Architecture-Centric Software Evolution by Software Metrics and Design Patterns

Juha Gustafsson, Jukka Paakki, Lilli Nenonen, and A. Inkeri Verkamo

Department of Computer Science, University of Helsinki
P.O. Box 26, FIN-00014 University of Helsinki, Finland, +358 9 191 44180
E-mail: {gustafss,paakki,lnenonen,verkamo}@cs.helsinki.fi

Abstract

It is shown how software metrics and architectural patterns can be used for the management of software evolution. In the presented architecture-centric software evolution method the quality of a software system is assured in the software design phase by computing various kinds of design metrics from the system architecture, by automatically exploring instances of design patterns and anti-patterns from the architecture, and by reporting potential quality problems to the designers. The same analysis is applied in the implementation phase to the software code, thus ensuring that it matches the quality and structure of the reference architecture. Finally, the quality of the ultimate system is predicted by studying the development history of previous projects with a similar composition of characteristic software metrics and patterns. The architecture-centric software evolution method is supported by two integrated software tools, the metrics and pattern-mining tool Maisa and the reverse-engineering tool Columbus.

1. Introduction

A software system is under constant evolution during its life cycle. The evolution can be roughly seen as divided into two categories of changes that the system is facing: first, the system is gradually developed from general requirements to functional code in several phases (such as design, implementation, and testing); and second, the functional code is modified, or maintained, according to various needs from the users of the system (such as new requirements, bug fixes, or porting into new environments). Implementation and management of all these numerous and often spontaneous changes in a controlled manner is one of the main reasons for the cost of developing large and complex software systems.

It is commonly accepted that a simple, well-defined structure makes it easier to maintain a software system without corrupting its quality by unintended side effects [12]. Another universal fact is that the earlier the quality of software is assured, the more effective quality assurance is with respect to the system's overall development costs [2]. Both of these objectives can be met by carefully designing the architecture of the system, by measuring its quality before entering the actual implementation phase, and by verifying that the code indeed implements the intended architecture and the design decisions captured by it. Recently, this architectural focus has been widely emphasized both in the software industry and the research community, as can be seen from the emergence of software architectures as a central discipline in software engineering [21].

Maisa (Metrics for Analysis and Improvement of Software Architectures) is a currently ongoing project developing methods and tools for architecture-based measurement and assurance of software quality. The core idea in Maisa is to measure and predict the quality of a software system already in its design phase, by computing various kinds of metrics from its architecture, given as a set of UML diagrams [18]. In addition to computing standard metrics such as size and complexity, the Maisa tool approximates the quality of the architecture also by automatically mining instances of software (design) patterns and anti-patterns [3, 4, 11] from it.

Maisa keeps track of the evolution of the software system by storing the measurements in a built-in project database from which various statistics can be generated, either phase-wise over the current project or over several projects. In addition to architectural metrics, measurements of the final system (such as size and complexity of the software code) can be stored in the database.

The scope of Maisa has been extended beyond the software design phase by its integration to the reverse-engineering tool *Columbus* [9]. Columbus abstracts the structure of a software system (written in C or C++) into a UML class diagram. This class diagram, when considered as one view over the system's underlying architecture,

Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR'02)
1534-5351/02 $17.00 © 2002 IEEE

can be given as input to Maisa for checking that the implementation architecture (code) conforms to the design architecture.

We proceed as follows. First, the evolutionary support provided by Maisa and Columbus is summarized in Chapter 2. Then, the Maisa method for architectural quality measurement is presented in Chapter 3. The integration of Maisa and Columbus for verifying architectural conformance between the code and the design is described in Chapter 4. The statistical support of Maisa for managing software evolution is addressed in Chapter 5, followed by a discussion of related work in Chapter 6. Finally, experience and future work are summarized in Chapter 7.

2. Software Evolution Management by Maisa and Columbus

Maisa and Columbus provide support for a software development approach where the system is analysed and its quality is measured long before the testing phase. The metrics tool Maisa concentrates on the design phase and the reverse-engineering tool Columbus on the subsequent coding phase. Thus, the combination of the tools covers those phases of a typical software development project where the architecture of the system is in a central role. Accordingly, Maisa and Columbus can be characterised as a toolset for measuring, tracking, and managing architecture-centric software evolution.

Using Maisa and Columbus, one can manage the evolution and assure the quality of a software system in several ways. First, the technical quality of the system's architecture can be measured by computing various metrics over it. Second, the decisions made by the designers can be recovered by extracting instances of (design) patterns from the architecture. Third, symptoms of poor design decisions can be found by extracting instances of anti-patterns from the architecture. Fourth, by using the quality history of architecturally similar systems as a base, one can predict the quality of the system to be built from the architecture. If any of these steps reveals potential quality problems in the architecture, the problems should be studied, an improved architecture designed, and the new version of the architecture analysed by Maisa, until a sufficient level of architectural quality has been reached and the coding phase can start.

Finally, after the code has been implemented, its conformance to the intended architecture can be verified by using Columbus to recover the structure (class diagram) of the code and by using Maisa to compare it to the architecture as analysed in the design phase. If the structure of the code does not match the system's architecture, the deviations should be studied, the code restructured, and the architecture recovery process repeated until the software patterns found from the code are the same as those in the reference architecture (and the conventional software metrics over the code are within acceptable range). During all these stages the measurements are stored in Maisa's project database both for phase-wise statistical analyses of the current project and for supporting quality predictions by project-wise comparisons.

The architecture-centric software evolution model is depicted in Figure 1. Solid arrows represent information generated by Maisa, and dashed arrows represent quality improvements implemented by software designers on the basis of reports from Maisa. The double arrow between design architecture and implementation architecture stands for a metrical and structural comparison; Maisa provides a facility for comparing two architectural measurements. The subject architectures, metrics, pattern instances, predictions, comparison results, as well as statistical charts can be studied in Maisa, which provides a specific view for each purpose.

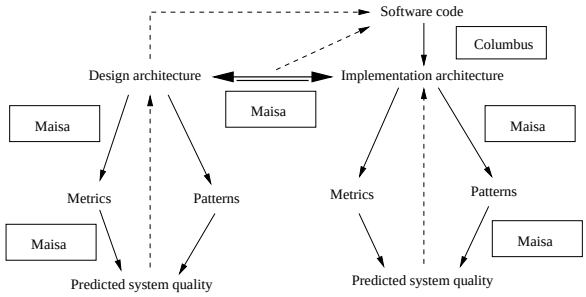

Figure 1. Architecture-centric software evolution

3. Metrics and Software Patterns in Maisa

The key concept in Maisa is to calculate both traditional metrics such as the cyclomatic complexity of a state diagram [6] and pattern based metrics such as the relative number of classes (i.e. the relative amount of classes that are contained in at least one design pattern instance) as early as possible. In practice this means using design diagrams, as they contain the minimum amount of information on which to base these calculations.

The Maisa tool produces three types of results:

- Simple object-oriented metrics (e.g., number of classes, number of attributes, number of associations, number of objects, depth of the inheritance tree)
- Pattern mining results (e.g., number of **Singleton** patterns and the respective instances in the diagram)
- Pattern-based metrics (e.g., relative number of classes)

One advantage of using simple metrics is that many of them can be calculated accurately already in the design phase. From the moment we have drawn our first class, we are ready to calculate the number of classes. Additionally, most simple metrics don't require any extensive computation, so with relatively little effort we can get lots of useful information for the user.

Currently, we mainly use class diagrams for pattern-based metrics. Maisa also calculates simple metrics from collaboration and state diagrams. Additionally, extended activity diagrams are used for performance analysis [22].

3.1. Constraint satisfaction in pattern mining

With pattern mining the situation is slightly different from the computation of regular metrics. In principle, we can search for patterns in any diagram (no matter how coarse it might be). In practice we need either extremely detailed diagrams or substantial help from the user, if we want this search to be precise. Another problem is that the patterns themselves are often quite vaguely defined.

The *constraint satisfaction problem* (CSP) [13, 16] is a generic technique that can be applied to a wide variety of tasks, in our case mining patterns from software architectures or software code [8, 19]. The CSP consists of variables and a set of constraints restricting the values that can be assigned to these variables. Unary constraints restrict the values of a single variable, while binary constraints represent a condition for a pair of variables. The CSP is often modelled as a graph, where the nodes represent the variables and the arcs represent the constraints.

We define our pattern mining problem as a CSP in the following way:

- The variables (nodes) represent the roles of a pattern.
- The variable domains are initialised to contain all the names (identifiers) of the diagram(s) in question.
- Unary constraints represent conditions for a single role (e.g. the element in role X must be an abstract class).
- Binary constraints represent conditions between two roles (e.g. the class in role X must be a subclass of the class in role Y).

For each pattern we have a result, i.e. the bindings that describe this particular pattern. The number of these bindings depends on the pattern in question. A binding is a pair $\{role, element\}$, where *role* is the name of the role and *element* is the diagram element that appears in that role, e.g. in the **Factory Method** pattern two of the roles are *Product* and *Creator* that are to be bound to two classes in the architecture diagram.

3.2. Constraining search space

Even with small variable domains, trying out all combinations takes too much time. In addition, most combinations would make no sense. In our case, this means that if we require that a certain variable can only have *class*-typed values, we can prune all attributes, methods etc. from its domain. Therefore we must find a way to effectively prune out impossible candidates, and for this we use the AC-3 algorithm [16].

The AC-3 algorithm is based on the concepts of *node* and *arc consistency*, which correspond to unary and binary constraints. For instance, node consistency means that all attribute entities would be pruned from the domain of a variable having a constraint that allows only solutions of type *class*. Arc consistency is defined in a similar fashion, but for a pair of nodes. We delete all values from the domain of the originating node, for which there are no legal arcs.

3.3. Applying AC-3 algorithm

The first and most trivial requirement is node consistency. Node i is *node consistent*, iff $\forall x \in D_i, P_i(x)$ holds. The following algorithm ensures node consistency.

```
procedure NC-1:
begin
    for i ← 1 until n do
      begin
          D_i ← {x ∈ D_i | P_i(x)}
      end
end
```

Thus, for example, all attribute-entities will be pruned by NC-1 from the domain of a variable having a constraint that allows only solutions of type *class*.

Arc consistency is defined in a similar fashion: Arc (i, j) is *arc consistent*, iff $\forall x \in D_i$ such that $P_i(x)$ holds, $\exists y \in D_j$ such that $P_j(y)$ and $P_{ij}(x, y)$. A more detailed discussion of arc consistency can be found in [17].

A single arc can be revised using the following procedure REVISE that returns a boolean value. The idea is similar to that behind node consistency. We delete all values from the domain of the originating node D_i, for which there are no 'legal' arcs:

```
procedure REVISE((i,j)):
begin
    DELETE ← false
        for each x ∈ D_i do
        if ∄y ∈ D_j such that P_ij(x, y), then
          begin
              delete x from D_i
              DELETE ← true
```

```
            end
        return DELETE
    end
```

The AC-3 algorithm first utilizes the node consistency algorithm and then the arc consistency revision algorithm as follows. We denote the entire CSP graph with G and the respective set of arcs (constraints) with $arcs(G)$. Additionally we denote the current (non-consistent) set of arcs with Q, which means that the algorithm halts as soon as Q is empty.

```
procedure AC-3:
begin
  NC-1
  Q ← {(i,j)|(i,j) ∈ arcs(G), i ≠ j}
  while Q not empty do
    begin
      select and delete any arc (k,m) from Q
      if REVISE((k,m)) then
        Q ← Q∪{(i,k)|(i,k) ∈ arcs(G), i ≠ k, i ≠ m}
    end
end
```

After the domains have been made consistent, we search for correct bindings among the remaining values that satisfy the current set of constraints. In the simple case we have only one value for each variable.

3.4. Extensions

Many design patterns are related to each other in the sense that they have common elements (see e.g. meta-patterns in [20]). These relationships may be taken advantage of in two ways: the ordered search of patterns and the use of auxiliary facts. The available information is updated as the search proceeds: when a particular pattern is being searched for, new facts pertaining to it are added. These new facts can be utilized later when searching for other patterns. Consider, for example, that we are searching for instances of the **Abstract Factory** pattern which has the **Factory Method** pattern as its prerequisite. We can make use of this relationship by dynamically adding new facts of type **FactoryMethod**, while searching for instances of the **Factory Method** pattern.

3.5. Example

The data in CASE tools has usually not been designed to be easily exported or even uniform, so the output of CASE tools and the requirements of Maisa are not directly compatible. Therefore we have defined an intermediate format for transferring the diagrams to Maisa. All UML diagrams are exported as a set of Prolog facts. These facts are then read into the Maisa tool. The same notation is also used for defining the patterns.

Maisa ignores any additional information such as comments that the Prolog files may contain. The format is very easy to extend as one only needs to add appropriate parser elements for Maisa to recognize the new patterns. The internal data structure for the diagrams is very close to UML structures. As can be seen later, we can also use reverse-engineered data in the same format.

The procedure itself is straightforward. The diagram is disassembled into components, and a fact (or a set of facts) is generated for each component. Other necessary information (such as the class a particular method belongs to) is included in these facts. For example, for each attribute in a class diagram we generate a fact that states the identifier and the name of that attribute as well as the class it belongs to.

For example, the **Singleton** [11] pattern is specified by the following facts:

```
class("Singleton").
attribute("Singleton.instance").
has("Singleton","Singleton.instance").
typeof("Singleton.instance","Singleton").
static("Singleton.instance").
```

The semantic intent of this pattern is to ensure that a class has only a single instance. This description states that a **Singleton** candidate (class) must have a static attribute whose type is the same as the class itself.

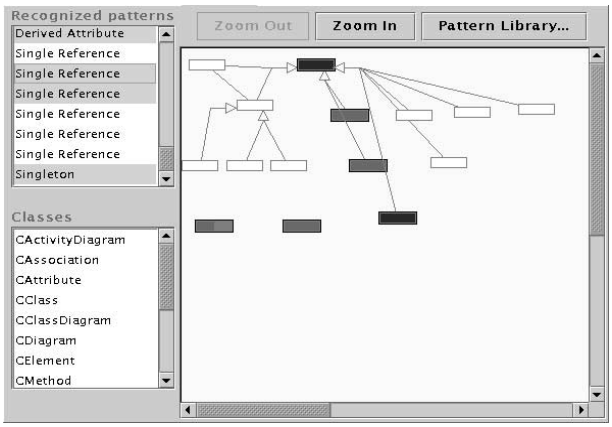

Figure 2. Design pattern visualization in Maisa

Figure 2 shows the pattern panel of Maisa. The right-hand side contains the class diagram that has been measured. The left-hand side contains two lists: the upper one shows the pattern instances found in the diagram and the

lower one shows the components of the diagram. When an item has been selected in either of the lists, the corresponding components are highlighted in the diagram. In this case, all the elements participating in any of the three selected pattern instances are highlighted. Regular metrics are shown in another panel (see Table 1).

4. Verifying Architectural Conformance

Columbus [9] is a reverse-engineering tool developed by the Research Group on Artificial Intelligence in the University of Szeged and the Software Technology Laboratory of Nokia Research Center. In addition to data extraction, Columbus supports a number of other related tasks including visualization. Columbus is based on plug-ins (e.g. for extracting or exporting), so the system is easily extendible.

4.1. Integration of Columbus and Maisa

The analysed project in Columbus is represented by an abstract semantic graph, which can be accessed through an application programming interface. Maisa, however, is implemented in Java, so the interface cannot be used directly, but an additional plug-in is required to transform the output of Columbus to the intermediate format used in Maisa.

Because this high-level information has been generated from source code, it is much more detailed than that generated from design diagrams. To take advantage of this, Maisa can use multiple pattern definitions (i.e. different sets of constraints). An association, for example, can be implemented in several ways. If we use reverse-engineered information, we know this implementation decision exactly. Thus, we may have more precise pattern definitions yielding more precise results.

4.2. Finding a pattern

If we want to search for the **Singleton** design pattern (see Chapter 3.5), we may use facts generated from a design diagram or we may analyse C++ code with Columbus.

A **Singleton** could be declared in C++ as:

```
class System {
public:
  static System* Instance();
protected:
  System() {};
private:
  static System* instance;
};
```

with the implementation:

```
System* System::instance = 0;

System* System::Instance() {
  if (instance==0) {
    instance=new System();
  }
  return instance;
}
```

Either way, we should obtain at least the following facts:

```
class("System").
method("System.Instance()").
public("System.Instance()").
static("System.Instance()").
has("System","System.Instance()").
returns("System.Instance()","System").
method("System.System()").
protected("System.System()").
has("System","System.System()").
attribute("System.instance").
private("System.instance").
static("System.instance").
has("System","System.instance").
typeof("System.instance","System").
```

When matching this description with the definition of the **Singleton** pattern, Maisa produces the following bindings, with the role on the left-hand side and the diagram (or code) element on the right-hand side:

```
Singleton.instance = System.instance
Singleton = System
```

This result also appears in the pattern panel (cf. Figure 2).

5. Statistical Architecture Evolution

The statistic diagrams can be drawn by Maisa either from a single set of metric results related to a certain version of an architecture or from a history data repository, where the calculated metric data has been explicitly saved. To support time series analysis, the repository contains as detailed metric results as possible, i.e. each metric calculated per each project, diagram and/or class when appropriate. The metric values are supplied with timestamps which express the calculation moment of the metric.

Maisa supports four statistic diagram types. They are tailored to the needs of architectural analysis.

Time series diagram presents the values of one metric with respect to time as a scatter plot, the time running

on the x axis. This diagram can only be drawn from repository data.

Distribution diagram presents the distribution of one metric as a histogram. The system does not use normal approximation but the histogram is constructed from a classified sample.

Correlation diagram presents the correlation of two metrics as a scatter plot.

Pattern distribution diagram presents the distribution of the recognized pattern instances as a histogram. The diagram is drawn from a nominal sample, one bar per pattern. This is the only diagram offered for non-numeric metrics.

When generating a diagram, the user can bound the sample data to all architectures or a certain architecture. When examining pattern instance count, the sample data can be bound further to instances of all patterns or a certain pattern. Thus, Maisa offers quite an extensive compilation of diagrams describing the architecture from different perspectives. The system also calculates the appropriate statistic key figures from the corresponding distribution.

Maisa is not dependent of any specific quality system but the quality of an architecture is indicated by a group of metrics, selected by the user. Let us study the example architecture (shown in Figure 2). Let us assume that it has been measured eight times with the results shown in Table 1.

NOC	NOMA	Patterns	Pattern instances	Relative NOC
7	3	0	0	0/7 = 0
7	21	1	10	6/7 = 0.857
13	47	2	30	12/13 = 0.923
15	50	2	32	15/15 = 1
15	53	3	40	15/15 = 1
15	45	2	35	13/15 = 0.867
15	49	2	38	14/15 = 0.933
15	57	3	40	15/15 = 1

Table 1. Metric results of the example architecture.

In Figure 3 we see the pattern coverage of the architecture in a time series diagram, expressed as relative NOC, the values varying from zero to one. The points in the diagram correspond to the metric calculations of one version of the architecture. The pattern coverage has risen during the design phase. The sixth measurement and the succeeding ones are from the code. The pattern coverage in the sixth measurement has dropped, because the pattern structure of the code has not been equivalent to the designed architecture. This indicates that the quality of the architecture has also dropped. After improving the code, the last measurement shows the code to fit the design, with the pattern-based quality now assured.

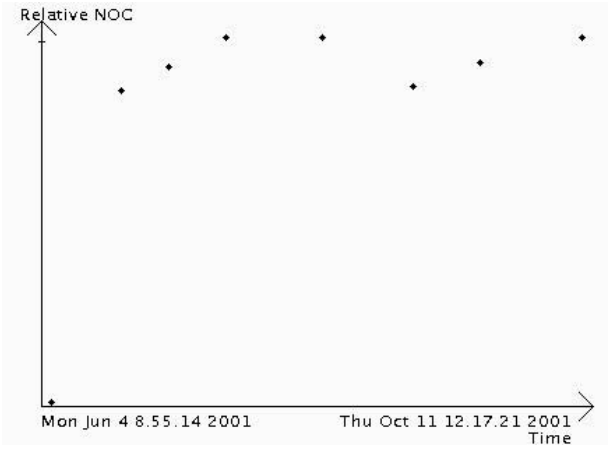

Figure 3. Pattern coverage of the sample architecture in time series.

All the above listed metrics describe the architecture from their own perspective, none of them indicating the alteration of the quality in full. In Figure 3, the pattern coverage seems to rise quite rapidly. However, there is only one class in the second and third version that is not covered by any pattern – very possibly the same class in both cases – but when the architecture contains fewer classes, each one weighs more. The same phenomenon can be seen in the sixth and seventh measurement with two and one non-pattern class respectively. The number of classes remains the same after the first three versions. The number of pattern instances rises between the fourth and fifth version, which cannot be seen in the diagram, because the pattern coverage remains the same. It must be noted that relative NOC is not the same metric as NOC per number of pattern instances. The former measures the percentage of classes that are covered with at least one pattern, whereas the latter does not indicate how smoothly the pattern instances are distributed among the classes. Thus, the pattern coverage is more descriptive than mere pattern count, pattern instance count or number of classes – though not a perfect one.

The formerly collected metric data can be used to reason the quality of the system under development. The repository can be searched for reference systems whose architectural metrics profile is similar to that of the current system (with one subprofile as in Figure 3). The quality of the best matching reference system can then be used as a predictor for the ultimate quality of the current system.

6. Related Work

Recent research on software measurement includes empirical studies about a large telecommunications system [5] and code decay [7]. Although there are metrics that are available early on (such as the number of events for a class, pointed out in [5]), the emphasis is still mostly on code-based evaluation.

There has also been some research on how well the traditional metrics estimate the quality of software and whether software estimation is even possible [15].

Research on mining design patterns [1] is increasing. One problem with the UML is that it lacks a precise way of modelling design patterns, or software architectures in general. As a result, some proposals for a more explicit support for design patterns in UML have been presented [10, 14].

Software metrics, design pattern mining, reverse-engineering, and statistical analysis have been independently studied quite actively, but the combination of all these as provided by Maisa (and Columbus) is a unique approach. So far, the experiences with this kind of an architecture-centric software evolution tool set are most promising.

7. Conclusions and Future Work

We have presented Maisa, a Java-based tool for evaluating software architectures. Besides supporting many traditional object-oriented metrics, Maisa is also capable of mining design patterns from a software architecture expressed as a project of UML diagrams. This can be utilized in numerous ways, the simplest of which is the visualization of design pattern instances.

While Maisa has originally been developed to help in software design, we have also presented some examples of how Maisa can be used cooperatively with other software tools (in our case the Columbus reverse-engineering tool) to keep track of the change history of a software product. To better support cooperative use, we are considering other exchange formats (such as the XMI) for Maisa.

Currently we are conducting further experiments with larger industrial systems as well as incorporating UML-based performance evaluation to Maisa.

Also, as part of our experiments, we are extending the pattern library with domain specific patterns. This is done in close cooperation with our industrial partners.

Maisa will become a part of a larger UML toolset, which will be developed in UML++, a collaborative research project of the universities of Helsinki and Tampere, and the Tampere University of Technology.

Acknowledgements

The Maisa development project is funded by the National Technology Agency of Finland and by four IT companies: Kone, Nokia Mobile Phones, Nokia Research Center, and Space Systems Finland. The integration of Maisa and Columbus has been implemented by Rudolf Ferenc and László Müller from the University of Szeged. The pattern visualization facility in Maisa has been designed and implemented by Minna Majuri and the pattern mining facility has been implemented by Pauli Misikangas.

References

[1] F. Bergenti and A. Poggi. IDEA: A design assistant based on automatic design pattern detection. In *Proceedings of 12th International Conference on Software Engineering and Knowledge Engineering (SEKE 2000)*, pages 336 – 343, 2000.

[2] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[3] W. Brown, R. Malveau, H. McCormick III, and T. Mowbray. *AntiPatterns — Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons Inc., 1998.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture : A System of Patterns*. John Wiley & Sons Inc., 1996.

[5] M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering*, 26(8):786 – 796, 2000.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476 – 493, 1994.

[7] S. Eick, T. Graves, A. Karr, J. Marron, and A.Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1 – 12, 2001.

[8] R. Ferenc, J. Gustafsson, L. Müller, and J. Paakki. Recognizing design patterns in C++ programs with the integration of Columbus and Maisa. In *Proceedings of the 7th Symposium on Programming Languages and Software Tools (SPLST'2001)*, pages 58 – 70, 2001.

[9] R. Ferenc, F. Magyar, Á. Beszédes, G. Márton, M. Tarkiainen, and T. Gyimóthy. Columbus 2.0 — tool for reverse engineering large object oriented software systems. Technical Report TR-2000-002, University of Szeged, 2000.

[10] M. Fontoura and C.J.P.de Lucena. Extending UML to improve the representation of design patterns. *JOOP*, 13(11):12 – 19, 2001.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] V. Gibson and J. Senn. System structure and software maintenance performance. *CACM*, 32(3):347 – 358, 1989.

[13] V. Kumar. Algorithms for constraint-satisfaction problems. *AI Magazine*, 13(1):32 – 44, 1992.

[14] A. Le Guennec, G. Sunyé, and J.-M. Jézéquel. Precise modeling of design patterns. In *Proceedings of ≪UML≫ 2000. Third International Conference*, pages 482 – 496, 2000.

[15] J. Lewis. Limits to software estimation. *ACM Software Engineering Notes*, 26(4):54 – 59, 2001.

[16] A. Mackworth. Consistency in network of relations. *Artificial Intelligence*, 8(1):99 – 118, 1977.

[17] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225 – 233, 1986.

[18] *OMG Unified Modeling Language Specification Version 1.3*. ©1999 Object Management Group, Inc.

[19] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A. I. Verkamo. Software metrics by architectural pattern mining. In *Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, pages 325 – 332, 2000.

[20] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[21] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[22] A. Verkamo, J. Gustafsson, L. Nenonen, and J. Paakki. Design patterns in performance prediction. In *Proceedings of the Second International Workshop on Software and Performance (WOSP 2000)*, pages 143 – 144, 2000.