



Why Program Adaptability Is So Important

Bruce Taylor

Introduction

Every software requirements specification contains, either implicitly or explicitly, a set of desirable attributes. In addition to the implicit ones of correctness and timeliness, there are also performance specifications, fault tolerance characteristics and a whole host of others usually called the “-ilities:” survivability, legibility, maintainability, and so on. And, even if these aren’t in the requirements specification, they are implicit in good software design and implementation practices.

But one of the implicit characteristics, “adaptability” has become so important in recent years that it should be considered specially, and it should be optimized, even at the expense of other characteristics. Adaptability has become the key characteristic of well-designed applications because modern software systems are expected to service rapidly changing business environments efficiently and with robust reliability over a time span of decades. If a system is not designed with this sort of adaptation in mind, it can only degrade to chaos in the face of ever-changing requirements.

Adaptability isn’t a mysterious attribute: it can be designed into your product just like any of the other characteristics. But to build an adaptable system you need to pay attention both to broad engineer-

ing strategies and to the details of implementation.

The Traditional Attributes

In the early days of programming, when memory capacity was measured in kilobytes and cycle times were measured in milliseconds, program size and speed were the most important characteristics. But today, the volume of compiled code is irrelevant because real memory is plentiful and all operating systems implement virtual memory. Speed is still important, but with processor clock rates and optimizing compilers, the speed of code execution is not likely to be the bottleneck for system performance. So program size and speed are becoming much less important than they have been.

Fault tolerance and reliability are, of course, very important. But with databases that incorporate distributed transactions, update replication, and hot swapping of backup systems, data integrity is no longer the sole responsibility of the application. Similarly, server clustering and load balancing hardware takes much of the responsibility for availability out of the designer’s hands. In short, the availability of robust turnkey systems means that reliability is no longer the vital design goal that it has traditionally been.

Characteristics like maintainability and comprehensibility became important

when we realized that program maintenance is as expensive as program development, if not more so. These characteristics arose so that programmers who had never seen the code could extend it and fix bugs in it. But in the agile era, the emphasis is shifting away from maintaining an existing code base to relatively frequent refactoring of the code, so the importance of these maintenance-oriented characteristics is beginning to fade.

Adaptability

The overwhelming demand of modern software systems is that they adapt to solve problems that mutate constantly, and sometimes very quickly. A system may be barely out of its beta testing when it must be changed to accommodate new features like:

1. Offload the heavy computational loads from the application server to an auxiliary computation server,
2. Support databases from multiple vendors, sometimes in the same application,
3. Re-host the application to a new operating system or hardware platform,
4. Change from a client/server model to a web services model.

These are the sorts of pervasive, system-wide changes that make architects swear and programmers despair, and we all wish that requirement were more stable; but rapidly mutating requirements are the reality of the new, agile development process.

Requirements don't stop changing when the application is released: in fact the release of the first version might cause a blizzard of requirements changes as users discover how to fit the application

into their business models. When the lifespan of a successful application is measured in decades, we can realistically expect that every line of application code will eventually be refactored, rewritten, or discarded. In the face of this reality, software adaptability becomes the most importance characteristic that we can build into a system.

Designing Adaptable Software

So, how should software engineers make software systems more adaptable to meet rapidly changing requirements? Most adaptability techniques are just the basics of good software design, but when they are applied consistently and systematically, they add up to adaptability:

Practice rigorous data hiding

Design components that export the business functions, not data. Always protect the source of data with accessor functions, and use service objects to hide complex data.

Maintain architectural integrity

The system as a whole, and every subsystem, should have a well-defined architecture, and components that perform similar functions should have similar architectures. Use standardized, well-understood design patterns whenever possible. Clearly segregate presentation, processing, and persistence functions, and enforce the separation vigorously.

Design systematic interfaces

When you design a service interface, make sure it is complete and follows a consistent logical model. Using an *ad hoc* model that suits only today's requirements will put you in a poor posi-

tion to extend or adapt the component in the face of changing requirements.

Write transparent code

Use the simplest algorithm that gets the job done, and resist the urge for premature optimization. If an application is to have a decades-long life, it is important that the next generation of engineers understand the intent of the designers, and not just their coding techniques.

Adhere rigidly to standards

Carefully choose and document a set of architectural, design, and coding standards, and stick with them. When you come to a situation that seems to demand that you bend the standards, first check to see if your standards still reflect the reality of the application and its environment.

Pay attention to coupling

Make sure that component interfaces promote low levels of coupling. For example, every method that uses an “operation code” parameter to control its operation becomes a problem when new operation codes must be defined and used consistently in multiple subsystems.

Encapsulate all platform function

Every platform-dependent function, including database access, file operations, and application server functions should be wrapped in a facade layer that exposes only the features that your application needs. This protects your application when you need to re-host your application on another database, application server, or operating system.

Keeping it Adaptable

If you have been foresightful enough to release a very adaptable application, how can you ensure that it retains the same high resistance to change as it evolves? You will find that this sort of maintenance is a balancing act between adhering to existing standards and patterns, and being willing to discard old patterns when they are no longer useful.

Maintain your architectural vision

Over the years, the architectural and design staff will come and go, but the application will live on. It is important that you hand down to new staff the design history of the application, and the set of architectural and design patterns that went into it. This institutional memory will help them understand the original intent of the designers and how to preserve the application’s adaptability over time.

Understand the scope of changes

When you make a change in a component, it’s not enough to understand the local changes; you need to understand how the change will affect other components that work with it. The most obvious example is a change to the database schema: you need to understand the change’s effect on the entire system, and adapt every affected component to use it effectively. There are more subtle examples, like the synchronization patterns of multi-threaded applications, or changes to the user interface model. Whatever the change, make sure that you implement it consistently throughout the whole application.

Refactor, don’t patch

For all but the simplest modifications, consider refactoring the component instead of simply patching the code. There will be a great temptation to take the path of least resistance, especially when the change is completely within a component, but these sorts of ad hoc patches tend to accumulate and interact to produce a component that can't be easily changed because it no longer fits in with the rest of the system architecture. Over time, the effort you save in maintenance will more than offset the time you spend redesigning and refactoring components on a regular basis.

Conclusion

New software marketplaces have requirements that mutate rapidly and continuously, and this trend is not likely to reverse. On the contrary, when the lifespan of applications is measured in decades, virtually all of the original system requirements will change as the business models change.

So to be successful in a changing market, modern applications must be adaptable in many ways and dimensions. This need to handle changing requirements is so strong that adaptability has become the primary quality of successful software.

Building adaptable software isn't a mysterious art: it is mostly a result of understanding the nature of software change, of adopting good architecture and design patterns, and of rigorously enforcing design and coding standards.

About Bruce Taylor



Bruce Taylor is the principal of CoachingProgrammers.com, an executive coaching firm located near Boston, Massachusetts. Bruce helps software organizations of all sizes to create low-stress, supportive, adaptable working environments, so that the engineers, leaders, and managers can work as effectively as possible. He provides executive coaching for senior managers who are creating superior organizations, management coaching for technical leaders who are adapting to new agile practices, and individual coaching for engineers who are upgrading their skills. Bruce has a Masters in Computer Science from Duke University, a Masters in Community Psychology, and a Certificate in Job Stress and Healthy Workplace Design, both from the University of Massachusetts.