Abstraction Layered Architecture: Writing Maintainable Embedded Code

John Spray[1] and Roopak Sinha[2]

[1] Tru-Test Group, Auckland, New Zealand
john.spray@trutest.co.nz
[2] Department of Information Technology & Software Engineering
Auckland University of Technology, Auckland, New Zealand
roopak.sinha@aut.ac.nz

Abstract. The brisk pace of the growth in embedded technology depends largely on how fast we can write and maintain software contained within embedded devices. Every enterprise seeks to improve its productivity through maintainability. While many avenues for improvement exist, highly maintainable code bases that can stay that way over a long time are rare. This article proposes a reference software architecture for embedded systems aimed at improving long-term maintainability. This reference architecture, called the Abstraction Layered Architecture (ALA), is built on the existing body of knowledge in software architecture and more than two decades of experience in designing embedded software at Tru-Test Group, New Zealand. ALA can be used for almost any object-oriented software project, and strongly supports domain-specific abstractions such as those found in most embedded software.

Keywords: software architecture, maintainability, readability, reusability, embedded software, embedded systems

1 Introduction

Tru-Test Group (henceforth, Tru-Test) is a New Zealand based company which manufactures numerous embedded solutions for livestock management, with many code bases existing for well over 20 years. A closer inspection of these code bases revealed useful insights into how some architectural practices can lead to better maintainability and lower complexity. While many code bases at Tru-Test gradually unravelled into *big balls of mud* [5] and some of these had to be abandoned, a few non-trivial examples thrived despite ongoing long-term maintenance. In fact these software parts had undergone regular maintenance for many years with almost trivial effort. Our perception was that they were two orders of magnitude easier to maintain than our worst code bases. This paper reports our attempt to uncover what makes software more maintainable, and to then integrate our findings into a reference architecture that can be used for future development.

It is said that 90% of commercial software is under maintenance [13], so any improvements here can provide high rewards. Maintainable software is easier to update and extend, which helps a company's profitability by reducing ongoing software development costs. A review of maintainable code bases at Tru-Test found that the many accepted software engineering *best practices* were helpful but not sufficient by themselves. Code-level practices (like clear naming, appropriate commenting, coding conventions, low cyclomatic complexity, etc.), module-level practices (encapsulation, programming to interfaces, etc.) and design-level practices (separation of concerns using design patterns like dependency injection, using object-oriented design, etc.) are all useful. However, individually they concern themselves with relative micro-structures within software code. The more maintainable code bases also featured robust *in-the-large* architectures. This paper focuses on architecture level interventions, which relate to high-level design decisions, structures, and constraints that, if followed, can achieve measurable improvements in maintainability.

For a developer already juggling a large set of requirements, quality attributes and deadlines, coming to a solution that also satisfies a large set of principles is often impossible. We hypothesize that it is possible to emerge a reference architecture that satisfies the principles of maintainable software without knowing the requirements, and that using this reference architecture is significantly easier than trying to satisfy all the maintainability and complexity principles concurrently. This hypothesis was broken into three research questions, as follows, leading to the main contributions of this article:

- RQ1 *What are the key system, sub-system and code-level practices that improve maintainability?* This sets the foundation for this work - we reuse and build on existing insights into writing maintainable software and consciously and deliberately avoid inventing new names for known terms. Sec. 2 provides a summary of these principles for writing maintainable code.
- RQ2 *How, and to what extent, can the practices identified in RQ1 be used to emerge a reference architecture?* This part of the research involves the creation of a reference software architecture that optimises maintainability and complexity. The creation of this proposed architecture, called *abstraction layered architecture* (ALA) is covered in Sec. 3.
- RQ3 *How can we evaluate the impact of the architecture proposed in RQ2 on maintainability?* We test the impact of ALA on software maintainability through both a re-architecting of the code base of an existing commercial product from Tru-Test, and through the addition of more features to the product. ALA shows measurable improvements in maintainability relating to all its sub-characteristics as listed by ISO/IEC 25010. The evaluation phase is described in Sec. 4.

2 Principles for Writing Maintainable Software

The principles listed in this section may not constitute an exhaustive list, but have been found to be the most important for writing maintainable software.

These principles were identified primarily through an internal review of all code bases at Tru-Test for identifying the key qualities of code-bases that remained robustly maintainable over the long-term. We also carried out a subsequent literature search for identifying design and development techniques and practices useful for writing maintainable code. At the conclusion of these investigations, we identified the following principles, which are listed in no particular order.

P1–The first few strokes: Christopher Alexander, the creator of the idea of design patterns in architecture states, "As any designer will tell you, it is the first steps in a design process which count for the most. The first few strokes which create the form, carry within them the destiny of the rest." [1].

The primary criteria for logically decomposing a system into discrete parts is well known to have a high impact on maintainability [10]. An "Iteration Zero" (the first Agile iteration) is needed to create the primary decomposition. It will not emerge from refactoring.

P2–Abstraction: Ultimately the only way of achieving knowledge separation is abstraction [14]. An abstraction is the brain's version of a module. It is the means we use to make sense of an otherwise massively complex world and it is the only means of making sense of any non-trivial software system. A great abstraction makes the two sides completely different worlds. A clock is a great abstraction. On one side is the world of cog wheels. On the other someone trying to be on time in his busy daily schedule. Neither knows anything about the details of the other. SQL is another great abstraction. On one side is the world of fast algorithms. On the other is finding all the orders for a particular customer. How about a domain abstraction, the calculation of loan repayments. On one side, the world of mathematics with the derivation and implementation of a formula. On the other the code is about a person wanting to know if they can afford to buy a house. If abstractions do not separate two different worlds like this, then we are probably just factoring out common code. We need to find the abstraction in that common code, and make it separate out something complicated which is really easy to use, like a clock.

P3–Knowledge Dependencies: The dependencies that matter are "knowledge dependencies" [3], not runtime dependencies [9]. Knowledge dependencies occur at code design-time (code read time, code write time). In order to understand and maintain a module, what knowledge do you need? Run-time dependencies are not important - they can go in any direction, and be circular. Often runtime dependencies in code are implemented as knowledge dependencies, destroying the abstractions.

P4–Zero Coupling: The concepts of coupling and cohesion have been studied extensively in literature [12]. A common misconception is that, because components in a system must interact to do anything useful, they must, at the least, be loosely coupled. The confusion arises from the use of the words 'dependency', or 'uses' for both runtime and design-time (knowledge) dependencies as noted in P3. It is important that runtime dependencies are always implemented completely inside an abstraction. For example, let's say abstractions A and B will exchange data at runtime. There must be an abstraction C that knows about the runtime dependency, and, for example, instantiates A and B and uses dependency injection to connect them. A and B must know zero about each other. Not only do A and B remain mutually zero coupled, the knowledge inside C is also mutually zero coupled with the knowledge inside both A and B. The only coupling remaining is the necessary knowledge coupling inside C on the abstractions A and B.

P5–Composition not Collaboration: In the example in **P4** above, C is a composition of A and B. Ultimately, composition is the only necessary relationship between abstractions of an architecture. Often architectures are described with components and connectors. The connector is often a runtime dependency. Thinking of components A and B as connected will induce us to let A or B have knowledge of each other. If A and B collaborate, however subtly, there will be a detrimental knowledge dependency between them, which will destroy them as abstractions. This is especially problematical when there is only one instance of each component. The lack of reuse makes it less likely to think of them as knowledge independent abstractions. Whenever we draw two components and connect them with a line, we should think of that as shorthand for two composition relationships. The drawing of instances of A and B connected by a line is just code completely contained inside C. From the point of view of A, B and C and all other abstractions in the system, the only relationship between them should be composition.

P6–Layers: Layers provide a framework for controlling dependencies. They should obviously be down the chosen layers, not across or within a layer and certainly not upwards.

Following on from principle **P3**, the only dependencies allowed are knowledge dependencies. This significantly changes how we do layering. Layering should only reflect the design time view. It should not contain layers based on runtime dependencies. Apart from [11], the layering metaphor is frequently used to represent runtime dependencies. For example, layering schemes such as GUI / Business logic / database, 3-tier, the OSI communications model are all based on runtime dependencies. Those dependencies run both ways. For instance, at run-time a database on its own is just as useless as a GUI on its own, and data will flow in both directions. To fit these systems into knowledge layers, they need to be rotated ninety degrees. Now the metaphor for them becomes a chain. Their component abstractions would generally all go into one layer, like A and B in our previous examples. One additional abstraction, like C in our previous example, would go in a higher layer. It would instantiate the required abstractions for a given application, configure them and connect them together.

P7–Stable Dependencies Principle: From $P5$ and $P6$ we have abstractions arranged in layers connected only by composition relationships going down. Ripple effects of change, are now confined to these composition relationships. To reduce the likelihood of the ripple effects, we reduce the likelihood of changing abstractions in lower layers. There is a relationship between abstraction, stability and reuse in that they tend to increase together. The lower layers should have increasing stability, and therefore increasing abstraction and reuse [8]. In higher layers, the abstractions are more specific so that is where the majority of change will be. All knowledge specific to the application requirements or other changeable things such as hardware are put in the highest layer abstractions.

P8–Abstraction granularity: There is a threshold point that should occur at about 100 to 500 lines of code that relates to our brain's capacity to handle complexity. Abstractions larger than this size may be too complex and need decomposing. If the average size is too small, abstractions will become numerous, again increasing the complexity.

P9–Primary separation - requirements from implementation: The first division line of decomposition is to separate requirements from implementation. This is the same principle used by DSLs. The requirements are expressed, succinctly, in terms of domain abstractions that you invent. Only internal DSLs are used (we don't want the disadvantages that external DSLs entail). The representation of the requirements knows nothing of the implementation and the implementation knows nothing of the requirements. Both depend on abstractions. The representation of requirements may typically take only about 1% of the total code.

P10–Fluent expression of Requirements: Maintainability is directly proportional to the ease with which new or changed requirements can be implemented into an existing system. More maintainable code bases allow requirements and the top-level application code that expresses them, to have a high degree of one to one correlation.

P11–Diagrams: Architectures must distill out details. We make a distinction between the use of Diagrams and Models (or boxes and lines). Models, as we define them, can leave out details arbitrarily, and these details can turn out to be important at the architectural level. Diagrams, as we define them, can only leave out details inside abstractions. Diagrams are therefore protected from change caused by the details. Diagrams are also executable. Diagrams are true source code.

Models should not be used as documentation of the large-scale structure of our code, as in for example an informal UML model. That would mean that the actual large scale-structure of the code is implicit and distributed in the detailed code. The structure should be explicit and in one place.

Diagrams and text are tools for different situations. Text is better for representing linear chains of relationships, or small tree structures that can be represented through indenting. Diagrams are better in situations where there are arbitrary relationships between the elements, such as in state charts.

The lines on a diagram show the connections and the structure visually. The lines also do it anonymously - without use of identifiers that you would otherwise need to do searches on to find the connections. Diagrams also provide an alternative and much better way to control scope than encapsulation does. Encapsulation is not particularly visible at read-time, and limits scope only to a boundary. A line on a diagram explicitly limits the scope to only those places where it connects.

Existing literature presents architectural tactics to deal with only some of these principles, but we still lack a cohesive reference architecture like ALA for achieving maintainability by design. Standards such as ISO/IEC 25010 define maintainability and its sub-characteristics [6]. Other works, such as the Architecture-Level Modifiability Analysis (ALMA) provide a way to evaluate a given architecture for maintainability [2]. ALMA and ALA both have the same goal. ALMA uses change scenarios to evaluate modifiability of a given architecture. ALA is a reference architecture that is pre-optimized with respect to modifiability. A loose analogy would be solving a mathematical equation. ALMA is analogous to a numerical technique whereas ALA is analogous to a symbolic technique. ALMA requires iteration to find an optimal solution. ALA solves for the optimal solution directly. That solution is the reference architecture. ALMA measures the quantity of interest, modifiability, directly and does so in the context of a domain, so is potentially more accurate (after some iterations). ALA makes the assumption that because the reference architecture satisfies the stated modifiability principles, modifiability is already optimized. The two approaches are complimentary. Compare modifiability with dependability (correctness). The two fundamental techniques here are understandability and testing. The developer first creates code that should be correct by understanding it, and then tests if it is actually correct by testing it. Using one without the other would not work well. Similarly ALA provides an architecture that should be modifiable, but still needs testing that it is actually modifiable.

3 Abstraction Layered Architecture

Abstraction Layered Architecture (ALA) was documented using the Software Architecture Documentation (SAD) process and template [4]. The following subsections follow the structure provided by SAD, and we highlight the key aspects of each part of the overall architecture document.

3.1 Architecture Background and Drivers

ALA is geared towards making embedded code more maintainable. Embedded code bases often contain entities (objects or components) which integrate different programming paradigms like logical, event and navigation flow together. More generally, we consider any object-oriented system written using any language which contains some degree of control or data flow and user interactions. For pure algorithmic problems, like those that essentially carry out sequential and nested function calls, ALA reduces to the well known functional decomposition strategy for functional programs, but adds emphasis on creating functions at discrete abstraction layers. We identify the following *architectural drivers*, based on the sub-characteristics of maintainability as per ISO/IEC 25010 [6]:

– *Modularity* is the degree to which parts of the system are discrete or independent. It depends on the coupling between components, calculated as the

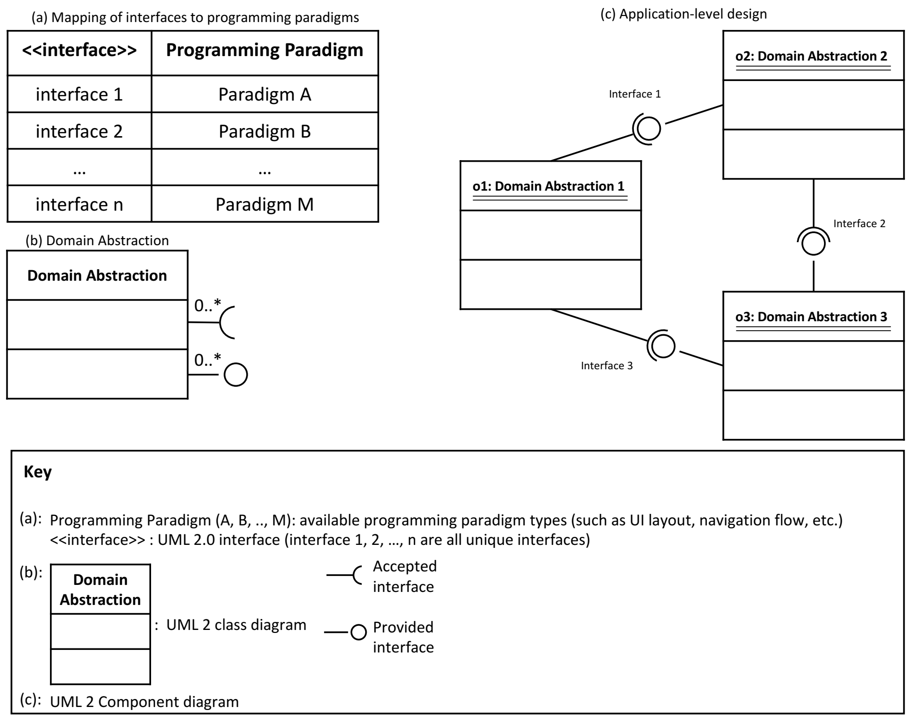
ratio between the number of components that do not affect other components and the number of components specified to be independent. It also requires each component to have acceptable cyclomatic complexity.
- *Reusability* relates to the degree to which an asset within one component or system can be used to build other components and/or systems. Reusability depends on the ratio of reusable assets to total assets, as well as the relative number of assets conforming to agreed coding rules.
- *Analysability* is the degree to which we can assess the impact of localized changes within the system to other parts of the system, or identifying individual parts for deficiencies or failures. Analysability depends on the relative numbers of logs in the system, and suitability and proportion of diagnosis functions that meet causal analysis requirements.
- *Modifiability* is the ease at which a part of the system can be modified without degrading existing product quality. It depends on the time taken for modifications themselves, and having measures to check the correctness of implemented modifications within a defined period.
- *Testability* relates to the ability to easily test a system or any part of it. It depends on the proportion of implemented test systems, how independently software can be tested, and how easily tests can be restarted after maintenance.

3.2 Views

We use the *4+1* model of documenting a reference software architecture [7]. The *logical view*, which decomposes the overall code base into smaller packages, is the most important aspect of ALA due to its direct impact on maintainability. The other views are also affected and are discussed briefly after we present the logical view.

Logical View Fig. 1 shows a representation of the top layer of ALA. Fig. 1(a) shows ALA's focus on the creation of clear interfaces which conform to specific programming paradigms. For instance, we can have explicit, named interfaces for data flow, event flow and navigation flow in a system. Most embedded code bases would benefit from multiple programming paradigms meshed together, and this mapping of interfaces to programming paradigms provides clarity in their use during the creation and maintenance of the application.

Fig. 1(b) introduces the concept of a *domain abstraction.* In general, a domain abstraction is a class which explicitly uses named interfaces, selected from the list of available interfaces in Fig. 1(b). A domain abstraction can *accept* an interface, or *provide* an interface, consistent with UML class and component diagrams. Interfaces do not need to be one way. For instance, an interface accepted by a class may not necessarily feed data into the class, and can also receive data. However, the two kinds of interfaces can help in understanding the general flow of data at the application-level (Fig. 1(c)). Another point to note is that domain abstractions can have multiple interfaces and can within themselves use several



<<interface>>	Programming Paradigm
interface 1	Paradigm A
interface 2	Paradigm B
...	...
interface n	Paradigm M

(a): Programming Paradigm (A, B, .., M): available programming paradigm types (such as UI layout, navigation flow, etc.)
<<interface>> : UML 2.0 interface (interface 1, 2, ..., n are all unique interfaces)

(b): Domain Abstraction : UML 2 class diagram

(c): UML 2 Component diagram

Fig. 1. Primary Representation of the Logical View in ALA

programming paradigms represented by these interfaces. This is in line with the tight coupling between aspects like event flow and navigation flow in a code base.

Fig. 1(c) shows the top-level application code. This is a UML Component diagram containing objects of named domain abstractions, connected or *wired* using compatible interfaces. The idea here is to allow the top-level application design to closely mimic functional requirements. Then, carefully chosen domain abstraction instances can simply be wired or re-wired together as needed. In all, ALA proposes the following four layers, (illustrated in Fig. 2):

1. *Application layer*, as shown in Fig. 1(c), contains knowledge of a specific application, no more and no less. Each requirement or feature of the application is succinctly represented by instantiating and wiring together the objects of domain abstractions defined in the second layer.
2. *Domain Abstractions layer* contains all knowledge specific to the domain, like the domain abstractions shown in Fig. 1. A domain may roughly equate to a company. Its abstractions are reusable across all potential applications in the domain.
3. *Framework layer* contains all knowledge of programming paradigms and their associated frameworks and the interfaces shared by domain abstractions. This layer also abstracts out how the Domain Abstraction layer and

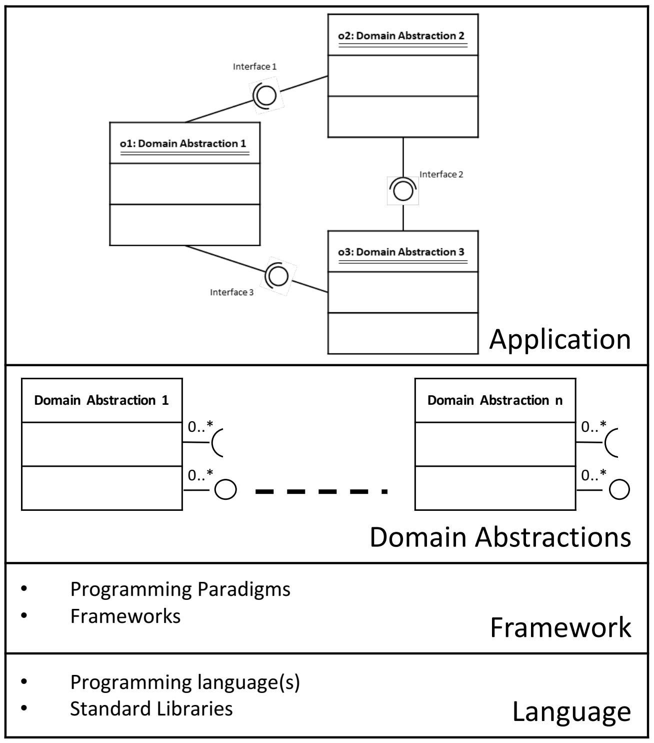


Fig. 2. The four layers in ALA

Application layer execute. For example, a common execution model framework here is 'Event driven'. A common service is a timer service. This layer contains knowledge and services that have potential to be more widely applicable than the domain layer, and consequently are more abstract, stable and reusable than those in the domain layer. We may not have to write anything in this layer ourselves as its ubiquity means that someone else may have already done it. The Applications and Domain Abstractions will only need to change if we change the programming paradigm or service abstractions.

4. *Language layer* contributes the most generic knowledge, that of the programming language(s) and associated libraries. This layer is included for completeness, but it is so generic, reusable and stable that we would never implement it for ourselves. We would always just choose the language(s) suitable for the types of Applications, Domain Abstractions and Frameworks we are going to make. All of those higher three layers will have knowledge dependencies on this language choice, but if the language is stable, those knowledge dependencies should never be a problem.

These layers are adopted from a similar set of layers proposed in [11]. The layers are relatively discrete, meaning that ideally each layer would be roughly an order of magnitude more abstract, more stable and more reusable than the one immediately above it. Having said that, code contained within a layer need not be completely flat. For instance, in the domain abstractions layer, we can have intra-layer hierarchies where abstractions could be built using local compositions.

Four major layers may seem like a small number. But note that the human brain can be built from just six composition layers - (protons, electrons, etc), atoms, (protein molecules), (cells or neurons), neural net structures, brain. Sometimes an additional layer may be needed. For example, a features layer could be introduced between the Application and Domain layers. A given application is then a composition of features.

Development view The development view constrains the process of designing and developing a system. ALA requires significant up-front design, in which the domain abstractions are identified from all known functional requirements. The need for some upfront design puts us clearly outside the camp of the agile purists who might say that the design will emerge over time, and clearly in the camp of the iteration zeroists. After this zero-th sprint spent on design, most of the domain abstractions will be known and any remaining architectural design can be done iteratively, but it remains deliberate and emergence is not encouraged. In ALA, the first application design involves taking one requirement at a time and writing it in terms of suitably invented domain abstractions, until all known requirements have been designed. In this respect the Domain Abstractions together with their shared interfaces form a DSL for concisely implementing requirements. The shared interfaces of the domain abstractions define the grammar. In terms of elements, form and rationale, Domain Abstractions are the elements, the shared interfaces give the form, and this paper provides the rationale.

ALA requires two skill levels. It needs the skills of a software architect competent with all the principles outlined in this article, for the architectural design and the on-going architectural refactoring. It then requires only average development skills for coding the domain abstractions and interfaces, as these are already stand alone. TDD suddenly starts to work well here as contractors can be used for the development roles, because they need to know only about the abstractions they work on. When they go, they will not take any other knowledge with them.

Most modifications to a mature system usually only affect the top layer. The top layer will typically contain only 1 to 10% of the total code. Addition of new functionality may require introducing or generalizing domain abstractions.

Process View The process view is concerned with the runtime structure of a code base. ALA supports both single and multi-process/threaded systems due to its emphasis on ensuring that domain abstraction instances are wired through using the right interfaces logically. How these instances and objects bind at run time is a decision that can be taken later.

Physical view The physical view allows mapping software to resources like available hardware. ALA does not explicitly constrain the physical view, but the application-level design shown in Fig. 1(c) can be modified to annotate where each part of the diagram executes.

4 Evaluating ALA

We carry out two kinds of evaluations for ALA. Firstly, at the architectural level, we identify the mechanisms that ALA provides for supporting each of the quality attributes identified in Sec. 3.1. These mechanisms are listed in Tab. 1. Overall, as can also be seen in Fig. 1(c), ALA supports our original goal of ensuring functional requirements can be mapped onto the application level in a one-to-one manner (research question RQ2). The second set of evaluations were based on using ALA on a Tru-Test product. These experiments are described in the following subsections.

4.1 Re-architecting an existing product

We chose to re-architect the XR5000, shown in Fig. 3, a hand-held embedded device used for managing several activities on a dairy farm. The device features a number of soft-keys for user actions. The user action associated with a soft-key depends on which screen is currently active. The XR5000 is the latest in a family of such devices produced by Tru-Test, and the code base for the product has been maintained and modified over many years. The XR5000 legacy code base represented a common "big ball of mud" scenario. It contained approximately 200 KLOC. It had taken 3 people 4 years to complete. One additional feature (to do with animal treatments) had taken an additional 3 months to complete - indicative of the typical increasing cost of incremental maintenance for a code base of this type.

For re-architecting this product using ALA, we first did an 'Iteration Zero' (two weeks) to represent most of the requirements of the XR5000. This produced an application diagram with around 2000 nodes. Fig. 4 shows a part of the application diagram. Tab. 2 shows the various kinds of interfaces used and their associated programming paradigms as per Fig. 1.

The size of the diagram was interesting in itself. The actual representation of requirements was about 1% of the size of the legacy code. The nodes were instances of around 50 invented domain abstractions. The diagram was not a model in that it was, in theory, executable. Most requirements were surprisingly easy to represent at the application level. There were occasional hiccups that took several hours to resolve, but as more abstractions were brought into play, large areas of functionality would become trivial to represent. This was a positive beginning.

Table 1. ALA's support for Maintainability Sub-Characteristics as per ISO/IEC 25010

QA	**ALA mechanisms**
Modularity	The solution consists entirely of modules (that are abstractions). No module need be large because it can always be broken up into a composition of other abstractions. Cyclomatic complexity can be dealt by hierarchical layer-based decompositions Cyclomatic complexity is reduced because modules based on abstractions naturally have a single responsibility. Upfront design ensures high cohesion within domain abstractions
Reusability	Reusability increases typically by an order of magnitude as we go down each layer Two layers are dedicated to two levels of reuse, layer 2 for reuse at the domain level, and layer 3 for reuse at the programming paradigm level Interfaces and domain abstractions are reusable types Domain abstractions conform to coding rules via interfaces The interfaces that exist for connecting domain abstractions are at the reuse level (and abstraction level) of the framework layer.
Analysability	Any piece of code, being inside an abstraction, is small and coherent in itself, and the only external knowledge dependencies needed to understand it are on abstractions in lower layers. These abstraction dependencies, being composition relationships, are necessary to the meaning of the higher layer abstraction content.
Modifiability	The knowledge contained inside abstractions tends to be naturally cohesive and therefore easy to change. The knowledge contained inside abstractions is zero-coupled with that in all other abstractions - zero ripple effects. Dependencies are restricted to true knowledge dependencies, so zero ripple effects from run-time dependencies. All Dependencies are composition relationships on abstractions, (not their contained knowledge) so ripple effects occur only if the nature of the abstraction itself changes. Abstractions tend to be naturally stable entities - reducing ripple effects. Abstractions are an order of magnitude more stable in a lower layer, further reducing ripple effects.
Testability	All abstractions can be tested individually within a layer because they are already zero coupled with their peers. Testing mocks can easily be wired to them. Inter-working of domain abstractions can be tested with straightforward integration tests by wiring each possible combination of abstraction. Higher layer abstractions are generally tested with their composition of lower layer abstractions intact. Automated acceptance testing via the external interfaces is not significantly easier in ALA as the system appears as a black box to these type of tests. However 'under the skin' acceptance testing can be easier because all I/O abstractions can be replaced by wiring in modified versions that can mock the hardware instead.

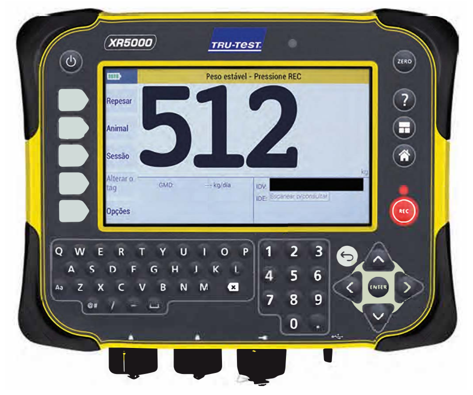


Fig. 3. The XR5000 embedded device

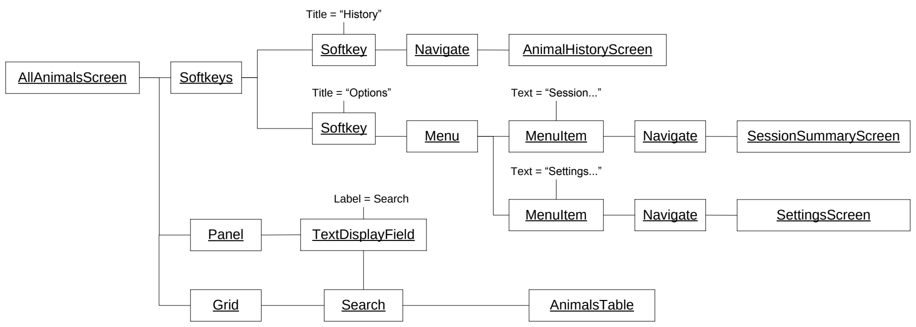


Fig. 4. Sample application-level diagram for a part of the ALA-based XR5000 code base

4.2 Adding a new feature

The diagram created during the re-architecting experiment deliberately did not include the aforementioned "treatments" feature. The next experiment was to add this feature to the application. This involved adding database tables, fields to existing tables, a settings screen, a data screen, and event-driven behaviours. The incremental time for the diagram additions was of the order of one hour. Obviously testing was needed to be considered also, and the 'Table' abstraction also needed additional work so it could migrate the data in its underlying database, a function the product had not needed up until this point. Although somewhat theoretical, the experiment was evidence to us of a potential order of magnitude improvement in incremental maintenance effort.

The big question now was, could the application diagram be made to actually execute? Fortunately we were allowed to fund a summer undergraduate student for 3 months to try to answer this question. It was a simple matter to translate the application diagram into C++ code that instantiated the abstrac-

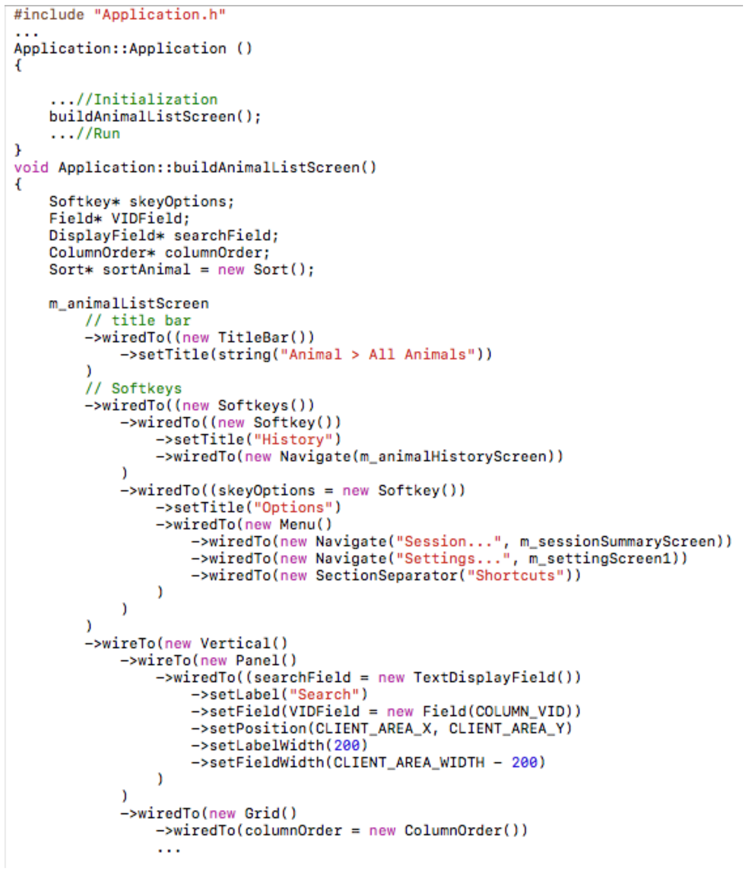
Table 2. Mapping of interfaces to programming paradigms for the XR5000

Interface(s)	**Programming paradigm**
IUiLayout, IMenuItem	UI layout
IDestination	Navigation flow
IEventHandler	Reactive
ITable	Data flow or Stream
iAction	Activity flow

tions (classes), wire them together using dependency injection setters, configure the instances using some more setters, and use the fluent interface pattern to make all this straightforward and elegant. As an example, the wired code for the diagram sample shown in Fig. 4 is shown in Fig. 5. Thanks to the composability offered by the interfaces of the domain abstractions, wiring instances in code follows exactly the same structure as the application diagram. We have omitted the interface types and kinds (provided or accepted) since we can only legally connect two instances through compatible interfaces. Also, the distinction between provided and accepted interfaces is more useful when defining the domain abstractions, and not so much during the wiring of their objects because both kinds of interfaces allow bidirectional flow of information.

The student's job was to write the classes for 12 of the 50 abstractions in the application. These 12 were the ones needed to make one of the screens of the device fully functional. The initial brief was to make the new code work alongside the old code, (as would be needed for an incremental legacy rewrite) but the old code was consuming too much time to integrate with so this part was abandoned. The learning curve for the student was managed using daily code inspections, explaining to him where it violated the ALA principles, and asking him to rework that code for the next day. It was his job to invent the methods he needed in the interfaces between his classes to make the system work, but at the same time give no class any knowledge of the classes it was potentially communicating with. It took about one month for him to fully "get" ALA and no longer need the inspections. As a point of interest, as the student completed classes, the implementation of parts of the application other than the one screen we were focused on became trivial. He could not resist making them work. For example, as soon as the 'Screen', 'Softkey' and 'Navigation action' classes were completed, he was able to have all screens displaying with soft-keys for navigating between them, literally within minutes.

The 12 classes were completed in the 3 months, giving the screen almost full functionality - showing and editing data through to an underlying database, searching, context menus, etc. Some of the 12 domain abstractions were among the most difficult needed for the XR5000, and most of the interfaces had to be designed, so there is some validity for extrapolation. Also, performance issues were considered during the implementation. For example, the logical flow of data from a Table to a Grid was actually implemented by passing a list of objects in the opposite direction that describe how the data is transformed along the



```
#include "Application.h"
...
Application::Application ()
{

    ...//Initialization
    buildAnimalListScreen();
    ...//Run
}
void Application::buildAnimalListScreen()
{
    Softkey* skeyOptions;
    Field* VIDField;
    DisplayField* searchField;
    ColumnOrder* columnOrder;
    Sort* sortAnimal = new Sort();

    m_animalListScreen
        // title bar
        ->wiredTo((new TitleBar())
            ->setTitle(string("Animal > All Animals"))
        )
        // Softkeys
        ->wiredTo((new Softkeys())
            ->wiredTo((new Softkey())
                ->setTitle("History")
                ->wiredTo(new Navigate(m_animalHistoryScreen))
            )
            ->wiredTo((skeyOptions = new Softkey())
                ->setTitle("Options")
                ->wiredTo(new Menu()
                    ->wiredTo(new Navigate("Session...", m_sessionSummaryScreen))
                    ->wiredTo(new Navigate("Settings...", m_settingScreen1))
                    ->wiredTo(new SectionSeparator("Shortcuts"))
                )
            )
        )
        ->wireTo(new Vertical()
            ->wireTo(new Panel()
                ->wiredTo((searchField = new TextDisplayField())
                    ->setLabel("Search")
                    ->setField(VIDField = new Field(COLUMN_VID))
                    ->setPosition(CLIENT_AREA_X, CLIENT_AREA_Y)
                    ->setLabelWidth(200)
                    ->setFieldWidth(CLIENT_AREA_WIDTH - 200)
                )
            )
            ->wiredTo(new Grid()
                ->wiredTo(columnOrder = new ColumnOrder())
                ...
```

Fig. 5. Code Snippet relating to Fig. 4

way. These objects are eventually turned into SQL in a 'Database interface' class within the Table abstraction. We can estimate that the 50 classes may have taken about one man-year to complete for the student. This compares with the 12 man-years to complete the original, conventionally written code. An interesting observation is that the original architecture diagram did not need to change as a result of the implementation of its composite abstractions.

5 Concluding Remarks

Abstraction Layered Architecture or ALA is an attempt to integrate principles that seem to produce code bases that are easy to maintain over a long time. These principles were identified via a review of Tru-Test code bases, both successful or unsuccessful from a maintenance point of view, and supplemented by a review of existing literature on this subject. This set of principles was then used to emerge a reference architecture based on layering of abstractions. We later show how ALA meets the key sub-characteristics of maintainability as per ISO/IEC 25010. More importantly, we show how an existing product at Tru-Test was re-architected and extended using ALA to produce a more maintainable and compact code base in a fraction of the time it took for the original code base.

This paper opens up several exciting directions for future research. We aim to continue developing ALA to incorporate other practices for maintainability, several of which are becoming more apparent as Tru-Test's software operations scale up. Investigating the use of ALA in non-embedded code bases such as for enterprise systems, and gathering empirical data on its effectiveness are some other future directions.

References

1. Alexander, C.: The nature of order: the process of creating life. Taylor & Francis (2002)
2. Bengtsson, P., Lassing, N., Bosch, J., van Vliet, H.: Architecture-level modifiability analysis (alma). Journal of Systems and Software 69(1-2), 129–147 (2004)
3. Cataldo, M., Mockus, A., Roberts, J.A., Herbsleb, J.D.: Software dependencies, work dependencies, and their impact on failures. IEEE Transactions on Software Engineering 35(6), 864–878 (2009)
4. Clements, P., Garlan, D., Little, R., Nord, R., Stafford, J.: Documenting software architectures: views and beyond. In: Proceedings of the 25th International Conference on Software Engineering. pp. 740–741. IEEE Computer Society (2003)
5. Foote, B., Yoder, J.: Big ball of mud. Pattern languages of program design 4, 654–692 (1997)
6. ISO/IEC: ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Tech. rep. (2011)
7. Kruchten, P.B.: The 4+1 view model of architecture. IEEE Software 12(6), 42–50 (Nov 1995)
8. Martin, R.C.: Agile software development: principles, patterns, and practices. Prentice Hall (2002)
9. Nicolau, A.: Run-time disambiguation: coping with statically unpredictable dependencies. IEEE Transactions on Computers 38(5), 663–678 (1989)
10. Ossher, H., Tarr, P.: Using multidimensional separation of concerns to (re) shape evolving software. Communications of the ACM 44(10), 43–50 (2001)
11. Page-Jones, M., Constantine, L.L.: Fundamentals of object-oriented design in UML. Addison-Wesley Professional (2000)
12. Perepletchikov, M., Ryan, C., Frampton, K.: Cohesion metrics for predicting maintainability of service-oriented software. In: Quality Software, 2007. QSIC'07. Seventh International Conference on. pp. 328–335. IEEE (2007)
13. de Souza, S.C.B., Anquetil, N., de Oliveira, K.M.: A study of the documentation essential to software maintenance. In: Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information. pp. 68–75. ACM (2005)
14. Visser, E.: Webdsl: A case study in domain-specific language engineering. In: International Summer School on Generative and Transformational Techniques in Software Engineering. pp. 291–373. Springer (2007)