A Software Tool for Optimal Class-Hierarchy Design Based on the Use-Relationship Among Functions and Variables

Nigel Gwee†
Computer Science Department
Southern University and A&M College
Baton Rouge, LA 70813, USA
nigel@cmps.subr.edu

Sukhamay Kundu
Computer Science Department
Louisiana State University and A&M College
Baton Rouge, LA 70803, USA
kundu@csc.lsu.edu

Abstract

We present a software tool for creating an optimal class-hierarchy from the use-relationship among data-items and functions based on the method in Kundu and Gwee [3]. The tool determines the classes and their inheritance relationship, including the assignment of variables and functions to the various classes and their appropriate access levels (private, etc.). We define a class-design language for modeling the classes and the use-relationship and a script language for manipulating the classes. The input to the tool and its operations are based on the constructs of these languages. We illustrate the tool with a small non-trivial class design problem.

1 INTRODUCTION

Although various metrics have been suggested [1, 2] to measure the quality of a class hierarchy, a general definition of an optimal class hierarchy remains elusive: an optimal class hierarchy for software efficiency may not necessarily be optimal for maintainability. Kundu and Gwee [3] consider a class hierarchy to be optimal if it minimizes unnecessary access of functions to data and to other functions, and if it avoids unduly long chains of single inheritance classes. Based on this, they presented a method to create an optimal class hierarchy design from a given use-relationship among a set of functions and data-items (which includes the inputs and outputs of those functions). Their method first extracts a partial ordering of the functions from the use-relationship and thereby creates an initial class hierarchy, and then uses a series of operations to improve the class hierarchy via refactoring and decomposition of classes and merging of variables and functions [5].

We present here a tool called OCHD (Optimal Class Hierarchy Designer) to demonstrate the feasibility of the method in [3]. We illustrate the tool with a small but non-trivial class design problem. The input to the tool OCHD can be a use-relationship or an arbitrary class hierarchy specified by the class-design language; in the second case, the tool first derives a use-relationship from the class hierarchy. The tool can be used interactively to manipulate the classes or run in batch mode, with the operations specified in the script language.

A given class hierarchy H can have several faults with respect to the design specification described by a use-relationship. For example, H may fail to satisfy the basic properties: (1) It uses exactly the same functions and variables as given in the use-relationship: (2) It allows each function to access the variables it needs via the inheritance relationship in H. The OCHD tool can detect these faults in a given class hierarchy against a use-relationship. Moreover, it can show the best placement of missing variables and functions within the existing classes in H, or add new classes to the existing class hierarchy to accommodate them, or create a new class hierarchy from scratch. Our tool can also facilitate the integration of a group of class hierarchies designed separately (not illustrated in this paper owing to space limitations).

2 OUTLINE OF THE PARTIAL ORDER METHOD

We briefly describe the method for creating a class hierarchy from the partial ordering of functions given in [3]. Given a finite set of functions F, a finite set of variables V, and a *use-relationship* $U(F, V)$, where $U = \{\langle f, v \rangle : f \in F \text{ and } v \in V, f \text{ reads or writes } v\}$, let $V(f_i) = \{v_j : \langle f_i, v_j \rangle \in U(F, V)\}$, and let $[f_i]$ be the equivalence class of all $f_j \in F$ for which $V(f_i) = V(f_j)$. We define a partial ordering on F as follows: $f_i < f_j$ if and only if $V(f_i) \supset V(f_j)$. Clearly, this gives a partial ordering on the equivalence classes $[f_i]$. We can now define a class C_i for each $[f_i]$, with C_i having member variables $V(C_i) = V(f_i) - \cup\{V(f_j) : f_i < f_j\}$ and member functions $F(C_i) = [f_i]$. We also make C_i a subclass of C_j if $f_i < f_j$.

If we are given a class hierarchy CH, say in C++ style, then $F = \{ f_i : f_i \text{ is a function in some class in } CH\}$ and similarly for V. We determine $U(F, V)$, or equivalently, each $V(f_i)$, as follows: for each f_i, we initialize $V(f_i)$ to be the variables directly used by f_i (from the source code of f_i). Then, we expand each $V(f_i)$ by letting $V(f_i) = V(f_i) \cup V(f_j)$, the union taken over for all f_j directly called by f_i. Finally, we repeat this expansion step until none of the $V(f_i)$'s change. Observe that for functions f_i that are higher up in CH the final value of $V(f_i)$ will tend

†This work is partially supported by the Louisiana Board of Regents under Contract Number LEQSF (2006-07)-ENH-TR-64 awarded to Southern University and A&M College, Baton Rouge, Louisiana.

to be determined before those of functions that are lower down in *CH*.

The above computations of *F*, *V*, and *U*(*F*, *V*) are essentially step (1) of the following algorithm to optimize *CH*. Steps (2)–(4) build an initial class hierarchy that may contain some redundant artifacts, which are eliminated in steps (5)–(7).

Algorithm: Build Optimal Class Hierarchy (BOCH)
Input: A class hierarchy *CH*, specified using the language CHSL given in section 3.
Output: An optimal class hierarchy *OptCH* consistent with *CH*.

1. [Preprocessing] Extract *F*, *V*, and *U*(*F, V*) from *CH*.
2. [Build function equivalence classes] Collect all functions f_i from *F* and partition them into equivalence classes based on *U* (*F*, *V*).
3. [Build classes from function equivalence classes] For each equivalence class, create a class containing the functions of that equivalence class and the variables these functions use (read, write, or read-write).
4. [Establish class-subclass relationship] For all pairs of functions f_i and f_j, if $f_i < f_j$ then make C_i a subclass of C_j.
5. [Remove subclass transitivity] Form transitive reduction of the acyclic class-subclass relationship.
6. [Prune instance variables] For each class C_i and for each variable v_j in C_i, remove v_j from C_i if v_j is in some parent class C_k of C_i. (Note that this does not eliminate duplicate variable occurrences in classes unrelated by "<".)
7. [Resolve remaining occurrences of duplicate variables by use of referencing] For each such variable v_i, arbitrarily choose one occurrence, say in class C_i, as the primary and treat each of the other occurrences of v_i as a reference to the primary occurrence. (The choice of C_i is left to the user)
8. [Merge classes] For all pairs of classes C_i and C_j, if C_i is a unique subclass of C_j and C_j is a unique parent class of C_i, then merge C_i and C_j into a single class C_k.

3 Class Hierarchy Specification Language CHSL

The class hierarchy input to the tool OCHD is specified by CHSL, which is described below using an extended BNF grammar. CHSL specifies classes in an abstract form using some of the data types and other concepts from C++ (one can easily modify or extend CHSL to include the data types and concepts of other object-oriented languages). For example, we specify a method by only giving its return type, its parameters, the variables used, and the methods called by it, without specifying the method's body. This suffices for determining an optimal class hierarchy.

```
‹class hierarchy› := ‹hierarchy name› ‹newline›
        ‹num classes› ‹newline›
        ‹class›^n Predicate: n = Value (‹num classes›)

‹hierarchy name› := ‹identifier›
‹identifier› := ‹alpha› | ‹alpha› ‹alphanumeric›^+
‹alpha› := a | b | ... | Z
‹alphanumeric› := ‹alpha› | ‹digit›
‹digit› := 0 | ‹positive digit›
‹positive digit› := 1..9
‹newline› := '\n'

‹num classes› := ‹positive integer›
‹positive integer› := ‹positive digit›
        | ‹positive digit› ‹non-negative integer›
‹non-negative integer› := 0 | ‹positive integer›

‹class› :=  ‹class name› ‹newline›
        ‹num parent classes› ‹parent class name›^m ‹newline›
                Predicate: m = Value (‹num parent classes›)
                AND names must have been previously defined
        ‹num instance variables› ‹newline›
        ‹instance variable declaration›^p
                Predicate: p = Value (‹num instance variables›)
        ‹num methods› ‹newline›
        ‹method›^q ‹newline›
        Predicate: q = Value (‹num methods›)

‹class name› := ‹identifier›
‹num parent classes› := ‹non-negative integer›
‹parent class name› := ‹identifier ›
        Predicate: ‹identifier› must have been previously defined

‹num instance variables› := ‹non-negative integer›
‹instance variable declaration› := ‹access level› ‹variable declaration› |
        ‹access level› ‹reference declaration›
‹access level› := private | protected | public
‹variable declaration› := ‹cit› ‹variable name›
‹reference declaration› := reference ‹class::variable name›
‹cit› := ‹constness› ‹indirection› ‹type›
‹constness› := ε | const
‹indirection› := ε | pointer | constPointer
‹type› := void | bool | char | short | int | long | float | double | ‹user-
defined type›
‹user-defined type› := ‹class name›
        Predicate: ‹class name› must have been previously defined
‹variable name› := ‹identifier›

‹num methods› := ‹non-negative integer›
‹method› := ‹method declaration›
        ‹num variables used› ‹class::variable name›^r ‹newline›
        ‹num methods used› ‹class::method name›^s ‹newline›
                Predicate: r = Value (‹num variables used›)
                Predicate: s = Value (‹num methods used›)
‹method declaration› := ‹access level› ‹abstractness› ‹cit›
        ‹method name› ‹num parameters› ‹parameter declaration›^u
                Predicate: u = Value (‹num parameters›)
‹abstractness› := ε | abstract
‹method name› := ‹identifier›
‹num parameters› := ‹non-negative integer›
‹parameter declaration› := ‹variable declaration›

‹num variables used› := ‹non-negative integer›
‹num methods used› := ‹non-negative integer›

‹class::variable name› := ‹class name›::‹variable name›
        Predicate: ‹class name› must have been previously declared and
        ‹variable name› must be from that class
‹class::method name› := ‹class name›::‹method name›
      Predicate: ‹class name› must have been previously declared and
      ‹method name› must be from that class
```

Note that we express reference declarations in CHSL differently from C++; the rationale for this will become clear later (figs. 5 and 6). Also, the determination of *V* in the use-relationship *U*(*F*, *V*) is now slightly different: a reference declaration "reference *C*::*v*" in any class is simply an alias of the variable *v* in class *C* and does not contribute to *V*, nor to any *V*(f_i). The non-terminal symbol ⟨class::variable name⟩ in the rule for ⟨reference declaration⟩ stands for the variable referenced. The non-terminal symbol ⟨class::variable name⟩ in the rule for ⟨method⟩ stands for a variable directly used by ⟨method name⟩ in the ⟨method declaration⟩. Similarly, the symbol ⟨class::method name⟩ stands for a method directly called by the same ⟨method name⟩ above.

Example 1. Shown below is a description in CHSL of a class hierarchy containing a single monolithic class with all the variables and functions (methods) of a small but non-trivial design problem considered in [3], involving a door with a lock and several operations on it. Despite only a few variables and functions to consider here, one can still design several alternative class hierarchies that look reasonable at first sight. A thorough analysis, however, reveals that there is only one optimal design [3].

The sets F, V, and $V(f_i)$ for $f_i \in F$ in this design problem are given by

F = {Lock, Unlock, Close, Open}
V = {lockedStatus, origKey, openClosedStatus}
V(Lock) = {lockedStatus, origKey, openClosedStatus}
V(Unlock) = {lockedStatus, origKey}
V(Close) = {openClosedStatus}
V(Open) = {openClosedStatus, lockedStatus}

We use the convention of class names starting with ‘C_’ and member variable names starting with ‘f’.

```
ch // class hierarchy name
1 // number of classes
C_MonolithicClass // class name
    0 // number of parent classes and their names (if any)
    3 // number of member variables
    protected bool fLockedStatus
    protected const int fOrigKey
    protected bool fOpenClosedStatus
    4 // number of methods
    public void Unlock 1 int key // class declaration and parameters
        2 // number of variables directly used
        C_MonolithicClass ::fLockedStatus
        C_MonolithicClass ::fOrigKey
        0 // number of methods directly called
    public  void Close 0
        1
        C_MonolithicClass ::fOpenClosedStatus
        0
    public  void Open 0
        2
        C_MonolithicClass ::fLockedStatus
        C_MonolithicClass ::fOpenClosedStatus
        0
    public  void Lock 1 int key
        3
        C_MonolithicClass ::fLockedStatus
        C_MonolithicClass ::fOrigKey
        C_MonolithicClass ::fOpenClosedStatus
        0
```

Fig. 1 shows the corresponding class. In the interests of space, we have omitted some information and shown the data in a more compact form. Clearly, this class structure is undesirable, e.g., Unlock() has unnecessary access to the variable fOpenClosedStatus. This and other shortcomings are addressed in Examples 2–4.♦

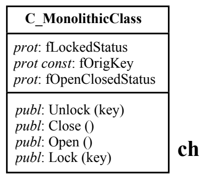


Fig. 1. Specification of functions and variables in Example 1 as a single monolithic class.

4 CLASS HIERARCHY OPTIMIZATION SCRIPTING LANGUAGE CHOSL

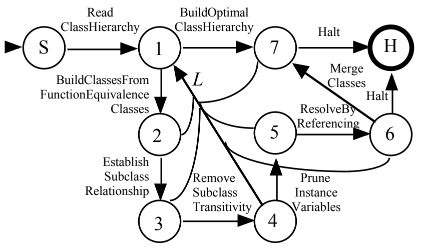
CHOSL gives the user control of the class design process. The user can selectively invoke the various optimizing operations and observe their effects. Immediate feedback on intermediate operations can be useful in indicating certain weaknesses in the current class hierarchy, and the need for redefining one or more methods (and variables) to simplify the class hierarchy.

CHOSL is partially specified by the following grammar, showing only a subset of the various operations. The operations shown are directly related to the steps of the algorithm BOCH in section 2.

```
‹script› := ‹operations› ‹newline› ‹halt› ‹newline›
‹operations› := ‹operation› | ‹operation› ‹newline› ‹operations›
‹operation› :=  ‹build optimal class hierarchy›
        | ‹build classes from function equivalence classes›
        | ‹establish subclass relationship› | ‹remove subclass transitivity›
        | ‹prune instance variables› | ‹resolve by referencing›
        | ‹merge classes›
        …    …    …
‹build optimal class hierarchy› := BuildOptimalClassHierarchy
        ‹source class hierarchy name› ‹destination class hierarchy name›
‹build classes from function equivalence classes› :=
        BuildClassesFromFuncEquivClasses
        ‹source class hierarchy name› ‹destination class hierarchy name›
‹establish subclass relationship› := EstablishSubclassRelationship
        ‹source class hierarchy name› ‹destination class hierarchy name›
‹remove subclass transitivity› := RemoveSubclassTransitivity
        ‹source class hierarchy name› ‹destination class hierarchy name›
‹prune instance variables› := PruneInstanceVariables
        ‹source class hierarchy name› ‹destination class hierarchy name›
‹resolve by referencing› := ResolveByReferencing
        ‹source class hierarchy name› ‹destination class hierarchy name›
‹merge classes› := MergeClasses
        ‹source class hierarchy name› ‹destination class hierarchy name›
‹halt› := Halt
‹source class hierarchy name› := ‹alphanumeric string›
‹destination class hierarchy name› := ‹alphanumeric string›
```

OCHD builds an internal representation (see section 5) of the input class hierarchy from its CHSL specification. The next operation ⟨build optimal class hierarchy⟩ invokes a series of sub-operations ⟨build classes from function equivalence classes⟩, …, ⟨merge classes⟩, which correspond to steps (3)–(8) of algorithm BOCH, and finally, the operation ⟨halt⟩. The semantics of these operations are explained and illustrated in Examples 2–4 below. The sub-operations could be invoked in various combinations on class hierarchies created previously. Operations not shown above include refactoring functions (MoveInstanceVariable and MoveMethod), writing class hierarchies onto an external file (WriteClassHierarchy), and printing class hierarchies onto the screen (Print-ClassHierarchy).

Fig. 2 shows a finite state automaton for typical uses of various operations. The transition *L* indicates a modification to a class or to a class hierarchy. For example, a modification may involve addition or deletion of methods or of variables to an existing class, addition or deletion of a class in a hierarchy, or of a child-parent relationship between some classes in a hierarchy (these operations were omitted in the description of CHOSL). Not shown are transitions to other states beside State 1 that could be made from States 2–7, depending on the nature of the modification (see Examples 3 and 4). Note that, as the automaton indicates, merging of classes is an optional operation. Note also that refactoring of a method can be modeled as a series of additions of methods in CHSL specification, including modifying the list of variables and methods used directly by the refactored functions (see Example 3).



Read ClassHierarchy
BuildOptimal ClassHierarchy
Halt
S
1
7
H
BuildClassesFrom FunctionEquivalence Classes
L
Merge Classes
Halt
ResolveBy Referencing
2
5
6
Establish Subclass Relationship
Remove Subclass Transitivity
Prune Instance Variables
3
4

Fig. 2. Finite-state automaton for operations in CHOSL: S = start-state; H = halt-state.

Example 2. We now illustrate the class hierarchy optimization process by the tool OCHD for the class hierarchy 'ch' in fig. 1. Our goal is to group methods and the variables they use into classes, in order to minimize unnecessary access to variables by various methods.

Fig. 3 shows the classes corresponding to $[f_i]$ using the operation "BuildClassesFromFuncEquivClasses ch ch2". The tool assigns names to the classes according to the names of the functions f_i. The duplications of variables in different classes, e.g., fLockedStatus, are merely redundant artifacts of step (3) of the algorithm, and will be eliminated and/or resolved in subsequent steps.

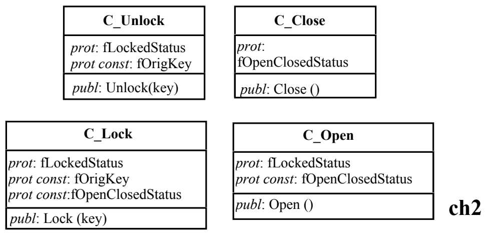


Fig. 3. Classes built from function equivalence classes.

Fig. 4 shows intermediate class hierarchies resulting from the processing of BOCH. Class hierarchy ch3 represents the class-subclass relationship corresponding to the partial ordering "<" using "EstablishSubclass-Relationship ch2 ch3". Note that only some of the duplications of variables have been eliminated. If we view class hierarchy ch2 as four separate but interdependent class hierarchies (each consisting of one class) due to the multiple occurrences of some variables, then ch3 represents an integration of these four hierarchies.

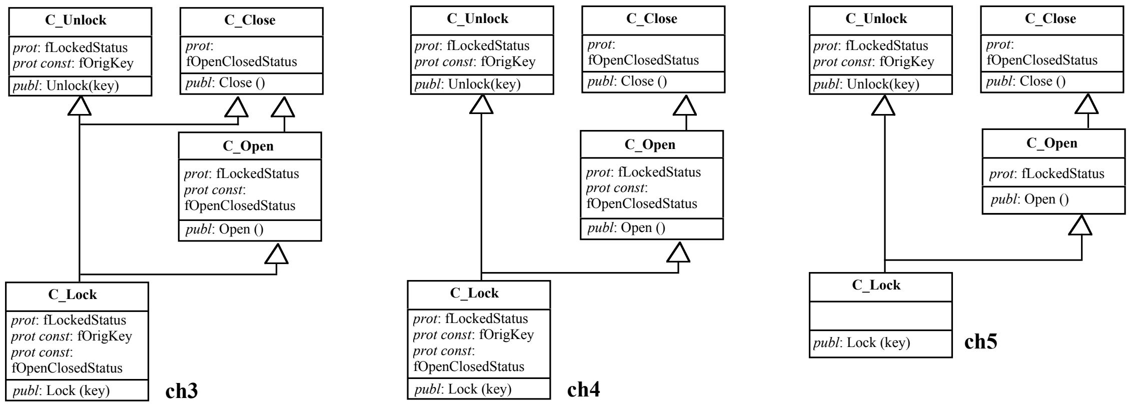


Fig. 4. Intermediate classes **ch3**: Establishment of class-subclass relationship based upon partial ordering of methods; **ch4**: Class hierarchy after removal of class-subclass transitivity; **ch5**: Class hierarchy after pruning of member variables.

The class C_Lock in ch3 appears to have three parent classes, but this is only an illusion: actually the "parent class" C_Close appears here as another processing artifact of the algorithm. Since C_Lock's true parent class C_Open's own parent class is C_Close, we can eliminate the latter as C_Lock's parent class. Eliminating such artifacts using "RemoveSubclassTransitivity ch3 ch4" produces ch4. Next, elimination of variables from classes whose parent(s) contain(s) the same variable in the hierarchy ch3 using "PruneInstanceVariables ch4 ch5" produces ch5.

Finally, fig. 5 shows the result of resolving the duplicate occurrence of fLockedStatus using "ResolveByReferencing ch5 ch6", followed by merging classes C_Close and C_Open, which have a unique parent-child connection, using "MergeClasses ch6 ch6" (here C_Unlock::fLockedStatus is taken as the primary occurrence). This last operation does allow an unnecessary access of the Close() method to fLockedStatus, and was undertaken simply to reduce the number of classes. It is an optional step (as reflected in the finite-state automaton of fig. 2).

Note that if we start with the class hierarchy ch2 (fig. 3) or any of hierarchies ch3–ch5 (fig. 4), OCHD will produce the same optimal class hierarchy ch6 (fig. 5). ♦

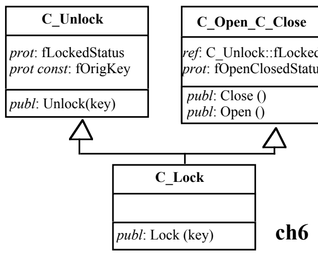


Fig. 5. Class hierarchy after resolving of duplicate variable occurrences and merging of classes with a unique parent-child connection.

To eliminate the referencing of fLockedStatus and to use only subclassing as the means for variable access, we could manually factor out fLockedStatus by creating a separate class to contain it and cause both C_Unlock and C_Open to be subclasses of this new class (Example 3). Alternatively, we could factor out the part of Open() that needs to use fLockedStatus (Example 4).

Example 3. Class hierarchy ch7 (fig. 6) is a new but related class hierarchy to ch5 (fig. 4). This can be created by adding to ch5 the class C_GetLockedStatus, including its member variables and methods, and also the subclass relationship between it and C_Unlock. In C_Open, fLockedStatus is explicitly specified as a reference variable. BOCH will now modify the access means of the variable fLockedStatus by Open() from referencing to subclassing. Class hierarchy ch8 is the optimized class hierarchy obtained using "BuildOptimalClassHierarchy ch7 ch8". The latter can also be obtained "from scratch" as follows. First, add information about the method GetLockedStatus() to the CHSL specification in Example 1, let the list of methods called by Unlock(), Open(), and Lock() consist of the single item GetLockedStatus(), and finally remove fLockedStatus from the list of variables used by those three methods. Let ch1' denote the new monolithic class. We can get ch8 from ch1' by applying "BuildOptimalClassHierarchy ch1' ch8" without going through ch7 and, in particular, without creating any reference declarations to fLockedStatus. We would still get ch8 if we do not remove flockedStatus from the list of variables used by Unlock(), Open(), and Lock(). ♦

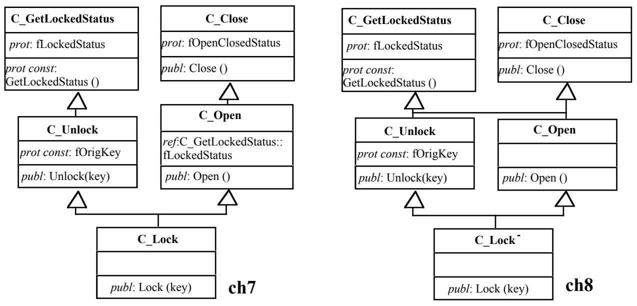


Fig. 6. **ch7**: Class hierarchy after creating a separate class to hold fLockedStatus; **ch8**: Class hierarchy after building new hierarchy from **ch7**.

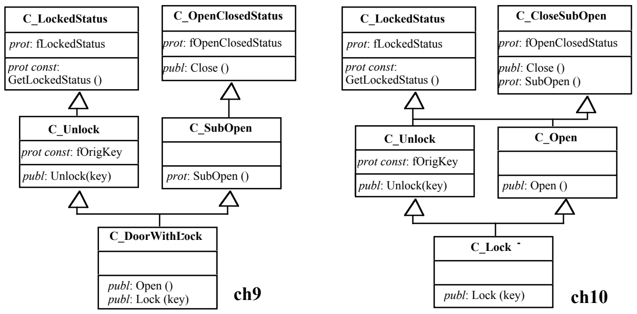


Fig. 7. **ch9**: Class hierarchy with refactoring of Open() to contain a submethod SubOpen(); **ch10**: Class hierarchy after building new hierarchy from **ch9**.

Example 4. In addition to creating a new class to allow use of fLockedStatus, we could refactor the method Open() by isolating into SubOpen() the part that does not require use of fLockedStatus ("Extract Method") [5, p. 110]. Fig. 7 shows this modified class hierarchy (ch9). Note that this class hierarchy departs slightly from a class-subclassing based strictly upon a partial ordering, since Open() never uses fOrigKey, and thus *V*(Open) is not a proper superset of *V*(Unlock). The class hierarchy ch10 is obtained using "BuildOptimalClassHierarchy ch9 ch10". Note its similarity to ch8 (fig. 6). The class hierarchy ch10 can also be obtained by modifying the

CHSL specification of ch1' in Example 3 by adding SubOpen() and specifying that it uses fOpenClosedStatus (but not fLockedStatus) and adding SubOpen() to the list of methods called by Open(). ♦

5 IMPLEMENTATION OF OCHD

We briefly describe some aspects of the software implementation of OCHD. Fig. 9 shows the class hierarchy CHModel which is used in OCHD for modeling the components (instance variables, methods, classes, etc.) of the user's input class hierarchy *CH*. The tool OCHD assigns unique ID's to all components of *CH* in order to identify and retrieve them for analyzing *CH*. Note that the user may use the same variable name in two different classes. The top-level class C_Component in fig. 9 contains data structures common to all other classes. The instance variable fID belongs to a type of class C_ID (not shown) that enables unique ID's to be created.

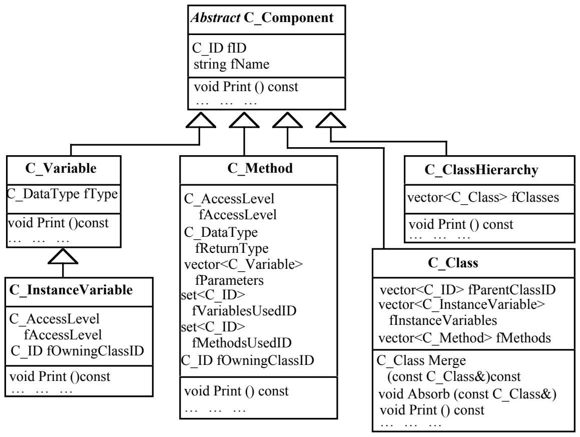


Fig. 9. The class hierarchy CHModel in OCHD to model an input class hierarchy.

The instance variable C_ClassHierarchy::fClasses stores all the classes in *CH*. In turn, C_Class stores its instance variables and methods in vectors of type C_InstanceVariable and C_Method respectively. Additionally C_Class stores its parents in a vector of type C_ID. This pattern of reference to components by their unique ID's can be seen also in C_Method:: fVariablesUsedID. In this case, however, the ID's are stored in a set (the template class 'set') instead of in a vector simply because the procedure used to optimize *CH* employs primarily set operations.

Most of the methods in the above classes are access (and, where feasible, modifying) methods for the instance variables of each class (e.g., C_Component GetID()), and are not shown in the class diagram. A few, however, perform some manipulation, e.g., C_Class::Merge (const C_Class& class2) and C_Class::Absorb (const C_Class& class2); the difference between these two methods is as follows: C_Class::Merge returns a new class that contains the instance variables and methods of both class2 and the class to which this message is given, without modifying the latter, whereas C_Class::Absorb transforms the class to which this message is given by inserting a copy of the instance variables and methods of class2.

The class C_ClassDesignAlgorithms (not shown) contains the optimization algorithm BOCH, which makes use of the methods in CHModel (fig. 9). The methods in C_ClassDesignAlgorithms are declared *static* because they rely solely on attribute data stored in the components of CHModel.

6 CONCLUSION

The optimal class hierarchy design based on the use-relationship as produced by OCHD can provide a useful basis for comparing initial designs by the user. Since the use-relationship does not capture the whole semantics of variables and functions, this optimal design may not be completely satisfactory to the user. Nevertheless, the comparison can highlight certain potential drawbacks that may be present in the initial designs. In Examples 2–4, the presence of extensive multiple inheritance (figs. 4, 6, 7) suggests the need for alternative means of providing access to variables in ancestor classes. Factoring out parts of methods that use those variables into submethods, and using object composition [4, p. 20] are two ways to eliminate multiple inheritance.

One way to enhance OCHD is to include other optimization algorithms that will result in class hierarchies better suited, say, for maintainability. Another possibility is to include special features of object-oriented languages like Java packages.

The interface of OCHD can also be further developed to produce a UML-like graphical view of the class hierarchy. One can also use a graphical interface to input new class specifications.

7 REFERENCES

[1] Choinzon, M., Y. Taki, and Y. Ueda. Quality metrics for object-oriented design assessment. *IEICE Technical Report*, vol. 104, no. 725, pp. 19–24, 2005.

[2] Choinzon, M. and Y. Ueda. Detecting defects in object oriented designs using design metrics. *Proc. 7th Joint Conference on Knowledge-Based Software Engineering*. In *Frontiers in Artificial Intelligence and Applications*. Vol. 140, pp. 61–72, 2006.

[3] Kundu, S. and N. Gwee. A formal approach to designing class-structures using a partial-order. *Proc. 29th Annual International Computer Software and Applications Conference*, 2005.

[4] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.