Standing on the Shoulders of Giants

Björn Bon

As the demand for software increases dramatically, reusable software is becoming more and more important. People are looking for software systems that contain maintainable and reusable pieces of software, but these systems are still hard to find. Developing these kind of systems starts with a proper decomposition of the main problem into sub-problems. This article describes a method to evaluate such a decomposition in an objective way. The method focuses on the evaluation of software architectures with respect to fundamental design criteria like minimal coupling and maximal cohesion.

The Need for Reuse

"If I have seen further than most men, it is because I stood on the shoulders of giants"
—Isaac Newton, 1676

In 1676, Isaac Newton made this famous statement. He realised he had defined some important laws that would be the basis of modern physics. He also realised that he did not do it on his own. Newton's strength was applying the available knowledge and extending it.

Science is a typical area where reuse is applied. In other fields reuse is important as well. It is used in classic architecture, but also in the television industry. This may lead to boring programmes, but it is a very efficient way of working.

In the last years, the demand for software has dramatically increased. The software world has replied in terms of better design tools (compilers and high level languages) and better development processes, like the *Capability Maturity Model*. Another trend is the reuse of software. As in other disciplines, reuse is probably the best answer to the increasing demand for software. However, reuse may be dangerous. It is very important to know the context in which the software can be reused. If the dependencies and the required context of the software to be reused is not known, this may lead to unpredictable systems (Ariane 5). Another problem with reusable software is that we often need to do some redesigning or reimplementation, before the software can be integrated in its new environment. The result is a new piece of software that is again not reusable in its pure form.

Software pieces with a clearly defined function and with a minimal and well known required context are still rare. Only these types of software pieces can be reused in larger systems without modifications and so really be reused. If we want to be able to meet the increasing demand for software in the future, it is important to get to the level where we are able to define software that can be reused in its most pure form, without the need of any modification.

The next chapter answers the question why we are not at that level. It answers this question by describing the software design evolution and the role of education. Then a chapter follows that describes the fundamental design criteria to come to reusable and maintainable software. This chapter is the basis of the proposal as described in the chapter called 'Architectural Evaluation.' The fourth chapter maps the design criteria on an evaluation model for software architectures. With a simple example the model is applied to two software architectures.

The Software Design Evolution

The first computers (right after World War 2) were slow, unreliable and very small in terms of memory and processing power. Programming these machines meant programming at the level of binary machine instructions. This was not a trivial job, but as the size of the programmes was rather small, one was able to do it.

With the increasing processing power and memory sizes of the computers, the development of high level programming languages was stimulated. This resulted in many programming languages like FORTRAN and COBOL. Today, the use of high level and object-oriented programming languages like EIFFEL and C++ are quite usual. In the non-embedded industry the so-called fourth generation languages are applied more and more.

Clearly, a lot of effort is put in new technologies with respect to software development. It is also clear that most effort is put in technologies that focus on the last part of the software development process, the coding phase. Of course, coding is one of the major parts of software development, but with the many fancy tools currently available it is tempting to start coding right away without thinking about the definition of clean reusable software pieces. This leads to a lot of code that cannot be reused.

Another disadvantage of the fast technological developments is that more and more attention is paid to new technology, new tools and making these new tools work. It seems that the real important issue is forgotten, which is: well designed software.

Good software design is independent of technology and language. It has to deal with the decomposition of a problem into smaller sub-problems. The way the solution of a sub-problem is implemented is not that important (it may be implemented by a C++ class, a COM component or a query on a database). If the solution of a sub-problem is called a module, then it is very important that the responsibility of the modules is clear and that the dependencies between the different modules are minimal. In this way we can design a system containing modules with maximal *cohesion* and minimal *coupling.* This increases the reusability and maintainability of the software.

Educational institutes do not pay much attention on architectural issues like coupling and cohesion. They focus more on the design of efficient algorithms. In recent years the situation has improved. This is partly caused by the introduction of design patterns [1]. A next step may be paying more attention to topics like architectural patterns. At this level, complete software systems can be described in terms of large software modules that interact with each other. For most studies, this is a difficult subject, because a software system is very specific for its domain. The goal of most software studies is not to learn about a certain domain like copiers. It is not possible to eductate the generic software architect who can solve any software problem. A software architect for banking applications has different capabilities than a television software architect. However, they do have some things in common. They both must be able to deal with fundamental design criteria as described in the next chapter.

Fundamental Design Criteria

At an abstract level, there are some generic software requirements:

1. Meeting the functional requirements
2. Maintainability
3. Reusability

By reuse, we often mean that it must be easy to integrate parts of the software into other software systems. This way, new software systems can be realised in less time.

To meet the requirements mentioned above, the requested functionality must be decomposed into subfunctions. This typically takes place in the architectural phase. In this phase, the software system is divided into modules where each module represents a sub-function. Such a module must meet the following criteria:

1. Minimal coupling: the number of modules one module interfaces with must be minimal.
2. Maximal cohesion: the functionality of a module must be single, clear and pure.
3. Encapsulation of data: the implementation details of a module must be hidden. Interfacing

between modules is done by well defined interfaces that are as simple and small as possible.
4. Documentation: the interface and behaviour of a module must be well documented.

Minimal Coupling

The degree of coupling between modules in a system is an indication of the maintainability of the system as a whole and the reusability of each separate module. The coupling between modules should be as small as possible. This means that a module must interface with the least possible number of other modules. It also means that the interface between modules (or the strength of the coupling) must be as weak as possible. The number of modules a module is coupled to can be derived from the logical static structure of the system. The direction of the coupling is important. If module A calls a function of module B, A depends on B. So module B is necessary to make module A function.

Maximal Cohesion

Cohesion is an indication of the correlation between functions in a module. A system containing modules with a high cohesion leads to maintainable systems. If the system requires a functional change, only one module in the system needs to be modified, being the module responsible for that certain function that needs a change. Modules with a high cohesion are reusable in other systems, because they implement a well defined function independent of the rest of the system.

Data Encapsulation

New component technologies (COM, Javabeans) and object-oriented languages (C++) provide mechanisms to separate interfaces from implementation. Of course, these technologies do not solve the real problem of data encapsulation, which is determining what parts need to be public (interface) and what parts are details (implementation). If these parts are separated well, the result is a module with a minimal interface. This way, the module is more predictable, because the way the module is controlled is restricted to the minimal interface.

Documentation

In the software development process, the generation of proper documentation is very important. From a reuse point of view, documentation about how to use a module is more important than a description of some implementation details. Tools can support the documentation process by means of auto-documentation and consistency checks.

Architectural Evaluation

The previous chapter described some fundamental software design criteria. The question is how to evaluate a software system using these criteria in an objective way. Most of the times, architectures are evaluated by means of feelings. It just looks right or it is a nice concept.

The degree of coupling can be easily measured by counting the number of modules a module is connected to. Measuring the degree of cohesion is less easy. This chapter presents a proposal to evaluate the degree of coupling and cohesion of modules in a software system. Based on this evaluation, the architecture can be adapted and evaluated again. In this way, structural improvement is possible. It is also possible to compare two architectures solving the same problem. At the end of this chapter, this process is explained by means of an example.

Evaluation Model

Below, four steps are described to evaluate an architecture with respect to coupling and cohesion. Encapsulation of data is implied by high cohesion in combination with a minimal interface (minimal coupling). We assume that the co-operation and the interfaces of the modules of the architecture are known and documented.

1. Describe the logical architecture of the system. Indicate which modules depend on which modules by using arrows. For each module, describe its interface. This can be done in a formal way but also in plain English.

2. Make a list of the degree of coupling per module. For each module, its incoming and outgoing arrows are counted. Many incoming arrows mean that many other modules depend on the first module. This decreases the maintainability of the module. Many outgoing arrows mean that the module depends on many others. This decreases the reusability of the module.
3. Make a list of the cohesion of the modules. Making the list is based on the SAAM method, published by the Carnegie Mellon Software Engineering Institute [2]. The SAAM method is based on the following substeps:
   (a) Make scenarios. By scenarios, we mean possible modifications or extensions to the system. This is the toughest and most important step of the evaluation process. The more scenarios we have, the better the system can be evaluated with respect to cohesion. Making scenarios means involving many people like users, maintainers, and architects.
   (b) Determine which scenarios affect which modules. Since the interfaces of the modules are known, it must be possible to take this step.
   (c) Evaluation: When one scenario affects multiple modules, this usually means that the functionality is distributed over multiple modules. This decreases the maintainability of the system. When different scenarios affect the same module, this module contains different functions, which has a negative effect on the reusability of the module.
4. Overall evaluation of the architecture. Determine what modules have to be reusable and/or maintainable. Compare this with the results acquired by step 2 and 3.

Example

The example presents two architectures of a cruise control system for cars. The architectures are evaluated in parallel by applying the steps of the previous section. First, a brief description of the cruise control system is given.

Requirements: The driver determines the requested speed of the car by a button on the dashboard. Based on this desired speed, the current speed, and the revolutions per minute, the system chooses the correct gear and the correct amount and mixture of fuel and air.

Step 1: Description Logical Architecture

Figure 1 describes solution 1. The arrows indicate the direction of the dependencies. For example, the *MotorMonitor* calls the functions *SetSpeed(v)* and *SetRevolution(r)*, which are implemented by the *ControlModule*. Solution 1 contains five modules, which are described below.

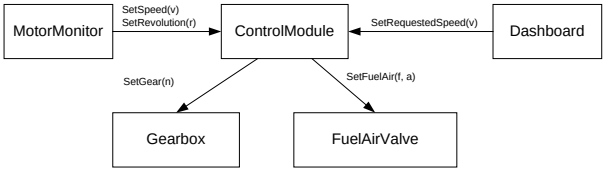

Figure 1: Step 1: Cruise control, solution 1

- The *MotorMonitor* measures the current speed and the revolutions per minute of the motor. When one of these attributes changes, the function *SetSpeed(v)* or *SetRevolution(r)* is called. These functions are API functions of the *ControlModule.*
- The *Dashboard* Module is responsible for reading the button of the dashboard (requested speed). When the requested speed changes, the *Dashboard* Module calls the *SetRequestedSpeed(v)* function, which is implemented by the *ControlModule.*
- Based on the current speed, the requested speed, and the revolutions per minute, the *ControlModule* computes the correct gear, and the correct amount and mixture of fuel and air.
- The *Gearbox* module is responsible for physically selecting the right gear. This module presents an interface function *SetGear(n).*
- The *FuelAirValve* controls the physical valve that is responsible for the amount and mixture of fuel and air.

Figure 2 describes solution 2. This solution contains six modules which are described below.

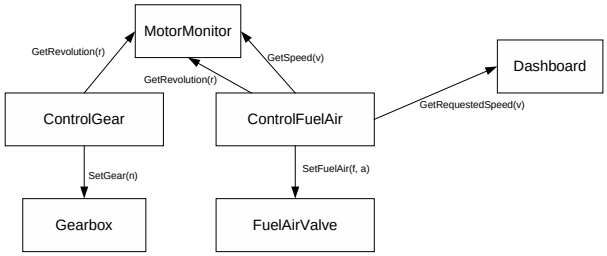

Figure 2: Step 2: Cruise control, solution 2

- The *MotorMonitor* measures the current speed and the revolutions per minute of the motor. Other modules can access the information by calling the interface functions *GetRevolution(r)* and *GetSpeed(v).*
- The *Dashboard* Module is responsible for reading the button of the dashboard (requested speed). Other modules can access the information by calling the interface function *GetRequestedSpeed(v),* which is implemented by the *Dashboard* module.
- The *ControlGear* module calls the *GetRevolution(r)* function, implemented by the *MotorMonitor*, to obtain the current number of revolutions per minute. Based on this input, it computes the correct gear and calls the *SetGear(n)* function of the *Gearbox* to activate the correct gear.
- The *ControlFuelAir* module calls the *GetSpeed(v), GetRevolution(r),* and *GetRequestedSpeed(v)* function. Based on this input, it computes the amount and mixture of fuel and air. After the computation, the *SetFuelAir(f, a)* function of *FuelAirValve* is called to activate the new amount and mixture of fuel and air.
- *Gearbox*, see solution 1.
- *FuelAirValve*, see solution 1.

Although many aspects of the architectures are not described, we have sufficient information to do the next steps of the evaluation process.

Step 2: Make a List of the Coupling

Table 1 presents the coupling of solution 1. For each module, the table indicates the number of incoming connections (maintainability) and outgoing connections (reusability). The numbers in the cells indicate the number of connections. The lower the number the better. We see that the *Gearbox* and *FuelAirValve* are very well reusable. The *MotorMonitor* and *Dashboard* are less reusable, because they depend on another module. However, they are more maintainable.

Module	**Maintainability** incoming arrows	**Reusability** outgoing arrows
MotorMonitor	0	1
Dashboard	0	1
ControlModule	2	2
Gearbox	1	0
FuelAirValve	1	0

Table 1: The coupling of solution 1

Table 2 describes the coupling of solution 2. The table has the same structure as Table 1. The *MotorMonitor*, *Dashboard*, *Gearbox*, and *FuelAirValve* are reusable. The *ControlGear*, and *ControlFuelAir* are more maintainable.

Module	**Maintainability** incoming arrows	**Reusability** outgoing arrows
MotorMonitor	3	0
Dashboard	1	0
ControlGear	0	2
ControlFuelAir	0	4
Gearbox	1	0
FuelAirValve	1	0

Table 2: The coupling of solution 2

Step 3: Make a List of the Cohesion

Make Scenarios In the example, we choose the following 4 scenarios:

1. The car is equipped with another type of motor. This means that the computation of the amount and mixture of fuel and air changes.
2. The car has faster acceleration by postponing changing up the gear.
3. The user interface of the cruise control changes.
4. The way the physical gear box is controlled changes.

Determine the Impact of the Scenarios Table 3 presents the cohesion of solution 1. Each column

	MotorMonitor	Dashboard	ControlModule	Gearbox	FuelAirValve
Other Motor			Influences		
Faster Acceleration			Influences		
New User-Interface		Influences			
Change in physical control of gearbox				Influences	

Table 3: Cohesion of solution 1

represents a module. The rows represent the scenarios. The cells indicate whether a scenario implies a change in a module. A row containing multiple cells with the keyword '*Influences*' implies that one function affects multiple modules. This means that a function is distributed over multiple modules. This decreases maintainability. Note that this situation does not occur in this example. A column containing multiple cells with the keyword '*Influences*' implies that one module contains more than one function. The responsibility of the modules is not clear (low cohesion), which implies that reuse may be difficult.

Table 4 presents the cohesion of solution 2. This solution contains modules that have a higher cohesion than the modules of solution 1. Controlling the *Gearbox* and the AirFuelValve are performed by two separate modules. In this way, the modules have a clearer responsibility. This is useful with respect to maintainability and reusability. Note that if we merge scenario 2 and 4, we do have a row with multiple keywords '*Influences*'. This may lead to other conclusions. Making the right scenarios requires a lot of attention and is the most difficult and most fundamental part of the evaluation.

Step 4: Overall Evaluation

The cohesion tables show that solution 2 is better than solution 1. Solution 2 offers modules that are more independent. The solution 2 modules also have a higher cohesion.

Evaluation

This small example shows that it is possible to quantify the coupling and cohesion of architectures using some simple techniques. When looking at large architectures, these techniques are useful to identify the strengths and weaknesses of the architecture in a more objective and structural way. In this way, we can get to the level where we are able to make better architectures that contain software pieces that are reusable in its most pure form.

When evaluating the cohesion of an architecture, it is important to know that the evaluation is based on the scenarios available. Making these scenarios requires a lot of attention from different kinds of people.

Conclusion

In every area in our society, people are (re)using each others work. In the software area, reuse may be the solution to realise future software systems more efficiently. This means that investments in solid software architectures have to be made at this very moment, in order to generate maintainable and reusable modules. Investments in new technology are certainly not sufficient, because this new technology mainly focuses on the coding phase, and less on using essential design criteria that are used in software design.

Investing in solid architectures starts with evaluating the architecture. This article described a model that can help doing this. Simple reporting techniques, like tables, enable the identification of weak spots in the architecture. When determining the cohesion of an architecture, we should note that the

	MotorMonitor	Dashboard	ControlGear	ControlFuelAir	Gearbox	FuelAirValve
Other Motor				Influences		
Faster Acceleration			Influences			
New User-Interface		Influences				
Change in physical control of gearbox					Influences	

Table 4: Cohesion of solution 2

evaluation is related to the scenarios we devised. This means that devising the scenarios should get a lot of attention from several people. The next step would be adapting the architecture, in order to improve the weak spots. The result will be maintainable and reusable software modules. Using Isaac Newtons words, this will again result in a new giant on whose shoulders we can stand.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley.

[2] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements, *Scenario-Based Analysis of Software Architecture*. Available via http://www.sei.cmu.edu/-architecture/scenario_paper/index.html.

About the Author

Björn Bon is with Alert Automation Services bv. At this moment, he is assigned to Philips Semiconductors, where he is working on the definition of reusable software components for Consumer Electronics Systems. He holds an M.Sc. in Information Technology and a Master of Technological Design in Software Technology, both from the Eindhoven University of Technology.