Decomposition and Interfaces Revisited

Andreas Rau
Systems Engineer
STZ Softwaretechnik
73730 Esslingen, Germany
email: andreas.rau@stz-softwaretechnik.de

ABSTRACT
Abstraction and hierarchical decomposition are key principles for mastering complexity in science and engineering. In Software Engineering, the interfaces that result from the application of those principles play a crucial role, particularly when building large and complex systems. This article revisits general principles behind abstraction and decomposition and presents a collection of fundamental principles and design goals to guide the modularization of a system and the definition of interfaces.

Abstraction, decomposition, interfaces

1. Introduction

Abstraction is one of the fundamental principles in science and engineering. In a complex world such as ours, only the removal of irrelevant details clears the view for understanding of the various problems and tasks we are faced with, which is a prerequisite for solving or mastering them - one at a time. Since the solving of problems and the mastering (automation) of tasks is the main application of computers and software, abstraction is an indispensable tool also for software engineering. It is the principle of abstraction that enables us to "divide and conquer" the problems that can be solved by means of computers and software. Following this principle, every complex problem or task is broken down into simpler problems or steps that can solved by an algorithm. This is reflected by the modular structure of software. Modules serve many important purposes in a software development process. A module

- is a self-contained unit of work that can be assigned to a developer for more or less independent and parallel implementation.
- is a countable part of the solution. The status of all modules constitutes the status of the project and can be used for tracking progress.
- is a unit for test and integration. Step-by-step unit testing and integration is required to achieve adequate test depth and minimize test effort.
- is a (potential) candidate for reuse.

However, any given module will not automatically fullfill all the above purposes. This might not matter for simple systems and in small projects, however for complex systems and large systems, their ignorance can be lethal. This is because such projects

- require sharing the work between many people, and it is essential for them to be largely independent of each other. Otherwise the overhead for communication can explode to the point where it effectively halts the project.
- involve many, many modules. When those modules are not independent and changes in one module keep affecting others, the project is going in circles, nothing ever gets finished and tracking progress becomes nearly impossible.
- include major effort for testing and the definition and management of test-cases. If modules can only be tested with enourmous effort or not at all, or if they keep changing (and thus require changing of their test cases and test beds), this effort will explode and the project either will not meet its deadlines or will deliver poor quality.
- are typically not developed in a linear fashion, but in iterations, i.e. through incremental changes. The resulting product has to be maintained over a long lifetime. Because of its complexity and the amount of effort it required to build, it cannot simply be re-build if requirements change. However, changes of requirements are very likely, even during development. If the effects of these changes ripple throughout the whole system, incremental development and maintainance are not safely and economically possible.
- are based on powerful and complex concepts and technologies that are too expensive to re-implement and re-test every time. It is far more reasonable and economical to re-use a time-tested component from a library or a third-party vendor.[1]

Therefore, modules and their interfaces have to be designed carefully. As in other areas of life, some extra effort

[1] Upon a closer look, even everyday software like web-browsers includes a mix of components (e.g. pluggins) from different vendors.

early can save a lot of headaches *later*. A variety of principles and methods have been established to help to decompose a given system into "good" modules. We shall discuss shortly what "good" means in this context. Unfortunately some of the corresponding terminology can easily be confused with abstraction itself (see [Ber] for a discussion). Interfaces play a crucial role in this decomposition and thus deserve our special attention.

The following section discusses the importance of interfaces in the decomposition process and presents a collection of principles and goals from various sources to aid their proper definition. Note: the common notion of a module typically involves a collection of functions with the module interface being the collection of their signatures. For the discussion of interfaces, the terms module and function are used interchangeably to reference products of decomposition with different sizes.

2. Interfaces

Interfaces are created whenever something is decomposed into parts, and they have a substantial impact on the development process and the final product:

- From a systems point of view, they are the glue for putting modules together, and - as in real-life - if the glue is bad, the system will break.
- From a process point of view, interfaces are both the means to isolate developers, so they can work independently, and an agreement on how to integrate their finished products into a whole.
- From a developer's perspective, an interface is all that is visible of an abstracted functionality. That is, for somebody looking at a black-box component from outside, the interface "is" the component.

The characteristics of an interface mainly depend on where a decomposition is made. If the point is well-chosen, complexity is indeed reduced as intended, whereas if it is ill-chosen, complexity might even increase. So the question that generations of researchers have tried to answer is "what is the optimal strategy for decomposition?". Unfortunately, none of them has come up with a definitive and formal answer so far (for a discussion, see [MP86]). So when it comes to finding the right place to cut, we must still rely on a number of basic principles, collective experience, rules of thumb and common sense.

2.1 Interface Requirements

Following is a list of basic principles and criteria for the functionality to capture within a module and its interface.

Uniqueness This refers to the principle "Do things once and only once" which is especially popular in the object-oriented community. The idea is to break down a problem into a set of small[2] functions each performing *only one simple task* (e.g. find an item in a list) or encapsulating a *single design decision* (e.g. how to store a list of items). Such functions can be independently and easily understood, implemented and tested and may be combined (reused) in many ways to implement functions of increasing complexity. Since everything is "done only once", it is very easy to maintain, because changes need be made only in one place.

Completeness The intention here is not to group arbitrary statements or half-cooked concepts but to encapsulate something complete that can be named, described and tested by itself. This can either be a function, step in a procedure or a design-decision (see [Par72]). Being able to name an abstraction is actually a good litmus test for proper decomposition.

Maximum Cohesion and Minimum Coupling The idea behind this is to maximize cohesion within modules and minimize coupling (dependencies) between modules. This can be achieved by putting only related functions together in a module (see also [Mye75]). Doing so simplifies understanding, independent development and testing of components. It is also a prerequisite for interface stability.

Interface Stability Interfaces can be viewed as a contract between independent modules/developers (see [Mey88, Mey92]). Just like a real contract, interfaces should stable after being negotiated and designed thoroughly. This is because changes will affect other developers/components and thus contradict the isolating function of the interface. Note: A good design will still allow some flexibility for foreseeable future extensions, so that when they become necessary, the interface need not be changed.

Interface Minimality A module should import and export only the minimal amount of information necessary to carry out its function. In particular, no extra data should be carried through a module unchanged. In short, an interface should convey "as little data and information as possible but as much as needed". This simplifies understanding of the interface. It also helps a module not to give away its inner workings (information-hiding, see [Par72]). For example, if a function needs only a few parts of a compound (e.g. a structure) it might be wise only to pass in this information individually, in order not to make the function

[2] screensize is a good rule of thumb

depend on the structure of the compound. On the other hand, if a function modifies part of a compound, it is better to pass it in and out as a whole, so that not every caller needs to know its structure and re-assign the changed values (see also Consistency and Documentation below). In addition, also the encapsulated functionality should be minimal to promote reuse, maintainability and comprehensibility.

Interface Documentation If somebody wants to use the functionality encapsulated by the module and visible through the interface, there must be documentation suitable both for judging the fitness of the functionality and for properly (re)using it. This includes a summary of the functionality itself, as well as the number, name, position, type and semantics of all the arguments and return values in the interface. Furthermore, for compound and pointer arguments, it should be documented which elements are (only) read from and which are modified. This might seem like a trivial requirement, but since nobody likes to write documentation, it is often not adhered to.

Interface Explicity This asks for all values used or created by a function to show explicitly in the interface as arguments or return values. The implicit use of global variables as inputs or outputs should be restricted as much as possible. Such a restriction is very similar to the Law of Demeter [LH89] for object-oriented software. It leads to predictable behavior by ruling out side-effects[3] and makes a function more straightforward to comprehend and use. Furthermore, it simplifies testing by minimizing the effort for testbeds (see testability below).

Interface Consistency In order not to confuse other developers and avoid mismatches, interfaces (in a family of functions) should be consistent with respect to the name, position and type of common arguments and the implementation of common mechanisms such as error-handling (e.g. by return value or extra argument). Then, developers can quickly recognize common elements and concentrate on the differences.

None of the above is really hard-and-fast. Thus, it should be obvious that the activities in analysis and design (synthesis) where the decomposition normally takes place cannot be performed mechanically or fully automated. They largely remain creative activities.

2.2 Interfaces and Quality

Interfaces also directly affect a number of common quality attributes, and there are design principles to improve them. Following is a list of such quality attributes. Where a corresponding design principle exists, it is mentioned in parantheses:

Testability (Design for Test) Testability of the parts of a complex system is essential, since it cannot be tested in-depth as a whole: It is simply impossible to precisely stimulate all the internal interfaces through the external interface. Therefore, the functionality of every part must be tested bottom-up *before* integration with other parts. A minimum requirement for the testability of a part is to be able to specify/document it by itself and such that test cases can be derived from this information. Additionally, it should have a minimum number of dependencies to other parts and simple interfaces, so that it can either be executed by itself or existing dependencies can be satisfied by a minimal context and simple stubs and drivers.

Reusability The (re)use of existing parts is important for successfully implementing a complex new system. Complex systems are not built from scratch ([Bro95]), and reuse is not only a matter of efficiency, but also of reliability: As reflected by the bathtub curve, time-tested components, i.e. components which are used either often or for a long time, become more reliable - the more they are exposed to different inputs, the more bugs are found and removed. The amount of new bugs introduced in this process and the resulting degradation of reliability depends on the maintainability of the component (see below). In general, by reusing existing parts instead of repeating old mistakes and "re-inventing the wheel", resources are set free for the design and testing of the new parts and the whole.

One important prerequisite for reusability that is often neglected is a well-organized library of well-documented components so that a given functionality can be found and understood when needed. Another important aspect is the flexibility of the part itself for use in different environments. There is an important tradeoff between achieving this flexibility and not ending up with some complicated all-purpose component that can neither be understood, maintained or tested. For interfaces, this means that they must be wide enough so that they can support all *reasonable* variations yet narrow enough so that they can still be understood, precisely specified and tested.

Readability/Understandability Readability refers to the extent to which a software is understandable, typically to another person than the original author. But also the author himself might have difficulties in understanding his own software after, say, a year, if it is not well structured and documented. Without understanding what a piece of software is supposed to do and what it actually does, it is impossible to review it, design test cases for it, evaluate its test results, pinpoint and fix bugs in it or safely make changes to it. Thus, readability is not only important in its own right but also as a

[3] Side-effects are a know source of hidden dependencies and hard-to-find bugs.

prerequisite for other goals like testability and maintainability. Interfaces, being the entry point to a function, are also the starting point for understanding it and limit its (side-)effects by explicitly defining inputs and outputs and isolating one function from another.

Maintainability/Changeability (Design for Change) Software is (seemingly) easy to change, and it *will* be changed many times, both throughout the development process (especially an iterative one as in [Boe88]) and during its lifetime because of changed requirements or bugs that need to be fixed. As a matter of fact, there are a lot more programmers maintaining old software than writing new software. Therefore, it is cruicial that software is not "write-only". As already described in [Wei71], many people write software, but not nearly as many read any. Nor is all software equally readable. Maintainability is very closely related to readability (see above), but it is not the same: Before changing a piece of software, the maintainance programmer must understand it in order to determine what/where to change and what side-effects the change might have on other components. That is the readability part. But even the most understandable software can be hard to change, if it is ill designed, i.e. when there are too many dependencies and side-effects that make the impact of changes unpredictable, when data structures are not extensible, etc. In this context, interfaces are particularly important, because they transport the data and implement or prevent dependencies.

2.3 Kinds of Interfaces

Interfaces can be classified by location or purpose, and each kind has to be treated differently in the development process. The discussion above was centered around internal interfaces of a system. Here are some more kinds, distinguished by location.

External interfaces reflect the context of a system and isolate its inner workings from it. Being the only means of interaction between the system and the environment, they effectively define and limit its possible functionality. The context can already be informally specified very early in development. It is crucial that it is completed, agreed upon by all (external) stakeholders and stable before moving on with the decomposition, because changes at a later time will break down to and affect internal interfaces.

Internal interfaces are a result of the decomposition of the system. By isolating individual modules, they facilitate concurrent development and testing and are the glue to put the pieces back together during integration.

Portability and maintainability layers can appear both inside and outside of a system. On the outside, they are used as wrappers to make a system independent of particular characteristics of its environment, especially those that are beyond the control of the system/project or that may change (e.g. over time or between platforms). Inside a system, they are used to encapsulate design-decisions or generic functions in a single place. Both on the inside and the outside, their ultimate goal is to limit the effect of changes.

Another possible distinction of interfaces is by purpose. This includes their generality and potential for reuse:

Specific interfaces are internal interfaces to functions that do some kind of processing that is particular to a system and that cannot be re-used. Hence, they need not be overly flexible and should be held as simple as possible. All generic functionality should be factored out to maintainability layers or frameworks.

Framework interfaces are external interfaces to functions that offer generic services (e.g. for GUIs or data storage) and can be re-used in different contexts. Therefore, they must offer additional flexibility for various situations. They are typically more expensive to build and only pay for themselves by reuse in the long run.

Programming Interfaces (APIs) are external interfaces to generic libraries or frameworks by which the latter can be used to implement a new programm in the problem domain given by the framework. Of course, they can also be used to extend a framework or build a portability layer on top of it.

Open/Extension interfaces are external interfaces that can be used to extend the capabilities of a general system or tool. Typical examples for this are pluggins, macros, scripts, filters, ...

Please note, that these different kinds of categories are not exclusive. A framework interface for example is typically used as a programming interface. When defining an interface, one should think about its intention and try not to address several purposes in one.

3. Conclusion

Paradigms, notations and programming languages change, but there are certain universal principles that must not be forgotten, and the importance of interfaces is one such principle. Interfaces are one key vessel for achieving the goal of managing and reducing complexity and avoiding to increase it through complicated rules, architectures, etc., especially for safety-critical systems. Or as in [Hat98]: "Software safety, like safety in other engineering disciplines, is about the enforced avoidance of known problems, the avoidance of needless complexity and the adherence to simple and well-established engineering principles which are observed to behave safely.".

References

[Ber] Edward V. Berard. Abstraction, encapsulation and information hiding. Whitepaper, http://www.itmweb.com/essay550.htm.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer, 21 (5)*, 1988.

[Bro95] Frederick P. Brooks. *The mythical man month: essays on software engineering*. Addison Wesley Publishing Company, Inc., 20th anniversary edition, 1995.

[Hat98] Les Hatton. *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*. McGraw-Hill International Series in Software Engineering, 1998.

[LH89] Karl Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[Mey92] Bertrand Meyer. Applying ”design by contract”. *IEEE Computer*, 25(10):40–51, 1992.

[MP86] Stephen M. McMenamin and John F. Palmer. *Essential Systems Analysis*. Prentice Hall, 1986.

[Mye75] Glenford J. Myers. *Reliable Software Through Composite Design*. Van Nostrand Reinhold Company, New York, 1975.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.

[Wei71] G.M. Weinberg. *The Psychology of Computer Programming*. Van Nostrand Reinhold, New York, 1971.