Assessing Sofware Quality by Micro Patterns Detection

Giuseppe Destefanis

DIEE

University of Cagliari

Advisor: Prof. Michele Marchesi - Co-Advisor: Prof. Giulio Concas

Curriculum: ING-INF/05

Cycle XXV

A thesis submitted for the degree of

PhilosophiæDoctor (PhD) in Computer Science and Electronic Engineering

A.Y. 2011/2012

Contents

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

One of the goals of Software Engineering is to reduce, or at least to try to control, the defectiveness of software systems during the development phase. Software engineers need to have empirical evidence that software metrics are related to software quality. Unfortunately, software quality is quite an elusive concept, software being an immaterial entity that cannot be physically measured in traditional ways. In general, software quality means many things.

In software, the narowest sense of product quality is commonly recognized as absence or low incidence of bugs in the product. It is also the most basic meaning of confermance to requirements, because if the software contains too many functional defects, the basic requirement of providing the desired function is not met.

To increase overall customer satisfaction as well as satisfaction toward various quality attributes, the quality attributes must be taken into account in the planning and design of software.

To improve quality during development, we need models of the develompment process, and within the process we need to select and deploy specific methods and approaches, and employ proper tools and technologies. It is necessary to know measures of the characteristics and quality parameters of the development process and its stages, as well as metrics and models to help ensure that the development process is under control to meet the product's quality objectives.

Software quality metrics tend to measure whether software is well structured, not too simple and not too complex, with cohesive modules that minimize their coupling. Many quality metrics have been proposed for software, depending also on the paradigm and languages used there are metrics for structured programming, object-oriented programming, aspect-oriented programming, and so on.

The use of traditional metrics as quality indicators is very difficult. The Lines of Code (LOC) metric (very related to faults), is difficult to use, you can not say to a team of developers to develop classes by imposing a predefined number of lines of code. The use of the micro patterns (introduced by Gil and Maman (1)) metrics, that capture concepts of good or bad programming (like anti patterns) can help developers to focus on those classes that belong to categories of micro patterns prone to fault. The relationship between traditional metrics and micro patterns is useful for enabling these new metrics to evaluate software quality. Micro patterns are similar to design patterns, but their characteristic is that they can be identified automatically, and are at a lower level of abstraction with respect to design patterns (4).

This thesis tackles the problem of measuring software quality in Object Oriented (OO) systems by using such novel approaches based on micro patterns that can be a useful metrics in order to measure the quality of software by showing that certain categories of micro patterns are more fault prone than others, and that the classes that do not correspond to any category of micro patterns are more likely to be faulty. Many empirical studies were performed to validate empirically CK suite under these two aspects, showing an acceptable correlation between CK metrics values and software fault-proneness and difficulty of maintenance.

In OO, micro patterns can help to identify the portions of code that should be improved (for example those where encapsulation is not respected), and highlight other portions that make up good design practices. The design patterns, defined in the early nineties (4) were an important breakthrough at analysis and design level, but are difficult to be automatically supported at the coding level. There are tools claiming to help finding the usage of design patterns in code, but in practice they are used in a very limited way. On the contrary, micro patterns are defined at coding level, and it is relatively easy to recognize them automatically, thus being able to implement formal conditions on the structure of the class.

1.1 Thesis overview

The thesis is organized according to this scheme:

- Chapter 2 provides an overview of the concept of software metrics;
- Chapter 3 presents an overview of the design patterns catalogs (4);
- Chapter 4 discusses the micro patterns catalog using the definitions made by Gil and Maman (1);
- Chapter 5 discusses the interpretation of Micro Patterns given by Arcelli and Maggioni (2) (3);
- Chapter 6 present the study of the evolution of five particular micro patterns (anti patterns) in different releases of the Eclipse and NetBeans systems, and the correlations between anti patterns and faults. The analysis confirms previous findings regarding the high coverage of micro patterns onto the system classes, and show that anti patterns not only represent bad Object Oriented programming practices, but may also be associated to the production of worse quality software, since they present a significantly enhanced fault proneness.
- Chapter 7 present a study that aims to show, through empirical studies of open source software systems, which categories of micro patterns are more correlated to faults. Gil and Maman demonstrated, and subsequent studies confirmed, that 75% of the classes of a software system are covered by micro patterns. In this chapter is also analyzed the relationship between faults and the remaining 25% of classes that do not match with any micro pattern. We found that these classes are more likely to be fault-prone than the others. We also studied the correlation among all the micro patterns of the catalog, in order to verify the existence of relationships between them.
- Chapter 8 present a study on micro patterns in different releases of two software systems developed with Object Oriented technologies and Agile process. In this chapter we present some empirical results on two case studies of systems developed with Agile methodologies, and compare them to previous results obtained for non Agile systems. In particular we have verified that the distribution of micro patterns in a software system developed using Agile methodologies does not differ from the distribution studied in other systems, and that the micro patterns fault-proneness is about the same. We also analyzed how the distribution

of micro patterns changes in different releases of the same software system. We demonstrate that there is a relationship between the number of faults and the classes that do not match with any micro patterns. We found that these classes are more likely to be fault-prone than the others even in software developed with Agile methodologies

- Chapter 9 present the Java tool used in order to extract from the source code the informations about micro patterns distributions.
- Chapter 10 discusses the related works in the field.

2

Software Metrics

Measurement is an inherent and fundamental activity of all engineering disciplines that provides the ability to control activities, products and resources of a specific process. Compared with other traditional engineering disciplines, software engineering is the youngest and most immature, but it is surprising how fast, above all during the last decade, software engineering has increased its power to deliver and produce large, reliable and high quality software systems. New methodologies, techniques, process models and tools have been proposed and successfully applied to improve and support the production of high quality software systems. The aim of software engineering is to provide the technologies which apply an engineering approach to the development and support of software products and processes. Software engineering activities includes managing, planning, modeling, analyzing, designing, implementing, testing and maintaining. To achieve this goal measurement plays a key role, allowing to understand, control and improve software development processes and products. Measurement is fundamental during each step of software development:

- to support project planning;
- to determine the strengths and weaknesses of the current processes and products;
- to evaluate the quality of specific processes and products;
- to assess project progress;
- to take corrective actions and evaluate the impact of such actions.

Measurement becomes essential to provide visibility to entities and relationships, to quantitatively assessing problems and to suggest adequate solution scenarios. It happens at all different levels of details and abstraction, helping engineers to make the better decision about methodologies, techniques and tools to improve or to simply understand the status of processes, products and resources.

2.1 Software Metrics Classification

The first obligation of a measurement process is to establish what to measure. This is a general concept whose validity also address software engineering. Thus, the primary objective in applying a measurement approach upon software development is to identify the entities and attributes to be measured. Following Fenton (34) software entities may be of three different types:

- Processes: a collections of related software engineering activities.
- Products: an artifacts produced as output of any software engineering activity.
- Resources: an artifacts required as input of any software engineering activity.

The activities of a process are related in some way which depends on time. Resources and products are related with processes. Each process activity is characterized by the resources it uses and by the products it produces. Any artifact can be considered both a resources or a product according to the specific associated process. More specifically, an artifact produced by a specific activity can feed another activity. Measurement is the activity of quantify in some way any attribute of a process, product or resource. It is possible to characterize its attributes as:

- Internal Attributes of a process, product or resource are those that can be measured in terms of the sole process, product or resource. Thus, internal attribute can be measured considering the process, product or resource on its own, independently of its behavior.
- External attributes of a process, product or resource are those that can not be measured in terms of the sole process, product or resource. Thus, external attribute can only be measured considering the process, product or resource related

with the external environment and context. In this case the behavior becomes as important as the entity itself.

For any industrial process quality and productivity are indeed the most important parameters to be controlled and quantitatively assessed. This is true for industrial software development, too. Nevertheless, a formal definition of terms such like quality and productivity is far to be simple. Typically, business level roles, such as managers and customers, are more interested on external attributes, because quality and productivity are expressed in terms of such kind of attributes. For example, Boehem expressed software quality in terms of portability, reliability, efficiency, human engineering, testability, understandability and modifiability. Thus, quality attribute is decomposed in other external attributes, which in turn are still far to be simply definable. In practice, the decomposition goes on in order to finally express external attributes in terms of internal attributes, which are directly measurable. Similarly, McCall proposed an alternative model, which became the basis for the ISO 9126 software quality standard. The standard express quality again in terms of external attributes like functionality, reliability, usability, efficiency, maintainability and portability. While external attributes are highly important on business level, internal attributes are fundamental as they are the ones that can be directly measured. Internal attributes represent the last step on the decomposition of external attributes, that is, external attributes are typically quantified in terms of internal attributes. Early and modern approaches such like top-down design, low-coupling-high-cohesion structured design, object orientation and design patterns are all based on a common principle: good internal structure leads to good external quality. This principle underlines the tight relationships between internal and external attributes and points that the establishment of a good level of desirable internal attribute leads to the consequent establishment of a good level of the high abstraction desirable attributes.

2.2 Object-Oriented Metrics

This section starts with a synthetic description of OO concepts, to achieve the application and the extension of software measurement principles for OO systems. After that the main OO metrics proposed in the software engineering literature will be presented. OO is an optimal paradigm to successfully apply the divide et impera principle for managing the high complexity of large systems. OO systems are composed of objects which represent the abstraction of a real objects within an existing application domain. OO systems are made of interacting objects at runtime, which reside in the computer memory and hold the data to be processed. Objects are characterized by state and behavior; the state represents the internal information structure, whereas the behavior represents the way they can interact with each other. Objects interact by sending messages, and when this occurs they are coupled. Objects are described by classes, the basic OO building components; classes specify how to instantiate objects. People who create, maintain or improve OO software face problems such as handling software components. This mental burden can be described as cognitive complexity. High cognitive complexity can determine component with undesirable external qualities, like in- creased bug-proneness or reduced maintainability. Cognitive complexity depends on the internal characteristics of the class and its interconnections with others. OO metrics try to measure this complexity taking into account three elements:

- Coupling: it measures the static usage dependencies among class in an object-oriented system.
- Cohesion: it describes to which extent the methods and attributes of a class belong together.
- Inheritance: it evaluates to what extent classes reuse code.

2.2.1 CK Metrics Suite

Chidamber and Kemerer (35) have suggested a suite of six object oriented design metrics. The theoretical basis for the definition of such metrics, is the set of ontological principles proposed by Bunge (36) and later applied to the formal definition of object oriented systems by Wand and Weber (37). The Bunge ontology has largely interested researchers involved in the formal modeling of object orientation, since it deals with concepts inherent to the real world, which is the claimed basis of object oriented approach. According to Bunge ontology the world is viewed as composed of substantial individuals having a finite set of properties. These concepts can be formally translated and encapsulated in the object oriented paradigm definition, where an object is equivalent to a substantial individuals collectively considered with its properties. Similarly, attributes such as coupling, cohesion, complexity and inheritance can be defined in terms of Bunge ontology. This approach provides for the first time a theoretical and formal basis of metrics definition. As pointed by Chidamber and Kemerer themselves, not only previous attempt to define object oriented metrics, but more generally the large part of traditional metrics definition lacked such theoretical basis. In this section we will present the CK Suite metrics and discuss them in terms of their implementation and empirical validation. All the following definitions are directly referred to the original paper.

- *WMC - Weighted Methods per Class*

  Definition: consider a class C, with methods M1,...,Mn that are defined in the class. Let c1,...,cn be the complexity1 of methods. Then:

  $$\sum_{i=1}^{n} c_i$$

  If all method complexities are considered to be unity, then WMC is the number of methods (NOM) for the class C:

  $$WMC = NOM = n$$

  Interpretation: this metric measure class complexity. The number of methods and their complexity are predictors of the effort required to develop and maintain the class. The larger the number of methods, the greater the potential impact on children classes, as they inherit all methods of parent classes. Moreover, a class with a large number of methods are likely to be application specific, limiting the possibility of reuse.

- *DIT - Depth of Inheritance Tree*

  Definition: the DIT metric is the number of ancestors of a given class, that is the number of nodes (classes) to be crossed to reach the root of the inheritance tree. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.

  Interpretation: DIT is a meusere of how many ancestors may potentially affect a class. The deeper a class is in the hierarchy tree, the greater the probability of inherit a large number of methods, thus increasing class complexity. On the other hand, the deeper a class is in the inheritance tree, the greater the potential level of reuse of inherited methods.

- *NOC - Number of Children*

  Definition: the NOC metric is the number of immediate subclasses subordinated to a class in the class hierarchy.

  Interpretation: NOC is a measure of how a class can potentially affect its subclasses. The greater the number of children, the larger the level of reuse. On the other hand, a high value for this metric may warn a misuse of inheritance mechanism.

- *CBO - Coupling Between Object Classes*

  Definition: the CBO metric for a given class is a count of the number of other classes to which it is coupled.

  Interpretation: two classes are coupled when methods declared in one class uses methods or instance variables defined by other class. High coupling negatively affects modularity of the design. In order to promote encapsulation and improve modularity, class coupling must kept to the minimum.

- *RFC - Response For a Class*

  Definition: the RFC metric is the cardinality of the response set for a class. Given a class C, with methods M1,...,Mn, let Ri with i = 1, .., n to be the set of methods called by Mi. Thus, the response set for the class C is defined as:

  $$RC = \bigcup_{i=1}^{n} c_i$$

  the RFC metric is then given by:

  $$RFC = |RC|$$

  Interpretation: the response set for a class is the number of methods that can be potentially executed in response of a message received by an object of that class. This is a measure of the communication of the class with other classes in the system. The greater the number of methods that can be invoked by a class, the larger the complexity of the class. RFC gives a measure of the effort required for maintain a class in terms of testing time.

- *LCOM - Lack of Cohesion in Methods*

  Definition: Given a class C, with methods M1,...,Mn, let Ii with i = 1,..,n, to be the set of instance variables accessed by Mi. We define the cohesive method set and the non-cohesive method set as:

  $$CM = \{(M_i, M_j) | Ii \cap I_j \neq \phi, i \neq j\}$$

  $$NCM = \{(M_i, M_j) | Ii \cap I_j = \phi, i \neq j\}$$

  the LCOM metric is then given by:

  $$LCOM = \begin{cases} |NCM| - |CM|, & if\, |NCM| > |CM| \\ 0, & otherwise \end{cases}$$

  Interpretation: the LCOM metric is refers to the notion of degree of similarity in methods. The degree of similarity is formally given by:

  $$\sigma(M_i, M_j) I_i \cap = I_j$$

  The larger the number of similar methods, the more cohesive the class. This approach is coherent with traditional notion of cohesiveness, which is intended to account the interrelation of different part of a program. Cohesiveness of methods in a class is a desirable attribute, since it promotes encapsulation. Lack of cohesion implies classes should probably split into two or more specialized subclasses.

The object oriented approach has been claimed as a key technology to better manage software complexity and to provide improvement of systems quality. The object oriented research mainly has been studying the relationship between object oriented metrics and quality in terms of defects, if $|NCM| > |CM|$ otherwise specifically those reported during customer acceptance testing. Defect density is considered to be an indicator of fault proneness at class level and then a measure of maintenance effort. Similarly, defect density is reasonably related with extent of code change, which is again a surrogate measure of maintainability. Early studies found the existence of a relationship between defects and both traditional complexity measures, such as the McCabe ciclomatic complexity, and size measures, such as the LOC family metrics. More recently many empirical experiments have been performed to find similar relationships between defects and object oriented metrics such like the Chidamber and Kemerer metrics previously described.

Although the CK metrics have been validated by a large number of studies, the literature presents also criticism mainly on a theoretical basis (7) (8). Furthermore, the CK suite does not cover potential complexity that arises from certain other OO design factors such as encapsulation and polymorphism.

3

Design Pattern

3.1 Introduction

Design patterns were introduced in 1995 by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides almost as a philosophical current (4). A pattern describes a problem that frequently occurs and proposes a possible solution in terms of the organization of classes / objects that are generally recognized like a good solution to solve the problem.

Design patterns are characterized by four main elements:

- Name - mnemonic reference that allow to increase the vocabulary of technical terms and allows us to identify the problem and the solution in one or two words;
- Problem - a description of the problem and the context in which the pattern could provide a solution;
- Solution - describes the basic elements that constitute the solution and the relationships between these
- Consequences - specifies the possible consequences that the application of the proposed solution can provide. Refer for example to possible problems of space or efficiency of the solution, or with applicability to specific programming languages;

When a design pattern is generally accepted by a community of developers this pattern becomes an excellent tool for the documentation of the software. There are examples of applications fully described through the use of design patterns (JUnit is one of this).

The Design Pattern are referred to the reuse of a design point involving different aspects like:

- Economic aspects: they reduce the time and the cost for the design of the individual modules;
- Technical aspects: they reduce project risks and the possibility of errors in the project. Solution implemented and reused multiple times and you can predict a priori characteristics (eg behavior of non-functional)
- Anthropological Appearance: simplifies the understanding of the project by providing a clear third level of abstraction;

The definitions of collections of patterns in a specific application domain, is the first step to establish a context for reuse. In the book of the Gang of Four patterns are described by the following points:

- Name: summarizes the essence of the pattern. The objective is to expand the vocabulary of the project.
- Intent: It is a description that tries to answer to two questions: what does the Design Pattern do? Which design problems the dp aims to solve?
- Aka (Also known as): Any other names that identify the pattern.
- Motivation: Scenario that illustrates the problem and how the pattern provides a solution.
- Applicability: situations of applicability of the pattern and directions on how to recognize these situations.
- Structure: the graphical representation of the involved classes (using UML) and their relationships.
- Participants: the classes involved in the pattern, their relationships and their responsibilities (class diagram, object ed activity diagram)

- Collaborations: How the different classes work together to achieve the goals (sequence diagram).
- Consequences: advantages and disadvantages of the use of the pattern and possible side effects in the use of the pattern.
- Implementation: tips and techniques for implementation with reference to specific programming languages.
- Sample source code: code snippets that provide guidance on the implementation.
- Known uses: examples of the use of the patterns in existing systems.
- Related Patterns: Differences and most important relationships with other patterns and typical concomitant use.

The design patterns are classified by two criteria. The first criterion, called purpose, reflects what a pattern does. Patterns can have either creational, structural, or behavioral purpose.

- Creational patterns concern the process of object creation.
- Structural patterns deal with the composition of classes or objects.
- Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

The second criterion, called scope, specifies whether the pattern applies primarily to classes or to objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are staticfixed at compile-time.

Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labeled class patterns are those that focus on class relationships. Note that most patterns are in the Object scope. Creational class patterns defer some part of object creation to subclasses, while Creational object patterns defer it to another object. The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects. The Behavioral class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone.

Design patterns solve many of the day-to-day problems object-oriented designers face, and in many different ways. Here are several of these problems and how design patterns solve them.

Design Patterns help to simplify the design of a software system:

- Identification of necessary objects: decomposing a system into objects is the most difficult task in the object oriented design. The task is difficult because many factors come into play: encapsulation, granularity, dependency, flexibility, performance, evolution, reusability, and on and on. Design patterns help to identify less-obvious abstractions and the objects that can capture them. For example, objects that represent a process or algorithm don't occur in nature, yet they are a crucial part of flexible designs.

- Determining Object Granularity: objects can vary in size and number. They can represent everything down to the hardware or all the way up to entire applications. The application of a particular pattern implies the evaluation of the granularity of the system. There are patterns that allow to represent complete subsystems with very simple objects. (The Facade pattern describes how to represent complete subsystems as objects, and the Flyweight pattern describes how to support huge numbers of objects at the finest granularities.)

- Specifying Object Interfaces: Every operation declared by an object specifies the operation's name, the objects it takes as parameters, and the operation's return value. This is known as the operation's signature. The set of all signatures defined by an object's operations is called the interface to the object. An object's interface characterizes the complete set of requests that can be sent to the object. Any request that matches a signature in the object's interface may be sent to the object.

- Specifying Object Implementations: An object's implementation is defined by its class. The class specifies the object's internal data and representation and defines the operations the object can perform. There are patterns focused on inheritance, whereas other patterns are focused on composition.

3.2 Good programming practices

Class inheritance is basically just a mechanism for extending an application's functionality by reusing functionality in parent classes. CLass iheritance allow to define a new kind of object rapidly in terms of an old one and to get new implementations almost for free, inheriting most of what it is needed from existing classes.

The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition. Class inheritance lets you define the implementation of one class in terms of another's. Reuse by subclassing is often referred to as white-box reuse. The term white-box refers to visibility, with inheritance, the internals of parent classes are often visible to subclasses. Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or composing objects to get more complex functionality. Object composition requires that the objects being composed have well-defined interfaces. This style of reuse is called black-box reuse, because no internal details of objects are visible and objects appear only as black boxes.

Inheritance and composition each have their advantages and disadvantages. Class inheritance is defined statically at compile-time and is straightforward to use, since it's supported directly by the programming language. Class inheritance also makes it easier to modify the implementation being reused. When a subclass overrides some but not all operations, it can affect the operations it inherits as well, assuming they call the overridden operations.

Class inheritance has some disadvantages:

- it is not possible to change the implementations inherited from parent classes at run-time, because inheritance is defined at compile-time;
- parent classes often define at least part of their subclasses' physical representation.

Because inheritance exposes a subclass to details of its parent's implementation, it's often said that inheritance breaks encapsulation.

Object composition is defined dynamically at run-time through objects acquiring references to other objects. Composition requires objects to respect each others' interfaces, which in turn requires carefully designed interfaces that don't stop you from using one object with many others. But there is a payoff. Because objects are accessed solely through their interfaces, we don't break encapsulation. Any object can be replaced at run-time by another as long as it has the same type. Moreover, because an object's implementation will be written in terms of object interfaces, there are substantially fewer implementation dependencies. Object composition has another effect on system design: favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task. Your classes and class hierarchies will remain small and will be less likely to grow into unmanageable classes. On the other hand, a design based on object composition will have more objects (if fewer classes), and the system's behavior will depend on their interrelationships instead of being defined in one class.

3.2.1 Delegation

Delegation is a way of making composition as powerful for reuse as inheritance. In delegation, two objects are involved in handling a request: a receiving object delegates operations to its delegate. This is analogous to subclasses deferring requests to parent classes. But with inheritance, an inherited operation can always refer to the receiving object through the this member variable in C++ and self in Smalltalk. To achieve the same effect with delegation, the receiver passes itself to the delegate to let the delegated operation refer to the receiver.

3.3 Creational Patterns

Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created,composed, and represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object. Creational patterns become important as systems evolve to depend more on object composition than class inheritance. As that happens,emphasis shifts away from hard-coding a fixed set of behaviors toward defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones. Thus creating objects with particular behaviors requires more than simply instantiating a class.

Below there is a brief description of the creational patterns:

- Abstract Factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

- Builder: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

- Factory Method: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

- Prototype Pattern: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

- Singleton: Ensure a class only has one instance, and provide a global point of access to it.

3.4 Structural Patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfacesor implementations. As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together. Another example is the class form of the Adapter pattern. In general, an adapter makes one interface (the adaptee's) conform abstraction of different interfaces. inheriting privately from an adaptee interface in terms of the adaptee's.

Below there is a brief description of the structural patterns:

- Adapter: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

- Bridge: Decouple an abstraction from its implementation so that the two can vary independently.
- Composite: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Container: Meccanismo di ereditarieta che tiene conto dell'incapsulamento.
- Decorator: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Facade: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Flyweight: Use sharing to support large numbers of fine-grained objects efficiently.
- Proxy: Provide a surrogate or placeholder for another object to control access to it.

3.5 Behavioral Patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Below there is a brief description of the behavioral patterns:

- Chain of Responsibility: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Command: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

- Interpreter: Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Iterator: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Mediator: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Memento: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- Observer: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- State: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Strategy: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Visitor: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

4

Micro patterns catalog

Micro patterns are similar to design patterns, but are at a lower level of abstraction, closer to the implementation. These metrics could be used especially for the analysis of Object Oriented code because they are defined as class metrics; their functionality is to detect characteristic information of the entity class.

The term Micro derives from the application of this condition to the entity class; there are metrics such as milli patterns that determine conditions on packages or on cluster of classes, and the nano patterns metrics that determine conditions on the classes methods.

Gil and Maman cataloged 27 micro patterns, automatically recognizable, that are related to a variety of programming practices in Java from inheritance, to data encapsulation, to the emulation of typical practices of procedural programming (1). They developed a tool that implements the algorithms needed for the detection of micro pattern using a language based on First Order Predicate Logic (FOPL). However, they did not provide any detail on the implementation of these algorithms, leaving a certain degree of freedom on the interpretation of the definitions. For example, the condition all methods must be public can be interpreted in a restrictive manner by extending this condition even to inherited methods of a class, or applying it in a less restrictive way, only to methods declared by the class.

Design patterns (4), described in Chapter 3, were difficult to implement and it is not possible to recognize them in automated manner, micro patterns instead are designed to be mechanically recognizable by either the formal terms of the structure of the class.

Figure 4.1: Micro patterns catalog

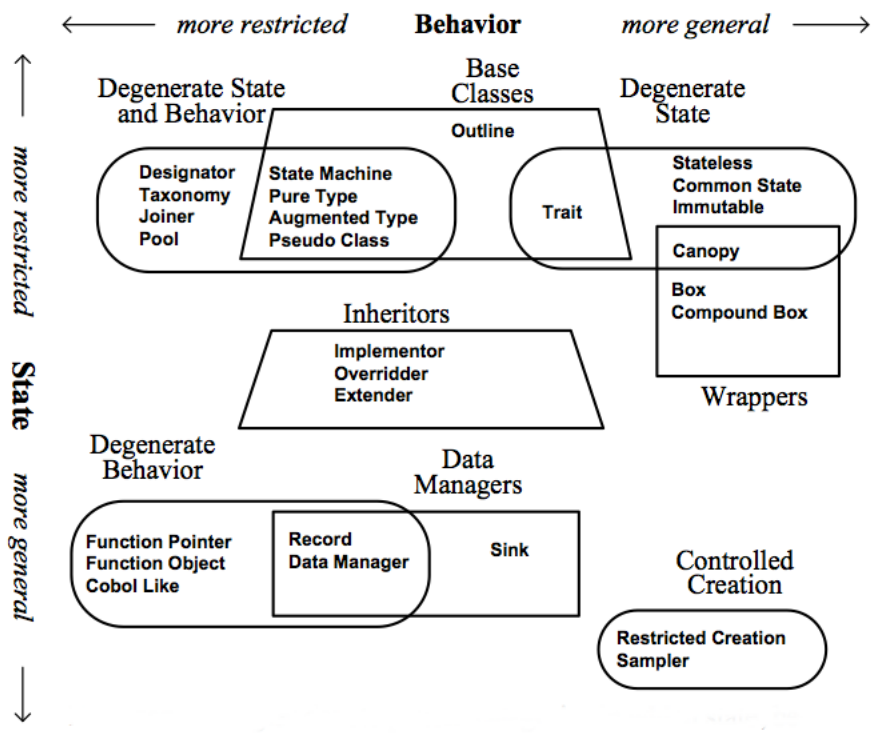


Fig 5.1 shows a global map of the catalog, including the 8 categories, and the palcement of the 27 micro patterns into these. The X dimension corresponds to class behavior and categories at the left hand side of the map are those of patterns which restrict the class behavior more than patterns which belong to categories at the right. The Y dimension of the figure correspond to class state: categories at the upper portion of the map are of patterns restricting the class state more than patterns which belong to categories at the bottom of the map.

Below there is a brief overview of all categories with the corresponding micro patterns given by Gil and Maman.

Out of the 27 patterns in the catalog, there are 21 patterns in which the state, behavior or the creation are degenerate in one way or another. This section describes the 12 patterns out of these which have no other category. The remaining 9 patterns are described together with their other respective category.

4.0.1 Degenerate State and Behavior

The first, and most simple category of micro patterns, includes those interfaces and classes in which both state and behavior are extremely degenerate. This degeneracy means, in most cases, that the class (or interface) does not define any variables or methods. Despite these severe restrictions, classes and interfaces which fall into this group are useful in tasks such as making and managing global definitions, class tagging, and more generally for defining and managing a taxonomy.

In addition to the patterns listed below, this category also contains the Pure Type, Augmented Type, Pseudo Class, and State Machine micro patterns which are described in the Base Classes category.

- Designator. The most trivial interface is an empty one. Interestingly, vacuous interfaces are employed in a powerful programming technique, of tagging classes in such a way that these tags can be examined at runtime. For example, a class that implements the empty interface Cloneable indicates (at run time) that it is legal to make a field-for-field copy of instances of that class. Thus, a Designator micro pattern is an interface which does not declare any methods, does not define any static fields or methods, and does not inherit such members from any of its superinterfaces. A class can also be Designator if its definition, as well as the definitions of all of its ancestors (other than Object), are empty. Pattern Designator is the rarest, with only 0.2% prevalence in the software corpus used by Gil and Maman. It was included in the catalog because it presents an important JAVA technique, which is also easily discernible.

- Taxonomy. Even if the definition of an interface is empty it may still extend another, potentially non-empty, interface. Consider for example interface DocAttribute (defined in package javax.print.attribute). This interface extends interface Attribute in the same package without adding any further declarations. Interface DocAttribute is used, similarly to the Designator micro pattern, for tagging purposes specifically that the attribute at hand is specialized for what is known as "Doc" in the JRE. An empty interface which extends a single interface is called a Taxonomy, since it is included, in the subtyping sense, in its parent, but otherwise identical to it. There are also classes which are Taxonomy. Such a class must similarly be empty, i.e., add no fields nor methods to its parent. Since constructors are not inherited, an empty class may contain constructors. A Taxonomy class may not implement any interfaces. This micro pattern is very common in the hierarchy of JAVAs exception classes, such as: EOFException which extends IOException. The reason is that selection of a catch clause is determined by the runtime type of the thrown exception, and not by its state.

- Joiner. An empty interface which extends more than one interface is called a Joiner, since in effect, it joins together the sets of members of its parents. For example, the interface MouseInputListener joins together two other interfaces: interface MouseMotionListener and interface MouseListener. An empty class which implements one or more interfaces is also a Joiner. For example, class LinkedHashSet marries together class HashSet and three interfaces Cloneable, Serializable and Set.
- Pool. The most degenerate classes are those which have neither state nor behavior. Such a class is distinguished by the requirement that it declares no instance fields. Moreover, all of its declared static fields must be final. Another requirement is that the class has no methods (other than those inherited from Object, or automatically generated constructors). A Pool is a class defined by these requirements. It serves a the purpose of grouping together a set of named constants. Programmers often use interfaces for the Pool micro pattern. For example, package javax.swing includes interface Swing-Constants which defines constants used in positioning and orienting screen components. The pattern, also called "constant interface anti-pattern", makes it possible to incorporate a name space of definitions into a class by adding an implements clause to that class.

4.0.2 Degenerate Behavior

The degenerate behavior category relates to classes with no methods at all, classes that have a single method, or classes whose methods are very simple.

- Function Pointer. Very peculiar are those classes which have no fields at all, and only a single public instance method. An example is class LdapNameParser (which is defined in package com.sun.jndi.ldap.LdapNameParser). This class has a single parse method, with (as expected) a string parameter. Instances of Function Pointer classes represent the equivalent of a function pointer (or a pointer to procedure) in the procedural programming paradigm, or of a function value in the functional programming paradigm. Such an instance can then be used to make an indirect polymorphic call to this function. The task of function composition (as in the functional programming paradigm), can be achieved by using two such instances.

- Function Object. The Function Object micro pattern is similar to the Function Pointer micro pattern. The only difference is that Function Object has instance fields (which are often set by the class constructor). Thus, an instance of Function Object class can store parameters to the main method of the class. The Function Object pattern matches many anonymous classes in the JRE which implement an interface with a single method. These are mostly event handlers, passed as callback hooks in GUI libraries (AWT and Swing). Hence, such classes often realize the COMMAND design pattern.

- Cobol like. Formally, the Cobol like micro pattern is defined by the requirement that a class has a single static method, one or more static variables, and no instance methods or fields. This particular programming style makes a significant deviation from the object oriented paradigm. Although the prevalence of this pattern is vanishingly small, instances can be found even in mature libraries. Beginner programmers may tend to use Cobol like for their main class, i.e., the class with function public static void main(String[] args) The prevalence of Cobol like is not high, standing at the 0.5% level in the corpus used by Gil and Maman. However, they found that it occurs very frequently (13.1%) in the sample programs included with the JAVA Tutorial (30) guides. The Degenerate Behavior category also includes two other patterns: Record, which has no methods at all, and Data Manager, in which all methods are either setters or getters. The two also belong in the Data Managers category, and are described below with the other patterns of that category.

4.0.3 Degenerate State

The Degenerate State category pertains to classes whose instances have no state at all, or that their state is shared by other classes, or that they are immutable. In this category Gil and Maman also find the Trait pattern which is defined under its other, Base Classes, category, and the Canopy pattern (defined under Wrappers).

- Stateless. If a class has no fields at all (except for fields which are both static and final), then it is stateless. The behavior of such a class cannot depend on its history. Therefore, the execution of each of its methods can only be dictated by the parameters. Micro pattern Stateless thus captures classes which are a named collection of procedures, and is a representation, in the object-oriented world, of a software library in the procedural programming paradigm. A famous example of the Stateless micro pattern is the Arrays class, from package java.util.

- Common State. At the next level of complexity, stand classes that maintain state, but this state is shared by all of their instances. Specifically, a class that has no instance fields, but at least one static field is a Common State. For example, the class System manages (among other things) the global input, output, and error streams. A Common State with no instance methods is in fact an incarnation of the modular

- Immutable. An immutable class is class whose instance fields are only changed by its constructors. The Canopy is an immutable class which has exactly one instance field. Its description is placed under its other category, Wrappers. More general is the Immutable micro pattern, which stands for immutable classes which have at least two instance fields. Class java.util.jar.Manifest is an Immutable class since assignment to its two fields takes place only in constructors code.

4.0.4 Controlled Creation

There are two patterns in this category, which match classes in which there is a special protocol for creating objects. The first pattern prevents clients from creating instances directly. The second pattern provides to clients ready made instances.

- Restricted Creation. A class with no public constructors, and at least one static field of the same type as the class, matches the Restricted Creation micro pattern. Many Singleton classes satisfy this criteria. A famous example is java.lang.Runtime.

- Sampler. The Sampler matches classes class with at least one public constructor, and at least one static field whose type is the same as that of the class. These classes allow client code to create new instances, but they also provide several predefined instances. An example is class Color (in package java.awt) with fields such as red, green and blue.

4.0.5 Wrappers

Wrappers are classes which wrap a central instance field with their methods. They tend to delegate functionality to this field. The main pattern in this category is Box. The case that the wrapper protects the field from changes is covered by Canopy. There are cases in which there is an auxiliary field; these are captured by pattern Compound Box.

- Box. A Box is class with exactly one instance field. This instance field is mutated by at least one of the methods, or one of the static methods, of the class. Class CRC32 (in the java.util.crc package) is an example of this micro pattern. Its entire state is represented by a single field (int crc), which is mutated by method update(int i).

- Canopy. A Canopy is a class with exactly one instance field which can only changed by the constructors of this cass. The name Canopy draws from the visual association of a transparent enclosure set over a precious object; an enclosure which makes it possible to see, but not touch, the protected item. Class Integer, which boxes an immutable int field, is a famous example of Canopy. As explained above, since the Canopy pattern captures immutable classes, it also belongs in the Degenerate State category.

- Compound Box. This is a variant of a Box class with exactly one non-primitive instance field, and, additionally, one or more primitive instance fields. The highly popular Vector class matches the Compound Box pattern.

4.0.6 Data Managers

Data managers are classes whose main purpose is to manage the data stored in a set of instance variables.

- Record. JAVA makes it possible to define classes which look and feel much like Pascal record types. A class matches the Record micro pattern if all of its fields are public and if has no methods other than constructors and methods inherited from Object. Perhaps surprisingly, there is a considerable number of examples of this pattern in the JAVA standard library. For example, in package java.sql there is the class DriverPropertyInfo which is a record managing a textual property passed to a JDBC driver.

- Data Manager. Experienced object-oriented programmers will encapsulate all fields of a Record and use setter and getter methods to access these. A class is a Data Manager if all of its methods (including 5 inherited ones) are either setters or getters. Recall that Data Manager micro pattern (just as the previously described Record) also belong to the Degenerate Behavior category.

- Sink. A class where its declared methods do not call neither instance methods nor static methods is a Sink. Class JarEntry of package java.util.jar.JarEntry is an example of Sink.

4.0.7 Base Classes

This category includes five micro patterns capturing different ways in which a base class can make preparations for its subclasses.

- Outline. An Outline is an abstract class where two or more declared methods invoke at least one abstract methods of the current (this) object. For example, the methods of java.io.Reader rely on the abstract method read(char ac[], int i, int j) Obviously, Outline is related to the TEMPLATE METHOD design pattern.

- Trait. The Trait pattern captures abstract classes which have no state. Specifically, a Trait class must have no instance fields, and at least one abstract method. The term Trait follows the traits modules of Ducasse, Nierstrasz and Black. For instance, class Number (of package java.lang) provides an implementation for two methods: shortValue() and for method byteValue(). Other than this implementation, class Number expects its subclass to provide the full state and complement the implementation as necessary.

- State Machine. It is not uncommon for an interface to define only parameterless methods. Such an interface allows client code to either query the state of the object, or, request the object to change its state in some predefined manner. Since no parameters are passed, the way the object changes is determined entirely by the objects dynamic type.

  This sort of interface, captured by the State Machine pattern, is typical for state machine classes. For example, the interface java.util.Iterator describes the protocol of the standard JAVA iterator, which is actually a state machine that has two possible transitions: next() and remove(). The third method, hasNext() is a query that tests whether the iteration is complete. In the state machine analogy, this query is equivalent for checking if the machines final state was reached.

- Pure Type. A class that has absolutely no implementation details is a Pure Type. Specifically, the requirements are that the class is abstract, has no static members, at least one method, all of its methods are abstract, and that it has no instance fields. In particular, any interface which has at least one method, but no static definitions is a Pure Type. An example is class BufferStrategy, which is found in package java.awt.image.BufferStrategy. As the documentation of this class states, it "represents the mechanism with which to organize complex memory". The concrete implementation can only be fixed in a subclass, since, "Hardware and software limitations determine whether and how a particular buffer strategy can be implemented.". Indeed, this class has nothing more than four abstract methods which concrete subclasses must override.

- Augmented Type. There are many interfaces and classes which declare a type, but the definition of this type is not complete without an auxiliary definition of an enumeration. An enumeration is a means for making a new type by restricting the (usually infinite) set of values of an existing type to smaller list whose members are individually enumerated. Typically, the restricted set is of size at least three (a set of cardinality two is in many cases best represented as boolean). For example, methods execute and getMoreResults in interface java.sql.Statement take an int parameter that sets their mode of operation. Obviously, this parameter cannot assume any integral value, since the set of distinct behaviors of these methods must be limited and small. This is the reason that this interface gives symbolic names to the permissible values of this parameter. Formally, an Augmented Type is a Pure Type except that it makes three or more static final definitions of the same type. Pattern Augmented Type pattern is quite rare (0.5%), probably thanks to the advent of the Enum mechanism to the language.

- Pseudo Class. A Pseudo Class is an abstract class, with no instance fields, and such that all of its instance methods are abstract; static data members and methods are permitted. A Pseudo Class could be mechanically rewritten as an interface. For instance, class Dictionary, the abstract parent of any class which maps keys to values, could be rewritten as an interface. Pseudo Class is an anti-pattern and is not so common; its prevalence is only 0.4%.

4.0.8 Inheritors

The three disjoint patterns in this category correspond to three different ways in which a class can use the definitions in its superclass: implementing abstract methods, overriding existing methods and enriching the inherited interface. The catalog does not include patterns for classes which mix two or more of these three.

- Implementor. An Implementor is a non-abstract class such that all of its the public methods were declared as abstract in its superclass. An example is class SimpleFormatter, which is defined in the java.util.logging package). This class has single public method, format(LogRecord logrecord), which was declared abstract by the superclass, Formatter (of the same package).

- Overrider. A class where each of its declared public methods overrides a non-abstract method inherited from its superclass. Such a class changes the behavior of its superclass while retaining its protocol. A typical Overrider class is the BufferedOutputStream class.

- Extender. An Extender is a class which extends the interface inherited from its superclass and super interfaces, but does not override any method.

For example, class Properties (in java.util) extends its superclass (Hashtable) by declaring several concrete methods, which enrich the functionality provided to the client. None of these methods overrides a previously implemented method, thus keeping the superclass behavior intact. Note that an Extender may be regarded as an instantiation of a degenerate mixin class over its superclass.

Table 4.1: Micro pattern definition by Gil and Maman

	Micro pattern	Definition
1	Designator	Interface with no members.
2	Taxonomy	Empty interface extending another interface.
3	Joiner	Empty interface joining two or more superinterfaces.
4	Pool	Class which declares only static final fields, but no methods.
5	Function Pointer	Class with a single public instance method, but with no fields.
6	Function Object	Class with a single public instance method, and at least one instance field.
7	Cobol Like	Class with a single static method, but no instance members.
8	Stateless	Class with no fields, other than static final ones.
9	Common State	Class in which all fields are static.
10	Immutable	Class with several instance fields, which are assigned exactly once, during instance construction.
11	Restricted Creation	Class with no public constructors, and at least one static field of the same type as the class
12	Sampler	Class with one or more public constructors, and at least one static field of the same type as the class
13	Box	Class which has exactly one, mutable, instance field.
14	Compound Box	Class with exactly one non primitive instance field.
15	Canopy	Class with exactly one instance field that it assigned exactly once, during instance contruction.
16	Record	Class in which all fields are public, no declared methods.
17	Data Manager	Class where all methods are either setters or getters.
18	Sink	Class whose methods do not propagate calls to any other class.
19	Outline	Class where at least two methods invoke an abstract method on this.
20	Trait	Abstract class which has no state.
21	State Machine	Interface whose methods accept no parameters.
22	Pure Type	Class with only abstract methods, and no static members, and no fields.
23	Augmented Type	Only abstract methods and three or more static final fields of the same type.
24	Pseudo Class	Class which can be rewritten as an interface: no concrete methods, only static field
25	Implementor	Concrete class, where all the methods override inherited abstract methods.
26	Overrider	Class in which all methods override inherited, non-abstract methods.
27	Extender	Class which extends the inherited protocol, without overriding any methods.

5

Interpretation of Micro Patterns

5.1 Introduction

The question posed by Gil and Maman in the original work about micro patterns is: "Can design be traced and identified in software?" .

The prime candidates of units of design to look for in the software are obviously design patterns. Despite the voluminous research ensuing it, attempts to automate and formalize design patterns are scarce. Specific research on detection of design patterns exhibited low precision, typically with high rate of false negatives. Indeed, as Mak, Choy and Lun (29) say, "...automation support to the utilization of design patterns is still very limited".

To overcome this predicament, Gil and Maman define the notion of traceable patterns, which are similar to design patterns, except that they are mechanically recognizable and stand at a lower level of abstraction. A pattern is traceable if it can be expressed as a simple formal condition on the attributes, types, name and body of a software module and its components.

The term "mechanically recognizable" means that there exists a Turing machine which decides whether any given module matches this condition. Thus, a condition "the module delegates its responsibilities to others" is not recognizable. On the other hand a predicate such as "each method invokes a method of another class with the same name", can be automatically checked.

It is required that these patterns are not random; they must capture a non-trivial idiom of the programming language which serves a concrete purpose. Yet, by definition, traceable patterns stand at a lower level of abstraction than that of the classical collection of design patterns (4). This is because traceable patterns are tied to the implementation language and impose a condition on a single software module.

5.2 A new interpretation

The original categorization of micro patterns consisted in eight overlapping categories, which are described in Chapter 5. In order to allow the detection of types presenting micro pattern flavours through a similarity-based approach, Arcelli and Maggioni first propose a categorization of micro patterns in three groups. Considering the definitions provided for the micro patterns, it is possible to note that each of them is based on one of the following three aspects:

- The analysis of the attributes belonging to a type;
- The analysis of the methods declared within a type;
- The analysis of both attributes and methods that characterize a type;

Arcelli and Maggioni define the set A as the set of micro patterns that can be identified by only analyzing the attributes of a type. Starting from (1), this set is clearly defined as A = Stateless, Common state, Immutable, Box, Compound box, Canopy, Trait. The authors define the set M as the set of micro patterns that can be identified by analyzing only the methods declared within a type. Therefore, M = Data manager, Sink, Outline, State machine, Implementor, Overrider, Extender. Finally, the set AM is the set containing those micro patterns that are identified by analyzing both the attributes and methods belonging to a certain type. Hence, AM = Designator, Taxonomy, Joiner, Pool, Function pointer, Function object, Cobol like, Restricted creation, Sampler, Record, Pure type, Augmented type, Pseudo class. These categories contain all the micro patterns, as M + A + AM =7+7+13=27.

Arcelli and Maggioni want to revisit the micro patterns also considering the NOA and NOM metrics, in order to support the detection of types presenting micro pattern flavours. To do so, they introduce three values:

- the attributes similarity ratio (ASR),
- the methods similarity ratio (MSR),

- the global similarity ratio (GSR).

For each micro pattern belonging to the A category, ASR measures the amount of attributes of a given type which satisfy the attributes conditions specified for that micro pattern, with respect to the total number of attributes declared within the type.

For each micro pattern in M, MSR calculates the amount of methods of a given type which satisfy the methods conditions specified for that micro pattern, with respect to the total number of methods declared within the type. Finally, for each micro pattern in AM, GSR considers both attributes and methods as being homogeneous entities of a type. Therefore, GSR measures the amount of attributes and methods (considered altogether) of a given type which satisfy the attributes and methods conditions specified for the micro pattern, with respect to the total number of attributes and methods declared within the type.

The attributes and methods considered in the evaluations are only those strictly belonging to the analyzed classes. The authors do not consider any inherited attribute or method.

ASR, MSR and GSR are calculated considering the NOA and NOM of each given type, hence taking into account the whole set of attributes and methods that characterize each of them. These similarity ratios are percentage rates, and are calculated in a different way depending on the micro pattern of interest.

Moreover, they are to be intended as an indication of how much a given type is similar to a certain micro pattern. The higher the value of these measures, the more the type is close to a correct and complete micro pattern realization. If a type has a similarity ratio of 100% to a certain micro pattern, the whole set of its attributes and/or methods satisfy the constraints specified by the micro pattern, and hence it is a precise instance of it.

Instances with a 100% similarity ratio are therefore those that can also be identified by the precise matching approach proposed by Gil and Maman (1).

For some micro patterns (Designator, Taxonomy, Joiner, Trait, Pure type, Pool and Record) Arcelli and Maggioni specify an upper bound of 3 or 5 methods and/or attributes defined by a type. This because the authors think it is too restrictive to consider only those types that do not define any attributes and/or methods at all. The authors verified that specifying a higher upper bound would result in detecting a larger number of instances presenting flavours of these patterns, but that are of scarce interest if we consider the purpose and the specifications of these micro patterns.

Designator:

- **MSR:** If $NOM{+}NOM\ inh. > U.B.$ then $MSR = 0$ else $MSR = 1 - \frac{NOM+NOM\ inh}{U.B.}$
- **ASR:** If $NOA{+}NOA\ inh. > U.B.$ then $ASR = 0$ else $ASR = 1 - \frac{NOA+NOA\ inh.}{U.B.}$

Taxonomy:

- **MSR:** If $NOM > U.B.$ then $MSR = 0$ else $MSR = 1 - \frac{NOM}{U.B.}$
- **ASR:** If $NOA > U.B.$ then $ASR = 0$ else $ASR = 1 - \frac{NOA}{U.B.}$

Joiner: the conditions for this micro pattern are the same of micro pattern Taxonomy, but in this case the number of extended interfaces must be equal or greater than two.

- **MSR:** If $NOM > U.B.$ then $MSR = 0$ else $MSR = 1 - \frac{NOM}{U.B.}$
- **ASR:** If $NOA > U.B.$ then $ASR = 0$ else $ASR = 1 - \frac{NOA}{U.B.}$

Pool:

- **MSR:** If $NOM > U.B.$ then $MSR = 0$ else $MSR = 1 - \frac{NOM}{U.B.}$
- **ASR:** in any case $\frac{static\ fields}{NOA}$

Function Pointer:

- **MSR:** If $public\ methods = 1$ then $MSR = 1$ else $MSR = 0$
- **ASR:** If $NOA = 0$ then $ASR = 1$ else $ASR = 0$

Function Object:

- **MSR:** If $public\ m. = 1$ then $MSR = 1$ else $MSR = 0$
- **ASR:** If $NOA = 0$ then $ASR = 0$ else $ASR = 1$

Cobol like:

- **MSR:** If $(static\ m. = 1)$ *and* $(NOM > 1)$ then $MSR = 0$ else $MSR = \frac{static\ methods}{NOM}$

- **ASR:** in any case $\frac{static\ fields}{NOA}$

Stateless:

- **MSR:** -
- **ASR:** If $NOA = 0$ then $ASR = 1$ else $ASR = \frac{static\ fields + final\ fields}{NOA}$

Common State:

- **MSR:** -
- **ASR:** If $NOA = 0$ then $ASR = 0$ else $ASR = \frac{static\ fields}{NOA}$

Immutable:

- **MSR:** If $NOM = 0$ then $MSR = 1$
- **ASR:** If $NOA > 1$ then
  $NOA = \frac{Number\ of\ fields\ modified\ by\ method}{NOA}$
  else $NOA = 0$

Restricted Creation:

- **MSR:** If $(public\ constructors = 0)$ then $MSR = 1$ else $MSR = 0$
- **ASR:** If $static\ same\ class\ fields >= 1$ then $ASR = 1$ else $ASR = 0$

Sampler:

- **MSR:** If $public\ constructors >= 1$ then $MSR = 1$ else $MSR = 0$
- **ASR:** If $static\ same\ class\ fields >= 1$ then $ASR = 1$ else $ASR = 0$

Box:

- **MSR:** -
- **ASR:** If $NOA = 1\ and\ (non\ final\ fields = 1)$ then $ASR = 1$ else $ASR = 0$

Compound Box:

- **MSR:** -

- **ASR:** If $NOA >= 1\ and\ (non-primitive\ fields = 1)$ then $ASR = 1$ else if $NOA > 1\ and\ (non-primitive\ fields > 1)$ then $ASR = 1 - \frac{non\ primitive\ fields}{NOA}$

Canopy:

- **MSR:** -
- **ASR:** If $NOA = 1$ then $ASR = \frac{Number\ of\ fields\ modified\ by\ constructor}{NOA}$ else $ASR = 0$

Record:

- **MSR:** If $NOM > U.B.$ then $MSR = 0$ else $MSR = 1 - \frac{NOM}{U.B.}$
- **ASR:** in any case $\frac{public\ fields}{NOA}$

Data Manager:

- **MSR:** If $NOM = 0$ then $MSR = 0$ else $MSR = \frac{(getter\ methods + setter\ methods)}{NOM}$
- **ASR:** -

Sink: a propagating method is a method which invokes at least another method within its body, either defined in the same class or in another type.

- **MSR:** If $NOM = 0$ then $MSR = 0$ else $MSR = \frac{propagating\ methods}{NOM}$
- **ASR:** -

Outline:

- **MSR:** If $Methods\ invoking\ an\ abstract\ method\ of\ the\ same\ class >= 1$ then $MSR = 1$ else $MSR = 0$
- **ASR:** -

Trait:

- **MSR:** If $abstract\ methods >= 0$ then $MSR = 1$ else $MSR = 0$
- **ASR:** If $NOA = 0$ then $ASR = 1$ else $ASR = 0$

State Machine:

- **MSR:** If $NOM = 0$ then $MSR = 0$
  else $MSR = \frac{propagating\ methods}{NOM}$
- **ASR:** -

Pure Type:

- **MSR:** $MSR = \frac{abstract\ methods}{NOM}$
- **ASR:** If $NOA = 0$ then $ASR = 1$ else $MSR = \frac{1}{NOA}$

Augmented Type:

- **MSR:** -
- **ASR:** If $(static\ final\ same\ class\ fields) >= 3$ then $ASR = 1$
  else $ASR = \frac{static\ final\ same\ class\ fields}{U.B.}$

Pseudo Class:

- **MSR:** In any case $\frac{(abstract\ m.+static\ m.)}{NOM}$
- **ASR:** in any case $\frac{static\ fields}{NOA}$

Implementor:

- **MSR:** If $NOM = 0$ then $MSR = 0$ else $MSR = \frac{implementing\ methods}{NOM}$
- **ASR:** -

Overrider:

- **MSR:** If $NOM = 0$ then $MSR = 0$ else $MSR = \frac{ovveriding\ methods}{NOM}$
- **ASR:** -

Extender:

- **MSR:** If $NOM = 0$ then $MSR = 1$ else $MSR = 1 - \frac{ovveriding\ methods}{NOM}$
- **ASR:** -

6

Anti patterns

In this chapter we aim to study and clarify the relationship between a subset of micro patterns and software defects, a field of study not covered in the literature on this topic, by analyzing software systems with a large number of classes. At the same time, we aim to confirm the following results found in the literature:

- three out of four classes match at least one micro pattern in the catalog (1);
- the definitions of micro patterns in terms of NOM and NOA can be validated. Specifically, we will show that some design principles, that can be related to the absence of specific micro patterns, yield better results than others, studying the relationship between micro patterns in term of the number of faults introduced in the software.

6.1 Definition of the anti-patterns

Let us consider the five micro patterns called anti-patterns, because they are associated to bad programming practices: Pool, CobolLike, Record, Pseudo Class, and Function Pointer.

We want to investigate their relationship with software defects. The anti-pattern Pool is defined when a class declares only fields of type static final, and declares no methods. Classes with this micro pattern are the most degenerate. In fact they are classes with neither state nor behavior. With respect to the state, it would be not enough to declare fields of type final, since final-but-not-static fields are simply constants, whose value can be different for different instances of the same class. This would represent the existence of a state. But in the case of static final constants, the value would be the same in different instances of the same class, representing a unique immutable state. Methods inherited from the class java.lang.Object and constructor methods are not considered. This anti pattern is also called "constant interface anti pattern", and good Object Oriented programming practices require this behavior to be assigned to an interface rather than to a class (13).

The Cobol Like anti pattern is a class having a single static method, one or more static variables, and no instance methods or fields. Cobol Like classes do not declare any method or instance field. Classes of this kind are very far from the Object Oriented programming paradigm and should be very rare. The Cobol Like anti pattern can be applied to classes and abstract classes.

The Record anti pattern is a class declaring only public fields and with no methods at all. Here again methods inherited from the java.lang.Object class and constructors are not considered. The Record micro pattern is very far from the Object Oriented programming paradigm since it violates the information hiding principle. It can be applied to classes and abstract classes.

The Pseudo Class is an abstract class declaring only abstract methods and not declaring fields which are not static. It can be substituted by an interface, thus must be considered an anti pattern. It can be applied to abstract classes.

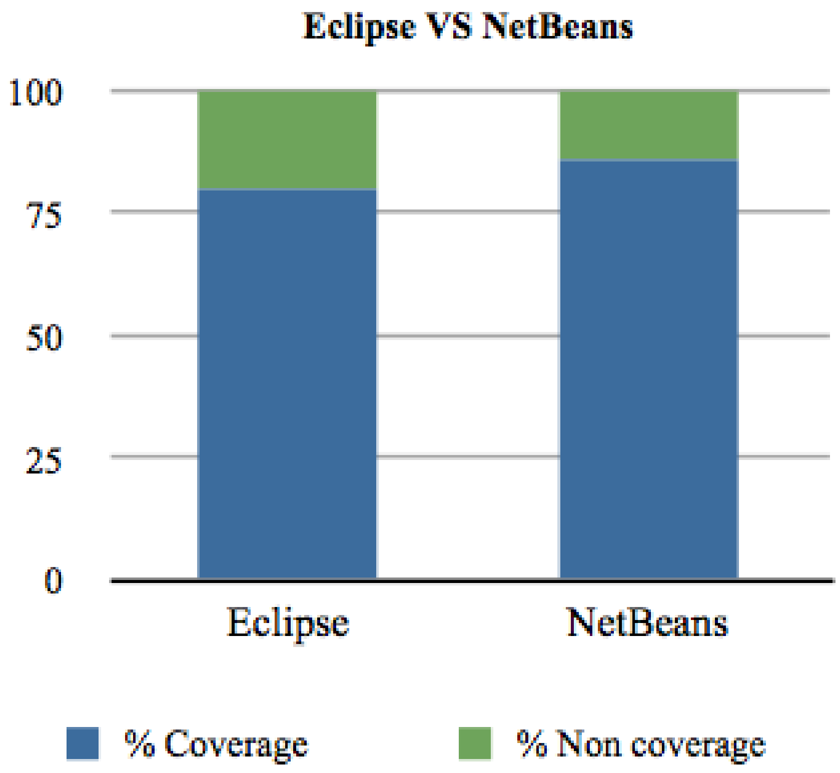
The Function Pointer micro pattern is a class declaring only one public method and no fields. The instances of this kind of classes are equivalent to the pointers to a function in the paradigm of procedural languages and consequently are away from the Object Oriented programming paradigm. It can be applied to classes and abstract classes.

Considering the standard definitions of Pool and Record micro patterns, Arcelli and Maggioni judge too restrictive the constraints that classes implementing them must not declare any method (2) (3). Thus they prefer to define an upper bound (U.B.) in order to consider valid instances of these patterns also classes defining at most 5 methods (U.B. set to 5). We also analyzed the parameter that evaluates the micro pattern considering both ASR and MSR, the GSR (Global Similarity Ratio), defined as GSR = min(ASR,MSR). GSR is a real number between zero (complete absence of the micro pattern) and one (presence of the micro pattern as defined in (1)). Intermediate values indicate a partial presence of the micro pattern.

6.2 Experimental results

In order to test the hypothesis formulated by Gil and Maman, the analyzed dataset includes twenty Eclipse releases and three NetBeans releases. It hosts a total of 335545 classes, Gil and Maman used 70000 classes for their study. Our results confirm their claims: "Three out of four classes match at least one micro pattern in the catalog" (1). The coverage relative to Eclipse is about the 80%, while in NetBeans it is about 86%, as reported in Fig. 6.1 for all the 27 micro patterns. This means that these systems can be characterized almost entirely in terms of micro patterns.

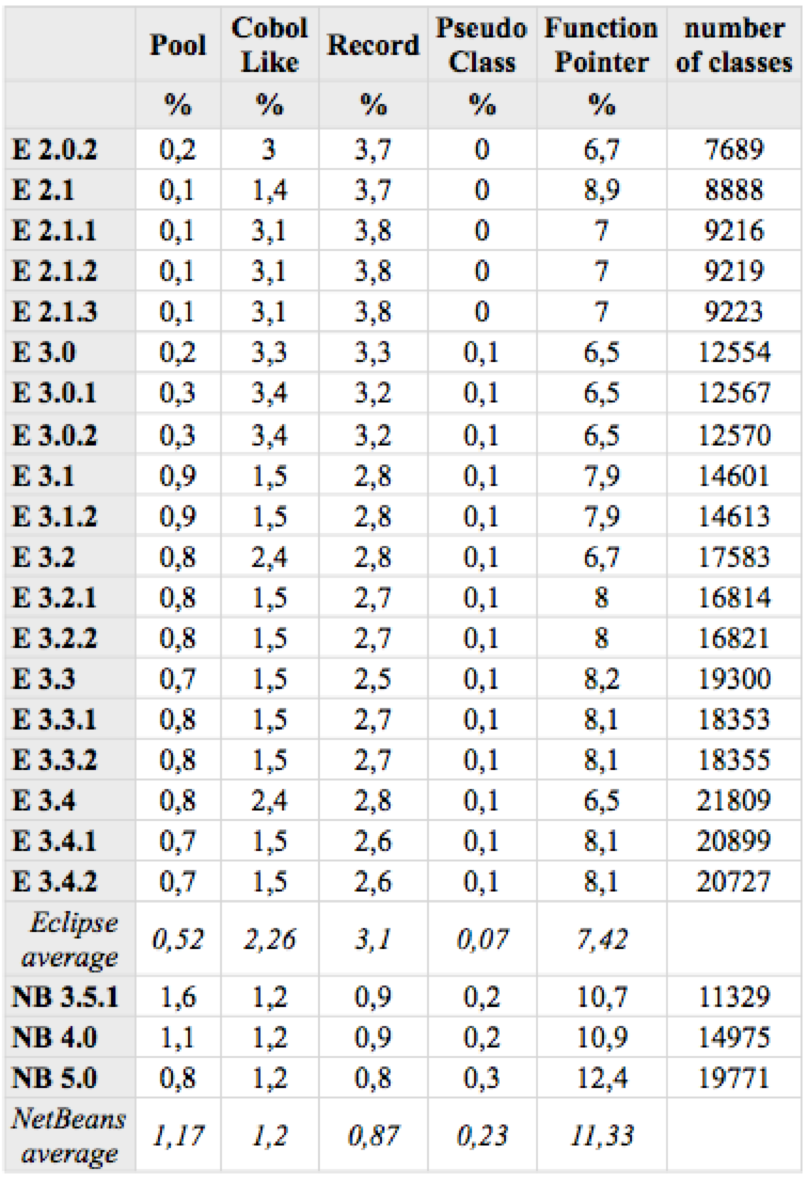
Figure 6.1: Coverage of all 27 micro patters



6.3 Relationship with faults

Figures 6.3, 6.4, and 6.5 report the relationships among anti patterns and faults in Eclipse 2.1, 3.0 and 3.1. In these tables we consider only perfect matches between

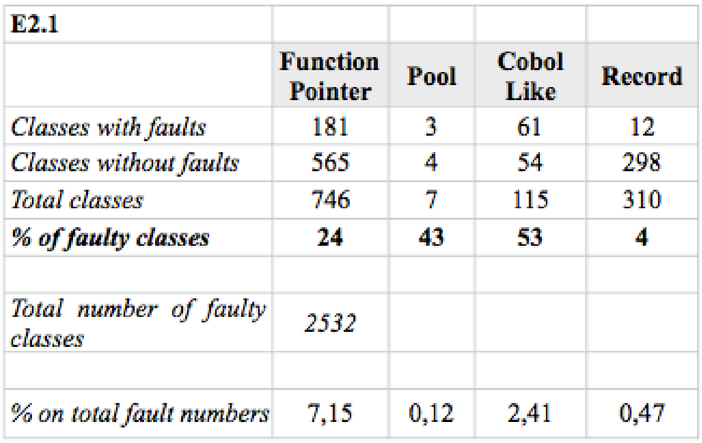
Figure 6.2: Percentual Presence of Anti Patterns



	Pool	Cobol Like	Record	Pseudo Class	Function Pointer	number of classes
	%	%	%	%	%	
E 2.0.2	0,2	3	3,7	0	6,7	7689
E 2.1	0,1	1,4	3,7	0	8,9	8888
E 2.1.1	0,1	3,1	3,8	0	7	9216
E 2.1.2	0,1	3,1	3,8	0	7	9219
E 2.1.3	0,1	3,1	3,8	0	7	9223
E 3.0	0,2	3,3	3,3	0,1	6,5	12554
E 3.0.1	0,3	3,4	3,2	0,1	6,5	12567
E 3.0.2	0,3	3,4	3,2	0,1	6,5	12570
E 3.1	0,9	1,5	2,8	0,1	7,9	14601
E 3.1.2	0,9	1,5	2,8	0,1	7,9	14613
E 3.2	0,8	2,4	2,8	0,1	6,7	17583
E 3.2.1	0,8	1,5	2,7	0,1	8	16814
E 3.2.2	0,8	1,5	2,7	0,1	8	16821
E 3.3	0,7	1,5	2,5	0,1	8,2	19300
E 3.3.1	0,8	1,5	2,7	0,1	8,1	18353
E 3.3.2	0,8	1,5	2,7	0,1	8,1	18355
E 3.4	0,8	2,4	2,8	0,1	6,5	21809
E 3.4.1	0,7	1,5	2,6	0,1	8,1	20899
E 3.4.2	0,7	1,5	2,6	0,1	8,1	20727
Eclipse average	*0,52*	*2,26*	*3,1*	*0,07*	*7,42*	
NB 3.5.1	1,6	1,2	0,9	0,2	10,7	11329
NB 4.0	1,1	1,2	0,9	0,2	10,9	14975
NB 5.0	0,8	1,2	0,8	0,3	12,4	19771
NetBeans average	*1,17*	*1,2*	*0,87*	*0,23*	*11,33*	

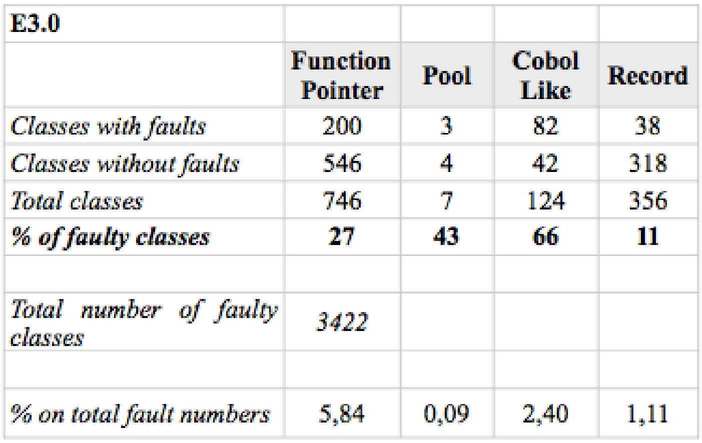
classes and anti patterns (GSR = 1). The anti pattern Pseudo Class appears only between 0% and 0.1% of classes in the 20 releases examined, thus we decided to neglect this anti pattern. The first row reports the number of classes exhibiting a given anti pattern and presenting at least one fault. The second row reports the number of classes exhibiting a given anti pattern with no faults. The third row reports the total number of classes into the system matching the anti pattern.

Figure 6.3: Eclipse 2.1



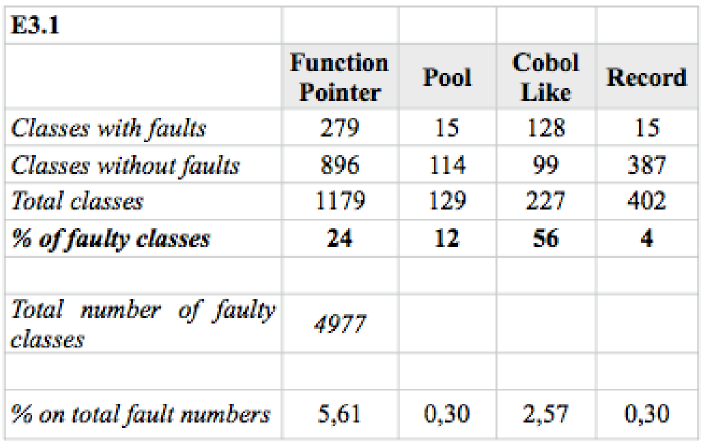
E2.1				
	Function Pointer	**Pool**	**Cobol Like**	**Record**
Classes with faults	181	3	61	12
Classes without faults	565	4	54	298
Total classes	746	7	115	310
% of faulty classes	**24**	**43**	**53**	**4**
Total number of faulty classes	*2532*			
% on total fault numbers	7,15	0,12	2,41	0,47

Figure 6.4: Eclipse 3.0



E3.0				
	Function Pointer	**Pool**	**Cobol Like**	**Record**
Classes with faults	200	3	82	38
Classes without faults	546	4	42	318
Total classes	746	7	124	356
% of faulty classes	**27**	**43**	**66**	**11**
Total number of faulty classes	*3422*			
% on total fault numbers	5,84	0,09	2,40	1,11

Figure 6.5: Eclipse 3.1

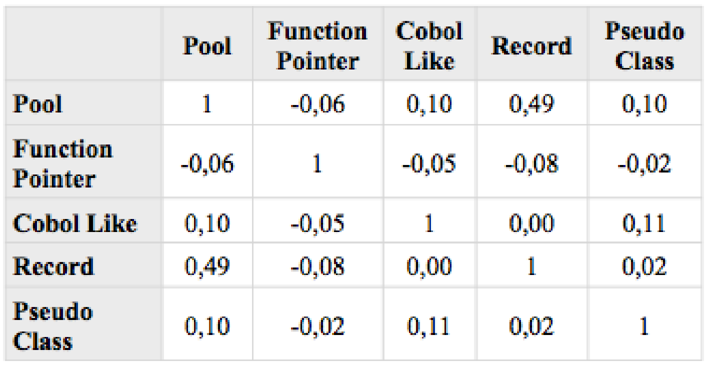


E3.1				
	Function Pointer	**Pool**	**Cobol Like**	**Record**
Classes with faults	279	15	128	15
Classes without faults	896	114	99	387
Total classes	1179	129	227	402
% of faulty classes	**24**	**12**	**56**	**4**
Total number of faulty classes	*4977*			
% on total fault numbers	5,61	0,30	2,57	0,30

6.4 Correlation between Anti Patterns

We characterized all software systems by a matrix of GSR (Global Similarity Ratio) coefficients, where each row identifies a class and each column contains a GSR coefficient for each micro pattern. The GSR coefficient is computed according to the definitions described in Chapter 5, and ranges from a minimum value of zero, denoting the absence of the micro pattern, to a maximum of one, denoting the presence of the micro pattern. A value between zero and one denotes partial presence for the micro pattern. The correlation between micro patterns provides information on the independence (or dependence) between a particular micro pattern and another. The results for the Pearson

Figure 6.6: Correlation between anti patterns



	Pool	**Function Pointer**	**Cobol Like**	**Record**	**Pseudo Class**
Pool	1	-0,06	0,10	0,49	0,10
Function Pointer	-0,06	1	-0,05	-0,08	-0,02
Cobol Like	0,10	-0,05	1	0,00	0,11
Record	0,49	-0,08	0,00	1	0,02
Pseudo Class	0,10	-0,02	0,11	0,02	1

correlation indicate that there is no correlation in general between the anti patterns, except among Pool and Record. This means that each anti pattern characterizes the system independently. The only significant correlation is 0.49 among Pool and Record. This is a quite expected result due to the definitions of the two micro patterns: the Record is a class that declares only public fields and no methods, while the Pool is a class that declares only

6.5 Discussion

The data shown tracked software defects only for three of the twenty Eclipse releases: 2.1, 3,0 and 3.1. It is interesting to compare these data for all the system classes with those obtained by our analysis for the anti patterns on Tables in figures 6.3, 6.4, and 6.5. Perhaps the most interesting result regards the Cobol Like anti pattern. In fact, while the use of any anti patterns should be avoided according to Object Oriented programming practices, our results show that in the particular case of the Cobol Like anti pattern such programming practice produces software more fault prone. A comparison among the average fault presence in all the Eclipse classes and in the Cobol Like classes indicates an enhanced fault proneness in the latter. We use the following indicators:percentage of faulty classes; average fault number; percentage of faults contained in the corresponding percentage of the total system (similar to the Pareto 80-20 rule).

The first indicator for Cobol Like classes is 53%, 66% and 56% in Eclipse 2.1, 3.0 and 3.1, respectively, while the values obtained for the overall system are 28%, 27% and 34%. The average fault number is 1.89, 3.19 and 1.93 for Cobol Like classes for the three Eclipse versions, while is 0.669, 0.867 and 0.866 for the overall three versions. As regards the last indicator, we have 3.6% of faults contained in 0.013% of system classes in Eclipse 2.1, 3.63% of faults contained in 0.009% of system classes in Eclipse 3.0, and 3.47% of faults in 0.015% in Eclipse 3.1. We computed the statistical t-test and Mann-Whitney test on these three indicators, which confirmed the higher fault proneness for Cobol Like classes to a high degree of significance.

With regards to other anti patterns, even if the indicators above do not show meaningful results for any increased fault proneness, the absolute values of faults for the corresponding classes suggest that the fault presence can be reduced eliminating the anti pattern classes. For example, the use of a Pool class may cause the presence of faults which could be avoided with the proper use of an interface in its place.

6.6 Threats to validity

We identified three main threats to the validity of our results. We examined 20 Eclipse versions and 3 NetBeans versions. Both are similar environments (IDE for Java programming), and thus may be not representative of all environments or programming languages. This constitutes a threat to the external validity of our findings. Eclipse and NetBeans are Open Source software. Commercial software is typically developed using different platforms and technologies, with strict deadlines and cost limitation, and by developers with different experiences. This might provide different micro pattern distributions, which is another threat for the external validity. Another threat regards the relationships among anti patterns and faults, which has been studied only for three Eclipse versions.

7

Micro pattern Fault Proneness

7.0.1 Introduction

In this chapter, we explore the relationship between use (or non-use) of micro patterns and the presence of defects in classes.

The 27 micro patterns proposed by Gil and Maman were shown by them to be present in 75% of classes they analysed. Some of those patterns are regarded as "anti-patterns" (13) representing practices that are considered to be poor design practice Thus classes can be divided into 2 categories: MP and NMP — those that match one or more of the 27 micro patterns, and those that match no micro pattern.

We examine the relationship between the category a class belongs to and the presence of defects in that class.

One potential complication in this kind of research is that there may be non-trivial relationships between the micro patterns themselves. As part of our research, we identify these relationships.

Our main research questions are:

- RQ1: Are micro patterns related to each other?
- RQ2: Are some micro patterns more fault-prone than others?
- RQ3: Does fault-proneness differ for NMP classes?

We show that there are relationships between micro patterns that have similar definitions, and our results about the correlation show that. Some micro-patterns are more fault prone than others.

It is important to highlight the case of the Extender micro pattern which is one of the most fault prone: this fact can open further discussion on the use of inheritance in Java and in OOP.

Regarding the NMP category, our studies show that these classes are more likely to fault.

7.1 Methodology

We used the Java tool described in Chapter 9 in order to extract from a generic software system the data relative to the micro patterns distribution.

We also analyzed the parameters that evaluate the micro pattern considering both ASR and MSR, the GSR (Global Similarity Ratio), defined as GSR = min(ASR,MSR). GSR is a real number between zero (complete absence of the micro pattern) and one (presence of the micro pattern as defined in (1)). Intermediate values indicate a partial presence of the micro pattern.

We divided the software systems analyzed for this work in two categories of different dimensions (in terms of number of classes) called Big Systems and Small Systems. The "Big Systems" category is composed by three Eclipse releases, and the "Small Systems" category is composed by Tomcat 6.0, Ant 1.6, Ant 1.7, Lucene 2.2 and Lucene 2.4.

For Eclipse we computed the number of faults in each class. We matched information recovered from software repositories such as Bugzilla (22) and Issuezilla, with commit operations performed by developers in CVS (23), associating bugs with the corresponding class. The bugs dataset for the Eclipse software projects are publicly available, as well as the commit operations. The details of the process of bugs extraction from the repositories can be found in (24) (25) (26). In order to calculate the faults of the Small Systems category we used the information taken from the Promise repository (27).

7.2 Results

Each software system analyzed is characterized by a GSR matrix where each row represents the value for a class and each column contain a GSR value for each micro pattern. The GSR coefficient ranges between zero and one. The correlation between columns of the GSR matrix provides an important information about the relationship between different micro patterns, for example if the matching of one micro pattern with a class implies the matching of an other micro pattern with the same class. All the correlation matrices of the systems analyzed are very similar (this demonstrates that the correlation between micro patterns is independent of the size of the system) and the correlation matrix presented in Table 7.1, is the correlation matrix of Eclipse 3.1.

Below we list the strongest correlations observed:

- *Joiner - Taxonomy*: the highest correlation value is between these two micro patterns. This is an expected result, because the set of Joiner classes is a subset of the Taxonomy classes. For the purpose of our study it is not necessary to differentiate Joiner and Taxonomy so we consider only the Taxonomy micropattern (which contains both definitions).
- *Common State - Stateless*: Another strong correlation occurs between the Common State micropattern and the Stateless micropattern. Even in this case the definitions are similar and the correlation is justified: when a class does not declare a state because it contains only constants, it is Stateless. But constants are static and consequently a class of this type is also Common State.
- *Compound Box - Box - Canopy*: A similar consideration can be made for micro patterns Compound Box and Box and for micro patterns Box and Canopy: the conditions for the first are a subset of the conditions for the second.
- *Pool - Stateless*: The correlation value between Pool and Stateless stems from the fact that the Stateless micropattern requires the presence of concrete methods (Stateless classes are Pool classes with methods).
- *Pure Type - Trait*: Among the micro patterns Pure Type and Trait there is a strong correlation because both require abstract classes.
- *Pseudo Class - Augmented Type*: Among the micro patterns Pseudo Class and Augmented type there is a strong correlation value because both require abstract methods and static fields.

Figure 7.1 shows the average percentage of presence of each micro pattern for Big System and for Small Systems. Table 7.2 shows, for each micro pattern, the proportion

Table 7.1: Correlation between micro patterns

	D	T	J	P	FP	FO	CL	S	C	I	RC	S	B	CB	CY	RD	DM	SK	OL	TT	SM	PT	AT	PC	IR	OR	EX
DESIGNATOR	1.00	0.52	0.52	-0.06	-0.09	-0.08	-0.05	-0.11	-0.12	-0.13	-0.04	-0.03	-0.13	-0.14	-0.10	-0.06	-0.06	-0.16	-0.04	-0.03	0.35	-0.04	-0.01	-0.02	-0.09	-0.14	-0.24
TAXONOMY	0.52	1.00	1.00	-0.03	-0.05	-0.04	-0.03	-0.06	-0.06	-0.07	-0.02	-0.02	-0.07	-0.07	-0.05	-0.03	-0.03	-0.08	-0.02	-0.02	0.26	-0.02	-0.01	-0.01	-0.05	-0.07	-0.12
JOINER	0.52	1.00	1.00	-0.03	-0.05	-0.04	-0.03	-0.06	-0.06	-0.07	-0.02	-0.02	-0.07	-0.07	-0.05	-0.03	-0.03	-0.08	-0.02	-0.02	0.26	-0.02	-0.01	-0.01	-0.05	-0.07	-0.12
POOL	-0.06	-0.03	-0.03	1.00	-0.06	0.09	0.07	0.58	0.50	0.22	0.06	-0.01	-0.08	-0.05	-0.06	0.31	-0.03	0.04	-0.02	-0.02	-0.05	-0.01	0.06	0.06	-0.04	-0.05	0.00
FUNCTIONPOINTER	-0.09	-0.05	-0.05	-0.06	1.00	-0.08	-0.05	-0.11	-0.12	-0.12	-0.03	-0.03	-0.12	-0.13	-0.09	-0.06	-0.06	0.16	-0.02	0.06	-0.08	0.04	-0.01	-0.02	0.10	0.13	-0.02
FUNCTIONOBJECT	-0.08	-0.04	-0.04	0.09	-0.08	1.00	0.02	0.05	0.05	0.01	0.00	-0.00	0.22	0.20	0.23	0.04	-0.05	0.04	-0.01	-0.03	-0.07	0.00	0.03	0.02	0.03	0.11	0.02
COBOLLIKE	-0.05	-0.03	-0.03	0.07	-0.05	0.02	1.00	0.31	0.51	0.21	0.08	0.10	-0.02	-0.00	-0.04	0.02	0.01	0.05	-0.01	-0.02	-0.04	-0.02	-0.00	0.10	-0.05	-0.06	-0.05
STATELESS	-0.11	-0.06	-0.06	0.58	-0.11	0.05	0.31	1.00	0.88	0.54	0.13	0.03	-0.15	-0.06	-0.11	0.14	0.00	0.03	0.02	-0.04	-0.10	-0.01	0.12	0.18	-0.06	-0.07	0.09
COMMONSTATE	-0.12	-0.06	-0.06	0.50	-0.12	0.05	0.51	0.88	1.00	0.51	0.27	0.12	-0.10	-0.02	-0.11	0.12	0.01	0.04	0.01	-0.04	-0.11	-0.02	0.11	0.18	-0.06	-0.08	0.06
IMMUTABLE	-0.13	-0.07	-0.07	0.22	-0.12	0.01	0.21	0.54	0.51	1.00	0.14	0.04	-0.17	-0.09	-0.13	0.52	0.01	-0.05	0.03	-0.05	-0.11	-0.02	0.11	0.14	-0.07	-0.07	0.04
RESTRICTEDCREATION	-0.04	-0.02	-0.02	0.06	-0.03	0.00	0.08	0.13	0.27	0.14	1.00	-0.01	0.00	0.02	-0.03	0.04	0.02	0.02	0.01	-0.01	-0.03	0.00	0.07	0.08	-0.03	-0.02	-0.02
SAMPLER	-0.03	-0.02	-0.02	-0.01	-0.03	-0.00	0.10	0.03	0.12	0.04	-0.01	1.00	-0.00	0.01	-0.00	0.01	0.06	-0.01	-0.00	-0.01	-0.03	-0.01	-0.00	0.02	-0.02	-0.00	0.04
BOX	-0.13	-0.07	-0.07	-0.08	-0.12	0.22	-0.02	-0.15	-0.10	-0.17	0.00	-0.00	1.00	0.69	0.77	-0.06	-0.03	0.05	-0.01	-0.05	-0.11	-0.00	-0.02	-0.03	0.06	0.12	0.03
COMPOUNDBOX	-0.14	-0.07	-0.07	-0.05	-0.13	0.20	-0.00	-0.06	-0.02	-0.09	0.02	0.01	0.69	1.00	0.55	-0.06	0.02	0.03	0.01	-0.05	-0.12	0.00	-0.01	-0.00	0.02	0.11	0.03
CANOPY	-0.10	-0.05	-0.05	-0.06	-0.09	0.23	-0.04	-0.11	-0.11	-0.13	-0.03	-0.00	0.77	0.55	1.00	-0.04	-0.06	0.06	-0.02	-0.04	-0.08	-0.00	-0.01	-0.02	0.05	0.14	0.00
RECORD	-0.06	-0.03	-0.03	0.31	-0.06	0.04	0.02	0.14	0.12	0.52	0.04	0.01	-0.06	-0.06	-0.04	1.00	-0.04	-0.02	-0.02	-0.02	-0.06	-0.02	0.04	0.03	-0.05	-0.06	-0.05
DATAMANAGER	-0.06	-0.03	-0.03	-0.03	-0.06	-0.05	0.01	0.00	0.01	0.01	0.02	0.06	-0.03	0.02	-0.06	-0.04	1.00	0.06	0.01	-0.02	-0.05	-0.01	-0.00	0.01	-0.04	-0.06	0.08
SINK	-0.16	-0.08	-0.08	0.04	0.16	0.04	0.05	0.03	0.04	-0.05	0.02	-0.01	0.05	0.03	0.06	-0.02	0.06	1.00	-0.03	0.11	-0.14	0.14	0.05	0.05	0.03	-0.03	-0.15
OUTLINE	-0.04	-0.02	-0.02	-0.02	-0.02	-0.01	-0.01	0.02	0.01	0.03	0.01	-0.00	-0.01	0.01	-0.02	-0.02	0.01	-0.03	1.00	0.19	-0.03	0.30	0.07	0.16	-0.03	-0.02	0.04
TRAIT	-0.03	-0.02	-0.02	-0.02	0.06	-0.03	-0.02	-0.04	-0.04	-0.05	-0.01	-0.01	-0.05	-0.05	-0.04	-0.02	-0.02	0.11	0.19	1.00	-0.03	0.71	-0.00	-0.01	-0.03	-0.02	-0.01
STATEMACHINE	0.35	0.26	0.26	-0.05	-0.08	-0.07	-0.04	-0.10	-0.11	-0.11	-0.03	-0.03	-0.11	-0.12	-0.08	-0.06	-0.05	-0.14	-0.03	-0.03	1.00	-0.04	-0.01	-0.02	-0.08	-0.12	-0.20
PURETYPE	-0.04	-0.02	-0.02	-0.01	0.04	0.00	-0.02	-0.01	-0.02	-0.02	0.00	-0.01	-0.00	0.00	-0.00	-0.02	-0.01	0.14	0.30	0.71	-0.04	1.00	0.09	0.15	-0.04	-0.04	-0.02
AUGMENTEDTYPE	-0.01	-0.01	-0.01	0.06	-0.01	0.03	-0.00	0.12	0.11	0.11	0.07	-0.00	-0.02	-0.01	-0.01	0.04	-0.00	0.05	0.07	-0.00	-0.01	0.09	1.00	0.63	-0.01	-0.01	-0.01
PSEUDOCLASS	-0.02	-0.01	-0.01	0.06	-0.02	0.02	0.10	0.18	0.18	0.14	0.08	0.02	-0.03	-0.00	-0.02	0.03	0.01	0.05	0.16	-0.01	-0.02	0.15	0.63	1.00	-0.02	-0.03	-0.01
IMPLEMENTOR	-0.09	-0.05	-0.05	-0.04	0.10	0.03	-0.05	-0.06	-0.06	-0.07	-0.03	-0.02	0.06	0.02	0.05	-0.05	-0.04	0.03	-0.03	-0.03	-0.08	-0.04	-0.01	-0.02	1.00	-0.05	-0.07
OVERRIDER	-0.14	-0.07	-0.07	-0.05	0.13	0.11	-0.06	-0.07	-0.08	-0.07	-0.02	-0.00	0.12	0.11	0.14	-0.06	-0.06	-0.03	-0.02	-0.02	-0.12	-0.04	-0.01	-0.03	-0.05	1.00	-0.05
EXTENDER	-0.24	-0.12	-0.12	0.00	-0.02	0.02	-0.05	0.09	0.06	0.04	-0.02	0.04	0.03	0.03	0.00	-0.05	0.08	-0.15	0.04	-0.01	-0.20	-0.02	-0.01	-0.01	-0.07	-0.05	1.00

of classes that match with that micro pattern, and the proportion of faults that are in classes that match with that pattern. We divided the table in two parts, in order to show the distribution of micro patterns in Small Systems (like Tomcat, Ant and Lucene) and in Big Systems (like Eclipse). Micro pattern Joiner is absent from Table 7.2, as explained in the Results section, and micro pattern Augment Type is absent because the presence of this micro pattern is negligible in the systems analyzed.

Considering the data, the more fault-prone micro patterns were as follows:

- Compound Box with 11.2 % of faults on the 8.1 % of the classes;
- Canopy with 5.6 % faults on the 3.1 % of the classes;
- Restricted Creation with 2.3 % of faults on the 0.6 % of the classes;
- Extender with 16.1 % of faults on the 14.6 % of the classes;
- Sampler with a 1.8 % of faults on the 0.9 % of the classes;
- Stateless with 2.1 % of faults on the 2.1 % of the classes.

Our results confirm those obtained by Kim et al. (28) about the fault-proneness of Compound Box, Sampler, and Common State micro pattern. On the other hand we found complitely new results for others micro patterns. For example in the Big Systems the micro pattern Extender shows a pronounced fault proneness since 14.5 % of classes contain 26.4% of faults (7.3). In particular NMP classes result more fault prone than others with a high significativity (table 4). It is interesting to check what is the fault-proneness of those classes that do not match with any of the micro patterns catalog (NMP category). We consider two categories of membership for the classes of a system:

- NMP: classes that do not matches any micro pattern in the catalog.
- MP: classes that matches with at least one micro pattern of the catalog;

The data in Table 7.3 shows that the classes that do not match with any micro pattern are more fault-prone than other classes. In the case of the Small System category there is an average of 33% on the 20% of total classes. The same happens in the categories of Large Systems, (three versions of Eclipse) where the average of faults is 31% on the 28% of total classes.

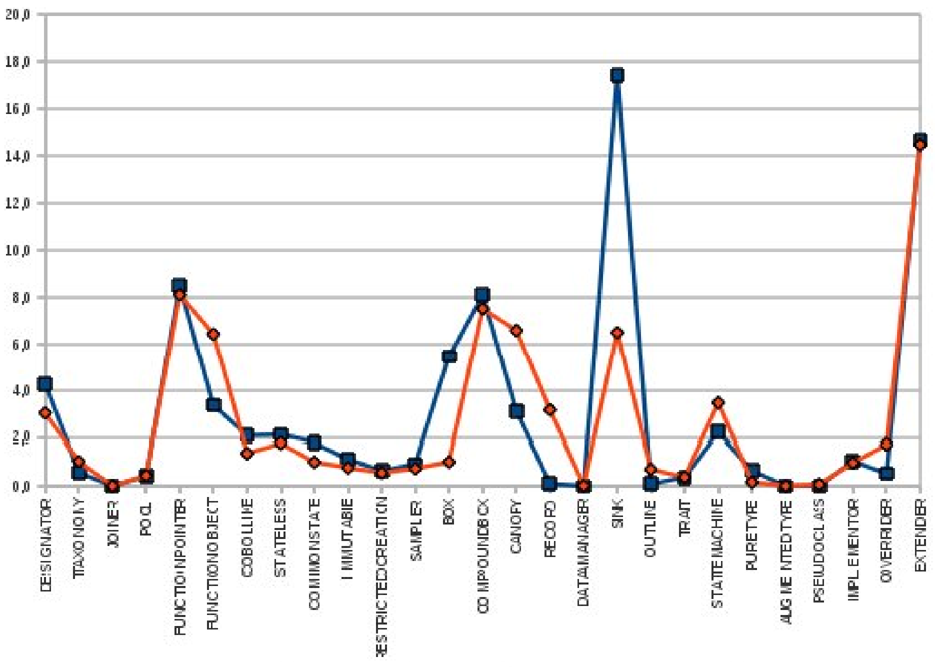
Table 7.2: Fault proneness

Micropattern	Small Systems		Big Systems	
	MP %	FAULT %	MP%	FAULT%
DESIGNATOR	4,3	0,2	3,0	1,0
TAXONOMY	0,6	0,1	1,0	0,1
POOL	0,4	0,4	0,4	0,1
FUNCTIONPOINTER	8,5	3,3	8,1	3,6
FUNCTIONOBJECT	3,3	2,3	6,4	4,8
COBOLLIKE	2,1	1,2	1,4	3,2
STATELESS	2,1	2,6	1,8	1,8
COMMONSTATE	1,8	1,7	1,0	1,4
IMMUTABLE	1,1	1,1	0,8	1,0
RESTRICTEDCREATION	0,6	2,3	0,5	2,1
SAMPLER	0,9	1,8	0,7	3,0
BOX	5,5	2,6	1,0	0,7
COMPOUNDBOX	8,1	11,2	7,5	7,7
CANOPY	3,1	5,6	6,6	4,7
RECORD	0,1	0,1	3,1	0,3
DATAMANAGER	0,0	0,0	0,0	0,0
SINK	17,4	10,8	6,5	1,9
OUTLINE	0,1	0,0	0,7	1,9
TRAIT	0,4	0,1	0,4	0,4
STATEMACHINE	2,2	1,8	3,4	0,7
PURETYPE	0,6	0,4	0,2	0,0
AUGMENTEDTYPE	0,0	0,0	0,0	0,0
PSEUDOCLASS	0,0	0,0	0,1	0,1
IMPLEMENTOR	1,0	0,9	1,0	0,5
OVERRIDER	0,5	0,5	1,7	1,2
EXTENDER	14,6	16,1	14,5	26,4

Table 7.3: Average of faults

		NMP	MP
Ant 1.6	% Classes	18	82
	% Faults	21	79
Ant 1.7	% Classes	20	80
	% Faults	22	78
Lucene 2.2	% Classes	23	77
	% Faults	44	56
Lucene 2.4	% Classes	19	81
	% Faults	32	68
Tomcat 6.0	% Classes	22	78
	% Faults	44	56
	Avg Classes	20	80
	Avg Faults	33	67
Eclipse 2.1	% Classes	28	72
	% Faults	27	73
Eclipse 3.0	% Classes	29	71
	% Faults	33	67
Eclipse 3.1	% Classes	28	72
	% Faults	32	68
	Avg Classes	28	72
	Avg Faults	31	69

Figure 7.1: Coverage of all 27 micro patterns (black line: Small Systems - grey line: Big Systems)



In order to verify the validity (in statistical terms) of the observed phenomenon, we performed the chi-square test on the observed data. We assumed to have from classes two populations belonging to MP classes, and n2 belonging to NMP classes, where each class has respectively the probability P1 or P2 to show the characteristic *A: "fault in the class"* . In a random sample from the first population, r1 members have the charateristic A and then a relative frequency equal to r1/n1; in the second population the relative frequency is r2/n2, with P1∼r1/n1 and P2∼r2/n2 for large numbers. The data can be exposed as a contingency table reported below.

Table 7.4: Contingency table

	A: Fault in the class		
	With fault	Without fault	
MP classes	r1	n1-r1	n1
NMP classes	r2	n2-r2	n2
	r1+r2	(n1-r1)+(n2-r2)	n1+n2

The total number of observations is indicated in the lower right, the four inner cells represent the observed frequencies.

We can therefore formulate the following: *Hypothesis H0: the relative frequency of the characteristic A is equal in the two populations. The difference observed in the samples is casual.*

To perform the test we consider a particular class and we check if it belongs to the MP classes category or to the NMP classes category, and if it presents a number of faults equal or greater than 1, it will become part of the population of the "classes with fault", otherwise it will became part of the population of "classes without fault".
The number of classes of the three Eclipse versions analyzed are sufficiently large to perform the test on each version.

Table 7.5: Contingency table Eclipse 2.1

	A: Fault in the class	
	With fault	Without fault
MP classes	1943	4276
NMP classes	796	1621

Table 7.6: Contingency table Eclipse 3.0

	A: Fault in the class	
	With fault	Without fault
MP classes	3441	5079
NMP classes	1654	1777

Table 7.7: Contingency table Eclipse 3.1

	A: Fault in the class	
	With fault	Without fault
MP classes	3460	6617
NMP classes	1517	2397

7.2.0.1 Big Systems

Eclipse 2.1 contains 8636 classes whose distribution is shown in Table 7.5. The chi-square test show the following results: $\chi^2 = 2.21$, $p = 0.136$. In this case the test turns out to be not significant. This fact emphasizes that the results obtained with the release of Eclipse 2.1 about the faults distribution, are different from those obtained in releases 3.0 and 3.1. In the release 2.1, the classes more fault-prononess appear to be those belonging to the MP category (even if only by one percentage point). The release Eclipse 3.0 contains 11951 classes whose distribution is shown in Table 7.6. The chi-square test show the following results: $\chi^2 = 60.8$, $p = 6.18e^{-15}$. In this case the test result is significant, and allows us to reject the hypothesis H0. We argue that the two classes of membership are not due to chance. The release Eclipse 3.1 contains 13991 classes whose distribution is shown in Table 7.7. The chi-square test show the following results: $\chi^2 = 23.86$, $p = 1.03e^{-6}$. Also in this case the test result is significant, and allows us to reject the hypothesis H0. We argue that, also for the release Eclipse 3.1, the two classes of membership are not due to chance.

Table 7.8: Contingency table Tomcat 6.0

	A: Fault in the class	
	With fault	Without fault
MP classes	46	598
NMP classes	31	155

Table 7.9: Contingency table Ant 1.6

	A: Fault in the class	
	With fault	Without fault
MP classes	70	218
NMP classes	24	39

7.2.0.2 Small Systems

The distribution of the classes for Tomcat 6.0 is shown in Table 7.8 The chi-square test show the following results: $\chi^2 = 14.44$, $p = 0.00014$. Also in this case the test result is significant, and allows us to reject the hypothesis H0. We can argue that even

Table 7.10: Contingency table Ant 1.7

	A: Fault in the class	
	With fault	Without fault
MP classes	137	492
NMP classes	36	80

Table 7.11: Contingency table Lucene 2.2

	A: Fault in the class	
	With fault	Without fault
MP classes	109	81
NMP classes	35	21

Table 7.12: Contingency table Lucene 2.4

	A: Fault in the class	
	With fault	Without fault
MP classes	158	116
NMP classes	45	20

for Tomcat 6.0 the two classes of membership are not due to chance, and the classes belonging to the category NMP are more prone to faults.

The contingency table obtained for Ant 1.6 is shown in Table 7.9. The chi-square test show the following results: $\chi^2 = 4.33$, $p = 0.037$. Also in this case the test result is significant, and allows us to reject the hypothesis H0.

The contingency table obtained for Ant 1.7 is shown in Table 7.10. The chi-square test show the following results: $\chi^2 = 4.19$, $p = 0.040$. Also in this case the test result is significant, and allows us to reject the hypothesis H0. The contingency table obtained for Lucene 2.2 is shown in Table 7.11. The chi-square test show the following results: $\chi^2 = 4.55$, $p = 0.033$. Also in this case the test result is significant, and allows us to reject the hypothesis H0. The contingency table obtained for Lucene 2.4 is is shown in Table 7.12. The chi-square test show the following results: $\chi^2 = 2.46$, $p = 0.11$. In this case the test turns out to be non significant.

According to these results, we can now answer to the research questions.

RQ1: Are micro patterns related to each other?

The answer is yes and the strong correlations are between Joiner - Taxonomy, Common State - Stateless, Compound Box - Box - Canopy, Pool - Stateless, Pure Type - Trait, Pseudo Class - Augmented Type.

RQ2: Are some micro patterns more fault-prone than others?

The answer is yes and the order is: Compound Box, Canopy, Restricted Creation, Extender, Sampler, Stateless.

RQ3: Does fault-proneness differ for NMP classes? The answer is yes. Data shown that the classes that do not match with any micro pattern are more fault-proneness than the other classes.

7.3 Threats to validity

Threats to external validity are related to generalization of our conclusions. With regard to the Big Systems category, we considered three versions of Eclipse, and this could affect the generality of the discussion. In future work we will consider different systems. With regard to the category of Small Systems, we analyzed two versions of Lucene and two versions of Ant (besides Tomcat), future studies should treat more systems. The software systems analyzed are Open Source software. Commercial software is developed using different platforms and technologies, with strict deadlines and cost limitation, and by developers with different experiences. This might provide different micro pattern distributions. Another threat regards the relationships among micro patterns and faults, which has been studied only for three Eclipse versions and for the Small Systems category.

8

Micro pattern in Agile software

8.1 Introduction

Given the purpose of micro patterns, a question naturally arises as to whether there is a relationship between the use of different patterns and the quality of the code. In particular there are no studies investigating the diffusion and the distribution of micro patterns in software systems developed using Agile methodologies (5).

In this chapter we will present the possible use of micro patterns metrics to indirectly assess the quality of the developed software, by showing the relationship between micro patterns and faults and in this context, we assess the ability of micro patterns to discriminate the usage of Agile practices.

We present results on different releases of two software systems on two industrial case-study. We understand that the presented evidence is anecdotal, but with real software projects it is very difficult to plan multi-project researches of this kind. This is because software houses tend to be very secretive about their projects. We hope that other researchers will try to replicate the presented results on similar projects whose data they can access.

The target of our research is the evolution of a software project consisting of the implementation of floss-AR, a program to manage the Register of Research of universities and research institutes. floss-AR was developed with a full object-oriented (OO) approach and released with GPL v.2 open source license.

The second system is a Web application, which has been implemented through a specialization of an open source software project, jAPS (Java Agile Portal System) (9), that is a Java framework for Web portal creation. This system is certified as a software developed using Agile methodologies.

In order to verify the use of Agile methodologies during the development phases of the analyzed systems, we submitted a questionnaire to the developers such as to have greater knowledge about Agile methodologies used.
We decided to organize our research answering to the following research questions:

- **RQ1:** Do software systems developed with Agile methodologies have a different distribution of micro patterns with respect to non Agile open source systems?
- **RQ2:** Is the micro patterns faults-proneness the same for Agile and non Agile software?
- **RQ3:** Does the micro patterns distribution change during software evolution? If yes, how?

8.2 Methodology

The goal is to investigate the possible relationship between Agile methodologies and micro patterns. We submitted to the developers of the floss-AR software system, a questionnaire in order to evaluate the effective use of Agile methodologies in the early stages of software development (20). We used the Java Tool described in Chpater 9 in order to extract from the software systems analyzed the data relative to the micro patterns distribution. We analyzed the two systems developed using Agile methodologies and we have studied the distribution and the evolution of micro patterns through different releases.

The micro patterns catalog contains several categories that in the literature are considered like anti patterns (15) as descriptive of bad programming practices not related to the object orientation techniques.

In (6) Destefanis et al. show that there are other micro patterns categories prone to fault and that the classes of a software system that does not belong to any category of micro patterns are more prone to faults. In this research we analyzed the different releases of the floss-AR system in order to verify if:

- also in this case there is a relationship between the number of faults and anti micro patterns;

- there is a relationship between number of faults and micro patterns more fault prone;
- there is a relationship between number of faults and classes that do not belong to any micro patterns category.

The analysis cannot have statistical significance (because it is performed on a single system), but it is however interesting and a good starting point to further studies. To establish the link between source code and fix operation we adopt the traditional heuristics proposed by Bachmann and Bernstein (14):

1. Scan through the change logs for bug report in a given format (e.g. fix bug, fix issue and so on).
2. Exclude all false-positive bug numbers (e.g. r420, 2009-05-07 10:47:39 -0400 and so on).
3. Check if there are other potential bug number formats or false positive number formats, add the new formats and scan the change logs iteratively.
4. Check if potential bug numbers exist in the bug- tracking database with their status marked as fixed.

Based on these heuristics we mine the source code repository (such as CVS and SVN) for commit that fixed a bug. Knowing how many time a class have been debugged and knowing the micro patterns associated (if any) to the class we could then evaluate the fault proneness of micro patterns for the system analyzed.

8.3 Results

In this section we present the results of the survey to developers and on the analysis performed on the source code of the Agile systems. In particular we show how the Agile development impacts on the micro patterns statistics, and on the fault proneness of micro patterns, anti patterns and the set composed by the classes that do not match with andy micro patterns of the catalog (no micro patterns category: NMP).

8.3.1 Survey

The results of the survey clearly show that Agile development has been applied for the floss-AR system. Tabs. 8.1 8.2 8.3 resume the survey's results.

The questions are divided in three groups according to the format of the possible answers. The first question requires an answer with 5 possibilities, in the second set the questions are posed in a YES or NO form, while in the third set the questions require a short sentence answer.

For developing floss-AR the following Agile practices have been applied:

- Pair programming
- Stand Up Meeting
- Refactoring
- On Site Customer

According to further discussions with the developers team, we are also able to identify four main phases of development:

- Phase 1 (Initial Agile): a phase characterized by the full adoption of all practices, including testing, refactoring and pair programming. This is the phase leading to the implementation of a key set of the system features. In practice, specific classes to model and manage the domain of research organizations, roles, products, and subjects were added to the original classes managing the content management system, user roles, security, front end and basic system services. The new classes include service classes mapping the model classes to the database, and allowing their presentation and user interaction.

- Phase 2 (Cowboy Coding): this is a critical phase, characterized by a minimal adoption of pair programming, testing and refactoring, because a public presentation was approaching, and the system still lacked many of the features of competitors' products. So, the team rushed to implement them, compromising the quality.

- Phase 3 (Refactoring): an important refactoring phase, characterized by the full adoption of testing and refactoring practices and by the adoption of a rigorous pair programming rotation strategy. The main refactorings performed were Extract Superclass, to remove duplications and extract generalized features from classes representing research products, and corresponding service classes, and Extract Hierarchy applied to a few big classes, such as an Action class that managed a large percentage of all the events occurring in the user interface. This phase was needed to fix the bugs and the bad design that resulted from the previous phase.

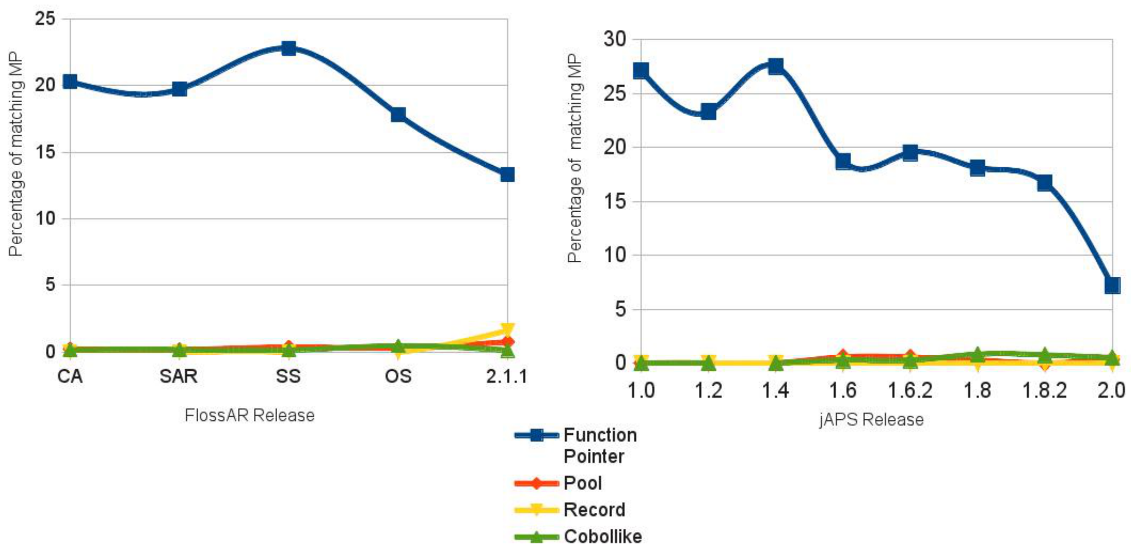
- Phase 4 (Mature Agile): Like Phase 1, this is a development phase characterized by the full adoption of the entire set of practices, until the final release.

8.3.2 Source code analysis

We next report the results on how Agile methodologies can impact on the micro patterns distribution and on the fault proneness of the code. In Tabs. 8.4 8.5 we report the micro patterns distributions for each release of the floss-AR and Japs systems, in order to show how such distributions evolve from one release to the next.

Both systems respect the Gil and Maman statement that about 75% of classes belong to at least one micro pattern. This means that micro patterns are good descriptors also for software developed with Agile methodologies. The distributions of micro patterns among classes roughly respect the same proportions found for software developed with traditional methodologies (6). In fact previous results show that Extender, Sink and Function Pointer are the most common micro patterns, while Taxonomy, Pool, Sampler and Record are almost absent. One key point is the behavior of anti patterns, which are indicators of bad programming practices (13). The overall anti patterns behavior is captured by Function Pointer, because classes belonging to others anti patterns, like Pool or Record, are a very small fraction of the total number of classes. Such behavior is displayed in Fig. 8.1 (left side), which shows an overall decreasing trend in the usage of anti patterns. This suggests that the constant application of Agile methodologies during software development across different releases may impact positively the software quality, carrying as side effect the reduction in the use of bad programming practices.

Figure 8.1: Left side: floss-AR - Right side: Japs



8.3.3 Micro patterns and faults

Next we examine the relationship among micro patterns and faults in the floss-AR releases. The top part of Tab. 6 shows the distribution of faulty classes among non micro patterns (NMP) and micro patterns (MP). It must be noted that NMP classes are only 25% of the total classes, and nevertheless they own the larger percentage of faulty classes, except for the last release, where the percentage of faulty classes is the same as the percentage of NMP in the entire release. This result for the first four releases is in agreement with those reported in (6), where NMP own most of the faults. This means that software developed through the adoption of Agile methodologies does not differ from other software with respect to such distribution. The result for the last release is somehow unexpected, and we cannot explain it with the data at our disposal. Further analysis are needed in order to understand the reasons for this inversion in the fault proneness.

The bottom part of Tab. 6 shows how faults are distributed among the different MP categories: anti micro patterns (AMP), fault-prone MP, and other MP, where fault prone MP are identified by the analysis performed in (6). Also in this case the total percentage of faulty classes in the last release is different than in previous releases, but the distribution among AMP, fault prone MP, and other MP is again respected. These results confirm that also in Agile systems the most fault prone micro patterns are Extender and Compound Box, and that also the AMP classes are more fault prone than others.

8.3.4 Discussion

According to these results, we can now answer to the research questions:
RQ1: Do software systems developed with Agile methodologies have a different distribution of micro patterns with respect to non Agile open source systems?
The answer to this research question is negative. According to tabs. 4, 5, the distributions of classes across micro patterns is roughly the same described in (6), where 8 systems were analyzed. They are very similar for both Japs and floss-AR, in all the releases analyzed. This result suggests that the use of Agile methodologies and programming practices does not influence the distribution of micro patterns in the classes.
RQ2: Is the micro patterns faults-proneness the same for Agile and non Agile software?
The answer to this question is positive except for the last release of floss-AR. Comparing the results obtained for the first 4 releases of floss-AR analyzed (Tab. 6, top part) NMP classes are by far the most fault prone classes. The more detailed analysis reported in Tab. 6 (bottom part) shows that among the classes matching with at least one micro pattern the Extender and Compound box micro patterns as well as the anti patterns are the most fault prone. This result confirms the findings reported in (6) and shows that the fault prone micro patterns distributions in Agile software is similar to the one found in systems developed without the adoption of Agile methodologies.
RQ3: Does the micro patterns distribution change during software evolution? If yes, how?
The answer to this research question is not univocal. In general we have shown that across all the releases the micro patterns distribution remains the same, with the exception of the anti patterns classes. In fact we found a decrease of the percentage of anti patterns classes in both systems across the releases. This may be related to the continuous adoption of Agile methodologies during development and maintenance.

8.4 Threats to validity

Threats to construct validity are related to the Agile methodologies not used during the system's development (like TDD and continuous integration). This may influence our conclusion that the use of agile methodologies may improve software quality, given that agile development has been adopted partially. Another threat to construct validity is related to the relationship between micro patterns and faults. We assume, based on previous works, that MP are related to software defectiveness. This result has not been generalized to all software systems, thus not necessarily the micro patterns catalogue is directly related to software defectiveness. Nevertheless we believe that our work can build a first step in this direction. Threats to internal validity are related to the fact that with different values of micro patterns could be possible to observe different correlations. Threats to external validity are related to generalization of our conclusions. With regard to the system studied in this work we considered only open source systems written in Java, and this could affect the generality of the discussion and thus our results are not representative of all environments or programming languages. Commercial software is typically developed using different platforms and technologies, with strict deadlines and cost limitation, and by developers with different experiences. This might result in different micro patterns distributions, which is another threat for the external validity. Another threat regards the relationships among anti patterns and faults, which has been studied only for the floss-AR system. Finally we have another threat to conclusion validity: there is not an estimated error on the recognition of a particular micro pattern for a given class.

Table 8.1: floss-AR developers survey (5 developers)

Question	Very good	Good	Discrete	Adequate	Not adequate
How would you describe the collaboration of the team?	4	1	0	0	0

Table 8.2: floss-AR developers survey (5 developers)

Question	Yes	No
The collaboration inside the team increased the productivity?	5	0
Did you take part in developing the whole system?	3	2
Do you have favourite programming styles?	2	3
Have the project decisions been discussed together with the team?	5	0
Did you interact directly with the customer?	4	1
Did you use refactoring?	5	0

Table 8.3: floss-AR developers survey (5 developers)

Question	Answer
Which Agile methodologies did you use during development?	• Pair Programming • Stand Up Meeting • Refactoring • On Site Customer
How often did you interact with the customer?	1-2 times per month
How often did you use refactoring?	2-3 times per month

Table 8.4: jAPS micropattern distribution (%)

MP	1.0	1.2	1.4	1.6	1.6.2	1.8	1.8.2	2.0
DESIGNATOR	2.14	1.79	2	3.3	3	4.32	6.83	9.6
TAXONOMY	0	0	0	0	0	0	0	0
POOL	0	0	0	0.55	0.54	0.27	0	0.35
JOINER	0	0	0	0	0	0	0	0
FUNCTIONPOINTER	27.1	23.3	27.5	18.7	19.5	18.1	16.7	7.18
FUNCTIONOBJECT	0.71	6.1	0	2.2	2.7	1.89	2.02	1.22
COBOLLIKE	0	0	0	0.27	0.27	0.81	0.75	0.5
STATELESS	0.71	0	1	0.82	0.82	1.08	1.01	1.22
COMMONSTATE	0	0	0	0	0	0	0	0.17
IMMUTABLE	0	3.2	0	0.82	0.82	0.81	0.75	0.87
RESTRICTEDCREATION	0.35	0.4	0.33	0.55	0.54	0.54	0.5	0.17
SAMPLER	0	0	0	0	0	0	0	0
BOX	4.64	15.4	3.98	0.27	0.27	0.27	0.25	1.4
COMPOUNDBOX	7.5	10	12.3	7.1	17.9	7.02	6.83	11.9
CANOPY	0	0	0	0	0	0	0	0
RECORD	0	0	0	0	0	0	0	0
DATAMANAGER	0.35	0.35	0	0	0	0	0	0
SINK	15.3	3.9	15.6	4.14	3.5	2.7	2.78	2.45
OUTLINE	0	0	0	0	0	0	1.0	0.35
TRAIT	0	0	0	0	0	0	1.3	1.1
STATEMACHINE	0.71	0	0.66	0.82	0.82	0.54	0.5	5.4
PURETYPE	0	0	0	0	0.5	0.8	0.3	0.2
AUGMENTEDTYPE	0	0	0	0	0	0	0	0
PSEUDOCLASS	0	0	0	0	0	0	0	0
IMPLEMENTOR	0	0.71	0.3	0.27	0.27	0.27	0.25	0.35
OVERRIDER	0	0	0.3	0.82	0.82	0.54	0.5	0.87
EXTENDER	25	27.9	27.5	36.1	35.9	37.2	34.6	25.1
TOTAL	84	73	85.7	77	76.6	76.4	74.4	75.1

Table 8.5: floss-AR micro patterns distribution (%)

MP	CA	SAR	SS	OS	2.1.1
DESIGNATOR	1.5	1.5	1.6	1.38	0.9
TAXONOMY	0	0	0	0	0
POOL	0.2	0.2	0.36	0.3	0.76
JOINER	0	0	0	0	0
FUNCTIONPOINTER	20.2	19.7	22.8	17.8	13.31
FUNCTIONOBJECT	2.5	2.4	2	4.45	1.53
COBOLLIKE	0.17	0.17	0.14	0.46	0.13
STATELESS	0.4	0.3	0.29	1.07	2.57
COMMONSTATE	0.2	0.2	0.14	0.15	0.06
IMMUTABLE	0.2	0.2	0.14	0.76	0.06
RESTRICTEDCREATION	0.1	0.1	0.29	0.30	0.06
SAMPLER	0	0	0	0	0
BOX	2	2	3.21	0.15	13.79
COMPOUNDBOX	7.9	8.2	7.45	10.4	12.61
CANOPY	0	0	0	0	0
RECORD	0	0.2	0	0.2	1.6
DATAMANAGER	0	0	0	1.68	1.74
SINK	18.9	18.6	17.2	3.53	14.77
OUTLINE	0	0	0	0.3	1.1
TRAIT	0.33	0.3	0.29	1.2	0.13
STATEMACHINE	0.17	0.17	0.29	0.15	0.06
PURETYPE	0	0	0	0.3	0.1
AUGMENTEDTYPE	0	0	0	0	0
PSEUDOCLASS	0	0	0	0	0
IMPLEMENTOR	1.7	1.22	1.46	2.61	0.69
OVERRIDER	0.33	0.34	0.29	1.07	0.2
EXTENDER	28.4	28.8	27.7	28.4	16.58
TOTAL	85.1	84.8	85.8	75.5	81.6

Table 8.6: FlossAR fault-prone analysis

		OS(%)	CA(%)	SAR(%)	SS(%)	2.1.1(%)
Distribution of faulty classes	NMP	63.12	62.41	71.63	70.92	23.4
among NMP and MP	MP	36.87	37.58	28.36	29.07	76.59
Percentage of MP faults						
Fault Percentage of AMP		12.76	12.05	7.8	7.8	23.4
Fault Percentage of fault-prone MP faults		18.43	14.89	11.34	13.47	32.62
Fault Percentage of other MP		5.67	10.63	9.21	7.8	20.56

9

The Java Tool

This chapter introduces the Java plug-in developed in order to extract the basic informations about the source code, which is essential for subsequent analysis based on micro patterns metrics.

9.1 The Eclipse Rich Client Platform

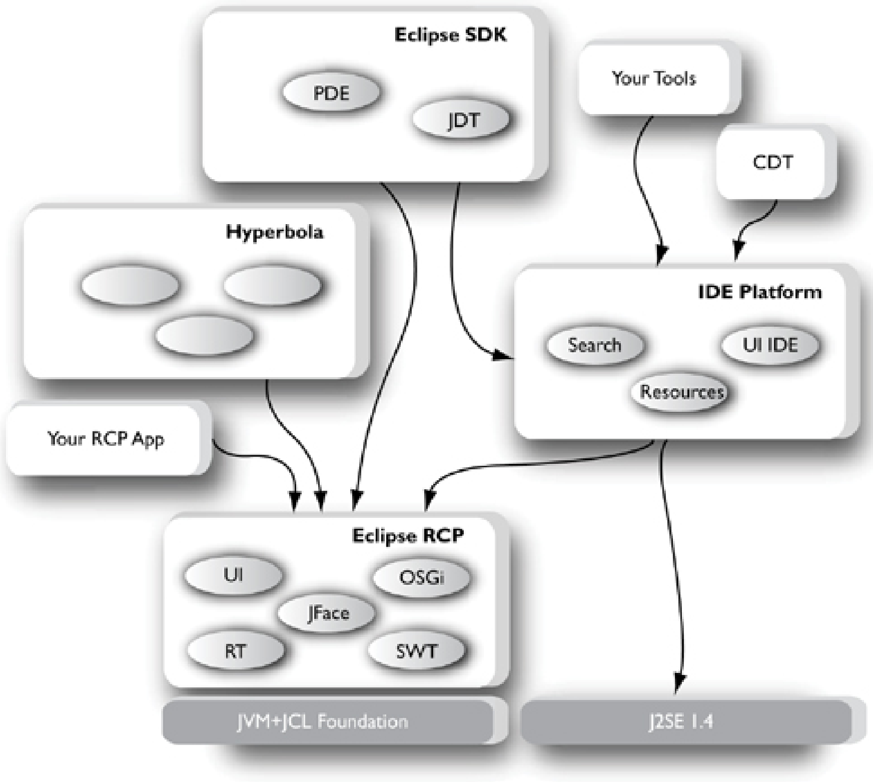
The term rich client was coined in the early 1990s with the rush to build client applications using the likes of Visual Basic and Delphi. The dramatic increase in the number and popularity of these client applications was due in part to the desire for a ”rich” user experience (33).

Rich clients support a high-quality end-user experience for a particular domain by providing rich native user interfaces (UIs) as well as high-speed local processing. Rich UIs support native desktop metaphors such as drag and drop, system clipboard, navigation, and customization. When done well, a rich client is almost transparent between end users and their workfostering focus on the work and not the system. The term rich client was used to differentiate such clients from terminal client applications, or simple clients, which they replaced.

The basic unit of functionality is called a plug-in (or a bundle in OSGi terms), the unit of modularity in Eclipse. Everything in Eclipse is a plug-in. An RCP application is a collection of plug-ins and a framework on which they run. An RCP developer assembles a collection of plug-ins from the Eclipse base and elsewhere and adds in the plug-ins he or she has written. These new plug-ins include an application and a product definition along with their domain logic. In addition to understanding how Eclipse manages plug-ins, it is important to know which existing plug-ins to use and how to use them, and which plug-ins to build yourself and how to build them.

Small sets of plug-ins are easy to manage and talk about. As the pool of plug-ins in your application grows, however, grouping abstractions are needed to help hide some of the detail. The Eclipse teams define a few coarse sets of plug-ins, as shown in Figure 9.1.

Figure 9.1: System architecture view



9.2 The Java plug-in

We developed a RCP plug-in in order to extract from a generic software system the data relative to the micro patterns distribution.

Although the application is born to be a Rich Client, this is an application intended for developers; we have preferred to focus on execution speed compared to a complex graphical user interface by removing some of the predefined classes.

The tool works in two steps: the first step consists in parsing the source code, and in generating a series of files containing information relative to the various classes, fields, methods, calls and so on; in the second step this set of files is examined for computing the micro patterns.

The files produced in the first steps are the following:

- *nodes.txt*: the list of classes and interfaces declared in the system. Contains: CU[1] full path, class name, class type, LOC. Class name is the name of the class (or abstract class, or interface). Class type can be: C (class), A (abstract class), I (interface).

- *methods.txt*: the list of all the methods defined in classes and interfaces of the system. Contains: CU full path, class name, method name, numbers of parameters, parameter list, method type, modifiers, LOC. Method name is the name of the method or constructor. Parameter list is the list of parameters separated by a space. Each parameter is represented by the full name of the class (package name.class name). Type method: m (method), c (constructor). Modifiers list is the list of modifiers of the method separated by a space (nothing is written if a modifiers is a default modifier).

- *fields.txt*: the list of all variables (instance variables and class variables) defined in the classes and interfaces of the system . The file contains: CU full path, class name, field name, field type, modifiers. Field name is the name of the variable (instace, classes variable, constant). Field type is the full name of the type of the field.

- *methodInvocation.txt*: the list of all calls to other methods contained within methods defined in the classes of the system. This file contains: CU full path, class name, name of calling method, number of parameters, lists of parameters, name of called method, number of parameters, lists of parameters, class of called method.

[1]The compilation units (CU) are files containing one or few classes. Most of them contain just one class (for example in Eclipse 90% of CU have just one file). Sometimes they may contain 2 or 3 classes, where one class is typically large, and the others are much smaller and support services for the first one.

The class of the called method is the full name of the class where is declared the called method.

- *fieldAccess.txt*: the list of all direct accesses to a instance variable or class variable (field) contained within thmethods defined in the classes of the system. This file contains: CU full path, class name, name of calling method, number of parameters, lists of parameters, the field name to which the method accesses, full name of the class that owns the field.
- *fieldWriteAccess.txt*: the structure is like that of FieldAccess.txt. In this file are reported only the fields in which access is a write operation. The method accesses the field to change the value. In the file FieldAccess instead are also reported fields to which the method leads to simple read operations.
- *links.txt*: the list of all the dependencies of a class system from another class, contained in system or not contained (including base classes as String, Date or Integer, but not primitive types like int or double). If a class depends on another more than once, there are many link declarations as there are the dependencies. The file contains: CU full path of starting class, full name of the starting class, full name of the arrival class, relation type.
  The relation type can be: D, IH, I.
  - D = dependence
  - IH = inheritance
  - I = implements

In the second step the tool calculates the presence of the micro patterns for each class of the analyzed system, using the files produced in the first step. The tool uses the definitions given by Arcelli and Maggioni. The class is assigned to only one micro pattern, the one with the higest GSR (defined in (2) (3)).

10

Related Works

After the work of Gil and Maman that defines the catalog of the micro patterns (1), several works appeared in this field. Arcelli and Maggioni suggest a novel approach to the detection of micro patterns which is aimed at identifing types that are very close and similar to a correct micro pattern implementation, even if some of the methods and/or attributes of the type do not comply with the constraints defined by the micro pattern (2) (3). The new interpretation is based on the number of attributes (NOA) and the number of methods (NOM) of a type. Kim et al. performed micro-pattern evolution analysis on three open source projects, ArgoUML, Columba, and jEdit, to identify micro pattern frequencies, common kinds of pattern evolution, and bug-prone patterns showing that the pattern evolution kinds that are bug-prone are somewhat similar across projects (28). The bug-prone pattern evolution kinds of two different periods of the same project are almost identical. Similar studies to those discussed in our work have been conducted for design patterns (4): Heuzeroth et al. presented an approach to support the understanding of software systems by detecting design patterns automatically using static and dynamic analyses (10).

Aversano et al. report an empirical study showing that for three open source projects, the number of defects in design-pattern classes is in several cases correlated with the scattering degree of their induced crosscutting concerns, and also varies among different kinds of patterns (11). Zimmer presented a classification of the relationships between design patterns, which led to a new design pattern and to an arrangement of the design patterns into different layers (31), and Noble describe and classifies the common relationships between object-oriented design patterns (32).

Tasharofi et al. (17) provide a set of high-level process patterns for Agile development which have been derived from a study of seven Agile methodologies based on a proposed generic Agile Software Process. These process patterns can promote method engineering by providing classes of common process components with can be used for developing, tailoring, and analyzing Agile methodologies.

Concas et al. in (16) studied and discussed the evolution of the classical software metrics and their behavior related to the Agile practices adoption level. The authors show that, in the reported case study, a few metrics are enough to characterize with high significance the various phases of the project. Consequently, software quality, as measured using these metrics, seems directly related to Agile practices adoption.

11

Conclusions

The objective of the work conducted during these three years of my PhD and described in this thesis, was to enable the use of micro patterns as a means of support to the evaluation of the quality of the source code. The starting point was the big job done by Francesca Arcelli and Stefano Maggioni in the redefinition of micro patterns in terms of NOM and NOA (2) (3), definitions at the base of the plug-ins that we developed for the analysis of source code.

- The objectives of the work described in Chapter 6 was to try to quantify the relationship between anti patterns (metrics that suggest a wrong OO programming practices) and faults in a software system. We observed from empirical studies on three Eclipse releases that 10% of the faults in these systems belong to classes classified as anti patterns. By implementing a refactoring of these classes that follow the implementation described in anti patterns definition (usually very simple classes to analyze), it might be possible to eliminate the 10% of faults acting on 13% of classes of the system. Note that these 13% classes are typically very simple - being without methods, or with just one method - and should anyway be refactored to comply with good OO style. Performing this refactoring should also enable to fix a small but not negligible percentage of all system faults. We chose to study a high number of Eclipse releases (20) to evaluate the temporal evolution of the percentage of occurrence of anti patterns in later versions of a software system, extending Gil and Maman empirical work (1). Finally, we found that the use of anti patterns causes an increment of the fault proneness in the interested classes. For the future we will extend our analysis to investigate the stabilization of a micro pattern over time in order to obtain information on the maturity of a software system.

- The objectives of the work described in Chapter 7 were to analyze the correlation between all the micro patterns of the catalog proposed by Gil and Maman, to analyze the fault proneness of each micro pattern and to analyze what happens in the classes that do not match with any micro pattern.

  We used the Java plug-ing (described in Chapter 9) in order to extract the data relative to the micro pattern distribution in a generic software system.

  We divided the software systems analyzed into two categories of different sizes (in terms of number of classes) called Big Systems and Small Systems, with the aim to understand if there are the same features in systems of different sizes. The "Big Systems" category is composed by three Eclipse releases, and the "Small Systems" category is composed by Tomcat 6.0, Ant 1.6, Ant 1.7, Lucene 2.2 and Lucene 2.4.

  We observed from empirical studies on the two categories that a correlation between different micro patterns exists and that the percentage of presence of each micro pattern in a system is very similar for Big Systems and Small Systems (as shown in Figure 7.1). With regard to the fault proneness of each micro pattern, we found that there are micro patters more fault prone than others. The final result regards the fault proneness of classes that do not match with any micro pattern of the catalog: we found that thess classes are more fault prone then classes that matching at least one micro pattern.

  The use of micro patterns may be helpful to evaluate the quality of a software project during the development process. A tool like the one used for the research conducted in the present work could be used in order to monitor the different stages of development, and possibly to control the temporal evolution of each category of micro patterns. It can be seen from our empirical results that classes that do not correspond to any micro patterns are more fault-prone and this supports that the use of a design methodology increases the quality of the code.

- The objectives of the work described in Chapter 8 were the analysis of micro patterns distribution in Agile open source software and the analysis of the relationship between MP-NMP and faulty classes. For the floss-AR system we analyzed the change log for bug report and extracted fix operation according to the traditional heuristic proposed in (14). We also submitted to the floss-AR developers team a questionnaire in order to evaluate the effective use of Agile methodologies, while for Japs this is certified on the web site (9).

Our analysis shows that the micro pattern distribution among classes is the same for the two systems, and remains roughly the same as the one found in non agile systems. Thus the adoption of agile methodologies does not influence such distribution. For example, Gil and Maman statement's that about 25% of classes does not match with any micropattern, is confirmed also in the two agile systems analyzed, for all the releases.

The analysis of fault prone classes shows that in agile systems the Extender and Compound box micro patterns are fault prone, as well as the anti micro patterns classes. In particular the most fault prone classes are those not belonging to any micro pattern. The last release of floss-AR represents an exception to this rule, even if the percentage of faulty classes belonging to NMP (23.4%), is still larger than the percentage of NMP classes in all the systems (18.4%).

Finally we found that the micro patterns distribution across the releases is unchanged, with the exception of the anti pattern classes, which displays a decreasing trend.

We can conclude that micro patterns may be helpful to evaluate the quality of an Agile software project during the development process. A tool like the one used in the present work could be used in order to monitor the different stages of development, and possibly to control the temporal evolution of each category of micro patterns. It can be seen from our empirical results that classes that do not correspond to any micro patterns are more fault-prone and this supports that the use of a design methodology increases the quality of the code.
Considering the natural adaptiveness of Agile development it could be useful to monitor the evolution of the most fault-prone micro patterns in order to increase the software quality and decrease the amount of defects.

12

Appendix

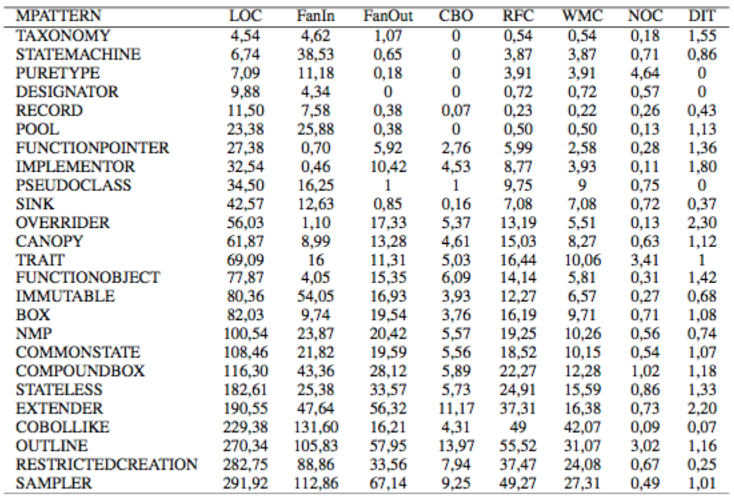
12.1 Consideration about the relationships between Micro Patterns and Traditional Software Metrics

Many empirical studies were performed to validate empirically CK suite under these two aspects, showing an acceptable correlation between CK metrics values and software fault-proneness and difficulty of maintenance. On the side of the relationship between micro patterns and software quality, our studies shown that there are different categories of micro pattern more prone to be faulty. In this appendix we present a preliminary analysis of the relationship between traditional software metrics and micro patterns in three versions of Eclipse (2.1, 3.0, 3.1).

For each category of micro pattern we have identified the corresponding classes and we calculated the average of the considered metrics evaluating three versions of Eclipse (2.1, 3.0, 3.1). It is interesting to see the average LOC (metric highly correlated with faults) in the different categories of micro patterns. Results are shown in fig. 12.1.

As regards the category named NMP, comprising the classes of the system that do not belong at any micro pattern of the catalog (25 % of the classes of the system), previous studies demonstrate that this is a category prone to faults. The fact that the average LOC value is not among the highest, indicates that these classes are not particularly complex (like core classes of the system). Another result regards anti patterns. While these are usually considered bad programming practices they in general present low values on average, for some critical metrics with respect to other micro patterns. For example Function Pointer possess an average a low value of the LOC

Figure 12.1: OO class metrics (Average for each categories of micro patterns)



MPATTERN	LOC	FanIn	FanOut	CBO	RFC	WMC	NOC	DIT
TAXONOMY	4,54	4,62	1,07	0	0,54	0,54	0,18	1,55
STATEMACHINE	6,74	38,53	0,65	0	3,87	3,87	0,71	0,86
PURETYPE	7,09	11,18	0,18	0	3,91	3,91	4,64	0
DESIGNATOR	9,88	4,34	0	0	0,72	0,72	0,57	0
RECORD	11,50	7,58	0,38	0,07	0,23	0,22	0,26	0,43
POOL	23,38	25,88	0,38	0	0,50	0,50	0,13	1,13
FUNCTIONPOINTER	27,38	0,70	5,92	2,76	5,99	2,58	0,28	1,36
IMPLEMENTOR	32,54	0,46	10,42	4,53	8,77	3,93	0,11	1,80
PSEUDOCLASS	34,50	16,25	1	1	9,75	9	0,75	0
SINK	42,57	12,63	0,85	0,16	7,08	7,08	0,72	0,37
OVERRIDER	56,03	1,10	17,33	5,37	13,19	5,51	0,13	2,30
CANOPY	61,87	8,99	13,28	4,61	15,03	8,27	0,63	1,12
TRAIT	69,09	16	11,31	5,03	16,44	10,06	3,41	1
FUNCTIONOBJECT	77,87	4,05	15,35	6,09	14,14	5,81	0,31	1,42
IMMUTABLE	80,36	54,05	16,93	3,93	12,27	6,57	0,27	0,68
BOX	82,03	9,74	19,54	3,76	16,19	9,71	0,71	1,08
NMP	100,54	23,87	20,42	5,57	19,25	10,26	0,56	0,74
COMMONSTATE	108,46	21,82	19,59	5,56	18,52	10,15	0,54	1,07
COMPOUNDBOX	116,30	43,36	28,12	5,89	22,27	12,28	1,02	1,18
STATELESS	182,61	25,38	33,57	5,73	24,91	15,59	0,86	1,33
EXTENDER	190,55	47,64	56,32	11,17	37,31	16,38	0,73	2,20
COBOLLIKE	229,38	131,60	16,21	4,31	49	42,07	0,09	0,07
OUTLINE	270,34	105,83	57,95	13,97	55,52	31,07	3,02	1,16
RESTRICTEDCREATION	282,75	88,86	33,56	7,94	37,47	24,08	0,67	0,25
SAMPLER	291,92	112,86	67,14	9,25	49,27	27,31	0,49	1,01

and of the CBO. Since LOC is directly related to bugs, this mean that on average Function Pointer classes will be less fault prone than others. Furthemore, the coupling with other classes is lower, in general, than for other micro patterns.

These features are also common to other anti patterns, like Record, Pool, Pseudoclass. Thus, while anti patterns are usually believed to represent bad programming practices, from the point of view of theirs relationship with software metrics well established to be ralated to software quality, they appear not so bad. In fact in general they will be less fault prone and will keep coupling low. Threats to external validity could affect our data. We considered three versions of Eclipse, and this could affect the generality of the discussion. In future work we will consider different systems. The software systems analyzed are Open Source software. Commercial software is developed using different platforms and technologies, with strict deadlines and cost limitation, and by developers with different experiences. This might provide different micro pattern distributions.

13

List of Publications Related to the Thesis

- Concas G., Destefanis G., Marchesi M., Ortu M, Tonelli R., Micro patterns in Agile Software, XP 2013, 14th International Conference on Agile Software Development, June 3rd - 7th, Vienna
- Cocco L., Concas G., Marchesi M., Destefanis G., Agent-Based Modelling and Simulation of the Software Market, Including Open Source Vendors Journal of Information Technology Management
- Destefanis G., Tonelli R., Tempero Ewan, Concas G., Marchesi M., Micro Pattern Fault Proneness, Euromicro SEAA 2012 Cesme Izmir (Turky)
- Concas G., Marchesi M., Destefanis G., Tonelli R. . 2012. An Empirical Study of Software Metrics for Assessing the Phases of an Agile Project. International Journal of Software Engineering and Knowledge Engineering
- Destefanis G., Tonelli R., 2012. Mixing SNA and Classical Software Metrics for Sub-Projects Analysis 11th International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS 12) Cambridge (UK), February 2012
- Destefanis G., Tonelli R., Concas G., Marchesi M., 2012. An Analysis of Micro-Anti-Pattern Effects on Fault Proneness in Large Java Systems SAC 2012, Riva del Garda (Trento)

References

[1] J. Y. Gil, I. Maman. Micro pattern in Java Code, Proceedings of the 20th Object Oriented Programming Systems Languages and Applications, San Diego, CA, USA, pp:97116, 2005. 2, 3, 23, 36, 37, 43, 44, 45, 52, 79, 81

[2] F. Arcelli and S. Maggioni. Metrics-based Detection of Micro pattern to improve the Assesment of Software Quality. Proceedings of 1st Symposium on Emerging Trends in Software Metrics (ETSM 2009), Italy, May 2009. 3, 44, 78, 79, 81

[3] Maggioni, Stefano, and Francesca Arcelli. "Metrics-based detection of micro patterns." Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics. ACM, 2010. 3, 44, 78, 79, 81

[4] E. Gamma, R. Helm, R. Jhonson, J. Vlissides. Design Pattern: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995. 2, 3, 13, 23, 36, 79

[5] Agile Manifesto, URL: www.agilemanifesto.org. 63

[6] Destefanis, G., Tonelli, R., Tempero, E., Concas, G., Marchesi, M. (2012, September). Micro Pattern Fault-Proneness. In Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on (pp. 302-306). IEEE. 64, 67, 68, 69

[7] N.I. Churcher and M.J. Shepperd, Comment on - "A Metric Suite for Object Oriented Design", Correspondence of IEEE Transaction on Software Engineering, 21(3) pp. 263-65, 1995 12

[8] B. Henderson-Sellers, L.L. Constantine and I.M. Graham, "Coupling and cohesion: towards a valid metrics suite for object-oriented analysis and design, Object Oriented Systems, 3(3), pp.143-58, 1996 12

[9] JAPS: Java agile portal system. URL: http://www.japsportal.org. 63, 83

[10] D. Heuzeroth, T. Holl, G. Hogstrom, W. Lowe. Automatic Design Pattern Detection, IWPC 03 Proceedings of the 11th IEEE International Workshop on Program Comprehension 79

[11] L. Aversano, L. Cerulo, and M. Di Penta. Relationship between design pattern defects and crosscutting concern scattering degree: an empirical study IET Softw. October 2009 Volume 3, Issue 5, p.395409 79

[12] S. Dorairaj, J. Noble, and P. Malik, "Understanding Team Dynamics in Distributed Agile Software Development", ;in Proc. XP, 2012, pp.47-61.

[13] J. Bloch. Effective Java Programming Language Guide. Addison-Wesley, June 2011. 44, 51, 67

[14] A. Bachmann and A. Bernstein. Software process data quality and characteristics: a historical view on open and closed source projects. IWPSE-Evol '09 Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops, ACM, 2009. 65, 83

[15] Destefanis, G., Tonelli, R., Concas, G., Marchesi, M. (2012, March). An analysis of anti micro patterns effects on fault proneness in large Java systems. In Proceedings of the 27th Annual ACM Symposium on Applied Computing (pp. 1251-1253). ACM. 64

[16] G. Concas, M. Marchesi, G. Destefanis, R. Tonelli, An empirical study of software metricsfor assessing the phases of an agile project, International Journal of Software Engineeringand Knowledge Engineering, Vol 22, 2012, pp.525-548 80

[17] Process Patterns for Agile Methodologies. Samira Tasharofi, Raman Ramsin 01/2007; In proceeding of: Situational Method Engineering: Fundamentals and Experiences, Proceedings of the IFIP WG 8.1 Working Conference, 12-14 September 2007, Geneva, Switzerland 80

[18] Robert Cecil Martin. 2003. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[19] Empirical studies of agile software development: A systematic review. Tore Dyba, Torgeir Dingsoyr. SINTEF ICT, S.P. Andersensv. 15B, NO-7465 Trondheim, Norway

[20] Rashina Hoda, James Noble, and Stuart Marshall. 2010. How much is just enough?: some documentation patterns on Agile projects. In Proceedings of the 15th European Conference on Pattern Languages of Programs (EuroPLoP '10). ACM, New York, NY, USA, , Article 13 , 13 pages. 64

[21] Agile Processes in Software Engineering and Extreme Programming, Lecture Notes in Business Information Processing Volume 31, 2009, pp 247-248, Software Product Line Engineering Approach for Enhancing Agile Methodologies, Jabier Martinez, Jessica Diaz, Jennifer Perez, Juan Garbajosa

[22] Bugzilla. http://www.bugzilla.org/. 52

[23] Cvs. http://www.nongnu.org/cvs/. 52

[24] R. Tonelli, G. Concas, M. Marchesi, A. Murgia. An analysis of SNA metrics on the Java Qualitas Corpus ISEC 11 Proceedings of the 4th India Software Engineering Conference 52

[25] G. Concas, M. Marchesi, A. Murgia, S. Pinna, R. Tonelli. Assessing traditional and new metrics for object-oriented systems, Proceedings of the WETsOM10, Cape Town, South Africa 52

[26] I. Turnu, G. Concas, M. Marchesi, S. Pinna, R. Tonelli. A modified Yule process to model the evolution of some object-oriented system properties Information Sciences, Volume 181, Issue 4, 15 February 2011, Pages 883902 52

[27] G. Boetticher, T. Menzies and T. Ostrand, PROMISE Repository of empirical software engineering data http://promisedata.org/ repository, West Virginia University, Department of Computer Science, 2007 52

[28] S. Kim, K. Pan, J. Whitehead. Micro Pattern Evolution, MSR 06, May 22-23, 2006, Shanghai, China 55, 79

[29] Mak, Jeffrey KH, Clifford ST Choy, and Daniel PK Lun. "Precise modeling of design patterns in UML." Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on. IEEE, 2004. 35

[30] Campione, Mary, Kathy Walrath, and Alison Huml. The Java tutorial: a short course on the basics. Addison-Wesley Longman Publishing Co., Inc., 2000. 27

[31] Zimmer, Walter. "Relationships between design patterns." Pattern languages of program design 1 (1995): 345-364. 79

[32] Noble, James. "Classifying relationships between object-oriented design patterns." Software Engineering Conference, 1998. Proceedings. 1998 Australian. IEEE, 1998. 79

[33] McAffer, Jeff, Jean-Michel Lemieux, and Chris Aniszczyk. Eclipse rich client platform. Addison-Wesley Professional, 2010. 75

[34] Hall, Tracy, and Norman Fenton. "Implementing effective software metrics programs." Software, IEEE 14.2 (1997): 55-65. 6

[35] Chidamber, Shyam R., and Chris F. Kemerer. "A metrics suite for object oriented design." Software Engineering, IEEE Transactions on 20.6 (1994): 476-493. 8

[36] Bunge, M., Treatise on Basic Philosophy: Ontology II: the World of Systems, Riedel, Boston, MA, 1979 8

[37] Wand, Y., Weber, R., An Ontological Model of an Information System, IEEE Transaction on Software Engineering, 16(11) pp. 1282-92, 1990. 8