A Comparison of Defensive Development and Design by Contract™

Donald G. Firesmith
FiresmithD@AOL.com

Abstract

This paper briefly defines and discusses assertions and their uses before summarizing Design by Contract and Defensive Development. This provides a foundation for the following comparison of their similarities as well as their respective strengths and weaknesses. The paper concludes by arguing that Defensive Development is superior to Design by Contract, largely because of how they differ in assigning the responsibility for checking and ensuring preconditions.

1. Introduction

Assertions are a class-level, quality assurance technique used to address the following important problems:

- **Abstraction**. Because each class and type (e.g., Java interface) should model a single abstraction, it should capture the essential characteristics of the concept being modeled while ignoring the inconsequential or diversionary details. Although each class should also be given an English definition of the concept it models, the resulting informal narrative text can be vague and ambiguous. Even if its responsibilities for doing, knowing, and enforcing are also documented, they are still captured in narrative English text. How can one unambiguously document the abstraction of a class or type?
- **Correctness**. How do you know that a class or type is correct? How should this correctness be enforced? A relatively formal, yet practical, technique is needed to ensure that a class or type correctly captures its abstraction.
- **Encapsulation**. Encapsulation is the combination of localization and information hiding.
  - **Localization**. How should the specification (the 'what' and 'why') and implementation (the 'how') of a class be localized? Unless the specification is localized with the implementation, it becomes increasingly unlikely that they will remain consistent as the design and software are iterated and maintained. However, the specification should clearly be separated from the resulting implementation. Otherwise, it becomes hard to know whether the software meets its requirements (i.e., the implementation fulfills the specification). How does one localize defect detection and handling software with the associated operational software that contains the defects without mixing the two and thereby cluttering up the software?
  - **Information hiding**. Information hiding makes it harder and less efficient to determine from outside an object whether or not the abstraction of the object has been violated. The object should not have to export read and write accessors just for testing purposes because this decreases encapsulation and thus increases coupling. Such accessors may also be misused as a trapdoor for purposes other than testing.

- **Inheritance**. How can one ensure that inheritance is used properly to capture the "a kind of" specialization relationship between a child and its parent(s)? How can one avoid the loss of polymorphic substitutability? How can one ensure that test software is inherited as well as operational software?
- **Exception handling**. When should exception handling be used? How can developers avoid misusing it as a glorified "goto" (or rather "come from")?
- **Defensive development**. It is important that an object protect its abstraction from violation no matter what messages are sent to it, what exceptions are raised to it, or in what order these interactions occur. Moreover, if the abstraction of an object is violated, it is important that this violation be immediately recognized so that appropriate action can be taken. This action could be to notify the developer during testing so that debugging can occur or to ensure the robustness of the delivered objects.
- **Robustness**. More and more applications are mission or safety critical. It is insufficient if the application only works under normal conditions. Applications must also continue to function appropriately, even with bad input or the failure of controlled hardware. Such applications often have severe requirements concerning reliability and operational availability. Such systems must be fault tolerant and therefore self-testing and self-correcting. How should such software be built in order to meet required levels of robustness, reliability, operational availability, and fault-tolerance?
- **Testing**. How can testability be maintained when encapsulation inherently makes classes less observable and controllable (and therefore less testable)? What design for testability techniques for classes provide the most testability? How can objects be made self-testing so that the defect that caused the failure is identified when it happens so that debugging can be performed? Information hiding also emphasizes blackbox testing that does not depend on knowledge of the hidden implementation of the class. What should be the basis for class-level blackbox testing?
- **Reuse**. How can we be know that a class is suitable for reuse without spending an inordinate amount of time researching its implementation? How do we know that it will perform as advertised when reused in a new environment?

2. Assertions

An **assertion** is a rule in the form of a condition that must be true at certain times during the execution of the software. An assertion is a Boolean expression that constrains certain properties of the software. When properly used, assertions form an essential part of the formal specification of a class or type, documenting the required behavior of its instances.

Typical examples of assertions include:

- The value of an attribute or parameter must be of a certain type (strong typing) or fall within a certain range.
- The multiplicity of a certain association must fall within a given range (e.g., a certain link must not be void).
- A certain formula describes the value that must be returned by an operation.
- An operation must raise a certain kind of exception (strong typing) under certain circumstances.

- The valid states of an object are defined in terms of certain values of certain attributes.

When properly used, assertions provide the following key benefits:

- Assertions capture business rules and the responsibilities for enforcing them.
- Assertions simplify operational code by separating it from rule checking and rule violation handling code.
- Assertions increase understandability by capturing the designer's intent and formally specifying the protocol of the class or type.
- Assertions more formally specify the abstraction than do narrative English comments.
- Assertions increase quality and reliability. By specifying the assertions of a class, developers take a major step towards ensuring that the class actually meets its specification.
- Assertions monitored at runtime form a basis for:
  - Systematic class-level testing and debugging.
  - Ensuring robustness.

Assertions come in the following major kinds:

- **Protocol assertions**. These assertions are visible to senders and document the protocol of the associated class or type:
  - **Invariants** – define the valid states of an object that exist before and after the correct execution of its visible[1] operations[2].
  - **Preconditions** – define the conditions that must be true before a specific operation can execute correctly.
  - **Postconditions** – define the conditions guaranteed to result from the correct execution of a specific operation.
- **Implementation assertions**. These assertions are hidden from senders and constrain the implementation of the class or type:
  - **Ad hoc assertions** – define a condition that must be true at an arbitrary point in some operation.

Because of encapsulation, protocol assertions should not involve private properties of the object; instead, they should be specified in terms of its logical properties (i.e., externally meaningful concepts). The ability to use precisely defined queries to capture logical properties provides great power to assertions because the implementation of these queries can be arbitrarily complex (e.g., quantifiers such as "there exists" and "for all"). However, information hiding suggests that such queries should be used primarily for assertion checking because arbitrary use by senders of the object increases coupling. The Law of

---

[1] A visible operation is visible to some external class. Different implementation languages recognize multiple kinds of visibility. For example, "visible" in Java means one of the following: public, protected, and package; hidden thus means private.

[2] Whereas preconditions and postconditions can be used with non-object-oriented models and software (e.g., use cases), invariants only make sense when applied to objects.

Demeter[3] also suggests that assertions should not include the logical properties of logical properties.

Thus, assertions can be Boolean expressions involving the following:

- Logical properties of the object.
- Message parameters.
- Logical properties of message parameters.
- Exceptions handled.
- Logical properties of exceptions handled.

3. Design by Contract™

Design by Contract is the collaboration-level specification and design approach developed by Bertrand Meyer, the creator of the Eiffel programming language. Thus, Design by Contract is directly supported in the Eiffel language and a fundamental part of the Eiffel mindset. Design by Contract views each interaction between two objects as if it were a legal contract between a customer and a service provider. Each such contract documents the respective obligations and benefits of each party; whereby the obligations of one party result in benefits for the other party. By analogy, the interaction between the operation sending the message (the customer) and the associated operation implementing the message (the service supplier) can be viewed as a contract between them specified in terms of assertions. For example, consider the pop operation of a bounded stack. The associated contract's obligations and benefits are documented in the following table:

BoundedStack.pop()	**Obligations**	**Benefits**
Customer (message sender)	Satisfy following preconditions: • Stack not empty.	Obtain from postconditions: • Top item is returned. • Stack is properly updated.
Supplier (receiver operation)	Satisfy following postconditions: • Stack is not full. • New size equals old size minus 1.	Obtain from preconditions: • Can assume that the stack is not empty; no need to check.

The metaphor of contracting can be extended to subcontracting. A superclass uses *inheritance* to subcontract out some of its obligations to its subclasses. A sender uses *delegation* to subcontract out some of its obligations to its receivers.

Design by contract is based on the following principles and assumptions:

- A run-time violation of an assertion is caused by a fault (defect, bug) in the software.
- The customer (sender object) is responsible for establishing the precondition. Therefore:
  - A violation of a precondition implies a defect in the customer's code.

[3] The Law of Demeter (a.k.a., the "Don't talk to strangers" pattern) minimizes coupling and enforces encapsulation by restricting who should receive messages from an object. An object should only send messages to itself, its properties (i.e., attributes, parts, entries, members, and objects that it is directly linked to), the parameters of messages sent to it, and any exceptions that are raised to it.

  - Thus, every logical property appearing in the precondition of an operation must be visible to every customer of the operation so that the customers can verify/ensure the precondition.

- If the customer does not establish the precondition of the supplier, then the supplier can do anything it wants because the result of calling the supplier operation is undefined. The following are direct quotes from Bertrand Meyer, the inventor of Design by Contract:
  - "If the client's part of the contract is not fulfilled, that is to say if the call does not satisfy the precondition, then the class is not bound by the postcondition. In this case, the routine may do what it pleases: return any value; loop indefinitely without returning a value; or even crash the execution in some wild way." [2]
  - "The definition of class correctness leaves the routines of the class free to do as they please for any class that violates the precondition or the invariant [2]
- In concurrent software in which a customer cannot guarantee the precondition of the supplier because of race conditions, the precondition becomes a wait condition rather than a correctness condition. The supplier waits until the precondition holds before executing the requested operation.
- The supplier (receiver object) is responsible for establishing the postconditions and invariants. Therefore, a violation of a postcondition or invariant implies a defect in the supplier itself.

4. Defensive Development

Defensive Development [2] is a class-level specification, design, and implementation approach designed to defend an abstraction from misuse or bugs. Unlike Design by Contract, it places all responsibilities for ensuring the abstraction of a class on the class itself. It is interested in both correctness and robustness. It is based on the following important principles:

- Each class and type captures a single abstraction. As such, it is correct and complete to the extent that it properly captures all essential aspects of the concept it models while ignoring all unimportant or diversionary details.
- Because you cannot defend an abstraction that is not specified, assertions and their associated exceptions are used to formally specify the behavior of the abstraction. Because assertions and exceptions capture a critical part of the abstraction, they must therefore be localized with the abstraction they specify.
- Objects must be instantiated in a valid state (i.e., ensure its invariants). Otherwise, it is already too late to defend them because their abstractions were already violated on creation.
- To the extent practical, an object should defend itself by not allowing its abstraction to be violated (e.g., respond inappropriately, be put in an invalid state) regardless of:
  - What messages it receives.
  - What exceptions it handles.
  - In what order the interactions occur (i.e., its state).

- If an object's abstraction is violated, the object should recognized that fact and react appropriately:
  - Reestablish its valid state.
  - Use *appropriate* exceptions to warn its senders that it cannot respond appropriately.
- Senders are *not* responsible for establishing the preconditions of receivers; they often cannot be responsible if multiple objects are sending messages to the same receiver. Neither is it the responsibility of the receiver to establish its own preconditions. Instead, the checking of all assertions is the responsibility of the receiver class that specifies them, whereas the handling of assertion violations resides in both the receiver (reestablish invariants and throw exception) and sender (handle failure).
- Because concurrency may lead to race conditions between multiple senders attempting to use the same receiver, visible operations of concurrent classes must be synchronized (i.e., critical regions guaranteed to run to completion without interruption) if correctness is to be guaranteed.

5. Comparison of Defensive Development and Design by Contract

Defensive Development and Design by Contract have a great deal in common:

- They are both based on the use of assertions.
- They use the same kinds of assertions.
- They both consider the receiver responsible for ensuring invariants and postconditions.
- They both allow the raising of exceptions upon assertion violations.
- They both therefore derive many of the same benefits from assertions.

Because of how it assigns responsibility for ensuring preconditions and emphasizes exception handling, Defensive Programming is superior to Design by Contract for the following reasons:

- **Defensive Development improves reliability**. Reliability is the combination of correctness and robustness. Design by Contract is only concerned with correctness, whereas Defensive Development is also concerned with robustness. Although it may be OK in the legal world for one party to abandon a contract if the other party violates the contract, such behavior is neither acceptable nor practical for software. It is critical that software continues to function, even if one party violates the contract. Whereas Design by Contract would allow the receiver operation to do anything on violation of its preconditions, Defensive Development mandates that an appropriate exception be thrown, thereby allowing the sender to take appropriate actions. Defensive Development also supports robustness by providing more guidance on exception handling. Defensive Development therefore produces software that is more reliable and robust than software produced by Design by Contract.
- **Defensive Development decreases message coupling.** The two different techniques result in different numbers of messages being sent between objects:
  - Design by Contract usually forces the sender to first send a query message to the receiver object to determine if each precondition of a desired operation is true before sending the associated message. The counter argument that the sender need not send

queries in order to guarantee the precondition does not usually hold in either practice or examples provided by proponents of Design by Contract.

- With Defensive Development, however, the sender assumes that the preconditions hold when sending messages. Thus, there is no need to send query messages to check the precondition, and only in those rare occasions in which a precondition is violated is an exception raised and handled.
- By requiring fewer messages, Defensive Development causes less message coupling between classes than does Design by Contract.
- The sequence diagrams in Figures 1 and 2 use stacks to illustrate how Defensive Development and Design by Contract differ with regard to interactions.

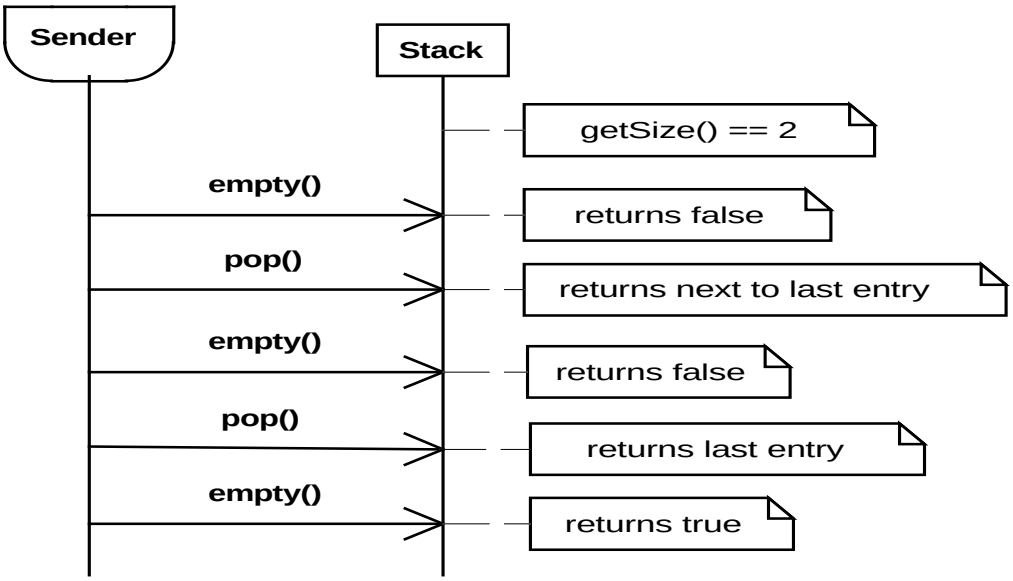

Figure 1. Interactions using Design by Contract

- **Defensive Development better supports encapsulation**. Defensive Development supports encapsulation better than Design by Contract:
  - With Design by Contract, the sender must ensure the receiver's preconditions. Thus, either the sender must send query messages concerning every logical property in the preconditions or the receiver must export the associated physical properties. Either the query operations must be public (instead of protected) or else (and worse) the associated physical properties must be public. Neither case maximizes encapsulation.
  - With Defensive Development, the receiver is responsible for ensuring its own preconditions. Therefore, many of the logical properties can be hidden if they only are used in the preconditions.
- **Defensive Development improves localization and cohesion.** Design by Contract violates the guideline "Never do for an object what an object can do for itself." Instead of having the supplier responsible for its own preconditions, the associated precondition checking code is scattered across all of the senders of the associated message. Defensive Development better localizes precondition checks with the operations having the preconditions, thereby improving the cohesion of the class.

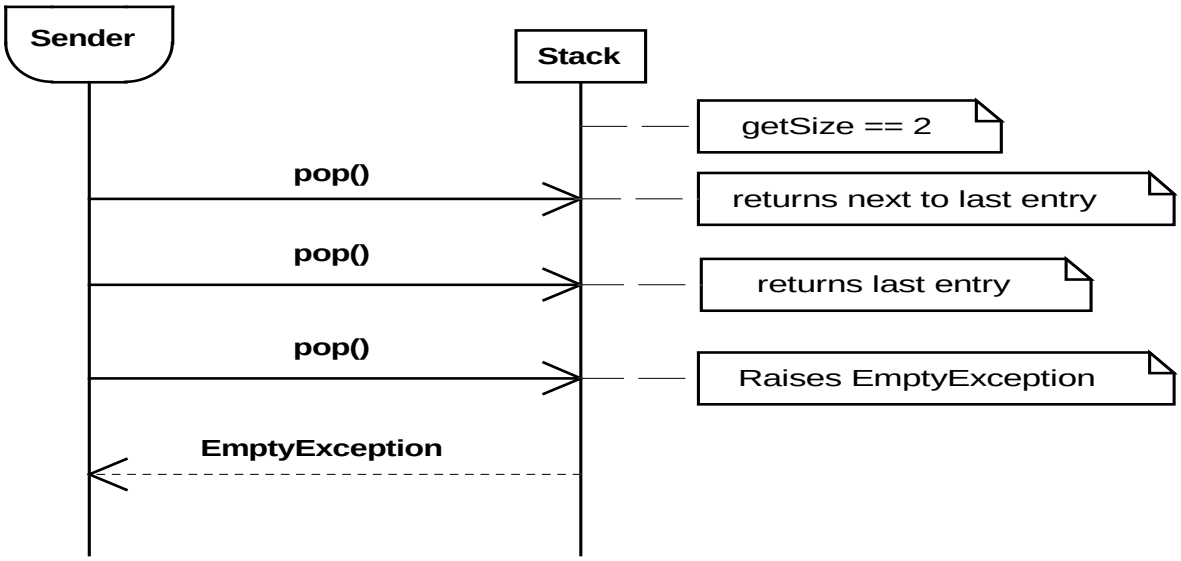

Figure 2: Interactions using Defensive Development

- **Defensive Development eliminates concurrency bugs.** More and more classes must exist in a concurrent environment. Design by Contract and Defensive Development greatly differ in how they handle concurrency:
  - Design by Contract would have the sender check or establish the precondition of the receiver. However, even if one sender were to establish the preconditions or determine that they were true, a second sender could send another message that causes a precondition violation before the first sender could send its intended message. Such race condition defects cause intermittent failures and are therefore very difficult to test for and debug. For this reason, Design by Contract only uses preconditions to ensure the correctness of sequential software; it uses preconditions to signify wait conditions for concurrent software.
  - Because the receiver of the message is responsible for monitoring its own preconditions and responding correctly, Defensive Development does not need to change the meaning or uses of preconditions when dealing with concurrent software.
  - The sequence diagrams in Figures 3 and 4 use bounded-stacks to illustrate how the behavior resulting from Defensive Development and Design by Contract differ greatly with regard to concurrency. Note that parallel lines signify that the senders and the bounded stack are concurrent (i.e., execute in *parallel* on their own threads). The shield on the bounded stack further indicates that it is thread safe. Note that Design by Contract requires the use of asynchronous messages, whereas Defensive Programming allows synchronous messages.
- **Defensive Development decreases complexity.** Design by Contract simplifies the supplier code (by not having it check the preconditions) at the cost of increasing the complexity of the code of each customer and increasing the number of interactions between the customer(s) and the supplier. Because a single supplier often has multiple customers that must each redundantly check the preconditions, Design by Contract results in a net increase in overall complexity.

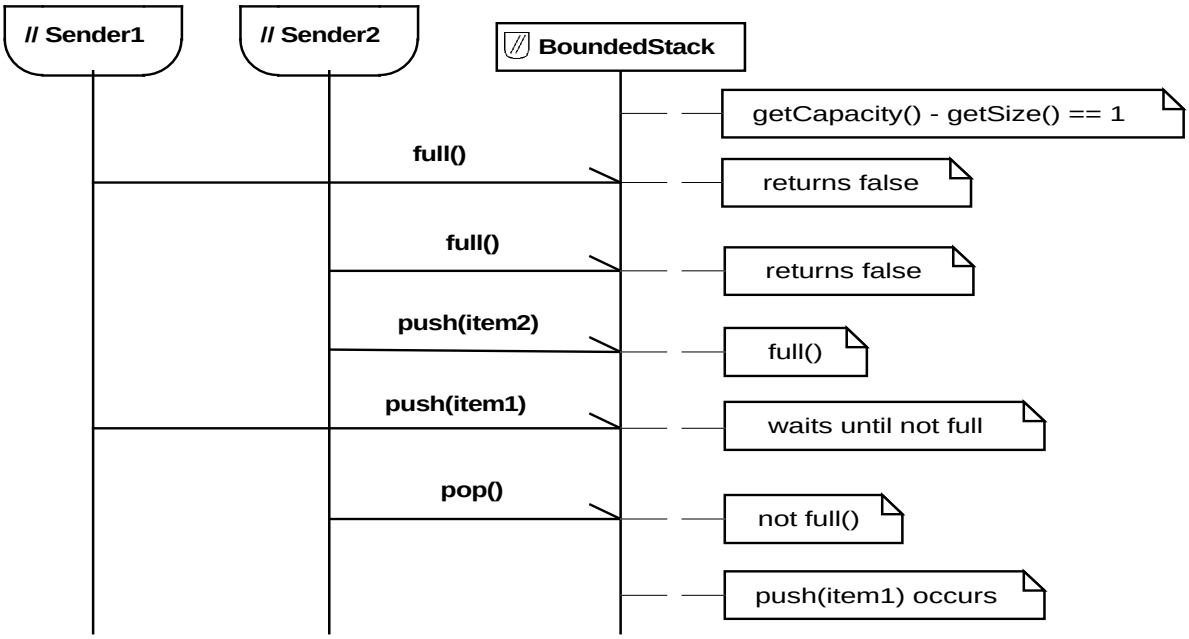

Figure 3: Concurrency using Design by Contract

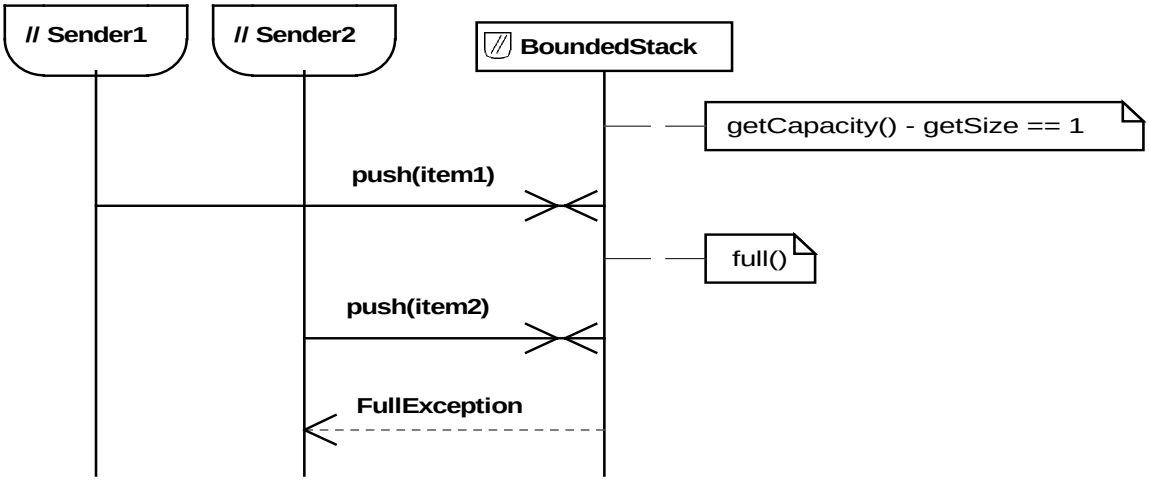

Figure 4: Concurrency using Defensive Development

- **Defensive Development eliminates redundant code by reusing precondition-checking code.** Whereas Defensive Development checks preconditions only once (at the beginning of the called operation), Design by Contract redundantly includes precondition checking code in each sender. Thus, Design by Contract increases overall code size and the costs of the associated maintenance. If the preconditions change, each sender must be found and modified in an identical manner; something that would be unnecessary if the receiver was responsible for enforcing its own preconditions.
- **Defensive Development increases maintainability**. It increases maintainability by decreasing complexity, eliminating redundant client code, and better separating normal from abnormal code.
- **Defensive Development increases performance**. Because assertions should only rarely be violated, exception raising and handling code will only rarely execute. By not

requiring (typically unnecessary) query messages between classes to check preconditions, Defensive Development significantly increases the performance of the resulting code. This is especially true when the instances of the classes belong to different processes on different processors.

- **Design by Contract is based on false assumptions**. Design by Contract is partially based on the following false assumptions:
  - Only one customer per supplier.
  - The violation of a precondition is a symptom of a defect in the sender. Such a violation could be the result of bad user input, or merely the fact that multiple concurrent senders are interacting via the same receivers (i.e., a race condition occurs).
  - The violation of a postcondition is a symptom of a defect in the receiver. This is an over simplification because such a violation could be the result of a failure of either a collaborator of the receiver or a hardware device that is controlled by the receiver. Thus, violations of postconditions may signify the robustness of the receiver rather than a defect.
- **Defensive Development increases consistency**. Defensive Development provides a single consistent way of handling all assertions (the receiver is responsible), sequential and concurrent classes treat preconditions the same way, and most commercial-off-the-shelf (COTS) class libraries are based on defensive programming (at least logically). Design by Contract does not.
- **Defensive Development provides greater separation of concerns**. Defensive Development physically separates normal processing code, code to detect assertion violations, and code to handle assertion violations (exceptions). However, Design by Contract commingles precondition violation detection code and normal processing code in the sender and does not emphasize the use of exception handling code.
- **Defensive Development produces specifications that are more complete**. By specifying the specific exception to be raised by each violated assertion, Defensive Development provides more complete specifications that does Design by Contract, which tends to emphasize assertions over their associated exceptions.

6. Conclusion

The proper use of assertions solves many important problems with the specification, design, implementation, and verification of classes and types. Whereas Design by Contract has greatly advanced software engineering and is the de facto industry standard approach for the use of assertions, Defensive Development (when properly based on assertions) provides all of the advantages of Design by Contract but also avoids many of Design by Contract's limitations, especially those associated with how to handle preconditions. This paper therefore argues that Design by Contract should be superseded by Defensive Development.

7. References

[1] Bertrand Meyer, *Object-Oriented Software Construction, Second Edition*, Prentice Hall, Englewood Cliffs, New Jersey, 1997.

[2] Donald G. Firesmith, "Pattern Language for Testing Object-Oriented Software," *Object Magazine, Vol. 5, No. 8*, SIGS Publications Inc., New York, New York, January 1996, pp. 32-38.