We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,300

Open access books available

172,000

International authors and editors

190M

Downloads

Our authors are among the

154

Countries delivered to

TOP 1%

most cited scientists

12.2%

Contributors from top 500 universities

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Flexible Design by Contract

Koen Vanderkimpen and Eric Steegmans

*K.U.Leuven
Belgium*

1. Introduction

Advances in computer science are made at many different levels and in many areas. One important area of research is reusability, a goal achievable in many ways, from the improvement of programming languages to the development and use of components off the shelf. In this chapter, we focus on methodology to improve reusability.

More precisely, we improve the practice of design by contract, a system of rules for specification of code that aims at improving reliability and clarity of code, two factors that contribute to reuse. The practice, however, conflicts with another practice, that of maximizing the adaptability of code, which is also an important factor to improve reuse. More on this conflict and on the limits of design by contract is explained in section 2.

Our first improvement to design by contract, the flexibility principle, aims to reconcile the methodology with adaptability. The second contribution, the distinction between restrictions and permissions, aims to improve the expressiveness of contracts.

The flexibility principle divides contract clauses between flexible and rigid clauses. The latter are inherited, the former are only valid in the containing class, which allows for more adaptations in subclasses. The second distinction, between restrictive and permissive clauses, allows splitting the domain of possible states for an object in three: a part that's valid, a part that's invalid, and a third left undefined, which can still be adapted in the future. Traditional contracts define only valid and invalid states, leaving little room for future change. We elaborate on our principle in section 3, defining the four important kinds of contract clauses (two times two) and their behaviour under inheritance.

Of course, a programming practice can reap great benefits of language support to make it more useful. That's why, in section 4, we introduce properties, part of our previous research, as the basis to which we will apply flexible design by contract. To do this, we demonstrate the readonly modifier for properties, and we show how this modifier is an application of our principle.

In section 5, we demonstrate the principle in full, applying it to the contract clauses of invariants, preconditions and postconditions. We demonstrate how our work improves the coherence of design by contract by creating symmetry between permissions and restrictions and the way they are strengthened or weakened in subclasses.

In section 6, we provide an evaluation of our work. Section 7 elaborates on a possible implementation. In section 8, we discuss related work, and we end with a conclusion and discussion of future work in section 9.

2. Problem Statement

The goal of design by contract, as explained in (Meyer, 1992), is to improve the reliability of software systems. The methodological guidelines enforce a relationship between the client and the supplier, here called the client and the developer. This reliability, however, comes at a price: the contract of a class can only be changed in certain specific ways that hamper the flexibility with which future changes can be made. Furthermore, a contract does not completely specify exactly what is correct, what is incorrect, and what is left undetermined. This section first elaborates on the benefits of programming by contract and the detrimental effect of unbridled adaptability in the absence of contracts, and then explores both of the principle's problems of flexibility and expressivity.

2.1 Adaptability versus Contracts

To improve adaptability is often considered to be the best way to improving reusability. We can, for example, make systems as customizable as possible, so that they can be reused in many different ways by changing parameters and making adaptations to the code. This kind of adaptability is often found in large software systems that need to be customized for each client individually. It is often accomplished by putting hooks in the code, by which the normal control flow is interrupted and custom code injected. It can also be done by injecting code using aspect oriented software development. Inheritance, with its ability to override the current behaviour, is also a powerful tool to accomplish this.

Contracts pose a serious obstacle when implementing adaptability in such a way, because the behaviour of subroutines that leave hooks and other insertion points for foreign code, is difficult to predict and even more difficult to correctly specify. A contract would limit the options for inserted code, while the ability to insert code often limits what can be described in the contract, or even makes it impossible to create any contract.

Another way to improve reuse is to divide the system in reusable building blocks, called components, which perform certain tasks with high reliability and usually offer some form of abstraction. This abstraction allows the component to be reused in different ways, and in different systems. The reliability of the component, combined with good documentation, makes sure developers will want to reuse the component, because they know what it will do and they know that it will work. A good approach to achieving reliability in such components is the use of contracts. This principle not only improves robustness and correctness, two key components of reliability, by formalizing the component's behaviour, but also automatically provides developers with good documentation, which increases the potential for reuse by lowering the learning curve required to reuse the code. Software written with a clear contract will tell developers what it can and cannot do, so any potential reuse is more easily identified.

To conclude, adaptability and the use of contracts cannot easily be combined. The problem lies in the difficulty of writing a good contract to specify a component that is designed for high adaptability. This reason for this is that class contracts are usually about what is constant: what will always be true about a system; and the more adaptable we make a system, the fewer constant features it will have. We believe, however, in improving the reliability of software systems by the rigorous use of class contracts, and strongly discourage making a system adaptable to the point where class contracts lose any meaning. The detrimental effect of class contracts on the potential adaptability of a system, should,

however, be minimized. In that manner, both reliability and adaptability may lead to increased reusability.

2.2 Flexibility

The first problem of design by contract we consider, is its lack of flexibility. Once we introduce a contract clause in a class, it is automatically inherited by all subclasses, and it can only be changed in certain ways, depending on what kind of clause it is.

```
public class Stack {
    /**
     * push object o on the stack
     *
     * Precondition for this method: none
     * Which means precondition: true
     * The method can always be executed
     */
    public void push (Object o) { ... }
}

public class BoundedStack extends Stack {
    /**
     * New Precondition: getNbItems() < getCapacity()
     * The stack must not be full!
     */
    @Override public void push (Object o) { ... }
}
```

Fig. 1. The Classes of Stacks and BoundedStacks

Consider the example in Figure 1; it is a Java-version of a class of stacks and a subclass of BoundedStacks, based on an example in (Meyer, 1997). The example clearly does something illegal: it introduces a precondition for a method in a subclass, which overrides a method in the superclass without precondition. This is against the Liskov principles of substitution (Liskov, 1994), which state that a precondition may only be weakened in subclasses, not strengthened. The lack of any precondition is identical to a precondition of 'true', which is the weakest possible precondition, so introducing any precondition in the subclass breaks the contract.

Yet exactly this illegal introduction of a precondition is often what we need when creating new subclasses based on classes built by a third party that did not know what our exact requirements would be. It is a good example of the lack of flexibility in specification of contracts. The developer of the class of stacks never considered the possibility of the introduction of a precondition, so he did not provide one, making it impossible for any client to do so either. Meyer (Meyer, 1997) solves this issue by introducing an abstract precondition in the class of Stacks, that can be made concrete by the clients; indirectly, this is a strengthening of the precondition, but by using an undetermined postcondition (read: the absence of a postcondition, which actually defaults to a true postcondition), he stays within the rules and avoids being accused of 'cheating'.

There are two problems with Meyer's approach: the first is the lack of coherence between the different kinds of assertions available: the lack of any assertion, whether it is a pre- or postcondition, means the assertion is true, but this still defines a big difference between the two kinds of assertions, because they are inherited differently. Preconditions in subclasses are OR'ed with those in superclasses, whereas postconditions are AND'ed. This results in preconditions that can only be made weaker (once you have true, the or-operation will make the result always true) and postconditions that can only be made stronger (once you have false, the and-operation will make the result always false).

The second problem lies in the fact that the developer of Stack needs to know he has to make an abstract precondition. If the developers of the two classes are different, however, they usually do not know such things. The superclass provider may guess that a precondition will be necessary in a subclass, but how is he to know which one and how it will be introduced?

So what we need is the option to leave certain things undetermined in a superclass, so subclass developers can still choose for themselves. Having a postcondition of 'true' comes close to non-determination, but for the precondition, there's no easy way to introduce it (a precondition of 'false' would make the execution of the method in the superclass impossible, regardless of the fact we can still weaken it in subclasses). Furthermore, we also need a more symmetrical approach to assertions, which will make it easier to reason about them.

2.3 Expressivity

When discussing flexibility, we already mentioned the problem of inconsistency: preconditions and postconditions being handled differently. The problem is actually much wider than simply the default precondition being false and the default postcondition being true: it is generally difficult to express that which is left undetermined. Lack of any assertion, whether it be pre- or postcondition, should mean that the assertion is left unspecified. We should generally be able to express what is certainly a precondition, what is certainly not a precondition, and what is left in the middle. The same is true for postconditions and invariants.

Consider the example in Figure 2: again, we've made an illegal strengthening of the precondition. Nevertheless, what we want to express here, is that a general Person can have an age from 0 to 200, but a Customer – clearly a subclass of Person, because not everyone is our customer – needs to be older than 18.

For this to work, we would need to be able to introduce a distinction between necessary preconditions and sufficient preconditions. For a Person, it is necessary and sufficient to have an age between 0 and 200. For a customer, however, this is also necessary but not sufficient.

What we would like to express is that values outside of the range between 0 and 200 are always illegal values, but the legality of the values inside that range is left undetermined for now. This is only possible by using tricks such as using a boolean method in the precondition, of which the postcondition can be strengthened. The method would state 'if (age<0 || age>200) then result==false', leaving unspecified when it returns true.

To conclude this section, we list what we need to be able to specify: first, we want to be able to specify that which is not to change in subclasses, whether it is an assertion that restricts certain things or one that permits certain things. Second, we also want to be able to specify what is not to change in future versions of some class (also called: a change in time). Being

able to express what will not change will provide guarantees to clients. Third, we want the option of being able to change what is still allowed to change in flexible ways, giving us full control over all which is not yet specified. Being able to specify anything left unspecified however we wish, will give us, and our clients, the possibility to make more adaptable software.

```
public class Person {  
  
    /**  
     * Precondition: argument should be a reasonable number  
     * arg >= 0 && arg <= 200  
     */  
    public void setAge(int arg) {...}  
}  
  
public class Customer extends Person {  
  
    /**  
     * Precondition: our customers must be adults  
     * arg >= 18  
     */  
    public void setAge(int arg) {...}  
}
```

Fig. 2. The Classes of Person and Customer

3. Flexible Design by Contract

This section explains the two pillars of our solution to improve the flexibility and expressiveness of design by contract. First we explore flexibility, which is accomplished by the distinction between flexible and rigid contract clauses. Second, we look into restrictions versus permissions, a classification of contract clauses which allows a more powerful specification than the traditional restrictive clauses.

3.1 The Flexibility Principle

The first constituent of our principle is the flexibility principle. In programming by contract, the definitions of all possible clauses in a contract state that the clause is valid for a class and all its subclasses. The flexibility principle changes the way in which this validity is determined: a contract clause will no longer automatically be always valid, nor valid for all subclasses. Instead, this will be optional and configurable. The definitions of the principle state as follows:

The Flexible Restriction Principle classifies contract clauses in two kinds: flexible clauses and rigid clauses. Flexible clauses are not inherited through subclassing, and are therefore not subject to any principles of substitutability. Rigid clauses are the traditional kind.

Actually, three different levels of guarantee are defined by this principle. The least level, not included in the definition, guarantees nothing: anything not specified by a flexible or rigid

restriction is left undetermined and thus unguaranteed. The flexible level is therefore an intermediate level: anything guaranteed by a flexible clause, is guaranteed for the current class, whatever versions of the class are still to come. It is thus a guarantee in time. The highest level of guarantee is given by a rigid clause: the specification given by such clauses remains guaranteed not only through time, but also through inheritance. Hence, this can be called a guarantee in all dimensions.

It would not make sense to establish a form of guarantee that would be fixed for inheritance, and thus inherited by subclasses, but not in time, which means it could be changed in future versions. This would cause a fragile base class problem (Szyperski, 1998), as such an assertion could be dropped in a future version of a superclass, possibly invalidating a subclass depending on it.

The two kinds of clauses, and the absence of clauses, can now be defined:

A flexible contract clause is a contract clause asserting a fact about objects of the current class. It may or may not be inherited in subclasses, where it can also, as a convenience, be converted into a rigid clause. It can be changed in time, but only according to well-defined principles of substitutability

A rigid clause is a contract clause asserting a fact about objects of the current class and all of its subclasses. It is automatically inherited and can only be changed in subclasses according to well-defined principles of substitutability

Anything that cannot be asserted by a contract clause, whether because of the lack of any clauses or because the existent clauses do not cover every possible ascertainable fact, is left undetermined, it can be changed at will

A simpler way of understanding these three definitions is as follows: as long as we have no contract at all, we can still add contract clauses. Once we introduce certain contract clauses, we can only add new clauses as long as these do not somehow conflict with the already existing clauses. We could also change these existing clauses, but only in ways that do not conflict with their previous version. In subclasses, we only have to take the rigid classes of the superclass into account, not the flexible ones. For example, suppose we have a method returning an integer. In its initial version, the method has no postcondition and can thus return any number in the integer range determined by the language. We then add a traditional, restrictive postcondition, stating the result has to be greater than or equal to 10. We call this a restrictive postcondition because it restricts any values smaller than 10, but does not specify anything about values greater than 10 (see the next subsection). If this is a flexible clause, the subclasses are still able to add any postcondition, even that the result would have to be smaller than 10. In the class itself, or also in subclasses if our postcondition were rigid, we can only add a postcondition that does not conflict. For example, we could state that the method only returns numbers smaller than 30. The new postcondition would be combined with the existing one using boolean conjunction. We could also introduce a postcondition stating the result has to be greater than 15, which can be combined with the existing clauses, but can also replace the greater-than-10 clause without conflict.

The flexibility principle allows developers to specify contracts for a certain class without compromising the possibility to specify adapted contracts in its subclasses. Any contract

clause is enforced for objects of that specific class, but not automatically for instances of subclasses. The developer can, however, also provide rigid clauses to deal with important restrictions that also apply to subclasses, and supplement these rigid clauses with flexible ones, that function more as guidelines, which can speed up the decision making process of subclass developers by providing easily accessible examples of further assertion statements. Furthermore, if a programming or specification language provides syntactic sugar to easily transform a flexible clause into a rigid one, the flexible clauses can also function as 'pick and choose' kind of clauses, speeding up the actual specification work of subclass developers by providing reusable assertions.

3.2 Restrictive versus Permissive

Sometimes it becomes necessary in a specification to express not only what is certainly incorrect, but also what will always be correct. Traditionally, this is only possible in a contrived way. For example, suppose we want to express, as an invariant, that the value of a certain integer field must always be positive, and is always allowed to be bigger than 18, leaving the interval between 0 and 18 unspecified. This would allow subclass developers to decide about that interval, and only that interval. Using only restrictions, the most we can do in a straightforward manner, is force subclass developers to accept that negative values are unacceptable; we cannot force them to leave the 18+ interval alone, since they are usually only allowed to make restrictions stronger.

To express these kinds of permissive domains, we introduce permissions. They are the opposite of restrictions, which are the traditional contract clauses. When permissions converge with restrictions, they become immutabilities. In the next section, we will see what the permissive qualification alters about the three simplest contract clause types: invariants, preconditions and postconditions. In this section, we provide the general definition:

A permission is a contract clause that expresses a fact that, although it will sometimes be false, is always allowed to be true. No other contract clause may be introduced that contradicts this fact. A rigid permission can be made more permissive, or less restrictive, in subclasses, as long as it does not contradict other contract clauses.

A restriction is a contract clause that expresses a fact that must always be true. It is never allowed to be false. No other contract clause may be introduced that contradicts this fact. A rigid restriction can be made more restrictive, or less permissive, in subclasses, as long as it does not contradict other contract clauses.

An immutability is a permission and a restriction at the same time. It is a fact that must always be true. No other contract clause may be introduced that contradicts this fact. A rigid immutability cannot be changed in subclasses.

The difference between these definitions is apparently small: a permission is always allowed to be true, whereas a restriction is simply always true. Yet they are two opposites that perfectly complement each other, allowing the expression of complex specifications. They relate to each other in the following way: a permission may be made less restrictive in subclasses, but not less restrictive than any restrictions about the same fact that could be present. The reciprocal is true for the restriction: it can be made more restrictive in sub-

classes, but not more so than any present permission about the same fact. When both a restriction and permission are maximally made more or less restrictive, they can no longer be changed in subclasses. This we call an immutable contract clause. The different possibilities for contract clauses are put together in Figure 3.

Different contract clauses in a class will be combined as follows: restrictions of the same kind (invariants, preconditions...) will be logically conjoined to make the complete restriction. Permissions of the same kind will also be logically conjoined, and also conjoined with the complete restriction. This is because we cannot permit anything that is already restricted, so any permission always implies the active restrictions.

Fig. 3. State Diagram of Restrictions and Permissions

When inheriting rigid contract clauses, they will always be logically conjoined with the clauses of the same kind that are added in the subclass: restrictions with new restrictions of

the same kind (invariants, preconditions,...) and permissions with permissions of the same kind. This coherence is different from traditional design by contract, where inherited preconditions were combined using the disjunctive boolean operator instead of conjoined like postconditions.

When using flexible contract clauses, they are similarly conjoined with the rigid clauses of the same kind, whether inherited or local, to form the complete assertion for the current class.

4. Application to Properties

In this section we will digress from flexible design by contract and introduce properties; part of our previous research. In the following section, our principles will be applied to properties to demonstrate them in detail. Properties were first introduced in (Vanderkimpfen et al., 2007a) and further elaborated on in (Vanderkimpfen et al., 2007b). They are a language feature introduced in statically typed object oriented programming languages to replace the concept of fields with a more versatile concept. In section 4.2, we will demonstrate that the readonly modifier for properties is already an early example of flexible design by contract.

4.1 Basic Properties

The main contribution of properties is the greatly increased flexibility when they are used under inheritance. Many things about a property, such as its being a stored or a derived property, can be overridden in subclasses. The traditional fields in languages like JAVA (Gosling et al., 2005) do not leave room for redefinitions at all, which is why good practices such as using beans have been established. With properties, there's no more need for beans; all their benefits are now part of the language.

Properties are best demonstrated using an example. Consider Figure 4, it demonstrates two classes: the class of Persons, and its subclass of BornPersons. The first class has a simple property age, denoting, in a number of years, the age of a Person. The keyword property denotes that the age is a property, which means it is actually the combination of a field, an accessor or getter, and a mutator or setter. Since this property has a primitive type, we call it a primitive property. Unlike the properties in the subclass, the age property in Person does not have a property block, which means it will be a default stored property. Default stored properties have a private value, stored in a field with the same type as the property, a default getter, which simply returns the stored field, and a default setter, which stores its argument in the field. The purpose of properties is to allow access to a field through the methods that encapsulate it, but with the same syntax as if it were a public field. An example of the use of a property is seen in the first and last lines of the setter for the age property in the subclass, where the birthDate property is first accessed, and later assigned to. This access actually uses the getter and setter defined in the birthDate property.

So far, we see no difference with C# properties, which are syntactic sugar for a field with a getter and a setter. The main benefit of our properties lies in its use with inheritance. Suppose we were not allowed to change the class of Persons, but we still wanted to make a better implementation, using a birth date to represent a Person's age instead of an integer denoting the age in years. For this purpose, we introduce a BornPerson class, subclassing Person and introducing a birthDate property. The age of a person must now be derived from that person's birthDate, which means we have to override the age property to make

sure the `birthDate` property is used behind the scenes. The `age` property is now completely derived, which can be inferred by the compiler: the `value` keyword representing the private field to store the property in is absent from the implementation. The only other special keyword is `argument`, which denotes the argument given to the setter. Since our properties are more than just syntactic sugar and since the private values of stored properties are kept completely encapsulated inside the properties at all times, we can drop the private field from the storage space needed to store an object of type `BornPerson`.

A second benefit of properties can be seen in the implementation of the getter and setter for the `birthDate` property: by default, we do not have to declare the private value, since it will have the same type as the property. For the `birthDate` property, however, we were able to declare a private value of a completely different type: the primitive type `long`. This is an acceptable choice, as long as we also make sure that the getter and setter of the property take this difference in types into account (otherwise, this would create an error at compile-time). This mechanism is so versatile, we can even change the type when overriding the property.

```
public class Person {
    public property int age;
}

public class BornPerson extends Person {
    public property Date birthDate {
        long value = 0;
        get {
            if (value==0) return null;
            return new Date(value);
        }
        set {
            if (argument==null) value = 0;
            value = argument.getTime();
        }
    }
    public property int age {
        get {
            return (new Date()).getYear()-birthDate.getYear();
        }
        set {
            Date date = birthDate;
            date.setYear((new Date()).getYear()-argument);
            birthDate = date;
        }
    }
}
```

Fig. 4. The Classes of Persons and BornPersons

4.2 Readonly Properties

As the name implies, a readonly property is a property that can only be read. This means the property cannot be assigned to by clients. As demonstrated in figure 5, it can be introduced

by adding the readonly modifier to the declaration. For default properties, the result will be the absence of a setter, and hence a compile-time error will occur when an attempt to assign to the property is made. For a readonly property with customizations, it would be a compile-time error to introduce a setter inside the property block. Readonly properties should not be confused with constant properties: they can still change, only not through assignment.

In (Vanderkimpen et al., 2007b), we introduced one more modifier: the immutable modifier. The reason for this is that a readonly property can still be overridden to become writable. According to the substitution principles, this is acceptable: only when accessed through its subtype, the redeclared property can be written to. Clients relying on the supertype declaration experience no problems, because a readonly property is not the same as a constant property: it can still change, only not through assignment. An immutable property, contrary to a readonly property, can no longer be made writable in subclasses, making it a stronger constraint.

```
public class Person {  
    public readonly property String firstName;  
    public readonly property String lastName;  
}  
  
public class BornPerson extends Person {  
    public property String firstName;  
    public immutable property String lastName;  
}
```

Fig. 5. Readonly Modifiers

In the example in Figure 5, we introduce names in the class of Persons: a first name and a last name, both readonly. In the class of BornPersons, the first name is made writable: from now on, it can be changed and this option can no longer be taken away in further subclasses. The lastname property, on the other hand, is made immutable, which means the option of making it writable is unavailable for any further subclasses.

Making a property readonly is a restriction: the ability to change it is completely taken away, and thus maximally restricted (so actually it is an immutability). Furthermore, this restriction can be seen as flexible, as it is only applied to the current class (although by default repeated in subclasses, but this is only a matter of choice and a syntactic convenience). Finally, it could also be seen as a 'false' precondition for the setter of the property, because having a precondition of false means it cannot be used.

Making a property writable, on the other hand, is the absence of a restriction or permission. It could actually be seen as a maximal permission, but this would hamper future change, because it would also be rigid. This is because the writable status is automatically inherited. In the absence of any other restrictions or permissions, the ability to write the property is thus left undetermined, with the syntactical effect of being able to assign to it.

Finally, the immutable modifier for properties, like the readonly modifier, is a maximal restriction, but unlike the readonly modifier, it is rigid, as it is automatically inherited in subclasses.

So, what we have demonstrated here, is that the readonly modifier for properties is actually already a special case of the application of flexible design by contract to the precondition of

a property's setter. Readonly is a flexible immutability, immutable is a rigid immutability and writable is the absence of any assertion. In figure 5, we show a state diagram demonstrating the possibilities of the readonly modifier for properties. The full application of our principle to preconditions in general, and also to other kinds of contract clauses, will be elaborated in the next section.

Fig. 6. Readonly State Diagram

5. The Principle in General

In section 3, we explained the new principle's general semantics, introducing the concepts of flexibility versus rigidity and restrictive versus permissive contract clauses. In this section, we apply these formalisms to invariants, preconditions and postconditions, introducing syntax to apply these contract clauses to the property concept from section 4. We present a set of short keywords to indicate these constructs as flexible or rigid and as restrictions or permissions. Furthermore, some single-word modifiers, which map to invariants, can be easily adapted to be used with our principle, similar to the readonly concept in the previous section. These modifiers are not supported in Java by default, so we will also provide a short explanation on them.

5.1 Invariants

In design by contract (Meyer, 1992), an invariant is a clause in the contract denoting a fact that will always be true. For a class contract, this fact will be true for all objects of that class throughout the object life cycle, starting immediately after construction and ending immediately before destruction. Usually, with 'objects of that class', objects of that class and all its subclasses are meant. Using the flexible restriction principle, however, we will make a clear distinction between invariants valid for a single class and invariants valid for the whole hierarchy. Additionally, we will make a distinction between the traditional restrictive invariants and the newly introduced permissive invariants.

For the scope of this work, we limit the invariants in our contracts to invariants that revolve around specific properties. Consequentially, the invariants will be added to the property block. We call these invariants 'property invariants', as opposed to the regular class invariants. They do, however, have the same semantics.

We explain all these different invariants by means of an example. In these examples, Java code will sometimes be complemented with pseudo code (in italics) to increase readability and conciseness.

In Figure 7, we see the class of Files, having several invariant clauses for its property name. A first distinction is made between flexible and rigid invariants: all contract clauses will by default be flexible, unless the keyword **rigid** is added, that's why the invariants without the **rigid** keyword are flexible. A second distinction is the presence or absence of the **permit** keyword, which indicates a clause as being a permission, rather than a restriction, which is, again, the default. Finally, all invariants are indicated as such by the keyword **invar**, which is followed by an optional name and an obligatory boolean statement enclosed in braces, indicating the actual invariant.

The class of Files has four invariant clauses. The first two are rigid restrictions; they tell us, that for all files, including subclasses of File, the name must never contain a slash or a delete character and the name must be between 0 and 256 characters. This is the traditional invariant clause: any object of type File for which the boolean statement is not true, is violating the class invariant. The invariant clause is inherited in all subclasses and can never be made less restrictive, according to the Liskov substitution principles (Liskov 1994).

```
public class File {
    public property String name {
        rigid invar(!name.contains(/,DEL) && name.length()<256);
        rigid invar(name.length()>=1);
        rigid permit invar(name.length()<=12);
        rigid permit invar(name.matches(alphanumerics));
        permit invar wildcard(!name.contains(?,*));
        permit invar (name.length()<=16);
        invar (name.length()<=32);
        invar notnull(name!=null);
    }
}
```

Fig. 7. The Class of Files

The third and fourth clauses indicate rigid permissions. They tell us, that for all files, including subclasses of File, if the name has at most 12 alphanumeric characters, it will be a valid name. This is the opposite of the traditional invariant: any object of type File for which the boolean statements are all true, is obeying the class invariant. The invariant clause is inherited in all subclasses and cannot be made more restrictive, which means it also defines a limit to how strong the class invariant can become.

In the absence of flexible invariant clauses, the combination of the rigid restrictions and permissions determines how the full domain of possible objects is divided into objects that respect the class invariant, objects that violate it, and objects for which this is undetermined. The third group are the objects for which the restriction conditions all evaluate to true, but not all of the permissions evaluate to true. For example, a filename that does not contain slashes or delete signs, but does contain non-alphanumeric characters, is not a restricted filename, but we cannot be certain it will always be a permitted one.

The situation is slightly different when flexible invariants are present. In such a case, the flexible invariant clauses further shrink the domain of undetermined values for an object's

properties, if that object is a direct instance of the current class. Flexible contract clauses are logically conjoined with their rigid counterparts.

Consider the fifth and sixth clause in Figure 7. They are flexible permissive invariant clauses, which tell us that, for instances of File, names shorter than 16 characters and not containing wildcards, will always be valid names, when they also respect the restrictive invariant clauses. Note that the fifth clause is named as the wildcards clause; this will prove convenient when the clause becomes important for a subclass, as it is easily referred to. Furthermore, an error-reporting tool able to detect the violation of invariants can create more meaningful error messages by using the name of the invariant clauses involved.

The last two clauses are flexible restrictions, further restricting the rigid restrictions for instances of File to the point where names of files cannot be null and can at most count 32 characters.

Please note a hypothetical conflict between the wildcards flexible permission and the first rigid restriction: it seems as if names not containing wildcards would always be valid names, even if they contain deletion sign or slashes, because the permission does not repeat those characters in the list of prohibited characters. This is of course not a real conflict: the rules we introduce to combine restrictions and permissions state that permissions always automatically imply any active restrictions, which means that only objects that respect the restrictions as well as the permissions are valid objects (objects respecting only the restrictions but not the permissions have an undefined validity; objects that don't comply with the restrictions are, of course, invalid). This is also the case for inherited rigid restrictions and permissions, which are AND'ed with the restrictions and permissions of the subclass.

An illustration of the invariant clauses determining filename length can be seen in figure 8. There is still a range of values not covered by any of the invariants: it is the range between 16 and 32. This is left undetermined by the developer: no guarantees are given that these values will be legal, and likewise, no guarantee is given that they will be illegal. This allows the developer of the class to further restrict or slacken the invariant in future versions of the class of Files. Clients are guaranteed the current restrictions and permissions; use of the undetermined range of values is at their own risk.

Fig. 8. Filename Length

In the subclass of files for ms-dos operating systems, shown in Figure 9, the invariants for the filename are altered. The first rigid clause further restricts the list of available characters for filenames; obviously, the new illegal symbols cannot be alphanumeric characters, since they are already explicitly permitted in the superclass. The second line in the contract states that the name will only contain majuscules.

The third contract clause in DosFile is an immutable invariant clause: it restricts the length of filenames to 12 characters or less. Since there is already a rigid permission, defined in the

superclass `File`, that allows filenames to have up to 12 characters, we cannot restrict the length to shorter than this, and we must therefore make this clause an immutability. This means it cannot be further restricted or slackened; it is a rigid permission as well as a rigid restriction.

```
public class DosFile extends File {
    public property String name {
        rigid invar (!name.contains(0x00-0x1F, ,",:,<,>,\,|));
        rigid invar (name.equals(name.toUpperCase()));
        immutable invar (name.length() <= 12);
        rigid invar wildcards, notnull;
    }
}
```

Fig. 9. The Class of DosFiles

The fourth invariant clause is a syntactic convenience: the flexible invariant clauses from the superclass are normally not inherited, but we wish to repeat them nevertheless, and even make them rigid. Since they have names, we can easily do this by modifying them with the right keywords, which of course is only syntactic sugar for repeating the code.

When discussing inheritance of class contracts, we cannot fail to mention the Liskov principles of substitution (Liskov, 1994). According to these, traditional class invariants can only be made more restrictive than the invariants in the superclass, as objects of the subclass must comply with the same rules as objects of the superclass, and optionally some extra rules. The situation is different using the flexible restriction principle. First and foremost, flexible invariants are not inherited at all, so we can ignore them for the remainder of this discussion. The system of restrictions and permissions, however, has a notable effect. Ordinarily, anything not explicitly forbidden by a class invariant is allowed; this is no longer the case: only that which is explicitly permitted is by default allowed, and vice versa, only that which is explicitly restricted is by default forbidden. The most important consequence of this is that a part of the domain of legal objects can remain undetermined, which allows us to later enlarge the part that is legal. Using only restrictions, we were only allowed to increase the part of the domain that was illegal.

The new principles of substitution are thus as follows: a rigid restrictive invariant can be made more restrictive, as long as it does not conflict with a rigid permissive invariant, and a rigid permissive invariant can be made less restrictive, as long as it does not conflict with a rigid restrictive invariant. When both are impossible because of conflicts, we have created an immutable invariant clause, which means the domain of legal values for what concerns that specific invariant clause, is completely determined.

5.2 Preconditions

A precondition, in programming by contract, is a fact that has to be made true by the caller of a method before that method may be called. Failure to do so would result in undefined behaviour. Preconditions, when applied to properties, will be facts that have to be true before calling the setter of the property, and thus before assigning a value to the property.

Consider Figure 10: the argument keyword represents the argument that will be passed to the setter, as explained in section 4. The first precondition of the name property in the figure, states that the property must not be assigned values containing slashes or delete characters.

It is a rigid precondition, which means it can be made more restrictive. The rest of the preconditions in the example are strongly akin to the invariants from section 5.1. This is typical: in order to guarantee the class invariant, the developer introduces preconditions that make it illegal for the client to invalidate the object by assigning a value to it that would be against the invariant. It is, however, not mandatory; for example, we omitted a precondition reflecting the permissive invariant that makes alphanumeric file names legal.

This classical similarity between invariants and preconditions pleads for a way to declare in short that the preconditions are meant to enforce the invariants. This would be a beautiful addition to our concepts, but is, for now, out of scope. We will re-examine it in section 9, together with other future work.

```
public class File {
    public property String name {
        rigid pre (!argument.contains(/,DEL));
        rigid pre (1<=argument.length()<256);
        rigid permit pre (argument.length()<=12);
        permit pre wildcardsarg (!argument.contains(?,*));
        permit pre (argument.length()<=16);
        pre (argument.length()<=32);
        pre notnullarg (argument!=null);
    }

    someMethod() {
        File someFile = getSomeFile();
        someFile.name ?= "myFileNamedTooLong"; //returns true
        someFile.name ??= "myFileNamedTooLong"; //returns false
    }
}
```

Fig. 10. The Class of Files, Preconditions

In the method `someMethod` in Figure 10, the single question mark-equation mark operator checks if a given value satisfies the restrictive preconditions of a property setter. The operator takes an object or primitive value of the same type on its left-hand and right-hand side. Its result will always be boolean. It differs from the regular assignment operator because it does not actually assign anything, it only checks the preconditions. When it returns true, we know our argument is not invalid, but we are not sure it is valid, as it could still be in the undetermined domain of values.

The double question mark-equation mark operator, on the other hand, checks the permissive preconditions and returns true only if these are met. The restrictive preconditions are of course automatically met in this case, since they are implied by the permissive ones. This check can thus give a stronger guarantee if it returns true, because we know the argument is valid for certain. In the example, the second check returns false because the permissive precondition '`<=16`' is not met.

Both operators are an easy way for clients that wish to make an assignment to a property depending on its preconditions. It would be an easy exercise to introduce assignment operators that would check the preconditions and then make the assignment if they are met, which is simply syntactic sugar for an if-statement; we leave this to future work.

An example of rigid preconditions being inherited can be seen in Figure 11.

```

public class DosFile {

    public property String name {
        rigid pre (!argument.contains(0x00-0x1F, ",,:,<,>\\,|));
        immutable pre (argument.length() <= 12);
        rigid pre wildcardarg, notnullarg;
    }
}

```

Fig. 11. Precondition in the class of DosFiles

In the class of DosFiles, the first rigid restrictive precondition strengthens the precondition by disallowing more characters. The second line makes the permission to have a name shorter than 12 characters immutable, meaning this is now also a restriction. The third line turns the flexible permissions wildcardarg and notnullarg of the superclass into an immutable precondition: the argument is now required not to contain wildcards and must also be non-null. Note that a restriction for the argument to consist only of capitals, is not given. This means the implementation of the setter will somehow have to make sure that the name is capitalized in order not to break the class invariant for DosFile. We will see to this when discussing postconditions.

From the example, we see that the substitution principles for preconditions are analogous to those for invariants. One important fact must be mentioned: the traditional precondition, when inherited, can only be made weaker, as opposed to the traditional invariants, which can only be made stronger. This is because the client is key in design by contract, and the actions of the client cannot be further restricted. Using the flexible restriction principle, this inconsistency is absent: restrictive preconditions can be made more restrictive, just like restrictive invariants. This correspondence is granted by the fact that our principles allow part of the domain of legal values to remain unspecified. From client-centric, we have evolved to a system where the provider still has some freedom in adapting the specification.

5.3 Postconditions

A postcondition, in programming by contract, is a fact that has to be made true by a method by the end of that method. Postconditions, when applied to properties, pertain only the result of the setter, which usually means they will be little used with simple single-valued properties. The default postcondition for non-primitive properties is shown in Figure 12. For primitive properties, it would make use of identity (==). This default does not need to be provided and is overridden by the user when a postcondition is introduced in the property block. A more useful set of postconditions is shown for the copy method in Figure 12.

Postconditions differ from invariants in that they need not always be true. They only have to be true immediately after the execution of a method. The flexible restriction principle, applied to postconditions, alters the definition of the standard postcondition in the following way: The postcondition must be true after execution of an assignment to a property, if the assignment was dynamically bound to the property's setter as defined in the class declaring the postcondition. Adding the rigid keyword to the postcondition ensures that the postcondition must always be made true after executing the method call, regardless of which class the object containing the property belongs to. Just as was the case with invariants, this does not break the substitution principles, because the definition of the flexible postcondition is different from the definition of a rigid one.

To illustrate the difference between restrictive and permissive postconditions, we consider the File example again, in Figure 12. The rigid restriction of the copy method tells us that we can expect the method to always create a file with a name that is at least equal to the copied file's name, ignoring case and dropping any part beyond the first 12 characters. The flexible restriction strengthens this further by stating the names need to be exactly equal, but only for direct instances of File.

The rigid permission of the copy method tells us that if the new name equals the existing name, the postcondition will always be fulfilled. This means a permission is the maximum postcondition that we'll ever need to be achieved by the method's developer, and the restrictive postcondition is the minimum required. Another way of explaining this, is that a restrictive postcondition is the necessary condition a method needs to fulfil, and the permissive postcondition a sufficient condition. Clients can rely on the necessary condition to be met, but not on the sufficient one. The (rigid) permission can be made less restrictive in subclasses, while the restrictive postcondition, like traditional postconditions, can only be made stronger.

```
public class File {
    public property String name {
        post(this.name.equals(argument)) //default postcondition
    }

    public File copy(Directory target) {
        rigid pre(target!=this.getDirectory())
        rigid post
            (result.name.substring(0,11).equalsIgnoreCase(
                name.substring(0,11))
        rigid permit post(result.name.equals(name))
        post(result.name.equals(name))
    }
}

public class DosFile {
    public property String name {
        post(this.name.equals(argument.toUpperCase()))
    }

    public File copy(Directory target) {
        immutable post(result.name.equals(name))
    }
}
```

Fig. 12. Postconditions in the Class of Files

In the subclass of DosFiles, also in Figure 12, the postcondition of the setter needs to be changed from the default, because the invariant for DosFile (in Figure 9) names state they need to be capitalized, so this is exactly what the setter will do. The rigid postcondition of the copy method becomes immutable: it needs to be rigid, because the name of the result being equal to the current name is a necessity, and it is also equally restrictive as the rigid permission in the superclass, which means it cannot be further restricted or weakened anymore.

5.4 More Modifiers

In specification languages like JML (Leavens, 1999), many single-word assertions are introduced on fields or methods. These can sometimes be brought back to some invariant or pre- or postcondition.

An example is the nullable keyword. In JML, all declarations are by default declared non-null, which means the declaration in question cannot evaluate to null. The nullable keyword is introduced to declare that something is allowed to be null.

Applying our principle, we could state that any property declaration has, by default, a flexible immutable invariant stating that the property is not allowed to be null. If we want the property to be able to become null, we can use the same keyword, nullable, to declare a flexible permission invariant stating that the property is valid when null. This, as usual, replaces the default, so it is not conflicting.

Another option is to enforce non-null even further, by making it a rigid immutable invariant. We introduce the keyword 'nevernull' for this purpose. At the other extreme, we could make the nullable permission rigid, by modifying the property with an 'alwaysnullable' keyword.

As we can see, our principles are applicable to many forms of assertions, even single-word modifiers, like nullable.

6. Evaluation

The flexible restriction principle adds qualifiers to clauses of a class contract denoting their precise meaning. At first sight, this adds complexity to programming by contract. It replaces three standard kinds of contract clauses – the precondition, postcondition and invariant – by the same number of clauses, multiplied by two times two distinctions: either a flexible or a rigid clause, and either a restrictive or a permissive clause. This distinction, however, also simplifies the paradigm. The division in four kinds of clauses can be seen as a modification of the clauses by additional meta-information that makes the clause easier to use. Once you know what kind of clause you're dealing with, you know how it impacts the contract, how far its influence goes, and by how much it limits future change. Furthermore, the three revised standard contract clauses are now completely consistent in the way they can be used and transformed.

Another benefit is the increased flexibility with which contracts can be written, thanks to the flexibility principle. The distinction between no clause, a flexible clause and a rigid clause allows developers to exactly determine which parts of the contract are fixed, and which are still flexible, and thus open to future adaptation. Lack of adaptability is an oft-used reason not to use programming by contract, a paradigm which, even in its traditional form, has a great impact on software quality. The increased flexibility established by our ameliorations makes this reason obsolete.

This increased flexibility nevertheless comes at a cost: we are now forced to think about assertions when we write code. How do we know which facts are true or false for all subclasses? How do we know what is sure not to change in future versions of the current class? How much flexibility do clients need? How many guarantees? Of course, this need to think, just as the lack of adaptability, is likewise a popular reason not to use traditional programming by contract. Our enhanced principles are designed to make it easier to reason about these issues in a controlled way. For each assertion conceivable, developers can now

take two recurring steps to decide how to put them in the contract: first, decide if you want to restrict the options, or to delimit a domain of permitted options. Second, decide if you want this assertion in subclasses or not. In traditional programming by contract, these decisions had to be taken implicitly and they impacted how the actual assertions needed to be written instead of simply being expressed by a number of contract clause modifiers. Making developers think about these questions in a controlled way will have a positive effect on the quality of the code they write.

Besides ease of use and flexibility, there is also an increase in expressiveness, thanks to the permission principle. This makes the writing of contracts easier, since certain assertions can now be expressed within a single clause, whereas they traditionally required the use of contrived mechanisms using auxiliary methods.

The possibility to add meta-information to contract clauses allows developers of the class to decide exactly which parts of the implementation are under a strict contract, fostering better robustness, and which parts are more suitable to change by subclasses. A criticism that can be made, is that the flexible assertions used to indicate the latter parts, do not provide any real guarantees: because of polymorphism, clients may not know if they are using instances of a class, or of its subclasses. For the developers of a class, this is not really a problem, since they have full control. For clients, a small amount of discipline when using the class is enough to ensure the flexible guarantees. For example, clients could create a subclass in which all flexible guarantees are made rigid, neutralizing the need for instances to be of a certain type as long as they use this subclass's static type. At the same time, the clients could introduce changes to any parts of the class contract that are still flexible. These changes can either be a fortification of the restrictions of the contract, improving robustness, or a fortification of its permissions, where more customization is required.

Another way of making sure objects of a certain type are used, is to create them in factories, and have those factories state in their postconditions that the created objects will be of a certain type, which guarantees the flexible assertions of that type will be applicable to the created objects. In Java (Gosling et al., 2005) for example, this can be done with the 'getClass' method and the '.class' expression, both resulting in types that can be compared with the '==' operator (using the instanceof would prove of no use, since it returns true even for subtypes).

7. Implementation

Implementing the proposed principle in Java would require language support for properties and class contracts. Several languages, Eiffel (Meyer, 1997), for example, exist already that integrate contracts in the language, as opposed to creating a separate specification language to be used on top of the programming language, like the Java Modelling Language (Leavens, 1998) for Java. Both of these approaches have the benefit of seamlessness: the fact that documentation and code should be one, which means: in the same document. This seamlessness is the best way to ensure that the code and its specification remain consistent, which is indispensable for reliable reuse.

A benefit that direct language support has over third party languages is easier acceptance: if the assertions are part of the language, nothing else is needed to start using them, so developers will more easily be inclined to do so.

In any case, language support, whether in a separate documentation language or in the core programming language, requires a compiler, and optionally modifications to the virtual machine (for runtime support). We are currently working on adapting the Java compiler to support our properties, and support for design by contract is future work.

Language support for flexible design by contract would allow for a myriad of useful tools. IDEs could do syntax checking on our assertions, and simple analysis tools could check them for inconsistencies, such as restrictions that restrict further than an already present permission. More advanced tools could check if the program code adheres to the provided assertions, and hence check if it is correct. How far this can go depends on the state of the art of tools for static verification. As it is now, a lot can be accomplished with certain tools (Chalin, 2006; Barnett et al., 2006), but complete verification of the correctness of general program code is not entirely possible. The development of such tools for our concepts, while extremely beneficial, is out of scope.

8. Related Work

The two main ingredients of this work are inheritance and class contracts. We are not the first to focus on these concepts to improve reuse of software.

Concentrating on inheritance, the notion of weak subtyping (Lowe et al., 1998) alleviates the principles of substitutability (Liskov & Wing, 1994) by looking at conformance not in general, but for a concrete application. This is called weak conformance. The difference with our approach is that we do not slacken the principles of substitution, but introduce new concepts which allow the easy identification of parts in the code or contract that are not subject to them.

Weak subtyping is not to be confused with the notion of weak behavioural subtyping (Dhara & Leavens, 1995). The latter defines a different weakening of the subtyping rules, which, as its most important contribution, allows mutable types to be subtypes of immutable types. The possible unexpected side effects resulting from this are prevented by prohibiting direct aliasing between objects of different types.

Weak behavioural subtyping is a weak version of Liskov's strong behavioural subtyping, simply called behavioural subtyping in (Liskov & Wing, 1994), which explicitly forbids mutations of immutable supertype state by mutable subtypes. Using strong behavioural subtyping, inheritance between such types can usually be accomplished by providing an abstract supertype to both the mutable and immutable versions of the type in question. Such a supertype leaves mutability undetermined. This is similar to our notions: our version of programming by contract explicitly states that anything left unspecified is also undetermined. The difference with our approach is the scope: certain assumptions about unspecified constraints still appear in Liskov's notions of subtyping that do not occur in ours. For example, we do not automatically presume that the only state-changes that occur are the explicitly specified ones.

The same example can be found in the Java Modelling Language (Leavens et al., 1999), a language to write contracts for Java-code (Gosling et al., 2005), JML introduces an assignable clause, specifying the allowed state-changes done by methods; the default for this clause is '`\everything`', which means there is no constraint on what state-changes are affected by the method. In general, the assignable clauses in JML are called frame axioms (McCarthy & Hayes, 1987). One could argue the default frame axiom be that anything not mentioned is

not allowed to change. Unfortunately, this is often too restrictive for systems that need to be highly adaptable, and therefore, our principles make this optional, stating that anything left undetermined is not under any contract. However, the use of properties (Vanderkimpen et al., 2007a) as a means of implementing our principles, fosters encapsulation (Snyder, 1986) and high cohesion, which lower the chance for unexpected (or undocumented) side effects and thus reduces the frame problem (Borgida et al., 1995). The JML is widely adopted and several tools exist for both static analysis (Chalin et al., 2006) as runtime checking of the contracts. Beyond the new principles introduced by this chapter, using JML differs from our approach because we make the specification part of the programming language, making it more natural in use.

Other languages exist that incorporate class contracts in the code. An important example is Eiffel (Meyer, 1997), which was invented by Bertrand Meyer, who also was the first to introduce the notion of design by contract (Meyer, 1992). Our approach is based on this notion, and expanded with the new concepts of flexibility and permissions.

Inheritance and contracts have also been combined before in a different way. Reuse contracts (Steyaert et al., 1996) allow developers and users of reusable code to specify a protocol for reuse. The contract's main contribution is the identification of method dependencies, which, in the absence of contracts, can cause problems like the fragile base class problem (Mikhajlov & Sekerinski, 1998). Subclassing Contracts (Ruby & Leavens, 2000) are a similar construct, but add this dependency identification to the more general class contracts of JML. Whatever their name, such contracts are indeed an interesting addition to general contracts and can thus also be combined with this work.

The distinction between inherited and uninherited parts of a contract, reflected in our approach by the flexibility principle, is also not entirely new. In Verified Design by Contract (Crocker, 2004) a distinction is made between postconditions and postassertions. Postassertions are actually the traditional, inherited postconditions, stating the facts that need to be true after execution of a method. They must be provable consequences of the postconditions, which are not inherited and thus have to be supplied anew in the subclass in order to still be able to prove the inherited postassertion. These postconditions, however, are more expressive than the traditional ones; they are in fact complete, which means they specify precisely what state is changed and how it is changed, which brings us back to frame axioms. The biggest difference between VDBC and our approach is precisely this completeness: in our approach, we are deliberately incomplete in specifications, to allow maximum flexibility of the design.

9. Conclusion & Future Work

In this chapter, we have shown how design by contract can impede the development of adaptable software because of its lack of flexibility. We have also shown how the expressivity of many contracts is often too limited. Focusing on the importance of both adaptable and reliable software, we have introduced flexible design by contract, a methodology aimed at reconciling the reliability of design by contract with a programming style that fosters adaptability.

Flexible design by contract introduces the distinction between flexible and rigid assertions, the latter of which concern whole type hierarchies, as in traditional design by contract, while the former focus only on a single type, not its subtypes. Furthermore, flexible design by

contract clarifies the third option, not to provide an assertion, as undetermined, maximizing the opportunities for future change.

A second distinction principle introduced by flexible design by contract is the distinction between restrictions and permissions, which allow for the division of a domain of possible values for a certain feature in three parts: valid, invalid and undetermined. This goes against the tradition of making anything left unspecified automatically valid or invalid, depending on which of the two has been specified.

We have demonstrated how both these changes to design by contract not only increase flexibility and expressiveness, but also the coherence of the paradigm, by eliminating certain ambiguities such as allowing preconditions only to weaken under inheritance, but postconditions only to strengthen.

We have shown how flexible design by contract can be applied to properties, part of our previous research, by adding invariants, preconditions and postconditions to property declarations. We have also discussed how an implementation in the form of language support can help developers use our methodology and how tools can be used to better test software written this way.

Finally, we have considered the benefits and drawbacks of our principles, and contrasted them with existing work.

Future work could include the addition of more kinds of contract clauses, such as the exception clause, which will be similar to the precondition clause, and clauses that can express the frame axioms of the code. Another branch of future work involves implementation in the form of adaptations to the Java compiler, and tool support. Finally, we note that there are often important similarities between invariant clauses and precondition clauses. Devising a short-hand notation to make preconditions implement the invariants can thus be future work that would strongly alleviate the burden of writing semi-duplicate specifications.

10. References

- Barnett, M.; Chang, B.; DeLine, R.; Jacobs, B. & Leino, K. (2006). Boogie: A Modular Reusable Verifier for Object-Oriented Programs, In: *Formal Methods for Components and Objects*, de Boer, F.; Bonsangue, M.; Graf, S. & de Roever, W.-P., (Ed.), 364-387, Springer, 978-3-540-36749-9, Heidelberg, Germany
- Borgida, A.; Mylopoulos, J. & Reiter, R. (1995). On the Frame Problem in Procedure Specifications. *IEEE Trans. Softw. Eng.*, 21, 10, (October 1995) 785-798, 0098-5589
- Chalin, P.; Kiniry, J.; Leavens, G. & Poll, E. (2006). Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2, In: *Formal Methods for Components and Objects*, de Boer, F.; Bonsangue, M.; Graf, S. & de Roever, W.-P., (Ed.), 342-363, Springer, 978-3-540-36749-9, Heidelberg, Germany
- Crocker, D. (2004). Safe Object-Oriented Software: The Verified Design-by-contract Paradigm, In: *Practical Elements of Safety: Proceedings of the Twelfth Safety-critical Systems Symposium, Birmingham, UK, 17-19 February 2004*, Redmill, F. & Anderson T. (Eds.), 19-43, Springer-Verlag, 978-1-852-33800-8, New York, United States
- Dhara, K & Leavens, G. (1995). Weak Behavioral Subtyping for Types with Mutable Objects. *Electronic Notes in Theoretical Computer Science*, 1, (1995) 91-113, 1571-0061

- Gosling J.; Joy, B.; Steele, G. & Bracha, G. (2005). *The Java™ Language Specification (Third Edition)*, Addison Wesley, 978-0321246783, United States
- Leavens, G.; Baker, A. & Ruby, C. (1998). JML: a Java modeling language. In: *OOPSLA '98 Addendum: Addendum to the 1998 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, ACM, 1-58113-286-7, Vancouver, Canada
- Leavens, G.; Baker, A. & Ruby, C. (1999). JML: a Notation for Detailed Design. In: *Behavioral Specifications for Businesses and Systems*, Kilov, H.; Rumpe, B. & Simmonds, I. (Ed.), 175-188, Springer, 978-0792386292, Heidelberg, Germany
- Liskov, B. & Wing, J. (1994). A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems*, 16, 6, (November 1994) 1811-1841, 0164-0925
- Lowe, W.; Neumann, R.; Trapp, M. & Zimmerman, W. (1998). Weak subtyping – yet another notion of inheritance, *Technology of Object-Oriented Languages, 1998, Tools 26, Proceedings*, 333-345, 0-8186-8482-8, Santa Barbara, CA, USA, August 1998, IEEE Computer Society, Washington, DC
- McCarthy, J. & Hayes, P. (1987). Some philosophical problems from the standpoint of artificial intelligence. In: *Readings in nonmonotonic reasoning*, Ginsberg, M. (Ed.), 26-45, Morgan Kaufmann Publishers Inc., 0-934613-45-1, San Francisco, United States
- Meyer, B. (1992). Applying “Design by Contract”. *Computer*, 25, 10, (October 1992) 40-51, 0018-9162
- Meyer, B. (1997). *Object Oriented Software Construction (2nd Ed.)*, Prentice-Hall, 0-13-629155-4, United States
- Mikhajlov, L. & Sekerinski, E. (1998). A study of The Fragile Base Class Problem. In: *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, Jul, E. (Ed.), 355-382, Springer-Verlag, 3-540-64737-6, London, United Kingdom
- Ruby, C. & Leavens, G. (2000). Safely creating correct subclasses without seeing superclass code. In: *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Rosson, M. & Lea, D. (Eds.), 208-228, ACM, 1-58113-200-X, New York, United States
- Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages, In: *OOPSLA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, 38-45, ACM, 0-89791-204-7, New York, United States
- Steyaert, P.; Lucas, C.; Mens, K. & D'Hondt, T. (1996). Reuse Contracts: Managing the Evolution of Reusable Assets, In: *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 268-285, ACM, 0-89791-788-X, San Jose, California, United States
- Szyperski, C. (1998). *Component software: beyond object oriented programming (2nd Ed.)*, Addison-Wesley Professional, 0-201-17457-20, United States
- Vanderkimpen, K.; van Dooren, M. & Steegmans, E. (2007a). Flexible Object Representation using Properties, *Proceedings of the International Multiconference on Computer Science and Information Technology*, 1103-1112, 1896-7094, Wisła, Poland, October 2007, PTI
- Vanderkimpen, K.; van Dooren, M. & Steegmans, E. (2007b). Using Customizable Properties to make Object Representation a First-class Citizen. *Computer Science and Information Systems*, 4, 2, (December 2007) 27-46, 1820-0214

Engineering the Computer Science and IT

Edited by Safeeullah Soomro

ISBN 978-953-307-012-4

Hard cover, 506 pages

Publisher InTech

Published online 01, October, 2009

Published in print edition October, 2009

It has been many decades, since Computer Science has been able to achieve tremendous recognition and has been applied in various fields, mainly computer programming and software engineering. Many efforts have been taken to improve knowledge of researchers, educationists and others in the field of computer science and engineering. This book provides a further insight in this direction. It provides innovative ideas in the field of computer science and engineering with a view to face new challenges of the current and future centuries. This book comprises of 25 chapters focusing on the basic and applied research in the field of computer science and information technology. It increases knowledge in the topics such as web programming, logic programming, software debugging, real-time systems, statistical modeling, networking, program analysis, mathematical models and natural language processing.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Koen Vanderkimpen and Eric Steegmans (2009). Flexible Design by Contract, Engineering the Computer Science and IT, Safeeullah Soomro (Ed.), ISBN: 978-953-307-012-4, InTech, Available from: <http://www.intechopen.com/books/engineering-the-computer-science-and-it/flexible-design-by-contract>

INTECH
open science | open minds

InTech Europe

University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China

Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2009 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the [Creative Commons Attribution-NonCommercial-ShareAlike-3.0 License](https://creativecommons.org/licenses/by-nc-sa/3.0/), which permits use, distribution and reproduction for non-commercial purposes, provided the original is properly cited and derivative works building on this content are distributed under the same license.

IntechOpen

IntechOpen