On the Inevitable Intertwining of Specification and Implementation

William Swartout and Robert Balzer
USC/Information Sciences Institute

Contrary to recent claims that specification should be completed before implementation begins, this paper presents two arguments that the two processes must be intertwined. First, limitations of available implementation technology may force a specification change. For example, deciding to implement a stack as an array (rather than as a linked list) may impose a fixed limit on the depth of the stack. Second, implementation choices may suggest augmentations to the original specification. For example, deciding to use an existing pattern-match routine to implement the search command in an editor may lead to incorporating some of the routine's features into the specification, such as the ability to include wild cards in the search key. This paper elaborates these points and illustrates how they arise in the specification of a controller for a package router.

CR Categories and Subject Descriptors: D.2.1. [**Software Engineering**]: Requirements/Specifications—*methodologies*

General Terms: Design, Documentation, Languages

Additional Key Words and Phrases: implementation

For several years we [1, 2, 3, 4] and others [5, 6, 7, 9, 10, 11] have been carefully pointing out how important it is to separate specification from implementation. In this view, one first completely specifies a system in a formal language at a high level of abstraction in an implementation-free manner. Then, as a separate phase, the implementation issues are considered and a program realizing the specification is produced. Depending on the development methodology being employed, this realization is produced either manually (Software Engineering), semiautomatically (Program Transformation), or automatically (High-level Languages and Automatic Programming). The key issue here is not how one arrives at the realization, but rather, that all current software methodologies have adopted a common model that separates specification from implementation.

Unfortunately, this model is overly naive, and does not match reality. Specification and implementation are, in fact, intimately intertwined because they are, respectively, the already-fixed and the yet-to-be-done portions of a multi-step system development. It is only because we have allowed this development process to occur, unobserved and unrecorded, in people's heads that the multi-step nature of this process was not more apparent earlier. Only with the appearance of development methodologies such as stepwise refinement and program transformation did this essential multi-step aspect become clear.

It was then natural, though naive, to partition this multi-step development process into two disjoint partitions: specification and implementation. But this partitioning is entirely arbitrary. Every specification is an implementation of some other higher level specification. Thus simply by shifting our focus to an earlier portion of the development, part of the specification becomes part of the implementation. This explains why it is so hard to create a good specification—one which is high level enough to be understandable, yet precise enough to define completely a particular class of behavior.

The standard software development model holds that each step of the development process should be a "valid" realization of the specification. By "valid" we mean that the behaviors specified by the implementation are a subset of those defined by the specification. However, in actual practice, we find that many development steps violate this validity relationship between specification and implementation. Rather than providing an implementation of the specification, they knowingly redefine the specification itself. Our central argument is that these steps are a crucial mechanism for elaborating the specification and are necessarily intertwined with the implementation. By their very nature, they cannot precede the implementation.

To distinguish such steps from valid implementation steps, we will call them *specification modifications.* They arise from two sources: physical limitations and imperfect foresight. We will consider these in turn.

This research is supported by the Air Force Systems Command, Rome Air Development Center under Contract No. F30602 81 K 0056. Views and conclusions contained in this report are the authors' and should not be interpreted as respresenting the official opinion or policy of RADC, the U.S. Government, or any person or agency connected with them.

Author's Present Address: William Swartout and Robert Balzer, University of Southern California, Information Sciences Institute, Marina del Rey, CA 90291.

The systems we implement employ physical devices (including computers). These devices have limitations (such as speed, size, and reliability) which are specific to the device. Often, one finds cost-effective partial solutions rather than total solutions. This introduces either a restriction that limits the domain of input (e.g., names can only be eight characters) or introduces the possibility of error. In the latter case, one must then define what to do when such errors arise. In either case, the semantics of the specification has been changed, and the alteration is only meaningful in terms of an already fixed implementation decision.

Clearly, such specification modifications cannot precede the implementation decisions they are predicated upon. These "imperfect implementations" are in fact quite common and include modifications due to finite resources or economic considerations and modifications that limit the domain of a specification to a subset of "expected" situations. One reason the specification modifications are not well recognized is that they are usually folded into the "initial" specification, which necessarily therefore also includes (implicitly) the associated implementation decisions, rather than existing explicitly as separate development steps.

The second source of specification modifications is our lack of foresight. The systems we specify and build are complex. We are unable to foresee all the implications and interactions in such systems. During implementation we examine these implications and interactions in more detail in terms of the more concrete implementation being created. Often we find undesirable effects or an incomplete description. This insight provides the basis for refining the specification appropriately. Which version of the specification is modified (i.e., where in the development the modification is inserted) depends upon which implementation decisions need to be reconsidered because of the new insight, and which implementation decisions it is dependent upon.

Such improved insight may (and usually does) also arise from actual usage of the implemented system. These changes reflect not only unanticipated implications and interactions in the implemented system, but also changing needs generated by the existence of the implemented system. Incorporation of these specification modifications is precisely the same as above, i.e., they must be integrated at some appropriate spot in the development.

Thus a much more intertwined relationship exists between specification and implementation than the standard rhetoric would have us believe. Implementation is a multi-step process. Each stage of this process is a specification for what follows. However, many of the steps in this development are not mathematically "valid." They do not implement the specification, they alter it. Many of these specification modifications arise from physical limitations of one form or another. Such "partial" or "imperfect" implementations provide the structure for elaborating the specification to handle the imperfections. The rest of the specification modifications arise from our lack of insight concerning the systems we are implementing. Inadequacies or incompletenesses are discovered during implementation and/or use, and result in the need to revise some appropriate version of the specification and reconsider some of the implementation decisions.

If we were to try to retain the old model of separation of specification from implementation, then we would have to define specification as that portion of the development process beyond which only valid implementation steps occurred (i.e., no specification modifications), and implementation was the rest. Unfortunately, such a distinction can only be made after the fact, and hence is not useful for system builders.

These observations should not be misinterpreted. We still believe that it is important to keep unnecessary implementation decisions out of specifications and we believe that maintenance should be performed by modifying the specification and reoptimizing the altered definition. These observations should indicate that the specification process is more complex and evolutionary than previously believed and they raise the question of the viability of the pervasive view of a specification as a fixed contract between a client and an implementer.

An Example

Consider the following specification of the controller for a package router:[1]

The package router is a system for distributing *packages* into destination *bins.* The packages arrive at a *source* station, which is connected to the bins via a series of *pipes.* A single pipe leaves the source station. The pipes are linked together by two-position *switches.* A switch enables a package sliding down its input pipe to be directed to either of its two output pipes. There is a unique path through the pipes from the source station to any particular bin.[2]

Packages arriving at the source station are scanned by a reading device which determines a destination bin for the package. The package is then allowed to slide down the pipe leaving the source station. The package router must set its switches ahead of each package sliding through the pipes so that each package is routed to the bin determined for it by the source station.

After a package's destination has been determined, it is delayed for a fixed time before being released into the first pipe. This is done to prevent packages from following one another so closely that a switch cannot be reset between successive packages when necessary. However, if a package's destination is the same as that of the package which preceded it through the source station, it is not delayed, since there will be no need to reset switches between the two packages.

There will generally be many packages sliding down the pipes at once. The packages slide at different and unpredictable speeds, so it is impossible to calculate when a given package will reach a particular switch. However, the switches contain sensors strategically placed at their entries and exits to detect the packages.

[1] This specification was obtained from [8].

[2] This is equivalent to viewing the router as a binary tree having switches as nodes, pipes as branches, and bins as leaves.

The sensors are placed in such a way that it is safe to change a switch setting if and only if no packages are present between the entry sensor of a switch and either of its exit sensors. The pipes are bent at the sensor locations in such a way that the sensors are guaranteed to detect a separation between two packages, no matter how closely they follow one another.

Due to the unpredictable sliding characteristics of the packages, it is possible, in spite of the source station delay, that packages will get so close together that it is not possible to reset a switch in time to properly route a package. *Misrouted* packages may be routed to any bin, but must not cause the misrouting of other packages. The bins too have sensors located at their entry, and upon each arrival of a misrouted package at a wrong bin, the routing machine is to signal that package's intended destination and the bin it actually reached.

When we received this specification, we considered it to be an excellent example of an abstract specification which had successfully separated the description of intended behavior from the implementation of that behavior. It was only during our attempt to formalize this example into our specification language that we came to the disturbing realization that the "excellent specification" was contaminated with many implementation decisions. For example, someone has made the decision to use gravity to move the boxes from one switch to the next. Alternately, this "package mover" could have been implemented by, say, a set of conveyor belts. If conveyor belts had been chosen, it might have been possible to make them more dependable than the gravity/chute implementation, and if so, the specification for the controller might not have to deal with "misrouted boxes" at all. Moving up a level, the choices of organizing the switches into a tree and making it binary are both implementation decisions. In fact, the package router could have been implemented using a gantry crane that would pick up boxes at the source and drop them in their appropriate bins. If that were the case, it would not have made much sense to talk about trees, switches, and package movers. Thus we can see that in this example (and we take it to be fairly typical) implementation decisions are made before specification is complete and these decisions can have a major effect on the further specification of the system. Turning things around, if we wanted a specification that contained no implementation commitments it would have to represent information about all the possible implementation technologies, a potentially enormous task.

The package router specification given above also illustrates how an implementation choice can force a modification to the specification. The goal of any package router is to distribute packages into their correct bins. However, particular implementation decisions already present in the specification presented (chiefly, the decision to slide boxes down chutes) introduce the possibility of boxes bunching up, preventing the system from routing all boxes to their correct destinations (because the switches do not and cannot have infinitely fast switching time). The notion of "misrouted boxes" must be introduced to specify what should happen when the goal cannot be achieved. If a different implementation decision had been made which assured that boxes would arrive correctly, the notion of misrouted boxes would be irrelevant.

Conclusion

From the standpoint of constructing aids for capturing the specification and evolution of programs, interleaving specification and implementation into a single development structure will result in a more coherent and realistic structure for making modifications. By contrast, if we attempted to construct such an aid keeping complete specifications and implementation separate, we necessarily would have trouble capturing specification changes like those described above which are forced by implementation decisions.

While the interleaving of specification and implementation seems to occur quite frequently in practice, work directed toward formal specification and aids for creating such specifications seems to have paid little attention to this phenomenon. We have attempted here to illustrate several situations where this interleaving plays an important part in the software development process. Therefore, our software development aids must begin to address these issues.

Received 11/81; revised 1/82; accepted 2/82

References

1. Balzer, R. M. Dataless programming. Full Joint Computer Conference, 1967, pp. 535–545.
2. Balzer, R. M., N. M. Goldman, and D. S. Wile. On the transformational implementation approach to programming. Proceedings of the Second International Conference on Software Engineering, October 1976, pp. 337–344.
3. Balzer, R. M., and N. M. Goldman. Principles of good software specification and their implications for specification languages. Proceedings of the Specifications of Reliable Software Conference, Boston, Massachusetts, April, 1979, pp. 58–67. (Also presented at the National Computer Conference, 1981.)
4. Balzer, R. M. Transformational implementation: An example. *IEEE Trans. Software Engineering 7*, 1 (Jan. 1981), 3–14. Also published as USC/Information Sciences Institute RR-79-79, May 1981.
5. Bauer, F. L. Programming as an evolutionary process. Proceedings of the Second International Conference on Software Engineering, Oct. 1976, pp. 223–234.
6. Burstall, R. M., and J. Darlington. Some transformations for developing recursive programs. Proceedings of the International Conference on Reliable Software, Los Angeles, Calif., April 1975, pp. 465–472.
7. Dijkstra, E. W. Notes on structured programming. In *Structured Programming*, Academic Press, New York, 1972.
8. Hommel, G. (Ed.) Vergleich verschiedener Spezifikationsverfahren am Beispiel einer Paketverteilanlage. Kernforschungszentrum Karlsruhe GmbH, August, 1980. PDV-Report, KfK-PDV 186, Part 1.
9. Knuth, D. E. Structured programming with goto statements. *Computing Surveys 6*, 4 (Dec. 1974).
10. Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Comm. ACM 15*, 12 (Dec. 1972), 1053–1058.
11. Wirth, N. Program development by step-wise refinement. *Comm. ACM 14*, 4 (Aril 1971).