

Chapter 2

Introduction to Testing

*“There are two kinds of failures: those who thought and never did,
and those who did and never thought”*

Laurence J. Peter

2.1 Testing Challenges

C. Kaner [Ka04] wrote: “The quality of a great product lies in the hands of the individuals designing, programming, testing, and documenting it, each of whom counts. Standards, specifications, committees, and change controls will not assure quality, nor do software houses rely on them to play that role. It is the commitment of the individuals to excellence, their mastery of the tools of their crafts, and their ability to work together that makes the product, not the rules. [...] Much of the polishing may be at your suggestion, as the tester.”

I agree fully with his opinion because I experienced very often that an individual’s initiative and engagement overcame a team’s inertia and made things move.

The powerful combination of the right methods and test tools is only effective if individuals are willing to learn processes and support to use them in their daily testing job. Processes are the basement of interaction and communication between teams to efficiently produce and exchange deliverables. Ensuring that a software application can scale and perform to meet service-level agreements (SLAs) and user expectations is one of the most difficult yet critical tasks in the testing process. A software application that meets all its functional requirements but not the end user's responsiveness expectations will ultimately be a failed software development project.

The National Institute of Standards and Technology (NIST) in the US stated that 80 percent of the software development costs of a typical project are spent on finding and correcting defects. The Gartner Group came to the conclusion that 50 percent of deployed applications are rolled back.

2.1.1 Business and IT

The role of IT and the contribution it brings to business effectiveness remains largely misunderstood. In my career as an IT professional, I experienced quite frequently the underlying management's belief that IT projects consistently under deliver. However, most of the current research in this area points at endemic business failure being the real challenge: unrealistic goals, poor business cases, and a lack of vision. The IT organization is therefore under very high pressure to deliver new applications, to provide support to a rapidly increasing number of systems, and to do both efficiently with stagnant staff levels and overall declining IT budgets. To deliver innovative solutions, CIOs turn to collaborative software tools and a multi-channel, services-oriented architecture (SOA). This will enable large companies to manage costs more efficiently and to reduce time-to-market. Project scopes have increased dramatically and the ever growing complexity of the IT systems will reach new highs. In this context, IT teams are expected to manage an increasingly complex world of systems and applications.

2.1.2 The Human Factor

The most critical success factor in all projects is undoubtedly human behavior.

Old habits are hard to break, and the tendency to reach completeness of a software component or application within a closed unit persists in large projects. It leads, in many cases, to software delivery units working in a rather isolated way with a tendency towards "silo-thinking." The hierarchical structure in many organizations reinforces the barriers across which project artifacts must be delivered at the different stages of development and testing. As a matter of fact, business analysts and domain experts often work in isolation, rather than collaborating with IT leaders at the very beginning of the project. An explanation of this attitude could be that some mistrust exists, which can be rooted in bad experiences in collaborative work made in the past.

Finally, restructuring the organization is a permanent factor of unsteadiness which generates know-how transfer problems, handling errors, and the demotivation of individuals with a negative impact of project results.

My personal experience is that a skilled and motivated (testing) team represents the most valuable asset in many situations because it can:

- Compensate missing analytic skills or management deficiencies
- Support junior programmers (training on the job)
- Provide training and advice to business analysts (help design test cases/show how to use efficiently the test platform)
- Establish a disciplined and methodical approach to problem solving to the many people involved in the project
- Promote and support the test process network philosophy.

As applications grow larger and are deployed internationally, they require involving larger, more diverse, and geographically dispersed teams. This new dimension makes teamwork an important factor in the production and testing of reliable and user-friendly software products. A team forces members to focus on the big picture, enabling a better understanding of the interrelationship between components, and therefore on the success of the entire product. A key point is also to involve team members in the decision process as frequently as is possible, creating trust and reinforcing the overall motivation of the team.

2.1.3 Old and New Worlds

Legacy Applications

A legacy application may be defined as any application based on older technologies and hardware, such as mainframes, that continues to provide core services to an organization. Legacy applications are frequently large, monolithic and difficult to modify. Migrating or replacing a legacy application to a new platform often means reengineering the vast majority of business processes as well. It is generally an extremely cost-intensive and risky endeavor.

According to a strategic paper published by Ovum in 2006 [GBMG06]: “Legacy is the problem child of IT, hampering growth, agility and innovation; the people with the in-depth application skills and knowledge required to keep it going are nearing retirement; it soaks up the IT budget. At the same time, legacy is the life-blood of public and private sector businesses around the world; we cannot do without it.

No one plans for legacy, it just happens and, left alone, the problems get worse. Legacy renewal is a journey that must be driven by business leaders, navigated by IT leaders and fuelled by vendors; anything less will fail. In this report, we explain why. We provide a summary view of current tools used to address the problem and why these alone are inadequate. There are important lessons for business leaders, IT leaders and vendors.”

While IT organizations struggle to integrate new technologies into their portfolios, the amount of legacy technology still in production continues to rise.

Ovum estimates that the worldwide investment in legacy technology amounts to \$ 3 trillion. This includes investment in hardware platforms, software licenses, application development (and maintenance) costs, and third party services. This does not include operational costs, or the costs associated with people and processes using those systems.

Ovum also estimates that the current inventory of production Cobol running on proprietary mainframes is 150–200 billion lines of code. While staggering, this figure does not do justice to the breadth of platform subsystems, programming languages (2GL, 3GL, and 4GL) flat files, databases, TP monitors, and screen formats that lock these applications into a technology time capsule.

Most companies in the banking sector claim to be running (and maintaining) 80–100 millions lines of legacy code.

Forrester Research, in September 2006, came to similar conclusions: “Legacy technology continues to run core business solutions for medium, large, and 2000 global companies. [...] The custom-built legacy applications running on mainframes are still the best transaction-processing platforms available.”

Graham Booch, Chief Scientist at IBM, estimates the volume of legacy code worldwide in production at a staggering 800 billions lines of code, the vast majority of it written in COBOL. The maintenance of the legacy code today exceeds 80% of the total development costs for software.

ERP products such as systems applications and products (SAP) are monolithic, mainframe-based solutions first released early in the 1970s, and in 1979, for SAP R/2.

The SAP’s client/server version (R/3) was released in 1992. The Butler Group Review concluded in its SOA strategy report in September, 2006:

“One of the challenges of complex, packaged applications such as ERP is that it can be difficult to map to what an organization actually needs, and often customization is required. In a typical ERP installation, less than 25% of the standard ERP code, and less than 25% of customized code, was actually used.”

This causes big problems for the testing community.

In the 1990s, the deregulation of financial services generated a flood of mergers and acquisitions leading to new business scenarios. To address ever-growing consumer needs and increased global competition, banks have introduced new technology in the front applications cooperating with existing legacy systems. Legacy systems, also named “heritage systems,” are applications that have been inherited from an earlier era of information technology. They don’t necessarily refer exclusively to older systems, but could be new systems which have been designed under the same design paradigms. Re-engineering archaic applications is not an easy task and requires using contemporary design paradigms, relational databases, and object orientation. These systems are not responsive to change, whether they are wrapped in a new service layer or encapsuled. Their maintenance is a financial burden.

On the other side, the market pressure forces large companies to provide new business services based on a service-oriented architecture (SOA) to create flexible and highly adaptative event-driven applications. The strategy adopted to renew the IT platforms consists in putting service-oriented wrappers around the old systems (legacy) and integrating them at the mid-tier through SOAs. This has resulted in massively complex IT architectures, multi-year migration initiatives, and soaring costs. The overall compatibility of peripheral applications (front-end) and core business systems has to be tested adequately inside and outside both IT worlds. Figure 2.1 shows that backward and lateral compatibility are main issues in the highly heterogeneous IT architecture found in the vast majority of large companies and organizations throughout the world.

IT faces the Gordian knot of ever-increasing business and legal requirements with constantly reduced budgets and exploding maintenance costs.

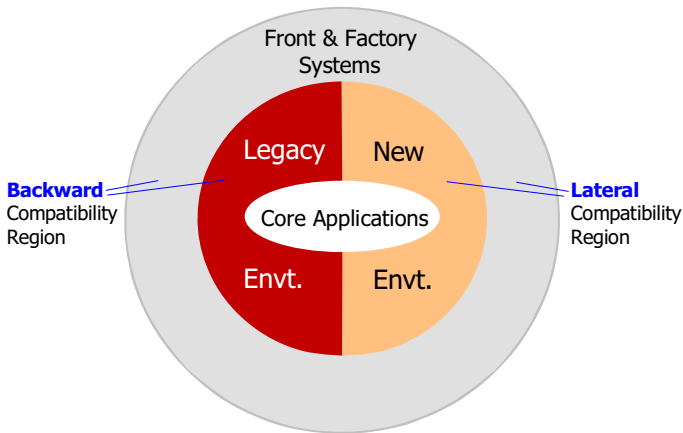


Fig. 2.1 Backward and lateral compatibility impacts legacy and new applications

2.1.4 Banking Platform Renewal

In financial services, 96% of European banks have either started to renew their overall application landscape or have concrete plans to do so. According to “IT View and Business Trends” by Jost Hoppermann, the necessary project budgets for software and service are huge: up to €250 million in extreme cases. Software and services costs for European bank’s renewal initiatives, spread across at least 10 years, will be in the €100 billion range.

Undertaking the Redesign

An article entitled “Better Operating Models for Financial Institutions,” published in November 6, 2005, in “McKinsey on IT” magazine, reveals the multiple challenges the finance industry is facing presently:

“In applying the five elements of an operating model to end-to-end processes, banks must be careful to balance some ongoing fundamental operating requirements against the need to create capacity for new growth. Thus, they must continue to deliver a broad portfolio of products and services across a wide-ranging client base while managing the associated risks. Complicating the challenge are shortcomings common to many financial institutions’ operations and IT organizations, including highly fragmented operations and technology platforms (sometimes the result of inadequate post-merger integration); inconsistent alignment around which capabilities are best shared and which should remain specific to a particular line of business; and a lack of firm-wide standards in, for example, data, interfaces and functional performance. [...] Complicating matters are increasingly stringent regulatory regimes, which may be standardizing across international boundaries but appear unlikely to loosen anytime soon.”

Fortunately, if some external trends are making life more difficult, others favor the changes these companies need to meet:

1. **Global sourcing:** Foremost among these trends is the rising quality and accessibility of resources in low-cost locations, combined with inexpensive telecom bandwidth for managing operations remotely. Financial industries have led the way in tapping global labor markets to run their back-office operations, and now they can exploit their experience of coordinating these activities across remote locations to offshore even more advanced operations.
2. **Technology:** The financial sector is also ahead of others in adopting IT systems for the rapid and accurate capture of data, inexpensive information storage, and high processing capacity and performance, as well as the ability to share and process data across multiple locations.
3. **Process improvement:** A third enabling factor is the financial sector's adoption of process improvement methodologies, including lean techniques and Six Sigma, that got their start in industrial settings but are now increasingly applied to service organizations. A core principle of these techniques is to change mindsets by teaching employees how to improve a process constantly. Once trained, these employees can become effective internal change agents, spurring transformation across the organization.

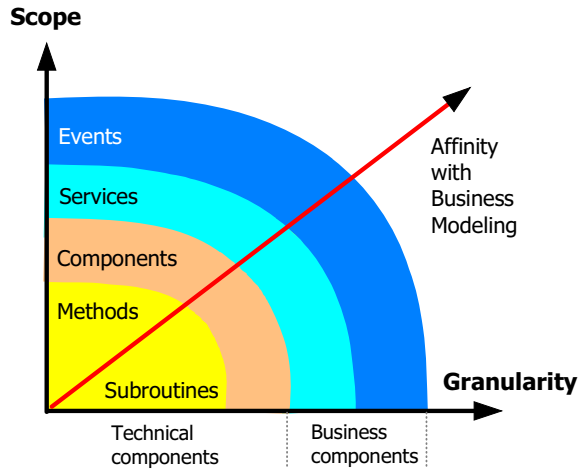
IT platforms renewal (hardware, processes, software, training) is a long-term endeavor. Therefore, it is essential to implement adequate (e. g., SOX-compliant) management and IT processes, and to use extensively integrated test platforms and tools to success on these multi-year migration projects. This book reflects experiences gained in such a challenging and tantalizing context.

2.1.5 Complex Testing

Testing large systems today is a daunting task because development, test, and production environments consist of multiple client/server machines, mainframes, personal computers, portable devices, various operating systems, a multitude of applications written in different programming languages, proprietary database systems, a complex network, and stringent security rules for accessing data and applications. The degree of complexity will reach a higher level with the introduction of composite service-oriented (SOA) applications, as depicted in Fig. 2.2 at the end of this section.

Thus, effective testing requires setting up and reusing multiple environments at will, whether for initial testing, regression testing, acceptance, or system tests. The test data pool must also be set up or restored in the same flexible way. Large companies use dedicated platforms for each test environment: CT/CIT, IIT, AIT, MIT, STE, UAT, including staffing and infrastructure either on mainframe or preferably on high-end UNIX systems. The system test configurations for AIT and STE testing are very close to those used in the productive environment. Therefore, the data processed reaches from 20% (AIT) to 100% (STE) of the production volume, which is pretty cost-intensive.

Fig. 2.2 SOA composite applications



Source: Gartner, Inc. "SOA: Composite Applications, Web Services and Multichannel Applications" by Yefim Natis, 2006

The resources needed to test applications and systems aim to achieve precise goals, vital for company operations and customer satisfaction:

- to assess conformance to specification
- to assess quality
- to block premature product releases
- to conform to regulations
- to find defects
- to find safe scenarios for use of the product
- to help managers make signoff decisions
- to maximize defects discovery
- to minimize safety-related lawsuit risks
- to minimize technical support costs
- to verify the correctness of the product

2.1.6 Global Testing

Software development projects in large organizations are often geographically distributed and multinational in nature, introducing new layers of organizational and technical complexity. Different versions of software have to be maintained in parallel for one or more core systems, along with different mid-tier services. To deal with global testing, the best approach is to establish a centralized testing with a main location responsible for the core product and other locations giving feedback about the rollouted versions: those being tested locally This solution requires a networked incident tracking and channeling process (ITC) supported by

a central test tool or test repository. These topics will be discussed later in detail in Sect. 5.2, “Core Testing Processes” and Sect. 8.2, “Test Progress Reporting.”

2.1.7 The Value of Testing

The real purpose of testing is a permanent question. Every time a tester discovers a defect that is repaired before software is shipped to the customer, he saves money because the nonconformity is reduced and additional testing effort decreases. One can argue that the added value is increased. Others mean that saving money this way is creating value for the organization. Testing's main purpose is to verify the good functioning of the software developed. This is the most critical and valued information for the customer: the product works as designed with the required quality. For the company or organization delivering the product, it reduces or minimizes a reputation for financial losses that could rise because of the quality aspects of the software. Testing must provide actual, correct, timely, exact, verifiable, and complete information to the decision makers. This information – which can be quantified with test metrics – is the real added value of testing.

In 2006, Mercury (Nasdaq: MERC) and the Economist Intelligence Unit conducted a worldwide survey in 21 countries, asking 758 IT managers and CIOs about: “Driving Business Value from IT – Top Challenges and Drivers for 2005”.

The results showed that in recent years, the gap between business requirements for an IT organization and IT's delivery capability grew dramatically. Three main topics drive the IT market today:

1. Strategic outsourcing
2. Regulatory requirements: IAS, Sarbanes-Oxley, LSF and anti-money laundering (AML)
3. The complexity of the IT applications

In this context, development, integration and maintenance of the software requires a high level of knowhow and experience from the IT professionals. Sixty percent of all IT managers in this study stated that the testing of applications is the driving force generating added value.

The Cost of Testing

Economic costs of faulty software in the U.S. have been estimated to be 0.6% of the nation's gross economic product (GDP). Annual costs of inadequate infrastructure for software testing is estimated to range from 22.2 to 59.5 billion US dollars by the American National Institute of Standards and Technology (NIST).

The European testing market alone is estimated to reach a volume of 65 billion Euros in 2008 with sustained growth.

The volume of required tests can grow exponentially, considering the number of specific parameters, environmental factors, and other variables existing in the project context. Depending of the project size and number of software components to test, testing activities can generate costs from 25% to more than 100% of the total development costs. Therefore, the test strategy needs to be adapted to the project size, integrating the right risk profile (cutting-edge technology/strategic initiative) and taking into account the business priorities. Test volume and test intensity should be in correlation with the potential risks of failures leading to financial losses and/or reputation damages for the company.

Realistic planning will help modulate the test efforts as difficulties arise. The prioritization of quality goals is necessary to focus testing activities on the main aspects most important to stake holders. Quality models such as ISO 9126 allows setting consistent quality goals and should be implemented in your project.

2.2 The Significance of Requirements

Since decades, requirements analysis is the most error-prone process on the way to a well-functioning software product. This situation can barely be improved because the causes of nonconformity are largely human-based errors: the misunderstanding of a customer’s needs, insufficient knowledge in the field of expertise, poor communication, non-documented facts, and so on.

Not only individuals but also teams can produce requirements which need a lot of rework. Incomplete or imprecise requirements include errors generating a cascade of subsequent faults in the design and implementation phases of the software which cause failure at run time [IEEE90], [Ch96]. Figure 2.3 shows the causal chain issued by requirements.

This explains why defects generated by requirements errors can’t be detected early enough and are costly to remove in terms of time and resources. As stated in numerous surveys [LV01], [BP84], [Wa06] these fundamental errors account for 50% of all defects found in the software.

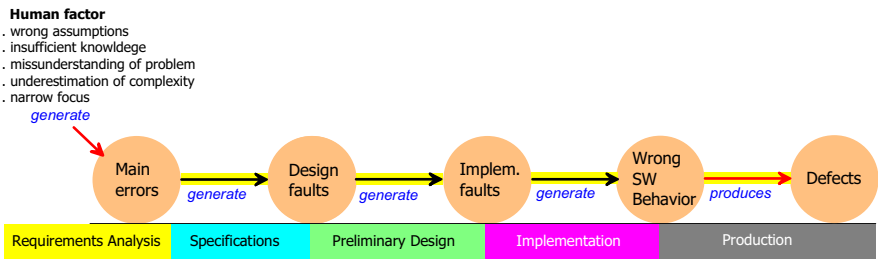


Fig. 2.3 The impact of requirements in the software production chain

2.2.1 *What is a Requirement?*

A requirement is:

1. A condition or capacity needed by a user to solve a problem or achieve an objective.
2. A capability that must be met or possessed by a system or software component to satisfy a contract, standard, specification, or other formally imposed documents. Requirements may be functional or non-functional.
3. A documented representation of a condition or capability as described in 1 or 2.

2.2.2 *Meeting the Unknown*

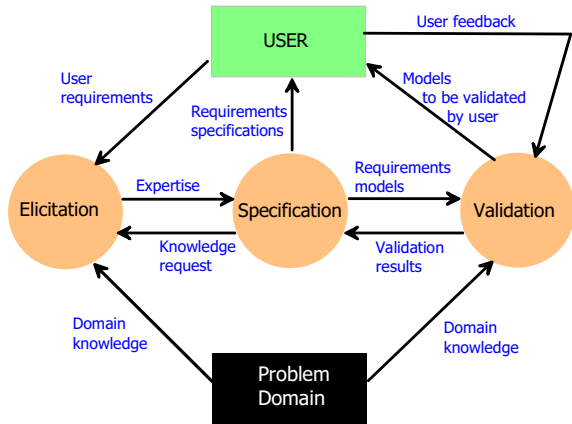
Each adventure begins with a large amount of fear and uncertainty. At a project start the situation is very similar:

- Uncertainty about the technology selected to solve the problem
- How it will work
- Uncertainty about the design proposed
- Uncertainty about the code implemented
- Uncertainty about the runtime behavior of the application
- Uncertainty about the production and maintenance costs
- Uncertainty about the customer's acceptance
- Uncertainty about the delivery schedule

The end product is – in most of the cases – very different from what the project team thought it would deliver when it started. The main reason is that *incremental requirements* emerge as soon as users see the preliminary software at work. These “additional” customer needs are, in fact, part of the original solution but were not properly identified in the early stages of the requirements analysis. To reduce inherent uncertainty, the project team has to address three aspects:

- Defining what the product will do
- Accurately designing and developing the product
- Knowing exactly what will be developed

Most companies using the waterfall (V-model) method try to eliminate all uncertainty before beginning development (upfront thinking) but experienced teams know pretty well the limits of this method. In many cases, project leaders tend to add some prototyping to know more about the customer's needs. But this is clearly not a substitute to compensate for a lack of business expertise, the analysis of existing solutions and insufficient knowledge of the operational systems in use. Prototyping can be best considered as a fallback position, not the answer in and of itself. Requirements engineering is the metaprocess which addresses the problem,

Fig. 2.4 RE process diagram

Source: Locopoulos

but that topic is not part of this book. We can briefly mention that the requirements engineering (RE) process includes four sub-processes:

- Gathering
- Elicitation
- Specification
- Validation

Locopoulos defined the framework for RE processes, as illustrated in Fig. 2.4.

2.2.3 *Characteristics of Requirements*

Business requirements have to be complete and need to reflect a high level of overall consistency. This can be achieved by addressing the following aspects:

1. Completeness
 - Functional completeness: Is the functions domain clearly delimited?
 - Content completeness: Are all significant values properly described?
 - Internal completeness: Are all cross-references available?
 - External completeness: Are all mandatory documents available?
 - Omissions: Is something important missing?
2. Consistency
 - Terminological consistency: Are all terms and abbreviations unique and correctly explained?
 - Internal consistency: Are all requirements free of contradictions or incompatibilities?
 - External consistency: Are all external objects compatible with the requirements?
 - Circular consistency: Do cyclic references exist?

3. Other Properties

- Requirements must also be: detailed, comprehensive, attainable, testable, revisable and realistic.

Table 2.1 summarizes the different types of requirements.

Table 2.1 Requirement types

Category	Description
Functional requirements	Requirements that define those features of the system that will specifically satisfy a consumer need, or with which the consumer will directly interact
Interface requirements	Requirements that define those functions that are used in common by a number of projects in different business domains
Operational requirements	Requirements that define those “behind the scenes” functions that are needed to keep the system operational over time
Technical requirements	Requirements that identify the technical constraints or define conditions under which the system must perform
Transitional requirements	Requirements that define those aspects of the system that must be addressed in order for the system to be successfully implemented in the production environment, and to relegate support responsibilities to the performing organization
Typical requirements	A listing and description of requirements that a typical enterprise/business domain might possess in the problem area

Note: Adapted from NYS

2.2.4 Requirements Elicitation

To gather requirements dealing with large and complex systems, a global approach is needed that takes into account all relevant (scientific/strategic) objectives, mission, advanced concepts, cross requirements, technology roadmaps, and long term visions. This is the “will be” requirements summary. A proof of concept in the early stages of elicitation can be necessary to investigate operational constraints, available technologies and the overall behavior of the target solution. For this purpose, simulation and system modelling are used. The results of these investigations deliver valuable information about potential deficiencies and strengths of the system to be developed.

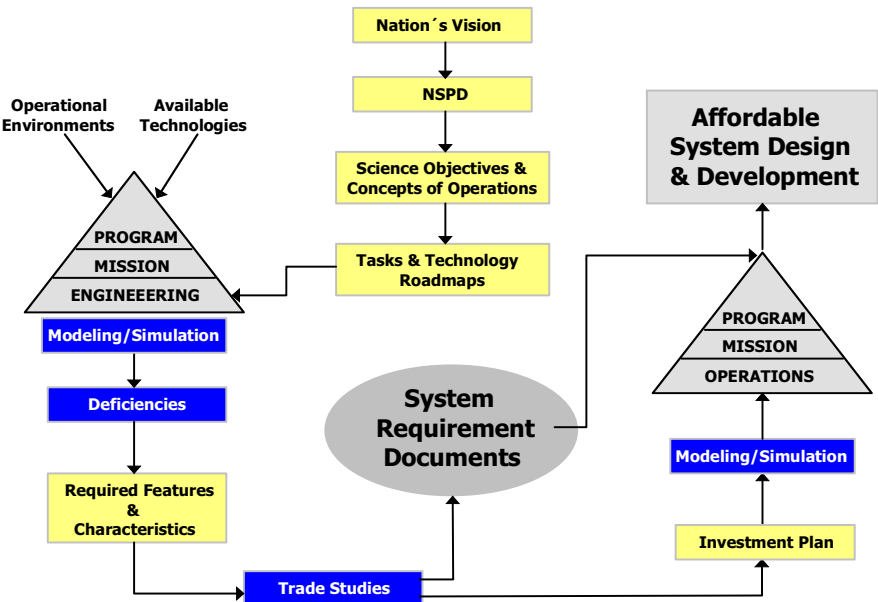
The next step is the refinement of the system’s features and characteristics followed by financial and commercial studies. Once finalized, the requirement documents are ready for review and approval, as shown in Fig. 2.5.

2.2.5 Main Problems with Requirements

The most frequent causes of defects will be analyzed in Sects. 7.2 and 7.3. In fact, defects are mainly produced because basic errors are produced at the very beginning of the software production chain in the form of defective requirements. Figure 2.6 depicts the testing perimeter with sources of defects coming from new business requirements. Fuzzy requirements are mainly generated in areas concerning business processes, business logic, and business rules, leading to defective coding and wrong software implementation.

An important aspect in managing requirements is to maintain a bidirectional traceability between core requirements (the business view) and the product-related requirements (the technical view). The maintenance burden of any large-scale software applications along the software development life cycle (SDLC) will be highly impacted by overlapping requirements, due to a defective solution management process. Adding to that, compliance issues increase the pressure on IT to deliver better software.

In an article entitled “Traceability: Still a Fact of Life in 2008,” Edward J. Corraia wrote: “While calls continue to repeal Sarbanes-Oxley compliance and accountability laws, for now they’re still a reality for any public company. So SDLC traceability remains critical for compliance with regulations that, by some estimates, shave about 4 percent off the bottom line. ‘Traceability is a fundamental



NSPD: National Security Presidential Directive Source: National Aeronautics and Space Administration (NASA)

Fig. 2.5 Requirement elaboration in a US government agency (NASA) (Source: NASA)

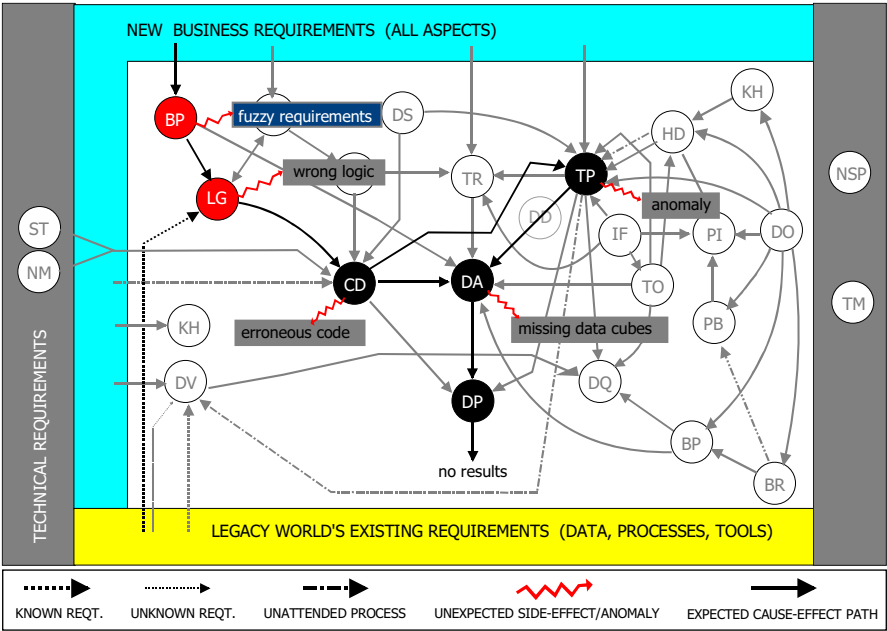


Fig. 2.6 Diagram showing unclear requirements as the defect source

part of any software development process, and in most cases is critical for satisfying any compliance or regulatory constraint,’ says John Carrillo, senior director of strategic solutions at Telelogic. ‘Most often, traceability is accomplished through a top-down approach – using best practices and solutions that link features and requirements down to the changed source code.’

The ability to track requirements has become central to innovation in a wide range of products and services, he says, including many in which software traditionally didn’t play a major role, such as the automobile. Today, software in the average car includes more than 35 million lines of code, and automakers spend between two and three billion dollars a year fixing software problems.’ In such scenarios, requirements are ‘the thread that ties all phases of the product life cycle together.’ So Carrillo calls the alignment between the organization’s developers and application requirements ‘the quality cornerstone of nearly every successful large software development strategy.’”

Traceability links define relationships between requirements and design artifacts; they help to support a number of essential activities related to requirements validation, impact analysis, and regression testing. In practice, these techniques have focused upon the functional requirements of the system or solution. Therefore, the nonfunctional requirements (stability, reliability, performance, scalability, security, safety) are not considered. After closer analysis, these non-functional requirements tend to decompose into lower level requirements that are

true functional requirements in nature. As a result, these functional changes may introduce unexpected side-effect and cause a long-term degradation of the system quality.

The best way to address this major challenge is to establish a verification and validation procedure assigned permanently to a joint team composed of business domain experts and IT specialists which qualifies the requirements coordination and supervises the elaboration of final change orders. This is the traditional approach in large projects to manage requirements. However, due the high volume and networked nature of requirements, the work of the experts group is error-prone, relatively slow and expensive. To achieve better results, up-to-date modeling technology and the appropriate tools can accelerate this process, produce much better results and offer full traceability. In recent years model-based testing (see Sect. 3.5.1) has proven to be very valuable in many branches, producing software of the highest quality.

Using model-based testing (MBT), errors and/or omissions, built in requirements can be detected automatically, early, and systematically. This is due to the fact that model-based testing provides a unique bidirectional traceability to requirements:

- It provides a direct requirement-test traceability
- It provides a requirement model in conjunction with model-test traceability.

See Fig. 3.9 in Sect. 3.5.1 for more details.

2.2.6 Risks Associated with Requirements

The most important risks which can occur in this area are the following:

- The number of requirements can become unmanageable if they are not under control.
- Requirements are related to one another and also to other deliverables in the solution delivery process.
- Requirements need to be managed by multiple cross-functional entities.
- The requirements modification process is not implemented or insufficiently implemented.
- Requirements are not reviewed at all or inadequately reviewed.
- Requirements describe only the design specification.
- There is an over-improvement of the requirements.
- Functional requirements are only considered.
- The user or customer is not enough involved.
- Critical requirements are overlooked.
- The business case is incomplete.

To reduce risks in large software projects, managers should bear in mind that people view the project from their own perspective with different expectations. When the initial objectives, assumptions, and requirements are not clearly and

precisely defined in writing, there is no (good) common starting point. An inherent expectation gap exists which can be a major cause of trouble at the end of the project.

2.2.7 *Recommendations*

To reduce significantly the volume and impact of changing requirements during the critical phases of a project (e. g., development and testing), it is strongly recommended to apply these simple rules:

- Understand perfectly the customer's business case.
- Integrate all customer needs included in the business case, but avoid an over-specification of the requirements.
- Requirements must be reviewed and the modifications agreed upon by all parties.
- Focus on what is really needed and must be delivered.
- Don't mix *what* is to be developed with *how* it will be developed.
- Work closely with the project's stakeholders to be able to understand, in an early stage of the project, how changes can be incorporated in alternate plans and strategies.
- Customers and management must clearly understand the impact on software delivery schedule, the inherent risks, the costs and the quality issues related to modified requirements.
- Maintain a constant dialogue with the customer to detect potential acceptance problems or other mishaps regarding the target solution.
- Documentation is of prime importance to precisely describe what the end user wants for the solution and to guide the development work later on.
- Interview notices are helpful and can be attached to formal requirements.
- Most organizations have a defined process for maintaining requirements documentation, but the process is quite frequently not enforced.
- Control over changes is essential to ensure that the design, development and test activities are only performed on approved requirements. At the reverse, the released product is guaranteed to have defects (more than expected!).
- It is a good practice to assign versions and approve each requirements change.
- Use change-case modeling to anticipate changes of your product or solution.
- Utilize domain experts to perform requirements engineering tasks.
- Collect requirements from multiple viewpoints and use formal methods where applicable.
- To be consistently successful, the project team must assume ownership of the requirements process. Take responsibility for the quality of the requirements.
- Know that an organization reaching CMM level 5, considers a request to change requirements as a non-conformance: an opportunity to analyze the process, to improve it, and reduce changes in the future.

- Remember that people write about problems and failures as much as successes. Take the chance to profit from their experiences.
- Requirements tracking with the user's involvement should cover the whole cycle: analysis, validation and verification, architecture design, and architecture design verification. This will be shown later in Fig. 2.21.

2.3 The Nonconformity Problem

ISO 9000:2000 defines conformity as a non-fulfillment of a specified requirement, or more specifically: "A departure of a quality characteristic from its intended level or state that occurs with severity sufficient to cause an associated product or service not to meet a specification requirement."

Software quality assessment is a difficult task because quality is multi-dimensional in essence. It depends largely on the nature of the product and this needs a clear definition of the most important criteria defining the end product considered.

Testing helps quality assessment by gathering the facts about failures but it can't verify the correctness of the product. This is impossible to do by testing alone.

It can be demonstrated that the software or solution is not correct, or it can be attested that a minimum of faults was found in a given period of time using a predefined testing strategy. Positive tests help to check that the code does what it is intended to do; this viewpoint is largely used in the testing community but it is not sufficient to make good testing. Sound rules, good processes, and best practices are important ingredients for good testing but they can't automatically guarantee good results. Those are produced by motivated individuals producing quality craftsmanship, working in homogeneous teams which often must compensate the weaknesses of the project organization, scarce resources, and management deficiencies.

2.3.1 *How Defects are Born*

The basic mechanism of default production by [Ch96] is illustrated in Fig. 7.2.

I adapted the diagram to point out the following facts:

1. Inherent problems are located mainly outside the testing world
2. These problems are the source of basic errors
3. These errors generate faults as soon as some triggers activate them.

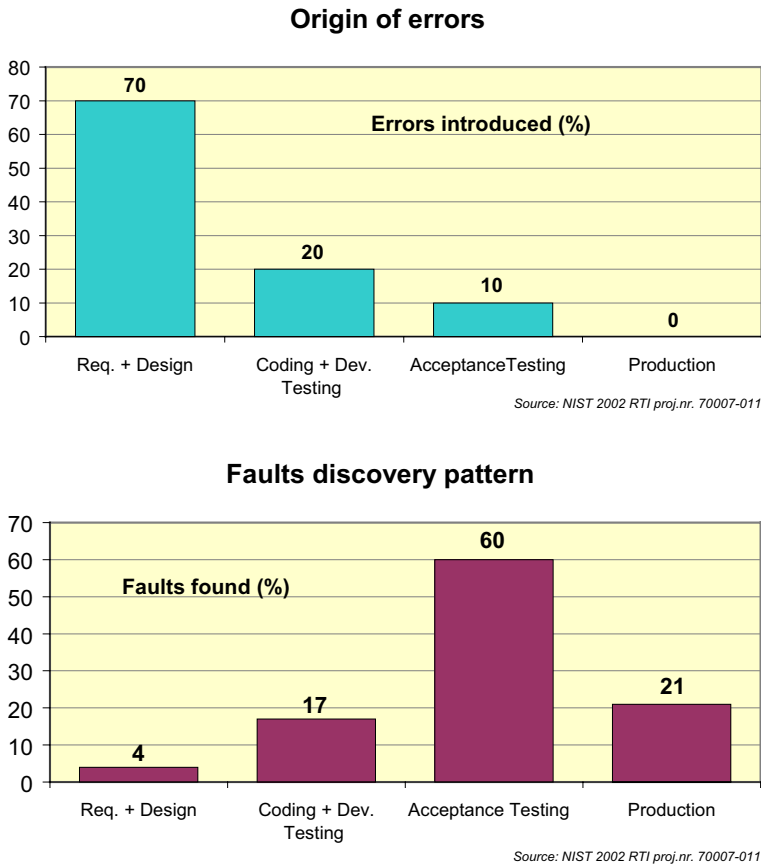


Fig. 2.7 Errors distribution pattern

We will examine, in Sect. 7.2, “causal chains explained” how apparent causes of defects and related symptoms are linked together. In a survey published in 2006, Mercury Interactive reveals the primary factors causing IT initiatives to fail to produce the expected benefits for the business. The four most important of them are: project management, requirements definition, software rollout and poor software quality.

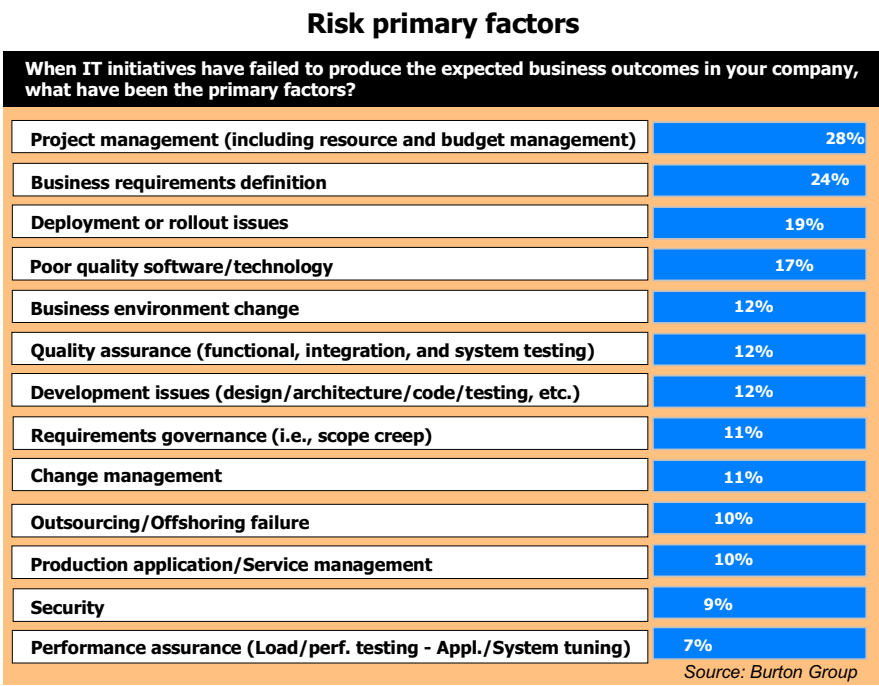


Fig. 2.8 Classification of risk factors in large IT projects

2.3.2 Nonconformity to Standards and Rules

The spectacular failure of the launching of the Ariane V is a good example of the disrespecting of basic engineering rules.

On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded.

The failure of the Ariane 501 was caused by the complete loss of guidance and attitude information 37 seconds after the start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was due to specification and design errors in the software of the inertial reference system.

The internal SRI software exception was caused during the execution of a data conversion from a 64-bit floating point to a 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer.

The second example is the Patriot missile failure. On February 25, 1991, during the Gulf War, an American patriot missile battery in Dharaan, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people.

A report of the General Accounting Office, GAO/IMTEC-92-26, entitled “Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia” reported on the cause of the failure. It turned out that the cause was an inaccurate calculation of the time since booting due to computer arithmetic errors. Specifically, the time in tenths of a second as measured by the system's internal clock was multiplied by 1/10 to produce the time in seconds. This calculation was performed using a 24 bit fixed point register. In particular, the value 1/10, which has a non-terminating binary expansion, was chopped at 24 bits after the radix point. The small chopping error, when multiplied by the large number, giving the time in tenths of a second, led to a significant error. Indeed, the Patriot battery had been up around 100 hours, and an easy calculation shows that the resulting time error due to the magnified chopping error was about 0.34 seconds. (The number 1/10 equals $1/24 + 1/25 + 1/28 + 1/29 + 1/212 + 1/213 + \dots$. In other words, the binary expansion of 1/10 is 0.0001100110011001100110011001100... The 24 bit register in the Patriot stored, instead, 0.00011001100110011001100 introducing an error of 0.00000000000000000000000011001100... binary, or about 0.000000095 of a decimal. Multiplying by the number of tenths of a second in 100 hours gives $0.000000095 \times 100 \times 60 \times 60 \times 10 = 0.34$.) A Scud travels at about 1,676 meters per second, and so it travels more than half a kilometer in this time. This was far enough away that the incoming Scud was outside the “range gate” that the Patriot tracked. Ironically, the fact that the bad time calculation had been improved in some parts of the code, but not all, contributed to the problem, since it meant that the inaccuracies did not cancel.

2.3.3 *Aging of Product Components*

In highly complex technical systems like NASA's space shuttle – the first ever certified as CMMI level 5 – aging hardware components can become erratic and influence directly or indirectly other components and the flight control system itself in unpredictable way. It is ostensibly a similar problem in a fuel sensor that ruined the shuttle's long-awaited return to flight. NASA did enlist a cross-country team of hundreds of engineers to figure out what went wrong. The agency has indefinitely delayed the launch.

In his Congressional testimony, the former ASAP chairman Blomberg, said that the safety group believes that the postponements of shuttle safety upgrades, and the delay in fixing aging infrastructure, was troubling. Also, the failure to look far enough ahead to anticipate and correct shortfalls in critical skills and logistics availability, he said, “will inevitably increase the risk of operating the space shuttle ... The problem is that the boundary between safe and unsafe operations can seldom be quantitatively defined or accurately predicted,” Blomberg advised. “Even the most well meaning managers may not know when they cross it. This is particularly true for an aging system.”

2.3.4 *Environmental Changes*

As we discussed in the previous chapter, software decay is unavoidable. It is quite remarkable that applications and systems developed decades ago can still function in our present time. These survivors from the technological Jurassic Park face the challenge to remain stable in a Web-oriented and Java-programmed environment. To test these systems requires a careful approach because the side-effects of software changes can become unpredictable and/or erratic. The positive aspects regarding these applications is that they are reasonably well-documented, making maintenance easier, as is much of modern software.

2.3.5 *Outdated Tests*

The aging of test artifacts is a major concern in large-scale projects, because many releases must be developed and maintained in parallel, necessitating the setting-up and running of a number of different test environments over long periods of time.

For each new release, all artifacts for regression tests must be checked on actuality, completeness, and suitability. In particular, scripts and test data should be verified carefully to avoid erroneous results and/or wrong system outcomes. Business and technical rules, in particular, need to be verified to avoid side-effect defects, including connectivity problems.

In this category fail defective or outdated firewall rules which are a frequent cause of defects impacting Web-based applications. User interfaces evolve rapidly as well, making scripts for automated tests obsolete.

Therefore, to achieve test readiness using older tests requires setting up a dedicated organization managing the test assets for each test environment with precise rules.

2.3.6 *Conformity Assessment*

Conformity assessment means checking that products, materials, services, systems, and people measure up to the specification of a relevant standard. Today, many products require testing for conformance with specifications, safety, security, and other regulations. ISO has developed many of the standards against which software products and applications are assessed for conformity, including standardized test methods. ISO has collaborated with IEC (the International Electrotechnical Commission) to develop and publish guide and standards in this matter. See also: <http://www.esi.es/>, <http://www.iso.org/iso/en/ISOOnline.frontpage>, and <http://www.wssn.net/WSSN/>.

2.3.7 The Costs of the Nonconformity

The ROI generated by software quality enhancement processes in large projects was analyzed in depth by the Air Force Research Laboratory Information Directorate (AFRL/IF). The study, entitled “A Business Case for Software Process Improvement” was prepared by Thomas McGibbon from the Data Analysis Center for Software (DACS) under contract number SP0700-98-4000. The publication date was 30 september 1999. The paper can be ordered at: <http://www.dacs.dtic.mil/techs/roispi2/>. It was recently enhanced by a tool named the ROI dashboard.

The ROI Dashboard

In response to increasing interest and attention from the software engineering and software acquisition community for benefits data from software technical and management improvements, the DACS presented the ROI Dashboard. The ROI Dashboard augments and updates the DACS report with the latest published data on benefits. The ROI Dashboard also graphically displays open and publicly available data and provides standard statistical analysis of the data.

2.3.8 Mission Impossible?

Working out realistic requirements seems to be one of the most challenging task on the way to good software solutions being delivered on time and with the required quality. In TechWatch, edited by the Butler Group – dated May, 2005 – Martin Butler and Tim Jennings analyzed the situation regarding ROI calculations of IT projects in large companies. They stated in this report: “The history of IT over-promising and under-delivering is a long one, and so it is hardly surprising that senior management had become skeptical where the delivery of IT systems is concerned. If you ask IT management why they over-promise, the answer is quite unanimous – we have to do this to compete with the other alternatives management have. What generally happens is that everyone bidding for work over-promises – driven by the silly demands of managers that are looking for the cheapest solution in the shortest amount of time. Generally speaking, the managers have no idea whether what is being promised is feasible or not – it’s often just a mindless macho exercise to try and screw the suppliers down as tight as possible. In response, the suppliers have become cunning and creative when constructing their proposals.” Looking back at more than thirty years in IT development, management, and consulting, I do agree with this crude analysis. In my opinion, things began to get really worse in the 1990s, because the business started to drastically reduce IT costs by setting unrealistic goals and schedules for IT projects year after year. The IT culture in large organizations today is massively cost-driven, and

even the most important asset – people – is managed as a “resource,” interchangeable at will. This could have serious consequences in the future on the stability and security of large and complex business systems, because some knowhow gets lost each time an IT professional leaves a project or the company. The more experienced he or she was, the bigger is the loss.

2.3.9 Complexity

The IT world’s most pre-eminent characteristic today is complexity. The Tech Watch report concludes: “Poor quality thinking always results in complexity – and IT is full of poor quality thinking.” As a result, we attempt hugely complex tasks with even more complex technologies, and wonder why the whole thing goes pear shaped.

A renaissance is overdue: a simplicity renaissance. Application development resembles the contorted models of the universe that existed before we realised that the universe does not revolve around the Earth. If we see information technology becoming simpler to use and understand, then we will be making real progress. Until then, expect to see even the world’s brightest organizations and people struggle with escalating complexity and the resulting problems it creates. The multiple facets of software complexity will be explained later in Chap. 7.

2.4 Test Artifacts

A test artifact is an abstract object in the digital world to be tested by a software component or a solution to verify the correct response/result of the code developed.

2.4.1 Classification of Test Artifacts

A release flash is commonly used to communicate to developers and testers the content of a new software release. The document describes the kind of test artifacts implemented in the current software version and in which context they must be tested. Test artifacts are as follows:

1. Software modifications: minor or major changes made to the current software
2. Change requests: actual requests to modify the current software
3. Last-minute fixes: hot fixes made in the productive environment and to be validated in the test world
4. Current defects: test incidents in the actual software release to be validated from a technical and/or business point of view

5. New functionality: functions developed in the actual release
6. Performance optimization: technical improvements to the current application and/or to standard software components
7. Software patches: service packs to be implemented in the actual software release

Test data, scripts and documentation in relation to the test artifacts are test assets managed in the central test repository. A release flash template can be found in Chap. 10.

2.4.2 Information Life Cycle

Data in the business world reflects basically relationships between partners, contracts, and products. All events, statuses, and requests must be collected, managed, processed, and communicated to the customer in a timely and coordinated manner. To achieve this goal, four main functions cover the information cycle:

- Information management
- Collection and processing
- Archiving and dissemination
- Exploitation and production

The resulting data flows are shown in Fig. 2.9.

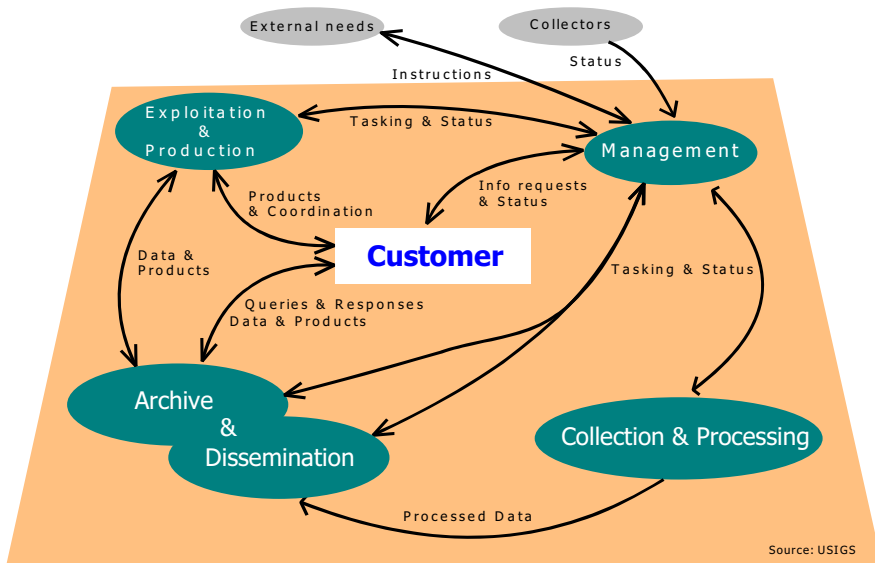


Fig. 2.9 Information cycle

2.4.3 Data Life Cycle

As illustrated in Fig. 2.10, data has a very long life cycle. Business data can be alive for decades and thus require long-term archiving. Test data also has to be archived up to seven years to be SOX-compliant. Inside this main cycle, test data is used during a life period of one to three years, depending on the release under test. Other test artifacts have a much shorter LC, generally a couple of months.

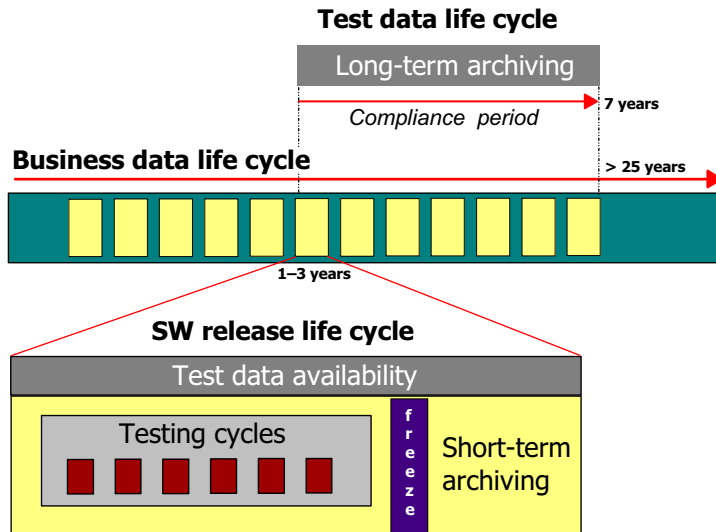


Fig. 2.10 Test data life cycle

2.5 Testing Predictability

Performing design for testability has many advantages, including more improved test coverage, simpler test design, and easier fault finding, offering more options for test automation. This will be discussed later in Sect. 3.5.4, “Test automation.”

By creating infrastructure support within the application – and specifically, to the interfaces of the software under test (SUT) – three notions can be supported efficiently [BIBuNa04]:

1. The *predictability* of the tests, which supports a means for conclusively assessing whether the SUT performed correctly or not.
2. The *controllability* of the tests, which permits the test mechanism to provide inputs to the system and drive the execution through various scenarios and states to foster the need for systematic coverage of the SUT functionality.
3. *Observability* of the system outputs that can lead to a decision as to whether the outputs are desirable (correct) or faulty.

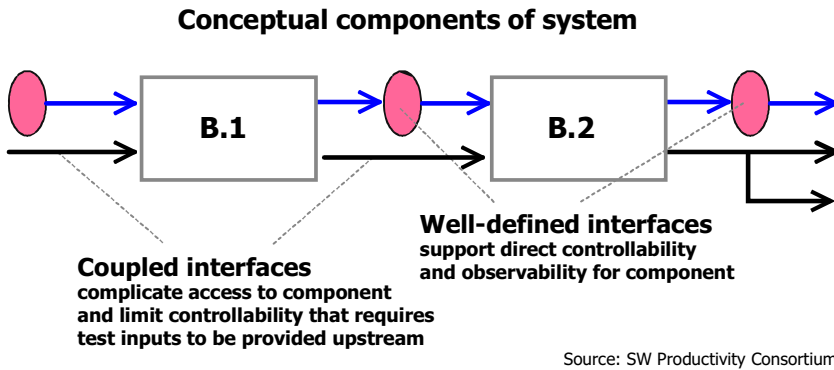


Fig. 2.11 Coupled interfaces

It is important for the design engineers to expose some of the internals of the software under test – like component interfaces – to provide more controllability and observability of internal information that passes into and out of the system through program-based interfaces.

The design for testability should occur at many or all of the layers of the software system architecture, because it results in less coupling of the system components. If the component interfaces are coupled to other components, the components are typically not completely controllable through separate interfaces. This can complicate the modeling and testing process, and blur testing predictability. Figure 2.11 shows a conceptual view of system components with well-defined and coupled interfaces.

Modern software components may be customizable through table-driven configurations enabling a single application version to address a broad range of customer needs. Table driven systems (TDSs) use a vast variety of business rules which pose a challenge for testing because the predictability of the results depend in many cases on the relationships between rules. The “Rule-to-rule associations” problem addresses a visibility issue because [GI03, S. 7]:

- A rule can be an exception to another rule
- A rule enables another rule
- A rule subsumes another rule
- A rule is semantically equivalent to another rule
- A rule is similar to another rule
- A rule is in conflict with another rule
- A rule supports another rule

The rules used in business information systems are of the form: event-condition-action (ECA). The large number of rules implemented in a productive environment makes difficult the analysis of interactions, since the execution of a rule may cause an event triggering another rule or a set of rules. These rules may in turn trigger further rules with the potential for an infinite cascade of rule firings to occur.

In relational databases (see Sect. 4.3.6), procedures can be attached that can be activated (or “triggered”) whenever a row is manipulated or changed in a table. The appropriate routine covering each condition of a rule is in relation to the corresponding table business rules.

2.5.1 *Business Rules*

The Business Rules Group (<http://www.businessrulesgroup.org>) gives two definitions of a business rule reflecting the dual views of business and technology in this matter.

From the *information system perspective*:

“A business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or control or influence the behaviour of the business.”

From the *business perspective*:

“A business rule is a directive, which is intended to influence or guide business behavior, in support of business policy that is formulated in response to an opportunity or threat.”

The aim of a business rule is to separate the business logic from data feeds, presentation layers and system interfaces of all kinds. Basically, a business rule is independent of the modeling paradigm or technical platform and is defined and owned by business entities.

The term “business rules”, however, has different meanings for business and IT, depending on whether the perspective is data-oriented, object-oriented, system-oriented, or expertise-oriented.

Teradata takes a holistic approach to enterprise data management by organizing the universe of data services into three main categories:

1. Data modelling and business views including:
 - Metadata management
 - Master data management
 - Data quality
 - Data integration
 - Data security and privacy
2. Data governance: Data governance includes the processes, policies, standards, organization, and technologies required to manage the availability, accessibility, quality, consistency, auditability, and security of the data in a enterprise.
3. Data stewardship: Data stewardship defines the continual, day-to-day activities of creating, using, and retiring data. Data stewards are responsible for formal accountability and the management of all enterprise data.

In Sect. 4.2, master data management will be presented which covers metadata, business rules, and reference data aspects.

Rule Traceability

The ability to manage experience and expertise gained in the business domains – and not documented for sure – is a major driver for rule management. The following questions should be addressed [GI03, S. 6]:

- When was the rule created?
- Where can more information about the rule be found?
- Where is the rule implemented?
- What new design deliverables need to address the rule?
- Who can answer particular questions about the rule?
- To what areas of the business does the rule apply?
- What work tasks does the rule guide?
- In what jurisdictions is the rule enforced?
- Over what period of time is the rule enforced?

It is important to establish a good process of uncovering and extracting the existing rules (implicit or explicit) that govern the operation of the business to understand where business rules fit best with business strategies. This shall insure the continuing business alignment of the project.

A new aspect adds to the urgency to (re-)discover the business rules or better manage them is regulatory compliance. The need to establish or verify business controls is the objective of the Sarbanes-Oxley Act of 2002 – best known as SOX 404. These controls are essential rules that apply to internal operations and procedures to be formally defined and captured in a central repository. The GUIDE business rules project has defined four categories of business rules:

1. Definitions of business terms,
2. Facts that relate these terms to each other,
3. Constraints on actions that should or not should take place in a particular scenario, and
4. Derivations of one type of information can be transformed into or derived from another.

The main question to answer is if all rules relevant to a business have been found. From a pragmatic point of view, it is suggested to take a formal testing or simulation approach, either manually driven with data walkthroughs, or using the simulation applications that form part of most of the BPM solutions available on the market. To improve business and systems acceptance business rules should be used as a basis for test plans.

From a technical point of view, business rules can be implemented with relational tables or/and using UML Object Constraint Language (OCL). OCL is a declarative language designed for object-oriented software. It can be used to constrain class diagrams, to specify pre- and post-conditions of methods, and to specify the guards of transition within an UML state machine, to name a few.

2.5.2 *Business Rules Management (BRM)*

Business rules constitute the part of an application most prone to change over time. New or altered regulations must be adhered to, new business methods get implemented, competitive pressures force changes in policy, new products and services are introduced, and so on. All of these require changes to the decisions that control behaviors and actions in business applications. The people best able to gauge the need for new operational behaviors, to envision the new decision criteria and responses, and to authorize implementation of new business policies are seldom technically trained in programming techniques. Policymakers want business applications designed to let them accomplish business tasks such as introducing new promotions, changing discount levels, altering rating criteria, or incorporating new regulations. But creating such modification and management applications is a task often comparable to building the underlying business systems themselves! Traditional organizational behavior is, for business policymakers, to gather a set of business changes that should be made, submit them as a formal request to a programming department, sign off the programming interpretation of the changes, and wait for a scheduling opportunity to have the changes implemented in a new software release. The delays in this type of cycle are apparent, but there is no alternative in traditionally implemented software systems. Because business rules are separated from and independent of the underlying system code that keeps a business application operating, they should be changed without impacting the application logic.

Auditability

An aspect of business rules management that should not be overlooked is the importance of keeping track of tasks carried out and decisions made. Since business rules control key decision processes, it is crucial to have clear access to what rules were in play when decisions were made, who made changes to rules, and why changes were made. Part of this process is dependent on a commitment to good change management procedures within the organization. But, the software being used to manage the rules should also provide the functionality to support enterprise control and auditing. For example, to record the state of an entire rule service at any point in time, a “snapshot” may be taken of the files containing the rules. This can be recovered, reused, or examined later for auditing and results testing/comparison against later versions. Rules management and maintenance applications should also add documentary information to rule creation, and changes showing the author, the time, and the date of the change. [GI03, S. 14]

Implementing BRM

For all the reasons mentioned before, BRM should be part of the company’s knowledge infrastructure. For testing activities, BRM plays a central role by driv-

ing the extraction process for test data generation, as described later in Sect. 4.2. The rules discovery process is an integral part of BRM and is usually owned by an expert committee with representatives of the business and technical domains. Defining, modifying, and enhancing rules address primary business aspects impacting IT operations as well. For this reason, a permanent BRM team involving domain experts, business managers, analysts, process designers and IT experts has to be established to steer the BRM process successfully.

From the technological perspective, a dedicated BRM framework is also required to cover the manual and automated elements of the process and sub-processes. Some tools are available which can extract rules from the business logic that is either defined within programs or hard-wired in legacy systems. Figure 2.12

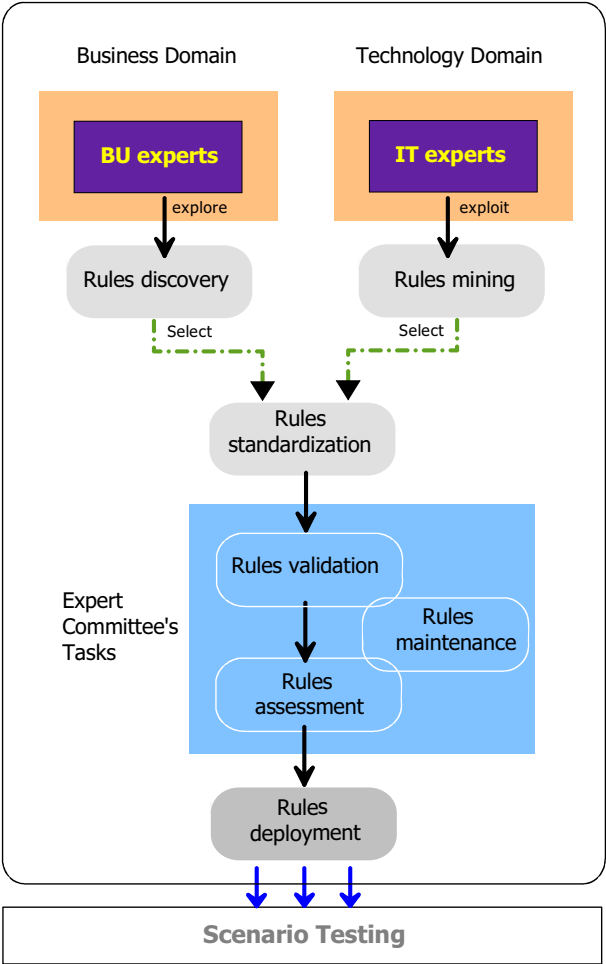


Fig. 2.12 Business rules management

shows the layout of a BRM framework with business and IT actors working together on specific tasks.

A rule engine will provide the necessary functionality to manage the rules and their associations, and to support the different tasks inside the BRM framework as depicted in the diagram.

The Rule Engine – Specific Requirements

The Rule engine component requires a number of capabilities for the rules management system to function properly. There are at least six major ones in a complete system, including:

- Support refining existing object/data models
- Support interoperation with other data sources
- Manage simultaneous requests for decisions from different applications
- Perform condition-based selection of the right rules to fire, in the right order
- Integrate with auditing software to record what rules were used in the course of making a decision
- Offer support for complex decision chains without needing explicit control from the calling application

The Rules Management GUI – Specific Requirements

The Rules Management GUI requires a number of capabilities for the Rules Management System to function properly. There are at least three major ones in a complete system, including:

- Supporting the management of the rule set and service partitioning (role-based access), service assembly, and deployment
- Supporting the execution path testing
- Supporting generation rule maintenance applications that completely hide the structured rule language – the applications that allow rules to be safely constructed and modified by the right people with the right data entry controls

Rules Management Integration Specifications

For effective use of a rules management system, business logic should be independent from the mechanisms used to manipulate data or implement decisions. Rules should be implemented independent of any external system that may use it. For example, in a process that requires a mechanical task, e. g., lighting a bulb, the rules management system should not include facilities to physically control input and output. Instead, it should contain clearly defined integration points for systems that are built to accomplish these mechanical tasks. [GI03, S. 27]

2.5.3 *Software Reliability*

The NASA Software Assurance Standard, NASA-STD-8739.8, defines software reliability as a discipline of software assurance that:

- Defines the requirements for software controlled system fault or failure detection, isolation, and recovery
- Reviews the software development processes and products for software error prevention and/or reduced functionality states
- Defines the process for measuring and analyzing defects, and derives the reliability and maintainability factors.

The level of quality of the software which results in a controlled and predictable behavior is the result of reliability techniques developed in the past twenty years. Reliability techniques are divided in two categories:

- Trending reliability: This tracks the failure data produced by the software to determine the reliability operational profile of the application or system over a specified period of time.
- Predictive reliability: This assigns probabilities to the operational profile of a software solution expressed in a percentage of failure over a number of hours or days.

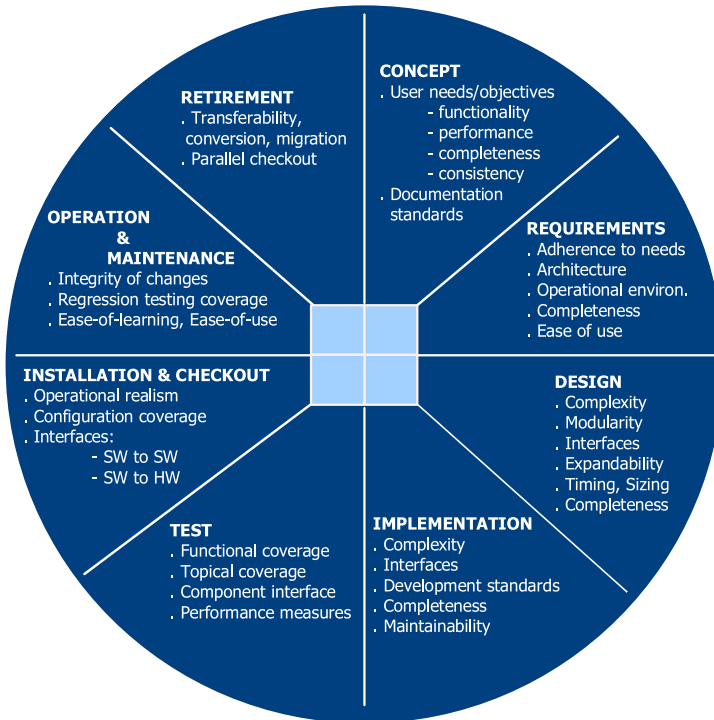


Fig 2.13 IEEE software reliability diagram (Source: IEEE Std.982.2-1988)

Software reliability is a quality characteristic which quantifies the operational profile of a system and tells the testers how much additional testing is needed. In general, software reliability failures are caused by incorrect logic, incorrect statements, or incorrect input data. Predictability in testing can be mastered reasonably well if the following conditions are fulfilled:

- A good knowledge of the use cases
- A good knowledge of the business rules to be applied
- A good understanding of the test data required
- The availability of a test library for automated testing

But, first of all, testing must be deterministic: that means, that to design a test case you must know the state of the data before starting the test suite and to be able to predict anything at the end. Don't forget to check the initial state of the test system and the database; reinstate them if needed and document the procedure before starting further test activities.

The IEEE diagram presented in Fig. 2.13 shows the components contributing to overall software reliability.

2.5.4 Software Quality Criteria [ISO 9126]

The software under test has to work as contractually specified with the customer or stake holder, providing key characteristics as defined by ISO 9126. These are as follows:

1. Efficiency: A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under the following stated conditions:
 - Throughput
 - Time behavior
 - Optimized resources consumption
2. Functionality: A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs, including:
 - Accuracy
 - Compliance
 - Correctness of the results
 - Predictable behavior
 - Good interoperability
 - Good scalability
 - Good security
 - Interoperability

3. Maintainability: A set of attributes that bear on the effort needed to make specified modifications, including:
 - Analyzability
 - Changeability
 - Modifiability
 - Stability
 - Testability
 - Understandability
 - Verifiability
4. Portability: A set of attributes that bear on the ability of software to be transferred from one environment to another, including the following:
 - Adaptability
 - Conformity
 - Installability
 - Replaceability
 - Serviceability
5. Reliability: A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time, including the following:
 - Fault tolerance
 - Maturity
 - Recoverability
6. Usability: A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users, including the following:
 - Learnability
 - Understandability
 - Operability

Very often, the full range of criteria can't be fully and timely satisfied, because some software components have different development schedules or tight integration windows. The test intensity needs to be modulated accordingly to take into account such contingencies.

2.6 Software Development Methods

In this chapter we examine the pros and cons of the most popular software development methods, focusing on their usability in large-scale projects. We assumed that most readers know the basics of phase-oriented software development.

2.6.1 V-Model

The V-model is used in the vast majority of large projects as the methodology of choice to develop and test complex systems and software solutions. The V-model is organized in three main blocks including development, solution build and integration. The development cycle is the main engineering process which covers:

- The system design review
- The product design review
- The critical design review

A continuous requirements tracking process should be implemented to verify and validate the correctness of design compared to original customer’s requirements. The next cycle is the solution build which is characterized by three processes:

- The pilot design
- The pilot implementation
- The pilot verification

The final cycle is the integration process including the following:

- The unit test (UT)
- The component test (CT)
- The integration test (IT)

The integration test must guarantee that the functionality of the new software work correctly end-to-end, without any problems. Figure 2.14 gives a top view of the V-model.

We will see later in this chapter that the V-model used in the real-world includes refinement loops providing flexibility and continuous improvement of the work in progress.

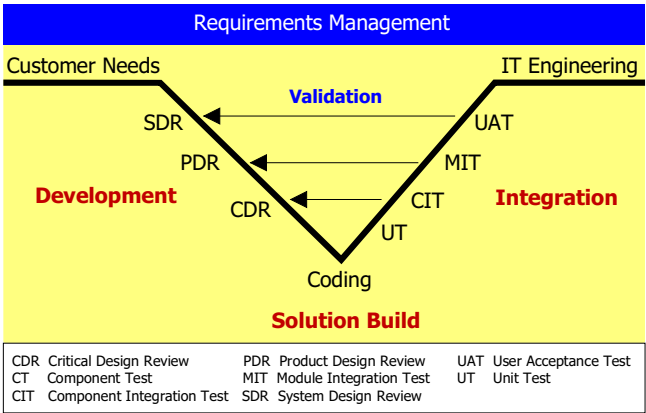


Fig. 2.14 V-Model

The V-model is a requirements-oriented approach which requires discipline, a high degree of coordination, and strong communication skills from all teams involved. Since documents define the content of the project and its final product, it is important for all parties involved to work with accurate document artifacts. The documentation process is therefore of great importance to support the team-work of analysts, developers, testers, and project supervisors.

Activities Carried Out in the V-model Framework

Along the waterfall life cycle, the development team does the following:

1. Starts by researching and writing the product's requirements
2. Writes the high-level design documents
3. Writes low-level design documents
4. Writes code.

The test team does the following:

1. Writes its acceptance tests as soon as the requirements are reviewed
2. Writes its system tests in response to the high-level design
3. Writes integration tests in response to detailed design
4. Writes unit tests in response to the coding
5. Runs the system tests.

The benefits of this planned method are reasonably well-defined requirements, well-thought-out test cases and test sets based on scenarios for all test environments.

This methodological approach works exceptionally well in situations where the goal of the testing is to determine service-level agreement (SLA) compliance, or when independent validation and verification (IV&V) is required. It is also easier to implement GUI-level automated tests because, in theory, the user interface is locked down relatively early. The waterfall cycle includes little incremental, iterative, or parallel development.

A drawback of this model is that the project team has to look very closely at detail level before costs, complexity, dependencies between software components, and implementation difficulty of those details are known. The project manager has to deliver a software solution that works reasonably well, on time, and within budget. Under the waterfall, the features are designed up front, funding and time are spent on designing and developing all of the features, until finally, the code comes into system testing. If the code is late, or the product has a lot of bugs, the typical trade-off between reliability and time to market has to be solved pragmatically. The testers need more time to test the product and get it fixed, while stakeholders want to deliver the product before it runs too far behind schedule.

2.6.2 Agile Software Development

“Faced with the conflicting pressures of accelerated product development and users who demand that increasingly vital systems be made ever more dependable, software development has been thrown into turmoil. Traditionalists advocate using extensive planning, codified processes, and rigorous reuse to make development an efficient and predictable activity that gradually matures toward perfection. Meanwhile, a new generation of developers cites the crushing weight of corporate bureaucracy, the rapid pace of information technology change, and the dehumanizing effects of detailed plan-driven development as cause for revolution.” [Barry Boehm, USC].

New initiatives to improve productivity and reduce the latent time between idea and software delivery are emerging periodically. They tend to avoid formalism and reduce drastically planning tasks to enable:

- A shorter development time (feature-driven)
- A strong interaction with the customer
- Maximum iterative cycles.

Adopting agile development could add potential risks in a project because important aspects of the software production are not properly addressed with this method:

- Minimal software documentation
- The poor auditability of testing activities in this context
- Integration problems with CMM-developed components
- A difficult synchronization of the development process.

This approach works best for small projects (up to 10 people) but don't really accelerate the software production in large-scale projects which are risk-managed and plan-driven. The following diagram illustrates the positioning of the agile approach compared to well-established CMM-development methods:

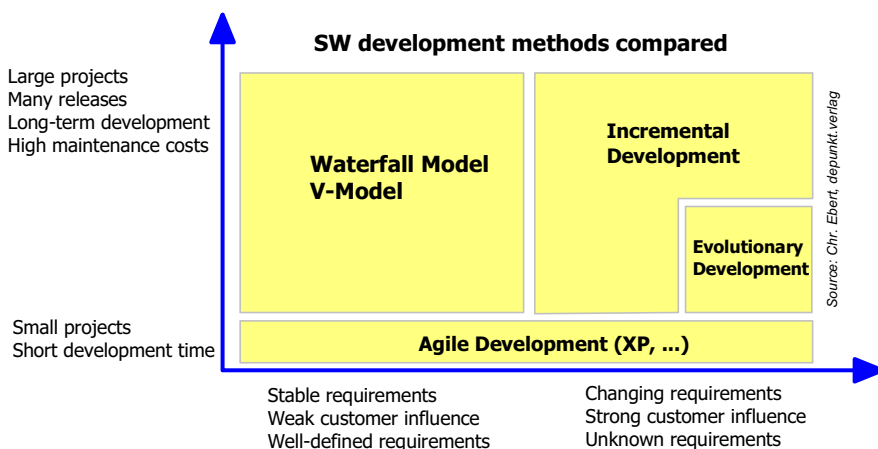


Fig. 2.15 Software development methods compared

2.6.3 What is Agility?

Agility means being able to adapt to environmental shifts quickly by changing products, services, prices, and business processes. From an IT viewpoint, it means being able to reconfigure and extend application systems quickly, without rewriting much code or changing the databases. It also means implementing multi-channel, or “channel neutral” applications that support desktop, mobile, and back-office work simultaneously.

The agility of company information systems cannot be achieved by merely accelerating systems built on traditional architectural principles. Agility must be built into the core of business processes and their information systems. (Source: Gartner Group)

The agile or “extreme programming” method (XP) features a modular, people-centric approach – as opposed to process-centric – software development and testing approach. With agile programming, feedback was the primary control mechanism, meaning that a fledgling design progressed through the steps of development only after tests and modifications were performed at each possible juncture. Proponents of agile programming argued that software development existed *entirely* in the design/development stage, and that the manufacture step was nothing more than insignificant compiling of code, an afterthought. (Source: Wipro Technologies)

Software development and testing activities must be harmonized at the methodological level to best address the main business requirements: faster product delivery at reduced (production) costs. Large organizations and many of those that are undertaking massive reengineering projects, facing high quality and safety requirements use traditional methods (e. g., CMM). This is particularly the case in the finance industry.

Based on my experience in large software projects, I can attest that agile methods are used in a V-model driven context, for clearly identified software components. The contribution of agile development and testing was a valuable one for components like business monitoring tools, DWH queries, and web services prototypes. The ability to be agile involves optimized use of established technologies, but more important, working with established processes and procedures which can be enriched with the positive feedback of refinement loops.

The important point to remember with agile methods is to reduce the uncertainty level of software maturity as quick as possible.

The frequency of iterations in agile mode is dependent on the kind of problems encountered in the development phase, such as:

- An inadequate design
- Incomplete/unclear requirements
- Technical gaps
- Emerging user’s requirements (on-the-fly)
- Lateral or backward compatibility problems
- Other causes not precisely identified

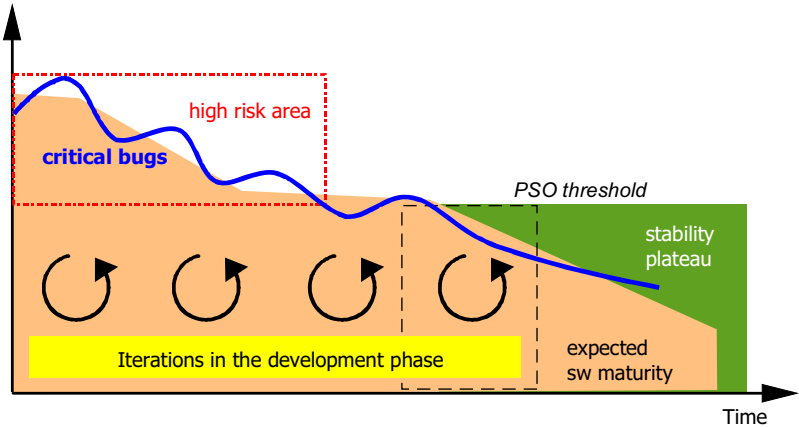


Fig. 2.16 Uncertainty reduction in agile development

Figure 2.16 shows the relation between development risk and uncertainty level. The iterative approach contributes to reduce significantly the failure rate in the software production and deployment processes, improving agility. Figure 2.17 shows that up-front agile development integrates existing/modified requirements and new features continuously via iterations up to software delivery. The at-end agile development provides production documentation, re-factoring, and training during the last iteration. Extended requirements – like SOX-404 and other legal requirements – are fulfilled after the last iteration, before PSO.

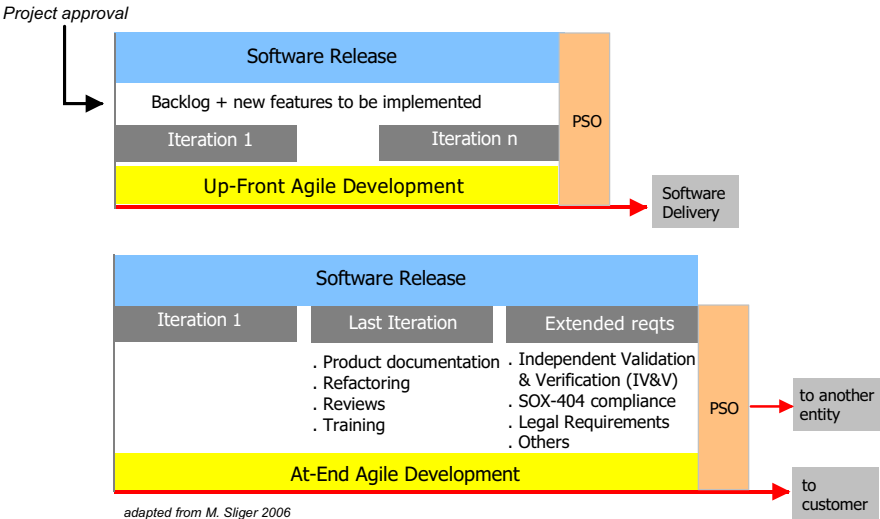


Fig. 2.17 Agile front-end

2.6.4 Iterative Development

In an evolutionary model like XP, programming follows an evolutionary model: each feature is tested as soon as it is added to the product, bringing it up to an acceptable quality before adding the next feature. In the evolutionary case, the tradeoff lies between features and time to market, rather than between reliability and time to market. Testers play a much less central role in the late-project controversies because the issue is not whether the product works (it does) but instead, whether there's enough product. The iterative approaches (spiral, RUP, XP, evolutionary development, etc.) are reactions to the risks associated with premature misspecification.

2.6.5 Waterfall and Agile Methods Compared

The most preeminent differences of the two methods are reflected in Fig. 2.18. The teams working cooperatively in an agile mode need to synchronize their activities and compare the development progress periodically with the core team using the waterfall method.

Agile teams rely on four levels of testing while developing software:

1. Developer testing
2. Story testing
3. Exploratory testing
4. User acceptance testing

These layers include some redundancy but each tends to reach a higher quality from a different perspective. In story testing for example, small slices of user-visible functionality can be analyzed, designed, coded and testing in very short time intervals (days), which are then used as the vehicle of planning, communication and code refinement.

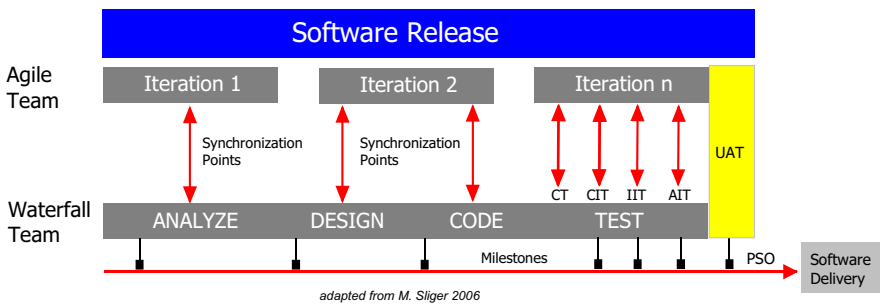


Fig. 2.18 Waterfall-agile comparison

2.6.6 Staged Delivery Method

In a staged delivery model, features are added incrementally, but the product is delivered to the customer after each “stage”. Requirements and architectural design are typically done up front and planned releases are made as each set of features is finished. This is an incremental method which can be used in large V-model projects, but more as a back-fall procedure. The solution manager can justify a staged delivery of the product to the stakeholders to overcome temporary resources shortage or to better control the rollout of releases in multiple geographical locations. In this case we speak about “wave rollout”. The staged delivery method is shown in Fig. 2.19.

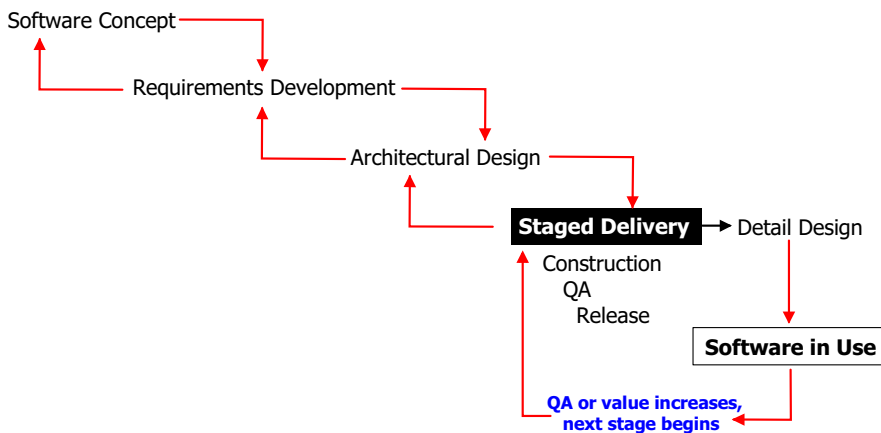


Fig. 2.19 Staged delivery method

2.6.7 Selection of the Right Development Method

Developing and testing software in a complex IT environment requires many ingredients, including well-documented artifacts and a comprehensive methodological framework. The waterfall model is ideally suited to fulfill tight requirements to produce high-quality results.

The waterfall model is most suitable to your needs if:

- The requirements are known in advance
- The requirements have no high-risk implications
- The requirements have no uncertainty or hidden risks
- The requirements satisfy stakeholders' expectations
- The requirements will be stable during the development phase
- The design architecture is viable and approved
- The sequential approach fits into the project's time frame.

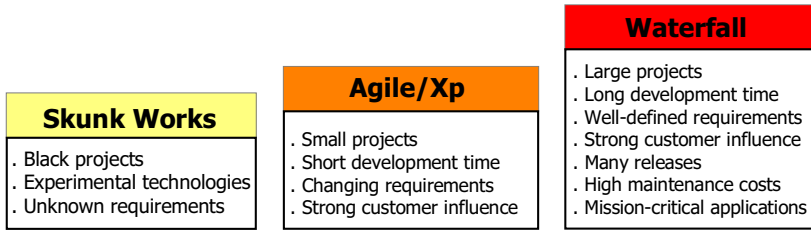


Fig. 2.20 Software development methodologies compared

The evolutionary approach is best for your project if:

- The initial release is good enough to keep stakeholders involved
 - The architecture is scalable to accommodate future system growth
 - The operational user organization can adapt to the pace of evolution
 - The evolution dimensions are compatible with legacy system replacement.
- (Source: ESEG, UMD)

The diagram shown in Fig. 2.20 reflects the positioning of the three software development approaches discussed here.

Large-scale projects can adopt agile development for some parts of the software to be manufactured, but some factors (e.g., the size of the development teams, the cultural environment, and communication and coordination issues) can be considered as insurmountable obstacles.

However, examining the V-model as used in a real-world context gives interesting clues about agility aspects in this rigid methodological framework. Refinement loops is the distinctive characteristic of an agile V-model.

The first loop is the continuous requirements tracking which covers four distinct areas:

- Analysis
- Validation and verification
- Architecture design
- Architecture verification

Agility begins with the analysis of requirements and continuous feedback to originators insuring more accurate deliverables produced in development and testing:

- Scenarios must be created, enhanced and validated
- Functions and limitations are worked out
- Operating and maintenance instructions must be adapted and documented

In the architectural design phase, additional functions and limitations can be added with corresponding documentation.

This second loop addresses even more the flexibility issue through prototyping:

- Pilot design
- Pilot implementation
- Pilot verification

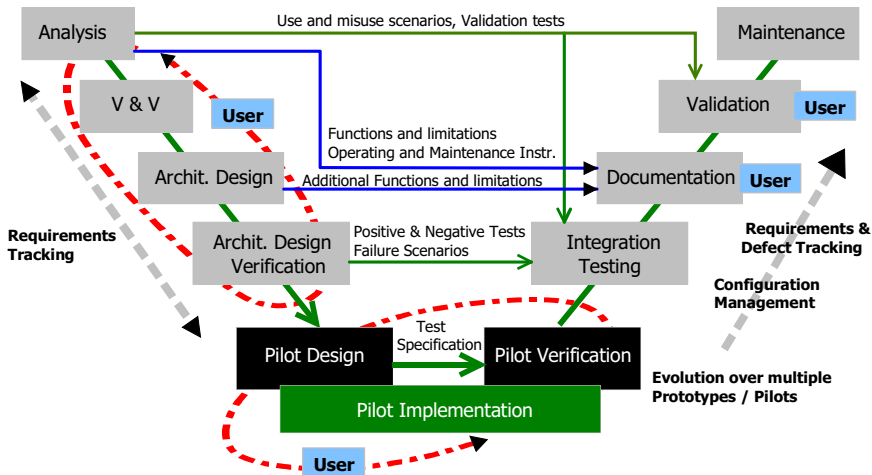


Fig. 2.21 V-model in practice

In large-scale projects, prototyping is chosen to produce a number of prototypes either for critical software components with new functionality and new design, or for end-user tools and Web interfaces.

The test specification in this phase needs to be adapted permanently to reflect the evolution of the product under development. During the architectural design verification phase, positive and negative tests and failure scenarios are elaborated for use later in the integration phase.

Scaling Agile Methods

The question if eXtreme programming works for large projects has been answered positively so far. Sanjay Murthi, President of SMGlobal Inc., reported about his experience in a project involving two development centers: “I had the opportunity to use eXtreme programming (XP) on a large software product release. The team consisted of more than fifty people located in two different development centers. At first, I was skeptical that XP could work for us. I knew that, historically, XP has been most successful with small and very committed teams. By placing importance on regular builds and working systems, we frequently caught problems early. By maintaining good version control, we also had confidence that we could easily roll back to a previous build if things went bad. Unit tests helped ensure that new functionality didn’t break the old. Overall, integration became easier and less worrisome. Working with requirements that weren’t yet fully defined made it possible to start work faster, fleshing out the requirements as we moved along. Regular demonstrations of the system as it was being built helped satisfy stakeholders that their needs were being met.

However, managers still have to weigh several factors before deciding whether to use agile methods. Their success depends on three major factors: the skills and enthusiasm of the people involved, the processes used and level of adherence to them; and lastly, the management and reporting systems in place.

Inadequate communication can disrupt any project and cause any methodology to fail. If teams don't communicate well with each other and with other teams, their skills and level of enthusiasm don't matter. They will end up creating poorly designed or just plain bad solutions.

When should managers consider agile methods, and how should they start using them? Traditional development methodologies can help deliver projects on time and budget, and give management a good handle on project status throughout. However, if you're moving to a new technology platform or your project calls for fluid requirements, these older methods aren't suitable."

Traditional methods assume fairly static requirements and impose high cost burdens because of the high volume of processes and people needed to complete a task. They focus on intermediate deliverables, which can become wasted work should project goals and requirements change. You can define projects by four factors: scope, time, resources, and risks. Traditional methodologies work well when all of these factors are quantified and understood.

Getting all of your teams to use similar techniques is important, particularly when it comes to time and cost estimation. Bad estimation can lead to poor distribution of work. (Source: Dr. Dobbs, 2002)

2.7 The Testing Value Chain (TVC)

The testing value chain is one of the most important instruments to plan and monitor all test activities in order to achieve production sign-off (PSO) in the terms agreed by all parties. The TVC begins with the base-lining and ends with the roll-out of the product.

Each test phase in the value chain is validated by an SO process which is binding for all contractual parties involved in the project. There are four distinct SOs, as listed in Fig. 2.22.

Initialization phase	➔	Business case sign-off (BSO)
Concept phase	➔	Requirements sign-off (RSO)
Development phase	➔	Specification sign-off (SSO)
Rollout phase	➔	Production sign-off (PSO)

Fig. 2.22 An SO agreement is required for each testphase completed

2.7.1 The SO Organization

A sign-off office coordinates all activities related to the SO process, and it is responsible to collect and valid all deliverables in a timely fashion. The office tracks the progress of work and organizes reporting meetings in accordance to the overall project plan. Participants to the SO process are: solution managers, project leader and team representatives of IT security, Software Development, Application Architecture, Data Standards, SW Integration, IT, and Operations and Testing. Figure 2.23 shows the software development sequences and the SOgates.

Each phase in the TVC must starts with a situation clearly documented and properly communicated to all project members. The term *baseline* applies to each set of commitments, which signifies to the parties involved in the project the progress of work through the passage of time. Each baseline includes deliverables – like documentation and software artifacts – being reviewed and delivered at defined points in time (milestones). For example, before starting CT or UT work, the logical and physical models must be available. Along the TVC chain, the integration and test phases are validated using entry and exit criteria, assessing the quality of the work produced (the deliverables).

In the real-world environmental factors (e.g., business pressure) are the main causes of lowering the barriers to satisfy the TVC's criteria at a low price, generating inevitably quality problems in all subsequent phases. Figure 2.24 shows the testing value chain with test phases in relation to the software development process.

Before starting the test activities in a given test environment, the following items must be identified, provided and agreed upon:

- An approved release baseline
- Test objectives
- The required test data
- The Responsibilities of all roles involved in the project.

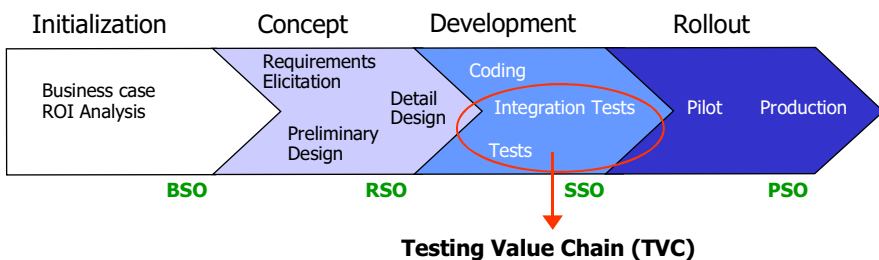


Fig. 2.23 Testing value chain gates

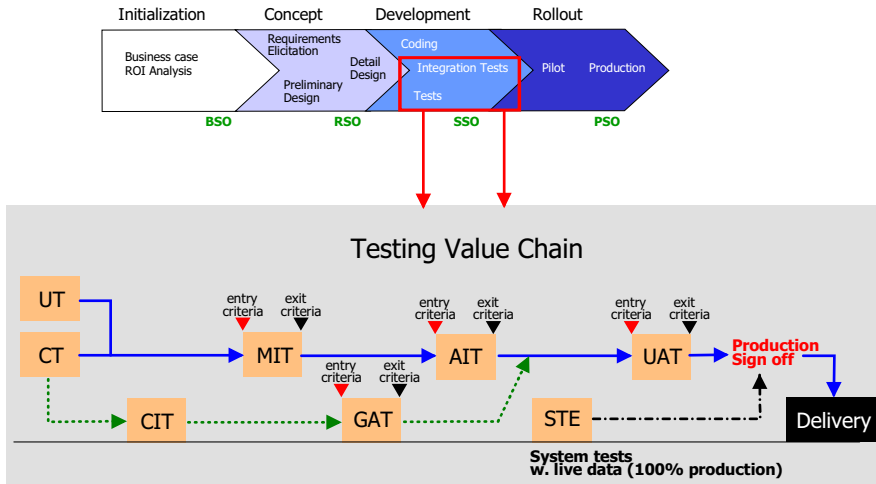


Fig. 2.24 Testing value chain phases

2.7.2 Quality Gates

Along the TVC, mandatory check points should be defined to assure adherence to processes and quality standards. These are called *quality gates* or *Q-gates*.

Q-gates apply these simple rules:

- Minimum criteria definition for entering the next phase
- Q-gates do not replace any QA activities in the project
- Q-gates shall be verified by a project-independent function
- Results should be monitored by line management

Introducing the concept of Q-gates will give a burst of improvement to the organization in charge of the software production, provided that following recommendations are followed:

1. Establish management commitment on standardized QA criteria for entering test phases (Q-gates) controlled by a project-independent QA responsible
2. Establish Q-gates
 - Define quality criteria and expected output artifacts for each project phase as criteria for entering the next phase
 - Provide documentation of processes with roles and responsibilities, expected output documents and quality criteria
 - Provide adequate training after the process rollout
 - Establish q-gate reviews for important project releases by an independent QA function reporting to management.

Q-Gates Identification

Q-gates are named according to the phases described previously:

- QG1: Business Case Signoff
- QG2: Software Architecture Signoff
- QG3: Software Requirements Signoff
- QG4: Software Design Signoff
- QG5: Handover to Test
- QG6: Handover to Acceptance
- QG7: Acceptance
- QG8: Production Signoff

Benefits of Q-Gates

If applied correctly, Q-gates offer many advantages:

- Less rework in late project phases
- Objective criteria on project status
- Problems are solved in the phase where they were produced
- Less consulting effort on process rollouts
- QA remains in the project's responsibility
- Adherence to QA standards is checked with reasonable effort.

Source: SQS

<http://www.springer.com/978-3-540-78503-3>

The Testing Network
An Integral Approach to Test Activities in Large
Software Projects

Henry, J.-J.P.

2008, XII, 438 p. 180 illus., Hardcover

ISBN: 978-3-540-78503-3