Dependable software

Bertrand Meyer, ETH Zurich

Cite as follows: Bertrand Meyer, *Dependable Software*, to appear in *Dependable Systems: Software, Computing, Networks*, eds. Jürg Kohlas, Bertrand Meyer, André Schiper, Lecture Notes in Computer Science, Springer-Verlag, 2006.

ABSTRACT

Achieving software reliability takes many complementary techniques, directed at the process or at the products. This survey summarizes some of the most fruitful ideas.

1 OVERVIEW

Everyone who uses software or relies on devices or processes that use software — in other words, everyone — has a natural interest in guarantees that programs will perform properly. The following pages provide a review of techniques to improve software quality.

There are many subcultures of software quality research, often seemingly sealed off from each other; mentioning process-based approaches such as CMMI to programming language technologists, or tests to people working on proofs, can be as incongruous as bringing up Balanchine among baseball fans. This survey disregards such established cultural fences and instead attempts to include as many as possible of the relevant areas, on the assumption that producing good software is hard enough that "every little bit counts" **[60]**. As a result we will encounter techniques of very diverse kinds.

A note of warning to the reader seeking objectivity: I have not shied away from including references — easy to spot — to my own work, with the expectation (if a justification is needed) that it makes the result more lively than a cold inspection limited to other people's products and publications.

2 SCOPE AND TERMINOLOGY

The first task is to define some of the fundamental terms. Even the first word of this article's title, determined by the Hasler Foundation's "Dependable Information and Communication Systems" project, requires clarification.

Reliability and dependability

In the software engineering literature the more familiar term is not "dependable" but "reliable", as in "software reliability". A check through general-purpose and technical dictionaries confirms that the two have similar definitions and are usually translated identically into foreign languages.

There does exist a definition of dependability **[1]** from the eponymous IFIP Working Group 10.4 **[39]** that treats reliability as only one among dependability attributes, along with availability, safety, confidentiality, integrity and maintainability. While possibly applicable to a computing system as a whole, this classification does not seem right for their software part, as some attributes such as availability are not properties of the software per se, others such as confidentiality are included in reliability (through one of its components, security), and the remaining ones such as maintainability are of dubious meaning for software, being better covered by other quality factors such as extendibility and reusability **[57]**.

As a consequence of these observations the present survey interprets dependability as meaning the same thing, for software, as reliability.

Defining reliability

The term “software reliability” itself lacks a universally accepted definition. One could argue for taking it to cover all “external quality factors” such as ease of use, efficiency and extendibility, and even “internal quality factors” such as modularity. (The distinction, detailed in **[57]**, is that external factors are the properties, immediate or long-term, that affect companies and people purchasing and using the software, whereas internal factors are perceptible only to software developers although in the end they determine the attainment of external factors.)

It is reasonable to retain a more restricted view in which reliability only covers three external factors: *correctness*, *robustness* and *security*. This doesn’t imply that others are irrelevant; for example even the most correct, robust and secure system can hardly be considered dependable if in practice it takes ages to react to inputs, an *efficiency* problem. The same goes for *ease of use*: many software disasters on record happened with systems that implemented the right functions but made them available through error-prone user interfaces. The reasons for limiting ourselves to the three factors listed are, first, that including all others would turn this discussion into a survey of essentially the whole of software engineering (see **[33]**); second, that the techniques to achieve these three factors, although already very diverse, have a certain kindred spirit, not shared by those for enhancing efficiency (like performance optimization techniques), ease of use (like ergonomic design) and other external and internal factors.

Correctness, robustness, security

For the three factors retained, we may rely on the following definitions:

- Correctness is a system's ability to perform according to its specification in cases of use within that specification.
- Robustness is a system's ability to prevent damage in cases of erroneous use outside of its specification.
- Security is a system's ability to prevent damage in cases of hostile use outside of its specification.

They correspond to levels of increasing departure from the specification. The specification of any realistic system makes assumptions, explicit or implicit, about the conditions of its use: a C compiler's specification doesn't define a generated program if the input is payroll data, any more than a payroll program defines a pay check if the input is a C program; and a building's access control software specification cannot define what happens if the building has burned. By nature, the requirements defined by robustness and security are different from those of correctness: outside of the specification, we can no longer talk of "performing" according to that specification, but only seek the more modest goal of "preventing damage"; note that this implies the ability to *detect* attempts at erroneous or hostile use.

Security deserves a special mention as in recent years it has assumed a highly visible place in software concerns. This is a phenomenon to be both lamented, as it signals the end of a golden age of software development when we could concentrate on devising the best possible functionality without too much concern about the world's nastiness, and at the same time taken to advantage, since it has finally brought home to corporations the seriousness of software quality issues, a result that decades of hectoring by advocates of modern software engineering practices had failed to achieve. One of the most visible signs of this phenomenon is Bill Gates's edict famously halting all development in February of 2001 in favor of code reviews for hunting down security flaws. Many of these flaws, such as the most obnoxious, buffer overflow, are simply the result of poor software engineering practices. Even if focusing on security means looking at the symptom rather than the cause, fixing security implies taking a coherent look at software tools and techniques and requires, in the end, ensuring reliability as a whole.

Product and process

Any comprehensive discussion of software issues must consider two complementary aspects: *product* and *process*.

The products are the software elements whose reliability we are trying to assess; the process includes the mechanisms and procedures whereby people and their organizations build these products.

The products of software

The products themselves are diverse. In the end the most important one, for which we may assess correctness, robustness and security, is code. But even that simple term covers several kinds of product: source code as programmers see it, machine code as the computer executes it, and any intermediate versions as exist on modern platforms, such as the bytecode of virtual machines.

Beyond code, we should consider many other products, which in their own ways are all "software": requirements, specifications, design diagrams and other design documents, test data — but also test plans —, user documentation, teaching aids…

To realize why it is important in the search for quality to pay attention to products other than code, it suffices to consider the results of numerous studies, some already decades old **[10]**, showing the steep progression of the cost of correcting an error the later it is identified in the lifecycle.

Deficiencies

In trying to ascertain the reliability of a software product or process we must often — like a detective or a fire prevention engineer — adopt a negative mindset and look for sources of *violation* of reliability properties. The accepted terminology here distinguishes three levels:

- A *failure* is a malfunction of the software. Note that this term does not directly apply to products other than executable code.
- A *fault* is a departure of the software product from the properties it should have satisfied. A failure always comes from a fault, although not necessarily a fault in the code: it could be in the specification, in the documentation, or in a non-software product such as the hardware on which the system runs.
- An *error* is a wrong human decision made during the construction of the system. "Wrong" is a subjective term, but for this discussion it's clear what it means: a decision is wrong if it can lead to a fault (which can in turn cause failures).

In a discussion limited to *software* reliability, all faults and hence all failures result from errors, since software is an intellectual product not subject to the slings and arrows of the physical world.

The more familiar term for "error" is *bug*. The upper crust of the software engineering literature shuns it for its animist connotations. "Error" has the benefit of admitting that our mistakes don't creep into our software: we insert them ourselves. In practice, as may be expected, everyone says "bug".

Verification and validation

Even with subjectivity removed from the definition of "error", definitions for the other two levels above remains relative: what constitutes a "malfunction" (for the definition of failures) or a "departure" from desirable properties (for faults) can only be assessed with respect to some description of the expected characteristics.

While such reference descriptions exist for some categories of software product — an element of code is relative to a design, the design is relative to a specification, the specification is relative to an analysis of the requirements — the chain always stops somewhere; for example one cannot in the end certify that the requirements have no fault, as this would mean assessing them against some higher-level description, and would only push the problem further to assessing the value of the description itself. Turtles all the way up.

Even in the absence of another reference (another turtle) against which to assess a particular product, we can often obtain some evaluation of its quality by performing *internal* checks. For example:

- A program that does not initialize one of its variables along a particular path is suspicious, independently of any of its properties vis-à-vis the fulfillment of its specification.
- A poorly written user manual may not explicitly violate the prescriptions of another project document, but is problematic all the same.

This observation leads to distinguishing two complementary kinds of reliability assessment, *verification* and *validation*, often combined in the abbreviation "V&V":

- Verification is *internal* assessment of the consistency of the product, considered just by itself. The last two examples illustrated properties that are subject to verification: for code; for documentation. Type checking is another example.
- Validation is *relative* assessment of a product vis-à-vis another that defines some of the properties that it should satisfy: code against design, design against specification, specification against requirements, documentation against standards, observed practices against company rules, delivery dates against project milestones, observed defect rates against defined goals, test suites against coverage metrics.

A popular version of this distinction **[10]** is that verification is about ascertaining that the product is "doing things right" and validation that it is "doing the right thing". It only applies to code, however, since a specification, a project plan or a test plan do not "do" anything.

3 CLASSIFYING APPROACHES

One of the reasons for the diversity of approaches to software quality is the multiplicity of problems they address. The following table shows a list of criteria, essentially orthogonal, for classifying them.

Criteria for classifying approaches to software reliability

A priori (*build*)		A posteriori (*assess and correct*)
Process		Product
Manual		Tool-supported
Technology-neutral		Technology-specific
Product- and phase-neutral	*vs*	Product- or phase-specific
Static (uses software text)		Dynamic (requires execution)
Informal		Mathematical
Complete (guarantee)		Partial (some progress)
Free		Commercial

The first distinction is cultural almost as much as it is technical. With *a priori* techniques the emphasis is methodological: telling development teams to apply certain rules to produce a better product. With *a posteriori* techniques, the goal is to examine a proposed software product or process element for possible deficiencies, with the aim of correcting them. While it is natural to state that the two are complementary rather than contradictory — a defense often used by proponents of “a posteriori” approaches such as testing when criticized for accepting software technology as it is rather than helping to improve it — they correspond to different views of the software world, one hopeful of prevention and the other willing to settle down for cure.

The second distinction corresponds to the two dimensions of software engineering cited above: are we working on the *products*, or on the *processes* leading to them?

Some approaches are of a methodological nature and just require applying some practices; we may call them *manual*, in contrast with techniques that are *tool-supported* and hence at least partially automated.

An idea can be applicable regardless of technology choices; for example process-based techniques such as CMMI, discussed below, explicitly stay away from prescribing specific technologies. At the other extreme, certain techniques may be applicable only if you accept a certain programming language, specification method, tool or other technology choice. We may talk of *technology-neutral* and *technology-specific* approaches; this is more a spectrum of possibilities than a black-and-white distinction, since many approaches assume a certain class of technologies — such as object-oriented development — encompassing many variants.

Some techniques apply to a specific product or phase of the lifecycle: specification (a specification language), implementation (a static analyzer of code)… They are *product-specific*, or *phase-specific*. Others, such as configuration management tools, apply to many or all product kinds; they are *product-neutral*. "Product" is used here to denote one of the types of outcome of the software construction process.

For techniques directed at program quality, an important division exists between *dynamic* approaches such as testing, which rely on executing the program, and purely *static* ones, such as static analysis and program proofs, which only need to analyze the program text. Here too some nuances exist: a simulation technique requires execution and hence can be classified as dynamic even though the execution doesn't use the normal run-time environment; model-checking is classified as static even though in some respect it is close to testing.

Some methods are based on *mathematical* techniques; this is obviously the case with program proofs and formal specification in general. Many are more *informal*.

A technique intended to assess quality properties can give you a *complete* guarantee that they are satisfied, or — more commonly — some *partial* reassurance to this effect.

The final distinction is economic: between techniques in the public domain — usable for free, in the ordinary sense of the term — and commercial ones.

4 PROCESS-BASED APPROACHES

We start with the least technical approaches, emphasizing management procedures and organizational techniques.

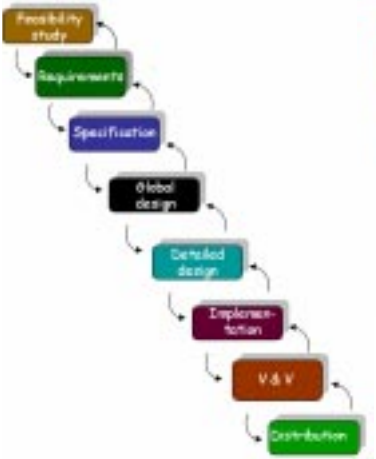
Lifecycle models

One of the defining acts of software engineering was the recognition of the separate activities involved, in the form of "lifecycle models" that prescribe a certain order of tasks (see the figure on the adjacent page). The initial model is the so-called "waterfall" **[11]**, still used as a reference for discussions of the software process although no longer recommended for literal application. Variants include:

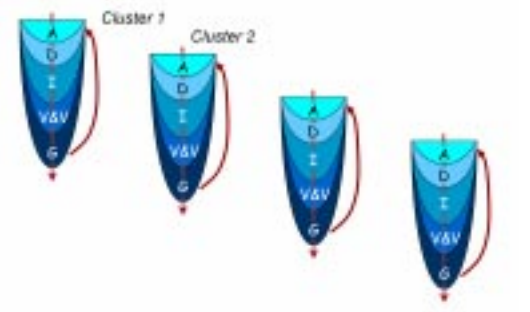
- The "V model" which retains the sequential approach of the waterfall but divides the process into two parts, the branches of the "V"; activities along the first branch are for development, those in the second branch are for verification and validation, each applied to the results of one of the steps along the first branch.
- The "Spiral model" **[11]** which focuses on reducing risk in project management, in particular the risk caused by the all-or-nothing attitude of the Waterfall approach. The spiral model suggests isolating subsets of the system's functionality that are small enough to be implemented quickly, and when they have been implemented taking advantage of the experience to proceed to other parts of the system. The idea is connected with the notion of rapid prototyping.
- The "Rational Unified Process", distinguishing four phases, inception, elaboration, construction and transition, with a spiral-like iterative style of development and a set of recommended "best practices" such as configuration management.
- The "Cluster model" **[51]** **[57]**, emphasizing a different form of incrementality — building a system by layers, from the most fundamental to the most user-oriented — and a *seamless* process treating successive activities, from analysis to design, implementation and maintenance, as a continuum. This model also introduces, as part of the individual lifecycle of every cluster, a *generalization* step to prepare for future reuse of some of the developed elements.

The figure shows pictorial representations of some of these models.

Waterfall



Cluster



V-shaped

Spiral (from **[11]**)

Lifecycle models, illustrated

Whatever their effect on how people actually develop software, the contribution of lifecycle models has been a classification and definition of the activities involved in software development, even when these activities are not executed as *phases* in the precise order mandated by, for example, the waterfall model. Software quality benefits in particular from:

- A distinction between *requirements*, the recording of user requirements, and *specification*, their translation into a systematic form suitable for software development, where rigor and precision are essential.
- Recognition of the importance of Verification and Validation tasks.
- Recognition of post-delivery activities such as maintenance, although they still do not occupy a visible enough place. Many software troubles result from evolutions posterior to the initial release.
- In the Cluster model, the presence, for each cluster, of the generalization task to prepare for reuse.
- Also in the Cluster model, the use of a seamless and reversible approach which unifies the methods, tools, techniques and notations that help throughout the software process, rather than exaggerate them. (The textbook counter-example here is the use of UML for analysis and design **[56]**.)
- The growing emphasis on *incrementality* in the development process, even if this concept is understood differently in, for example, the spiral, cluster and RUP models.

Organizational standards

Another process-related set of developments has had a major effect, largely beneficial, on some segments of the industry. In the early 1990s the US Department of Defense, concerned with the need to assess its suppliers' software capabilities and to establish consistent standards, entrusted the Software Engineering Institute with the task of developing a "Capability Maturity Model", whose current incarnation, CMMI **[74]** (the I is for Integration) provides a collection of standards applicable to various disciplines, rather than a single model for software. Largely independently, the International Standard Organization has produced a set of software-oriented variants of its 9000-series quality standards, which share a number of properties with CMMI. The present discussion is based on CMMI.

Beyond its original target community, CMM and CMMI have been the catalyst for one of the major phenomena of the IT industry starting in the mid-nineties: the development of offshore software production, especially in India **[63]**. CMMI qualification provides suppliers of outsourcing development services with quality standards and the associated possibility of independent certification, without which customers would not be have known how to trust distant, initially unknown contractors.

CMMI is (in the earlier classification) product-neutral, phase-neutral and technology-neutral. In its application to software it is intended only to determine how well an organization controls its development process by defining and documenting it, recording and assessing how it is applied in practice, and working to improve it. It doesn't prescribe what the process should be, only how much you are on top of it. You could presumably be developing in PL/I on IBM 370 and get CMMI qualification.

CMMI assesses both the *capability* level of individual "process areas" in (such as software) in an organization, and the *maturity* of an organization as a whole. It distinguishes five levels of increasing maturity:

- *Performed*: projects happen and results get produced, but there is little control and no reproducibility; the process is essentially reactive.
- *Managed*: processes are clearly defined for individual projects, but not for the organization as a whole. They remain largely reactive.
- *Defined*: proactive process defined for the organization.
- *Quantitatively managed*: the control mechanisms do not limit themselves to qualitative techniques, but add well-defined numerical measurements.
- *Optimizing*: the mechanisms for controlling processes are sufficiently well established that the focus can shift on improving the organization and its processes.

Through their emphasis on the process and its repeatability, CMMI and ISO standards help improve the quality of software development. One may expect such improvements of the process to have a positive effect on the resulting products as well; but they are only part of the solution. After a software error — one module of the software was expecting measures in the metric system, another was providing them in English units — was identified as the cause of the failure of the NASA Mars Orbiter Vehicle mission **[82]**, an engineer from the project noted that the organization was heavily into ISO and other process standards. Process models and process-focused practices are not a substitute for using the best technological solutions. Tailored versions of CMMI that would not shy away from integrating specific technologies such as object technology could be extremely useful. In the meantime, the technology-neutral requirements of CMMI can be applied by organizations to get a better hold on their software processes.

Extreme programming

The Extreme Programming movement **[6]** is a reaction against precisely the kinds of lifecycle models and process-oriented approaches just reviewed. XP (as it is also called) emphasizes instead the primacy of code. Some of the principal ideas include:

- Short release cycles to get frequent feedback.
- Pair programming (two people at a keyboard and terminal).
- Test-driven development.
- A general distrust of specification and design: *testing* is the preferred guide of development.
- Emphasis on programmers' welfare.

Some of these practices are clearly beneficial to quality but were developed prior to XP, in particular short release cycles (Microsoft's "daily build" as described in 1995 by Cusumano and Shelby **[19]**, see also **[54]**) and the use of frequent testing as part of development (see e.g. "quality first" **[55]**). Those really specific to XP are of limited interest (while sometimes a good practice, pair programming cannot be imposed indiscriminately, both because it doesn't work for some people and because those who find it useful may not find it useful all the time) or, in the case of tests viewed as a *replacement* for specifications, downright detrimental. See **[75]** and **[64]** for critiques of the approach.

Code inspections

A long-established quality practice is the inspection, also known as *review*: a session designed to examine a certain software element with the aim of finding flaws. The most common form is *code* inspection, but the process can be applied to any kind of software engineering product. Rules include:

- Small meeting: at most 8 people or so, including the developer of the element under review.
- The elements under review and any supporting documents must be circulated in advance; the participants should have read them and identified possible criticisms before the meeting. The allotted time should be bounded, for example 2 or 3 hours.

- The meeting must have a moderator to guide discussions and a secretary to record results.
- The moderator should not be the developer's manager. The intent is to evaluate products, not people.
- The sole goal is to identify deficiencies and confirm that they are indeed deficiencies; correction is not part of the process and should not be attempted during the meeting.

Code inspections can help avoid errors, but to assess their usefulness one must compare the costs with those of running automated *tools* that can catch some of the same problems without human intervention; static analyzers, discussed below, are an example.

Some companies have institutionalized the rule that no developer may check in code (integrate it into the repository for a current or future product) without approval by one other developer, a limited form of code inspection that has a clearly beneficial effect by forcing the original developer to convince at least one other team member of the suitability of the contribution.

Open-source processes

A generalization of the idea of code inspection is the frequent assertion, by members of the open-source community, that the open-source process dramatically improves quality by enabling many people to take a critical look at the software text; some have gone so far as to state that "*given enough eyes, all bugs are shallow*" **[73]**.

As with many of the other techniques reviewed, we may see in this idea a beneficial contribution, but not a panacea. John Viega gives **[78]** the example of a widely used security program in which "*in the past two years, several very subtle buffer overflow problems have been found… Almost all had been in the code for years, even though it had been examined many times by both hackers and security auditors One tool was able to identify one of the problems as potentially exploitable, but researchers examined the code thoroughly and came to the conclusion that there was no way the problem could be exploited*." (The last observation is anecdotal evidence for the above observation that tools such as static analyzers are potentially superior to human analysis.)

While is no evidence that open-source software as a whole is better (or worse) than commercial software, and no absolute rule should be expected if only because of the wide variety of products and processes on both sides, it is clear that more eyes potentially see *more* bugs.

Requirements engineering

In areas such as embedded systems, many serious software failures have been traced **[45]** to inadequate requirements rather than to deficiencies introduced in later phases. Systematic techniques for requirements analysis are available **[76]** **[40]** to improve this critical task of collecting customer wishes and translating them into a form that can serve as a basis for a software project.

Design patterns

A process-related advance that has had a strong beneficial effect on software development is the emergence of design patterns **[32]**. A pattern is an architectural scheme that has been recognized as fruitful through frequent use in applications, and for which a precise description exists according to a standard format. Patterns provide a common vocabulary to developers, hence simplifying design discussions, and enable them to benefit from the collective wisdom of their predecessors.

A (minority) view of patterns **[62]** **[65]** understands them as a first step towards the technique discussed next, reusable components. Patterns, in this interpretation, suffer from the limitation that each developer must manually insert the corresponding solutions into the architecture of every applicable system. If instead it is possible to turn the pattern into a reusable component, developers can directly reuse the corresponding solution through an API (Abstract Program Interface). The observation here is that *it is better to reuse than to redo*. Investigations **[65]** suggest that with the help of appropriate programming language constructs up to two thirds of common design patterns can be thus **componentized**.

Trusted components

Quality improvement techniques, whether they emphasize the process or the product, are only as good as their actual application by programmers. The magnitude of the necessary education effort is enough to temper any hope of major short-term improvements, especially given that many programmers have not had the benefit of a formal computer science education to start with.

Another practical impediment to continued quality improvement comes from market forces. The short-term commercial interest of a company is generally to release software that is "good enough" **[83]**: software that has barely passed the threshold under which the market would reject it because of bad quality; not excellent software. The extra time and expense to go from the first to the second stage may mean, for the company, losing the market to a less scrupulous competitor, and possibly going out of business. For the industry as a whole, software quality has indeed improved regularly over time but tends to peak below the optimum.

An approach that can overcome these obstacles is increased reliance on reusable components, providing pre-built solutions to problems that arise in many different applications, either regardless of the technical domain (*general-purpose* component libraries) or in particular fields (*specialized* libraries). Components have already changed the nature of software development by providing conveniently packaged implementations, accessible through abstract interfaces, of common aspects such as graphical user interfaces, database manipulation, basic numerical algorithms, fundamental data structures and others, thereby elevating the level at which programmers write their applications. When the components themselves are of good quality, such reuse has highly beneficial effects since developers can direct their efforts to the quality of the application-specific part of their programs.

Examining more closely the relationship of components to quality actually highlights two separate effects: it is comforting to know that the quality of a system will benefit from the quality of its components; but we must note that reuse magnifies the bad as well as the good: *imperfections* can be even more damaging in components than in "one-of-a-kind" developments, since they affect every application that relies on a component.

The notion of *trusted component* **[58]** **[61]** follows from this analysis that one of the most pressing and promising tasks for improving software quality is the industrial production of reusable components equipped with a guarantee of quality. Producing such trusted components may involve most of the techniques discussed elsewhere in this article. For some of the more difficult ones, such as program proving, application to components may be the best way to justify the cost and effort and recoup the investment thanks to the scaling effect of component reuse: once a component has reached the level of quality at which it can really be trusted, it will benefit every application that relies on it.

5 TOOLS AND ENVIRONMENTS

Transitioning now to product-oriented solutions, we examine some of the progress in tools available to software developers — to the extent that it is relevant for software quality.

Configuration management

Configuration management is a both practice (for the software developer) and a service (from the supporting tools), so it could in principle be classified under “process” as well as under “product”. It belongs more properly to the latter category since it’s tools that make configuration management realistic; applied as a pure organizational practice without good tool support, it quickly becomes tedious and ceases being applied.

Configuration management may be defined as the systematic collecting and registering of project elements, including in particular the ability to:

- Register a new version of any project element.
- Retrieve *any* previously registered version of any project element.
- Register dependencies, both between project elements and between registered *versions* of project elements (e.g. *A* relies on *B*, and version 10 of *A* requires version 7, 8 or 9 of *B*).
- Construct composite products from their constituents — for example, build an executable version of a program from its modules — or reconstruct earlier versions, in accordance with registered dependencies.

A significant number of software disasters on record followed from configuration management errors, typically due to reintroducing an obsolete version of a module when compiling a new release of a program, or using an obsolete version of some data file. Excuses no longer exist for such errors, as acceptable configuration management tools, both commercial and open-source, are widely available. These tools, while still far from what one could hope for, have made configuration management one of the most important practices of modern software development.

Source code is not the only beneficiary of configuration management. Any product that evolves, has dependencies on other elements and may need restoring to an earlier state should be considered for inclusion in the configuration management repository. Besides code this may include project plans, specification and design documents, user manuals, training documents such as PowerPoint slides, test data files.

Metrics and models

If we believe Lord Kelvin's (approximate) maxim that all serious study is quantitative, then software and software development should be susceptible to measurement, tempered of course by Einstein's equally famous quote that not everything measurable is worth measuring. A few software properties, process or product, are at the same time measurable, worth measuring and relevant to software reliability.

On the process side, **cost** in its various dimensions is a prime concern. While it is important to record costs, if only for CMMI-style traceability, what most project managers want at a particular time is a *model* to estimate the cost of a future project or of the remainder of a current project. Such models do exist and can be useful, at least if the development process is stable and the project is comparable to previous ones: then by estimating a number of project parameters and relying on historical data for comparison one can predict costs — essentially, person-months — within reasonable average accuracy. A well-known cost model, for which free and commercial tools are available, is COCOMO II **[12]**.

During the development of a system, **faults** will be reported. In principle they shouldn't be comparable to the faults of a material product, since software is an intellectual product and doesn't erode, wear out or collapse under attack from the weather. In practice, however, statistical analysis shows that faults in large projects can follow patterns that resemble those of hardware systems and are susceptible to similar statistical prediction techniques. That such patterns can exist is in fact consistent with intuition: if the tests on the last five builds of a product under development have each uncovered one hundred new bugs each, it is unlikely that the next iteration will have zero bugs, or a thousand. *Software reliability engineering* **[69][46]** elaborates on these ideas to develop models for assessing and predicting failures, faults and errors. As with cost models, a requirement for meaningful predictions is the ability to rely on historical data for calibration. Reliability models are not widely known, but could help software projects understand, predict and manage anomalies better.

More generally, numerous **metrics** have been proposed to provide quantitative assessments of software properties. Measures of complexity, for example, include: "source lines of code" (SLOC), the most primitive, but useful all the same; "function points" **[25]**, which count the number of elementary mechanisms implemented by the software; measures of the complexity of the control graph, such as "cyclomatic complexity" **[48][49]**; and measures specifically adapted to object-oriented software **[35][59]**. The EiffelStudio environment **[30]** makes it possible to compute many metrics applied to a project under development, including measures regarding the use of contracts (section

8), and to compare them with values on record. While not necessarily meaningful in isolation, such measures elements are a useful control tool for the manager; they are in line with the CMMI's insistence that an organization can only reach the higher levels of process maturity (4 and 5) by moving from the qualitative to the quantitative, and should be part of the data collected for such an effort.

Static analyzers

Static analyzers are another important category of tools, increasingly integrated in development environments, whose purpose is to examine the software text for deficiencies. They lie somewhere between type checkers (themselves integrated in compilers) and full program provers, and will be studied below (page 26) after the discussion of proofs.

Integrated development environments

Beyond individual tools the evolution of software development has led to the widespread of integrated tool suites known as IDEs for Integrated (originally: Interactive) Development Environments. Among the best known are Microsoft's Visual Studio **[66]** and IBM's Eclipse **[27]**; EiffelStudio **[30]** is another example. These environments, equipped with increasingly sophisticated graphical user interfaces, provide under a single roof a whole battery of mechanisms to write software (editors), manage its evolution (configuration management), compile it (compilers, interpreters, optimizers), examine it effectively (browsers), run it and elucidate the sources of faults (debuggers, testers), analyze it for possible inconsistencies and errors (static analysis), generate code from design and analysis diagrams or the other way around (diagramming, "Computer-Aided Software Engineering" or CASE, reverse engineering), change architecture in a safe way through tool-controlled transformations (refactoring), perform measurements as noted above (metric tools), and other tasks.

This is one of the most active areas in software engineering; programmers, for whom IDEs are the basic daily tools, are directly interested in their quality, so that open-source projects such as Eclipse and EiffelStudio benefit from active community participation. The effect of these advanced frameworks on software reliability, while diffuse, is undeniable, as their increasing cleverness supports quality in several ways: finding bugs through static and dynamic techniques; avoiding new bugs through mechanisms such as refactoring; generating some of the code without manual intervention; and, more generally, providing a level of comfort that frees programmers from distractions and lets them apply their best skills to the hardest issues of software construction.

6 PROGRAMMING LANGUAGES

The evolution of programming languages plays its part in the search for more reliable software. High-level languages contribute both positively, by providing higher levels of expression through advanced constructs freeing the programmer (in the same spirit as modern IDEs) from mundane, repetitive or irrelevant tasks, and negatively, by ruling out certain potentially unsafe constructs and, as a result, eradicate entire classes of bugs at the source.

The realization that programming language constructs could exert a major influence on software quality both through what they offer and what they forbid dates back to *structured* programming [22] [20] which, in the early seventies, led to rejecting the **goto** as a control structure in favor of more expressive constructs — sequence, conditional, loop, recursion. The next major step was *object-oriented* programming, introducing a full new set of abstractions, in particular the notion of class, providing decomposition based on object types rather than individual operations, and techniques of inheritance and genericity.

In both cases the benefit comes largely from being able to reason *less operationally* about software. A software text represents many possible executions, so many in fact that it is hard to understand the program — and hence to get it right — by thinking in terms of what happens at execution [22]. Both structured and object-oriented techniques make it possible to limit such operational thinking and instead understand the abstract properties of future run-time behaviors by applying the usual rules of logical reasoning.

In drawing the list of programming languages' most important contributions to quality, we must indeed put at the top all the mechanisms that have to do with **structure**. With ever larger programs addressing ever more ambitious goals, the production and maintenance of reliable software requires safe and powerful modular decomposition facilities. Particularly noteworthy are:

- As pointed out, the **class** mechanism, which provides a general basis for stable modules with a clear role in the overall architecture.
- Techniques for **information hiding**, which protect modules against details of other modules, and permit independent evolution of the various parts of a system.
- **Inheritance**, allowing the classification and systematic organization of classes into structured collections, especially with *multiple* inheritance.
- **Genericity**, allowing the construction of type-parameterized modules.

Another benefit of modern languages is **static typing** which requires programmers to declare types for all the variables and other entities in their programs, then takes advantage of this information to detect possible inconsistencies in their use and reject programs, at compilation time, until all types fit. Static typing is particularly interesting in object-oriented languages since inheritance supports a flexible type system in which types can be compatible even if they are not identical, as long as one describes a specialization of the other.

Another key advance is **garbage collection**, which frees programmers from having to worry about the details of memory management and removes an entire class of errors — such as attempts to access a previously freed memory cell — which can otherwise be particularly hard to detect and to correct, in particular because the resulting failures are often intermittent rather than deterministic. Strictly speaking, garbage collection is a property of the language implementation, but it's the language definition that makes it possible, as with modern object-oriented languages, or not, as in languages such as C that permit arbitrary pointer arithmetic and type conversions.

Exception handling, as present in modern programming languages, helps improve software robustness by allowing developers to include recovery code for run-time faults that would otherwise be fatal, such as arithmetic overflow or running out of memory.

A mechanism that is equally far-reaching in its abstraction benefits is the "closure", "delegate" or "**agent**" **[62]**. Such constructs wrap operations in objects that can then be passed around anonymously across modules of a system, making it possible to treat routines as first-class values. They drastically simplify certain kinds of software such as numerical applications, GUI programming and other event-driven (or "publish-subscribe") schemes.

The application of programming language techniques to improving software quality is limited by the continued reliance of significant parts of the software industry on older languages. In particular:

- Operating systems and low-level system-related tend to be written in C, which retains its attractions for such applications in spite of widely known deficiencies, such as the possibility of buffer overflow.
- The embedded and mission-critical community sometimes prefers to use low-level languages, including assembly, for fear of the risks potentially introduced by compilers and other supporting tools.

The "Verifying Compiler Grand Challenge" **[38]** **[77]** is an attempt to support the development of tools that — even with such programming languages — will guarantee, during the process of compiling and thanks to techniques described in the following sections, the reliability of the programs they process.

7 STATIC VERIFICATION TECHNIQUES

Static techniques work solely from the analysis of the software text: unlike dynamic techniques such as tests they do not require any execution to verify software or report errors.

Proofs

Perhaps the principal difference between mathematics and engineering is that only mathematics allows providing absolute guarantees. Given the proper axioms, I can assert with total confidence that two plus two equals four. But if I want to drive to Berne the best assurance I can get that my car will not break down is a probability. I know it's higher than if I just drive it to the suburbs, and lower than if my goal were Prague, Alma-Ata, Peking or Bombay; I can make it higher by buying a new, better car; but it will never be one. Even with the highest attention to quality and maintenance, physical products will occasionally fail.

Under appropriate assumptions, a program is like a mathematical proposition rather than a material device: any general property of the program — stating that *all* executions of the program will achieve a certain goal, or that *at least one* possible execution will — is either true or false, and whether it is true or not is entirely determined by the text of the program, at least if we assume correct functioning of the hardware and of other software elements needed to carry out program execution (compiler, run-time system, operating system). Another way of expressing this observation is that a programming language is similar to a mathematical theory, in which certain propositions are true and others false, as determined by the axioms and inference rules.

In principle, then, it should be possible to prove or disprove properties of programs, in particular correctness, robustness and security properties, using the same rigorous techniques as in the proofs of any mathematical theorem. This assumes overcoming a number of technical difficulties:

- Programming languages are generally *not* defined as mathematical theories but through natural-language documents possessing a varying degree of precision. To make formal reasoning possible requires describing them in mathematical form; this is known as providing a *mathematical semantics* (or "formal semantics") to a programming language and is a huge task, especially when it comes to modeling advanced mechanisms such as exception handling and concurrency, as well as the details of computer arithmetic since the computer's view of integers and reals strays from their standard mathematical properties.

- The theorems to be proved involve specific properties of programs, such as the value of a certain variable not exceeding a certain threshold at a certain state of the execution. Any proof process requires the ability to express such properties; this means extending the programming language with boolean-valued expressions, called *assertions*. Common languages other than Eiffel do not include an assertion mechanism; this means that programmers will have to resort to special extensions such as JML for Java **[43]** (see also Spec#, an extension of the C# language **[5]**) and annotate programs with the appropriate assertions. Some tools such as Daikon help in this process by extracting tentative assertions from the program itself **[31]**.

- In practice the software's actual operation depends, as noted, on those of a supporting hardware and software environment; proofs of the software must be complemented by guarantees about that environment.

- Not all properties lend themselves to easy enunciation. In particular, "non-functional" properties such as performance (response time, bandwidth, memory occupation) are hard to model.

- More generally, a proof is only as useful as the program properties being proven. What is being proved is not the perfection of the program in any absolute sense, nor even its quality, but only that it satisfies the assertions stated. It is never possible to know that *all* properties of interest have been included. This is not just a theoretical problem: security attacks often take advantage of auxiliary aspects of the program's behavior, which its design and verification did not take into account.

- Even if the language, the context and the properties of interest are fully specified semantically and the properties relevant, the proof process remains a challenge. It cannot in any case be performed manually, since even the proof of a few properties of a moderately sized programs quickly reaches into the thousands of proof steps. Fully automated proofs are, on the other hand, generally not possible. Despite considerable advances in computer-assisted proof technology (for programs as well as other applications) significant proofs still require considerable user interaction and expert knowledge.

Of course the effort may well be worthwhile, especially in two cases: life-critical systems in transportation and defense to which, indeed, much proof work has been directed; and reusable components, for which the effort is justified — as explained in the discussion of Trusted Components above — by the scaling-up effect of reuse.

Here are some of the basic ideas about how proofs work. A typical program element to prove would be, in Eiffel notation

```
decrement
        -- Decrease counter by one.
    require
        counter > 0
    do
        counter := counter – 1
    ensure
        counter = old counter – 1
        counter >= 0
    end
```

This has a program body, the **do** clause, and two assertions, a "precondition" introduced by **require** and a "postcondition" introduced by **ensure** and consisting of two subclauses implicitly connected by an **and**. Assertions are essentially boolean expressions of the language with the possibility, in a postcondition, of using the **old** notation to refer to values on entry: here the first subclause of the postcondition states that the value of *counter* will have been decreased by one after execution of the **do** clause.

Program proofs deal with such annotated programs, also called *contracted* programs (see section 8 below). The annotations remind us that proofs and other software quality assurance technique can never give us absolute guarantees of quality: we can never say that a program is "correct", only assess it — whether through rigorous techniques like proofs or using more partial ones such as those reviewed next — *relatively to* explicitly stated properties, expressed here through assertions integrated in the program text.

From a programmer's viewpoint the above extract is simply the text of a routine to be executed, with some extra annotations, the precondition and postcondition, expressing properties to be satisfied before and after. But for proof purposes this text is a *theorem*, asserting that whenever the body (the **do** clause with its assignment instruction) is executed with the precondition satisfied it will terminate in such a way that the postcondition is satisfied.

This theorem appears to hold trivially but — even before addressing the concern noted above that computer integers are not quite the same as mathematical integers — proving it requires the proper mathematical framework. The basic rule of *axiomatic semantics* (or "Hoare semantics" **[37]**) covering such cases is the assignment axiom, which for any variable *x* and expression *e* states that the following holds

require *Q* (*e*) **do** *x* := *e* **ensure** *Q* (*x*)

where *Q* (*x*) is an assertion which may depend on *x*; then *Q* (*e*) is the same assertion with every mention of *x* replaced by *e*, except for occurrences of **old** *x* which must be replaced by *x*.

This very general axiom captures the properties of assignment (in the absence of side effect in the evaluation of *e*); its remarkable feature is that it is applicable even if the source expression *e* contains occurrences of the target variable *x*, as in the example (where *x* is *counter*).

We may indeed apply the axiom to prove the example's correctness. Let *Q1* (*x*) be *x* = **old** *x* – 1, corresponding to the first subclause of the postcondition, and *Q2* (*x*) be *x* >= 0. Applying the rule to *Q1* (*counter*), we replace *counter* by *counter* + 1 and **old** *counter* by *counter*; this gives *counter* – 1 = *counter* – 1, which trivially holds. Applying now the same transformations to *Q2* (*counter*), we get *counter* – 1 >= 0, which is equivalent to the precondition *counter* > 0. This proves the correctness of our little assertion-equipped example.

From there the theory moves to more complex constructions. An inference rule states that if you have proved

require *P* **do** *Instruction_1* **ensure** *Q*

and

require *Q* **do** *Instruction_2* **ensure** *R*

(note the postcondition of the first part matching the precondition of the second part) you are entitled to deduce

require *P* **do** *Instruction_1* ; *Instruction_2* **ensure** *Rt*

and so on for more instructions. A rule in the same style enables you to deduce properties of **if** *c* **then** *I1* **else** *I2* **end** from properties of *I1* and *I2*. More advanced is the case of loops: to prove the properties of

```
from
    Initialization
until
    Exit
loop
    Body
end
```

you need, in this general approach, to introduce a new assertion called the **loop invariant** and an integer expression called the **loop variant**. The invariant is a weakened form of the desired postcondition, which serves as approximation of the final goal; for example if the goal is to compute the maximum of a set of values, the invariant will be "*Result* is the maximum of the values processed so far". The advantage of the invariant is that it is possible both to:

- Ensure the invariant through initialization (the **from** clause in the above notation); in the example the invariant will be trivially true if we start with just one value and set *Result* to that value.

- Preserve the invariant through one iteration of the loop body (the **loop** clause); in the example it suffices to extend the set of processed values by one element *v* and execute **if** *v* > *Result* **then** *Result* := *v* **end**.

If indeed a loop possesses such an invariant and its execution terminates, then on exit the invariant will still hold (since it was ensured by the initialization and preserved by all the loop iterations), together with the *Exit* condition. The combination of these two assertions gives the postcondition of the loop. Seen the other way around, if we started from a desired postcondition and weakened it to get an invariant, we will obtain a correct program. In the example, if the exit condition states that we have processed all values of interest, combining this property with the invariant "*Result* is the maximum of the values processed so far" tells us that *Result* is the maximum of all values.

Such reasoning is only interesting if the loop execution actually terminates; this is where the loop variant comes in. It is an integer expression which must have a non-negative value after the *Initialization* and decrease, while remaining non-negative, whenever the *Body* is executed with the *Exit* condition not satisfied. The existence of such an expression is enough to guarantee termination since a non-negative integer value cannot decrease forever. In the example a variant is *N* — *i* where *N* is the total number of values being considered for the maximum (the proof assumes a finite set) and *i* the number of values processed.

Axioms and inference rules similarly exist for other constructs of programming languages, becoming, as noted, more intricate as one moves on to more advanced mechanisms.

For concurrent, reactive and real-time systems, boolean assertions of the kind illustrated above may not be sufficient; it is often convenient to rely on properties of **temporal logic** [47], which given a set of successive observations of a program's execution, can express, for a boolean property *Q*:

- **forever** *Q*: from now on, *Q* will always hold.
- **eventually** *Q*: at some point in the future (where "future" includes now), *Q* will hold.
- *P* **until** *Q*: *Q* will hold at some point in the future, and until then *P* will hold.

Regardless of the kind of programs and properties being targeted, there are two approaches to producing program proofs. The **analytic** method takes programs as they exist, then after equipping them with assertions, either manually or with some automated aid as noted above, attempts the proof. The **constructive** method [24] [2] [68] integrates the proof process in the software construction process, often using successive *refinements* to go from specification to implementation through a sequence of transformations, each proved to preserve correctness, and integrating more practical constraints at every step.

Proof technology has had some notable successes, including in industrial systems (and in hardware design), but until recently has remained beyond the reach of most software projects.

Static analysis

If hoping for a proof covering all the correctness, reliability and security properties of potential interest is often too ambitious, the problem becomes more approachable if we settle for a subset of these properties — a subset that may be very partial but very interesting. For example being able to determine that no buffer overflow can ever arise in a certain program — in other words, to provide a firm guarantee, through analysis of the program text, that every index used at run time to access an item in an array or a character in a string will be within the defined bounds — is of great practical value since this rules out a whole class of security attacks.

Static analysis is the tool-supported analysis of software texts for the purpose of assessing specific quality properties. Being "static", it requires no execution and hence can in principle be applied to software products other than code. Proofs are a special case, the most far-reaching, but other static analysis techniques are available.

At the other extreme, a well-established form of elementary static analysis is *type checking*, which benefits programs written in a statically typed programming language. Type checking, usually performed by the compiler rather than by a separate tool, ascertains the type consistency of assignments, routine calls and expressions, and rejects any program that contains a type incompatibility.

More generally, techniques usually characterized as static analysis lie somewhere between such basic compiler checks and full program proofs. Violations that can typically be detected by static analysis include:

- Variables that, on some control paths, would be accessed before being initialized (in languages such as C that do not guarantee initialization).
- Improper array and string access (buffer overflow).
- Memory properties: attempt to access a freed location, double freeing, memory leak…
- Pointer management (again in low-level languages such as C): attempts to follow void or otherwise invalid pointers.
- Concurrency control: deadlocks, data races.
- Miscellaneous: certain cases of arithmetic overflow or underflow, changes to supposedly constant strings…

Static analysis tools such as PREfix **[72]** have been regularly applied for several years to new versions of the Windows code base and have avoided many potential errors.

One of the issues of static analysis is the occurrence of *false alarms*: inconsistency reports that, on inspection, do not reveal any actual error. This was the weak point of older static analyzers, such as the widely known *Lint* tool which complements the type checking of C compilers: for a large program they can easily swamp their users under thousand of messages, most of them spurious, but requiring a manual walkthrough to sort out the good from the bad. (In the search for errors, of course, the “good” is what otherwise would be considered the bad: evidence of wrongdoing.) Progress in static analysis has been successful in considerably reducing the occurrence of false alarms.

The popularity of static analysis is growing; the current trend is to extend the reach of static analysis tools ever further towards program proofs. Two examples are:

- Techniques of *abstract interpretation* **[18]** with the supporting ASTRÉE tool **[9]**, which has been used to prove the absence of run-time errors in the primary flight control software, written in C, for the Airbus A340 fly-by-wire system.

- ESC-Java **[21]** and, more recently, the Boogie analyzer **[4]** make program proving less obtrusive by incrementally extending the kind of diagnostics with which programmers are familiar, for example type errors, to more advanced checks such as the impossibility to guarantee that an invariant is preserved.

Model checking

The **model checking** approach to verification **[36] [17] [3]** is static, like proofs and static analysis, but provides a natural link to the dynamic techniques (testing) studied below. The inherent limitation of tests is that they can never be exhaustive; for any significant system — in fact, even for toy examples — the number of possible cases skyrockets into the combinatorial stratosphere, where the orders of magnitude invite lyrical comparisons with the number of particles in the universe.

The useful measure is the number of possible *states* of a program. The notion of state was implicit in the earlier discussion of assertions. A state is simply a snapshot of the program execution, as could be observed, if we stop that execution, by looking up the contents of the program's memory, or more realistically by using the debugger to examine the values of the program's variables. Indeed it is the combination of all the variables' values that determines the state. With every 64-bit integer variable potentially having 2^{64} values, it is not surprising that the estimates quickly go galactic.

Model checking attempts exhaustive analysis of program states anyway by performing *predicate abstraction*. The idea is to simplify the program by replacing all expressions by boolean expressions (predicates), with only two possible values, so that the size of the state space decreases dramatically; it will still be large, but the power of modern computers, together with smart algorithms, can make its exploration tractable. Then to determine that a desired property holds — for example, a security property such as the absence of buffer overflows, or a timing property such as the absence of deadlock — it suffices to evaluate the corresponding assertion in all of the abstract states and, if a violation of that assertion (or *counter-example*) is found, to check that it also arises in the original program.

For example, predicate abstraction will reduce a conditional instruction **if** $a > b$ **then**... to **if** p **then**..., where p is a boolean. This immediately cuts down the number of cases from 2^{128} to 2. The drawback is that the resulting program is only a caricature of the original; it loses the relation of p to other predicates involving a and b. But it has an interesting property: *if the original violates the assertion, then the abstracted version also does*. So the next task is to look for any such violation in the abstracted version. This may be possible through exhaustive examination of its reduced state space, and if so is *guaranteed* to find any violation in the original program, but even so is not the end of the story, since the reverse proposition does not hold: a counter-example in the abstracted program does not necessarily signal a counter-example in the original. It could result from the artificial merging of several cases, for example if it occurs on a path — impossible in an execution of the original program — obtained by selecting both p and q as true where q is the abstraction of $b > a + 1$. Then examining the state space of the abstracted program will either:

- Not find any violations, in which case it *proves* there was none in the original program.
- Report violations, each of which might be an error in the original or simply a false alarm generated by the abstraction process.

So the remaining task, if counter-examples have been found, is to ascertain whether they arise in the original. This involves defining the path predicate that leads to each counter-example, expressing it in terms of the original program variables (that is to say, removing the predicate abstraction, giving, in the example, $a > b$ **and** $b > a + 1$) and determining if any combination of values for the program variables can satisfy the predicate: if such a combination, or *variable assignment*, exists, then the counter-example is a real one; if not, as in the case given, it is spurious.

This problem of *predicate satisfiability* is computationally hard; finding efficient algorithms is one of the central areas of research in model checking.

The focus on counter-examples gives model checking a practical advantage over traditional proof techniques. Unless a software element was built with verification in mind (through a “constructive method” as defined above), the first attempt to verify it will often fail. With proofs, this failure doesn’t tell us the source of the problem — and could actually signal a limitation of the proof procedure rather than an error in the program. With model checking, you get a counter-example which directly shows what’s wrong.

Model checking has captured considerable attention in recent years, first in hardware design and then in reactive and real-time systems, for which the assertions of interest are often expressed in temporal logic.

8 DESIGN BY CONTRACT

The goal of developing software to support full proofs of correctness properties is, as noted, desirable but still unrealistic for most projects. Even a short brush with program proving methods suggests, however, that more rigor can be highly beneficial to software quality. The techniques of *Design by Contract* go in this direction and deliver part of the corresponding benefits without requiring the full formality of proof-directed development.

The discussion of proofs introduced Eiffel notations such as

- **require** *assertion* -- A routine precondition
- **ensure** *assertion* -- A routine postcondition

associated with individual routines. They are examples of **contract** elements which specify abstract semantic properties of program constructs. Contracts apply in particular to:

- Individual routines: **precondition**, stating the condition under which a routine is applicable; **postcondition**, stating what condition it will guarantee in return when it terminates.
- In object-oriented programming, classes: **class invariant**, stating consistency conditions that must hold whenever an object is in a stable state. For example, the invariant for a "paragraph" class in a text processing system may state that the total length of letters and spaces is equal to the paragraph width. Every routine that can modify an instance of the class may assume the class invariant on entry (in addition to its precondition) and must restore it on exit (in addition to ensuring its postcondition).
- Loops: **invariant** and (integer) **variant** as discussed above.
- Individual instructions: "assert" or "check" constructs.

The discipline of Design by Contract **[53] [57] [67]** gives a central role to these mechanisms in software development. It views the overall process of building a system as defining a multitude of relationships between "client" and "supplier" modules, each specified through a contract in the same manner as relationships between companies in the commercial world.

The benefits of such a method, if carried systematically, extend throughout the lifecycle, supporting the goal of *seamlessness* discussed earlier:

- Contracts can be used to express *requirements* and *specifications* in a precise yet understandable way, preferable to pure "bubbles and arrows" notations, although of course they can be displayed graphically too.

- The method is also a powerful guide to *design* and *implementation*, helping developers to understand better the precise reason and context for every module they produce, and as a consequence to get the module right.

- Contracts serve as a *documentation* mechanism: the "**contract view**" of a class, which discards implementation-dependent elements but retains externally relevant elements and in particular preconditions, postconditions and class invariants, often provides just the right form of documentation for software elements, especially reusable components: precise enough thanks to the contracts; abstract enough thanks to the removal of implementation properties; extracted from the program text, and hence having a better chance of being up to date (at least one major software disaster was traced **[41]** to a software element whose specification had changed, unbeknownst to the developers who reused it); cheap to produce, since this form of documentation can be generated by tools from the source text, rather than written separately; and multi-purpose, since the output can be tuned to any appropriate format such as HTML. Eiffel environments such as EiffelStudio produce such views **[30]**, which serve as the basic form of software documentation.

- Contracts are also useful for *managers* to understand the software at a high level of abstraction, and as a tool to control *maintenance*.

- In object-oriented programming, contracts provide a framework for the proper use of *inheritance*, by allowing developers to specify the semantic framework within which routines may be further refined in descendant classes. This is connected with the preceding comment about management, since a consequence is to allow a manager to check that refinements to an design are consistent with its original intent, which may have been defined by the top designers in the organization and expressed in the form of contracts.

- Most visibly, contracts are a **testing** and **debugging** mechanism. Since an execution that violates an assertion always signals a bug, turning on contract monitoring during development provides a remarkable technique for identifying bugs. This idea is pursued further by some of the tools cited in the discussion of testing below.

Design by Contract mechanisms are integrated in the design of the Eiffel language **[52]** **[28]** and a key part of the practice of the associated method. Dozens of contract extensions have been proposed for other programming languages (as well as UML **[80]**), including many designs such as JML **[43]** for Java and the Spec# extension of C# **[5]**.

9 TESTING

Testing **[70] [8]** is the most widely used form of program verification, and still for many teams essentially the only one. In academic circles testing has long suffered from a famous comment **[23]** that (because of the astronomical number of possible states) “testing can only show the presence of bugs, but never to show their absence”. In retrospect it’s hard to find a rational explanation for why this comment ever detracted anyone from the importance of tests, since it in no way disproves the usefulness of testing: finding bugs is a very important task of software development. All it indicates is that we should understand that finding bugs is indeed the sole purpose of testing, and not delude ourselves that test results directly reflect the level of quality of a product under development.

Components of a test

Successful testing relies on a **test plan**: a strategy, expressed in a document, describing choices for the tasks of the testing process. These tasks include:

- Determining which parts to test.
- Finding the appropriate input values to exercise.
- Determining the expected properties of the results (known as *oracles*). Input values and the associated oracles together make up *test cases*, the collection of which constitutes a *test suite*.
- Instrumenting the software to run the tests (rather than perform its normal operation, or in addition to it); this is known as building a **test harness**, which may involve *test drivers* to solicit specific parts to be tested, and *stubs* to stand for parts of the system that will not be tested but need a placeholder when other parts call them.
- Running the software on the selected inputs.
- Comparing the outputs and behavior to the oracles.
- Recording the test data (test cases, oracles, outputs) for future re-testing of the system, in particular *regression testing*, the task of verifying that previously corrected errors have not reappeared.

In addition there will be a phase of *correction* of the errors uncovered by the test, but in line with the above observations this is not part of testing in the strict sense.

Kinds of test

One may classify tests with respect to their **scope** (this was used in the earlier description of the V model of the lifecycle):

- A *unit test* covers a module of the software.
- *Integration test* covers a complete cluster or subsystem.
- A *system test* covers the complete delivery.
- *User Acceptance Testing* involves the participation of the recipients of the system (in addition to the developers, responsible for the preceding variants) to determine whether they are satisfied with the delivery.
- *Business Confidence Testing* is further testing with the users, in conditions as close as possible to the real operating environment.

An orthogonal classification addresses **what** is being tested:

- *Functional* testing: whether the system fulfills the functions defined in the specification.
- *Performance* testing: its use of resources.
- *Stress* testing: its behavior under extreme conditions, such as heavy user load.

Yet another dimension is **intent**: testing can be *fault-directed* to find deficiencies but also (despite the above warnings), *conformance-directed* to estimate satisfaction of desired properties, or *acceptance testing* for users to decide whether to approve the product. *Regression testing*, as noted, re-runs tests corresponding to previously identified errors; surprisingly to the layman, errors have a knack for surging back into the software, sometimes repeatedly, long after they were thought corrected.

The testing technique, in particular the construction of test suites, can be:

- *Black-box*: based on knowledge of the system's specification only.
- *White-box*: based on knowledge of the code, which makes it possible for example to try to exercise as much of that code as possible.

Observing the state of the art in software testing suggests that four issues are critical: managing the test process; estimating the quality of test suites; devising oracles; and — the toughest — generating test cases automatically.

Managing the testing process

Test management has been made easier through the appearance of **testing frameworks** such as JUnit **[42]** and Gobo Eiffel Test **[7]** which record test harnesses to allow running the tests automatically. This removes a considerable part of the burden of testing and is important for regression testing.

An example of a framework for regression testing of a compiler, incorporating every bug ever found since 1991, is EiffelWeasel **[29]**. Such automated testing require a solid multi-process infrastructure, to ensure for example that if a test run causes a crash the testing process doesn't also crash but records the problem and moves on to the next test.

Estimating test quality

Being able to estimate the quality of a test suite is essential in particular to know when to stop testing. The techniques are different for white-box and black-box testing.

With white-box testing it is possible to define various levels of **coverage**, each assuming the preceding ones: *instruction* coverage, ensuring that through the execution of the selected test cases every instruction is executed at least once; *branch* coverage, where every boolean condition tests at least once to true and once to false; *condition* coverage, where this is also the case for boolean sub-expressions; *path* coverage, for which every path has been taken; *loop* coverage, where each loop body has been executed at least *n* times for set *n*.

Another technique for measuring test suite quality in white-box approaches is **mutation testing** **[79]**. Starting with a program that passes its test suite, this consists of making modifications — similar, if possible, to the kind of errors that programmers would make — to the program, and running the tests again. If a "mutant" program still passes the tests, this indicates (once you have made sure the mutant is not *equivalent* to the original, in other words, the changes are meaningful) that the tests were not sufficient. Mutation testing is an active area of research **[71]**; one of the challenges is to use appropriate mutation operators, to ensure diversity of the mutants.

With black-box testing the previous techniques are not available since they assume access to the source code to set up the test plan. It is possible to define notions of *specification coverage* to estimate whether the tests have exercised the various cases listed in the specification; if contracts are present, this will mean analyzing the various cases listed in the preconditions. *Partition testing* **[81]** is the general name for techniques (black- or white-box) that split the input domain into representative subsets, with the implication that any test suite must cover all the subsets.

Defining oracles

An oracle, allowing interpretation of testing results, provides a decision criterion for accepting or rejecting the result of a test. The preparation of oracles can be as much work as the rest of the test plan. The best solution that can be recommended is to rely on contracts: any functional property of a software system (with the possible exception of some user-interface properties for which human assessment may be required) can be expressed as a routine postcondition or a class invariant.

These assertions can be included in the test harness, but it is of course best, as noted in the discussion of Design by Contract, to make them an integral part of the software to be tested as it is developed; they will then provide the other benefits cited, such as aid to design and built-in documentation, and will facilitate regression testing.

Test case generation

The last of the four critical issues listed, test case generation, is probably the toughest; *automatic* generation in particular. Even though we can't ever get close to exhaustive testing, we want the test process to cover as many cases as possible, and especially to make sure they are representative of the various potential program executions — as can be assessed in white-box testing by coverage measures and mutation, but needs to be sought in any form of testing.

For any realistic program, manually prepared tests will never cover enough cases; in addition, they are tedious to prepare. Hence the work on automatic test case generation, which tries to produce as many representative test cases as possible, typically working from specifications only (black-box). Two tools in this area are Korat for JML **[13]** and AutoTest for Eiffel **[15]** (which draws on the advantage that — contracts being native to Eiffel — existing Eiffel software is typically equipped with large numbers of assertions, so that AutoTest can be run on software *as is*, and indeed has already uncovered a significant number of problems in existing programs and libraries).

Manual tests, which benefit from human insight, remain indispensable. The two kinds are complementary: manual tests are good at depth, automatically generated tests at breadth. In particular, any run that ever uncovered a bug, whether through manual or automatic techniques, should become part of the regression test suite. AutoTest integrates manual tests and regression tests within the automatic test case generation and execution framework **[44]**.

Automatic test case generation needs a strategy for selecting inputs. Contrary to intuition, *random* testing **[34]**, which selects test data randomly from the input domain, can be an effective strategy if tuned to ensure a reasonably even distribution over that domain, a policy known as *adaptive random testing* **[14]** which has so far been applied to integers and other simple values (for which a clear notion of distance exists, so that "even distribution" is immediately meaningful). Recent work **[16]** extends the idea to object-oriented programming by defining a notion of object distance.

10 CONCLUSION

This survey has taken a broad sweep across many techniques that all have something to contribute to the aim of software reliability. While it has stayed away from the gloomy picture of the state of the industry which seems to be de rigueur in discussions of this topic, and is not justified given the considerable amount of quality-enhancing ideas, techniques and tools that are available today and the considerable amount of good work currently in progress, it cannot fail to note as a conclusion that the industry could do much more to take advantage of all these efforts and results.

There is not enough of a reliability culture in the software world; too often, the order of concerns is cost, then deadlines, then quality. It is time to reassess priorities.

Acknowledgments

The material in this chapter derives in part from the slides for an ETH industry course on Testing and Software Quality Assurance prepared with the help of Ilinca Ciupa, Andreas Leitner and Bernd Schoeller. The discussion of CMMI benefited from the work of Peter Kolb in the preparation of another ETH course, “Software Engineering for Outsourced and Offshored Development”. Bernd Schoeller and Ilinca Ciupa provided important comments on the draft.

“Design by Contract” is a trademark of Eiffel Software.

The context for this survey was provided by the Hasler Foundation’s grant for our SCOOP work in the DICS project. We are very grateful for the opportunities that the grant and the project have provided, in particular for the experience gained in the two DICS workshops in 2004 and 2005.

REFERENCES

Note: All URLs listed were active in April 2006.

[1] Algirdas Avizienis, Jean-Claude Laprie and Brian Randell: *Fundamental Concepts of Dependability*, in *Proceedings of Third Information Survivability Report*, October 2000, pages 7-12, available among other places at citeseer.ist.psu.edu/article/avizienis01fundamental.html.

[2] Ralph Back: *A Calculus of Refinements for Program Derivations*, in *Acta Informatica*, vol. 25, 1988, pages 593-624, available at crest.cs.abo.fi/publications/public/1988/ACalculusOfRefinementsForProgramDerivationsA.pdf.

[3] Thomas Ball and Sriram K. Rajamani: *Automatically Validating Temporal Safety Properties of Interfaces*, in *SPIN 2001*, Proceedings of Workshop on Model Checking of Software, Lecture Notes in Computer Science 2057, Springer-Verlag, May 2001, pages 103-122, available at tinyurl.com/qrm9m.

[4] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, Wolfram Schulte: *Verification of object-oriented programs with invariants*, in *Journal of Object Technology*, vol. 3, no. 6, Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs, June 2004, pages 27-56, available at www.jot.fm/issues/issue_2004_06/article2.

[5] Mike Barnett, K. Rustan M. Leino and Wolfram Schulte: *The Spec# Programming System: An Overview*, in *CASSIS 2004: Construction and Analysis of Safe, Secure Interoperable Smart devices*, Lecture Notes in Computer Science 3362, Springer-Verlag, 2004, available at research.microsoft.com/specsharp/papers/krml136.pdf; see also other Spec# papers at research.microsoft.com/specsharp/.

[6] Kent Beck and Cynthia Andres: *Extreme Programming Explained: Embrace Change*. 2nd edition, Addison-Wesley, 2004.

[7] Éric Bezault: Gobo Eiffel Test, online documentation at www.gobosoft.com/eiffel/gobo/getest/index.html.

[8] Robert Binder: Testing *Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.

[9] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux and Xavier Rival: *ASTRÉE: A Static Analyzer for Large Safety-Critical Software*, in *Applied Deductive Verification*, Dagstuhl Seminar 3451, November 2003, available at www.di.ens.fr/~cousot/COUSOTtalks/Dagstuhl-3451-2003.shtml. See also ASTRÉE page at www.astree.ens.fr.

[10] Barry W. Boehm: *Software Engineering Economics*, Prentice Hall, 1981.

[11] Barry W. Boehm: *A Spiral Model of Software Development and Enhancement*, in Computer (IEEE), vol. 21, no. 5, May 1988, pages 61-72.

[12] Barry W. Boehm et al.: *Software Cost Estimation with COCOMO II*, Prentice Hall, 2000.

[13] Chandrasekhar Boyapati, Sarfraz Khurshid and Darko Marinov: *Korat: Automated Testing Based on Java Predicates*, in Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA), Rome, July 22--24, 2002, available at tinyurl.com/qwwd3.

[14] T.Y. Chen, H. Leung and I.K. Mak: *Adaptive random testing*, in *Advances in Science - ASIAN 2004: Higher-Level Decision Making*, 9th Asian Computing Science Conference, ed. Michael J. Maher, Lecture Notes in Computer Science 3321, Springer-Verlag, 2004, available at tinyurl.com/lpxn5.

[15] Ilinca Ciupa and Andreas Leitner: *Automated Testing Based on Design by Contract*, in Proceedings of Net.ObjectsDays 2005, 6th Annual Conference on Object-Oriented and Internet-Based Technologies, Concepts and Applications for a Networked World, 2005, pages 545-557, available at se.ethz.ch/people/ciupa/papers/soqua05.pdf. See also AutoTest page at se.ethz.ch/research/autotest.

[16] Ilinca Ciupa, Andreas Leitner, Manuel Oriol and Bertrand Meyer: *Object Distance and its Application to Adaptive Random testing of Object-Oriented Programs*, submitted for publication, 2006, available at se.ethz.ch/~meyer/publications/testing/object_distance.pdf.

[17] Edmund M. Clarke Jr., Orna Grumberg and Doron A. Peled: *Model Checking*, MIT Press, 1999.

[18] Patrick Cousot: *Verification by Abstract Interpretation*, in *International Symposium on Verification Theory & Practice Honoring Zohar Manna's 64th Birthday*, ed. Nachum Dershowitz, Lecture Notes in Computer Science 2772, Springer-Verlag, 2003, pages 243-268.

[19] Michael Cusumano and Richard Selby: *Microsoft Secrets*, The Free Press, 1995.

[20] Ole-Johan Dahl, Edsger W. Dijkstra and C.A.R. Hoare: *Structured Programming*, Academic Press, 1971.

[21] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe: *Extended Static Checking*, Research Report 159, Compaq Systems Research Center, December 1998, available at ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-159.pdf.

[22] Edsger W. Dijkstra: *Go To Statement Considered Harmful*, in *Communications of the ACM*, Vol. 11, No. 3, March 1968, pages 147-148, available at www.acm.org/classics/oct95/.

[23] Edsger W. Dijkstra: *Notes on Structured Programming*, in **[20]**; original typescript available at www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

[24] Edsger W. Dijkstra: *A Discipline of Programming*, Prentice Hall, 1978.

[25] Brian J. Dreger: *Function Point Analysis*, Prentice Hall, 1989.

[26] Paul Dubois, Mark Howard, Bertrand Meyer, Michael Schweitzer and Emmanuel Stapf: *From Calls to Agents*, in *Journal of Object-Oriented Programming* (JOOP), vol. 12, no. 6, September 1999, available at se.ethz.ch/~meyer/publications/joop/agent.pdf.

[27] Eclipse pages at www.eclipse.org.

[28] ECMA/ISO: *Eiffel: Analysis, Design and Programming Language*, standard ECMA 367, accepted in April 2006 as ISO standard, available at www.ecma-international.org/publications/standards/Ecma-367.htm.

[29] Eiffel open-source development site at eiffelsoftware.origo.ethz.ch/index.php/Main_Page.

[30] Eiffel Software: EiffelStudio documentation, online at eiffel.com.

[31] Michael D. Ernst, J. Cockrell, William G. Griswold and David Notkin: *Dynamically Discovering Likely Program Invariants to Support Program Evolution*, in *IEEE Transactions on Software Engineering*, vol. 27, no. 2, February 2001, pages 1-25, available at pag.csail.mit.edu/~mernst/pubs/invariants-tse2001.pdf.

[32] Erich Gamma, Richard Helms, Ralph Johnson and John Vlissides: *Design Patterns*, Addison-Wesley, 1994.

[33] Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli, *Software Engineering*, 2nd edition, Prentice Hall, 2003.

[34] Richard Hamlet: *Random Testing*, in Encyclopedia of Software Engineering, ed. J. J. Marciniak, 1994, available at tinyurl.com/rcjxg.

[35] Brian Henderson-Sellers: *Object-Oriented Metrics: Measures of Complexity*, Prentice Hall, 1995.

[36] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis and Sergio Yovine: *Symbolic Model Checking for Real-Time Systems*, in *Logic in Computer Science*, Proceedings of 7th Symposium in Logics for Computer Science, Santa Cruz, California, 1992, pages 394-406, available at tinyurl.com/lb5fm.

[37] C.A.R. Hoare: *An axiomatic basis for computer programming*, in *Communications of the ACM*, Vol. 12, no. 10, October 1969, pages 576 - 580, available at tinyurl.com/ory2s.

[38] C.A.R. Hoare and Jayadev Misra: *Verified Software: Theories, Tools, Experiments, Vision of a Grand Challenge Project*, October 2005, foundation paper for the VSTTE conference **[77]**, available at vstte.ethz.ch/pdfs/vstte-hoare-misra.pdf.

[39] IFIP Working Group 10.4 on dependable computing and fault tolerance: home page at www.dependability.org.

[40] Michael Jackson: *Problem Frames: Analysing and Structuring Software Development Problems*, Addison-Wesley, 2001.

[41] Jean-Marc Jézéquel and Bertrand Meyer: *Design by Contract: The Lessons of Ariane*, in *Computer* (IEEE), vol. 30, no. 1, January 1997, pages 129-130, available at archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html.

[42] JUnit pages at SourceForge: junit.sourceforge.net.

[43] Gary T. Leavens and Yoonsik Cheon: *Design by Contract with JML* (Draft), at ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf; see also other JML papers at www.cs.iastate.edu/~leavens/JML/.

[44] Andreas Leitner, Ilinca Ciupa, Bertrand Meyer and Mark Howard: *Reconciling Manual and Automated Testing: The AutoTest Experience*, submitted for publication, 2006.

[45] Nancy G. Leveson: *System Safety in Computer-Controlled Automotive Systems*, SAE Congress, March 2000, available at sunnyday.mit.edu/papers/sae.pdf.

[46] Michael R. Lyu (ed.): Handbook of Software Reliability Engineering, IEEE Computer Society Press and McGraw-Hill, 1995; also available online at www.cse.cuhk.edu.hk/~lyu/book/reliability/.

[47] Zohar Manna and Amir Pnueli: *The temporal logic of reactive and concurrent systems*, Springer-Verlag, 1992.

[48] Thomas J. McCabe: *A Complexity Measure*, in *IEEE Transactions on Software Engineering*, vol. 2, no. 4, December 1976, pages 308-320.

[49] Thomas J. McCabe and Charles W. Butler: *Design Complexity Measurement and Testing*, in *Communications of the ACM*, vol. 32, no. 12, December 1989, pages 1415-1425.

[50] Bertrand Meyer: *Introduction to the Theory of Programming Languages*, Prentice Hall, 1990.

[51] Bertrand Meyer, *The New Culture of Software Development: Reflections on the Practice of Object-Oriented Design*, in *Advances in Object-Oriented Software Engineering*, eds. D. Mandrioli, B. Meyer, Prentice Hall, 1991.

[52] Bertrand Meyer: *Eiffel: The Language*, 2nd printing, Prentice Hall, 1992.

[53] Bertrand Meyer: *Applying "Design by Contract"*, in *Computer* (IEEE), 25, 10, October 1992, pages 40-51.

[54] Bertrand Meyer: *Object Success*, Prentice Hall, 1995.

[55] Bertrand Meyer: *Practice to Perfect: The Quality First Model*, in *Computer (IEEE)*, May 1997, pages 102-106, available at se.ethz.ch/~meyer/publications/computer/quality_first.pdf.

[56] Bertrand Meyer: *UML: The Positive Spin*, in *American Programmer*, 1997, available at archive.eiffel.com/doc/manuals/technology/bmarticles/uml/page.html.

[57] Bertrand Meyer: *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.

[58] Bertrand Meyer, Christine Mingins and Heinz Schmidt: *Providing Trusted Components to the Industry*, in *Computer* (IEEE), vol. 31, no. 5, May 1998, pages 104-105, available at se.ethz.ch/~meyer/publications/computer/trusted.pdf.

[59] Bertrand Meyer: *The Role of Object-Oriented Metrics*, in Computer (IEEE), vol. 31, no. 11, November 1998, pages 123-125, available at se.ethz.ch/~meyer/publications/computer/metrics.

[60] Bertrand Meyer, *Every Little Bit Counts: Towards Reliable Software*, in *Computer* (IEEE_, vol. 32, no. 11, November 1999, pages 131-133, available at se.ethz.ch/~meyer/publications/computer/reliable.pdf.

[61] Bertrand Meyer: *The Grand Challenge of Trusted Components*, in *ICSE 25* (International Conference on Software Engineering, Portland, Oregon, May 2003), IEEE Computer Press, 2003.

[62] Bertrand Meyer: *The Power of Abstraction, Reuse and Simplicity: An Object-Oriented Library for Event-Driven Design*, in *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, eds. Olaf Owe, Stein Krogdahl, Tom Lyche, Lecture Notes in Computer Science 2635, Springer-Verlag, 2004, pages 236-271, available at se.ethz.ch/~meyer/publications/lncs/events.pdf.

[63] Bertrand Meyer: *Offshore Development: The Unspoken Revolution in Software Engineering*, in *Computer* (IEEE), January 2006, pages 122-124, available at se.ethz.ch/~meyer/publications/computer/outsourcing.pdf.

[64] Bertrand Meyer: *What will remain of Extreme Programming?*, in *EiffelWorld*, Vol. 5, no. 2, February 2006, available at www.eiffel.com/general/monthly_column/2006/February.html.

[65] Bertrand Meyer and Karine Arnout: *Componentization: the Visitor Example*, to appear in *Computer* (IEEE), 2006, draft available at se.ethz.ch/~meyer/publications/computer/visitor.pdf.

[66] Microsoft: Visual Studio pages at msdn.microsoft.com/vstudio.

[67] Richard Mitchell and Jim McKim: *Design by Contract by Example*, Addison-Wesley, 2001.

[68] Carroll Morgan: *Programming from Specifications*, 2nd edition, Prentice Hall, 1994, available at web.comlab.ox.ac.uk/oucl/publications/books/PfS/.

[69] John Musa: *Software Reliability Engineering*, 2nd edition, McGraw-Hill, 1998.

[70] Glenford J. Myers, Corey Sandler, Tom Badgett and Todd M. Thomas: *The Art of Software Testing*, 2nd edition, Wiley, 2004.

[71] Jeff Offutt: Mutation testing papers at www.ise.gmu.edu/~ofut/rsrch/mut.html.

[72] John Pincus: presentations (mostly PowerPoint slides) on PREfix and PREfast at research.microsoft.com/users/jpincus/.

[73] Eric Raymond: *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O' Reilly, 1999; earlier version available at www.firstmonday.org/issues/issue3_3/raymond/.

[74] Software Engineering Institute, CMMI site, available at www.sei.cmu.edu/cmmi.

[75] Matt Stephens and Doug Rosenberg: *Extreme Programming Refactored: The Case Against XP*, aPress, 2003.

[76] Axel van Lamsweerde: *Goal-Oriented Requirements Engineering: A Guided Tour*, in Proceedings of the 5th IEEE International Symposium on Requirements Engineering, August 2001, available at tinyurl.com/mscpj.

[77] Verified Software: Theories, Tools, Experiments: International IFIP conference, ETH Zurich, October 2005, see VSTTE conference site at vstte.ethz.ch.

[78] John Viega: *The Myth of Open-Source Security*, 2000, available at www.developer.com/tech/article.php/626641; follow-up article, *Open-Source Security: Still at Myth*, September 2004, available at www.onlamp.com/pub/a/security/2004/09/16/open_source_security_myths.html.

[79] Jeffrey M. Voas and Gary McGraw: *Software Fault Injection: Inoculating Programs Against Errors*, Wiley, 1998.

[80] Jos Warmer and Anneke Kleppe: *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd edition, Addison-Wesley, 2003.

[81] Elaine J. Weyuker and Bingchiang Jeng: *Analyzing Partition Testing Strategies*, in IEEE *Transactions on Software Engineering*, vol. 17, no. 9, July 1991, pp. 97-108.

[82] Wikipedia: entry "Mars Climate Orbiter", available at en.wikipedia.org/wiki/Mars_Climate_Orbiter.

[83] Edward Yourdon: *When Good Enough Software Is Best*, in *Software* (IEEE), vol. 12, no. 3, May 1995, pages 79-81.