

A Value-Based Approach to Predicting System Properties from Design

Chris Scaffidi

School of Computer Science
Carnegie Mellon University
+1-412-268-3564
www.cs.cmu.edu/~cscscaffid
cscscaffid+isri@cs.cmu.edu

Ashish Arora

Sloan Software Industry Center &
H John Heinz III School of Public
Policy and Management
Carnegie Mellon University
+1-412-268-2191
www.softwarecenter.cmu.edu/Arora.htm
ashish@andrew.cmu.edu

Shawn Butler

School of Computer Science
Carnegie Mellon University
+1-703-628-2195
www.cs.cmu.edu/~shawnb
shawn.butler@cs.cmu.edu

Mary Shaw

Sloan Software Industry Center &
School of Computer Science
Carnegie Mellon University
+1-412-268-2589
www.cs.cmu.edu/~shaw
mary.shaw@cs.cmu.edu

Abstract

Traditional engineering requires evaluating designs before implementing them. Evaluating a design predicts the properties of a reasonable implementation and the value of these properties to a stakeholder. Software engineering has some (though not enough) relevant evaluation techniques but lacks frameworks to compare, develop, and apply those techniques in a manner that respects how value varies by stakeholder. We present an adaptation of economists' value models that, given a design and a development method, predicts value to a client. We give examples supporting our approach. Even in its preliminary state, our approach helps to explain and characterize design evaluation techniques and shows sufficient promise to justify further development.

Categories & Subject Descriptors:

D.2.m [Software]: Software Engineering – *miscellaneous*;

K.6.0 [Management of Computing and Information Systems]:

General - *economics*

General Terms: Design, Economics

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

1. Background: the view from economics

Traditional engineering discipline calls for early evaluation of designs; similarly, software engineering recognizes that problems cost substantially more to find and fix after implementation begins than during requirements and design [2]. However, software engineering lacks a systematic approach to early evaluation. Here we describe an approach to early evaluation derived from the classic economic model for value.

Much 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

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDSER'05, May 15, 2005, St. Louis, Missouri, USA.
Copyright 2005 ACM 1-59593-118-X/05/0005...\$5.00.

Here, the vector z represents the quantity sold of each product, and $B(z)$ is the total revenue from selling those products. The vector y represents the quantity of each input used, and $C(y)$ is the total cost of those inputs. $F(y, z)$ is a vector, as well, so $F(y, z) \leq \theta$ represents a list of equations representing constraints [6].

2. An economic approach to predictive design evaluation for software

Bearing in mind the special characteristics of software, we are exploring an approach to design evaluation based on the economic model [9]. We must adapt this model in certain respects, including in particular that the values of attributes are not drawn from the continuous numbers and that the model must reflect the uncertainties that lie between the given design and an implementation whose properties underlie the value. The setting we have in mind is that

- a design d and a description of one or more clients' preferences θ are available, both possibly incomplete;
- we have techniques to predict some aspects of the value v to the client that will result from implementing the design d ;
- actual code is not available for analysis, either because code is unwritten or unavailable, or because analysis is intractable;
- the design d will be implemented with method m and yield an implementation with properties x .

Naturally, the predictions are not guarantees; they therefore have associated uncertainties. Further, the value itself will generally have several components [7]. Specifically, let

- d be the design or partial design being evaluated, expressed in some appropriate notation¹ \mathbf{D}
- m be a development method for implementing a design, expressed in some appropriate notation \mathbf{M}
- x be in \mathbf{A}^n , a multidimensional space whose dimensions correspond to attributes of interest to the client. These dimensions form a subset of all (eventually) observable artifact properties. In the simplest case, the attributes' values are rational numbers and \mathbf{A}^n is \mathbf{R}^n , but more generally the values could be of other types, such as ranges, probability density functions, or other expressions of uncertainty
- θ 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

¹ For the fastidious, \mathbf{D} is the set of legal expressions in the design notation, whatever that notation may be. Similarly, Θ and \mathbf{M} are sets of legal expressions in whatever notations are appropriate.

may prefer speedier implementations and/or be risk averse). θ is the only representation of these preferences, so two clients with the same θ are indistinguishable in this model.

- v be in V^n , a multidimensional value space; in the simplest case, values are scalar dollars but more generally v may distinguish different classes of costs and values².

Estimating the value of a design d requires comparing benefits and costs of the implementation properties predicted to result from d , provided of course, the design is feasible. This resembles many interesting economics problems cast as maximizing a function subject to constraints—for instance, maximizing utility subject to budget constraints, or minimizing the cost of production given current technology and input prices.

In adapting the model used by economists to apply to software design evaluation, we preserve the form of the net utility 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 relate the design and its implementation with some software development method. We allow multidimensional values because software design makes demands on resources that may be incommensurable or valued differently by different clients [7]. As a result, no unique optimal design may exist, but it is useful to capture aspects of interest to a software architect / designer.

The 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 also even for smaller-scale abstractions. For example, the algorithmic complexity of an algorithm predicts a property of a correct implementation of the algorithm, namely the rate at which execution time grows as a function of problem size. Selecting an $O(n \log n)$ algorithm over an $O(n^2)$ algorithm offers reasonable assurance, but not an absolute guarantee, that the algorithm will be correctly implemented to achieve the $O(n \log n)$ scalability.

Specifically, let

- $U: D \times \Theta \rightarrow V^n$, the value function, be a function composing benefits and costs to predict overall value of a design d to a client with preferences θ ; for simplicity we take it to be vector subtraction in V^n (the domain of U must include the domains of B and C)
- $B: A^n \times \Theta \rightarrow V^n$, the benefit function, be the predicted value of properties x with respect to the client's preferences θ , possibly re-evaluated dynamically; B captures the client's concerns and hence does not depend on the design or the method of implementing the design
- $C: D \times A^n \times M \rightarrow V^n$, the cost function, be the predicted cost resulting from implementing design d to achieve properties x using development method m ; 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 detailed terms of the proposed contract

plus any other costs the client will incur in deploying and maintaining the software.

- $P: D \times M \rightarrow A^n$, the method predictor, be the properties x to expect from implementing design d with method m
- $F: D \times A^n \times M \rightarrow \{\text{true}, \text{false}\}$, the feasibility function, 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 space $D \times A^n \times M$ and captures external constraints such as physical and legal limitations

Then the value of a design is

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

This model can accommodate analysis techniques having various degrees of detail and precision. It allows for complex client preferences, permits rich attributes and value terms including confidence indicators or estimates, and accommodates the time value of resources. The model captures the information required to treat determination of software value in economic terms, including cost, uncertainty, and future contingencies.

3. Predicting properties of products

Our approach to reasoning about the value of designs involves the four functions above: the method predictor P , cost function C , benefit function B , and net utility U . In this section, we show how existing techniques can be cast as method predictors $P: D \times M \rightarrow A^n$. We first examine predictors of usability, then of development effort and product attributes in A^n .

3.1 Predictors of 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 an example method predictor P . The first strand concerns user interface design, while the second involves architectural design.

First, the GOMS family of user interface evaluation techniques [5] 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 subgoals) 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 subgoals.

Second, Bass et al have identified 26 ways in which architectural design affects software usability [1]. 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) and logging in order to roll back canceled computations.

To understand how to compose these strands of research into a method predictor P , consider designing spreadsheet editing software. Suppose that the use cases include importing data from Microsoft Access, and that the space of implementation attributes A^n is represented with tuples of the form

² For the fastidious, all the v in a single analysis must represent values of a single stakeholder, typically the client. To do otherwise would confound the client's and the producer's valuations, which would be dangerous. E.g., purchase cost is negative to the client but positive to the producer.

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

The design description \mathbf{D} might involve vectors of four Booleans:

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

Consider evaluating three designs (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 >

To predict what implementation attributes $\mathbf{x} \in \mathbf{A}^n$ would likely result from a given design $\langle d_1, d_2, d_3, d_4 \rangle \in \mathbf{D}$, 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 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 $\mathbf{x} = \langle 4s, 2s \rangle \in \mathbf{A}^n$. In contrast, the second design yields $\mathbf{x} = \langle 4s, 104s \rangle$ (since canceling requires 4 seconds to import all the rows, plus 100 seconds to manually undo each edit). Finally, the third design yields $\mathbf{x} = \langle 300s, 200s \rangle$ (since canceling requires 150 seconds to copy and paste half of the rows, plus another 50 seconds to manually delete them).

This example shows not only how to cast those research results relating design and usability as method predictor P , but also how to compose two distinct models relating design and usability.

3.2 Predictors of development attributes

The COCOMO II Early Design Model offers another predictor, giving estimates of the development effort and development time based on the design [3]. We draw three examples from COCOMO II. In casting COCOMO II as a method predictor, we take the design to represent the design \mathbf{D} , the cost drivers of the Early Design Model to represent the method \mathbf{M} , and we include the time and effort of development and the size of the implementation in the product attributes \mathbf{A}^n . In addition, we show how to treat

several of COCOMO's effort multipliers as product attributes rather than characterizations of the development method. The first example estimates the size of the implementation. The second explores choices of design alternatives. The third reformulates the COCOMO II model slightly to derive tradeoffs among some additional product attributes.

3.2.1 Estimate the design's size

Although the Early Design variant of COCOMO enables analysis at the design stage, the COCOMO formulas require an estimate of the implementation's size in units of standard lines of code [3]. COCOMO recommends obtaining this element by counting the number of function points in the design and converting those function points into an estimate for standard lines of code. Function points fall into five categories:

- Internal Logical Files³
- External Interface Files
- External Inputs
- External Outputs
- External Queries

Each function point count is weighted by a conversion factor (depending on the design element's complexity), summed over all five categories, and then multiplied by the average number of lines of code per function point. The relation between function points and lines of code depends on the programming language.

For example, based on guidelines in [3], we might evaluate our spreadsheet design space \mathbf{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 C++, and using the conversion factors provided in [3], 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 proposed designs have the following approximate sizes:

Design	\mathbf{D} Representation	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 provides the size estimates required by COCOMO II.

3.2.2 Use size to relate product attributes

COCOMO II Early Design model predicts project effort in person-months (PM)⁴ based on the size of the software as inferred from the design and a characterization of the development method

³ Note that function point "files" are logical. They may or may not actually be implemented as entities resident in a file system.

⁴ It also predicts time for development in calendar months; the form of the model is essentially similar.

in terms of cost drivers called effort multipliers (EM) and scale factors (SF) plus two organization-specific fitting parameters A and B [3]. Upon examination, we find that eight of the cost drivers (plus A and B) describe aspects of the development method, but four cost drivers actually describe properties of the product. We separate these as follow:

- 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 between person-months 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 additional dimension, in addition to PM.
- 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 \mathbf{M} as a vector, where each element corresponds to one element of EM_D or SF . We will also include the fitting parameters A and B 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”⁵.

Augmenting our example product attribute space \mathbf{A}^n with our five new product properties yields the new representation:

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

Representing the method description as outlined above yields:

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

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

$$\begin{aligned} PM &= \text{Size}^E * S \\ S &= A * \Pi EM_i \\ E &= B + 0.01 * \Sigma SF_i \end{aligned}$$

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 the next section we show how to evaluate the interaction among these properties. For simplicity we will henceforth use the nominal values $A=2.94$ and $B=0.91$.

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 \mathbf{M} in tables provided as part of the definition of COCOMO⁶.

⁵ 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.

⁶ 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.

For example, if $\mathbf{x} \in \mathbf{A}^n$ shows minimal schedule pressure, reliability, reusability, and platform difficulty, in an ideal software organization with process perfectly suited to this project, then S and E will take on minimal values: S evaluates to 0.23, and E evaluates to 0.91. 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 evaluates to 178.4, and E evaluates to 1.23. Taking the median (“nominal”) values, S evaluates to 2.94 and E evaluates to 1.10.

For each proposed spreadsheet import design, we can insert these minimal, nominal, and maximal values of S and E, along with the size estimates from Section 3.2.1, into the formula for PM:

Design	Minimal PM	Nominal PM	Maximal PM
All features	0.54	8.21	560.57
Database	0.13	1.46	81.90
Copy/paste	0.12	1.30	71.98

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 $\mathbf{m} \in \mathbf{M}$ 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 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 4, in addition to the tradeoffs as examined in Section 3.2.3.

3.2.3 Evaluate tradeoffs among product attributes

In order to highlight the tradeoffs among the product attributes, 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

$$\begin{aligned} PM &= (\text{Size}^E * S_D) * S_p \\ S_p &= \Pi EM_p \text{ where } EM_p \text{ ranges over } EM_p \\ S_D &= A * \Pi EM_D \text{ where } EM_D \text{ ranges over } EM_D \\ E &= B + 0.01 * \Sigma SF_i \text{ where } SF_i \text{ ranges over } SF \end{aligned}$$

This reorganizes the model into the form

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

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 $\mathbf{m} \in \mathbf{M}$ (A, B, and the elements of EM_D and SF), this becomes

$$\begin{aligned} PM &= 2.94 * 100^{1.0997} * \Pi EM_p \\ &= 465.3153 \Pi EM_p \end{aligned}$$

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. It is not immediately clear whether raising these qualities would increase the product's benefit enough to justify the increased costs that might result from a concomitant increase in the person-months. To address that question, we briefly turn to the benefit function B and cost function C in the next section.

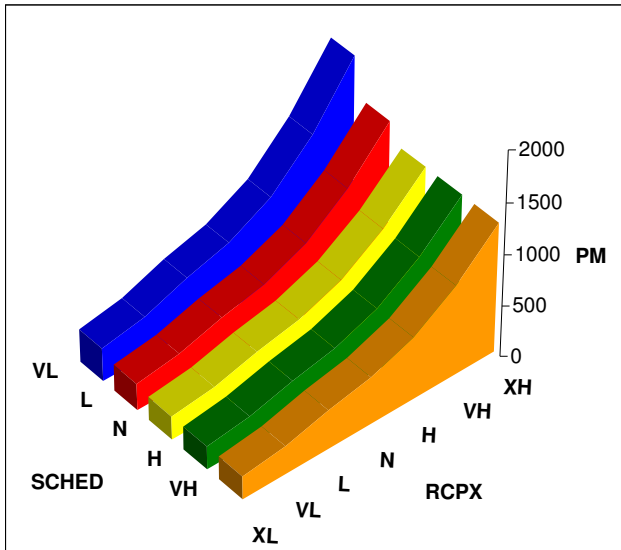


Figure 1. Tradeoffs among SCHED, RCPX, and PM

4. Other functions

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), researchers have made strides in assessing broad aspects of feature valuation by customers. For example, 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 [4].

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 [8].

In terms of costs, it is easy to convert the resulting person-months estimate 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 boxed 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.

Consequently, although we have outlined an approach for reasoning about the value of a software design and demonstrated how to begin putting our model into practice, a good deal of work remains. In the future, we will continue to identify ways to use existing research to evaluate the functions within our model, thereby continuing to extend its applicability and usefulness.

Acknowledgements

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

References

- [1] 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.
- [2] B. Boehm and V. Basili. Software Defect Reduction Top 10 List. *IEEE Computer*, 34, 1 (Jan 2001), 2-4.
- [3] B. Boehm et. al. *Software Cost Estimation with COCOMO II*, Prentice-Hall, 2000.
- [4] 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.
- [5] 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.
- [6] P. Layard and A. Walters. *Microeconomic Theory*, McGraw-Hill, New York, NY, 1978.
- [7] V. Poladian, S. Butler, M. Shaw, and D. Garlan. Time is Not Money: The Case for Multi-Dimensional Accounting in Value-Based Software Engineering. In *Fifth Workshop on Economics-Driven Software Research*, 2003, 19-24.
- [8] V. Poladian, D. Garlan, and M. Shaw. Software Selection and Configuration in Mobile Environments: A Utility-Based Approach. In *Fourth Workshop on Economics-Driven Software Research*, 2002.
- [9] M. Shaw, A. Arora, S. Butler, and C. Scaffidi. In Search of a Unified Theory for Early Predictive Design Evaluation for Software, *Carnegie Mellon University School of Computer Science Technical Report CMU-CS-03-105*, and *Institute for Software Research, International CMU-ISRI-04-118*, in preparation, http://shaw-weil.com/marian/DisplayPaper.asp?paper_id=84