Aspects of Software Adaptability

Article *in* Communications of the ACM · October 1996

DOI: 10.1145/236156.236170 · Source: DBLP

2 authors:

Mohamed Fayad
AITG Inc Aeeh Press Inc i-SOLE inc SJSU
479 PUBLICATIONS **4,665** CITATIONS
SEE PROFILE

Marshall Cline
15 PUBLICATIONS **321** CITATIONS
SEE PROFILE

4. Booch, G. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.
5. Borghi, G. and Brugali, D. Autonomous map learning for a multi-sensor mobile robot using diktiometric representation and negotiation mechanism. In *Proceedings of ICAR'95* (Sant Feliu de Guixols, Spain, Sept. 20–22, 1995), pp. 521–528.
6. Brooks, R.A. A Robust layered control system for a mobile robot. *IEEE J. Robotics and Automation, RA-2,* 1 (Mar. 1986).
7. Chin, R.S. and Chanson, S.T. Distributed object-based programming system. *ACM Comput. Surveys 23* (Mar. 1991), 91–124.
8. E., Helm, R., Johnson, R., Villisides, J. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, NY, 1994.
9. Johnson, R. Documenting frameworks using patterns. In *Proceedings of OOPSLA '92* (Vancouver, B.C., Oct. 1992).
10. Minoura, T., Pargaonkar, S., Rehfuss, K. Structural active object systems for simulation. In *Proceedings of OOPSLA '93* (Washington D.C., Oct. 1993).
11. Object Management Group. *The Common Object Request Broker: Architecture and Specification.* September 1992.

AMUND AARSTEN is a Ph.D. candidate at the Politecnico di Torino in Torino, Italy; email: amund@polito.it
DAVIDE BRUGALI is a Ph.D. candidate at the Politecnico di Torino in Torino, Italy; email: brugali@polito.it
GIUSEPPE MENGA is a professor and chair of Automatic Controls at the Politecnico di Tornino in Torino, Italy; email: menga@polito.it

Aspects of Software Adaptability

Mohamed Fayad and Marshall P. Cline

In today's rapidly changing business environment, adaptability is a critical weapon for survival. Businesses must be adaptable in order to meet increasingly narrow market windows. This need for adaptability at the business level has changed the focus in many businesses from efficiency to opportunity, from reducing costs to generating revenue. For example, an efficient but inflexible system might reduce costs, but might also make it impossible for the business to engage in a new revenue-generating opportunity. Those businesses that desire this adaptability are redoubling their efforts to make their underlying people- and software-systems adaptable, particularly for those systems that could inhibit business-level adaptability.

Because of these forces, software developers need to deal with change like never before. This need for adaptable software systems is driving the move toward object-oriented (OO) technology in many circles. Certainly one of the promises of OO has been its ability to make software more adaptable.

Using OO, however, does not guarantee that the resulting software will be adaptable. Adaptability must be explicitly engineered into the software, even with OO. Furthermore, adaptability is not a generic quality of the software system as a whole: Software systems are adaptable in specific, designated ways, if at all. Therefore the adaptability must not only be explicitly engineered into the software, it must be engineered into the software in places where it will do the most good to the business.

It is no longer acceptable if a software system is correct and solves the problem for which it was designed. Ideally the system will be able to grow and change to solve slightly different problems over time. This corresponds to the three stages of the evolution of software development: Build the right thing, build the thing right, and support the next thing.

Build the right thing corresponds to validation. The requirements must be accurately understood. There continues to be a great deal of effort spent trying to figure out what the right thing really is. In today's world, however, the definition of "right" changes between the time when the perceived sponsor says what they want and the time the software is delivered.

Because the notion of "the right thing" is a moving target, the only way to satisfy the sponsor's desires is to make the software adaptable. Only if the software is adaptable will it be reasonable to change the thing that gets built (which will be "wrong" by then) into the thing that the sponsor wants at that time.

Build the thing right corresponds to verification, correctness, and defect rate. Although everyone has always been concerned with reliability and faithful translation of the requirements into a solution, the market was not always willing to pay a lot of extra money for reliability. Although there still is not a general willingness to pay extra for low defect rates, there seems to be an increasing intolerance for defects. In cases where standards have created a level playing field, customers today appear to have less brand loy-

alty than ever, and are increasingly willing to buy a competitor's product if that will be more reliable. There are probably many reasons for this shift in attitude, among them the fact that software currently runs core business functions rather than just being a tool for finance, and because the average software customer is less technical due to the huge increase in the market breadth. But whatever the reasons, customers are demanding simpler interfaces and more reliable results.

Because fixing defects invariably means changing the software, the best way to build the thing right is to engineer adaptability into the system. Only if the software is adaptable will it be reasonable to introduce changes and/or fix defects with a low probability that the new changes will introduce new defects.

Support the next thing. There are several ways that today's software can provide significant value when building tomorrow's software. One such approach is to build reusable parts that are viewed as "Lego" pieces that can be snapped together later to build something different. This approach clearly works in the small (e.g., with simple class libraries) and in relatively low-level areas such as GUI and database frameworks. However, in our experience it has less success at the higher levels in business applications. There are many reasons for this, but the simplest is the general tension between reusability and usability: The more reusable something is, the more dials and knobs it needs, and this in turn makes it less easy to use.

On the other hand, making something easier to use typically reduces its ability to be reused in a different domain or for a different purpose (single-purpose tools are usually easier to use than multipurpose tools). Because of this and other factors, the high-level components in one product are not generally reusable when building a different product, and making these high-level components more reusable across problem domains typically increases the cost of using them at all.

The other way to leverage current software investment is engineer adaptability into the system, which makes it feasible to adapt the existing system to fit tomorrow's needs. This is slightly different than the reusable parts approach, since the asset in this second approach is the adaptable system, whereas in the reusable parts approach the asset is the pile of parts with which the systems are built. Another way to look at the difference between these approaches is to notice that the reusable parts approach relies on programmers to build tomorrow's systems from today's parts, whereas the adaptable system approach is like building a spreadsheet: The adaptable system can be customized by end users, or at least by a lower caliber of developer.

Spreadsheets were an early example of adaptable systems that were completely customizable by end users. The most recent example is the Web browser, since, in combination with the Web authoring tools, normal users can publish static content in a complex distributed application with a simple WYSIWYG editor.

There are four factors systems generally need to be adaptable; two imply high-level changes (extensibility and flexibility), and two imply low-level changes (tunability and fixability).

Extensibility means it is easy to change the system's capabilities in amount, but not in kind. For example, a system is extensible if it is easy to add another graphical device, or another drawing shape, or another of what it already has.

Flexibility means it is easy to change the system's capabilities in kind. For example, taking something that was a graphical system and making it sensory- or sound-based. Flexibility is often harder than extensibility, especially when on-the-fly changes are desired.

Performance tunability can also be thought of as a change activity. A tunable system can be easily changed in ways that affect performance. For example, CORBA objects are location-independent, which allows objects to be physically moved around on the network (e.g., to reduce network traffic) without impacting very much, if any, code. Java is even more extreme in this aspect, since the objects can be moved to any machine on the network on the fly. Naturally none of this negates the need to do up-front performance engineering work [1]. However the complexity of current systems makes it more and more unlikely that this performance engineering work will be perfect, if for no other reason than the fact the software has changed since the performance engineering work was done, therefore it is increasingly necessary to engineer tunability into today's system.

Fixability is defined as the ability to fix one thing without breaking two other things. This can be very difficult in large systems. It requires, among other things, separation of interface from implementation, and fairly tight specification of the behavior of each component. How design patterns fit in: The most powerful design patterns exist to support some kind of change. Each of these patterns provides a specific axis of change, a specific hinge, where the system will be adaptable. Very generally speaking, use a design pattern wherever you need such a hinge.

References

1. Berg, B., Cline, M., Girou, M. Lessons learned from the OS/400 OO project. *Commun. ACM 38*, 10 (1995), 54–64.
2. Cline, M. and Fayad, M. *Object-Oriented Design Patterns.* (1995) Course notes, 800 pages.

Mohamed Fayad (fayad@cs.unr.edu) is an associate professor at the University of Nevada, Reno. ***Marshall P. Cline*** (cline@parashift.com) is the president of Paradigm Shift, Inc., Potsdam, N.Y.