Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2011, Article ID 579292, 21 pages
doi:10.1155/2011/579292

Research Article

The Study of Resource Allocation among Software Development Phases: An Economics-Based Approach

Peleg Yiftachel,[1,2] Irit Hadar,[3] Dan Peled,[4] Eitan Farchi,[5] and Dan Goldwasser[6]

[1] *Caesarea Rothschild Institute for Interdisciplinary Applications of Computer Science, University of Haifa, Mount Carmel, Haifa 31905, Israel*
[2] *Israeli Center of Excellence, EMC Cooperation, 7 Hamada Street, Herzelia 46733, Israel*
[3] *Department of Information Systems, University of Haifa, Mount Carmel, Haifa 31905, Israel*
[4] *Department of Economics, University of Haifa, Mount Carmel, Haifa 31905, Israel*
[5] *Group of Software Performance Analysis Reviews and Quality (SPARQ), IBM Haifa Research Laboratory, Haifa University Campus, Mount Carmel, Haifa 31905, Israel*
[6] *Department of Computer Science, University of Illinois, Urbana-Champaign, IL 61801, USA*

Correspondence should be addressed to Peleg Yiftachel, peleg.yiftachel@gmail.com

Received 10 August 2011; Accepted 1 November 2011

Academic Editor: Gerardo Canfora

This paper presents an economics-based approach for studying the problem of resource allocation among software development phases. Our approach is structured along two parallel axes: theoretical and empirical. We developed a general economic model for analyzing the allocation problem as a constrained profit maximization problem. The model, based on a novel concept of *software production function*, considers the effects of different allocations of development resources on output measures of the resulting software product. An empirical environment for evaluating and refining the model is presented, and a first exploratory study for characterizing the model's components and developers' resource allocation decisions is described. The findings illustrate how the model can be applied and validate its underlying assumptions and usability. Future quantitative empirical studies can refine and substantiate various aspects of the proposed model and ultimately improve the productivity of software development processes.

1. Introduction

Fundamental disagreements often arise with regard to the "correct" allocation of resources to various software development phases (SDPs). For example, persuasive arguments are made for devoting substantial effort to requirements analysis and design, in order to avoid the costly consequences of modifications in later development stages [1–4]. However, the pressure to provide an executable product, which can be tested and presented to the customer sooner rather than later, can sometimes constitute a consideration for shifting resource allocation to implementation instead. Yet another approach can be seen in Test-Driven Development methods [5, 6], where much emphasis is placed on testing, both before and after implementation.

These tradeoffs are relevant when analyzing graphs showing the cost of detecting and correcting a fault as a function of the phase in which it is detected. Such graphs—prevalent in the literature—demonstrate the dramatic increase in cost when defects are detected later in the development process [7–9]. This observation is often used to support the claim that more resources should be allocated to early SDPs. Obviously, this claim can be pursued ad absurdum: dedicating all (or almost all) resources to the requirements and design phases would leave insufficient resources for implementation. Yet developers[1] cannot find any guidance as to how many resources should be allocated to the various SDPs.

Why are such fundamentally different approaches being advocated for allocating resources to SDPs? Does a correct allocation for a given software project exist, and can it be identified?

Misallocating resources among SDPs can have serious consequences. Tens of billions of dollars are estimated to be spent annually in the US alone due to software faults or

aborted projects [7]. Such huge costs may be attributed to various factors such as lack of understanding of the customer's domain or inadequate software development tools. The contention of this paper is that a large portion of these costs may result from incorrect allocation of software development resources among SDPs (see also [8]).

The most popular model dealing with economic perspectives of software development is the Constructive Cost Model (COCOMO) [2], which focuses on cost estimation. While its goal is not optimizing the allocation of resources among the SDPs, this model uses empirical data on software development costs to estimate the cost of any given project. The estimate is based on the project's characteristics and those of the developing environment. The COQUALMO model [10], which is based on COCOMO, discusses relationships between costs, schedule, and quality, but without addressing the resource allocation problem. Emam [11] emphasizes the importance of quality for profitability. Some works (e.g., [1, 7, 12]) present ex-post surveys of the actual resource allocation in different software projects. Heijstek and Chaudron [13] measure effort distribution among the SDPs according to the Rational Unified Process (RUP) hump chart. While much can be learned through hindsight from such cases, they do not provide a basis for a comprehensive method of up-front resource allocation planning. In contrast, Babu and Suresh [14] suggest how to allocate limited development resources across M development phases in order to maximize the average or the minimum of P quality factors, subject to various constraints. The problem is cast as a static nonlinear maximization problem, which assumes known functional forms and parameter values (not based on empirical data), but ignores crucial factors such as the iterative nature of modern software development processes, and omits various quality aspects from the objective function of the problem analyzed.

Value-Based Software Engineering (VBSE) [15] considers economic aspects within the entire software development lifecycle. It can be defined as a software development paradigm in which business value considerations are engineered into software processes, best practices, activities and tasks, management and technology decisions, as well as tools and techniques used throughout the software lifecycle.

Biffl et al. [15] surveyed different methods for solving multicriteria decision problems, presented them as constrained optimization models, and discussed their applicability to decisions in software engineering by analyzing, for example, whether a specific requirement should be included in a specific project/release. However, we did not find in the literature any empirically based decision making model that concretizes these generic optimization models to the specific problem of resource allocation across SDPs.

Some VBSE-related works address subproblems of the one we focus on. For example, Huang and Boehm [16] present a value-based approach for determining how much software assurance is enough before release. Jalote and Vishal [8] address the problem of allocating resources to different quality control stages, in order to minimize the (scalar) measure of the numbers of bugs to be removed during quality control.

To address the absence of guidelines on how to allocate resources to various SDPs, we propose here a conceptual framework for studying the problem of resource allocations among SDPs. While there are several types of resources, we refer here to the time invested by developers as a basic resource. Basing such a framework on scientifically sound principles while making it practical at the same time is a real challenge. In fact, well-known sources [17, 18] suggest that there are no silver bullet solutions to such essential software engineering problems.[2] Moreover, even if a solution does exist, it is likely to depend on specific product and developer characteristics and thus lack generality. Finally, studying this problem empirically poses many difficulties associated with defining, identifying, and measuring the variables. For example, how should we define and measure the output of the software development process, a necessary metric for comparing the consequences of alternative allocations of development resources?

The difficulties of the problem do not absolve us from recognizing its economic importance and should not curb efforts to resolve it. Our goal in this paper is to suggest an economics-based approach for studying the problem of resource allocation among SDPs.

A note on terminology: there is no common, standard terminology in the literature for the different activities and their classification according to what we call phases. For our purpose, it is important to distinguish between different activities, which are concrete instances of development work that need to be associated with one of the phases they belong to. Section 4 deals with this topic explicitly.

The rest of this paper is organized as follows. Section 2 provides a detailed overview of our research approach. Section 3 presents the economic model of software development, which captures the interplay between the allocations of development resources and resulting output. Section 4 proposes how to define and measure SDPs, and Section 5 proposes how to define and measure the resulting outputs. Section 6 analyzes the relationships among these variables. Section 7 illustrates how the model can be used to analyze incorrect resource allocation decisions when developing software in an empirical setting. We conclude and suggest how this research approach can be further developed in Section 8.

2. An Overview of Our Approach

Developing an appropriate analytical framework for optimizing resource allocations among SDPs should be based on a suitable formal theoretical model that can be evaluated empirically, as in other VBSE research [15]. In our case, the theoretical model should formulate the resource allocation problem in a manner that captures the consequences of alternative feasible allocations of resources among SPDs for given objectives of the decision maker.

Our approach for studying this problem is structured along two parallel axes: (1) *the theoretical axis* consists of formulating the problem as an economic optimization model, based on characterizing the model components in a precise and measurable manner; (2) *the empirical axis* is designed

as an empirical environment in which case studies are used to explore and guide the construction of the theoretical structures and provide an empirical basis for refining the model and the characterization of its components.

Ultimately, the approach should generate a working model that can be used to prescribe the allocation of resources to the various SDPs in a manner that efficiently achieves the objectives of the software developer. This section provides a bird's eye view of our approach along these two axes.

(1) The Theoretical Axis. We present a novel formulation of an economic optimization model for software development and make some headway in characterizing its components. This model, detailed in Section 3, allows us to analyze the problem of resource allocation among SDPs as a constrained value maximization problem.

For this conceptual model to be useful, its various components should be carefully characterized. In particular, (a) variables representing inputs should correspond to work performed in different development phases; (b) variables representing outputs should correspond to artifacts of the inputs; (c) the *software production function* (to be defined in Section 3) should capture the relationships between inputs and outputs (these components will be analyzed and discussed in Sections 4–6). Of course, the different variables, that is, the inputs and outputs of the model, must be measurable, which poses a challenge because software studies are known to suffer from nonstandardized measurements [19].

(2) The Empirical Axis. We strive to create an empirical environment which will enable us to identify, quantify, and verify the key components of the economic model. That is, we develop an empirical research method and construct appropriate measurement tools, in order to obtain a quantitative representation of the variables in the model and the relationships among them. We demonstrate the feasibility of this empirical track and along the way develop solutions to some of the critical challenges involved in implementation of our model. For this aim we adopt the qualitative research approach and conduct an exploratory study. When aiming to explore and understand a phenomenon and its different aspects, rather than statistically corroborating a hypothesis or a theory, it is appropriate to use a qualitative research approach [20]. Qualitative research approaches have gained recognition in general in empirical research, and specifically in software engineering research (see [21], e.g.). These approaches are appropriate when studying software engineering aspects related to human behavior [21] as indeed is done in this research.

Employing qualitative research methods (see, e.g., [22]), we construct the first generation of research tools and illustrate how they can be used in the future for gathering the empirical data needed to complete the mathematical specification of the model and evaluate its usefulness. The data gathered in this study allows us to (a) verify the model's utility in interpreting developers' actual conduct during the software development process: (b) obtain some initial information about the general nature of the functional forms appearing in the model, and (c) understand the possible reasons for observed deviations in software developers' conduct from the model's prescriptions. Such an exploratory study is important for guiding further research activities [19] and towards future quantitative research.

Our approach emphasizes the need to base the precise specification of the *software production function* on empirical observations. Accordingly, as part of this research, we have developed an environment for *empirical observation* in which four subjects, each serving as a case study, worked independently on developing a software project, while we monitored and measured their activities during the entire software development process.

The participants in this study were undergraduate seniors in a management information systems (MISs) department, enrolled in the course "Human Aspects of Software Engineering." The students individually developed a product, called the taxi ordering server (TOS), based on *identical* requirements and constraints (see Appendix A). We provided the participating *developers* an explicit *profit* objective, a function of the *quality* of their finished software product. We monitored and recorded the time allocated by each participant to the different development phases. For this purpose, we developed a tool, called *Econometric*, which automatically records the time spent in each development environment (e.g., SRS document, UML coding CASE tool, unit testing environment). *Econometric* also included a reflection document filled in by the students each time they moved between phases, explaining what they had accomplished in their recent activity, to which phase they planned to move next, and why. The students submitted intermediate versions of their work-in-progress every two weeks.

For evaluating the output of the students' work, we developed a *Quality Checker* (QC). The QC calculates several quality factors of the developed software. Given a prespecified objective function and based on these factors, the QC graded the students' projects. This grade was a significant component of the final course grade, hence served as a value for the students and provided them an incentive to optimize their output. Based on these evaluations, we analyzed the impact of the time allocated to each phase on the quality of the resulting software product. The economic model and its components' characterization were iteratively refined based on the data obtained during the empirical study. For more details about the empirical study, see Appendix A and [23].

This environment generated very detailed and multifaceted data, which allowed us to study the actual allocations of work time among SDPs and relate these allocations to developers' capabilities and expected personal reward. In Sections 4, 5, and 6, we present data from this empirical study and demonstrate how they feed back into the construction of the model.

While this observational environment is admittedly artificial, and unlikely to be available in real-life situations, it is highly informative and useful for constructing this custom-made research methodology for investigating the problem of resource allocation among SDPs. We view this study as the

start of an ongoing effort to formalize the description of software development for the purpose of enhancing its productivity. As we accumulate more empirical data about resource allocations and their resulting output, the models and their components will be refined and perhaps customized to different types of software products. This interplay between theory and empirical findings will eventually lead to better utilization and productivity of software development resources.

3. A Value-Based Model for Resource Allocation among Software Development Phases

In this section we present a theoretical framework for modeling allocation of resources among SDPs. The model presents a desirable allocation of resources as a solution to a value-maximizing optimization problem, in the spirit of Value Based Software Engineering (VBSE). VBSE is a relatively young discipline, which has been successfully applied to various aspects of software design and management [15]. However, while VBSE has successfully integrated various theories such as utility theory, decision theory, and dependency theory to address particular software engineering problems, it has not yet applied elements from *economic production theory*, which this paper does. In fact this paper is the first to apply the VBSE approach to the study of resource allocation among SDPs.

The following two subsections describe two different versions of the model; the second one should be viewed as a dynamic extension of the first.

3.1. Software Development as a Production Process: The Static Model. We borrow from economic theory the idea that any production process can be represented as a function which maps production factors (inputs) into producible output [24]. However, software development cannot be evaluated by *physical* produced units, as is typically done in economics. Instead, we consider *software development output* as consisting of a set of features, each with its own quality, and denote this output by Q. This notion of quality is *external quality* [25], as viewed from the customer's perspective, and is one of the main drivers of the developer's *value* from the product. For presenting the economic model in its simplest form in this section we treat Q as a scalar. In fact, this output has multidimensional structure (see details in Section 5). However, nothing in the model construction depends on this scalar representation.

Our model focuses on the case in which the resource is development time, to be allocated to work on four different development phases. A basic assumption underlying the model is that the same output can be developed with different combinations of inputs, albeit with different costs. Accordingly, we define $Q = f(t_1, t_2, t_3, t_4)$ as the *software production function* that maps work time inputs on development phases into the software development output.

A general concept of software production function was used in its equivalent dual cost function form by several sources (e.g., Boehm et al. [2, 3], Hu [26], Pendharkar et al. [27]). They present the effort needed to develop software as a function of its size and possibly attributes of the developing organization. Since our focus is the impact resource allocation on software development output, we revert to the traditional economics concept of production function and extend it in two ways relative to these aforementioned references. First, we include both size and quality attributes in our characterization of software development output. Second, we distinguish between efforts extended on different development phases as distinct production factors. In order to capture the flow dynamics of inputs and output in software development, Madachy [28] presents a general dynamic software production function, without distinguishing input by development phases. The idea that the allocation of resources across development phases is linked to the quality of the resulting software was first presented by us in [29]. Heijstek and Chaudron [30] provide empirical evidence on the effects such allocations have on defect detection during development.

The inputs (t_1, t_2, t_3, t_4), respectively, denote the time allocated to each of the following four development phases: Requirements, Design, Implementation, and Testing, respectively.[3] The function f can also depend on additional developer-specific human and organizational factors. As explained in [23, Chapter 5.2], such factors can be based on COCOMO II scale factors and effort multipliers [2]).

The value of the output Q to the developer depends on how Q is perceived by the relevant stakeholder. The identity of the stakeholders can vary from customers purchasing the software to division managers in a large software development organization. We assume that the developer is trying to maximize a *value function*, denoted by $V(Q)$, which is increasing in Q. This value function captures all the factors that constitute the developer's reward for the developed product. For instance, $V(Q)$ could be the monetary income generated by selling a software product with properties represented by Q. Other factors affecting product demand, such as advertising and promotion, or deployment of the software product in the client organization, are currently ignored in this model, which focuses on the software development process. Alternatively, $V(Q)$ can be the salaries, bonuses, and income equivalents of promotion opportunities enjoyed by a software developer working in a large organization.

The essence of the value-based software development optimization problem consists of maximizing $V(Q)$ subject to a budget constraint on the total costs of the inputs chosen by the developer. While it is always possible to improve the resulting product along some dimension, the developer will seek to utilize the budget by choosing a combination of inputs that contributes the most to $V(Q)$.

The relationship between the inputs and the resulting output is at the heart of this approach, and we elaborate accordingly on how it can be constructed. While the available resources can be allocated in different ways in the course of the development process, this will affect both the development output and its costs. To illustrate how one can capture the intricate relationship between work on different development phases and the resulting software product, we consider an alternative representation of the same software production function in terms of development phase artifacts.

Let qi denote the quality of the artifact of phase i, $i \in \{1, 2, 3, 4\}$, and define the software development output as $Q = \hat{f}(q_1, q_2, q_3, q_4)$.

These phase qualities correspond to *internal quality* as defined by [25] and are further explained in Section 5. Since the resulting output of each phase depends on the time allocated to that phase and may also be affected by the qualities of the artifacts of other phases, this alternative representation of the input-quality relationship is equivalent to the former in terms of working time on each of the four phases. The dependence among the artifacts of different phases is modeled by assuming that $q_1 = h_1(t_1)$ and $q_i = h_i(t_i, q_{i-1})$; $i = 2, 3, 4$.

This formulation emphasizes the interdependence of the artifacts from different phases, as well as the fact that the resulting artifact of each phase depends on the resources allocated to it. With w_i denoting possibly different costs associated with inputs to different phases, the budget-constrained value-maximizing problem is formulated as

$$\underset{t_1,t_2,t_3,t_4}{\text{Max}} \left\{ V(Q) \mid \sum_{i=1}^{4} w_i t_i = b \right\}, \tag{1}$$

where $Q = \hat{f}(q_1, q_2, q_3, q_4)$; $q_1 = h_1(t_1)$, $q_i = h_i(t_i, q_{i-1})$, $i = 2, 3, 4$, and b is the given budget for the project.

The assumption that the project's budget will be exhausted holds whenever additional development work has a positive marginal effect on the product and an unused portion of the budget does not increase the developer's value. One can easily convert this constrained optimization description of the problem to an unconstrained problem, where no budget constraint is given to the developer and the inputs are chosen to maximize the value net of development costs. For such a version of the model, see [23].

The relationships between the different components of the model are summarized in Figure 1.

The schematic views presented in Figure 1, as well as its underlying economic model, are abstract and can be customized to suit any level of the software developing entity, by specifying appropriate rewards and cost functions.

For an individual developer or team leader, Q can include the amount of features under his responsibility and their quality. The value $V(Q)$ will include the developer's salary and bonuses, which are assumed to be influenced by the work performed. The cost will include the alternative uses of the developer's time.

For a software developing firm, $V(Q)$ will typically include the income generated by a development output Q, and possibly the indirect market positioning gained by having such a product available in the market at that time. The cost, at this level, is the budget devoted by the firm for developing the given product.

3.2. The Dynamic Economic Model. Capturing the dynamic nature of software development is important for accommodating current iterative development methodologies, including incremental and even lean development such as in agile methodologies. The production function $\hat{f}$ presented in the previous subsection ignores the fact that the software development output is also influenced by the dynamic allocation of resources to different phases, rather than just by the aggregate amount of resources allocated to each phase. The richness of the dynamics of software development is explored in depth by Madachy [28].[4] Accordingly, we describe in this subsection the essence of a dynamic version of that model, which is presented in full in Appendix B. The dynamic allocation of resources can be derived by solving a dynamic programming problem of allocating resources to development phases.

This dynamic allocation is based on an incremental quality improvement function, which describes the transformation of an existing software development output Q into its next state, Q', as a result of applying the subsequent unit of development resources to one of the phases, $Q' = Q + g(Q, a)$. Here $a \in \{1, 2, 3, 4\}$ indicates to which of the four development phases the next resource unit is allocated, and $g(Q, a)$ is the change in the development output Q resulting from that allocation. Consequently, the output will be affected by the order in which subsequent resource units are allocated to the different phases, not just by the total allocated to each phase. Moreover, the dynamic allocation of development resources also allows us to stop the development process at any stage. As long as there are remaining resources available for further development, the optimal decision at each stage involves two choices: (1) to stop further development and release the product in its current form or continue the development process, (2) if development continues—to which phase to allocate the next unit of development time.

This dynamic allocation of resources to different development phases can be presented as a decision tree, as in Figure 2. The allocation of each subsequent unit of development resource to one of the phases depends on the current state of the development process, takes into account the implications of allocating the next unit to one of the phases, and recognizes that decisions on allocating subsequent resource units will be made in the future. This is the essence of the dynamic programming approach we propose for the dynamic model. The formal dynamic model can be viewed as an optimization formulation of a certain type of System Dynamics model advocated by [28].

The main advantage of allocating resources dynamically is that it takes into account, by construction, the path of development work on each of the four phases, and not just the aggregate time spent on each phase. Furthermore, this dynamic model can be extended by including random effects that influence the transition of the development output Q or changes in available resources or the configuration of features to be included in the software product. However, in order to construct this dynamic programming formulation, one must have a valid description of the incremental quality improvement function, g, at each possible state of the software development output index.

4. Characterization of the Software Development Phases

This section characterizes the SDPs and suggests a procedure for associating each activity during the software development

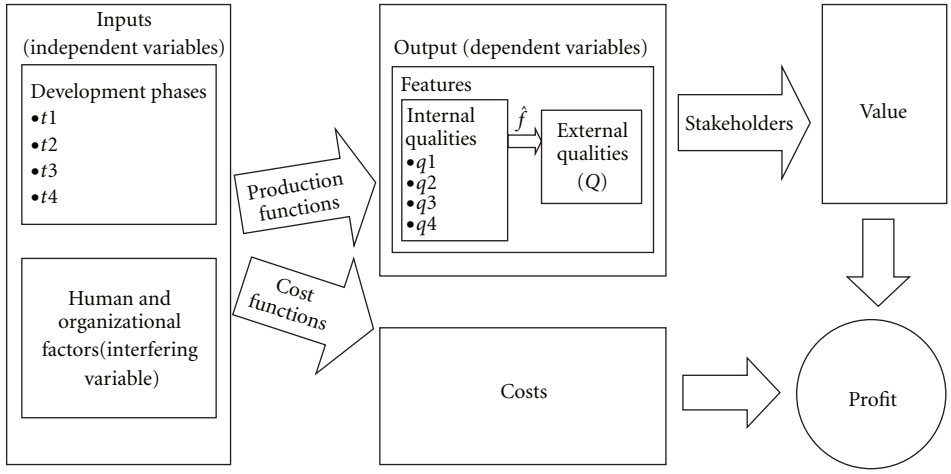

Figure 1: A schematic view of the economics-based approach to allocating software development resources among SDPs.

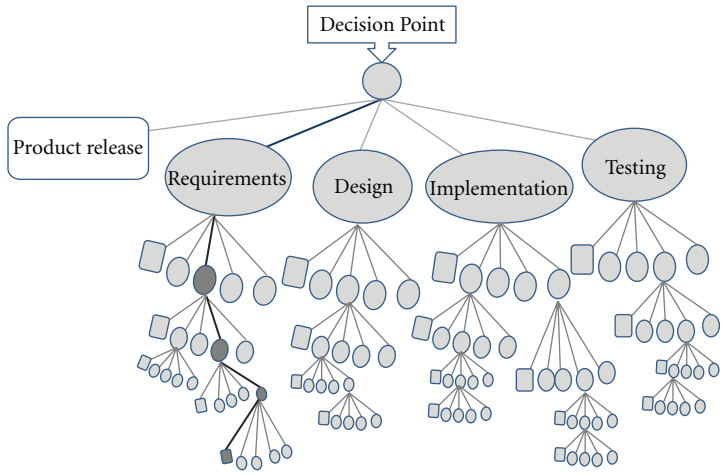

Figure 2: A schematic view of dynamic allocation of resources among development phases.

process with one of these phases. It further identifies two properties of the phases, which we assume to have influence on the process output.

Several well-known sources in the literature have categorized the software development phases in different ways (e.g., [2, 9, 31]). First, they have provided somewhat different lists of the phases participating in the software manufacturing process. Second, they have provided different definitions regarding which development activities belong to each phase. For example, the requirements and design phases are commonly distinguished based on viewing the requirements phase as dealing with *what* the software should do, while the design phase deals with *how* it should be done (IEEE definition, [32]). However, as pointed out by Berry [33], there are cases where it is not possible to precisely specify the *what* without going into the *how*. In such cases, attempts to measure work time allocated to each phase require a rule for determining whether a particular observed development activity should be considered as Requirements or Design. This problem is not confined to distinguishing between these two phases. It holds true as well for the Design and Implementation (e.g., writing pseudocode), Implementation and Testing phases (e.g., debugging/unit testing), and Testing and Requirements (e.g., writing tests prior to development, such as in XP methodology).

Our approach is based on identifying each atomic development activity and classifying it to one of the different development phases according to several criteria. To this end, we need a conclusive definition of the development phases and the boundaries between them, such that (1) each development activity can be classified to exactly one phase, (2) the time allocation for each phase can be measured, and (3) the quality of artifacts of each phase can be defined and measured.

To achieve this objective, each development activity was classified as belonging to a specific development phase according to the *issue* dealt with by the developer. We relied on the observation that each software development activity deals with one of the following four possible *issues*: *what* is to be done, *how* it should be done, *carrying out* the above *how*,

and *checking* the results. Accordingly, every development activity can be uniquely mapped to one of the following phases:

(1) *Requirements*: the set of activities that result in a definition of *what* the software should do;

(2) *Design*: the set of activities that result in the definition of *how* the software should meet its requirements;

(3) *Implementation*: the set of activities that result in the computer program which is the physical, executable artifact that will eventually be delivered to the customer;

(4) *Testing*: the set of activities whose goal is to verify the implemented artifacts.

Note that activities aimed at validating the requirements or design, such as requirements review and design review, are included in the requirements and design phases, respectively.

The definitions aforementioned are based on known standards [32] and common knowledge.[5] However, as explained previously, it is still difficult to uniquely assign each activity to one of the phases. Some refinements are necessary to resolve this fuzziness and sharpen the boundaries between the phases. In what follows, we propose a direction for such refinements.

For example, with regard to the boundary between Design and Implementation, we classify everything that can be represented in the UML class diagram [9] semantics as Design, whereas anything more specific is part of the Implementation. More specifically, definitions of class hierarchy, method signatures, and class data members are classified as design, and more detailed code is classified as implementation. According to this distinction, if a piece of code is composed of one long method, for example, we assume that less design has been done than in a case in which the same functionality is composed of several shorter methods.

Unlike most works in the literature (e.g., [1, 2, 31, 32]), here we distinguish between coding and implementation. We see this distinction as the distinction between the working environments used and the actual phase to which the development activity belongs. For example, when coding a method signature, the classification of this activity to the relevant phase depends on the context. If this signature is coded according to a design document created earlier, the developer merely implements what has been defined in the design; hence this coding would be classified as an Implementation activity. However, if the developer codes the signature of a method without predetermining it in the Design, this activity should be classified as a Design activity.

This classification can be extended in light of Ralph and Wand [34], who suggest that the working phase is derived from the existing technology that can be used by the developer. If the developer deals with choosing between existing technologies, for example, implemented objects, it is considered Implementation. If the developer deals with technology that does not yet exist, for example, an object planned but still not implemented, it is considered Design. Furthermore, according to [34], we can view the collection of all the possible things we can do with the technology primitives as all the possible design alternatives, while the requirements are a constraint on this collection.

Another difficulty in associating activities with development phases stems from the developer's perspective. Consider, for example, the boundary between Requirements and Design. The requirements of a system component are actually part of the system level design. Similarly, the requirements of subcomponents constitute part of the component design. In the taxi ordering server (TOS) project used in our empirical study (see Appendix A), for example, when thinking about *how* to build the system so it will meet the requirements, a decision to include a Parser component might be reasonably considered to be a part of this *how*, and hence a Design activity. The parser's responsibility is to check if a given text input is syntactically correct and, if it is, to build a data structure, namely, a logic representation of the text. Looking at the exact same activity when focusing on the Parser development leads to view it as requirements specification, since it deals with the capability required from this component.

In order to address this concern, we refine the above phase definitions by distinguishing between the developments phases with regard to a specific decomposition level of the system discussed. Decomposition, according to Paulson and Wand [35], is "the breakdown of a complex system into smaller, relatively independent units. It is the main tool available to simplify the construction of complex man-made systems." In our context, viewing the system as a whole, as required by the customer, defines the lowest decomposition level (level 0). Each successive breakdown, first to the system's direct components, then to their subcomponents, and so forth, defines a higher decomposition level. We define that each development activity is classified based on the *issue* it deals with, with respect to a specific decomposition level. Accordingly, for a given decomposition level, our definition resolves the fuzziness between the definition of *what* (Requirements) and *how* (Design). We can similarly use this decomposition analysis when assigning unit-testing activities.[6] When considering *checking* of given software unit from the unit perspective, any test with regard to that unit is considered as Testing. However, viewing the same activity from the entire system's perspective will be considered as Implementation. This approach is similar to the one presented in [15, Page 160], where the analysis of project value requires multiple views that are painted using multiple measurement models. Each measurement model is like a lens, where each *lens* is used by an observer in order to make inferences about the properties of an object of interest.

After classifying the phases, we further characterize the development activities according to additional attributes. While there may be various such attributes, we find the following two to be the most relevant in terms of affecting the output:

(1) *the environment in which the work is conducted.* Word editor, UML tool, programming IDE, and so forth,

(2) *latitude: the existence/nonexistence of choice regarding the next activity.* Accordingly each activity can be described by one of the following:

(a) process-forced activity: a situation in which there is only a single choice for the next action at a given point; Such activities can be described in the context of our dynamic model (presented in Section 3) as contemporaneous choices that avoid obviously inferior outcomes;

(b) strategically chosen activity: the situation where there is more than one possibility for the next activity (that will positively affect the product's quality).

For example, at the beginning point of a project, the requirements phase is usually the only choice. This is a process-forced activity. However, after processing the first requirement, one has a choice to continue processing the next requirement or designing/implementing the system to meet the requirement already processed. A more common example is making Design/Requirements decisions during Implementation. This can be a *process-forced activity* if the developer gets stuck when writing the code, for example, when deciding what to write in the *else* condition of an *if-else* statement, or a *strategically chosen activity* when the developer decides in advance to leave a certain design decision to be made during Implementation.

In our empirical study, the above described attributes were identified and various combinations of them were found. In what follows we present several examples as written in the students' reflection documents (additional examples appear in [23]). Note that the emphasis in the texts was added by the authors. For each example we indicate the activity profile in the following template: [<***issue***>, <*work environment*>, <*latitude*>] according to the explanation above.

In the following example, a requirement activity is executed in the code environment, for the purpose of defining interfaces with an external system (activity profile: **requirement**, code environment, strategically chosen activity):

> "I am reviewing the code in order to understand the system interface and the way it is supposed to serve the system requirements. This whole time is part of the requirements analysis. [Reflection document—John, May 15]".

In the next example, a strategic decision to design a specific element during Implementation is made, when more details are available (Activity profile: **design**, code environment, strategically chosen activity):

> "I decided to design this [the class ConfigurationManager] while dealing with the code, because **this way I can think of more possibilities**. For example, if I would design this using Paradigm [UML CASE tool], I would probably only think of the problems that may occur in reading the file (if it's damaged or empty) or problems in finding the file. Working with Eclipse I understand I need a class to hold parameters. There could also be a problem if one of the parameters is wrong. I didn't postpone it [the design] because of laziness; it's the dealing with the code that enables me to conclude the design required here. [Interview—Mary, August 30, regarding her May 29 Reflection]".

What follows is a different example for strategic work on Design (Activity profile: **design**, UML, strategically chosen activity):

> "I am working on a Sequence Diagram for the process of reading data from file. The design documents are essential for understanding the code later. The code is quite complicated; it will be much easier to understand it using UML. I stopped coding in order to organize what already exists and document it in the code and the design documents. [Reflection document—Ann, July 2]".

The following example shows a requirements activity that is forcibly executed during Implementation, but not in the code environment (Activity profile: **requirements**, Software Requirement Specification document, forced):

> "The coding phase of the requirement for order message processing and canceling, and their legality check according to certain parameters, obligates me to go back to the requirements document and understand these requirements better than I did in the initial stage of the project. [Reflection document—John, June 29]".

The importance of the solution presented in this section lies in reducing the fuzziness of the development phase definitions as they appear in the literature. We illustrated how each activity in the development process can be classified into one of the four phases according to the suggested definitions, employing a multidecomposition level structure. For a given decomposition level, activities can be allocated definitively to a development phase. Accordingly, it is now possible to produce a clear set of instructions for measuring time allocated to the different phases. Note that the guidelines presented in this section are not the only solution possible; their importance lies in their consistency. Whatever boundaries are defined to separate the phases, they should be consistent.

In addition, we suggest that the contribution of each time unit invested in a certain development phase is determined not merely by the issue dealt with by the developer but also by the specific profile of related attributes: environment and latitude. The possible combinations of these attributes may bring to numerous decision variables. Therefore, identifying which of these attributes substantially influences the output is helpful in quantifying the development effort.

5. Characterization of the Software Development Output

This section proposes an approach towards a measurable definition of the software development output in a way that reflects what is produced and what will ultimately be used by the customer. This approach can be seen in contrast to, for

example, an output measure based on the number of lines of code—which is of no interest to the customer.

Following the VBSE approach, note the distinction in our economic model between the definitions of *software development output* and the *value* derived from that output. The former is defined in terms of the product's properties, whereas the latter is based on how the output is perceived by the relevant stakeholder. This section deals only with defining the *software development output*. The output is a fundamental concept, whose definition must underlie any formulation and estimation of production, value, and cost functions.

Finding an appropriate output definition may also contribute to other areas of software engineering management [36]. Finding a way to measure **how much** software is produced will allow analyzing and comparing the productivity of different manufacturers, projects, development strategies, and invested resources. It will facilitate project planning, cost estimations, and so forth.

At the root of the difficulty in defining and measuring software output is the fact that software engineering deals with the *development* of a single prototype rather than mass *production* of identical goods. A basic and common approach for measuring output is the notorious lines of code (LOC) measure, which does not reflect the essence of the product, as noted previously. A more advanced approach is the function points (FPs) measure [2], reflecting the volume of the software functionality. The problem with this and similar approaches is that they do not take into account the *quality* of the product, as noted by Sommerville [31, Chapter 26]. Recent approaches that do take quality into economic consideration (e.g., [8, 10, 11, 30]) deal only with specific aspects of quality, such as bugs per volume unit (as measured in LOC or FP terms), and do not deal with the problem of resource allocation among SDPs to improve this measure.

The output of the software manufacturer is defined here as a combination of software features and their qualities. We use the term *feature* to denote the software volume unit(s) that produces a particular functionality (note that this differs from the IEEE definition). Accordingly, *a feature* is an application capability that can be measured and weighted according to its importance for a representative user. Naturally, a feature may be decomposed into subfea-tures. Although there are several ways to characterize a feature (volume unit), for example, the function points methodology [2], the object points methodology [2], or end-user features specified informally as "stories," [6], the characterization of the feature does not affect our approach to the resource allocation problem. For the sake of simplicity, we illustrate our approach for a single feature.

For quality modeling we use the notation of ISO 9126 [25, 37–39], which lists six main quality characteristics: Functionality, Reliability, Usability, Efficiency, Maintainability, and Portability. The quality is first examined from the customer's perspective, denoted as *external quality*. We assume that the customer combines the software features and the above six quality characteristics in what we call the *software product matrix (SPM)* presented in Figure 3.

Columns in the SPM represent different functional features of the software product. Rows correspond to the external qualities of the above six characteristics.

Each numeric entry in the SPM represents the measure of the corresponding row's quality characteristic for a given column's functional feature. Each subfeature and each quality subcharacteristic can be further decomposed, with appropriate weights [37, 38]. Some quality attributes, for example, Portability, may not be decomposable for all software features. Some of the entries may be *nonapplicable* when a certain quality factor is irrelevant for a particular feature.

The output of each of the four main development phases contributes in a different way to the different factors of the external quality. We formalize these contributions in terms of phase artifact qualities, which can be presented along a third dimension of the SPM matrix. This quality definition fits the ISO 9126 notation of internal quality, which refers to static properties of software that are considered independently of its execution (see [25] for more details).

Figure 4 presents a schematic view of the interrelationships between the 2-dimensional SPM and the development work along the different phases. This relationship is presented as a 3-dimensional matrix, called the *phase contribution box*.

5.1. External Quality Characterization. An external quality of a given software feature is defined by quality characteristics, which are used to evaluate the software behavior during execution from the user's point of view, with regard to this feature. The key to correct assessment of the quality factors is a comprehensive and properly weighted list of scenarios. This list should represent typical loads or stress tests under which the product should perform. The tests include running the software to obtain performance scores as well as experimenting with, for example, product modifications or portability to different operating systems.

A quality factor in general is a decomposable entity. It is composed of subfactors, each having a relative importance (weight). These subfactors, in turn, can be further decomposed. Thus, when discussing quality we refer to two elements: quality factors and their weighting [37, 38]. Weighting is largely affected by system usage, as illustrated in usage-based statistical testing techniques such as the Cleanroom approach [40].

A software developer is usually not explicitly aware of all the ISO quality characteristics or of their relative importance in the particular product being developed. However, given the accepted assumptions of the *economics of uncertainty* models, we assume that the developer behaves as if s/he does know them.

In our empirical study we created an artificial setting in which a taxi ordering server (TOS) application's quality factors and their weightings were explicitly defined to the students developing it (see Appendix A). The factors and their weightings were handed to the developers only down to a certain level, below which they had to "guess" the relative weightings. Figure 5 illustrates this notion by means of a directed tree representing the artificial setting of the

Features / Quality factors	Feature 1		Feature 2				Feature 3	Feature 4			⋯
	Feature 1.1	Feature 1.2	Feature 2.1	Feature 2.2	Feature 2.3	Feature 2.4		Feature 4.1	Feature 4.2	Feature 4.3	
Functionality											
Reliability											
Usability											
Portability											
Efficiency											
Maintainability											

Figure 3: An example of the Software Product Matrix (SPM).

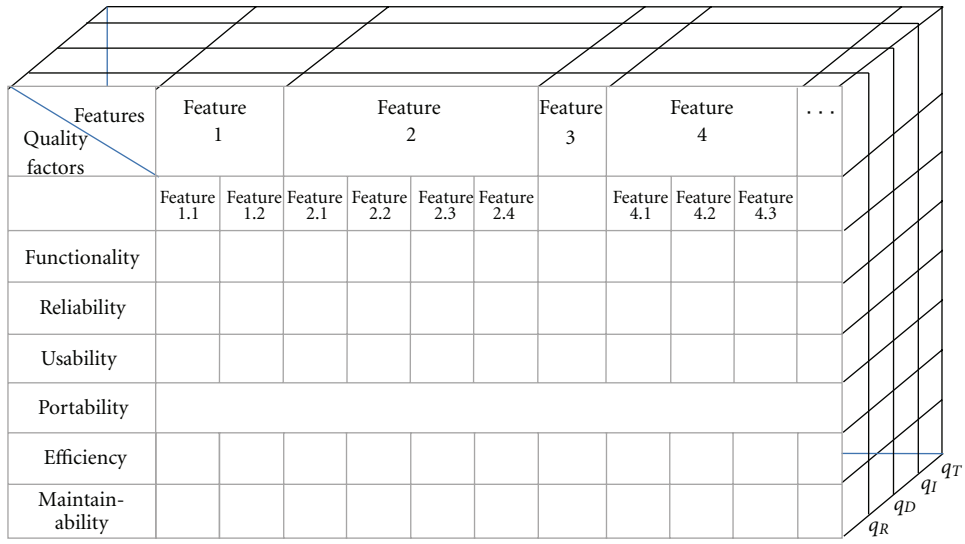

Figure 4: Example of the phase contribution box.

empirical study in terms of quality definitions. The nodes of the tree represent the quality factors. Descendants of a node in the tree represent its subfactors. The weightings on the arcs represent the relative importance of each quality factor.

According to ISO 9126, the quality factor *Functionality* consists of the subfactors' Suitability, Accurateness, Interoperability, Compliance, and Security. In our study, only Compliance and Suitability have positive weights (0.96 and 0.04, resp.). *Suitability* is defined by ISO 9126 as "Attributes of software that bear on the presence and appropriateness of a set of functions for specified tasks" and *Compliance* is defined as "attributes of software that make the software adhere to application-related standards or conventions or regulations in laws and similar prescriptions" [25].

Using this general definition of *Compliance*, we divided the *Compliance* quality subfactors into two subfactors: C_1, the degree to which the application adheres to related standards in the case of *correct* inputs, and C_2, a similar measure of the application's response to *incorrect* inputs. We assigned equal weights to each of these subfactors.

Thus far, we have seen the upper levels of external quality, explicitly defined for the students. Lower levels are assumed to have been defined by the students individually within the general definitions given for the upper levels. That is, in accordance with economic theory, we assume that the students make an educated guess regarding these subfactors and their weights or understand them without necessarily being aware of it.

Down the tree one can find quality subfactors relating to a more specific set of requirements. For instance, down the subtree spread under the subfactor C_2, one can find, for example, a quality factor at the (almost) lowest level, defined as the ability to adhere to the application-related standards in case of incorrect syntactic inputs originating from more than

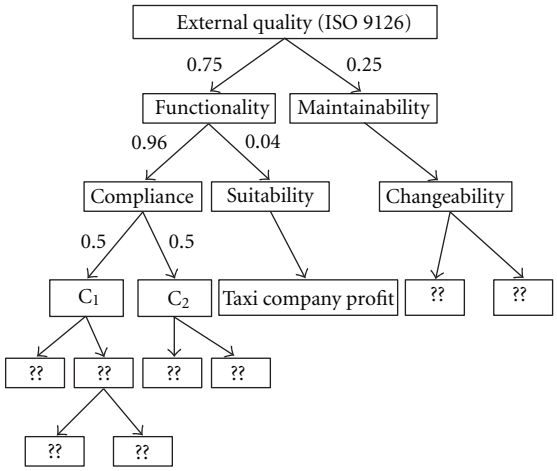

Figure 5: A directed tree representing the TOS external quality.

one space left between the words (tokens) in the passenger's message.

We assume that the relative importance of each of the external quality factors stems from the customer utility function and that they all "diffuse" into the developer's mind via the product demand function. We also assume that a rational software developer would try to make sure that his/her weighting evaluation is as accurate as possible. At the firm level, statistical testing techniques, such as in the Cleanroom approach [40], might be available. However, the individual developer tries to understand the importance of each quality subfactor using available means. This notion is well illustrated in an e-mail sent by one of the students, John, when trying to gather information from the "customer":

> "[· · ·] My question: is it critical to check that there is only a single space character between 2 words or is it less critical and therefore can be ignored, so I can move to implement other more critical functionalities? [Email—John, August 1st]".

In the above example, we see that John identified an implicit quality subfactor and tried to obtain information about its importance. That is, he understood that having more than one space between two tokens in the passenger's message is a syntax error; however, as he was very busy, he justly considered whether he should invest his time handling this error or continue to work on more important (i.e., more valuable) tasks. This example is consistent with our assumption that the software development process is a rational process of allocating resources for producing maximum quality units.

Generally, student behavior was consistent with the basic assumption that development resources are allocated primarily to achieve the highest possible quality measures. All students expressed time and again in their reflection documents their desire to increase (certain aspects of) quality when describing their decisions to move between development phases. This finding is consistent with [41], reporting that the external quality of software is the most important factor in the evaluation models of senior IS managers.

5.2. Internal Quality Characterization. According to the ISO 9126 standard [25], internal quality is measured using internal metrics, which apply to static properties of software that are considered independently of its execution. Our approach associates internal quality with *phase artifact quality*. Internal quality measures quantify the artifacts produced by different development phases.

Our method for evaluating phase artifact quality is to do it feature by feature. We look at any particular feature and evaluate the contribution of each of the four development phase artifacts to each of its six external quality factors. This is analogous to slicing our phase contribution box along a fixed feature and extracting a 2-dimensional matrix. We call it the *feature qualities matrix* (FQM). A schematic view of this matrix is presented in Figure 6.

In the upper four rows of the matrix, cell (i, j) contains the contribution of the artifact quality of the ith phase on the jth external quality factor. These four rows show artifact qualities of the four development phases with respect to the different quality factors. Naturally, for a given quality factor (column), the different qualities of each of the four phases are assumed to affect the external quality. The bottom row of the FQM holds the scores of the feature's external qualities.

The main challenge here is to devise sensible measures for each of the FQM internal quality cells, for measuring the contribution of each phase's artifact to the external quality. This will provide a practical means to estimate the assumed functional relation between the internal qualities and the external quality (see the definition of the $\hat{f}$ function in Section 3). These measures should be consistent with the phase definitions, so that it will be possible to estimate the functional relation between the resources allocated to a certain phase and the quality of this phase's artifact.

In what follows we illustrate how one can go about filling in the entries of the FQM. This illustration is based on the measurements taken in our empirical study [23].

In filling in the phase artifact cells for the Maintainability column, for example, one should consider how to define internal metrics for each of the four phases. The literature suggests many static evaluation methods for software Maintainability (e.g., [42–46]). Let us consider the effect of the quality of the Design phase artifact on Maintainability (the *second* cell of the Maintainability column in Figure 6). The metric to be selected should include factors such as documentation, modularity, and legibility, which will also be assigned weights.

Lindroos [42] summarizes several static metrics for the evaluation of Maintainability that seem highly connected with the external maintainability factor.[7] Among the metrics listed by Lindroos, coupling between objects (CBOs) and response for a class (RFC) seem to be good candidates to be included in the evaluation metrics of modularity (via encapsulation), which bear high significance in maintainability.

The CBO score of a class is the number of classes to which the class under consideration is coupled. A high

	Functionality	Reliability	Usability	Portability	Efficiency	Maintainability
q_R						
q_D						
q_I						
q_T						
External quality (Q)						

Figure 6: A schematic view of the FQM.

coupling measure has a detrimental effect on modularity and the ability to reuse the class in the future. The higher the coupling measure, the higher the sensitivity of the design to changes; that is, more effort is required for modification, fault removal, or environmental change (all ISO criteria for the evaluation of Changeability, which is one of the subcharacteristics of Maintainability). Therefore, the CBO measure is assumed to affect the external Maintainability score. In particular, CBO reflects the contribution of the design phase (as defined in Section 4) to Maintainability, as it refers to issues that can be represented by the class diagram semantics.[8]

Other metrics for evaluating the contribution of the Design artifact to Maintainability may include the documentation of the design, that is, how exhaustive and well-written the software design document is, to what extent the design complies with the requirements, and how well the UML diagrams (if used) reflect the actual design as it appears in the code.

The cells reflecting the contribution of the three other phases on Maintainability can be examined in the same way. For instance, metrics for code modularity and documentation will probably be among the factors used to evaluate the effect of the Implementation's artifact on Maintainability (the third cell of the Maintainability column in Figure 6). Metrics such as the popular Maintainability Index (MI) [47], or metrics included in this index, such as the Halstead Complexity measures or the McCabe Cyclomatic Complexity measure, might be appropriate here since the aspects of Implementation they deal with do influence Maintainability. For example, a high value for the McCabe Cyclomatic measure indicates high code complexity, which makes the program less maintainable.

The challenge of finding metrics consistent with the definitions of development phases is indeed difficult to realize. One problem is developing metrics that strictly differentiate between Design and Implementation. The MI, for example, seems to evaluate elements linked to both Design and Implementation. However, given our suggested distinction between these two phases—everything that can be represented in the class diagram semantics is considered as part of the Design, whereas anything more specific in that context is part of the Implementation—the CBO indeed evaluates the quality of the Design, as it measures the relations among classes. Likewise, the McCabe index indeed evaluates Implementation. It should be emphasized, however, that the CBO cannot serve as the only metric for Design modularity and should be combined with other metrics.

Thus far we have examined the effect of the qualities of the artifacts of the different phases on Maintainability. For this purpose, we have not gone into the nuances of the distinctions between the various subfactors of Maintainability, since it is reasonable to assume a high correlation between the effect of an internal index on Analyzability, for instance, and its effect on Changeability or Testability. However, this is not necessarily the case for the other five ISO 9126 external quality factors. In the case of Functionality, for example, the distinction between the different subfactors is very important. For instance, the effect of the phase artifacts' qualities on Compliance is different from their effect on Suitability or Accurateness.

Let us consider the effect of the development phase artifacts on the Compliance factor. As suggested in the previous subsection, the external quality of this factor can be measured by the appropriate response of the software to different types of input. The metric evaluating external Compliance, therefore, examines the fulfillment of the external functional requirements. This examination results in a collection of test cases. For each of these test cases, the application can be operated under a quality checker to examine whether the required functionality has indeed been fulfilled. The external Compliance score consists of the weighted percentage of the cases that passed the test.

In order to fill in the four cells representing the effect of the different phase artifacts on the external Compliance, we have to statically evaluate the contribution of each phase to the external *Compliance* score, that is, to evaluate what percentage of the tests will pass. This can be done by defining the collection of required functions of the given feature.

The effect of the Requirements artifact on *Compliance* can be measured by examining what percentage of the required functions have been specified as functional requirements and their quality in terms of consistency, completeness, and unambiguity. The effect of the Design artifact on *Compliance* can be measured in part by examining what percentage of the required functions have been given

solutions in the design documents. Similar metrics should be used for Implementation (by considering what percentage of the required functioned is actually implemented in the code) and Testing (by examining functional coverage).

The remaining cells in the FQM can be filled and applied for each feature of a given software product, gradually filling the entire phase contribution box for the output evaluation of the entire software, and thus quantifying the contribution of each phase to each of the external quality factors.

5.3. Summary. This section characterizes software development output as a bundle of features, each of them provided at a certain quality level. We explained the principle of defining quality in the scope of a single feature, based on the ISO 9126 quality model. We emphasized the distinction between phase artifact quality (internal quality) and the quality as perceived by the customer (external quality). We exemplified this principle by presenting internal and external quality metrics for a small subset of ISO 9126 quality characteristics as measured in our empirical study (see [23]). We presented a workable framework, called the *feature qualities matrix* (FQM), which organizes all quality scores of a given feature.

Each cell in the FQM matrix holds a certain quality score of the feature, based on predefined heuristic. We demonstrated how to define measurable metrics for several cells in the FQM while focusing on Functionality and Maintainability. The demonstration was based on students' projects. However, in this specific sense we see no major difference from real, industrial projects. The practice of measuring coupling between objects (CBOs) as a metric for evaluating the effect of the design quality on maintainability or the Maintainability Index (MI) as a metric for evaluating the effect of the implementation is independent of the projects' scale. Thus we find the external validity intact in this context.

The evaluation of other cells in the FQM can in practice be done similarly, based on the presented guidelines. For example, consider the quality factor of Efficiency, which was not demonstrated. According to ISO 9126, "time behavior" is one of the subcharacteristics of Efficiency. In algorithmic oriented features, the external quality of time behavior is based on processing and response times. The effect of Design on time behavior will be measured by means of the algorithm's time complexity. The effect of Implementation will be measured by static code analysis metrics. As a simple example, consider a C++ metric that counts instances of function parameters, which are too big and would be better handled via references.

Having a metrics for all the cells of the FQM is not required for its practical use. Some software development teams receive the requirements specification as a mature and processed document. Such teams can ignore the Requirements row of the matrix. Likewise, for some software projects the Portability or the Usability factors may not be applicable. Focusing on the internal and external qualities of the ISO 9126 subfactors that are most relevant for the feature can make the FQM a practical tool for storing and organizing quality data. The output evaluation presented in this section can be extended beyond the scope of a single feature by incorporating into our framework a methodology for distinguishing between features and weighting them, using, for example, the function points methodology [2]. Accordingly, we believe that software development output can indeed be defined in sensible and measurable units. Building a CASE tool for handling output evaluation will undoubtedly help reduce the complexity of this evaluation process.

6. Properties of the Software Production Function

The previous two sections dealt with the characterization of the economic model's variables. This section discusses the nature of the relationships between the dependent and independent variables. Since a comprehensive formulation of production functions should be based on a vast empirical basis which has yet to be built, this section only illustrates what can be learned about the theoretical model from empirical studies and how that model can help in interpreting the developers' observed conduct in experimental settings.

While the mathematical structures of these production functions were not fully characterized in this research, we maintain the assumption that such underlying functions do exist. The empirical study yielded some inferences about the functional forms of the relationships between work on development phases and the resulting development output. This section presents two key characteristics that emerged from this study regarding the nature of these functions: *substitutability among development phases* and *decreasing marginal product*.

6.1. Substitution among Production Factors. The economic model proposed in Section 3 relies on knowing how different combinations of inputs, in the form of working hours invested in the Requirement, Design, Implementation, and Testing, affect the resulting software. More specifically, it is assumed that the software developer has the latitude to increase one of the inputs while decreasing the other or choose alternative sequences of time allocations among the phases, without changing the output index score of the produced software.

The curve in Figure 7 is a schematic picture of all combinations of two inputs, (holding the level of other inputs fixed), that yield the same output, known in economics as *isoquant*. Each isoquant is indexed by the constant level of output it represents, $Q_1 < Q_2$, and so forth. The existence of such a curve reflects the *substitutability* of production factors. Its downward slope indicates that if you scale down the use of one of the input factors, you need to scale up the use of another factor in order to preserve the same output. This ability to use alternative combinations of production factors allows the developer to choose the least costly combination of inputs that produces the desired output.

In the empirical study, we made two observations with regard to the nature of this substitution. One was the fact that developers chose vastly different input combinations and

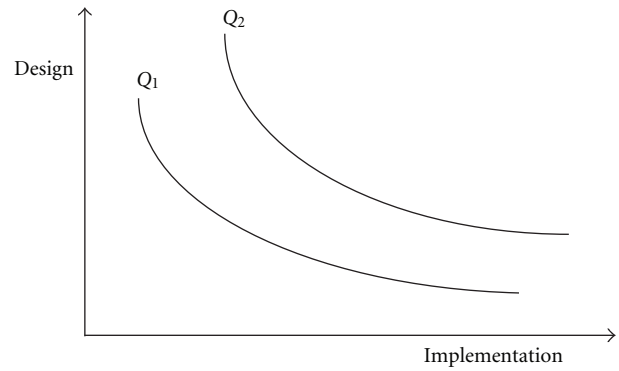

Figure 7: Schematic visualization of *substitution* in software production functions.

yet managed to produce similar outputs. This observation supports the general idea that one can produce a particular software product by different combinations of inputs. Thus, a nontrivial decision must be made here, namely, which input combination yields the highest return to the developer. The degree of substitution present among SDPs in performing a particular task depends on the aggregation level of the task. More elementary tasks allow less substitution among phases, down to, for instance, multiplying two numbers. Such a task still requires some requirement analysis, for example, whether the numbers are real, integers, or complex numbers, and some design work on alternative ways to carry out the requirement, with very limited substitution between them. On the other extreme, a large multifeatured software product admits a large degree of substitution among phases, including shifting among phases associated with different features or different quality characteristics.

Our second observation was that personal attributes of the developer can influence the degree of substitutability among the development phases.

Here is an example of how substitution is affected by Java development experience:

> "In the beginning of the project I used to perform many tests, since I had no confidence in my programming. Now that *I have more experience working with Java language*, I don't perform as many tests as I used to; I know how to conduct Unit Testing more efficiently. [Interview—Mary, August 30]".

6.2. Decreasing Marginal Product. Production processes having this property exhibit decreasing returns as one of the inputs is increased while holding other inputs constant. Such processes usually imply the desirability of an internal allocation of resources, with some positive amount of each input. As more and more resources are allocated to the same input, the improvement obtained becomes smaller, until at some point it pays off to shift resources to another input. This is known in economics as *decreasing marginal product*. In the context of a dynamic production process, this property determines *when* to shift the allocation of resources among inputs, and *how much* to allocate to each input.

In static optimization problems, the decreasing marginal product property is a necessary condition for concavity. Concave optimization problems can be solved by means of first order conditions. Moreover, their solutions are typically continuous functions of the parameters of the problem, allowing for straight forward sensitivity analysis of the solution with respect to parameters of the problem. Thus, to the extent that such functional structure is validated by empirical studies, the optimization model becomes a handy tool for extracting the optimal allocation of resources. The decreasing slope of the *isoquants* in Figure 7 reflects the decreasing returns property of each input in isolation.

In fact, availability of analytical solutions to the resource allocation problem is not essential given the advance of simulation-based models. Such models can be augmented by expanding the software production process module (e.g., [28, Section 3.3.5]), according to the software production function proposed in this paper. Thus, the impact of alternative allocations of resources on various aspects of the developed output can be traced via simulations, along with its interactions with other aspects of managing the software project.

Our observations are consistent with the *decreasing marginal product of working time allocated to each of the phases*. Throughout the development process, the students usually moved from one phase to another when they felt that the previous phase's quality is satisfactory and their resources will have a higher contribution if applied to a different phase at that point. This notion is clearly exemplified in Ann's report on her shift from Design to Implementation:

> "I moved to Implementation after achieving a **reasonable level** of the UML documents relative to the time I have left to complete the project. [Reflection document—Ann, May 26]".

In a follow-up clarification Ann was asked to explain what she meant by "reasonable level of UML documents," and this was her response:

> "Reasonable level [of UML documents means]: a description of 2 basic functions and their sequence; which class performs which function; organizing the functionality according to a logical, sense-making, order. It's enough for me to describe the basic functions at this stage in addition to the main functions and not develop on paper [that is, in the UML diagrams as opposed to the code environment] all the functions, because I don't have enough time for it. [Ann, May 26]".

Ann clearly has a firm perception of what "reasonable level of design" means. She is also well aware of the time constraint. It seems that what she had determined as reasonable level of design is the design construction up to the point where the marginal product generated by an additional time-unit of Design is lower than the marginal product of Implementation. That is, given the current state of her development, her next working hour will yield higher return

in Implementation than in Design, because she already made significant headway in Design.

This phenomenon was observed repeatedly with respect to all students in the study. Here is another example:

> "Completed the Design of methods and parameters to be implemented **to an adequate level** and moved on to Implementation of methods' code [Reflection document—John, June 5]."

Nonetheless, we also observed shifts from one phase to the next, when the developer explicitly acknowledges that the previous phase is incomplete. For example,

> "Due to time limitations I stopped at the stage of basic definition of the Requirements and did not go into its depth. I defined major functions and their respective tasks, and the data structures. If I had more time, I would have gone into more detail. [Reflection document—Ann, May 11]".

The shift from one phase to another despite recognized imperfections in the former, as opposed to completing one phase before moving to the next, is consistent with allocating development resources to activities that yield the highest *marginal* contribution.

7. Using the Model to Analyze Developer Decisions

The model presented in this paper provides a framework within which we can analyze various developer decisions made at different points in the development process, identify the reasons for these decisions, and suggest how to improve them in future projects. We exemplify these capabilities of the model by analyzing suboptimal decisions made by the students developing the software in our empirical setting.

We have identified cases where the time allocation decisions made by the students were clearly not consistent with the model and indeed led to poor results in which students received zero scores on Functionality (during interim project submissions). In order to understand this result and the developers' conduct preceding it, we investigated the sources of the mistaken allocation decisions and arrived at the following possible reasons for them.

(i) The students failed to allocate sufficient time to Requirements and Testing. For example, mistakes in the format of the response message sent by the TOS to an incoming order request, a common reason for getting a zero score by the quality checker, could have been avoided by reading the requirements more carefully. Likewise, the software developed by some of the students failed to detect incorrect inputs despite our explicit instruction to detect them and respond with appropriate error messages. A main reason for this failure was that developers neglected to include appropriate tests of incoming orders.

(ii) Given that the students had been informed that Functionality would account for 75% of their product's external quality (i.e., a major part of their course grade), they naturally tended to invest much effort in it. However, we found that some of them perceived Functionality improvement as stemming almost exclusively from time spent on Implementation, failing to appreciate the contribution of other development phases to this software quality factor. Thus, allocating time to Requirements and Design would have resulted, in their opinion, only in an increase of the Maintainability factor, which accounted for only 25% of the external quality. Moreover, we found that students invested time in infrastructural activities (e.g., writing a proper requirements document in the requirements phase, writing a UML diagram properly describing the design and other documents laying out the design rationale, and writing a test plan for the testing phase), mainly when forced to do so. For example, in an interview, Mary said "I considered writing a design document. Since you did not say it's a must, and I thought it will take me too much time, I decided not produce it after all."

(iii) The students tended to overinvest in the Suitability aspect of Functionality at the expense of Compliance, though the latter had been defined as a much more important component of Functionality. Suitability was measured by the profits generated by the TOS scheduler, whereas Compliance was measured by the percentage of appropriate responses of the software to correct and incorrect sets of inputs. Not only was the weight of Compliance 24 times higher than that of Suitability (96% versus 4% of the Functionality grade, respectively, as was explicitly explained in the project description given to the students), but also the marginal impact of profits on Suitability was decreasing, thus resulting in very low profit in return for the time invested. After the development process was complete, we examined students' motivations while attempting to identify what caused such poor results. Our examination revealed that some students simply did not understand the quality definitions, while others did, but knowingly chose to operate differently. For example, in a follow-up interview Ann said "I was aware of the weights given to different quality factors, but personally the scheduling function was the one that challenged and interested me the most." This case exemplifies a developer maximizing her own individual objective function, which deviates from the objective function given.

(iv) The students tended not to use helpful software tools with which they were not familiar. It appears that they ascribe more importance to evident time costs than to the unknown efficiency gains. Consequently, they may allocate too much time to developing new "tools" instead of using available ones, and this often distorts their decision as to how to allocate time among the development phases. For example, in order to apply syntax checking for incoming messages, some of the students chose to develop

their own parser rather than use the existing software package supplied by Java (i.e., the regex package). This was a clear error, since using regex would have been more efficient.

(v) Cognitive aspects such as attention span, concentration ability, memory capacity, and diligence seem to have significant influence on the chosen allocation of time resources. For example, in many cases students chose their next activity according to how concentrated/tired they felt at that time or due to a concern that they might forget to conduct a certain activity at a later stage.

We believe that the relevance of the value-based underpinning of this work extends beyond the scope of the field-study presented in this paper. It is our working assumption that each software developer should operate to increase the product's external quality, since this is the factor that drives the demand for the product. Accordingly, development process decisions should be taken while considering their consequences in terms of external quality. In cases where external quality properties are practically not measurable, we suggest to use internal quality instead. Future research may estimate the functional relations between internal and external qualities, to yield proxies of the project's external qualities based on simple measurements of internal qualities.

8. Conclusions and Future Work

Optimal allocation of resources among software development phases is a complicated problem that has not been addressed systematically, despite its possible ramifications for software project quality and cost. The research framework presents two goals for addressing this problem. The first is to enhance our understanding of resource allocation decisions, their complex nature, and their influence on the output. The second goal, which will be handled in the long-term on the basis of this and future research, is prescriptive: proposing how to modify existing resource allocation approaches or develop new ones, in order to improve the productivity of the software development processes.

This paper presents an economics-based approach for studying this problem. The underlying assumptions of our work have been that it is feasible and useful to treat resource allocation among SDPs as rational decisions whose aim is to maximize a well-defined objective, and that these allocation decisions should be based upon measurable notions of inputs, outputs, costs, and resulting quality.

Our approach is structured along two axes: theoretical and empirical, building and refining a theoretical model based on empirical data. The paper presents the first steps conducted on these two axes. We have developed an economic model for analyzing the resource allocation problem as a constrained maximization problem and characterized its main building blocks: input in the form of work performed in different SDPs, software development output, and the *software production function*, which maps inputs into outputs. We have constructed an empirical environment for evaluating and refining the model, and conducted the first empirical study for examining software developers' time allocation decisions and their resulting output.

The empirical study was quite limited in its setting compared to typical projects in the software industry. First, our subjects were students rather than professional software developers, which was a limitation dictated by the need to let them all develop the same product (according to an identical set of customer requirements). However, Kitchenham et al. [19] argue that using students as subjects instead of software engineers is not a major problem (as long as the researchers are not specifically interested in experts). Second, the students worked individually, as 1-person teams, which is a limited simulation of real development teams in industry. However, our focus here is on variables and tradeoffs in decision-making processes at the individual level. Regardless of these limitations and the simplicity of the empirical setting, we have learned much about the nature of the variables and functions of the model as well as its practical application that could not have been obtained without the empirical evidence generated by this study.

To validate our approach and the proposed model, the following questions must be answered.

(1) Can each activity during the development process be appropriately classified into one of the development phases?

(2) Can the software development output be defined in measurable units which capture what the producer develops and what is consumed by the consumer?

(3) Is there a systematic relationship among the resources allocated to different SDPs and the resulting output?

We found positive answers to each of these questions, although more work is needed to refine the specifications of the model's components. These answers were spelled out in Sections 4, 5, and 6, where we characterized the model components and described the empirical study using the first version of the measurement tools we have developed.

We regard the work presented here as an illustrative first step. Its novelty lies in casting the resource allocation among SDPs as a familiar economic optimization problem, utilizing standard economic analytical tools, while developing the underlying concept of the *software production function* along with empirical methods for its evaluation. The usefulness of our approach will increase as we improve measurements of inputs and outputs and accumulate more reliable information about, and estimations of, the mathematical relationships among variables embedded in the software production function.

Even at this illustrative stage, our work demonstrates the usefulness of the model as an *evaluation* tool. For example, the *feature qualities matrix* (FQM), or even a subset of it, can be used as a practical tool for storing and organizing data regarding comprehensive aspects of the developed software quality.

The reasoning of the *software production function* can also be helpful in practice. Development organizations that will keep data of inputs and outputs across different versions of the same product, or across different features from the

same family, may find systematic relations between them and improve their analytic capabilities.

We found in the developers' behavior within our study support for the underlying assumption of rational process for maximizing a quality-based goal function (see Sections 5 and 6). On the other hand, we found significant and systematic deviations between actual allocations of resources among SDPs and those prescribed by the model and were able to use the model to identify some of the sources of these deviations.

In addition to indications supporting our model, we find that the model should be refined in several ways. For instance, more emphasis should be put on the allocation of resources to different *features* in the required application and on the order in which these features should be developed. We have observed that decisions regarding time allocation among features are made together with those regarding resource allocation among development phases. We can view each feature as having its own production function, and extend the model to include combinations of features and phases. Namely, each decision will refer to both: to which feature and to which phase to allocate the next time unit. Future work may use Function Points Analysis for deeper characterization of the term *feature*. Recent work such as the one by Batista et al. [48] can be helpful for this regard.

Our approach can also indirectly contribute to related fields in software engineering. For instance, phase-sensitive versions of the cost estimation model [3] will yield sharper and more accurate project effort and cost estimates, by adopting the phase characterizations presented in Section 4. A straightforward extension of our economic models can be used to model and analyze the optimal output level for releasing the software, thus contributing to the discipline of Release Engineering (e.g., [15, Chapter 9], and [49]). Future work should strive to further sharpen the characterization of variables in the model and estimate the relationships embedded in it, paying special attention to the incentive mechanisms under which software developers operate.

Additional work is needed to further formalize the characterization of the development phases and combine it with industrial development methods such as the RUP model (see [2, 13]). The characterization of the output should be concretized further by taking into account the different features embedded in a software product, using Function Point counting [2], and by developing a method for measuring and weighing subfeatures and their qualities. This will facilitate the comparison between different philosophies underlying alternative resource allocation strategies, even across different software projects.

Our proposed definition of internal qualities as phase artifact qualities should be augmented by metrics that capture the contribution of each development phase to each of the external quality characteristics. This can be accomplished, as illustrated in this work, by filling in more entries in the FQM, which organizes the quality scores in the context of a single feature.

We believe that a large part of software development consists of systematic work along well-defined routes, with measurable inputs and outputs. Our work demonstrates that that portion, at least, can be analyzed and improved upon as done for other production processes.

Appendices

A. The TOS Application Task

The toy-application developed by the students was a Taxi Ordering Server (TOS). The purpose of the system was to supply simple taxi ordering services to potential passengers, within a given country and under the constraint of a given number of taxi cabs. The TOS should supply the passengers the following services.

(1) *TaxiOrder*. The passenger notes his or her location, destination, required pickup time, and his/her name. In case of an incorrect request message, the TOS will notify the passenger. Otherwise, if the TOS succeeds in scheduling a taxi for this task, it will respond by giving the passenger a unique Ride-ID number. If it does not succeed, it will send a rejection message.

(2) *OrderCancellation*. The passenger cancels an order by specifying its Ride-ID to the TOS. The TOS has to respond after checking whether the message is correct.

Given the expected revenue and the costs of the ride orders, the TOS has to optimize order scheduling, in order to maximize the taxi company's profit.

The four development phases created a simulated software lifecycle: the students specified the detailed engineering requirements, then designed, implemented, and tested the software.

The relative importance (weights) of the various ISO 9126 [25] quality factors of the application to be developed was given to the students. According to our economic approach, such weights are needed so that the developer can decide, given the underlying software production function, how to allocate time in the most profitable way. Naturally, the weights were given to a specific collection of ISO quality factors [37], whereas the developers still had to use their own judgment in order to decide how to weigh subfactors for which no specific weights were given.

The quality measure of the TOS application was defined as an aggregate measure of two key quality factors: Functionality and Maintainability (henceforth F and M, resp.), whereas other factors included in the ISO definition received zero weight. (We use the word *weight* loosely and mean the elasticity of the score with respect to each quality factor.) The resulting formula of the TOS quality was $Q = F^{0.75} \cdot M^{0.25}$, representing the economic tradeoff between these quality factors.

The Functionality quality factor was divided into two subfactors: Suitability and Compliance (henceforth S and C, resp.), while the other ISO subfactors received zero weight each. The Compliance subfactor was assigned a weight of 96% and the Suitability subfactor was assigned a weight of 4%. The resulting formula for the Functionality factor was $F = C^{0.96} \cdot S^{0.04}$. Note that it was not our intention here to represent a typical development scenario, but rather to include in our study environment opportunities to study

developers' behavior; hence, suitability was assigned an extremely low weight.

The company's profit is calculated on the basis of the payment received from passengers and the operating costs per unit of distance driven. In light of the general ISO definition of Suitability, we define this quality subfactor of the TOS application as $S = e^{-1000/\text{profit}}$. This measure exhibits decreasing marginal contribution of profits to Suitability.

Based on the ISO general definition of Compliance, we divided the Compliance quality subfactor in the TOS application into two subfactors: C_1, the degree to which the application adheres to related standards in the case of *correct* inputs, and a similar measure of the application's response to *incorrect* inputs, C_2. We assigned equal weights to each of these components, so that the resulting formula was $C = C_1^{0.5} \cdot C_2^{0.5}$. Accordingly, the Functionality quality factor is

$$F = C^{0.96} \cdot S^{0.04} = \left(C_1^{0.48} \cdot C_2^{0.48}\right) \cdot \left(e^{-1000/\text{profit}}\right)^{0.04}. \quad \text{(A.1)}$$

The Maintainability factor was defined in terms of the subfactor of Changeability only. In the context of the TOS application, Changeability can be evaluated in terms of how long it takes a reasonably proficient software engineer to make a particular change in the original application. The overall quality measure of the application given to students was

$$\begin{aligned} Q &= F^{0.75} \cdot M^{0.25} \\ &= \left(C^{0.96} \cdot S^{0.04}\right)^{0.75} \cdot M^{0.25} \\ &= \left(C_1^{0.36} \cdot C_2^{0.36} \cdot \left(e^{-1000/\text{profit}}\right)^{0.03}\right) \cdot M^{0.25}. \end{aligned} \quad \text{(A.2)}$$

The external quality score of Functionality was evaluated by our QC application.[9] The QC calculated the score of each of the Functionality subfactors and produced a scalar score according to the formula presented previously.

The C_1 and C_2 compliance subfactors were calculated by the QC on the basis of exhaustive and representative test suites, for correct and incorrect inputs, respectively, where the application either passed or failed each test. The Suitability score was derived from the C_1 test suite (the correct inputs) according to the resulting profits of the taxi company. Profits were calculated by adding all payments received for rides, minus the costs of these rides, which depend on their length.

The QC yielded a detailed report that included a list of all test cases run on the TOS, a "Pass" or "Fail" message for each test case, an indication for the source of the TOS failure, and finally a summary report for the values of Compliance, Suitability, and Functionality. The quality of the product determined the students' project grades (thus their incentives). The grade was calculated as follows: 2/3 according to the QC and the other 1/3 according to the quality of the reflection document.

B. A Dynamic Model of Resource Allocation among Software Development Phases

In Subsection 3.2 we explained the principles of our dynamic model for resource allocation among SDPs. This appendix presents a complete formal version of this model. Suppose that there are N units of time, or money, which can be used to develop a software product. These resource units can be sequentially allocated, one unit after the other, to any of the *four* development phases. This dynamic model allows us to allocate successive units of the resource among the development phases, taking into account the progress that has been made so far, and the remaining available resources.

Let Q_n be the software development output index after n of the N units of the resource have been allocated, $n = 0, 1, 2, \ldots, N$, and let $V(Q_n)$ be the value of a software product with quality index Q_n to the software developer. We are considering a specific product, with a given set of features, so that only the quality of artifacts can be affected by different allocations of resources. The (monetary) cost of assigning any resource unit to a development phase is given by w_i, $i = 1, 2, 3, 4$. The goal is to find the sequence of assignments of N available units of development resources to the four development phases in order to achieve the highest possible value to the developer. The optimizing sequence should also allow development to be stopped at any time, thus saving the unused development resources.

Let $\varphi_n(Q)$ be the optimal value that can be obtained from an in-process software project with a current development output vector Q, after using n out of N units of the resource. At the heart of this formulation is the incremental improvement function, $g(Q, a)$, describing the change in each component of the quality state vector Q when the next unit of resource is allocated to development phase: a, $a \in \{1, 2, 3, 4\}$. With this function, the optimal value functions $\varphi_n(Q)$ are defined recursively, for $n = N, N-1, N-2, \ldots, 0$, as follows.

For $n = N$,

$$\varphi_N(Q) = V(Q). \quad \text{(B.1)}$$

For $n = N-1, N-2, \ldots, 0$,

$$\varphi_n(Q) = \text{Max}\left\{V(Q),\ \underset{a_{n+1} \in \{1,2,3,4\}}{\text{Max}} \left\{w_{a_{n+1}} + \varphi_{n+1}(Q + g(Q, a_{n+1}))\right\}\right\} \quad \text{(B.2)}$$

The term $V(Q)$ on the RHS represents the option to stop development, release the project in its current state Q, and receive the value associated with Q. Alternatively, if further development is to be undertaken, then the assignment of the next unit of the resource a_{n+1} must be determined, taking into account its cost $w_{a_{n+1}}$, and the fact that subsequent development decisions will also be made in this optimal fashion.

This recursive system of equations is solved backwards. When all N available units of the resource have been used up, there is no option but to stop development, so that $\varphi_N(Q) = V(Q)$. When only the last unit of the resource remains to be assigned, the optimal value function corresponding to that

case and the optimal phase assignment of the last unit of the resource are given by

$$\varphi_{N-1}(Q) = \mathrm{M}ax\left\{ V(Q)\ ,\ \underset{a_N\in\{1,2,3,4\}}{\mathrm{M}ax}\{w_{a_N} + V\left(Q + g(Q, a_N)\right)\}\right\}. \tag{B.3}$$

Here we use the fact that the next optimal value function $\varphi_N(Q)$ is $V(Q)$.

Once the function $\varphi_{N-1}(\cdot)$ is found for every possible Q, the optimal value function when two units of the resource remain to be assigned, $\varphi_{N-2}(\cdot)$, is found in a similar manner, and so on. Starting with an initial development output Q_0, (e.g., $Q_0 = 0$), one finds the value of optimally allocating the N available resource units among the phases as $\varphi_0(Q_0)$. Note that this optimal value takes into account the option to stop development before all resources have been used up.

Note that while the model in this appendix uses external quality Q as the state variable, an equivalent modeling in terms of phase artifact (internal) qualities $q = (q_1, q_2, q_3, q_4,)$ can be obtained, by using the relationship $Q = \hat{f}(q)$ from Section 3, and an appropriately defined incremental improvement function $\hat{g}(q, a)$.

The dynamic programming formulation solves for the value-maximizing (net of costs) sequence of resource allocation, taking into account both the cumulative and complex influence of that sequence on resulting quality, and the cost of assigning different resource units to different development phases. This approach can be easily extended to reflect random disturbances in the development process, provided that one can specify a probability structure governing this stochastic development process. Alternatively, the implications of different sequences of resource allocations can be explored using simulation-based models, as in [28].

Acknowledgments

The authors gratefully acknowledge the financial support and encouragement of the Caesarea Rothschild Foundation Institute for Interdisciplinary Applications of Computer Science (C.R.I.) at the University of Haifa. The authors wish to thank Daniel Berry, Yair Wand, and Regev Porat for their contributions to this research.

Endnotes

1. Throughout the paper we use the generic term "developer" for all hierarchical levels in the software organization: from the entire software house to the individual engineer.
2. Yiftachel and Hadar [36] explain the link between the problems of defining and measuring software development output, which is a critical component of the resource allocation problem, and what Brooks [17] defines as an essential problem.
3. From here on the 4 phases will be noted with capital initials to indicate that the phase is discussed. For example, when we note "Design", we mean "the design phase".
4. Reference [28] proposes various System Dynamics models to effectively manage such processes and demonstrates their capabilities. However, it does not deal specifically with allocation of resources across software development phases, except to illustrate the concept of product peer inspection in Section 5.3.2.
5. The literature suggests additional phases omitted from this paper. Specifically, planning and management are not considered at all in this model, being orthogonal to the development phases we refer to. Integration and maintenance are considered within the existing phases (e.g., each maintenance process can be viewed as comprised of several activities, each associated with one of the four phases).
6. Unit testing is viewed by many as part of the implementation (e.g. [31]), although it is a *checking* operation.
7. We refer in our examples to measures related to the object-oriented paradigms. Of course, when using other development paradigms, the choice of measures needs to be done accordingly.
8. CBO is a simple metric for evaluating the Design artifact quality in the sense of modularity. The literature suggests more complicated metrics; for the sake of simplicity we illustrate our approach using CBO.
9. In this study we did not measure the *external* maintainability factor. However, we evaluated the effects of design (q_2) and implementation (q_3) on maintainability. For details see [23].

References

[1] S. R. Schach, *Object-Oriented and Classical Software Engineering*, McGraw-Hill, New York, NY, USA, 5th edition, 2002.
[2] B. W. Boehm, C. Abts, A. W. Brown et al., *Software Cost Estimation with COCOMO II*, Prentice-Hall, Englewood Cliffs, NJ, USA, 2000.
[3] B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1981.
[4] G. Smith and L. Wildman, "Model checking Z specifications using SAL," in *the International Conference of Z and B Users (ZB '05)*, H. Treharne, S. King, M. Henson, and S. Schneider, Eds., pp. 87–105, Springer, 2005.
[5] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Boston, Mass, USA, 2000.
[6] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 226–237, 2005.
[7] G. Tassey, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, National Institute of Standards and Technology, 2002.
[8] P. Jalote and B. Vishal, "Optimal resource allocation for the quality control process," in *Proceedings of the 14th International Symposium on Software Reliability Engineering*, Denver, Colo, USA, November 2003.

[9] S. R. Schach, *Introduction to Object-Related Analysis and Design*, McGraw-Hill, New York, NY, USA, 2004.

[10] B. Steece, S. Chulani, and B. Boehm, "Determining software quality using COQUALMO," in *Case Studies in Reliability and Maintenance*, W Blischke and D Murthy, Eds., Wiley, Sidney, Australia, 2002.

[11] E. K. Emam, *The ROI from Software Quality*, Auerbach Publications, Boston, Mass, USA, 2005.

[12] Y. Yang, M. He, M. Li, Q. Wang, and B. Boehm, "Phase distribution of software development effort," in *Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 61–69, Kaiserslautern, Germany, 2008.

[13] W. Heijstek and M. R. V. Chaudron, "Effort distribution in model-based development," in *the 2nd Workshop on Model Size Metrics, and the 10th International Conference on Model Driven Engineering Languages and Systems*, Nashville, Tenn, USA, 2007.

[14] AJG Babu and N. Suresh, "Modelling and optimizing software quality," *International Journal of Quality and Reliability Management*, vol. 13, no. 3, pp. 95–103, 1996.

[15] lS Biffl, A Aurum, BW Boehm, H Erdogmus, and P. Grünbacher, *Value-Based Software Engineering*, Springer, New York, NY, USA, 2005.

[16] L. Huang and B. Boehm, "Determining how much software assurance is enough: a value-based approach," in *the 7th Workshop on Economics-Driven Software Research*, St. Louis, Mo, USA, 2005.

[17] F. P. Brooks, "No silver bullet refired," in *The Mythical Man-Month*, pp. 207–226, Addison Wesley/Longman, Boston, Mass, USA, 1995.

[18] D. M. Berry, "The inevitable pain of software development: why there is no silver bullet," in *Proceedings of 9th International Workshop on Radical Innovations of Software and Systems Engineering in the Future*, October 2002.

[19] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard et al., "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721–734, 2002.

[20] M. Bassey, "Methods of enquiry and the conduct of case study research," in *Case Study Research in Educational Settings*, chapter 7, pp. 65–91, Open University Press, Buckingham, UK, 1999.

[21] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[22] N. K. Denzin and Y. S. Lincoln, Eds., *Handbook of Qualitative Research*, Sage, Thousand Oaks, Calif, USA, 2000.

[23] P. Yiftachel, *Resource Allocation among Software Development Phases*, M.S. thesis, Computer Science Department, University of Haifa, 2006.

[24] S. T. Hackman, *Production Economics*, chapter 2, Springer, Berlin, Germany, 2008.

[25] ISO/IEC TR 9126, "Software engineering-product quality," 19-12-2000.

[26] Q. Hu, "Evaluating alternative software production functions," *IEEE Transactions on Software Engineering*, vol. 23, no. 6, pp. 379–387, 1997.

[27] P. C. Pendharkar, J. A. Rodger, and G. H. Subramanian, "An empirical study of the Cobb-Douglas production function properties of software development effort," *Information and Software Technology*, vol. 50, no. 12, pp. 1181–1188, 2008.

[28] R. J. Madachy, *Software Process Dynamics*, Prentice-Hall/IEEE Press, Englewood Cliffs, NJ, USA, 2008.

[29] P. Yiftachel, D. Peled, I. Hadar, and D. Goldwasser, "Resource allocation among development phases: an economic approach," in *Proceedings of Economic Driven Software Engineering Research Workshop, the 28th International Conference on Software Engineering*, pp. 43–48, Shanghai, China, 2006.

[30] W. Heijstek and M. R. V. Chaudron, "On early investments in software development: a relation between effort distribution and defects in RUP projects," Tech. Rep., Leiden University, Leiden Institute of Advanced Computer Science, 2008.

[31] I. Sommerville, *Software Engineering*, Pearson Education Limited, London, UK, 7th edition, 2005.

[32] IEEE 610.12-1990, "A Glossary of Software Engineering Terminology," Institute of Electrical and Electrical and Electronic Engineers, Inc, 1990.

[33] D. M. Berry, "What, Not How? When Is 'How' Really 'What'? and Some Thoughts on Quality Requirements," Tech. Rep., Computer Science Department, University of Waterloo, 2001.

[34] P. Ralph and Y. Wand, "A proposal for a formal definition of the design concept," in *Design Requirements Engineering: A Ten-Year Perspective*, K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and B. Robinson, Eds., pp. 103–136, Springer, Berlin, Germany, 2008.

[35] D. Paulson and Y. Wand, "An automated approach to information systems decomposition," *IEEE Transactions on Software Engineering*, vol. 18, no. 3, pp. 174–189, 1992.

[36] P. Yiftachel and I. Hadar, "Defining and measuring software development output: a light at the end of the tunnel for an essential problem. Presented at the workshop, No Silver Bullet: A Retrospective on the Essence and Accidents of Software Engineering," in *the International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '07)*, Montreal, Canada, October 2007.

[37] M. King, "Living up to standards," in *the 10th Conference of the European Chapter of the Association for Computational Linguistics (EACL '03)*, pp. 65–72, Budapest, Hungary, 2003.

[38] V. Poladian, S. Butler, M. Shaw, and D. Garlan, "Time is not money: the case for multi-dimensional accounting in value-based software engineering," in *the 5th Workshop on Economics-Driven Software Research*, pp. 19–24, Portland, Ore, USA, 2003.

[39] M. A. Côté, W. Suryn, C. Y. Laporte, and R. A. Martin, "The evolution path for industrial software quality evaluation methods applying ISO/IEC 9126:2001 quality model: example of MITRE's SQAE method," *Software Quality Journal*, vol. 13, no. 1, pp. 17–30, 2005.

[40] D. P. Kelly and R. S. Oshana, "Improving software quality using statistical testing techniques," *Information and Software Technology*, vol. 42, no. 12, pp. 801–807, 2000.

[41] B. Anderson, A. Bajaj, and W. Gorr, "An estimation of the decision models of senior IS managers when evaluating the external quality of organizational software," *Journal of Systems and Software*, vol. 61, no. 1, pp. 59–75, 2002.

[42] J. Lindroos, "Code and design metrics for object-oriented systems," in *the Seminar for Quality Models for Software Engineering*, Helsinki, Finland, 2004.

[43] S. R. Chidamber and C. F. Kemerer, "Metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[44] Y. Wand and R. Weber, "An ontological evaluation of systems analysis and design methods," in *Information System Concepts: An In-Depth Analysis*, E. D. Falkenberg and P. Lindgreen, Eds., pp. 79–107, North-Holland, Amsterdam, The Netherlands, 1989.

[45] S. C. Misra, "Modeling design/Coding factors that drive maintainability of software systems," *Software Quality Journal*, vol. 13, no. 3, pp. 297–320, 2005.

[46] Y. Cai and K. J. Sullivan, "A value-oriented theory of modularity in design," in *the 7th International Economics-Driven Software Engineering Research Workshop*, St. Louis, Mo, USA, 2005.

[47] K. D. Welker and P. W. Oman, "Software maintainability metrics models in practice," *Crosstalk Journal of Defense Software Engineering*, vol. 8, no. 11, pp. 19–23, 1995.

[48] V. A. Batista, D. C. C. Peixoto, E. P. Borges, W. Pádua, R. F. Resende, and C. I. P. S. Pádua, "ReMoFP: a tool for counting function points from UML requirement models," *Advances in Software Engineering*, vol. 2011, Article ID 495232, 7 pages, 2011.

[49] J. M. Akker, S. van den Brinkkemper, G. Diepen, and J. Versendaal, "Determination of the next release of a software product: an approach using integer linear programming," in *Proceedings of the CAiSE '05 FORUM*, pp. 119–124, 2005.