Issues in the Design and Implementation of a Schema Designer for an OODBMS

Jay Almarode
Instantiations, Inc.
Portland, Oregon

1. Introduction

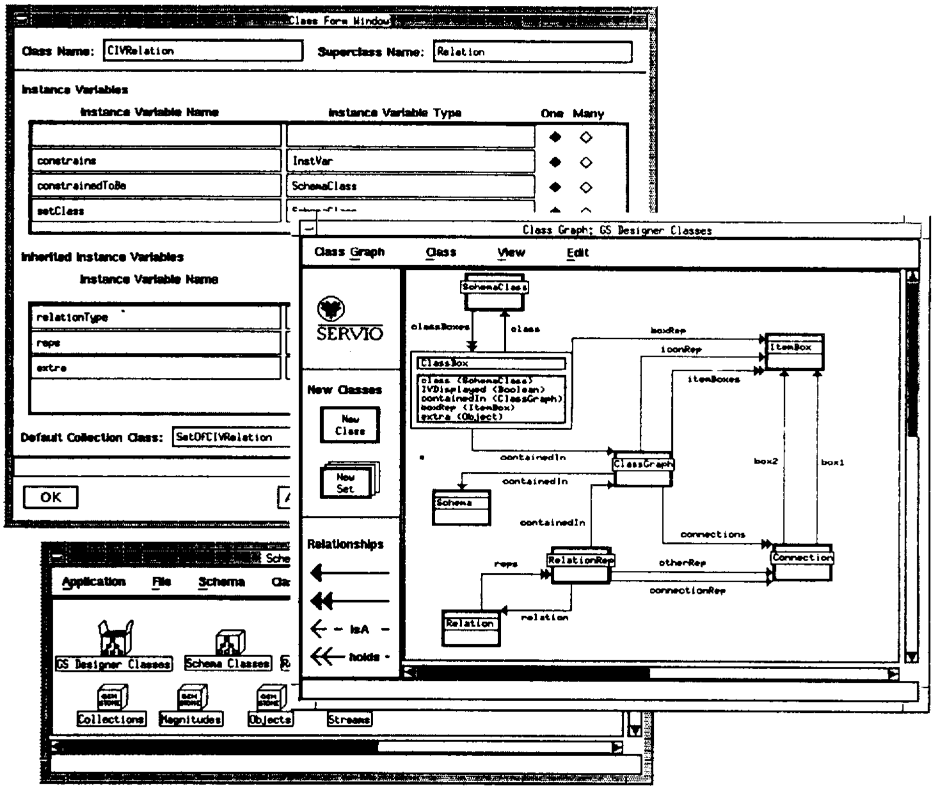
In an earlier paper [Almarode Anderson 90], we described the GemStone Visual Schema Designer, a graphical schema editor for the GemStone† object-oriented database management system. The GemStone Visual Schema Designer (GS Designer) allows the user to interactively define classes and relationships between them by drawing a graphical object model of the class definitions. GS Designer allows the user to create, modify and delete GemStone class definitions using a mouse and keyboard interface and bit-mapped graphics in a windowing environment. The tool utilizes state of the art user interface primitives and direct graphical manipulation to provide an easy-to-use, intuitive interface to operations. GS Designer is a commercial product designed to meet the needs of real application development.

In the course of designing and implementing the tool, a number of issues were encountered that are unique to object-oriented databases. These issues include order of database update, references to classes external to a particular schema, and concurrency conflict. This paper discusses some of the key problems encountered, proposed solutions to the problems, and a rationale for the solutions that were chosen. This paper is organized as follows. Section 2 gives an overview of the GS Designer. Section 3 discusses the order of updating class definitions in the database and Section 4 discusses the problems of external references to classes outside the schema. Section 5 discusses concurrency conflicts that can arise under optimistic concurrency control. We conclude with a brief discussion of future directions for the GS Designer.

2. Overview of the GemStone Designer

The main organizing principle of the GS Designer is the class graph. A class graph is a named collection of classes related to one another by various kinds of relationships. A particular class may be contained in more than one class graph, and a schema may consist of many class graphs. Collectively, all of the class graphs may be thought of as the schema for the application. The relationships currently implemented include generalization (realized by the superclass / subclass hierarchy), constrained instance variables, and constrained collections. Future relationships to be implemented include aggregation and association. All classes are part of a single superclass / subclass hierarchy (rooted at class **Object**), so any class class graph that the user creates or manipulates is a sub-graph connected to the class hierarchy, although these connections may not be displayed. Class graphs are used to partition all the classes of an application schema into logical subdivisions. Within a class graph, relationships between classes may selectively be displayed or hidden. Thus, a class graph is a versatile mechanism for viewing the meta-data of an application, and for managing the complexity of large schemas. An important design principle of the GS Designer is that the user should not be allowed to create an invalid class graph. All class graphs should represent a consistent database state, and the user should not be allowed to draw an incorrect schema given the information currently available to the GS Designer.

Figure 1 illustrates the various windows of the GS Designer. The bottom window is the schema window. It contains an icon for each class graph in the schema. All schemas automatically include the four GemStone class graphs that contain built-in classes, such as **Integer, String, Dictionary**, etc. The right-most window is a class graph window. In this window, each rectangle represents a class in the schema. An arrow represents a relationship between two classes. The solid single-headed and double-headed arrows represent named instance variables for a class. The arrow labeled **IsA** represents the superclass relationship between classes, and the arrow labeled **holds** represents a constrained collection. The top window is a class form window. This window allows the user to define a class textually rather than by drawing relationship arrows. An important feature of the GS Designer is that all windows are consistent and are updated dynamically. This means that a change in one window will be reflected in all windows that are appropriate.


Class Form Window

Class Name: CIVRelation
Superclass Name: Relation

Instance Variables

Instance Variable Name	Instance Variable Type	One	Many
		◆	◇
constrains	InstVar	◆	◇
constrainedToBe	SchemaClass	◆	◇
setClass			

Inherited Instance Variables

Instance Variable Name
relationType
reps
extra

Default Collection Class: SetOfCIVRelation

ClassBox
class (SchemaClass)
IVDisplayed (Boolean)
containedIn (ClassGraph)
boxRep (ItemBox)
extra (Object)

Figure 1.
The various windows of the GS Designer

The following list describes some of the main operations that can be performed by users of the GS Designer.

- Create a new class graph in the application schema. The new graph initially contains no classes.
- Delete the class graph. The class graph is deleted from the schema but classes in the graph remain, since they may be referenced by classes in other graphs.
- Create class definitions in the database. New classes are instantiated in the database and existing classes are modified if necessary.
- Save the schema as a graph object in the database. Saving the schema does not make any modifications to the class definitions in the database.
- Import an existing class into the class graph. The user may import a class from the built-in class library, from another class graph within this schema, or from outside the schema. A class my be imported according to the relationships it has with other classes, by cutting and pasting from another class graph window, or by name look-up.
- Generate a textual report. A text file of class definitions may be output for documentation purposes.
- File out the schema. A schema may be output in a form that can be input to other databases.
- File out the class definitions. Individual class definitions may be output so that the class definitions can be instantiated in other databases.

Another important feature of the GS Designer is that class and relationships may have multiple representations that can be manipulated. For example, a class may be represented in a class graph as a simple rectangle with its class name displayed and some of its instance variables displayed as arrows emanating from the rectangle. Another representation for a class in a class graph is a rectangle with all of the class's instance variables and their constraining classes listed inside. Still another representation for a class is a separate window which shows its superclass and lists all instance variables defined by the class and all its inherited instance variables. Since classes and relationships have multiple representations, it is possible to perform the same conceptual operation in many ways. For example, to reconstrain an instance variable, the user may type in a new constraint in the class form widow, or may graphically drag an arrow representing the instance variable from one class rectangle to another. A principle of good user interface design is to allow the user a number of ways to perform the same operation. Dynamic updating of all windows provides instant feedback of the results of the operation.

The remainder of this paper discusses some of the issues that were addressed during the design and implementation of the GS Designer.

3. Updating Class Definitions in the Database

One of the first issues that was addressed in the design of the GS Designer was when to update class definitions in the database. One alternative is to update a class definition in the database immediately after each operation on a class. Another alternative is to allow a number of operations to be performed and then allow the user to explicitly update the database with all class modifications. Most schema editors for non-traditional data models, including ISIS [Goldman et al. 85], Pasta3 [Kuntz Melchert 89], Siderius [Albano et al. 88], SNAP [Bryce Hull 86], and VILD [Leong et al. 89], have chosen the latter approach. We are aware of only one schema editor, the PSYCHO schema editor [KimH 88], that provides both alternatives.

For the first release of the GS Designer, we chose to provide an explicit operation to update the database. We felt it was desirable for the user to be able to save the schema design separately from actually updating the definitions in the database. This allows the user to modify and view private changes in the schema without changing classes that may be contained in other schemas. In addition, the user may file out the private changes and create the class definitions in a different database without updating the actual definitions in the original database. Another reason is performance. Some class modifications are expensive, potentially causing the recompilation of numerous methods in the class hierarchy. This delay can be noticeable with an interactive graphical tool, and discourages the user from exploring alternative designs. The next release of the GS Designer will cause us to re-evaluate whether to provide the first solution (continuously updating the database as changes are made) in addition to the second solution. The next release will provide integrated capability for defining and compiling methods for a class. Before a method can be compiled, any changes the user has made will need to be transmitted to the database, so that references to instance variables will be valid. In such a scenario, when the user opens a window for method definition for a class, an implicit operation will update the class's definition in the database.

Given that the user can make many modifications to more than one class before updating those changes to the database, the order that the changes are made in the database is important. The order of updates must ensure that the database is always in a consistent state, especially since methods may be invoked during the update operations. The ordering problem is particularly evident when a subclass reconstrains an instance variable inherited from a superclass. This reconstraining is allowable as long as the new constraint is a subclass of the superclass's constraint. Figure 2 illustrates an example of a subclass reconstraining an inherited instance variable. In this example, **Class2** may reconstrain inherited instance variable *y* as long as the new constraint is a subclass of **Dictionary**. When multiple deletions and additions of instance variables are performed, the order of class modifications must be performed correctly or this rule could be broken. This is best illustrated with an example. Figure 3 show two classes: **Class1** with no instance variables, and its subclass, **Class2**, with a single instance variable named *x* constrained to be a string. Now suppose that the user deletes instance variable *x* and redefines it for **Class1** to be constrained to an integer. The resulting class graph is pictured in Figure 4. When the class definitions are updated in the database, there are two operations to be performed: deleting *x* defined on **Class2** and adding *x* defined on **Class1**. If the addition of *x* is attempted first, it would not be allowed since *x* would be defined in a subclass and the constraint in the subclass would not be valid.

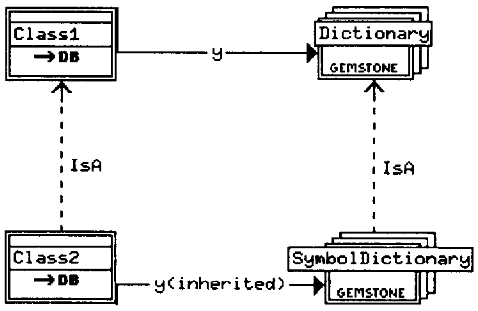


Figure 2.
An example of a correctly reconstrained inherited instance variable

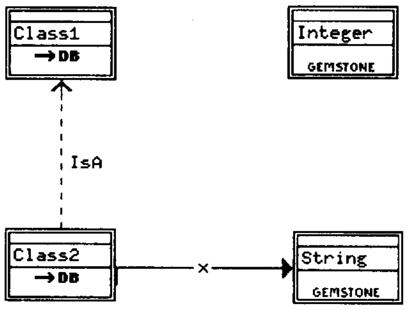


Figure 3.
A subclass with an instance variable

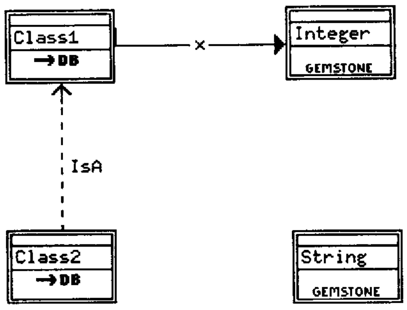


Figure 4.
After the instance variable has been moved to the superclass

A similar ordering problem occurs if the addition of a new instance variable causes a subclass's instance variable to become inherited. Figure 5 shows **Class2** with an instance variable *z* constrained to be **SmallInteger**. If the user adds a new instance variable *z* to **Class1** to make the instance variable in **Class2** a reconstrained inherited instance variable, as pictured in Figure 6, the following operations must be performed. First, the instance variable *z* must be deleted from **Class2**. Next, the new instance variable must be added to **Class1**. Finally, the inherited instance variable *z* for **Class2** must be constrained to SmallInteger.

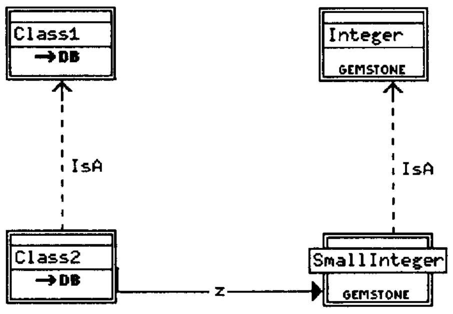


Figure 5.
A subclass with an instance variable

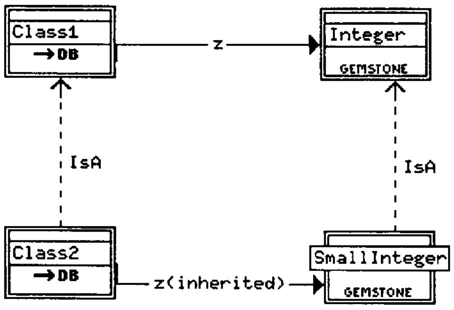


Figure 6.
After the instance variable has been made inherited

Although the user did not explicitly delete instance variable *z* for **Class2**, it must implicitly be deleted before *z* is added to **Class1**; otherwise, an error would occur for attempting to add a same-named instance variable that is defined by a subclass.

In light of the ordering problems described above, we have developed a general algorithm to perform multiple schema modifications. The purpose of the algorithm is to modify the actual class definitions in the database to conform to the definitions as pictured in the GS Designer. The pseudo code to perform multiple schema modifications is as follows:

```
1.  Create any new class definitions in the database

2.  For each class, remove any instance variables from the database definition
    that are not in the schema definition

    2a.  Find an instance variable in the database definition that is not in the
         schema definition

    2b.  Check all superclasses to see if this instance variable is inherited
```

2c. If the instance variable is not inherited, delete it

2d. If the instance variable is inherited, reconstrain the instance variable to the superclass's constraint

3. For each class, add new or reconstrained instance variables to the database definition

3a. Find an instance variable in the schema definition that is not in the database definition

3b. Check all subclasses to see if this instance variable makes a subclass's instance variable become inherited

3c. If so, delete the subclass's instance variable

3d. Add the new instance variable

3e. Reconstrain any subclass's instance variables that were removed in step 3c.

4. External References

As mentioned previously, the GS Designer provides an operation to import classes that are defined outside the schema. This provides a way to bring classes defined in other schemas or by other tools under the control of the GS Designer. However, importing external classes leads to some problems in handling external references. When a class is imported into a schema, it is possible that the class references other classes not in the schema. One solution is to import the referenced class also, but this class may also reference a class outside the schema and so on. Unless some discipline is placed on how external classes are imported into the schema, a potentially large number of classes may be imported. This is unnecessary if all the user wishes to do is to reference a single class, for example, to constrain an instance variable.

The GS Designer solves this problem by giving the user control over how deeply nested the import operation is invoked. The user is allowed to import a class whose definition with respect to the GS Designer is 'incomplete', i.e. the class is imported although not all of its instance variables (and their constraining classes) may be included or displayable. When a class is imported incompletely, the class is read only; only complete classes may be modified. The GS Designer provides three levels of nesting for the import operation, as follows:

- Import the class only. The class definition in the GS Designer may be incomplete if it references classes not in the schema. This operation is useful if the user only plans on referencing the imported class.

- Import the class and its connections. The class definition is guaranteed to be complete. If it references external classes, those will be imported as 'class only' as described above. This operation is useful if the user plans on modifying the imported class.

- Import the class and its network. The class and all other classes imported will be complete. This operation imports the transitive closure of all classes and their references. This is useful if the user wants to bring a number of inter-related classes under the control of the GS Designer.

In all of the operations above, importing a class will implicitly import its superclass at the same nested level. The GS Designer also supports promoting incomplete definitions by providing a 're-import' operation. This allows an incomplete class definition to be changed to a complete definition by importing its external references. Two operations are provided to promote an incomplete class: re-import with connections, and re-import with network. These operations work on incomplete classes already in the schema and will import their referenced classes according to the semantics described above.

Though the import operation is both useful and flexible, problems can arise when the schema designer has only a partial definition of a class available to it (i.e., when a class within the schema references a class outside the schema or vice versa). An example of this occurs when a class within the schema has a subclass created in another schema by a different user of GS Designer, or through another interface to the database (such as the C language interface, Smalltalk interface, or the command line interpreter interface). In this situation, the GS Designer cannot recognize name conflicts and other errors between the superclass and its external subclasses until the superclass's modifications are updated in the database definition and the compiler determines the conflict. GS Designer then informs the user that an error resulted from an external class definition. The error message gives the name of the external class that caused the error, and also suggests that the user re-import the class into the schema to avoid future errors of the same type.

The import feature can also give rise to a deadlock situation. This problem is best illustrated with an example. Figure 7 shows two class graph windows for two different users of the GS Designer (i.e. two different sessions interacting with the database). The background window shows user 1's schema with class **Person** instantiated in the database. The foreground window shows user 2's schema which also contains class **Person**. Note, however, the class is locked by user 1 and is therefore not editable by user 2. (GS Designer locks class definitions when the schema is loaded, and doesn't allow modifications to classes with subclasses locked by another user). Figure 8 shows that user 2 has created the class **Employee** as a subclass of **Person** in the database. At this point, user 1's schema does not include class **Employee**, and name conflict errors could result, as described earlier. The deadlock situation is illustrated in Figure 9, when user 1 imports the subclass **Employee** into his/her schema. Once the subclass is imported, the GS Designer recognizes that class **Employee** is locked

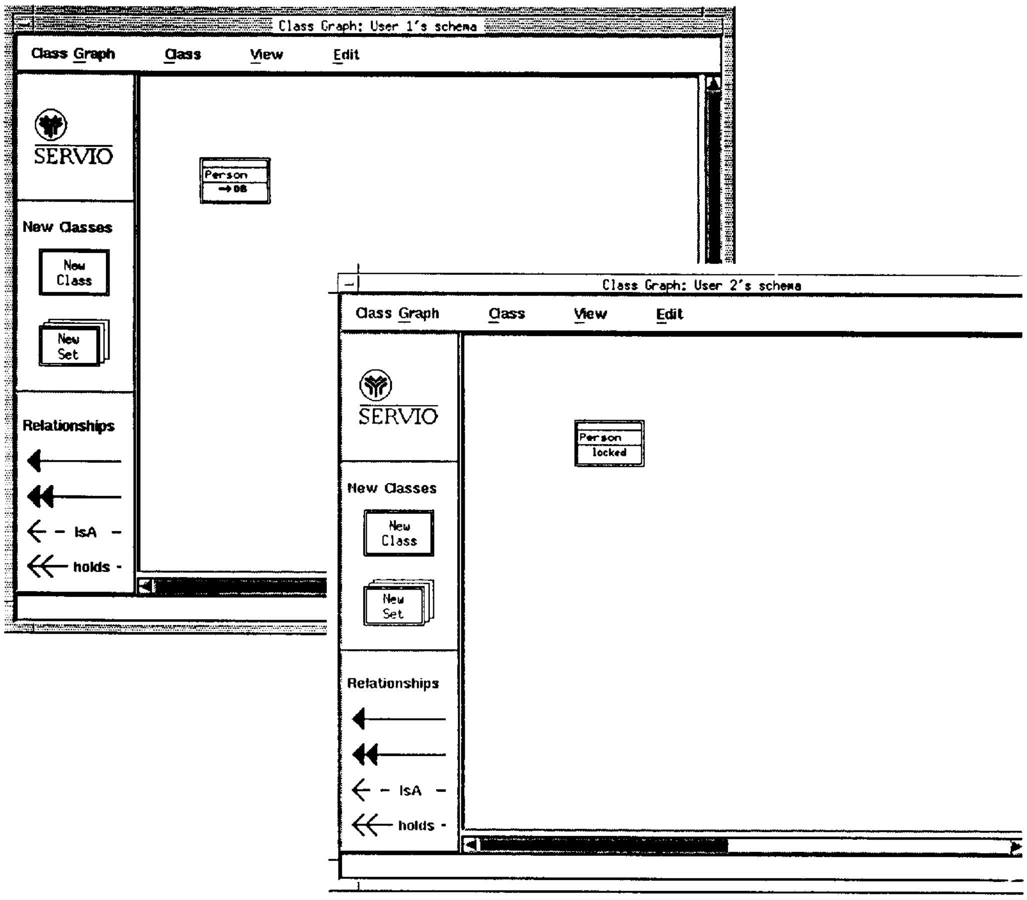


Figure 7.
Two class graphs with the same class for two different users

by another user and disallows any modification of class **Person**. This is appropriate behavior because a change in the superclass may affect the subclass, and the subclass is currently locked by user 2. Importing **Employee** eliminates the possibility of name conflict errors as described earlier; however, no modifications may be made to **Person** by either user until one of the users exits his/her schema.

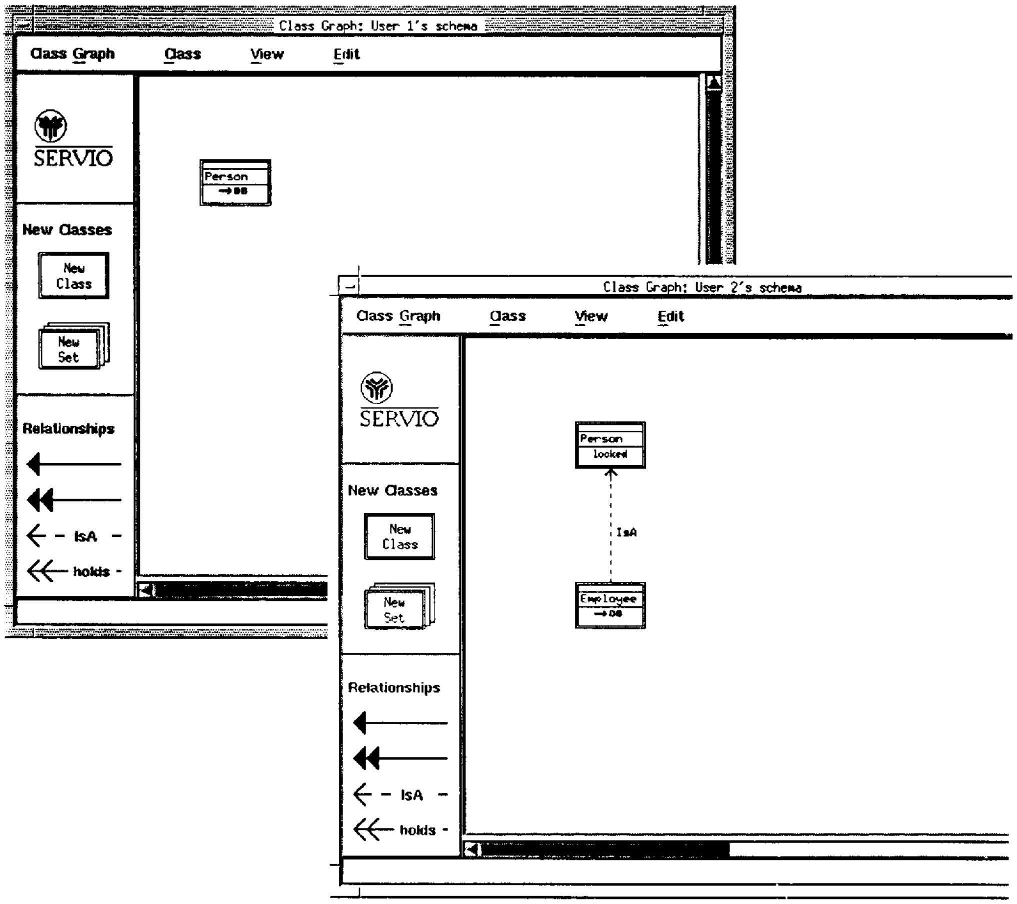


Figure 8.
User 2 creates a subclass

5. Concurrency Conflicts

As mentioned previously, the GS Designer acquires locks to prevent concurrency conflicts between multiple uses of the system. An important issue is how much of a class definition to lock without limiting other users. This is especially important in database management systems, such as GemStone, that use optimistic concurrency control (as described in [Maier et al. 86]) as well as locking. With this mechanism, concurrency conflicts are recognized at commit

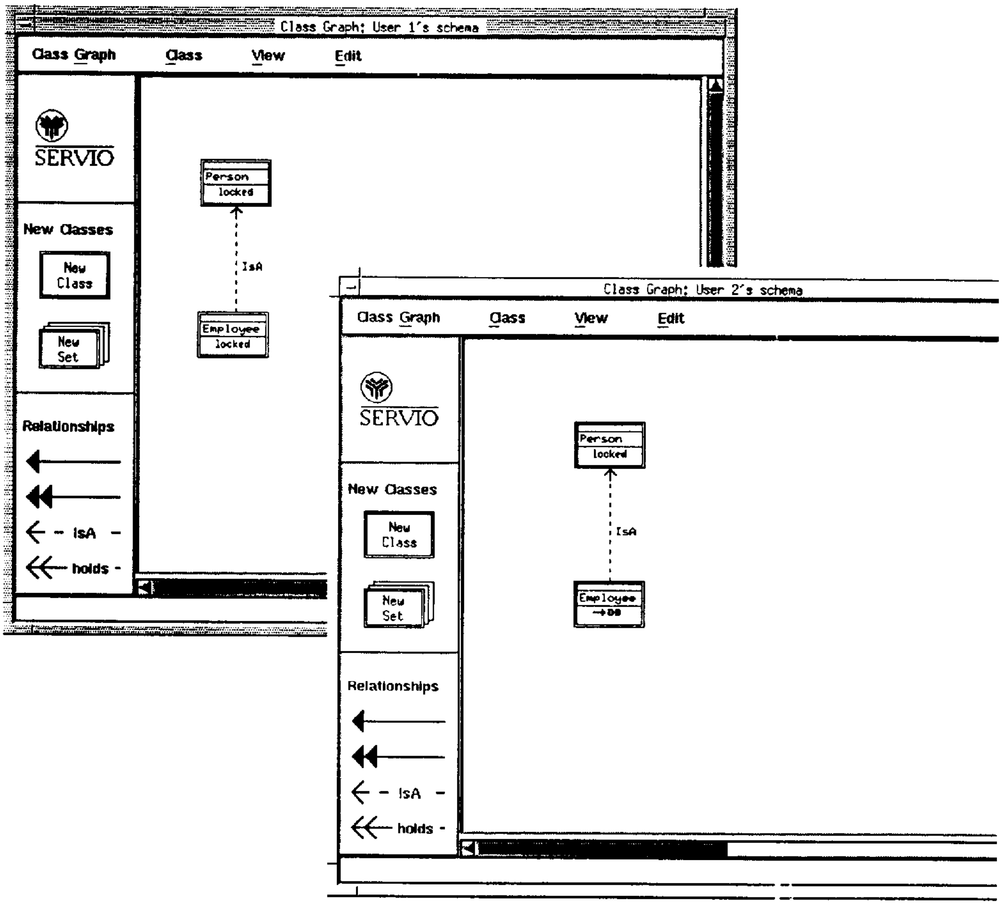


Figure 9.
User 1 imports the subclass

time if an object that was read or written has since been modified by another user. Although the GS Designer locks class definitions, schemas, and dictionaries that may be updated by the user, it is still possible for the operation that updates class definitions in the database to be unable to commit due to optimistic concurrency control. This is because the class creation and modification methods may alter nested objects that are not locked. The GS Designer could lock these nested objects, but doing so would limit other users. An example illustrates how this might happen.

Figure 10 shows two class graph windows belonging to two different users interacting with the database. Both class graphs contain **Class1**, with user 1 prevented from making any modifications because user 2 acquired the lock on the class first. The two class graphs also show that each user has specified a subclass of Class1, but has not created the class definition in the database yet. When a class is created in the database, it updates a class variable in the superclass. This class variable maintains a set of all subclasses of the class, and is a nested object referenced by the superclass definition. In this case, whichever user creates the subclass first will cause the other user to be unable to commit because the set of subclasses will have been modified by another user since this transaction began.

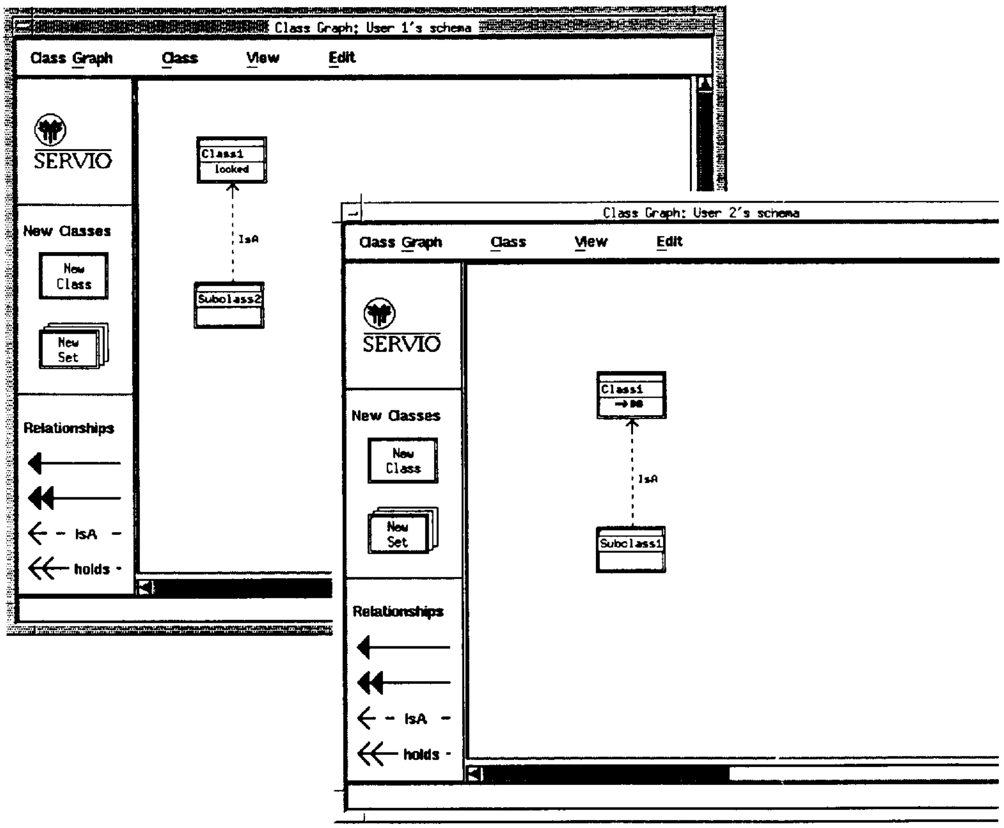


Figure 10.
Both user 1 and user 2 create subclasses of the same class

One solution to the problem would be to lock all nested objects that may be modified by other users. However, we felt this would be too limiting. As demonstrated in the previous example, if the set of subclasses were locked by user 2, then user 1 or any other users would not be able to create a subclass of **Class1**. The solution that is used by the GS Designer is to abort the current transaction when an optimistic concurrency control conflict occurs and then retry the operation again. Aborting the current transaction refreshes the database cache so that any modifications that occurred are now visible to the user. The GS Designer reduces this possibility by locking all class definitions, the schema, and the dictionary from which the schema was read. Locking the dictionary guarantees that the user will always be able to save the schema (i.e. no work is lost).

6. Conclusion

This paper has discussed some of the issues that were addressed in the first release of the GS Designer. We have shown how the order of class modifications can lead to problems unless performed in the correct order, and given an algorithm of how the GS Designer performs multiple class modifications in the database. We have also shown how references to classes external to the schema can lead to problems when multiple designers are updating the database, and discussed the *import* operations to support incomplete definitions in the schema. Finally, we have illustrated problems with locking and optimistic concurrency control when subclasses are created by multiple users in different database sessions.

The emphasis throughout the design and implementation of the tool has been on providing intuitive, consistent operations for the user without limiting other users of the database. The tool is intended to aid in the sharing of class definitions as well as in their creation and modification. To this end, we expect to add a number of additional features in the next release. We expect to integrate the GS Designer with a database administration tool that will allow the user to set authorizations and to easily move objects to different name spaces. In the future, we also expect to couple the GS Designer with a forms package and a query tool, so that users will be able to specify indices and query the database graphically. With the increased complexity of inheritance, behavior, and complex objects in the database, building such tools for an object-oriented database management system will provide a challenge to all database researchers and implementors.

7. Bibliography

Almarode, J., Anderson, L. "GemStone Visual Schema Designer: A Tool for Object-Oriented Database Design", IFIP TC2 Working Conference on Database Semantics, Windermere, UK, 1990.

Albano, A., L. Alfo, S. Coluccini, R. Orsini. "An Overview of SIDERIUS: A Graphical Database Schema Editor for GALILEO", Proc. of the Int. Conf. on Extending Database Technology - EDBT '88, Venice, Italy, 1988.

Bryce, F., R. Hull. "SNAP: A Graphics-based Schema Manager", Proc. of the IEEE 2nd Int. Conf. on Data Engineering, Los Angeles, 1986.

Goldman, K. J., S. A. Goldman, P. C. Kanellakis, S. B. Zdonik. "ISIS: Interface for a Semantic Information System", Proc. of ACM-SIGMOD 1985 Int. Conf. on Management of Data, Austin, Texas, 1985.

Kim, H. J. "Issues in Object-Oriented Database Schemas", The University of Texas at Austin, PH.D. 1988.

Kuntz, M., R. Melchert, "Ergonomic Schema Design and Browsing with More Semantics in the Pasta-3 Interface for E-R DBMSs", 8th Int. Conf. on Entity-Relationship Approach, Toronto, Canada, 1989.

Leong, M.,S. Sam, D. Narasimhalu. "Towards a Visual Language for an Object-Oriented Multi-Media Database System", Proc. of the IFIP TC2/WG 2.6 Working Conf. on Visual Database Systems, Tokyo, 1989. Published as Visual Database Systems, Elsevier Science Publishers B .V, 1989.

Maier, D., J. Stein, A. Otis, A. Purdy. "Development of an Object-Oriented DBMS", OOPSLA '86.