"Freedom" Service−Oriented Methodology White Paper

An Introduction to Requirements Encapsulation and Definition

by
Rick Lutowski
rick@jreality.com

Virtually everyone in the software industry considers it a truism that many, if not most, requirements will be forward traceable to more than one code module. While this is true using today's methodologies, it need not be true in the very near future. A few software engineers, including this writer, have been encapsulating every requirement of a requirements specification individually within a single code module for years. Perhaps you are thinking, "That would require a magician or IT from Mars." Close. The trick is to use IT developed for those who are trying get there. As for magicians, there is no such thing; magic is mostly a matter of proper perspective.

Here is that perspective.

Guidance

A long web−time ago (1981) on a project faded far from collective memory, Dr. David Parnas and Ms. Kathy Britton, both of the Naval Research Laboratory Software Cost Reduction (SCR) Project, published a paper proposing that the then−emerging concept of information−hiding could be used to encapsulate three major types of information. The *A−7E Software Module Guide* referred to encapsulated information as module "secrets" because the implementation details of the information was known only to the encapsulating module. By hiding knowledge of implementation details behind the stable interface of the module, other modules which utilize that information need not be affected when the implementation details of the "secret" information changes. The result is more maintainable software with lower life−cycle cost.

The *Module Guide* implied maximum life−cycle cost savings would result when three types of information with high probability of change were all made secrets hidden within modules. These three types of hidden information are:

1. Hardware−software interfaces
2. Software design decisions
3. Required behavior

The first type of module secret listed in the *Module Guide* is software interfaces to hardware. Hardware programmatic interfaces are encapsulated within hardware−hiding modules, which are "programs that need to be changed if any part of the hardware is replaced". These types of modules came into common use in operating system design under the moniker of "device drivers." All modern operating systems now use device drivers, even if the remainder of the operating system is not written using information−hiding concepts.

The second category of module secrets are software decisions. Software decision−hiding modules encapsulate "software design decisions based upon mathematical theorems, physical facts, and programming considerations such as algorithmic efficiency and accuracy". Software decision hiding now forms the backbone of modern object−oriented (OO) development, particularly in the form of data structure encapsulation.

However, the most interesting proposed type of information−hiding module consists of "programs that need to be changed if there are changes in the sections of the requirements document that describe the required behavior". The encapsulated secret of these "behavior−hiding" modules is the required behavior of the software that forms the heart of the requirements document. Unlike the other two types of secrets, encapsulation of required behavior never caught the eye of the industry. As a result, almost everyone today has forgotten that Parnas and Britton proposed that it could even be done.

Almost everyone.

Synergy

Ten years after the inception of the SCR Project, the paths of the author and a member of the original SCR team (David Weiss) crossed at the Software Productivity Consortium. It was this synergy that made the author aware of the *Module Guide* and its claim that information−hiding could be applied to requirements. Intrigued, the author began searching for a practical approach to requirements encapsulation years after the SCR project let the idea slide. (In an email to the author, Parnas indicated that requirements were not initially recognized as a problem by the SCR team, perhaps explaining why requirements−hiding was not more fully pursued.) The lack of a solution from SCR may well explain why requirements encapsulation never caught the eye of the industry.

Pursuit

It quickly became clear that devising a practical approach to requirements encapsulation would be challenging. Consider typical requirements statements such as:

1. The system shall compute XYZ.
2. The data base query should be cached.
3. The system shall have 99.9% availability.
4. The system shall be written in Java.

It is not hard to imagine encapsulating items (1) and (2) each in their own single module. However, encapsulating items (3) and (4) in individual modules does not seem possible. SCR claimed required behavior could be encapsulated. Item (4) is not behavior, perhaps explaining the problem there. However, item (3) is a behavioral characteristic but does not appear to be encapsulatable either. Might the concept of requirements encapsulation be flawed? With conviction in the validity of the SCR concept, the focus shifted to a more fundamental level. The nature of requirements themselves came under scrutiny, starting with the question "Exactly what are requirements?"

Many define requirements in general and vague terms −− "What the system shall do, but not how it shall do it." (ubiquitous)

Some use a more legalistic definition, such as "A capability that must be met or possessed to satisfy a contract or specification." (*IEEE Standard Glossary* type of definition).

A few enumerate selected parts of the software, such as "Requirements are the interface to users, external systems, and the hardware." (Brooks, *No Silver Bullet* paper).

One person (I do not recall exactly who) drew a dotted circle on the back of an envelope then, motioning to the outside of the circle, said enigmatically, "Requirements might be out here somewhere."

Many authors do not state a definition at all, either content the term is self−explanatory or, perhaps, too wise to tread into a minefield.

It soon became apparent that there indeed was a problem concerning the definition of requirements. The fact that different people gave such widely differing definitions, and the fact that most of the definitions were very vague or non−specific, was symptomatic. The bottom line was, and still is:

The industry has no precise definition of requirements, and no consensus, even at a vague level, on what requirements are!

Clearly, one cannot encapsulate something unless one knows what it is. Before a solution to the requirements encapsulation problem could be found, a precise (not vague) technical (not legal) definition of requirements had to be found.

Definition

The work of the late Dr. Harlan Mills provided the critical insight. Dr. Mills, a former IBM Fellow noted for his research on software "clean room" development for zero−defect software, was one in the "too wise to tread" camp. Instead of definitions, Dr. Mills employed a model upon which to base solutions to hard software problems. He chose a well−understood model proven effective in other engineering disciplines. This model was the concept of a "black box". While Mills never actually said so, it was not hard to infer from his work what he thought requirements were:

Requirements are the black box view of the software system.

The black box view of a system is precisely equal to the external interface of the system −− by definition of a black box. The above definition of requirements is thus equivalent to:

Requirements are the external interface of the software system.

Note that this is very close to the definition given by Brooks in his classic *No Silver Bullet* paper. However, failure to recognize a black box as the underlying conceptual model caused Brooks to include hardware interfaces in his definition, which is incorrect. Subsequent methodology development and usage has validated the black box definition. Requirements are precisely the software system external interface −− to humans, to external systems, and to the external environment. Nothing more; nothing less.

Freedom

This clarification of vision via Dr. Harlan Mills was forthcoming a year or two after leaving the SPC for the Space Station Freedom Project (SSFP). The SSFP Software Support Environment (SSE) standards and methods team at Johnson Space Center −− consisting of Jeff Kantor, Ron Blakemore and the author −− used the newly clarified precise definition of requirements as the basis of a requirements encapsulation methodology for use by SSFP. This methodology was documented in SSE project deliverable document #42, which evoked a bit of wry humor −− in *The Hitchhikers Guide to the Universe*, "42" was the alleged "universal answer," which aptly mirrored the team's enthusiasm at having cracked the requirements encapsulation problem.

After termination of the SSFP, the author, recognizing the merits of the methodology, continued to apply and refine it. In 2002, the evolved methodology was adopted for use on the Freedom Ship project by Chris Jacoby, Neeraj Tulsian, Shing Lin, Travis Watkins, and the author. This variant was documented on the Freedom Ship intranet by the author with the help of Gang Qi. In recognition of its invention for use by Space Station Freedom and its subsequent adoption for use by Freedom Ship, the requirements encapsulation methodology was named "Freedom."

The definition of requirements as the black box view of the software system leads to a complete solution to the requirements encapsulation problem. It offers other significant benefits and guidance as well, which fully merit the name "Freedom". The following explains how encapsulation follows naturally from the definition, and summarizes some (but not all) of the other "freedoms" that result.

Encapsulation: Freedom to Change

Freedom recognizes a requirements specification as being the specification of an interface, i.e., the software system external interface. Encapsulating requirements is conceptually no different than encapsulating hardware since they are both types of interfaces. The industry has decades of experience encapsulating interfaces to hardware. Thus, encapsulating requirements involves no great leap in OO technology, but does demand we clarify our thinking regarding requirements. It is essential to recognize that requirements are the software system external interface, by definition of a black box, and a requirements specification is a specification of that interface.

Clarity: Freedom from Confusion

Freedom practitioners clearly understand the difference between requirements and design information; the difference is as obvious as a black wall. Requirements lie on one side of the wall, design on the other (the fellow who drew the dotted circle had the right idea!) The determining criteria is external visibility −− can a user (human or external system) see the information? Yes equals requirements; No equals design. One by−product of clarity is completeness. Knowing when requirements are complete is simply a matter of knowing when the specification of the external interface is complete. Feedback from users via a User Interface Prototype is quite effective at determining initial requirements completeness, and when design based on those requirements should commence.

1:1 Mapping: Freedom from Traceability

Encapsulation of requirements effectively ensures a 1:1 mapping between individual code modules and requirements. The result is that the traditional problem of requirements traceability becomes trivial to the point of essentially disappearing. The time and expense of maintaining traceability maps is eliminated, reducing development time and cost. At the same time, progress tracking in terms of requirements implemented to date is greatly simplified and not compromised by traceability errors, making scheduling and reporting easier and more accurate.

Neutrality: Freedom from Design Pollution

When a user (human or external software system) sees the software, they see the interface (the requirements), including a "look and feel" or protocol, controls and/or valid commands that can be issued, and responses that come back. They do not see code modules (the design and implementation). The users do not know, or care, if the black box internals are built using OO, functional design, expert systems, or even burned into silicon. Since requirements contain no information beyond what the users (human and external software systems) perceive, they contain no information about the design or implementation. Hence, black box requirements are design−neutral and implementation−neutral.

Formality: Freedom from Documents

Freedom recognizes that formal documentation exists to serve customer needs, not a methodology. With the sole exception of a user manual, required for any software with a human user interface, Freedom does not specify generation of formal documentation. No requirements document, no design document, no test plan document, and so forth means no effort wasted on book publishing. Freedom does insist that technical files for these work products be developed, but these need not be publication quality simply for methodology's sake. Product quality does not depend on formal presentation of information, only on information content. Content in files or in a data base is quite sufficient, and even superior to published document formats since raw machine readable data is more easily subject to automated checks than the same data in publication format. However, customer needs govern. If the customer requires formal documentation for project monitoring and tracking or other purposes, and is willing to pay for it, then Freedom can easily accommodate the need. Freedom has no need of formal books beyond the needs of the customer.

Interface−Centricity: Freedom from Scenarios

Use Cases are diagrams depicting use of the system. Use of the system is a specification of a process, which is quite different from a specification of an interface. Thus, by definition, Use Cases do not specify requirements, as widely claimed. Use Cases may comprise the main body of a user manual, or serve as supplemental material in the appendix of a requirements document (should formal documentation be produced), but they are not appropriate in the main body of a requirements document because Use Cases, usage scenarios and other process−centric notations do not specify requirements. Because black box requirements are interface−centric, Freedom incorporates two new and effective techniques for specifying external interfaces to record requirements. These techniques are Functionality Trees, which identify and organize stimuli, and Behavior Tables, which specify responses to the stimuli.

Development: Freedom from Unnecessary Effort

Many developers ignore the external interface until near the end of the project. For requirements, they use things like Use Cases, CRC Cards, Data Flow Diagrams, Statecharts, or one of dozens of other techniques. In so doing, they expend an amount of work (R). Later, they get around to specifying the external interface, because without it the software would be useless, expending an amount of work (I). In contrast, Freedom recognizes that specification of the external interface IS requirements, and expends the amount of work (I) up front and in lieu of (R). Hence, Freedom beats current methodologies by the time and cost that other approaches expend performing the unit of work (R), minus whatever part of (R) they may fortuitously devote to specification of the external interface (usually very little).

Maintenance: Freedom to Evolve

In addition to the development savings above, Freedom’s ability to encapsulate requirements continues to pay dividends throughout the life of the software (if OO design and implementation are used; recall Freedom requirements are neutral and do not dictate OO). With current approaches, requirements changes are difficult to back fit into the code. Encapsulating requirements makes requirements easier to change in the same way that encapsulation of data structures makes data structures easier to change. Both contribute to a decrease in the life−cycle cost of the software. Hence, Freedom calls the post−release activity "Evolution" to emphasize ease of enhancement of requirements, rather than the traditional "Maintenance," which connotes bug fixing. Since maintenance costs are widely recognized as comprising as much as 80% of the life−cycle cost of an application, the post−development savings of requirements encapsulation can be expected to dwarf the development savings (which are in and of themselves sufficient reason to use Freedom).

Silver?

For the above reasons, and others not covered here, it is likely that no other methodology can beat Freedom for cost efficiency, except an improved version of Freedom itself. Long after Freedom becomes widespread it will continue to improve, leading to ever greater savings. Might requirements encapsulation be the key to the elusive "silver bullet" of Brooks −− reductions in the cost of software comparable to that traditionally attained by hardware? Founding software engineering on the same conceptual model used by hardware engineers, the black box model, offers a promising point of departure.

More information on requirements encapsulation and the Freedom methodology is available on the Freedom web site at http://www.jreality.com/freedom/

References

"A−7E Software Module Guide," by K.H. Britton and D.L. Parnas, Naval Research Laboratory, NRL Memorandum Report 4702, December 8, 1981.

"No Silver Bullet −− Essence and Accidents of Software Engineering," by Frederick P. Brooks, Jr., *Computer*, Vol 20 No 4, April 1987, pp 10−19.