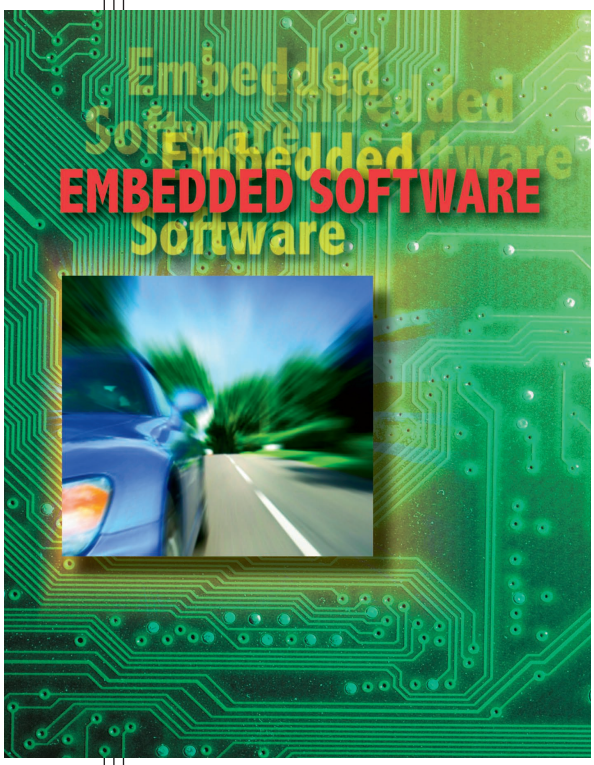


EMBEDDED SOFTWARE: FACTS, FIGURES, AND FUTURE

Christof Ebert, *Vector*
Capers Jones, *Software Productivity Research*

Due to the complex system context of embedded-software applications, defects can cause life-threatening situations, delays can create huge costs, and insufficient productivity can impact entire economies. Providing better estimates, setting objectives, and identifying critical hot spots in embedded-software engineering requires adequate benchmarking data.

Embedded software shapes our world. It is difficult to imagine day-to-day life without it. Examples of embedded software include pacemakers, cell phones, home appliances, energy generation and distribution, satellites, and automotive components such as antilock brakes. Embedded software creates both huge value and unprecedented risks.

Pacemakers are a good example of how embedded software helps millions of persons live a better life. Yet between 1990 and 2000, firmware errors accounted for about 40 percent of the half million devices recalled.[1] For a person with a pacemaker, the odds that the device's manufacturer will recall it or issue a defect warning over a one-year period are about one in 15. For implantable cardioverter defibrillators—more sophisticated devices that can deliver a strong electric shock to avert sudden death—the odds of a warning are even higher: nearly one in six.

The worldwide market for embedded systems is around 160 billion euros, with an annual growth of 9 percent. Figure 1 shows the size and annual volume of selected embedded software.[2,3] While these statistics are comparable to the world's biggest software packages, such as Microsoft Windows, embedded software is far more complex due to the real-time and interface constraints that do not affect IT, application, or desktop software.

The embedded and information systems communities tend to exist in almost complete isolation from one another. This holds for conferences as well as for organization layout and products. Embedded-software engineers typically don't attend mainstream computer shows or software engineering conferences, but rather attend their domain-specific events, such as the SAE Convergence series, because they relate software engineering to specific industry domain challenges and solutions.

 Published by the IEEE Computer Society

We therefore want to provide an overview of techniques and methods that impact embedded-software engineering. Instead of diving into one specific development methodology, we want to bootstrap a measurement-driven approach focused on improvements in embedded-software engineering.

MEASURING EMBEDDED SOFTWARE

In 2008, there were some 30 embedded microprocessors per person in developed countries with at least 2.5 million function points of embedded software.[2,4] As more devices become automated and consumers acquire more such devices, the volume of embedded software is increasing at 10 to 20 percent per year depending on the domain. Embedded microprocessors account for more than 98 percent of all produced microprocessors, thus vastly surpassing computing power in the IT industry.

Figure 2 shows the evolution of some embedded systems in terms of software size over time—namely onboard software in spacecraft, telecommunications switching systems, automotive embedded software, and the Linux kernel, which serves as the basis of many embedded systems. Windows Mobile and other embedded operating systems are evolving at the same pace. We use these examples because we have been working on these components.

The growth rate of embedded software has accelerated over the past decades. For instance, new cars currently have 20 to 70 electronic control units with more than 100 million object code instructions, totaling close to 1 Gbyte of software in a premium car. Value creation in cars is primarily determined by embedded software, resulting not only in increased cost and complexity, but also in increased potential defects from embedded software. While mechanical defects are decreasing in rate, defects caused by electronic systems are increasing rapidly.

But how do we assess the defect density of embedded software? How do we evaluate supplier schedules? To estimate, set objectives, and identify critical hot spots in development, testing, and project management, we need industrial benchmarking data, such as expected defects. Where do you get such initial data? This data might not be readily available, or it is not yet scalable for new products, methodologies, or projects.

While researchers have begun to publish increasing amounts of data for standard software, this is not the case for embedded-software development. There are many informal claims for tools, languages, and methodologies, but empirical data on their actual effectiveness in terms of quality or productivity is rarely collected. The reason is simply that embedded software tends to "disappear" within the surrounding systems. It is highly specific to its environment, thus making empirical studies difficult.[5]

To provide facts and figures, we draw from our measurement experiences and present quantitative data accumulated over our combined 60 years of embedded software engineering experience.[2,3,6] Knowing that it is often difficult to use simple numbers to characterize a situation, we also provide concrete and fact-based guidance from our own experiences so you can use it as a baseline in your projects. Clearly, this is not a substitute for your own measurement database, but it does provide a starting point.

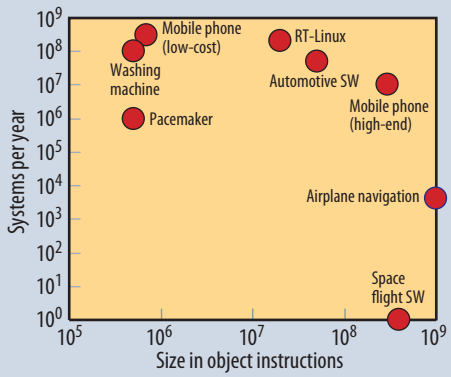


Figure 1. Embedded software size and deployment.

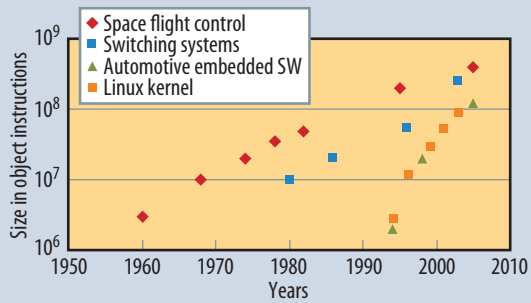


Figure 2. Complexity growth of embedded systems.

PROJECT LESSONS AND COST ESTIMATION TOOLS

We started by looking into our own project lessons learned and the cost-estimation tools on which we have been working for decades. We have continuously verified this data with client visits and published experiences.[1,3,4,6-13] Over time, this experience has provided a broad basis of data points from embedded projects and products. From this we derived simple rules of thumb (or heuristics) that are applicable even in situations where no historic information is accessible.

This article uses function points as defined by the International Function Point Users Group and assumes version 4.2 of the IFPUG counting rules.[3,4] Adjustments to the data would be needed for COSMIC function points or other variations on IFPUG counting methods.

To improve your own embedded development processes, and to ensure that benchmark data applies to your environment, we strongly suggest building your own

history database with baselines for estimation, quality planning, and the like. To get started without much overhead, we recommend the following lean set of effective project indicators:[2,6]

- *Requirements status and volatility.* Requirements status and change rate is a basic ingredient for tracking progress based on externally perceived value. Always remember that you are paid for implementing requirements, not for generating code. Too many changes indicate that the project was not sufficiently prepared.
- *Product size and complexity.* Size can be measured in function points or as code size in lines of code (LOC) or statements. Be prepared to distinguish between what is new and what is reused or automatically generated code.

Embedded systems heavily influence design and engineering constraints of their respective surrounding systems—and vice versa.

- *Effort.* This is a basic monitoring parameter to ensure you stay on budget. Effort is estimated up front for the project and its activities. Afterward, these effort elements are tracked.
- *Schedule and time.* Monitor results, increments, and milestones to ensure that you can keep the scheduled delivery time. Similar to effort, time is broken down into increments or phases that are tracked based on what has been delivered so far. Note that milestone completion must be aligned with defined quality criteria to avoid detecting poor quality software too late.
- *Project progress.* This is the key measurement during the entire project execution. Progress has many facets and should monitor deliverables and how they contribute to achieving the project's goals. Typically, there are milestones for the big steps and earned value and increments for the day-to-day operational tracking. Earned value techniques look to the degree with which results such as implemented and tested requirements or closed-work packages relate to effort spent and elapsed time. This lets us estimate the cost and remaining time to complete the project.
- *Quality.* This is the most difficult measurement, as it is hardly possible to forecast accurately whether the product has already achieved the quality level expected for operational usage. Quality measurements need to predict quality levels and track discovered defects against estimated defects. Reviews, unit test, and test progress and coverage are the key measurements to indicate quality. Reliability models are established to forecast how many defects still need to be found. Note that quality attributes are not only functional but also relate to performance, security, safety, diagnosability, and maintainability.

This set of measurements applies to project tracking and oversight from a product- and contractor-management perspective and thus keeps measurements lean yet effective. These measurements are state of the practice in embedded-software engineering and thus are necessary if you need to justify development practices and your risk management in, for instance, litigation.[13] Consider the Heisenberg Uncertainty Principle for Software: Accurate estimating and measurement change the project. The more you know what's going on, the more you can influence and improve. Measurements have impact, and with more impact, their usage and benefits will grow.

DEVELOPMENT PRACTICES

Embedded-software systems pose extraordinary challenges to the software engineer due to their complexity. The main source of complexity is the large number of subtle and often unexpected interactions among the various parts of these systems, which have the following common features:

- functionality represented by states and events;
- real-time behavior of events and expected actions;
- combined software/hardware systems equipped with distributed software, computers, sensors, and actuators;
- high demands on availability, safety, information security, and interoperability; and
- long-lived systems in which embedded software is expected to work reliably.

Embedded-software development practices vary to a high degree across industries. Mostly, they evolve at different speeds and without much cross-fertilization. One reason is that embedded developers often do not really consider themselves "software engineers." By training, many of them are electrical engineers, automotive engineers, or telecommunications engineers, or have some other background. They don't want to be viewed as software engineers because software engineering has a lower professional status than more mature forms of engineering—particularly the types of engineering that have certification and licensing requirements.

Design and engineering constraints

Embedded systems heavily influence design and engineering constraints of their respective surrounding systems—and vice versa. To illustrate, we researched

automotive embedded electronic control units (ECUs) over time and analyzed how they impact design decisions. While in the 1980s the majority of electronics in a car came from the radio and the engine controller, automotive electronics increased significantly to provide safety, and, more recently, comfort functions. In the past, primary car buying criteria were power, speed, and design. Today, buyers demand energy efficiency, safety, and comfort.

Figure 3 shows the evolution of embedded systems in cars since 1985. The upper line shows the number of ECUs in high-end models at release time, while the lower line shows the electric energy consumption of these ECUs.

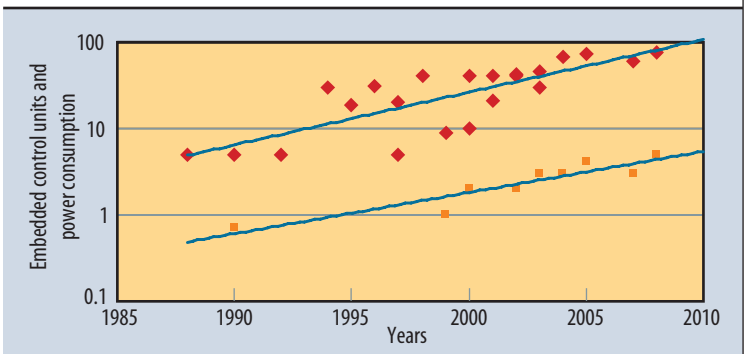


Figure 3. Automotive embedded systems (upper line: number of control units) and overall electrical energy consumption (lower line: consumption in kW).

Programming environments

A common denominator across all embedded-software domains is the use of programming languages that allow direct access to interfaces, memory, and so forth. More than 80 percent of all companies are using C and to some degree C++. More than 40 percent are using assembler for lower-level interfaces. Java is increasingly used for GUI and application programming. Eclipse-based development tools dominate engineering workbenches due to the many different tools that need to be federated, such as modeling and simulation tools (for example, Matlab/Simulink, Rose, and Tau), testing environments (LabView, CANoe, HIL/SIL, and emulators), product life-cycle management environments (Teamcenter and eASEE), configuration management tools (Subversion and CVS), requirements tools (DOORS and Caliber), and of course compilers, debuggers, and the like. Due to the intensive supplier interaction and collaboration, which is much higher than in traditional IT, tools such as DOORS or Matlab/Simulink have respective market shares of more than 50 percent.

Software practices

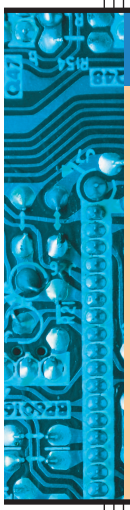
Because of the high-reliability and quality requirements for embedded applications, as well as the often stringent performance requirements, since the 1990s, the embedded world has focused on software practices that yield high quality. Some examples of the practices used for embedded software include the following:

- quality function deployment for requirements prioritization and traceability of quality;
- model-driven design and test;
- mathematical modeling for reliability, power consumption, thermal, and performance analysis;
- formal design and code inspections;
- automated static code analysis for memory, performance, and security;
- broad automatic testing;
- Six Sigma for software development;
- adoption and fairly rapid ascent to CMMI levels 3 and above (in fact the CMMI was created by explicit demand from the embedded-software industries);
- components that are explicitly designed for reuse; and
- selected agile principles such as feature-driven design and daily Scrum sessions for status and quality topics.

The embedded domain tends to be more formal in its development practices than either the IT or desktop software domains. Because many embedded applications are safety-critical by nature (for example, medical, industry automation, automotive, or transport), developers have been forced to systematically introduce and use formal methods that concentrate on quality. Some industries are under extreme quality and schedule pressure, resulting in extensive oversight criteria. For example, in the satellite business, deadlines and quality are obviously not negotiable, and the same holds for domains such as automotive electronic suppliers or the industry automation domain where huge external systems are waiting for the in-time availability of high-quality embedded software controllers.

DEFECT POTENTIALS AND REMOVAL

Quality as a multifactor set of requirements is more complex and important in embedded systems than in application software or information systems. As many embedded devices have immediate impact on the user, often under safety conditions, defect potentials and removal must be closely monitored and improved. The embedded-software domain sometimes stumbles and releases products that are dangerous or fail to work effectively. There have been serious issues with medical instruments, antilock brakes, and home appliances. Failures and poor quality in embedded software can sometimes cause death or serious injury. As a result, some embedded devices such as medical instruments have serious liability issues.



EMBEDDED-SOFTWARE RESOURCES

The following list provides additional sources of information about the topics discussed in this article.

- ***IEEE Software*, special issue on software development for embedded systems, May/June 2009; www.computer.org/portal/site/software**
- **Newsletter and archive on embedded-software engineering and technologies: www.embedded.com**
- **Function Point calculation and benchmarks: International Function Point Users Group (IFPUG); www.IFPUG.org**
- **Benchmarks on a variety of IT and software projects: International Software Benchmarking Standards Group (ISBSG); www.ISBSG.org**

Therefore, top-notch quality control is a mandatory feature of many embedded applications.

Quality plays a pivotal role in efficiency and cost improvement. Quality leads, and productivity follows. For productivity to improve at all, quality must be improved faster and to a higher level than productivity. Attempts to improve productivity without improving quality first are not effective.

Software quality

Finding and fixing bugs is overall the most expensive activity in software development. Studies of software quality indicate a strong correlation between application size and the total number of defects that must be eliminated. A simple rule of thumb can provide an approximate but useful estimate of potential defects for embedded applications: Raise the size of the application in function points to the 1.22 power, and the result will yield the approximate total number of defects that must be found and eliminated. This rule of thumb includes all major sources of defects: requirements, design, coding, and documentation, as well as "bad fixes" or secondary defects that are accidentally included in attempts to repair other defects. Note that this rule of thumb is for embedded software. Other forms of software such as information systems or commercial packages would need different exponents. The predicted number of defects will be somewhat higher than will cause failures.

A similar rule of thumb can predict the approximate number of test cases. Since embedded software tends to perform more kinds of testing and have more test cases than other forms, the rule of thumb is: Raise the size of the application in function points to the 1.24 power to determine the approximate number of test cases that are likely to be used.

To illustrate these two rules of thumb, we consider an application of 1,000 function points coded in C. Given that the ratio of C statements to function points is approximately 100 to 1 (understanding that the exact value differs depending on the specific C dialect), 1,000 function points is equivalent to 100,000 C statements or 150 KLOC. A thousand function points raised to the 1.22 power equals 4,570 potential defects in all categories, or 30 defects per KLOC. Of these total defects, about 20 percent would be high-severity defects. While this may seem like a large number of defects, a significant percentage—in the range of 95 to 99 percent, depending on the organization's maturity level—will be found prior to delivery.

This amount of code requires some 5,350 test cases based on industrial benchmarks. Your own number could be much higher for two reasons. First, test-driven development and criticality-based testing multiplies this number by at least a factor of 10. Second, embedded-software organizations typically have a test-case redundancy of around 30 to 50 percent due to collecting, but never revisiting, old test cases.

Verification and validation

Code verification and validation stages for embedded software encompass peer reviews, static code analysis, subroutine and algorithmic testing, unit testing, component testing, functional testing including hardware-in-the-loop (HIL) and software-in-the-loop (SIL), integration testing, system testing, and qualification and acceptance testing. Each of these V&V steps will typically yield 30 percent of defect-removal effectiveness. This provides 97 percent overall code defect-removal effectiveness if all 10 steps are performed adequately. In our experience, maturity level 3 organizations in the embedded-software domain demonstrate 98 percent removal effectiveness, typically with high emphasis on reviews and test methodology.[3,6,10] Maturity level 5 organizations in embedded-software systems, such as Boeing or Motorola, achieve 99 percent and higher removal effectiveness.

By contrast, the average defect-removal effectiveness at release of application software and information systems is only about 85 percent.[6] You can imagine what that means for software such as Windows given its millions of LOC. Overall, the embedded domain does a somewhat better job in terms of defect-removal levels than other forms of software.

Compared to other forms of software such as desktop applications, and business information systems, the embedded-software domain tends to use more sophisticated software quality assurance, better quality measurements, formal inspections, and more test stages. Table 1 shows a typical pattern of defect prevention, verification, and validation activities used for embedded-software development based on the SPR database.[4,6] Note that the list is based on history data and does not prescribe certain techniques or effort. It includes all common forms

Table 1. Typical pattern of embedded software defect prevention and removal.

Activities	Assignment scope in function points	Production rate in function points per month	Defect-removal effectiveness (percent)	Bad fix injection (percent)
Manual reviews				
Design inspections	1,000	160	85.0	4.0
Code inspections	200	60	85.0	4.0
Quality function deployment	1,000	200	82.0	3.0
Test plan inspection	750	125	80.0	5.0
Test script inspection	300	175	78.0	4.0
Document review	3,500	1,000	77.0	2.5
Pair programming review	2,500	200	75.0	5.0
Bug repair inspection	300	90	70.0	3.0
Quality assurance review	2,500	750	45.0	7.0
Manual testing				
Subroutine testing	5	100	50.0	2.0
Component testing	1,250	150	40.0	3.0
System testing	2,000	200	40.0	7.0
New function testing	125	110	35.0	5.0
Regression testing	150	150	30.0	7.0
Unit testing	50	90	25.0	4.0
Automated testing				
Static code analysis	15,000	10,000	87.0	1.0
System test	500	200	40.0	8.0
Regression test	500	175	37.0	7.0
Unit test	500	250	35.0	4.0
New function test	500	200	35.0	5.0
Qualification				
Usability testing	5,000	2,000	65.0	4.0
Preseries testing	15,000	500	45.0	5.0
Acceptance testing	5,000	3,000	40.0	7.0

of defect prevention and removal used for embedded applications. Any specific embedded application might use only a subset of the activities shown.

Interpreting this list requires introducing and explaining specific terms. Assignment scope is the amount of work assigned to one engineer. Assignment scopes are used to determine staffing levels. Production rate is the amount of work that one person can perform in a given period such as an hour or a month.

Late defect correction in embedded software costs much more than for other software types due to the close hardware interaction and the demands from certification bodies such as the US Food and Drug Administration for more intense regression testing. The clear focus is thus on detecting defects as early as possible in the phase where they are inserted. Sixty percent of a system's defects come from 20 percent of its components (modules, classes, or units).[3] However, the distribution varies based on environment characteristics such as processes used and quality goals. Ten percent of all code accounts for 90 percent of outage time, whereas 20 percent of all defects need 60 to 80 percent of correction effort.[10] Most of the avoidable rework comes from a small number of software defects, where avoidable rework is defined as work done to mitigate the effects of errors or to improve system performance.

Adapt your V&V strategies to this Pareto imbalance and focus expensive reviews and manual test on critical areas, while performing basic regression testing bottom up with unit test, test-driven development, and automatic integration test routines based on operational profiles. Note that test cases and test scripts often contain defects,

are not enough focused on what matters, and are highly redundant. It is therefore necessary to review test strategy and test cases as much as the design and implementation. Model-driven approaches help in making code more consistent but do not replace reviews because they would just delay early errors from the modeling down to the validation where the model is finally checked versus reality. Don't collect test cases like stamps, but rather use test-driven development and implementation-independent test strategies to improve your test quality.

PROJECTS AND PRODUCTIVITY

A client developing embedded software asked us to evaluate his company's engineering efficiency. He had organically grown his embedded automotive software business and gradually introduced people, processes, and tools. The company's software technology was unquestionably far above average. Its embedded-software development had grown from support, to mechanical and hardware engineering, to being the major value driver and cost factor.

Cost and efficiency increasingly are the focus of embedded-software development.

But how to improve efficiency was unclear to management. To start, let's look at planning.

Planning

A good predictor is the Putnam formula, which states that project effort is proportional to size to the power of 3 divided by duration to the power of 4 and again divided by productivity to the power of 3. For embedded software, this effort multiplies by a factor of 2 to 4. High-dependability software multiplies the base effort by a factor of 3 to 10. For maintenance projects, the base effort multiplies by a factor of 2 to 5. The minimum project duration in months is 2.5 times effort in person-years to the power of 1/3.[2,10]

As an example, take an automotive embedded controller on which five persons would typically work for two years. The minimum duration could be 6 to 8 months with intensive front-loading and priority-driven design with parallel verification and validation. This can be achieved by a short requirements and design phase of 2 to 3 months and intensive parallel verification and validation.

Allocating engineers to several projects in parallel reduces productivity. Experience shows that productivity is reduced in steps depending on the amount of context switching due to the different assignments—for example, interruptions by phone calls from the second project while doing design in the first. As a rule, consider a 30 percent overall productivity decrease if an engineer is working on several independent assignments.

Business case validity seems to be optimal with 5 to 10 percent delays. Zero is overly expensive, but more than 10 percent decreases customer satisfaction. A common tradeoff is permitting few customer requirements changes (or by sales and marketing) that make projects a bit late but add tangible value for the customer.

Cost and efficiency increasingly are the focus of embedded-software development. New entrants from low-cost countries have shown that high reliability is not designed bottom up with expensive components and methods but can be achieved with low-cost redundancy. Cost pressure in some embedded industries has caused double-digit reductions for the same software year over year. Today, embedded-software engineering needs to deliver on time with excellent quality at a continuously decreasing cost per unit.

Requirements and test

The two major cost drivers in embedded-software development are requirements and test.

Requirements. Requirements are the single major driver. We often develop the wrong things due to not reviewing and analyzing requirements, missing and vague requirements, or confusing needs and requirements. Forty percent of all software defects in embedded systems result from insufficient requirements and analysis activities. The typical effort allocated to requirements engineering is 3 to 7 percent of total project cost. It is 5 to 10 percent for all requirements-management-related activities during the life cycle, which includes change management during the project. Doubling this effort has the potential to reduce life-cycle cost by 20 to 40 percent, thus yielding a direct ROI of 4, not considering benefits such as better reuse. The cost reduction mostly stems from reduced error rates during elicitation and analysis, earlier defect removal during specification and requirements verification, and improved consistency across work products.

Test. Testing after code completion consumes 30 to 40 percent of embedded-development resources and—depending on the project life cycle (sequential or incremental)—requires a lead time of 15 to 50 percent of total project duration. The minimum lead time is achieved when test strongly overlaps development, such as in incremental development with a stable build that is continuously regression tested and integration of software artifacts is split into groups of check-ins tested in "stage areas," which then are connected to further stage areas, developing an integration "tree." In this case, there is only the system test at the end, contributing to lead time on the critical project path. On the other hand, testing practiced in a classic waterfall approach—which still is widely seen in embedded-software development due to the many interfaces and external dependencies— significantly increases lead time

due to repetitive component integration overheads and heavy extra effort from late changes.

How to reduce test effort? First, by detecting defects close to the phase where they are created: Reviews, models, and code analysis will help. Second, by removing redundancy: Across projects, at least 30 percent of all test cases are redundant, as embedded-software engineers have the tendency to add test cases "to be on the safe side" and don't control them by means of coverage or related effectiveness criteria. Review your test strategies, use coverage tools, and apply orthogonal test-case arrays to reduce test redundancies.

Schedule pressures

Beware of the negative impact of time pressure. We often find companies that compress schedules to a point that makes engineers skip necessary V&V activities, only to find later they need extra time and incur costs for repair. V&V activities are necessary and need to be planned up front. Organizations insisting on requirements reviews on each project and requirement change have quality, schedule, and efficiency advantages of more than 20 percent simply because they start with the right requirements and have fewer changes afterward.

A surprising finding is that software projects that achieve 95 percent or higher in total defect-removal effectiveness tend to have shorter development schedules and lower development costs than similar projects of the same size that achieve only 85 percent (or lower) defect-removal efficiency.[3,4,6] This occurs because testing is the main portion of development where schedule delays mount up and costs begin to exceed budgets. Applications that enter testing with an excessive volume of defects cannot exit the testing phase because they don't work. By contrast, similar projects using formal inspections, static analysis, and other methods in addition to testing will have shorter test schedules because a majority of defects have already been eliminated.

Productivity. Improving productivity can reduce the duration of a task or project (given that all other factors are known) by up to 25 percent. This implies excellent team building and teamwork, strong planning and monitoring on the critical path, strong method and tool support, high parallelism, and early defect removal.

Such mechanisms are not sustainable, however, and demand strong follow-up. They bear the risk of high stress levels and attrition of team members if pressure is maintained for too long. New defects are inserted with changes and corrections, specifically those late in a project that are done under pressure. Corrections create some 5 to 30 percent new defects depending on time pressure and underlying tool support. Sometimes, secondary and tertiary bad fixes occur. One of the authors was an expert witness in a lawsuit where four consecutive attempts to repair a bug failed, and each attempt added at least one new bug. In particular, late defect removal while being on the project's critical path causes many new defects because quality assurance activities are reduced and engineers are stressed. This must be considered when planning testing, validation, or maintenance activities.

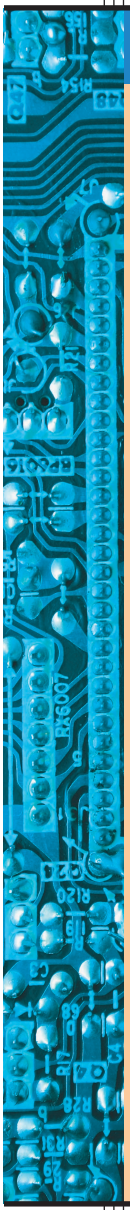
Outsourcing. Be aware that outsourcing and distributed development of embedded software is difficult and is often canceled before it delivers any real savings. Dividing a business process across the world with shared responsibilities costs extra money and requires rework effort. Our own experience shows that with two locations, you should budget 20 to 30 percent overhead, and for three to four locations, the overhead is some 30 to 40 percent.[12] This overhead is due to additional interfaces, management, team effort, collaboration support, quality control, reviews, and so on. Reported cost reduction from global software engineering is much less than the commonly touted 50 to 70 percent savings if only labor costs are compared—as the media often do. In our experience, outsourced embedded-software engineering projects report a 10 to 15 percent cost reduction after a two- to three-year learning curve. Initially, outsourcing demands up to 20 percent additional effort. For India, communication and automotive supplier companies report that the effective savings after a three-year period is 15 to 20 percent.

Applications that enter testing with an excessive volume of defects cannot exit the testing phase because they don't work.

CHALLENGES AND SOLUTIONS

By 2015, massively parallel computing systems will evolve to the individual device level, with systems on chip being produced on wafer scale. Sensors and processors will include mechanical or biological systems, optical devices, wireless connectivity, and voice recognition. With highly networked systems, energy distribution will change from a centralized architecture to many small distributed units, such as solar cells, wind turbines, and others. Users will receive online-demand information from smart meters, and utilities will boost their capacity management from the many embedded yet powerful batteries in e-vehicles. With sensors arriving at the biological level, implants will ease diagnosis and facilitate seamless medical support and, where needed, immediate yet remote assistance.

Many scenarios can be derived from these major trends.[2,14,15] To provide value for embedded-software engineers, we distill from these trends four design principles with concrete guidance for improving embedded-software engineering.



EMBEDDED SOFTWARE

Christof Ebert, *Vector*
Jürgen Salecker, *Siemens Corporate Technology*

Embedded systems have overwhelming penetration around the world. Innovations are increasingly triggered by software embedded in automotive, transportation, industrial-automation, medical-equipment, communication, energy, and many other kinds of systems. They use about 98 percent of all the microprocessors produced worldwide.

Embedded software differs significantly from desktop and enterprise software, mostly in environmental conditions—particularly real-time and performance expectations, safety needs, low production costs (because of high volumes), heterogeneous environments, changing platforms, long life spans, and maintenance difficulties. They communicate with their environment (other embedded devices, enterprise systems, or mechanical or biological systems) in many ways—via sensors, actors, specialized human interfaces, and general-purpose communication links.

IEEE Software, another IEEE Computer Society magazine, is dedicating its May/June issue to embedded software. The issue shows how environmental conditions impact embedded-software engineering. Emphasis is on the state of the practice and current development techniques and trends. Above all, it provides many hands-on industrial experiences from which all of us can learn, independent of the domain we're engaged in and the type of software we use in our day-to-day engineering work.

In "Trends in Embedded-Software Engineering," Peter Liggesmeyer and Mario Trapp summarize current advances in embedded-software engineering such as model-driven development (MDD). You might argue that such techniques are already used in IT and application software development. True and not so true, as the Point-Counterpoint discussion by Les Hatton and Michiel van Genuchten highlights with interesting insights. One of the most relevant trends in embedded-software engineering is the move toward more abstraction and thus being able to better manage complexity throughout the life cycle. In "UML-Based Model-Driven Development for HSDPA Design," Jesús Martinez, Pedro Merino, Alberto Salmerón, and Francisco Malpartida show how to introduce MDD to embedded-software development. The application and development of domain-specific languages is well suited for the embedded domain as well.

Complexity reigns in embedded software, as elsewhere. But power and performance restrictions demand we control complexity. One possible solution is the application of multicore microcontrollers, which are now entering the embedded domain. "Embedded Multiprocessor Systems-on-Chip Programming" by Jean-Yves Mignolet and Roel Wuyts will help professionals avoid common traps when entering this domain. Because of its embedded character with respect to critical environments and often life-threatening risks, embedded software faces high-quality requirements. Systematic, thorough, and completely traceable verification and validation are key to good quality. In "Formal Modeling and Verification of Safety-Critical Software," Junbeom Yoo, Eunkyong Jee, and Sungdeok (Steve) Cha show how such techniques are applied to safety-critical software in a nuclear-reactor protection system. In line with this article, the Software Technology department in the *Software* May/June issue introduces practical aspects of static code analysis as well as current practices and tools for embedded software.

In "A Case for Taking a More Agile Approach in Embedded-Systems Development," Michael Smith, James Miller, Lily Huang, and Albert Tran show how to keep good processes sufficiently lean to allow for flexibility and efficiency. "Experiences in Improving Flight Software Development Processes," by Ronald Kirk Kandt, emphasizes the impact that a higher level of software development maturity has had on a software engineering project at the Jet Propulsion Laboratory. High quality and performance requirements combined with fierce cost pressures demand strong, continuously improving engineering and management processes.

Enjoy reading the *IEEE Software* May/June issue.

***Christof Ebert** is a partner and managing director at Vector. Contact him at christof.ebert@vector-consulting.de.*

***Jürgen Salecker** is competence field manager for embedded systems at Siemens Corporate Technology. Contact him at juergen.salecker@siemens.com.*

Complexity management

Embedded software continues to grow by 10 to 30 percent per year depending on the application domain.[2] Increasing complexity means extra defects and cost.

Measure the complexity of your embedded software and control it. Use systematic processes both internally and with your suppliers to mitigate risks and allow for fast recuperation in case of insufficient performance. Use CMMI or SPICE and demand similar use from your suppliers. Both frameworks are designed for mixed software/hardware systems and help build a state-of-the-practice engineering environment.

Apply model-driven design and test to trace design decisions and foster fast change cycles. Round-trip engineering is not yet available on a systems scale but should be your target. Apply it on the highest feasible aggregation level, such as in embedded-controller design. Reduce code size and refactor your software periodically. Reduce variants and use platform- or product-line engineering to avoid any kind of ad hoc variation. Establish penalty schemes if code is duplicated or modified without an agreed-upon business case. Refactor your test strategies and test cases to focus on critical defect removal.

Value orientation

Embedded software is always under cost pressure. Apply the RACE principle: reduce accidents and control essence.[3] Accidents imply unnecessary overhead such as gold plating, rework due to late defect removal, or too many requirements changes. Essence is what customers pay for.

Grow systems engineering skills in your software teams. Take a systems perspective when deciding on software architecture, interfaces, or future evolution.

Embedded software cannot be evaluated in its own limited software scope. Establish reviews for your requirements from the perspective of both product management and testing. Specify requirements so that they show their respective marginal value and at the same time are testable. Ensure consistency and traceability of decisions. Install one change-control agent who decides on all incoming change requests and on all software to be released.

Safety and security

Risks from malfunctions of embedded software are much higher than those of application software. Security rapidly grows in relevance as embedded software communicates autonomously with other computing systems.

Embedded-software engineers must know and use a richer combination of defect prevention and removal activities than other software domains. Safety, security, or performance cannot be designed or tested in isolation. They influence each other as well as all functional and interface requirements.

Focus on architecture and performance requirements before diving into algorithms and functions. Use techniques such as disciplined traceability of requirements and changes, model-driven development, code analysis, test-driven development, and automatic testing to achieve a high-maturity process culture.

Energy efficiency

While early embedded software typically drove energy consumption beyond what was necessary for plain functionality, today it must control energy consumption. Energy efficiency is currently the major embedded trend. Software contributes to energy savings.

Use performance tools to identify idle processes and switch them off. Put hardware systems in sleep mode when performance is not demanded. Adjust processor speed to demands. Cache data to avoid bus and interface load. Deploy passive sensors that trigger wake-up functions. Make your embedded software independent of standby power supply by using capacitors and flash memories.

We have introduced systematic engineering concepts and productivity improvements in many companies around the world in embedded-software domains such as telecommunications, the automotive industry, and transportation. Within a few years, they managed to improve early detection by a factor of 2 to 5, reduce maintenance costs by more than 40 percent, and shorten development schedules by 15 to 50 percent.[3,6] This would have been impossible without measuring and assessing quality, cost, and schedule.

Will embedded software grow in revenues, and therefore jobs, as we have come to expect? Clearly, the worsening economic environment will impact all companies throughout the technology supply chain. But the overall structure of the embedded industry and the positioning of individual companies are vastly different from the last recession in 2001.

First, the demand will not cease given the widespread needs related to transportation, communication, automation, safety, and security. This means that risks can be distributed over more segments, including recession-resistant medical and industrial markets. Second, recent growth rates have been lower than before 2001, while cash positions have increased. Third, the embedded-systems market has become more global with new markets and R&D centers emerging in developing and faster-growing regions. Fourth, and not least, good embedded-software engineering skills remain scarce on a global scope because such skills are difficult to acquire and are immediately mission-critical. Certainly, we will see some impact from the current economic environment, but much less than in other industries. Compound average annual growth rates are 7 percent for embedded hardware, 10 percent for embedded software, and 15 percent for related engineering services.[2,15]

Embedded software has tremendous impact. It determines value and risks in many of today's products, independent of domain and usage. If done well, it is invisible and makes people spend money for the product. The equation is simple: More embedded software in a product increases sales and market share. But one wrong bit out of billions can bring a system down and can cause physical damage.

Embedded-software engineering must cope with the close relationship of value and risk. Competition drives quality. The goal is not perfection but professionalism. In the 1970s, Japanese industry, which had practically no international market share, started shipping technical equipment that showed value in terms of cost and quality. They stunned the industries in the US and Europe—and changed behaviors toward a strong focus on value and quality.

Of course, demands and markets have changed today, but the underlying rationale is the same: We must continuously look for better ways to develop embedded software. Smart embedded-software engineers will spend their careers engaged in this search for the better, and they will find continuous challenges, solutions, and lots of personal rewards. C

References

1. W.H. Maisel et al., "Recalls and Safety Alerts Involving Pacemakers and Implantable Cardioverter-Defibrillator Generators," *JAMA*, vol. 286, 15 Aug. 2001, pp. 793-799; http://jama.ama-assn.org/cgi/content/abstract/286/7/793.
2. BITKOM, "Studie zur Bedeutung des Sektors Embedded-Systeme in Deutschland" ["Embedded Systems Study in Germany"], 2008, (in German); www.bitkom.org/files/documents/Studie_BITKOM_Embedded-Systeme_11_11_2008.pdf.
3. C. Ebert and R. Dumke, *Software Measurement*, Springer, 2007.
4. C. Jones, *Estimating Software Costs*, McGraw Hill, 2007.
5. A.S. Berger, *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*, CMP Books, 2001.
6. C. Jones, *Applied Software Measurement*, McGraw Hill, 2008.
7. ISBSG, "The Benchmark," release 10, ISBSG; www.isbsg.org.
8. S. McConnell, *Professional Software Development,* Addison-Wesley, 2003.
9. C. Jones, *Software Quality—Analysis and Guidelines for Success*, International Thomson Computer Press, 1997.
10. F. Shull et al, "What We Have Learned about Fighting Defects," *Proc. 8th Int'l Symp. Software Metrics*, IEEE CS Press, 2002, pp. 249-258.
11. A. Endres and D. Rombach, *A Handbook of Software and Systems Engineering—Empirical Observation, Laws and Theories*, Addison-Wesley, 2003.
12. C. Ebert, "Global Software Engineering," *ReadyNotes*, IEEE CS Press, 2006; www.computer.org/portal/pages/ieeecs/ReadyNotes/ebert_abstract.html.
13. C. Jones, *Conflict and Litigation between Software Clients and Developers*, Software Productivity Research Inc., 2008.
14. ITEA, *Technology Roadmap for Software-Intensive Systems*, 2nd ed., 2004; www.itea2.org/itea_roadmap_2.
15. OECD, "Information Technology Outlook 2006"; www.oecd.org/sti/ito, 2006. Annually updated.

Christof Ebert *is a partner and managing director at Vector. His research interests include product management and productivity improvement. He received a PhD in electrical engineering from the University of Stuttgart, Germany. He is an IEEE senior member and distinguished visitor. Contact him at christof.ebert@vector-consulting.de.*

Capers Jones*, chief scientist emeritus of Software Productivity Research LLC, is also the president of Capers Jones & Associates LLC. His research interests focus on software productivity and measurement. He received a degree in English from the University of Florida. He is a member of the IEEE Computer Society and a lifetime member of the International Function Point Users Group. Contact him at cjonesIII@cs.com.*