

* The affected classes are shaded.
 * The affected data members and methods of affected classes are shaded in dark gray level.

Figure 9: Results of the Algorithm

the one that is most cost effective. It can also be used by software testers to find what areas are affected by the changes, so they can test only the affected areas and still feel confident about the quality of the software.

The algorithms described in this paper are detailed enough to be implemented. In the future, we hope to develop a metric system to quantitatively measure the impacts of the proposed changes, and develop an analysis tool that implements these algorithms. We also want to expand this technique to a distributed object environment, and analyze how changes propagate across heterogenous networks, databases, operating systems, and languages.

6. Acknowledgements

We would like to thank Oliver Jojic, Rick Wise and Bob Crandal for their valuable comments and suggestions on this paper.

7. References

[1] Stephane Barbey, and Alfred Strohmeier, "The Problematics of Testing Object-Oriented Software", SQM'94 Second Conference on Software Quality Management, volume 2, Edinburg, Scotland, UK, 1994.
 [2] S. A. Bohner, "A Graph Traceability Approach for

Software Change Impact Analysis", Ph.D. Dissertation, George Mason University, Fairfax VA, 1995.
 [3] Grady Booch, "Object-Oriented Analysis and Design with Applications," Second Edition, Benjamin/Cummings Publishing Company 1994.
 [4] Mary Jean Harrold and John D. McGregor, "Incremental Testing of Object-Oriented Class Structures", 14th International Conference on Software Engineering, IEEE Computer Society, pages 68--80, Melbourne, Australia, May, 1992.
 [5] Li Li and A. Jefferson Offutt, "Algorithmic Analysis of the Impact of Changes to Object-Oriented Software", George Mason University ISSE Department Technical Report February 1996, ISSE-TR-96-02.
 [6] Robert C. Martin, "Designing Object-Oriented C++ Applications Using The Booch Method", Prentice Hall, Inc. 1995.
 [7] Robert Orfali, Dan Harkey, Jeri Edwards, "The Essential Distributed Objects Survival Guide", John Wiley & Sons, Inc. 1996.
 [8] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.
 [9] M. D. Smith and J. J. Robson. Object-Oriented Programs - The Problems of Validation. In Proceedings of the 1990 IEEE Conference on Software Maintenance, pages 272-281, San Diego, CA, Nov 1990.

contaminate_none (0x00).

7. **Affected by other data member or methods:** If a method is affected by other data members or methods, the state of the object will change, and this method will become contaminated. If it is affected by other methods, the referenced method's changed behavior can change the behavior of this method. If this method is public, it will affect the data members or methods from the changed class, from its clients, and from its children (contaminate_all). If it is protected, it will affect the data members or methods from the changed class and from its children (contaminate_children, contaminate_current). If it is private, it will only affect the data members or methods from the changed class (contaminate_current).

4.3 Other changes

1. **Add a class:** When a new class is added, it does not yet have any classes that use it, so we assume its attribute is contaminate_none.
2. **Delete a class:** When a class is deleted, its data members and methods will no longer be available, so all related classes will be affected. The attributes of all the data members and methods are contaminate_all.

4.4 Examples

This section illustrates these algorithms through an example. The classes are heavily interrelated to better demonstrate the algorithms. Figure 8 shows a system with four classes, A, B, C, and D. We use our algorithms to analyze the impact to the system of changing the type of fa1 in A.

The TotalEffect algorithm first calls SetInit to set the initial values of the different sets. Every class in the system is put into the UncheckedSet. A is put into ACS and marked dirty, AMS[A] is initialized to empty and AFS[A]

to {_fa1}. Then A is picked from UncheckedSet, and FindEffectInClass, FindEffectAmongChildren, FindEffectAmongClients are called to check whether other members in A or in any class that uses A or inherits from A are affected by this change. ACS now contains A, A2, and B. The algorithm picks a class that has not yet been checked from UncheckedSet and applies the same procedures repeatedly until the UncheckedSet is empty. The final results of running these procedures are shown in Figure 9. The affected classes and their affected methods and data fields are shown as shaded areas.

5. Conclusions and future work

In this paper, we have analyzed the characteristics of object-oriented software to understand how encapsulation, inheritance, and polymorphism can influence the propagation of changes through a software system. Industry is still struggling with how to apply change impact analysis to object-oriented software. This paper takes a first step towards building a framework for viewing object-oriented maintenance. Although it is easy to identify and package objects, features such as inheritance make ripple effects of changes more important (and more difficult) to control. We categorize the different kinds of changes that can be applied to object-oriented software, and assign each type an influence attribute according to how the type of change influences other objects in the system. A simple and conservative approach was first described, then ways to optimize the algorithms according to the change type and the change attributes were presented. The complete set of algorithms that calculate the change propagation within the class, between the client and server class, and between the parent and children class are described in a technical report [5].

The technique described in this paper can be used by software developers to run "what if" analyses on different change proposals for the software system, and to choose

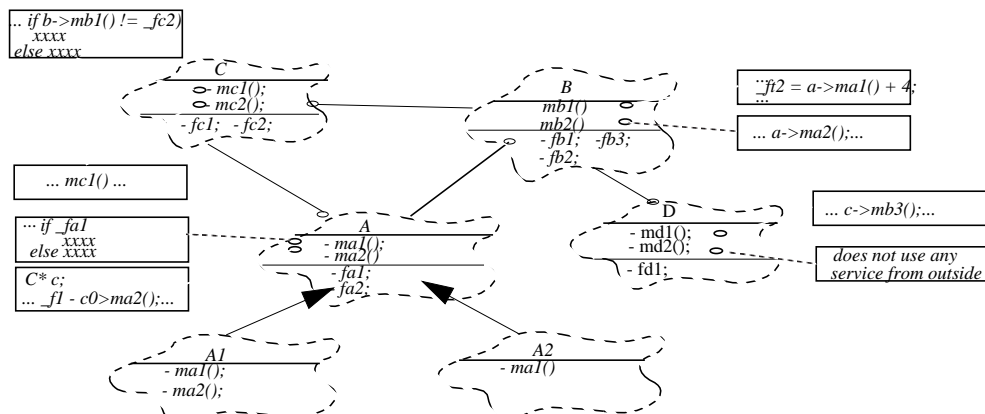


Figure 8: Class Diagram of an Example System

can bring to this data member. It may be possible to fix the change locally, such that this data member will not affect other parts of the system. In that sense, it is not contaminative. For example, when a referenced data member has been changed from public to private, that data member will no longer be available for its clients to reference. The developer can replace this data member by a corresponding member function that retrieves the value of this data member. So its clients can just substitute the data member with the corresponding method, and keep their interfaces the same. In this situation, its clients will not propagate this change further. But it may need more dramatic changes that cause its clients to have to change their interfaces. So we assume that if this data member is public, it will affect the data members or methods from the changed class, from its clients, and from its children (contaminate_all = 0x07). If it is protected, it will affect the data members or methods from the changed class and from its children (contaminate_children | contaminate_current = 0x03). If it is private, it will affect the data members or methods from the changed class only (contaminate_current = 0x01).

4.2 Changes types of methods

The types of changes that could be applied to methods are:

1. **Signature changed:** If the signature of a method has changed, for example, parameters have been added or deleted, it will affect any methods or data members that are related to this method. If the method is public, it will affect the data members or methods from the changed class, from its clients, and from its children (contaminate_all). If it is protected, it will affect the data members or methods from the changed class and from its children (contaminate_children, contaminate_current). If it is private, it will only affect the data members or methods from the changed class (contaminate_current).
2. **Axiom changed:** If the preconditions, postcondition, or axioms are changed, this will change the behavior or semantics of the method. This may or may not affect the methods or data members that reference this method. If the method is public, it will affect the data members or methods from the changed class, from its clients, and from its children (contaminate_all). If it is protected, it will affect the data members or methods from the changed class and from its children (contaminate_children, contaminate_current). If it is private, it will affect the data members or methods from the changed class only (contaminate_current).
3. **Implementation changed:** This type of change

affects the details of the implementation but not the interface. The semantics and behavior may or may not be changed. If this method is public, it will affect the data members or methods from the changed class, from its clients, and from its children (contaminate_all). If it is protected, it will affect the data members or methods from the changed class and from its children (contaminate_children, contaminate_current). If it is private, it will only affect the data members or methods from the changed class (contaminate_current).

4. **Delete a method:** When a method is deleted, it is no longer available to its clients, so all of its clients are affected. If the method is public, it will affect the data members or methods from the changed class, its clients, and its children (contaminate_all). If it is protected, it will affect the data members or methods from the changed class and from its children (contaminate_children, contaminate_current). If it is private, it will only affect the data members or methods from the changed class (contaminate_current).
 5. **Add a method:** When a new method is added, it does not yet have any classes that use it. So we assume it is non-contaminative to its client, but its children have to know the new method. Its attribute should be contaminate_children.
 6. **Scope changed:** There are six possible scope changes relevant to methods:
 - I) Public --> Private
 - II) Public --> Protected
 - III) Protected --> Private
 - IV) Protected --> Public
 - V) Private --> Public
 - VI) Private --> Protected
- I) Changing a method from Public to Private will affect any client classes and subclasses that reference this method, because it will no longer be available. The attribute is 0x06 (contaminate_client | contaminate_children = 0x06).
- II) Changing a method from Public to Protected will affect any client class that reference this method, but not the data members and methods in subclasses and in the changed class. Because this method will no longer be available for any client classes but will still be available for subclasses and the changed class, the attribute of this change is contaminate_client (0x04).
- III) Changing a method from Protected to Private will affect all the subclasses that are derived from this class, because it is no longer available to its subclasses. The attribute of this is Contaminate_children (0x02).
- IV) V) VI) will not affect any other classes except to reveal the state of the object. So its attribute is

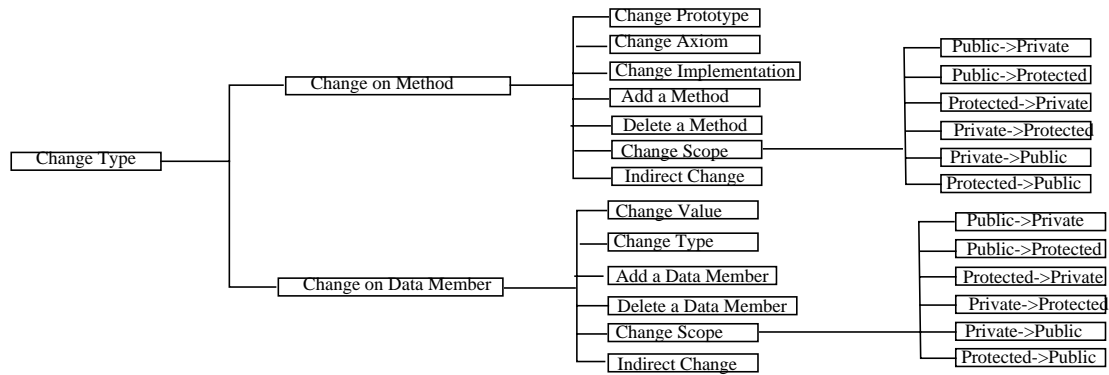


Figure 7: Change Category

nate any of the related classes (Contaminate_none = 0x00).

2. **Type changed:** If the type of the data member is changed, it will affect the methods or other data members that reference it. If this data member is public, it will affect the data members or methods from the changed class, its clients, and its children (contaminate_all = 0x07). If it is protected, it will affect the data members or methods from the changed class and its children (contaminate_children | contaminate_current = 0x03). If it is private, it will only affect the data members or methods from the changed class (contaminate_current = 0x01).

3. **Scope changed:** Let us assume the programming language has three levels of scope: public, protected and private (as in C++ and Ada). Public data members or methods constitute the interface of the class and can be seen by other classes. Protected data members or methods can only be seen by data members or methods within the class or from this class' derived classes. Private data members or methods can only be seen by the data members or methods within the class. A *scope change* is when a data member or method is moved from one scope level to another. There are six possible scope changes:

- I) Public --> Private
- II) Public --> Protected
- III) Protected --> Private
- IV) Protected --> Public
- V) Private --> Public
- VI) Private --> Protected

I) Changing a data member from Public to Private will affect all client classes and subclasses that reference this data member, because it will no longer be available. Since the data members and methods in the changed class can still see this data member, they will not be affected by this change. The attribute is 0x06 (contaminate_client | contaminate_children = 0x06).

II) Changing a data member from Public to Protected will affect client classes that reference this data member, but not the data members and methods in subclasses and in the changed class. Because this data member will not be available for any data members and methods in client classes but will still be available for those in subclasses and the changed class, this change will only affect the client classes of the changed class. The attribute of this change is contaminate_client (0x04).

III) Changing a data member from Protected to Private will affect all subclasses that are derived from this class. Changing the data member from protected to private makes this data member not available for any data members and methods in its subclasses. The attribute of this change is Contaminate_children (0x02).

IV) V) VI) These will not affect any other classes except to reveal the state of the object. So the attribute is contaminate_none (0x00) for all these classes.

4. **Delete a data member:** When a data member is deleted, it will no longer be available to its client, so all its clients will be affected. If this data member is public, it will affect the data members or methods from the changed class, from its clients, and from its children (contaminate_all = 0x07). If it is protected, it will affect the data members or methods from the changed class and from its children (contaminate_children | contaminate_current = 0x03). If it is private, it will only affect the data members or methods from the changed class (contaminate_current = 0x01).

5. **Add a data member:** When a data member is added, no classes use it yet. So we assume it is non-contaminative to its clients, but, according to the Liskov principle, its children have to know the new data member if it is public. Its attribute is contaminate_children (0x02).

6. **Affected by other data members or methods:** If a data member references other data members or methods that are contaminated, this data member may be affected. It is very hard to predict what kind of changes the change of the referenced data member or methods

the current changed class. These classes could be clients, subclasses of the current changed class or the changed class itself.

Contaminate_current: This type of change will only affect the data members and methods in the changed class.

Contaminate_children: This type of change will only affect subclasses that are derived from the changed class.

Contaminate_client: This type of change will only affect client classes that use the changed class.

Contaminate_none: This type of change will not affect any data members or methods belonging to either client classes or subclasses of the changed class, or belonging to the changed class itself.

These attributes can be represented by an *attribute byte*, in which **Contaminate_current**, **Contaminate_children**, and **Contaminate_client** each occupies one bit. If a change will affect all related classes (**Contaminate_all**), its attribute byte is 0x07 (**Contaminate_current** | **Contaminate_children** | **Contaminate_client**). If a change affects client and child classes, its attribute type is 0x06 (**Contaminate_children** | **Contaminate_client** = 0x04 | 0x

02 = 0x06). If a change does not affect any other class, its attribute is 0x00. Figure 7 summaries all the changes categories and the relationships among them. We explain each of these categories in detail in the following section.

4.1 Change types of data members

We classify the potential changes based on the syntactic elements changed and the actions used to make the change. The characteristics of each type of change and how each type can influence other parts of the system is analyzed. Although we have tried to cover all cases, we have no basis on which to claim this listing is exhaustive.

1. **Value changed:** If the value of a data member is changed, it will change the state of the object. This kind of change may or may not affect other data members or methods, depending on whether this change will change the state of the object. If the state of the object is changed, the execution of the program could lead to some paths that have not previously been executed. So if the change causes the object to change its state, it will contaminate all of the related methods in the changed object (**Contaminate_current** = 0x01). If it does not cause the object to change its state, it will not contami-

```

FindEffectAmongChildren(Cp)
BEGIN
  FOR each class Cc that inherits from Cp
    FOR each method m in Cc
      CASE (inheritance type of m)
        Extended Redefine:
        Virtual-extended-Redefine:
        Inherit:
        Virtual inherit:
          IF (mp ∈ Cp ∩ mp ∈ AMS[Cp])
            AMS(Cc) = AMS(Cc) ∪ {m}
            IF m is public
              PAMS(Cc) = PAMS(Cc) ∪ {m}
            ENDF
          OTHERS:
            /* no other case will be affected by the change of m in A */
          ENDCASE
        IF (m ∉ AMS[Cc]) AND (( FREF(m) ∩ AFS(Cp) ≠ Empty ) OR
          ( MREF(m) ∩ AMS(Cp) ≠ Empty ))
          AMS(c) = AMS(c) ∪ {m}
          IF m is public
            PAMS(c) = PAMS(c) ∪ {m}
          ENDF
        ENDFOR /* end of for each method */
      ENDFOR /* end of for each class */
    END FindEffectAmongChildren

```

Figure 6: Algorithm to Calculate the Change Effects Among Subclasses

Pre-conditions on a particular method in a class must be no stronger than those of the same method in a parent class.

Post-conditions on a particular method in a class must be no weaker than those of the same method in a parent class.

The invariant for a class must be a superset of the invariant for a parent's class.

Now we can analyze how the changes in the ACS propagate through parent and children classes by inheritance and polymorphism. From the attribute categories above, we know that any change in a child will not affect its parent because its parent cannot access the methods or data fields of its children. However, changes in a parent can affect its children. Smith and Roberson [9] think a change to a parent class can potentially affect all descendants. Below, we analyze impacts of changes through the inheritance categories of the methods in subclasses.

If the method or data field A in a child class is a new attribute, A is defined in M but not in P, or the signatures of A in M and P are different. Since A is not accessible in P, the new attribute in R will not affect the A in P. But it will affect R's children.

If a method or data field A in a child class is an inheritance attribute, A is locally bound to P. In this situation, if A in P changes, A needs to be retested in R, because the context of A in P is different from the context of A in R.

If the method or data field A in a child class is a total-redefined attribute, M redefines A without using P's version of A. So A's change in P will not affect A in R. But if A in R uses other methods in AMS[P], it will still be affected.

If the method or data field A in a child class is an extended-redefined attribute, M extended the functionality of A in P by adding extra functionality to A. The A in P is invoked in M. So any change of A in P will affect the A in R. A's change in R will not affect its parent P since its parent either does not have A or its version of A has a different signature. A's change in P will affect R, so R is in ACS.

Figure 6 show the algorithm that finds the impacts of changes through inheritance and polymorphism. In this algorithm, we use C_p denote the parent class and C_c denote the child class, m_p denote the method in parent class, and m express the method in child class.

3.5 Complexity estimation

Assuming the number of classes in the system is m , let

$$nm = \max(\text{number of methods in Class } i) \quad 1 \leq i \leq m \quad \text{formula 1}$$

$$nf = \max(\text{number of data fields in Class } i) \quad 1 \leq i \leq m \quad \text{formula 2}$$

$$n = \max(nm, nf) \quad \text{formula 3}$$

By analyzing TotalEffect (in Figure 2), we can tell that the overall complexity of the whole algorithm is:

$$O(\text{TotalEffect}) = \max(O(\text{SetInit}), O(m) * O(\text{the body of for loop in Figure 2})) \quad \text{Formula 4}$$

Since SetInit has to mark each class in the ACS as *dirty*, the worst case complexity of SetInit is $O(m)$. The complexity of the body of the FOR loop in TotalEffect is equal to $\max(O(\text{FindEffectInClass}), O(\text{FindEffectAmongChildren}), O(\text{FindEffectAmongClients}))$. By analyzing the FindEffectInClass (in Figure 4), we get the worst case complexity of FindEffectInClass(c) to be $o(m)$. By analyzing Figure 5 and Figure 6, we find the worst case complexity of FindEffectAmongChildren is $o(mn)$ and the worst case complexity of FindEffectAmongClients is $o(m^2n)$. So $O(\text{FOR loop in TotalEffect}) = o(m^2n)$. From Formula 4, we get $O(\text{TotalEffect}) = o(m^3n)$.

The overall algorithms actually calculate the transitive closure of all the affected classes. As we mentioned before, some classes that have already been checked could be remarked as *dirty* if their AMS or AFS were expanded. Since the number of members in AMS or AFS cannot be greater than n , the total running time of our analysis is $O(\text{TotalEffect}) = O(m^3n) \times O(n) = O(m^3n^2)$.

4. Algorithm improvement

The algorithms in section 3 offer a conservative approach to estimating the system-wide impacts of proposed changes. Because certain types of changes will not necessarily affect other parts of the system, some classes that are put into the ACS may not necessarily affect other classes. In other words, our algorithms include classes if they **might** be affected by the proposed changes. In this section, we categorize changes that can be applied to object-oriented software, analyze the characteristics of these categories, and discuss in detail what kinds of changes will affect other parts of the system and what kinds of changes will not affect other parts of the system.

There are many different kinds of changes that can be applied to object-oriented software. We categorize these changes by specifying the types of changes that could be applied to data members, methods, classes and objects. Each type of change is assigned one of the following five attributes according to how they influence other classes in the system. We describe these in terms of the class that is currently being considered for change, which we call the *changed class*.

Contaminate_all: This type of change will affect the data members and methods in any classes that are related to

designer specifies the modifier, which may contain various types of attributes that alter the parent class to get the resulting subclass. Although M transforms P into a new class R, M does not totally constrain R. We must also consider the inheritance relation since it determines the effects of composing the attributes of P and M and mapping them into R. The inheritance relation determines the visibility, availability and format of P's attributes in R. Since inheritance is deterministic, rules can be constructed to identify the availability and visibility of each attribute.

When a subclass redefines one of its parent's methods, it can either totally replace the method or simply expand its functionality. The impact of the parent's method on this subclass will be different depending on how the subclass expands the parent's method. If the subclass totally re-implements its parent's method, the change in the parent's method will not affect the subclass. If the subclass expands its parent's service based on the service the parent's method provides, any changes in the parent's method could affect this subclass. Because of this, we extend Harold and McGregor's [4] attributes classification by splitting the redefine and virtual redefine into extended redefine, total redefine, virtual-extended redefine, and virtual-total-redefine. As a result, methods in subclasses are divided into the following eight categories:

New attribute: A is an attribute that is defined in M but not in P or A is a member function attribute in M and P but has a different signature. In this case, A is bound to the locally defined attribute in M. A is accessible within R and accessible outside R if A is public; A is not accessible in P.

Inherited attribute: A is defined in P but not in M. In this case, A is bound to the locally defined attribute in P. A is accessible within R and accessible outside R if A is public; A is accessible both within and outside P.

Extended-redefined attribute: A is defined in both P and M with the same signature. The A in M will extend the functionality of A in P by using the services of A in P. In this case, A is bound to the locally defined attribute in M. A is accessible inside R and if it is public, outside R; A is not accessible in P.

Total-redefined attribute: A is defined in both P and M with the same signature. The A in M will replace the functionality of A in P by implementing the services without using the A in P. In this case, A is bound to the locally defined attribute in M. A is accessible within R and accessible outside R if A is public; A is not accessible in P.

Virtual-new attribute: A is specified in M but its implementation may be incomplete in M to allow later definitions or A is specified in M and P and its implementation may be incomplete in P, but A's signature

differs in M and P. In this case, A is bound to the locally defined attribute in M. A is accessible within R and if it is public, outside R; A is not accessible in P.

Virtual-inherited attribute: A is specified in P but its implementation may be incomplete in P to allow later definition, and A is not defined in M. In this case, A is bound to the locally defined attribute in P. A is accessible within R and if it is public, outside R; A is accessible both inside and outside P.

Virtual-extended-redefined attribute: A is specified in P but its implementation may be incomplete in P to allow for later definition and A is defined in M with the same signature as A in P. The A in M will extend the functionality of A in P by using the services of A in P in M's implementation. In this case, A is bound to the locally defined attribute in M. A is accessible inside and if it is public, outside R; A is not accessible in P.

Virtual-total-redefined attribute: A is specified in P but its implementation may be incomplete in P to allow for later definition and A is defined in M with the same signature as in P. The A in M will replace the functionality of A in P by implementing the services without using the A in P. In this case, A is bound to the locally defined attribute in M. A is accessible inside and if it is public, outside R; A is not accessible in P.

The inheritance relation determines visibility, availability and the format of P's attributes in R.

Polymorphism allows a reference to denote instances of various classes. It is usually constrained by inheritance. Polymorphism means that the same method can do different things, depending on the class that implements it. It lets two similar objects be viewed through a common interface and allows subclasses to override an inherited method without affecting the ancestor's methods [7]. If the inheritance scheme is subtyping, the denoted objects all have at least the properties of the root class of the hierarchy. Thus an object belonging to a derived class could be substituted into any context in which an instance of the base class appears, without causing a type error in any subsequent execution of the code. Martin calls this *total polymorphism*, as described by the Liskov Substitution principle: If for each object o1 of type S, there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T [6]. Less formally, the software can always pass a pointer or reference to a derived class to a function that expects a pointer or reference to a parent class. Since polymorphic names can denote object of different classes, it is impossible to predict which class will be executed until run time. This type of inheritance is also called *strict inheritance* and has the following characteristics:

system. This class needs to be checked again by the algorithms, so it is thrown back to the UncheckedSet to wait to be picked by the main loop in TotalEffect. FindEffectAmongClients is shown in Figure 5.

3.4 Inheritance

Inheritance is the mechanism that allows the developer to create new child classes -- known as subclasses or derived classes -- from existing parent classes. Inheritance represents a hierarchy of abstractions, in which a subclass inherits from one or more super classes. A child class shares the structure or behavior defined in its parent class. A child class can express differences with its parent class by modifying and adding properties.

Different languages accept different inheritance schemes (strict inheritance, subtyping, subclassing etc.). *Strict inheritance* is the simplest inheritance scheme; it

keeps the exact behavior of its parent. The inherited properties cannot be modified, and the derived class can only be redefined by adding new properties. *Subtyping* is the most commonly used scheme. In addition to properties of strict inheritance, subtyping allows the inherited properties to be redefined when the parent's operation is not appropriate for the subclass. In *subclassing*, the derived class is not considered to be a specialization of the base class, but a completely new abstraction that bases part of its behavior on part of another class. This scheme is also called *implementation inheritance*. The derived class can therefore choose not to inherit all the properties of its parent. In this paper, we assume the language uses subtyping. The algorithm can be easily modified for other inheritance schemes.

Inheritance can be thought of as an incremental modification technique that combines a parent P with a modifier M to get a resulting class R ($R = P \oplus M$). The subclass

```

FindEffectAmongClients (C0)
input: The ACS and the AMS, AFS for c. They could come from initialization or the
      result of previous execution.
output: The expanded ACS, the expanded sets: ACS, AMS, AFS, PAMS, and PAMS.
FOR each class C that uses C0
BEGIN
  OLDAMS[c] = AMS[c]
  OLDAFS[c] = AFS[c]
  FOR each method m in c
  BEGIN
    IF ( MREF(m) ∩ PAMS(C0) ≠ Empty ) OR ( FREF(m) ∩ PAFS(C0) ≠ Empty )
    BEGIN
      AMS(c) = AMS(c) ∪ {m}
      IF m is public
        PAMS(c) = PAMS(c) ∪ {m}
    ENDIF
  ENDFOR
  FOR each field f in c
  BEGIN
    IF ( (FREF(f) ∩ PAFS(c)) ≠ Empty ) OR ( MREF(f) ∩ PAMS(c) ≠ Empty )
    BEGIN
      AFS(c) = AFS(c) ∪ {f}
      IF f is public
        PAFS(c) = PAFS(c) ∪ {f}
    ENDIF
  ENDFOR
  FindEffectInClass(c)
  IF ((OLDAMS[c] ≠ AMS[c]) OR (OLDAFS[c] ≠ AFS[c]))
  BEGIN
    ACS = ACS ∪ {c}
    UncheckedSet = UncheckedSet ∪ {c}
  ENDIF
END FindEffectAmontClients

```

Figure 5: Algorithm to Calculate the Change Effects Among Clients

```

FindEffectInClass(c)
/* Find the effect within the class if certain data members or methods have changed */
input: The AMS and AFS sets of c. They could come from initialization or result from
      a previous execution.
output: New AMS and AFS in Class c. They include the original members plus any newly
       added members.
BEGIN
  Analyze the CFG and DFG of each method, constructing MREF & FREF sets for each
  method and data fields.
  /* initial searching of methods and data fields */
  FOR each method m in c
  BEGIN
    IF ( m ∉ AMS ) AND ( ( MREF(m) ∩ AMS(c) ≠ Empty ) OR ( FREF(m) ∩ AFS(c) ≠ Empty ) )
      AMS(c) = AMS(c) ∪ {m}
      IF m is public member
        PAMS(c) = PAMS(c) ∪ {m}
      ELSE CleanMethods = CleanMethods ∪ {m}
    ENDIF
  ENDFOR
  FOR each field f in c
  BEGIN
    IF ( f ∉ AFS ) AND ( ( FREF(f) ∩ AFS(c) ≠ Empty ) OR ( MREF(f) ∩ AMS(c) ≠ Empty ) )
      AFS(c) = AFS(c) ∪ {f}
      IF f is public attribute
        PAFS(c) = PAFS(c) ∪ {f}
      ELSE CleanFields = CleanFields ∪ {f}
    ENDIF
  ENDFOR
  REPEAT
    Pick one method m from CleanMethods
    IF ( MREF(m) ∩ AMS(c) ≠ Empty ) OR ( FREF(m) ∩ AFS(c) ≠ Empty )
      AMS(c) = AMS(c) ∪ {m}
      CleanMethods = CleanMethods ∪ {m}
      IF m is public member
        PAMS(c) = PAMS(c) ∪ {m}
      ENDIF
    UNTIL CleanMethods is stable
  REPEAT
    Pick one data field f from CleanFields
    IF ( FREF(f) ∩ AFS(c) ≠ Empty ) OR ( MREF(f) ∩ AMS(c) ≠ Empty )
      AFS(c) = AFS(c) ∪ {f}
      CleanFields = CleanFields ∪ {f}
      IF f is public attribute
        PAFS(c) = PAFS(c) ∪ {f}
      ENDIF
    UNTIL CleanFields is stable
  END FindEffectInClass

```

Figure 4: Algorithm to Calculate Change Effects Inside Classes

interact with it. An object also has a private component that implements the methods. The object's implementation is encapsulated -- that is, hidden from the public view [7]. In the presence of encapsulation, the only way to observe the state of an object is through its interface (public methods). The class hides the properties of its instances to conceal the data structure and the details of implementation. All the features of an object are usually hidden, such that the only way the state can be examined or modified is by invoking its interface formed by its public properties. The interface is a basis for a protocol that objects use to communicate with each other by requesting an object to invoke one of its operations. Methods and data members in the class can see all the properties within the class.

For each class c , the $AMS[c]$ and the $AFS[c]$ contain all the methods and data fields that could be affected by the specified changes. Since the only way to observe the state of an object or operate on an object is through its public member, an object's clients can only be directly affected by the changes in the public member. We define the $PAMS[c]$ as the public affected method set, and the $PAFS[c]$ as the public affected data field set. They contain the public affected members of the class c . Obviously, $PAMS[c] \subseteq AMS[c]$ and $PAFS[c] \subseteq AFS[c]$.

3.3.1 Finding effects within the class (FindEffectIn-Class)

When a method or data field in a class c changes, the effects within c can be found by $FindEffectInClass(c)$. Since the execution within each method is still sequential, we can apply CFG and DFG techniques to find the MREF and FREF sets of c . $FindEffectInClass$ (in Figure 4) checks each member in c that is not affected. If m references any method in AMS ($REF(m) \cap AMS(c) \neq \text{Empty}$) or m references any data field in the AFS ($REF(m) \cap AFS(c) \neq \text{Empty}$), m could be affected by the changes in the $AMS[c]$ and the $AFS[c]$. So it will be added to the AMS and to the $PAMS$ if m is public. This sounds reasonable, but unfortunately this algorithm has a flaw.

Assume a class has methods $m1, m2, m3, m4, m5$. $m1$ and $m2$ are in the AMS ; if $m3$ references $m5$ and $m5$ references $m2$, $m3$ references $m2$ indirectly, $m2 \in AMS$, so $m3$ should belong to the AMS . But when we check $m3$, since $m5$ has not been checked yet, $m3$ could not find any reference in the AMS set, so the algorithm thinks it is clean and fails to put it in the AMS . To fix this problem, we put the methods or data fields that cannot find any reference in the AMS or the AFS set in a temporary clean set. After having checked all the methods and data fields in the class, the algorithm repeatedly examines all the members in this clean set to check whether there are more methods or data fields that could be affected until no more members can be removed from the clean set.

3.3.2 Finding effects among clients (FindEffectAmongClients)

If class A sends messages to class B , A uses B , and we say that class A is class B 's *client*. Encapsulation builds a wall between the class and its clients. Assume the current affected class is c_0 and we want to determine which client of c_0 will be affected. Because of the encapsulation, the clients of c_0 can only access this class through its public members, which means its clients can only be affected by this class's members that belongs to $PAMS[c_0]$ and $PAFS[c_0]$.

$FindEffectAmongClients$ examines each client class of c_0 and puts any methods or data fields that reference methods or data fields in the $PAMS$ or $PAFS$ of c_0 into their own AMS or AFS . If any of these members are public, they are put into the $PAMS$ and $PAFS$ of these client classes.

When the algorithm checks the client c of c_0 , it first saves the $AMS[c]$ and $AFS[c]$ to $OLDAMS[c]$ and $OLDAFS[c]$. At the end, the algorithm checks whether any new members have been added to AMS or AFS by comparing the AMS with the $OLDAMS$ or AFS with the $OLDAFS$. If there are new members in a client class, it means this client class might influence some already checked classes in the

```

SetInit()
BEGIN
  UncheckedSet = {All the classes in the system}
  ACS = {The set of classes proposed to change}
  Mark each class in the ACS dirty
  FOR each class in the ACS  $C_i$ 
  BEGIN
     $AMS[C_i] = \{\text{The set of methods changed in } C_i\}$ 
     $AFS[C_i] = \{\text{The set of fields changed in } C_i\}$ 
  ENDFOR
END SetInit

```

Figure 3: Initialization Algorithm

$$\begin{aligned} \text{AMS}_n(C) &= \{m \mid \forall m \text{ in } C, \exists x, \exists c, \text{ s.t. } x \in \text{FREF}(m) \\ &\wedge x \in \text{AFS}_{n-1}(c)\} \cup \{m \mid \forall m, \exists n, \exists c, \text{ s.t. } n \in \text{MREF}(m) \\ &\wedge n \in \text{AMS}_{n-1}(c)\} \end{aligned}$$

$\text{AMS}_{n-1}(c)$, where c is any class in the system:

$$\begin{aligned} \text{AFS}_n(C) &= \{f \mid \forall f \text{ in } C, \exists x, \exists c, \text{ s.t. } x \in \text{FREF}(f) \\ &\wedge x \in \text{AFS}_{n-1}(c)\} \cup \\ &\{f \mid \forall f \text{ in } C, \exists m, \exists c, \text{ s.t. } m \in \text{MREF}(f) \\ &\wedge m \in \text{AMS}_{n-1}(c)\} \end{aligned}$$

3. Algorithms

This section presents four algorithms that combine to analyze the ripple effects through the system when a component is being considered for change. The algorithms calculate the transitive closure of each class in ACS. They pick an unchecked class from the system, check all the classes that are directly related to this class via encapsulation or inheritance, then add all the classes that could potentially be affected by this class to the ACS. The ACS of the entire system is the union of all the ACSs of each class in the system. TotalEffect is the main algorithm that picks an unchecked class c from the system and calls the other algorithms. It calls FindEffectInClass(c) to calculate the change propagation inside c , FindEffectAmongClient(c) to determine the client classes that could be affected by c , and FindEffectAmongChildren(c) to determine the subclasses that could be affected by c . All the affected classes are put into the ACS.

3.1 Total effect

The TotalEffect algorithm initializes the ACS, the AMS and the AFS sets of each class in ACS using SetInit. SetInit puts every class in the system into an *UncheckedSet*. TotalEffect picks one class from the UncheckedSet, then uses FindEffectInClass(c) to analyze the effects within the class, FindEffectAmongChildren(c) to analyze

the effects in the system according to inheritance, and FindEffectAmongClient(c) to analyze the effects in the system according to encapsulation. The following subsections explain these three algorithms in detail. During execution, if the AMS or AFS of any checked class increases, they are put back into UncheckedSet for further examination. Figure 2 shows the high level flow of the algorithm.

3.2 Initialization

The Initialization algorithm in Figure 3 initializes the data structures to satisfy the preconditions of the algorithms. The user can specify what methods and data fields of what classes to analyze. SetInit will set the initial value of ACS to the classes that have been proposed to change. For each class c in ACS, it will set the AMS[c] to the methods that have been proposed to change, and AFS[c] to the data fields that have been proposed to change.

3.3 Encapsulation

In traditional programming, the basic unit is a procedure. In object-oriented programming, methods or member functions are the actions that can be performed on objects. They manipulate and express the state of the object. They define the interface to other classes and in many ways are not logically independent. The control flow analysis and data analysis techniques are not directly applicable to the object level, since there is no sequential order in which the operation will be invoked. Thus, we can treat classes as the basic unit for analysis, and focus on classes and objects.

Encapsulation is a way to separate the implementation of a data object from its specification. An object does this by managing its own resources and limiting the visibility of what others should know. An object publishes a public interface that defines how other objects or applications can

```
TotalEffect()
input: The set of changed classes and their changed methods and data fields.
output: The affected classes and their methods, data fields in the system.
BEGIN
  SetInit() // see Figure 3
  WHILE ACS has dirty classes
    Pick one class c from UncheckedSet
    FindEffectInClass(c) // see Figure 4
    FindEffectAmongChildren(c) // see Figure 6
    FindEffectAmongClient(c) // see Figure 5
  ENDFOR
END TotalEffect
```

Figure 2: Algorithm to Calculate the Total Effect in the System

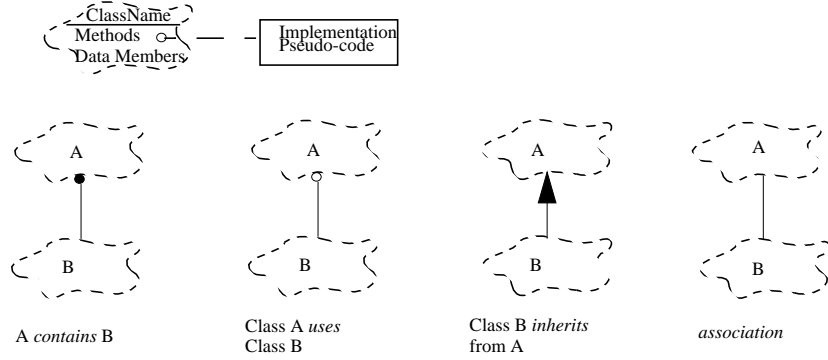


Figure 1: Some Booch notation for class diagrams

such like keys, steering wheels, etc. A class can *inherit* the instance variables, interfaces, and instance methods of another class as if they were defined within it. This expresses the generalization/specialization relationship. For example, a Sedan is a specialization of a general car. The class from which another class inherits is called *parent* or *superclass*. The class that inherits from parent is called a *child*, *subclass* or *derived class*. If a class has more than one parent, this kind of relationship is called *multiple inheritance*. *Association* is a semantically weak relationship. It only states there is some relationship between the classes expressed without explicitly stating what kind of relationship. It could be contains, use, or inheritance. This is usually used in the analysis and design phases when some relationships among classes are still not clear or we just want to represent a general relationship among the classes.

2.1 New definitions

In structured programming, one thinks in terms of inputs, functions and outputs. In object-oriented programming (OOP), the approach is different -- a message is passed to an object requesting an operation on the object. Objects have methods and data fields; the methods specify the allowable operations on the objects' private data, and the data fields specify the state information for the object. When a data field or method changes, it could affect other classes through message passing. We define the *affected class set* (ACS) to be the set of classes that could potentially be affected, the *affected method set* of c ($AMS[c]$) to be the set of methods that could potentially be affected in class c , and the *affected field set* of c ($AFS[c]$) to be the set of data fields that could potentially be affected in class c .

If x is a member of a class, $REF(x)$ (*reference set of x*) is the set of members that are referenced by x , in other words, member m is in $REF(x)$ if m is used by x . $REF(x)$ represents the set of members that could affect x if they change. When necessary we use $FREF(x)$ (*data field refer-*

ence set of x) to denote reference set composed of data fields, and $MREF$ to denote reference set composed of methods. The $AMS[c]$ (*affected method set*) includes the set of all methods in c that reference any method in the $AMS[c']$ or any field in the $AFS[c']$ (*affected field set*), c' can be c or any other class c use. The $AFS[c]$ (*affected field set*) contains all the fields in c that are defined or eventually recursively defined by a field in the $AFS[c]$ or a method in the $AMS[c]$. The $PAMS$ (*public affected method set*) is the subset of the AMS that is composed of public methods of c . The $PAFS[c]$ (*public affected field set*) is the subset of AFS that is composed of public data fields of c .

Induction formulae described below explain how to calculate the $AMS[c]$ and $AFS[c]$. Suppose we want to see what impacts a change could have on a system if the data members or methods in certain classes are changed. First we initialize the ACS to the set of classes that are to be changed, and initialize the AMS and AFS of each class in the ACS. For example, if it has been proposed to change the data member f_0 and the method m_0 for the class c in ACS, we have:

$$AFS(c) = \{f_0\}$$

$$AMS(c) = \{m_0\}$$

Assume that at step $n-1$, $AMS_{n-1}(c)$ contains all its affected methods in C , and $AFS_{n-1}(c)$ contains all its affected fields or data members in C :

$$AMS_{n-1}(c) = \{m | \forall m, m \text{ is an affected method in class } c\}$$

$$AFS_{n-1}(c) = \{f | \forall f, f \text{ is an affected field in class } c\}$$

Then at step n , the AMS of c will contain all its methods that reference any data field in $AFS_{n-1}(c)$ plus all its methods that reference any methods in $AMS_{n-1}(c)$, where c is any class in the system:

The AFS of c contains all its fields that use any field in $AFS_{n-1}(c)$ plus any field that references any method in

tem, they need to know how this update would impact the rest of system. Determining how a potential change might impact the system is referred to as *change impact analysis* [2]. Without impact analysis, engineers could make small changes that unintentionally cause major problems or have ripple effects throughout the system. To predict impacts of changes to object-oriented software, engineers can use the technique described in this paper to evaluate the effects of changes before committing them. During maintenance, when changes have been made to the system, we need to estimate how many classes need to be retested. Retesting too many classes in the system will increase the cost of testing, but retesting too few classes in the system might adversely effect the quality of the software. By applying this technique, testers can learn what classes are possibly affected by the change and retest only those classes.

Although there has been some research into this problem in the context of procedural software, maintaining object-oriented software is still more of an art than an engineering skill.

In this paper, we analyze a number of possible changes to object-oriented software, how these changes affect the classes in the system, and describe a set of algorithms that determine what classes will be affected by the changes. Due to space limitations, we only summarize most of the algorithms; the full details can be found in a technical report [5].

Section 2 presents object-oriented concepts and definitions used in the paper and presents the theoretical background for our algorithms. Section 4 first analyzes how encapsulation, inheritance, and polymorphism will affect the change propagation, and then describes a simple algorithm to estimate potentially affected classes. Algorithms are presented that calculate change propagation within classes, between client and server classes, and between parent and children classes. We also categorize possible changes to an object-oriented system and give each type of change a change attribute according to how these different types of changes can affect the other parts of the system, and discuss how to optimize the original algorithms according to these different change categories. An example system is given, and the algorithms described in this paper are applied to an example system to analyze the impact result. The complete set of algorithms are given in the technical report [5].

2. Definitions and background

An object-oriented system is composed of objects and classes. An object is composed of a set of *properties*, which define its state, and a set of *operations*, which define its behavior. The *state* of an object encompasses all the

properties of the object plus the current values of each of these properties. *Behavior* is how an object acts and reacts, in terms of its state changes and message passing [3]. The state of an object represents the cumulative results of its behavior. The constants and variables that represent an instance's state are called *Data Fields*, *Instance Variables* or *Data members* depending on the language. *Messages* are operations that one object performs upon another, and *Methods* or *Member Functions* are operations that clients may perform upon an object. In this paper, we use Data Field and Method. Member are used to refer either data field or method. A *Class* is the specification of an object; it is the "blueprint" from which an object can be created. A class describes an object's interface, the structure of its state information, and the details of its methods [6]. Objects are runtime instances of a class. An *Abstract Class* is a class that only partially describes an object. Usually some or all of its interface elements are without implementation.

A *control flow graph* (CFG) is a finite, connected directed graph $G = (N, E, N_s, N_f)$ where N is a finite set of nodes, $E \subseteq N \times N$ is a finite set of edges, $N_s \in N$ is the start node and $N_f \in N$ is the final node. A node in a CFG represents a statement or a *basic block*, i.e., a sequence of statements having the property that each statement in the sequence is executed whenever the first statement is executed. An edge (N_i, N_j) represents a possible flow of control between two statements or basic blocks, i.e. the statement (or block) represented by N_i is executed before the statement or basic block that is represented by N_j .

A *Data Definition* is an expression or part of an expression that modifies a data item. A *Data Use* is an expression or that part of an expression that references a data item without modifying it. A *def-use* pair is a definition and a use such that the definition may, under some executions, reach the use without going through another definition. A *data flow graph* (Def-Use) graph is a directed graph where the nodes and some edges are described by def-use relationships.

The *transitive closure* of a relationship R is the relation R^+ defined by cR^+d , if and only if there is a sequence $e_1Re_2, e_2Re_3, \dots, e_{m-1}Re_m$, where $m \geq 2$, $c=e_1$ and $d=e_m$. In this paper, we use Booch Notation [3] to express relationships among classes. Figure 1 shows Booch Notations that express some class relations used in this paper.

Class A *contains* class B if the instance of class B is held in one of the instance variables of A . This represents the "whole/part" relationship. For example, we can say a car has an engine, or a car has doors. Class A *uses* class B if A sends messages to B . For example, we say a person uses a car. The person tells the car to start-up, turn, and stop by sending messages to the car through car interfaces

Algorithmic Analysis of the Impact of Changes to Object-Oriented Software

Li Li
LCC L.L.C.
2300 Clarendon Blvd., Suite 800
Arlington, VA 22201
phone: 703-516-7394
email: lili@lccinc.com

A. Jefferson Offutt¹
ISSE Department, 4A4
George Mason University
Fairfax, VA 22030-4444
phone: 703-993-1654
email: ofut@isse.gmu.edu

Abstract

As the software industry has matured, we have shifted our resources from being primarily devoted to developing new software systems to primarily making modifications in evolving software systems. A major problem for developers in an evolutionary environment is that seemingly small changes can ripple throughout the system to have major unintended impacts elsewhere. As a result, software developers need mechanisms to understand how a change to a software system will affect the rest of the system. Although the effects of changes in object-oriented are restricted, they are also more subtle and more difficult to detect. This paper presents algorithms to analyze the potential impacts of changes to object-oriented software, taking into account encapsulation, inheritance, and polymorphism. This technique allows software developers to perform “what if” analysis on the effect of proposed changes, and thereby choose the change that has the least influence on the rest of the system. The analysis also adds valuable information to regression testing, by suggesting what classes and methods need to be retested, and to project managers, who can use the results for cost estimation and schedule planning.

Key words: Change Impact Analysis, Object-Oriented Software, Software Testing, Software Maintenance.

1. Introduction

Software systems have traditionally been decomposed into subsystems in a top down fashion according to their functionality. The object-oriented approach describes the system in terms of objects that make up the problem domain. Applying object-oriented tech-

nology can lead to better system architectures, and enforce a disciplined coding style. Rumbaugh [8] states that because the object classes provide a natural unit of modularity, an object-oriented approach produces a clean, well-understood design that is easier to test, maintain, and extend than non-object-oriented designs.

Despite the advantages of object-oriented technology, it does not by itself ensure the quality of the software, shield against developer’s mistakes, nor prevent faults. Barbey and Strohmeier think the object-oriented paradigm can also be a hindrance to testing, due to encapsulation, inheritance, and polymorphism [1]. This is an important issue to commercial clients that have begun to use object-oriented technologies for their critical system. Maintaining current object-oriented systems is more of an art (similar to where we were 15 years ago with procedural systems) rather than an engineering skill. We are beginning to see “legacy” object-oriented system in industry because of this. So, maintaining object-oriented systems is becoming an important area of concern for industry. The next “great frontier” will be (and already is for some) how to maintain these objects in large, complex systems. Although objects are more easily identified and packaged, inheritance (for example) makes the ripple effects of object-oriented systems far more important to control than we see in procedural system systems today. The algorithms described in this paper take an important first step towards building a framework for viewing object-oriented maintenance and its ripple effects through systems, and provide the basis for impact analysis of object-oriented effects through systems.

Software evolution refers to the on-going enhancements of existing software systems, involving both development and maintenance. As software ages and evolves, the task of maintaining it becomes more complex and more expensive, which is especially true for object-oriented systems.

When engineers consider an update to an existing sys-

1. Partially supported by the National Science Foundation under grant CCR-93011967.