

**METHOD AND SYSTEM FOR DIAGRAMING
COLLABORATIONS DEDUCED FROM
SMALL TALKCODE USING A DESIGN
VIRTUAL MACHINE**

RELATED INVENTION

IBM application Ser. No. 08/769,910 entitled "Method and System for Synchronizing Code with Design", filed concurrently herewith on Dec. 19, 1996.

FIELD OF THE INVENTION

The present invention relates in general to computer software, and in particular to a method and system for diagraming collaborations deduced from Smalltalk code using a design virtual machine.

BACKGROUND OF THE INVENTION

Software designs, much like abstract analogs (such as maps and blueprints), are built because they are useful for explaining, navigating, and understanding the richer underlying realities. With software, however, it is rare for even the most general design of an implemented system to be either complete or accurate. In many projects, senior programmers brainstorm on a white board, produce the program and produce just enough of a retrospective design to satisfy management. In projects with formal analysis and design stages, the design may be accurate when it is first made, but it seldom matches the final implementation. As code is developed it diverges from the design. These changes are rarely transferred back to the design documents because programmers seldom take the trouble to find and edit the design documents.

The lack of accurate design adds dramatically to the life cycle cost of software systems. Mismatches between design and code slow initial development of large systems because teams working on one portion of the system rely in part upon the design descriptions of other portions of a system. Inaccurate design has an even more dramatic effect on maintenance because maintenance done without understanding the underlying design is time consuming and prone to error.

Design and code can neither be completely separated from each other nor completely joined with one another. They overlap in that both describe the same system but are different because the intended audience of those descriptions are quite different. Design communicates the intent of the designers to other humans, while code communicates design intent to the machine. Humans share a vast common knowledge and can deal with abstractions but are weak at handling masses of detail. The machine is not hampered by details but is oblivious to abstraction and generality.

One prior art approach to synchronizing code and design supposes that if programmers are unable or unwilling to keep the code synchronized with design, perhaps programmers can be dispensed with and simply generate the code from the design. In some cases, such as when an application merely maintains a database, this approach works. However, for general programming this approach fails for several reasons. One of these reasons is that analysts and designers seldom, if ever anticipate all the details encountered in the actual coding. Programmers need to make changes that extend or "violate" the design because they discover relationships or cases not foreseen by the designers. Removing the programmers from the process does not impart previously unavailable omniscience to the designers. Additionally, most real world applications contain behavior

that is best described with algorithmic expressions. Programming code constructs have evolved to effectively and efficiently express such algorithms. Calling a detailed description of algorithmic behavior "design" simply because it is expressed in a formalism that isn't recognizable as "code" does not eliminate the complexity of the behavior.

Another previously known method is the automated extraction of object structure from code. Some tools are available that can create more or less detailed object structure diagrams directly from C++ class definitions that contain inheritance and attribute type information. Some Smalltalk systems provide similar attribute "type" information that allows these tools to be similarly effective. Without the attribute information, tools can only extract the inheritance structure. This method does not actually parse and model code other than C++ header files or Smalltalk class definitions. Therefore, this approach can at best identify "has-a" and "is-a" relationships. These relationships may imply collaboration but this approach does not specifically identify any of the transient collaborations that are important for understanding design. In addition, it does not provide any information about algorithms.

Another method is the automated deduction of design by analyzing code execution. Collaborations implicit in Smalltalk code are difficult to deduce statically from the code and may not be fully determined until run time. However, Smalltalk is strongly typed at runtime so it may be determined exactly what kind of objects are participating in all collaborations by examining the receiver and the arguments involved in all message sends. The resulting information can be used to specify the collaborations observed during the execution. This method suffers from the following problems: (1) it requires test cases to exercise the code; each of these test cases must construct an initial state which is sometimes elaborate; (2) the test cases themselves require careful construction and may become obsolete as the system changes; (3) the effort needed to construct and maintain these test cases can be a deterrent to routine use of this technique; and (4) full coverage by the test cases is difficult to obtain and the degree of coverage is difficult to assess. This undermines confidence in the resulting design. Without full coverage, the extracted collaboration design is likely to be incomplete in important ways. For instance, the way a system is designed to handle the exceptional cases can be more telling than the way it handles the common ones.

Larger-grain object oriented design involves just a handful of basic elements:

1) Object structure: diagrammatic specification of which objects are in a model and how they are statically related, these diagrams are referred to as Object Structure Diagrams (OSDs);

2) CRC (Class, Responsibilities, Collaborators): short textual description of object behavior (responsibilities) and a list of other related classes (collaborators); and

3) Object interaction: diagrammatic and textual representation of the timing of interaction between objects in response to a particular event or transaction within a system, these diagrams are referred to as Object Interaction Diagrams (OIDs).

These three design artifacts are different perspectives depicting how objects collaborate within an object oriented system. These diagrams are artifacts of the design process in that they represent the system at a level of abstraction different from code. This design information is intended to communicate certain details of a system and elide other details (such as implementation in code). Design artifacts are