How To Build Utterly Reliable Systems

fold

Table of Contents

Introduction

Someone once said of iMatix, “they can solve any problem, no matter how large”. We sometimes say, only half-joking, that if a project is not terrifying, it is not worth doing. At the same time, since we started our business in 1996, iMatix has never failed to deliver on a project, no matter how large and terrifying. So obviously we are doing something right. My goal in this paper is to explain how we turn “terrifying” into merely “tedious”.

The key requirement for all our projects – we do not make consumer software – is reliability. Clients obviously want their projects to work accurately, to be done on time, and economically, but reliability is always the prime requirement, and the hardest to achieve.

In this white paper I'll explain how iMatix builds utterly reliable software systems. We have done this for decades, the principles are well understood, and apply to all domains of software design, development, and use. Designing for reliability also makes the overall project easier to manage, and reduces the risk of cost and time overruns, and of functional errors.

In our experience the reliability of a system depends on two related aspects: the overall complexity of the system, and the individual reliability of its pieces. In general more complex systems have less reliable components and we believe this is direct cause and effect: developers are less able to build reliable components in a complex architecture.

So, the fundamental challenge facing all software architects, though only good architects realize this, is how to solve complexity. We solve any given technical issues simply by applying effort, money, resources. But we see even the most well-funded projects collapse under their own weight if badly designed. People confuse complexity for value, simplicity for naivety, when the truth is opposite. It is hard to build simple systems, easy to make complex ones.

General Principles

I'll look at the main principles that guide our work on large systems. Where possible I'll provide counter examples to demonstrate why failing to observe the principle causes problems.

Architecture Does not Just Happen

It should go without saying, but people seem to forget this. A successful architecture is the work of a skilled architect who invests in the job. Just as

Links

- Portfolio of iMatix's utterly reliable projects

a vibrant, livable city is the work of many generations of skilled planners, and a mega-slum is the result of unplanned growth, so software systems that have no architects will inevitably become slums.

We see several classic scenarios where no architect is put in charge of the overall design:

1. There may be no clear business owner of the overall system, so no-one is willing or able to nominate an architect and take responsibility for the global quality.
2. Competition between vendors can prevent a single person acting as architect.
3. There may be no competent architect available at all.
4. It may not be obvious to the business that there is an architecture issue.

As counter-example we discuss the AMQP project from 2006-2007. This workgroup consisted of pro-bono contributions by various firms. iMatix delivered to the workgroup an original version of the protocol, which we consider to have been well-designed (naturally, we were the architects). This version of AMQP is, by all accounts, simple to understand and implement, robust, and in general a product we are proud of.

The workgroup was, and still is, characterized by a lack of architect, and a lack of architecture. The design has become very complex, and the protocol process itself has become stuck under its own weight.

In this workgroup, the discussion of “do we need an architecture” was itself controversial. Competition between vendors to contribute to, and dominate, the protocol meant no single person could take responsibility for architecture. Architecture was inseparable from detailed design. And so on.

Just as the AMQP/0.8 release stands as an example of a well-architected protocol, the AMQP/0.9 working group release stands as an example of what it looks like when one allows a slum to grow in the heart of a city.

Architecting a software system does *not* mean micro-management. In fact, it has the opposite effect, of creating much more room for delegation and local ownership. We see, in the wider world of software, many examples of perfectly architected systems that work smoothly with unplanned and unpredictable change, yet produce overall systems that are simple (compared the problem they are solving) and reliable: Wikipedia, Debian GNU/Linux package management, Unix.

Architecture is About People

The term “software engineer” suggests that software is a material like steel or carbon fibre. In fact writing software is a lot more like cooking, designing clothes, or classic architecture of homes and offices. We do solve technical issues but we always solve them in the context of people. People have limitations, make mistakes, and need help in certain ways. This applies as much to those making the software as those using it.

People are, above all, very bad at remembering arbitrary complexity. We like patterns, things we can learn and predict, rules we can depend on.

At iMatix we assume that any developer – especially a very good one who works rapidly and with broad strokes – will make errors at a rate of one or two per ten lines of initial code as he/she writes it. The development process aims to uncover these errors as rapidly as possible, ideally catching 100% before the code is committed. We don't assume we can write perfect code, because we know from experience that is impossible. So we architect our tools and development environment to compensate.

This is the same challenge for a systems architect: assume every component is born full of errors. How do you uncover these as rapidly as possible, give the developers every assistance in finding and fixing those errors. How do we help developers *prove* their work to be correct?

These are rhetorical questions, which I answer in the following pages. I hope you accept my thesis that our real work is compensating for human, not technical, weaknesses.

As a counter example, I'll look at an example from the non-software world, city architecture. In the 20th century we saw many examples of technically accurate but inhumane architectures for homes and offices. Tower blocks of apartments that became ghettos. The destruction of old city centers in the name of modernization. Business areas that died every evening as the workers went home. Motorways called “long term parking lots”.

Divide and Conquer

A good architect starts by cutting large problems into smaller pieces, like a diamond cutter breaks a large stone into smaller pieces. Personal and collective experience guides the knife. There is an old joke about a compiler construction (a problem that computer scientists used to solve in the 20th century): “how many people does it take to write an N-pass compiler?” Answer: “N + 1, one person per pass plus a project leader”.

It took compiler writers a long time to understand how to split the problem into “passes” that could be solved independently by individual engineers. The ideal division of a large problem is rarely obvious.

However, there are some general rules that help:

- Pieces need to fit people. Ultimately, architecture is about fitting the problem to people. If a problem fits neatly to one developer or one small team, it has a good size. If a problem can only be solved by collaborating teams, it's badly sized.
- Interfaces define the fracture points. It's best to slice at the point where the interface is simplest. The interface will become a contract between individual developers, or teams. The simplest contract is the best one.
- Every problem can be deconstructed. If an architect cannot break a large problem into pieces, he or she is not competent. Sometimes lateral thinking is needed. But we have never seen large problems that could not be divided up.
- The architecture is a contract. It must be clear enough to create boundaries, between teams and layers, that cannot and never need to be crossed except through agreed interfaces.
- Decouple the change process. The architecture should package change into clean boxes so that the overall system can bes both stable and dynamic.

The overall architecture, showing the different pieces and how they interface, is the first product of an architect. If the architecture is too complex, if it has too many pieces or too many lines, the work is not finished.

One of my rules is: the architecture of any system, no matter how big, can fit onto one page. And I don't mean, using a 3pt typeface.

The Linux kernel is a great example of a successful architecture. It defines abstractions like “kernel modules” that deconstruct an immense problem into human-sized pieces. It creates contracts between different layers, so that code can survive for decades. It decouples the change process so that new functionality can be added at any stage, even to running kernels.

As a broader example, I'll compare two well-known operating systems, Windows and Unix. Windows is typified by large monolithic applications (like SQLServer) that solve everything from user interface to security in one package. Unix is typified by a vast number of generic pieces that can be combined according to a set of clear rules (pipes, libraries, etc.)

The Unix model is so successful that it's been adopted by FreeBSD and Linux and is arguably the most successful operating system design ever[1].

Windows, on the other hand, remains stubbornly expensive to develop, weak in terms of security, reliability and performance. In 2008 the RAM industry is facing a collapse because expected sales of Windows Vista have not materialized.

Formalize the Interfaces

Every interface is a contract between two parties (or rather, two categories of party) and must be formalized as far as reasonable. Typically we formalize some or all these aspects of an interface:

1. What information we exchange between the two parties (the semantics);
2. How we exchange this information (the syntax);
3. How we handle errors of different types;
4. How we negotiate, report, or extend the interface;
5. How we evolve the interface over time;

A good interface is regular, predictable, utterly consistent, and overall simple to understand. An interface that is “complex” is a recipe for unreliable applications. Further, a good interface is inflexible, it has no shades of gray, no flexibility when things go wrong. We want the problems in a system to be seen sooner, rather than later.

There are sadly many examples of complex and irregular interfaces. Simple successful one are much harder to find.

Identify and Eliminate Risk

Risk is always relative, and one reason we pay skilled people to help us design architectures is that they should be able to eliminate risk. There is a classic set of risks that face all non-trivial software projects, and each can be eliminated through careful architectural choices:

- The risk of dependency on specific operating systems, languages, or other platforms. We eliminate this by creating or reusing portability layers.
- The risk of vendor lock-in. We eliminate this by favouring open source technologies and open standards.
- The risk of dependency on key staff. We eliminate this by making it possible and economic to re-engineer any part of the architecture if necessary.
- The risk of human error. We eliminate this by assuming people make mistakes, and designing our processes to catch those (rather than demanding that people be perfect).
- The risk of design failure. We eliminate this by designing progressively, especially in new areas where we also need to learn. Sometimes we deliberately discard designs and start afresh.
- The risk of budget or schedule overruns. We (usually) eliminate this by working minimalistically, never implementing any functionality that is not needed.
- The risk of badly implemented components. We make sure every component is fully testable before it is plugged into the architecture.

Usually, reliable software does take longer to develop but is much cheaper to maintain.

Eliminate Dependencies

One of the biggest headaches in complex architectures is that changing one part has unexpected consquences elsewhere. You may have heard of programmers who experience this in their code – it is a sign that the code is poorly architected.

We make architectures easier to change by eliminating dependencies between components. For example, imagine if a Linux kernel module had dependencies to other kernel modules. Any change in one of these could affect the others in unforeseen ways.

The key notion here is the difference between “dependency” and “interface”. If two components need to communicate, they do this via a formal interface that can be documented and that allows each component to be tested in vitro.

An unwelcome dependency would be two applications that share the same database tables.

Ensure Full Testability

People sometimes speak of “unit tests” and “integration tests” but in our view these should be the same. That is, our goal should be that a unit should operate the same whether it is in a laboratory, or in a real architecture.

When we define formal interfaces, it becomes relatively easy to write testing frameworks that implement each half of the interface. For example, if we look at Linux kernel modules, I can write a dummy module that lets me test the kernel. I can write a dummy kernel that lets me test a module. When I put a tested kernel together with a tested module, I'd expect them to work perfectly together. If they don't, the fault is in the interface or in the tests. I can improve these, and the next time I write a module I know it will be fully testable.

Writing such test frameworks can be expensive but it lets team deliver near perfect code, and aim for utter reliability.

Conclusion

In this white paper I've explained our view at iMatix that reliability comes from good design, and that this means deliberate and careful attention to the overall architeture of a system, and the relationships between its components.

I've also explained a number of key techniques that we use to create more reliable systems, and at the same time, solve very large problems.

If you'd like more information, please contact me on ph@imatix.com.

Footnotes

1. In my opinion, Linux is the TCP/IP of operating systems and will spread inexorably to cover the whole computing universe