Assertions in Object Oriented Software Maintenance: Analysis and Case Study

Manoranjan Satpathy[1], Nils T Siebel[2] and Daniel Rodríguez[1]

[1]Applied Software Engineering Group
Department of Computer Science
The University of Reading
PO Box 225, Whiteknights
Reading RG6 6AY, UK

[2]Cognitive Systems Group
Institute of Computer Science and Applied Mathematics
Christian-Albrechts-University of Kiel
Olshausenstraße 40
24098 Kiel, Germany

m.satpathy@reading.ac.uk, nils@siebel-research.de, drg@ieee.org

Abstract

Assertions had their origin in program verification. For the systems developed in industry, construction of assertions and their use in achieving and proving program correctness is a near-impossible task. However, they can be used to show that some key properties are satisfied during program execution. In this paper we first present a survey of the special roles that assertions can play in object oriented software construction. We then analyse such assertions by relating them to the Case Study of an automatic surveillance system. In particular, we address the following two issues:

- *What types of assertions can be used most effectively in the context of object oriented software? How can you discover them and where should they be placed?*
- *During maintenance, both the design and the software are continuously changed. These changes can mean that the original assertions, if present, are no longer valid for the new software. Can we automatically derive assertions for the changed software?*

Keywords. *Assertions; Software Maintenance; Object Oriented Systems; Case Study*

1. Introduction

Assertions are formal constraints on software system behaviour which are inserted as annotations in the source program text. They had their origin in program verification [4, 9]. Program correctness is usually defined in relation to a specification and assertions can encode the semantic properties of a specification. Using assertions to show program correctness is in general a non-trivial task and therefore it is hardly followed in practice. However, many key properties of a program can still be encoded in a simple assertion language. In such a scenario, if a program executes without any assertion violation, it can give some confidence about the program's correctness. As Meyer says: an imperfect solution is better than none [15]. In a sense, assertions test a program without using any test data.

Assertions are Boolean expressions which evaluate to *true* when the program state satisfies the desired constraints. If an assertion evaluates to *false* then it means that the program has entered into an inconsistent state and hereafter the program behaviour cannot be relied upon. Assertions thus provide a powerful tool mechanism for automatic run-time detection of software faults during development, testing and maintenance. They succinctly and unambiguously specify important properties (mostly safety properties) of a program in a way which is easily understandable. In addition, assertions are expected not to produce any side effect on the program state.

Assertions are widely used in the software industry today, primarily to detect, diagnose and classify programming errors during testing. They are sometimes kept in product code to forestall the danger of crashes, and to analyse them when crashes occur. They are beginning to be used by compilers as hints to improve optimisation. Assertions are usually compiled differently for test runs and for code that is shipped to the customer. In ship code, the assertions are often omitted to avoid the run time penalty and the confusion that would follow from an error diagnostic or a checkpoint dump in view of the customer [10].

In this paper we discuss the use of assertions in the context of object oriented (OO) software; especially, their roles in software maintenance. We first analyse the assertions added during the re-engineering of a medium sized system written in C++. During the course of its evolution it was re-engineered to improve its maintainability [19]. Developers used assertions at this stage with the intention of making debugging easier, and also while performing unit testing dur-

ing the reverse engineering process. While assertions were used to get tangible benefits, at that stage the developers did not have sufficient knowledge of the special roles that assertions could play in OO software. The following questions arose from their experience.

- What are the categories of assertions which could be used in OO software? How to discover them and where to place them? In the first part of the paper we present a survey of these issues and analyse them in the context of our case study.
- The system studied here was continually undergoing thorough design changes and therefore the code needed to be refactored and adapted. In this process, even if assertions are present in the code they may no longer be consistent with the next version of the code obtained through refactoring. So the natural question was: How can old assertions be adapted so they would be consistent with the changed code (or design). This issue is also analysed in the context of our case study.

1.1. Realisation of Assertions

Some programming languages like `Eiffel` [15] and the latest releases of `Java` provide assertions as language features while some others like `C` and `C++` provide assertions as language extensions. `C` and `C++` provide a simple assertion facility in the form of a predefined macro called `assert(bool expression)` defined in the file `<cassert>`. If the Boolean expression given as an argument to `assert` evaluates to `false` then the program aborts. The failure of an assertion usually triggers a dump of the program state which can be analysed to discover the source of the error(s). The Boolean expression within the assertion may involve function calls; care must taken that such function calls do not produce any side effect on the program state. When an assertion evaluates to $false$ the program aborts. Some authors (e.g. [11, 17]) have used special annotations to specify assertions in a virtual language which is pre-compiled by a pre-processor to generate appropriate statements in the underlying language. The translated statements may have local variables which do not modify the intended program state. This makes the assertions more expressive.

The system of our case study was developed in `C++`. In the following discussion we will therefore use `C++` syntax.

1.2. Compile time assertions

The assertions of the above types are sometimes called *run time assertions* because they are evaluated at run time. However, certain properties can be tested during compilation through the use of *compile time assertions*.

Compile time assertions are checked at compile time; violation of such an assertion results in a compilation error. This type of condition checking is usually implemented using macros. A typical application is for checks involving the `sizeof` operator. In the following popular example, the macro `COMP_ASSERT` can check at compile time whether two variables have the same (storage) size:

#defi ne COMP_ ASSERT(x) extern dummy$[(x)\ ?\ 1 : -1]$
COMP_ASSERT(sizeof$(x) ==$ sizeof(y));

The use of the `const` keyword in `C++` can also be viewed as a compile time assertion. It can be used to specify that a particular variable (e.g. a call-by-reference parameter to a method) may not be changed by the method. A method may also be declared as `const`, meaning that it may not change the object's state. A compiler can determine the breach of `const`ness properties and generate an error.

The organisation of the paper is as follows. Section 2 discusses the related work. Section 3 makes a survey of assertions in the context of OO systems, and the ones that were used in the system that was examined. Section 4 presents the object of our case study. We discuss what types of assertions were used in that software and their benefit. Section 5 discusses how assertions of a system can be adapted in face of design changes. Section 6 discusses the lessons learnt from this case study. Section 7 concludes the paper.

2. Related Work

Assertions had their origin in program verification [4, 9]. The axiom system developed by Hoare [9] uses assertions to show the correctness of Algol-like programs. Dijkstra's Weakest Precondition (WP) semantics views program design as a goal-oriented activity [7]. The requirement of a program can be stated as an assertion, and thereafter the use of the WP semantics could help program designers to construct in a systematic manner not only the program but the intermediate assertions.

Voas discusses the use of assertions for enhancing the effectiveness of testing for OO systems [24]. He points out that factors like information hiding and data encapsulation have a tendency to mask errors which in turn makes OO program testing less effective; such errors can be unmasked by injecting assertions at appropriate places.

The work of Korel and Al-Yami describes two kinds of assertions: Boolean and executable [11]. Boolean assertions are made out of expressions. Executable assertions have local variables which can be assigned values. They are like functions in a programming language returning Boolean values. The authors have used these two kinds of assertions for mechanical generation of test data.

Shimeall and Leveson have carried out empirical studies to compare various fault tolerant and fault elimination tech-

niques [22]. In their experiments run-time assertions were used. The authors observed that such assertions mostly detected faults relating to parameter reversal, substitution, over-restriction, loop conditions and data structures. Assertions did not detect faults like missing path, missing functionality, incorrect formula, operation ordering faults etc. The assertions used detected many faults which were not detected by other techniques (code reading, structural testing etc).

Rosenblum classifies assertions (in the context of maintenance and testing of systems developed in `C`) into two main categories: (a) Specification of function interfaces and (b) Specification of function bodies [17]. The constraints belonging to the first category take a *black-box view* of the function and they are specified independently of the function implementations. In contrast, assertions of the second category take a *white-box* view. Rosenblum's classification scheme does not include assertions like loop invariants. He has also observed that there is no clear correlation between the location of a fault and the location of the assertion that revealed it.

Hiller has used executable assertions for error detection and recovery in case of embedded systems [8]. In such systems, the internal signals need to satisfy several formal constraints; these are tested through executable assertions. An error is detected if any of these constraints is violated. The proposed error recovery mechanism forces an erroneous signal into its valid domain by assigning it a *best effort* value according to the parameters determined by the classification of the signal—a technique otherwise known as *forced validity*.

Meyer has developed the notion of *Design by Contract* in the context of OO software construction [15]. Every method has a precondition and a postcondition. These are expected to be satisfied at function entry and exit, respectively. Assertions can check such conditions. The design by contract principle states that the job of precondition checking must be done by the client code (except in some special circumstances like when input is supplied by a human user or it comes from an external system) and a method must check its postcondition before transferring control to the client code. This division of labour between the client and the server methods is the basic idea behind the design by contract principle.

3. A Survey of Assertions in OO Software

Assertions are meant to encode the key properties of OO programs. These can be classified into two categories: (i) intra-class properties and (ii) inter-class properties. Intra-class properties are expected to hold within a class, whereas inter-class properties are expected to hold when two classes interact through inheritance, method calls or aggregation.

3.1. Intra-class properties

Class Invariants (CIs)

Behind every class there is an underlying abstract data type (ADT) [15]. An ADT consists of *(a)* the name of the ADT which may have parameters, *(b)* a set of functions with their signatures, *(c)* a set of axioms to restrict the behaviour of functions and *(d)* preconditions on the functions. A class is an implementation of the underlying ADT; its functions become class methods, function preconditions become method preconditions and axioms become class invariants. Class invariants also include the implementation invariants; for instance, an ADT representing a stack may not have a stack size but its implementation has a size restriction on the size of the stack. The constructor is supposed to establish the class invariants; these are expected to hold thereafter before and after every other method invocation. Therefore, an assertion can be placed as a postcondition of the constructor to indicate that the initialization satisfies the class invariant. Further, similar assertions may be placed as precondition or postcondition of other class methods.

Method preconditions

Depending on the need, an assertion can be placed at a function entry that the entry point satisfies the class invariant. A precondition can also show the validity of or dependency between input arguments. In particular, such assertions can show:

- consistency between function arguments. For example, a method `add_vector(A, B, Result)` could have an assertion of the form `A.get_dimension() == B.get_dimension()`.
- appropriate bounds on input values. These include constraints such as (i) subrange restriction, (ii) a pointer variable not being `NULL`, or (iii) a string argument having a terminator character.
- validity of the context in which the function is called. For instance, a function is to be called only when the global constraint, say `initialised == true`, holds.

Postconditions

An assertion may be placed to show that the class invariant holds at a function exit. In addition, assertions at function exit points can show:

- dependency of return values on function arguments. As an example, after the execution of swap(x,y), it can be checked that the values of x and y were indeed swapped.

- effect of the call on the class state. For example, at the exit point of a call to the method `delete_record(record)`, an assertion can verify that the record has indeed been deleted.
- a call not having any undesirable side-effect on the class state. As an example, an assertion can show that a call does not modify a certain class variable.
- validity of return values. For example, assertions can check (i) a return pointer not being `NULL`, or (ii) a string variable having a terminator character etc.

Assertions within function bodies

Besides preconditions and postconditions, a function body needs to preserve some important semantic properties. A function body may contain a long sequence of control statements which are likely to introduce faults. We summarise some important properties within a function body which can be encoded in the form of assertions.

- Strengthening of condition in the `else` part (`default` case) of an `if` statement (`switch` statement): Let us assume, in an `if` statement, the condition is `(x > 100)`. It means that the `else` part is executed whenever the condition `(x <= 100)` holds. In such a case, we can strengthen the condition in the `else` part by an assertion depending on the context.
- Consistency of related data: Some sort of consistency relation may exist between two data; whenever one variable's value is within some range the other variable's value can be expected to be in some other range. Such consistency can be checked at various points inside a function body.
- Loop invariants: These are the constraints which must be true before and after every loop iteration, and they can be encoded as assertions. However, in practice, they are less often used [15, 17].
- Loop variants: This is a non-negative variable whose value decreases with each iteration. Such a constraint can be encoded as an assertion whose purpose is to show loop termination.
- Array index invariants: Assertions can ensure prior to an array access that the index lies within allowable limits. It is often recommended that these should not be deactivated in the code even after delivery.
- Assertions for reliability: To ensure that a key function works correctly, sometimes it is not enough to rely on the outcome of a single algorithm. As an example, speed is critical in the page layout code of the Microsoft Word and therefore it is written in assembly language. However, this code changes regularly as new features are added and it is highly likely that bugs creep into the assembly code. To catch the layout bugs, Word programmers write the C version of the layout code. Both routines run during development or debug mode and an assertion checks if there is a mismatch between the results of the two layout programs [13].
- Assertions to detect the impact of known errors: This type of assertion will be discussed in Section 4.2.
- Other assertions: An assertions can check common errors like division by zero, arithmetic overflow, underflow etc.

3.2. Inter-class properties

When two classes interact through inheritance, aggregation or method calls, they need to preserve some properties.

- *CI of a Derived Class:* If Inv_B is the class invariant of a base class, and Inv_D is the local CI of the derived class, the resulting CI of the derived class is $Inv_B \wedge Inv_D$ which must hold after initialization of the derived class and then before and after each method invocation. In `C++` the derived class constructor usually calls the base class constructor; therefore, assertions can be placed as postconditions of the derived and base class constructors that they satisfy necessary class invariants.
- *Indirect Invariant Effect* [15]: In presence of pointers, even if a class preserves its own class invariant, it all by itself cannot preserve certain key properties. Consider a double linked list, in which each node object has a forward pointer and a backward pointer. Then consider a property like: Whenever A points to B through its forward pointer, it means that B must point to A through its backward pointer. An assertion can be used to check that such a property is not violated.
- Assertions to satisfy the *Liskov Substitution Principle* (LSP) [12]: The principle says: functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it. In `C++` the types of the input parameters of a virtual function and its redeclaration must remain identical and the return type of the redefined function must be a subtype of the virtual function in the base class. Even if this holds, the LSP may still be violated if the base class virtual function has preconditions and post conditions in the form of assertions. In this context, we have Meyer's *Method Redeclaration Rule* [15]: if

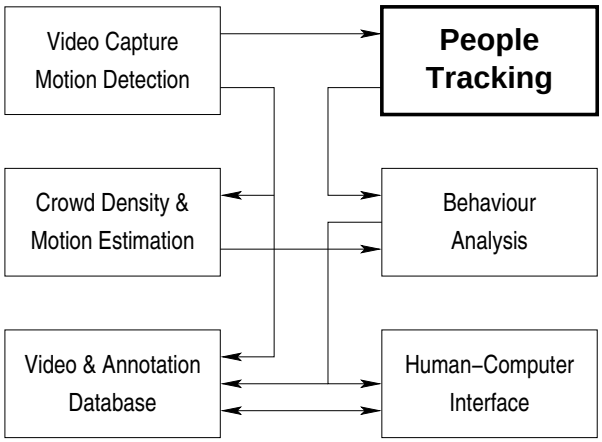

Figure 1. ADVISOR **System Overview**

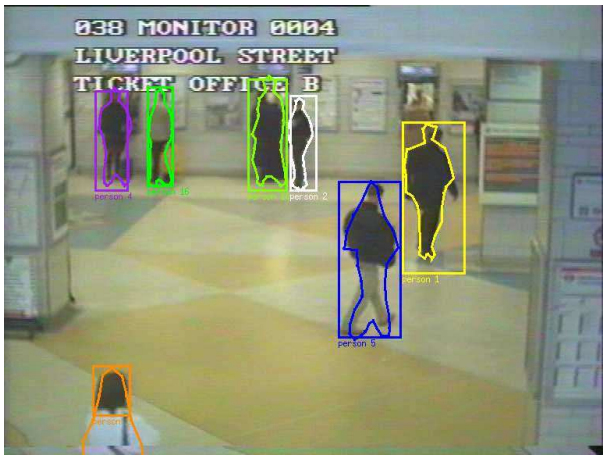

Figure 2. People Tracking Results

$\{Pre\}body\{Post\}$ respectively represents the precondition, body and postcondition of a base class polymorphic method and $\{Pre'\}body'\{Post'\}$ represents the same for a redeclaration of the same polymorphic method, then the following constraint must be satisfied: $Pre \Rightarrow Pre' \ \wedge \ Post' \Rightarrow Post$ ("$\Rightarrow$" means logical implication). Therefore, assertions in the redeclared method must respect the redeclaration rule or an assertion can be placed to check this correspondence.

- *Design by Contract* [15]: The Design by Contract principle advocates that the assertion meant for a precondition checking should be placed in the client code, and the assertion for postcondition need to be placed in the server code. We claim that this division of responsibility in some cases violates object oriented principles. For example, if a client method wants to pop an element out of a stack, then the client code must check that the stack is non-empty. In order to do this, the client either needs direct access to the private data of the stack class or is has to use some utility method for the check. However, for security reasons such a method might in some cases not be in the public interface. Therefore, in cases like these the strict use of Design by Contract principles violates the encapsulation principle of object orientation.

4. The Software Studied for this Analysis

4.1. Context and Brief History

The system studied for this article is the People Tracking subsystem of ADVISOR[3], an integrated system for automated surveillance of people in underground stations [20].

[3]Annotated Digital Video for Intelligent Surveillance and Optimised Retrieval

Figure 1 shows an overview of the system as it was originally designed[4]. ADVISOR was built as part of a European research project involving 3 academic and 3 industrial partners using a total man power of 36 person years. The task of the People Tracker is to automatically analyse images from a number of camera inputs. The system has to detect all the people in the image and track them in realtime as a stream of video images is continuously fed to the system. The image in Figure 2 shows an example of the visualised output from the People Tracker.

The People Tracker software (PT) over its lifetime of 10 years has been used in 3 different academic projects in 2 universities. We will focus on the most recent project, ADVISOR, in which the software has been adapted to form part of the next-to-market ADVISOR system. The original PT was written in 1993–1995 using C++ as part of a PhD research work [1]. The main focus during development was to provide functionality to carry out experiments on people tracking. Starting in 2000, the PT was adapted for its use within ADVISOR. This new application required a number of major changes at different levels. New functionality was added within the PhD project of an author of this article [21]. The software was completely re-engineered (see [19] for details) and during this time assertions were added.

4.2. Initial Use of Assertions in the PT

The initial PT of 1995 used no assertions. However, some explicit checks (e.g. array bounds) were performed. Within ADVISOR, 175 run-time assertions and 367 compile-time assertions (const keyword) were added both to existing and new code. Wherever appropriate, existing checks were changed to assertions. The assertions

[4]The design was changed later but this is the state examined.

mostly checked (a) that a function was called in the right context, (b) an array index lying in the right subrange, (c) a pointer being non-null at a function entry or exit; and (d) the consistency of related data within a function body. All these fall into the classification scheme of Rosenblum [17].

A New Type of Assertion

In addition to the categories above, a new type of assertion was used in PT which we term *Assertions to Detect the Impact of Known Errors*. The PT contains at places incomplete implementations of algorithms and other potential sources of problems. This is due to limited resources available during maintenance—a common problem in the software industry. Assertions are used to direct the attention of maintainers to those incomplete implementations which create difficulties.
Example: A method handles a large amount of data of different types. The implementation covers most of the possible combinations of data values of a number of variables, but not all. This means that if a special combination of data values is encountered the program does not know what to do. In "normal" operation this special case does not arise. Therefore a low priority is assigned to the task of correcting the problem. An assertion is added at this stage to determine whether the situation sometimes occurs. If the assertion fails, the task of fixing the problem will be assigned a higher priority[5].

Benefits of the Initial Use of Assertions in PT

The developers found that assertions significantly reduced debugging effort when detecting bugs. Instead of replacing debugging output and the use of a graphical debugger, assertions were a natural *addition* to these mechanisms.

Out of the 175 run-time assertions in the code about 10 ($5.7\,\%$) failed during development and testing. The developers also reported that assertions, when placed well, were helpful in determining not only the existence but also the location of the error. We present two such cases here.

- Range checks: When assertions for range checks fail it is easy to determine where the variable was last changed. Automated search of variable names in source files and "watch" facilities in modern debuggers make this search easy. Furthermore, let us assume that modification of a variable has led to the violation of a subrange constraint. Then let it be the case that the variable was modified along two different paths leading to the point of assertion violation. Then more assertions could be placed along both the paths to narrow down the slice of the program that caused the error.

[5] In this particular case the assertion never failed even after many hours of program run time and several 10,000 evaluations of the assertion.

Assertion Category	# cases added
Compile time assertions (`const`)	367
Preconditions	73
Postconditions (in constructors)	80
Postconditions (in other methods)	46
Loop invariants	not used
Loop variants	2
Cases for LSP	3
Assertions after qualified calls	40
Impact of Known Errors	20
Strengthening of `if`/`case` cond.	15
Range checking	84
Consistency of related data	45

Table 1. Types of assertions in the PT

- Consistency checks: Similarly, using tools like debuggers and development environments often allowed to locate the bug easily using the failed assertions as a starting point. However, the developers noted that consistency checks, more than other assertions, require extensive domain knowledge to be used effectively.

4.3. Use of OO-specific Assertions in PT

We have added OO-specific assertions discussed in the previous section to the PT. Table 1 shows the number of assertions of various categories which are now in the PT. The PT contains 408 run-time and 367 compile-time assertions in its 47,457 lines of `C++` code. Assertions were inserted into 46 out of the 186 source files. These files make up 24,631 lines of code (52 % of the total code). Considering its size the system contains few assertions. One reason is that this is an ongoing work and we are still in the process of adding more and more assertions of the new categories and studying their impact on the system behaviour. We have not yet added loop invariants; the reason is that they are hard to express in terms of simple assertions. However, we have observed that by adding additional operations (especially query operators), we can express the loop invariants and variants with ease.

5. Adapting Assertions under Design Changes

In OO programming, behaviour-preserving source-to-source transformations are called refactorings [14]. The granularity of refactoring can be high like splitting a large class into a base and derived class pair, or the granularity can be low like creating a new instance variable. Opdyke has taken the approach that any high level refactoring can be implemented in terms of several low-level primitive refactorings [16]. Such applications are compositional in the sense that if each of the primitive refactoring is correct

then it would imply that the high level refactoring is correct. Banerjee and Kim have defined a set of invariants which must be preserved with a view to making the refactoring transformations behaviour-preserving [2]. Opdyke has identified preconditions (or enabling conditions) which must hold so that a refyactoring step satisfies the invariants. Roberts has augmented Opdyke's refactoring steps by adding postconditions to the transformations [18]. Van Gorp et al. [6] have discussed how the pre- and postconditions of refactorings could be expressed in the Object Constrained Language (OCL) in a UML framework [23]. Mens and Tourwe have observed that even the simplest design changes require a large number of primitive refactorings; therefore, to increase the scalability and performance of a refactoring tool, frequently used sequences of primitive refactorings are combined into composite refactorings and then applied atomically [14].

Software engineers usually think about refactorings at the design level though at the same time they must be aware of the detailed code-level issues [6]. Therefore, a design should always be consistent with the code. In our case, we have encoded important intra-class and inter-class properties in OCL and placed them in the class diagram. While performing refactorings, we use such properties to generate consistent assertions for the refactored code. In the appendix, Figures 3, 4 and 5 show the specification of class properties using OCL.

If a source program has been annotated with assertions, and the code is subjected to refactoring transformations, then it is natural that the original assertions would no more be consistent with the refactored code. In this article, we take the view that, along with the refactoring rules, additional rules could be used either to recover or transform the original assertions so that the new assertions are consistent with the refactored code. In the following, we will cite two refactoring steps and illustrate how rules attached to them could recover (or generate) consistent assertions for the refactored code.

Refactoring: Inline method

In this refactoring, a method's body replaces the method calls. Let a method $method1(a, b, c)$ have *Pre* and *Post* as the pre- and the post conditions respectively. If the method body replaces a call to $method1(expr_1, expr_2, expr_3)$, then the rules:

- At the point of the call, prior to the code segment which replaces the call, put the assertion $Pre([expr_1/a] \;||\; [expr_2/b] \;||\; [expr_3/c])$. Here $e_1[e_2/a_2]$ means substituting all occurrences of argument variable a_2 in expression e_1 by the expression e_2 and || means parallel substitution. What the above expression means is that the occurrences of a, b and c in the predicate *Pre* are simultaneously substituted by the expressions $expr_1$, $expr_2$ and $expr_3$ respectively.
- Immediately after the code segment that replaces the call $method1(expr_1, expr_2, expr_3)$, one can put the assertion $Post([expr_1/a] \,||\, [expr_2/b] \,||\, [expr_3/c])$.

Refactoring: Replace Conditional with Polymorphism

In the following, the behaviour of an object of class EdgeDetector varies depending on whether the member variable `search_method` has the value NEAREST, MAXIMUM or STATISTICAL. In such a case, a base class is created with `find_edge()` as an abstract method. And the derived classes NEARESTEdgeDetector, MAXIMUMEdgeDetector and STATISTICALEdgeDetector will have specialised polymorphic versions of the method `find_edge()`. In the appendix, we show the refactoring of this class. Figures 3 and 4 show the original class EdgeDetector and the case after the refactoring, respectively. The relevant assertions have been shown in the OCL.

```
typedef enum {NEAREST, MAXIMUM, STATISTICAL} category;
class EdgeDetector{
      fbat max _scale;
      fbat curr ıı _val;
      category search_method;
      EdgeDetector(...) {...};
      edge_status fi nd_ edge (){
      switch (search_method){
            case NEAREST: .... ;
            case MAXIMUM: .....;
            case STATISTICAL: ....;
            } }
};
```

For the refactoring of the above type, apply the following rules to obtain the assertions for the refactored classes. In the Appendix, we show how through the application of such rules, we recover assertions for the refactored code.

1. The class invariant of the new base class remains the same as that of the original class.
2. Each of the derived classes corresponds to a particular case of the conditionals in the original class; therefore, each such class will have a CI stating that the category type remains constant in the class (e.g. `search_method == MAXIMUM` is a CI of the class MAXIMUMEdgeDetector). Accordingly, the constructors in the derived classes can have postconditions to check that this invariant holds.
3. The postcondition of the polymorphic function in the derived classes remain the same as the postcondition of the method containing the conditionals.

6. Discussion

We have learnt the following from our case study.

Assertions in OO Software. In addition to the types of assertions used in non-OO languages like C, OO languages require some special kinds of assertions. Viewing a class as the implementation of an ADT helps one to discover some intra class properties like the class invariant, operation pre- and postconditions and constraints over the member variables in a systematic way. The following types of assertions are particularly important for OO programs.

- *Checking the class invariant:* Constructors through their postconditions can ensure that the class invariants are satisfied. Such a checking should always be followed in practice unless the class invariant is too trivial. This type of assertion checking is particularly useful during the refactoring phase. For example, when a new base or a derived class is created, care needs to be taken as to how, both the base class and the derived class constructors together preserve the class invariants of the derived class; assertions in constructors are useful in this case. Constructors can also perform checks on input parameters through preconditions.
- Preservation of the LSP is crucial when a derived class is expected to have an *IS-A* relationship with the base class. We have used Meyer's *method redeclaration rule* to preserve the LSP. So it is our experience that use of assertions to preserve this rule should always be used along with polymorphism. This brings in an added clarity in understanding polymorphic functions.

Design by Contract. As we have pointed out in Section 3.2 the Design by Contract principle compromises a basic principle of object orientation. Therefore, checking of preconditions should be done by the server code whenever checking them by the client amounts to accessing private/protected data. Furthermore, Meyer advocates always to use assertions after qualified calls. However, such assertions on many occasions turn out to be postcondition checking.

Consistency between related data. Assertions to check the consistency between related data is an effective way for detecting bugs in a method; however, to use such assertions effectively one needs to have good domain knowledge.

Use assertions to detect the impact of known errors. When code contains known bugs or incomplete implementations of functionality these are sometimes not corrected because of lack of resources. In order to detect the case where a problem may arise from these conditions, appropriate assertions can be added to the code. This has been found to be useful in the project examined here.

Use assertions to test the system integrability. The `ADVISOR` system consisted of six subsystems which were developed at geographically different places. In such a scenario, a subsystem is usually checked by generating simulated data. However, after integration it is necessary to check that all the assumptions about the interfaces and the incoming or the outgoing data have really been addressed. Assertions are a nice mechanism to check this.

Consistency between design and implementation. While refactorings are performed over source code, decision about them are taken at the design level. As we have discussed in the previous section, key properties can be encoded in the design which could be used to generate important assertions. Therefore, it is imperative that the design should remain consistent with the implementation.

Extreme Programming. Refactoring is integral to the approach of *Extreme Programming* [3]. Therefore our approach of deriving assertions for the refactored code (or design) from the previous code (or design), we believe, is an effective one.

7. Conclusion

In this paper, we briefly presented the re-engineering of the People Tracking subsystem of the surveillance system `ADVISOR`. Some commonly used assertions were used for debugging purposes but without detailed planning. To use more effective assertions, we then made a survey of assertions meant for OO systems. Using these new assertions we have made some important observations.

During maintenance of OO systems, refactorings take place at regular intervals which make the already used assertions obsolete. In this context we outlined some rules which could be attached to refactoring steps (primitive or composite); such rules can recover assertions which are consistent with the refactored code. We have the following plans for our future work:

- We are still in the process of studying the impact of all assertions that we have added to the PT subsystem.
- Our work of generating assertions by attaching rules to refactoring steps is at an initial stage. Our plan is to define rules for all important refactorings and to integrate the rules in a refactoring tool.

Acknowledgements

We wish to thank Profs. Rachel Harrison, Gerald Sommer and Steve Maybank for encouragement and support. Thanks also to the anonymous referees whose comments improved the presentation of the paper. Daniel Rodríguez wishes to thank the Spanish CICYT for supporting part of this work under the Argo Project (TIC2001-1143-C3).

References

[1] A.M. Baumberg. *Learning Deformable Models for Tracking Human Motion*. PhD thesis, School of Computer Studies, University of Leeds, October 1995.

[2] J. Banerjee, W. Kim, H. Kim and H.F. Korth, Semantics and Implementation of Schema Evolution in Object-Oriented Databases, ACM SIGMOD Conference, 1987.

[3] K. Beck and M. Fowler, *Planning Extreme Programming*, Addison Wesley, 2001.

[4] R. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, XIX American Mathematical Society, 1967, pp. 19–32.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[6] P. van Gorp, H. Stenten, T. Mens and S. Demeyer, Towards Automating Source-Consistent UML Refactorings, Proc. of the Unifi ed Modelling Language Conference, 2003.

[7] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[8] M. Hiller, Error Recovery Using Forced Validity Assisted by Executable Assertions for Error Detection: An Experimental Evaluation, DSN 2000, June 2000, pp. 24-31.

[9] C.A.R. Hoare, An Axiomatic basis for computer programming, Communications of the ACM, Vol. 12 (10), October 1969, pp. 576–580,583.

[10] C.A.R. Hoare, Assertions: a personal perspective, website: `http://research.microsoft.com/~thoare/`

[11] B. Korel, A.M. Al-Yami, Assertion-Oriented Automated Test Data Generation, Proc. of the ICSE, 1996, pp. 71-80.

[12] B.H. Liskov and J.M. Wing, A Behavioural Notion of Subtyping, ACM TOPLAS, 16(6): 1811–1841, 1994.

[13] S. Maguire, *Writing Solid Code*, Microsoft Press, 1993.

[14] T. Mens and T. Tourwe, A Survey of Software Refactoring, *IEEE Trans. on Software Engineering*, Vol. 30(2), February 2004, pp. 126–139.

[15] B. Meyer, *Object Oriented Software Construction*, Prentice Hall, 1997.

[16] W.F. Opdyke, *Refactoring Object Oriented Frameworks*, Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

[17] D.S. Rosenblum, A Practical Approach to Programming with Assertions, IEEE Transactions on Software Engineering, Vol 21(1), 1995, pp. 19-31.

[18] D.B. Roberts, *Practical Analysis for Refactoring*, Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

[19] M. Satpathy, N.T. Siebel, D. Rodríguez, Maintenance of Object Oriented Systems through Re-engineering: A Case Study, Proc. of ICSM'02, Montréal, pp. 540–549.

[20] N.T. Siebel and S.J. Maybank. Fusion of multiple tracking algorithms for robust people tracking. In *Proceedings of ECCV 2002*, København, May 2002, pp. 373–387.

[21] N.T. Siebel. Design and Implementation of People Tracking Algorithms for Visual Surveillance Applications. Ph.D. thesis, Department of Computer Science, The University of Reading, Reading, UK, 2003.

[22] T.J. Shimeall and N.G. Leveson, An Empirical Comparison of Software Fault Tolerance and Fault Estimation, IEEE Transactions on SE, Vol. 17(2), February 1991, pp.173-182.

[23] J. Warmer and A. Kleppe, *The Object Constrained Language (2nd Edn.)*, Addison Wesley, 2003.

[24] J. Voas, How Assertions can Increase Test Effectiveness, IEEE Software, March/April 1997, pp. 118-122.

Appendix

Refactoring: Replacing Conditional with Polymorphism

```
class EdgeDetector {
/* CI: max_scale ≥ 1.0   &&   0 ≤ curr_u_val ≤ 1.0 */
      fbat max_scale;
      fbat curr_u_val;
      category search_method;
public:
      EdgeDetector(category c) {...};
      virtual edge_status find_edge(...) = 0; /* abstract method */
}

class NEARESTEdgeDetector : public EdgeDetector {
/* local CI:: search_method == NEAREST */
public:
      NEARESTEdgeDetector(CATEGORY c) : EdgeDetector(c) {
/* postcondition: search_method == NEAREST*/
      ...
      }
      fbat find_edge(...){
/* precondition: search_method == NEAREST*/
            ....
      }
}
```

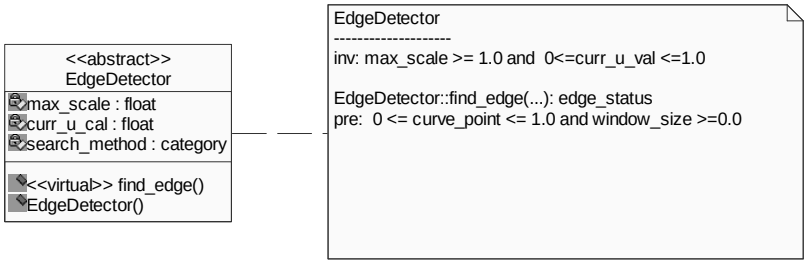

Figure 3. Class EdgeDetector: before refactoring

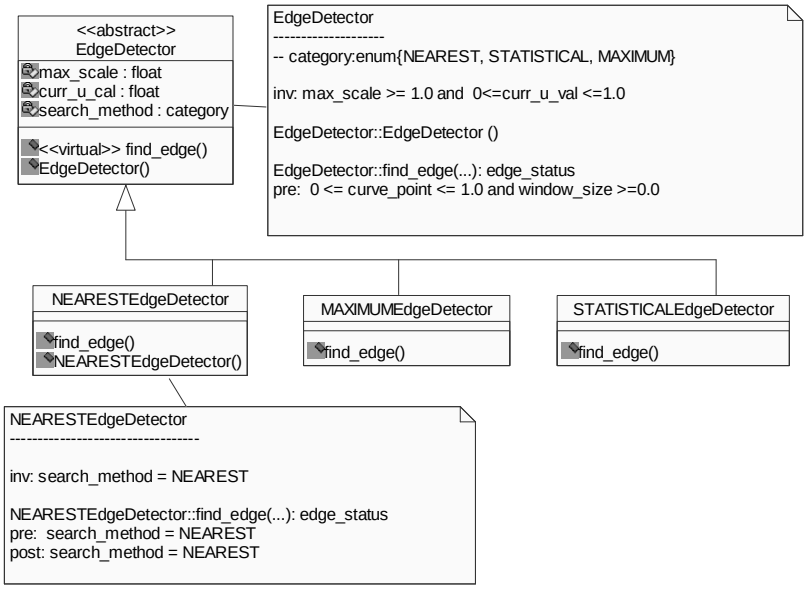

Figure 4. Refactoring: Replacing conditionals with polymorphism

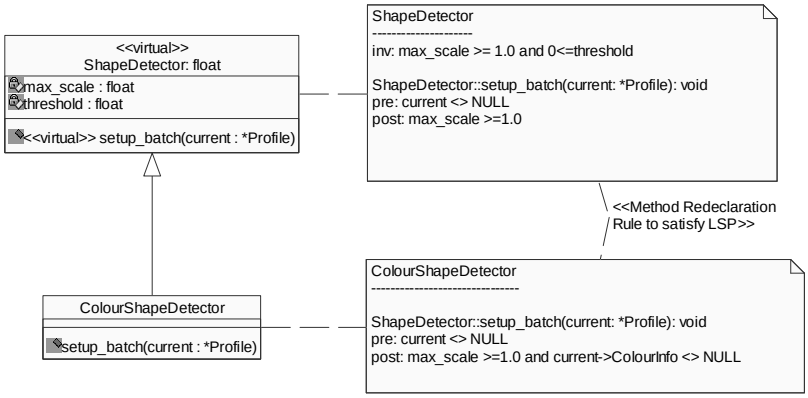

Figure 5. Use of Method Redeclaration Rule to satisfy Liskov's Substitution Principle