A Formal Approach To Designing A Class-Subclass Structure Using A Partial-Order on The Functions

Sukhamay Kundu
Computer Sc. Dept, Louisiana State University
Baton Rouge, LA 70803
(kundu@csc.lsu.edu)

Nigel Gwee
Computer Sc. Dept, Southern University
Baton Rouge, LA 70813
(nigel@cmps.subr.edu)

Abstract

We present a formal method for designing the class-subclass structure based on a partial order on the functions, which is derived from the use-relationship between the functions and the data items. We can regard this method as a first step in building a theory of refactoring and design-patterns. Our method can identify the functions which should be factored into subfunctions, including their desired signatures, in order to reduce their use-complexity and thus simplify the class-subclass structure. It can also identify the data items that need to be decomposed or consolidated for simplifying the class-subclass structure. We illustrate our method with several examples.

Keywords: class hierarchy, partial order, refactoring.

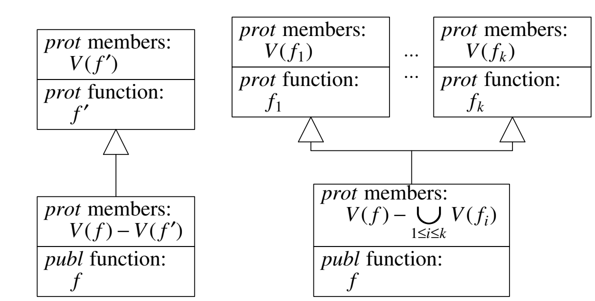
1. Introduction

There have been many advances in object-oriented design over the last decades, which include the development of UML [4], design-patterns [2], and a variety of refactoring techniques [1]. Although refactoring is very useful in improving the design of an existing program and its class structure [3, 7], we still lack a general method for determining the class structure that can be used as a design tool before writing the code. This lack can also be viewed as a main reason for the proliferation of design-patterns, each one addressing a specific type of application software. If we have a general method for class design, then it should be possible to derive the known design-patterns as special cases of such a method.

A key difficulty in designing a class structure comes from the two conflicting needs: (1) keeping the data "close" to the functions which use them to support information hiding, and (2) eliminating the duplication of data and operations. Moreover, it is often the case that we need to decompose the original functions and data (obtained from the requirements, say, as a first step in the design process) in order to meet the two conflicting needs, which complicate the class design task. The notion of refactoring will remain useful even if we have a general class-design method because there will always be a need for incremental rewriting of existing code during the lifetime of a software as we add new functionalities. However, we need a more direct and scientific approach, a science of design, for determining a proper class-subclass structure instead of an evolutionary approach. In [5], the notion of concept lattice is used for grouping the members of a given class according to their use-patterns in order to determine whether the class can be decomposed into two or more disjoint classes. Our work is focused on determining the total class-subclass structure as a whole instead of one class at a time. The fact that a class structure resembles a partial order instead of the more specialized structure of a lattice makes the notion of concept lattice ill suited for determining the whole class-subclass structure.

2. Class-Structure Design

The basic class-subclass relationship fits well with the principle of information-hiding. If a function f uses the data items (variables) $V(f)$ and the computations of f contains a subcomputation which can be isolated (i.e., factored out) as a subfunction f' which uses only a proper subset of data items $V(f') \subset V(f)$, then we can organize f and f' in the form of a class-subclass structure as in Fig. 1(i). Here, both f and f' may have parameters. A more general case is where we can isolate several subcomputations $f_1, f_2, \cdots, f_k$ within f such that $V(f_i) \cap V(f_j) = \varnothing$ for $i \neq j$; in this case, we can organize the functions f_i as in Fig. 1(ii) using multiple parents. The members $V(f_i)$ in the respective classes are shown here as "protected" (as in C++) for the sake of definiteness. One can apply this processes recursively to each f_i.



(i) A simple class structure to isolate a subcomputation f' of f.

(ii) A class structure to isolate several disjoint subcomputations f_i of f.

Figure 1. Forming a class-subclass structure by isolating subcomputations f' (f_i) of a function f.

In non-object oriented programming, the factoring of a subfunction f_i from a function f helps in two ways: (1) it reduces the code size by replacing repeated occurrence of the code for f_i in different parts of the program by calls to f_i, and (2) it simplifies the task of program understanding and makes the program modifications easier. For the class-design, the factoring of f_i from f helps in another way: (3) it reduces dependencies among different classes that are not related by the class-hierarchy (via the use of "friends" and "references", say), by keeping the operations and their related data items close to each other, and thereby simplifying the class structure. Note that the requirement $V(f_i) \subset V(f)$ applies only for (3), and not for (1)-(2), where we may have $V(f_i) = V(f)$.

The primary focus in the design of class-structure are the functions; they either directly correspond to the actions (internal/external behavior) of the software or support (are parts of) those actions. In contrast, for database systems, where the operations are standardized, the data occupy the central position. But in a typical software, the functions are of primary interest and data plays a secondary role to support the functions. If we are given a particular organization of the functions in the form of classes, the placement of various data items can be obtained by following the principle of keeping each data item close to the functions that use it without duplication. This means if a data item v is used by the functions $F(v) = \{f_i: 1 \le i \le k\}$, then place v at the nearest common ancestor class $C_{F(v)}$ of the functions $F(v)$, if exists. If there is no unique nearest common ancestor $C_{F(v)}$, then we can put v in one of the nearest common ancestor classes and use a reference to it from the other nearest ancestor classes. Alternately, we can create a new nearest common ancestor (abstract) class to hold v.

2.1. Why use a partial order

The inheritance property in a class structure can be viewed as a partial order, with different class structures giving different partial orders. More precisely, given a class-subclass structure S, we can define a partial order on the classes C_i in S by $C_i < C_j$ if C_i is a subclass of C_j; it is clear that the binary relation "<" is transitive and antisymmetric. Let $F(C_i)$ be the public and protected member functions in class C_i, and similarly let $V(C_i)$ be the public and protected member variables (fields) in C_i; we ignore the private member fields and functions since they are not accessible to functions in other classes (save the friends). Let $F(S) = \bigcup F(C_i)$ (union over all $C_i \in S$), $V(S) = \bigcup V(C_i)$, $VF(C_i) = V(C_i) \cup F(C_i)$, and finally $VF(S) = V(S) \cup F(S) = \bigcup VF(C_i)$. We can define a partial order on $F(S)$ by $f_i < f_j$ if $f_i \in F(C_i)$, $f_j \in F(C_j)$, and $C_i < C_j$; this means f_i has access to (ability to make use of) a larger set of variables and functions in the other classes than f_j although f_i may actually use only a small part of them. We can similarly define a partial order on $V(S)$ by $v_i < v_j$ if $v_i \in V(C_i)$, $v_j \in V(C_j)$, and $C_i < C_j$; this means v_i can support (can be used by) a smaller set of functions than v_j in the other classes. The above observation suggests that to design a class-structure we need to first determine a partial order among the functions (cf. our previous remarks that the functions are our primary focus). One way to obtain a partial order on the functions is via the use-relationship between the functions and the data items as shown below.

2.2. Use-relationship and partial order

Given a finite set of functions F and a finite set of variables V, let $U(F, V)$ be the *use-relationship*, where $(f_i, v_j) \in U$ mean that the function f_i uses (reads, writes, or both) v_j. We do not distinguish at this point between read-use and write-use although these details are needed to determine, for example, if a reference to a variable should be "constant". Also, let $V(f_i) = \{v_j: (f_i, v_j) \in U(F, V)\} \subseteq V$, the subset of variables used by f_i, and likewise let $F(v_j) \subseteq F$ be the subset of functions which use v_j. We define below an equivalence relation $\approx_U$ on F and a partial order $<_U$ on the resulting equivalence classes.

Def. 1. We define $f_i \approx_U f_j$ (or, simply, $f_i \approx f_j$, when no confusion is likely) if $V(f_i) = V(f_j)$; we write $[f_i]$ for the equivalence class of f_i. We also define $f_i <_U f_j$ (or, simply, $f_i < f_j$) if $V(f_i) \supset V(f_j)$, which gives a partial order on the equivalence classes $[f_i]$. We can similarly define an equivalence relation "≈" on V and a partial ordering "<" on the equivalence classes $[v_i]$ by letting $v_i < v_j$ if $F(v_i) \subset F(v_j)$ □

The simplest case of a use-relationship is one where any two $V(f_i)$ and $V(f_j)$ are either identical or disjoint; this happens, in particular, if each $V(f_i)$ is a singleton set $\{v_k\}$. In this case, the partial order "<" of the functions is empty, i.e., any two functions f_i and f_j are either unrelated by "<" or are equivalent, and the same is true for the variables v_k. At the other extreme, we may have the partial order on $[f_i]$ to be a linear order (which happens if for any two $V(f_i)$ and $V(f_j)$, one of them is a subset of the other). In this case, the partial order on $[v_j]$ is also linear. It is easy to see that the converse holds in both the cases, i.e., a linear order on $[v_i]$ implies a linear order on $[f_i]$ and the empty partial order on $[v_i]$ implies the empty partial order on $[f_i]$. Thus, in the extreme cases, the partial order on the functions does not give additional information than that on the variables. However, it will be seen from the example design problems considered in the next Section that the partial order on the functions are in general more informative. One way to explain this difference is the following. First, if we view $U(F, V)$ as an abstract relation and interchange the rows and columns, that is, let $V' = F$ be an abstract set of variables, $F' = V$ an abstract set of functions, and let $U(F', V') = \{(v_j, f_i): (f_i, v_j) \in$

$U(F, V)\}$, then the partial order on F' obtained from $U(F', V')$ is the same as the partial order on V obtained from $U(F, V)$. This may suggest that the partial order on the variables should be as informative as the partial order on functions. However, this argument is invalid because if f_i is a function of v_j and possibly some other variables, then we cannot say that v_j is a function (mathematically speaking) of f_i although that would be the case if f_i is an one-to-one function of just the variable v_j so that we can regard v_j to represent the inverse function f_i^{-1}. Moreover, the inverse functions (if they exist) typically do not appear in practice as naturally as the functions themselves. (The meaning of $(f_i, v_j) \in U(F, V)$ is also somewhat more complex than the mathematical notion of f_i being a function of v_j because of read-write use.)

Def. 2. We define the *use-complexity* of a function f_i by $UC(f_i) = |V(f_i)|$ and the *use-complexity* of a variable v_j by $UC(v_j) = |F(v_j)|$.

If we factor out a subfunction f_i from a function f as illustrated in Figs. 1(i)-(ii), f_i necessarily needs to have a smaller use-complexity than f. The use-complexity of f, which calls each of the subfunctions f_i, is not affected by the factoring.

2.3. From partial order to class structure

We view the class-structure design from the partial order "$f_i < f_j$" as a two-step process. In the first step, we obtain an initial design of the class structure which parallels the partial order $f_i < f_j$ and thereby captures the rudimentary semantics of the functions as represented by the use-relationship $U(F, V)$. To be precise, we define one class C_i for each equivalence class $[f_i]$ and let C_i be a subclass of C_j ($C_i < C_j$, in symbols) if $f_i < f_j$; the member variables of C_i are given by $V(C_i) = V(f_i) - \bigcup\{V(f_j)\colon f_i < f_j\}$ and the member functions of C_i are given by $F(C_i) = [f_i]$. The second step involves simplifying and improving the initial design by the factoring of some of the functions and redefining some of the variables, if possible. This step requires deeper insights into the meaning of the functions and cannot be possibly fully automated like the first step. We present below two detailed examples to illustrate the second step, where both examples use factoring and the second example use the redefinition (more specifically, consolidation) of variables to simplify the initial design.

The initial class design C_i indicated above already has an important "cohesion" property [6] for its member variables (and functions) that is worth mentioning. Let $\bar{V}(C_i) = \bigcup\{V(C_j)\colon C_i \le C_j\} = V(f_i)$, the set of variables in V which are accessible to the functions $[f_i]$ in class C_i. Suppose we define a graph $G(V)$ on the variables V in $U(F, V)$ as follows: two nodes (variables) v_i and v_j are adjacent if for some f_k, we have both $v_i, v_j \in V(f_k)$. Put another way, G is the union of the complete graphs on the vertex sets $V(f_k)$. We call a subset $V' \subseteq V$ *connected* if the subgraph of G on V' gives a connected graph. It is clear that $V' = V(f_i)$ is connected and, more generally, each $V(C_i)$ is connected. On the other hand, if $V(f_i) \cap V(f_j) = \emptyset$ and there is no f_k such that $V(f_k)$ intersects both $V(f_i)$ and $V(f_j)$, then $V(f_i) \cup V(f_j)$ is not connected. We summarize this observation below.

Theorem 1. For the class structure C_i defined from the partial order $f_i < f_j$, both the member variables $V(C_i)$ in C_i and the member variables $\bar{V}(C_i) = V(f_i)$ in its ancestor classes form a connected (cohesive) set.

We briefly remark that if we try to define the classes C_i' in a similar way based on the partial order $v_i < v_j$, with the member variables in C_i' given by $V(C_i') = [v_i]$, we may not be able to put a function f_j in any C_i'. This happens if there is no unique highest common descendent of the classes C_i' which contains f_j. If we introduce new subclasses for a highest common descendent to put f_j, then we basically get back the same result as that given by the partial order of the functions.

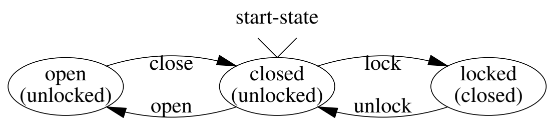
3. Class Design Application

We present now two class design problems of different complexities. The first one illustrates the basic use of the partial order and the resulting feedbacks about the need to factor some of the functions to simplify the class structure. The second example illustrates a more complex issue of redefining (consolidating) different variables with identical domains to facilitate factoring and using the factored part in different places by introducing parameters. The classes-structure obtained by our method is superior to those obtained by intuitive and adhoc approaches.

3.1. First design problem

Consider the problem of designing the classes for the operations {open, close, lock, unlock} of a door with a lock. Assume that a door can be locked only if it is closed. The finite-state model in Fig. 2(i) shows which operation can follow which other operations; for example, the open-operation can follow only a close or an unlock-operation. We assume that a door is delivered by the manufacturer in the closed-unlocked position for the safety purpose, and thus we let the state "closed (unlocked)" in Fig. 2(i) to be the start-state. Assume that a door can be unlocked from either side, irrespective of the side from which it was locked. We show that the additional operations: (1) extendedOpen, which unlocks the door first if it was locked and then opens the door, and (2) extended-Lock, which first closes the door if it was open and then locks the door, do not affect the final class-design.

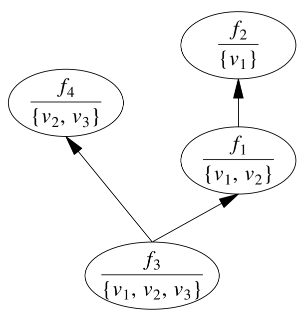
Fig. 2(ii) shows the variable-use relationship, where v_1 = openClosedStatus has values in {open, closed}, v_2 =



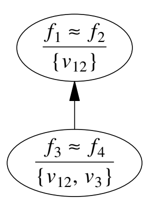
(i) The state-diagram for a door-with-lock; the state "locked" would be absent for doors without a lock.

Variable-use relationship	f_1: Open	f_2: Close	f_3: Lock	f_4: Unlock	f_5: Ext-Open	f_6: Ext-Lock
v_1: open-ClosedStatus	x	x	x		x	x
v_2: locked-Status	x		x	x	x	x
v_3: origKey			x	x	x	x

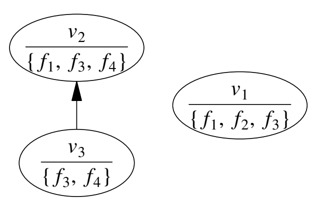
(ii) The variable-use relationship for the door-operations; $f_3 \approx f_5 \approx f_6$ and $[v_i] = \{v_i\}$ for each i.



(iii) The partial order $f_i < f_j$; also shown in each node is $V(f_i)$.



(iv) The result of merging v_1 and v_2 into a single variable v_{12}.



(v) The partial order $v_i < v_j$; also shown in each node is $F(v_i)$.

Figure 2. The state-diagram, use-relationship, and the partial-orders for functions and variables for a door.

lockedStatus has values in {locked, unlocked}, and v_3 = origKey represents the key supplied by the door manufacturer. We assume that lock and unlock operations involve the use of a key, which must match origKey in order to succeed. Fig. 2(iii) shows the partial order $f_i < f_j$ and Fig. 2(iv) shows the modified partial order if we replace the variables $\{v_1, v_2\}$ by a single variable v_{12} = openCloseLockedStatus, taking values in {open, closed, locked} corresponding to the states in Fig. 2(i). The considerable loss of detail in the partial order in Fig. 2(iv) is an indication that v_1 and v_2 should not be merged; we will show that the classes obtained from Fig. 2(iv) are not as good as those obtained from the partial order in Fig. 2(iii). Fig. 2(v) shows the partial order $v_i < v_j$ based on the use-relationship in Fig. 2(ii); it too has fewer details compared to Fig. 2(iii) and this is in agreement with our previous remark that the functions should be our focus in designing class-structures. For the variables $\{v_{12}, v_3\}$, we get the partial order $v_3 < v_{12}$ instead of Fig. 2(v); the fact that this is structurally identical to Fig. 2(iv) (and is linear) is not an indication that v_1 and v_2 should be merged.

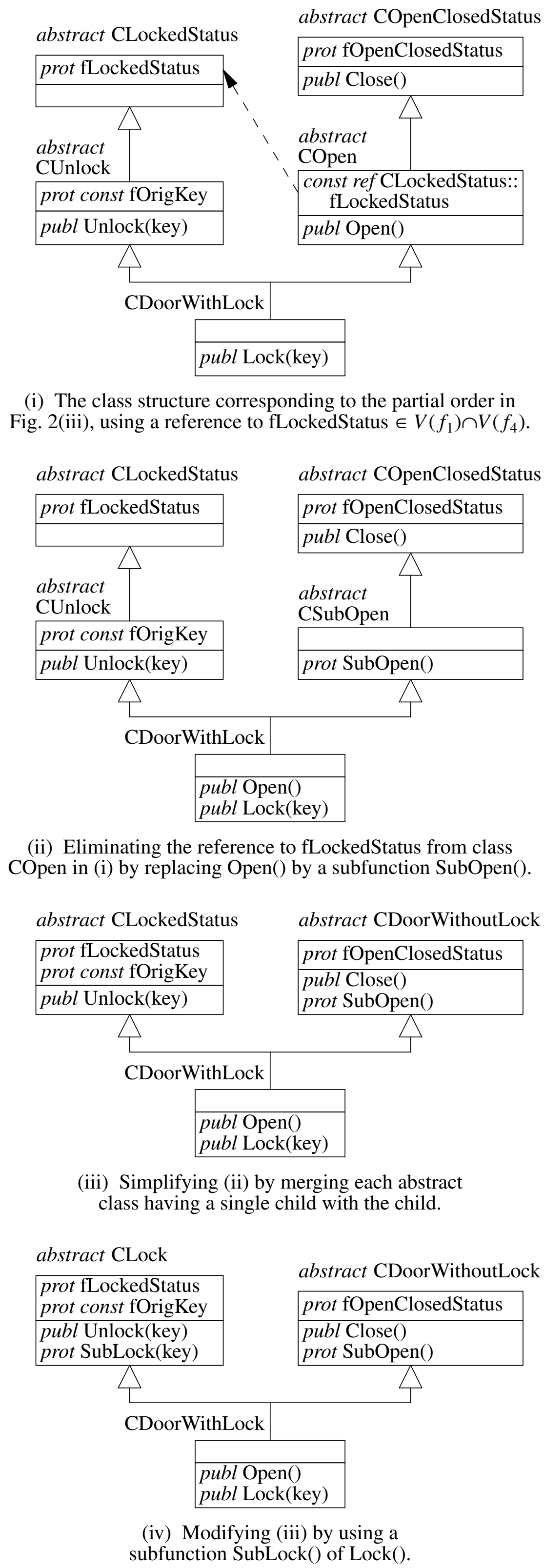
3.1.1. Solution class-structure

Fig. 3(i) shows the class-structure obtained from the partial order in Fig. 2(iii). We use the convention that field-names in a class begin with 'f', class-names begin with 'C', and function-names begin with a capital letter. The constructor of the class `CDoorWithLock` initializes `fLockedStatus` to "unlocked" and `fOpenClosedStatus` to "closed" in keeping with the start-state in Fig. 1; it also initializes the constant `fOrigKey`. A slight variation of Fig. 3(i) would be to interchange the role of `COpen` and `CUnlock` with respect to `CLockedStatus`, i.e., make `COpen` a subclass of `CLockedStatus` and eleminate the reference to `fLockedStatus`, and then replace the subclass relationship of `CUnlock` to `CLockedStatus` by a reference to `fLockedStatus` (as in current `COpen`) from `CUnlock`, but without the restriction "*const*"; this latter fact makes the variant slightly less preferable to Fig. 3(i). Fig. 4 shows a simple implementation of the functions in Fig. 3(i).

```
bool COpenClosedStatus::Close() //see Fig. 3(i)
{ if (open == fOpenClosedStatus)
      { fOpenClosedStatus = closed; return(true); }
  else return(false);
}

bool COpen::Open() //see Fig. 3(i)
{ if ((closed == fOpenClosedStatus) &&
       (unlocked == fLockedStatus))
      { fOpenClosedStatus = open; return(true); }
  else return(false);
}

bool CUnlock::Unlock(key) //see Fig. 3(i)
{ if ((key == fOrigKey) && (locked == fLockedStatus))
      { fLockedStatus = unlocked; return(true); }
  else return(false);
}

bool CDoorWithLock::Lock(key) //see Fig. 3(i)
{ if ((key == fOrigKey) &&
       (unlocked == fLockedStatus) &&
       (closed == fOpenClosedStatus))
      { fLockedStatus = locked; return(true); }
  else return(false);
}
```

Figure 4. Implementations of the functions for the classes in Fig. 3(i).

The classes in Fig. 3(ii) result from factoring the part of `Open()` that does not depend on v_2 = `fLockedStatus` as a subfunction `SubOpen()` as shown in Fig. 5 and replacing `COpen::Open` with `SubOpen`; this eliminates the reference used in Fig. 3(i). Also, the new `Open()`-function, with the factored part replaced by a call to `SubOpen()` (see Fig. 5) is moved down in the hierarchy to allow its access to `fLockedStatus`. Unlike Fig. 1, here we keep the factored part in the original place of the function and the original function moves down.



(i) The class structure corresponding to the partial order in Fig. 2(iii), using a reference to fLockedStatus $\in V(f_1) \cap V(f_4)$.

(ii) Eliminating the reference to fLockedStatus from class COpen in (i) by replacing Open() by a subfunction SubOpen().

(iii) Simplifying (ii) by merging each abstract class having a single child with the child.

(iv) Modifying (iii) by using a subfunction SubLock() of Lock().

Figure 3. The final class-design based on the partial order in Fig. 2(iii) is shown in part (iv); it uses the decomposition of Open() and Lock(key) functions shown in Fig. 5.

```
bool CSubOpen::SubOpen() //in Fig. 3(ii)
{ if (closed == fOpenClosedStatus)
      { fOpenClosedStatus = open; return(true); }
  else return(false);
}


bool CDoorWithLock::Open() //in Fig. 3(ii)
{ if (unlocked == fLockedStatus)
      return(SubOpen());
  else return(false);
}

bool CLock::SubLock(key)
{ if ((key == fOrigKey) && (unlocked == fLockedStatus))
      { fLockedStatus = locked; return(true); }
  else return(false);
}

bool CDoorWithLock::Lock(key)
{ if (closed == fOpenClosedStatus)
      return(SubLock(key));
  else return(false);
}
```

Figure 5. The results of factoring out `SubOpen()` from `Open()` and `SubLock()` from `Lock()`.

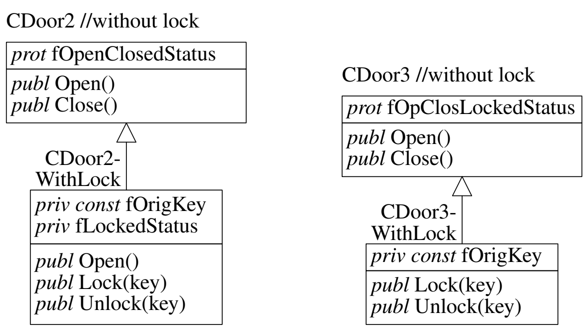
The classes in Fig. 3(iii) result from Fig. 3(ii) by merging each abstract class having only one child with the child. We have renamed the merged class `COpen-ClosedStatus` as `CDoorWithoutLock` to reflect its proper role. If we make `CDoorWithoutLock` a non-abstract class, then this design suffices for modeling doors without a lock and with a lock. In that case, the field `fOpenClosedStatus` will be initialized to "closed" by the constructor of `CDoorWithoutLock`.

Fig. 3(iv) shows a similar modification of Fig. 3(iii) where we factor out the part of `Lock(key)` that does not depend on `fOpenClosedStatus` as a subfunction `SubLock(key)` as shown in Fig. 5, moving the factored part up into the class `CLockedStatus`, and renaming the latter as `CLock`. We can use this design with a slight modification for modeling doors with multiple locks, which is logically the same as a lock with multiple keys. We add now a protected and constant member field `fNumLocks` in `CLock`, which is to be initialized by the constructor of `CDoorWithLock`; the member fields `fLockedStatus` and `fOrigKey` will now be dynamic arrays of size `fNumLocks`, and the parameter "key" in each of `Unlock`, `SubLock`, and `Lock` function will be an array of keys with `fNumLocks` items. The `SubLock`-function returns true if `fLockedStatus` = unlocked and there is an item by item match between the parameter key-array and the `fOrigKey`-array. To lock a door with multiple locks, we still refer to the unique `fOpenClosed-Status` for the door-part. A similar remark holds for `Unlock`-function.

Note that instead of multiple inheritance in Fig. 3(iv), we could use composition. For example, we can have a door without lock as a main object and use the lock as a component (the alternate choice of using a lock as the main object with the door without lock as a component is less desirable).

3.1.2. Some adhoc designs

Fig. 6 shows two alternative designs for doors with and without lock obtained by intuitive and adhoc approaches. In Fig. 6(i), the function `CDoor2::Open()` is the same as `SubOpen()` in Fig. 3(iv). The design in Fig. 3(iv) is superior to the one in Fig. 6(i) in two ways; first, the `Unlock`-function here has an unnecessary access to the variable `fOpenClosedStatus`, and second we cannot use this design for modeling doors with multiple-locks. The design in Fig. 6(ii) is obtained by replacing the variables v_1 = `fOpenClosedStatus` and v_2 = `fLockedStatus` by a single variable v_{12} = `fOpen-ClosLockedStatus` (see Fig. 2(iv)). While this design has the advantage that we do not need to redefine `Open`-function for the class `CDoor3WithLock`, it too cannot be used for doors with multiple locks; the fact that the value "locked" for the variable `fOpenClosLocked-Status` is irrelevant for the class `CDoor3` is an additional minor irritant. Note that this class-design corresponds to the partial order shown in Fig. 2(iv).



(i) A design by a piecewise approach using doors with a lock as a subclass of doors.

(ii) The result of combining fOpenClosedStatus and fLockedStatus in (i).

Figure 6. Some other class designs for doors with and without a lock using intuitive and adhoc approaches.

3.2. Second design problem

Now consider the problem of designing the classes for the operations {deposit, withdraw, findBalance, addInterest, monthlyReport} of a bank-account. Assume that for each account-number there is a savings and a checking account (which are distinguished, say, by an additional ending character *S*/*C*). Also, we are concerned with these operations after a specific account is identified and thus the account-number itself is not used by any of the functions except f_1 = monthlyReport-function, which prepares the monthly-activity report; the addInterest-function applies only to the savings-account. For simplicity, we assume a fixed interest-rate and that the interest amount is computed on the basis monthly-average balance, which can be obtained from the monthly-activity list; the latter contains all activities (successful or unsuccessful) that can affect the account-balance. There is a separate activity-list for each savings and checking account. A checking account has a minimum-balance requirement, and a penalty is applied for each occurrence of the minimum-balance violation when a withdrawal succeeds.

Fig. 7(i) shows the use-relationship; it uses a separate set of variables for savings account activities and checking account activities. The columns with the label '*S*' are for a saving account and the columns with the label '*C*' are for a checking account. We sometimes refer to f_i for a savings account by f_i^S and similarly refer to f_i for a checking account by f_i^C. Fig. 7(ii) shows the partial order $f_i < f_j$ in the context of savings accounts and Fig. 7(iii) shows the same for the checking accounts. They clearly show the similarities of the class structures needed for the two types of account, and they also suggest certain reorganizations of both data and functions to increase the similarity and hence the sharing of common parts of the two class-structures. They are:

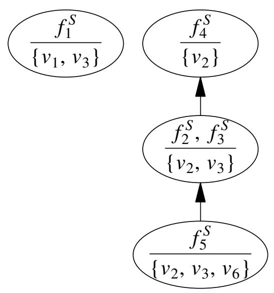
(a) Consolidate the variables v_2 and v_4 (which have the same domain) into a single variable v_{24} = accountBalance and likewise the variables v_3 and v_5 into v_{35} = monthlyActivityList. Different instances of these variables will represent the desired entities for the two types of account, and the functions using these variables will be applied to the appropriate instances. Fig. 7(v) shows the new variable-use relationship based on v_{24} and v_{35}.

(b) Factor out a part f_{31}^C from f_3^C which uses only $\{v_4, v_5\}$ so that corresponding to $f_2^S \approx f_3^S$ for savings account (see Fig. 7(ii)) we have $f_2^C \approx f_{31}^C$. Fig. 8 shows such a factoring of f_3^C, where f_{31}^C does the same computation as f_3^S applied to checking account; we denote by f_{32}^C for the modified form of f_3^C, which combines f_{31}^C with the remaining computations of the original f_3^C. Fig. 7(iv) shows the modified form of partial order in Fig. 7(iii) using the factor f_{31}^C.

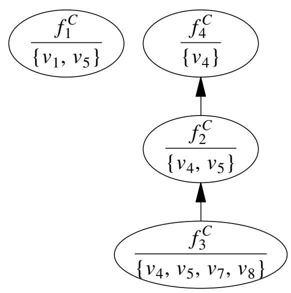
Fig. 9(i) shows the class-structure `CSavingsOr-Checking` obtained from the partial orders in Fig. 7(ii) and Fig. 7(iv) after we merge the classes for $\{f_4\}$, $\{f_2, f_3$ (or $f_{31})\}$, and $\{f_1\}$ just to keep the number of classes small. The identical columns in Fig. 7(v) for $\{f_i^S, f_i^C\}$, $1 \leq i \leq 4$ (with f_3^C replaced by f_{31}^C) suggest the creation of the common parent class `CSaviongsOrChecking` for `CSavings` and `CChecking`. The class `CAccount` shown in Fig. 9(ii) aggregates a savings account and a checking account, in addition to having its own member field `fAccountNumAddress`. In this way, we avoid duplication of account number, etc. in a savings and checking account pair. Note that we have actually factored out from the original f_1 = `MonthlyReport()` which has $V(f_1) = \{v_1, v_{35}\}$, the part `CSavingsOrCheck-ing::MonthlyReport()` which uses only v_{35}. The

Variable-use relationship	f_1: Monthly Report (S)	(C)	f_2: Deposit (S)	(C)	f_3: Withdraw (S)	(C)	f_4: GetBalance (S)	(C)	f_5: AddInterest (S)
v_1: accountNumAddress	x	x							
v_2: savingsBalance			x		x		x		x
v_3: savingsMonthActList	x		x		x				x
v_4: checkingBalance				x		x		x	
v_5: checkingMonthActList		x		x		x			
v_6: interestRate									x
v_7: checkingMinBalance						x			
v_8: penalty						x			

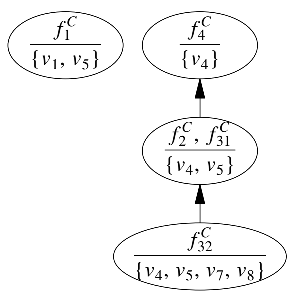
(i) The variable use-relationship for the bank operations; here, $v_7 \approx v_8$, S = savings-account, C = checking-account.



(ii) The partial order $f_i^S < f_j^S$ for a savings-account; also, shown in each node is $V(f_i^S)$.



(iii) The partial order $f_i^C < f_j^C$ for a checking-account; also, shown in each node is $V(f_i^C)$.



(iv) The modified form of (iii) after factoring f_3^C into f_{31}^C and f_{32}^C, where $f_{31}^C = f_3^S$ applied to checking account.

Variable-use relationship	f_1 (S)	(C)	f_2 (S)	(C)	f_3 (S)	(C)	f_4 (S)	(C)	f_5 (S)
v_1: accountNumAddress	x	x							
v_{24}: accountBalance			x	x	x	x	x	x	x
v_{35}: monthlyActivityList	x	x	x	x	x	x			x
v_6: interestRate									x
v_7: checkingMinBalance						x			
v_8: penalty						x			

(v) The new use-relationship after identifying v_2 and v_4 into v_{24} = accountBalance, and v_3 and v_5 into v_{35} = monthlyActivityList.

Figure 7. The use-relationships and partial-orders for the bank example before/after variable consolidations.

For savings-account:

```
bool SavingsWithdraw(amount) //amount > 0.0
{ bool returnVal = false;
  if (amount <= fAcctBalance) {
     fAcctBalance -= amount;
     returnVal = true; }
  update fMonthActList; //details omitted
  return(returnVal);
}
```

For checking-account:

```
bool CheckingWithdraw(amount) //amount > 0.0
{ bool returnVal = false;
  if (amount + fMinBalance <=  fAcctBalance) {
     fAcctBalance -= amount; returnVal = true; }
  else if (amount + fPenalty <= fAcctBalance) {
          generatePenaltyNotification; //details
                                       //omitted
          fAcctBalance -= (amount + fPenalty);
          returnVal = true; }
  update fMonthActList;  //details omitted
  return(returnVal);
}
```

(i) Withdraw-functions for savings and checking accounts.

```
bool CheckingWithdraw(amount) //amount > 0.0
{ if (amount + fCheckingMinBal <=  fCheckingBal)
     return(SavingsWithdraw(amount));
  else if (amount + fPenalty <= fCheckingBal) {
          generatePenaltyNotification;
          return(SavingsWithdraw(amount+fPenalty));
       }
  else return(false);
}
```

(ii) A revised form of checking-account `withdraw`-function after factoring out savings-account `Withdraw`-function.

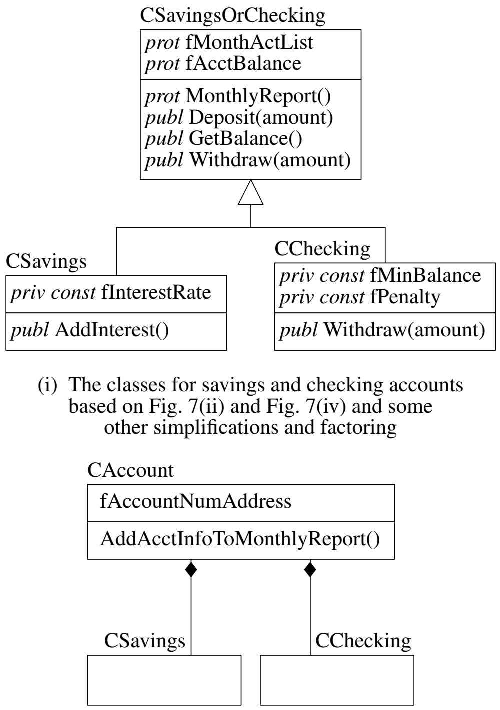
Figure 8. A factoring of Withdraw-function for checking accounts where the factored part is the same as the Withdraw-function for savings accounts.

function `AddAcctInforToMontlhyReport()` in Fig. 9(ii) simply adds account number, etc to the report formed by `CSavingsOrChecking::MonthlyReport()`. The function `CSavings::AddInterest()` updates `fMonthActList` as in the case for withdraw-functions in Fig. 8. The constants `fInterestRate`, `fMinBalance`, and `fPenalty` will be implemented as reference to the corresponding bank-information so that they have the same value for all accounts in a bank (and set by the bank). The presence or absence of `GetBalance` has no effect on the class-structure.

We have not employed a finite-state model for this design problem simply because it does not provide additional insight into the interactions of the functions modeled here.

4. Conclusion

We have presented here a simple, powerful formal method for designing the total class structure of an application based on the partial order on the functions, which is derived from the use-relationship between the functions and their data items. Our method can identify the need to factor out parts of a function as subfunctions, including the subset of data items to be used by the subfunctions, in order to simplify the class-structure. Although functions with high use-complexity will typically be candidates for factoring, but this need not always be the case. Likewise,



(ii) An aggregation of the two subclasses of CSavingsOrChecking.

Figure 9. The class structure for modeling the bank-operations.

our method can identify different data items which play similar roles in a group of functions which are otherwise identical and which can be therefore consolidated into a single variable (data) to serve a common "abstract" role (interface) among those functions. This allows replacing those functions by a single function, moving that function to a higher class in the hierarchy, and thereby simplifying the class structure. This may require some factoring of the functions before the data consolidation can take place. As a general principle, we recommend that each variable initially represent a small amount of information and let the data consolidation step determine if two or more of them should be merged. The results obtained here can be regarded as a first step in building a general theory for designing classes. Our future research will involve refining the theory to support the known design-patterns.

5. Reference

1. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
2. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley, 1995.
3. W.F. Opdyke and R.E. Johnson, Creating abstract superclass by refactoring, *Proc. 21st Annual Computer Science Conference,* pp. 66-73, Indianapolis, 1993.
4. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual*, Addison-Wesley, 1999.
5. G. Snelting and F. Tip, Understanding class hierarchies using concept analysis, *ACM Trans. on Prog. Languages and Systems,* 22(2000), pp. 540-582.
6. W.P. Stevens, G.J. Meyers, and L.L. Constantine, Structured design, *IBM Systems Journal,* 13(1974), pp. 115-119.
7. R.J. Wirfs-Brock and R.E. Johnson, A survey of current research on object-oriented design, *Comm. of Assoc. of Computing Machinary*, 33(1990), pp. 105-124.
8. R. Godin, H. Mili, G.W. Minieau, R. Missaoui, A. Arfi, and T.-T. Chau, Design of class hierarchies based on concept lattice, *Theory and Practice of Object Systems,* 4(1998), pp. 117-134.