Enhancements of Detecting Gang-of-Four Design Patterns in C# Programs

Anna Derezińska [0000-0001-8792-203X] and Mateusz Byczkowski

Institute of Computer Science, Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
A.Derezinska@ii.pw.edu.pl

Abstract. Gang-of-Four design patterns are valuable architectural artefacts in object-oriented design and implementation. Detection of design patterns in an existing code takes an important role in software evolution and maintenance. A lot of work has been devoted to development of methods and tools that support automatic detection of design patterns. There have been scarcely any attempts to detect design patterns in C# programs. We have focused on the refinement and extension of the approach of A. Nagy and B. Kovari. In this paper we discuss the rules for mining of a subset of GoF design patterns in C# applications. These rules have been used to enhance the program that detects design patterns in C# applications. The mining results of both tools were compared.

Keywords: Software maintenance, Design patterns, Design pattern detection, Gang of Four, C#.

1 Introduction

Gang-of-Four design patterns belong to the mostly used pattern category in software development [1]. These general architectural ideas have successfully been applied while implementing programs in different notations, including the C# language [2].

Detection of design patterns in an existing code is an important issue of software evolution and maintenance [3,4]. Knowledge of patterns used in a program might support comprehension of its design, especially if we are not in touch with an author of the code, we need to maintain a legacy software or one prepared by external collaborators via outsourcing, etc. Information about patterns pointed up could also be applicable in software refactoring or in quality software evaluation [5,6].

Many approaches to design pattern detection have been studies and implemented in various software engineering tools [7-10]. However, there is a gap in mining and evaluation design patterns in C# programs.

The work of Nagy and Kovari [11] has been focused on a language-independent solution. They employed general structure-based criteria. One of implementations of this approach was the only one used to evaluate C# programs. Based on this solution,

This is a pre-print of a contribution published in:
A. Derezinska, M. Byczkowski, Enhancements in detecting Gang-of-Four design patterns in C# programs. L. Borzemski et al. (Eds.): ISAT2018, AISC 852, pp. 277-286, Springer, 2019.
The definite authenticated version is available on-line via https://doi.org.10.1007/978-3-319-99981-4_26

we have proposed similar criteria that were refined in case of several design patterns. Moreover, the list of considered patterns was extended with five additional ones.

In this paper, we have presented the structural-based criteria to design pattern detection in C# programs. The proposed criteria were implemented in a tool used for static analysis of C# programs and applied in some case studies. According to experiment results, it can be seen that even the minor refinements could have a significant effect on the design pattern detection.

In the paper *Design Patterns* will be abbreviated as DP. This paper is organized as follows. The next section summarises approaches to DP detection and other related work to DP in C# applications. In Sec. 3 criteria of DP detection used during static analysis of C# programs are presented. Sec. 4 describes briefly tool support to analysis of C# programs and comparison of results. Finally, Sec. 5 concludes the paper.

2 Background and other Related Work

In general, design pattern detection approaches can be divided into two main types: those focused on the static program analysis, and the second that take into account some behavioral program features. Static program analysis is mainly based on the structural analysis, in which inter-relations between classes are examined. Structural approaches use direct code analysis, usually in some intermediate form of the code as AST (Abstract Syntax Tree), or other graph representations based on the reverse engineering techniques. In behavioral analysis, constraints could be satisfied by a candidate instance to be classified as a “valid” design pattern.

Yu et al. [9] used a code representations of Class-Relationship Directed Graphs in which sub-patterns are discovered and jointed to compare with DP templates and method signatures. Combination of static analysis with subsequent dynamic verification can be found in ePAD^{evo} [12]. Guéhéneuc et al. [13] combined former experiences with constraint programming and numeric signatures in design motifs to deal with complete and incomplete occurrences of DP.

One of the problems is existence of many variants of a pattern. Stencel and Wegrzynowicz used static analysis and SQL to find many implementations of DP in Java code [14]. Multiple variants were treated in [15] by machine learning.

Design pattern identification could be proceeded by a filtering step in order to improve performance and precision rate. A metric-based approach to preliminary selection of pattern candidates and their further evaluation was described in [16].

Many different tools have been developed to support design pattern mining in a source code. They can mainly detect patterns in Java or C++ programs, but also in Eiffel, Smalltalk, UML or XML.

Rasool et al. [7] have evaluated different tools that recognize design patterns in source code. However, neither of six compared tools, nor 30 only reviewed in the paper, were detecting patterns in C# programs. Moreover, it has been noted that many tools give only results about the number of recovered patterns but do not show an exact location in the source code. In a PhD thesis by Zanoni [8], many pattern detection tools have been described. Though, there were no candidates to assist mining of

C# programs. More recent list of design pattern detecting tools could be found in [10]. But also in this survey, no tools dealing with C# have been mentioned.

Design patterns could be developed in the C# programming language as in other object-oriented ones. The basic discussion of pattern implementation and their variants related to C# has been presented by Metsker [2]. However, not much work has been published that considered research or experiments on C# programs.

Gatrell et al. [17] examined a C# system in order to confirm a hypothesis that pattern-based classes were less change prone than other classes. In this work detecting of 13 GoF DP was not automated but based on manual code inspection.

Rasool et al. [18] reported on development of a customizable approach to feature-based pattern recognition. Apart from parsers supporting Java and C++, a module for C# was also implement. However, experiments were dealt only with two first languages and not with any C# programs.

To the best of our knowledge, the only tool practically used for mining DP in C# programs was one of implementations of the approach described in [11]. Nagy and Kovari employed static program analysis of an AST form and structure-based criteria for DP. The approach was not bounded by developer intentions, and could also reveal patterns that were not deliberately designed. Our work follows their method.

3 Criteria of Pattern Detection in C# Programs

3.1 Preliminaries

We have modified the approach of [11] focusing only on the variant for the C# language. Nagy and Kovari intended to have a programming language independent solution, hence, language-independent criteria. In their tool, there were separate modules for processing of a source code, C# and Java code in particular. A data model from one of those modules was transmitted to a common criterion matching algorithm that used a list of criteria to design pattern analysis, and returned DP instances.

Based on the evaluation of many papers presented by Budgen [19], the mostly studied patterns have been *Composite*, *Observer* and *Visitor*. *Observer* and *Composite* appeared also to be the mostly useful patterns according to the conducted survey. Results of *Visitor* were mixed, as it was useful for limited purposes. Therefore, selecting new patterns to extend the tool, we have added *Observer* and *Visitor*, as *Composite* was already one of the patterns implemented in the tool [11]. Besides, two structural patterns were supplemented, namely *Adapter* and *Proxy*, that were not counted to a group of “patterns of little real use”. In result, we propose the refined detection criteria that cover the list of DP supported in the original tool plus the additional five patterns mentioned above.

3.2 Criteria of Design Pattern Detection

The presented DP detection criteria are based on static analysis, i.e. general object-oriented structural program features, but taking into account objectives of C# as a source program. In general, classes that are components in a pattern are recognized by interrelations they meet and additional conditions of their fields and methods. In the following criteria, classes will be denoted by capital letters: A, B, etc. If two classes are named by different letters, it is assumed that they are different, e.g. A≠B.

A rule of each DP is illustrated by a class diagram. The diagrams do not define the rules but improve their readability. A diagram presents only one structural variant of a pattern, if a rule implies existence of two possible relations, e.g. an inheritance or realization of an interface. Usually only the solution with a base class is shown.

The class diagrams (Fig.1. - Fig.10.) do not replicate exactly the known original models that explain DP at the modelling level [1]. They should be interpreted as a UML reverse engineering design at the implementation level. Therefore, generalizations reflecting code inheritance relations are shown, but simple associations or aggregations are not present. In some cases, class fields are presented that are checked in the rule. They are mainly code level realizations of associations of various kind.

Elements in the diagrams are additionally annotated with stereotypes «..» that relate to their roles in a pattern. It should be noted that they are not a part of the code under concern, but could only easy comprehension of particular design pattern components. They could be added to elements recognized as design pattern members in their comments or code annotations, if source code modifications were considered.

1. *Singleton*

a) There exists class A that has only private and protected constructors.
b) Class A includes exactly one non-public and static field which is of the same type as the class A.
c) Class A includes exactly one public static method that returns an object of type A.

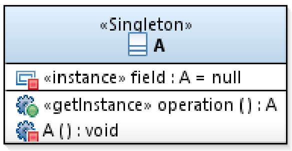

Fig. 1. Singleton

2. *Factory method*

a) There are classes A and B such as B inherits from A or class B implements an interface A.
b) Class B includes a method *operation()* that overrides the method of class A.
c) There exist also classes C and D such as D inherits from C or D implements an interface C.
d) According to its signature the method *operation()* returns an object of type C. In its body in B, the method returns an object of type D.

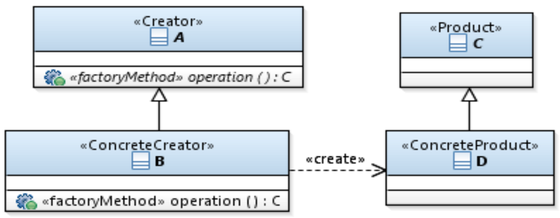
«Creator»
A
«factoryMethod» operation () : C
«Product»
C
«ConcreteCreator»
B
«factoryMethod» operation () : C
«create»
«ConcreteProduct»
D

Fig. 2. Factory method

3. *Proxy* (Fig. 3)

a) There are classes A, B and C such as B and C inherit from A or classes B, C implement an interface A.
b) Class C includes a field that stores a reference to an object of type B.
c) Class A includes a method *operation()*. The method is overridden in B and C.
d) The method *operation()* from class C calls the method *operation()* from class B.

4. *Decorator* (Fig. 4)

a) There exist classes A, B and C such as B and C inherit from A, or classes B, C implement an interface A.
b) Class C includes a field that stores a reference to an object of type A.
c) Class C includes a constructor C that has a parameter – an object of type A.
d) Class A includes a public, protected or default method *operation()* that is overridden in class B and in class C.
e) The method *operation()* from class C calls the method *operation()* from class A.

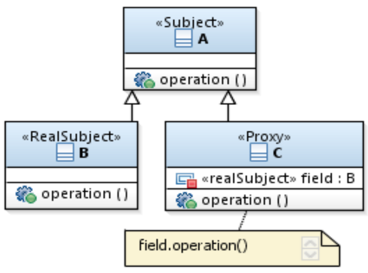

Fig. 3. Proxy

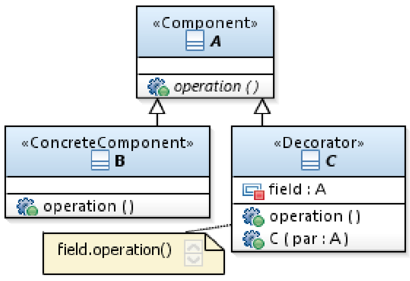

Fig. 4. Decorator

5. *Composite* (Fig. 5)

a) There exist classes A, B and C such as B and C inherits from A.
b) Class C cannot inherit from B.
c) Class C includes a collection of elements of type A.
d) Class A includes a method *operation()* that is overridden in B and C.
e) Inside the method *operation()* in class C, method *operation()* of the class A is called for elements of type A.

6. *Adapter* (Fig. 6)

a) There exists classes A, B and C such as C inherits from A and implements an interface B.
b) Class A does not implement interface B.
c) Class A has its public method *specificOperation()*. Interface B has a public method *operation()* and class C has a public method *operation()* such as it overrides the method from interface B. The method *specificOperation()* from class A is called inside the method *operation()* from class C.

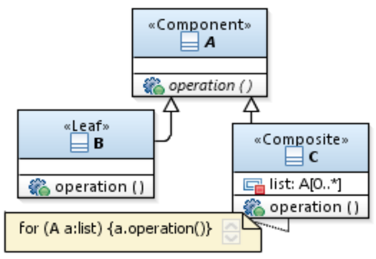

Fig. 5. Composite

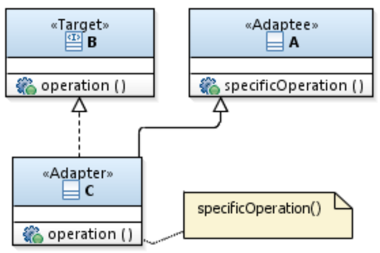

Fig. 6. Adapter

7. *Mediator (a variant with two communicating classes).*

a) There exists classes A, B and C such as B and C inherit from A.
b) There exists classes D and E such as E inherits from E.
c) Each of classes A, B, and C has at least one field that stores a reference to an object of type D.
d) Class E includes one field storing a reference to an object of type B, and one field storing a reference to an object of type C.

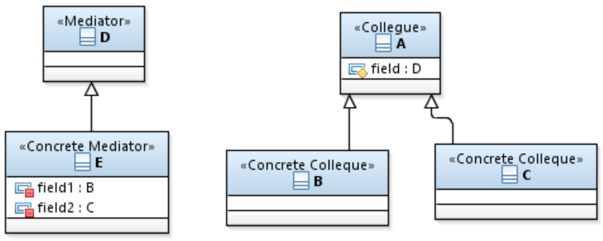

Fig. 7. Mediator

8. *Chain of responsibility*

a) There exist at least two classes B and C that inherit from class A.

b) Classes A, B, and C includes at least one field that has a reference to class A.

c) Class A has a public, non-static method *operation()* that is overridden in class B and in class C.

d) Methods from classes B and C call the method *operation()* for an object o type A.

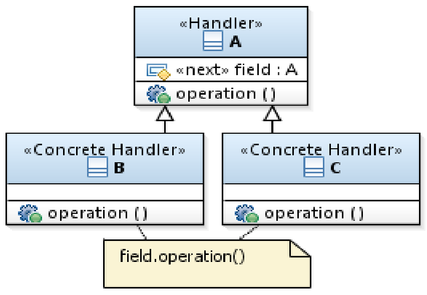

Fig. 8.. Chain of responsibility

9. *Observer* In the case of this design pattern, there are omitted conditions that assume existence of a base class for the observer.

a) There are classes A and B.
b) Class A includes a collection of elements of type B.
c) Class A has a method that calls another method included in class B for any component of the collection.
d) The method in class B has a parameter that is an object of type A or one of base class of class A.

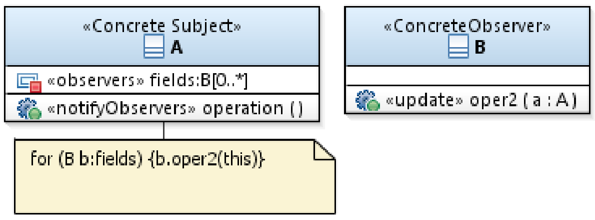
«Concrete Subject»
A
«observers» fields:B[0..*]
«notifyObservers» operation ()
for (B b:fields) {b.oper2(this)}
«ConcreteObserver»
B
«update» oper2 (a : A)

Fig. 9. Observer

10. Visitor

a) There exist classes A, B, C and D such as B inherits from A or implements an interface A, and D inherits from C or implements an interface C.
b) Class A includes a method *operation()* that has a parameter – an object of type C.
c) Class B includes a method that overrides the method operation() from class A.
d) Class C includes a method *oper2()* that has a parameter – an object of type B.
e) Class D includes a method that overrides the method *oper2()* from class C.
f) The method from class B has a parameter of an object of type C, and calls the method from class C.

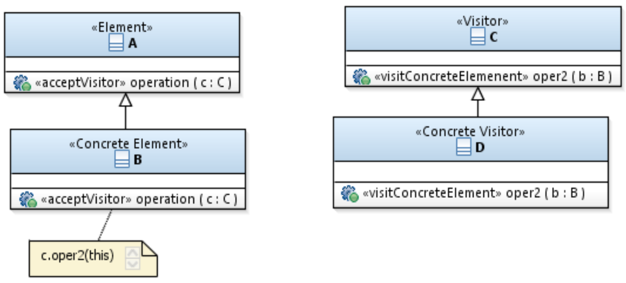
«Element»
A
«acceptVisitor» operation (c : C)
«Concrete Element»
B
«acceptVisitor» operation (c : C)
c.oper2(this)
«Visitor»
C
«visitConcreteElemenent» oper2 (b : B)
«Concrete Visitor»
D
«visitConcreteElement» oper2 (b : B)

Fig. 10. Visitor

4 Design Pattern Detecting with C# Analyzer

C# Analyzer has been developed to integrate different functionalities basing on static analysis of C# programs. The mail goals of the tool are the following:

- Identification of classes and methods that implement selected design patterns.
- Calculation of software metrics.
- Support of a project comprehension.
- Support of software quality analysis based on the above results.

The tool was designed as an extension of the Visual Studio environment. A significant part of C# Analyzer is a module responsible for detection of DP. It inputs code of a program under analysis, and creates a data model, i.e. its syntax tree. The tree is generated using .NET platform compiler - Roslyn. Then, the source program is analyzed according to the given criteria. The tool points at classes that participate in DP under concern. The detection program is a modified version of the tool implemented for C# programs by [11]. The main revisions have referred to the following areas:

1. mechanism of creating model data,
2. criteria to qualify a code extract as a design pattern,
3. addition of new patterns to be detected,
4. identification of all class components in a pattern.

A modification within the first area has been taking into account relations to interfaces in the data model, which can be used in C# programs. Another adaptation implies usability of fields and methods that are inherited from a base class. Based on a recurrent approach, all base classes of a given class are visited starting from the top level class in its inheritance tree. During return routine, the corresponding methods are substituted by overriding methods, and the fields accordingly. In result, all fields and methods accessible due to public or protected modifiers are recognized in the data model of a descendant class.

The program has been extended with detection facility for additional DP (*Adapter*, *Composite*, *Observer*, *Proxy* and *Visitor*), and refinement of other DP detection criteria. The criteria for detection of the design pattern set supported by the current tool are given in Sec. 3.2.

The former detector has identified and outputted only a main class that was accepted as a central component of a pattern. Currently, all classes that are pattern components are recognized. In an output file, they are annotated with the name of a pattern being detected, and a role served in the pattern, i.e. symbols A, B, C, D,… according to the diagrams illustrating the criteria (Sec. 3.2).

Experiments conducted on a set of programs have confirmed usefulness of the solution [20]. Not all DP supported by the detector occurred in the tested programs. Therefore, a set of C# benchmark programs was prepared illustrating application of different DP. All kinds of the patterns were identified in these experiments.

In comparison to the former approach [11], we could detect new patterns and pattern instances corresponding to more structures of C#. On the other hand, some potential patterns were excluded due to criteria refined with regard to the C# language.

In Tab. 1. we have shown the numbers of different patterns detected in the Log4net C# library. It was used in experiments both with the previous tool [11] and C# Analyzer. Therefore, we could compare DP detection abilities of the tools. In the experiments, 22 pattern instances were detected by the old tool whereas 28 by the new one.

Only in case of *Singleton*, both tools recognized exactly the same pattern instances. Diversities in other cases followed from the differences in the pattern recognition criteria. In evaluation of *Decorator*, the former tool only used limited criteria 4.a and 4.c. While taking into account additional criteria (4.b, 4.d, 4.e) two cases were not classified as *Decorator*. The similar situation refers to *Composite*. All four pattern instances were not accepted due to an extra condition (5.d) about a collection of elements of type A implemented in class B. An opposite situation can be experienced in case of *Factory Method*. This pattern specification takes into account two additional relations, namely: detection of interface realization and inheritance of fields and methods. Therefore, 3 additional occurrences of *Factory Method* were recognized apart from 10 instances previously accepted. Finally, *Proxy* was not addressed before but was detected by the current tool.

Table 1. Numbers of design patterns detected by tools in the Log4net C# library

Tool	Design pattern occurrences					
	Singleton	Decorator	Composite	Factory method	Proxy	Sum
Tool [11]	5	3	4	10	-	22
C# Analyzer	5	1	0	13	9	28

5 Conclusions

In this paper we have presented a design pattern detection approach based on static code analysis towards structural feature mining. The approach was adopted to the C# programing language and implemented in the enhanced tool. The tool was combined within the C# Analyzer system that was integrated with the Visual Studio. Other components of the system refer to software metrics. Evaluation of results of pattern detection in the relation to the software metrics is described in [20].

Introduced improvements have resulted in higher number of recognized pattern instances mainly due to additional types of pattern detected. However, more refined criteria have eliminated some of pattern instances accepted formerly, as they counted now being false positive. It should be noted, that the former tool applied to C# programs was an instance of the language-neutral concept of pattern detection. All in all, it could be questionable whether it is not better to have language-specific solutions based on a general one, as proposed in this paper.

The presented tool could further be extended with detection of remaining GoF design patterns and other patterns. Moreover, other criteria to design pattern recognition could be added, taking into account e.g. naming conventions, comments, or metadata. Open issue remains also investigation of different pattern variants in respect to a general structure and specific features of the C# language, performance evaluation of detection facilities, as well as thoroughly estimation of detection precision.

References

1. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: Elements of reusable object-oriented software. Addison-Wesley, Boston
2. Metsker SJ (2004) Design patterns in C#. Addison-Wesley, Boston

3. Ampatzoglou A, Charalampidou S, Stamelos I (2013) Research state of the art on gof design patterns: A mapping study. Journal of Systems and Software 86(7), 1945–1964. doi: 10.1016/j.jss.2013.03.063
4. Mayvan BB, Rasoolzadegan A, Yazdi ZG (2017) The state of the art on design patterns: A systematic mapping of the literature. J. of Systems and Software 125, 93-118. doi: 10.1016/j.jss.2016.11.030
5. Ali M, Elish M O (2013) A comparative literature survey of design patterns impact on software quality. In: Proc. of International Conference on Information Science and Applications (ICISA). IEEE, pp 1-7. doi:10.1109/ICISA.2013.6579460
6. Pradhan P, Dwivedi A K, Rath SK (2015) Impact of design patterns on quantitative assessment of quality parameters. In: Second International Conference on Advances in Computing and Communication Engineering (ICACCE). IEEE, pp 577-582. doi: 10.1109/ICACCE.2015.102
7. Rasoola G, Maedera P, Philippow I (2011) Evaluation of design pattern recovery tools. Procedia Computer Science 3, pp 813–819. doi:10.1016/j.procs.2010.12.134
8. Zanoni M (2012) Data mining techniques for design pattern detection. PhD thesis, Universita degli studi di Milano Bicocca.
9. Yu D, Zhan Y, Chen Z (2015) A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. J. Syst. Softw 103, 1–16. doi: 10.1016/j.jss.2015.01.019
10. Al-Obeidallah M G, Petridis M, Kapetanakis S (2016) A survey on design pattern detection approaches. International Journal of Software Engineering (IJSE), 7(3) 41-59.
11. Nagy A, Kovari B (2015) Programming language neutral design pattern detection. In: Proc. of 16th IEEE International Symposium on Computational Intelligence and Informatics (CINTI). IEEE, pp. 215-219, doi:10.1109/CINTI.2015.7382925
12. De Lucia A, Deufemia V, Gravino C, Risi M, Pirolli C (2015) ePadEvo: A tool for the detection of behavioral design patterns. In: Proc. of ICSME, IEEE, pp 327-329. doi: 10.1109/ICSM.2015.7332480
13. Guéhéneuc Y-G, Guyomarc'h J-Y, Sahraoui H (2010) Improving design-pattern identification: a new approach and an exploratory study. Software Quality Journal 18(1) 145–174. doi:10.1007/s11219-009-9082-y
14. Stencel K, Węgrzynowicz P (2008) Implementation variants of the Singleton design pattern. In: Meerrsman R, Tari Z, Herrero P (eds) On the move to meaningful Internet systems: OTM 2008 Workshops, LNCS vol. 5332, Springer, Berlin Heidelberg, pp 396-406. doi: 10.1007/978-3-540-88875-8_61
15. Zanoni F, Fontana A, Stella F (2015) On applying machine learning techniques for design pattern detection. J. Syst. Softw. 103, 102–117. doi: 0.1016/j.jss.2015.01.037
16. Issaoui I, Bouassida N, Ben-Abdallah H (2015) Using metric-based filtering to improve design pattern detection approaches. Innovations in Systems and Software Engineering, 11, pp 39-53. doi: 10.1007/s11334-014-0241-3
17. Gatrell M, Counsell S, Hall T (2009) Design patterns and change proneness: a replication using proprietary C# software. Working Conference on Reverse Engineering (WCRE). pp 160-164.
18. Rasoola G, Mader P (2014) A customizable approach to design patterns recognition based on feature types. Arab J Sci Eng 39, 8851-8873. doi:10.1007/s13369-014-1449-0
19. Budgen D (2013) Design Patterns: Magic or Myth? IEEE Software 30(2) 87–90. doi: 10.1109/MS.2013.26
20. Derezinska A, Byczkowski M.: Evaluation of design pattern utilization and software metrics in C# programs. unpublished