PART 1: EXPLORATION PHASE

3 EXISTING THEORY

"I conceive that a knowledge of books is the basis on which all other knowledge rests."
-- George Washington --

3.1 Introduction

In this Chapter the literature study, to identify theory relevant to software release decisions, is presented. The study object of this study, as outlined in Figure 3-1, is software release decision-making with the presence of strategic value.

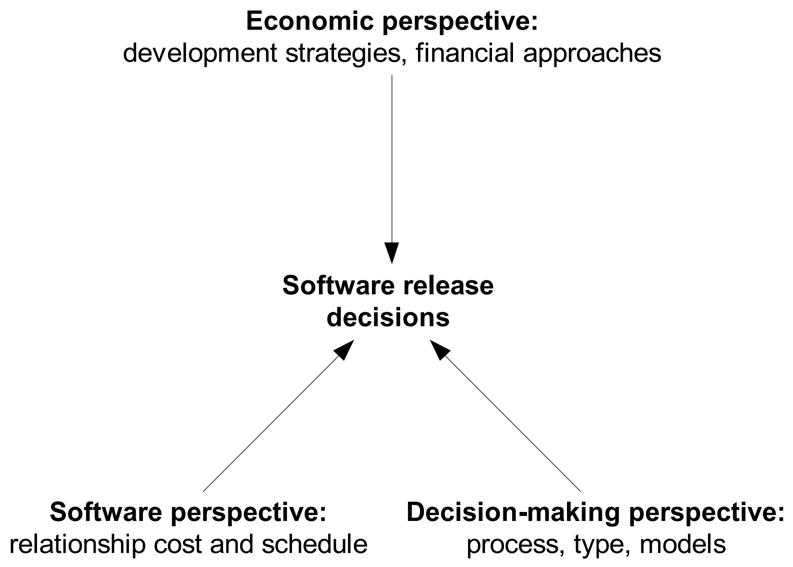
Economic perspective:
development strategies, financial approaches
Software release decisions
Software perspective:
relationship cost and schedule
Decision-making perspective:
process, type, models

Figure 3-1: Identified Perspectives for Software Release Decisions

The three disciplines involved are:

Software Management. The study focuses on the decision-making process for software products. Opposed to physical goods, software has unique characteristics as information goods (Varin 2000). Software product development has some specific difficulties compared to other engineering disciplines, mainly in its unpredictability (Standish Group 2001, 2004). The focus of the literature study from this perspective is the theory on the relationship between cost and schedule in software product development and maintenance. These are important trade-off parameters in software product development, from a project's start, when selecting the product development strategy, until the release decision, deciding which market entry strategy will be most appropriate.

Decision-making. An important aspect in the study is the decision-making process itself. Boehm and Sullivan (2000, p.2) note that there is an increasing need to understand how software design decisions, including release decisions, relate to business value creation, and they conclude that, in general, a gap exists between technical decisions and organization-level value maximization. Experimental research in psychology shows that engineering judgement is more trusted than it should be (Strigini 1996). The study of decision-making literature aims to reveal issues such as typical elements of a decision-making process, a classification of decision types, a description of decision-making models, models of choice, and decision success criteria that can be applied to decision-making. Motivation for this approach is that no literature could be identified which applied decision-making theory to software release decisions. At this stage, no attention is paid to behavioural aspects of decision-making, which will be addressed later in Chapter 8.

Economics. Strategic software release decisions are decisions with a possibility of large financial loss outcomes to the manufacturer and customers/end-users of the product. This includes the presence of high costs to reverse a decision once made, and, in general, these are decisions with a long-term horizon. From an economic perspective, software manufacturers make software products where the objective is, at least in theory, to maximize economic value, with strategic value an important input to economic value. Maximizing economic value depends on the market environment and internal strategic, and functional, characteristics of a manufacturer's organization (Heijltjes and Witteloostuijn 2003). These determine the long-term business strategy to be followed and provide the basis for deriving a product development strategy for a specific product, or service proposition, and guidelines for organising development efforts (Rajala *et al.* 2003). A product development strategy sets the boundaries for the product, or service, to be developed, and this determines the initial release criteria. Theories about different product development strategies – setting the initial project boundaries and market entry strategies for release trade-offs – for different software manufacturer types are discussed in this study.

The constraints laid on this review of existing theory are in agreement with Wisker (2001, p.127), who stresses the contextualising role of a literature review; it should not summarise all written work in a field, but rather highlight those aspects directly related to the study. The review of existing theory in this Exploration phase continues with an additional review of existing theory in the Design phase. In this phase, the focus is on exploring the study objective, whereas in the next phase the focus is on theory helping to answer additional [secondary] research questions, and identification of relevant theory on the methodology to be designed.

Areas of interest identified from the economic perspective are discussed in Section 3.2, the software management perspective in Section 3.3, and the decision-making perspective in Section 3.4. All Sections conclude with a number of questions as input for the exploratory case studies. The economic perspective is the starting point, as it offers an introduction to the areas of interest discussed in the software management perspective. A summary and conclusions are given in Section 3.5.

3.2 Economic Perspective

3.2.1 Product Development Strategies

To develop a product development strategy for a specific product, or service, an organization first determines its primary strategic orientation, or business strategy.[20] An organization cannot be all things to all people, and should focus on what will distinguish it in the marketplace. In literature many techniques are available that support the analysis, choice and implementation of the business strategy for an organization.

Heijltjes and Witteloostuijn (2003) distinguish three different schools, as follows:

- ❖ *Formulation School.* This school focuses on the fit between environment and strategy. A well-known technique is the *five forces analysis* (Porter 1980); a means of identifying the forces which affect the level of competition in an industry. Distinction is made between the organization and its competitors, the customers or end-users of the product, the suppliers, the substitutes for the product and new potential entrants from other markets. Porter distinguishes three generic strategies to out-perform competitors or maintain a market position against competition (Porter 1980, pp.35-40):
  - Overall cost leadership: produce the same or better product at less cost;

[20] A business strategy is considered to be different to a corporate strategy, stating an organization's overall direction in terms of products, services, resources and growth.

- Product differentiation: produce a unique product;
- Cost, or differentiation, focus on a particular buyer group, market segment or geographic market.

Porter considers that organizations failing to make a choice are likely to become stuck midway in the market.

Other techniques in this school are:

- *PEST Analysis* (Johnson and Scholes 1999, pp.104-107). This technique considers the environmental factors [political, economic, social and technological] affecting the organization [in the past, at present and in the future].
- *Value Chain Analysis* (Johnson and Scholes 1999, pp.156-160). This method describes the relationship between primary and support activities within, and around, an organization, and relates the result to an analysis of the competitive strength of the organization.
- *Comparative Analysis and Benchmarking* (Johnson and Scholes 1999, pp.179-186). Distinction is made in a historical analysis by comparing the present position of an organization with previous years, and by a comparison with industry norms, or best practices, through analysing the relative performance of an organization in the same industry, or public service, against an agreed set of performance indicators.
- *SWOT Analysis* (Johnson and Scholes 1999, pp.190-192). This analysis method summarises the key issues from an analysis of the business environment and the strategic capability of an organization. Carrying out an analysis using the SWOT framework helps an organization focus its activities on areas where the organization is strong and where the greatest opportunities lie.

❖ *Implementation School*. This school focuses on matching organizational and structural elements to an organization's strategy, with the motivation that manufacturing technology and management of resources are of central importance to the competitive positions of organizations (Heijltjes 1995).

❖ *Integration School*. This school combines the perspectives of the previous two schools, combining environmental and organizational characteristics with strategy (Miller 1988).

Defining a business strategy requires an identification of the market, and Iberele (2003) describes different software manufacturer types, each with typical characteristics, as in Figure 3-2.

Käkölä (2002) divides the market for software products and services into five major industry segments:

❖ *Professional software services* [planning, building, integrating and maintaining customized software systems for individual customers].

❖ *Enterprise solutions* [relying on both products and professional services that adapt and integrate products for customers needs].

❖ *Packaged mass-market software* [design and sale of software products for the public].

❖ *Internet-based applications* rented by Application Service Providers.

❖ *Embedded software*, including services.

Successful software business strategies are:

❖ *Operational excellence* – with a string focus on high quality and low costs.

❖ *Customer intimacy* – providing complete solutions and seeking long-term relationships.

❖ *Product or service innovations* – offering the best product in mass-markets.

(Treacy and Wiersema 1995; Rifkin 2001)

Software Manufacturer Type	Typical Characteristics
Custom systems written on contract	Software made for one particular buyer Budget and schedule fixed Penalties for late delivery
Custom systems written in-house	Used to improve efficiency/effectiveness of internal organization Limited number of end-users Possible conflicting interests between IT-department/end-users
Commercial software (business-to-business)	Software sold to other businesses Many different buyers Critical to the buyer's business
Mass-market software (business-to-consumer)	Software sold to individual buyers High volume buyers Market windows and buying seasons
Commercial/mass-market firmware	Cost of distributing fixes very high [physical items] Many to high volume buyers Failures can have fatal consequences

Figure 3-2: Characteristics of Software Manufacturer Types
(Iberle 2003) [21]

When defining the business strategy for software products, traditional supply-demand pricing relationships do not necessarily apply. Software is an *information good*, and as such it carries the following properties (Varin 2000; Messerschmitt and Szyperski 2001):

- *Experience good*. Consumers of software have limited ability to clearly see what they need. This introduces uncertainty for potential consumers and end-users of software. The quick pace of change, and high level of technology, enforces non-transparency of purpose and quality of software.
- *Returns to scale*. Software typically has a high fixed cost of production but a low marginal cost of reproduction and distribution. The fixed costs for software are also sunk costs. If a software application is not successful the developments costs are usually not recoverable.
- *Public good*. Software is typically non-rival and often non-excludable. Non-rivalry is a property of software itself and means the consumption of software does not diminish the volume available to other consumers. Non-excludability means one customer cannot exclude another from using the software.

Traditional economy shows inverse properties: consumers can better see what they need and what is for sale - two products are made at twice the cost of one, private goods are often rival and excludable (DeLong and Froomkin 1999). These properties matched the economy for centuries past. The fit for the software industry is less satisfactory and may influence supply-demand pricing relations (Messerschmitt and Szyperski 2001).

The availability of an overall business strategy for the portfolio of products, and services, enables an organization to derive a product development strategy for the competitive positioning of a specific product proposition and guidelines on how to organize development efforts. Card (1995) sees the product development strategy to be chosen for a specific product

[21] An issue not addressed in this overview is compliance to standards. In several markets standards have been defined to ensure, for example, the safety of software at a cost to the software manufacturer. Examples are: defence industry, aerospace industry and medical devices industry. These standards will have an effect on the way software is produced and released, especially if warranty conditions apply.

depending on the number of potential buyers and the competition level in the market, as in Figure 3-3.

		Competition Level	
		Low	High
Buyers	Few	Unique functional requirements	Lowest development cost
	Many	First mover	Highest quality

Figure 3-3: Model of Software Markets
(Card 1995)

Product development strategies identified by Card (Linders and Sassenburg 2004) are:

- ❖ *First mover*. Involves getting a product to market fastest. This is typical of software manufacturers involved with rapidly changing technology or products with a rapidly changing fashion [small market window]. Pursuit of this strategy typically leads to trade-offs in optimizing functional requirements, development cost and quality [e.g. reliability, safety].
- ❖ *Lowest development cost*. Focused on minimizing development cost or developing products within a constrained budget. It occurs, for example, when software manufacturers develop under contract for other parties, or where a company has severely constrained financial resources. It involves trade-offs with product features, time-to-market and quality.
- ❖ *Unique functional requirements*. Focuses on having the highest level of functional requirements [including aspects like the latest technology and/or product innovation]. It involves a trade-off of time-to-market, development cost and quality.
- ❖ *Highest quality* [or lowest operational cost]. Focuses on assuring high levels of product quality. This is typical of industries requiring high quality because of the significant costs to correct a problem after product release [e.g. recalls in a mass market], the need for high levels of reliability [e.g. aerospace industry], or where there are significant safety issues [e.g. medical devices]. It corresponds with minimizing operational cost, and involves a trade-off of functional requirements, time-to-market and development cost.

Not all product development projects within a product portfolio, or product development program, will necessarily have the same product development strategy: these may be different and often in conflict with each other (Mallick and Schroeder 2003). Some projects may intend the building of market share, others for revenue generation and others for market maintenance. Moore considers a product development strategy does not necessarily have to be constant over time, and presents a model, showing how a product development strategy shifts during the evolvement of a product (Moore 1995), as in Figure 3-4.

Market Description	Introduction	Early Adopters	Mainstream	Late Majority	End of Life
Buyer Profile	Technology enthusiasts	Visionaries	Pragmatists	Conservatives	Sceptics
Importance	1. Time-to-market 2. Product features 3. Reliability	1. Time-to-market 2. Product features 3. Reliability	1. Reliability 2. Time-to-market 3. Product features	1. Reliability 2. Product features 3. Time-to-market	1. Reliability 2. Product features 3. Time-to-market

Figure 3-4: Product Development Strategy
(Moore 1995)

Moore (1995) investigates why many new technology companies start with new inventions and rapid market growth, but collapse within the next three years. He explains the phenomenon by recalling an earlier model of mindsets towards the adoption of technologies. Initial success is gained by selling products to technology enthusiasts and visionaries, who are quick to grasp the implications and care less about, for example, reliability. However, when the initial market becomes saturated, the attempt to sell the technology to pragmatists might fail as they care more about stability or reliability. Moore uses the metaphor of a chasm: the company leadership discovers too late that it does not communicate with the pragmatists, as illustrated in Figure 3-5.

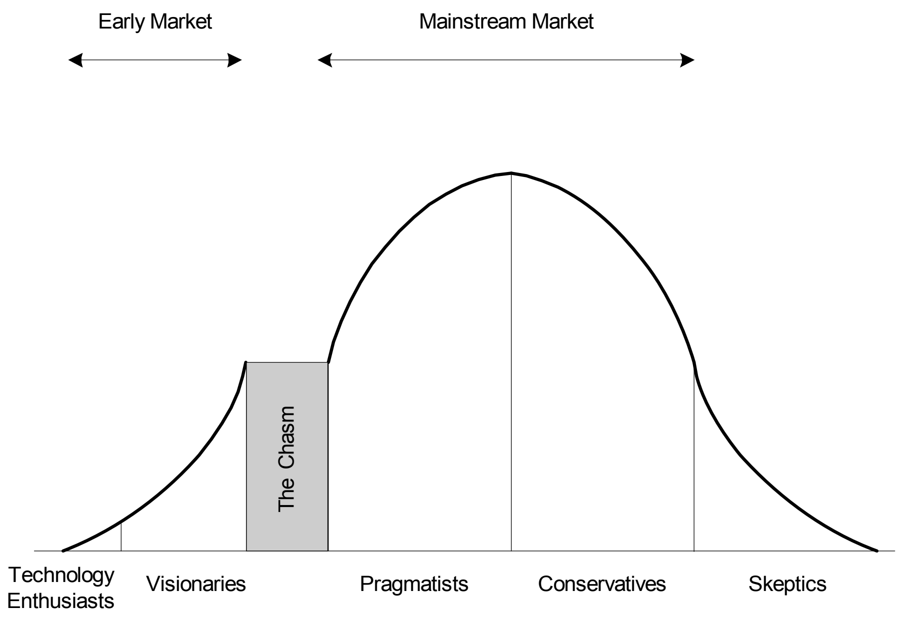

Figure 3-5: Market Chasm
(Moore 1995)

3.2.2 Market Entry Strategies

When a product development strategy for a specific product is chosen, an organization defines the initial boundaries of the project, as developing the product in terms of required product characteristics [product features, consisting of functional and non-functional requirements], schedule and costs. This can be regarded as the initial definition of the release criteria. However, the unpredictable nature of product development arising from incomplete and unstable requirements, or poor project performance, is a potential source for undermining the initial product development strategy. The external market environment may be dynamic, forcing an organization to continuously revisit the initial development strategy. As product development proceeds and releasing a product comes into sight, the trade-off between time-to-market and product performance arises. A market entry strategy will not necessarily be in line with the initial product development strategy.

In Figure 3-6 examples of possible profit models, when a manufacturer is faced with a release decision, are given (Sawhney 1999). When, for example, the entry of a new product is delayed in a market with heavy competition, the probability of the manufacturer capturing the advantages of early adopters will decrease, with a negative impact on revenue and profit. This effect might be less dramatic in a market with limited competition. A ‘rushed’ entry into market with a low quality product might also have a negative impact on revenues generated. Here the overall profit level may also decrease further from high operational costs for defect repairs.

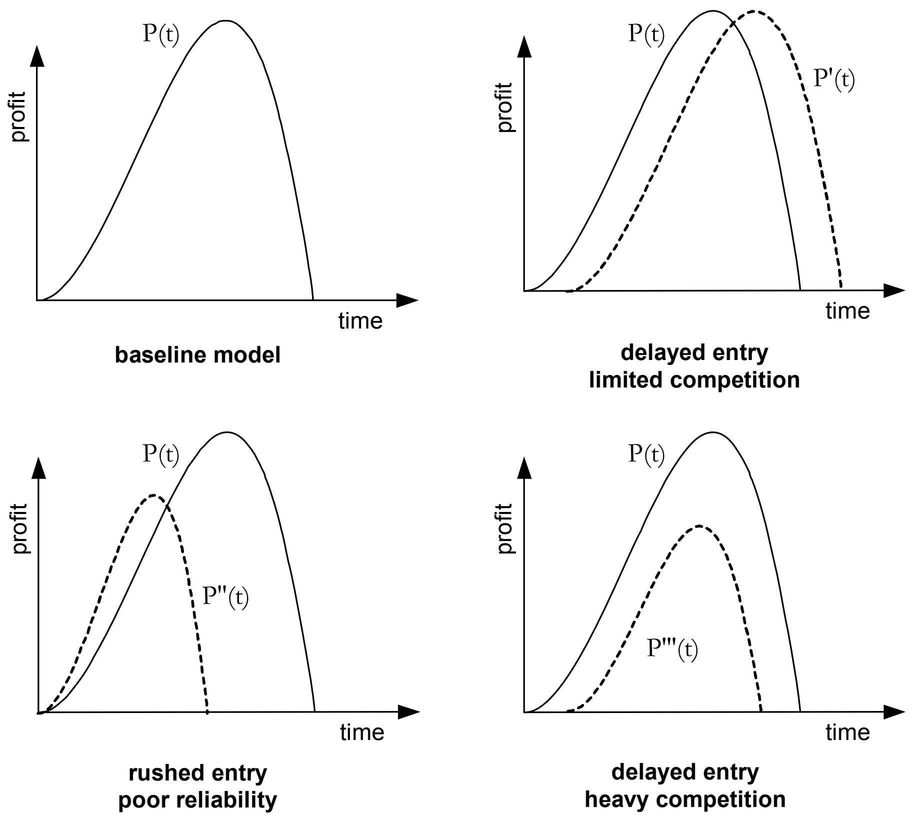

Figure 3-6: Profit Models
(Sawhney 1999)

The entry timing trade-off question is not unique to the software industry. Hatch and Macher (1999), for example, study the trade-off between time-to-market and manufacturing performance using the semiconductor industry as the empirical setting. The trade-off question is similar: find the optimal time to commercialise a new technology; whether to economise on time-to-market or economise on technology maturity. Delaying the transfer increases the cost of lost revenues and falling prices – transferring the technology too early creates manufacturing problems that could lead to product failure. The automotive industry is also confronted with the entry timing trade-off. Spending too much on quality issues, including safety, can result in a car manufacturer not making any money on their product, while spending too little can lead to a disaster. One can use a cost-benefit analysis to decide the economic justification, but in the case of safety-critical software products, human lives will be at stake. How does one measure the economics of saving a life, or preventing pain, when assigning a monetary value to human life? This is difficult and might even be unethical.[22]

Time-to-market is a crucial factor for product success, and ‘first mover’ advantage is a well-known strategy, but theoretical work has shown that, under certain conditions, there are later-mover advantages (Dutta *et al.* 1995; Hoppe and Lehmann-Grube 2001; Lin and Saggi 2002). Tellis and Golder (1996) find supporting empirical evidence, showing that the failure rate for first movers, or pioneers, is high. Their results show there are second-mover, or follower, advantages for dynamic quality competition. Lévesque and Shepherd (2004) study the choice of entry strategy in emerging and developed markets taking the perspective of an entrepreneur or *entrant*. They develop a dynamic model capturing the concomitant consideration of two

[22] Carchia (1999) summarizes a classic example where ethical aspects were involved, as in the release of the Ford Pinto. Design flaws involving the fuel system were not properly corrected. Ford used a cost-benefit analysis, placing a price on human life, and decided that it was not economically justifiable to fix the known problem immediately. The result was 27 people killed and Ford ended up paying millions of dollars in legal settlements to accident victims, recalling cars to install a part to fix the problem, and dealing with a tarnished reputation (Birsch and Fielder 1994).

important entry strategic decisions, namely, ‘should a manufacturer enter now or wait and enter later?’ and ‘to which degree should a manufacturer mimic the entry mechanisms of other manufacturers?’ They developed an optimal stopping rule prescribing an entrepreneur’s entry and mimicry decision. The resulting decision rules suggest one of three decisions of entry strategy: (1) delay entry, (2) enter with low mimicry entry mechanism, or (3) enter with a high mimicry entry mechanism. They argue that the cost/benefit ratio from using a high mimicry entry strategy is lower for manufacturers entering emerging economies than it is for manufacturers entering developed economies.

Release policies specific for software products have been studied. Cottrell (2000) examines the software release decision under *monopoly* and *duopoly* market structures for different *network benefits*, as in Figure 3-7.[23] For the monopolist, he concludes that, under a constant network benefit, it is in the monopolist’s interest to release a product with significant quality, while under a stochastically-evolving network benefit lower-quality products may be released. In the duopoly market structure, product quality is always lower. If the first entrant captures the entire network benefit software products are released immediately upon completion [often without significant testing], whereas with a muted network effect, the release is delayed. Although the value of this model as a high-level release policy is not questioned, it can be criticized as being too general to be useful in supporting software release decision-making. The focus is on reliability, ignoring the functional and other non-functional requirements of a product. The market entry strategy chosen will also depend on the relationship [history, reputation] between a manufacturer and customer(s)[24], and the phase of the product life-cycle (see Figure 3-4). Even knowing lower reliability might be acceptable in, for example, a duopoly market structure does not provide guidance in determining the acceptable reliability level, or the right release moment.

	Market Structure	
Network Benefit	**Monopoly**	**Duopoly**
Constant	Very high	Very low
Stochastic	High	Low

Figure 3-7: Product Quality Levels
(Cottrell 2000)

Arora *et al.* (2003) present an economic model for fixing software failures after the product has been released in the market. They claim that in the software industry repair, or patching, investment and time-to-market are strategic complements, as higher investments in patching capability allow a software manufacturer to enter the market earlier, offering the possibility of increasing the profit level over the total product life-cycle. They conclude that in a market with duopoly competition products are released earlier than in a monopoly market. The same criticism [not providing guidance in determining the acceptable reliability level, or the right release moment] is applicable to these models, but some specific assumptions in the model can also be criticized. It assumes the marginal cost of repairing multiple copies of defective software by issuing patches can effectively be zero for a large number of users. However, repair costs can be extremely high, for example, in the case of firmware. The model further assumes that users are fully aware of the quality of the software product, which is an invalid assumption with software being an information and experience product.

Software release decisions have also been studied in the field of game theory – an approach to decision analysis assuming competitors are likely to react to any moves an organization makes,

[23] Network effects arise when a product becomes more valuable as more users adopt the same, or compatible, products.
[24] Customers can decide whether to buy from an organization on the basis of past quality decisions [reputation]. When consumers are perfectly informed about an organization’s past quality decisions, they can punish an organization that produces low quality, by not patronizing it thereafter (Kranton 2003).

or that the organization will need to react to competitor moves. Games can be seen as conflict situations in which decision-makers have their own set of alternatives from which to choose. Zhao (2002) conducts a study in this area on competition between an *incumbent* and an entrant, focusing on three decisions: the amount of investment in a project, the timing of the new product introduction and the magnitude of performance improvement. He concludes that both the incumbent and the entrant should emphasize quality over time-to-market. Studies, simulating, for example, two players in a competitive market, with virtually identical software products (Zeephongsekul and Chiera 1995; Zeephongsekul 1996; Dohi *et al.* 2000), are also considered too general and lacking practical application.[25]

3.2.3 Conclusions on Software Release Decisions

Product development strategies, aligned with business strategies, are important as they define the scope of product development projects as the basis for the initial definition of release criteria. They depend on many factors, such as the external environment, the organization's strengths and weaknesses, the phase within the product's life-cycle and concurrency with other products in the same portfolio. Product development strategies normally change during product development due to dynamics in the external environment and/or differences between the actual and planned project performance, forcing a manufacturer to continuously revisit the initial release criteria. Models of different market entry strategies for different market structures are discussed, but assessed as too general and/or too theoretical to support software release decisions in a practical context.

To gain a better understanding of how software release decisions are made in a practical context, and from an economic perspective, the following areas of interest in exploratory case studies are identified:

1. *Which product development strategies do different software manufacturer types use?*
2. *Do law-like generalizations for product development strategies exist, as in Moore's model?*
3. *Which market entry strategies do different software manufacturer types use?*

3.3 Software Management Perspective

The focus of the literature study on this perspective is on theory concerning the relationship between cost and schedule in software product development and maintenance. The three issues investigated are:[26]

- ❖ Models describing the relationship between development cost and schedule [time]. These models support the selection and formulation of a product development strategy during the project proposal phase.
- ❖ Models to determine the optimal release time. These models support the derivation of the market entry strategy.
- ❖ Models to estimate expected post-release operational costs – an important input to the market entry trade-off.

[25] One diffculty in the general applicability of game theory results from the assumptions necessary to reduce the complexity of practical situations (Johnson and Scholes 1999, p.382).

[26] No attention is given to project management methodologies [e.g. PRINCE2 and PMBOK], maturity models [e.g. CMMI], standards [e.g. ISO/IEC 9001, ISO/IEC 15288, ISO/IEC 12207, and ISO/IEC 14598], testing standards and approaches [e.g. BS7925-2, IEEE 1008, TMap, TPI, and TMM], and operational methodologies [e.g. ITIL and IT Service CMM]. These are discussed in Section 1.4 and reveal limited attention to software release decisions. They do not, furthermore, address the relationship between cost and schedule in software product development and maintenance, which is of primary concern here.

3.3.1 Development Cost and Schedule

Software product development is characterized by unpredictability, and there are often large discrepancies between the initially planned and actual project objectives. The unpredictability of software product development is not new. Genuchten (1991), for example, studies this phenomenon to find reasons why software projects are often late. This unpredictability seems to increase. The Standish Group (2001) reported in 2001 an average cost overrun of 189%, an average schedule overrun of 222%, failure in meeting expectations of 61%, and only 28% of successful projects, a situation that has since worsened (Standish Group 2004).

Estimation of software cost and time schedules is a major topic in the software industry and has attracted the attention of researchers and practitioners, with software development cost estimation approaches developed. Several studies survey, and compare, existing software development cost estimation approaches (Fenton and Pfleeger 1997; Boehm and Abts 2000; Briand and Wieczorek 2000).

Boehm and Abts (2000, p.2) distinguish the following software estimation techniques:

- *Model-based*. These are mathematical, parameterised, models. By fitting data points from known projects the algorithm is derived.
- *Expertise-Based*. These use the opinions of experts who have past experience both in the software development techniques and the application domain.
- *Learning-Oriented*. Combine case studies [extrapolation from specific examples] and neutral networks [automation of improvements in the estimation process by building models that learn from previous experience].
- *Dynamics-Based*. Explicitly acknowledge that project factors, like effort and costs, are dynamic rather than static over time.
- *Regression-Based*. Used in conjunction with model-based techniques.
- *Composite*. Incorporate a combination of two or more techniques to formulate the most appropriate functional form for estimation.

The most widely applied models are the mathematical, parameterised models (Boehm and Abts 2000, p.4). These models derive their algorithms by fitting data points from known projects [regression]. All models have the form:

$$E = C_1 + C_2 \,.\, S^{C3} \qquad (3.1)$$

with:

E:	*effort in person months*
S:	*size of the product, expressed either in lines of code or function points*
C_1, C_2, C_3:	*constants*

To illustrate the practical implementation of such models, two examples of widely used models are discussed, namely COCOMO II (Boehm *et al.* 2001) and SLIM-Estimate (Putnam and Myers 1992). Both models are based on empirical relationships between project dimensions, as shown in Figure 3-8.

Driver	Schedule	Effort	Defect Density
Product size	+	+	+
Productivity	-	-	-

Figure 3-8: Empirical Relationships
(Sassenburg 2002a)

Increasing product size requires a lengthened schedule and/or increased effort, while defect density increases (see also Section 1.2). Increasing productivity has a positive effect on time schedule and/or effort, and defect density will decrease. But, as can be derived from equation 3.1, the relationships are not linear and accelerating a schedule is only a limited possibility.

COCOMO II distinguishes three separate models (Boehm *et al.* 2001):

1. The *Application Composition Model* involves prototyping to resolve potential high-risk issues such as user interfaces, software/system interaction, performance and technology maturity. The model might be used during the earliest phases of a project, and during any other prototyping later in a project. The function used to calculate effort takes, as input, the estimated number of so-called new object points [NOP, comparable to function points] and the productivity rate PROD [determined by both the developer's experience/capability and the maturity/capability of using an integrated CASE environment (ICASE)].

$$E = NOP / PROD \tag{3.2}$$

2. The *Early Design Model* involves exploration of alternative software/system architectures and concepts of operation. During this product design phase, not enough is generally known to support fine-grain cost estimation with a high level of accuracy. The function used to estimate effort *E* takes as input the estimated size, seven cost drivers and five scaling factors.[27]

$$E = 2.94\, Size^{B} \,.\, \Pi\, EM_i \tag{3.3}$$

$$B = 0.91 + 0.01 \,.\, \Sigma\, SF_j \tag{3.4}$$

3. The *Post-Architecture Model* involves the actual development and maintenance of a software product. Once the life-cycle architecture has been defined, detailed information should be available to assign values to seventeen cost drivers, whereas the five scaling factors are the same as used in the Early Design model.

$$E = 2.94\, Size^{B} \,.\, \Pi\, EM_i \tag{3.5}$$

In all models the formulae to estimate the needed development schedule *TDEV* is:

$$TDEV = 3.67\, E^{(0.33 + 0.2\,(B - 1.01))} \,.\, SCED\% / 100 \tag{3.6}$$

The SCED variable is used to express the required compression/expansion percentage for the schedule. It is also one of the cost drivers for effort. Compression of the development schedule results in a higher effort.

SLIM-Estimate is a constraint model, which can be applied to projects exceeding 70,000 lines of code. This model, developed by Putnam and Myers (1992), assumes that effort for software projects is distributed similarly to a collection of Rayleigh curves. Different Rayleigh curves are used for design and code, test and validation, maintenance and management. This assumption is based on the work of Norden (1963), who observes that the Rayleigh curve is a good approximation of the manpower curve for various hardware development processes.

[27] The cost drivers are multiplicative factors that determine the effort required to complete a software project. They are grouped into four categories: product, platform, personnel and project. The selection of five scale drivers is based on the rationale that they are a significant source of exponential variation on a project's effort or productivity variation. (Boehm *et al.* 2001).

The model is described by two equations. The first equation, *software equation*, describes the development effort as proportional to the cube of the size, and inversely proportional to the fourth power of the development time:

$$Size \quad = \quad C \, . \, E^{1/3} \, . \, TDEV^{4/3} \qquad (3.7)$$

with:

C:	*technology factor*
E:	*total project effort in person years (after product design)*
TDEV:	*lead-time to delivery in years (after product design)*

The technology factor C is a composite cost driver and reflects, for example, process maturity, management and engineering practices, state of the development environment, skills and experience level of the team, and the complexity of the application. These cost drivers can be compared with those described in COCOMO II.

The second equation, *manpower-buildup equation*, describes the effort as proportional to the cube of the development time:

$$D_0 \quad = \quad E \, / \, TDEV^3 \qquad (3.8)$$

with:

D_0:	*manpower acceleration*
E:	*total project effort in person years*
TDEV:	*lead-time to delivery in years*

Combining the two equations leads to the formula for effort (Fenton and Pfleeger 1997, p.444):[28]

$$E \quad = \quad (Size \, / \, C)^{9/7} \, . \, D_0^{\,4/7} \qquad (3.9)$$

These equations can support software manufacturers in making trade-offs between functional requirements, quality, cost and schedule [time-to-market]. They further offer the possibility of setting realistic objectives prior to the development of a software product. The cost versus schedule trade-off is an especially important one. Having computed a nominal value for schedule, a software manufacturer may face the question of adjusting the schedule to either deliver the software product at an accelerated pace, or to improve efficiency. Any adjustment to the schedule will have a cost impact. There is an optimal time T_o, in terms of a combination of development schedule and required resources [costs]. Increasing development time, up to T_o, as the optimal development time in terms of costs, can reduce costs. As a schedule is further stretched, the costs again begin to climb. Accelerating the schedule is possible, but the cost penalties are severe. As shown in Figure 3-9, costs increase exponentially as the schedule is accelerated.

Both models acknowledge that a barrier exists for schedule acceleration called the *impossible region*, as in Figure 3-9. Empirical research shows that using existing development approaches, it is impossible to accelerate a schedule beyond this barrier (Putnam and Myers 2003), as a minimal development time T_{min} exists.

[28] Note this equation is similar the one used in COCOMO II: schedule is proportional to the cube root of effort and effort is proportional to size, to a power constant greater than 1.

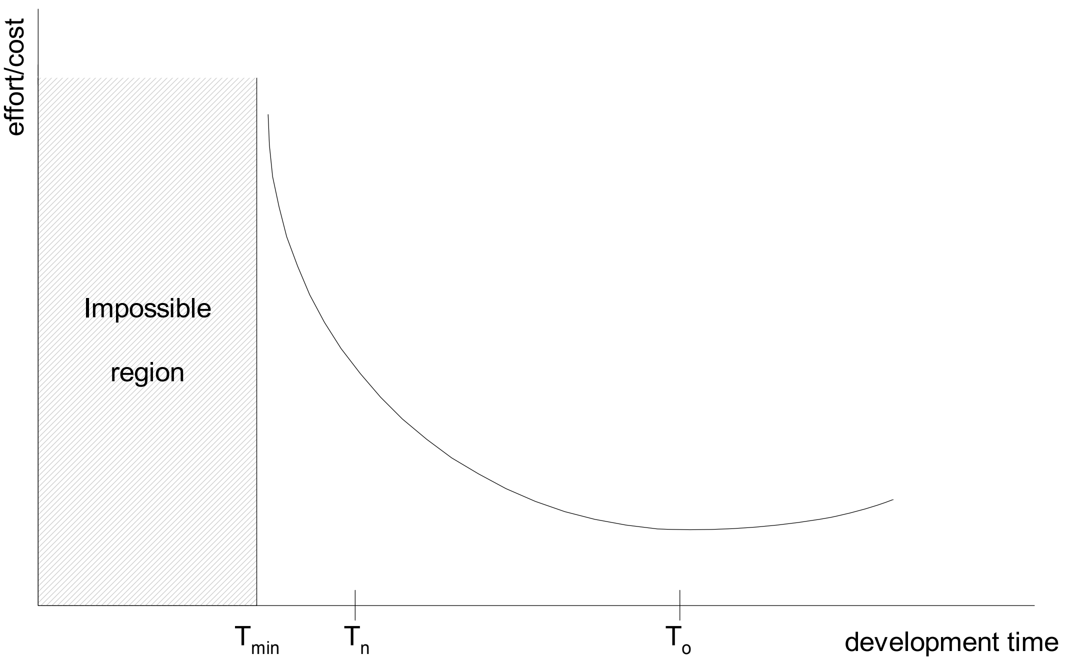

Figure 3-9: Development Time versus Effort/Costs

For software manufacturers willing to accelerate their schedules for product development, two options can be considered.

- ❖ In the first place, as discussed in Section 1.3, an option is to improve software productivity, for example, by implementing a process improvement program. However, it is concluded that the scope for improvement is limited when comparing the best software manufacturer organizations with the worst (Jones 1998).
- ❖ Secondly, the schedule can be accelerated by reducing the net software size to be developed, by either re-using existing internal software and/or using software developed by a third party.

These traditional regression-based models have problems and can be criticized (Fenton and Pfleeger 1997, pp.445-448; Shepperd and Schofield 1997). Fenton *et al.* (2004) note that these classic models provide limited support for managerial decision-making when faced with trade-off questions, such as:

- ❖ For a problem of a given size, and given limited resources, how likely is a project to achieve a software product of suitable quality?
- ❖ Where a resource-constraint and quality cannot be sacrificed, what are the staff requirements to build the software product with these limited resources?

Fenton *et al.* (2004) promote the idea of causal models, using Bayesian nets, to provide relevant predictions when making such tradeoffs. A *Bayesian net* is a graphical network together with an associated set of probability tables. The nodes in the net represent uncertain variables and the arcs in the net represent causal/relevance relationships between the variables. The probability tables for each node provide the probabilities of each state of the variable of that node. For nodes without parents there are just marginal probabilities while for nodes with parents these are conditional probabilities for each combination of parent state values. Once a Bayesian net is established, evidence about variables [as soon as available] can be entered. All the probabilities are updated accordingly, offering valuable information on variables one is interested in predicting.

In Figure 3-10, the overall causal model for resource estimation in the form of a Bayesian net is given, consisting of six subnets:

- *Distributed communications and management*: variables that capture the nature and scale of the distributed aspects of the project, and the extent to which these are well managed.
- *Requirements and specification*: variables relating to the extent to which the project is likely to produce accurate and clear requirements and specifications.
- *Process quality*: variables relating to the quality of the development processes in the project.
- *People quality*: variables relating to the quality of people working on the project.
- *Functionality delivered*: all relevant variables relating to the new functionality delivered on the project, including effort assigned to the project.
- *Quality delivered*: all relevant variables relating to both the final quality of the system delivered and the extent to which it provides user satisfaction.

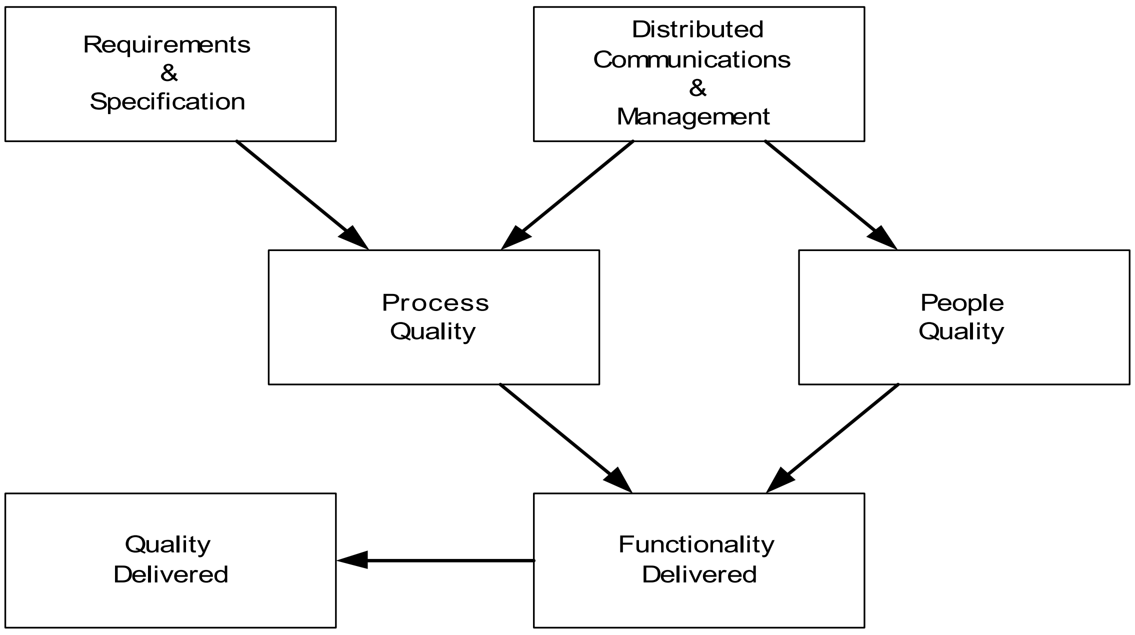

Figure 3-10: Schematic for a Project Bayesian Net
(Fenton *et al.* 2004)

Some initial results using this approach have been reported (Fenton *et al.* 2004), but further research in this area is required.

Although the models presented can be of help during the various phases of a project, the usefulness of the software cost estimation models is limited towards the end of product development, when the project is faced with the release decision itself. Projects normally end with a testing phase, in which the crucial question is: When can testing be stopped so the product can be released? To answer this question, additional models have been developed to determine the optimal release time, and an overview of these models is given in the next Section.

3.3.2 Optimal Release Time

The crucial question during the testing phase of a product is: when can testing be stopped so the product can be released? Literature on software testing has appeared since the beginning of computer science (Goodenough and Gerhart 1975; Myers 1979). Software testing involves any activity aimed at evaluating an attribute, or capability, of a software product and determining that it meets specified requirements (Hetzel 1993). Software testing involves both *verification* and *validation* of the software product. Verification is the activity that ensures 'the product is built right'. Validation is the demonstration that the software implements each of the stated software requirements, for a specific intended use, correctly and completely, confirming 'the right product is built'.

Although software is not unlike other physical processes where inputs are received and outputs produced, it differs in the manner in which it fails and this influences the verification and validation process:

- ❖ Detecting and removing design defects in software is difficult, as unlike most physical systems such as hardware, most of the defects in software are not manufacturing defects but design errors, which are harder to visualize, classify, detect and correct (Lyu 1995). This is also true for reliability. The unreliability of hardware systems tends to be dominated by physical failures, whereas the reliability in software arises from human intellectual failures (Keene 1994). Once the software is released, the residing defects are embedded in the product and remain latent until activation. Other distinct characteristics of software compared to hardware are (Keene 1994):
  - *Wear-out*. Software does not have energy related wear-out phases.
  - *Repair*. Software failures can possibly be fixed with periodic restarts.
  - *Time dependency / life cycle*. Software reliability is not a function of operational time.
  - *Environmental factors*. These factors do not influence software reliability.
- ❖ Detecting and removing design defects in software is difficult, as the possible input combinations thousands of users can make across a given software interface are simply too numerous for testers to apply them all (Whittaker 2000). Because software is a digital system, testing boundary values is insufficient for verification. In theory, all the possible values need to be tested and verified. Complete testing is, however, not realistic.[29] In practice, software programs have multiple different inputs, where timing, unpredictable environmental effects and human interactions generate virtually infinite numbers of distinct input combinations. Testing may be effective in showing the presence of defects, but it is inadequate for showing their absence. As described in Section 1.2.2, the number of tests required achieving any given level of test coverage increases exponentially with software size.
- ❖ Another complicating factor stems from the dynamic nature of software (Harrold 1999). If a failure occurs during testing software version *n* and the code is changed, the new software version *n+1* may now work for the test case(s) where it didn't previously work. But its behaviour on other pre-error test cases, previously passed, is not necessarily guaranteed. Any specific fix can (a) fix only the problem reported, (b) fail to fix the problem, (c) fix the problem but damage something that was previously working, or (d) fail to fix the problem and damage something else (Whittaker 2000). To account for this possibility, testing should be restarted. However, the incurred costs of re-executing all

[29] Consider, for example, a well-known program which finds the greatest common divisor of two positive integers, with an upper limit of 1,000 on the input values. To exhaustively test this program it is necessary to choose *every* combination of the two inputs, leading to [1,000 x 1,000] possible distinct input combinations If the program is modified to allow input in the range 1 to 1,000,000, and include a further input [so it is now finding the greatest common divisor of three numbers] then the input domain grows to [1,000,000 x 1,000,000 x 1,000,000] distinct input combinations. If it was possible to create, run, and check the output of each test in one-hundredth of a second, then it would take over 317 million years to exhaustively test this program.

previous test cases are often prohibitive. So, the question arises how much retesting [regression testing] of version *n+1* is necessary using the tests that were run against version *n*.

Independent of how the testing process is organized and how test cases are selected (See Beizer 1995; Hetzel 1993), at some point in time questions will arise as to how reliable the software product is; how long the software will run before it fails; and how expensive the software will be to maintain? Reliability, defined as the probability that a product will operate without failure under given conditions for a given time interval, is an important non-functional requirement to take into account when the release trade-off question is raised. If testing, as the last project stage, is stopped too early, significant defects could be released to intended users and the software manufacturer could incur the post-release cost of fixing resultant failures later. If testing proceeds too long, the cost of testing and the opportunity cost could be substantial. Littlewood and Strigini (2000) point out that the increasing ubiquity of software stems from its general-purpose nature, causing serious problems in achieving sufficient reliability and demonstrating its achievement.[30]

Two types of software reliability models are available (Reliability Analysis Center 1996):

- *Software reliability prediction models* [also referred to as *quality management models*, see Kan 2003] address the reliability of the software early in the life-cycle, at the requirements, design or coding level, using historical data. The reliability is, for example, predicted using fault density models and uses code characteristics, such as lines of code and nesting of loops, to estimate the number of faults in the software.
- *Software reliability estimation models* [also referred to as *reliability growth models*, see Kan 2003] evaluate current and future reliability from faults, beginning with the integration, or system testing, of the software. The estimation is based on test data. These models attempt to statistically correlate defect detection data with known functions, such as an exponential function. Figure 3-11 summarises the main differences between these two types.

Issues	Prediction Models	Estimation Models
Data reference	Uses historical data	Uses data from the current software development effort
When used in the development cycle	Usually made prior to coding and test phases; can be used as early as concept phase	Usually made later in the product life-cycle during testing (after some data have been collected)
Time frame	Predict reliability at some future time	Estimate reliability at either present or some future time

Figure 3-11: Differences between Prediction and Estimation Models
(Reliability Analysis Center 1996)

Software reliability prediction models use characteristics of the software and the software development process, throughout the development cycle, and extrapolate to operational behaviour. Some examples are:

- *Phase-Based Model* (Gaffney and Davis 1988; Kan 2003). This model assumes that faults in the different development phases follow a Rayleigh density function. Further assumptions are that the staffing level is directly related to the number of faults discovered during development and that estimates for the code size are available during the early phases of the development cycle. The formula used is:

[30] Sources for this phenomenon are the difficulty and novelty of problems tackled, the complexity of the resulting solutions, the need for short development cycles and the difficulty of gaining assurance of reliability, because of the inherently discrete behaviour of discrete systems (Littlewood and Strigini 2000).

$$\Delta Vt = E\,[\,e^{-B(t-1)^2} - e^{-Bt^2}\,] \qquad (3.10)$$

Parameters *B* [based on defect discovery phase constant] and *E* [total lifetime fault rate, expressed in fault density] are estimated as data becomes available. The time *t* is the life-cycle phase index [e.g. 1 for Requirements Analysis, 2 for Software Design].

- *Rome Laboratory or RADC Model* (Rome Laboratory 1987). In this model several factors related to fault density are selected: application type, development environment, requirements and design representation metrics [such as reviews], and software implementation metrics [such as language type, modularity and complexity]. These factors are used to compute the initial fault density prediction δ_0; which value is used to predict the initial failure rate.
- *COQUALMO* (Chulani 1999). This model has two sub-models: the Defect Introduction (DI) and Defect Removal (DR) models. The DI-model is formulated using the product, process, computer and personnel attributes [based on COCOMO II] and predicts the number of faults to be introduced in the different development phases. The DR-model estimates the number of faults removed by several defect removal activities [like reviews], each of them with six different levels of detection/removal efficiencies.
- *Orthogonal Defect Classification* (Chillarege *et al.* 1992). This concept enables in-process feedback to developers by extracting signatures on the development process from defects. The methodology classifies software defects and provides a set of concepts that supports guidance in the analysis of defects data. 'Orthogonal' refers to the non-redundant nature of information captured by the defects attributes and their values used to classify defects (Bassin *et al.* 2002; Butcher *et al.* 2002; Linders and Sassenburg 2004).

Although software reliability prediction models can be applied during the entire product development process, software reliability estimation models have been formulated to find the optimal release time for software products. These models have in common the support of the trade-off between three dimensions cost, time and quality during the test phase, i.e. when the project is nearing the release date. These models take the general form (Xie 1991; Boland and Singh 2002):[31]

$$\begin{aligned} C_{to}(t) &= I_t(t) + M_s(t) \\ &= i_1 \,.\, m(t) + i_2 \,.\, t + m_1 \,.\, [\, m(\infty) - m(t)\,] \end{aligned} \qquad (3.11)$$

with:

$C_{to}(t)$:	*total cost of testing and removing faults during testing/operation phase*
$I_t(t)$:	*total cost of testing and removing faults during testing phase*
$M_s(t)$:	*total cost of testing and removing faults during operation phase*
i_1:	*expected cost of removing a fault during testing phase*
i_2:	*expected cost per unit time of testing*
m_1:	*expected cost of removing a fault during operation phase*
$m(t)$:	*expected mean number of faults detected in time (0,t]* [32]

The mean value function *m(t)* is often described as an increasing, strictly concave function. There are many functions that can satisfy these conditions. In many studies the Goel-Okumoto [or exponential] model is used (Kimura *et al.* 1999; Xie and Yang 2003). The exact form of the mean value function is the objective of many studies and is derived from the failure rate of a

[31] The original equation is $C(t) = c_1 \,.\, m(t) + c_2 \,.\, [\, m(\infty) - m(t)\,] + c_3 \,.\, t$. This equation has been slightly rewritten by making an explicit distinction between the total cost of testing and removal of faults during the testing phase $I_t(t)$ and the total cost of testing and removal of faults during the operation phase *M(t)* to enable comparison with equations used later in Chapters 6 and 7.

[32] The mean value function is derived from the intensity function *h(x)*: $m(t) = \int h(x)dx$, and offers the possibility of calculating the cumulative MTBF [Mean Time Between Failures]: $MTBF(t) = 1 / m(t)$. This is often a stated requirement for reliability.

software product under test. In the Goel-Okumoto model, a non-homogeneous Poisson process (NHPP) is used to model the failure process. The failure rate generally decreases due to detection and removal of faults. At any time [present time], it is possible to observe the history of the failure rate [present failure rate], as in Figure 3-12. Software reliability estimation models forecast the curve of the failure rate from statistical evidence.

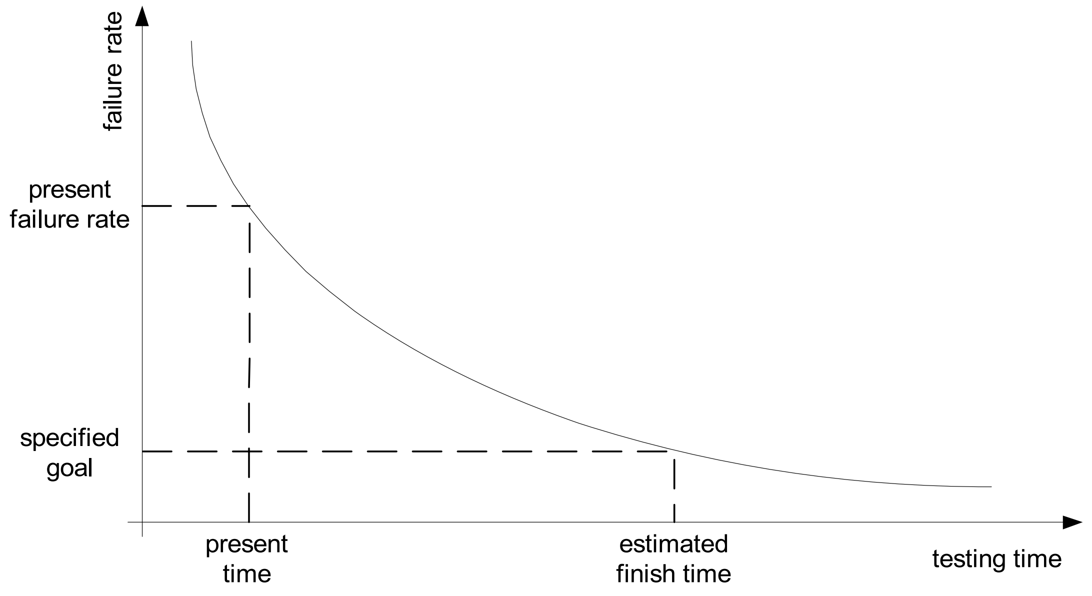

Figure 3-12: Basic Software Reliability Estimation Modelling

Gokhale *et al.* (1996) investigate available software reliability estimation models, and classify these as depicted in Figure 3-13.[33]

Characteristics of the different categories of models are (Gokhale *et al.* 1996):

- *Error Seeding*. The software product with an unknown number of faults is seeded with a known number of faults and subjected to rigorous testing. An estimate of the actual number of faults is obtained by determining the ratio of the discovered seeded faults and discovered actual faults. Mill's Hypergeometric model falls in this category (Mills 1972).
- *Input-domain*. The reliability of the software is measured by exercising the software with a set of randomly chosen inputs. An estimate of the actual number of faults is obtained by determining the ratio of the number of inputs that resulted in successful execution and the total number of inputs. The model proposed by Thayer *et al.* (1976) belongs in this category.
- *Homogenous Markov Models*. These models assume that the initial number of faults in a software product is unknown. The failure intensity depends upon the number of residual faults. Examples of models in this category are the Jelinski-Moranda model (Jelinski and Moranda 1972), the Littlewood model (Littlewood 1981) and the Goel-Okumoto Imperfect Debugging Model (Goel 1985).
- *Non-homogenous Markov Models*. The number of faults is assumed to be a random variable, most often assumed to display the behaviour of a Non-Homogeneous Poisson Process (NHPP). Distinction is made between finite and infinite models, addressing whether or not the expected number of failures observed during an infinite period of time is finite. Examples include the Goel-Okumoto model (Goel 1985) and the Delayed S-shaped model (Ohba 1984).

[33] See also Kan (2003), who distinguishes *time between failure models* and *fault count models*.

- *Semi-Markov Models*. The initial number of faults is assumed to be unknown but fixed. The failure intensity depends on the number of residual faults in the software and time elapsed since the last failure, an example being the Schick-Wolverton model (1973).
- *Others*. Some time-domain models are developed, which can neither be classified as homogeneous Markov, non-homogeneous Markov models nor as Semi-Markov models: examples being Bayesian models like the Littlewood-Verall Bayesian model (Abdel-Ghally *et al.* 1986).

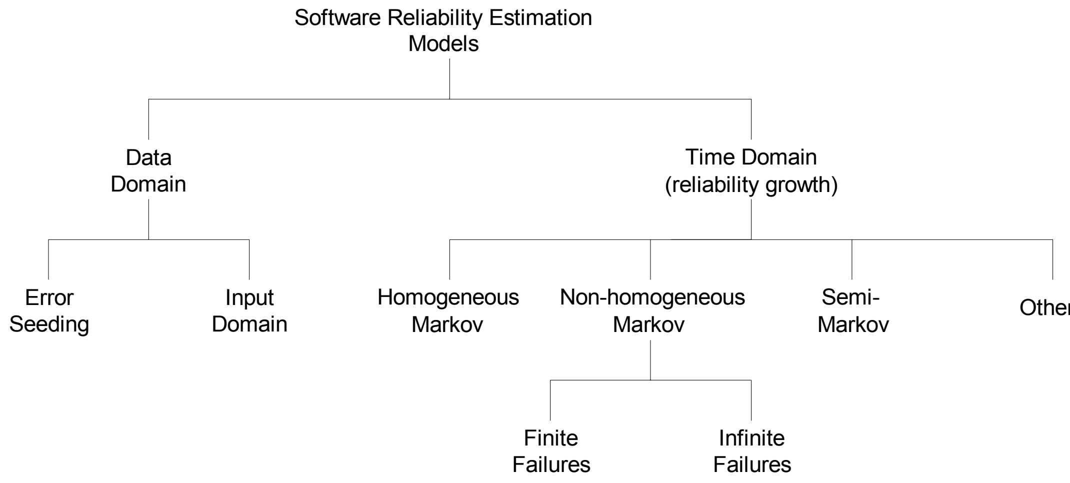

Figure 3-13: Classification of Software Defect Prediction Models
(Gokhale *et al.* 1996)[34]

Using a reliability model, one can determine the optimal release time T^*; defined as the time that minimizes the total cost *C(t)*, which is the sum of test cost and operational cost:[35]

$$dC_{to}(t)/dt = 0 \qquad (3.12)$$

Many models can be found in literature, all based on the initial cost model described by Okumoto and Goel (1980). Models are described, for example, by Yamada and Osaki (1986); Brettschneider (1989), presenting a simplified decision-making model; Ehrlich *et al.* (1993); Yang and Chao (1995); Boland and Singh (2002); Hou *et al.* (1997), incorporating the penalty cost when the software is delivered after the scheduled delivery time; Pham and Zhang (1999), incorporating warranty and risk cost; Xie and Yang (2003), incorporating the effect of imperfect debugging on software cost; Huang *et al.* (1999), incorporating ways to improve test efficiency; Kimura *et al.* (1999) incorporating warranty cost; Yamada *et al.* (1993), incorporating life-cycle distribution and applying discount rate; Pham and Zhang (2003), incorporating test coverage: Kapur and Garg (1989), incorporating penalty cost; and Leung (1992), incorporating a budget constraint.

Software testing is a trade-off between functionality, budget, time and quality, where the objective is, at least in theory, to maximize economic value. However, 'What happens if the project is behind schedule, or over budget, when it reaches the testing phase?' In general, reducing functionality is not an option as the requirements have already been specified,

[34] A model is Markovian if the next test results only depend on the presently encountered state [present time, the number of faults found and remaining and the overall parameters of the model].

[35] This can be seen as a *cost of software quality* approach. This approach distinguishes between control costs (prevention and appraisal) and failure costs [internal and external] described for example by Knox (1993), Krasner (1998) and Dobbins (1999). The control and failure costs have an inverse relationship to one another: as the investment in achieving quality increases [higher control costs], the costs due to lack of quality decrease [lower failure costs]. This relationship can be shown as a set of two-dimensional curves, which plot costs against a measure of quality. The total cost of quality, obtained by adding the two curves, will show a minimum prior to achieving 100% of the quality measure, being the point of diminishing returns (Sassenburg 2002a).

designed, documented and coded.[36] So, budget and time [release now without spending more, but with a certain number of defects] are traded against improved quality through continued testing [release later with less defects but additional test budget]. In theory, the software reliability estimation models presented not only offer the possibility of determining the optimal release time, but also the possibility of determining the effects of a time, or budget, constraint. Where a software manufacturer is faced with a time-constraint [fixed schedule delivery] or a budget-constraint [limited cost], it is possible to make predictions of the operational cost based on the reliability level at release time. A software manufacturer might further be faced with a reliability-constraint [minimal reliability requirement], and these models can help determine the time and test cost for reaching this requirement, and the expected operational cost after product release.

Assumption	Reality
Faults are repaired immediately when discovered	Faults are not repaired immediately. A work-around may be to leave out duplicates and to accumulate test time if a non-repaired fault prevents other faults from being found. Fault repair may introduce new faults. It might be the case that newly introduced faults are less likely to be discovered as retesting is not as thorough as the original testing.
No new code is introduced in testing	It is frequently the case that fixed or new code is added during the test period. This may change the shape of the fault detection curve.
Faults are only reported by the testing group	Faults may be reported by lots of groups due to parallel testing. If the test time of other groups is added, there is a problem of equivalency between an hour of the testing group and an hour of other groups [types of testing may differ]. Restricting faults to those discovered by the testing group eliminates important data.
Each unit of time is equivalent	The appropriate measure of time must relate to the test effort. Examples are: calendar time, execution time and number of test cases. However, potential problems are: the test effort is asynchronous [calendar time], some tests create more stress on a per hour basis [execution time] and tests do not have the same probability of finding a fault.
Tests represent operational profile	It is hard to define the operational profile of a product, reflecting how it will be used in practice. It would consist of a specification of classes of input and the probability of their occurrence. In test environments tests are continually being added to cover faults discovered in the past.
Tests represent adoption characteristics	The rates of adoption, describing the number and type of customers who adopt the product and the time when they adopt, are often unknown.
Faults are independent	When sections of code have not been as thoroughly tested as other code, tests may find a disproportionate share of faults.
Software is tested in isolation	The software under testing might be embedded in a system. Interfaces with e.g. hardware, can hamper the measurement process [test delay due to mechanical or hardware problems, re-testing with adjusted mechanical or hardware parts].
Software is a black-box	There is no accounting for partitioning, redundancy and fault-tolerant architectures. These characteristics are often found in safety-critical systems.
The organization does not change	When multiple releases of a product are developed, the organization might significantly change, e.g. the development process and the development staff. After the first release, a different department might even execute the development of the next release. It may also heavily influence the test approach by concentrating on the changes made for corrective maintenance and preventive maintenance [new functionality].

Figure 3-14: Assumptions of Models versus Reality

(Hamlet 1992; Hecht *et al.* 1997; Wood 1997; Whittaker 2000; Li *et al.* 20)

[36] This is one of the reasons that agile development methods (see Section 1.4.2.) are promoted as they offer, at least in theory, the possibility to reduce functionality as well.

The usefulness of the software reliability estimation models has been heavily criticized. Criticism is twofold:

- In the first place, most models assume a way of working that does not reflect reality (Hamlet 1992; Hecht *et al.* 1997; Wood 1997; Whittaker 2000; Li *et al.* 2003), meaning that the quality of assumptions is low. See Figure 3-14. As a result, several models can produce dramatically different results for the same data set (Gokhale *et al.* 1996, p.1; Fenton and Pfleeger 1997, p.390), meaning that the predictive validity is limited. Because no two models provide exactly the same answers, care should be taken to select the most appropriate model for a project, and too much weight should not be given to the value of the results (Wallace and Coleman 2001). Grams (1999) claims that the models make strong statements for basically unaltered software, but in the case of new innovative software products, the models are completely worthless.[37]
- Secondly, Fenton and Neil (1999) studied the most-widely used models and observe many shortcomings. They conclude that, as a result, these models provide little support for determining the reliability of a software product. Their study also shows that the number of pre-release faults is not a reliable indicator of the number of post-release failures.[38] The problem is that many software manufacturers use the pre-release fault count as a measure for the number of post-release failures, e.g. the reliability of the released product.

The lack of practical applicability of traditional verification approaches for non-functional requirements, like reliability and safety, has led to the exploration of new approaches. Fenton and Neil argue that Bayesian nets offer a model that takes into account the crucial concepts missing from classical approaches (Fenton *et al.* 1998; Neil and Fenton 2005), analogously as was done for software cost estimation methods (see Section 3.3.1). An example of such a Bayesian net for defect prediction is shown in Figure 3-15. The nodes in the net represent uncertain variables and the arcs in the net represent causal/relevance relationships between the variables. Classical prediction methods do not take these relationships into account, but focus on correlation between variables [e.g. size and defects].

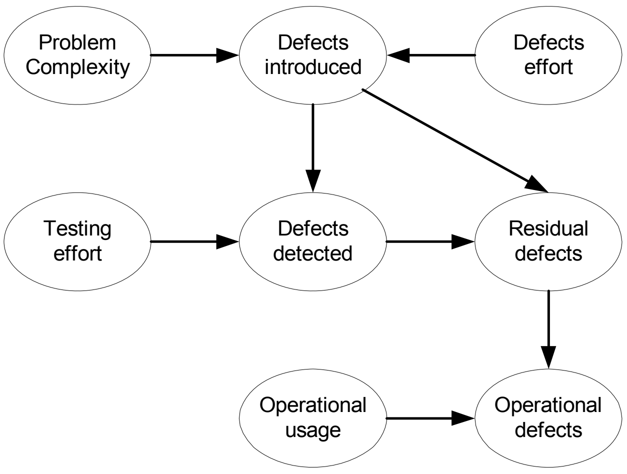

Figure 3-15: Bayesian Net
(Fenton and Neil 2001)

[37] This may explain why these models are mostly applied in high-reliability industries like avionics and telecommunications, where high-reliability is achieved during extensive operational use (Littlewood and Strigini 2000). Rae and Robert (1995) define evaluation methods for reliability and other non-functional requirements, and propose different methods for different risk levels.

[38] Beizer (1990) uses an analogy with pesticide to illustrate the difficulty in software testing, known as the Pesticide Paradox: *Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual*. By eliminating the [previous] easy bugs one allows an escalation of features and complexity, but this time one is faced by super, subtler, bugs.

Although Bayesian nets attempt to model the crucial notion of uncertainty and may overcome some of the limitations of standard reliability techniques and, although positive results have been reported (Neil and Fenton 2005), its practical application to support software release decisions is assumed to be limited for large and complex software products due to the multitude of interdependent variables and the excessive assessment burden, which might lead to informal, and indefensible, quantification of the modelled variables. Further research in this area is required.

Another relatively new approach to construct and present well reasoned arguments that a system achieves acceptable levels of safety, is the development of safety cases, where arguments are structured using a technique called *Goal Structuring Notation* (*GSN*) (Kelly 1998). Weinstock *et al.* (2004) have broadened this work and developed the concept of dependability cases. This approach focuses on creating and documenting structured rationales that convincingly show how evidence gathered during system design and test, supports claims regarding non-functional requirements like dependability, real-time performance, reliability and maintainability (Reliability Analysis Center 2004). Ongoing research is required to investigate the practical application.

It is concluded that determining the optimal release time using existing theory is difficult, and it is probable that uncertainty will be present when making the software release decision in a practical setting. Capturing this uncertainty by using probability models has not been theoretically, or experimentally, verified, except in specific application domains (Whittaker 2000).

A question arising from this conclusion is how software release decisions are made in practice. Despite the multitude of stories about post-release problems of prematurely-released software, limited literature is found on how the optimal release time is determined in practice. A study prepared for the National Institute of Standards & Technology, revealed a combination of the following non-analytical methods to decide when a software product is 'good enough' for release (RTI 2002):

- A 'sufficient' percentage of test cases run successfully.
- Statistics are gathered about what code is exercised during the execution of a test suite.
- Defects are classified and numbers and trends are analysed.
- Real users conduct beta testing and report problems that are analysed.
- Developers analyse the number of reported problems in a certain period of time. When the number stabilizes, or remains below a certain threshold, the software is considered 'good enough'.

During the last decade, the concept of 'good enough' quality was introduced in the software industry. Bach (1997) introduced this paradigm as an alternative opposed to compulsive formalism (Bach 1997).[39] 'Good enough' means:

- The software product has sufficient benefits.
- The software product has no critical problems.
- The benefits sufficiently outweigh the problems.
- In the present situation, and all things considered, further improvement would be more harmful than helpful.

In a practical context, the problem is however to determine, with sufficient certainty, that the software product has no critical problems. Especially for safety-critical systems, where the

[39] Many 'gurus' preach perfection through, for example, process maturity and continuous process improvement, stating that, in this way, all defects can be found or even prevented (Jones 1993; Humphrey 1997).

presence of failures should not be a viable option, the decision to stop testing would require this certainty.

For the present it is assumed that, in practice, software manufacturers follow this approach either consciously or unconsciously. However, it is an important issue in the case studies to reveal how software manufacturers determine the optimal release time in a practical setting. In the next Section, existing theory regarding the estimation of operational cost is presented.

3.3.3 Operational Cost

Software reliability estimation models have received criticism from different angles. It is concluded here, that two higher-order limitations regarding these models also exist:

- *Focus is on cash outflows, not on profit.* The models only take into account cash outflows, assuming that minimizing total cash outflows is the main objective.[40] However, in profit-oriented environments, for example, where software manufacturers sell products to their customers, the expected cash inflows should also be taken into account. In this case the optimal release time would not be determined by minimizing the total cash outflows but by maximizing the difference between cash inflows and cash outflows: maximizing economic value. Few of the models studied incorporate the time value of money.
- *Focus is on pre-release testing versus post-release corrective cash outflows, not total cash outflows*. Considering the total life-cycle cost of a software product, focus should not only be on the short-term operational cost for repairing failures [corrective maintenance cost], but also on the expected future cost for extending the product with additional functionality [adaptive and perfective maintenance cost]. Important factors influencing the long-term maintenance cost are, for example, the quality of the product design [the extent to which maintainability requirements are addressed], the quality of the product realization [the extent to which maintainability requirements are correctly implemented], and the quality of the documentation supporting the product [the extent to which the product is documented in an accessible way: e.g. specifications, design, code, test cases, build procedures].

The majority of resources in many software manufacturer organizations are consumed by software maintenance (Lientz and Swanson 1980; Harrison and Cook 1990; Abran and Nguyenkim 1991; Eastwood 1993; Erlikh 2000). Studies indicate that maintenance costs can be two or four times the initial costs of the system and this proportion is increasing. In 1980, Lientz and Swanson (1980) reported that more than 50% of the total life-cycle costs are devoted to maintenance, in 1993 Eastwood (1993) reported 75%, and Erlikh (2000) estimated this to be more than 90%. Lientz and Swanson found in their survey that around 75% of the maintenance effort was on adaptive [changes in the software environment] and perfective [new user requirements] maintenance, while corrective maintenance [fixing defects] consumed about 21% (Nosek and Palvia 1990). The high fraction of life-cycle costs expended on maintenance stresses the fact that the maintainability of a software product [defined as the probability that, for a given condition of use, a maintenance activity can be carried out within a stated time interval, using stated procedures and resources] is important, especially when it is first developed and released. Releasing software that is difficult to maintain might lead to unexpectedly high post-release costs of corrective maintenance[41] and adaptive and perfective maintenance on software that will soon no longer be able to evolve. This is another important non-functional requirement to take into account when the release trade-off is addressed.

[40] The possibility also exists for using these models to calculate reliability and test time for a cost constraint, or the cost and test time for a reliability constraint.

[41] Corrective maintenance is closely related to the achieved level of reliability. However, being able to correct problems also requires the availability of other artefacts like development documentation.

The *IEEE 1219* standard provides a seven-step model for the process of good quality software maintenance, where the requirements for control, management, planning and documentation are highlighted (IEEE 1998), and the appendix provides methods and tools, and discusses reverse engineering. The standard distinguishes the following categories of maintenance (IEEE 1998):

- *Corrective maintenance*: reactive modification of a software product; performed after delivery to correct discovered faults.
- *Adaptive maintenance*: modification of a software product; performed after delivery to keep the computer program usable in a changed, or changing, environment.
- *Perfective maintenance*: modification of a software product; performed after delivery to improve performance or maintainability. The improvement of performance can be described as *evolutive maintenance* and the improvement of maintainability as *preventive maintenance*.

In this study, the main concern on maintainability is to estimate the post-release operational cost, prior to the release decision. Although many studies exist in this area (Mancini and Ciampoli 1990; Gerlich and Denskat 1994; Jørgensen 1995; Niessink and Vliet 1997; Abran *et al.* 2002; De Lucia *et al.* 2002; Ahn *et al.* 2003), there are no generally accepted, or rigorous and successful, empirically validated models (Koskinen *et al.* 2003). Sneed (1995), for example, develops a method to extend current cost estimation methods to cover the estimation of maintenance costs, but it assumes the availability of certain metrics data and has undergone limited empirical validation (Koskinen *et al.* 2003). The *Maintainability Index* or *MI*, defined by Oman and Hagemeister (1994), gives an indication of how maintainable a software product is. Two equations are available; the second one takes into account the availability of comment in the code [assuming it has a positive influence on maintainability]:

$$MI = 171 - 3.42 \ln(aveV) - 0.23\, aveV(g') - 16.2 \ln(aveLOC) \quad (3.13)$$

$$MI' = MI + 50 \sin \sqrt{(2.46\, perCM)} \quad (3.14)$$

with:

$aveV$	=	*average Halstead Volume per module (related to the number of operators and operands used)*
$aveV(g')$	=	*average extended cyclomatic complexity per module (number of linearly independent test paths)*
$aveLOC$	=	*average lines of code per module*
$perCM$	=	*average percent of lines of comment per module*

However, one of the general problems is the lack of reliable metrics for software complexity – one of the main input drivers for estimation. Inputs like lines of code, function points and cyclomatic complexity all have severe limitations (Kemerer 1995).

IEEE (1988a, 1988b) defines the *Software Maturity Index* or *SMI*, which provides an indication of the stability of a software product and can be used as a metric for planning software maintenance activities. As *SMI* approaches *1*, the product begins to stabilise. In a formula:

$$SMI = [\, M_t - (F_a + F_c + F_d)\,] \,/\, M_t \quad (3.15)$$

with:

M_t	=	*number of modules in the current release*
F_c	=	*number of modules in the current release that have been changed*
F_a	=	*number of modules in the current release that have been added*
F_d	=	*number of modules in the current release that have been deleted*

This index cannot provide an accurate estimate of operational costs, and its main purpose is to demonstrate the evolution of a product over time.

Summarized, the conclusion is that no models are available to support accurate estimations of expected operational cost prior to a software release decision. Collecting and analysing historical data from similar projects is considered an important instrument to support such estimations.

3.3.4 Conclusions on Software Release Decisions

The focus of the literature study on this perspective is on theory about the relationship between cost and schedule in software product development and maintenance. Three types of models are reviewed: cost estimation models, software reliability prediction and estimation models and operational cost models. The cost estimation models, although criticized, are considered of help during the start of projects, enabling a software manufacturer to make trade-offs between cost and schedule and to define realistic objectives. The usefulness of reliability estimation models, aiming at predicting the reliability of the software early in the life-cycle using is considered high if historical data is available from similar projects. The usefulness of software reliability estimation models, aiming at determining the optimal release time, is considered severely limited: most models assume a way of working that does not reflect reality and studies reveal many shortcomings regarding the predictive validity. Estimating operational cost prior to a software release decision is a difficult area, although the majority of resources in many software manufacturer organizations are consumed by software maintenance. The availability of historical data from similar projects is considered important.

The limited usefulness of these models raises questions of how software manufacturers deal with this problematic area of development cost and schedule estimation, optimal release time determination [reliability], and operational cost estimation [maintainability]. Relevant questions identified for the exploratory case studies are:

4. *Which methods are used to estimate development cost and schedule?*
5. *Are different project and design alternatives considered?*
6. *To what extent are reliability requirements defined, deployed and evaluated?*
7. *To what extent are maintainability requirements defined, deployed and evaluated?*
8. *How do software manufacturers estimate post-release operational cost, for short-term corrective activities and long-term product enhancements, prior to the release decision?*

3.4 Decision-making Perspective

Decision-making involves selecting between different alternatives, and selecting the alternative that is most likely to result in attaining one, or more, objectives (Harrison 1987). Decision-making has many aspects and may involve one, or more, decision-makers. In organizations, it is likely that multiple decision-makers will be involved in collective decision-making. Decisions with little impact can often be made on the basis of informal rules and agreements, but as soon as common interests increase collective decision-making has to be institutionalized through a generally accepted decision-making process (Stokman *et al.* 2000, p.1). As the focus in this study is on strategic software release decisions by software manufacturer organizations, more decision-makers [stakeholders] will be involved and, where the characteristics of a *collective decision-making process* are of primary concern here, the elements of a collective decision-making process are investigated.

Another important aspect of decision-making is the characterization of the *decision type*. Decisions can be classified in different ways, for example, for routine and the recurrence of the decision (Harrison 1987); the level of uncertainty involved (Beroggi 1999); the number of times the decision itself may be repeated (Beroggi 1999); and the way information is processed (Rosenberger 2001). These different classifications are investigated.

Because of the presence of an almost infinite number of variables in decision-making, the decision-making process can benefit from the availability of *decision-making models*, with a small number of variables that are significant and understandable (Harrison 1987). An overview of decision-making models most suitable for collective, managerial decision-making is provided.

An important element in the decision-making process is the choice among different options [alternatives]. The best alternative may not always be readily apparent, forcing decision-makers in a collective decision-making process to select one alternative, or find a compromise among competing alternatives. An overview is given of different *models of choice*, a decision-maker(s) can select. Attention is also given to the definition of *decision success*, making a distinction between the quality of the decision outcome and the quality of the decision implementation.

3.4.1 Collective Decision-making Process

Decision-making is defined as the combined activity of comparing alternatives and the act of choice. However, Harrison (1987, pp.38-39) divides a decision-making process into six functions; broadening the scope with preceding and proceeding activities, as illustrated in Figure 3-16.

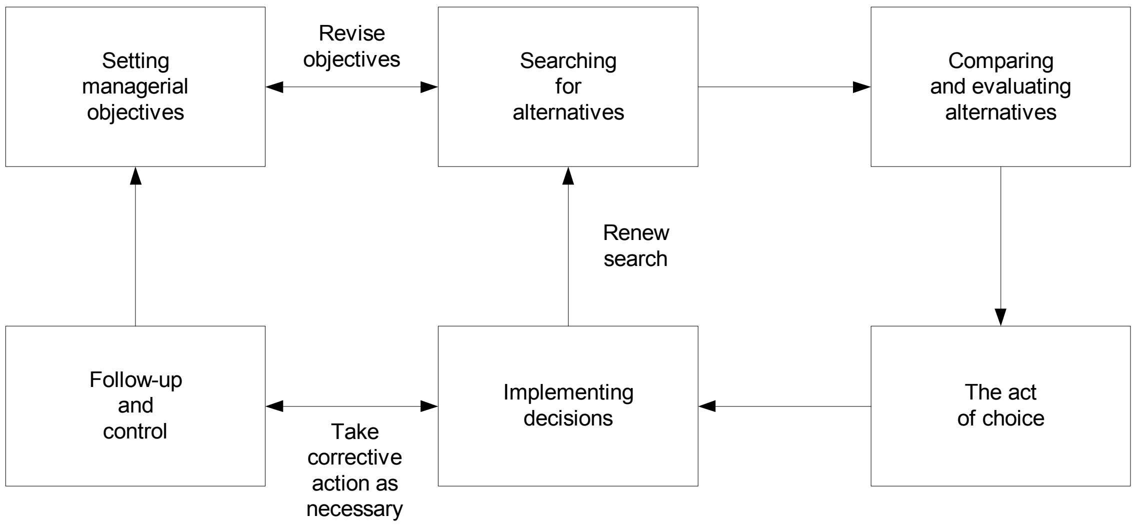
Setting managerial objectives
Revise objectives
Searching for alternatives
Comparing and evaluating alternatives
Renew search
Follow-up and control
Take corrective action as necessary
Implementing decisions
The act of choice

Figure 3-16: The Decision-making Process
(Harrison 1987, p.40)

In this framework, decision-making is illustrated as a dynamic process. Decision-making is considered to be a non-linear, recursive process. That is, most decisions are made by moving back and forth between the choice of criteria or objectives [the characteristics the choice should meet] and the identification of alternatives [the possibilities one can choose from]. The alternatives available influence the objectives applied, and similarly the objectives defined influence the alternatives to be considered (Maneck 1966).

The different functions of decision-making identified by Harrison (1987, pp.38-39) are:

- *Setting managerial objectives*. The decision-making process starts with the setting of objectives, and a given cycle within the process culminates on reaching the objectives. The next complete cycle begins with the setting of new objectives.

- *Searching for alternatives*. In the decision-making process, search involves scanning the internal and external environments of the organization for information. Relevant information is formulated into alternatives that seem likely to meet the objectives.
- *Comparing and evaluating alternatives*. Alternatives represent various courses of action that singly or in combination may help attain objectives. By formal and informal means alternatives are compared based on the certainty or uncertainty of cause-and-effect relationships and the preferences of the decision-maker(s) for various probabilistic outcomes.
- *The act of choice*. Choice is a moment in the ongoing process of decision-making when the decision-maker chooses a given course of action from a set of alternatives.
- *Implementing the decision*. Implementation causes the chosen course of action to be carried out within the organization. It is the moment in the total decision-making process when the choice is transformed from an abstraction into an operational reality.
- *Follow-up and control*. This function is intended to ensure the implemented decision results in an outcome in keeping with the objectives that gave rise to the total cycle of functions within the decision-making process.

The process starts with setting objectives, which requires the search for information or alternatives. The bi-directional arrow between 'Setting managerial objectives' and 'Searching for alternatives' in Figure 3.16 illustrates the dynamics of the decision-making process. Alternatives are compared and evaluated and the selected alternative is implemented. Follow-up and control of the implemented decision reveal to management the actual outcome of the decision, e.g. the degree of congruence between the actual outcome and the expected outcome.

This broader perspective is considered important, as in this research the methodology to be designed includes requirements for pro-active measures, meaning decision-making occurs over a varying span of time, and implementation aspects, to increase the likelihood of success (Trull 1966).

3.4.2 Decision Types

Decisions can be classified in many ways. Four different approaches are discussed here, being decision categories, the level of uncertainty involved, its occurrence interval and the way information is processed. These are important characteristics for the study phenomenon, being software release decisions.

Characteristic	Category I Decisions	Category II Decisions
Classifications	Programmable; routine; generic; computational; negotiated; compromise	Nonprogrammable; unique; judgmental; creative; adapative; innovative; inspirational
Structure	Procedural; predictable; certainty regarding cause/effect relationships; within existing technologies; well-defined information channels; definite decision criteria; outcome preferences may be certain or uncertain	Novel, unstructured, consequential, elusive, and complex; uncertain cause/effect relationships; nonrecurring; information channels undefined; incomplete information; decision criteria may be unknown; outcome preferences may be certain or uncertain
Strategy	Reliance upon rules and principles; habitual reactions; prefabricated response; uniform processing; computational techniques; accepted methods for handling	Reliance on judgment, intuition, and creativity; individual processing; heuristic problem-solving techniques; rules of thumb; general problem-solving process

Figure 3-17: Decision Characteristics
(Harrison 1987, p.21)

Simon (1977) distinguishes 'programmed' and 'non-programmed' decisions; close to the distinction made by Drucker (1967) between 'generic' and 'unique' decisions. Others define a three-point classification (e.g. Gore 1962; Delbecq 1967; Mintzberg 1973) or distinguish four decision strategies (Thompson 1967). Harrison (1987, p.21) concludes that the different classification schemes have much in common and can be reduced to two basic categories, with a categorization of the decision characteristics given in Figure 3-17:

- *Category I decisions*. This category includes the routine, recurring decisions handled with a high degree of certainty.
- *Category II decisions*. This category includes the non-routine, non-recurring decisions handled with a high degree of uncertainty.

The decision category determines the level in an organization where a decision is to be made. Higher management should concentrate on non-routine decisions [Category II], routine decisions should be left to operating management [Category I]. Research shows that the choice patterns of a decision-maker are not necessarily the same for multiple, repeated decisions [Category I], or for unique decisions [Category II] (Keren and Bruine de Bruin 2003, p.350, referring to work by Lopes 1981; Keren 1991; Redelmeier and Tversky 1992). In the remainder of this thesis, the terms 'routine' versus 'non-routine' decisions are respectively used for Category I and Category II decisions.

Although the distinction between routine and non-routine decisions involves the level of uncertainty present [high degree of certainty for routine, high degree of uncertainty for non-routine], it is considered of interest to further detail the notion of uncertainty. According to Beroggi (1999, p.16), uncertainty in decision-making reflects two aspects: uncertainty about the assumptions on the environment of the decision problem and uncertainty related to the evaluation of the alternatives. Three levels of decision-making under uncertainty are distinguished in decision analysis literature (Beroggi 1999, pp.16-17):[42]

- *Certainty*. Alternatives are evaluated without consideration of any scenarios.
- *Informed Uncertainty* [risk]. Alternatives can be evaluated with considerations of some scenarios, whereby the chance of occurrence of each scenario can be quantified with probability, or possibility, values.
- *Complete Uncertainty*.[43] Alternatives can be evaluated with consideration of some scenarios but the occurrence chance of each scenario cannot be quantified.

A third classification of decision types is the number of times the decision itself may be repeated. Beroggi (1999, p.6) makes a distinction between a one-at-the-time decision and repetitive decisions. Although his classification can be compared with routine versus non-routine decisions, in the context of this study it is interpreted as a decision made once for a specific problem [a selection is made among available alternatives] or repeated [making the decision is postponed until the set of alternatives enables decision-makers to select an appropriate alternative].

A fourth perspective in classifying decision types is by looking at the way information is processed. Rosenberger distinguishes simultaneous or compensatory models, using an optimizing process for decision-making using all, or a portion of, the information available, whereas sequential or non-compensatory models use a discriminatory process to decision-making, sequentially using specific information to discriminate alternatives in a systematic fashion (Rosenberger 2001, p.11).

[42] Beroggi (1999, p.16) describes uncertainty in terms of partitions of the total uncertainty space, where, for each partition, different states are identified, while any combination of states is called a scenario.

[43] Note that the term 'complete uncertainty' is used throughout this thesis.

3.4.3 Decision-making Models

Models help to better understand the complex nature of decision-making. In practice, there are no limits to the models developed to serve a specific decision problem. However, on a higher abstraction level models can be distinguished that reflect certain assumptions and behaviour. Harrison (1987, p.152) describes a typology of four decision-making models, relying on the work of, for example, Anderson (1977) and Lindblom (1979):

- ❖ The *Rational Model* (classical). This is a normative model and has its foundation in quantitative disciplines like economics and mathematics. The model is based on the assumption that all significant variables in a decision-making process can be quantified to some degree. As the model operates within an artificially closed environment, it applies mainly to routine [Category I] decisions.
- ❖ The *Organizational Model* (neo-classical). This model combines the behavioural disciplines [like philosophy, psychology and sociology] with quantitative analysis to make a decision that fits the constraints caused by the external environment. It is suited for both routine [Category I] and non-routine [Category II] decisions.
- ❖ The *Political Model* (adaptive). This model is almost totally behavioural, and the primary criterion for decision-making is an outcome that is acceptable to many external stakeholders [bargaining, compromises]. It de-emphasises objective oriented outcomes.
- ❖ The *Process Model* (managerial). This model is oriented toward innovation, and organizational change, with a particular emphasis on long-term results. It is well suited for non-routine [Category II] decisions.

These models represent multi-dimensional perspectives on decision-making. The Rational Model is based on the quantitative disciplines of economics, mathematics and statistics. The Organizational Model combines these disciplines with behavioural disciplines, incorporating the constraints resulting from the external environment. The Political Model is primarily based on the disciplines of political science, philosophy, psychology and sociology, but, whereas the Rational Model is almost totally quantitative, this model is almost entirely behavioural. The Process Model has a strong emphasis on managerial decision-making, especially for cases where uncertainty is high.

The models, although not mutually exclusive, differ from each other in primary decision-making criterion and certain key assumptions, as illustrated in Figure 3-18.

Criterion	Rational (classic)	Organizational (neo-classical)	Political (adaptive)	Process (managerial)
Objectives	fixed	attainable	limited	highly dynamic
Information	unlimited	limited	limited	limited
Cognitive limitations	no	yes	yes	yes
Time and cost constraints	no	yes	yes	yes
Alternatives	quantifiable and transitive	partially quantifiable and intransitive	non-quantifiable and generally transitive	generally non-quantifiable and intransitive
Environment	closed	open	open	open
Outcome	quantitatively limited	qualitatively and quantitatively limited	environmentally limited	objectives-oriented
Time-horizon	short-term	short-term	short-term	long-term

Figure 3-18: Typology of Decision-making Models
(Harrison 1987, p.152)

The type of decision to be taken is an important factor in determining the most appropriate model, especially whether a decision is routine or non-routine.

3.4.4 Models of Choice

An overview is given of different models of choice that can be applied when choosing among different alternatives [third and fourth element in the framework presented in Figure 3-16]. As discussed in Section 3.4.2, models of choice can be classified by looking at the way information is processed: simultaneously or sequentially. The former relates to *compensatory models of choice*, the latter relates to *non-compensatory models of choice* (Rosenberger 2001, p.11). [44]

Simultaneous or non-compensatory models use a portion, or all, of the information available (Rosenberger 2001, p.11). Examples of such models are (Hogarth 1987; Rosenberger 2001):

- ❖ *Linear Dimension Model*. This is a straightforward model. Each dimension, or variable, is quantified and is given a weight reflecting its relative importance. The evaluation of each alternative is the sum of the weighted values on its dimensions. The alternative with the greatest sum for all dimensions is the obvious choice.
- ❖ *Additive Difference Model*. This model compares the alternatives, dimension-by-dimension, whereafter the differences are aggregated to see which of the alternatives is favoured by the aggregate net difference.
- ❖ *Ideal Point Model*. This model is similar to the additive difference model, except that, in this model, the perfect model is taken as a reference and its distance from this ideal point determines the attractiveness of each alternative.

Sequential or compensatory models use sequentially specific information to discriminate alternatives (Rosenberger 2001, p.11). Examples of such models are (Hogarth 1987; Rosenberger 2001):

- ❖ *Disjunctive Model*. This approach to choice seeks the best attribute, or characteristic, presumed to denote the best alternative. The decision-maker will permit a low score on a dimension provided there is a high score on one of the other dimensions. The alternative with the highest rating in its best characteristic is chosen.
- ❖ *Conjunctive Model*. The decision-maker sets certain cut-off points on the dimensions, and any alternative that falls below a cut-off is eliminated.
- ❖ *Lexicographic Model*. The model first ranks the characteristics in order of importance and then selects the best-rated alternative after the most important characteristic. If two or more alternatives rate equally, the next most important characteristic is used.
- ❖ *Elimination-by-aspects Model*. This is a combination of the three above-mentioned models. The model assumes that alternatives consist of a set of aspects or characteristics. At each stage of the process one characteristic is elected, according to a probabilistic scheme, and alternatives that do not include the aspect are eliminated. The process continues until only one alternative remains.

In literature, many variants and specific implementations of the models of choice presented can be found, often referred to as *multi-attribute decision-making models* (MADM) where the set of decision alternatives has been predetermined.[45] Triantaphyllou (2000) presents an overview, and some examples are:

[44] A compensatory model means a decision-maker is willing to allow compensation: a strong performance on one criterion can compensate for a weaker performance on some other criterion.
[45] No attention is given here to multiple objective decision-making models (MODM), where decision variables are infinite and subject to constraints. These problems may be solved through linear programming.

- ❖ Linear model: SMART (Edwards and Barron 1994).
- ❖ Outranking methods: TOPSIS (Hwang and Yoon 1981) and Electre (Roy 1991). [46]
- ❖ Outranking methods with qualitative data: Qualiflex (Paelinck 1976) and Regime (Hinloopen *et al.* 1983).
- ❖ Converting methods [subjective assessments of relative importance to a set of overall scores or weights by pair-wise comparisons]: Analytical Hierarchy Process (Saaty 1980) with the sub variants REMBRANDT (Lootsma 1992), and MACBETH (Bana e Costa and Vansnick 1994).[47]
- ❖ Fuzzy MCA methods (Chen and Hwang 1992).

Another technique to compare alternatives is a decision-tree combined with utility theory, like the *Subjective Expected Utility Model* or SEU-model (Savage 1954). The decision tree diagrams the paths of possible courses of action. For each path the tree displays the probability for an outcome or event and the end-nodes display the valuation of the payoff for each outcome.
Suppose, for example, a researcher has to decide how to fund his research project. He is faced with two possibilities: apply for funding or go to a casino. By assigning probabilities to the different outcomes [funding: 0.20 application accepted, 0.30 application partially accepted, 0.50 application rejected; casino: 0.01 jackpot, 0.99: no jackpot] and values [0.70: funding application fully accepted, 0.40: application partially accepted, 1.00: jackpot], it is possible to calculate that applying for funding is the best alternative, as in Figure 3-19: [48]

$$0.20 * 0.70 + 0.30 * 0.40 + 0.50 * 0.00 > 0.01 * 1.00 + 0.99 * 0.00 \qquad (3.16)$$

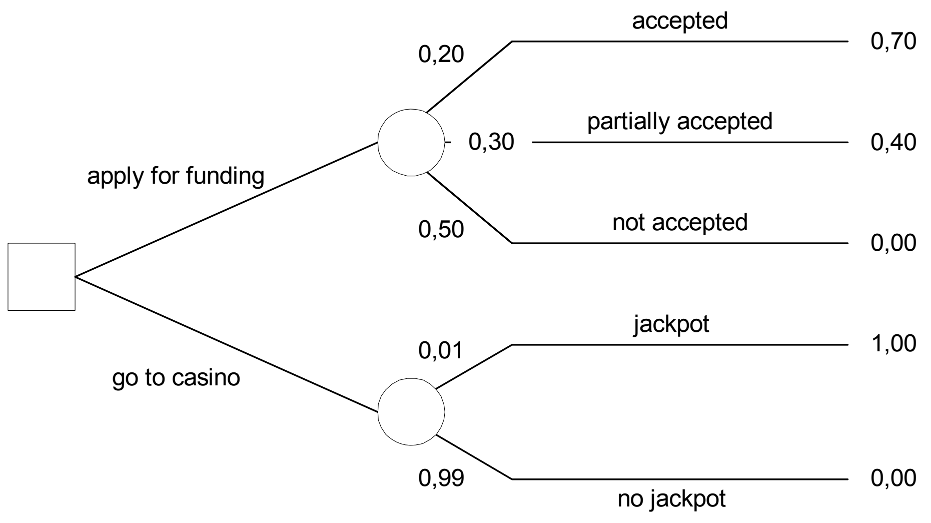

Figure 3-19: Example of a Decision Tree
(SEU-model)

Combinations of multi-criteria analysis methods and SEU-models or probabilistic statistical techniques are described (Keeney and Raiffa 1993).

Despite the wealth of available models of choice, there is little consensus on the model best suited to a specific situation. Guitouni and Martel (1998) note that most users select a model they are familiar with, rather than the optimal method, assuming that such a method exists.

[46] Option A outranks Option B if there are enough arguments to decide that A is at least as good as B, and there is no overwhelming reason to refute that statement.
[47] These methods are primarily used in a single decision-maker context.
[48] Note that, in theory, both options require an initial investment, to be subtracted from the end result.

3.4.5 Decision Success

In Section 1.6, the term decision success was introduced, defined as a decision that results in the attainment of the objective that gave rise to the decision, within the constraints that had to be observed to bring out such attainment (Harrison 1987, p.346). This refers to the extent to which a decision contributes to the achievement of organizational goals. Decisions with strategic value are important to an organization as they have important consequences, and resource demands, for the organization (Nutt 1998). The important concept to grasp is that decision success is not related to its outcome: a good decision can have either a good or a bad outcome. Similarly, a non-successful decision can still have a good outcome. In this thesis the quality of the research phenomenon, e.g. strategic software release decisions, is defined as the sum of the quality of the decision-making process and the extent to which the decision is successfully implemented (Dooley and Fryxell 1999), so there is an acceptable degree of congruence between the actual outcome and the expected outcome, irrespective of whether the actual outcome can be assessed as good or bad. It is assumed in this thesis that decision success requires a high quality of both decision process and decision implementation:

decision success = quality of decision outcome + quality of decision implementation

This definition is considered true for decisions with, or without, strategic value.

3.4.6 Conclusions on Software Release Decisions

The focus of the literature study on this perspective is on theory concerning decision-making, addressing the element of a collective decision-making process, classifications of decision types, decision-making models, model of choice, and decision success. At this stage, no link is made to the study phenomenon itself, as no literature was found applying decision-making theory to software release decisions.

This first orientation toward decision-making is used to identify a number of relevant questions for the exploratory case studies, to gain insights into software release decision-making in a practical context:

9. *To what extent is a formal collective decision-making process applied to software release decisions?*
10. *How can a software release decision be characterized?*
11. *Which decision-making models apply to software release decisions?*
12. *Which models of choice are used for software release decisions?*

3.5 Summary and Conclusions

In this Chapter, existing theory on software release decisions is discussed. Three perspectives are highlighted, namely economics, software management and decision-making.

For the economic perspective, it is concluded that the market entry trade-off is complex. Important factors influencing the trade-off are the external environment, the organization's capabilities and the business strategy chosen. The presence of a multitude of factors in a practical setting strongly indicates that no unambiguous, overall product development strategy can be defined for a specific manufacturer type or a specific software market. The dynamics of product development resulting from a changing external environment and/or unplanned project performance are likely to make each software release decision unique.

From the software management perspective, three issues are addressed. For the predictability of development costs and schedule, the usefulness of the software cost estimation models is concluded as being limited towards the end of product development, when the project is faced with the release decision itself. They do offer the possibility of support in defining project objectives prior to product development and evaluating them during product development. In determining the expected reliability based on defects found in different project phases, the reliability estimation models can potentially be used under the condition that historical data from similar projects is available. In determining the optimal release time [with focus on trading off reliability against time-to-market], the usefulness of existing software reliability prediction models is questioned. For the estimation of post-release operational cost, no generally-accepted or rigorous and successful empirically-validated models were found. The collection and analysis of historical data ffrom similar projects is recommended. These findings complicate software release decisions.

Decision-making is reviewed from a theoretical point of view. The elements of a managerial decision-making process are identified, as are different classifications of decision types. Available decision-making models differing from each other with respect to the primary decision-making criterion and key assumptions are discussed. For models of choice, various compensatory and non-compensatory models are presented, complemented with a combination of decision trees and subjective utility theory as another technique to compare alternatives.

Exploration of the study phenomenon from these three perspectives leads to the identification of a number of questions as input for the exploratory case studies. The results of these case studies, providing answers to the questions raised, are presented in the next Chapter.