Editor: **Diomidis Spinellis**
Athens University of Economics and Business, dds@aueb.gr

Lessons from Space

Diomidis Spinellis and Henry Spencer

We can lick gravity, but sometimes the paperwork is overwhelming. —Wernher von Braun

Post your comments online by visiting the column's blog:
www.spinellis.gr/tools

TWILIGHT SAW THE landing of Atlantis at the Kennedy Space Center on 21 July 2011, marking the end of the 30-year Space Shuttle program and leaving the Soyuz series of spacecraft as the only remaining major option for sending humans into space. With a history of 1,700 flights over an almost half-century lifetime, the Soyuz rocket and spacecraft are arguably a tremendously successful spaceflight design. Given the parallels between the complexity of human spaceflight and large software systems, what can we as developers learn from the Soyuz program?

Go for the 80 Percent

There's no question that the Space Shuttle design was more sophisticated and technically ambitious than that of Soyuz. So was the design of the Multics operating system, the PL/I language, and the OSI networking standards. Yet today's astronauts fly on Soyuz and much of the world runs on Unix, C, and the Internet. It seems that limiting a project's scope and complexity early on can have a dramatic payoff in its success and longevity.

Soyuz is an "80 percent solution." It's not everything its original designers wanted, never mind what its current users want, but it does what's needed and most of what's wanted. Its capacity is a fraction of the Space Shuttle's (see Figure 1); it's cramped, not reusable, and can't run Spacelab missions or deploy satellites. However, its bare-bones design reduced construction and operation costs along with the opportunities for problems, while still making Soyuz adaptable to the variety of roles it was called to fulfill.

The complexity of software-based systems is often staggering, so a frugal design's payoffs can be equally dramatic. Complexity feeds on itself. A lean and mean design results in a faster time to market, a smaller number of developers or contractors, a simpler management structure, sleeker processes, shorter build cycles, and fewer and shorter meetings (hurray!). Such a design also promotes agility; a small speedboat can run circles around a weighty freighter. In today's rapidly changing business environment, you're a lot safer with a design you can adapt in a matter of days or even ditch without qualms to start afresh.

However, too many software systems are built like the Space Shuttle, with a long list of ambitious goals and a resulting system that's badly compromised by trying to do too much. Gold-plated requirements frequently distract management and engineers, diverting talent and precious resources from those functions with the highest payoffs. Unfortunately, all too often the upfront cost of large-scale procurement processes promotes the all-but-the-kitchen-sink approach to requirements, penalizing small and nimble projects. The next time you see a fully configurable report generator in an information system's specifications, ask yourself whether a simple data export facility would suffice.

From the hardware perspective, today's stalled CPU clock speeds and memory access times as well as the power constraints of mobile devices also require us to reduce bloat. Lean designs can have lower processing and data throughput needs and can thus be more responsive and energy efficient.

0740-7459/11/$26.00 © 2011 IEEE

Leave Margins

The Space Shuttle's development was badly hampered when a performance shortfall required drastic weight reduction measures throughout the orbiter (such as deleting built-in work platforms that would have made maintenance quicker and cheaper). As best we can tell, Soyuz had a big enough design margin that it never went through such a traumatic experience.

The real shining example of the importance of margins, though, was the Apollo spacecraft and its *Saturn V* rocket. A fundamental decision had to be made early in the rocket development: How big was the payload? Consequently, the spacecraft designers had to commit to a weight limit quite early in their development process.

After considerable thought, they promised chief rocket developer Wernher von Braun that the final spacecraft would weigh 75,000 pounds at most, its definite absolute maximum. He could confidently build the rocket to carry that much payload. Von Braun thought about the history of earlier Apollo weight estimates, decided that he just didn't believe it, and told his department heads that while the official payload was 75,000 pounds, the real requirement was 90,000 pounds! This was going to make the rocket substantially bigger; among other changes, the first stage needed five engines, not four.

The wisdom of this became clear a few years later. The *Apollo 11* spacecraft, after stringent weight-reduction efforts, weighed just over 109,000 pounds at launch. Making the *Saturn V* lift that much was difficult, but it would have been utterly impossible if von Braun had taken the spacecraft builders' original rash promise at face value.

Software development works in the same way: estimates of execution speed, memory use, and development time early in a project are notoriously overoptimistic. This happens because the knowledge we acquire during the project's development increases in scope and complexity. Clearly, the earlier a working prototype can be had, the better, so that estimates can be based on actual data. Also, adding generous margins to early estimates (and any subsequent revisions) will almost always ease the pain of development and deployment.

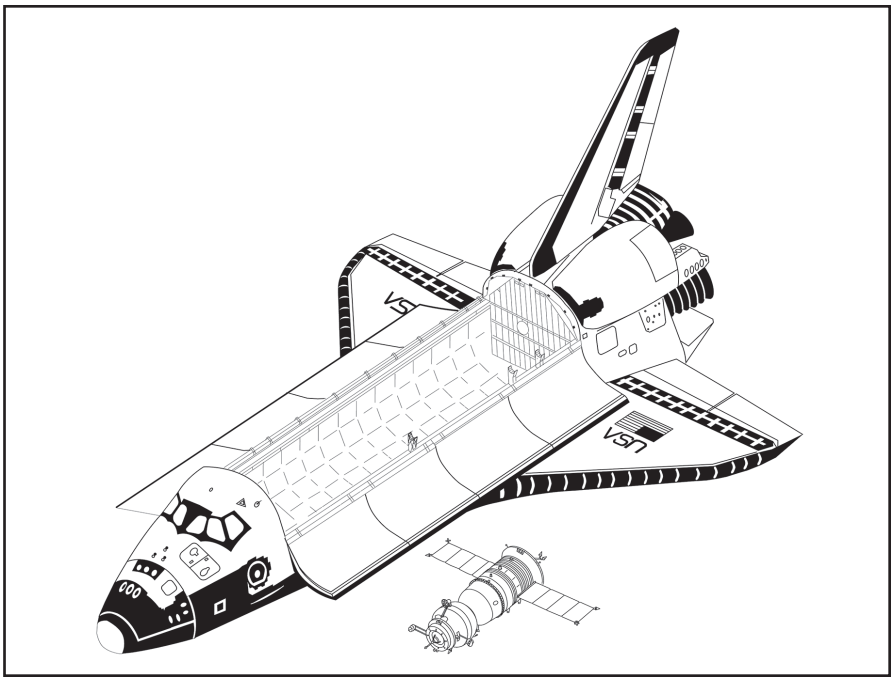

FIGURE 1. The Space Shuttle orbiter and the 1986–2003 Soyuz model drawn to scale. The Soyuz isn't reusable and doesn't offer the same capacity, but its bare-bones design reduced both costs and opportunities for problems.

If It Ain't Broke, Don't Fix It

Since 1967, the Soyuz spacecraft currently ferrying passengers to the International Space Station has evolved through more than a dozen variants. The Soyuz rocket family goes even further back in time, deriving its design from the 1960 Vostok launcher, which in turn was based on the 1957 R-7 intercontinental ballistic missile. Despite a limitation or two, Soyuz does its job well, and the Russians have seen no need to replace it. (The Buran spacecraft was never meant as a Soyuz replacement—it was a purely military project.) NASA has tried to replace the shuttle several times; the only reason it didn't was because it never managed to bring any of those plans to fruition.

Soyuz and its rocket have been upgraded several times; they're descendants of the ones that flew nearly half a century ago, not exact copies. But the improvements have been incremental and the transitions gradual. There have been no sudden "flag day" switchovers when the old system is decommissioned and the new one must work the first time it's tried, and no long gaps between systems. The software analogy is obvious: gradual evolution with a working program at each step, rather than massive rewrites. There are at least three good reasons for sticking to a good design rather than continually starting afresh.

First, software architecture and design remain art as much as science. Although we can easily spot signs of a deficient design, such as excessive coupling, we can only tell a good one once

it's proven itself in practice. Given that a miniscule subset of all possible designs will turn out to be truly stellar, it's sensible to recognize them as such and stick with them.

Then we have the people aspect. Designs do not exist in a vacuum; they depend on people who know how to make them tick and nurture them in the right direction. All too often, developers fresh on a project end up trying to hit a screw with a proverbial hammer. This happens because only a small part of the knowledge embodied in complex artifacts is documented and taught. The rest is hidden within the designs and the teams that build them. Relying on existing designs and teams allows us to capture that tacit knowledge and avoid costly mistakes. It's noteworthy that the people who built Soyuz were the same ones who built Voskhod, Vostok, and Sputnik. One thing the US space program hasn't done well at all is to keep successful teams together and give them more missions. Devising a mentoring program and meaningful career ladders for software developers is similarly important and challenging.

Finally, incremental improvements in software also help us keep the ecosystem we nurtured: our customer base and third-party contributors. Microsoft played this card with gusto as it stubbornly built the first versions of Windows on top of the creaking MS-DOS infrastructure, beating to the marketplace IBM's initially more ambitious OS/2 project.

Modularize and Specialize

In contrast to the Space Shuttle orbiter, which combines engines, habitation, and reentry functionality, the Soyuz program followed a modular design. A Soyuz spacecraft consists of an orbital module, which accommodates the crew during its mission; a small reentry module, which by decelerating through airbraking, a parachute, and retro-rockets, returns the crew back to Earth; and the service module, which contains instrumentation, propulsion, and the solar array. This assembly is launched by means of the three-stage Soyuz rocket.

Splitting the Soyuz structure between the rocket, orbital, reentry, and service modules let the designers optimize each part for its intended functions and avoid dangerous and costly interdependencies. The interface between a capsule and its rocket can be quite simple; the Apollo project had fewer than 100 wires connecting it to its Saturn rocket and just 36 wires between the Apollo command/service module and the lunar module. The Shuttle orbiter, more tightly integrated with the external tank and the solid rocket boosters, had more than 1,000 wires between it and the rest of its stack, requiring an entire work shift just to mate the electrical connectors during shuttle stacking. Worse, this is the difference between something one engineer can easily understand and something no single human has ever fully understood.

Soyuz's modular design also allowed its builders to create the Progress cargo hauling variant, and Zond, which removed the orbital module and beefed up the reentry-module heat shield, providing a minimal manned spacecraft for a round-the-moon flight. This versatility yielded economies of scope and scale further driving the Soyuz program's success.

The analogies with software designs are obvious and the outcomes similar. Microsoft paid the price for the tight integration between the Windows kernel, the user interface, and Internet Explorer through an endless stream of critical security vulnerabilities. Contrast this with the modular design of Unix tools that connect through the simple pipes-and-filters architecture, which has kept them popular for almost as long as Soyuz. Various software ecosystems, such as those of Emacs, Perl, TeX, Java, Eclipse, the iPhone, and the Unix package managers, also owe their success and popularity to a well-defined modular structure for add-ons.

A less obvious lesson concerns the handling of critical functionality. Customizing the Soyuz orbital module (a common requirement) doesn't affect the life-critical descent module. Similar opportunities abound in software and can promote flexibility. For instance, it's often a good idea to split a design's core and its interface using a scripting language. This aids performance, security, and reliability. The software analogy goes even deeper. One way people have successfully tackled the problem of safety-critical software is to split the software between two computers: one does nothing but run the hardware and enforce the safety invariants, while the other, with its fancy displays and complex data acquisition, can give orders to the hardware only through the first. Keeping the safety-critical hard-real-time code on a separate computer isolates it from misbehavior by the rest of the code much more effectively than any level of software precautions.

Designing a software system like the Shuttle—a gleaming, integrated whole meant to fully meet a long list of requirements—is appealing, not least because it often fits the customer's preconceptions well. In real life, though, it's often wiser to emulate Soyuz instead: a simple, modular solution that solves the most important problems and can evolve to handle future changes.

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). Contact him at dds@aueb.gr.

HENRY SPENCER is a freelance consultant for software development and spacecraft engineering, and an amateur space historian. His code is in orbit on five satellites—possibly more by the time you read this—built by the University of Toronto's Space Flight Laboratory.