Design by Contract: An Overview
CSCI 5828

Michael M. Vitousek

University of Colorado at Boulder
michael.vitousek@colorado.edu

March 21, 2012
1 Introduction
   - Motivation and Introduction
   - Simple Example
   - Contract Overview
2 Coding with Contracts
   - Using Contracts
   - Types of Contracts
   - Computational and Protocol Contracts
   - Contract failures
3 Design by Contract
   - Design by Contract Methodology
   - History
   - Another Example
4 Conclusions
   - Conclusions
   - Resources and References
Outline

1 Introduction
   - Motivation and Introduction
   - Simple Example
   - Contract Overview

2 Coding with Contracts
   - Using Contracts
   - Types of Contracts
   - Computational and Protocol Contracts
   - Contract failures

3 Design by Contract
   - Design by Contract Methodology
   - History
   - Another Example

4 Conclusions
   - Conclusions
   - Resources and References
Motivation (I)

- One of the basic problems of software development: does our program do what we think it does?
- The first question is the essence of verification in software engineering, and along with validation, it is one of the most important questions of software development.
- There are multiple ways to verify software:
  - The most common method is testing, including unit tests, integration tests, etc.
  - A much more strenuous approach is proof via formal methods — an extremely strong method of verification.
Motivation (II)

Unfortunately, both these previous methods have shortcomings:

- It is difficult to build test cases that give full code, condition, and or path coverage
- And developing a full proof of a program can be extremely difficult, and proofs can have errors (especially if not mechanically checked)

We would like to have a method of verification that provides stronger properties than testing alone, but which does not require the effort and overhead of formal methods
Contracts

- Fortunately, such a method exists!
- Design by Contract, a.k.a. Code Contracts
  - Named in reference to enforceable legal contracts
  - Contracts are formal propositions (i.e. boolean expressions) about the behavior of a software system [7]
  - Contracts let users specify strong requirements about programs and program values
- Design by Contract typically puts contracts the behavior of individual methods or variables, but is very flexible
- Contracts complement testing — if a program enters a faulty state unforeseen by testing, contracts can reduce the impact of the fault
A Simple Example (I)

- Consider the following pseudocode:

  ```
  function calculateThreadCount(double blockCoeff)
      int numCores := getNumCores()
      return (numCores/(1 − blockCoeff))
  end function
  ```

- This program is the correct method for determining the number of threads to use in a concurrent program.
- But it makes several assumptions:
  - First, it assumes that `blockingCoefficient` is not a negative number or a number greater than 1; if it is, the program would behave in an unexpected manner.
  - Second, it assumes that `blockingCoefficient` is not equal to 1; if it is, the program will throw an exception or return some special “not-a-number” value.
In reality, we want to have stronger constraints on the behavior of this procedure than that which is specified by its code.

Specifically, we want:
- \( 0 \leq \text{blockCoeff} < 1 \)
- \( 1 \leq \text{function’s output} \)

The condition on the argument is called a *precondition*, and that on the return value is called a *postcondition*. We’ll expand on these definitions later.
A Simple Example (III)

- Contracts let us encode these rules in the definition of the program:

  **Require:** 0 ≤ blockCoeff < 1

  ```
  function calculateThreadCount(double blockCoeff) 
  int numCores := getNumCores() 
  return (numCores / (1 − blockCoeff))
  end function
  ```

  **Ensure:** output ≥ 1

- Design by Contract is the methodology of software development that holds that such conditions like these are crucial parts of programming good software
Behavior of Contracts

- Contracts allow code segments to be directly linked to their specifications in a robust way.
- The use of contracts does not prevent failures from occurring — contracts are checked at runtime, so it is possible to compile and run programs that will violate contracts.
- But contracts cause such invalid programs to fail in an expected manner, earlier rather than later, “blaming” the code that caused the problem.
- Contract violations at worst cause the program to cease execution, rather than proceeding in an unexpected way that could cause confusing — or dangerous — failures later in execution.
Outline

1 Introduction
   - Motivation and Introduction
   - Simple Example
   - Contract Overview

2 Coding with Contracts
   - Using Contracts
   - Types of Contracts
   - Computational and Protocol Contracts
   - Contract failures

3 Design by Contract
   - Design by Contract Methodology
   - History
   - Another Example

4 Conclusions
   - Conclusions
   - Resources and References
Using contracts

- Contracts are often integral parts of the source code of a program.
- Contracts tend to be easy to define and write if you know the expected behavior of a program.
  - Contracts are a verification tool — they help the user determine if the program behaves as expected.
  - They are not generally useful for validation.
- Not all languages support native contracts, but some that do include:
  - Eiffel
  - Spec# (based on C#)
  - Racket (based on Scheme)
- Other languages may have libraries that enable non-native use of contracts.
The most common form of a contract is an assert statement

- Format: “assert \( b \)” where \( b \) is some boolean value
- If \( b \) is true, then the program proceeds normally
- If \( b \) is false, then typically an exception is thrown
- If the assert doesn’t cause an exception, then the program can assume that whatever property is being tested by \( b \) holds thereafter (until the state of the program changes!)

- assert statements can be used in many languages, even ones that don’t generally support contracts (Java, C#)
Preconditions and postconditions are at the heart of design by contract

- Preconditions specify what kinds of input are expected
- Postconditions specify what kinds of guarantees are provided by the output

Pre- and postconditions can be seen as similar to type annotations on arguments and return types

- They both put constraints on the types of values that can be passed in and out
- But pre- and postconditions can reflect a much more refined set of values than can be expressed in most type systems
Invariants

While pre- and postconditions specify what has to be true about a program before and after the execution of some of its code, invariants specify what stays the same.

An invariant is a property that is true when its code is entered, stays true throughout its execution, and remains true after it terminates.

Invariants have a broader scope than other contracts:

- Preconditions specify what is true at the point that execution starts.
- Postconditions specify what is true when it terminates.
- Asserts specify something about an arbitrary point in code.
- But invariants are true at all points in its code.
Other contracts

- Pre- and postconditions are the most important contracts used in Design by Contract, but there are other kinds of contracts we might want to enforce
  - Non-null: we can require that variables not be null references
  - Side effects: we can limit or specify the ways that a method can effect global state
  - Exceptions thrown: what kinds of errors should be allowed to occur
Similarly to the case of concurrent program design, how Design by Contract is applied depends on what kind of program is being designed.

“Computational programs” are those which take an input, perform some kind of computation on it, and produce some kind of output.

Contracts for computational programs specify “what we are tying to achieve with the contract” [7].

- They put constraints on the kind of data that is provided as the input,
- specify what has to be true of the resulting output for the computation to be correct,
- and give invariants about the state and form of the intermediate computation.
Protocol contracts

- In contrast, interactive programs depend on interaction with external systems — a user, other programs, or other computers.
- The computational view of contracts is insufficient for this situation.
  - In interactive situations such as a computer game the notion of a “correct output” is unclear.
- Instead, *protocol contracts* specify how a program interacts with other entities in its context.
- Protocol contracts on I/O dependent programs won’t specify the “correct answer” of the program, but rather who it should communicate with, how, and when.
Substitutability

- Both computational contracts and protocol contracts are designed to allow *substitutability*\[7\]
  - Two methods, functions, modules, etc are substitutable when they have the same set of contracts on them
  - Therefore, they act essentially the same way, regardless of differences in implementation

- Two different sorting algorithms will still have the same overall contracts about their behavior

- Two different AIs for a computer game will still perform the same kinds of queries and actions
Design by Contract is often applied to object-oriented languages that have important notions of inheritance and subtyping.

Subclasses are allowed to strengthen postconditions on the methods they share with their superclasses.

This is natural — by strengthening the postconditions, subclasses maintain at least the requirements of their superclasses on what kinds of values are returned.

Less obviously, subclasses are allowed to have weaker preconditions.

This ensures that any argument to a superclass’ method which passes its preconditions will also pass the subclass’ preconditions, and maybe the subclass doesn’t require all the properties that the superclass does — not a problem!
Contract failures (I)

- Up until now, we’ve mostly been talking about what contracts guarantee.
- But it’s also important to understand what happens when contracts are broken.
- When the boolean proposition of a contract is false when executed, the program must not continue as if nothing has happened — it should “fail hard”.
  - This typically results in the termination of the program’s execution in some way.
  - Assertion failures are not meant to be recoverable.
  - E.g., in Java, “assert false” will cause an AssertionError, and the JDK documentation discourages trying to catch these errors [10].
Contract failures (II)

- This approach can seem extreme — why can’t we try to recover from a contract violation?
- The reasoning behind this can be compared to type errors:
  - In a statically-typed language like Java, if we try to pass a boolean into a function that expects a double, our program won’t even compile.
  - Likewise, when an int with value 42 is passed into a function whose precondition specifies that its arguments are in the range \([0, 9]\), we have performed an operation so nonsensical that the program should simply halt.
- However, we do want to know what code is responsible for the contract violation, in order to debug properly.
In order to achieve this, we use *blame tracking*. Blame tracking arises from the intuition that the point in code at which a contract violation occurs may not be the point at which a fault exists in the code. Instead, it may be in the function that called the failing function, or the function that called *that* function. Blame tracking can precisely trace these failures back to their origin. Simple blame tracking can be achieved using similar techniques to stack traces. In more complex situations, such as violations induced by type casts, more advanced techniques under active research need to be used[1].
The basic idea behind the Design by Contract methodology is that the first elements of code written for a method, class, or program should be its contracts.

In DbC, contracts are a critical step between understanding what a program should do and implementing a program that does it:

- Once the programmer understands what a section of code should do, he or she can write contracts for it.
- Once contracts have been written, it is more clear what the actual implementation should be.
Contracts therefore serve several complimentary roles in software development, in addition to their role in preventing dangerous program behavior:

- Contracts embed the specification of a program into its code
- Contracts aid in code reuse by making clear what kinds of context a section of code can be used in
- Contracts, if sufficiently readable, document the behavior of a program

Design by Contract is therefore complimentary to other methodologies and means of verification, including automatic test generation [6]
What contracts aren’t

- Contracts are great, but developers using Design by Contract shouldn’t overstate its abilities
- Contracts don’t prevent programs from entering states that violate its contracts, they just prevent such programs from proceeding
- Contracts don’t replace unit testing
  - Unit testing should examine the behavior of code when its assumptions are met
  - Contracts prevent those assumptions from not being met
- Contracts aren’t behavior-driven development
  - Although pre- and postconditions look something like Cucumber scenarios, they operate at different levels
  - BDD scenarios are high level, black box tests, while contracts enforce very specific, low level properties
The theory of contracts was developed from the system of Hoare Logic.

Hoare Logic is a system for reasoning about the behavior of imperative programs based on the preconditions and postconditions of program statements [5].

This logic was refined into a software engineering and programming language by Bertrand Meyer in the Eiffel language [3].

Advances in the use of design by contract and related techniques have been made by the PLT group in Racket [1], and by Microsoft in the Spec# language [2].
Another example (I)

- Let’s consider how Design by Contract suggests we develop a section of code.
- Say we want to write a simple function that takes two numbers \(a, b\) and returns an object containing members \(k, r\) where \(k\) is the quotient of \(a\) and \(b\) and \(r\) is the remainder.
- That is, it should perform the Euclidean division algorithm.
- We begin by considering the constraints inherent to this algorithm:
  - \(a\) and \(b\) need to be integers, and \(b\) needs to be nonzero.
  - \(k\) and \(r\) need to be integers, and the property of \(a = bk + r\) needs to hold.
Another example (II)

- The constraint that the arguments and returned values need to be integers is best handled by the type system.
- But the other constraints are a job for contracts!
- Let’s come up with some pre- and postconditions:

  **Require:** \( b \neq 0 \)

  **function** `divisionAlgorithm(int a, int b)`

  //Unimplemented

  **end function**

  **Ensure:** \( a = b \times output.k + output.r \)

- Now that we have the pre- and postconditions, we can write the function itself such that it matches those specifications...
Another example (III)

...as such:

Require: \( b \neq 0 \)

function divisionAlgorithm(int \( a \), int \( b \))

\[
\text{int } r := a \ % \ b \\
\text{int } k := (a - r)/b \\
\text{return } \{k, r\}
\]

end function

Ensure: \( a = b \times output.k + output.r \)

Therefore, our initial development of contracts has:

- Helped us write the function
- Ensured that our function won’t have silent failures that propagate to other parts of the program,
- And documented the behavior of the function, improving readability and reusability
Outline

1 Introduction
   - Motivation and Introduction
   - Simple Example
   - Contract Overview

2 Coding with Contracts
   - Using Contracts
   - Types of Contracts
   - Computational and Protocol Contracts
   - Contract failures

3 Design by Contract
   - Design by Contract Methodology
   - History
   - Another Example

4 Conclusions
   - Conclusions
   - Resources and References
Contracts are elements of code that enforce constraints and specifications about the runtime state of a program

- Contracts can be adapted to lots of different situations, including both interactive and computational programs
- Contracts typically test low-level properties at method, class, or module boundaries
- When contracts are violated, the program should fail-stop to prevent non-conforming data from spreading through the program

Design by Contract uses contracts throughout the design process

- Contracts bridge the gap between the coder's mental understanding of what a method should do, and the implementation of the method
- Design by Contract is not a substitute for unit testing or verification, but it complements those approaches very well
Resources

- *Advances in Object-Oriented Software Engineering*, chapter “Design by Contract” by Bertrand Meyer [8]
- “Spec#” by Microsoft [9]
All references

- Matthew Flatt, Robert Bruce Findler, and PLT. The Racket Guide.
- Microsoft. Spec#. 