Design of a Methodology to Support Software Release Decisions

Sassenburg, J.A.

1 INTRODUCTION

"Complexity is the business we are in and complexity is what limits us."
-- Frederick P. Brooks --

1.1 Software Releasing: A Problematic Area

Software is a major worldwide industry and demand is increasing exponentially (Hoch *et al.* 1999; Boehm and Sullivan 2000; Littlewood and Strigini 2000; Fuggetta 2000; Humphrey 2002). Software pervades a multitude of products, in social, business and military human-machine systems. It includes information [technology] systems, developed for gathering, processing, storing, retrieval and manipulation of information, to meet organizational needs, and commercially-developed software products, sold to one, or more, customers or end-users. Software offering functionality to an end-user is also called *application software*, and might be a product on its own, or might be embedded in a larger system or machine. Figuratively speaking, application software sits on top of *system software*, which are low-level programs, such as operating systems and utilities for managing computer resources, that interact with the computer at a basic level (IEEE 1990; Humphrey 2001).

The increasing demand for application software brings with it an increase in our *dependence on software* (Littlewood and Strigini 2000):

- Software-based systems replace older technologies in safety or mission-critical applications.
- Software moves from an auxiliary to a primary role in providing critical services.
- Software becomes the only way of performing some function; not perceived as critical but whose failure would deeply affect individuals or groups.
- Software-provided services increasingly become an accepted part of everyday life, without any special scrutiny.
- Software-based systems increasingly integrate and interact, often without effective human control.

Despite the exponential increase in the demand for software and the increase in our dependence on software, many software manufacturers behave in an unpredictable manner (Boehm and Sullivan 2000; Standish Group 2001, 2004). In such an unpredictable software manufacturer organization, it is difficult to determine when a software product will be released, the features the product will have, the associated development costs or the resulting product quality. Without knowing when a software product will be released, a software manufacturer experiences difficulty in planning activities, such as product promotions, customer training and maintenance support. Resource utilization across projects may become inefficient, and difficult to manage, when projects fail to meet schedules. Customers have difficulties in planning for the introduction of a new software product into their organizations when a scheduled release date is missed.

There are many [indefinite] points of evaluation along the life-cycle of a software product. The various milestones in between the life-cycle stages in particular, draw the attention of researchers and practitioners in the Information Systems (IS) and Software Engineering (SE) disciplines. Important milestones are the upfront investment appraisal, the implementation or release decision, and disinvestment in an operational software product, as in Figure 1-1 (Sassenburg and Berghout 2004).

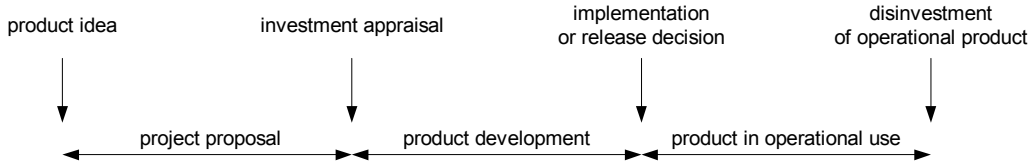

Figure 1-1: Important Milestones in the Life-cycle of a Product
(Sassenburg and Berghout 2004)

This thesis is about release decisions for software products; a crucial point of evaluation. For many software manufacturers, especially those operating in mass markets, this is the point of no return.

A software release decision is often a trade-off between early release to capture the benefits of an earlier market introduction [a larger installed base], and the deferral of product release to enhance functionality, or improve quality. At first sight, this trade-off seems not to be of any special nature, from a strictly economic perspective. If a software product is released 'too early', a software product with less functionality and/or significant defects would be released to intended users and the software manufacturer incurs post-release costs of later fixing failures. If a software product is released 'too late', the additional development cost, and the opportunity cost, of missing a market window could be substantial. These two alternatives need to be compared, to determine which alternative maximizes economic value [revenues minus costs].

In a practical setting, the decision to release a software product can be a problem, best illustrated with examples:

- In practice, cost and time constraints will normally be present in retrieving complete and reliable information. This search for information should be taken into account as an economic activity with associated costs and time. This leaves the software manufacturer with the problem of finding the optimal level of information, where marginal value equals marginal costs and thus marginal yield is zero. Gigerenzer (2004) holds this optimal level is difficult, if not impossible, to find.
- Decision-making in the real world is often unstructured (March and Olsen 1979), and normally involves various stakeholders, and there might, for example, be reasons to release a system or software product, due to political or business pressures, even though knowing it still contains defects. A study of spacecraft accidents, for example, reveals that, although inadequate system and software engineering occurred, management and organizational factors played a significant role, including the diffusion of responsibility and authority, limited communication channels and poor information flows (Leveson 2004).
- Research has revealed there are many obstacles to the successful implementation of almost any decision (March and Olsen 1979), including:
  - The reduced importance of a decision once it is made and implemented.
  - The control of the outcome of a decision by stakeholders not involved in its making.
  - The development of new situations and problems to command the attention of the decision-makers once the choice has been implemented.

It is likely these obstacles apply to software release decisions. When political, or business, pressure to release a software product is high, the stakeholders responsible for maintaining the product may be neglected. In more chaotic environments, the attention of decision-makers to the decision implementation is likely to decrease due to other problems and/or the arrival of new problems.

It is concluded here that the trade-off issue 'release now or later' is not always easy to solve. Given the multitude of factors influencing this decision, each software release decision is likely to be unique. The author is however unaware of any studies conducted in this problem area; addressing software release decisions from both an economic and a decision-making point of view. In this study, the dimensions of the problem of deciding when to release a software product are investigated, using the existing body of knowledge, complemented by a study of current practices in software manufacturer organizations. The results obtained, combined with a multitude of theories from different disciplines, are used to propose a methodology to improve software release decisions.

In Section 1.2 the increasing impact of software on society is discussed. In Section 1.3, attention is paid to the fact that both the complexity and importance of software release decisions are likely to increase in the next years or even decades. The extent to which existing methodologies, models and standards provide guidance in structuring software release decisions is discussed in Section 1.4. The argument for conducting this study is discussed in Section 1.5, addressing the benefits of a formal decision-making process. This leads to the definition of the primary research question in Section 1.6, the definition of the scope of this study in Section 1.7, the conclusions in Section 1.8, and the outline of this thesis in Section 1.9.

1.2 Increasing Impact of Software

Carr (2003) criticizes today's worship of information technology and software. Although he recognizes that software has deeply transformed today's society, he claims that it no longer offers a competitive advantage to organizations. Using examples of electric power production and trains he shows that software, as a proprietary technology, will no longer generate a competitive advantage. Instead, it can be expected that the usage of software will become standardized and no organization will benefit from purchasing and maintaining its own software solutions. Although this paradigm shift might be true, it does not imply that the production and consumption of software solutions will decrease. Taking the example of electric power production, used by Carr, it is obvious that, even after standardization instead of proprietary solutions, production and consumption of electric power has increased exponentially. The same holds for software, nowadays spreading from computers to, for example, the engines of automobiles to robots in factories to X-ray machines in hospitals.

1.2.1 Software Size

It is often reported that the size of software products is growing at an exponential rate following *Moore's Law* (Yang 1998; Humphrey 2002; STW Technology Foundation 2002; Asseldonk 2004; Broy 2004). The observation, made in 1965 by Gordon Moore, co-founder of Intel, noted that the number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented. Moore predicted this trend would continue for the foreseeable future. In subsequent years, the pace slowed a bit, but data density has doubled approximately every 18 months, and this is the current definition of Moore's Law. Expressed as 'a doubling every 18 months' or 'ten times every 5 years', Moore's law is applied in other technology-related disciplines suggesting the phenomenal progress of technology development in recent years. Expressed on a shorter timescale, Moore's law equates to an average growth of over 1% a week.

Software size can be described using different attributes. Fenton and Pfleeger (1997) use three attributes:

- *Length;* the physical size of the product. In general, the code length is the easiest to measure and is normally expressed in terms of lines of code.

- *Functionality;* amount of functionality delivered to end-users [in function or object points].
- *Complexity*. This attribute is interpreted in different ways: problem complexity [complexity of the underlying problem]; algorithmic complexity [complexity of the algorithm implemented to solve the problem]; structural complexity [structure of the software]; and cognitive complexity [ease of understanding the software].

It is assumed here that increasing the length, or functionality, of software will, in general, also lead to an increase of complexity, whatever perspective is taken (Rosenberg and Hyatt 1996; Solingen and Berghout 1999). According to Broy (2003), modern automobiles contain up to 10 million lines of software, distributed across 80 separate controllers and 5 bus systems. According to Yang (1998), the size of a typical embedded system is estimated to be 32 million lines of software in the year 2010.

1.2.2 Defect Potentials, Removal Efficiencies and Defect Densities

In software, the narrowest sense of product quality is commonly recognized as a lack of defects or 'bugs' in the product (Kan 2003). Using this viewpoint, or scope,[1] three important measures of software quality are:

- *Defect potential*, defined as the number of injected defects[2] in software systems, per size attribute.
- *Defect removal efficiency*, defined as the percentage of injected defects found and removed before releasing the software to intended users.
- *Defect density*, defined as the number of released defects in the software, per size attribute.

Size [function points]	Defect Potential [development]	Defect Removal Efficiency	Defect Density [released]
	Defect potential and density expressed in terms of defects per function point		
1	1.85	95%	0.09
10	2.45	92%	0.20
100	3.68	90%	0.37
1000	5.00	85%	0.75
10000	7.60	78%	1.67
100000	9.55	75%	2.39
Average	**5.02**	**86%**	**0.91**

Figure 1-2: Software Size *versus* Defect Potential, Removal Efficiency, and Density
(Jones 2002)

A study by Jones (2002) reveals that the defect potential, the defect removal efficiency, and the defect density at release time depend on the software size. From Figure 1-2 it follows that as software grows, defect potential increases and defect removal efficiencies decrease. The defect density at release time increases, and more defects are released to the end-user(s) of the

[1] This scope focuses on the reliability of the product [e.g., number of failures per *n* hours of operation, mean time to failure, or the probability of failure-free operation in a specified time]. The ISO/IEC 9126 (ISO 2001b) standard takes a broader scope and includes all non-functional requirements, being product properties that place constraints on the functional requirements. See also Section 4.3.2. The narrowed scope here is only used to demonstrate the presence of a software crisis in Section 1.3.3.
[2] The term 'injecting defects' was first used by Humphrey (1997, p.139), meaning the introduction of defects in a software program as a result of errors or mistakes made by software engineers.

software product. Larger software size increases the complexity of software and thereby the likelihood that more defects [both absolute and relative] will be injected. For testing, a larger software size has two consequences (Humphrey 2001):

- The number of tests required achieving a given level of test coverage increases exponentially with software size.
- The time to find and remove a defect first increases linearly and then grows exponentially with software size.

As software size grows, to just maintain existing levels of released defect densities, software manufacturers would have to exponentially improve their defect potentials and removal efficiencies. The situation could be even worse. When software grows, more functionality is offered to the end-users. Assuming they use the enhanced functionality, end-users are exposed to even more defects. Maintaining the existing level of released defect density is not enough and this should be further decreased.

Combining these results leads to the following two conclusions:

1. The size and complexity of software and the amount and variety of software products are growing exponentially.
2. The increased dependence of society on software means end-users will be exposed to more defects when software manufacturers are unable to exponentially improve their defect potentials and removal efficiencies.

Date	Casualties	Detail
2003	3	Software failure contributes to power outage across the North-eastern U.S. and Canada.
2001	5	Panamanian cancer patients die following overdoses of radiation, determined by faulty use of software.
2000	4	Crash of a Marine Corps Osprey tilt-rotor aircraft partially blamed on 'software anomaly'.
1997	225	Radar that could have prevented Korean jet crash hobbled by software problem.
1997	1	Software-logic error causes infusion pump to deliver lethal dose of morphine sulphate.
1995	159	American Airlines jet, descending into Cali, Colombia, crashes into a mountain. A report from Aeronautical Civil of the Republic of Colombia found that the software presented insufficient and conflicting information to the pilots, who got lost.
1991	28	Software problem prevents Patriot missile battery from picking up SCUD missile, which hits U.S. Army barracks in Saudi Arabia.

Figure 1-3: Examples of fatal Software-related Accidents
(Gage and McCormick 2004)

There is an increasing volume of evidence illustrating these conclusions, and an increasing number of software-related accidents are being reported. Gage and McGormick (2004) have published an overview of some known fatal software-related accidents as illustrated in Figure 1-3.

Leveson (1994) published a collection of well-researched accidents along with brief descriptions of industry-specific approaches to safety. Accidents are described in the fields of

medical devices,[3] aerospace, the chemical industry and nuclear power. Glass (1998) describes other accidents. Studying software release decisions has a social relevance due to the increasing impact of software on society. This decision is a crucial evaluation point, often irreversible once it has been made.

1.3 Improvement Areas

Do the conclusions in the previous Section mean there is still a software crisis? One could claim the software crisis is dead, in the sense that the term 'crisis' refers to a turning point and we have passed this point (Putnam and Myers 1997). In this case, a valid question is whether there has indeed been a turning point. To answer this question, the gains in software productivity and the most important improvement strategies of the last decades are first discussed.

1.3.1 Software Productivity

Software productivity is often expressed as a ratio of the amount of source code statements produced for some unit of time, such as lines of code per hour. Scacchi (1995) claims however that due to the number and diversity of variables influencing software productivity this is an oversimplification. In the same survey, he further concludes existing software productivity measurement studies are fundamentally inadequate, and potentially misleading. Combining different studies, Scacchi (1995) identifies a list of productivity drivers related to the development environment, the product and the project staff. Putnam and Myers (1992) follow this approach and define the so-called *Productivity Index*, combining different factors and making the argument that it is objective, measurable and capable of being compared on a numeric scale. It is a macro-measure of the total development environment. Putnam studies the trend in this Productivity Index and concludes that it has increased linearly over the last decade (Putnam 2000). Reifer (2004) confirms this, reporting an average linear productivity increase of 8 to 12 percent a year. In his study, productivity is viewed from a quality point of view by normalizing it to the quality of a software product when released to its intended customers or end-users.

For software productivity, another relevant issue is the efficiency of developing software products. In general, effort is being wasted for two main reasons: cancelled projects [the software product is never released] and software repair [testing and defect repairs]. Jones (1998) finds that 15% of the global software workforce is involved in projects that will never be deployed. Further, that 61% of software developers' time is spent on software repair, including testing. This confirms the observation of Boehm and Basili (2001) that some 40-50% of the effort on current software projects is spent on avoidable rework, excluding testing.

In Figure 1-4, the work pattern for a typical software engineer, after vacations, training and sick days, is given, with all activities other than time spent actually working on software projects factored out.

[3] Some of the most widely cited software related accidents in safety-critical systems involved a computerized radiation therapy machine called the Therac-25. Between June 1985 and January 1987, six known accidents involved massive overdoses by the Therac-25 with resultant deaths and serious injuries. They have been described as the worst series of radiation accidents in the 35-year history of medical accelerators (Leveson and Turner 1993). An analysis by the Center for Devices and Radiological Health of the Food and Drug Administration reveals that from 3,140 medical device recalls between 1992 and 1998, 242 (7.7%) were attributable to software failures (Center for Devices and Radiological Health 2002).

In addition to developing projects that never see the light of day, software personnel are involved in defect removal from projects eventually completed, and defect repairs during routine maintenance of software products already released.

Activities	Workdays	Percent
Testing and defect repairs	120	61%
Time on cancelled projects	30	15%
Productive time on projects	47	24%
Total	**197**	**100%**

Figure 1-4: Software Engineering Effort by Task
(Jones 1998)

Assuming software productivity is growing linearly, and not exponentially, a possible solution is to increase the amount of software produced per unit of time, by allowing more people to work in parallel. However, this solution provides limited results. It is only in the later project phases that work can be distributed effectively among more software engineers. Increasing the project team size will lead to higher overheads, thus likely to reduce the average productivity level.[4]

It is concluded that, until now, software productivity has mostly been increasing linearly, rather than exponentially, with most development effort still taken up by defect repair, and projects that will be prematurely terminated. This linear increase is insufficient to cope with the exponentially increasing software size.

1.3.2 Improvement Initiatives

Nearly two decades ago, Boehm (1987) identified a number of strategies for improving software productivity: get the best from people, make development steps more efficient, eliminate development steps, eliminate rework, build simpler products and re-use components. Re-use especially has been regarded as a high-potential solution (Poulin *et al.* 1993), but overall results have been disappointing (Card and Comer 1994; Schmidt 1999). In 2001, Boehm and Basili (2001) published a list of the most important factors in reducing defect injection and removal rates.

During the last decades, many software development organizations initiated software process improvement programs (Fuggetta 2000; Humphrey 2002). The intention of these initiatives is to improve the software manufacturer's performance by reaching, for example, the higher levels of process maturity models, such as the *Capability Maturity Model* or *CMM* (Paulk *et al.* 1993; Software Engineering Institute 1995), its successor the *Capability Maturity Model Integration* or *CMMI* (Software Engineering Institute 2002; Chrissies *et al.* 2004)[5], and *ISO/IEC 15504* (ISO 2003a, 2004b) in combination with the *ISO/IEC 12207* standard (ISO 1995, 2002b, 2004f).[6]

[4] See also (Brooks 1975) who claims that the increased overhead costs may outweigh any gain in productivity.
[5] The CMMI has a broader scope than the CMM with a focus on software development. The CMMI integrates four bodies of knowledge, or models, one can choose from [systems engineering, software engineering, integrated product and process development, supplier sourcing] and one can select two different representations: the staged representation [each maturity level exists of a number of process areas as in CMM] and a continuous representation [a maturity level is assigned to each process area as in ISO/IEC 15504].
[6] The revised version of the ISO/IEC 15504 standard describes the requirements for conducting assessments and making process capability profiles. The process descriptions themselves have been removed and integrated in the revised ISO/IEC 12207 standard.

The Software Engineering Institute twice yearly publishes a *Maturity Profile Update*. These profiles list the percentage of officially assessed software manufacturers performing at each maturity level using CMMI as the reference model. Over the last decade, improvements have been reported. The situation at the end of 2003 revealed that most organizations involved in the study are performing at the second maturity level [43.3%], with basic project management practices in place (Software Engineering Institute 2004). However, it is assumed the assessment of all software manufacturers worldwide would show a more dramatic picture, as the results reported to the Software Engineering Institute only include officially assessed software development organizations, namely organizations that are willing, or forced, to be assessed.

Jones (1998) compares the best software manufacturer organizations with the worst ones, to illustrate the scope for improvement. In Figure 1-5, a range of results for lagging, average and leading software projects is presented. These results suggest there is ample room for improvement. The defect injection rate in average organizations is almost twice that found in the best organizations. Further, by improvements to both the defect potentials and the defect removal efficiencies, there is room for a reduction by a factor of five in the number of defects actually delivered by average organizations.

Task	Leading	Average	Lagging
	[data expressed in terms of defects per function point]		
Requirements	0.55	1.00	1.45
Design	0.75	1.25	1.90
Coding	1.00	1.75	2.35
User manuals	0.40	0.60	0.75
Bad fixes	0.10	0.40	0.85
Total	**2.80**	**5.00**	**7.30**
Removal %	95%	85%	75%
Delivered	0.14	0.75	1.83

Figure 1-5: Comparison of Defect Potential and Removal Efficiency
(Jones 1998)

1.3.3 Software Crisis?

The term ‘software crisis’ has been used since the late 1960s to describe those recurring system development problems in which software development cause the entire system to be late, over budget, not responsive to the user and/or customer requirements, and difficult to use, maintain and enhance (Gibbs 1994). Royce (1991) emphasized this situation: *”The construction of new software that is both pleasing to the buyer/user and without latent errors is an unexpectedly hard problem. It is perhaps the most difficult problem in engineering today, and has been recognized as such for more than 15 years. It is often referred to as the ‘software crisis’. It has become the longest continuing crisis in the engineering world, and it continues unabated.”* It could be argued that the software crisis is dead in the sense that the software industry has passed the turning point, as under the right conditions software development can be managed. The positive side is that software productivity is increasing, some software manufacturers have succeeded in improving their maturity levels, and there is room for improvement, when comparing leading and average software manufacturer organizations. It could also be argued that one can still speak of a software crisis, in the sense that the turning point has probably not yet been reached. There is no indication that any improvement strategy can result in the performance improvements needed. It is questionable whether the effects of any improvement strategy can match the exponential growth of the variety and size of software products. It is

likely that, in the future, more software products will be released with higher defect densities, with the most likely consequence that end-users will be confronted with more defects. Put differently, the chronic character of software development problems is unlikely to be resolved in the future. The question that may arise is whether a possible solution to reduce or even eliminate the chronic software development problems is developing less and simpler software products. The study from Jones (2002), discussed in Section 1.2.2, reveals that as software size shrinks, defect potential decreases and defect removal efficiencies increase. In practice, it will be difficult to limit the development of new software products, however each software manufacturer will undoubtedly benefit from trying to minimize product size by implementing only those requirements for which a clear customer/user demand exists.

If the software industry is unable to find easy-to-implement improvement strategies, the typical software manufacturer organization is likely to become increasingly less predictable in terms of cost, quality and time-to-market, a trend confirmed by studies of the Standish Group (2001, 2004).[7] In markets with increasing competition and smaller market windows, software manufacturers might experience an increasing pressure to release software products prematurely, disregarding the total life-cycle effects (Berghout and Nijland 2001). In this case, uncertainties are:

- *Unknown product behaviour*. It is difficult, if not impossible, to guarantee customers/end-users the exact functional and non-functional requirements of the software product. This may lead to dissatisfied customers/end-users and to unforeseen, even potentially dangerous, situations. Apart from the fact that people's lives may be at risk, such situations can have an enormous financial impact on the software manufacturer.
- *Unknown operational maintenance cost*. The post-release or maintenance cost of the software may become unexpectedly high. If the exact status of the software with its documentation is unknown, a software manufacturer may be confronted with high maintenance costs for correcting failures. Future adaptive and perfective maintenance activities may be severely hampered.

The presence of these uncertainties may have a dramatic impact on a software manufacturer's market position. Releasing a software product too late might severely undermine its market position, releasing a software product prematurely might lead to recalls and warranty, or even liability, problems. This is illustrated by figures from the automotive industry; transformed from a mechanically-oriented industry to an electronic and software-oriented industry. In Figure 1-6, an overview is given of the number of officially reported recalls in the German and UK automotive industry (KBA 2003; Bates *et al.* 2004), showing a constant increase with an increasing percentage of recalls from the hardware/software area.[8]

[7] The latest Chaos report of the Standish Group (2004) reveals the software industry is losing ground. Only 28% of software projects succeed these days, down from 34% a year or two ago. Outright failures [projects cancelled before completion] are up from 15% to 18%. The remaining 51% of software projects are seriously late, over budget and lacking features previously expected.

[8] Including 'silent' recalls, the ones not officially reported, would reveal a more dramatic picture.

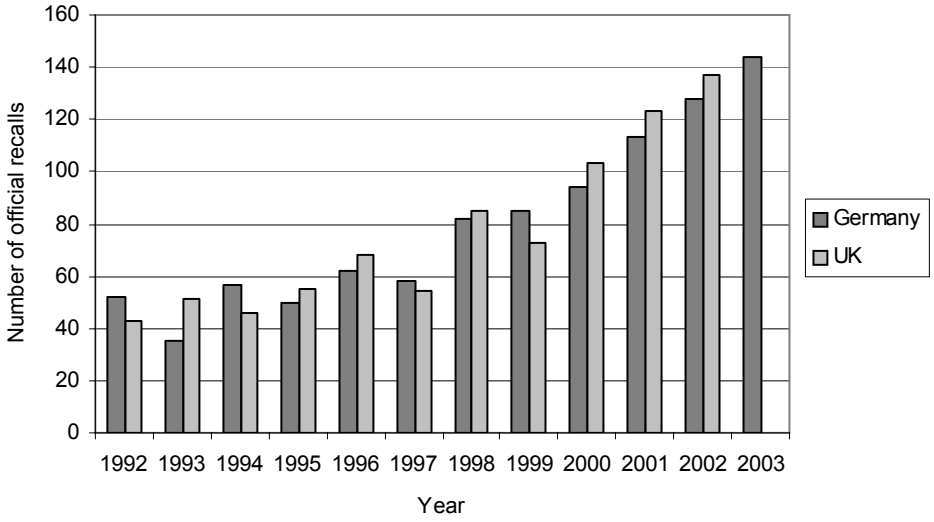

Figure 1-6: Recalls in German and UK Automotive Industry ~ 1992-2003
(KBA 2003; Bates *et al.* 2004)

Another study in the automotive industry, conducted by McKinsey (2001), concerns the financial impact of quality problems on profit margins. As in Figure 1-7, the economic significance of these problems is high.

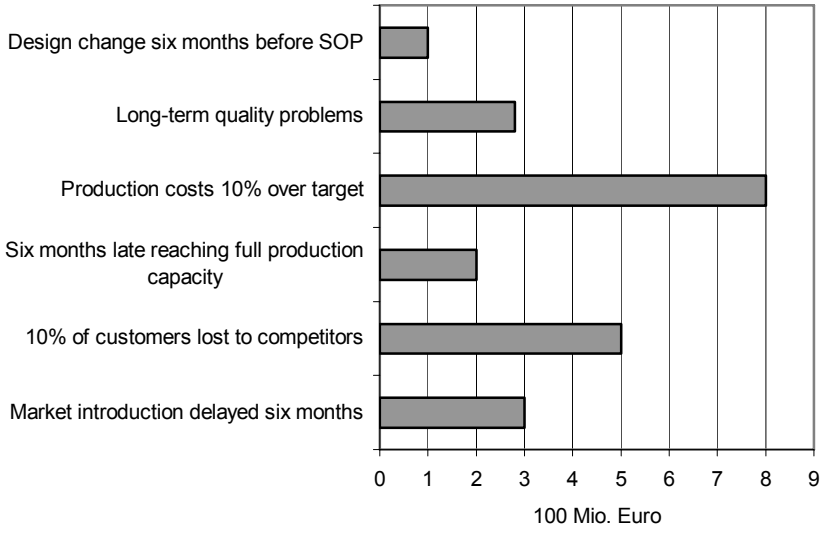

Figure 1-7: Cost of Quality Problems in Automotive Industry [loss of profit margin]
(McKinsey 2001)

Finally, when looking into the near future, it is expected that the market volume of automotive software and hardware [production costs] will grow significantly (Mercer Management Consulting 2002, Lederer *et al.* 2003), as illustrated in Figure 1-8.

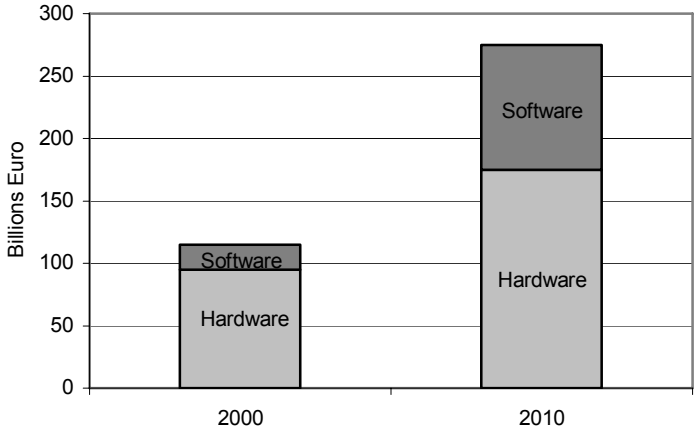

Figure 1-8: Expected Growth in Worldwide Market Volume [software and hardware production costs in automotive industry]
(Mercer Management Consulting 2002, Lederer *et al.* 2003)

From these figures, it is concluded that automotive manufacturers are facing uncertain times due to the unpredictability of their development process and resultant products, while software is expected to play an increasingly important role in new innovations, production costs, lead-time of product developments and released product quality.

It is concluded that software release decisions are likely to become more complex and important in the next years, or even decades, due to limited room for improvement and increasing uncertainty levels. A valid question to raise here is the extent to which software release decisions are addressed in existing methodologies, standards and models. This is discussed in the following Section.

1.4 Available Methodologies, Standards and Models

In this Section, some of the most widely known [software] project management methodologies, development methodologies, maturity models, standards and operational methodologies are discussed to investigate the extent to which software release decisions are addressed. Special attention is paid to whether they provide practical guidance in structuring software release decisions. Examples of this practical guidance are the distinction of different roles and responsibilities, the required information as input to the decision, the decision-making process itself and implementation aspects.

1.4.1 Project Management Methodologies

The Central Computer and Telecommunications Agency (CCTA), currently part of the Office of Government Commerce (OGC), developed *PRINCE* ['PRojects IN Controlled Environment']. This project management methodology was originally developed for use on software projects. The methodology was re-launched in 1996 under the acronym *PRINCE2* and is used in both the public and private sectors. PRINCE2 is a process-based approach to project management. Each process is defined with its key inputs and outputs, together with the specific objectives to be achieved and activities to be carried out. PRINCE2 has 8 main processes, comprised of 45 sub processes (Office of Government Commerce 2002). The eight processes are:

- Starting Up a Project (SU);
- Initiating a Project (IP);
- Directing a Project (DP);
- Controlling a Stage (CS);
- Managing Product Delivery (MP);
- Managing Stage Boundaries (SB);
- Closing a Project (CP);
- Planning (PL).

Software release is partially addressed in the process 'Closing a Project' (CP). The process covers the project manager's work in requesting permission to close the project, either at its natural end or for a premature close. The objectives of this process (Office of Government Commerce 2002) are to:

- Note the extent to which the objectives set out at the start of the project have been met;
- Confirm the customer's satisfaction with the products;
- Confirm that maintenance and support arrangements are in place [where appropriate];
- Make any recommendations for follow-on actions;
- Ensure that all lessons learned during the project are annotated for the benefit of future projects;
- Report on whether the project management activity itself has been a success;
- Prepare a plan to check on achievement of the product's claimed benefits.

This abstract description is of limited support in structuring software release decisions, as only the role and responsibilities of the project manager are addressed and some implementation aspects. The same conclusion holds for other project management methodologies like, for example, *PMBOK* ['Project Management Body Of Knowledge']. Where PRINCE2 provides a more prescriptive [although flexible] set of steps for the project manager and teams to follow, PMBOK offers the project manager a considerable array of information about proven practices in this field and invites the project manager to apply these where deemed appropriate. PMBOK also focuses strongly on planning processes; taking the perspective of the project manager (Project Management Institute 2000). Both PRINCE2 and PMBOK attempt, early in the life-cycle, to break down the project into smaller and smaller pieces using techniques such as creating Work Breakdown Structures. Identified sub-tasks are transformed into network diagrams, augmented with dependency and estimate information to determine the critical path for the project. As such, these methodologies focus strongly on planning. The over-riding theme is that if a project is planned properly, if progress is tracked against the plan and if corrective actions are taken, when work accomplished deviates from the plan, a project is likely to be successful. This is considered an incomplete and flawed view of projects because it does not, for example, take into account the inherent uncertainty associated with many of today's software projects.

It is concluded that these project management methodologies offer limited guidance in structuring software release decisions.

1.4.2 Development Methodologies

Generally, there are two categories of software development methodologies: [9]

- The traditional software development methodology utilizes a sequential approach whereby each phase of the project life-cycle is performed, and completed, before the next phase is started. As an example, a traditional approach may define the major phases of a project as: Requirement, Design, Code, Test and Deploy Phases. Each of the phases is performed only once, within the entire life-cycle, and each phase is performed to its completion before moving onto the next phase. The sequential models build on the assumption that the problem to be solved can be completely understood and described before a solution is designed. A design that satisfies all aspects of a problem can be specified before implementation. All implementation can be done before validation and delivery. This approach is most suitable for projects where the requirements are clearly stated and stable.[10] Examples of this approach are *Pure Waterfall* (McConnell 1996) and the *V-modell* (Dröschel and Wiemers 2000). The V-Modell was originally developed by and for the German government under the name V-Modell '97. Its successor V-Modell XT, released in 2004, does not prescribe a sequential approach and has the flexibility of adopting any development methodology (Rausch 2004).
- The opposite of the sequential approach is the iterative approach [sometimes called agile development methodology]. This approach is essentially a rapid prototyping framework, which assumes the requirements will become more complete, and change as the project progresses. The project life-cycle has short iterations, each including collection of requirements, designing, coding, testing and deploying for that small part of the software system. Examples of this approach are *Rapid Application Development* or *RAD* (McConnell 1996), *Rational Unified Process* or *RUP* (Kruchten 2000), *eXtreme Programming* or *XP* (Beck 2000), *Dynamic Systems Development Method* or *DSDM* (Stapleton and Constable 1997), *Scrum* (Schwaber and Beedle 2001), and *Feature-Driven Development* or *FDD* (Palmer and Felsing 2002).[11]

Available descriptions of development methodologies focus on the project life-cycle, stating the order in which activities should take place. For software release decisions, there is little practical guidance for distinguishing different roles and responsibilities, the information required as input to the release decision, is the release decision-making process itself and release implementation aspects. It is therefore concluded that these methodologies offer limited guidance in structuring software release decisions.

1.4.3 Maturity Models

The concepts of process or capability maturity are increasingly being applied to software product development, both as a means of assessment and as part of a framework for improvement. The principal idea of the maturity grid is that it describes typical organizational behaviour at a number of levels of 'maturity', for each of several aspects of the area under study. This provides the opportunity to codify what might be regarded as good practice [and

[9] See (McConnell 1996) for a detailed overview.

[10] An *incremental* development approach, whereby the total scope of work is broken up into a series of small sub-projects or increments, is considered a special implementation form of the sequential approach. In this approach the overall requirements of the final system or product are also known at the start of the development, however a limited set of requirements is allocated to each increment and with each successive [internal] release more requirements are addressed until the final [external] release satisfies all requirements. A special incremental development methodology is *Cleanroom* (Mills *et al.* 1987). The focus of Cleanroom involves moving from traditional, craft-based software development practices to rigorous, engineering-based practices. Cleanroom software engineering yields software that is correct by mathematically sound design, and software that is certified by statistically-valid testing.

[11] See (Abrahamsson *et al.* 2002) for an overview and comparison of agile development methodologies.

bad practice], along with some intermediate, or transitional, stages. Maturity approaches have their roots in the field of quality management. One of the earliest of these is Crosby's *Quality Management Maturity Grid* or *QMMG*, which describes the typical behaviour exhibited by an organization at five levels of 'maturity', for each of six aspects of quality management (Crosby 1979).

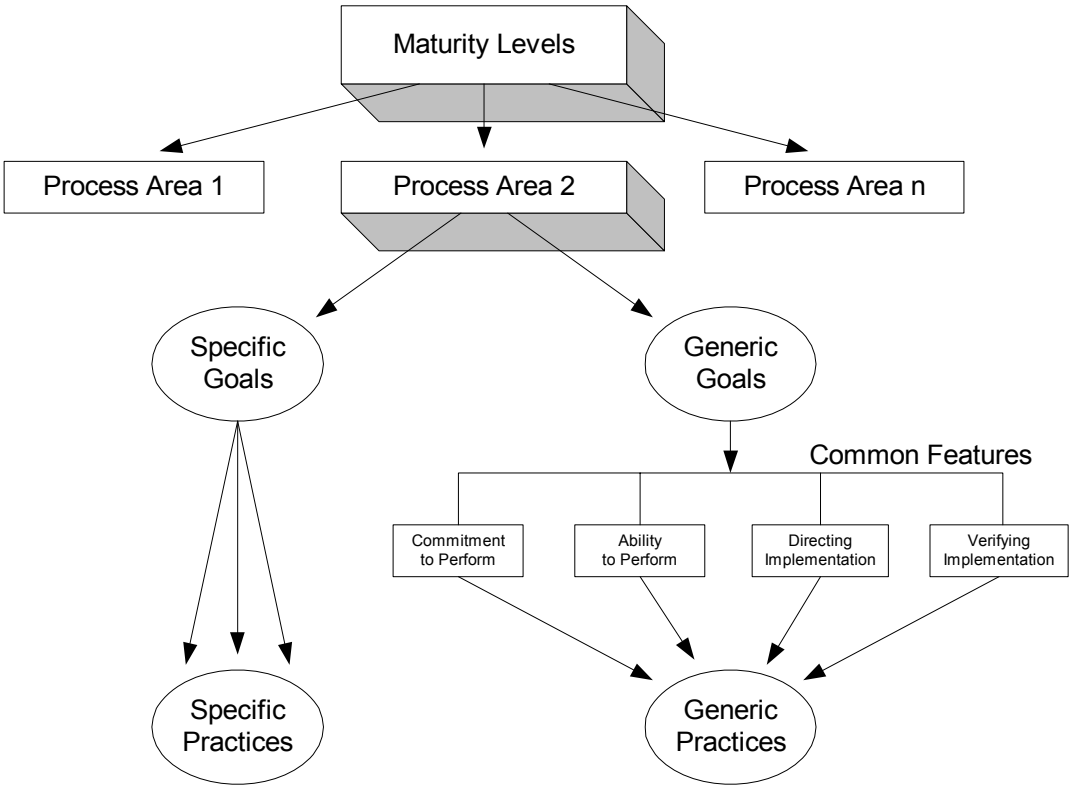

Figure 1-9: CMMI Components [staged representation]
(Software Engineering Institute 2002)

Some of the most widely-known models are the CMM (Paulk *et al.* 1993; Software Engineering Institute 1995) and its successor CMMI (Software Engineering Institute 2002; Chrissies *et al.* 2004).[12] In Figure 1-9 the model components of the staged representation of CMMI are given as an example.

Detailed descriptions of different aspects of software product development, the important aspects of software release decisions, are not explicitly addressed in these maturity models. In the staged representation of the software engineering variant of the CMMI, the process areas at maturity level 2 addressing project management issues (Software Engineering Institute 2002) are:

- Requirements Management;
- Project Planning;
- Project Monitoring and Control;
- Supplier Agreement Management;
- Measurement and Analysis;
- Process and Product Quality Assurance;
- Configuration Management.

[12] The ISO/IEC 12207 standard will be described in Section 1.4.4.

Software release decisions are only partially addressed in the Configuration Management process area, stating the requirement to control and monitor the release of work products built from the configuration management system. At maturity level 3, some process areas of interest are found, including Integrated Project Management [defining the need to establish and manage a project with the involvement of relevant stakeholders] and Decision Analysis and Resolution [to analyse possible decisions using a formal evaluation process that evaluates identified alternatives against established criteria]. However, these process areas state requirements in general terms that are not specific for software release decisions. It is concluded that these models offer limited guidance in structuring software release decisions.

Although not specifically a maturity model, another methodology aiming at capability improvements is *Six Sigma*.[13] This paradigm is often used by high-maturity organizations to implement *statistical process control* (*SPC*). Six Sigma originates from manufacturing environments and is concerned with statistical quality models of physical manufacturing processes that are inherently dynamic rather than static (Brassard and Ritter 2002). An integral part of every Six Sigma program is a DMAIC model: define, measure, analyse, improve and control (Wheeler 2005). This model is applied to existing processes. The form derived for the development of new products and processes is called *Design for Six Sigma* (*DFSS*), distinguishing the following five steps: define, measure, analyse, design and verify (DMADV). Six Sigma is relatively new to software development, and, although its applicability is criticized for various reasons (Binder 1997), it might be of support when defining and verifying criteria for tollgates for the later stages of the project life-cycle, examples being software testing and product rollout. A review of existing literature did not however reveal examples of tollgate definitions for the rollout stage, the latter one being the study object here.

1.4.4 Standards

The *ISO/IEC 9001* standard (ISO 2000a) has many similarities with maturity models like the CMMI, and addresses many of the same CMMI disciplines. The main difference is that ISO/IEC 9001 is a standard for quality systems, and not a maturity model. It is not an instrument for assessing capability and at best it provides a 'Pass/Fail' level of judgment. The standard emphasises an organization's quality management system including:

- Note the extent to which the objectives set out at the start of the project have been met;
- Management responsibility;
- Resource requirements;
- Measurement, analysis and improvement;
- Product realization.

The various aspects of software release decisions are not explicitly addressed in this high-level standard.

The same holds for other ISO/IEC standards. The *ISO/IEC 15288* standard describes the major processes at system level (ISO 2002a). Distinction is made between enterprise processes [enterprise management, investment management, system life-cycle management, resource management], agreement processes [acquisition, supply], project processes [planning, assessment, control, decision-making, risk management, configuration management], and technical processes [stakeholder needs definition, requirements analysis, architectural design, implementation, integration, verification, transition, validation, operation and maintenance, and disposal]. This standard is very global in nature and does not pay specific attention to [system] release decisions other than describing decision-making, stakeholder involvement, verification, transition and validation in general terms.

[13] 'x sigma' means the range *x* standard deviation above and below the average, or a span of *2x* [± sigma].

The *ISO/IEC 12207* standard describes the major component processes of a complete software life-cycle and the high-level relationships that govern their interactions (ISO 1995, 2002b, 2004f). It consists of the original standard (ISO 1995), Amendment 1 (ISO 2002b), and Amendment 2 (ISO 2004f). The evolvement of this standard is illustrated in Figure 1-10. The major revision was in 2002, when the process reference model of the technical report *ISO/IEC TR 15504* (ISO 1998a) was integrated in the standard. The standard covers the life-cycle of software from conceptualisation of ideas through retirement. Distinction is made between primary processes [acquisition, supply, development, operation, and maintenance], supporting processes [documentation, configuration management, quality assurance, verification, validation, joint review, audit, and problem resolution], and organizational processes [management, infrastructure, improvement, and training]. This standard is of a global nature and does not pay specific attention to software release decisions.

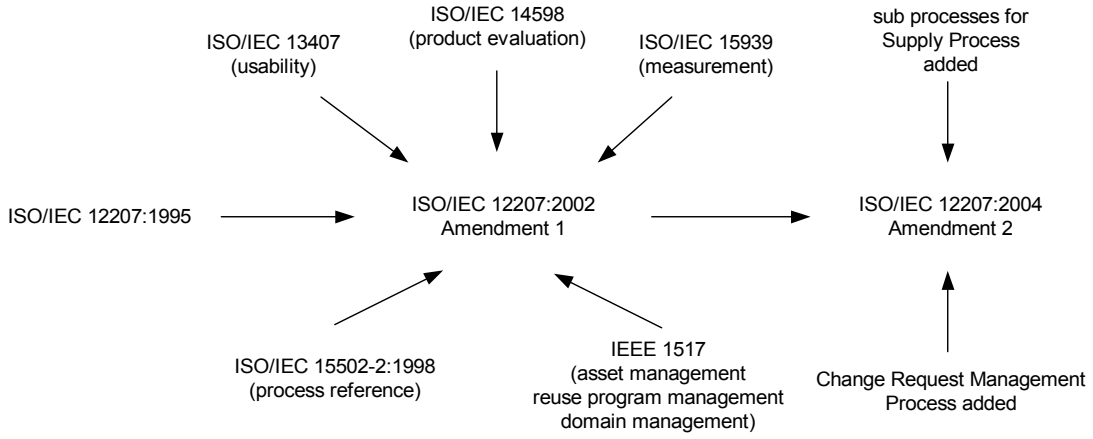

Figure 1-10: Evolvement of the ISO/IEC 12207 Standard
(ISO 1995, 2002b, 2004f)

The *ISO/IEC 14598* standard (ISO 1998b, 1999b, 1999c, 2000b, 2000c, 2001a) comes closer to the subject of software release decisions by describing the evaluation process for product quality attributes, as described in the *ISO/IEC 9126* standard (ISO 2001b, 2003b, 2003c, 2004e). Part 1 of the standard provides an overview of the other parts (Part 2 through Part 6) and explains the relationship between software product evaluation and the quality model. Additionally it describes the evaluation process in the following steps: establish evaluation requirements, specify, design and execute the evaluation. However, apart from this product evaluation description, the software release decision itself is not further addressed. It is therefore concluded that these standards offer limited guidance in structuring software release decisions. Most of the previously discussed methodologies, models and standards address software testing in global terms.

1.4.5 Testing Standards and Approaches

The British Standard Institute and IEEE have published testing standards specifically for software product development (British Standards Institute 1998; IEEE 1987). These standards, respectively *BS7925-2* and *IEEE 1008*, primarily define techniques and measures for component testing. Integration, system and acceptance testing are not covered by these standards.

Several approaches have been developed during the last decade to define and improve the testing process. Examples are *Test Management approach* or *TMap* (Pol *et al.* 2002), *Test Process Improvement* or *TPI* (Koomen and Pol 1999) and *Test Maturity Model* or *TMM*

(Burnstein *et al.* 1996a, 1996b). However, none of these approaches specifically address the release decision. Their purpose is to prescribe relevant processes with underlying practices.

These standards and approaches provide limited guidance for structuring software release decisions.

1.4.6 Operational Methodologies

Finally, some methodologies for IT service management of software products in an operational context [after they have been released] are reviewed to determine the extent to which they address software release decisions. Although these methodologies are not applied when transferring a software product from its development phase to operational use, they are concerned with the rollout of new releases of existing products.

IT Infrastructure Library or *ITIL* is probably the most widely-accepted approach to IT service management. It provides a cohesive set of best practices, drawn from the public and private sectors internationally. ITIL distinguishes various processes among which Release Management. The goals of Release Management are (Office of Government Commerce 2002b):

- The project management of software and hardware rollout;
- The design and implementation of efficient procedures for the distribution and installation of changes to IT infrastructures;
- To ensure that hardware and software changes are traceable, secure and that only correct, authorised and tested versions are installed;
- To communicate and manage expectations of the business/customer during the planning and rollout of new releases;
- To agree the exact content and rollout plan for the release, through liaison with the Change Management process;
- To implement new software releases or hardware into the operational environment using the Configuration Management and Change Management controlling processes;
- To ensure that master copies of all software are safe in a secure repository;
- To guarantee that all software/hardware rolled out, or changed, is secure and traceable, using the services of Configuration Management.

The *IT Service Capability Maturity Model* or *IT Service CMM* is a maturity growth model aimed at providers of IT services, such as management of hardware and software, operations, and software maintenance (Niessink *et al.* 2004). The structure of the model is equal to that of the Software CMM, but the contents of the IT Service CMM, however, are key process areas needed for mature IT service provision. The main difference from ITIL is that the IT Service CMM is organized as an ordered set of key process areas, whereas ITIL is organized as a framework of best practices, organized by different processes (Niessink *et al.* 2004).

As in the previously described methodologies, models, and standards, these methodologies do not explicitly address the important aspects of software release decisions, other than stating the requirement to control and monitor the release of work products built from the configuration management system and some project management aspects. It is concluded that these methodologies offer limited guidance in structuring software release decisions.

1.4.7 Summary

A review of some widely-known project management methodologies, development methodologies, maturity models, standards and operational methodologies reveal that software release decisions are only addressed in a limited way. Issues covering the definition of different roles and responsibilities; information required as input to the decision, the decision-making process itself and implementation aspects, are mostly ignored. Results are summarized in Figure 1-11.

Category	Examples	Focus	Aspects addressed for software release decisions
Project management methodologies	PRINCE2	Project planning	Role of project manager
	PMBOK	Project planning	Some implementation aspects
Maturity models	CMMI	Process capability	Product integrity
	Six Sigma / DFSS	Statistical process control	Decision analysis
Standards	ISO/IEC 9001	Quality system requirements	None
	ISO/IEC 15288	System life-cycle processes	Verification/transition/validation
	ISO/IEC 12207	Software life-cycle processes	Verification/validation
	ISO/IEC 14598	Product evaluation	Evaluation of product attributes
Testing standards, approaches	BS7925-2	Component testing	None
	IEEE 1008	Component testing	None
	TMap, TPI, TMM	Test process [improvement]	Test strategy and planning
Operational methodologies	ITIL	Service management	Role of project manager
	IT Service CMM	Service provision	Product integrity

Figure 1-11: Summary of Methodologies, Models and Standards

A preliminary investigation of other supporting literature reveals limited additional information.[14] Bays (1999) describes a software release methodology from both an operational point of view [addressing defect-tracking systems, source code control, and change control systems] and an organizational point of view. However, the organizational view is limited to a description of release engineering service strategies [overly flexible versus inflexible, centralized versus decentralized] and release management philosophies [in order of increasing maturity: reactive chaotic, reactive structured, consultant, and dictator]. This work however merely expresses the ideas and experiences of the author himself, while not referring to the existing body of knowledge or any research conducted.

Studying software release decisions has a scientific relevance, and will be a contribution to the existing body of knowledge, as this aspect has not been addressed in any of the above systems.

1.5 Benefits of a Formal Process

Decisions with little impact can often be made on the basis of informal rules and agreements, however as soon as common interest increases, collective decision-making has to be institutionalized through the elaboration of a more formal approach; a generally-accepted decision-making process to arrive at a decision based on the input of multiple stakeholders (Stokman *et al.* 2000, p.1). Whenever a formal decision-process is defined and established, the

[14] Not discussed here are software reliability estimation models that can support, at least in theory, finding the optimal release time [discussed in Chapter 3]. These are mathematical models and do not address decision-making.

risk exists that bureaucracy will occur: the process adds cost without adding value and/or slows an organization down. The justification for using a formal process is that it assists in identifying the functioning of a group in collective decision-making. By offering structure, it facilitates the early identification of [potential] problems that threaten decision success, and thus easier, and cheaper, solutions, or even prevention, of these problems. Furthermore, it helps to enhance understanding (Burke 1994; Winch 1995) and unity, or consensus, is much easier to achieve (Kolstoe 1985). The emphasis in a formal process is on reducing the risks associated with self-imposed controls and restrictions, and optimizing available resources by assigning process activities identified and eliminating those activities [costs] that add no value. This is important when common interest increases, or when stakes are high.

It can be argued that the definition, and implementation, of a collective decision-making process for software release decisions does not necessarily improve predictability for a software manufacturer in terms of cost, quality and time-to-market. Nearing the end of a project, the knowledge on the project has gradually increased, whereas the possibilities of action on the project are reduced: the irreversibility of decisions already made arises. Such a collective decision-making process should therefore include pro-active measures taken before the release decision itself, so a sound project start is ensured during the project proposal phase and time is allowed for the prevention of threats identified during product development. These measures should impact on the way a software manufacturer accepts and implement its projects prior to the release decision itself.

A formal process offers a structured mechanism to provide visibility of threats to release decision success. The net result of a formal approach is to help avoid preventable surprises late in the project, and improve the chance of meeting initial project commitments, and reducing the level of uncertainty. Reducing uncertainty has a cost, which should be balanced against the potential cost a software manufacturer could incur if the uncertainty is not reduced. It may not be cost-effective to try and reduce uncertainty too much. Formal approaches are of special concern when common interests increase, and when strategic value is present. In this study, a decision is considered as being of strategic value when large prospective financial loss outcomes to a software manufacturer and its customers/end-users of the software are present (Kunreuther *et al.* 2002). This is often true for software release decisions due to high costs for reversing the software release decision once made. Strategic value also has a long-term character as prospective loss outcomes may arise long after the decision has been made [for example, in cases where liability issues lead to lawsuits]. Decisions with strategic value should be made at a high level of the organization, require a formal decision-making process, and should be of concern to top management (Harrison 1987, p.30). Routine software release decisions, without strategic value, can be handled with a higher degree of certainty, and should be left to management at tactical, or even operational, level. Strategic software release decisions require a formal, collective decision-making process.

1.6 Primary Research Question

In the previous Sections, several arguments are put forward for a study of software release decisions:

1. Studying software release decisions has a social relevance due to the increasing impact of software on society; it is a crucial point of evaluation, often irreversible once the decision to release has been made (Section 1.2).
2. Software release decisions are likely to become more complex and important in future years, or even decades, due to the limited scope for improvements, and increasing uncertainty levels (Section 1.3).
3. Studying software release decisions has a scientific relevance as these are only addressed in a limited way in the most widely-known project management methodologies,

development methodologies, maturity models, standards, test standards and approaches, and operational methodologies. This will extend the body of knowledge in this area (Section 1.4).

4. Strategic software release decisions can benefit from the institutionalization of a collective decision-making process, offering a structured mechanism to provide visibility of threats to release decision success (Section 1.5).

Based on these arguments, the primary research question is formulated as:[15]

How to improve strategic software release decisions?

To investigate this research question, a methodology must be designed. The objective of this methodology is to describe the relevant practices that, when implemented successfully, help give a better chance of success when making strategic software release decisions. A successful decision is defined as one that results in the attainment of the objective that gave rise to the decision, within the constraints present (Harrison 1987, p.346). Such a methodology should enable a software manufacturer to understand the different aspects relevant to strategic software release decisions [descriptive character] and offer the possibility of assessing its capability in this area [judgmental character], thereby creating an instrument to identify possible improvement areas.

1.7 Scope of the Study

To narrow the scope of this study, the following limitations are applied:

- *Decision Type*. As in Sections 1.5 and 1.6, this study will focus on strategic software release decisions. When strategic value is involved, the increased common interest requires the institutionalization of a collective decision-making process. By offering structure, it facilitates the early identification of [potential] problems that threaten decision success and thus the easier and cheaper resolution, or even prevention, of these problems.
- *Release Type*. After a software product has been developed, different tests are conducted before the software product is put into operational use (Myers 1979; Sommerville 1995; Pressman 2000, Vliet 2000). The software product might first be internally released for *alpha testing* with the objective of finding and eliminating the most obvious defects. Subsequently, the software product can be released to a limited population of customers or end-users for *beta testing*, conducted to test the product for all functions in a breadth of field situations, and to find those failures likely to show in actual use. After the completion of this beta test, the software product can be officially released to its intended customers or end-users, and to the organizational authority responsible for post-release activities like software maintenance. In this study, the focus is on this last step, transferring the software product to operational use. The alpha and beta tests are considered the final steps of product development, prior to the software release decision, after which the software product is put into [full] operational use.
- *Software Type*. In Section 1.1, two categories of software are defined: system software and application software. Although there is not always a clear dividing line between these categories, the focus in this thesis is on application software. In this case customers/end-users are exposed to the software product released [instead of other software manufacturer organizations]. As mentioned in Section 1.1, a software product is either a standalone software product or software embedded in a system.
- *Software Manufacturer Type*. An important factor influencing the supply-demand/pricing relationship for software products is the relationship between the software manufacturer

[15] In Chapter 5, four additional secondary research questions are formulated.

and its customer(s) and/or end-user(s) of the software product (see Iberle 2003 for an overview of different software manufacturer types). This relationship between a software manufacturer and its customer(s) and/or end-user(s) influences the outcome of the software release decision. Releasing a software product to one known customer [custom system] will be different to releasing a software product to many unknown customers [mass market]. However, it is a requirement here that this relationship should not influence the methodology proposed. The purpose of the methodology is to identify the practices that help increase decision success, without focusing on a specific software manufacturer type.

- *Role of the Customer/End-user*. It is assumed that the software manufacturer takes a software release decision without direct involvement of customer(s) and/or end-user(s). However, when, for example, a custom system is written on contract, it is likely that they will be stakeholders in the decision-making process. An additional requirement for the methodology to be proposed is that this situation be taken into account.

1.8 Conclusion

In this Chapter important trends in the software industry are described. Our dependence on software is increasing, and as a result the total societal costs of computer failures increase. The size of software as well as the amount and variety of software products are growing exponentially. This growth implies that end-users will be exposed to more defects, if the software industry is not able to exponentially improve its defect potentials and removal efficiencies. Combined with increased competition and smaller market windows, it is likely that software manufacturers will be exposed to higher levels of uncertainties when releasing software. The decision to release a software product is expected to become an even more complex and important decision. Irrespective of this, a review of existing methodologies, models and standards reveals limited guidance in structuring software release decisions in a methodological way. It is the objective of this study to design a methodology for a better chance of success when making strategic software release decisions, by identifying, and managing, critical aspects. The definition of the scope of this study narrows the solution space of the formulated primary research question.

1.9 Thesis Outline

The remainder of this thesis divides into five parts:

- In Chapter 2 the research design for the study is described, including the chosen research philosophy, research approach and research strategies.
- In Part 1 (Chapters 3 and 4) the results of the *Exploration phase* are described, combining the results of the first literature study on existing theory of software release decisions and seven exploratory case studies. The questions addressed in these case studies are derived from the literature study.
- In Part 2 (Chapters 5 through 10) the results of the *Explanation phase* are described, and a supporting methodology for software release decisions is designed, based on the results obtained in the previous phase. This methodology is based on three styles of decision-making behaviour: maximizing, optimizing and satisficing behaviour, complemented by relevant issues for the decision implementation. An additional literature study is conducted in this phase, with the objective of finding answers to four additional secondary research questions, and identifying relevant theory on the methodology to be designed.

- In Part 3 (Chapter 11) the results of the *Testing phase* are described, with validation results of the methodology designed, in a practical context, through three additional case studies.
- In Chapter 12, the final part of this thesis, conclusions and recommendations are discussed. Conclusions are presented on the primary research question and secondary research questions. Recommendations are provided for further research in this area.

The outline of this thesis is depicted in Figure 1-12.

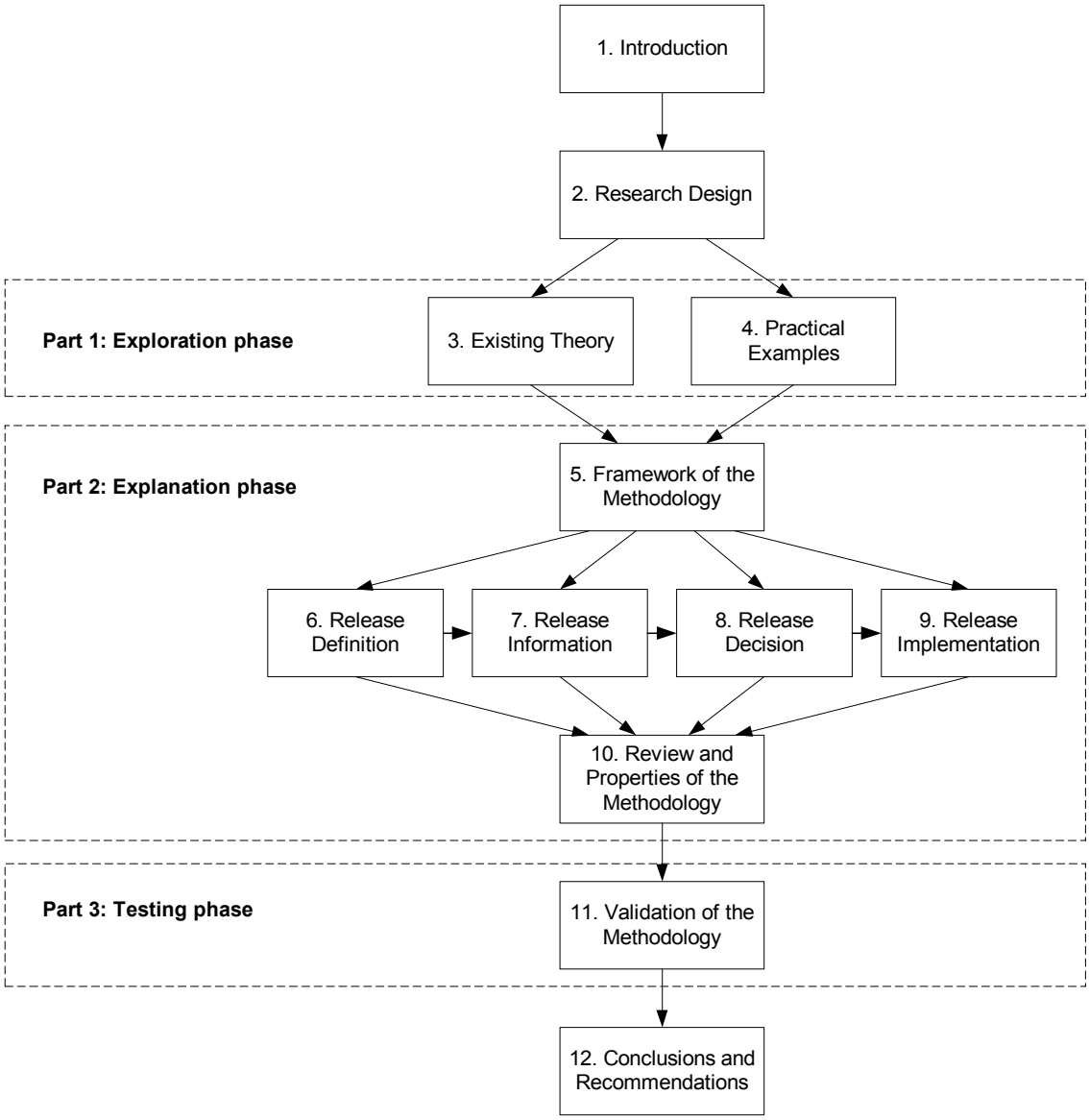

Figure 1-12: Thesis Outline