
MARS - Modular Architecture Recommendation System

Analysis of System Decompositions through
Coupling and Cohesion metrics

Master's Thesis submitted to the
Faculty of Informatics of the University of Lugano
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics

presented by
Alessio Böckmann

under the supervision of
Prof. Dr. Michele Lanza and Dr. Mircea Lungu

June 2010

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Alessio Böckmann
Lugano, Yesterday June 2010

Every big computing disaster has
come from taking too many ideas
and putting them in one place.

Gordon Bell

Abstract

Software architecture recovery and software re-engineering is of crucial importance for software maintenance and evolution. Through this process, the architect is able to obtain the original architecture from the “as is” system.

Such a recovery task requires automated tool support which aid in extracting data of the system in order to reason on the provided information which may be used for several different purposes (such as reducing the architectural drift given by the absence of documentation or identifying reusable components and isolated modules).

Due to the considerable size of many software systems, the raw metric information that can be extracted from the implemented system quickly becomes unmanageable and therefore such data needs to be represented in a wide variety of ways, each trying to uncover different aspects of the system.

The aim of the study is to devise a recommendation engine which informs the software architect whether some entities belonging to a given package should be moved to a different parent package. Such recommendations are provided on the basis of an assessment of the coupling and cohesion of modules within a system.

We discuss the impact and significance of coupling and cohesion within object-oriented systems and we show different views and interpretations of these metrics.

The direction that we follow is based on the Softwrenaut tool which, through a visualization of software systems, provides an exploratory approach for uncovering their architecture and evolution. We are going to introduce the MARS tool built as part of this thesis, explain various strategies and algorithms that we developed and discuss the results.

Acknowledgments

First of all I would like to thank my supervisor, prof. M. Lanza for his guidance and for the helpful comments that were given on earlier drafts of this dissertation.

I would like also to thank M. Lungu for his valuable help and constant support without which, the tool and subsequent thesis would not have been possible.

Last but not least, many colleagues and close friends, that cannot be extensively acknowledged but whose aid I am very grateful, have also been of great support and I would like to thank them all as well.

Contents

Abstract	v
Acknowledgments	vii
Contents	x
1 Introduction	1
1.1 Outline / Structure of the Document	2
2 Problem Description	3
2.1 Absence of a silver bullet	3
2.2 Common practices for modularizing systems	4
2.2.1 Partitioning according to the architectural style	4
2.2.2 Partitioning through high level logical properties	4
2.3 Evolutionary issues	5
2.4 Architecture Recovery	5
2.5 Software Visualization	7
2.6 Goals & Problem statement	8
3 Softwareaut & MARS	9
3.1 Softwareaut	9
3.1.1 MARS component	10
3.2 Modularizing Object-Oriented systems	11
3.3 Overview of coupling and cohesion	12
3.3.1 Coupling	12
3.3.2 Coupling summary	15
3.3.3 Using coupling as a quality metric	16
3.3.4 Cohesion	18
3.3.5 Using cohesion as a quality metric	20
3.3.6 Coupling-Cohesion correlation	21
3.4 MARS tool	22

4	Modularity recommendations	23
4.1	Types of Recommendation strategies	24
4.1.1	Inheritance strategy	25
4.1.2	Dependency-based strategies	27
4.2	Structure of recommendations	29
4.2.1	High level vs. Low level dependencies	31
4.2.2	Subject mobility ratio	32
4.2.3	Granularity of module expansion	34
4.2.4	Informational metrics	36
4.2.5	Strategies	37
4.3	Scenarios	49
5	Validation	53
5.1	AICup	53
5.1.1	First impression	55
5.2	SimpleSample	58
5.2.1	Decoupling optimization	60
5.3	Considerations on the strategies	63
5.4	Softwareaut	66
5.4.1	Considerations on the results	68
5.5	CodeCity	70
5.6	Improvements	72
6	Conclusions	73
6.1	Summary	73
6.2	Reflections	74
6.3	Future Work	75
	List of Figures	78
	List of Tables	79
	Bibliography	81

Chapter 1

Introduction

Software engineering has characteristics which span both realms of rigid science and artistic creativity. Considering that it is a relatively new science, as all of computer science is, many attempts have been made towards finding a rational, standardized and unified design process.

However, despite the improvements in software design which range from better development tools, enhanced formal verification techniques and more and more refined common practices (such as design patterns or, more generally, architectural styles), systems heavily on the ability and personal point of view of the individual developer, architect or engineer.

What distinguishes therefore software engineering from other areas is this balance between a structured, model-based development, and the intrinsic biased nature of the artifact that is developed. This bias comes from each individual who has worked on creating the artifact and who has put his/her own insights into developing the product. Software systems manifest this equilibrium at many levels, from the development process (e.g. Waterfall and Unified Process vs. XP and SCRUM) to the way the systems are actually composed in terms of design and architecture.

In this work we are concerned with the design of the system. In the context of software reverse and re-engineering, the architectural structure and design of a system are crucial to properly understand, improve and avoid its degradation.

The structure of a system, more often than not, reflects the perceived functionality of the system that the software engineer had in mind during design and development. It rarely represents the true internal structure and therefore its decomposition into modules can be misleading. A major issue therefore arises from this dichotomy between the intended structure of the system and its optimal composition.

In this thesis we present an approach to analyze system architectures through the use of coupling and cohesion metrics. We implemented the tool MARS which exposes such analysis and is integrated in the SoftwareNaut architecture recovery tool. The approach is intended to provide to the recovery architect a “recommendation” engine which proposes a set of move operations based on the amount of dependencies between entities in a (sub)system.

1.1 Outline / Structure of the Document

- In **Chapter 2** we introduce the problem of system decomposition as well as some terminology and pinpoint the aspects that will be discussed further on.
- In **Chapter 3** we present a brief overview of the object-oriented paradigm and introduce the coupling and cohesion metrics that will be at the center of the analysis.
- In **Chapter 4** we describe the recommendations system and the proposed solution as it has been implemented in the MARS tool
- In **Chapter 5** we apply the tool on simple examples and on real software systems and discuss the results as well as analyze the benefits and drawbacks of each recommendation strategy.
- In **Chapter 6** we conclude by summarizing the work and contributions and point out some possible future work or improvements on the subject.

Chapter 2

Problem Description

There are several definitions of architecture such as [3]. For this work we are going to adopt the definition provided by the IEEE 1471 standard ([9]) where architecture is: *“The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.”*

2.1 Absence of a silver bullet

It is a known fact that an incorrect modularization of software systems leads to a fast and “ripple-effect” degradation of the architecture. This is true in other research and development areas but is especially significant in the area of software design.

This constant degradation is also known as architectural aging (initially defined by [19]). The signs of architectural aging include:

Architectural drift increasing separation between the architecture (intended/ documented) and the as-is architecture. It leads to in-adaptability due to the obscurity and non-coherence of the final architecture.

Architectural erosion increasing violations of the architecture which lead to negative effects. Each violation contributes to the brittleness of the system.

Because of this, the “correct” partitioning and classification of functional modules is of utmost importance.

However, as we analyze further on, there is no absolute metric for measuring the correctness of a system decomposition. In fact, from this point of view, correctness is actually unachievable due to the various layers of abstraction that software systems present and that imply different “correct partitions” depending on which level of abstraction is being considered.

2.2 Common practices for modularizing systems

Due to the lack of formal correctness proofs for decomposing software systems, system architectures may follow various ad-hoc approaches (two of the most common will be presented) which depend on the specific system.

2.2.1 Partitioning according to the architectural style

Architectural styles often impose and drive the way that processes are grouped into modules. Simple or straightforward systems which make use of a single architectural pattern often are constrained by the style. For example, a software system which solely uses a Layered architectural style will most probably have a partitioning of modules which reflects the various layers of the system whereas a Pipe & Filter approach will produce modules based on the different stages of the pipeline within the general process.

As we move towards more fine-grained patterns, more constraints will be applied to the partitioning approach (for example the MVC pattern dictates not only the interaction between modules but also the clustering of processes into a specific model, view or controller module).

2.2.2 Partitioning through high level logical properties

For larger systems which make use of more than a single architectural style, the partitioning is not so straightforward, since the different styles may be competing in terms of how to modularize the system.

Therefore, a common practice for dividing the system is based on the high level logical behavior of modules. This partitioning which is clear during development is “imposed” by the architect and it should provide insights on the functionality that each component provides.

According to Parnas the connections between modules are the assumptions which the modules make about each other. In most systems we find that these connections are much more extensive than the calling sequences and control block formats usually shown in system structure descriptions.[18]

As an example, let us consider a process which is used to read from the local filesystem and another process which is responsible of writing to the same filesystem. These two components most probably will not share any data (although they may have several similar structures) since they are responsible of two highly different functionalities. From the abstract logical point of view that is being considered the two components will probably be clustered together into an IO module. However the same system may be partitioned differently, perhaps

into two separate modules (especially if the two steps of input and output to the filesystem are quite consistent).

The above example further clarifies the absence of an absolute measure of correctness when dealing with the structure of modules in a system, since there can be several different partitioning approaches, each (hopefully) based on a different rationale.

2.3 Evolutionary issues

In the context of software evolution, correct modularization is of central importance to properly maintain and adapt the system throughout its lifecycle.

Although a logical partitioning of modules is perhaps the most “human understandable” form of dividing the system into subcomponents (which implies an easier understanding of the functionality of a module), it lacks in providing a measurable quality of the partitioning and, more important, ignores the dependencies between the different modules.

Therefore, despite its intuitiveness, the approach of dividing a software system into subcomponents according to their high level logical functionality leads to an entangled system, where, in the worst case, each module is dependent on all other modules.

We must also consider the issues which may arise in the communication and understanding of the architecture. Because of this, developers might place the components in the wrong place.

For software evolution and maintenance therefore, such an approach implies that the system is doomed to a continuous degradation.

2.4 Architecture Recovery

A key objective in reverse engineering is the recovery of the system architecture [4]. Software Architecture Recovery (also known as Architecture Reconstruction or Reverse Architecting) is based on extracting detailed data from a system and returning a highly informative, condensed representation of the system to aid in its understanding.

The process of reverse engineering and re-engineering of a system shares similar characteristics with forward software engineering. It is therefore based on standards and theoretic approaches but also on the intuition of the engineer. However, the two have key differences in the starting point or input and in objectives that they must achieve. The input to reverse engineering is composed

of the final system and the recovery engineer must traverse backwards in the lifecycle of the system to understand all of the design decisions that have been made during development.

The difference in the initial input between the two engineering approaches, where on one hand we have requirements for forward engineering and systems for reverse engineering, implies that there is a wide difference in objectives between the two. Forward engineering is concerned with properly understanding the requirements and reflecting the solution in the system, whereas reverse engineering is interested in understanding the system and its design.

Understanding the system is however only the first goal and a means to obtain the final objective of upgrading the architecture to avoid/reduce the signs of architectural aging.

The architecture recovery method described by [10] concentrates on creating higher-level views of the architecture through the following “extract-abstract-present” process (depicted in figure 2.1).

Data extraction This phase consists in the retrieval of any kind of relevant information about the software system. Such informations may be given by the source code, documentation, previous versions in the repository, developers or any stakeholder who where working on the project at the time etc.

Knowledge organization This phase consists in organizing and abstracting the information retrieved from the previous phase to a higher level (the approach proposed by Krikhaar is through Relational partition algebra rules used to abstract to higher architectural levels).

Information presentation After having organized the extracted information, the abstracted architecture must be presented in some sort thus requiring a visualization.

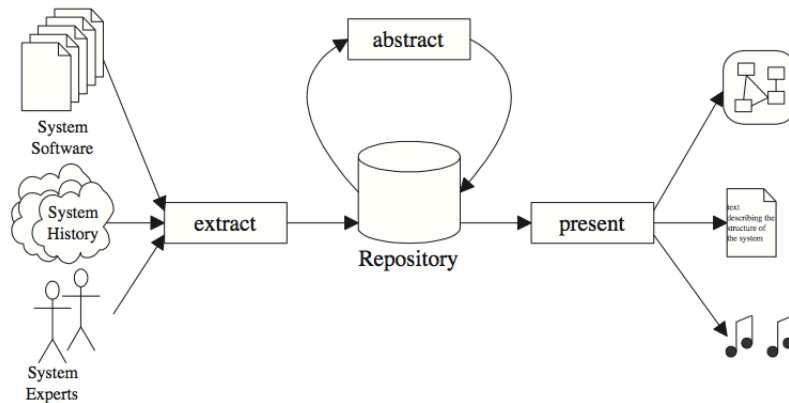


Figure 2.1: Software Architecture Recovery process

2.5 Software Visualization

A key point of the architecture recovery process is the presentation of the gathered information which, given the high number of possible sources of data and the amount of low-level information which can be extracted, can result in being an additional problem on its own. However a proper visualization technique can be used to greatly aid the recovery process by leveraging the intuitiveness and pre-attentive responses of the reverse engineer. In fact visualization provides an ability to comprehend huge amounts of data.[21].

We may distinguish between Scientific Visualizations (concerned with visualizing physically-based objects), and Information visualization related instead to abstract intangible data. Visualization of software systems clearly falls under the second category and provides us a way to simultaneously see, compare and analyze large pools of data.

Software visualization is important also in forward engineering since “the motivation for visualizing software is to reduce the cost of software development and its evolution. Software visualization can support software system evolution by helping stakeholders to understand the software at various levels of abstraction and at different points of the software life cycle”.[1]. In the context of Software evolution however, visualization is of fundamental importance, since as systems evolve, they become more complex and more interconnected, thus requiring more resources to be spent in maintenance. On top of that, since the environment changes, software systems need to be continuously evolving otherwise they become less and less useful in that environment. Because of this, an automated tool support for visualizing the systems is important to assess the architecture.

Classic and more recent state-of-the-art software visualization tools include the program slices approach proposed by Ball and Eick [2], *Rigi* [17] and *CodeCrawler* [11], as well as 3-dimensional-based approaches such as proposed by Gall, Jazayeri and Riva ([8]), *SourceViewer3D* [16] and *CodeCity* [22].

In our work we augment SoftwareNaut [13] by adding a recommendation tool to the existing reverse engineering system.

2.6 Goals & Problem statement

From the architecture recovery point of view we are therefore interested in improving the structural architecture of a system. To achieve this, we need a way to explore and analyze the system and a way to offer and present some recommendations on how to improve the architecture of the system under question. Regarding this last point, as has been described earlier, the approach for providing recommendations should be based on an automated algorithm coupled with the interaction, evaluation and confirmation of these recommendations on behalf of the user.

Summarizing, given a software system as input:

1. obtain direct metrics (related to the source code) regarding the strength and amount of dependencies between components of the system.
2. compute a series of recommendations by comparing the dependencies within a module (cohesion of the module) with the dependencies external of the module (coupling).

To this purpose we present the MARS plugin for the SoftwareNaut reverse engineering tool.

Chapter 3

Softwareonaut & MARS

3.1 Softwareonaut

Softwareonaut is a tool aimed at visual architecture exploration. Its purpose is to aid the reverse engineer in the analysis of a system by providing a top-down exploration of the modules and their dependencies and can scale to large software systems.

The exploration metaphor used in Softwareonaut (depicted in figure 3.1) allows the reverse engineer to proceed in the understanding of the system through an iterative exploration (or expansion) of containing modules to reveal the structure of the subsystem. At each level of expansion, detailed information regarding the entity and its dependencies towards other modules is provided. The tool therefore follows a bottom-up approach to represent the system by gathering various low level metrics and information and presenting them through an exploratory and informative view. The tool scales to large systems and can be used to explore evolving systems as well as software ecosystems (for additional information on the exploration of evolving systems or of the system decompositions see [13], [14] and [15]).

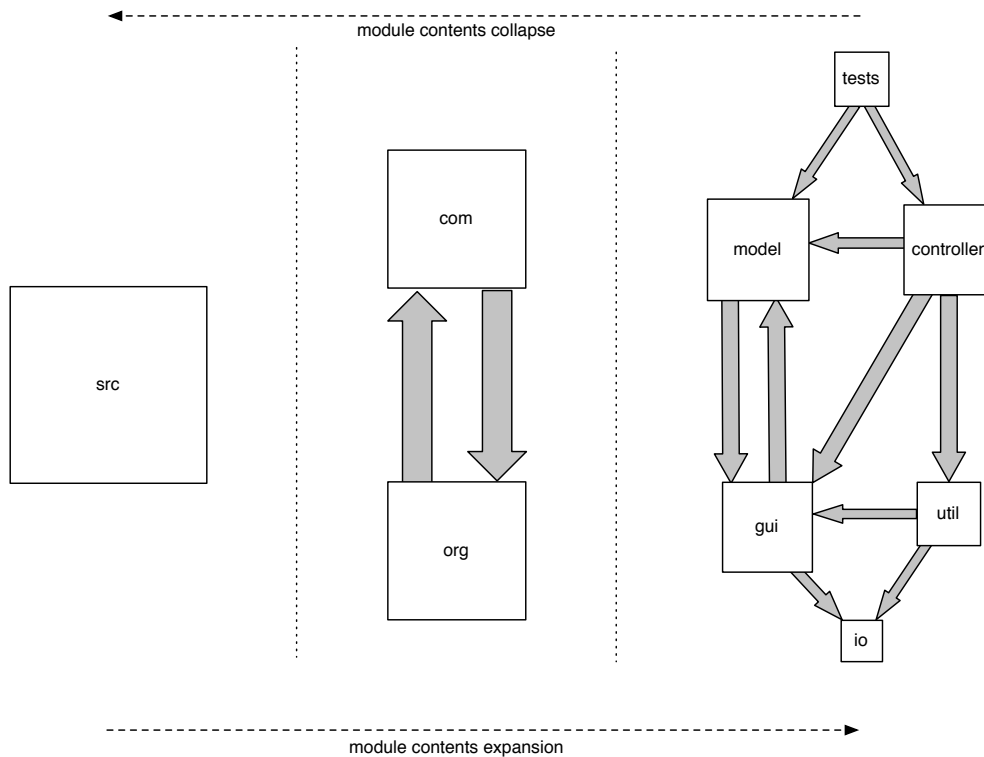


Figure 3.1: Top-down exploration metaphor used in Softwareaut

3.1.1 MARS component

The component that was developed as part of this dissertation was built in order to assist the reverse engineer in the understanding of a system through the exploratory approach offered by Softwareaut.

The system makes use of the same underlying architecture and model used by Softwareaut. It was built with the Smalltalk environment and uses the MOOSE re-engineering environment as well as the FAMIX meta-model to represent and extract direct metrics from software systems (see [5],[6]).

Our work therefore concentrates on providing suggestions which we are going to refer to as *recommendations* to the recovery architect. Such recommendations are based on an analysis of the system from a structural point of view and are mostly based on the number of method invocations across different modules. Through these suggestions we aim at assisting the reverse engineer in discovering the critical modules of the system at hand and possibly improve its architecture by lowering/removing the number of cross-package dependencies.

3.2 Modularizing Object-Oriented systems

Object-Oriented development provides concepts and means for having a more structured design of the system. In the Object Oriented paradigm, the core structure - the *object*, provides the functionality that is specified by its *class* (the definition of the object's behavior). Objects communicate between themselves through message passing.

Objects therefore are intended to characterize a concept or a unit of data that possibly has a concrete realization in the real world. They contain both the data and the processes for manipulating information.

This first level of modularization, known as *encapsulation* or *information hiding* provides a means for which objects can hide certain inner details (both related to their data structures or methods). Information hiding leads to objects which respond only to a specific set of messages which is a subset of all the possible messages and functionality that it provides.

The communication between objects therefore is based on the *interface* that an object reveals to the rest of the system and through which other objects can communicate. In some languages such as Java, interfaces are clearly defined and separated from the rest of the structures of the language (such as classes and methods).

This powerful mechanism, if followed properly, allows systems to be highly structured, in accordance with the principles of *single responsibility* and *separation of concerns*.

These characteristics which distinguishes OO from other programming paradigms requires a way to include genericity that for example other functional or imperative languages provide. This is achieved through the use of *inheritance* and *subtyping* which ultimately leads to *polymorphism*.

Through these mechanisms, the developer is able to create relations between objects in the sense of grouping different classes into a common hierarchy.

3.3 Overview of coupling and cohesion

3.3.1 Coupling

Coupling has been defined for the first time in the realm of procedure-oriented systems by Stevens et al. as “the measure of the strength of association established by a connection of one module to another.”[20].

Eder et al. state that: “Strong coupling complicates a system, since a module is harder to understand, change, or correct by itself if it is highly interrelated by other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules.” [7].

Complexity of systems therefore is reduced by having weak (also known as loose) coupling between components. Highly interrelated system on the other hand manifest strong (or tight) coupling which implies that modules have many mutual dependencies and communicate constantly.

Coupling categories

As we analyzed earlier in section 3.2 methods represent the communication means through which objects interact with each other. Objects therefore, when tied between each other in interaction, establish what is also known as invocation coupling.

The following list (ordered from best/loosest to worst/tightest) contains the classical different known degrees of coupling (summarized also in table 3.1). In object-oriented systems, these various types of coupling can be mapped directly to the general class of *invocation coupling*, with slight differences according to the peculiar characteristics of OO.

1. *Invocation coupling*

Message coupling

This the loosest type of coupling if we don't consider the absence of coupling at all (which for large systems with multiple modules is anyways unachievable). Message coupling consists of dependencies between modules which are based on communication through a public interface. The various modules therefore do not share any of their internal data or processes but instead reveal only the module's global functionality. An example of message coupling is the asynchronous communication between modules through event dispatchers and event handlers commonly used in User Interface toolkits.

Data coupling

A common form of coupling in many systems, data coupling occurs when modules share their internal data and expose it to other modules. This happens when we have parameter passing in messages between modules. The atomic data belonging to a single module is therefore “released” to the public or to the communicating module. This coupling form is considered to be the best when joined with message coupling since only the relevant information is passed as parameter to the object being invoked and not any additional data which may not be useful.

Stamp coupling

Similarly to data coupling, stamp coupling happens when compound data or data structures are shared between modules. It represents a higher degree of coupling not only because the data that is shared is more complex and probably more useful and central to the system functionality, but also because the whole data structure or record is passed between modules even when only a subpart of it is required leading to additional overhead in communication.

Control coupling

Control coupling occurs when the procedural flow of execution of a module is controlled by another module (such as through the parameter passing of a boolean flag). The effect is to create a tight coupling between the two modules, where one is directly dependent on the output and the information provided by the other module. This degree of coupling excludes other tighter forms of coupling (such as external, common or content) since it is solely based on parameter passing between objects. Control coupling, although it's not considered as negatively as other forms of coupling, can lead to a ripple-effect degradation of the system, since changes in a module affect other modules through hidden changes in the control or in the behavior of the controlled module.

The OO paradigm does not prohibit control coupling but, subtyping, dynamic binding and ultimately polymorphism, offer mechanisms to avoid it.

External coupling

This kind of coupling reflects high and tight dependencies between modules which share data which is not owned or internal to any of the modules. The data is externally imposed and constrain the way the modules can be constructed. Examples of external coupling are data formats, communication protocols and pre existing interfaces. If

we consider the method level as a module, external coupling is given by the use of instance variables which do not represent the state of the object or of the class. In case of classes and packages, external coupling is given by the use of global data and variables.

Common coupling

Similarly to external coupling, common coupling is based on shared data external to the communicating modules. In this case however, the shared data is usually unstructured and is defined by a module of the system (such as a global variable), and therefore modification of such data implies that all modules dependent on it change functionality. Common coupling is forbidden by the OO paradigm since there are no constructs for creating a globally shared space where objects and classes connect to.

Content coupling

The highest level of coupling occurs when the encapsulation within a module is ignored and other modules directly access (and possibly modify) the internal functionality or data of the module. This implies that the other modules must have knowledge of the internal details of the package with the consequence that each module is highly dependent on the others.

The object-oriented paradigm limits, through encapsulation and information hiding, the amount of direct access to the implementation of a class or method. However, a low use of information hiding (public visibility of the contents of a module), or, for example the use of the `friend` modifier in C++ which allows to override the private or protected data of a class, can easily lead to content coupling.

Classes and packages, being the container components which methods belong to, can also manifest the different degrees of interaction coupling that have been described. On top of that, classes expose other forms of coupling due to their more structural nature (as opposed to the behavioral nature of methods) and to the additional OO properties which must be considered.

Considering this last aspect we can define an additional type of coupling based on the inheritance relationships between classes.

2. *Inheritance coupling*

Can be clearly applied only to classes but the coupling between the class modules propagates through the abstraction layers showing itself also at the package level (this occurs when the module containing a child class is different than the parent's).

Inheritance coupling is of high importance because of the improvement possibilities that it offers in terms of refactoring and generalization but also due to the negative impact on reusability for hierarchies split among different packages. The effects of inheritance coupling may reveal themselves in cases where we have large inheritance hierarchies and where a super type definition/signature is modified. Such changes affect the whole underlying hierarchy which means that the degree of coupling is directly related to the extent of the inheritance hierarchy.

3.3.2 Coupling summary

Table 3.1 summarizes the different levels of coupling that we just described.

Coupling type	Degree	Description
inheritance	-	subtype relationship between class modules
message	lowest	message passing/interface communication
data	medium	shared data
stamp	medium	shared data structures
control	high	control of a module is yielded to another module
external	high	common protocols/interfaces/data format
common	high	shared global data
content	highest	full access to the internal details of a module

Table 3.1: Summary table of different coupling categories

3.3.3 Using coupling as a quality metric

As described earlier in section 3.2, classes in object-oriented systems already provide a modularization of the functionality. We therefore have different levels of abstraction for what concerns the modules that are being analyzed (typically classes and packages).

When the module is a class, the concept of coupling represents the number of method invocations which are external to the class. In the model that is being proposed, the class coupling refers instead only to the messages sent to classes belonging to other packages (any outgoing messages within the same package are instead treated as cohesive links, as will be described in 3.3.4).

Coupling provides a metric which can be used to describe and analyze software systems and is directly mapped to the quality of the system. It can also be used to analyze the lifetime of a system and to evaluate its maintainability. In order to loosen the coupling of a system, two main approaches can be followed:

1. Reduce the coupling by refining and modify the code in order to lower the degree of coupling. For example, such an approach would be based on removing control coupling by making heavier use of polymorphism thus obtaining the lower degree stamp coupling. After a further iteration the stamp coupling would be reduced to message coupling. Such an approach however is not applicable in the context of architecture recovery since it requires a deep knowledge of the system and can be hardly automated.
2. The approach that we are considering instead is based on the extent and amount of coupling that modules have. From this point of view, a method invocation external to the package represents a dependency between the two modules which increases the coupling between the two classes/packages.

Coupling can be applied at any layer of grouping, such as at the class or package level. We can thus define the coupling of a class or package as the number of dependencies towards other packages in the system.

More precisely:

external class coupling the number of dependencies which the class has towards classes contained in other different packages.

$$coup_i = \frac{k_i}{n_i} \quad (3.1)$$

where k is the number of inter package dependencies relative to class k_i and n is the total number of dependencies.

package coupling the coupling of the package represents the total number of dependencies towards other packages.

$$coup_j = \sum_{class}^{C_j} coup_{class} \quad (3.2)$$

where C_j is the total number of classes contained in the package.

Coupling therefore is used as a quantitative metric and encompasses all categories that have been described earlier and that are grouped as invocation coupling with the exception of the inheritance coupling which is treated differently (further described in 4.1).

We thus define two categories of coupling (table 3.2): Invocation and inheritance coupling where one reflects the number of dependencies that there are between different modules and the other the number of subtype relations between classes.

Coupling category	Metric used	Module level
Inheritance	Definition location (top of inheritance tree)	class, package
Invocation	quantitative (invocation counts)	method class, package

Table 3.2: Coupling categories associated with their main metric and abstraction level

As has been described, due to the nature of OO systems, there are various ways of interpreting a module based on the abstraction level. We limit our search to the class and package levels since the method level is too fine-grained when dealing with architecture recovery. Nonetheless it is worth mentioning that, as equation (3.2) represents the summation of the class coupling obtained by (3.1), this itself is composed by the coupling of the methods, therefore the coupling of a class i can also be expressed as:

$$coup_i = \sum_{method}^{M_i} coup_{method} = \sum_{method}^{M_j} \frac{k_{method}}{n_{method}} \quad (3.3)$$

where M_i is the total number of methods contained in class i , k_{method} is the number of method invocations external to the package and n_{method} is the total number of invocations.

3.3.4 Cohesion

The concept of cohesion was also defined by [20] and is highly correlated with coupling. Cohesion provides a measure of the focus of a module in the sense of how strongly related are the components and responsibilities within the module.

Quality systems present a high cohesion which implies that the modules within the systems are robust and maintainable as well as reusable. On the other hand, low cohesive systems have modules that are not single-focused therefore highly depending on others.

Cohesion categories

Similarly to coupling, there are different categories for cohesion (summarized in table 3.3):

Functional cohesion

The highest and strongest level of cohesion is given by functional cohesion in which all and only the module parts which contribute to the single functionality of the module are grouped together. In case that there is no coupling with other modules, functional cohesion allows the packages to be treated completely independently and can be modified or removed without the risk of affecting other modules or the whole system.

Sequential cohesion

This cohesion type can be found especially in systems which follow a Pipes & Filter architecture or where a specific functionality of a module part is directly related to the output of another module part. The cohesion therefore is based on the sequential steps that are necessary to produce the desired functionality (for example the preprocessing, lexical analysis and syntax analysis of source code in a compiler process).

Communicational cohesion

Communicational cohesion is directly linked to the shared data of a module and to the communication that module parts share between themselves. Module parts are therefore grouped based on the amount of interaction that they share.

Procedural/Temporal cohesion

Procedural and temporal cohesion are similar in the sense that in both cases, module parts are grouped according to the specific point in time or execution when they are called. For example, we have procedural cohesion when a part of a module (which is responsible of checking file permissions) is grouped together with another part which handles the actual opening of a file in case there are sufficient privileges.

Logical cohesion

Logical cohesion is the most common kind of cohesion and has been described earlier in 2.2.2. Module parts are grouped therefore according to their logical functionality (their perceived functionality) even if the parts are very different in nature.

Coincidental cohesion

From its intuitive name, coincidental cohesion represents the grouping of functionality into modules without any rationale therefore through an arbitrary or random decision. An example of coincidental cohesion is a module which contains commonly used “utility” functions, which may or may not have some degree of cohesion.

Cohesion summary

Table 3.3 summarizes the cohesion categories that we just described.

Cohesion type	Degree	Description
coincidental	lowest	random grouping into modules
logical	low	grouping based on a logical categorization
procedural	medium	grouping based on execution time of module parts
communicational	high	module parts all operate on the same shared data
sequential	high	parts follow a well-defined sequence of execution
functional	highest	parts all contribute to a single responsibility

Table 3.3: Summary table of different cohesion categories

3.3.5 Using cohesion as a quality metric

Given our definition of coupling we can already see that the dependencies that are not to be considered as coupling fall under the category of cohesion and vice-versa. We can therefore define cohesion in a similar way as:

external class cohesion the strength or amount of dependencies which the class has towards classes within its same module.

$$coh_i = \frac{k_i}{n_i} \quad (3.4)$$

where k is the number of intra package dependencies and n is the total number of dependencies.

package cohesion represents the total number of dependencies and communication links between the classes contained in the module.

$$coh_j = \sum_{class}^{C_j} coh_{class} \quad (3.5)$$

where C_j is the total number of classes contained in the package

The cohesion categories can therefore be defined in the same way as for coupling (see table 3.4).

Cohesion category	Metric used	Module level
Inheritance	Definition location (top of inheritance tree)	class, package
Invocation	quantitative (invocation counts)	method class, package

Table 3.4: Cohesion categories associated with their main metric and abstraction level

3.3.6 Coupling-Cohesion correlation

The high correspondence between coupling and cohesion is given by the fact that both positive and negative shifts on the quality scale of one measure, tend to inversely affect in the same way the other measure.

Figure 3.2 depicts this correspondence. We can see that systems can have a high external coupling and low cohesion but also high cohesion. These are the candidates which will be considered later when providing recommendations. The objective is to minimize the external coupling of an entity in order to push it as far as possible under the curve which represents the ratio between coupling and cohesion.

Note that the linearity between coupling and cohesion in systems is a simplification used for explaining the relation between these two metrics and that the actual curve may be different. The choice of whether to provide a move recommendation of an entity depends on the point of the curve that the system falls on.

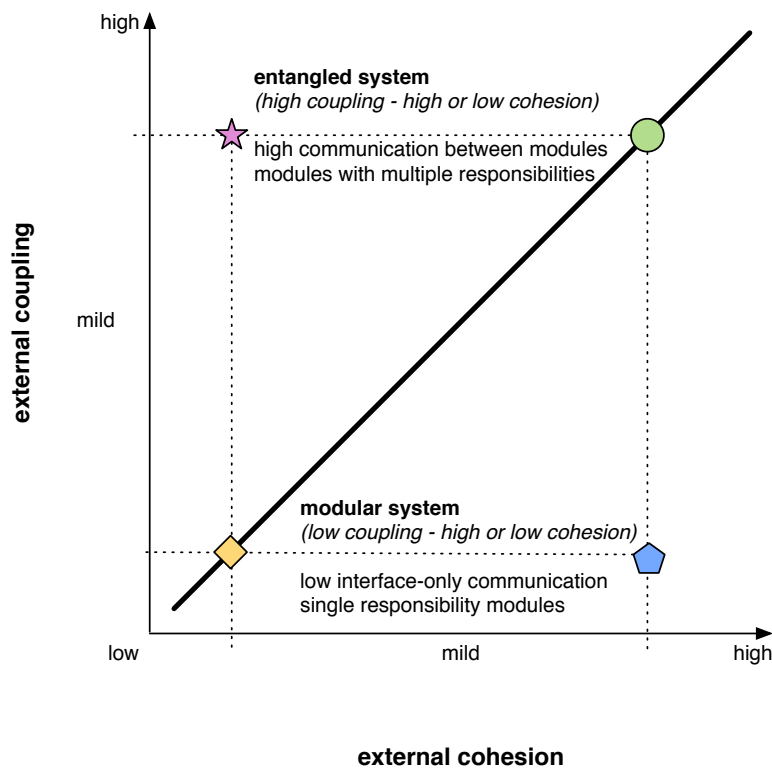


Figure 3.2: Correlation between coupling and cohesion

3.4 MARS tool

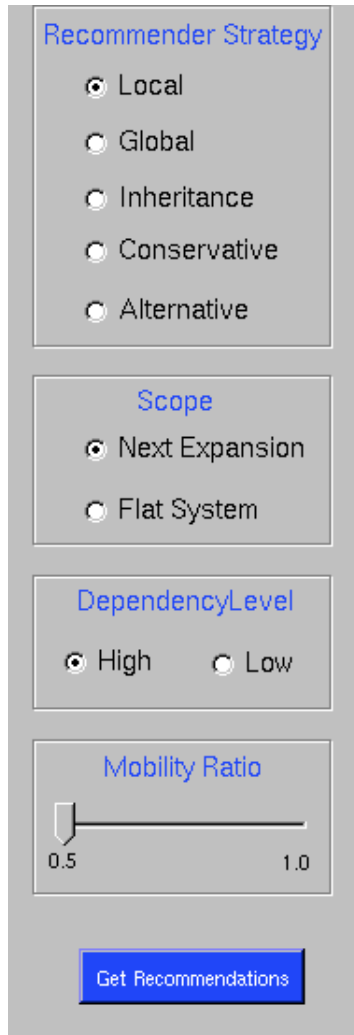


Figure 3.3: MARS tool options & parameters sidebar

The MARS tool is integrated into Softwareaut and presents a new window through the toolbar button shown in figure ?? on each system model change.

Figure 3.3 depicts the sidebar of the MARS window where the recovery architect user can set various parameters to restrict or widen the search space.

A button, which is not depicted in the figure, allows the user to export the recommendation results in csv format (comma-separated entries for importing in a spreadsheet application). Another “shortcut” button is provided for running all of the strategies on the fully expanded system (i.e. flat system scope). The qualitative results (i.e. the subject, the source and target packages) are grouped as before into an exportable csv file.

Once the desired scope granularity, dependency level and strategy have been selected, the resulting recommendations are displayed in a side view in a tabular form.

We will now present the approach for providing recommendations and describe in detail each of the specific parameters that we just mentioned.



Figure 3.4: Softwareaut toolbar

Chapter 4

Modularity recommendations

As has just been described and depicted in figure 4.1 we base our analysis from the coupling perspective in the sense that we aim to reduce it when it is higher than the cohesion. Of course, given our definition, the complementary approach can also be taken by considering the cohesion and providing recommendations in case it is lower than the external dependencies of the entity.

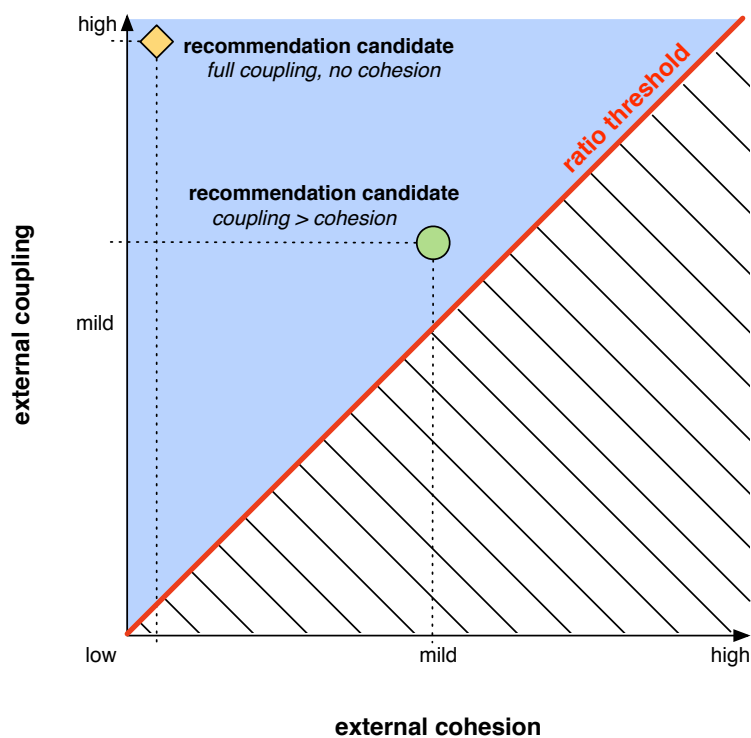


Figure 4.1: Recommendation threshold based on coupling/cohesion ratio

Recommendations therefore are the main structure of the MARS tool and contain the information relative to the state of the system. This corresponds to the current exploration view of the system decomposition which is provided by Softwareaut. Recommendations also hold the predicted improvement that would arise from performing the recommended move.

In order to provide the recommendations, the system must undergo a further expansion of its entities (which may be packages, classes, methods) in order to extract the detailed internal state of the modules. Each child entity is then processed as possible candidate for becoming a recommendation result based on the specific setting parameters (described later in section 4.2) and according to the main recommendation strategy which is being applied.

4.1 Types of Recommendation strategies

We propose two general strategies for producing recommendations. These reflect the two coupling/cohesion categories (inheritance and invocation) that have been described earlier. We thus distinguish between a structure-based strategy which perform an analysis of the inheritance hierarchies and dependency-based strategies which are responsible of analyzing invocation dependencies between modules.

4.1.1 Inheritance strategy

The recommendations are based on an analysis of the structural properties of the system. One of these properties are, in the case of object-oriented systems, the inheritance and subtype hierarchy of the classes. Classes hold both behavior as well as data (and data structures), therefore analyzing the inheritance hierarchy allows to assess the stamp and data coupling of the system (as well as the logical cohesion which is given by the structure of the hierarchy).

Several strategies for providing recommendations can be applied. These strategies all have the common goal of reducing the coupling and increasing cohesion at the package level. The direction that we chose as strategy is based on grouping classes which extend a common superclass into the location of the declaration of that superclass. This choice is due to the fact that the static declaration of classes and their relationship to other classes is clearly a structural design choice made by the developer. As such, the location of a class which has subclasses is taken as the reference target of all subclasses since it represents the global definition of the interface.

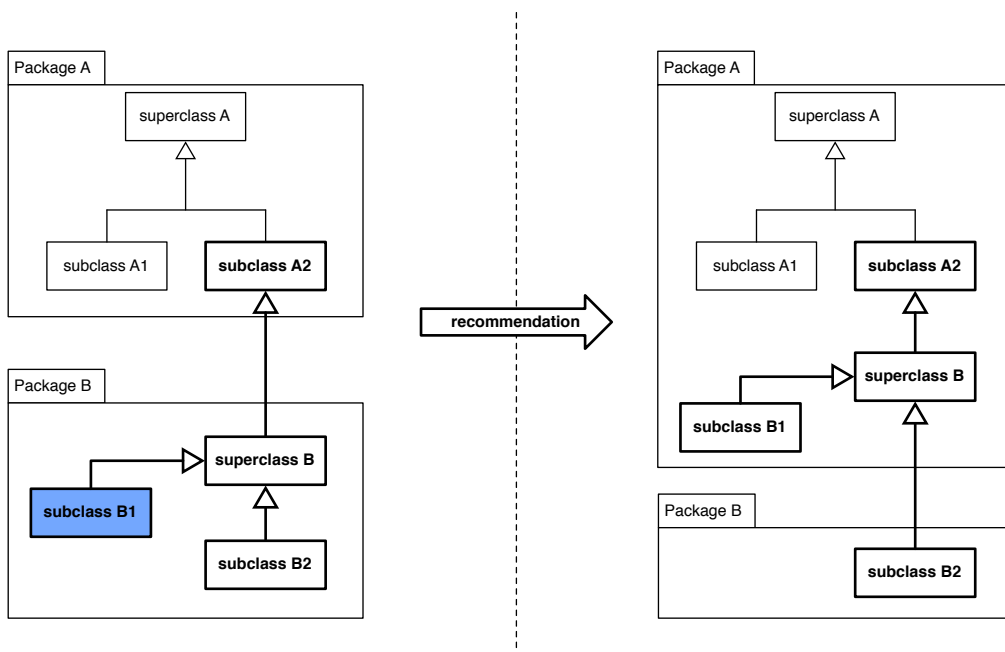


Figure 4.2: Example situation for a recommendation given by the inheritance analyzer

The process can be found in Algorithm 1 and Algorithm 2. The output consists of recommending a move of any subclass to the package of the root superclass in case the two packages are different. This has the effect of grouping all inheritance hierarchies into a single module. Figure 4.2 depicts the strategy that is followed by the recommendation system and that has just been described.

Algorithm 1 inheritanceRecommender(Model M)

```

recommendations ← new List.
for each package module  $P_i$  in  $M$  do
  for each contained class  $C_j$  do
    recommendations add(recursiveSuperClass( $C_j$ ))
  end for
end for
return recommendations

```

Algorithm 2 recursiveSuperClass(Class C_j)

```

if  $C_j$  hasSuperclass then
  if superClass( $C_j$ ) isInTheSubSystem then
     $C_j$  ← superClass( $C_j$ )
    recursiveSuperClass( $C_j$ )
  end if
end if
return recommendations

```

4.1.2 Dependency-based strategies

The general recommendation strategy for the dependency based analysis is based on obtaining the external coupling and cohesion for each entity (class) that is being analyzed. This implies that each entity that is being analyzed, undergoes a hidden expansion if the modules in order to gather metrics' information. Different strategies for obtaining and representing the coupling and cohesion metrics will be discussed further on but for now we can distinguish two cases where we can apply recommendations:

Entity with higher coupling than cohesion (coupling > cohesion)

We have seen that coupling and cohesion are complementary metrics therefore a recommendation strategy that would lower coupling would be equivalent to one that increases cohesion (assuming that both metrics are represented in the same manner).

The direction that will be considered from here on points at reducing the coupling of a system, therefore entities which are susceptible to recommendations are those with a higher number of dependencies towards other packages than towards its own package (an example can be found in figure 4.3).

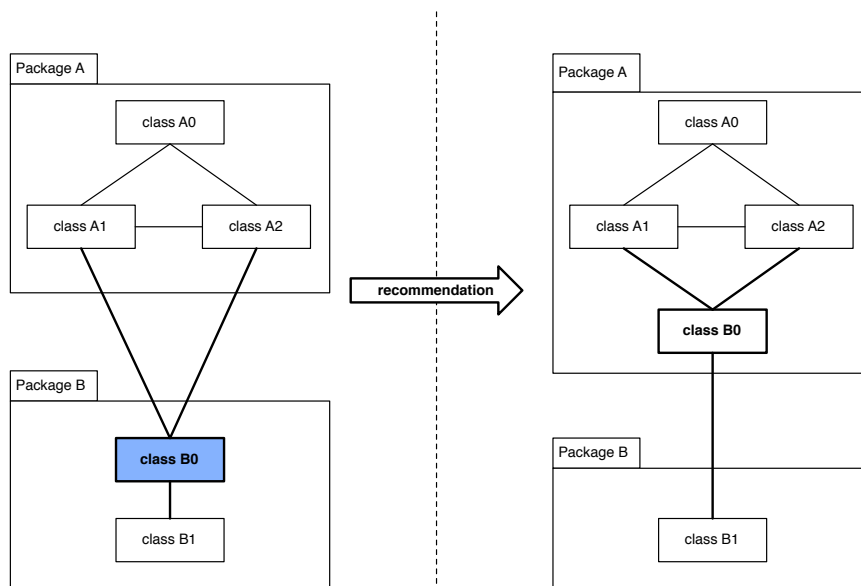


Figure 4.3: Example situation for a recommendation given by the dependency analyzer (in case the coupling > cohesion)

Entity with no cohesion (cohesion = 0)

From this point of view we can also consider entities with no cohesion but with some degree of coupling (see figure 4.4). This specific case of the generalized approach described earlier is noteworthy since there is little need to perform an in-depth analysis because the entity has no measurable reason to be contained where it is. Because of this, any movement towards one of its coupled neighbors would lower the coupling of the system.

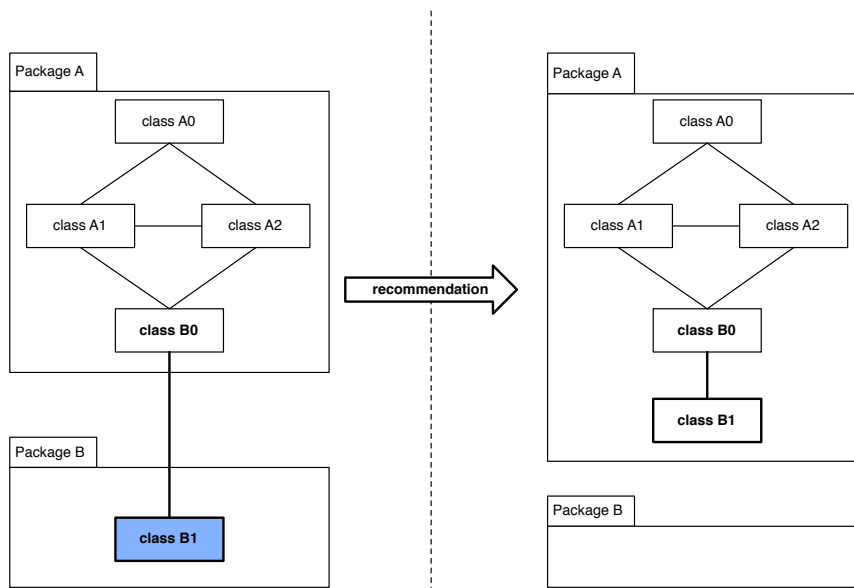


Figure 4.4: Example situation for a recommendation given by the dependency analyzer (in case the cohesion = 0)

Other situations or states that the system may show do not require recommendations since the ratio between coupling and cohesion is sufficient (meaning that entities have more cohesion than coupling) and cannot be improved. More specifically we can distinguish the following sub-states:

Entity with loose coupling and strong cohesion (coupling < cohesion)

Balanced entity (coupling = cohesion)

Specific situations of this general state include the cases when we have a balanced class, therefore when the two metrics are equal. We take a conservative approach on this situation since a move recommendation would not bring any improvement.

Orphan class (coupling = cohesion = 0)

Another case when recommendations are not provided is when there isn't enough information relative to the metrics such as when there are orphan classes with no dependencies towards any entity.

Entity with no coupling (coupling = 0)

As for the previous case we do not have enough information regarding the possible targets of a recommendation since there are no dependencies towards other modules. This situation represents the optimal desired situation where a module component contributes only to its module functionality.

Now that the condition for providing a recommendation has been defined we can distinguish various strategies for calculating the coupling and cohesion metrics.

4.2 Structure of recommendations

The output recommendations are object structures which contain the following fields:

subject	an entity which is susceptible to a recommendation
from	a package (which contains the subject of the move)
to	a target package
strategy	the strategy used by the recommender
dependencyLevel	<i>high-level</i> (a dependency edge implies an increase of 1 in the coupling) or <i>low-level</i> dependencies (the coupling is the sum of all invocation edges)
scope	<i>next expansion</i> (contents of the viewed system) or <i>flat system</i> (full expansion of the packages)
rank	$\text{couplingImprovement} \cdot \text{subjectMobility}$
subjectMobility	a number (the coupling of the subject over its total number of dependencies).
systemCoupling	number representing the total coupling of a system (relative to the strategy being used)
couplingImprovement	a number (the gain in terms of decreased coupling that a move would produce)
description	textual description of the reason for the move

The main fields are the `subject`, `from`, and `to` which define the recommendation. Recommendations are given based on a particular strategy which is applied to the system under analysis.

Recommendations contain parameters which can be modified through the graphical user interface to restrict or widen the search space. These are the `subjectMobility`, `dependencyLevel` and `scope` which we will describe in detail in the following section.

Other fields provide information such as `couplingImprovement` and `systemCoupling` which are numerical metrics which provide information on the improvement of the recommended move and on the full (sub)system respectively. The rank is represented as the product between the mobility ratio and the coupling improvement.

4.2.1 High level vs. Low level dependencies

The first difference which distinguishes the recommendation strategies is the *dependency level*. We refer to it as high or low based on the following definitions:

High level Coupling and cohesion are based on the sheer number of dependencies between classes. Only the high level dependencies between modules are considered therefore only the existence or absence of connections between packages.

Low level The low level view instead counts the true number of dependencies between modules. In this case the coupling and cohesion of a module is given by the cardinality of its dependencies (therefore by the cumulative number of method invocations). This view is different from the previous since it counts the weight of dependencies between modules instead than the existence of dependency links between modules..

Referring to figure 4.5, the number of high level dependencies is 3, whereas the low level dependencies based on number of method invocations amounts to 40. We leverage these different views of the metrics in the scenarios discussed in 4.2.

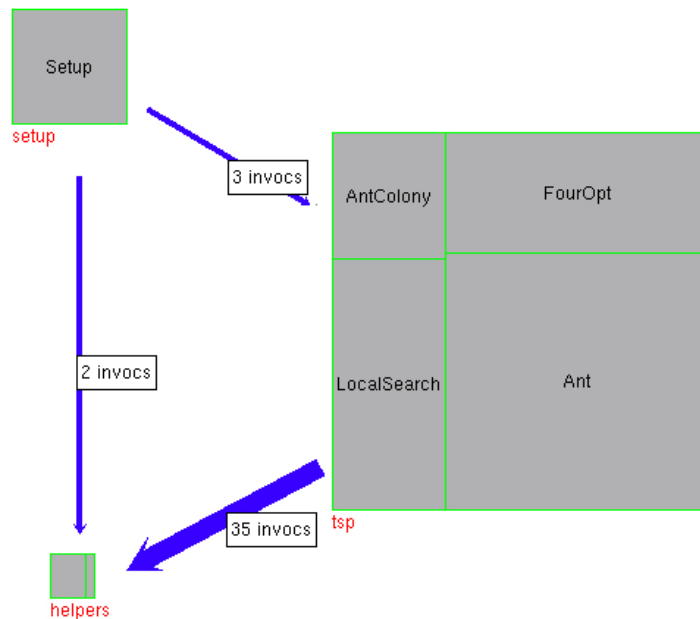


Figure 4.5: Example system for the dependency level

4.2.2 Subject mobility ratio

Referring to the initial description on how to use coupling as a quality metric (see equation 3.1), the mobility of a class represents an important parameter which can be used as threshold for the recommendation solution.

First of all, such a metric represents the balance between coupling and cohesion, and since we are considering the problem from the coupling point of view, this ratio must be > 0.5 in order to have any improvements. Anything less would imply that the entity has more dependencies within its module than towards the external modules.

The mobility ratio can be therefore seen as the coupling “over”/divided by the coupling and cohesion. The ratio is thus equal to 1 when the cohesion is zero, which means that the entity has no absolutely no reason of being where it is (from the structure point of view).

The mobility of an entity is therefore enclosed between 0 and 1 where, when such a value is ≤ 0.5 we have a stable (more cohesed than coupled) entity whereas any values > 0.5 represent a high mobility of the subject which therefore becomes a candidate entity to be recommended (see figure 4.6).

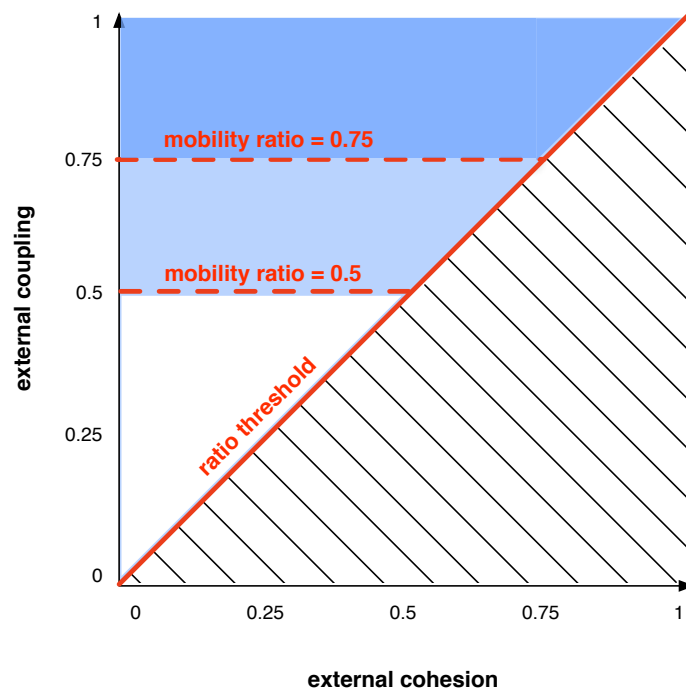


Figure 4.6: Recommendation threshold based on the subject mobility ratio

Such an approach can be used to perform an initial overview of the state of the system by setting the mobility ratio to 1 which would result in recommendations with a high confidence on which entities are wrongly placed and should be moved.

4.2.3 Granularity of module expansion

This setting (which we also referred to as “scope”) allows the recommendation tool to expand the system under analysis down to the individual class level. Given for example a system modularized in nested packages (such as in the example in figure 4.7) the default behavior which we assumed up to now (i.e. performing a hidden expansion of all of the modules to extract the metrics) corresponds to the **next expansion** setting. The alternative option is the **flat system** which performs the hidden expansion recursively down to the class level.

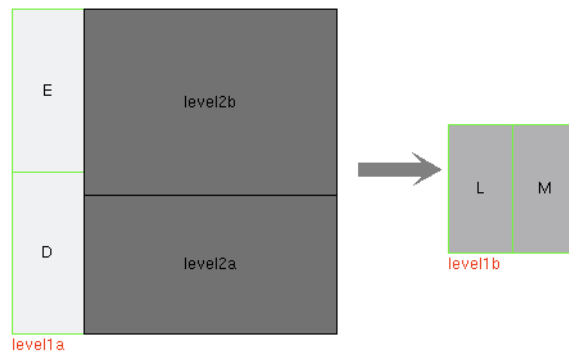


Figure 4.7: Current view of the explanatory system for the scope parameter

1. **next expansion:**

This is the default setting for the recommendation tool and corresponds to obtaining the recommendations for the current exploration view given by Softwareonaut. This option allows the system to be inspected in a step-wise manner by expanding modules or modifying the system through the exploration and metric filters which are provided by Softwareonaut. The returned results therefore contain as recommendation subjects the entities which are contained in the current modules being explored. Figure 4.8 depicts how the system viewed in figure 4.7 is seen by the tool

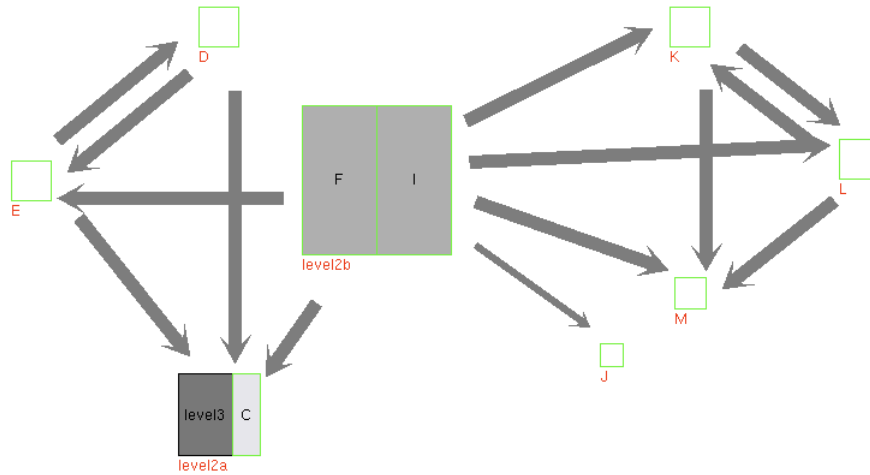


Figure 4.8: System as viewed by the next expansion scope

2. flat system:

This option which naturally returns more recommendation results is based on recursively expanding the contents of the packages which are being explored on the current SoftwareNaut view. It therefore corresponds to a full expansion of the modules down to the class level and returns recommendations on those subjects. Referring to the example system, figure 4.9 depicts how it is seen by the tool in the flat system expansion. Packages level2band level2a (which itself contains package level3) are expanded down to revealing the classes F,G,H,I and A,B,C.

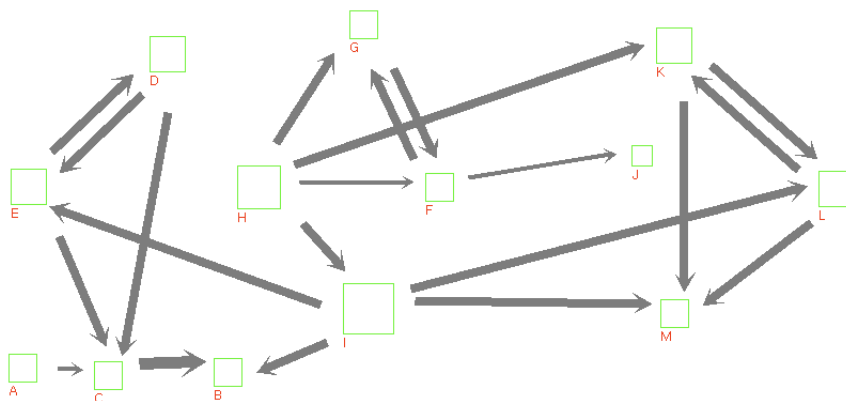


Figure 4.9: System as viewed by the flat system expansion scope

4.2.4 Informational metrics

System and improvement coupling

Additional information which is provided by the recommendation system contains the amount of coupling of the whole system, as well as the discount coupling / gained cohesion which would follow from performing the recommended move. These two metrics are provided only with invocation-based strategies and provide two measurable values for comparing recommendations.

For each dependency-based strategy, the improvement which is given by a recommendation represents the number of dependencies that would be eliminated by performing the recommended move operation. The number of dependencies is relative to the strategy that is being used (therefore either high-level or low-level dependencies).

All the dependency strategies compute the improvement given by moving an entity j to the most coupled package p_{max} as follows:

$$improvement_j = d_{t+1} - d_t$$

Where d_{t+1} is the amount of inter package dependencies there would be after performing the recommended move and d_t is the value representing the current number of dependencies.

The value of d_{t+1} which again represents the coupling of the entity after performing the move is given by:

$$d_{t+1} = d_t - (d \rightarrow p_{max})_t + i_t$$

where $(d \rightarrow p_{max})_t$ is the amount of dependencies towards the most coupled package. Thus $d_t - (d \rightarrow p_{max})_t$ represents the remaining coupling/number of external dependencies there would be after the move and i_t is the cohesion/amount of package-internal dependencies of the entity. We need to consider also this since performing atomically the recommended move would imply that all of the internal dependencies would become external to the package after the move.

rank

The rank represents the product between the improvement of the recommended move and the mobility ratio of the subject. This value is thus directly proportional to both the gain in terms of coupling and to the mobility ratio (which is highly influenced by the cohesion).

4.2.5 Strategies

As described earlier, strategies can be applied at the high or low dependency level. As such, each of the following strategies will have a high and a low level version.

Local strategy

This corresponds to a greedy strategy which restricts the scope of the search to the package the entity under analysis is most coupled with. We therefore analyze a subsystem composed of two packages and the dependencies between them. We thus refer to it as a **Local** strategy.

The process and behavior are defined in algorithm 3 and depicted in figure 4.10 for the high level local strategy, and in algorithm 4 and figure 4.11 for the low level local strategy, respectively.

Algorithm 3 dependencyRecommender - High Level Local Strategy

```

for each package module  $P_i$  do
  for each contained class  $C_j$  do
     $intraDepList \leftarrow \text{getInternalDependencies}(C_j, \text{highlevel})$ 
     $NP_j \leftarrow \text{getMostCoupledPackage}(C_j, \text{highlevel})$ 
     $interDepList \leftarrow \text{getExternalDependencies}(C_j, NP_j, \text{highlevel})$ 

     $coupling_j \leftarrow \text{length}(interDepList)$ 
     $cohesion_j \leftarrow \text{length}(intraDepList)$ 

    if  $coupling_j > cohesion_j$  then
       $subject \leftarrow C_j$ 
       $from \leftarrow P_i$ 
       $to \leftarrow NP_j$ 
      add new Recommendation( $subject, from, to$ ) to  $recommendations$ 
    end if
  end for
end for
return  $recommendations$ 

```

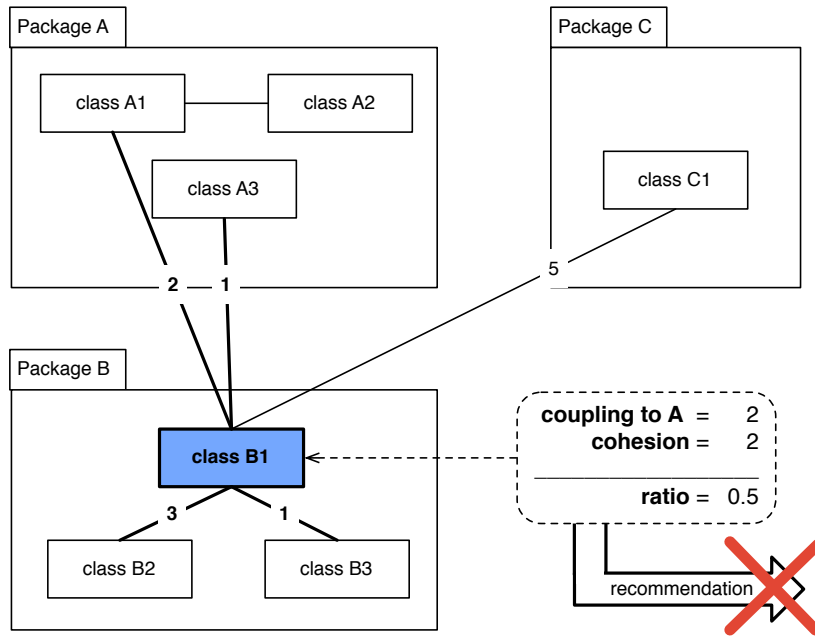


Figure 4.10: Example state as seen by the High-level Local Strategy

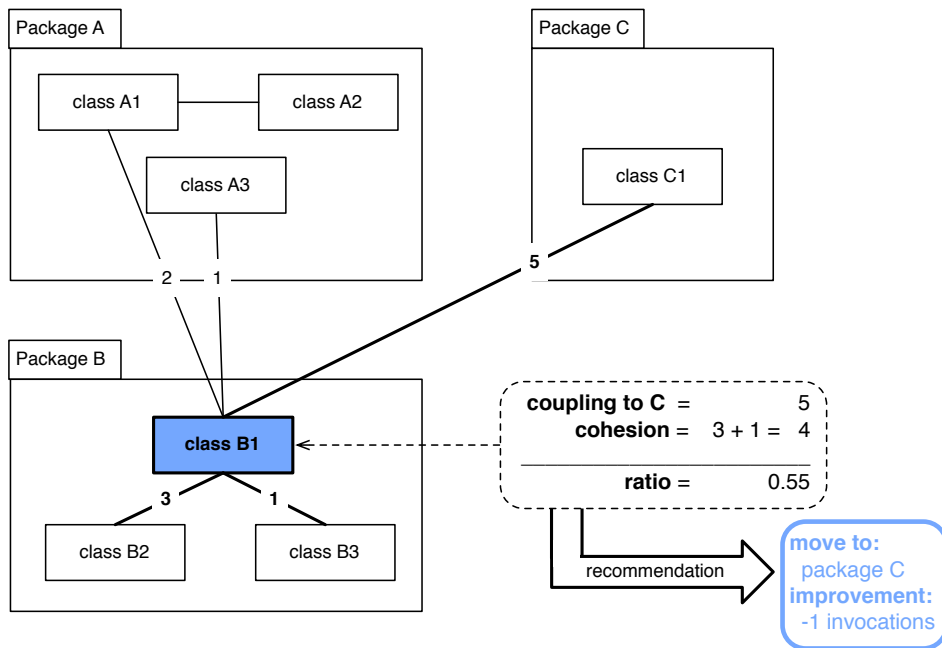


Figure 4.11: Example state as seen by the Low-level Local Strategy

Algorithm 4 dependencyRecommender - Low Level Local Strategy

```

for each package module  $P_i$  do
  for each contained class  $C_j$  do
     $intraDepList \leftarrow \text{getInternalDependencies}(C_j, \text{lowlevel})$ 
     $NP_j \leftarrow \text{getMostCoupledPackage}(C_j, \text{lowlevel})$ 
     $interDepList \leftarrow \text{getExternalDependencies}(C_j, NP_j, \text{lowlevel})$ 

    for each dependency  $d_k$  of  $interDepList$  do
       $\{d_k \text{ is the sum of the number of invocations}\}$ 
       $coupling_j \leftarrow coupling_j + d_k$ 
    end for
    for each dependency  $d_k$  of  $intraDepList$  do
       $cohesion_j \leftarrow cohesion_j + d_k$ 
    end for

    if  $coupling_j > cohesion_j$  then
       $subject \leftarrow C_j$ 
       $from \leftarrow P_i$ 
       $to \leftarrow NP_j$ 
      add new Recommendation( $subject, from, to$ ) to  $recommendations$ 
    end if
  end for
end for

```

The assumption here is that any external coupling not directly related to the subsystem which we are analyzing will remain the same after the recommended move operation. We are therefore able to reduce the external coupling of the whole system if we lower the coupling of the subsystems which compose it.

With respect to the high level view, since it looks only at the existence of dependency links, this strategy tries to remove the dependencies between modules without making distinction on their weight. This means that if we have an entity depending on two modules and another entity heavily relying on another module (as in figure 4.10), the first entity will be chosen as recommendation candidate since it has a higher degree of coupling than the second. We have instead the opposite situation if we consider the low level dependencies as in figure 4.11.

Global strategy

The global strategy, contrasted with other strategies, considers the full number of dependencies of a class, both internal and external to the package, as coupling and cohesion respectively. This means that, as can be seen in figures 4.12 and 4.13, because a move can be done to one and only one package, the external coupling is overestimated by considering all the external couplings of an entity. This strategy however has been implemented (refer to algorithms 5 for the high-level global strategy and 6 for the low-level global strategy) to provide an upper bound on the number of returned results in order to get an overview of all the potential candidates for move operations.

Algorithm 5 dependencyRecommender - High Level Global Strategy

```

for each package module  $P_i$  do
  for each contained class  $C_j$  do
     $intraDepList \leftarrow \text{getInternalDependencies}(C_j, \text{highlevel})$ 
     $NP_j \leftarrow \text{getMostCoupledPackage}(C_j, \text{highlevel})$ 
     $NPList_j \leftarrow \text{getCoupledPackages}(C_j)$ 
    for each package  $NP_k$  do
       $interDepList \leftarrow \text{getExternalDependencies}(C_j, NP_k, \text{highlevel})$ 
    end for

     $coupling_j \leftarrow \text{length}(interDepList)$ 
     $cohesion_j \leftarrow \text{length}(intraDepList)$ 

    if  $coupling_j > cohesion_j$  then
       $subject \leftarrow C_j$ 
       $from \leftarrow P_i$ 
       $to \leftarrow NP_j$ 
      add new Recommendation( $subject, from, to$ ) to  $recommendations$ 
    end if
  end for
end for
return  $recommendations$ 

```

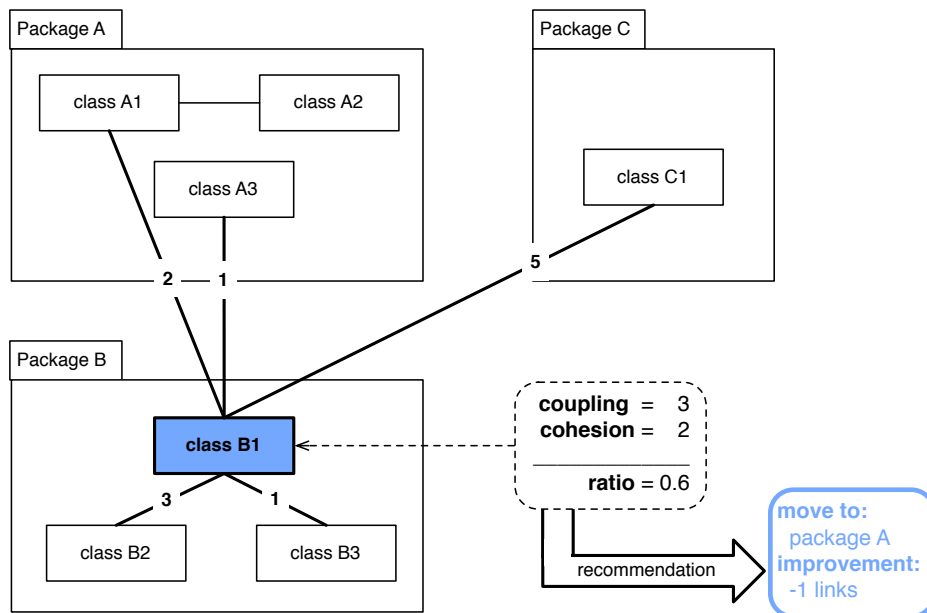


Figure 4.12: Example state as seen by the Low-level Global Strategy

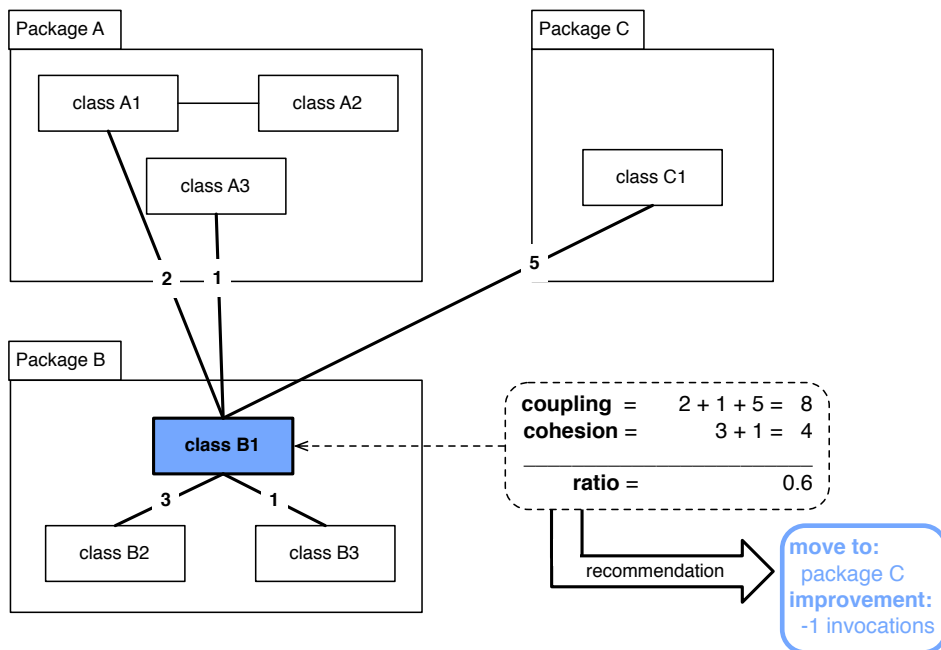


Figure 4.13: Example state as seen by the Low-level Global Strategy

Algorithm 6 dependencyRecommender - Low Level Global Strategy

```

for each package module  $P_i$  do
  for each contained class  $C_j$  do
     $intraDepList \leftarrow \text{getInternalDependencies}(C_j, lowlevel)$ 
     $NP_j \leftarrow \text{getMostCoupledPackage}(C_j, lowlevel)$ 
     $NPList_j \leftarrow \text{getCoupledPackages}(C_j)$ 
    for each package  $NP_k$  do
       $interDepList \leftarrow \text{getExternalDependencies}(C_j, NP_k, lowlevel)$ 
    end for

    for each dependency  $d_k$  of  $interDepList$  do
      { $d_k$  is the sum of the number of invocations}
       $coupling_j \leftarrow coupling_j + d_k$ 
    end for

    for each dependency  $d_k$  of  $intraDepList$  do
       $cohesion_j \leftarrow cohesion_j + d_k$ 
    end for

    if  $coupling_j > cohesion_j$  then
       $subject \leftarrow C_j$ 
       $from \leftarrow P_i$ 
       $to \leftarrow NP_j$ 
      add new Recommendation( $subject, from, to$ ) to  $recommendations$ 
    end if
  end for
end for
return  $recommendations$ 

```

Since the global strategies compute the coupling of the entity as the full number of dependencies of that entity, the mobility ratio (recalling: the external dependencies of the entity over the total dependencies) is over estimated implying a higher mobility ratio and thus a higher chance to become a recommendation candidate.

Conservative strategy

The conservative strategy analyzes the system in the same way the Local strategies do. It is therefore a special case of the local strategy. The difference is the higher weight which is given to the cohesion of a class which is set to twice as much as the external coupling. This directly influences the mobility ratio of the class by lowering it, thus following a conservative approach and returning less results compared to other recommendation strategies.

The approach is depicted in figures 4.14 for the high level conservative strategy and in 4.15 for the low level conservative strategy (and defined in algorithms 7 and 8 respectively).

Algorithm 7 dependencyRecommender - High Level Conservative Strategy

```

...
{same as High Level Local Strategy with multiplier (=2) check}
if  $coupling_j > (multiplier * cohesion_j)$  then
  ...
end if
...
return recommendations

```

Algorithm 8 dependencyRecommender - Low Level Conservative Strategy

```

...
{same as Low Level Local Strategy with multiplier (=2) check}
if  $coupling_j > (multiplier * cohesion_j)$  then
  ...
end if
...
return recommendations

```

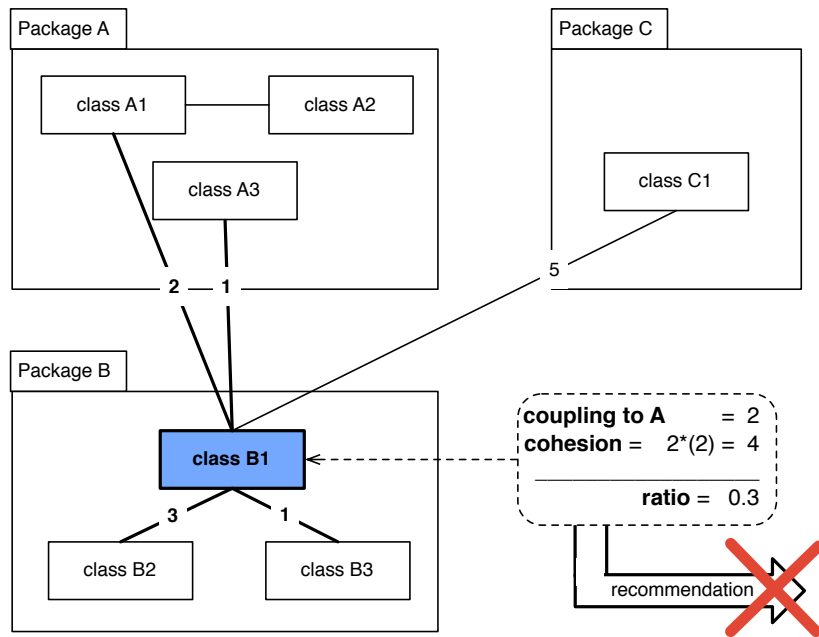


Figure 4.14: Example state as seen by the High-level Conservative Strategy

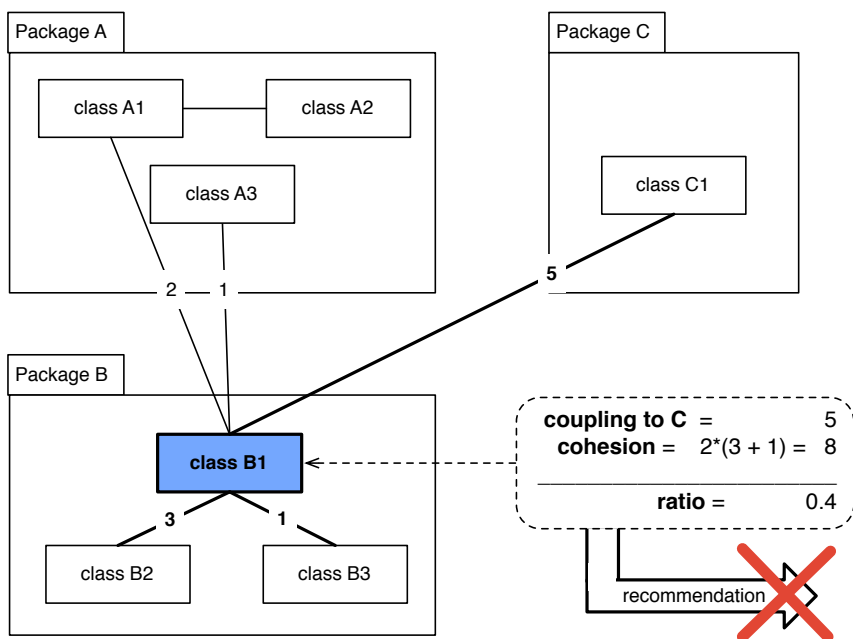


Figure 4.15: Example state as seen by the Low-level Conservative Strategy

Alternative strategy

This strategy differs in the view of the coupling of an entity. In this case we observe the local best candidate and compare it to the rest of the dependent entities. We therefore use such an “alternative” view in the sense that only if the dependencies towards the most coupled package (the best candidate) is higher than the rest of the dependencies then such an entity can be considered as a candidate (with the following analysis of whether its coupling is greater or less than its cohesion).

Again, the depictions of the high-level and low-level variant of the alternative strategy just described can be found in figures 4.16 and 4.17. For the algorithms refer to the pseudocodes defined in 9 and 10.

Algorithm 9 dependencyRecommender - High Level Alternative Strategy

```

for each package module  $P_i$  do
  for each contained class  $C_j$  do
     $intraDepList \leftarrow$  getInternalDependencies( $C_j, highlevel$ )
     $NP_j \leftarrow$  getMostCoupledPackage( $C_j, highlevel$ )
     $interDepList \leftarrow$  getExternalDependencies( $C_j, NP_j, highlevel$ )
     $NPList_j \leftarrow$  getCoupledPackages( $C_j$ )
    for each package  $NP_k$  do
       $fullInterDepList \leftarrow$  getExternalDependencies( $C_j, NP_k, highlevel$ )
    end for

     $coupling_j \leftarrow$  length( $interDepList$ )
     $cohesion_j \leftarrow$  length( $intraDepList$ )
     $totalCoupling_j \leftarrow$  length( $fullInterDepList$ )

     $remaining_j \leftarrow$  ( $totalCoupling_j - coupling_j$ )
    if  $coupling_j > cohesion_j$  &  $coupling_j > remaining_j$  then
       $subject \leftarrow C_j$ 
       $from \leftarrow P_i$ 
       $to \leftarrow NP_j$ 
      add new Recommendation( $subject, from, to$ ) to  $recommendations$ 
    end if
  end for
end for
return  $recommendations$ 

```

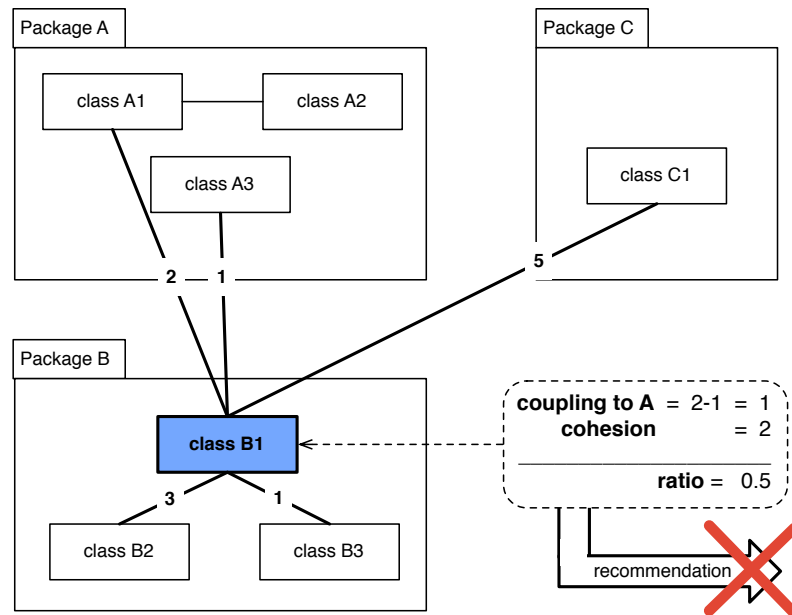


Figure 4.16: Example state as seen by the High-level Alternative Strategy

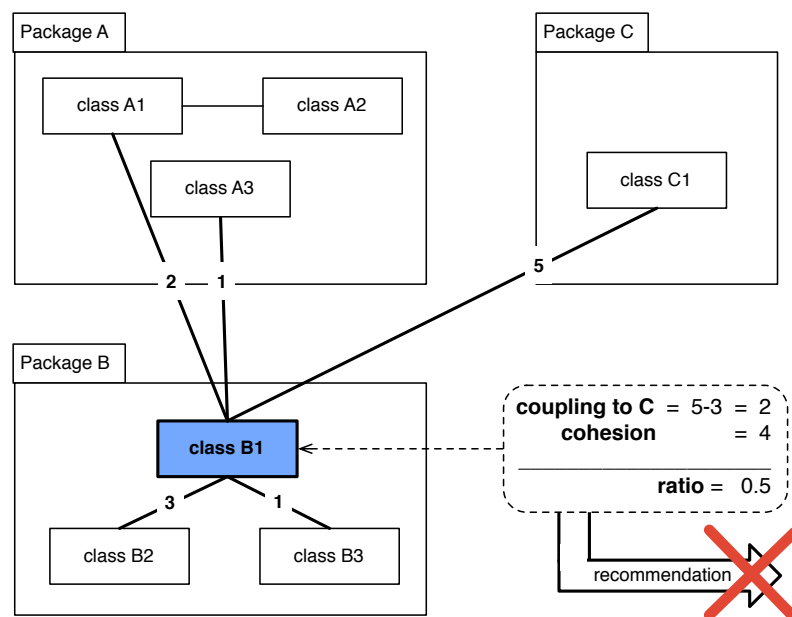


Figure 4.17: Example state as seen by the Low-level Alternative Strategy

Algorithm 10 dependencyRecommender - Low Level Alternative Strategy

```

for each package module  $P_i$  do
  for each contained class  $C_j$  do
     $intraDepList \leftarrow \text{getInternalDependencies}(C_j, lowlevel)$ 
     $NP_j \leftarrow \text{getMostCoupledPackage}(C_j, lowlevel)$ 
     $interDepList \leftarrow \text{getExternalDependencies}(C_j, NP_j, lowlevel)$ 
     $NPList_j \leftarrow \text{getCoupledPackages}(C_j)$ 
    for each package  $NP_k$  do
       $fullInterDepList \leftarrow \text{getExternalDependencies}(C_j, NP_k, lowlevel)$ 
    end for

    for each dependency  $d_k$  of  $interDepList$  do
       $cohesion_j \leftarrow cohesion_j + d_k$ 
    end for
    for each dependency  $d_k$  of  $intraDepList$  do
       $cohesion_j \leftarrow cohesion_j + d_k$ 
    end for
    for each dependency  $d_k$  of  $fullInterDepList$  do
       $totalCoupling_j \leftarrow totalCoupling + d_k$ 
    end for

     $remaining_j \leftarrow (totalCoupling_j - coupling_j)$ 
    if  $coupling_j > cohesion_j$  &  $coupling_j > remaining_j$  then
       $subject \leftarrow C_j$ 
       $from \leftarrow P_i$ 
       $to \leftarrow NP_j$ 
      add new Recommendation( $subject, from, to$ ) to  $recommendations$ 
    end if
  end for
end for
return  $recommendations$ 

```

This strategy is similar to the global strategy in the sense that the external coupling metric (towards the target package) does not reflect the true value. This strategy has been added in order to provide an alternative view of the critical modules offered by the global strategy. In fact the result of this strategy is that in order for an entity to be added to the recommendations, its external coupling towards the package it's most coupled with must be greater than its cohesion (as usual). On top of that, it has to be greater than the added sum of the remaining external coupling of the entity therefore tending towards a conservative approach.

Strategy	Method	metric used	scope of analysis
Inheritance	-	class inheritance	full system
Global	High-level	dependency links	full neighborhood
	Low-level	n. of invocations	full neighborhood
Local	High-level	dependency links	best neighbor
	Low-level	n. of invocations	best neighbor
Conservative	High-level	dependency links	best neighbor
	Low-level	n. of invocations	best neighbor
Alternative	High-level	dependency links	best - remaining neighbors
	Low-level	n. of invocations	best - remaining neighbors

Table 4.1: Overview of the recommendation strategies

4.3 Scenarios

Based on the two recommendation categories and the strategies which can be applied, we can define a number of scenarios of usage of the tool which are meant to represent the different aims and approaches of the reverse architect.

First impression

This usage scenario corresponds to the proposed way to tackle a system for the first time. We define two opposite approaches as based on the desire to assess the state of the system from a greedy approach which would return recommendations with high confidentiality or from a softer approach based on obtaining an overview of all the critical modules and entities.

1. *Greedy*

In order to quickly receive a list of true candidates that reflect classes which are wrongly packed due to their number of external dependencies we can apply global strategies with the highest mobility ratio. This would return all classes that have no cohesion within their package (we thus avoid the issue with the global strategies which may lead to spurious results).

2. *Soft*

By applying the Inheritance strategy and the Global strategies we receive a larger set of results that would represent the noteworthy classes which have an inheritance relationship between each other or a higher coupling than cohesion. As discussed previously, by using the global dependency strategies what we obtain in return is an informative set of results which may not contain improvements but that are useful for identifying the critical elements (from our inter and intra package dependencies point of view). Also the Inheritance strategy is intended for this use since inheriting from an external package, although clearly increasing the coupling of the entity, does not mean that the entity is in the wrong module.

Decoupling optimization

This usage scenario corresponds to the attempt to reduce the number of inter module dependencies. We therefore use high-level dependencies only and again we can follow two directions: a greedy way based both on local and global information (and with a mobility ratio = 1), which would return the modules that have no cohesion (as for the First impression greedy scenario), or a softer approach based on the same strategies but with a lower mobility ratio threshold.

1. *Greedy*

If our interest is to completely decouple as much as possible the packages, then the Inheritance informative strategy would be useful to assess all of the inheritance relationships which span across different packages. Also, using the Local High level strategy we would obtain all of the moves which would reduce the overall number of dependencies. Also global strategies could be applied if the result set is manageable in order to individually inspect each result and assess the validity of the recommendation.

2. *Soft*

Always using high-level dependencies, the Local strategy could be applied or, depending on the complexity of the system under analysis, if the result set is too large, the High-level Conservative strategy could be used to reduce the size of the returned results.

Weighted optimization

The architecture recovery process can also be aimed at reducing the global number of invocations (thus low-level dependencies). This is thus a weighted optimization since it tries to reduce the number of invocations between packages rather than eliminate the dependency links.

Also here we propose a greedy and soft approach which use the same Low-level Local and Conservative strategies to obtain the recommendations.

1. *Greedy*

The mobility ratio is raised to 0.7 to limit the number of results.

2. *Soft*

The mobility ratio is left to the default (0.5).

Also here as for the Decoupling optimizations, the local strategy could be applied.

Table 4.2 summarizes the three usage scenarios which we just described.

First impression:

Approach	Strategies	High-level dependencies	Low-level dependencies	Subject Mobility
greedy	<i>G</i>	✓	✓	1
soft	<i>I G L</i>	✓	✓	0.5

Decoupling :

Approach	Strategies	High-level dependencies	Low-level dependencies	Subject Mobility
greedy	<i>I L</i>	✓		1
soft	<i>L C</i>	✓		0.5

Weighted:

Approach	Strategies	High-level dependencies	Low-level dependencies	Subject Mobility
greedy	<i>L C</i>		✓	0.7
soft	<i>L C</i>		✓	0.5

(where *C* : Conservative, *I* : Inheritance, *G* : Global, *L* : Local)

Table 4.2: Proposals for usage scenarios

Chapter 5

Validation

We evaluated the tool on various software architectures ranging from simple corner case examples used for testing to larger case studies with real systems. We will present now our findings first in two simple example systems to depict the functionality and describe the recommendations, followed by two other systems, namely SoftwareNaut and CodeCity which have been used as subjects for the recommendations evaluation.

5.1 AICup

This initial simple system can be used to describe the application of a scenario and the corresponding results. When the system is loaded in SoftwareNaut, the default view which is presented is the top level module containing the system. By expanding such package we are presented with the top level architecture of the system (a depiction can be found in figure 5.1). This corresponds to the current level of exploration of the system. Figure 5.2 depicts instead the additional hidden expansion of the modules done by the recommendation system to gather the metrics.

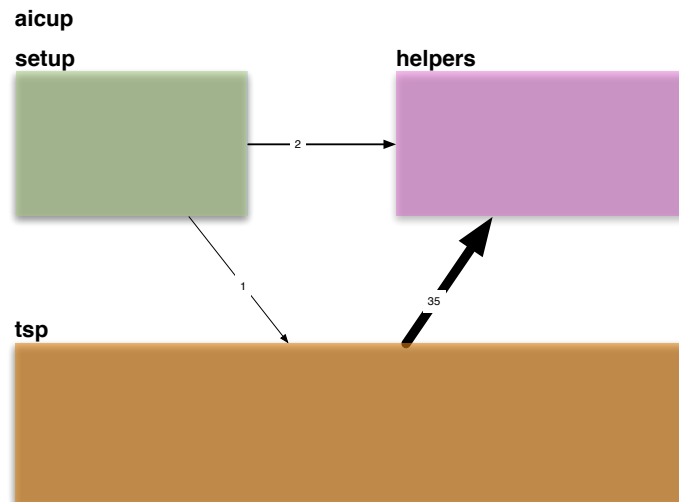


Figure 5.1: Current exploration view for the simpleSample example system

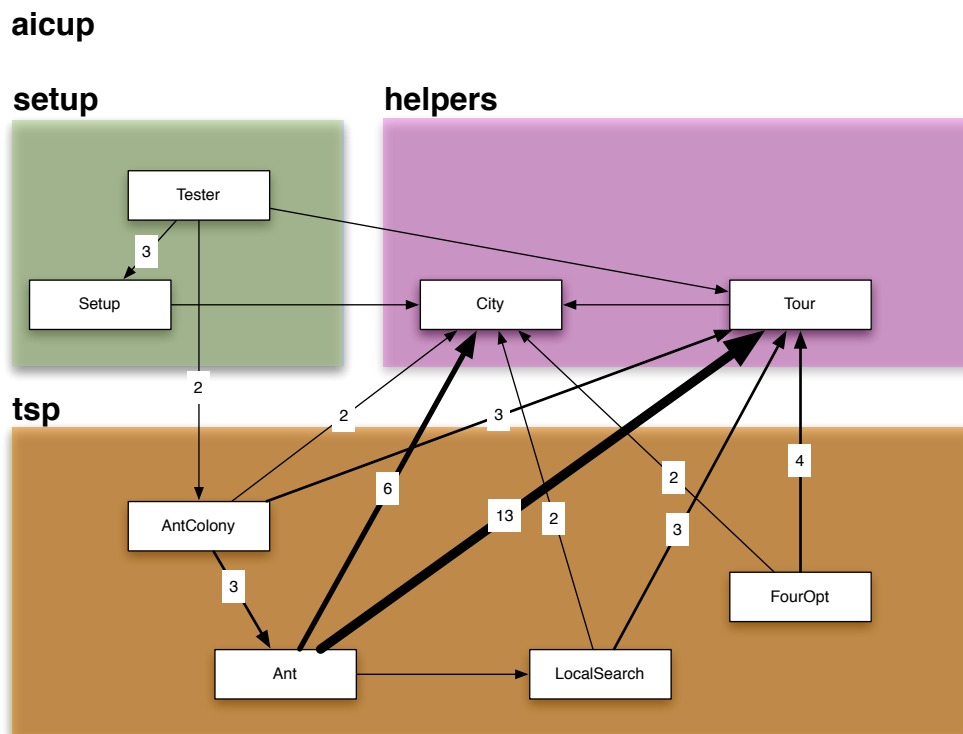
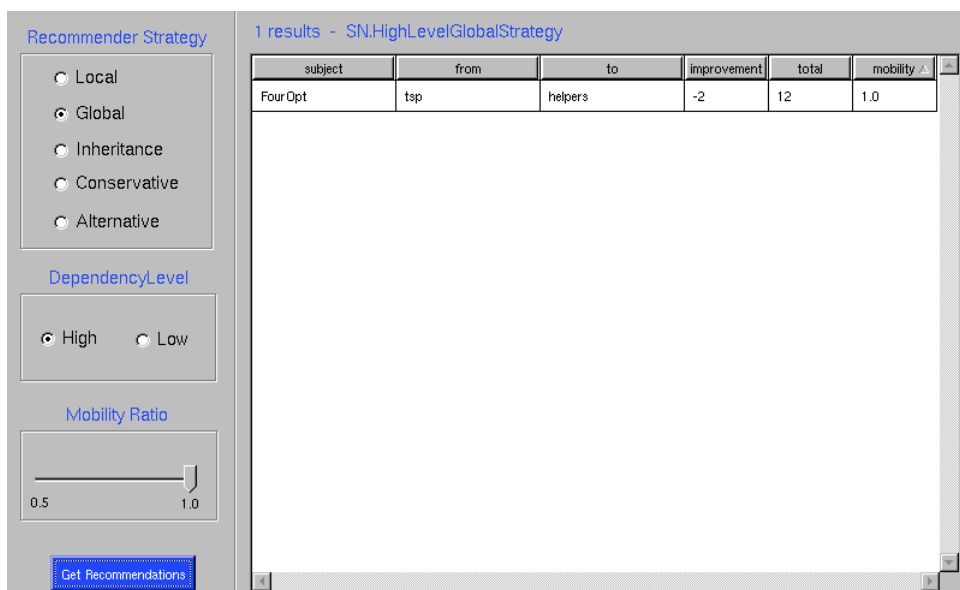


Figure 5.2: Internal dependency details for the AICup example system

5.1.1 First impression

greedy approach

1. Global strategy, high-level dependencies, subject mobility=1
As depicted in figure 5.3 the result is a single recommendation. In fact, referring back to figure 5.1, class `FourOpt` has no internal dependencies within the package `tsp`. The high level improvement which would be obtained by performing this move would be to remove the two dependency links that tie `FourOpt` to package `helpers`.



subject	from	to	improvement	total	mobility
FourOpt	tsp	helpers	-2	12	1.0

Figure 5.3: AI Cup example - Recommendation results returned by the High Level Global Strategy

2. Global strategy, low-level dependencies, subject mobility=1 (results depicted in figure 5.4)
Same as previously but from a low level dependencies point of view (the improvement would be to remove the six method invocations towards classes in package `helpers` .

subject	from	to	improvement	total	mobility
FourOpt	tsp	helpers	-6	40	1.0

Figure 5.4: AI Cup example - Recommendation results returned by the Low Level Global Strategy

soft approach

1. Local strategy, high-level dependencies, subject mobility=0.5
As depicted in figure 5.5 the results set grows when the mobility ratio is set to the lowest value. What stands out is the classes City and Tour which have a high mobility ratio and provide the best improvement in terms of dependency links removed.
2. Local strategy, low-level dependencies, subject mobility=0.5 (figure 5.6)
When using the low-level dependencies we can see that the two classes City and Tour which in the previous case were undistinguishable now are in relation and we can see that performing the recommended move Tour, from: helpers, to: tsp gives a double improvement than performing the one for City. Another noticeable result is to move Ant which would also provide a good improvement. This last recommendation is the one which could be followed since it would provide a clear separation between model classes such as Ant, City, Tour and algorithms and procedure-based classes (FourOpt, LocalSearch, AntColony).

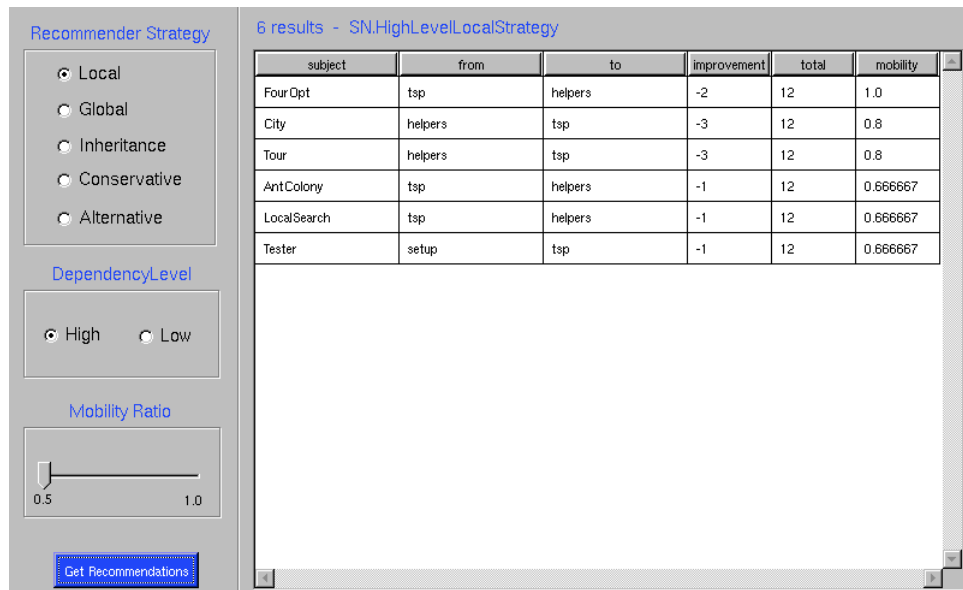


Figure 5.5: AI Cup example - Recommendation results returned by the High Level Local Strategy

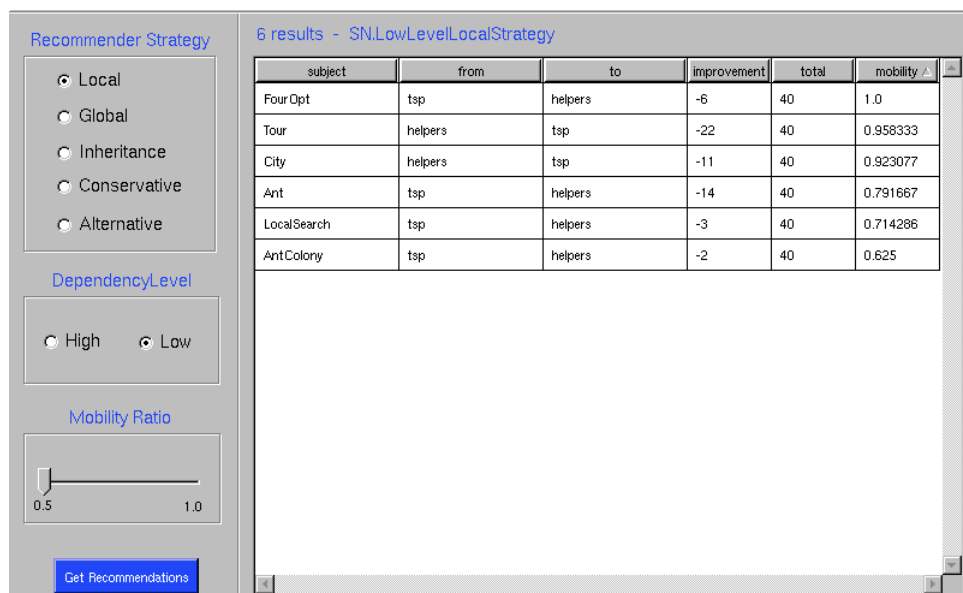


Figure 5.6: AI Cup example - Recommendation results returned by the Low Level Local Strategy

5.2 SimpleSample

The SimpleSample system is composed of 6 packages and 14 classes. It contains two inheritance hierarchies and is partitioned according to an MVC pattern into controller, ui and model packages plus a util package. The current exploration level of the system is the top level of the architecture (depicted in figure 5.7).

In this example (the internal state as seen by the recommendation tool is depicted in figure 5.8) we apply all of the strategies and compare the results.

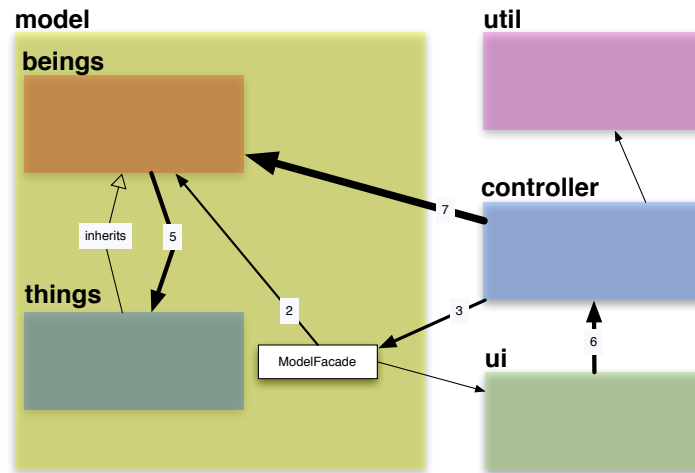


Figure 5.7: Current exploration view for the simpleSample example system

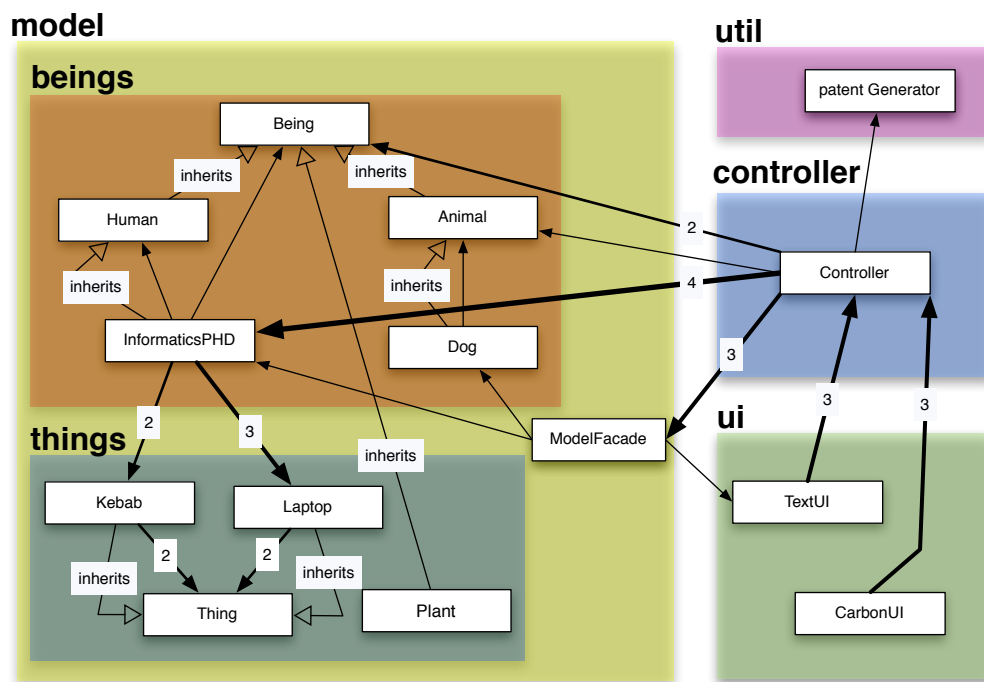


Figure 5.8: Internal dependency details for the simpleSample example system

5.2.1 Decoupling optimization

We define the correct partitioning of the system in figure 5.9 and apply all of the strategies first taking a high level view of the dependencies and then from a low-level view with a higher mobility threshold (as for the *soft weighted optimization* scenario).

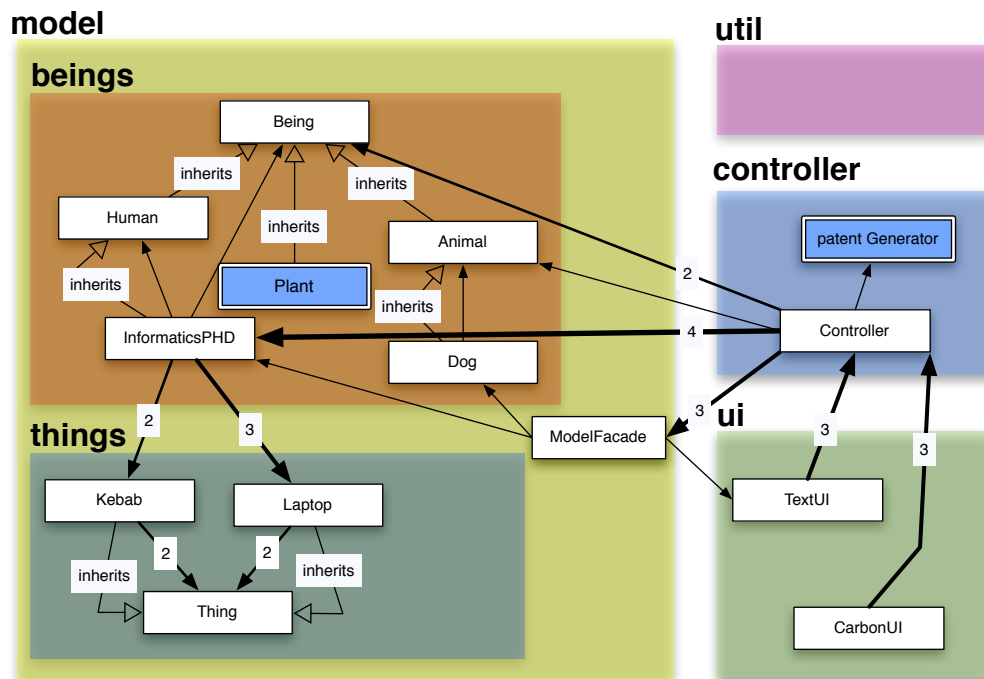


Figure 5.9: simpleSample example: Valid moves

1. Inheritance strategy

The inheritance strategy correctly returns a single result (figure 5.10) which corresponds to moving class `Plant` from: `things` to the `beings` package.

2. Global strategy, high-level dependencies, subject mobility=0.5 (figure 5.11)

The global strategy recommends all of the classes which have a high number of external dependencies among which we have the `Controller` and `InformaticsPHD` classes.

3. Local strategy, low-level dependencies, subject mobility=0.5

As we take a more restricted view, the incorrect recommendations are eliminated (such as `InformaticsPHD`, see figure 5.12).

4. Conservative & Alternative strategy, low-level dependencies, subject mobility=0.5 (figure 5.13)

The conservative and alternative strategies do not recommend to move the `Controller` class and only return the single `PatentGenerator`.

subject	from	to	improvement	total	mobility
Plant	things	beings	'N/A'	'N/A'	'N/A'

Figure 5.10: Inheritance Strategy applied to SimpleSample

Recommender Strategy

- Local
- Global
- Inheritance
- Conservative
- Alternative

DependencyLevel

- High
- Low

Mobility Ratio

0.5 ————— 1.0

Get Recommendations

3 results - SN.HighLevelGlobalStrategy

subject	from	to	improvement	total	mobility
PatentGenerator	util	controller	-1	10	1.0
Controller	controller	model	-1	10	0.818182
InformaticsPHD	beings	things	0	10	0.666667

Figure 5.11: High Level Global Strategy applied to SimpleSample

Recommender Strategy

- Local
- Global
- Inheritance
- Conservative
- Alternative

DependencyLevel

- High
- Low

Mobility Ratio

0.5 ————— 1.0

Get Recommendations

2 results - SN.HighLevelLocalStrategy

subject	from	to	improvement	total	mobility
PatentGenerator	util	controller	-1	10	1.0
Controller	controller	model	-1	10	0.6

Figure 5.12: High Level Local Strategy applied to SimpleSample

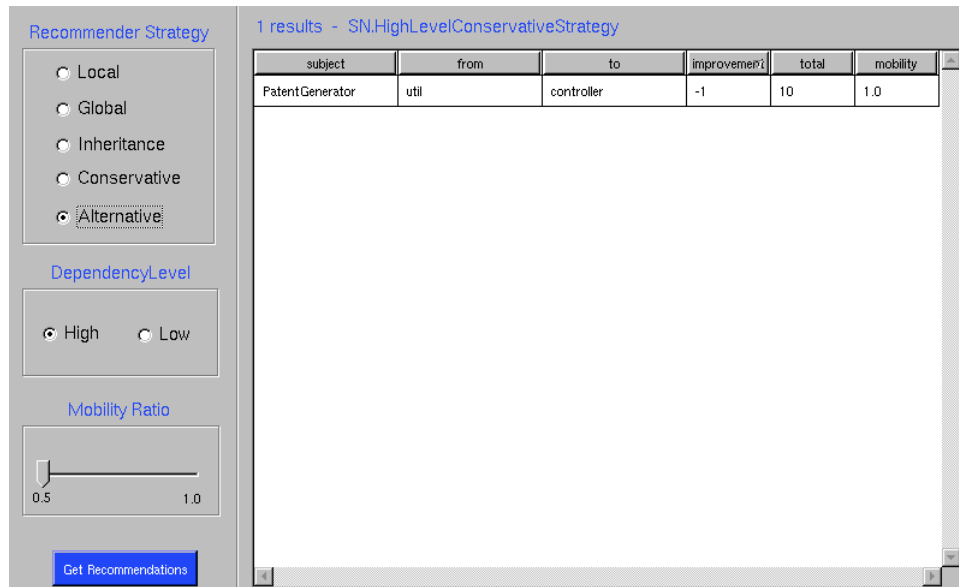


Figure 5.13: High Level Conservative Strategy applied to SimpleSample

5.3 Considerations on the strategies

As shown previously, the different strategies compute the amount of dependencies in different ways and under different scopes. Specifically:

Global Strategies

Global strategies tend to provide more recommendations on average since they compare the full coupling of the subject against its cohesion. Clearly, if an entity has many external dependencies it will become a recommendation candidate even if those dependencies are highly scattered among many targets. Since one can move an entity to only place, the improvement would be rather small (if not even a degradation if it's < the cohesion of the entity). The previous external dependencies of the entity would remain external with the exception of the ones towards the package the entity was just moved.

Global strategies therefore provide relatively many results which indicate noteworthy entities in the sense of classes with fewer dependencies within their module with respect to other modules.

Local Strategies

Local strategies are not concerned with the previous issue since their view of the system is limited to the subject and a target package only. The assumption here is that even if we are not considering all of the system, any external dependency would still remain external after the move.

These strategies first perform a local analysis of the best target and then compare the amount of dependencies towards that target with the dependencies within the entity's package (and of course propose a candidate only in the case of the first being greater than the second).

Having solved the issue with the global strategies returning low or non-improvement results, the local strategies, give the assurance that moving the entity towards the proposed target would reduce the coupling because it would remove more external dependencies than the ones it would introduce (which is the previous cohesion which would become coupling after the move).

Conservative & Alternative strategies

Conservative strategies are based on local strategies and naturally return fewer results since, in order to ensure a low mobility of classes, the cohesion of a class is boosted (as in the conservative strategy with a multiplier = 2), or a further comparison is done between the amount of dependencies towards the target package and all of the other targets (as in the alternative strategy). In this way we enforce that only if a class has a much higher coupling than cohesion or if it is highly coupled with a module and basically only with that module then we propose a recommendation.

The alternative strategies have not shown improvements when compared with other strategies due to the chance that it will return false positives (as for the global strategies).

Multiple rounds of recommendation strategies

As described in the proposed usage scenarios the strategies can be applied sequentially to obtain a narrower view of the system (meaning less results) starting from a wide view of the critical entities. We therefore propose to use the global and alternative strategies to obtain an upper and a lower bound on the number of returned recommendation results. The local and conservative strategies can then be applied to obtain more precise results.

On top of that, given the exploratory nature of SoftwareNaut, the size of the recommendation results may vary considerably depending on the parameters, exploration views and filters which can be applied in SoftwareNaut (refer to [12] and [13] for full details).

The operations which influence the recommendations returned by the tool are the filtering (removal/hiding of dependency edges or node entities) and the level of exploration of the system which starts from a high level package-only view and can be explored down to the individual classes and methods.

5.4 Softwareaut

The first system which will be used to validate the recommendation tool is Softwareaut. Since we are dealing with real systems the complexity of their architecture cannot be depicted as for the previous simple examples.

Therefore we have applied all of the strategies on both the high-level and low-level dependency views and evaluated the returned recommendations based on our knowledge of the system and about its correct structure. The current exploration view is however depicted in figure 5.14. We apply the recommendations to the subsystem composed of the MARS package and Softwareaut namespace which itself contains several packages.

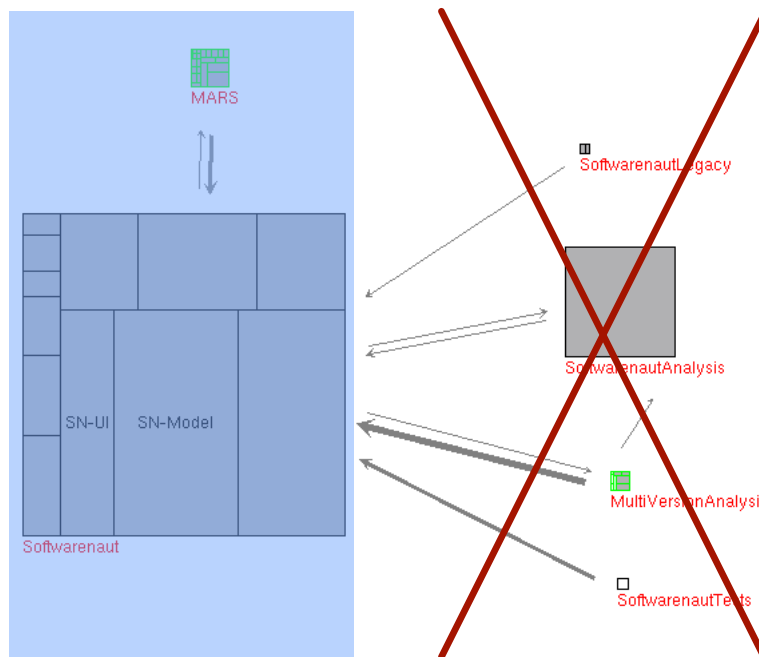


Figure 5.14: Current exploration level and considered modules for Softwareaut

The detailed results can be found in figure 5.15 which contains the recommended targets for each recommendation candidate in a spreadsheet. For reasons of space, the quantitative information (such as the mobility ratio, improvement and system coupling) are not included although, as stated several times, that information would provide valuable insights on the importance/confidence in the returned recommendation and therefore on the level of “trust” that the reverse engineer should put into it.

The evaluated results can be summarized through the known precision and recall measures as follows:

$$\textit{precision} = \frac{\textit{true positives}}{\textit{total retrieved positives}}$$

$$\textit{recall} = \frac{\textit{true positives}}{\textit{total true positives}}$$

where *total true positives* = *true positives* + *false negatives*
and *total retrieved positives* = *true positives* + *false positives*

However, since the true number of move recommendations is not known, we cannot compute the precision of the tool. Regarding the recall however the tool returns 20 correct results over the 101 total returned thus obtaining a value slightly lower than 20%.

5.4.1 Considerations on the results

The results contained in figure 5.15 show that a high number of false positive results are given for subjects which have a high external coupling but which logically belong where the developer had originally placed them. Such entities are for example, in the case of Softwareonaut, the figures, layouts, decorations and panel classes which are grouped in the SN-ModuleFigures, SN-Layouts, SN-DecorationFigures, and SN-Detail-POV packages. These packages in fact represent collections of their contained classes, all of which are grouped according to their main logical functionality. Almost half of the recommendations are related to moving such highly linked classes to the package they are most coupled with. We can therefore say that a possible improvement on the recommendations would consist in analyzing the naming conventions which are used for packages and classes and exclude any entities that have some degree of matching with their parent package. However, implementing this solution would imply that we assume that the system follows a “proper” naming convention which of course may not be the case. We therefore keep the approach as it is and return a higher number of (possibly redundant or wrong) recommendations since otherwise some entities could be mistakingly ignored on the basis of a similar pattern between classes and their package.

With respect to the correct results (marked as YES in the table), these contain conflicting recommendation targets (which mostly depend on the specific high-level or low-level dependency views). On the other hand, results which are partially correct (meaning that the recommendation is not incorrect but that the specific choice of whether to perform the recommended move depends on the amount of improvement that would result from that move), have a higher agreement about the target package among the different strategies.

Finally, an unexpected result arises from the fact that the low-level dependency based strategies apparently perform worse than their high level counterparts. This result was unexpected since the low level strategies naturally have more material to work with and should therefore return more precise results or at least the same as the ones returned by the high level views.

SUBJECT	FROM	TO (HIGH LEVEL LOCAL)	TO (LOW LEVEL LOCAL)	TO (HIGH LEVEL GLOBAL)	TO (LOW LEVEL GLOBAL)	TO (HIGH LEVEL CONSERVATIVE)	TO (LOW LEVEL CONSERVATIVE)	TO (HIGH LEVEL ALTERNATIVE)	TO (LOW LEVEL ALTERNATIVE)	EVALUATION
Preferences_class	CodeCityViewConfig	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	YES
Preferences	CodeCityViewConfig	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	YES
CityEasyl	CodeCityScripting	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	MAYBE
DisarmonyMapGUI	CodeCityGUI	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	MAYBE
AbstractSelectionDialog_class	CodeCityGUI	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	MAYBE
ViewConfiguratorGUI	CodeCityGUI	CodeCityVisualMappings	CodeCityVisualMappings	CodeCityVisualMappings	CodeCityVisualMappings	CodeCityVisualMappings	CodeCityVisualMappings	CodeCityVisualMappings	CodeCityVisualMappings	MAYBE
LinearConverter_class	CodeCityUtils	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	MAYBE
RelationCollector	CodeCityUtils	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	MAYBE
NewConfiguration	CodeCityGUI	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	MAYBE
DisarmonyMapGUI_class	CodeCityGUI	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	MAYBE
VersionControlDialog_class	CodeCityGUI	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	MAYBE
ConfigurationRepository_class	CodeCityViewConfig	CodeCityVisualMappings	CodeCityVisualMappings	CodeCityVisualMappings	CodeCityVisualMappings	CodeCityVisualMappings	CodeCityVisualMappings	CodeCityVisualMappings	CodeCityVisualMappings	NO
ConfigurableFieldManager_class	CodeCityViewConfig	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	NO
BindingBox_class	CodeCityLayouts	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	NO
SnapshotBuilder	CodeCityViewBuilders	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	NO
SnapshotNodePlacer	CodeCityViewBuilders	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	NO
SnapshotViewBuilder_class	CodeCityViewBuilders	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	NO
TimeTravelDisplayModel	CodeCityMVC	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	NO
AbstractLayout	CodeCityLayouts	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	NO
SnapshotViewBuilder	CodeCityViewBuilders	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	NO
FunctionalMapper	CodeCityVisualMappings	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	NO
BlueGreenYellowColorScheme	CodeCityColorSchemes	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	NO
ColorLinearMapper	CodeCityVisualMappings	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	NO
MapperDialog	CodeCityGUI	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	NO
BoxPlotConverter	CodeCityGUI	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	NO
AbstractNodeGlyph	CodeCityMVC	CodeCityLayouts	CodeCityLayouts	CodeCityLayouts	CodeCityLayouts	CodeCityLayouts	CodeCityLayouts	CodeCityLayouts	CodeCityLayouts	NO
TimelineDisplayModel	CodeCityMVC	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	NO
BlockMapper	CodeCityGUI	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	NO
PhotographerDialog_class	CodeCityGUI	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	NO
TimelineViewBuilder_class	CodeCityViewBuilders	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	NO
SnapshotDisplayModel	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	CodeCityMVC	NO
SearchTermDialog	CodeCityGUI	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	NO
AbstractLayout_class	CodeCityLayouts	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	NO
TimeTravelViewBuilder	CodeCityViewBuilders	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	NO
CityScriptExamples_class	CodeCityScripting	CodeCityLayouts	CodeCityLayouts	CodeCityLayouts	CodeCityLayouts	CodeCityLayouts	CodeCityLayouts	CodeCityLayouts	CodeCityLayouts	NO
DisplayModel	CodeCityLayouts	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	NO
SpaceManager	CodeCityGlyphs	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	NO
AbstractGlyph	CodeCityGlyphs	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	NO
TimeTravelEdgeBuilder	CodeCityViewBuilders	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	CodeCityGUI	NO
Photographer	CodeCityUtils	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	NO
SnapshotEdgeBuilder	CodeCityViewBuilders	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	NO
PhotographerDialog	CodeCityGUI	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	CodeCityGlyphs	NO
HistoryInfrastructureBuilder	CodeCityViewBuilders	CodeCityUtils	CodeCityUtils	CodeCityUtils	CodeCityUtils	CodeCityUtils	CodeCityUtils	CodeCityUtils	CodeCityUtils	NO
ConstantMapper	CodeCityVisualMappings	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	NO
PhotoAutomator	CodeCityUtils	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	CodeCityViewConfig	NO
SpaceManager_class	CodeCityLayouts	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	CodeCityViewBuilders	NO

Figure 5.17: Summary table with the returned results given by different strategies applied to CodeCity

The recall in this case reaches 25%. As for Softwareaut, the majority of negative results are related to parts of the system which represent collections or groupings of classes which all follow a common logical goal and purpose.

5.6 Improvements

The main issues and problems on the results that are provided by the recommender are related to the recommendations which have a mobility ratio of 1. These entities, from the dependency point of view that we are considering, have zero dependencies within their containing module and therefore have no reason to be packed in the module in which they are. However, as has been found both for Softwareaut and CodeCity, several of these entities are grouped together because of their high level logical properties. Because of this, although performing the recommended move would reduce the coupling and therefore the complexity in the communication in the system, the move would actually degrade the architecture by moving entities which are supposed to have a single, well-defined purpose (such as a View/UI entity) to other packages which define other different high level responsibilities.

Because of this, referring back to the problem which we introduced in section 2.2.2, the issue of modularizing systems is composed of two opposite forces: from the “system point of view”, the dependencies metrics describe the amount of communication with other modules and the amount of single responsibility of the subject module. Through these metrics we are able to remove the coupling between pairs of modules thus improving the general architecture. However, modularizing the system according to the other direction, represented by the architects point of view, would lead to a system which is more understandable and “human-readable” but less close to how the system really is (in terms of communication of the modules). Because of this, we repeat and enforce the suggestion of using the recommendation system in parallel with the exploration of the system through Softwareaut (therefore using the nextexpansion scope). Not only does this approach return a fewer number of results which are more manageable for the reverse engineer, but also allows the recovery architect to observe simultaneously the system and evaluate if a specific returned recommendation should be taken into consideration or not (by deciding, based on the improvement and mobility ratio of the recommended subject, whether the improvement gain is worth the breaking of the naming conventions used throughout the system).

Chapter 6

Conclusions

6.1 Contributions

In this work we have proposed an approach to improve the architecture of the system based on two metrics which correspond to the external and internal dependencies of a module or package. Such a direction is aimed at lowering the total number of dependencies within the (sub)system under analysis. We presented two general approaches for providing recommendations - one, just mentioned, based on the dependencies between entities, and the other based on the inheritance relationships between classes.

For the dependency-based recommendations we distinguished between high and low level dependencies to specify two different degrees of granularity for the metrics (one based on the existence of communication between entities and the other on the weighted method invocation counts).

Additionally we developed a set of dependency-based strategies for analyzing the system from different points of view (such as the global and alternative strategies which concentrate on the coupling of an entity or the conservative strategy which stresses the cohesion of the subject).

Finally, we included in the recommendation results a number of additional parameters and data to better inform the recovery architect user about the specific situation of a candidate. This additional information is provided mainly by the mobility ratio and by the coupling improvement fields which indicate the cohesion of the entity and the quality of the returned recommendation in terms of discounted dependencies.

Through the proposed approach we are able to reduce the complexity of the system in terms of cross-package dependencies and inheritance relationships.

6.2 Reflections

The two metrics on which the recommendation tool is based, provide valuable information on the strength of connections between modules and on the degree of single responsibility that a module has.

Given that these two values provide important insights in the quality of the system under analysis, we must say that software architecture is a much more complex problem which cannot be reduced to the simple degree of coupling or cohesion. The problem is therefore inherently intricate and two metrics, despite their expressiveness, cannot fully grasp the complexity of the problem, let alone solve it.

The approach that we have proposed therefore is inherently based on a constant and iterative evaluation of the recommendations done by the user, which has the arduous task of assessing and judging the results in order to identify the critical entities (based on the additional numerical information which is returned).

As has been mentioned several times, since the recommendations are based solely on the two external coupling and cohesion metrics, the proposed candidates represent entities which, from an external dependencies point of view, are incorrectly placed. However, there are several other reasons which may interfere with following such recommendations. We already discussed the cases where the naming conventions and high level logical properties become of higher importance than the improvements which would follow from moving an entity to the recommended location. On top of that we may have situations where the architectural style imposes a clear modularization and thus any recommended moves would break the architecture. Finally we should also consider that, since the tool is integrated in an architecture recovery system, there may not be the possibility to arbitrarily move certain parts of the system.

However, as we claimed before, the tool is intended as a recommender which can range from conservative to quite generous in terms of dependency thresholds. Through the different views and parameters we can assess the architectural disharmonies and pinpoint the critical modules which are at the center of these issues thus aiding the reverse engineer in the recovery process.

6.3 Future Work

We briefly outline some possible roads that can be taken and future improvements to the tool.

Improved Inheritance analysis:

The single strategy which is concerned with the Inheritance analysis could be improved in order to return better and more informative results. It would also reduce the number of recommendation results which are returned which, at the moment can be very high for systems which make heavy use of inheritance hierarchies.

A possible improvement would be to consider the type of relationship between child and parent classes. If a child class simply extends a parent class, this would be considered as the lowest form of coupling between the two modules. In case a class instead, refines the definition of the superclass, by changing the signatures or adding by adding new definitions, then the coupling would be more intense.

A more refined recommendation strategy could analyze this fact and include in the candidate list only the classes which have a refinement inheritance relationship between each other.

Evolutionary timeline:

If data regarding the lifetime of the software system is incorporated in the analysis, then several recommendation strategies could be devised based on this added information. For example, the mobility ratio of a class could be proportional to its age since each version release which has a specific class left in the same location would strengthen the fact that the class should be left untouched.

Advanced visualization:

The recommendation system in MARS returns information through a table and adds visual cues regarding the selected recommendation (such as changing the text color of the selected subject class, origin and target packages).

However, more advance visualization techniques can be added which aid the exploration especially for large systems. For example a simple enlarging / border stroke on the shape under inspection would clearly make the subject identifiable. In order to indicate the direction of the move, a clear large arrow pointing to the target package region could be used or also an animation effect.

Feedback to the system model:

The recommendation system is always based on the model of the system which is provided by the user. To further integrate the tool and to aid the reverse engineer, the returned recommendations could be selectively chosen by the user as correct or not after which the selected recommended moves could be actually applied to the system. This would change the structure of the software system and could be used in order to iteratively modify it based on the returned recommendations.

It is important to note that this last addition would require careful planning and probably the inclusion of other metrics or heuristics to improve the recall. Detailed logs, visualizations or even further a timeline-based navigation of the performed moves would also need to be developed to indicate the changes to the system and control them.

Figures

2.1	Software Architecture Recovery process	7
3.1	Top-down exploration metaphor used in Softwarenaut	10
3.2	Correlation between coupling and cohesion	21
3.3	MARS tool options & parameters sidebar	22
3.4	Softwarenaut toolbar	22
4.1	Recommendation threshold based on coupling/cohesion ratio . .	23
4.2	Example situation for a recommendation given by the inheritance analyzer	25
4.3	Example situation for a recommendation given by the dependency analyzer (in case the coupling > cohesion)	27
4.4	Example situation for a recommendation given by the dependency analyzer (in case the cohesion = 0)	28
4.5	Example system for the dependency level	31
4.6	Recommendation threshold based on the subject mobility ratio . .	32
4.7	Current view of the explanatory system for the scope parameter .	34
4.8	System as viewed by the next expansion scope	35
4.9	System as viewed by the flat system expansion scope	35
4.10	Example state as seen by the High-level Local Strategy	38
4.11	Example state as seen by the Low-level Local Strategy	38
4.12	Example state as seen by the Low-level Global Strategy	41
4.13	Example state as seen by the Low-level Global Strategy	41
4.14	Example state as seen by the High-level Conservative Strategy . .	44
4.15	Example state as seen by the Low-level Conservative Strategy . . .	44
4.16	Example state as seen by the High-level Alternative Strategy . . .	46
4.17	Example state as seen by the Low-level Alternative Strategy	46
5.1	Current exploration view for the simpleSample example system .	54
5.2	Internal dependency details for the AICup example system	54

5.3	AI Cup example - Recommendation results returned by the High Level Global Strategy	55
5.4	AI Cup example - Recommendation results returned by the Low Level Global Strategy	56
5.5	AI Cup example - Recommendation results returned by the High Level Local Strategy	57
5.6	AI Cup example - Recommendation results returned by the Low Level Local Strategy	57
5.7	Current exploration view for the simpleSample example system .	58
5.8	Internal dependency details for the simpleSample example system	59
5.9	simpleSample example: Valid moves	60
5.10	Inheritance Strategy applied to SimpleSample	61
5.11	High Level Global Strategy applied to SimpleSample	62
5.12	High Level Local Strategy applied to SimpleSample	62
5.13	High Level Conservative Strategy applied to SimpleSample	63
5.14	Current exploration level and considered modules for Softwareaut	66
5.15	Summary table with the returned results given by different strategies applied to Softwareaut	67
5.16	Current exploration level for CodeCity	70
5.17	Summary table with the returned results given by different strategies applied to CodeCity	71

Tables

3.1	Summary table of different coupling categories	15
3.2	Coupling categories associated with their main metric and abstraction level	17
3.3	Summary table of different cohesion categories	19
3.4	Cohesion categories associated with their main metric and abstraction level	20
4.1	Overview of the recommendation strategies	48
4.2	Proposals for usage scenarios	51

Bibliography

- [1] K. Babu, P. Govindarajulu, and A. Kumari. Development of the conceptual tool for complete software architecture visualization: Darch. *IJCSNS*, 2009.
- [2] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, Vol. 29, No.4, pp. 33-43, April 1996.
- [3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley Professional, 2003, 2003.
- [4] E. Chikofsky and J. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE sotware*, 1990.
- [5] S. Ducasse, M. Lanza, and S. Tichelaar. The moose reengineering environment. *Smalltalk Chronicles*, Aug 2001.
- [6] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: An extensible language-independent environment for reengineering object-oriented systems. *Proc. Second Int'l Symp. Constructing Software Eng. Tools (CoSET 2000)*, June 2000.
- [7] J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems. 1994.
- [8] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. *Proceedings of the International Conference on Software Maintenance*, pp. 99D108, 1988.
- [9] IEEE. Ieee recommended practices for architectural description of software intensive systems. *IEEE technical report*, 2000.
- [10] R. Krikhaar. Software architecture reconstruction. *PhD thesis, Universiteit van Amsterdam*, 1999.

-
- [11] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 2003.
 - [12] M. Lungu and M. Lanza. Softwrenaut: Cutting edge visualization. *Proceedings of Softvis 2006 (3rd International ACM Symposium on Software Visualization)*, pp. 179-180, 2006.
 - [13] M. Lungu and M. Lanza. Softwrenaut: Exploring hierarchical system decompositions. *Proceedings of CSMR*, 2006.
 - [14] M. Lungu and M. Lanza. Exploring inter-module relationships in evolving software systems. *Proceedings of CSMR*, 2007.
 - [15] M. Lungu, M. Lanza, and T. Gîrba. Package patterns for visual architecture recovery. *Proceedings of CSMR*, 2006.
 - [16] J. I. Maletic, A. Marcus, and L. Fen. Source viewer 3d (sv3d): a framework for software visualization. *Proceedings of the 25th International Conference on Software Engineering*, pp. 812-813, 2003.
 - [17] H. A. Mueller and K. Klashinsky. Rigi - a system for programming-in-the-large. *Proceedings of the 10th International Conference on Software Engineering*, pp. 80Ð86, 1988.
 - [18] D. L. Parnas. *Information Distribution. Aspects of Design Methodology*. North-Holland Publishing Company, 1972.
 - [19] D. Perry and A. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 1992.
 - [20] W. Stevens, G. Myers, and L. Constantine. *Structured Design*. Yourdon Press, 1979.
 - [21] C. Ware. *Information Visualization*. Morgan Kaufmann, 2000.
 - [22] R. Wetzel and M. Lanza. Visualizing software systems as citites. *Proceedings of VISSOFT*, 2007.