



Closing the loop – The Influence of Code Analysis on Design

Peter Amey

Publication notes

© Springer-Verlag
Published in Lecture Notes in Computer Science 2361
J Blieberger and A Strohmeier (Eds.):
Reliable Software Technologies – Ada-Europe 2002
7th Ada-Europe International Conference, Vienna, Austria, June 2002

Closing the Loop: The Influence of Code Analysis on Design

Peter Amey

Praxis Critical Systems, 20, Manvers St., Bath BA1 1PX, UK
peter.amey@praxis-cs.co.uk

Abstract. Static code analysis originally concerned the extraction from source code of various properties of a program. Although this kind of reverse engineering approach can uncover errors that are hard to detect in other ways, it is not a very efficient use of resources because of its retrospective nature and the late error detection that results. The SPARK language and its associated Examiner tool took a different approach which emphasises error *prevention* (“correctness by construction”) rather than error *detection*. Recent work with SPARK has shown that very early application of static analysis can have a beneficial influence on software architectures and designs. The paper describes the use of SPARK to produce designs with demonstrably low coupling and high cohesion.

1 Introduction

Software development lifecycles usually make a clear distinction between *development* activities and *verification* activities. The classic V model is a good example. The distinction made some sense when verification was largely synonymous with testing since, clearly, you cannot dynamically test something until you have produced it. Unfortunately, the same thinking has affected the way that static analysis tools have been developed and deployed. There is now compelling evidence that the use of analysis tools *throughout development*, especially prior to compilation, can bring significant economic and technical benefits [1, 2]. Some of this arises straightforwardly from earlier error detection but more subtle effects involving the influence of analysis on design appear to be equally, or perhaps even more, significant. The paper explores these effects and the important benefits that can accrue.

2 Static Analysis - an Historical Overview

The early development of static analysis tools, which resulted in MALPAS [3] and SPADE [4] in the UK, was largely concerned with the reverse engineering of existing source code; this can be termed *retrospective analysis*. The reasons for this were straightforward: certification authorities and major consumers of software such as the UK Ministry of Defence, were being offered complex, software-intensive systems which came with very little evidence for their suitability and safety for deployment. These agencies either had to accept the vendors’ assurances that the system was satisfactory

or find some way of exploring and understanding the system for themselves. In this environment retrospective static analysis tools offered one of the very few ways forward. The view that analysis is something that should take place near the end of the project lifecycle prevails to this day and has been reinforced by some more recent analysis tool developments. For example ASIS-based tools perform their analyses on information supplied by the compiler; clearly the code must be compilable before the analysis can begin. Similarly, analysis based on techniques such as abstract interpretation requires a complete, linkable closure of the program to be effective; again this is only available late in the project lifecycle.

3 The Disadvantages of Retrospective Analysis

Postponing static analysis to late in the development process has both technical and economic drawbacks. Technically it tends to have only a very limited impact on the quality of the finished product. By definition it cannot influence the way the code has been designed and constructed so its role is limited to the uncovering of errors in the supposedly finished product. Even here the benefit is diminished because of the reluctance to change code which is perceived as being “finished”. Significant deficiencies, uncovered by analysis, may be allowed to remain in delivered code because the cost of correction and re-testing is considered too high. These deficiencies remain and may pose a risk to future development and maintenance activities. This reluctance to change code, despite errors being exposed by analysis, was a significant conclusion of a static analysis experience report presented at SIGAda 2000 [5]; it is an inevitable consequence of retrospective analysis approaches.

4 Correctness by Construction

In the UK, the developers of the SPADE toolset learned from its initial deployment that the real issues involved were associated with design rather than just with analysis. Well written programs were straightforward to analyse whereas badly written ones defied the most powerful techniques available. Therefore the need was not for ever more powerful analysis techniques but rather support for *constructing* correct programs. This reasoning led to SPARK [6–8] and its support tool, the SPARK Examiner. A key design feature of the SPARK language is the use of annotations to strengthen package and sub-program specifications to the point where analyses could be performed without needing access to their implementations; this allows analysis of incomplete programs. The ability to analyse program fragments allows static analysis to be deployed very early in the development process, prior to compilation, which leads to significant benefits. Some of this is very straightforward: errors detected by static analysis performed at the engineer’s terminal as he writes code are corrected immediately. They never make it to configuration control, don’t have to be found by testing and don’t generate any re-work. The economics of this are very powerful; see for example the Lockheed C130J mission computer project where error densities were reduced by an order of magnitude while, at the same time, costs were reduced by three quarters [1, 2].

More subtly, the use of analysis as an integral part of the development process alters the way in which engineers approach the writing of code. They quickly get used to the kinds of constructs that the SPARK Examiner complains about and avoid using them. They tend to write simpler, more prosaic code which is more often correct and also turns out to be easier to understand, test and maintain. These effects are most pronounced where the process involves formal verification or code proof: the style of coding soon changes to make the code easier to prove; see for example [9].

More recently, Praxis Critical Systems has begun to see ways of gaining even more leverage from the use of static analysis; this goes beyond error detection and low-level code style issues and has an impact on program architectural design itself. The approach makes use of properties measured by static analysis directly to drive design decisions. The result is significant improvements in certain important design properties.

5 SPARK Annotations

SPARK's annotations are special comments introduced by the prefix "#". They have significance to the SPARK Examiner although they are, of course, ignored by an Ada compiler. The purpose of annotations is to strengthen the specification of packages and subprograms. A full description of the system of annotations can be found in [6–8]. For the purpose of analysing design qualities the key annotations are: *own variables*, *global annotations* and *dependency relations*.

An *own variable annotation* indicates that a package contains persistent data or "state". For example:

```
package P
--# own State;
is
...
end P;
```

tells us that package P contains some persistent data that we have chosen to call `State`. The own variable `State` can be an abstraction of the actual variables (known in SPARK as *refinement constituents*) making up the package's state.

A *global annotation* indicates the direct or indirect use of data external to a subprogram and the direction of data flows involved. For example:

```
procedure K (X : in Integer);
--# global in out P.State;
```

tells us that the execution of procedure K causes the own variable `State` in package P to be both referenced and updated.

Finally, a *dependency relation* indicates the information flow dependencies of a subprogram, i.e. which *imported* variables are used to obtain the final values of each *exported* variable. For example:

```
procedure K (X : in Integer);  
--# global in out P.State;  
--# derives P.State from P.State, X;
```

tells that the final value of `P.State` after execution of `K` is some function of its initial value and of the parameter `X`. Other annotations can provide a more precise relationship between imports and exports for proof purposes but these are beyond the scope of this paper.

A less significant annotation, *inherit*, governs inter-package visibility. For the purposes of this paper it can be considered to be the transitive closure of a package's *with* clauses; *inherit* shows direct and indirect package use whereas *with* shows only direct use.

SPARK annotations provide a strong indication of the degree of coupling between objects. A data item declared inside a subprogram does not appear in its annotations at all (even if it is of some complex abstract type declared in another package); a parameter appears only in the *derives* annotation; and data manipulated globally appears in both the *global* and *derives* annotation.

6 Desirable Design Properties

Well designed programs have a number of heavily interconnected properties that are described below. A good design will strike a balance between these various characteristics and extremes should be treated with suspicion. For example a design which placed the entire program in a single procedure might score well on encapsulation but would be very poor when measured for cohesion!

Encapsulation Encapsulation is the clear separation of specification from implementation; this is sometimes described as a “contract model” of programming. It is an important principle that users of an object should not be concerned with its internal behaviour. Were this not the case then loose coupling could not be achieved. The principle of encapsulation applies not only to data and operations but even to type declarations: for example, a limited private type provides more encapsulation than a full Ada type declaration.

Abstraction Abstraction is a necessary component of encapsulation. Abstraction allows us to strip away (ignore) certain levels of detail when taking an external view of an object; the detail is hidden by being encapsulated in the object. The term “information hiding” is frequently used in this context. This term is somewhat inappropriate for critical systems about which reasoning is important: information, by definition, *informs*. We cannot reason in the absence of information; however, we can ensure that our reasoning takes place at an appropriate level of abstraction, which we achieve by hiding *detail*. Hiding unnecessary detail allows us to focus on the *essential* properties of an object. The essential properties are those which support encapsulation by allowing use of an object without the need to be concerned with its implementation.

Loose coupling Coupling is a measure of the connections between objects. Highly coupled objects interact in ways that make their separate modification difficult.

Undesirable, high levels of coupling arise when abstractions are poor and encapsulation inadequate. Software components can be strongly or weakly coupled. The OOD literature is not entirely consistent as to which forms of coupling are weak and which are strong and therefore less desirable. SPARK provides a simple and clear distinction: the appearance of a package name only as a prefix in an *inherits* annotation represents weak coupling (use of a service) but its appearance in a *global* or *derives* annotation indicates strong coupling (sharing of data).

Cohesion Whereas coupling is measured *between* objects, cohesion is a property *of* an object. Cohesion is a measure of focus or singleness of purpose. For example, a car has both door handles and pistons but we would not expect to find both represented by a single software object. If they were, we would not have high cohesion and modifying the software to support a 2-door rather than 4-door model (or to replace a Straight-4 with a Vee-8 engine) would involve changing rather unexpected parts of the design. There is a clear distinction between the isolation provided by highly-cohesive objects and the need to arrange such objects in hierarchies as described below.

Hierarchy Here we recognise that in the real-world objects exhibit hierarchy. Certain objects are contained inside others and cannot be reached directly. When we approach a car we can grasp a door handle but not a piston: the latter is inside the engine object which is itself inside the car. Many OOD methods are prone to producing unduly flat networks of objects (that would make door handles and pistons of equal prominence) which can easily encourage extra, undesirable, coupling between objects.

7 The Influence of Analysis on Design

To produce a “good” design, we seek to optimise the properties described above. This is where life gets difficult: we can all agree that loose coupling is desirable but how do we measure it and achieve it? Here the analysis performed by tools such as the SPARK Examiner is a direct help. For example, the Examiner performs *information flow analysis* [10] and this is a direct and sensitive measure of coupling between program units. High coupling manifests itself as large SPARK flow annotations (*global* and *derives* annotations) and loose coupling by small ones. We can seek to manipulate the design so that the annotations are in their smallest and most manageable form. This results in loose coupling and is consistent with the observation that well written programs are easy to analyse. Information flows, and coupling, are a consequence of where “state”—persistent data—is located in the program’s design. Values of state variables have to be calculated and set using information from other parts of the system and current values of state have to be conveyed from their location to the places which need those values; these are information flows. Information flows lead to couplings between components. Again the language rules of SPARK and the analysis performed by the Examiner come to our aid: own variable annotations describe the presence of state variables even though they are concealed according to the rules of Ada; this makes it possible to reason about them. There are also rules which govern the permitted refinements of abstract state variables on to their concrete constituents.

8 Information Flow Driven Design

The information obtained from analysis makes it possible to steer designs in the direction of highest cohesion and loosest coupling by constantly seeking to minimise information flow. There must, of course, be some information flow or the program would do nothing, but what we are seeking is to limit the flow to that essential to perform the desired task and not allow it to be dominated by extraneous flows between objects that a better design would eliminate. The control that allows information flow to be minimised is the location and choice of abstraction of system state. We can steer the program design in the required direction in two ways. The first is to make use of knowledge of how the analysis works to predict the kinds of designs that will work best and incorporate this knowledge into a set of design heuristics. The second is to carry out static analysis of the emerging system early (and often) and to be ready to refactor the design as soon as undesirable information flows—and hence coupling—start to emerge. This information flow driven design method is outlined below; a fuller description can be found in [11]. An approach that meets these goals has the following steps:

1. Identification of the system boundary, inputs and outputs.
 - (a) Identify the boundary of the *system* for which the software is intended.
 - (b) Identify the *physical* inputs and outputs of the system.
2. Identification of the SPARK software boundary within the overall system boundary.
 - (a) Select a *software* boundary within the overall system boundary; this defines the parts of the system that will be statically analysed.
 - (b) Define boundary packages to give controlled interfaces across the software boundary and translate physical entities into problem domain terms. These packages, to use the terminology of Parnas and Madey [12], translate *monitored variables* into *input data items* and *output data items* into *controlled variables*.
3. Identification and localization of system state.
 - (a) Identify the essential state of the system; what *must* be stored?
 - (b) Decide in what terms we wish the SPARK information flow annotation of the main program to be expressed. Any state located outside the main program will appear in its annotations, any local to it will not. We seek here an annotation that describes the desired behaviour of the system in terms of its problem domain rather than simply a description of a particular software solution. See [13] for examples of annotations as a *system* rather than *software* description.
 - (c) Using these considerations, assess where state should be located and whether it should be implemented as abstract state machine packages or instances of abstract data types.
 - (d) Identify any natural state hierarchies and use refinement to model them.
 - (e) Test to see if the resulting provisional design is a loop-free partial ordering of packages and produces a logical and minimal flow of information. Backtrack as necessary.
4. Handling initialization of state.
 - (a) Consider how state will be initialized. Does this affect the location choices made?

- (b) Examine and flow analyse the system framework.
- 5. Implementing the internal behaviour of components.
 - (a) Implement the chosen variable packages and type packages using top-down refinement with constant cross-checking against the design using the Examiner.
 - (b) Repeat these design steps for any identified subsystems.

9 Case Study - A Cycle Computer

The case study can be found in full in [11]. For the purposes of this paper we simply show the result of following the process described in Section 8 and the annotations of the main, entry-point subprogram that results. The consequences of *not* achieving the optimum design, as revealed by the size and form of the SPARK annotations, can then be shown.

9.1 Requirements

The cycle computer consists of a display/control unit to mount on the handlebars of a bicycle and a sensor that detects each complete revolution of the front wheel. The display unit shows the current instantaneous speed on a primary display and has a secondary display showing one of: *total distance*, *distance since last reset*, *average speed* and *time since last reset*. The display/control unit has two buttons: the first *clears* the time, average speed and trip values; and the second switches between the various secondary display *modes*. Unfortunately, but typically of many software projects, the hardware has already been designed: see Figure 1.

There is a clock that provides a regular tick (but not time of day) and the sensor, a reed relay operated by a magnet on the bicycle wheel, provides a pulse each time the wheel completes a revolution.

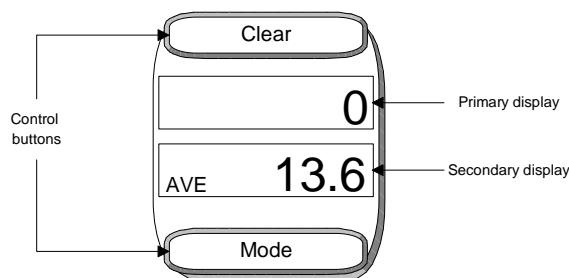


Fig. 1. Cycle Computer Hardware

9.2 Outline Design

Following the design process described in Section 8 we identify boundary packages for:

1. The clock.
2. The sequence of timed pulses received from the wheel sensor (pulse_queue).
3. A boundary abstraction encompassing both the display screens.
4. A boundary abstraction encompassing both the buttons.

Some state data needs to be stored for the controller to perform its function.

1. To calculate instantaneous speed we need (for smoothing purposes) a rolling average, over a short period, of values from the pulse queue.
2. For average speed we need to calculate an average of values from the pulse queue but over the entire period since the clear button was last pressed.
3. To display total distance travelled we need a count of the total number of pulses received since the system was first activated.
4. For trip distance we need a count of the total number of pulses since the last reset.
5. For the elapsed time or stopwatch function we need to record the clock tick value at the last reset.
6. Finally, all calculated values depend on the wheel size which must be stored in a suitable location. For simplicity the routines that would be needed for the user to be able to program the wheel size into the device have been excluded.

Items 1 to 4 are items of abstract state best considered internal to the controller; they should be implemented as instances of abstract data types.

Item 5 is a simple integer-typed value best implemented as a discrete Ada variable local to the main program.

The final item, wheel size, offers more interesting design choices. It could be implemented as a scalar local variable in the main program; however, since all displayed speeds depend on its value it is better to consider it to be external to the controller. We therefore store it in an abstract state machine package; this package also provides a suitable location for the operations to set a new wheel size value.

State initialization does not appear to present any problems; that in boundary packages is implicitly initialized by the environment and that declared local to the main program can be initialized during the first stages of program execution. Again, the wheel size requires some consideration. It is almost certainly best to regard the wheel size state as being initialized because we want to make it an import to the main program (to show that it affects the displayed speed). This suggests the need to set up a default wheel size during elaboration of the variable package which will be used unless a different size is programmed by the user. It is now fairly simple to produce and analyse a package specification skeleton of the entire system. The main program annotation (excluding facilities for programming the wheel size) should be of the form:

```
--# global in      Clock.State,  
--#                Pulse_Queue.State,  
--#                Buttons.State,
```

```

--#           Wheel.Size;
--#           out Display.State;
--# derives Display.State
--#           from Clock.State,
--#           Pulse.Queue.State,
--#           Buttons.State,
--#           Wheel.Size;

```

These annotations are expressed in terms of the problem domain (e.g. the state of the display) and represent the minimum information flows required to provide the desired behaviour. Completion of the system requires only the implementation of the various type packages and variables.

10 Refactoring of Designs After Analysis

It is instructive to observe the result of making different design decisions from those outlined above.

10.1 Missed Abstractions

If we fail to observe that the two displays and the two buttons are tightly interconnected and therefore neglect to place an abstraction around them then their individual states would become visible in the SPARK annotations of the main program. Instead of reasoning about the abstract `Display.State` we have the more detailed `PrimaryDisplay.State` and `SecondaryDisplay.State` (and similarly for two buttons). The changes affect the main program annotations thus (changes from the optimal annotations are shown in italics):

```

--# global in      Clock.State,
--#                Pulse.Queue.State,
--#                ClearButton.State,
--#                ModeButton.State,
--#                Wheel.Size;
--#           out PrimaryDisplay.State,
--#                SecondaryDisplay.State;
--# derives PrimaryDisplay.State
--#           from Clock.State,
--#                Pulse.Queue.State,
--#                Wheel.Size &
--#           SecondaryDisplay.State
--#           from Clock.State,
--#                Pulse.Queue.State,
--#                ClearButton.State,
--#                ModeButton.State,

```

```
--#                               Wheel.Size;
```

This annotation is longer and more complex than the original and reveals the close coupling between, for example, the displays. Most things which affect the primary display also affect the secondary display; this is consistent with our understanding of the system. If, for example, the bicycle goes faster the higher speed shows on the primary display but the average speed will also change and this affects the secondary display. The only difference between the two dependencies is that the clear and mode buttons affect only the secondary display; this is useful information when considering the display packages themselves but can be considered unnecessary detail at the main program level. Examination of the flow relations directly prompts us to provide abstractions for the displays and buttons to the overall benefit of the design.

10.2 Badly Positioned State

Badly handled OOD approaches often lead to proliferation of objects to handle particular events or parts of the system behaviour. We could, for example, introduce a “pulse count handler” to replace our original design choice of storing pulse counts locally within the main program. The handler package would take the form of an abstract state machine that tracked each wheel pulse received and could provide trip and overall distance information. Because this object is external to the main controller its state becomes visible in the main program annotations which will change as follows:

```
--# global in           Clock.State,  
--#                   Pulse_Queue.State,  
--#                   Buttons.State,  
--#                   Wheel.Size;  
--#                   out Display.State;  
--#                   in out Pulse_Handler.State;  
--# derives Display.State  
--#                   from Clock.State,  
--#                   Pulse_Queue.State,  
--#                   Pulse_Handler.State,  
--#                   Buttons.State,  
--#                   Wheel.Size &  
--#                   Pulse_Handler.State  
--#                   from Pulse_Handler.State,  
--#                   Pulse_Queue.State,  
--#                   Buttons.State;
```

This is significantly less clear than the original. We can probably accept that the display is affected by the `Pulse_Handler` but we also have to deal with an entirely new clause in the flow relation showing that the `Pulse_Handler` depends on itself and the `Pulse_Queue` (but not, for example, the clock). Furthermore, the new items in the annotation are not in problem domain terms; `Pulse_Handler` is an internal design

artefact not part of the external system we are building. Again consideration of information flow, revealed by static analysis, prompts us to regard pulse count handling as an internal rather than an external property of the system and thus improve our design.

Note that both of the poor design choices in Section 10 can be detected and corrected *before* the packages concerned are implemented; we are analysing only emerging design constructs.

11 Conclusions

Project risk and project cost are both increased if processes for error detection are weighted towards the back end of the project life cycle; this is equally true for static analysis and dynamic testing. Static analysis, in contrast with dynamic testing, can be carried out early in the project lifecycle provided that the language and tools used permit analysis of incomplete programs. Early analysis has a straightforward benefit because errors found early are easy and cheap to correct; however, there is a more subtle benefit from early analysis concerned with program design. Static analysis, as performed by the SPARK Examiner, measures properties of the program, such as information flow, which are powerful indicators of design quality. High levels of information flow indicate excessive coupling or poor state abstractions and are revealed by large or complex SPARK annotations. We can therefore conceive static analysis as being much more than a verification activity: it is actually a powerful design aid. The benefits of this are enormous: good design features such as loose coupling provide benefit throughout all remaining lifecycle stages. The code is easier to analyse and test; easier to understand; and easier to modify.

Praxis Critical Systems have recently delivered two systems whose designs were influenced by early analysis aimed at minimising information flow and we have had feedback from another SPARK user who has taken the same approach. All three of these projects have shown the benefits to be both practical and valuable.

References

1. Croxford, Martin and Sutton, James: *Breaking through the V&V Bottleneck*. Lecture Notes in Computer Science Volume 1031, 1996.
2. Sutton, James: *Cost-Effective Approaches to Satisfy Safety-critical Regulatory Requirements*. Workshop Session, SIGAda 2000.
3. B D Bramson. *Malvern's Program Analysers*. RSRE Research Review 1984.
4. Bernard Carré. *Program Analysis and Verification in High Integrity Software*. Chris Sennett (Ed). Pitman. ISBN 0-273-03158-9.
5. C. Daniel Cooper. *Ada Code Analysis: Technology, Experience, and Issues*. Proceedings SIGAda 2000.
6. Finnie, Gavin et al: *SPARK - The SPADE Ada Kernel*. Edition 3.3, 1997, Praxis Critical Systems¹.
7. Finnie, Gavin et al: *SPARK 95 - The SPADE Ada 95 Kernel*. 1999, Praxis Critical Systems¹.

¹ Praxis Critical Systems documents are available on request from sparkinfo@praxis-cs.co.uk

8. Barnes, John: *High Integrity Ada - the SPARK Approach*. Addison Wesley Longman, ISBN 0-201-17517-7.
9. King, Hammond, Chapman and Pryor: *Is Proof More Cost-Effective than Testing?*. IEEE Transaction on Software Engineering, Vol. 26, No. 8, August 2000, pp 675-686.
10. Bergeretti and Carré: *Information-flow and data-flow analysis of while-programs*. ACM Transactions on Programming Languages and Systems 1985¹, pp37-61.
11. Amey, Peter: *The INFORMED Design Method for SPARK*. Praxis Critical Systems 1999, 2001¹.
12. D L Parnas and J Madey. *Functional Documentation for Computer Systems*, in *Science of Computer Programming*. October 1995, pp41-61.
13. Amey, Peter: *A Language for Systems not Just Software*. Proceedings, SIGAda 2001².

² Also downloadable from www.sparkada.com