Object Type Hierarchies Analysis
to Identify Crosscutting Concerns

Mario L. Bernardi and Giuseppe A. Di Lucca
Department of Engineering - RCOST, University of Sannio, Italy
\{mibernd, dilucca\}@unisannio.it

Abstract
In Object Oriented (OO) systems inheritance, implementations and type nesting relationships are used as one of the principal way to implement static crosscutting. Thus Type Hierarchies are one of the cause introducing the scattering and tangling of code in the components along the hierarchies. This paper presents an approach based on the analysis of type hierarchies to automatically identify the scattered and tangled Type Fragments (i.e. a portion of a type in terms of its members and relationships) implementing crosscutting concerns in OO systems. The approach defines a model to represent Concerns as sets of Type Fragments and a stepped process to identify the Type Fragments composing each Concern and to perform a crosscutting analysis among them. This information is useful and needed to re-engineer the crosscutting concerns into aspects, or to migrate an existing object oriented system towards the aspect oriented paradigm. The approach has been applied to several software systems producing valid results, presented and discussed in the paper, about its feasibility and effectiveness.

Keywords: Reverse Engineering, Aspect Mixing, Code Analysis, Aspect Oriented Programming, Software Evolution, MOF

1. Introduction
Separation of concerns and their implementation in well defined modules is one of the challenges of software engineering. A software system can be considered as composed by a set of concerns cooperating each-other to meet the system specification. Thus a system can be modelled by its concerns: a principal (or dominant) decomposition is used to represent the concerns related to the main functionalities and behaviours the system has to provide its users, while a secondary decomposition models all the (secondary) concerns related to functionalities and behaviours shared or included by the principal ones or supporting them. Secondary concerns are usually crosscutting ones: their behaviour is related to, and can be interleaved with, the behaviour of some other concerns and their implementation can crosscut the boundaries of, and overlap, the modularization units defined by the principal decomposition. The implementation of crosscutting concerns, by 'traditional' programming languages, such as the Object Oriented (OO) ones, induces the generation of code scattered and tangled across more components (e.g. classes, methods in OO systems). This can produce a high level of code duplication [9] that makes the crosscutting concerns hard to maintain, evolve and reuse.

In OO systems, the usage of Type Hierarchies (and mainly the ones rooted in Abstract Classes or Interfaces) is a main cause introducing scattering and tangling in the modules implementing a Type along the hierarchy. As an example, a method declared by an Interface (i.e., a Type root of a hierarchy) and implemented by more than one class will have its behaviour scattered in all the classes implementing it and tangled with the behaviour of the other methods implemented in each class along the hierarchy. A main reason for that is that usually inheritance, interfaces' implementations and type nesting, are used as a way to implement crosscutting behaviours (i.e.,
concerns) by super-imposition [12]: each type in the system that need to contribute to a certain concern is forced to implement an interface, to inherit or to contain another type. This is because in OO systems just hierarchical decomposition is allowed as modularization mechanism.

To confirm and verify this assumption a set of existing Java systems have been analyzed. This paper presents the results of this analysis together with the method used to identify the crosscutting among the concerns introduced by Type Hierarchies. The method allows to identify the Types Fragments (i.e., a portion of a Type in terms of its members and relationships) involved in the implementation of a concern and the crosscutting among them. In particular, each type introducing a new external behaviour (e.g. a new method) is initially associated to a separate concern; we call such a type the “Seed” of the concern. The results of the analysis confirmed the goodness of the assumption made: interfaces and type containment, i.e., superimposition, actually are a main way used to implement crosscutting concerns in an OO system.

The knowledge of which portions of a system are involved in the implementation of crosscutting concerns is very useful because it can drive the re-engineering or refactoring of an existing OO systems towards the Aspect Oriented Programming (AOP) paradigm. The evolution/migration of existing OO systems towards AOP ones is a way to eliminate, or reduce at a minimum, the scattered and tangled code implementing the crosscutting concerns (for example OO systems coded by Java can be migrated towards system implemented by AspectJ [3]). The way how re-engineering/refactoring can be performed is not the focus of this paper and it is not dealt with. Our focus is just to verify that Types in Hierarchies are seeds for crosscutting concerns and thus they can be considered as a starting point to mine aspects in existing systems. However, it is just worthwhile to note that the information about the concern composition by the Type Fragments is also useful to reduce the refactoring effort since the focus can be limited to the identified Type Fragments in order to define the related aspects into which encapsulate them.

The paper is structured as follows. Section 2 describes the meta-model exploited to represent concerns at class level. Section 3 presents the approach to identify the crosscutting concerns due to Seeds in a Java system. Section 4 describe the main features of the ConAn tool developed to automatically support the defined approach. Section 5 illustrates the results of the analysis carried out on several Java systems. In Section 6 some relevant related works are discussed. Section 7 contains conclusive remarks and briefly discusses future work.

2. A Model to Represent the Relationships among Types and Concerns in OO Systems

In OOP language (such as Java), the usage of super-imposition and inheritance/implementation of declarations of types can cause the raising of crosscutting. This crosscutting is related to the system static structure expressed as a type hierarchy based on inheritance, implementation and type nesting relationships, thus we refer to it as static crosscutting. Static crosscutting is present and identifiable at type level by looking for the presence of a substitutable method \(^1\) declared in a super-type (e.g. an abstract class or an interface) and inherited/implemented by one or more sub-types: this produces the scattering of that method among the sub-classes defining it and the tangling with the other methods defined by each sub-class.

The Figure 1 shows, as a UML class diagram, the model defined in [1] exploited to represent the static structure of an OO system in terms of its Concerns, Types, and the relationships among such Concerns with the portion of Types implementing them. The grain of model is the class, i.e. a method is seen as a whole black-box (just its declaration is considered, while its internal code is not). The portions by which a Type can be decomposed into, are called “Type Fragments”, i.e. a Fragment is a portion of a type in terms of its members and relationships (inheritance, implementation, and containment). A system is modeled as a set of Types (i.e. Value, Reference, and

\(^1\)A method is said substitutable when it is a concrete implementation (or overriding) of the one declared in a super-Type along the inheritance hierarchy.
Array Types \(^2\) where Reference Types are Interfaces and Classes and an Interface is implemented by one or more Classes. Reference Types are composed by Reference Type Fragments (that can be Class- or Interface- Fragments). A Reference Type Fragment is in turn composed by Fields and Methods. A Concern is represented by mean of a set of Reference Type Fragments (in the paper, for brevity, we refer to them also as Type Fragments or just Fragments). A ReferenceType can inherit from another ReferenceType as well as can contain another ReferenceType (e.g. an inner class). In model instances, relationships on types are mapped to the Type Fragments associated to one of the concerns. The meta-model has been defined as a MOF model and implemented by means of the EMF (Eclipse Modeling Framework) framework. In model instances, each Type Fragment is identified by the fully qualified name of the code component implementing it (e.g., the fully qualified name of a class method).

The model allows to represent different concerns separately depicting the complete system as the composition of all the concerns. The crosscutting relationships among the concerns can be identified by analysing their internal structure in terms of the Type Fragments associated to each concern: concerns whose fragments generate tangling and scattering are considered as crosscutting. A merge operation has been defined on the model to join concerns into a more abstract one. The merge of two concerns \(C_a\) and \(C_b\) creates a new concern \(C_c\) in place of \(C_a\) and \(C_b\) by merging their type fragments.

\(^2\)Array types are treated as separated types since they must specify the type of array’s components.
3. The process to Identify Concerns along Type Hierarchies

The analysis of the system’s Type Hierarchies exploits the method defined in [1].

The process to identify the Concerns along the system’s Type Hierarchy exploits the method defined in [1]. The process is made up by the following main steps:

1. Analysis of Type Hierarchy to find Concerns’ Seeds

2. Type Fragments Identification and Model Instance Generation

3. Clustering of Concerns

4. Crosscutting Analysis

that are shortly illustrated in the next sub-sections (more details can be found in [1])

3.1  Analysis of Type Hierarchy to find Concerns’ Seeds

A static code analysis of the system Type Hierarchy is performed, looking at inheritance, implementation and containment relationships. In particular any Reference Type that introduces, in all the paths from a root to a leaf of the type hierarchy graph, the declaration of a new set of members is to be identified.
The Figure 2-(a) shows an example consisting of three Type Hierarchies rooted in the Interfaces “I1”, “I2” and “I3”. Following the paths from the roots down to leaves, all the Types are to be considered. The three root interfaces introduce basic behaviour (i.e. methods) for their abstractions and hence they are considered Seeds of as many as Concerns. The Classes C1, C2, C3, C4, C5, and C6 are not seeds since they don’t introduce any new member.

3.2 Type Fragments Identification and Model Instance Generation

To create an instance of the model all the Type Fragments implementing each Seed have to be identified. Initially each Seed is associated to one concern. These initial Concerns do not contain any Type Fragment. The association between Concerns and Seeds is made regardless of any semantic meaning about them. Just the structural information, extracted by the traversal of the type hierarchy, is exploited. At the end of the traversal, each concern will be associated to the Fragments of all Typos that implement the corresponding single Seed.

The traversal of the hierarchy type graph starts from the leaves and goes up to the roots. Each type t encountered during the traversal is analysed to detect what Seeds it implements. For each seed s, implemented by t, a type fragment tf is created and added to the concern c associated to the seed s.

The sub-figures 2-(b), 2-(c), and 2-(d) show the results of the traversal on the example of Figure 2-(a) for the three Concerns associated to the Seeds “I1”, “I2” and “I3”. In the figure the UML stereotype “IF” and “CF” are, respectively for “Interface Fragment” and “Class Fragment”. For each Seed, starting from the leaves, the Fragments are created for each method that is substitutable with the considered seed. For example, from the classes “C1” and “C2” result the two fragments “C1” and “C2” including just the implementation of the methods declared by the seeds “I1” and “I3”.

3.3 Clustering of Concerns

The resulting model instance contains a Concern for each identified Seed. This model may be too detailed because the identified Concerns are strictly connected to their implementation and the richness of details could “hide” the more abstract actual Concerns. When this is the case, groups of Concerns (i.e., the associated Seeds) can be grouped together to merge the associated low-level abstraction Concerns into more abstract ones. Just as an example, we could have that some Seeds
contribute to the implementation of a concern more general than the single one associated to each Seed (e.g. different Seeds, each one associated to a Concern implementing the Persistence of a specific object, can be all together associated to the more general Persistence Concern). The same is when a Seed is the specialization of only another one, or when there are several Seeds that are “synonyms”, i.e. different Seeds implemented by Type Fragments performing the same responsibilities. As an example, looking at the Figure 2, we could hypothesize that Concerns associated to Seeds $I_1$ and $I_2$ participate to the implementation of a more abstract concern $I$ and hence could be grouped together. In this case the concern $I$ is derived merging the concerns associated to $I_1$ and $I_2$ as represented in 2-(b), and 2-(c). The results of the merging operation is depicted in the figure 3.

The automatic Hierarchical Agglomerative Clustering (HAC) algorithm defined in [1] is exploited to group seeds on the base of a combination of a structural and a lexical distance.

3.4 Crosscutting Analysis of Concerns

A crosscutting matrix is traced down by the relationships among the type fragments located in different concerns. The matrix is generated by traversing the concern model instance, identifying the scattering and tangling at fragment level, for each couple of concerns. A matrix value $R_{ij}$ represents the number of fragments that are tangled between the Concern in row $i$ and the one in column $j$. The matrix is symmetrical as the definition of crosscutting we consider (equivalent to that provided by Kiczales [8] and its equivalent formalization provided by Ostermann [9]). In particular, given two concerns $A$ and $B$ and a crosscutting relationship ($\equiv$), it holds that:

$$A \equiv B \Rightarrow B \equiv A$$

(1)

$$A \not\equiv A$$

(2)

$$A \equiv B \wedge B \equiv C \Rightarrow A \equiv C$$

(3)

Given such definition, within the defined model, scattered concerns are the concerns containing at least two Type Fragments. Then two Concerns $C_x$ and $C_y$ are crosscutting if and only if: (i) they both contain more than one Type Fragment (hence generating scattering) and (ii) there are at least two Type Fragments $f \in C_x$ and $g \in C_y$, with the same fully qualified name (this means that the two concerns are introducing members or relationships on the same Type, thus generating tangling).

In order to support the process is automatically supported by a prototype tool called “Concern Analyser” (ConAN) [2] described in the next section.

4. The Tool “Concern Analyser”

The tool “Concern Analyser” (ConAn) [2] has been developed, on top of the Eclipse platform, to automatically perform and support the process described in the previous section. The tool architecture, depicted in Figure 5, is built on top of JDT tools. It is composed of three main components: the Model Handler, the JDT Extension, and, on top of these ones, the Presentation Layer. The JDT Extension adds to the JDT Core component the functionality needed to build a complete structural representation of a Java system taking into account program elements (types and their members up to methods and fields) and their relationships (inheritance, implementation and type nesting).

The Model Handler uses JDT Extension and EMF to generate an instance of the proposed model. The Model Handler traverses the type hierarchy graph of the system and extracts the information needed to find the seeds and to instantiate the model by associating a concern to each seed (the Model Generator component in the figure is responsible for that). In order to perform the clustering of seeds the tool provides both semi-automatic and automatic supports. The automatic one is based on the HAC algorithm, described in [1]. The manual clustering is supported by the
Model Handler and Model Editor components that provides the user with "merge" and "split" operations on instances of the model. The user is allowed to select which groups or seeds realize a single (more general) concern and give it a name and a description. The user can modify/modify the groups defined by the automatic HAC, too. When the user has decided which group of seeds realize a single concern, the tool performs a transformation on the model instance: all the concerns associated to selected seeds are merged together, as specified in Section 2.

The figure 8 shows a snapshot of the ConAn user interface. Several views, implemented in the Presentation Layer and grouped into an Eclipse Perspective, can be used to gain information about the internal structure of concerns or crosscutting.

On the bottom of the figure is the Model Editor rendering the structure of the model instance as a tree. This has been developed by a customization of the default editor generated by the Eclipse EMF Tools from the MOF meta-model specification. Some operations have been added to: perform the seed merging/splitting, compute some metrics, and to generate a UML2 class diagram for the internal structure of a concern.

The tool allows to compute dimensional metrics such as: the number of fragments, the number of fragments' members belonging to each concern, the Degree of Scattering for Classes (DOSC) [5]. This let to gain comprehension about the structure of a concern and to compare its scattering degree with the one of all other concerns (useful to find the most crosscutting concerns' members in each type).

On the right side of the Model Editor is a window showing the Crosscutting Matrix. By clicking on a matrix cell, a list of the involved fragments is provided and one of them can be located in the model editor.

At the left side of the model editor is a window with a UML2 class diagram of a concern showing
the type fragments composing it and their relationships. This is handled by the UML2 View that is integrated with UML2 Tools provided by the Eclipse Modeling Tools.

In the upper side of the figure 6, there are the windows representing the Crosscutting View. The Crosscutting View, is a customized version of the Eclipse Visualiser, that provides information about the crosscutting at the member granularity. This view, represents classes (or package) as boxes and each line in a class represents a member. Each colour represents a concern and each line has the colour of the concern the member is a part of. The boxes and the lines are associated to the model elements: a double-click on them allows to open types and members associated to a concern in both Java and Model editors.

The Visualizer Menu, on the right side of the Visualizer window in the Figure 6, allows a user to select the concerns of interest. The user can highlight all the concerns affecting a single module (class or package) by a click of left-button of the mouse.
5. Case Study

Several open source java software systems were analysed to verify and validate the assumption that Type Hierarchies are one of the cause of the introduction of crosscutting concerns, and the effectiveness of the method to identify the Seeds and the Type Fragments involved in the concerns implementation. The Table 1 reports the list of the largest analyzed systems. For each system, it is provided the name, the version number, the LOC size, the number of Types (classes and interfaces), the number of methods implemented in the system, the number of the identified seeds (distinguishing between Class and Interface seeds), the total number of Fragments in the concern meta-model instance, and the number of the Concerns resulting after the Seeds’ clustering step.

The first system is XWorks, a medium/small command-pattern framework used in J2EE environments, providing 'Inversion of Control' (IoC), an expression language, automatic data type
conversion, validation and pluggable configuration. The second and third systems are two major
versions of the HotDraw graphical editor framework designed to show good use of design patterns.
The fourth system, OSWorkflow, is a middle-sized real-world mature java workflow engine, while
the fifth, FreeMind, is an open-source mind-map tool written by Java. The last system is the por-
tion of the Java Foundation Classes (JFC) related to the I/O handling. It is a very large real-world
mature java software system heavily used in the industry. It allowed us to test the robustness
and the scalability of the performance vs. the size of the system to analyze of the method and the
prototype supporting tool.

The table 4 highlights that, in all the systems, all the Interfaces were found to be actually Seeds
of Concerns while just few classes (with respect to the total number of classes) were found as Seeds.
This indicates us that Interfaces are more prone to be associated to Concern (as we expected, being
them root of Hierarchies). The number of Type Fragments seems to be independent from the
number of Seeds and methods.

For the sake of brevity details and discussion about the results are provided only for the Free-
Mind system.

The initial instance of the model of the FreeMind system contained 176 initial concern Seeds,
these were grouped, by the clustering step, into the 29 Concerns reported in the Table 2, sorted
on the descending number of Seeds. For each Concern, they are reported: the number of Seeds
associated to it by the clustering step, the number of internal Type Fragments for both class and
interface Fragments and, in the last column, the Seeds resulted after a validation step carried out by
a “manual” code inspection made by an expert to assess the correctness of the results and assumed
as “gold standard” (the red and bold values highlights differences with the values resulting from
the automatic clustering). The “Minimap Model Actions” and “Types” were the largest clusters of
Seeds created by the clustering algorithm. The former concern implemented all the kind of actions
that users can perform in the system while the latter was devoted to the definition of all types of
mind-map nodes.

| Concern | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | Number of Fragments |
| Seeds   | 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| 176| Figure 6: Crosscutting Matrix of the FreeMind System.
The Figure 7 reports the Crosscutting matrix computed for the FreeMind system. The matrix allows to make an evaluation of the degree of crosscutting of each concern with respect to the number of tangled and scattered fragments. For instance, the "Event Handling" concern is crosscutting with 16 other concerns (of the 20 total) and has a very high number of fragments tangled with many of those concerns (121 fragments with "Action Validation", 121 with "Persistence" and 167 with "Types"). The same holds for concerns like "Unmarshallable Entities" and the "MindMap Model Actions". These concerns would be the most critical ones in re-engineering or migration approaches, requiring an effort to migrate them into aspects greater than the ones for concerns with lower values in the table.

![Crosscutting matrix for FreeMind](image)

**Figure 7**: FreeMind's crosscutting view of Persistence, Event Handling, Types and Action Validation concerns

A crosscutting view shows the members "belonging" to each concern in the system modularization units (at class or package level). The Figure 8 shows the crosscutting view for the "Action Validation", "Types" and "Event Handling" concerns at class level. Looking at the figure we can note that the most of the classes contain members from the these concerns. The figure allows to better highlights the high level of crosscutting of the concerns "Event Handling" and "Persistence". It's worthwhile to note that even if "Persistence" and "Event Handling" have a high degree of crosscutting, they also have several core units that are responsible to implement the main infrastructure of the concern. In the figure 8 the last boxes (i.e., classes) on the right side of the view are mainly devoted to such concerns and implement their main functionalities, that are invoked from other system components. This is not true for "Action Validation" concern that is completely decentralized (scattered within "Actions" classes). A validation to assess the quality of the results was done performing the identification of seeds and fragments and all the other method's activities "by hand" by an expert (i.e., the "gold standard"). In some cases the automatic clustering step grouped together concerns that were clustered in a different way by the manual analysis, producing a different set of concerns or some difference in the concerns' composition. For the FreeMind system, the last column of Table 2 highlights the clusters of seeds that differs from those identified by the expert. One of the major difference was reported for the Hooks concern. The clustering algorithm grouped together the concerns Hooks (made up of 5 class Seeds) and Hooks Registration (made up of 1 interface seed and 1 abstract class seed) into a single concern named "Hooks". The manual clustering considered them as separate concerns introducing a separate concern for "Hooks Registration" (not shown in the table). This however had a little impact on the model since the introduced concern was only 2 Fragments in size. Other differences from the expert choices were related to some types wrongly clustered in the Util concern, because of suffixes closer to other seeds in Util. This was for 1 Seed in Types ("TimeWindowConfigurationStorageType"), 1 Seed in MindMap Registry ("MindMapLinkRegistry seed") and 1 Seed in Views ("EdgeView"). In this case the changes in terms of number of Fragments were very small for the first two cases (3 Fragments from Util passed to Types and 2 to MindMap Registry) while more relevant for the EdgeView seed (12 Fragments passed from Util to View). Summarizing, we note that we had 8 Seeds wrongly clas-
tered, on 176 (less than 5%), and the addition of 1 concern on a total of 28 (less than 4%). Similar consideration are valid for the other analysed systems: for them the percentage of bad clusters was not greater than 7%. The worst result was obtained for the XWorks system in which several parts of the system seems to have been created automatically (maybe by a code generator) that makes limited the usage of interfaces. Summarizing, the results confirmed that Type Hierarchies are one of the main cause of introducing crosscutting concerns in OO systems, as well as that the analysis of the Type Hierarchies actually can reveal the static crosscutting concerns implemented in the system. Moreover the method used to identify such concerns by searching for Seeds and their Type Fragments showed to have a good level of effectiveness. The manual inspection required, in all the cases, a greater effort than one required by the method and the supporting tool (usually about an average of the 43% of more time was required). Of course the effort for manual analysis was depending on the system size, the number of seeds, and the number of fragments.

6. Related Work

The proposed approach is related to the ones about aspect mining techniques. Most aspect mining techniques are based on static source code analyses and manipulations. Others techniques focus on code execution analysis and elaborating the data gathered at run-time (dynamic techniques). Hybrid techniques, exploiting both structural and run-time information, are also used to improve the quality of static and dynamic approaches.

Marin et. al., in [4], propose a technique that combines together a metric based approach [7], a token-based one (exploiting identifier lexical analysis), and the dynamic based approach proposed in [11].

The approach proposed in [12] by Tonella and Cecatto focuses on crosscutting concerns produced by the scattered implementation of methods declared by interfaces that do not belong to the principal decomposition. Such interfaces, called “aspectizable” are identified and automatically migrated to aspects. This work shares with our approach the considerations that in OO systems, the interface can be used to model both main abstraction and secondary properties (i.e. they are Seeds for concerns). The main difference between the two approaches is the kind of elements analysed and the main goals. Our main goal was to verify that Type Hierarchies introduce seeds for crosscutting concerns, by analysing them at class level, i.e. without analysing the code inside methods bodies and their properties, while the approach of Tonella and Cecatto is based on the clustering of calls to methods (mining dynamic crosscutting concerns).

An approach to model concerns in source code that influenced the definition of our model was proposed in [10]. Since we are interested to build a static crosscutting representation taking into account the kind of relationships of a Java system, our model is much more dependent on language constructs of object-oriented single inheritance languages than the approach proposed in [10]. While this can restrict the range of applicability of our model, it allows to reason about the complete structure of the system, explicitly taking into account all the kind of modules, their members and their relationships (inheritance, implementation, overriding and type containment).

7. Conclusions and Future Work

The hypothesis that Type Hierarchies are one of the cause allowing the introduction of seeds for concerns that are crosscutting was verified to be valid. The method used to identify the Seeds, the Type Fragments composing the Concerns and their clustering into more general system’s concerns has been verified to have a good effectiveness and to be efficient. Indeed it allowed to find seeds and concerns with a good precision with respect the ones resulted by the ‘manual’ analysis carried out by an expert and to sensibly reduce the needed effort.

Future work will be mainly addressed to consider fragments of code inside methods in order to identify also dynamic crosscutting. Empirical validation of the approach will be carried out too.
References


Authors

Mario Luca Bernardi received the Laurea degree in Computer Science Engineering from the University of Naples “Federico II”, Italy, in 2003 and the Ph.D. degree in Information Engineering from the University of Sannio in 2007. Since 2003 he has been a researcher in the eld of software engineering and he is author of several papers published in journals and conference proceedings. His main research interests include software maintenance and testing, software reuse, software reverse engineering and reengineering and web engineering with particular interest on systems modularity, component-based software engineering and aspect-oriented paradigm. He has served as reviewer of papers submitted to conferences and journals in the eld of software engineering.
software maintenance and program comprehension.

Giuseppe A. Di Lucca received the Laurea degree in Electronic Engineering from the University of Naples “Federico II”, Italy, in 1987 and the Ph.D. degree in Electronic Engineering and Computer Science from the same University in 1992. He is currently an Associate Professor of Computer Science at the Department of Ingegneria of the University of Sannio. Since 1987 he has been a researcher in the field of software engineering and his list of publications contains more than 80 papers published in journals and conference proceedings. His main research interests include software engineering, software maintenance, software testing, reverse engineering, software reuse, software reengineering, software migration, aspect-oriented software development, and web engineering. He serves both as a member of the program and organizing committees of conferences, and as reviewer of papers submitted to some of the main journals and magazines in the field of software engineering, software maintenance and program comprehension.