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 evolvability. While many code bases at Tru-Test gradually unravelled into *big balls of mud* [4] 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 partial 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 [12], 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. Moreover, the more maintainable code bases also featured robust *in-the-large* architectures. This paper focuses on architectural interventions, which relate to high-level design decisions, structures, and constraints that, if followed, can help achieve measurable improvements in maintainability.

We hypothesize that providing architectural support which allows functional requirements to be mapped directly into an application improves maintainability. 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 combined into a reference architecture that allows functional requirements to be mapped onto application level code in a one-to-one fashion?* This part of the research involves the creation of a reference software architecture that features high maintainability. 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 through two parallel investigations. The first investigation involved an internal review of all code bases at Tru-Test, focussed on identifying the key qualities of code-bases that remained robustly maintainable

over the long-term. The second investigation followed a systematic literature review, in which design and development techniques and practices useful for writing maintainable practices were surveyed. 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 [9]. An "Iteration Zero" (the first Agile iteration) is needed to create the primary decomposition. It will not emerge from refactoring.

P2–Coupling and High Cohesion: The concepts of coupling and cohesion tell us to view software in two different scales [11]. There is a threshold point that should occur between the two at about 10 to 100 lines of code (which relates to our brains' capacity to handle complexity). Below the threshold, code is cohesive - it all relates to each other tightly for a single responsibility. Above the threshold, they are parts that are loosely coupled.

P3–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.

P4–Dependencies: The dependencies that matter are "knowledge dependencies" [2], not runtime dependencies [8]. 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.

P5–Stable Dependencies Principle: To control the ripple effects of change, dependencies, even at a class and function level, should be in the direction of greater stability [7].

P6–Layers: Layers provide a framework for dependencies [10]. They should be down the chosen layers, not across or within a layer and certainly not upwards. This means that classes in lower layers should be more stable, and therefore more abstract (less specific to the application).

P7–Abstractions: Ultimately the only way of achieving knowledge separation is abstraction [13]. 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.

P8-Diagrams: Diagrams should not be used as an overview of the structure of your code, as in for example a UML model. That would mean that the large scale structure of the code has become buried in the details. Such diagrams leave out detail arbitrarily, and these details generally turn out to be important. Diagrams are very useful when they are true source code. When using components, a diagram is usually the best way to show the explicit wiring. The lines on the diagram show the connections and the structure explicitly. 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. Diagrams handle "unstructured" structure better than text. There are many situations, such as state machines, where the natural structure does not conform to a small tree which is the natural limit of text based programming.

P9–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.

3 Abstraction Layered Architecture

This section presents an overview of Abstraction Layered Architecture (ALA) documentation, carried out using the Software Architecture Documentation (SAD) process and template [3]. 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 [5]:

- *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 [6]. 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 the primary representation of the logical view 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, event and navigation flows in a system. Most embedded code bases would benefit from multiple programming paradigms meshed

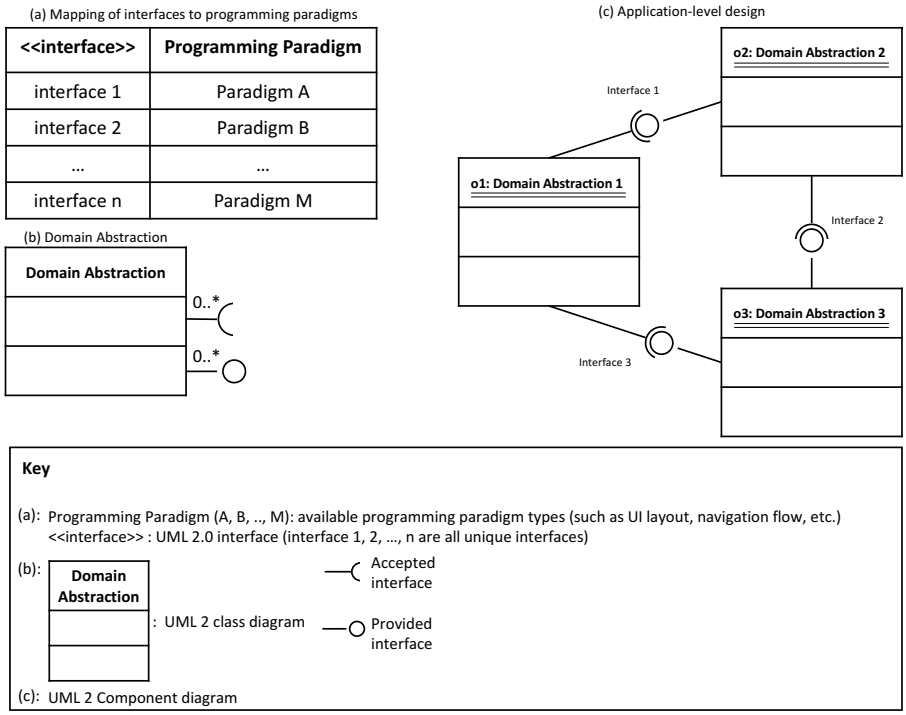

Fig. 1. Primary Representation of the Logical View in ALA

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 names its 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 information into the class, and can also receive information. However, the two kinds of interfaces can help in understanding the general flow of information at the application-level (Fig. 1(c)). Another point to note is that domain abstractions can have multiple interfaces and can within themselves combine several programming paradigms to enable these interfaces. This is in line with the tight coupling between aspects like event flow and navigation flow within an embedded 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* to each other 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, while low-level implementation details are relegated to lower implementation layers.

Fig. 1(c) shows the top-level application logic, which is supported by the *layers* below. In all, ALA proposes four layers, illustrated in Fig. 2 and listed from top to bottom as follows:

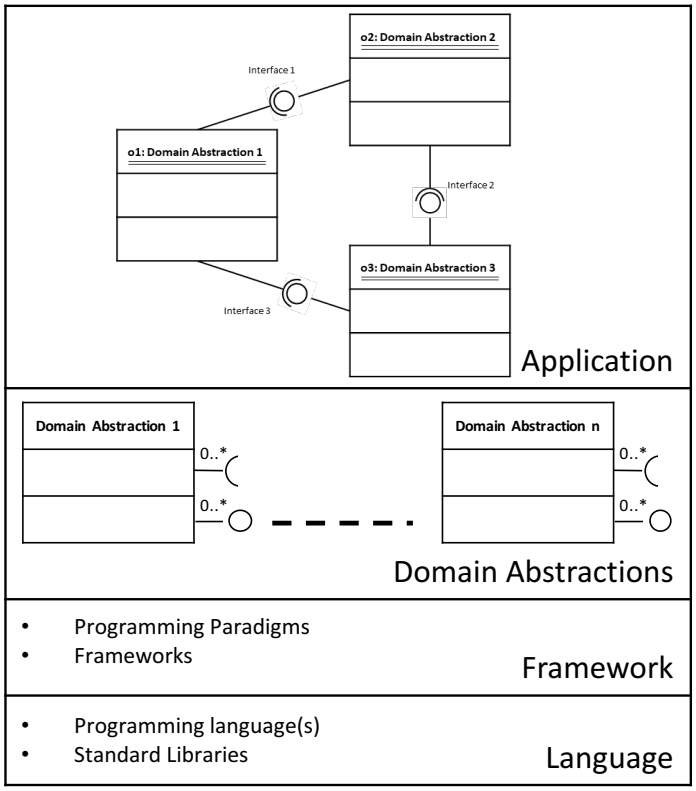

Fig. 2. The four layers in ALA

1. *Application layer*, as shown in Fig. 1(c), contains knowledge of a specific application, no more and no less. The application instantiates and wires together the objects of domain abstractions defined in the second layer.
2. *Domain Abstractions layer* contains all knowledge specific to the domain, like the implementation of the domain abstractions and interfaces shown in Fig. 1.
3. *Framework layer* contains all knowledge specific to the types of computing problem(s) a company's products solve, such as available programming paradigms and associated frameworks. This layer abstracts out how the Domain Abstraction layer and Application layer execute.

4. *Language layer* contributes the most generic knowledge, that of the programming language and associated libraries.

These layers are adopted from a similar set of layers proposed in [10]. Code contained within a layer need not be completely flat. For instance, in the domain abstractions layer, we can have intra-abstraction hierarchies where abstractions could be built using techniques like composition and decorators. However, as a principle, abstractions must have zero coupling when configured at the application level.

Development view The development view constrains the process of designing and developing a system. ALA requires significant up-front design, in which domain abstractions are identified and created through a thorough review of 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, remaining architectural design can be done iteratively, but it remains deliberate and emergence is not encouraged. In ALA, design involves taking one requirement at a time and designing for it, until all known requirements have been designed for.

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 also requires strong development skills for coding the domain abstractions and interfaces. 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, and when they go, they will not take any other knowledge with them.

Whenever new functionality needs to be added, usually at most the first two layers of the code base (see Fig. 2) will be affected. In such situations, decisions around creating domain abstractions or reusing existing ones are crucial. While reuse is encouraged, it must not be done at the expense of increasing module dependencies or polluting the interfaces. Most changes to the application are confined to the application layer, while addition of new functionality may require introducing or updating domain abstractions. In general, the lower layers are progressively more stable, each requiring significantly lesser modifications than the layer above it during maintenance.

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 qualities 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.

Fig. 3. The XR5000 embedded device

Table 1. ALA's support for Maintainability

QA	**ALA mechanisms**
Modularity	–zero coupling between layer 2 domain abstractions and layer 1 applications –zero coupling between programming paradigms (layer 3) and domain abstractions (layer 2) –cyclomatic complexity can be dealt by hierarchical layer-based decompositions –cyclometric 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 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	–requirement changes are overlaid first on the top-level application, with changes decomposed and localized to interface and domain abstraction –Use of abstractions (rather than just modules) together with the emphasis on knowledge dependencies rather than run-time dependencies
Modifiability	domain –abstractions have zero coupling domain abstractions and interfaces can be checked individually –module dependencies are always on interfaces that are at least one abstraction level more stable
Testability	–domain abstractions can be tested individually –interworking of domain abstractions can be tested with straightforward integration tests by wiring each possible combination of abstraction –application can be tested easily using mocked or modified versions of I/O abstractions

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.

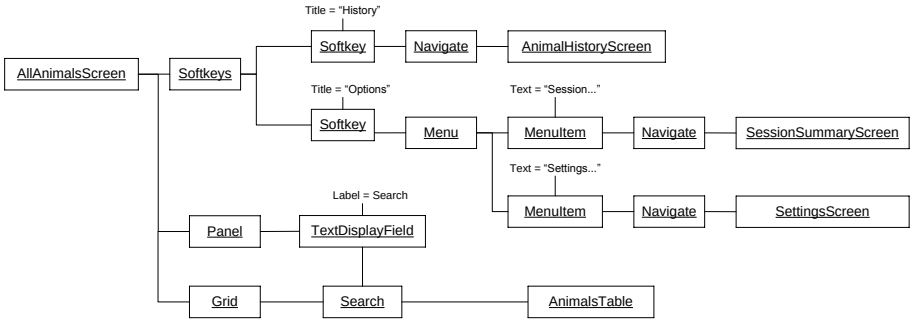

Fig. 4. Sample application-level diagram for a part of the ALA-based XR5000 code base

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

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 abstractions (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.

```
#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

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 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.

5 Concluding Remarks

Abstraction Layered Architecture or ALA is an attempt to integrate best practices that seem to produce code bases that are easy to maintain over a long time. These practices 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 practices were 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. 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)
3. 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)
4. Foote, B., Yoder, J.: Big ball of mud. Pattern languages of program design 4, 654–692 (1997)
5. 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)
6. Kruchten, P.B.: The 4+1 view model of architecture. IEEE Software 12(6), 42–50 (Nov 1995)
7. Martin, R.C.: Agile software development: principles, patterns, and practices. Prentice Hall (2002)
8. Nicolau, A.: Run-time disambiguation: coping with statically unpredictable dependencies. IEEE Transactions on Computers 38(5), 663–678 (1989)
9. Ossher, H., Tarr, P.: Using multidimensional separation of concerns to (re) shape evolving software. Communications of the ACM 44(10), 43–50 (2001)
10. Page-Jones, M., Constantine, L.L.: Fundamentals of object-oriented design in UML. Addison-Wesley Professional (2000)
11. 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)
12. 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)
13. 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)