~ ~ ~ Working Draft of October 24, 2005~ ~ ~

~ ~ ~ This will be obsolete on November 30, 2005. Do not circulate after that date ~ ~ ~
~ ~ ~ After December 1, 2005 refer instead to http://shaw-weil.com/marian/DisplayPaper.asp?paper_id=84 **~ ~ ~**

In Search of a Unified Theory for Early Predictive Design Evaluation for Software

Mary Shaw, Ashish Arora, Shawn Butler, Vahe Poladian, Chris Scaffidi

Revised October 2005
CMU-CS-05-139
CMU-ISRI-05-114

This report supersedes CMU-CS-04-133/CMU-ISRI-04-118 of June 2004

School of Computer Science and Sloan Software Industry Center
Carnegie Mellon University
Pittsburgh, PA 15213-3890
{mary.shaw, shawn.butler, vahe.poladian, chris scaffidi}@cs.cmu.edu, ashish@andrew.cmu.edu

Mary Shaw is a member of the Institute for Software Research International and the Computer Science Department in the School of Computer Science and of the Sloan Software Industry Center in the H John Heinz III School of Public Policy and Management
Ashish Arora is a member of the Sloan Software Industry Center in the H John Heinz III School of Public Policy and Management
Shawn Butler and Chris Scaffidi are members of the Institute for Software Research International in the School of Computer Science
Vahe Poladian is a member of the Computer Science Department in the School of Computer Science

Keywords: Engineering design, design evaluation, design selection, unified design model, early predictive design evaluation.

Abstract

Traditional engineering design discipline calls for designs to be evaluated long before they are implemented. Early design evaluations predict properties of the artifact that will result from a proper implementation of the design and the value of those properties to the client or end user. The predicted properties can include costs as well as functionality, performance, and quality measures. Software engineering has some such evaluation techniques but the discipline lacks a systematic way to explain, compare, develop, and apply them. We discuss the role of early predictive design evaluation in software design, show how a variety of specific predictors serve this role, and propose a unifying framework, *Predictive Analysis for Design* (PAD) for design evaluation techniques. We are especially interested in techniques that predict the value of the finished software system to its client or end user and that make the predictions before the expense of software development or integration is incurred. We show that our PAD framework, even in its preliminary state, is sufficiently expressive to be useful in explaining and characterizing design evaluation techniques. We argue that the PAD framework shows sufficient promise to justify further development toward a unified theory of early predictive design evaluation.

Table of Contents

Table of Examples

1. Early Evaluation in Engineering Design

Good engineering practice seeks cost-effective solutions to practical problems. It recognizes that the benefit of a design must be balanced against the cost of selecting that design as well as the cost of implementing the design. As a result, the objective is commonly cost-effectiveness rather than optimality.

The ability to consider a variety of designs and select the one most likely to deliver best value is an important element of this good practice. For software engineering, a significant point of leverage is improving our ability to explore a variety of designs to select the few that are worthy of deeper commitment. This requires substantial improvement in our capabilities for evaluating preliminary designs to understand the value they will deliver to clients.

This report proposes a model for Predictive Analysis of Design (PAD) that frames the evaluation questions and shows how to incorporate a variety of techniques – both existing and new – for predicting the properties that an implementation of a design will have and the value of the result to a client.

1.1 Engineering design and software design

In traditional engineering disciplines, it is usually very expensive to physically implement a design. As a consequence, engineering design involves sketching multiple preliminary designs, predicting the pertinent properties of reasonable implementations of those designs, selecting one or a few sketches for further development, and iterating through several cycles of refinement and evaluation before proceeding to detailed design and construction. Moreover, external constraints from both the problem and its context often constrain the space of viable designs.

Traditional engineering provides models and techniques for early design analysis that predicts properties of the product or artifact described by the design. For example, civil engineers use stress analysis models; these models can provide the basis for families of detailed designs such as the Pennsylvania simple bridge template. These models and design families consider the expected use as well as the technical properties of the implementations, and they can predict costs of materials and construction as well as structural properties. Indeed, traditional engineering seeks cost-effective solutions, and it recognizes that early decisions about a design strongly influence the cost and quality of the product.

Complex, expensive, software-intensive systems should also benefit from the discipline of performing early predictive design analysis. Such evaluations should predict properties that an implementation of the design can reasonably be expected to have, together with the uncertainty associated with the prediction. Most software development methods, however, are largely-linear processes that refine a single initial design. While most processes do allow for feedback and repetition of a phase, this typically provides a way to correct or refine the development, not a systematic way to generate and test a variety of alternatives.

We believe that software engineering would benefit from a generate-evaluate-select model similar to that of traditional engineering. This approach relies on the ability to make good predictions about the values of the alternative designs. Unfortunately, most software analysis techniques require access to source code or execution traces – in either case, the techniques assume that code exists. Code-based analyses are, of course, more precise than analyses that do not depend on code, but their dependence on code means they cannot be performed during early design, before actual code is available.

1.2 The need for better early design evaluation

Early decisions in software design set the overall capability and structure of the system. They are hard to change later in development, and they have a profound effect on delivered system quality. Boehm and Basili report that "finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase."[10]. In addition, Boehm reports that software cost and size estimates are least accurate in early stages of design, with estimates in the requirements phase often differing from final cost by factors of 2 or more[9].

Early-stage design decisions must, by their nature, be made on the basis of preliminary design information, without access to actual code. A number of empirically based effort predictors, such as COCOMO II[9], provide early estimates of development *cost*, and a number of analytic techniques, such as algorithmic analysis, provide early predictions about execution time. However, relatively few models support comparable early-stage predictions about other *technical properties* to expect of the actual implementations, let alone the value of these properties to a given client.

Yet these properties, including usability, security, fitness to purpose, fit with existing systems, scalability, evolvability, legality and other user- and context-specific capabilities, matter greatly to clients. Indeed, they may be as important to ultimate value of the software as the classic properties of general functionality, performance, dependability and cost.

The good engineering practice of sketching, evaluating, and iterating through designs depends critically on being able to evaluate a preliminary design in order to predict the properties it will have in practice. Models and techniques for doing this are severely lacking for software design. To the extent that they exist, design evaluation models for software usually emphasize functionality, performance,

reliability, and other quality properties. When models consider cost or risk it is usually based on an aggregate size rather than on design details or time-adjusted values. Further, they usually evaluate a design out of context, without reference to the preferences of its intended users or clients.

Thus we identify a point of leverage for software engineering research: creating models and techniques that do *early predictive design evaluation* without requiring access to the code to predict properties that can reasonably be expected of implementations of those designs.

1.3 A unified approach to early design evaluation

To date, very little technically-informed work has been aimed at valuing early-stage software designs. The available tools and techniques tend either to be impressionistic and technically problematic (e.g., reports of various "best practices" and consultants' recommendations), or theoretically sound but of limited applicability (e.g., abstract and/or small-scale models). Thus even crude estimates of value, based on technically informed appraisals of economic potential and sound valuation methodology are likely to be substantial improvements over current early-stage design practice and to provide a scientific basis for future, more refined, theories.

We believe that tools and techniques for early-stage analysis, by providing advance indications of implementation properties, will facilitate better early design decisions. These tools and techniques should be applicable in other contexts as well. The need to predict system properties without reference to code also arises when the code is not available for analysis or when the analysis would be intractable or more expensive than the value of the result. For instance, similar difficulties arise when designers are selecting COTS products for which only claims about the code, but not the code itself, are available and when designers are selecting strategies for evolving legacy systems for which the code may be available but very difficult to analyze with confidence.

In this report we lay out a unifying model for early predictive design evaluation. This model provides a framework for understanding the relation among individual predictive evaluation techniques. We believe that in the short run this model will provide an effective means of explaining, comparing, and teaching the techniques; further, we believe that over time this model can help to unify the collection of distinct techniques.

Our specific objectives are threefold:

- First, we seek to enrich the set of evaluation tools and techniques available for predicting properties of code from the design, before the code exists.
- Second, we aim to develop methods for describing user preferences about diverse properties in such a way that they can be used to predict value.
- Third, and most significantly, we wish to develop a way to integrate those preferences into a consistent and robust decision-making framework.

The framework we envision is based on principles of economic rationality and draws upon multi-attribute utility theory and the financial theory of real options.

The framework sketched here is intended as a basis for developing economically sound methods for early-stage design evaluation. More detailed objectives include:

- Ability to provide useful predictions without requiring access to code. We emphasize analysis that can be done with information that is available early in design
- Ability to estimate value of a software design, defined as benefit net of cost adjusted for time, uncertainty and optionality. We specifically want to address multi-dimensional benefits and costs, and in this work we focus specifically on value to the client or end user.
- Ability to provide useful predictions at the macro level, based on necessarily incomplete information. We are not trying to predict exact properties at the code level.
- Value estimates that are predictive. That is, they should be based on evaluating designs rather than code.
- Analyses that reflect user preferences and valuation functions, as well as those elements of the design that validly predict the implementation's properties along dimensions the user values.
- Systematic, applicable and useful knowledge, including both theoretical and empirical/statistical models. These include existing models from other fields, including engineering systems design, economics, finance and management.

The remainder of this report explores the character of a model for early predictive design analysis. Section 2 lays out the economic basis and describes the model. Section 3 develops the model with a series of examples that show the need for various elements of the model. Section 4 presents some scenarios showing how predictions of design value can be used. Section 5 examines some open questions and discusses ways to validate the model. Section 6 catalogs the predictors used as examples in this report and provides a starting point for a catalog of predictive techniques.

2. The Value of a Design

We are interested in the value of a design to a client or end user. More specifically, we are interested in predicting the value to a specific user or user community of an implementation of the design. In order to be useful, such predictions must depend only on the types and amounts of information that is available in early stages of the design, before the implementation is undertaken. Such a model will

also help us: (1) select from possible alternative designs by evaluating and then ranking them, (2) potentially explain to the user how their preferences affect the valuation of alternative designs, (3) provide the user assurances that the ranking among the designs will not change if uncertain input information is limited to specific range. Models that satisfy these requirements may also be well suited to predictions in other settings with incomplete knowledge. In particular, we are interested in prediction when the code is, for whatever reason, not available for analysis.

Our model of value includes not only the usual measures such as functionality, performance, and dependability, but also measures such as cost, risk, user distraction, usability, and fitness to purpose that are less common in software development; we will also consider measures that express a degree of uncertainty.

Our focus is

- very high level design, before "software development methods" start elaborating the box and line diagrams; to a certain extent we are interested in other situations where code may exist, but it is for some reason not suitable for analysis
- evaluation that considers costs and benefits to the user as well as the capabilities of the software
- evaluation that reflects value as perceived by the client, though we believe an essentially similar model can capture value for other stakeholders

A successful model must be

- open-ended, especially with respect to the evaluation models it handles and to the design notations and properties it supports;
- general and expressive, to accommodate a variety of specific models; in particular, it must accommodate multi-dimensional values for costs and benefits and for capabilities [41], and it must support composition of individual models;
- flexible, in that it accommodates estimates at various levels of detail and precision.
- appropriately correct and precise, in that it preserves the precision and accuracy inherent in the specific models and makes the degree of precision and accuracy of composite estimates clear

In the long run, we seek a unified quantitative model for predicting the value, in a technical sense, of a software design – that is, the value the system can be expected to have if we implement the design according to a given method. Naturally, these values will be imprecise and approximate, because knowledge early in design is imprecise and approximate.

We base our initial formulation on the economists' model of value. The remainder of this section presents the underlying economic model and our initial formulation for software design evaluation. Subsequent sections examine specific aspects of the framework, ways it may be used, and the challenge of developing and evaluating it.

2.1 The view from economics

Economists take value to be benefit net of cost. The value to a specific stakeholder evaluates benefit and cost from the viewpoint of that stakeholder. Different stakeholders see costs and benefits differently – for example, a consumer may not care whether a software system is maintainable, the vendor of that system may see maintainability as a significant benefit – the value of a specific product may be perceived differently by different stakeholders.

Economists use value as an integral part of decision making, for example about what set of products to manufacture from available input resources. As presented by any intermediate economics textbook, such as [31], a large part of microeconomics focuses on optimization problems, the most quintessential of which is the maximization of profit. Typically, a firm can create varying quantities of certain products, each of which requires a certain amount of various inputs. The firm's goal is typically to maximize the total revenue minus the cost of the inputs, as represented by the mathematical formulation

$$Max[B(z) - C(y)] \quad \text{such that } F(y, z) \leq 0$$

Here, the vector element z_j represents the quantity sold of product j, and $B(z)$ is the total revenue from selling those products. The vector element y_i represents the quantity of input i consumed, and $C(y)$ is the total cost of those inputs. $F(y, z)$ is a vector, as well, so $F(y, z) \leq 0$ represents a list of equations representing constraints on the problem (discussed further below).

In many competitive scenarios, the price of each product p_j can be assumed to be a constant. In that case, $B(z) = p \cdot z$. Likewise, in many situations, the cost per unit of input can be assumed to be a constant c_i, which implies $C(y) = c \cdot y$.

Connecting z to y is much more difficult. The form of this "production function" varies by industry, by firm, and by production technology. Consequently, it can be hard to concretely express the relationship between z and y. However, for illustrative purposes, it can be assumed that the function for the amount of product j can be represented with the Cobb-Douglas production function,

$$z_j = \prod_i y_{ij}^{\alpha_{ij}}$$

Here, y_{ij} represents the amount of input i used up in the generation of product j, and α_{ij} is a number between 0 and 1 representing the "elasticity" or efficiency of converting input i into product j. (For example, the production of steel might take the form steel_quantity = iron_quantity$^{0.7}$ x coal_quantity$^{0.2}$. Depending on what units are used for each material, additional constants may be required in this equation.) Note that under this formulation,

$$y_i = \sum_j y_{ij}$$

Substituting into the optimization equation,

$$Max[(\sum_j p_j \cdot \prod_i y_{ij}^{\alpha_{ij}}) - (\sum_j c_j \cdot \sum_i y_{ij})]$$

such that $F(y, z) \le 0$

The constraints typically express two types of requirements:

- Manufacturing cannot consume negative amounts of input. That is, $y_{ij} \ge 0$, which is typically written as $-y_{ij} \le 0$
- The amount of each input is constrained by natural or market forces. That is, $y_i \le q_i$, where q_i represents a constant upper bound on consumption of that input.

Once the problem has been expressed in the form described above, it is ripe for a Kuhn-Tucker analysis, which can produce an analytic solution. The basic process involves realizing that each constraint equation can be rewritten using an equals sign (rather than $\le$) through the introduction of a new variable for each constraint equation.

For example, the $-y_{ij} \le 0$ can be rewritten as $\lambda_{ij} y_{ij} = 0$, where λ_{ij} is zero any time that $y_{ij} < 0$, and λ_{ij} can be anything when y_{ij} is 0. Likewise, a new variable μ_i can be introduced for each of the constraints on total supply of each input. These λ and μ are variables that can be tweaked in order to achieve an optimal solution.

Because the left hand side of each constraint equation is now equal to zero, it can be added to $B(z) - C(y)$. This facilitates lumping all the constraints into the main optimization equation.

With these modifications, the basic optimization problem can be rewritten in the form

$$\begin{aligned} \underset{y_{ij}, \lambda_{ij}, \mu_i}{Max} \quad & [(\sum_j p_j \cdot \prod_i y_{ij}^{\alpha_{ij}}) \\ & - (\sum_j c_j \cdot \sum_i y_{ij}) \\ & + (\sum_{ij} \lambda_{ij} \cdot y_{ij}) \\ & + (\sum_i \mu_i \cdot (-q_i + \sum_j y_{ij})) \ ] \end{aligned}$$

The Kuhn-Tucker methodology for solving this essentially involves taking partial derivatives of this expression with respect to each y_{ij} and setting the partial derivatives to zero. Then, it is possible under many conditions to solve analytically for each variable to produce an optimal solution. (The proof for this procedure depends on visualizing the optimization expression as a function suspended over in a multi-dimensional space, and essentially looking for "flat" spots—a generalization of the single-dimension optimization procedure taught in introductory calculus.)

The key lessons of this procedure for are the following:

- Economists optimizing over the difference between a benefit function (which depends on outputs) and a cost function (which depends on inputs).
- The outputs are connected to the inputs by a production function
- A variety of constraints on inputs govern this problem.
- In economics, the variables are usually assumed to be continuous, which is vital to a Kuhn-Tucker reformulation of the problem as maximization over the space of input allocations (y_{ij}).

The most important lesson is the mindset behind the goal. Specifically, economists usually assume that a firm will attempt to maximize its profit. Other formulations are possible but less typical, such as optimizing the utility to the customer (benefits to customer minus costs to customer).

That final lesson is the main point of departure for the following discussion, which will emphasize a mindset of optimizing total utility *from the customer's standpoint*. Moreover, as will soon become clear, the other four bulleted lessons find their most natural expression in somewhat different forms when applied to the context of software engineering.

2.2 A model for predictive analysis of value early in design

We turn now to software design. We would like to make software design more like engineering, in which analysis of early design sketches shapes the direction of the design activity. We would like to do this with explicit attention to the value a product delivers to a user. To do this, we adapt the economic value model to the specific domain of software.

Software design is inherently different from the simple economics case because the product we're trying to evaluate does not yet exist at the time we need to evaluate it and our value space is not a simple continuous vector space. Value prediction for software requires a three-step analysis: first, determine whether an implementation is actually feasible; second, predict the properties that an implementation of a design is likely to have, and finally translate those implementation properties to the utility they have for a specific stakeholder. In the present work, we address value to the user, customer, or other client.

We propose a model for *Predictive Analysis of Design* (PAD). This model explicitly separates the tasks of predicting the properties of an implementation, determining feasibility of implementation, and associating client utility with the properties of an implementation.

The PAD model provides a unified framework, a sort of normal form, for describing and unifying predictive design evaluation techniques. The setting we have in mind is that we have available

- a design ***d*** and a (possibly incomplete) description of one or more clients' preferences $\boldsymbol{\theta}$;
- techniques for predicting some properties ***x*** that will result if ***d*** is implemented with method ***m;***
- techniques for estimating the value ***v*** to the client of certain properties of an implementation;

We assume that actual code is not available for analysis, perhaps because it has not yet been written, it is not available for inspection, or analysis is intractable .;

Naturally, the predictions are not guarantees; they therefore have associated uncertainties; we consider this in sections 3.8 and 3.9. Further, values itself generally have several components; we consider this in section 3.3.

Specifically, let

d be the design or partial design being evaluated, expressed in some appropriate notation[1] **D**

$\boldsymbol{\theta}$ be the client's preferences or utility characteristics, expressed in some appropriate notation Θ; these characteristics may depend on time and risk (e.g., the client may prefer speedier implementations and/or be risk averse).; $\boldsymbol{\theta}$ is the only representation of these preferences, so two clients with the same $\boldsymbol{\theta}$ are indistinguishable.

v be in $\mathbf{V}^n$, a multidimensional value space capable of expressing both costs and benefits to the client; in the simplest case, values are scalar dollars but more generally ***v*** may distinguish different classes of costs and values[2].

x be in $\mathbf{A}^n$, a multidimensional space whose dimensions correspond to the attributes of interest to the client. These dimensions form a subset of all the (eventually) observable properties of the artifact. In the simplest case, the values of the attributes are real numbers and $\mathbf{A}^n$ is $\mathbf{R}^n$, but more generally the values could be ranges, probability density functions, or other expressions of uncertainty. (In common usage, ***x*** is a property list, though some models may use a positional representation with attribute assigned to fixed positions)

m be a development method for implementing a design, expressed in some appropriate notation **M**

In adapting the model used by economists, we preserve the form of the function in our value function ***U;*** we replace the inequality constraints with a feasibility predicate ***F***; we allow the benefit and cost functions to take on multidimensional values, and we introduce a mapping ***P*** to capture the relation between the design and the properties of its implementation with some software development method.

We allow multidimensional values because software design involves software characteristics and user value terms that may be incommensurable or valued differently by different clients [41]. It may follow that there is not a unique design that is optimal for all users, so representing the space of design tradeoffs is useful in capturing aspects of interest to a software architect / designer.

Standard economic model deals only with cost and benefit. We explicitly model a predictor ***P*** to deal with the uncertainties that lie between design and implementation. These uncertainties arise not only for large complex systems, but even for smaller-scale components. For example, the algorithmic complexity of an algorithm predicts the scaling qualities of the resulting code. Selecting an O(n log n) algorithm over an $O(n^2)$ algorithm offers some assurance, but not an absolute guarantee, that the algorithm will be correctly implemented to achieve the predicted performance scaling.

Specifically, let

U: $\mathbf{D} \times \Theta \rightarrow \mathbf{V}^n$, the value function, be a function composing benefits and costs to predict overall value of a design ***d*** to a client with preferences $\boldsymbol{\theta}$; for simplicity we take it to be vector subtraction in $\mathbf{V}^n$: $\boldsymbol{U = B - C}$.

B: $\mathbf{A}^n \times \Theta \rightarrow \mathbf{V}^n$, the benefit function, be the predicted value of properties ***x*** with respect to the client's preferences, possibly re-evaluated dynamically; ***B*** captures the client's desires and the value that he or she places on characteristics of the product; hence it does not depend on the design or the method of implementing the design or the resources required to implement it

C: $\mathbf{D} \times \mathbf{A}^n \times \mathbf{M} \rightarrow \mathbf{V}^n$, the cost function, be the predicted cost resulting from implementing design ***d*** to achieve properties ***x*** using development method ***m***; ***C*** captures the resources required to produce the implementation. If the developers are part of the client's firm, it captures the concerns of the developers about realizing the design. If the developers are at another firm, ***C*** describes the

[1] For the fastidious, **D** is the set of legal expressions in the design notation, whatever that notation may be. Similarly, Θ and **M** are sets of legal expressions in whatever notations are appropriate.

[2] All the ***v*** in a single analysis must represent values of a single stakeholder, typically the client. To do otherwise would be to confound the client's and the producer's valuations. Thus, ***v*** would include dimensions of usefulness, such as usability and time when the product is available, as well as dimensions that capture the price paid.[3] Note that function point "files" are logical. They may or may not actually be implemented as entities resident in a file system.

detailed terms of the proposed contract plus any other costs the client will incur in deploying and maintaining the software.

P: $\mathbf{D} \times \mathbf{M} \rightarrow \mathbf{A}^n$, the method predictor, be the properties ***x*** to expect from implementing design ***d*** with method ***m***

F: $\mathbf{D} \times \mathbf{A}^n \times \mathbf{M} \rightarrow \{true,false\}$, the feasibility predicate, be a predicate indicating whether design ***d*** can be realized at all with properties ***x*** through method ***m***; ***F*** determines the feasible regions in the property space **A** and captures external constraints such as physical and legal limitations

Then the value of a design is

$$\boldsymbol{U}(\boldsymbol{d};\boldsymbol{\theta}) = \boldsymbol{B}(\boldsymbol{x},\boldsymbol{\theta}) - \boldsymbol{C}(\boldsymbol{d},\boldsymbol{x},\boldsymbol{m}) \text{ for } \{\boldsymbol{x}: \boldsymbol{F}(\boldsymbol{d},\boldsymbol{x},\boldsymbol{m})\}$$

where $\boldsymbol{x} = \boldsymbol{P}(\boldsymbol{d},\boldsymbol{m})$

This model can accommodate analysis techniques having various degrees of detail and precision. We expect it to allow for complex client preferences, to accommodate rich attributes and value terms including confidence indicators on estimates, to express the time value of resources, and to capture the information required to treat value in economic terms, including consideration of cost, uncertainty, and future contingencies [6].

3. Elements of the Predictive Analysis of Design Model

Section 3 develops the Predictive Analysis of Design (PAD) model incrementally. Each section introduces a new element of the model and gives examples that show how that element contributes to the model. Some of the model elements are introduced first in simplified form, then elaborated in later sections. In each section, the parts of the model that have been introduced are summarized in a shaded box, and the elements introduced in that section are highlighted within the box. The full model is

$\boldsymbol{U}(\boldsymbol{d};\boldsymbol{\theta}) = \boldsymbol{B}(\boldsymbol{x},\boldsymbol{\theta}) - \boldsymbol{C}(\boldsymbol{d},\boldsymbol{x},\boldsymbol{m})$ for $\{\boldsymbol{x}: \boldsymbol{F}(\boldsymbol{d},\boldsymbol{x},\boldsymbol{m})\}$, where $\boldsymbol{x} = \boldsymbol{P}(\boldsymbol{d},\boldsymbol{m})$

d in **D** (expressions in a design notation)

$\boldsymbol{\theta}$ in $\boldsymbol{\Theta}$ (expressions of utility)

v in $\mathbf{V}^n$ (multidimensional value space)

x in $\mathbf{A}^n$ (vector of attributes, or property list)

m in **M** (characteristics of design method)

U: $\mathbf{D} \times \boldsymbol{\Theta} \rightarrow \mathbf{V}^n$

B: $\mathbf{A}^n \times \boldsymbol{\Theta} \rightarrow \mathbf{V}^n$

C: $\mathbf{D} \times \mathbf{A}^n \times \mathbf{M} \rightarrow \mathbf{V}^n$

P: $\mathbf{D} \times \mathbf{M} \rightarrow \mathbf{A}^n$

F: $\mathbf{D} \times \mathbf{A}^n \times \mathbf{M} \rightarrow \{true,false\}$

We explain each step with examples that apply established and informal analysis techniques to concrete cases. In this way we illustrate the range of properties that it is possible to predict, the range of techniques that can be accommodated in the PAD model, and the range of confidence and precision provided by various predictive techniques.

Section 4 presents some scenarios about how these value predictions can be used. It addresses choosing designs to maximize client utility, composing individual predictors, and exploiting the systematic organization provided by the framework to improve design practice.

The analysis techniques used in Sections 3 and 4 include

- Benchmarking against expected usage Example 3.1.1
- Feature value analysis (hedonic analysis): Example 3.1.2
- Size, time and effort prediction (COCOMO II): Example 3.2.1, Example 3.4.2, Example 3.4.3, Example 4.3.1, Example 4.5.1
- Simulation (Acme Architecture Simulator): Example 3.2.2
- Task model with empirical inputs: Example 3.2.3, Example 3.6.2, Example 4.2.1
- Multidimensional analysis: Example 3.3.1, Example 4.1.1
- Algorithmic complexity: Example 3.4.1
- Subjective relative ranking (CBAM) Example 3.5.1
- Cost-benefit analysis: Example 3.6.1, Example 3.6.3
- Multi-attribute decision theory (SAEM): Example 3.6.3
- Tradeoff analysis (ATAM): Example 3.7.1
- GOMS family of user interface evaluation techniques: Example 4.5.1

[[[*Editorial note*: eventually, Section 6 will provide a side-by-side comparison of these and other predictive models]]]

The examples include two scenarios that develop along with the model, together with a variety of standalone examples.

- Representing line diagrams: Example 3.1.1
- Value of spreadsheet features: Example 3.1.2
- Design of a new program feature: Example 3.2.1, Example 3.4.2, Example 3.4.3, Example 3.5.1, Example 3.7.1, Example 4.3.1, Example 4.5.1
- Choosing security components: Example 3.2.2, Example 3.6.3
- Configuration for mobile user: Example 3.2.3, Example 3.6.2, Example 4.2.1
- Command and control: Example 3.3.1
- Choosing an algorithm: Example 3.4.1
- Value of installing an application: Example 3.6.1
- Choosing among products; Example 4.1.1

3.1 Value based on product attributes

At heart, the value of a product is its value to some stakeholder. In PAD, the stakeholder of interest is the client or end user. In the simplest form of the model, value is computed in scalar dollars, and attributes of the design are both known and easily valued in dollars.

In particular, this simple case considers a design ***d*** with a small number of known scalar real attributes (***x*** in $\mathbf{R}^n$), easy mappings ***B, C*** from those attributes to dollars (***v*** in **R**), The value ***U*** of the design ***d*** is then the benefits minus the costs. Naturally, the benefits and costs can be considered separately. The simple model is:

$U(d) = B(x) - C(x)$

d in **D** (expressions in a design notation)

v in **R** (scalar dollars)

x in $\mathbf{R}^n$

U: $\mathbf{D} \rightarrow \mathbf{R}$

B: $\mathbf{R}^n \rightarrow \mathbf{R}$

C: $\mathbf{R}^n \rightarrow \mathbf{R}$

We illustrate this simple model with an example of selecting a representation for a rich data set, Example 3.1.1.

Other examples include cryptographic systems, in which benefits must be balanced with costs. In many such systems, the amount of computation required to encrypt and decrypt a message increases with the degree of security provided. Placing dollar values on computation effort and degree of security places such a system in this simple model.

Example 3.1.1 Choosing a data representation

Suppose you want to store and share a set of large line diagrams, such as blueprints or road maps. These diagrams will be used mainly through a particular standard drawing application, and they will be edited as well as viewed. Your standard drawing application supports a variety of data formats. Five of these, AI, PDF, WMF, EMF, and EPS, preserve the structure of the images (other supported formats convert the drawing to a raster image that cannot subsequently be edited). The benefits of these representations are equivalent to you, so you're only interested in the costs.

For a sample of typical files, you determine the following:

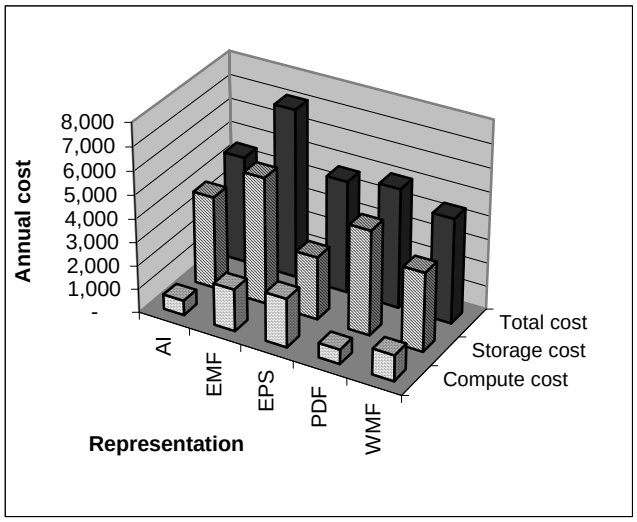
File type	Seconds to open (read)	Seconds to write (save or export)	File size (KB)
AI	6	93	6243
EMF	9	88	17908
EPS	5	17	20909
PDF	7	95	6243
WMF	5	86	11038

You estimate that out of every 50 times one of these files is opened, only 1 will result in changes and hence in a write. You estimate that the annual cost for every second of read/write time is \$10K and the annual cost for every KB of storage is \$0.1K (this reflects cost of bandwidth to move the files as well as raw storage and backup costs). Your task is to choose the least expensive representation.

By weighting reads 50:1 relative to writes and converting seconds and KBs to dollars, you arrive at the following, highlighting the minimum value in each cost column:

File type	Total time	Time cost	Space cost	Total cost
AI	393	\$ 3930	\$ **624**	\$ 4554
EMF	538	5380	1791	7171
EPS	267	**2670**	2090	4761
PDF	445	4450	**624**	5074
WMF	336	3360	1103	**4464**

You further visualize this as a chart to see how time and space contribute to overall cost:



Now you can choose a representation, namely WMF with the lowest overall cost. In terms of our model, you have five candidate designs. You have determined the attributes <time, size> for each, and you converted the result to a cost expressed in dollars; the best value for your application is the file format with the minimum cost.

Naturally, if your time and space costs change, or if the percentage of writes relative to reads changes, this analysis will also change.

Design *d* <format>	Attributes *x* <time,size>	Cost *v* <total \$>
AI	<393,6243>	\$4554
EMF	<538, 17908>	17908
EPS	<267, 20909>	4761
PDF	<445, 6243>	5074
WMF	<336, 11038>	**4464**

Naturally, if your time and space costs change, or if the percentage of writes relative to reads changes, this analysis will also change.

Example 3.1.2 Determining value of features

[[[*Editorial note:* Studies based on hedonic models, which attempt to correlate prices with the presence of features, reveal that spreadsheet software "which adhered to the dominant standard, the Lotus menu tree interface, commanded prices which were higher by an average of 46%." Moreover, features which result in positive network effects supply a powerful boost to prices: for every 1% increase in installed base, list prices were higher by 0.75%. At least for an "average" end-user, the benefit function ***B*** seems to rise steeply if spreadsheet software exhibits these characteristics [11] Anecdotally, we also understand that some software development organizations have experts who can make subjective judgments about the market value of a feature; the judgments of these experts has a significant effect on the features selected for development. An example based on this analysis is under development.]]]

3.2 Predicting attributes from design

The simple model assumes that the properties of an implementation are clear from a design. Often, this is not possible, and one of our principal objectives is the ability to predict the value of a design before actually investing in an implementation.

This section establishes an explicit relation between a design and properties of its implementation. It augments the design by introducing an explicit function to predict the properties of a design. The predictor function, **P,** predicts the attributes of a reasonable representation of a given design (for now, we will not worry about the development or implementation method, which we will consider in section 3.4).

We illustrate the use of predictor ***P*** with three examples: predicting code size from identifiable design elements, predicting technical properties with an architectural simulator, and predicting resource usage from a mobile computing design.

Other examples of interesting predictors include estimating development cost from design and code size (which we will consider later, in Section 3.4) predicting user time for a scenario with the keystroke model (which we consider in Example 4.5.1) and predicting round-trip time from queuing analysis of a design.

$\boldsymbol{U}(\boldsymbol{d}) = \boldsymbol{B}(\boldsymbol{x}) - \boldsymbol{C}(\boldsymbol{x})$ where $\boldsymbol{x} = \boldsymbol{P}(\boldsymbol{d})$

$\boldsymbol{d}$ in $\mathbf{D}$ (expressions in a design notation)

$\boldsymbol{v}$ in $\mathbf{R}^n$ (vector of scalars)

$\boldsymbol{x}$ in $\mathbf{R}^n$

$\boldsymbol{U}: \mathbf{D} \rightarrow \mathbf{R}$

$\boldsymbol{B}: \mathbf{R}^n \rightarrow \mathbf{R}$

$\boldsymbol{C}: \mathbf{R}^n \rightarrow \mathbf{R}$

$\boldsymbol{P}: \mathbf{D} \rightarrow \mathbf{R}^n$

Once we can predict an implementation's attribute set, which we represent as an element of a multi-dimensional real space, we can map these attributes into our benefit function **B** and cost function **C** to generate a net utility **U** = **B** − **C**. We will return to these three functions later, in Section 3.5.

Example 3.2.1 Predicting code size from function points

To demonstrate how an existing predictor fits the PAD model, we review the COCOMO II Function Point software sizing model, which predicts code size by examining the functionality embedded in a design [8]. We then introduce an example problem, a spreadsheet that supports data imports, to illustrate simple prediction of attributes, $\boldsymbol{P}$: $\mathbf{D} \rightarrow \mathbf{R}^n$. We return to this hypothetical "import to spreadsheet" problem in Example 3.4.2, Example 3.4.3, Example 3.5.1, Example 3.7.1, and Example 4.5.1.

The COCOMO II Function Point software sizing model begins by examining the design and counting the number of identifiable functional elements embedded in the design. Each functional element is categorized as one of five types and one of three complexity levels, as shown in the table below (reproduced from [9], which also describes how to properly categorize functional elements). Each cell of the table gives a function point count corresponding to the relative difficulty of actually implementing a functional element of that complexity and type.

	Complexity Level		
Type	**Low**	**Average**	**High**
Internal Logical Files[3]	7	10	15
External Interface Files	5	7	10
External Outputs	4	5	7
External Inputs	3	4	6
External Queries	3	4	6

For example, a design with five External Queries of Low complexity and two External Outputs of High complexity would yield 5*3+2*7 = 29 total function points.

This function point total is then multiplied by the average number of lines of code per function point. This multiplier varies by programming language; some samples appear below (taken from [9]).

Language	Lines of Code Per Function Point
ADA 95	49
C++	55
Java	53
PERL	27
Visual Basic 5.0	34

We now turn to a hypothetical example, the "import to spreadsheet" problem and show how the COCOMO II Function Point sizing model predicts code size, an attribute of an implementation..

Suppose that a vendor of office software has decided to enter the spreadsheet market. After reviewing requirements expressed by their user focus groups, as well as features offered by the competition, the firm determines that one significant need is a way to transport data from Microsoft Access into the spreadsheet. Soon, the firm begins evaluating possible designs which meet the requirements in different ways.

The design description **D** may use vectors of four Booleans:

< Does the design include connectors to databases? ,
Does it call for integrating with OS clipboard? ,
Does it include tracking and undoing changes? ,
Does it specify running tasks in their own threads?
>

The architects propose three particular designs $\boldsymbol{d} \in \mathbf{D}$ (where 1 and 0 indicate the presence and absence of design features, respectively):

A design with all four features, < 1 1 1 1 >
A design with only database connectors, < 1 0 0 0 >
A design with only copy/paste, < 0 1 0 0 >

These designs provide diffent levels of features and functionality in satisfying the given requirements. For example, the first design not only includes connectors to facilitate automatic extraction of data from the database, but also provides a variety of features to facilitate using the data after it arrives. The second design simply provides for automatic database import, but little in the way of additional usability functionality. The third design would require end users to manually copy and paste rows from the database into the spreadsheet.

Because of this wide range in functionality, these designs will result in implementations with differing amounts of source code. Although the product attribute space $\mathbf{A}^n$ would generally include many dimensions, we will focus on just one here, the projected code size,. We will consider additional dimensions in Section 3.4.

Based on guidelines in [8], we might evaluate our spreadsheet design space **D** as follows:

- Database connector: High complexity External Interface File
- Copy/paste: Average Complexity External Input plus Average Complexity External Output
- Track/undo: Average Complexity Internal Logical File plus Average Complexity External Input
- Multi-threaded architecture: High Complexity Internal Logical File (admittedly not a good match, but the best available)

Assuming that the implementation will be written in Java, and using the conversion factors shown above, we estimate the design size in thousands of lines of code (KSLOC) as

$$\text{Size} = (10*d_1 + 9*d_2 + 14*d_3 + 15*d_4) * 0.053$$

Thus, the proposals yield the following approximate sizes:

Design	Size (KSLOC)
All features < 1 1 1 1 >	2.54
Database < 1 0 0 0 >	0.53
Copy/paste < 0 1 0 0 >	0.48

This "import to spreadsheet" problem shows how to use an existing nethod to support the evaluation of one product attribute, code size. In Section 3.4, we examine the COCOMO II Early Design model more fully and use it to evaluate additional attributes of the product attribute space $\mathbf{A}^n$.

Example 3.2.2 Predicting technical properties with an architecture simulator

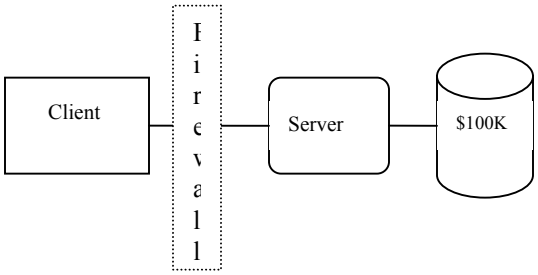
While the COCOMO II Function Point sizing model helps predict an element of cost by predicting KSLOC, Acme Simulator, an extension of Acme [25], can help predict a design's technical properties, such as performance and security. Acme is an architecture description language that software architects use for describing architectural structure, types and styles. Acme is useful in describing relatively simple designs, **D** producing a representation of a specific design ***d*** using the constructs of the architectural description language.

The Acme Simulator extends the capabilities of Acme by allowing the architect to describe data flows through the architecture and the capacity of the architecture's components and connectors. In addition, Acme Simulator extends the properties of component and connectors to allow the architect to describe security attributes of these architectural elements. The architect first uses Acme to describe a particular design and then uses the Simulator extensions to describe system loads and a threat model that describes the arrival rate and paths of various malicious

events. Once the architect describes systems loads and the threat model, the Acme Simulator simulates these loads and predicts the performance and security risk exposure of the design. Acme Simulator was developed as a tool that helps architects understand design tradeoffs.

Acme Simulator illustrates a predictor for attributes $\boldsymbol{P}$: $\mathbf{D} \rightarrow \mathbf{R}^n$, where $\boldsymbol{d}$ is the specified design and $\boldsymbol{x}$ is the set of technical attributes <performance, exposure>. In Acme Simulator, performance is evaluated as the average length of a transaction; for example, a transaction could take 2.1 seconds on average to complete. The architect defines Acme Simulator transactions, which are logical paths through a set of components. Exposure is the expected damage from a security compromise to the designs assets. Again, the architect identifies the design assets and assigns value.

Consider an example of a simple client-server-database architecture that has a database valued at $100K. The architect is considering adding a firewall between the client and the server, but must decide among three different types of firewalls, each with different levels of effectiveness and performance characteristics. The architect uses Acme to model the architecture and uses the extensions in Acme Simulator to specify the performance characteristics of each component, including the firewall. Furthermore, the architect specifies the periodic arrival rates of viruses and the expected damage, or exposure, to the database each time a virus reaches the database, which depends on the effectiveness of the firewall. The exposure of the design is the cumulative damage from viruses during the length of the simulation. Finally the architect specifies the system load, which is the average rate that a transaction starts.



	Firewall A	Firewall B	Firewall C
Performance	1.2 sec	.9 sec	.4 sec
Exposure	$50K	$63K	$75K

Based on the simulation specifications defined by the architect, Acme Simulator determines the average completion time for transactions and the cumulative damage of the database from viruses. Assuming that customers value both performance and security, the architect can use Acme Simulator to compare the performance and security risk exposure values of each design. From the table, the design with Firewall A appears to have the slowest performance, but the least amount of exposure. In contrast, the design that includes Firewall C has the fastest performance, but the most damage.

Although Acme Simulator might not be necessary for such a simple example, because the performance values and exposure values could be easily calculated, the values are not so easily calculated with more complex designs that could be subjected to numerous threats and have many different types of transactions. However, Acme Simulator is a tool that predicts technical attributes from a design and the example serves to reinforce how the Simulator fits into the unified framework.

Example 3.2.3 Predicting resource usage from configuration

This section presents an example from ubiquitous computing and demonstrates how the various elements of the example map to the PAD model.

The setting. A newspaper movie critic would like to work on a movie review before his flight departs. His task requires using a video player to view the various clips from an online movie database, a browser to search the web for information, and a text editor to take notes and update the review.

The challenges. The airport environment is dynamic and scarce in resources. Wireless bandwidth is limited and changing; finding a power outlet for a laptop computer can be difficult; computers in the airport kiosk lack in hardware resources and applications.

The problem. The movie critic cares about which applications he uses for the task, as well as the quality of service the applications deliver. Specifically, he has a preference for using the Windows Media Player over other video players, although he is willing to use others too. Also, he cares about the frame rate of the video and the sound quality, but not so much about the graphic detail of the frames. When using a web browser, he would rather have pages load fast than get rich content such as graphic images or embedded objects. We would like to help the movie critic to select a set of applications and to configure these applications in terms of quality of service delivered so that the user's preferences are satisfied optimally.

We describe how the example maps to the PAD model in three parts. In the first part, we describe the operating attributes of the problem, the design space, and how we predict the operating attributes from design points. In the

second part, captured in Example 3.6.2, we describe the prediction of user's utility with respect to operating attributes. And finally, in Example 4.2.1 we describe the optimization techniques we designed to solve the problem.

As indicated in the description of the scenario, the operating attributes are the user-perceived dimensions of the task. More formally, the operating attributes are the types of applications (formally called services) requested for the task: (1) play video, (2) edit text, (3) browse web, and the associated quality dimensions for each service. This table shows the quality dimensions of each service, and the potential applications to satisfy each service. In this case, $\mathbf{A}^n$ is a Cartesian product of 3 services and 5 quality dimensions:

Service	QoS Dimensions (units)	Available Applications
Play Video	Frame rate (frames per sec), frame size (pixels), audio quality (high, med, low)	RealPlayer, Windows Media
Edit Text	None	TextPad, MS Word
Browse Web	Latency (seconds), Content (text, med, high)	Internet Explorer, Firefox

The design space is the set of all configurations of the task. A configuration is the choice of applications, one per service, and the runtime state of the applications. This runtime state describes the predicted level of quality of service, and the resource demand to support this level of quality. Assume we have three scarce resources: CPU, network bandwidth, and virtual memory. Then the complete description of a design point ***d*** is:

d = { <Windows Media Player,
(24 fps, 300x200, high quality audio),
(25%, 256 Kpbs, 30 MB)>,
<MS Word,
( ),
(2%, 0 Kpbs, 28 MB>,
<Firefox,
(5 s, text),
(8%, 56 Kpbs, 10 MB)>}.

In practice, the design space for a typical task can have several thousand points.

Each point in the design space is a prediction about the runtime resource requirements. For example, <Windows Media Player, (24 fps, 300x200, high quality audio), (25%, 256 Kpbs, 30 MB)> is a prediction about the runtime resource requirements of watching a particular video stream encoded at 24 frames per second, has 300x200 pixel size, and high quality audio with Windows Media Player. The prediction is based on historical profiling (see, for example Odyssey [40]). The actual resource usage might vary depending on a particular video stream, but we hope that the variance is small enough to be practically useful in our case. When multiple applications are used simultaneously, we assume that the resources required are aggregated using vector addition. For example, the aggregate resource requirement for the above-mentioned configuration is (35% CPU, 312 Kbps, 68 MB), where the estimates are for CPU, network bandwidth, and virtual memory, respectively.

The description of the design space is not complete without estimates of resource availability. Assume, that at some point in time the resource availability is (100 % CPU, 220 Kpbs, 128 MB). We will see in Example 3.6.2 that the resource availability estimates constrain the feasible region of the problem.

3.3 Dealing with multidimensional quantities

Software design deals with attributes of many types; not all are scalar, and not all are interconvertable or interchangeable. Similarly, users often have many dimensions of value, and their preferences are not simple combinations of those dimensions. In predicting the value of a design, it is not enough simply to represent the costs and benefits as single scalar quantities such as dollars. Though convenient, this simplification fails to address the inherent incommensurability of different costs and benefits. As we have explained elsewhere (in [41]), these incommensurabilities can arise from a variety of sources, including differences in measurement scale level, granularity, economies of scale, and fungibility.

Actually converting different dimensions of cost and benefit into a single dimension requires accounting for the preferences of the customer, since each customer will have individual preferences for how to combine these dimensions (representing their own needs and context). As a result, we will now use a multidimensional range for the benefit function ***B***, cost function ***C***, and net utility function ***U***

When underlying spaces have multidimensional structure, it is often necessary to preserve the distinction of the individual dimensions when performing analysis. A classical example of this is the "colors of money" phenomenon in defense spending, where different budget accounts have different restrictions on the way funds are spent, and mingling funds from different accounts is prohibited; the funds in different accounts are said to be "of different colors". The source of this complication is the *fungibility* of different resources. Other important properties of resources include measurement scale, perishability, rivalry, etc. In general, when resources are not fully interchangeable, it is important to devise analysis

techniques that preserve the important properties of the resources. We call this *late binding*.

To respond to this common problem structure, we extend the PAD model by making both the attribute space and the value space vector spaces of arbitrary values. Note that in order for U to be defined, the ranges of B and C must be compatible.

$U(d) = B(x) - C(x)$ where $x = P(d)$
d in $\mathbf{D}$ (expressions in a design notation)
v in $\mathbf{V}^n$ (arbitrary values)
x in $\mathbf{A}^n$ (arbitrary values)
U: $\mathbf{D} \rightarrow \mathbf{V}^n$
B: $\mathbf{A}^n \rightarrow \mathbf{V}^n$
C: $\mathbf{A}^n \rightarrow \mathbf{V}^n$
P: $\mathbf{D} \rightarrow \mathbf{A}^n$

A class of problems where distinctions among the resources matter is multidimensional knapsack problems. In a multidimensional knapsack problem, there are multiple constraints, e.g. weight, volume, price, etc. The objective is to choose a subset of all the items that jointly maximize some measure of the value, while satisfying the knapsack constraints. For example, QRAM ([42]) is a scalable real-time solution to a multi-resource QoS problem, essentially a multidimensional knapsack problem. The QRAM algorithm computes a compound resource by taking weighted quadratic averages of the actual resources, and then solves a simpler one dimensional knapsack problem. To mitigate against obtaining an infeasible answer, the algorithm is done in phases, each time refining the weights. However, under some practically observable conditions, the answer that QRAM finds falls far from optimal. According to [27], the reason is that QRAM uses certain heuristic optimizations, which eliminate some feasible and valuable points in the design space from consideration. A solution presented in [27] correctly identifies such conditions and improves on the QRAM algorithm accordingly. Essentially, the conversion that QRAM does eliminated valuable design points *too early*; H-QRAM [27] delays the conversion to preserve those points.

Example 3.3.1 Preserving distinctions among resources

Let's look at a simplified example of a resource allocation involving multidimensional spaces. The example shows why early conversion of multiple resources into a single, unified resource might cause problems.

Continuing the configuration example from Example 3.2.2the first set of tables below show hypothetical runtime operational profiles of the two programs: Messaging and Real-time Video used by the battlefield commander for command and control. This operation profile includes quality of service that the program can deliver and the resource requirement. The resource requirement is in terms of percentage of bandwidth and CPU. The quality level information in the first column is provided by the application specification. The second and third columns give resource usage (percentage-of-resource-required/second) to achieve the specified quality level. The resource data depends on the runtime characteristics of the application and the data processed, which can be obtained using profiler tools. The value information in the fourth column is represents the value assessments of the battlefield commander, which can be obtained through elicitation interviews.

Messaging

Quality Level	CPU, %	BW, %	Value
None	0	0	-∞
Very Low	57	17	1
Low	61	23	12
Medium	72	27	55
High	79	29	68
Very High	98	32	75

Real-time Video

Quality Level	CPU, %	BW, %	Value
None	0	0	-∞
Bad	12	43	3
Acceptable	19	52	30
Good	23	69	45
Very Good	27	78	57
Excellent	34	93	89

The *unmodified* operational profiles of two applications, showing the bandwidth and CPU required for each quality level and the value to the battlefield commander

One (incorrect) approach to solving this problem is to take as given the external prices of CPU and Bandwidth, and convert these to a common currency. Assume the cost of one percent of available CPU is 2 units, and that of one percent of the available Bandwidth is 3 units. A total of 2 * 100 + 3 * 100 = 500 units of total resource are available. Table 4 presents the resource requirements in terms of the single currency. Column 2 shows the cost in common currency of providing that level of quality, and column 3 shows the percentage of that cost.

Messaging

Quality Level	Cost	Cost, %
None	0	0
Very Low	165	0.33
Low	191	0.38
Medium	225	0.45
High	245	0.49
Very High	292	0.58

Real-time Video

Quality Level	Cost	Cost, %
None	0	0
Bad	153	0.31
Acceptable	194	0.39
Good	253	0.51
Very Good	288	0.58
Excellent	347	0.69

Resource requirements of the battlefield applications in terms of the *converted cost measure*

Notice that according to table that depicts the resource requirements in terms of the converted cost measure, the best combination that can be achieved is High quality of Messaging and Good quality of Real-time Video, which costs 498 units, or just under 100%, and is valued at 113 (=68+43). However, after consulting the unmodified operational profiles, which show the resource requirement in bandwidth and CPU, we notice that CPU would be utilized at 101 percent, making that combination unattainable. The problem is that we have allowed conversion of unused Bandwidth into CPU, despite the inappropriateness of this conversion. Indeed, each quality point for either application can be obtained using only a unique CPU and bandwidth vector. The root of the problem is that CPU and bandwidth are not fungible, and our assumption of fungibility leads to an incorrect solution.

Another approach to this problem is to use derivatives, e.g. a calculus method called Lagrange Multipliers.

However, since the space of quality points is sparse, any kind of continuous approximation is likely to yield a solution that is also not in the space of available quality points.

QRAM offers a solution to this problem by iterating over the same solution we outlined several times, and refining the weights until a feasible, (near)-optimal solution is obtained.

3.4 Predicting attributes from design and method

As noted earlier, software valuation largely depends on the implementation, but during early design evaluation, we do not yet have an implementation to evaluate. Hence, we have introduced a predictor method ***P*** that predicts what implementation will result from a given design.

However, as touring a handful of software development firms will quickly reveal, the design alone does not predict the resulting implementation: the development method can have a powerful influence on a wide range of product attributes. For example, it can affect technical attributes, such as reliability and usability; it can also affect production attributes that affect the utility of the product, such as its time to market. It certainly affects consumption of inputs, such as staff effort and calendar time.

Hence, in this section, we extend the domain of our method predictor ***P*** to include another dimension, the development method ***m***. The development method is an element of an arbitrary notational space **M** selected to represent the range of development methods that might be employed in the implementation of a given design.

$U(d) = B(x) - C(x)$ where $x = P(d, m)$

d in **D** (expressions in a design notation)
v in $\mathbf{V}^n$ (arbitrary values)
m in **M** (characteristics of design method)
x in $\mathbf{A}^n$ (arbitrary values)
U: $\mathbf{D} \rightarrow \mathbf{V}^n$
B: $\mathbf{A}^n \rightarrow \mathbf{V}^n$
C: $\mathbf{A}^n \rightarrow \mathbf{V}^n$
P: $\mathbf{D} \times \mathbf{M} \rightarrow \mathbf{A}^n$

Example 3.4.1 Predicting execution profile from algorithmic complexity

[[*Editorial note:* Algorithmic complexity provides for an algorithm description of the way execution time or space scales as a function of problem size, under the assumption that the code competently implements the algorithm (this is, of course, not guaranteed by the algorithm). There is an established collection of analyzed algorithms, and their asymptotic growth is known. Obtaining a prediction requires simply consulting a standard reference. The example here will present an algorithm that has both space and size terms. ***d*** is the pseudo-code of the algorithm, **x** is the O(n) description of how time/space varies with problem size , **P** is the technique of algorithm analysis.]]

Example 3.4.2 Predicting development effort and time

As described earlier in Example 3.2.1, the Function Point sizing model within COCOMO II allows us to predict the code size from a design. In this section, we will consider how the COCOMO II Early Design Model offers two additional predictions, giving estimates of the development effort and time [9]. In casting COCOMO II as a method predictor, we take the design document to represent the design space **D,** and we include development effort and time in addition to code size in the product attributes $\mathbf{A}^n$.

COCOMO II uses the variable PM ("person-months") to represent development effort and predicts it as follows:

$$PM = Size^{E} * S$$

Here, Size is the code size in lines of code (which we estimated earlier in Example 3.2.1), and E and S are numbers that depend on additional firm, project, and product characteristics. We will return to these important issues in Example 3.4.3, but for now, we will hold these constant at E = 1.10 and S = 2.94 (which correspond to "nominal" levels of difficulty owing to firm, project, and product characteristics).

COCOMO II uses the variable TDEV ("time of development") to represent development time in months and predicts it as follows:

$$TDEV = PM^{F} * C * Scaling$$

Again, Size is the code size in lines of code, Scaling is an arbitrary factor indicating schedule compression by management, and the other variables are constants which can be adjusted but will remain constant at C=3.67 and

F=0.32 for now. (Scaling will generally be ignored in this section, but we will need to return to it in Example 3.4.3)

With this guidance from COCOMO II, we are now ready to return to our "import to spreadsheet" problem from Example 3.2.1. We had been evaluating three designs, each with a projected size. We can now use the PM and TDEV estimators shown above to fill in the development effort and time dimensions of our product space $\mathbf{A}^n$:

Design	Size (KSLOC)	Effort PM (PM)	Time TDEV (Months)
All features < 1 1 1 1 >	2.54	8.21	7.17
Database < 1 0 0 0 >	0.53	1.46	4.14
Copy/paste < 0 1 0 0 >	0.48	1.30	3.99

By extending Example 3.2.1, this example predicts a multidimensional product attribute vector ***x*** from a multidimensional design attribute vector ***d***. The design attributes were first used to predict one product attribute, code size, which was then used to generate two additional product attributes, development effort and time.

The presentation above may leave the impression that the development effort and time product attributes are fixed as soon as the code size attribute has been identified and that therefore this product attribute space has a "degenerate" dimensionality of 1 rather than a dimensionality of 3.

However, as we shall demonstrate in Example 3.4.3, COCOMO II bundles a great deal of complexity into the "constants" E, F, and S, meaning that development effort and time participate in a much richer interplay among product attributes. Later, in Example 4.3.1, we will see that separating each product attribute into a separate dimension yields a great deal of leverage for evaluating tradeoffs among these and other product attributes.

Example 3.4.3 Finding effect of product attributes on development time and effort

COCOMO II Early Design model predicts project effort in person-months (PM) and development time in months (TDEV) based on the size of the software as inferred from the design. It also depends on a characterization of the development method in terms of cost drivers called effort multipliers (EM) and scale factors (SF) plus four organization-specific fitting parameters A, B, C, and D [9]. It is these EM and SF quantities that drive the E, F, and S "constants" mentioned earlier in Example 3.4.2.

Upon examination, we find that eight of the cost drivers (plus A, B, C, and D) describe aspects of the development method, but four cost drivers actually describe product attributes. We separate these as follows:

- Properties demanded of the product: SCHED (adherence to a compressed development schedule), RCPX (required reliability and complexity), RUSE (investment in reuse), and PDIF (platform difficulty); call this set EM_P. Our main goal below is to recast COCOMO II in a way that highlights the relationship among person-months PM, development time TDEV, and these other four product properties. To achieve this, we will augment our product attribute space $\mathbf{A}^n$ to include each element of EM_P as an extra dimension, in addition to PM and TDEV.
- Properties of the development method: PERS (personnel capability), PREX (personnel experience), and FCIL (facilities); call this set EM_D. In addition, the scale factors PREC (precedentedness), FLEX (development flexibility), RESL (architecture resolution), TEAM (team cohesion), and PMAT (process maturity) also characterize the development method; call this set *SF*. We will represent our method description space **M** as vectors, where each element corresponds to one element of EM_D or *SF*. We will also include the fitting parameters A, B, C, and D as elements of this vector.

Each of the twelve COCOMO II cost drivers may take on a variety of values, ranging from "extra low" to "extra high"[4].

Continuing with our "import to spreadsheet" example problem, we augment our product attribute space $\mathbf{A}^n$ with our six additional product properties to yield the new representation:

< Time to import 100 rows from Access,
Time to begin an import then cancel,
SCHED, RCPX, RUSE, PDIF, Size, PM, TDEV >

Representing the method description as outlined above yields:

<A, B, C, D, PREC, FLEX, RESL, TEAM, PMAT,
PERS, PREX, FCIL>

COCOMO II then relates PM and TDEV to the other attributes as follows:

$$S = A * \Pi EM_i$$
$$S_{NS} = A * \Pi EM_i \text{ excluding } EM_{SCHED} \in EM_P$$

$$E = B + 0.01 * \Sigma SF_i$$
$$F = D + 0.002 * \Sigma SF_i$$

$$PM = Size^E * S$$
$$PM_{NS} = Size^E * S_{NS}$$

[4] Specifically, seven possible ratings exist ("extra low," "very low," "low," "nominal," "high," "very high," and "extra high"); however, technically, "extra low" and "extra high" are excluded options for some attributes, meaning that an alternative such as "very low" or "very high" must be used instead.

$$TDEV = PM_{NS}{}^{F} * C * Scaling$$

where EM_i ranges over EM_D and EM_P, and SF_i ranges over all *SF*.

For the following discussion, we will choose sample values for implementation properties in $\mathbf{A}^n$; in Example 4.3.1 we show how to evaluate the interaction among these properties. For simplicity we will henceforth use the nominal values A=2.94, B=0.91, C=3.67, and D=0.28.

Each EM_i and SF_i is converted from text (such as "very high") to a numerical value by looking up the cost driver elements of $\mathbf{A}^n$ and **M** in tables provided as part of the definition of COCOMO[5]. For example, if $\boldsymbol{x} \in \mathbf{A}^n$ shows minimal schedule pressure, reliability, reusability, and platform difficulty, in an ideal software organization with process perfectly suited to this project, then S, E, and F will take on minimal values (S=0.23, E=0.91, F=0.28). In contrast, for software with maximal schedule pressure, reliability, reusability, and platform difficulty, in an organization with the worst possible process, S and E take on maximal values (S=178.4, E=1.23, F=0.34). Taking the median ("nominal") values yields moderate values (S=2.94, E=1.10, F=0.32).

Note that S_{NS} is essentially the same as S, except that the cost driver corresponding to SCHED (schedule compression) is omitted. Thus, PM_{NS} is essentially the same as PM, except that it does not take into account any effects on development effort due to schedule compression. That is because TDEV incorporates this a different way, by multiplying by a Scaling constant corresponding to the different levels of EM_{SCHED}, as follows:

Scaling	EM_{SCHED}
1.60	1.00
1.30	1.00
1.00	1.00
0.85	1.14
0.75	1.43

For example, if the schedule is compressed by 15% (Scaling=0.85) compared to a nominal schedule, then EM_{SCHED}=1.14 rather than 1.0. However, this EM_{SCHED} is only used in calculating S and PM, not in calculating S_{NS} and PM_{NS}. COCOMO II only handles the EM_{SCHED} cost driver in this odd way, through an indirect Scaling variable; all the others are simply combined directly to yield the S, E, and F variables.

For each proposed spreadsheet import design, we can consider the best, nominal, and worst case conditions. The "easiest conditions" correspond to a project with minimal schedule compression (Scaling=1.6), complexity, reusability constraints, and so forth; conversely, the "hardest conditions" demand maximal schedule compression (Scaling=0.75), complexity, reusability constraints, and so forth. These yield the minimal and maximal values of S, E, and F, respectively, which we insert into the formulas for PM and TDEV:

Design	PM for easiest conditions	PM for nominal conditions	PM for hardest conditions
All features < 1 1 1 1 >	0.54	8.21	560.57
Database < 1 0 0 0 >	0.13	1.46	81.90
Copy/paste < 0 1 0 0 >	0.12	1.30	71.98

Design	TDEV for easiest conditions	TDEV for nominal conditions	TDEV for hardest conditions
All features < 1 1 1 1 >	4.93	7.17	21.37
Database < 1 0 0 0 >	3.31	4.14	11.04
Copy/paste < 0 1 0 0 >	3.22	3.99	10.56

(Note that, despite the tighter scheduling compression in the "hardest conditions," TDEV is still pushed up rather high, owing to the surfeit of pressures from other cost drivers.)

The predicted value of PM and TDEV for any given design thus varies greatly, depending on what other product attributes (SCHED, RCPX, RUSE, PDIF) are desired, and based on what development method $\boldsymbol{m} \in \mathbf{M}$ is in effect.

Clearly, this level of analysis alone does not suffice for selecting a design. First, because of the huge range between minimal and maximal values of S and E (and the resulting wide range between minimal and maximal PM), the firm must carefully evaluate how well their development method $\boldsymbol{m} \in \mathbf{M}$ suits the project at hand.

Second, recognizing that the dimensions of $\mathbf{A}^n$ represent linked variables, the organization must consider how much to invest in the schedule pressure, reliability, reusability, and platform difficulty attributes. Each of the choices of free variable represents a tradeoff decision. For example, adding schedule pressure gets the product to market faster but also raises PM. Likewise, supporting additional platforms may allow more customers to benefit from the product but at a higher PM. Alternatively, the organization

[5] Although organizations can recalibrate these tables, as well as A and B, using organization-specific data, the generic tables provided with the COCOMO II definition suffice for present purposes.

may choose to hold PM constant and instead trade off schedule pressure against platform support.

Thus, in order to act optimally, the organization must explicitly consider the benefit and cost functions, as described in Section 3.5, in addition to the tradeoffs as examined in Example 4.3.1.

3.5 Cost and benefit functions

In Section 3.1 we considered simple cost and benefit functions where the costs and benefits were expressed in dollars and the mapping from product attributes to dollars was simple. In Section 3.3 we saw that both product attributes and values may be multidimensional and nonscalar – indeed, they may not be expressed on a ratio scale. Here we add the complexity that cost depends not only on the attributes of the product but also on the cost of producing that product. The cost of production depends, of course, on the development method and, in some cases, the design. We consider similar complexities in the benefit function in Section 3.6

$U(d) = B(x) - C(d,x,m)$ where $x = P(d,m)$
d in $\mathbf{D}$ (expressions in a design notation)
v in $\mathbf{V}^n$ (arbitrary values)
m in $\mathbf{M}$ (characteristics of design method)
x in $\mathbf{A}^n$ (arbitrary values)
U: $\mathbf{D} \rightarrow \mathbf{V}^n$
B: $\mathbf{A}^n \rightarrow \mathbf{V}^n$
C: $\mathbf{D} \times \mathbf{A}^n \times \mathbf{M} \rightarrow \mathbf{V}^n$
P: $\mathbf{D} \times \mathbf{M} \rightarrow \mathbf{A}^n$

[[*Editorial note:* the whole discussion below this point, including the example, is really about benefit and utility. It will be moved to Section 3.6 in the next revision. We will install here an example that involves a rich function from product attributes and development method to user value. Perhaps trading precision for time/cost; dependability vs costs (e.g., redundancy, N-version programming; security vs encryption time)]]

Because not all customers will realize equal benefits and costs from product attributes, we have chosen to separate the modeling of benefits and costs into separate functions B and C.

Models of the benefit function B remain relatively undeveloped. Nonetheless, while the benefit of software features depends on the needs of specific users (as modeled in the preferences function θ, introduced in Setion 3.6), researchers have made strides in assessing broad aspects of feature valuation by customers. Previously, inExample 3.1.2, we described the overall market value of spreadsheet features.

We anticipate that continued studies along this line will reveal the broad outlines of the benefit function B for a given class of software. Resulting models of B will offer the highest predictive power if they take note of how B depends on the preferences function θ, representing the varying needs of diverse end-users. The ultimate goal is to select a design which maximizes the net utility $U = B-C$ to a client. Researchers have already begun exploring how to solve such problems, for example in the context of mobile systems [42] as described in Example 3.6.2.

In terms of costs, it is easy to convert the person-months estimated by an effort predictor such as COCOMO II into dollars by assuming a fixed conversion factor. While this may provide an adequate cost estimate for some organizations, others may require a more sophisticated approach. For example, firms developing shrink-wrapped software products will probably need to amortize the costs over some large number of customers. Others might not pass on any of these costs to the end user (as in the case of Internet Explorer), preferring instead to use the product to reinforce network effects for another product. Each strategy requires a unique cost function describing how product attributes generate costs to customers.

Example 3.5.1 Costs and benefits of an implementation

The Cost Benefit Analysis Method (CBAM) provides a way to evaluate the costs and benefits of a proposed implementation[34].[6] This section will focus on describing how to use CBAM to support evaluating the Benefit Function B: $\mathbf{A}^n \times \Theta \rightarrow \mathbf{V}^n$ of our model.

CBAM begins by having all stakeholders meet together, including customers and engineers. They identify a set of scenarios describing key criteria for evaluating the value of implementations. For each scenario, the customers specify what characteristics they would expect to see in a "best case" implementation, what they consider "desirable," and what they would barely tolerate in a "worst case" implementation.

Next, they specify how many points an implementation would win for achieving the worst, desirable, and best case versions of the scenarios. Finally, they assign weights for each scenario, indicating how much they value support for that scenario. The engineers then calculate a weighted sum of points for each implementation.

To demonstrate how CBAM supports evaluating our Benefit Function, let us return to the "import to spreadsheet" example introduced in Example 3.2.1and consider a simplified product attribute space $\mathbf{A}^n$ represented as

[6] CBAM generally follows ATAM, a method discussed later in this document in the context of the Feasibility Function. However, CBAM also can be used on its own, as shown above.

< Time to import 100 rows from Access,
Time to begin an import then cancel halfway thru,
Support for Windows and Linux >

Here, the first and second elements are real numbers in units of seconds, and the third can take one of three values:

L = Latest version of Windows only

M = Most versions of Windows back to version 3.11

H = Has support for all versions plus Linux

(As a mnemonic, this labeling also conveniently corresponds to the Low, Medium, and High platform difficulty PDIF values mentioned in the earlier discussion of COCOMO II.)

The stakeholders might identify the following scoring rules:

	Import	**Import + Cancel**	**Platform Use**
Best	< 2 sec 100 pts	< 2 sec 100 pts	H 100 pts
Desired	2-60 sec 70 pts	2-60 sec 90 pts	M 95 pts
Worst	>= 60 sec 5 pts	>= 60 sec 10 pts	L 40 pts
Weight	**35%**	**15%**	**50%**

Suppose that six sample implementations shown below are under consideration. Each of these offers a different combination $\boldsymbol{x}$ of product attributes $\mathbf{A}^n$. We can calculate the scenario score for each implementation as follows:

	Import	**Import + Cancel**	**Platform Use**
<4 2 L>	70 pts	100 pts	40 pts
<4 104 L>	70 pts	90 pts	40 pts
<4 104 M>	70 pts	90 pts	95 pts
<300 200 L>	5 pts	10 pts	40 pts
<300 200 M>	5 pts	10 pts	95 pts
<300 200 H>	5 pts	10 pts	100 pts

CBAM has thus produced a three-dimensional representation of the value space $\mathbf{V}^n$, where each dimension corresponds to the number of points that an implementations scores on one scenario. For example, the benefit of the first implementation would be <70 pts, 100 pts, 40 pts> $\in \mathbf{V}^n$.

CBAM generally would then use the scenario weights identified earlier (35%, 15%, and 50% in our example) to collapse these benefit vectors down to a scalar. The purpose of doing this is to provide a single "benefit" number which can be divided by a single "cost" scalar that CBAM generates costs using a tabular process fairly similar to the benefit approach outlined above. CBAM calls the ratio of scalar benefit to scalar cost the "return on investment" (a slightly different use of the term than in traditional finance), and suggests that the engineers should select the implementation which yields the highest ROI.

While collapsing costs and benefits into scalars might be a reasonable way to ultimately select an implementation, it is not reasonable if the weights depend on the context in which the application will ultimately be used. For example, in some contexts, the import features may find heavy use, whereas in others, the import features might be completely useless. Thus, for the purposes of our model, we prefer to stop here at a vector representation $\boldsymbol{v} \in \mathbf{V}^n$ without multiplying against the scenario weights, which are better represented as the user utility $\boldsymbol{\theta} \in \Theta$ discussed by the following section.

3.6 User utility

In this report we are chiefly concerned with predicting value to the user. This is a departure from most work in software engineering, which assumes that a specification or requirement document establishes the criteria for evaluating a software product. In actuality, though, different users have different expectations of software products – notwithstanding the commitments and limitations of the requirements, specification, or other product description.

To reflect this, we introduce an explicit term to represent the user preference: $\boldsymbol{\theta}$, the expression of user utility. We augment the model by adding $\boldsymbol{\theta}$ as a parameter of both the benefit function (since benefit is in the eye of the beholder) and the utility function (since we're evaluating utility of a design to a specific user.

$\boldsymbol{U}(\boldsymbol{d},\boldsymbol{\theta}) = \boldsymbol{B}(\boldsymbol{x},\boldsymbol{\theta}) - \boldsymbol{C}(\boldsymbol{d},\boldsymbol{x},\boldsymbol{m})$ where $\boldsymbol{x} = \boldsymbol{P}(\boldsymbol{d},\boldsymbol{m})$

$\boldsymbol{d}$ in $\mathbf{D}$ (expressions in a design notation)

$\boldsymbol{\theta}$ in Θ (expressions of utility)

$\boldsymbol{v}$ in $\mathbf{V}^n$ (arbitrary values)

$\boldsymbol{m}$ in $\mathbf{M}$ (characteristics of design method)

$\boldsymbol{x}$ in $\mathbf{A}^n$ (arbitrary values)

$\boldsymbol{U}: \mathbf{D} \times \Theta \rightarrow \mathbf{V}^n$

$\boldsymbol{B}: \mathbf{A}^n \times \Theta \rightarrow \mathbf{V}^n$

$\boldsymbol{C}: \mathbf{D} \times \mathbf{A}^n \times \mathbf{M} \rightarrow \mathbf{V}^n$

$\boldsymbol{P}: \mathbf{D} \times \mathbf{M} \rightarrow \mathbf{A}^n$

In the simplest case, $\boldsymbol{\theta}$ would be static. It could instead be a function of context or time, including user task shift and changing resources, and in addition, differ across users.

First, we can try to capture some current approaches by selecting the elements of $\boldsymbol{x}$ correctly.

Second, we can capture simple preferences with a preference function that's simply a weighted composition of the values ("we give 60% weight to correctness, 40% to reliability"). We can also handle complex preferences (nonmonotic utility, conditional preferences or joint utilities) through preference functions that do the appropriate stuff (whatever that is). Note that by separating the cost function we make it hard to say "we give 35% weight to on-time delivery, 35% to cost, 30% to correctness" -- we are in effect taking a hard line that the attributes are attributes of the product, not of the processes that produce it.

Third, we can capture the behavior of cost over time by incorporating present-value terms in both P and C;

Example 3.6.1 Value of installing an application

In order to make this model concrete we work through a simple example. A company that conducts e-commerce periodically suffers from denial-of-service attacks. The security manager is considering whether to invest in a specific type of load balancing software, which costs $20,000. This type of software should help mitigate the damage from these attacks, estimated at $50,400/year based on a recent risk analysis. The software will increase the system's ability to accept and handle incoming connections. Under a prolonged denial of service attack system performance will degrade and, at some point, the load balancing software is saturated and the system becomes unavailable.

In this case products are COTS, so $\boldsymbol{m}$ is simply purchase and install. We carry values $\boldsymbol{v}$ as scalar dollars. $\boldsymbol{d}$ is the software architecture with the addition of the load balancing component and $\boldsymbol{C}$(d, x, m) is the cost of the component, which includes purchase, implementation, and maintenance costs. Finally, the benefit of adding the load balancing component is the difference in lost profit between the architecture with the component and without the component

$$\boldsymbol{B_{lb}(x, \theta) = Cost\ of\ DoS - Cost\ of\ DoS_{lb}}$$

In this example the security manager is interested in two properties that will affect customer revenues: availability and performance. As system performance degrades, a percentage of legitimate users will not be able to use the system, so there will be a percentage of lost sales commensurate with the degree of performance. For the period that the system is unavailable the company loses all sales. Therefore, the vector of $\boldsymbol{x}$ is <performance, availability >. The security manager can determine the expected lost revenue due to unavailability and degraded performance. In addition to revenues lost directly, there may be some additional losses as some customers switch over to rivals. The fraction of lost future contribution is assumed to be proportional to lost current sales and is represented by $\boldsymbol{\theta}$. Suppose that the security manager has derived the following step function for $\boldsymbol{\theta}$:

If the system is unavailable for < 2 hours then $\theta = 1$
If the system is unavailable for 2- 4 hours then $\theta = 2$;
If the system is unavailable for > 4 hours then $\theta = 3$

Then, if

R = Average Contribution/hour of operation = $500

D = Average duration of DoS Attacks = 4 hours

L = Expected rate of attacks per year = 12

L * D* R = $24,000

$P(d_{lb}, m) = x = <$(availability, 50%)(performance, 60%)

$\boldsymbol{\theta}$ = 2 (assumed average availability with the load balancing software is 2 hours) the cost of a DoS under load balancing, DoS_{lb}, is defined as:

DoS_{lb} =

L * R * D * [(1- availability) $\boldsymbol{\theta}$ + (availability * performance)]

The benefit of load balancing is:

$50,400 – $31,200 = $19,200

So the value of the architecture with load balancing is:

$19,200 - $20,000 = -$800

Clearly, the security manager should consider a different product or a different risk-mitigation technique since the load-balancing product does not produce a positive net value.

Example 3.6.2 Predicting benefit of a configuration to a user

In this section, we continue describing the scenario about the movie critic working in the airport. Specifically, we describe the benefit function of the user with respect to the observable attributes of the design.

Recall that the observable attributes of the design include the application chosen for each service in the task and the quality of service dimensions of these applications. Given these observable attributes, we would like to be able to evaluate the points in the design space. We choose to represent the value space of the user as the one-dimensional space of positive real numbers. Further, we capture user's utility as a set of mathematical functions, each mapping an attribute to the value space and combine these functions using weighted addition. Thus, the utility of the design is a linear combination over the attributes. Notice that the utility functions are unique to each user, and can potentially be different depending on the task and context.

Let's see examples of utility functions in more detail. Suppose a user's utility for a specific video player is as follows: U(Windows Media Player)=1.0,

U(RealPlayer)=.8, and the weight for that choice is .5. Utility with respect to the quality dimensions of "play video" service is described in the table.

	Frame Rate	Frame Size	Audio
Weight	1.0	.5	1.0
Utility function	U(10 fps)=.1 U(18 fps)=.5 U(24 fps)=1.0	U(150x100) = .9 U(300x200) = 1.0 U(600x400) = 1.0	U(low)= .3 U(med = .5 U(high)=1.0

Using these utility functions, we can evaluate the design point: <Windows Media Player, (24 fps, 300x200, high quality audio), (25 %, 256 Kpbs, 30 MB)>. The value of that point is .5 * U(Media Player) + 1.0 * U(24 fps) + .5 * U(300x200) + 1.0 * U(high) = .5 + 1.0 + .5 + 1.0 = 3.0. Notice that the task contains other services, so to obtain the complete value of a configuration, we need the utility functions for "edit text" and "browse web" services, and need to do more similar arithmetic. For the sake of brevity, we omit showing the entire example, but we trust the reader has an idea of how user's utility helps compute the valuation of a specific point in the design space.

The utility functions allow us to evaluate different points in the design space, ultimately helping to find one configuration (one point in the design space) that is optimal or near-optimal from the user's point of view. In that respect, the utility functions are predictions of user's actual preferences.

In Example 4.2.1 we will show how the structure of the utility functions can be used to efficiently search the design space for alternatives that maximize value to the user.

Example 3.6.3 SAEM

The Security Attribute Evaluation Method (SAEM) is a cost-benefit analysis method that helps security engineers select security components for information systems. SAEM takes a security engineer through a three step process that: 1) prioritizes information system security threats 2) assesses the benefit of security components with respect to those threats, and 3) evaluates the security architecture for defense-in-depth gaps[16][17]. The result is that the security engineer can close gaps in the security architecture by selecting security components that provide the greatest value. SAEM uses an additive model from multi-attribute analysis techniques [35] to compute the relative risk of a threat and determine the value of a security component.

[[*Editorial note:* This example is in preparation. Here is the mapping from SAEM into the PAD Model]]

d is the base security architecture design, which consists of a set of security technologies

d_{st} is the security architecture design with the addition of security technology ***st***

θ is the risk tolerance of the organization

v is the Threat Index (TI) that SAEM computes using the additive value function

m is the selection of an established product

x is the set of assets that are at risk from a set of threats. The user is generally concerned about personal assets such as credit card numbers, medical information, etc.

$P(d_{st}, m)$ predicts the values of ***x*** using SAEM for design d_{st}

$C_{st}(d, x, m)$ is the cost of the security architecture with ***st***. Although not include in SAEM, cost would include purchase, maintenance, training, etc.

$F(d_{st}, x, m)$ eliminates infeasible combinations of the technology in the architecture, such as multiple authentication mechanisms.

$B_{st}(x, \theta)$ of a security technology is defined as:

$$\Sigma_{j=threats}(TI_t - TI_{t,st})$$

where TI_t is the Threat Index of a threat before the technology is added to the security architecture and $TI_{t,st}$ is the Threat Index of a threat after the technology has been added to the security architecture.

Therefore,

$$U_{st} = B_{st}(x,\theta) - C_{st}(d,x,m)$$

Intuitively, the benefit of a security technology is the difference in risk between a security architecture with and without the technology. Although the value of the technology depends on its contribution to risk reduction and the cost of technology, SAEM computes the benefit using non-dimensional TI units. Before the security manager could compute the value of a security technology, the benefit or cost function would need to be converted to similar units.

One limitation of the SAEM benefit function is that θ is not used to determine the value of a security technology. Ideally, the benefit of a security technology should be a function of its effectiveness in reducing risk and the user's tolerance for risk. For example, an organization that is risk tolerant may not see the need to add a security technology if they find that the current security architecture achieves an acceptable level of risk. In contrast, an organization that is risk adverse might find increased benefit in a technology that helps achieve an overall acceptable level of risk.

3.7 Feasibility

There is an old adage among engineers: "Good, fast, cheap—pick two." Sometimes it is impossible to produce a product with an intended level of quality within a certain schedule and budget, given the technology and development method currently available. Put in terms of

our model, it is sometimes infeasible to produce a certain set of product attributes $\boldsymbol{x} \in \mathbf{A}^n$ from a given design $\boldsymbol{d} \in \mathbf{D}$ using a development method $\boldsymbol{m} \in \mathbf{M}$.

In this section, we introduce the feasibility predicate ***F***, which indicates whether a given implementation can be achieved from a design using a certain development method. We focus on one specific qualitative method for evaluating feasibility, but other more quantitative approaches, such as deadlock detection with model checkers and rate monotonic analysis, would be appropriate in some circumstances.

$\boldsymbol{U}(\boldsymbol{d};\theta) = \boldsymbol{B}(\boldsymbol{x},\theta) - \boldsymbol{C}(\boldsymbol{d},\boldsymbol{x},\boldsymbol{m})$ for $\{\boldsymbol{x}: \boldsymbol{F}(\boldsymbol{d},\boldsymbol{x},\boldsymbol{m})\}$,
where $\boldsymbol{x} = \boldsymbol{P}(\boldsymbol{d},\boldsymbol{m})$

$\boldsymbol{d}$ in $\mathbf{D}$ (expressions in a design notation)

θ in Θ (expressions of utility)

$\boldsymbol{v}$ in $\mathbf{V}^n$ (arbitrary values)

$\boldsymbol{m}$ in $\mathbf{M}$ (characteristics of design method)

$\boldsymbol{x}$ in $\mathbf{A}^n$ (arbitrary values)

$\boldsymbol{U}: \mathbf{D} \times \Theta \rightarrow \mathbf{V}^n$

$\boldsymbol{B}: \mathbf{A}^n \times \Theta \rightarrow \mathbf{V}^n$

$\boldsymbol{C}: \mathbf{D} \times \mathbf{A}^n \times \mathbf{M} \rightarrow \mathbf{V}^n$

$\boldsymbol{P}: \mathbf{D} \times \mathbf{M} \rightarrow \mathbf{A}^n$

$\boldsymbol{F}: \mathbf{D} \times \mathbf{A}^n \times \mathbf{M} \rightarrow \{true,false\}$

Example 3.7.1 Determining feasibility with ATAM

The Architectural Tradeoff Analysis Method (ATAM) provides a qualitative way of evaluating whether a proposed design and development method can produce an intended set of implementation attributes [32]. The ATAM approach involves a series of meetings between stakeholders, including software engineers and customers, to discuss requirements as well as proposed architectures. These meetings aim toward three goals.

First, the conversation initially centers on clarifying what quality attributes would likely benefit the customer most. It may turn out, for example, that speed and flexibility are the most important attributes. However, such attributes are usually abstract and vague, so the stakeholders also attempt to operationalize what these concepts mean in the context of this particular system. Brief concrete scenarios are recorded in order to document this operationalization in a format that customers can understand and verify. This aspect of ATAM can be viewed as identifying which dimensions of the implementation space $\mathbf{A}^n$ merit focused consideration in the discussion that follows.

The second goal of the ATAM meetings is to understand the system's current architecture. Although ATAM was initially inspired by military software development projects, for which there usually exists an enormous base of legacy code, other situations abound where preexisting code strongly influences design decisions in the present. This aspect of ATAM can be viewed as identifying which dimensions of the design space **D** are "free variables."

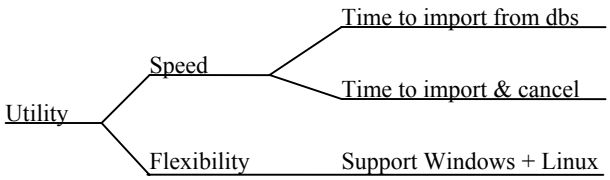
The final goal of ATAM is to identify sensitivity points, tradeoff points, and risks. A sensitivity point is like the fulcrum of a lever: it corresponds to a case in which the feasibility of an implementation attribute strongly depends on an architectural attribute. A tradeoff point occurs when two desirable implementation attributes each depend on the same design attribute, but in conflicting directions. Risks are situations in which there might exist no feasible way to design a system to meet important implementation requirements. In the case of tradeoff points and risks, ATAM suggests that the engineers should further investigate the potential problem, perhaps by applying more quantitative means such as queuing theory [21].

An example should help to demonstrate how ATAM supports evaluating the feasibility predicate ***F*** of our model. Consider again the spreadsheet being produced by a product development firm in our hypothetical "import to spreadsheet" problem (as introduced in Example 3.2.1). The important stakeholders might include liaisons to a user focus group, the marketing department (who have collected statistics on past customer purchases and complaints), and software engineers responsible for understanding the implications of architectural decisions.

After lengthy discussion, the participants may settle on speed and flexibility as being the most important considerations. This allows subsequent discussion to ignore other, less crucial dimensions of $\mathbf{A}^n$, such as delivery date and reusability. Of course, depending on how the cost function is modeled, it may be desirable to retain these other dimensions in $\mathbf{A}^n$; they would simply not be the focus when using ATAM to evaluate the feasibility predicate ***F*** in this example.

In order to operationalize "speed" and "flexibility" into something that supports evaluation, the participants further discuss what these two quality attributes mean in this context. They may decide to focus on two aspects of "speed" and one aspect of "flexibility;" each of these is described with a scenario and represented in a tree, as shown below.[7]

[7] Since utility trees are usually much larger than the example shown here, ATAM suggests associating labels with each scenario leaf to indicate whether the scenario offers High/Medium/Low value, and whether the scenario seems likely to be of High/Medium/Low difficulty to support. If the scenario leaves are labeled with value and estimated difficulty, then the engineers know to focus most on the leaves with high value and/or high difficulty.



After all the stakeholders have created the utility tree, the engineers take time to understand the existing system architecture. In the case of our spreadsheet system, they might meet with programmers of any previous product versions, or they might examine documentation. In this case, the engineers might find that the company already possesses plenty of platform-independent libraries to support cancel operations, and that this will not be a free variable. At the same time, recognizing that existing libraries already cover this issue allows the engineers to focus on other aspects of the problem, such as multi-platform import.

Finally, as the engineers evaluate what changes would be required in order to achieve the specified implementation, they identify sensitivity points, tradeoffs, and risks.

In our example, the engineers would likely note the sensitivity point that supporting import from Microsoft databases like Access will require interfacing with OLE, ODBC, or another Microsoft database access technology. Unfortunately, supporting Linux may hinge in an opposite direction on the same architectural issue, due to limited support for Microsoft database technologies on non-Microsoft platforms, resulting in another sensitivity point as well as a tradeoff. The engineers might even suspect the impossibility of simultaneously achieving both implementation goals, thereby identifying a risk.

Upon identification of tradeoffs and risks, the engineers can do further research to understand which combinations of product attributes are truly infeasible. In the case of the risk mentioned above, further investigation would hopefully reveal an abundance of 3rd party components supporting Microsoft interfaces on the Linux platform, thus alleviating concern about this risk.

Note that in performing this qualitative evaluation of our feasibility predicate $\boldsymbol{F}$: $\mathbf{D} \times \mathbf{A}^n \times \mathbf{M} \rightarrow \{true,false\}$, the implementation method $\boldsymbol{m} \in \mathbf{M}$ also appears as a parameter. If the firm's method is fixed, then this implies an additional constraint on what product attributes can be achieved. For example, if the firm lacks anyone familiar with Linux, then the engineers may immediately flag support for Linux as infeasible.

On the other hand, the implementation method might be a free variable for some firms, depending on their business model. For example, if the firm is amenable to working with open source communities, then the company could deliver a Linux version of the spreadsheet that completely lacks support for importing from Microsoft databases, but instead provide an open API whereby other developers can insert database interface plug-ins. The firm might then actively foster a community of developers who could create plug-ins not only for Microsoft databases, but also for a variety of other databases that would be prohibitively expensive for the firm to directly support.

To summarize, ATAM achieves several goals in a qualitative fashion through a series of meetings between stakeholders representing customer and engineering concerns. First, ATAM yields a utility tree outlining the most crucial implementation attributes. Second, it gives the engineers a chance to understand the existing architecture's strengths and weaknesses. Finally, the engineers combine these disparate issues to identify sensitivity points, tradeoffs, and risks. The result is a qualitative feasibility predicate $\boldsymbol{F}$, which provides guidance not only for further investigation, but also guidance for discussions aimed at resolving infeasible requirements.

3.8 Uncertainty in time

Cost-benefit decisions must often consider incurring costs now in anticipation of benefits later. These decisions often penalize the future benefits, for example by considering interest rates for money or other forms of the time value of resources.

Adding these considerations to the PAD model does not require a change to the signature, but it does increase the sophistication of the benefit $\boldsymbol{B}$ and cost $\boldsymbol{C}$ functions.

$\boldsymbol{U}(\boldsymbol{d};\boldsymbol{\theta}) = \boldsymbol{B}(\boldsymbol{x},\boldsymbol{\theta}) - \boldsymbol{C}(\boldsymbol{d},\boldsymbol{x},\boldsymbol{m})$ for $\{\boldsymbol{x}: \boldsymbol{F}(\boldsymbol{d},\boldsymbol{x},\boldsymbol{m})\}$, where $\boldsymbol{x} = \boldsymbol{P}(\boldsymbol{d},\boldsymbol{m})$

$\boldsymbol{d}$ in $\mathbf{D}$ (expressions in a design notation)

$\boldsymbol{\theta}$ in Θ (expressions of utility)

$\boldsymbol{v}$ in $\mathbf{V}^n$ (arbitrary values)

$\boldsymbol{m}$ in $\mathbf{M}$ (characteristics of design method)

$\boldsymbol{x}$ in $\mathbf{A}^n$ (arbitrary values)

$\boldsymbol{U}$: $\mathbf{D} \times \Theta \rightarrow \mathbf{V}^n$

$\boldsymbol{B}$: $\mathbf{A}^n \times \Theta \rightarrow \mathbf{V}^n$

$\boldsymbol{C}$: $\mathbf{D} \times \mathbf{A}^n \times \mathbf{M} \rightarrow \mathbf{V}^n$

$\boldsymbol{P}$: $\mathbf{D} \times \mathbf{M} \rightarrow \mathbf{A}^n$

$\boldsymbol{F}$: $\mathbf{D} \times \mathbf{A}^n \times \mathbf{M} \rightarrow \{true,false\}$

Because we can choose B and C freely, the PAD model allows for the level of analysis appropriate to each problem.

For example, we can capture the behavior of cost over time by incorporating present-value terms and expected schedules in both P and C.

[[[*Editorial note:*Here there will be an example of a buy vs license decision based on NPV of the purchase and license costs and of the expected benefits, together with

Erdogmus' model of economic value on software development [24]]]

3.9 Uncertainty in incidence (risk)

Although the change of value in resources over time is the most common kind of uncertainty to consider in estimating the value of a product, real projects must take other risks or uncertainties into account. Baldwin, for example, argues that software development inevitably creates options (in the financial sense) [6], and Sullivan [[need citation]] has explored the use of real options to guide software development decisions such as reuse..

$\boldsymbol{U}(\boldsymbol{d};\boldsymbol{\theta}) = \boldsymbol{B}(\boldsymbol{x},\boldsymbol{\theta})$ - $\boldsymbol{C}(\boldsymbol{d},\boldsymbol{x},\boldsymbol{m})$ for $\{\boldsymbol{x}: \boldsymbol{F}(\boldsymbol{d},\boldsymbol{x},\boldsymbol{m})\}$, where $\boldsymbol{x} = \boldsymbol{P}(\boldsymbol{d},\boldsymbol{m})$

$\boldsymbol{d}$ in $\mathbf{D}$ (expressions in a design notation)

$\boldsymbol{\theta}$ in Θ (expressions of utility)

$\boldsymbol{v}$ in $\mathbf{V}^n$ (arbitrary values)

$\boldsymbol{m}$ in $\mathbf{M}$ (characteristics of design method)

$\boldsymbol{x}$ in $\mathbf{A}^n$ (arbitrary values)

$\boldsymbol{U}$: $\mathbf{D} \times \Theta \rightarrow \mathbf{V}^n$

$\boldsymbol{B}$: $\mathbf{A}^n \times \Theta \rightarrow \mathbf{V}^n$

$\boldsymbol{C}$: $\mathbf{D} \times \mathbf{A}^n \times \mathbf{M} \rightarrow \mathbf{V}^n$

$\boldsymbol{P}$: $\mathbf{D} \times \mathbf{M} \rightarrow \mathbf{A}^n$

$\boldsymbol{F}$: $\mathbf{D} \times \mathbf{A}^n \times \mathbf{M} \rightarrow \{true,false\}$

We recognize that P yields a prediction, not a guarantee. At this point, we're lucky to get a prediction of x. Indeed, an important aspect of any predictive model is the confidence one has in its predictions, or put differently, how much the actual outcomes differ from the predicted ones. If we could be more precise about the uncertainties, P could yield a probability distribution over a multidimensional space. In principle, this can be accommodated simply by redefining x to be an element of the space of probability distributions defined over the space of properties relevant to the user.

Alternatively, if the values that properties take can be represented by real numbers, we can let $\phi(\boldsymbol{z};\ \boldsymbol{x})$ be the probability distribution over realized attributes, $\boldsymbol{z}$, that P yields, where $\boldsymbol{x}$ now represents the parameters of this distribution. For instance, if $\boldsymbol{z}$ is a scalar and $\phi()$ is Gaussian, then $\boldsymbol{x} = \{\mu, \sigma\}$, where μ is the mean and σ is the standard deviation.

Everything else remains unchanged except that $\boldsymbol{B}(\boldsymbol{x},\boldsymbol{\theta})$ is now to be interpreted as the expectation of the true benefit function, $U(z,\theta)$, where the expectation is taken over $\boldsymbol{z}$ with respect to $\phi()$. That is, $B(x,\theta) = \int U(z,\theta)\varphi(z;x)\,dz$.

Obviously, the cost function is to be similarly reinterpreted as the expectation of the underlying "true" cost function.

Although it is easy to imagine quite complex, even intractable, functions, the current state of software quality estimation is so primitive that even scalar predictions are hard to obtain; ranges, confidence intervals, and probabilities are a luxury.

4. Usage Scenarios

Section 3 developed and explained the PAD model for predicting value from design. We turn now from the task of predicting value for a specific design to some scenarios that build on that base by showing how such predictions can be used in practice. We begin with some scenarios about using value predictions to make design decisions. We follow with some examples of how PAD accommodates and encourages the composition of predictors in various ways. We close with scenarios about PAD-based analysis affecting the software development process itself.

4.1 Comparing products

A common case of evaluation in the absence of code is selection among different products. In this case the design is fairly simple: Each design is a product identifier, perhaps with selection of options. The attributes of the product may be known from the product specification or perhaps through benchmarking or other empirical observation. Many people informally – often subjectively – perform an analysis that weighs advantages against disadvantages.

We can make that process more precise. (Of course, if we can't get good data, it may not be more accurate.)

Example 4.1.1 Choosing a design with multidimensional costs and values

To make this concrete, we work through another example: Assume our client needs to buy a widget driver. Widget drivers in general provide features J, K, L, M, and N. The client can buy one based on product description, or she can commission one to be built semi-custom from available parts. Thus the set of design is simply the identification of the alternatives:

D in { design(W1), design(W2), design(W3)}

The client really needs a highly usable version of feature J; she must have some degree of K, she needs L occasionally, and M would be nice but not necessary; N is of no interest.

The alternative designs variously provide the features mentioned above, or perhaps subsets. So we can describe their capabilities with a functionality vector <J, K, L, M, N >, where each value is the quality of the feature on a 10-point scale.

For each alternative the client also has good values for purchase cost, delivery time, and risk:

$\boldsymbol{x}$, the predicted properties of the design, is of type <functionality, purch cost, time to delivery, sched risk>

The value space combines cost and time to delivery in a single acquisition cost charge:

v, the predicted values are of type
<functionality, acq cost, schedule risk>

The client provides a personal utility function consisting of an indication of which features are critical (J, K, and L) together with the relative weights for the significance of the features' ratings:

$\boldsymbol{\theta}$ = <<*true,true,true,false,false*>, <4,3,2,1,0>>

The cost function reports the cost and schedule risk, costing each week over 1 at $100; it is silent on functionality:

C(widget driver design, properties, method) =
C(***d*** ,***x, m***) =
<nil, $\boldsymbol{x}_2$+max(0,100*($\boldsymbol{x}_3$-1)), $\boldsymbol{x}_4$>
or, more informally,
C = <nil, cost+max(0, $100*(wks-1), sched risk>

The benefit function sets a minimum standard of quality, say 4, for the required features and applies the weights for the linear combination of quality points to get a figure of merit for functionality; the benefit function is silent on the benefits of the other properties. Expanded,

B(widget driver design, properties, user preference) =
B(***d*** ,***x,*** $\boldsymbol{\theta}$) =
< ((if $\boldsymbol{\theta}_{1i}$ and ($q(x_{1i}) < 4$)) then 0
else $\sum_i \boldsymbol{\theta}_{2i} * q(x_{1i})$,
where i in [1..5], q(z) in [1..10]),
nil, nil>

or, more informally,
B = <weighted quality if all required functions
are present (otherwise 0), nil, nil>

The client has three alternatives. In the cases of packaged software, the vendors' claims provide the predictions of the properties ***x***; in the case of semi-custom development, the acceptance test in the contract that provides the properties ***x***, but with medium-high risk.

Product W1, packaged software
func1 = <7, 8, 6, 10, 2>
x1 = <*func1*, $1000, 1 week, low>
M1 = buy

Product W2, packaged software
func2 = <10, 10, 0, 10, 9>
x2 = <*func2*, $500, 2 days (or .3 weeks), low>
M2 = buy

Product W3, semi-custom
func3 = <9, 10, 9, 8, 9>
---- tailored to my needs, but N came free
x3 = <*func3*, $2000, 8 weeks, medium-high>
M3 = contract fixed fee for schedule & performance

Now we can evaluate the value function. We don't need to worry about the feasibility of the ***x***'s, assuming we have confidence in the W3 producer.

V(design(W1))
= <28+24+12+10, nil, nil> - <nil, $1000, low>
= <74, -$1000, - low>

V(design(W2))
= <0, nil, nil> - <nil, $500, low>
= <0, -$500, - low>
---- missing feature implies zero value,
---- per the benefit function

V(design(W3))
= <36+30+18+8, nil, nil> -
<nil, $2000+$700, medium-high>
= <92, - $2700, - medium-high>

Comparing the designs, W2 is a non-starter because it's missing a required feature; its benefit is therefore zero and its value is negative. To decide between W1 and W3 the client must reduce multidimensional value to a form that allows comparison [41].

4.2 Comparing designs

PAD offers support for the engineering activity of sketching designs, predicting the values of their implementations, and deciding which are worth pursuing. Example 3.1.1, Example 3.6.1, and Example 4.1.1 selected examples by examining a few alternatives manually. This section presents a technique for optimizing over a family of designs, and Example 4.3.1 shows a way to describe a tradeoff space for design choices.

Let's consider how to use the model to choose a design

The optimal ***d*** is

argmax ***V***(***d; θ***) = ***B***(***x***,***θ***) - ***C***(***d***,***x,m***) for {***x***: ***F***(***d***,***x,m***) }

where ***x*** = ***P***(***d,m***).

Note that the optimal design is a function of ***θ***, the utility characteristics of the user. For a mass market product, ***θ*** would vary across different buyers (or types of buyers). If a single design is to be chosen, then either one has to fix ***θ*** (e.g., at its "typical" value, such as the modal value, if one exists) or, if ***θ*** is a real valued vector, by taking expectations over the ***θ***s for the different buyers targeted. In the latter case, the optimal design is

argmax E_θ ***V***(***d; θ***) = argmax {E_θ***B***(***x***,***θ***) - ***C***(***d***,***x,m***)}
for {***x***: ***F***(***d***,***x,m***) }, where ***x*** = ***P***(***d,m***)

Note that unless there are further restrictions imposed upon the benefit and cost functions, there may be many "optimal" designs. Indeed, unless both ***B()*** and ***C()*** have been reduced to scalars, in general there will be more than one such design ***d.***

Example 4.2.1 Optimizing over a set of configuration alternatives

In this section we sketch the solution to the runtime configuration problem introduced in Example 3.2.3.

Recall that the problem at hand is to select a suite of applications and simultaneously set their operating parameters in a way that best satisfies the preferences of a user in the face of resource scarcity.

User's utility captures affinity for specific applications, independent of the runtime operating parameters, and preferences for the runtime operating parameters.

We use this specialized structure of user's utility to design a efficient heuristic search algorithm. The algorithm works in two phases: (1) generate feasible suites of applications that jointly satisfy user's request and for each suite compute the component of the utility that comes from application affinity, (2) for each application suite, explore the design space (the space of operating parameters) to find the one with the maximum *predicted* utility to the user.

The first phase of the algorithm is an ad-hoc combinatorial exercise of generating all possible suites of applications. Computing the partial utility of a suite of application based on user's affinity for applications and sorting the suites of applications in the order of decreasing utility can be done trivially.

Searching the design space of each suite of applications is potentially an expensive operation. To avoid searching all combinations, we design a simple stop condition: stop exploring the design space of additional application suites, once we know that the maximum attainable utility from the remaining (unsearched) suites can not exceed the predicted utility of one of the suites already explored.

The complete details of the solution are described in [43].

4.3 Exploring tradeoff spaces

It is common in design to encounter situations in which you can increase one quality attribute only by decreasing another (or by increasing cost). PAD should support analytic descriptions of those tradeoffs. We find a quality tradeoff in an unexpected place, within the COCOMO II model.

Example 4.3.1 Evaluating tradeoffs among product attributes

In this example, we return to the discussion of COCOMO II [9] from Example 3.4.3in order to highlight the tradeoffs among product attributes. First, we reorganize the terms in the formula for person-months (PM). Having already split the set *EM* into subsets EM_P and EM_D corresponding to properties of the product and the development process, respectively, we rewrite the COCOMO II equations as

$$PM = (Size^{E} * S_D) * S_P$$
$$S_P = \Pi\ EM_p \text{ where } EM_p \text{ ranges over } EM_P$$
$$S_D = A * \Pi\ EM_d \text{ where } EM_d \text{ ranges over } EM_D$$
$$E = B + 0.01 * \Sigma\ SF_i \text{ where } SF_i \text{ ranges over } SF$$

This reorganizes the model into the form

$$PM = f(\boldsymbol{m}) * \Pi\ EM_p$$

thereby giving a relation among the five COCOMO-related elements of $\mathbf{A}^n$. For example, if we assume a 100KSLOC system size and nominal values for the elements of $\boldsymbol{m} \in \mathbf{M}$ (A, B, and the elements of EM_D and *SF*), this becomes

$$PM = 2.94 * 100^{1.0997} * \Pi\ EM_p$$
$$= 465.3153\ \Pi\ EM_p$$

This is too complex for simple visualization. However, if we fix RUSE and PDIF at their nominal values, we can display the tradeoff space between SCHED, RCPX, and PM as in Figure 1. This figure depicts the fact that increasing the implementation's quality attributes also raises the product's person-months required. Selecting an optimal level of each quality attribute requires an examination of the cost and benefit functions, as discussed earlier in Section 3.5.

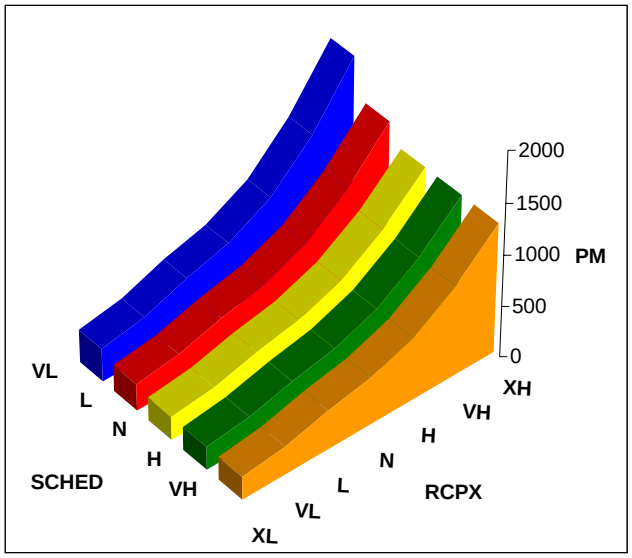


Figure 1. Tradeoffs among SCHED, RCPX, and PM

4.4 Composing evaluation functions

The examples of Section 3 show a great deal of variety among the representations of designs used by predictor functions. The attributes predicted by those functions are similarly varied. Section 6 tabulates these for easy comparison.

We are deliberately liberal about the inputs and outputs of PAD model elements because we believe that PAD should accommodate a wide variety of predictors, and we have no control over what they choose as parameters (nor should we).

There will be cases in which the result type of one predictor will match the input type of another predictor. PAD is deliberately intended to expose opportunities to compose predictors and to model that composition.

Section 3 provided an example of precisely this type of composition. Example 3.2.1 estimated code size based on function points, and Example 3.4.2 used that code size as

its input. This composition is part of the COCOMO II Early Design Model [9].

4.5 Jointly using separate predictors

As an alternative composition strategy, rather than composing predictors to yield *more* product dimensions as in previous examples, we can compose to yield *richer* product dimensions. This alternative strategy becomes attractive any time that the available predictors map into a product attribute space with impoverished dimensions. In such cases, composing multiple predictors facilitates mapping into a space that is less quantized and therefore more effective at highlighting utility differences among designs.

For instance, in Example 4.3.1, we introduce a line of architecture research that, at best, could have mapped designs into a product attribute space represented by a vector of Booleans. By composing this predictor with another technique, we are able to map into a fuller space represented by a real number in each dimension.

One strength of our model is that it allows for multiple composition strategies, and indeed that it allows for composition at all. This opportunity to compose strands of research allows the creation of spaces and predictors that capture more of a design problem's texture, thereby facilitating better decisions and happier customers.

Example 4.5.1 Predicting usability

As researchers have begun exploring relations between design features and software quality attributes, the impact of design decisions on an implementation's usability has become particularly well understood. Below, we weave together two strands of usability research to provide a richer method predictor ***P*** than either strand could provide alone. The first strand concerns user interface design, while the second involves architectural design.

First, Bass et al have identified 26 ways in which architectural design affects software usability [7]. For example, the ability to "cancel" computations is crucial to usability in some contexts, but it requires a pre-emptive multi-threaded architecture (with one thread responding to user inputs while another performs long-running but cancelable computations), plus logging in order to roll back canceled computations.

Second, the GOMS family of user interface evaluation techniques [30] expresses usability in terms of how quickly a user can complete tasks. Specifically, applying GOMS to predict usability requires specifying a set of user goals (with sub-goals) or use cases, estimating the time to execute relevant operations supported by the software design, and specifying "selection rules" that indicate which operations the user will choose in order to achieve each sub-goal. The total time required to achieve a goal thus approximately equals the total time required to perform the list of operations for the relevant sub-goals.

To demonstrate how these two strands together yield a richer method predictor ***P*** than either strand could alone, let us return once more to the "import to spreadsheet" problem originally introduced in Example 3.2.1. Our challenge is to design a spreadsheet supporting two use cases related to importing data from Microsoft Access. These use cases together define a simplified product attribute space $\mathbf{A}^n$ represented with tuples of the form

< Time to import 100 rows from Access,
Time to begin an import then cancel >

We have been evaluating three designs,

A design with all four features,	< 1 1 1 1 >
A design with only database connectors,	< 1 0 0 0 >
A design with only copy/paste,	< 0 1 0 0 >

which exist in a design space **D** represented as

< Does the design include connectors to databases? ,
Does it call for integrating with OS clipboard? ,
Does it include tracking and undoing changes? ,
Does it specify running tasks in their own threads?
>

To predict what implementation attributes $\boldsymbol{x} \in \mathbf{A}^n$ would likely result from each design $<d_1, d_2, d_3, d_4> \in \mathbf{D}$, we incorporate architectural considerations and a simple GOMS model of how users utilize features to attain goals.

Consider the first use case, importing from the database. Our GOMS model might specify that importing data takes 4 seconds for one hundred rows if the design includes database connectors, since the process would be largely automated. If the application lacked this feature but did support copy/paste, our GOMS model might specify that the user will copy/paste rows one at a time from Access, at a cost of 3 seconds per row.

Estimating the time to achieve the second use case, import-and-cancel, involves an architectural wrinkle. As noted earlier, canceling requires a multi-threaded architecture and a log. If the design lacks both features, the computation must proceed to its end, and then the user must reverse the computation by manually deleting each row; if the design includes logging and undo but not multi-threading, the computation must proceed to its end, and then the user can undo in one step. Our GOMS model might estimate that manually deleting each row costs 1 second, and that canceling occurs halfway through import (so multi-threading saves half of the automated import time, or about 2 seconds).

Under this composite model, leaving aside the feasibility predicate ***F*** and implementation method ***m***, the first design can be expected to cost 4 seconds to support simple import and 2 seconds to support import-and-cancel. This yields the tuple $\boldsymbol{x} = <4s, 2s> \in \mathbf{A}^n$. In contrast, the second design

yields $\boldsymbol{x}$ = <4s, 104s> (since canceling requires 4 seconds to import all the rows, plus 100 seconds to manually undo each edit). Finally, the third design yields $\boldsymbol{x}$ = <300s, 200s> (since canceling requires 150 seconds to copy and paste half of the rows, plus another 50 seconds to manually delete them).

Now, neither the GOMS strand of research alone nor the usability-architecture strand of research alone would have been able to make all of these predictions.

Consider using GOMS alone. Predicting the first product attribute would have been straightforward, since our simple GOMS model essentially predicts a cost of 4 seconds for any implementation with an automatic database import, and 300 seconds for any implementation that only supports copy/paste.

However, the GOMS model would stumble upon trying to predict the product attribute corresponding to the time to import and cancel halfway through. In order to make this prediction, we had to rely on the usability-architecture strand of research, which highlighted the necessity of a multi-threaded architecture to support immediate cancellation of a task. Without this insight, using GOMS alone, we would be left to simply assume whether imports could be immediately cancelled, regardless of the threading architecture; in fact, we might have been led to omit the threading model entirely from our design space model **D**. This could have led to selecting a design with a poor threading model, resulting in disappointed customers.

Conversely, the usability-architecture strand of research alone could not have yielded such a rich method predictor $\boldsymbol{P}$, either. This research remains in its qualitative stage, which means that the best we could have done without GOMS is to model our product attribute space as a vector of Booleans:

< Can automatically import from Access,
Can import and preemptively cancel halfway >

Our three designs would have yielded the implementations <true true>, <true false>, and <false false>, respectively. Obviously, this loss of richness reduces our ability to reason about tradeoffs among implementations. For example, the CBAM benefit evaluation discussed in Example 3.5.1 would be much less precise, since implementations could only be scored on the basis of *whether* they support the use case, rather than *how well* they support it. The loss of information could potentially result in indistinguishable benefit between two implementations, thereby making it difficult to decide among the options.

To conclude, combining these two strands of usability research yielded a method predictor that maps into a product attribute space with richer dimensions than either strand alone could have achieved.

4.6 Finding predictors of attributes of interest

[[*Editorial note:* When the comparative table of Section 6 is finished, it will support a kind of usage in which the designer identifies an attribute of interest and seeks a way to predict it from available information. The comparative table should support dependency analysis to show compositions of functions that can predict the attribute of interest. The designer could then choose a path that uses available information or, if information is not currently available, decide whether the value of the prediction justifies the cost of acquiring new data.]]

4.7 Deciding what design information to capture

[[Editorial note: When the comparative table of Section 6 is finished, it should support analysis of design information. The inputs of the predictors are designs in various notations. Examining the table will show what design information is required to enable prediction of various attributes. This should provide guidance about the types of design information it is worth capturing.]]

5. Open Questions

This is a preliminary report on an initial PAD framework. Development of this framework continues. We wrap up the discussion in this preliminary report with some discussion of problems we have not yet addressed and of the ways we are gathering evidence to support the validity and usefulness of the PAD model.

5.1 Is it sound?
No, it's light

[[[Q: what if one of the predictors in the inventory is wrong? A: You get the wrong answer. Our obligation is to provide a framework that lets you take the best advantage of the analysis tools at your disposal. Different people are more or less willing to use approximate or unsound analyses. Indeed, we can accept expert subjective judgment as a predictor for some element if nothing else is available. Open question: How to attach metainformation such as credibility or source of the information to an analysis]]]

5.2 Is the model correct?
Maybe not, it's a first cut

[[[We think this is close, but we know of some rough spots. For example, why doesn't C take theta as a parameter? Should development costs be treated as properties of the implementation? Is it problematic that the same types can appear as design elements and as attributes?]]]

5.3 Is it complete?
No, it's opportunistic

[[[Q: What I want to predict isn't covered. A: <get this from Ockerbloom's response about TOM, basically we don't make the problem worse and might make it better by finding a chain.

Besides, we need to learn to work with partial information and know what limitations come with that partial information. Bring in some of the arguments from the Credentials paper; think about the use of metainformation.]]]

5.4 Is it universal?

No, it takes a user view of value

[[[Q: I have this model. It doesn't fit. A: OK]]]

Aren't there other stakeholders? Yes

[[[Q: The user isn't the only stakeholder. A: Right. And we hope that Sullivan and co are working on a compatible complementary model for another stakeholder]]]

[[[Q: What about other stakeholders? They need value functions, too. A: Value is in the mind of a beholder. We have focused on value to the client. We hope that others will develop value functions that reflect the objectives of other stakeholders, especially producers.]]]

5.5 Does it work?

Maybe

[[[Q: what makes you think this works? A: This is a preliminary report. We have not done field studies on the framework. However, many of the examples are validated, and we can reason about how we preserve their value and about how we can provide better access to their value]]]

[[[Q: Isn't there a lot of mechanism? A: Not really We defined a framework that allows us to explain the contributions of a wide variety of models and techniques for predicting value. The framework shows how they can interact. The mechanism you see is really the mechanism of the diverse models, and if you want those results, you have to do the work.]]]

5.6 So, is it useful?

We think it can be

[[[There's lots to validate, but even in this early state we see early promise: By restating the models in this framework we show better what they actually do, make it easier to compare them. By collecting a set, we provide an organized collection of tools.]]]

Theories come in many forms; generally more compact theories are valued more highly. Historically, though, theories in software engineering have often started as qualitative descriptions and gained power as they matured.

[[[For a formalism to be truly useful, of course, it should support an associated calculus that allows us to draw formal conclusions *within* the formalism.]]]

We propose this unification as a step in that maturity. Our proposed unification will be valuable to the extent that it helps to improve the design decisions that software engineers make.

We have showed it range of expressiveness by exploring the fit of existing models to the model – both models intended for this purpose and some models principally intended for other purposes. In the process, we have noted some of the strengths and limitations of these models for early predictive design analysis.

In recasting these models, we have made it easier to compare them: their applicability, their precision, their assumptions. We believe (and will test this belief in the fall semester) that this uniform presentation will help students and designers to understand the models and select appropriate ones for their tasks.

More broadly, we hope that this explicit formulation of the value proposition will

- draw more attention to cost and other value attributes during early design
- encourage systematic extension of current models to handle a larger part of the value space
- stimulate work that will improve the operability of existing models,

and ultimately lead to a systematic unified theory to support early predictive design for software.

5.7 What does it not do?

Things that need code

[[[There are several types of analysis for which PAD is not intended and may provide little help. These are chiefly analyses that require access to actual code..

Ex 1: traditional verification of consistency between code and specifications – requires code

Ex 2: static analysis – similarly requires code.

Ex 3: algorithmic complexity – fits, with cost in big-O notation; also allows us to point out that implementations sometimes get it wrong]]]

6. Catalog of Published Predictors

[[In preparation. The first version will be a side-by-side comparison of the predictors used in the examples and discussion of this report, specifically identifying their inputs, outputs, and their fit within the PAD model.]]

7. Acknowledgements

Thanks to Carliss Baldwin, Frank Padberg, <others?> for constructive discussions on various drafts.

This work has been funded in part by the EUSES Consortium via the National Science Foundation (ITR-0325273), by the National Science Foundation under Grant CCF-0438929, by the Sloan Software Industry Center at Carnegie Mellon, and by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298. Any opinions, findings, and conclusions or recommendations expressed in this material

are those of the authors and do not necessarily reflect the views of the sponsors

[[[other authors, add your funding acknowledgements]]]

8. References

[[[There are lots of extra references here. I brought over all the citations from the proposal in hopes of preserving the cross-references. We need to prune out the ones we are not using.]]]

[1] Gene M. Amdahl, Gerrit A. Blaauw and Frederick P. Brooks, Jr. "Architecture of IBM System/360," *IBM Journal of Research and Development*, 8 (2), 1964: pp. 87-101.

[2] A. Arora, M. Ceccagnoli, and W. Cohen. "R&D and the patent premium", 2003, National Bureau of Economic Research Working Paper 9431, under review at *The RAND Journal of Economics*.

[3] A. Arora and A. Fosfuri. "Pricing Information: The market for (diagnostic) information tools", 2002, revision requested by *Management Science.*

[4] C. Gordon Bell and Allen Newell. *Computer Structures: Readings and Examples*. McGraw Hill 1971

[5] Carliss Y. Baldwin and Kim B. Clark. *Design Rules, vol 1 The Power of Modularity*. MIT Press, 2000.

[6] Carliss Y. Baldwin and Kim B. Clark. *The Fundamental Theorem of Design Economics*. Working paper, Harvard Business School. http://www.peoole.hbs.edu/cbaldwin/DR2/BCFundTheorDesignv4.pdf

[7] L. Bass, B. John, and J. Kates. Achieving Usability Through Software Architecture. *Technical Report CMU/SEI-2001-TR-005, Carnegie Mellon Software Engineering Institute*, Pittsburgh, PA, Mar 2001.

[8] Gerrit A. Blaauw and Frederick P. Brooks. *Computer Architecture: Concepts and Evolution*. Addison-Wesley 1997.

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

[10] Barry W. Boehm and Victor R. Basili. "Software defect reduction top 10 list," *IEEE Computer,* January 2001, pp. 2-4.

[11] E. Brynjolfsson and C. Kemerer. Network Externalities in Microcomputer Software: An Econometric Analysis of the Spreadsheet Market. *Management Science*, *42,* 12 (Dec 1996), 1627-1647.

[12] Shawn A. Butler. Security Attribute Evaluation Method: A Cost Benefit Analysis Method. 23rd International Conference on Software Engineering, 2001 [[[is this correct?]]]

[13] Shawn A. Butler. "Security design: why it's hard to do empirical research". *Workshop on Using Multidisciplinary Approaches in Empirical Software Engineering Research*, affiliated with the 22nd International Conference on Software Engineering (ICSE 2000).

[14] Shawn Butler, "Improving security technology selections with decision theory," *Third Workshop on Economics-Driven Software Research (EDSER-5)*, affiliated with the 23rd International Conference on Software Engineering, 2001.

[15] Shawn Butler, "Security attribute evaluation method: A cost-benefit approach," Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), pp. 232-240.

[16] Shawn A. Butler. *Security Attribute Evaluation Method*. Carnegie Mellon University School of Computer Science Technical Report CMU-CS-03-132, May 2003. PhD Thesis. http://shaw-weil.com/marian/DisplayPaper.asp?paper_id=78

[17] Shaw. Butler and Paul Fischbeck, "Multi-attribute risk assessment," Symposium on Requirements Engineering for Information Security, October 2002.

[18] Shawn A. Butler, Somesh Jha, and Mary Shaw. When good models meet bad data: Applying quantitative economic models to qualitative engineering judgments, *Proceedings of the Second Workshop on Economics Driven Software Engineering Research*, IEEE Computer Society, 2000.

[19] Shawn Butler and Mary Shaw, "Incorporating nontechnical attributes in multi-attribute analysis for security," *Fourth Workshop on Economics-Driven Software Engineering Research (EDSER-4)*, affiliated with the 24th International Conference on Software Engineering (ICSE'02).

[20] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley 2001.

[21] Robert Cooper. *Introduction to Queuing Theory*, North Holland Publishers, Second Edition, 1981.

[22] CourseForges Alliance. *CourseForges Wiki* for collaborative curriculum development in value-based software decision making. http://seg.iit.nrc.ca/yawc/courseforges/public/wiki.cgi

[23] EDSER: The International Workshop on Economics-Driven Software Engineering Research (affiliated each year with the International Conference on Software Engineering). See http://www.edser.org.

[24] Hakan Erdogmus. Comparative evaluation of software development strategies based on Net Present Value. *Position paper for the First Workshop on Economics-Driven Software Engineering Research (EDSER-1),* May 1999

[25] David Garlan, Robert Monroe, and Dave Wile. Acme: An Architecture Description Interchange Language. *Proceedings of CASCON 97*, Toronto, Ontario, November 1997, pp. 169-183.

[26] David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste, "Project Aura: Towards distraction-free pervasive computing," *IEEE Pervasive Computing*, special issue on "Integrated Pervasive Computing Environments", Volume 21, Number 2, April-June, 2002.

[27] Sourav Ghosh, et al. "Scalable Resource Allocation for Multi-Processor QoS Optimization". *Proc. 23rd International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[28] J.D. Herbsleb and A. Mockus. “Formulation and preliminary test of an empirical theory of coordination in software engineering”. *Proc Joint ESEC/ ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003. pp. 112-121.

[29] Michael Jackson. *Problem Frames*. Addison-Wesley 2001.

[30] B. John and D. Kieras. The GOMS Family of User Interface Analysis Techniques: Comparison and Contrast. *ACM Transactions on Computer-Human Interaction, 3*, 4 (Dec 1996), 320-351.

[31] P. Layard and A. Walters. *Microeconomic Theory*, McGraw-Hill, New York, NY, 1978

[32] R. Kazman, M. Klein, and P. Clements. ATAM: Method for Architecture Evaluation, Technical Report ESC/CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Aug. 2000.

[33] Risk Kazman, Jai Asundi, Mark Klein. Quantifying the Costs and Benefits of Architectural Decisions. 22nd International Conference on software Engineering, 2000, IEEE Computer Society.

[34] R. Kazman, J. Asundi, and M. Klein. Making Architecture Design Decisions: An Economic Approach. Technical Report CMU/SEI-2002-TR-035, ESC-TR-2002-035, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September 2002.

[35] R.L. Keeney and H. Raiffa, Decisions with Multiple Objectives, Cambridge University Press, 1999

[36] Chen Lee, et al. A Scalable Solution to the Multi-Resource QoS Problem. Proc IEEE Real-Time Systems Symposium (RTSS), 1999.

[37] Paul Luo Li, Mary Shaw, and James D. Herbsleb. “Selecting a defect prediction model for maintenance resource planning and software insurance”. Fifth Workshop on Economics-Driven Software Research (EDSER-5), affiliated with the 25th International Conference on Software Engineering, 2003, IEEE Computer Society, pp. 32-37.

[38] Paul Luo Li, Mary Shaw, James D. Herbsleb, Bonnie Ray, and P. Santhanam. “Empirical evaluation of defect projection models for widely-deployed production software systems”. Submitted to *ACM SIGSOFT 2004* / FSE-12.

[39] Paul Luo Li, Mary Shaw, Kevin Stolarick, and Kurt Wallnau, “The potential for synergy between certification and insurance,” Special edition of *ACM SIGSOFT* from the International Workshop on Reuse Economics in conjunction with the Seventh International Conference on Software Reuse (ICSR7), April 2002.

[40] D. Narayanan, J. Flinn, M. Satyanarayanan. Using History to Improve Mobile Application Adaptation. *Proc. 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, 2000.

[41] Vahe Poladian, Shawn A. Butler, Mary Shaw, and David Garlan. “Time is not money: The case for multi-dimensional accounting in value-based software engineering.” *Fifth Workshop on Economics-Driven Software Research* (EDSER-5), affiliated with the 25th International Conference on Software Engineering, 2003, IEEE Computer Society, pp. 19-24.

[42] Vahe Poladian, David Garlan, and Mary Shaw, “Software selection and configuration in mobile environments: A utility-based approach,” *Fourth Workshop on Economics-Driven Software Engineering Research* (EDSER-4), affiliated with the 24th International Conference on Software Engineering (ICSE'02), May 2002.

[43] Vahe Poladian, Joao Sousa, David Garlan, and Mary Shaw, “Dynamic configuration of resource-aware services,” *Twenty Sixth International Conference on Software Engineering*, (ICSE-2004), May 2004.

[44] Mary Shaw (ed). *Alphard: Form and Content*. Springer-Verlag, 1981, 321 pp. (Annotated Alphard papers.)

[45] Mary Shaw. Truth vs knowledge: The difference between what a component does and what we know it does, *Proceedings of the 8th International Workshop on Software Specification and Design*, 1996, pp. 181-185.

[46] Mary Shaw. "What makes good research in software engineering?" Invited keynote, European Joint Conference on the Theory and Practice of Software, ETAPS-2002. Appears in Opinion corner, *International Journal on Software Tools for Technology Transfer*, vol 4, DOI 10.1007/s10009-002-0083-4, June 2002.

[47] Mary Shaw, “Everyday dependability for everyday needs,” *Supplemental Proceedings of the 13th International Symposium on Software Reliability Engineering*, Annapolis, MD, November, 2002, pp. 7-11.

[48] Mary Shaw, “'Self-healing': Softening precision to avoid brittleness,” *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02)* Charleston, South Carolina, November 2002, pp. 111-113.

[49] Mary Shaw. “Writing good software engineering research papers.” Mini-tutorial, *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, IEEE Computer Society, 2003, pp. 726-736.

[50] Mary Shaw, Shawn A. Butler, Hakan Erdogmus, and Klaus Schmid. “CourseForges: Open source curriculum design for value-based software engineering. *Fifth Workshop on Economics-Driven Software Research (EDSER-5)*, affiliated with the 25th International Conference on Software Engineering, 2003, IEEE Computer Society, pp. 4-7.

[51] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[52] Mary Shaw and Jim Herbsleb. *Methods: Deciding What to Design*. Graduate course, Carnegie Mellon University. Fall 2003 offering at http://spoke.compose.cs.cmu.edu/methods-fall03/

[53] Herbert A. Simon. *The Sciences of the Artificial*. Third edition, MIT Press, 1996.

[54] João Pedro Sousa and, David Garlan. *The Aura Architecture: an Infrastructure for Ubiquitous Computing*. Technical Report, CMU-CS-03-183, School of Computer Science, Carnegie Mellon University, 2003.

[55] Kevin Sullivan (ed). *NSF Workshop on the Science of Design: Software and Software-Intensive Systems*. Preliminary report, February 2003. Online at NSF.

[56] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. "The structure and value of modularity in software design." *Proc Joint ESEC/ ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2001.

[57] R. Swati, A. Firauf, and R. Santosh. Architecture Simulator High Level Design Document. Carnegie Mellon West working paper, http://www-2.cs.cmu.edu/~shawnb/ARCH_AAD_ELA_HLDD_1.1_SR.doc

[58] J Walsh, A. Arora, W. Cohen, "Working through the patent problem" 2003, *Science,* 299(5609): 1021-1021, Feb 14.

[59] J Walsh, A. Arora, W. Cohen. "Research tool licensing and patenting and biomedical innovation", 2003, in Cohen and Merril (eds), *Patents in the Knowledge Based Economy*, NAS Press, Washington DC, *forthcoming*.

[60] William. A. Wulf, Mary Shaw, Paul N. Hilfinger, and Lawrence Flon. *Fundamental Structures of Computer Science*. Addison-Wesley, 1981, 621 pp. (Also, Instructor's Manual, with Deborah Baker and Dale Moore, 295 pages.) Republished in Addison-Wesley's World Student Series, 1982.

[61] Wm. A. Wulf, Ralph L. London, and Mary Shaw. "An Introduction to the construction and verification of Alphard programs." *IEEE Tr. Software Engineering*, SE-2, 4 (December 1976), pp.253-265. Presented at Second Int'l Conference on Software Engineering, October 1976. Original report reprinted in *Alphard: Form and Content* (Mary Shaw, ed), Springer-Verlag 1981. Named "most influential paper from ICSE-2" at Thirteenth Int'l Conference on Software Engineering, May 1991.

[62] Stress analysis

[63] Economic model citation from Ashish

[64] PA simple bridge template

9. Appendix A: Time is Not Money

This will be the text of the position paper, "Time is Not Money" by Vahe Poladian, Shawn A. Butler, Mary Shaw, and David Garlan [41]

10. Appendix B: Reconciling Quantitative and Qualitative Models

This will be text of the position paper, "When good models meet bad data: Applying quantitative economic models to qualitative engineering judgments" by Shawn A. Butler, Somesh Jha, and Mary Shaw [18].

11. Appendix C: Credentials

This will be text of the paper, "Truth vs knowledge: The difference between what a component does and what we know it does" [45].