is used for measuring the ripple effects of any changes on a class or an interface [16]. LCOM (Lack of Cohesion Methods). Similar to CBO, but stands at the methods level. A method references lesser external variables has lesser side effects when modified, and so it has higher analysability [16]. LOC (Lines of Code) [21]. If an asset had more lines of code, the difficulty of analysis activities might increase. CP (Commenting Percentage) [22]. Proper comments help maintainers identify where to make a change, enhancing analysability.
Modifiability	Modification efficiency	No changes required as these measures are directly applicable to any code base.
	Modification correctness	
	Modification capability	
Testability*	Test function completeness Autonomous testability Test restartability	CBO (Coupling Between Objects Classes) [16]. Less coupling with other components makes it easier to write a unit test case and run it. LOC (Lines of Code) [21]. A class with a bigger size is generally harder to test. WMC (Weighted Methods per Class) [16] measures the cyclomatic complexity of a class. A higher value of WMC means the class is more complex, and requires more effort to test. LCOM (Lack of Cohesion Methods) [16]. Similar with CBO, but stands at the method level inside a class. A method that references fewer external variables is easier to test, as less conditions need to be considered.

TABLE II: Mapping ISO 25023 measures for maintainability sub-characteristics to well-known metrics. (*No one-to-one relationships could be established between Analysability and Testability measures and known metrics.)

in Sec. IV. Subsequently, we carry out a deeper, quantitative analysis of ALA based on the ALA-DMA case study.

A. An Initial Qualitative Assessment of ALA

Fig. 6 summarises the results of an initial assessment of ALA using the refined metrics from Tab. II. Modularity is significantly high due to ALA's "zero coupling" mechanism and single responsibility of domain abstractions. Reusability is also high, because all the domain abstractions and programming paradigms are designed to be reused. Analysability is assessed to be low overall. ALA focuses on the architectural level only, which equates to external analysabilty (measured only by CBO). External analysability is expected to be high, again because of zero coupling. ALA has no constraints that affect internal analysability (the code inside domain abstractions) except for a relatively high limit of 500 LOC per abstraction. Also, the code inside ALA abstractions is highly cohesive, which increases internal coupling. However, with zero external coupling, the internal code of each abstraction is like a separate program. Internal analysability inside a small but independent program is not considered to be a problem commercially. However, it is expected that the measures of internal analysability like LOC, LCOM and CP, will be low in an ALA application. Modifiability is also hard to assess at this stage, and since it depends on modularity (high) and analysability (low) [6], so we infer it as medium. Testability is high, because test functions and criteria are easy to establish and run over domain abstractions. Moreover, it is effortless to achieve test goals due to the single responsibility of domain abstractions. Overall, this qualitative analysis and small-scale experiments conducted by our team showed that ALA holds promise from a maintainability perspective.

B. Quantitative Analysis

We assessed any maintainability improvements in ALA-DMA by comparing it with Legacy-DMA, which has been maintained for over two decades. Tab. III compares the two applications on age, development team's size, code base size in SLOC, and development time in man-years.

Project	age	team size	SLOC	man-years
Legacy-DMA	20+	1	70k	12
ALA-DMA	2	1-3	55k	2

TABLE III: Legacy-DMA vs ALA-DMA

1) `NDepend` *Dependency Graphs:* `NDepend` [38] visualises dependencies between classes and interfaces. Fig. 7

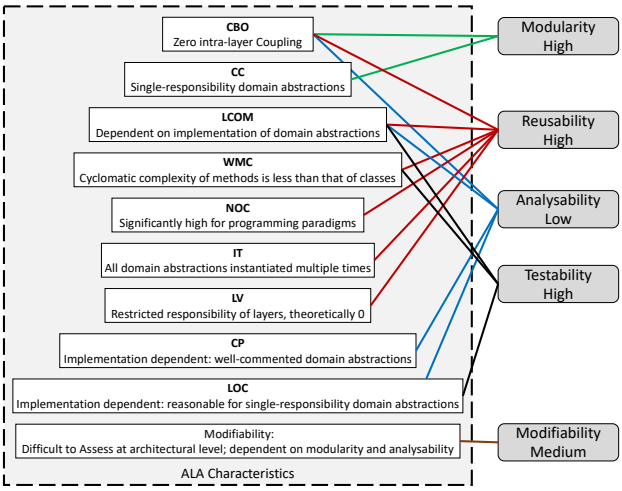

Fig. 6: A qualitative assessment of ALA's support for maintainability sub-characteristics based on ALA principles

and Fig. 8 show partial dependency graphs for ALA-DMA and Legacy-DMA, respectively. The ALA graph is clipped to about half its size vertically. Legacy-DMA's dependency graph was both much larger and exponentially denser, due to which Fig. 8 only includes a small clipped version. Complete graphs can be downloaded from https://github.com/johnspray74/ALAExample/tree/master/Application/Documents/NDependDiagrams.

The ALA-DMA dependency graph is well-organised with dependencies flowing downwards from the application diagram to the classes (domain abstractions), which in turn depend on programming paradigms (such as IUI as shown in Fig. 7). In contrast, the dependency graph for Legacy-DMA shows thousands of dependencies criss-crossing in the background resembling the big ball of mud pattern. Unsurprisingly, NDepend was unable to lay out the graph in layers.

As discussed in Sec. II-B, *good* dependencies flow from more concrete artifacts to more abstract artifacts, such as from domain abstractions to programming paradigms. Following this definition, if we remove the good dependencies from both diagrams to get an indication of coupling, then Fig. 7 would have connections at all. While we can assume that several lines in Legacy-DMA's graph might be removed, we expect that the graph would still look much the same.

2) Modularity: Modularity can be measured via the *component coupling* (CC and CBO) and *cyclomatic complexity adequacy* measures (see Tab. II). CBO includes both afferent coupling and efferent coupling that relate to, respectively, the number of components that depend on a component and the number of components a component depends on. While ALA distinguishes between good and bad dependencies, the CBO metric does not. Fig. 9 and 10 plot afferent and efferent coupling for both applications. ALA-DMA shows a marked improvement in afferent coupling and similar efferent coupling with lesser components having a high number of dependencies. The middle part of Fig. 10 shows the good dependencies between domain abstractions and programming paradigms. Overall CBO is lower for ALA-DMA.

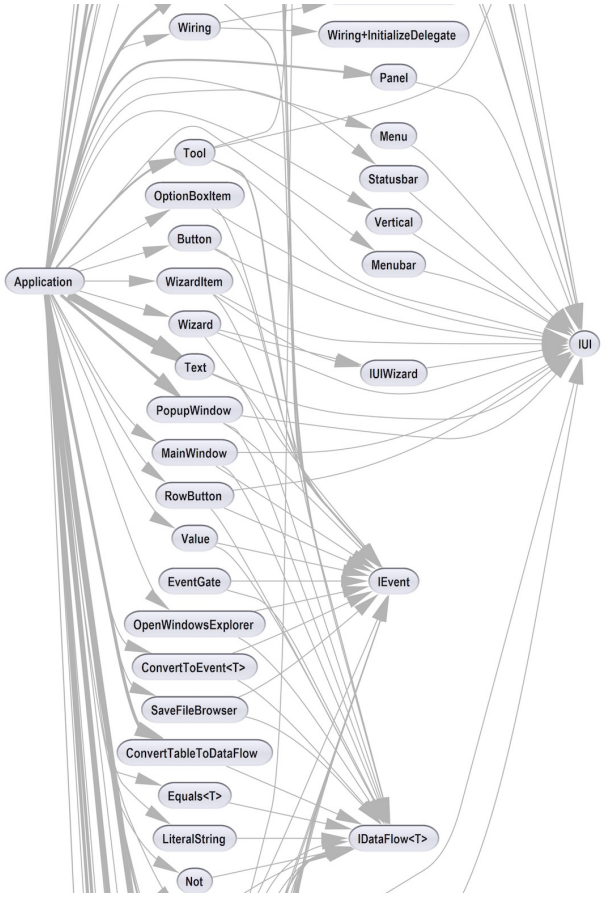

Fig. 7: A Partial Dependency Graph of ALA-DMA

Fig. 8: A Partial Dependency Graph of Legacy-DMA

CC is the ratio of the components implemented independently to those designed to be independent [6]. For ALA-DMA, this ratio is 100%, because there are no intra-layer dependencies between abstractions. For Legacy-DMA, CC cannot be measured because it is impossible to identify independent components from the original design of the application, which was constructed well over 20 years ago.

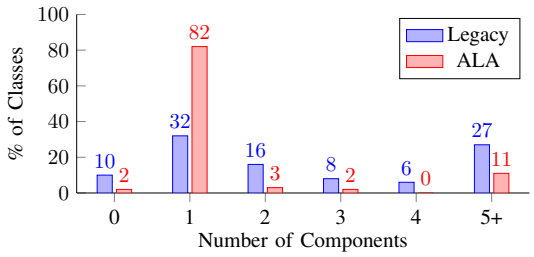

Fig. 9: Comparison of Afferent Coupling

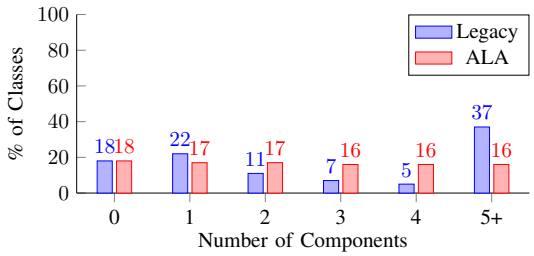

Fig. 10: Comparison of Efferent Coupling

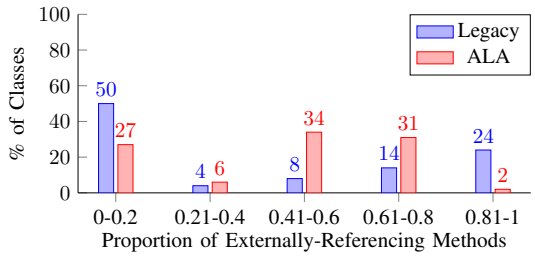

Fig. 11: Lack of Cohesion in Methods (LCOM) Comparison

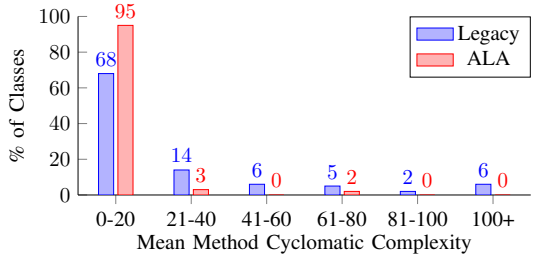

Fig. 12: Comparison of Weighted Methods per Class (WMC)

Nevertheless, ALA-DMA can be observed to be significantly more modular than the legacy application.

For cyclomatic complexity adequacy, the mean cyclomatic complexity values for ALA-DMA and Legacy-DMA are 8.32 and 27.83, respectively. ALA-DMA, therefore, shows a marked improvement in cyclomatic complexity.

Overall, this quantitative analysis concurs with the qualitative analysis, presented in Fig. 6, and we conclude that ALA helps produce code bases with high modularity.

3) Reusability: Reusability is measured using *reusability of assets* and *coding rules conformity* [6]. For reusability of assets, we identified CBO, LCOM, WMC, NOC and IT as suitable metrics. The first three aim to measure the possibility of reuse of assets, while the last two measure the actual reuse of assets. CBO measurements were reported in the modularity section, so we focus now on the other metrics.

Fig. 11 and 12 present the LCOM and WMC measurements for the two code bases. High LCOM and WMC values increase the complexity of a class, which limits its potential reuse [16]. The LCOM results indicate that ALA-DMA has lower reusability. On the other hand, the WMC results show that ALA-DMA has higher reusability. This result is, however, not a contradiction: LCOM measures class-level reusability whereas WMC measures method-level reusability.

Fig. 13 and 14 present the NOC and IT measurements, respectively. As inheritance is not used in ALA, NOC simply represents the number of interface implementations. A greater NOC value indicates greater reuse [16]. NOC measurements show that the percentages of reused interfaces and ancestor classes are approximately identical for the two code bases. The IT metric provides the actual number of classes that have been instantiated. We can see that ALA-DMA involves higher reuse of classes.

For coding rules conformity, we utilise the *layer violation* and *circular dependency violation* metrics, summarised in Tab. IV. The *Layer violation* of ALA is zero due to the absence of *bad* dependencies (discussed in Sec II-B). On the other hand, the Legacy-DMA has 141 circular dependencies among its 449 classes, with 68 correlated classes. ALA, therefore, scores significantly higher on coding rules conformity.

Code Base	Metric	Result	Coding Rule Conformity
ALA	LV	0%	100%
Legacy	CDV	15.14%	84.86%

TABLE IV: Coding Rules Conformity Results

As estimated in Fig. 6, ALA scores higher on reusability with a 13% mean improvement over the five metrics.

A "number of times instantiated" metric was computed much later when ALA-DMA was almost complete and comparable to legacy-ALA in terms of implemented features. This is a manual count of the number of instantiations of classes and interfaces. The average number of instantiations of domain abstractions and programming paradigm interfaces in ALA-DMA was 11.7 and 42, respectively. In comparison, the average number of instantiations of classes and interfaces in Legacy-DMA was 2.5 and 7, respectively.

4) Analysability: The metrics CBO, LCOM, LOC and CP (Tab. II) measure two aspects: the ease of locating the ripple effects of an intended change, and the ease of locating the parts with deficiencies or causes of failure.

Ripple effects of changes are directly associated with the extent of coupling. The internal coupling is measured by LCOM, which emphasises the methods and properties of a class. External coupling is measured by CBO, which considers dependencies between classes. These two metrics have been discussed earlier, and the results show that ALA-DMA has

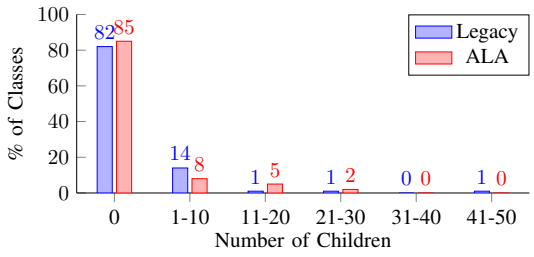

Fig. 13: Comparison of Number of Children (NOC)

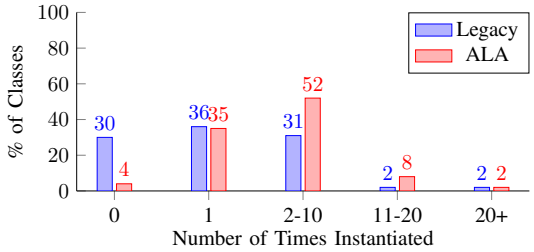

Fig. 14: Comparison of Instantiated Times (IT)

better external analysability, whereas Legacy-DMA has better internal analysability.

The ease of locating deficiencies is measured by LCOM, LOC and CP. LCOM is useful here because failures of assets usually occur due to internal defects. Fig. 15 and Fig. 16 show the LOC and CP measurements for the two code bases. LOC measurements demonstrate that ALA-DMA is more analysable, as classes have fewer lines of code. For CP, excessively low and high CP values indicate low analysability. Compared to the average commenting percentage of 19% in various open source projects [39], ALA-DMA had a much higher mean value of 33% and Legacy-DMA had a considerably lower mean value of 14%.

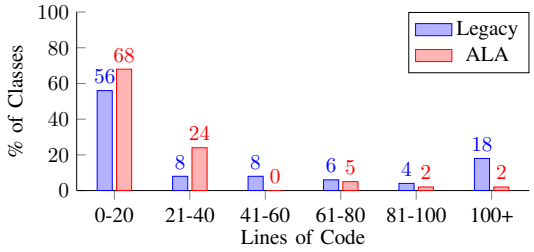

Fig. 15: Comparison of Lines of Code (LOC)

Overall, ALA offered a mean 7% improvement (over the five metrics used) over the legacy application. This result differs from our initial qualitative analysis presented in Fig. 6, where we inferred that ALA might reduce analysability,

5) Modifiability: The key measures for modifiability are *modification efficiency, modification capability and modification correctness*. The correctness of modifications needs to be tracked over the long term and therefore related measurements were not conducted. Modification efficiency and capability were computed over the implementation of the functional

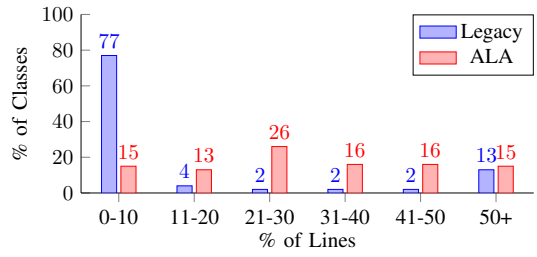

Fig. 16: Comparison of Commenting Percentage (CP)

requirement sets F1–F5 from Sec. II-A. Fig. 17 illustrates the decomposition of a sample user story into modification tasks for ALA-DMA. Fig. 18 shows a comparison of the time taken by the modification tasks for four additional user stories. Overall, the legacy application required 39 hours for completing these four user stories while ALA-DMA required 56.

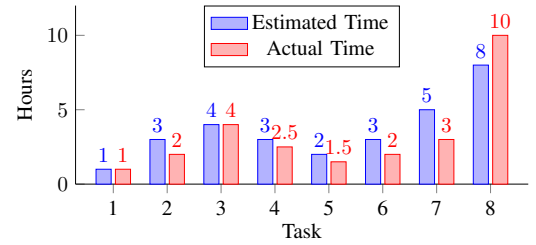

Fig. 17: Modification Efficiency for a sample user story

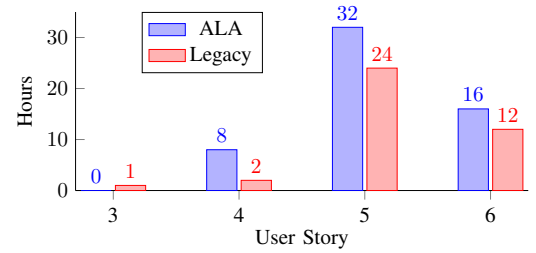

Fig. 18: Comparison of Estimation of Modifications

Modification capability is computed as the proportion of items modified within a specified duration to the number of items required to be modified within that duration. In sample measurements taken over two larger user stories that were implemented simultaneously in both applications, Legacy-DMA took an average of 29 hours to modify each item whereas ALA-DMA required 32.3 hours. ALA-DMA had a 10% lower modification capability than the legacy application.

We see that ALA exhibits low modifiability at the 3-month point in the development, a result that differs from the initial assessment where ALA was predicted to have medium modifiability (Fig. 6). However, Fig. 19 gives an indication of modifiability for ALA-DMA over a longer 16-month development period. It shows the percentage of commits to the applications layer where we directly implement

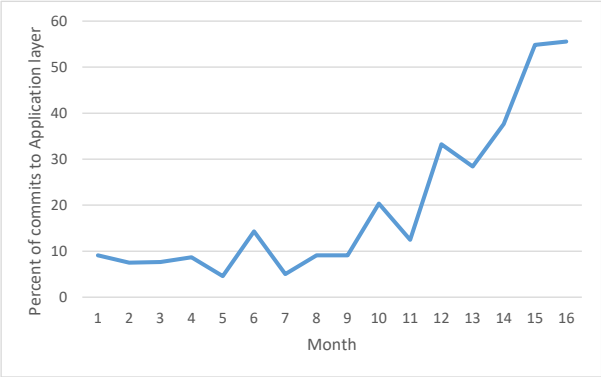

Fig. 19: Percentage monthly commits to the application layer

requirements. At the 3-month point, less than 10% of commits were to the application itself, with the rest going into domain abstractions and programming paradigms. After 16 months this had increased to 55%. It should be noted that the application accounts for only 10% of the total code. The domain abstractions and programming paradigms make up the remaining 90% of the code.

6) Testability: We measured testability using the CBO, LCOM, WMC and LOC metrics (Tab. II), all of which have been discussed in the previous sub-sections. High-coupling requires more testing effort and a high LCOM value makes components more difficult to test [16]. Based on these previous measurements, ALA-DMA has higher testability according to CBO and lower testability according to LCOM. WMC and LOC are correlated with the size and complexity of a class. Complexity correlates with the time needed to test the class [16]. Both these metrics show that ALA-DMA has higher testability than Legacy-DMA. A breakdown of the metrics shows that 60% of ALA's assets have high testability, whereas in the legacy application the number was 47%. Overall, our findings support our initial assessment in Fig. 6 that ALA achieves high testability.

C. Analysis, Discussion and Summary

Sec. V-B helps us answer **RQ3** and shows that ALA-DMA showed improvements in all sub-characteristics of maintainability except *modifiability*, which is dependent on modularity and analysability. Interestingly, we found ALA to have much higher modularity and analysability than the legacy application. A deeper analysis was performed to identify the reasons for the lower modifiability measurement. We recorded the kind of tasks carried out during perfective maintenance and any differences in the time taken to perform similar tasks over the course of the project. We found that there were two primary reasons for the lower modifiability, which are discussed below.

1) Construction of new abstractions: Perfective maintenance in ALA requires minimal effort if new user stories can be implemented by wiring instances of existing domain abstractions in the application layer. However, being a relatively new code base at the time of the measurement, several user stories required the creation of new domain abstractions. Boehm [40] states that developing reusable assets usually increases short-term effort.

Tab. V shows the new domain abstractions that were created when the two user stories used for the measurement were implemented, each of which required additional effort.

User story	New Abstractions Implemented
Connect to device	DeviceIdSCP
Display sessions	IIterator, ListOfFiles, SelectExternal, FileSessions, ConvertIteratorToTab.

TABLE V: Abstractions created for new requirements

2) Complexity of the application diagram: Tab. VI illustrates the task efficiency of the "Display sessions" user story. The last task (T8) involves finalising the wiring in the application diagram. Similar measurements were taken for all user stories. A few trends were observed. More complex user stories took, as expected, longer to wire than simpler user stories. Also, developer estimates related to T8 (or similar tasks in other user stories) were almost always too optimistic.

	T1	T2	T3	T4	T5	T6	T7	T8
Estimated(hrs)	1	3	4	3	2	3	5	8
Actual(hrs)	1	2	4	2.5	1.5	2	3	10
Efficiency(%)	100	150	100	120	133	150	167	80

TABLE VI: Efficiency of completing tasks in User Story 2

It took longer than expected to carry out the manual wiring of the application diagram in the later stages of development. This was because, as the diagram went over the ALA size constraint, keeping the manually generated wiring code in sync with the wiring diagram required more care than expected. This factor was later solved by auto-generating the wiring code from the diagram. Furthermore, the application diagram was separated into user story abstractions by creating an additional intermediate layer between the application and domain abstraction layers, such as a "plug-in" abstractions layer. These two solutions are key future directions of this work.

Overall modifiability can also be indicated by a comparison of the total effort for ALA-DMA and Legacy-DMA. ALA-DMA's development consisted of a 2-week up-front design by a senior engineer and the master's student, a 3-month development by the master's student alone, and two additional 3-month student internships of 3 students each. Approximately 2 man-years of effort in total have been spent. At this time about 90% of the functionality of the Legacy-DMA has been completed and tested. By comparison, the Legacy-DMA has conservatively required 12 man-years of accumulated effort.

ALA-DMA's application diagram contains about 2200 nodes and 3200 wirings, equivalent to about 5.4 KLOC (one line of code per node and one per wiring) representing 10% of the total code. Maintenance increasingly takes place in this 10% of the code, as opposed to potentially 100% of the code in the legacy-DMA.

The factors discussed above indicate that ALA has higher modifiability over the long-term. The domain abstractions and programming paradigms layers have become more stable and maintenance is carried out increasingly in the application layer.

Reusability would increase due to the same reason. Modularity and testability are expected to remain unaffected over the long-term, as these relate to individual abstractions that have capped sizes in ALA. Analysability is expected to remain unaffected, provided the ALA constraints are also applied in the application layer when added features cause a single diagram to become too large.

D. Threats to Validity

A number of provisions were made to mitigate the effects of possible threats to validity [41]. Conclusion validity was achieved through a statistical analysis of the quantitative data. Only statistically significant results were reported in Sec. V-B. We also note that the two code bases being compared were developed at different times, and it may be difficult to separate ALA's maintainability benefits from the deterioration in maintainability of the Legacy-DMA. This factor can be addressed by ongoing experiments on a wider range of systems, which is beyond the scope of this article. Internal and external validity was achieved by documenting progress every day over the duration of the project. Logs were appraised weekly to ensure that a systematic process of development, consistent with both ALA constraints and Datamars' Scrum-based development process, was followed. The development of the two applications were carried out by two independent teams. The findings of the qualitative analysis of ALA's effect on maintainability were triangulated with a deeper quantitative analysis to strengthen the overall findings of the article. For data validity, `NDepend` was used out-of-the-box on both applications. Any bias emanating from the fact that members of the research team developed ALA-DMA was mitigated by ensuring the assessments of suitability and effectiveness were carried out independently by the students and added to the daily logs.

VI. Conclusions

We tested the effectiveness of Abstraction Layered Architecture (ALA) [2] in achieving high maintainability through developing a commercial C# desktop application for Datamars, New Zealand. This experience led to proposing a generalised, iterative process for developing maintainable code using ALA. We also conducted a systematic literature review to categorise available metrics to measure each of the five sub-characteristics of maintainability, as defined by ISO 25010 and ISO 25023 [3], [6]. Using this categorisation, we provide an initial qualitative assessment of ALA, followed by a deeper quantitative analysis where the metrics were applied to the commercial code base we developed.

We found that ALA helped achieve higher *modularity, reusability, analysability and testability* than the conventional strategies used at Datamars, but lower *modifiability* in the short term. Reusability and modifiability did measurably increase over the long term. The growing complexity of the ALA application diagram as it goes over the ALA size constraint can reduce analysability in the long term. The ALA approach is to create an additional intermediate layer between the application and domain abstraction layers to address this issue, and this remains as a key future direction of this work. Another project relating to generating code automatically from the application diagram is currently underway.

For further scalability, we are exploring the use of node clustering methods to simplify the application diagram and the organisation of domain abstractions to make them easier to identify and use. Another interesting future direction is using machine-learning approaches to identify abstractions in conventional code bases to aid in refactoring such code bases into ALA-compliant code bases. In general, there is a need to clearly articulate how domain abstractions can be identified in any ALA project. This will require systematically collecting feedback from experienced developers and engineers working on a range of projects. Gathering detailed empirical evidence from developers on how ALA supports maintainability, as well as carrying out detailed comparisons with architectural alternatives such as pattern-oriented software architecture [42], microservices [14] and refactoring [43] approaches are both critical future directions. Analysing the impact of the zero coupling on other quality requirements remains an interesting next step. Distributing ALA applications onto multiple devices is another future direction that requires elaboration of the physical and process views of the architecture.

References

[1] H. Krasner, "The cost of poor quality software in the US: A 2018 report," *Consortium for IT Software Quality, Tech. Rep*, vol. 10, 2018.

[2] J. Spray and R. Sinha, "Abstraction layered architecture: Writing maintainable embedded code," in *European Conference on Software Architecture*, pp. 131–146, Springer, 2018.

[3] BSI ISO, "BS ISO/IEC 25010:2011 Systems and software engineering Systems and software Quality Requirements and Evaluation (SQuaRE) System and software quality models," *BSI Standards Publication*, 2011.

[4] K. B. M. Noor, "Case study: A strategic research methodology," *American Journal of Applied Sciences*, vol. 5, no. 11, pp. 1602–1604, 2008.

[5] J. de A.G. Saraiva, M. S. de Frana, S. C. Soares, F. J. Filho, and R. de Souza, "Classifying metrics for assessing object-oriented software maintainability: A family of metrics catalogs," *Journal of Systems and Software*, vol. 103, pp. 85 – 101, 2015.

[6] BSI ISO, "BS ISO/IEC 25023:2016 Systems and software engineering Systems and software Quality Requirements and Evaluation (SQuaRE) Measurement of system and software product quality," *BSI Standards Publication*, 2016.

[7] P. B. Kruchten, "Architectural Blueprints - The 4+1 view model of architecture," *IEEE Software*, vol. 12, no. 6, pp. 42–50, 1995.

[8] S. Balaji and M. S. Murugaiyan, "Waterfall vs. V-Model vs. Agile: A comparative study on SDLC," *International Journal of Information Technology and Business Management*, vol. 2, no. 1, pp. 26–30, 2012.

[9] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[10] K. Schwaber, "Scrum development process," in *Business object design and implementation*, pp. 117–134, Springer, 1997.

[11] R. E. Johnson, "Frameworks=(components+patterns)," *Communications of the ACM*, vol. 40, no. 10, pp. 39–42, 1997.

[12] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[13] S. Apel, D. Batory, C. Kästner, and G. Saake, "Software product lines," in *Feature-Oriented Software Product Lines*, pp. 3–15, Springer, 2013.

[14] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015.

[15] U. Kaur and G. Singh, "A review on software maintenance issues and how to reduce maintenance efforts," *International Journal of Computer Applications*, vol. 118, no. 1, 2015.

[16] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[17] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.

[18] V. Laing and C. Coleman, "Principal Components of Orthogonal Object-Oriented Metrics," *White Paper SATC-323-08-14, NASA Goddard Space Flight Center, Greenbelt, Maryland*, vol. 20771, 2001.

[19] B. M. Goel and P. K. Bhatia, "Analysis of reusability of object-oriented system using CK metrics," *International Journal of Computer Applications*, vol. 60, no. 10, pp. 32–36, 2012.

[20] S. Sarkar, G. M. Rama, and R. Shubha, "A method for detecting and measuring architectural layering violations in source code," in *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*, pp. 165–172, IEEE, 2006.

[21] A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code, and development effort prediction: a software science validation," *IEEE Transactions on Software Engineering*, no. 6, pp. 639–648, 1983.

[22] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *International Conference on Program Comprehension (ICPC)*, pp. 83–92, IEEE, 2013.

[23] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering–a systematic literature review," *Information and Software Technology*, vol. 51, no. 1, pp. 7–15, 2009.

[24] X. Cheng, "Abstraction layered architecture: Improvements in maintainability of commercial software code bases," Master's thesis, Auckland University of Technology, 2020.

[25] M. H. Halstead *et al.*, *Elements of software science*, vol. 7. Elsevier New York, 1977.

[26] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111–122, 1993.

[27] J. Chen and J. Lu, "A new metric for object-oriented design," *Information and Software Technology*, vol. 35, no. 4, pp. 232–240, 1993.

[28] M. Lorenz and J. Kidd, *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994.

[29] F. B. Abreu and R. Carapuça, "Object-oriented software engineering: Measuring and controlling the development process," in *International Conference on Software Quality*, vol. 186, pp. 1–8, 1994.

[30] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Application of artificial neural network for predicting maintainability using object-oriented metrics," *Transactions on Engineering, Computing and Technology*, vol. 15, pp. 285–289, 2006.

[31] Y. Zhou and H. Leung, "Predicting object-oriented software maintainability using multivariate adaptive regression splines," *Journal of Systems and Software*, vol. 80, no. 8, pp. 1349–1361, 2007.

[32] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 612–621, IEEE, 2018.

[33] R. Kazman, L. Bass, G. Abowd, and M. Webb, "SAAM: A method for analyzing the properties of software architectures," in *International Conference on Software Engineering*, pp. 81–90, IEEE, 1994.

[34] P. O. Bengtsson, N. Lassing, J. Bosch, and H. Van Vliet, "Architecture-level modifiability analysis (ALMA)," *Journal of Systems and Software*, vol. 69, no. 1-2, pp. 129–147, 2004.

[35] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Transactions on Software Engineering*, no. 5, pp. 510–518, 1981.

[36] W. A. Harrison and K. I. Magel, "A complexity measure based on nesting level," *ACM Sigplan Notices*, vol. 16, no. 3, pp. 63–74, 1981.

[37] R. D. Banker, R. J. Kauffman, and R. Kumar, "An empirical test of object-based output measurement metrics in a computer aided software engineering (case) environment," *Journal of Management Information Systems*, vol. 8, no. 3, pp. 127–150, 1991.

[38] P. Smacchia, "NDepend," *Product description on company website at http://www.ndepend.com*, 2007.

[39] O. Arafat and D. Riehle, "The commenting practice of open source," in *ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, pp. 857–864, ACM, 2009.

[40] B. Boehm, "The COCOMO 2.0 software cost estimation model," *American Programmer*, 1996.

[41] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research-an initial survey.," in *Seke*, pp. 374–379, 2010.

[42] F. Buschmann, K. Henney, and D. Schimdt, *Pattern-Oriented Software Architecture: On Patterns And Pattern Language*, vol. 5. Wiley, 2007.

[43] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

John Spray is Embedded Software Team Lead at Datamars. John did a degree in engineering before software engineering was a thing and started by writing a compiler in the days when there were none. Having seen a plethora of patterns, principles and styles of software architecture enter the popular memes, John has a unique industry perspective on the human struggle to understand this new discipline. His goal is to unify them into a single guiding theory to help others to organize their code. There's sometimes a little time left over to fly model airplanes and play tennis.

Roopak Sinha (Ph.D. '09, MCE 16, BE (Hons) 03, SFHEA 17, SMIEEE) is the Head of the Department of Computer Science and Software Engineering at Auckland University of Technology, New Zealand. Roopak's primary research interest is *Systematic, Standards-First Design of Complex, Next-Generation Software* applied to domains like Internet-of-Things, Edge Computing, Cyber-Physical Systems, Big Data, Home and Industrial Automation, and Intelligent Transportation Systems. He contributes actively to international standards on software and systems. He also works with New Zealand companies to systematically reduce standards-compliance costs in software-intensive systems. He has previously worked as an academic at INRIA Grenoble, France and The University of Auckland, New Zealand.

Arnab Sen received a BSc (2019) in Computer Science from The University of Auckland and an MPhil (2021) in Computer Science from Auckland University of Technology. His research interests include the optimisation of software architectures for long-term maintainability, and the creation of software development tools to help improve productivity. Of particular interest to him is exploring the impact of novel software tools and architectures in industrial settings. He is currently working as a Software Developer at the content management and insights company Pingar.

Xingbin Cheng is a mobile developer at Datamars (New Zealand). Before he started working for Datamars, Xingbin got a master's degree from Auckland University of Technology in computer and information science. Prior to that, he studied in Northeastern University and got his bachelor's degree in computer science and technology, and worked for a few companies in China. Xingbin's interests are software architecture and computer graphics. His research fields are software maintainability, including sub-characteristics i.e. modularity, reusability, analysability, modifiability and testability. He also devotes to improve user interface rendering in mobile platforms.