Improving Size Estimates Using Historical Data

A software project's estimate of effort commonly requires input specifying the project size, and a reliable size estimate depends on many factors. This study examines a completed C++ project and considers programming artifacts that we can readily trace to requirements and early class design.

James Bielak, *Greenstone Software Architecture & Consulting*

Several estimation techniques and tools are available for predicting the amount of time and effort needed to develop software systems. Most of these techniques require a wide variety of input factors, including historical data, system complexity measures, the development team's level of skill, any project constraints, and an estimate of the volume of code (the project's size). For any of these estimation techniques to be effective, the organization doing the estimation must meet certain conditions. In particular, we must clearly define the estimation process, the practitioners must receive the appropriate training, everyone must follow the process, and we must evaluate the results for effectiveness.

There are only a few recognized methods for estimating software size. Generating reliable time and effort estimates depends heavily on accurately exercising at least one of the existing size estimation techniques. As a team of developers writing a scientific analysis application for the research department of a large oil company, the capability to generate estimates and work within our budget was a major concern. But considering that our project team did not have a defined process and was not trained in any of the available estimation techniques, we effectively had no process to follow. However, we did have a large amount of data in the form of already completed code. We suspected that we could mine information from this historical project data, enabling (or at least assisting) us in our attempts to estimate the size of future work. This article describes our attempt to develop guidelines for improving size estimates, by studying how the count of programming elements might help predict the code's ultimate size.

Project Background

In 1996, Arco's Exploration Research and Technical Services department wanted to acquire an application framework that would help geoscientists

- apply the oil and gas industry's best exploration practices,
- incorporate new analysis methods,
- use a user-programmable interface for evaluating new research methods, and
- accommodate reading and writing of various industry data formats.

0740-7459/00/$10.00 © 2000 IEEE

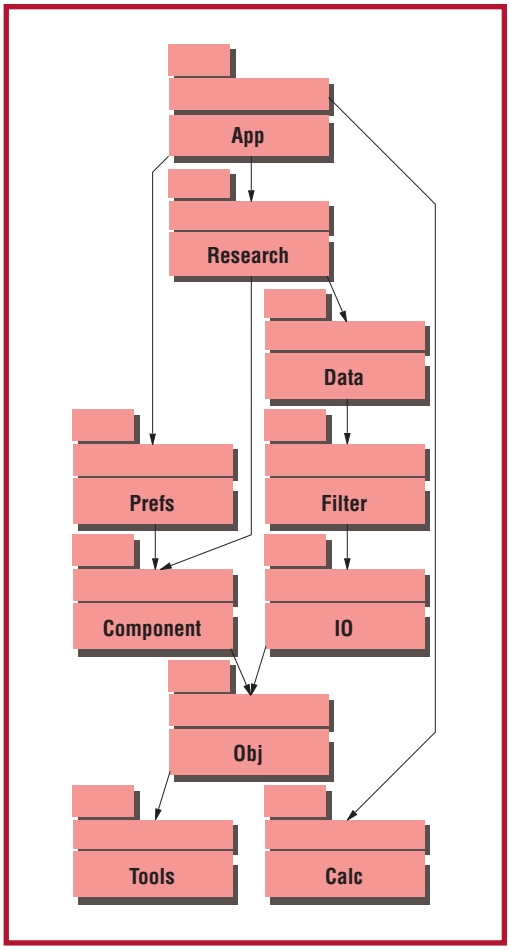

Figure 1. Direct Hydrocarbon Indicator Toolkit component packages.

The decision to build a proprietary in-house application created an opportunity to engineer an object-oriented, X/Motif-based C++ research and data-analysis application. After 24 months of development, the project team completed release 2.2 of the Direct Hydrocarbon Indicator (DHI) Toolkit in May 1999, fulfilling the goals to provide unique data viewing, analysis, and computation capabilities.

A staff of four to nine full-time developers created the component-based application containing the packages in Figure 1. The package layers reflect the application architecture, with high-level user interface components (App) depending on data delivery subsystems (Data), import/export facilities (Filter), reusable GUI components (Prefs and Component), and low-level abstract business objects (Obj) and utilities (Calc and Tools). Each package is internally composed of layers of discretely testable components with well-defined interfaces and dependencies; we consciously and aggressively avoided cyclical dependencies between components.[1]

The development team adopted a consistent programming style and participated in code reviews to catch defects early and ensure that we used standard coding style through each component. We used canonical methods established early in the project to handle X/Motif-to-C++ event notification throughout the development process. When development ceased, the DHI Toolkit comprised some 81,000 source lines of code (SLOC) of C++, distributed among 190 components.

Table 1 lists brief descriptions of the packages we used. We omitted three of the packages in Figure 1 from this study, because they contain contributions from scientific staff who did not participate in the development team's coding style (Research), employ languages other than C++ and contain non-object-oriented modules (Calculations), or contain support tools

Table 1

Direct Hydrocarbon Indicator Toolkit Software Package Descriptions

Package	Number of components	Description
Abstract business objects (Obj)	16	Basic abstract geoscience business objects, including projects, wells, well logs, markers, zones, and seismic data
Concrete business object input/output (I/O)	15	Concrete input/output behaviors for abstract business object persistence mechanisms
Data import/export filters (Filter)	14	Abstract factories, strategies, and GUI components for importing and exporting "foreign" data formats
Application data factory (Data)	6	Abstract data object factory providing GUI components for user data selection, delivering observers of abstract business objects to application contexts
Reusable GUI components (Component)	22	Reusable unit-labeling (feet/meters) text fields, data selection components, and calculation configuration components
Application support components (Prefs)	15	User preference, window layout, and session save/reinstantiation support
Application GUI (App)	73	User data display and editing, object property dialogs, and calculation setup dialogs

Estimation Techniques

Software estimation techniques generally fall into one of three categories:

- empirical techniques that relate observations of past performance to predictions of future efforts,
- regression models that are derived from historical data and describe the mathematical relationships among project variables, and
- theory-based techniques that are based on the underlying theoretical considerations of software development processes.[1]

For the purposes of this discussion, I merely draw a distinction between techniques that might help estimate size versus those used to estimate effort, schedule, and cost.

A well-known and widely used regression technique is Cocomo (*co*nstructive *co*st *mo*del).[2,3] The Cocomo II model estimates the effort, schedule, and cost required to develop a software product, accounting for different project phases and activities. This type of estimation method uses regression equations (developed from historical data) to compute schedule and cost by factoring in various project drivers such as team experience, the type of system under development, system size, nonfunctional product attributes, and so on.

The SLIM (*s*oftware *li*fecycle *m*anagement) method[4] is a theory-based technique[1] that uses two equations to estimate development effort and schedule. The software equation, derived from empirical observations about productivity levels, expresses development effort in terms of project size and development time. The manpower equation expresses the buildup of manpower as a function of development time.

Sizing techniques rely primarily on empirical methods. A few of these include the Standard and Wideband Delphi estimation methods, analogy techniques, the software sizing model (SSM), and function-point analysis. Observations and an understanding of historical project information can help predict the size of future efforts.

The Delphi methods[2,5] employ techniques for decomposing a project into individual work activities, letting a team of experts generate individual estimates for each activity and form a consensus estimate for the project. Estimation by analogy involves examining similarities and differences between former and current efforts and extrapolating the qualities of measured past work to future efforts.

The SSM decomposes a project into individual modules and employs different methods to estimate the relative size of software modules through pair-wise comparisons, PERT (identifying the lowest, most likely, and highest possible sizes) estimates, sorting, and ranking techniques. The estimates are calibrated to local conditions by including at least two reference modules of known size. The technique generates a size for each module and for the overall project.[6]

Function-point analysis[7] measures a software system's size in terms of system functionality, independent of implementation language. The function-point method is considered an empirical estimation approach[1] due to the observed relationship between the effort required to build a system and identifiable system features, such as external inputs, interface files, outputs, queries, and logical internal tables. Counts of system features are adjusted using weighting and complexity factors to arrive at a size expressed in function points. Although function-point analysis was originally developed in a world of database and procedural programming, the method has mapped well into the object-oriented development paradigm.[8]

References

1. R.E. Fairley, "Recent Advances in Software Estimation Techniques," *Proc. 14th Int'l Conf. Software Eng.*, ACM Press, New York, 1992.
2. B. Boehm, *Software Engineering Economics*, Prentice Hall, Upper Saddle River, N.J., 1981.
3. Barry Boehm et al., "Cost Models for Future Software Life Cycle Process: Cocomo 2.0," *Ann. of Software Eng. Special Volume on Software Process and Product Measurement*, J.D. Arther and S.M. Henry, eds., Science Publishers, Amsterdam, The Netherlands, Vol. 1, 1995, pp. 45–60.
4. L.H. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," *IEEE Trans. Software Eng.*, Vol. 4, No. 4, Apr. 1978, pp. 345–361.
5. K. Wiegers, "Stop Promising Miracles," *Software Development*, Vol. 8, No. 2, Feb. 2000, p. 49.
6. G.J. Bozoki, "Performance Simulation of SSM (Software Sizing Model)," *Proc. 13th Conf., Int'l Soc. of Parametric Analysts*, Int'l Soc. of Parametric Analysts, New Orleans, 1991, pp. CM–14.
7. A. Albrecht, "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Trans. Software Eng.*, Vol. SE-9, No. 6, 1983.
8. T. Fetcke, A. Abran, and T.H. Nguyen, "Mapping the OO-Jacobson Approach into Function Point Analysis," *Proc. TOOLS-23'97*, IEEE Press, Piscataway, N.J., 1998.

and utilities that are not directly traceable to specific requirements (Tools).

Estimation Study

The estimation tools available to our development team included Cocomo II and Estimate Pro V2.0.[2,3] (See the "Estimation Techniques" sidebar for a description of these and other estimator techniques.) Both of these tools generate estimates for the amount of effort, time, or cost, but they require input specifying the size of the work to be completed. The problem we faced throughout the project was estimating size. No one on the development team was trained in function-point analysis, so we loosely based our attempts at prediction on analogy methods and the Delphi principle of reaching consensus with individual, "expert" estimates. Without a defined, repeatable size-estimation process, these predictions were little better than outright guesses. Finally, our failure to record our predictions—and subsequently compare the actual size against the

estimate—did little to improve our skills.

As the project neared completion, we began to speculate on how the presence or absence of different programming elements affected code volume. In particular, we considered the impact of the number of

- GUI elements in a component;
- events and state changes handled by a window, dialog, or object;
- member functions per component; and
- reused system components used by a module

on the total size of a completed component. We identified the following programming elements as potential candidates for study because of their widespread use throughout the project, their apparent impact on size, and the likelihood of identifying, early in the design process, their eventual implementation:

- the number of GUI elements in a component estimated from drawings, storyboards, prototypes, or textual descriptions;
- events estimated by considering button pushes and subsequent behaviors, text field edits, invocation of dialogs, drawing element selections, moves, deletions, and so on;
- the number of member functions in class as an outcome of preliminary class design; and
- the presence of a reused component in code, which implies the need for manipulating or communicating with that object, which in turn contributes to code size.

We performed a quick exercise counting the elements listed earlier that compared the

Table 2

Definition Checklist for Source Statement Counts

Definition name: Logical source statements, DHI Toolkit (basic definition) — **Date: 1999** — **Originator: Arco**

Measurement unit Logical source statements

Statement type

When a line or statement contains more than one type, classify it as the type with the highest precedence

		Includes	Excludes
Executable	Order of precedence-> 1	x	
Nonexecutable			
Declarations	2	x	
Compiler directives	3	x	
Comments			
On their own lines	4		x
On lines with source code	5		x
Banners and nonblank spacers	6		x
Blank (empty) comments	7		x
Blank lines	8		x
C++ clarifications			
Null statement (for example, ";" by itself to indicate an empty body)			x
Expression statements (expressions terminated by semicolons)		x	
Expressions separated by semicolons, as in a "for" statement		x	
Block statements (for example, {…} with no terminating semicolon)		x	
"{", "}", or "};" on a line by itself when part of a declaration			x
"{" or "}" on a line by itself when part of an executable statement			x
Conditionally compiled statements (#if, #ifdef, #ifndef)			x
Preprocessor statements (#include *only*)		x	
Code inherited from superclasses			x
XmNresource resource statements, both set and get		x	

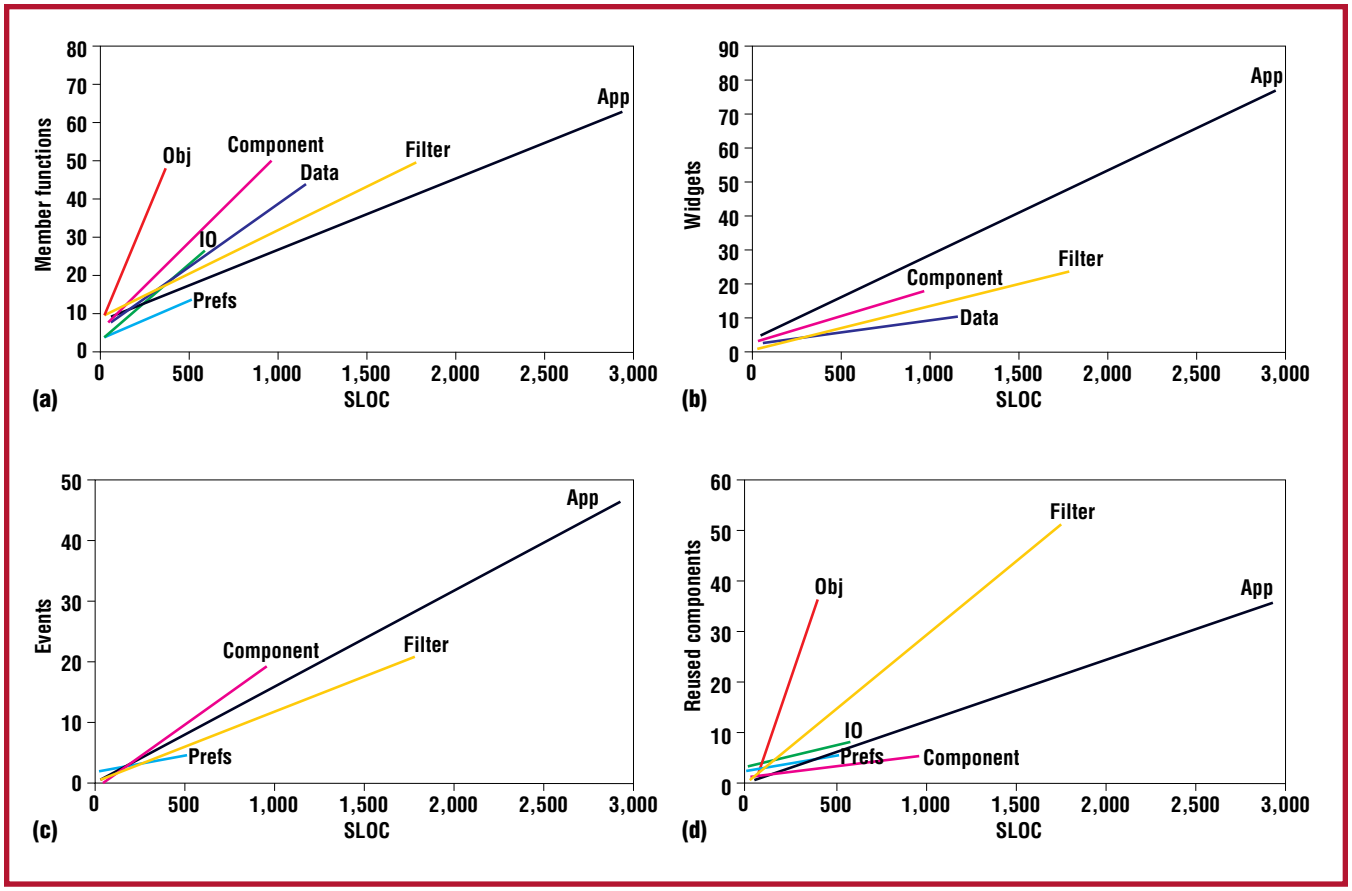

Figure 2. Programming elements versus source lines of code (SLOC) for project packages. Each trend line represents data from a separate project package: (a) shows regression lines comparing count of member functions versus size, (b) shows the number of widgets versus size, (c) shows how closely tied events and state changes are to the number of widgets in a component, and (d) shows the amount of component reuse in package components.

counts to the number of physical lines of code in the component. The trends we observed sparked interest in conducting a more comprehensive examination of the application code, using firm rules for counting the program elements and for defining a logical C++ source statement.

Component code elements are counted during line-by-line visual inspection, creating a tally of the number of instantiated Motif Widgets (such as GUI window elements like push buttons, text fields, and so forth), events and state changes handled, class member functions, and automatic and allocated (use of the "new" operator) instantiations of reused in-house components. Each compilation unit is considered a component, whether it contains one or multiple class definitions.

We determined the component size by using the Software Engineering Institute definition checklist for a logical source statement,[4] which we modified to include statements to set or retrieve Motif Widget resources. Table 2 shows the checklist used in this exercise.

Analysis

Each component contains one or more classes. Therefore, class member functions are common to all components, and they are easily counted. Figure 2a plots member function data and least squares regression lines for each component package. Table 3 lists the correlation coefficient of determination r^2 for each trend. An interesting feature of Figure 2a is that the spread of regression lines roughly reflects the position a package occupies within the overall architecture (see Figure 1). Comparing the slope of a package's trend to its level in the architecture shows that the amount of SLOC per member function increases for higher architectural levels. This feature might have interesting consequences for size estimation. At the business object level (Obj), components are relatively compact, with behaviors generally restricted to read, write, and data ac-

cessors. Indeed, several "set" and "get" functions consist of nothing more than one or two lines of code. At the other extreme, user interface member functions are on average several SLOC larger, carrying the burden of all the code required to process widget instantiation and configuration, event handling, and display updating.

Trend correlation coefficients (Table 3, column "Member functions") suggest that size estimates based on the count of member functions are less meaningful for architecturally higher packages than for low-level packages. Clearly, something besides the number of member functions contributes to the volume of code in components. Programming elements that are present in the higher levels of architecture, but generally absent from lower levels, are user interface widgets.

Figure 2b shows the count of widgets versus component size, from packages containing sufficient numbers of widgets. We can attribute the relationship between the number of these GUI elements versus component size to the amount of code required to instantiate and configure the widgets, as well as to handling widget and component behavior during callbacks. Table 3 (column "Widgets") shows mixed results from the attempt to correlate the count of widgets against size. Still, in the widget-intensive App package, the number of GUI elements appears to strongly influence and correlate well with component size.

A count of events or state changes handled by components (see Figure 2c) produces high scatter. Our suspicion that the amount of user event-handling code (push buttons, radio buttons, text field edits, data selection, and so forth) directly affects the overall size of a component in a predictable fashion seems inconclusive. Packages App and Filter produce the best correlations (Table 3, "Events" column), which might be expected given the fact that these two packages generate the highest degree of correlation between count of widgets and size (Figure 2b), coupled with the fact that widget callbacks are in general handled as event and state changes.

The final programming element we examined involves the amount that reused components affect module size. Figure 2d illustrates count versus size for six of the component packages. Each trend line reflects how components within individual packages use objects from lower levels in the architecture, but there is no coherent predictability in trend variance between packages. Business objects (Obj) are typically compact code, and hierarchical parent–child relationships between these business objects result in a high degree of reuse, generating a steep count versus size trend slope. Reusable GUI components (Component) have larger sizes relative to the number of objects they use. This is in part because they contain code to generalize and abstract their behavior for use in a broad range of conditions, coupled with minimal dependence or collaboration with other components.

Table 3

Least Squares Coefficient of Determination r^2 for Programming Elements

	Member functions	Widgets	Events	Reused components	Members + widgets	Member + widgets + events	Members + widgets + events + reused components
App	0.3503	0.6079	0.6011	0.4841	0.7161	0.7941	0.8660
Comp	0.6354	0.3407	0.3697	0.1496	0.8623	0.8680	0.8399
Data	0.8844	0.6587	(2 pts)	(3 pts)	0.9512	0.9770	0.9779
Filter	0.3426	0.4787	0.5455	0.6757	0.6555	0.7468	0.8125
IO	0.8083	(0 pts)	(0 pts)	0.0692	0.8083	0.8083	0.9227
Obj	0.8808	(0 pts)	(0 pts)	0.6423	0.8808	0.8808	0.9021
Prefs	0.4896	(2 pts)	0.0429	0.0958	0.8117	0.8353	0.8322
All data	0.3602	0.6497	0.6213	0.3705	0.7544	0.8213	0.8593

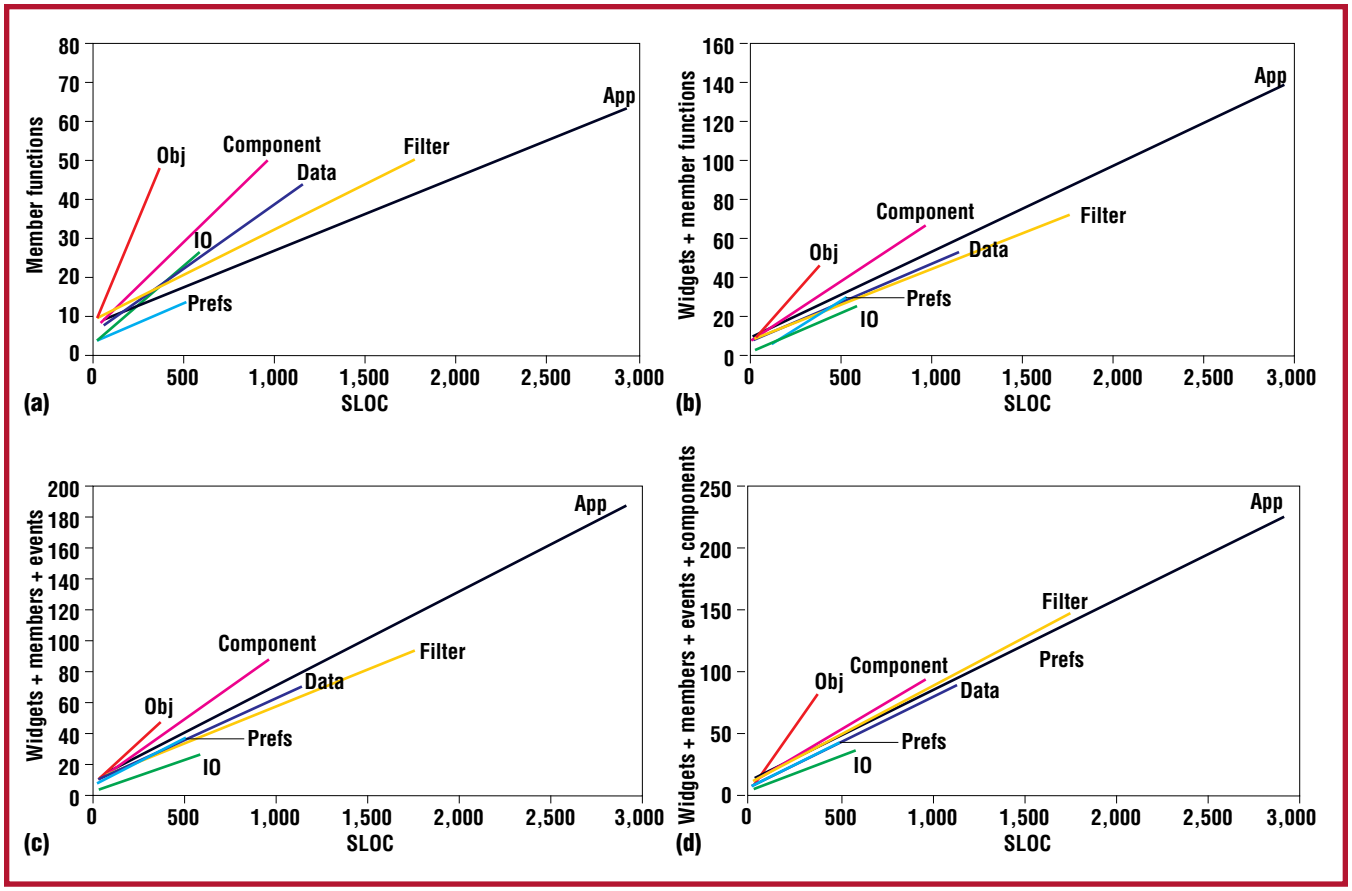

Figure 3. Summed programming elements versus source lines of code for project packages. A repeat of Figure 2a (a) plots the count of member functions to reflect a package's architectural position. Adding the number of widgets to a count of member functions (b) causes trends from different packages to converge. Adding the number of event changes to the data in (b) further refines a trend of count versus size (c), and the sum of all studied programming elements (d) shows very similar count versus size trends for the individual project packages.

A component is more than just a collection of member functions, widgets, or event-handling code. Component size is the sum of many things, including behavioral logic, the instantiation and manipulation of helper components, construction, initiation, destruction, and so on. Bearing this in mind, we tried summing the counted programming elements within modules to determine if we could refine our prediction of component size.

Figure 3 illustrates the results of summing the number of widgets with the number of member functions, adding to that the number of events, and finally adding the number of reused components. Table 3 shows that, in general, correlation between the number of summed elements versus size improves, because more program elements are included in the sum. Except for the trend line exhibited by the Obj package, all trends appear to converge toward a single slope. The Obj package, occupying a low level in the architecture and composed of highly specialized components, shows a trend significantly steeper than other packages.

We can use the count of different programming elements, which we can discern at either the requirements or preliminary class design stage, to roughly estimate the size of individual components. Identifying appropriate programming elements that are easily counted helped us establish relationships between the number of these elements and the resulting SLOC. Taken individually, each type of selected programming element used in the project's source code generated a trend correlating count with size, summarized in Figure 4. Table 3 ("All data" row) shows that correlation coefficients for the different elements across the entire project range from good to poor.

However, summing relevant programming elements refined the relationship between count and size, as Figure 5 illustrates.

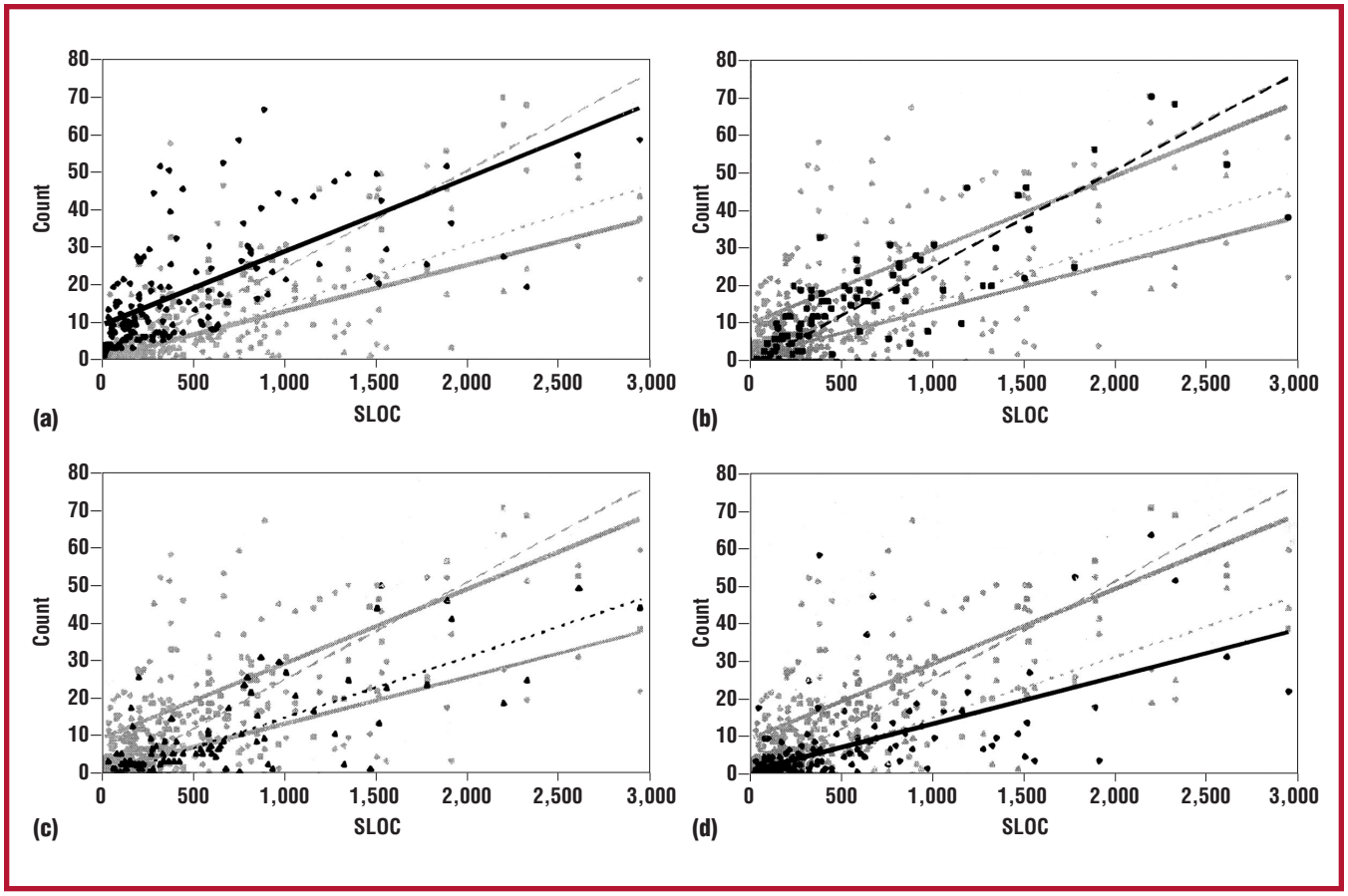

Figure 4. Project-programming elements versus source lines of code. Plotting all project component counts versus size, irrespective of where the package components come from, results in four trend lines for the different programming elements we studied. (a) Member functions (all components versus size result in fair scatter, (b) all components containing widgets versus size show a good trend. (c) Event changes for all components result in a trend that parallels (b) because events are used to handle widget actions. (d) The amount of component reuse from all components in the project produces high scatter and low correlation of count versus size.

By summing counted element types and plotting them against component size, we could include additional information in the estimate. This results in steeper trend lines that display better correlations between count and size. Upon reflection, this seems intuitively obvious. The more "things" in a program, and the more "stuff" it does, the more that data contributes to longer code (this is the basis for the function-point analysis technique). By being able to identify and count some of these things early in the development process, we can use such elements to estimate code size more confidently.

Results of our investigation showed that artifact counts are related to size in varying degrees. This observation is probably analogous to the way that function-point analysis estimates size by counting the number of identifiable external inputs, outputs, inquiries, files, and internal tables. Summing our counted elements increased correlation coefficients in trends we established, suggesting that providing more information to an estimate increases the prediction's reliability.

The results this technique obtained were most definitely influenced by our development environment. The small team of developers, sharing a similar coding style and common techniques for handling recurring programming idioms, had jelled to a high degree over the two-year project. During code reviews, it became difficult to identify a particular author due to our consistent style of programming. Any departure from the established coding style or programming idioms would definitely have an impact on the number of lines of code written, relative to the counted artifacts.

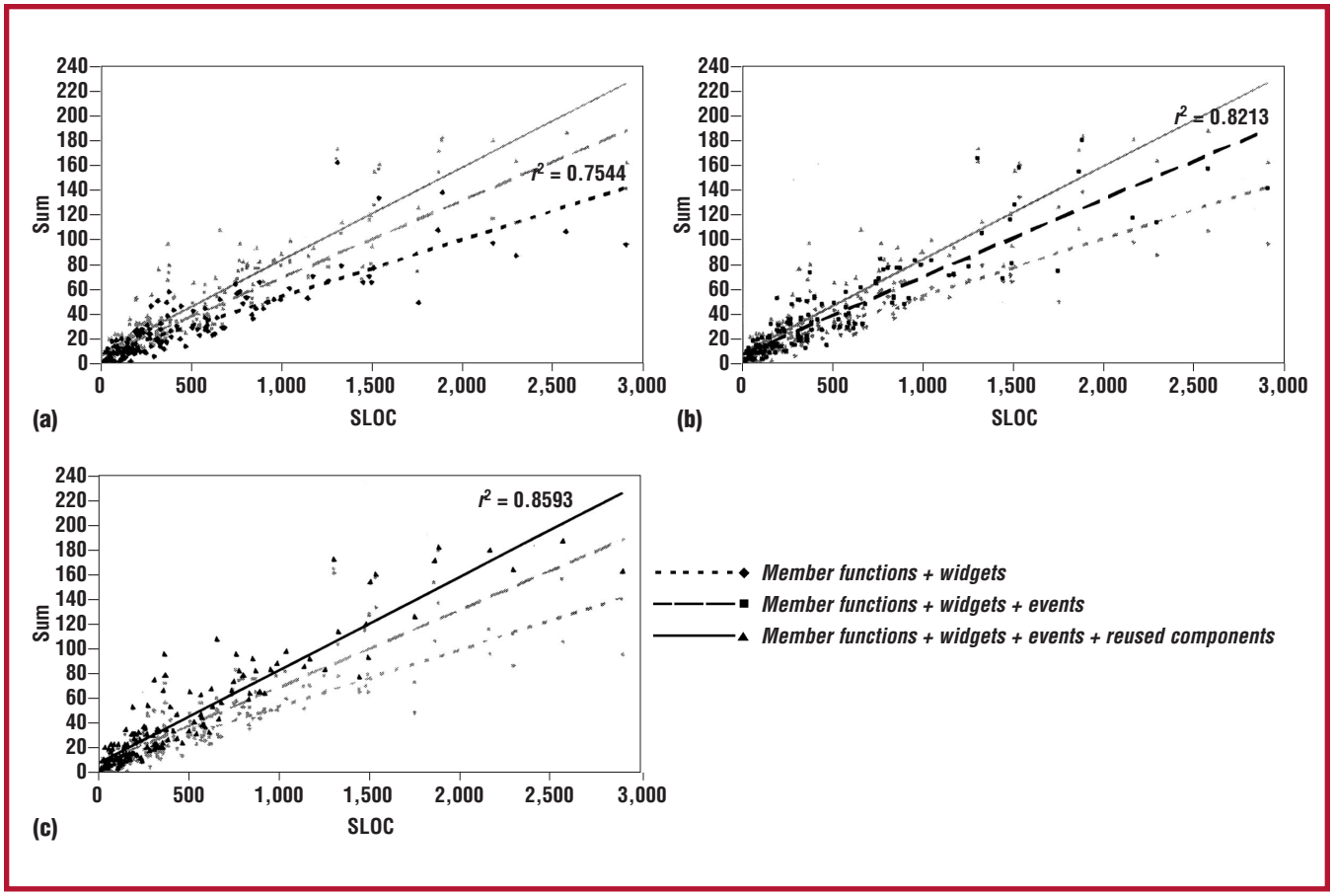

Figure 5. Summed project-programming elements versus source lines of code. (a) Member functions plus widgets, for all components with widgets, produce the best correlation coefficient so far (r^2 = 0.7544). (b) By adding in the number of events to the previous data, the trend's slope increases, as does the correlation coefficient (r^2 = 0.8213). (c) The best correlation (r^2 = 0.8593) is reached by summing all studied programming elements for each project component.

Consequently, count versus size data reported here will not directly apply to other development teams or different problem domains, languages, or systems that are significantly larger or smaller than the one in this study. Furthermore, the results I present here represent only one study of a single system. Generalizing this technique for application to other programming systems, or fully understanding the relationships we report, requires the study of many more systems and bringing more sophisticated analysis techniques to bear on historical data.

For our purposes, the relationships we discovered could help us sharpen our estimation skills, both as "expert" opinion-givers and during estimation by analogy. By mining our historical project data, we developed insights into count versus size that would let us examine our future predictions under the revealing light of past performance.

References

1. J. Lakos, *Large Scale C++ Software Projects*, Addison Wesley, Reading, Mass., 1996.
2. *USC Cocomo II.1997 Reference Manual*, USC Center for Software Eng., Univ. Southern California, Los Angeles, 1997.
3. *Estimate Professional V2.0 User's Guide*, Software Productivity Center, Vancouver, B.C., 1998.
4. R.E. Park, *Software Size Measurement: A Framework for Counting Source Statements*, Tech. Report CMU/SEI-92-TR-20 or ESC-TR-92-020, Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, Penn., 1992.

About the Author

James Bielak is president of Greenstone Software Architecture & Consulting, specializing in consulting, mentoring, and training software development teams. He has worked with the geoscience community for over 15 years, developing exploration, research, and production software. His current efforts focus on improving the software development process for a number of client companies. He received an MS in geology from the University of Montana. Contact him at Greenstone Software Architecture & Consulting, 2015 Greenstone, Carrollton, TX 75010-4000; jbielak@greenstoneconsulting.com.