Preprint¾ 1998 Workshop on Software and Performance

Predicting Memory Use from a Class Diagram using Dynamic Information

Gail C. Murphy and Ekaterina Saenko
Department of Computer Science, University of British Columbia
201-2366 Main Mall, Vancouver B.C. Canada V6T 1Z4
{murphy,saenko}@cs.ubc.ca

ABSTRACT

Increasingly, new applications are being built by composing existing software components rather than by coding a system from scratch. Using this approach, applications can be built quickly. Far too often, however, these applications do not exhibit acceptable performance. The benefits of construction through composition could be more reliably achieved if a software engineer could assess the performance impact of a component prior to its use. In this paper, we present our experiences of taking a straightforward approach to a piece of this problem: predicting the memory use of an existing component for a new application. The approach consists of annotating a structural view¾ a class diagram¾ of an object-oriented component with data gathered from sample existing uses of the component. The annotated view is then used as a basis for prediction. Our experience highlights several of the difficulties software engineers face in not only predicting, but also in analyzing, the space performance of existing object-oriented components.

1. Introduction

Increasingly, new applications are being built by composing existing software components rather than by coding a system from scratch. This mode of development can provide great benefits, including shorter development cycles to deliver more functionality [1].

Development benefits, however, do not generally come without an associated cost. When the components that are being composed and used are object-oriented libraries and frameworks, the cost is often increased space usage of the resulting application. Version 1.0 of the Java Abstract Window Toolkit (AWT), for example, mapped AWT components to native windows, requiring native data structures to be allocated in addition to the AWT structures. Version 1.1 of the toolkit provides alternative lightweight components to free applications from this restriction.

Generally, a software engineer is not able to assess the performance impact of a library or framework until the application is built. If the performance of the application is not sufficient, the engineer must then take some typically costly action, such as altering the use of or replacing the component.[1] To preserve the benefits intended from using existing components, it would be desirable if a software engineer was able to assess whether selected components will provide acceptable performance prior to their incorporation into a system.

In this paper, we present our experiences in addressing a piece of this problem: predicting the performance of an existing component in a new context. This problem has arisen in part of one of the author's research that involves the construction of source code analysis tools. Multiple components were available to provide parsing and program representation support. Since large applications were to be analyzed, the memory profiles of the existing components were an issue. Before committing to a choice of component, it would have been desirable to predict the memory usage for the types of input expected. Since no tool support or documentation was available to reason about this aspect of the components, a component, the SUIF framework [2], was selected using other criteria and the source analysis tools were implemented. The implemented tools do not scale as desired partly because of the memory usage of the chosen component.

To enable a better choice between components to be made the next time, we wanted to investigate if a straightforward approach based on gathering dynamic information about a component, and then reasoning about that data in terms of the component's structure might support prediction and facilitate comparison. We were interested in applying tools to support this approach that are readily available to most practicing software engineers.

Given the particular problem of interest, our focus was on a way to predict the maximum and average memory use of an existing component when providing specific behaviour. We assumed a situation in which the engineer had access to both sample applications demonstrating the use of the component, and the component's source code. We chose to focus on the collection of dynamic information about a component's memory use and chose to use a class diagram as the structural basis since the

[1]Through the rest of this paper, we use the term component to refer to an object-oriented library or framework.

component of interest, the SUIF compiler framework, was written in C++ [3]. Our early experiences with this approach have helped identify a number of problems in gathering and interpreting appropriate run-time information for use in prediction.

We begin, in Section 2, by describing existing approaches to analyzing and predicting the memory use of components. In Section 3, we then outline our approach to predicting memory usage through an annotated class diagram. Section 4 evaluates the predictions we made using our approach for the SUIF framework. In Section 5, we discuss our experiences and suggest avenues for further investigation to address the problems identified. Section 6 summarizes the paper.

2. Existing Approaches

Research in software performance modeling has largely focused on execution speed rather than on memory requirements [4]. Software developers who must consider the potential memory use of an application under development thus typically rely on ad-hoc methods applied to design documentation. One such ad-hoc approach, where the developer applies knowledge of the size and frequency of design entities, is outlined in Section 3.2. When the developer has extensive knowledge of the design, these ad-hoc approaches can produce useful results. However, in the case of interest in this paper, when the developer is making use of unfamiliar existing code, it is generally impossible for the developer to make reasonable estimates.

Consequently, when considering the reuse of components, developers are most often reliant on documentation provided with the component to assess potential memory use. Sometimes, the documentation explicitly addresses this issue. For example, the Booch data structure components [5] discuss memory issues explicitly, in part because the components include interfaces that allow the developer to control memory use.

In most cases, however, memory use is not addressed in the documentation. To gain a better understanding of a component's memory requirements, a developer might run sample applications with a self-selected set of test inputs. If the sample applications use functionality from the component similar to the desired application, this data may provide a reasonable basis for prediction. However, since sample applications are meant to demonstrate particular features of a component, it is more common that a developer must consider the execution of several sample applications to understand how the component might perform under the desired use. It is difficult to consolidate this coarse data into a meaningful basis for prediction as application overhead can be difficult to factor out.

Few tools to help an engineer analyze the memory use of the existing component exist beyond the system provided memory monitors, such as the Unix `ps` and `top` commands. One category of tools an engineer might use is the class of memory leak tools (e.g., mprof[2], dmalloc[3], etc.). These tools can help assure the engineer that the component is well-constructed, but they do not give the engineer a deeper understanding of overall memory use. Purify, a commonly used memory leak detector, provides additional memory tracking functionality, including an ability to describe all memory currently in use [6]. Although an engineer can use this feature to periodically describe the memory state of an application during execution, support to describe transient memory allocated and deallocated within the period is not provided.

Another category of tools includes run-time monitoring and tracing tools such as the Unix `prof` or `pixie` tools. Similar to performance modeling techniques, these tools focus on speed issues rather than on memory use.

Finally, an engineer might use a software visualization tool for dynamic information to provide more insight tools into the operation of an application than possible with the monitoring tools. IBM Research's Jinsight tool[4], for instance, displays information about the objects allocated and deallocated in Java program. These displays can likely help an engineer understand memory use in an existing program, but it is unclear if the visualizations would provide sufficient detail in an appropriate form to help an engineer predict the memory required by a component in a new context.

3. A Structural Approach

Existing approaches to help a software engineer assess the memory use of a component are insufficient because they do not provide a sufficiently detailed basis for prediction. To investigate the kind of support required to move towards a predictive basis, we tried applying a straightforward approach to mapping execution data to structural information. The structural information we chose to use was a class diagram.

The class diagram plays a central role in most object-oriented development. This structural view is part of all popular object-oriented development methods and is commonly used to describe an object-oriented system. For instance, class diagrams are a major part of describing many design patterns [7].

Because the memory use of an object-oriented system is closely aligned with the structural information shown in a class diagram, we wanted to be able to use this view to reason about and predict space performance of a component. More specifically, we wanted to be able to break down a component's memory use in terms of the class diagram view and to then roughly correlate how that memory view corresponds to the input to the component. Given a characterization of the expected input, we would

[2] mprof was developed and is distributed by Ben Zorn.
[3] Gray Watson developed and distributes dmalloc.
[4] See http://www.alphaworks.ibm.com/formula/jinsight.

then be able to predict the memory use of different components for the desired processing.

The approach we attempted to apply consists of three steps. In the first step (Section 3.1), the software engineer gathers data about a component's memory usage. Sample applications distributed with a component can be used to generate appropriate information. In the second step (Section 3.2), the dynamic information is annotated onto the class diagram. In the third step (Section 3.3), the engineer designates a particular class to use as a basis for projecting memory use. The engineer then uses this basis to predict the memory use of the component in the context of the desired application.

3.1. Gathering Data

We wanted to track every object allocation and deallocation that occurred when a component was used. We also wanted to know the size of each allocation and deallocation. Conceptually, these questions seem straightforward. In practice, this information was hard to determine (see Section 5.1).

To gather the desired data for a C++ component, we overloaded the C++ `new` and `delete` functions to track the size, in bytes, of the allocations and deallocations. Since we did not have access to type information within these overloaded functions, we used lexical scripts to alter each constructor and destructor to record the class of the object being allocated or deallocated. With this approach, we could gather an approximation of the execution information of interest.

3.2. Annotating Structural Information

A class diagram can show many different relationships between parts of a system. Two relations in particular that are useful in reasoning about memory use are the inheritance and association relations between classes [8]. Figure 1 shows a snippet of a class diagram with these relations for part of a hypothetical windowing system (based on [8], p.44). The figure shows that each panel in an interface consists of a number of panel items that may be buttons or text items.

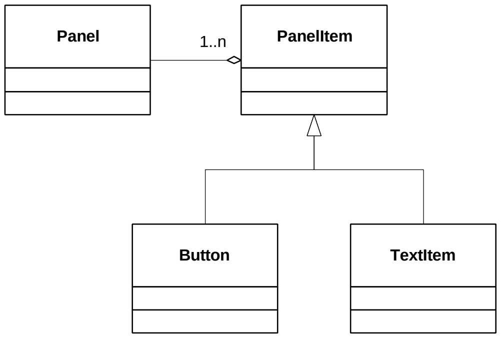

Figure 1. Sample class diagram.

A software engineer might use this class diagram information to reason about the eventual memory use of an application built with these classes. For instance, the average memory required for a panel might be calculated as the size required to represent the panel object plus the average number of buttons and text items in each panel multiplied by the average size of each of these objects respectively.

Predicting memory use in this manner requires the engineer to have access to structural information, and to approximate both the size and relative multiplicity of objects of the classes. When the engineer is the developer of both the component and the desired application, the structural information is available, and it is reasonable for the engineer to provide estimates of the values of interest. However, when the engineer is unfamiliar with the component, it is necessary to both determine the structural information and to provide guidance on suitable values.

Determining the structural information is relatively straightforward. Given the source code, it is reasonably easy to reverse engineer a class diagram for a system. We wrote a lexical script using the GNU awk program to approximate a class diagram with the two relations of interest for the SUIF framework.[5] Figure 2 shows a fragment of the extracted class diagram. The full diagram includes 118 classes with 180 inheritance and association links between the classes.

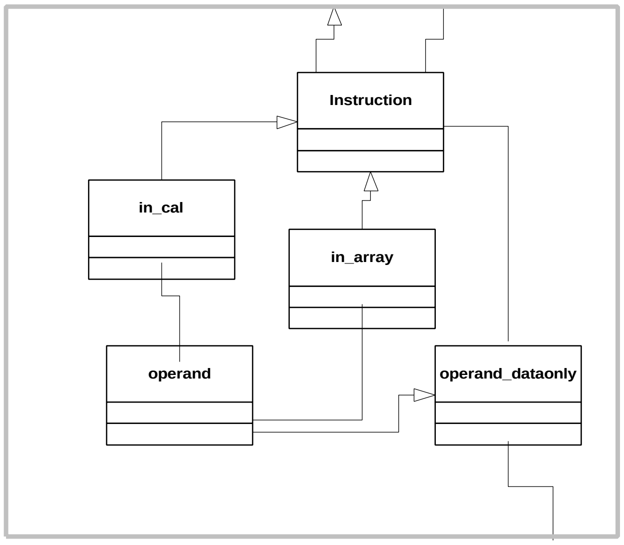

Figure 2. A fragment of the extracted SUIF class diagram. Lines with arrowheads represent inheritance. Lines without arrowheads represent association.

[5]We were considering primarily the base SUIF framework that is comprised of approximately 37 000 lines of C++ code.

Determining suitable values is more difficult. To ease the engineer's task, we wanted to annotate the extracted class diagram with dynamic information about the execution of the component. Object size is associated easily with the class diagram by computing the average and maximum size of objects of each class across the execution of sample applications with a variety of input.

Associating the relative numbers of objects of each class with the diagram is more challenging. Our approach divides this sub-problem into two parts. First, we extend the data gathering process described in Section 3.1 to also track limited information about the sample applications. Second, we process the gathered data in the context of the prediction of interest. We discuss the data gathering extension next. Section 3.3 discusses the processing of the data.

Since the memory use of an application changes across its execution and we were primarily interested in specific behaviours of the component, we had to take into account the usage undulations when determining relative numbers of objects. We accomplish this by asking the engineer to divide the execution of each sample application into different phases. Phases are chosen to isolate behaviours of the component; for instance, one phase might represent initialization of an abstract syntax tree, another phase might represent a search through the tree.

The engineer designates entry into the different execution pases by inserting print statements into the application code. For each phase (of each sample application's execution with each different input), we calculate the maximum number of objects of each class potentially active during the phase by ignoring all deallocation information. We then consider the deallocation information to calculate the number of objects of each class active on exit from each phase. The information gathered in each phase for each execution is referred to as a data set. All data sets collected are passed to the third step of our approach.

For SUIF, we collected the data of interest using two example applications distributed with SUIF and two different inputs (Section 4).

3.3. Using the Annotated Structure for Prediction

To support prediction, we wanted to allow the software engineer to designate a class for which they could provide a reasonable estimate of the number of objects at run-time. We then wanted to automatically present the engineer with a prediction of the maximum and average memory use of the component. A class for which the engineer might be able to provide a reasonable estimate is one that they believe corresponds to some interface between the desired application and the component.

The class for which an estimate will be given forms the *basis* class for the prediction. Given this class, we can process the data gathered about the maximum and average number of active objects of each class. More precisely, we determine, for each data set, the maximum and average number of active objects for each class relative to the basis class. We then compute, across all data sets, the maximum and average number of active objects (relative to the basis class).

For instance, in the case of the SUIF framework, we perused the extracted class diagram and noted the classes corresponding to different kinds of instructions in the SUIF intermediate representation. One class, `in_cal`, represents call statements in the analyzed program. We chose this class as a basis class because we could provide reasonable estimates on the number of call statements we expected in input to our eventual application.

Given an estimate of active basis objects, we can compute the average and maximum predicted size (in bytes) of the component by applying the algorithm the engineer might apply by hand, namely:

- scaling the average and maximum number of active objects of each class by the number of estimated basis objects,
- computing the bytes required to represent all active objects of each class through the use of the appropriate object size information, and then
- computing the sum of the bytes required to represent all active objects in the component.

4. Applying the approach

To get a sense of whether the approach shows any promise, we tried using it to predict the amount of memory that the SUIF framework might require to support the building of a call graph extractor for C code. We had previously built such an extractor using SUIF and it was available for comparison.

As described in Section 3, we modified the SUIF framework to gather the appropriate memory use information during execution and extracted the class diagram shown in Figure 2. The next step in our approach requires the gathering of data. A set of examples—the SUIF cookbook—is available from the framework developers. We chose three examples (i.e., prog1, prog2, and prog5) from this set that used features of the framework similar to the features that might be required by the call graph extractor. We then inserted execution phase statements into the example applications and gathered data about the maximum and average object counts using two sample inputs: the 8000 lines of source comprising the Unix adventure program (Version 6) and the 5100 lines of source comprising the GNU sort program (Version 1.21).[6]

[6] All execution data was gathered on an Alpha running Digital Unix 3.2. None of the applications reported memory leaks when run with purify on a SPARC 5 running Solaris 2.5.

Table 1. Predictions of memory use per SUIF-analyzed procedure for Call Graph Extractor. Average value is first followed by maximum value in parentheses. All values are reported in megabytes.

Basis Class/ Eg. Prog.	`in_cal` (75)		`tree_instr` (150)		`var_def` (45)	
	adventure	sort	Adventure	sort	adventure	sort
prog1	.52 (8.91)	2.79 (30.3)	3.88 (26.7)	1.36 (4.07)	.08 (5.33)	.11 (2.86)
prog2	.28 (8.93)	2.27 (3.04)	3.57 (26.7)	1.11 (4.07)	.06 (5.33)	.09 (2.87)
prog5	.75 (16.6)	3.53 (33.4)	4.37 (27.1)	1.62 (230)	.10 (6.0)	.14 (3.72)

We wanted to predict if the SUIF framework might scale to larger inputs when used for the call graph extractor. Continuing our approach, we examined the extracted class diagram and chose three different basis classes to try: `in_cal`, `tree_instr`, and `var_def`. We chose multiple basis classes because we wanted to investigate how well our approach was working. We chose these classes because we believe they represent items—namely calls, instructions, and variable definitions respectively—whose occurrence in the expected input we could estimate reasonably. We then used our approach to predict the memory use for inputs comprised of as many as 75 calls, 150 instructions, or 45 variable definitions in a procedure. Table 1 shows the resulting predictions. The predicted values are given per procedure since the chosen execution phases were mostly based on the processing of individual procedures.

The predictions resulting from our approach varied widely from several thousands of bytes to megabytes. For comparison, Table 2 shows the maximum and average memory gain per execution phase of the developed call graph extractor application. The table reports data for the two sample inputs as well as a represented desired input (the 25,000 lines of C code GNU plot, Version 3.5). These values were computed from data reported by the Unix `ps` command.

Given the wide variance in predictions and the variance from a rough comparison, does this approach show any promise? We believe it does show some promise for two reasons. First, the average values reported of several hundreds of thousands of bytes are, albeit by a large margin, within the actual range. In some cases, they are within range because a perusal of the data showed that they arise from cases where the projected size of the input existed as a case in the input used to annotate the structure. The values may also be within range simply by pure chance. Two questions to ask are why are most (or all) of the average values reported not within range and why are the predicted ranges so broad? A perusal of the data used to compute these values suggests it may be because we are tracking only total counts of objects, rather than determining the dependences between the objects. We discuss this point further in Section 5.2. An evaluation of the approach enhanced with association tracking would allow a more careful assessment of whether the reasonable values predicted were either because of being reported actual occurrences or simply by chance.

A second reason to continue to investigate the approach is that the predictions attempted here were based on substantially approximate data (Section 5.3). Further investigation using more accurate data would give a better indication of the usefulness of the approach.

5. Experiences

The process of applying our approach identified a number of problems that must be addressed to connect gathered performance data with structural information for the purpose of prediction.

5.1. Tracking Performance

The use of gathered performance data for prediction is attractive because it ensures the effects of the component's operating environment are considered appropriately. Unfortunately, in some cases, such as memory tracking, it is difficult to gather the desired performance data in the context of the component. For example, for C++ code, we had to alter each existing constructor and destructor. Although the base allocation and deallocation functions can be overridden, the engineer does not have access in these functions to the appropriate context; for instance, one cannot determine the constructor causing the call to the allocation function.

Languages with reflective features, such as Smalltalk [9], provide better support to address aspects of the data gathering problem. With meta-class support, for instance, it is typically possible to add tracing information into every

Table 2. Memory use per SUIF-analyzed procedure as computed from Unix ps. Average value is first followed by maximum value in parentheses. All values are reported in megabytes.

Input/ Prog.	**adventure**	**Sort**	**gnuplot**
Call Graph Extractor	.60 (1.11)	0.19 (0.57)	0.52 (1.2)

constructor and destructor. Reflective language features, however, do not solve the problem because they do not generally provide any help in tracking operating environment performance for the component. Garbage collection, for example, is typically part of the run-time operating environment, making it difficult to track memory deallocations using reflective language features. Similarly, reflective language features do not generally help an engineer gather data on how the operating system is affecting a component's performance.

Some existing tools do provide a degree of support for reporting both application-level and run-time environment information in the context of the application. The Unix prof tool, for example, reports on the performance of both system and application functions. One difficulty with existing tools of this form is that the link between the reported information and the source is either by name only or by reference to a line number in an application. Forming a correspondence between the reported data and the source thus often requires additional sophisticated tools that understand the semantics of the language.

Overcoming some of these difficulties may involve combining reflective language-level support with new interfaces into the operating environment. In addition, since the source for a component of interest will not always be available, methods for gathering and interpreting data from binaries will also be needed.

5.2. Mapping Execution Information to Structure

Using structural information to reason about gathered performance data requires mapping the execution information to the structure. In the approach described in this paper, the mapping is simple: performance data reported in terms of classes is mapped to classes in the structural information. This mapping provides some approximate bounds on the relative multiplicity of objects of varying classes. A more useful mapping would allow the gathered execution data to be attributed not only to classes in the design diagram, but also relations between the classes. For instance, in terms of Figure 1, it would be preferable to know how many `PanelItem` objects were created and destroyed per `Panel` object rather than overall during an execution phase. Supporting this mapping would require the abstractions in the design to be correlated to the source (e.g., where is the relation formed in the source) and would require additional data to be gathered during execution (e.g., the location in the source where a constructor was invoked).

In the longer-term, software engineers may find it useful to use, as a basis for prediction, structural information that is more abstract from the source than a class diagram. Software architectural information expressed in an architectural description language, for instance, may include components and connectors representing multiple source-level modules or classes, or multiple distributed pieces of code. Existing work in checking the conformance of designs to implementations (e.g., [10] or [11]) and in reverse engineering (e.g., [12] or [13]) may help address some of the mapping problems. New approaches to combine and summarize the gathered performance data associated with the source entities may be required.

5.3. Accuracy

What accuracy of the data gathered for prediction is necessary to be useful to a software engineer? The degree of accuracy will certainly depend on the task the engineer intends to perform based on the predicted values. If the engineer is attempting to size a real-time application, accurate data is likely needed. However, if the engineer is attempting to compare two components, more approximate data may suffice.

The data we gathered to use for the memory use predictions was highly approximate. One source of approximation was our use of lexical scripts to place trace information into the component. Since our scripts did not add any methods to a class, if a class did not have a constructor or destructor, it would not report trace information. An instantiation or deallocation may have been counted multiple times as the result of the call chain and limited filtering of the trace data. More accurate data could be collected by using syntax-based tools to ensure all component classes have appropriate trace information and by additional computation over the trace data. We also considered only dynamically allocated class memory in the component, ignoring static and automatic allocations. Incorporation of these values would also improve the data used for prediction.

6. Summary

Predicting the performance of an application early in a development cycle may help reduce development costs by averting costly changes that can arise when a developed application does not have acceptable performance. Increasingly, application development is incorporating the use of existing large components, such as object-oriented frameworks for user-interface development. Performance prediction methods that utilize knowledge of the execution of these components could help developers choose appropriate components to meet performance requirements.

In this paper, we have described our early experiences with an approach that maps execution data gathered from an existing component onto structural design information which is then used as the basis for predicting a component's memory use in a new context of use. Significant refinement of this approach is needed before a detailed assessment of its value can be undertaken. Even at this early stage, the process of applying the approach has highlighted problems that must be overcome to make the vision of predicting performance in the context of design information viable. These problems include difficulties in gathering and reporting acceptably accurate performance data in the context of the application, and appropriately mapping the gathered data to entities in the design.

Acknowledgements

This work was funded in part by the Natural Sciences and Engineering Research Council of Canada, and in part by the University of British Columbia.

REFERENCES

[1] K. J. Sullivan and J. C. Knight . Experience assessing an architectural approach to large-scale systematic reuse . *Proceedings of the 18th International Conference on Software Engineering*, pp. 220-229, 1996.

[2] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe , J. M. Anderson, S. W. K. Tjiang , S. W. Liao, C. W. Tseng, M. W. Hall, M. S. M. Lam, and J. L. Hennesy . SUIF: an infrastructure for research and parallelizing and optimizing compilers . *SIGPLAN Notices*, vol. 29, pp. 31-37, 1994.

[3] B. Stroustrup, *C++ Programming Language*. Addison-Wesley, 1986.

[4] C. Smith . Software performance engineering: a case study including performance comparison with design alternatives . *IEEE Transactions on Software Engineering*, vol. 19, pp. 720-741, 1993.

[5] G. Booch and M. Vilot . The design of the C++ Booch components. *Proceedings of OOPSLA ECOOP '90 Conference on Object-Oriented Systems, Languages, and Applications*, pp. 1-11, 1990.

[6] R. Hastings and B. Joyce . Purify: fast detection of memory leaks and access errors . *Proceedings of the Winter 1992 USENIX Conference*, pp. 125-136, 1991.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides . *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[8] J. Rumbaugh , M. Blaha , W. Premerlani , F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice Hall, 1991.

[9] A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley, 1983.

[10] G. C. Murphy, D. Notkin, and K. Sullivan . Software Reflexion Models: Bridging the Gap between Source and High-level Models . *Proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 18-28, 1995.

[11] M. Sefika, A. Sane, and R. H. Campbell . Monitoring compliance of a software system with its high-level design models. *Proceedings of the 18th International Conference on Software Engineering*, pp. 387-396, 1996.

[12] H. A. Müller and K. Klashinsky . A system for programming-in-the-large . *Proceedings of the 10th International Conference on Software Engineering*, pp. 80-86, 1988.

[13] D. H. Hutchens and V. R. Basili . System structure analysis: Clustering with data bindings . *IEEE Transactions on Software Engineering*, pp. 749-757, 1985.