MultiProperties

Koen Vanderkimpen

K. U. Leuven

Department of Computer Science

Leuven, Belgium

Email: koen.vanderkimpen@cs.kuleuven.be

Eric Steegmans

K. U. Leuven

Department of Computer Science

Leuven, Belgium

Email: eric.steegmans@cs.kuleuven.be

Abstract—Properties, which occur in C#, provide good support for features of objects with singular multiplicity and allow straightforward implementation and use of an object's encapsulated state. If, however, used for features with a higher multiplicity, usually represented by collection objects, they fail to provide a natural approach to proper encapsulation of the one-to-many aspect of such characteristics.

Lacking support by powerful concepts, such as properties, makes handling these associations with many-valued multiplicity necessitate much boilerplate code in mainstream programming languages. This makes the implementation of such features tedious and error-prone.

In this paper, we introduce multiproperties: properties specifically developed to make implementing one-to-many relationships in object oriented programming languages easier and more robust. Multiproperties make use of operator overloading to access these multivalued features in a simple and natural way. They are aimed at reducing boilerplate code and making encapsulation of an object's data automatic and configurable.

I. INTRODUCTION

IN OBJECT Oriented Programming, object state is usually handled by fields. To enhance information hiding [1], data encapsulation [2] is often used to protect these fields. This is often done by introducing so-called getters and setters and restricting access to the field itself by declaring it private. Unfortunately, this introduces a lot of boilerplate code in a program. In the java jdk 1.1, for example, about one fourth of all methods are field accessor methods [3].

This problem has been greatly alleviated by C# properties [4] and our own previous enhancements to the property language construct [5].

Properties encapsulating collections, however, are still lacking in functionality. For a default list in C#, only requiring the ability to index and insert items, we could use a property and count on the lists indexer. This prevents a lot of boilerplate code and allows easy access to the items in a List property. But if we require even the most basic extra functionality, such as adding and removing items, or if we require the use of a different collection than a list, we still need to call accessors as declared in the collection interfaces of the language.

This is not only a problem in C#: many imperative programming languages work in the same manner, a notable example being Java [6]. In C#, we could come closer to a solution using operator overloading. If we were to define subclasses of the regular collections that introduce operators such as '+' to add an element and '?' to see if an element is contained in

the collection, we could use these on properties and decrease boiler plate code even further.

The real problem, however, appears when we wish to make further customizations to the way the collections are accessed. A simple check to see if an object is not null, before adding it to a list, is not possible without entirely wrapping the used collection and providing a method to add the object that makes the check. If we don't wrap the collection, it can still be accessed by clients and null references could be added. If we do wrap it, we need to provide wrapper methods for every piece of functionality required, and moreover, we can no longer use a property, enforcing even more boilerplate code. An anonymous subclass of the collection may be the answer, but we would be making subclasses for each and every collection of which we need to customize the behaviour. Additionally, making anonymous classes creates even more boilerplate code and generally complicates our code even further.

An example of this problem is given in Fig. 1. It demonstrates a Library class, implemented in C#. The library contains a list of books, which must never contain the null reference. To make sure no null references ever get into the list, it is encapsulated, and a read-only property returns a copy of the list when it is read. This prevents aliasing, so clients cannot add the null reference from some other location in the program. The encapsulation unfortunately deprives clients of any useful functionality as provided by the List class in the API. To give users an instrument to manipulate the list, some auxiliary wrapper methods and an indexer haven been added, so at least books can be added and removed in some fashion. Search functionality and other advanced methods from the List class are, however, missing.

As we can see, the simple restriction of null elements in our list forces the introduction of a large amount of boilerplate code, while still being severely lacking in functionality. A solution would be the introduction of a class of lists that can accept a custom predicate based on which elements will be added. But addition is only one aspect; there could be restrictions on removing elements as well, or on any other operation. Making list subclasses, and perhaps other collection classes like sets, would lead to a proliferation of classes in the collection framework. More advanced solutions would require patterns [7] to enhance reuse, but ultimately, it will always be a tedious and error-prone task to support every required customization to a collection feature.

```

public class Library {
    private IList<Book> books = List<Book>();

    public IList<Book> Books {
        get {
            return new List<Book>(books);
        }
    }

    public void add(Book book) {
        if (book!=null) books.Add(book);
    }

    public void remove(Book book) {
        books.Remove(book);
    }

    public Book this[int i] {
        get {
            return books[i]
        }
        set {
            if (value!=null) books[i] = value;
        }
    }
}

```

Fig. 1. A class of Libraries in C#

This paper presents a more elegant solution to access control and customization for collections, by extending properties. We introduce multiproperties, which are properties for collections. Multiproperties allow the introduction of many more accessor methods besides the traditional getter and setter, but also provide default implementations for those methods to reduce boilerplate code. The accessor methods reflect most of those found in regular collections, such as lists, sets and bags. Furthermore, multiproperties automatically encapsulate collections while still providing easy access to them through operators. This encapsulation uses defensive copying whenever a reference needs to be passed to the client, so no aliasing can occur.

Multiproperties are an extension to our properties [5], which receive a short introduction in section 2. Originally developed as an extension to Java, our properties are already an improvement to C# properties because they are self-contained and provide automatic encapsulation. Furthermore, our properties are more flexible with regard to customizations and overriding when inherited, which culminates in their ability to be joined. The new concept of multiproperties will syntactically be an extension to C#, but is applicable to most C#- or Java-like languages.

In section 3, we introduce multiproperties as a special case of properties, used for collections, and discuss their basic features for use in isolated classes. These features are then evaluated in section 4. The final sections of the paper consist of related work (section 5), future work (section 6) and a conclusion in section 7.

II. ON PROPERTIES

Our property concept was established in [5] and further developed in [8]. In this paper, before broadening the concept towards multiproperties, we will first introduce a small syntactic change. As in C#, our properties can have getter and setter declarations, which, up till now, also as in C#, use the get and set keywords.

From now on, to declare a getter, we will use operator overloading on the dot (.) operator, and for the setter, operator overloading on the assignment (=) operator. This makes the meaning of these accessor methods more clear, as their goal is to allow referencing the property and assigning to it as if it were a simple field, for which we would use precisely those operators.

This syntactical change is, however, not the only way in which properties differ from their C# cousins. Their main contribution is their customizability when used with inheritance. The getter and setter can always be overridden¹ in subclasses, and the privately stored field, if any, automatically adapts to these changes. It is, for example, possible to redeclare a stored property (one backed by a field) into a derived property, and have the (private and hidden) field disappear from the subclass's stored state. Moreover, when stronger constraints require several properties from a superclass always to be equal in a subclass, they can be joined together in that subclass to enforce this equality. This reduces redundancy and the coinciding likelihood of errors while at the same time reducing memory consumption caused by obsoleted fields.

Another benefit of our properties is that they are self-contained: the private value, and the getters and setters for a property are all (optionally) declared inside a property block, instead of in any random location inside the class. This not only enhances readability, but also causes automatic strong encapsulation, as the private value can only be accessed within the property block.

The syntax of singular properties can be best described using an example. Consider the classes of Rectangle and Square in Fig. 2. In Rectangle, a property height is declared first, using defaults for the private value and for the getter, which simply returns the private value. These defaults have been included for demonstration purposes only; the interesting part is the setter, which is declared to override the default (which simply sets the private value to the given argument). The customized setter makes sure that the height cannot become negative.

The next property, width, is declared to behave exactly like the height property, using the *like* keyword. This means it will have the same customizations, and will thus likewise be non-negative. The third property, surface, is a derived property: it will not have a stored private value. The compiler can infer this because no mention of the value has been made within the property block. Moreover, the surface property is declared

¹Only if the property is declared virtual. For brevity, we assume all properties in this paper to have been declared virtual, even in the absence of the keyword.

```

public class Rectangle {
    public int height {
        //(default, no need to declare)
        int value = 0;
        = { //customized
            if (argument>=0) value = argument;
        }
        . { //(default, no need to declare)
            return value;
        }
    }

    public int width like height;

    readonly int surface {
        . {
            return width*height;
        }
    }
}

public class Square extends Rectangle {
    public side join width,height;
}

```

Fig. 2. Rectangle and Square

readonly, which means it is prohibited to declare a setter for it, and also to assign a value to it when using Rectangle objects in code.

In the subclass Square, the width and height of a Rectangle need to be equal at all times. For this purpose, the two superclass properties have been joined and renamed to side. This will keep the two equal automatically, and also decreases the amount of memory needed to store a square, because only one private value is left. When a Square object would be used polymorphically as a Rectangle, any assignments to its width and height would change both.

When overriding properties, everything internal can be overridden at will. This means we can, for example, change the private value to a completely different type, as long as we make sure the getter returns a covariant type and the setter accepts a contravariant type. More information on these inheritance mechanisms can be found in [8].

III. MULTIPROPERTIES

In this section, we introduce multiproperties, and demonstrate their use. Multiproperties are generic properties for collections. First, we show how operator overloading for collections can make working with them more intuitive. Second, we extend properties to multiproperties, which encapsulate operator-overloaded collections and make the introduction of features with multiple cardinality straightforward.

A. Operator Overloading for Collections

When using C#, we could subclass the standard collection framework and add operator overloading to every class to achieve the ease of use we want, but what we really need is for the interfaces of the framework to specify what the operators will do. This is not possible, because in C#, it is not possible to

define abstract operator overloading methods and, as a result, operator overloading in interfaces is also prohibited.

For this reason, the first language feature we advance, is default operator overloading for collection types. In Fig. 3, we demonstrate all operators available for a traditional List, and to what methods they correspond to.

Note that some operators are doubly overloaded. The '+'-operator, for example, can be applied to object's of the collection's item type, in which case it corresponds with a simple add, but also to an object of another collection with the correct generic type, in which case it corresponds with AddAll.

Since our concept is not language-dependent, we have chosen some liberties in what methods will be available through operators. The AddAll method, for example, which adds all items from a list given as argument, is called AddRange in C# (we have chosen the name of a similar method in Java). Furthermore, the method RemoveAll, the corresponding method to remove items based on a list given as argument, does not exist in C# (there is a method RemoveRange, but this method takes two integers as arguments to determine what should be removed). The motivation for this difference is that we achieve greater consistency. A RemoveAll method exists in C#, which takes a Predicate as argument to determine what should be removed. Adding this functionality as an operator (through further overloading) would be easy, but is out of scope.

The range of operators that can be used for collections given in Fig. 3, is of course specific to lists. For general collections, the indexation operators will not be available. Even for lists, the range is not exhaustive, but it does cover most operations habitually applied to such collections. Additionally, we do not cover maps, which can be regarded as collections of keypairs. We leave them as future work.

B. Collections as Properties

Now that we have established how operators work on collections in general, we can introduce multiproperties as a special kind of property for collections with robust encapsulating behaviour.

Usually, when we declare some collection property in a class, we do not want the user of our class to be able to change this collection outside of our class's accessor methods, as this is an important source of errors. This is why the default implementation of a multiproperty in Figures 4 and 5 completely encapsulates the collection and does not leak any references to it.

To denote a property as a multiproperty, we annotate the property keyword with the multi modifier. This will tell the compiler that the property is a multiproperty, and what kind of collection it represents. The rest of the declaration consists of the generic type indicating what will be stored inside what kind of collection.

To illustrate the example in detail, we consider Fig. 4. The first line inside the property block initializes the private value of the property to a concrete list object. This differs from

```

List<string> cats = new List<string>();
List<string> mycats = new List<string>();

//return size of list (Count)
cats.size;

//add item to list, (Add)
cats+="felix";

//add item to (copied) list, assign
//resulting List to list of cats, (Add)
//and the normal assignment operator)
cats=cats+"sansa"

//add an item to the(copied) list, but don't do
//anything with resulting (new) list (Add)
cats+="meowser";

//false: cats will not contain "meowser" (Contains)
cats?"meowser";

//remove an item, (Remove)
cats-="felix";

//add all items in mycats to cats (AddAll)
cats+=mycats;

//similar, but remove (RemoveAll)
mycats-=mycats;

//true: cats contains every item in mycats,
//which is empty (ContainsAll)
cats?mycats;

//prints "sansa", the first cat in the list
//(the indexer, already possible in C#)
print(cats[0]);

//sets the second item in the list (indexer,
//already possible in C#)
cats[1]="meowser";

//remove second element in list (RemoveAt)
cats[1]-;

//return 0, first index of "sansa" (IndexOf)
cats[?]="sansa"

//copy cats into a as an array (ToArray)
string[] a=cats[];

//copy cats into mycats as a list (ToList)
mycats=cats.list;

//returns a set of unique items in the list
cats.set;

//returns a bag with the items in the list
cats.bag;

```

Fig. 3. Using Operators on a List

the type of the property itself, which is `IList`, the interface for lists. The rest of the code in Fig 4 demonstrates the default implementations for the simple operations available on collections, such as the addition and deletion of elements. On that account, note that `'+='` does not behave completely equivalent to `'+'` followed by `'='`, although the ultimate effect is the same. This is for reasons of efficiency, but only in the context of default multiproperties, for which no aliasing can occur.

So far, except for that minor detail, there's no netto advantage over simply using a normal property with a collection as its type: the same operators could be used if operator overloading for the collections were available. Additionally, since the main goal of our work is to make collections easier to use for programmers, it is still possible to call and declare the methods on the multiproperty by their regular names, such as *add* and *contains*.

The real advantage can be seen in Fig. 5, where the accessor and assignment operators for the multiproperty are defined, along with some advanced methods implementing collection conversions. In this implementation, we see the encapsulation provided by the multiproperty: whenever a reference to the private collection would be leaked outside the property, a copy is generated instead. This prevents users from tampering with the proprietary collection inside our class.

```

public multi IList<Book> books {
    //all defaults:
    value = new List<Book>();
    + {
        return new List<>(value).Add(argument);
    }
    + {
        return new List<>(value).AddAll(argument);
    }
    += { //more efficient Add
        value.Add(argument);
    }
    += {
        value.AddAll(argument);
    }
    - {
        //Remove, RemoveAll, similar to Add
        //and AddAll
    }
    ? {
        //Contains, ContainsAll, similar to
        //Add and AddAll
    }
    [] {
        return value.ElementAt(index);
    }
    []- {
        value.RemoveAt(index);
    }
    []= {
        value.Insert(index, argument);
    }
}

```

Fig. 4. A List of Books Multiproperty

In cases where there is absolutely no constraint on how and when items are added or removed from the collection, using

```

= { //copy the collection, so no reference
    //to it will exist outside the property
    value = new List<>(argument);
}
. { //return a copy of the collection
    return new List<>(value);
}
array { //the type for the array is implicit
    //and equal to the property's collected type
    result = new [size];
    int i = 0;
    foreach (item in value) result[i++] = item;
    return result;
}
list {
    return new List<>(value);
}
bag {
    //comparable to the above
}
set {
    //comparable to the above
}

```

Fig. 5. A List of Books, Continued

a simple property is no problem, but from the moment we want to impose some kind of constraint on how, for example, objects are added to the list, we have a lot of work to do. In the current state of the art, we have two choices. The first is to define a regular property for the collection, which, even when readonly, can still directly be accessed as a default collection, and can be passed around throughout the whole program code. To impose our constraint, we subclass the collection class we wish to use, and override the required methods, implementing the constraint. As a result, we have to create a dummy class, and our property has a very specific type instead of an interface type.

The second choice is to write boilerplate code inside the current class to wrap the collection and all the methods we require to call on it, preventing aliasing. This is quite tedious, and becomes chaotic if we have more than one collection in our class. As shown in Fig. 5, the default implementation for a multiproperty fixes this problem, by automating the encapsulation of many methods. Using multiproperties, adding a single constraint becomes much easier. Suppose, for example, we want to throw an exception when the user tries to add the null reference to our collection. All we have to do, is override the ‘+=’-operator to do the required check and throw the exception if necessary. An example of such a customization is given in Fig. 6, which puts the books property inside a class of Libraries.

A final remark on the default multiproperty is about generics. In Fig. 5, a simple ‘<>’ operation is used instead of fully qualifying the generic type of the collection, when, for example, constructing a new list. This is not a syntax error: because of the encapsulation, the compiler can infer the correct type from the declaration of the property, so we do not need to repeat the type over and over again.

```

public class Library {
    public multi IList<Book> books {
        += { //defining either one of + or +=
            //is enough
            if (argument!=null) {
                value.Add(argument);
            } else {
                throw new Exception();
            }
        }

        //+ {...} is automatically adjusted
        //+ for collections uses + for singular
        //elements, so likewise, it does not need to
        //be overridden
    }
}

```

Fig. 6. The Library Class

IV. EVALUATION

Using multiproperties has several advantages. Above all, we prevent a lot of boilerplate code in the form of wrappers to encapsulate collections or (anonymous) subclasses to customize them. With multiproperties, simple and safe access to collections can be achieved on a pick and choose basis. All that’s needed is to override the necessary methods inside a property block. Moreover, wrapping a collection manually is repetitive and error-prone, and should therefore be replaced with the automatic approach achieved by multiproperties. The default multiproperty already behaves with full encapsulation, so usually, not much extra work is needed.

Another advantage is safety: using default multiproperties, references to collections can no longer be leaked to clients. Access to the collections can only be gained through methods over which the developer has full control. This practice prevents unexpected errors that could occur when the collections would be modified through an alias.

The major disadvantage of multiproperties is the extensive use of defensive copying to prevent aliasing of the encapsulated collection object. This will have a negative impact on runtime performance and memory consumption, and will require some constraint from the programmer’s side when using mutators other than ‘+=’ and ‘-=’.

To conclude, a final benefit is inherited from the use of our regular properties [8]. When multiproperties are inherited, the same possibilities as for regular properties are available, which means any method and thus any operator can be overridden to suit the intended behaviour of the subclass. Even the privately stored value can be changed, or discarded to make the property derived. An example of such a derived multiproperty could be a list of books by a certain author, which can be calculated from a more general list of books. Multiproperties could even be joined, just like regular ones, which would effectively merge the superclass collections in a single line of code.

V. RELATED WORK

Multiproperties are a logical extension of simple properties. In the Unified Modeling Language (UML) [9], they can both

be represented by associations. The only difference between the two, is the multiplicity of the link. Some properties, however, are represented by primitive or API-supported types, which do not require a relationship with another class in the diagram. They are represented as attributes in UML.

These associations, or relationships, have been studied before, and several mechanisms exist to turn them into a first-class concept for object-oriented programming. ReJ [10] for example, formally introduces relationships as an equal construct to classes in a middleweight version of Java [11]. Relationship Oriented Design [12], on the other hand, focuses on a methodology centered around the relationships between objects, instead of focussing on the language. Similarly, the Object-Relation model [13] proposes a closer integration of the Object-Oriented model and the Entity-Relationship (ER) model [14], using relationships as a subclass of collections. Finally, [15] raises the abstraction level of Object-Oriented languages with a code inheritance relation for the reuse of general purpose classes as building blocks in other classes. Among other features like mass renaming and component parameters, this conception allows easy reuse of predefined first-class relationships.

The difference between our approach and the approaches described above, is that we do not turn relationships into first-class citizens directly, but make the underlying Object-Oriented mechanisms part of the broader family of object properties, which provide first-class support for object state. Furthermore, we introduce direct language support to make properties in general as easily usable as possible.

Multiproperties make extensive use of defensive copying to protect the encapsulation of the collections. Another mechanism to prevent aliasing is to have an ownership concept in the language. Ownership types [16] restrict access to the representation of an object to the owner of that representation, usually the object itself. This makes aliasing effectively impossible. Some versions of ownership types allow relaxing this restriction or transferring the ownership [17]; or they extend access to inner classes in the same module [18]. The latter allows, for example, to create iterators that can use the encapsulated collection, so that some form of access to them from the outside is still possible.

Ownership types provide a good solution to the encapsulation problem, and may make the implementation of multiproperties easier. They do not, however, address the issue of boilerplate code, which is required to influence the encapsulated collection through its owner. Encapsulation alone is not enough: we also need controlled access. Multiproperties do not solve the aliasing problem as efficiently, but do provide an answer to the access issue as well.

To conclude related work: as described throughout the paper, support for the necessary mechanisms to automate collection encapsulation in C# [4], are inadequate. Of the mainstream languages, however, C# comes closest, with its support for operator overloading, its simple properties, and its indexers.

VI. FUTURE WORK

As discussed earlier, the use of defensive copying to encapsulate the collection objects is not very efficient. One could ask the question if this encapsulation is entirely necessary: perhaps some form of controlled aliasing will prove more useful. As a first future work, we would investigate if it is possible to allow aliasing of the collection object, but still control any access to it by intercepting calls and redirecting them to the customized accessors as declared in the containing multiproperty. This would mean, for example, that if we obtain an alias to a list of books from a library object, adding a book to this list would call the library's 'books+=...' method instead of the usual 'List+=...' operation. Having this mechanism would eliminate the need for defensive copying and may even make it easier for clients to work with multiproperties, as they could access them directly, without even the need to know about the containing object after the initial aliasing. It would, however, create polymorphism issues, as the distinction between a normal collection and a multiproperty's aliased collection became less clear.

The ultimate goal of our work is to make all of an object's internal state, the object representation, a first-class citizen, employing the easy to use and understand concept of properties. Up till now, we have support for unidirectional simple properties and multiproperties. What still needs to be developed, is support for bidirectional properties.

This means, for example, that a bidirectional relationship between two Person objects who are married, can be represented in the class Person as a property spouse, and that the consistency of this relationship (a person's spouse's spouse is that same person) is automatically guaranteed by the language without much effort from the programmer. Bidirectionality can be developed for singular properties as well as for multiproperties.

Another important aspect of object representation is how it is specified. We believe firmly in Design by Contract [19] and coinciding language support. That is why we will also introduce class contracts at the level of properties, in the form of invariants, preconditions and postconditions. A general methodology to do this has already been developed and will appear in [20], scheduled august 2009.

Finally, implementation of these language features by providing a transformation tool to translate them back to plain C#, would be very valuable and could make testing and benchmarking possible. A full compiler would be even better, because it allows changing the memory layout of an object, to drop redundant fields when they appear.

VII. CONCLUSION

We have shown how the representation of attributes with a multi-valued multiplicity is still inadequately supported in mainstream object-oriented programming languages. The traditional use of collections without any further support requires a lot of boilerplate code to implement from the moment any customization to the collection's default behaviour is required. Moreover, this repetitive endeavor often results in errors,

caused by faulty or insufficient encapsulation of an object's internal state, which does not prevent aliasing.

Additionally, we demonstrated how properties, a very useful language concept, becomes crippled when applied to features represented by some kind of collection. A simple getter and setter is not enough in such a case. More accessors are required to deal with addition, deletion and several more frequently used operations on such features.

To solve these issues, we have introduced multiproperties, a mechanism to extend properties with the necessary additional operations to simplify the use of multivalued features of objects. Multiproperties manage the proper encapsulation for collection type properties by making defensive copies. They also automate many aspects of their use, while still providing maximum flexibility for customization.

Finally, we situated multiproperties in the more comprehensive solution for the internal state of objects we are currently developing. Together with regular properties, bidirectional properties and design by contract, they support object representation as a first-class citizen.

REFERENCES

- [1] D. L. Parnas, P. C. Clements, and D. M. Weiss, "Enhancing reusability with information hiding," in *Software reusability: vol. 1, concepts and models*. New York, NY, USA: ACM Press, 1989, pp. 141–157.
- [2] A. T. Cohen, "Data abstraction, data encapsulation and object-oriented programming," *SIGPLAN Not.*, vol. 19, no. 1, pp. 31–35, 1984.
- [3] E. Rose and K. H. Gsbro Rose, "Java access protection through typing," *Concurrency and Computation: Practice and Experience*, vol. 13, p. 2001, 2000.
- [4] M. Corporation, *Microsoft C# Language Specifications*. Redmond, WA, USA: Microsoft Press, 2001.
- [5] K. Vanderkimpfen, M. van Dooren, and E. Steegmans, "Flexible object representation using properties," in *Proceedings of the International Multiconference on Computer Science and Information Technology*. Poland: PIPS, 2007, pp. 1103–1112.
- [6] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley, 1994.
- [8] K. Vanderkimpfen, M. van Dooren, and E. Steegmans, "Using customizable properties to make object representation a first-class citizen," *Computer Science and Information Systems*, vol. 4, no. 2, pp. 27–46, 2007.
- [9] M. Fowler and K. Scott, *UML distilled: applying the standard object modeling language*. Essex, UK, UK: Addison-Wesley Longman Ltd., 1997.
- [10] G. Bierman and A. Wren, "First-class relationships in an object-oriented language," 2005, pp. 262–286. [Online]. Available: http://dx.doi.org/10.1007/11531142_12
- [11] G. Bierman, M. Parkinson, and A. Pitts, "MJ: An imperative core calculus for Java and Java with effects," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-563, Apr. 2003. [Online]. Available: <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-563.pdf>
- [12] J. Noble and J. Grundy, "Explicit relationships in object oriented development," in *Proceedings of TOOLS 18: Technology of ObjectOriented Languages and Systems Conference*. Prentice Hall, 1995, pp. 211–225.
- [13] J. Rumbaugh, "Relations as semantic constructs in an object-oriented language," in *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*. New York, NY, USA: ACM, 1987, pp. 466–481.
- [14] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, 1976.
- [15] M. van Dooren and E. Steegmans, "A higher abstraction level using first-class inheritance relations," in *ECOOP*, 2007, pp. 425–449.
- [16] D. Clarke, J. Potter, and J. Noble, "Ownership types for flexible alias protection," *ACM SIGPLAN Notices*, vol. 33, no. 10, pp. 48–64, 1998.
- [17] D. Clarke and S. Drossopoulou, "Ownership, encapsulation and the disjointness of type and effect," in *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM New York, NY, USA, 2002, pp. 292–310.
- [18] C. Boyapati, B. Liskov, and L. Shriram, "Ownership types for object encapsulation," *ACM SIGPLAN Notices*, vol. 38, no. 1, pp. 213–223, 2003.
- [19] B. Meyer, "Applying "design by contract";," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.
- [20] A. Lazinic, Ed., *Engineering the Computer Science and IT*. Vienna, Austria: IN-TECH, 2009.