Matrix Based Problem Detection in the Application of Software Process Patterns.

Conference Paper · January 2007

Source: DBLP

CITATIONS
2

READS
23

2 authors:

Chintan Amrit
University of Twente
67 PUBLICATIONS **303** CITATIONS
SEE PROFILE

Jos van Hillegersberg
University of Twente
159 PUBLICATIONS **2,170** CITATIONS
SEE PROFILE

Detection of Problems in the Application of Software Process Patterns

Chintan Amrit & Jos van Hillegersberg

Department of IS & CM, BBT,
University of Twente, P.O. Box 217,7500 AE Enschede
{c.amrit, J.vanHillegersberg}@utwente.nl

Abstract. Software development is rarely an individual effort and generally involves teams of developers. In distributed and collocated teams we often find problems in the organizational process structures. Though process patterns have been around for many years, there has been little research in the area of quantifying the problem structures in order to effectively apply the patterns. We propose a methodology using adjacency boolean matrices which identifies problems in the software development process.

1 Introduction

Software Development projects often prove to be both a costly and risky endeavor. Poor software project execution continues to result, in the best cases, in missed deadlines, and in the worst cases, in escalations in commitment of additional resources as a cure-all for runaway projects [1].

When we treat the root causes of problems we not only eliminate the symptoms, but we also are in a much better position to develop and maintain quality software in a repeatable predictable fashion. To enable this, the industry has what are called software best practices. These best practices are commercially proven approaches to strike at the root of the software development problems [2]. Some examples of best practices are:

- Develop software iteratively.
- Manage requirements.
- Use component-based architectures.
- Visually model software.
- Verify software quality.
- Control changes to software.

Most of the time, during software development it is observed that only the knowledge of the best practices is not enough to guarantee successful completion of software projects. The problem with the usage of best practices as generic solutions is that they are not precisely quantified and are not solutions to specific problems one encounters during software development. Experienced software designers and developers try to reuse solutions which have worked in the past rather

than solve every problem from first principles. This methodology has led to the use of software patterns, which are proven solutions to recurrent software development problems. These patterns are mostly applied at the design stage of the product and not in the actual implementation part of the software development. Further, there has been no work done in quantifying the problematic areas which the patterns address. In this paper we focus on patterns related to software development in distributed or collocated teams and demonstrate a methodology to suggest the problem areas in the organizational process structure. This, we think would make it easier to implement these process patterns.

1.1 Patterns

While there are many ways to describe a patterns, Christopher Alexander who originated the notion of patterns in the field of building architecture described patterns as a recurring solution to a common problem in a given context and system of forces [3]. In Software Engineering patterns are attempts to describe successful solutions to common software problems [4]. Software Patterns reflect common conceptual structures of these solutions and can be used repeatedly when analyzing, designing and producing applications in a particular context. Patterns represent the knowledge and experience that underlie many redesign and re-engineering efforts of developers who have struggled to achieve greater reuse and flexibility of their software. The different types of patterns are:

- Design Patterns: Are simple and elegant solutions to specific problems in object oriented design [5].
- Analysis Patterns: Capture conceptual models in an application domain in order to allow reuse across applications [6].
- Organizational Patterns: Describe the structure and practices of human organizations [7].
- Process Patterns:Describe the Software Design Process.

The basic format of a pattern was devised by the "Gang of Four" [5], this consists of

1. Pattern Name: Is a handle we can use to describe a design problem, its solutions and consequences in a word or two.
2. Problem: Describes when and under which context one should apply the pattern.
3. Solution: Describes the elements that make up the design, their relationships, responsibilities and collaborations.
4. Consequences: Are the results and trade offs of applying the pattern.

Patterns are most generally represented in natural language and are typically published in printed catalogues. Pattern presentation is generally loosely structured and consists of a series of fields each having a meaning introduced via an informal definition or description. An example of such a structure is what is proposed in [5]. A fragment of this is illustrated in the Table 1.

Field	Explanation/Definition
Name	Ideally a meaningful name that will be part of the shared design vocabulary. Many existing patterns do not satisfy this requirement for historical reasons.
Also known as	Other names of pattern.
Intent	a short specification or rationale of the pattern, used as a principal index for goal oriented pattern search
Applicability	An outline of the circumstances in which the pattern may be applicable and, perhaps more importantly, when it should not be applied.
Structure	A diagrammatic representation of the pattern
Consequences	Discusses the context resulting from applying the pattern. In particular, trade-offs should be mentioned
Implementation	Advices on how to implement the patterns, and other language specific issues. The implementation will depend on the abstractions (objects, parameterized types,. . .) supported by the target language.
Known uses	Patterns are by essence derived from existing systems. It is therefore important that they be justified by their use in several real systems.
Related patterns	Patterns are often coupled or composed with other patterns, leading to the concept of pattern language; e.g. a visitor may be used to apply an operation to the closed structure provided by a composite.

Table 1. Fragment of a canonical form of pattern representation.

Identifying the problem areas related to process patterns [8,7] can prove difficult for large distributed or collocated teams working on large software projects. The way to solve this problem is to automate the process of detecting problems related to process patterns. One needs a proper formalism of problem scenarios in order to automate the process of problem detection and then leave it the manager's discretion, whether the particular pattern has to be applied. However, there has been little research done in the area of formalisms of such problem scenarios.
In this position paper we focus on the two patterns related to software development in distributed or collocated teams and demonstrate a technique that suggests the problem areas in the organizational process structure. This, we think would make it easier to implement these process patterns.
The rest of the paper is organized as follows, Section 2 describes the construction process of the process dependency matrices. Section 3 describes the matrix formulation with the developing in pairs pattern. Section 4 describes the matrix formulation with the Conway's law pattern. Concluding remarks are given in Section 5.

2 Construction of the Process Dependency Matrices

Dependency matrices have been used in Engineering literature to represent the dependency between people and tasks. Li et al. [9] use dependency matrices to analyze dependencies between components in a CBS. Here we create what we

call *process dependency matrices* in order to represent the connections between software modules, software developers as well as the modules each developer is working on. In the notation used here $A[i,j]$ represents a matrix, while a_{ij} represents the $(i,j)^{th}$ element of the matrix. From the CVS (Concurrent Versioning System) we can obtain two kinds of adjacency matrices. One is the $m*m$ adjacency matrix $SM[i,j]$ representing the software dependency graph [10](assuming there are m modules) and the other an $m*n$ adjacency matrix $SMA[i,j]$ representing the allocation of modules to the developers(assuming there are n developers working on the m modules). The software dependency matrix would represent:

- Function call dependency
- Inheritance dependency
- Aggregation dependency

The software dependency matrix would appear as:

$$SM[i,j] = \begin{bmatrix} sm_{11} & sm_{12} & \cdots & sm_{1m} \\ sm_{21} & sm_{22} & \cdots & sm_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ sm_{m1} & sm_{m1} & \cdots & sm_{mm} \end{bmatrix}$$

Which can be concisely represented as:

$$(sm_{ij}) = \begin{cases} 1 \; if \quad c_i \rightarrow c_j; \\ 0 \quad otherwise \end{cases}$$

Where c_i, c_j are software modules, whereas each sm_{ij} represents the relation between the modules as described above. Similarly, the $n*m$ Software Module Allocation adjacency matrix would appear as:

$$SMA[i,j] = \begin{bmatrix} sma_{11} & sma_{12} & \cdots & sma_{1m} \\ sma_{21} & sma_{22} & \cdots & sma_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ sma_{n1} & sma_{n1} & \cdots & sma_{nm} \end{bmatrix}$$

Where the rows represent the software developers working on modules which are represented along the columns. This matrix can be concisely represented as:

$$(sma_{ij}) = \begin{cases} 1 \; if \quad sd_i \rightarrow c_j; \\ 0 \quad otherwise \end{cases}$$

Where each sd_i represents a software developer, while each c_j represents a software module. Hence, we see that each sma_{ij} represents the relation between them , which is in this case whether the particular developer sd_i is developing the module c_j in question. The symmetric $n*n$ matrix $SND[i,j]$ representing

the work related communication network of the developers can be represented as:

$$(snd_{ij}) = \begin{cases} 1 \; if \; sd_i \rightarrow sd_j; \\ 0 \quad otherwise \end{cases}$$

That is each element snd_{ij} of the matrix $SND[i,j]$has a value of 1 if the i^{th} person is talking to the j^{th} person, where each row of the $SND[i,j]$ matrix corresponds to the same developers as the rows of the $SMA[i,j]$ matrix. We now utilize these matrices in order to describe the problem scenarios of two process patterns.

3 Developing in Pairs, Pattern

This pattern deals with pairing compatible programmers together so that they can produce more together than they can working individually [8]. Also there has been research claiming that pair programming produces better products in less time [11].Further, its better for the software product when all its modules has two developers working on it, as, when one of them leaves the company the other has an idea of what is to be done.
In our matrix based technique, we create a $n * t$ matrix $(PP[i,j])$ which has all the software modules in the matrix $(SMA[i,j])$ developed by at the most one programmer. Hence, if $t = 0$ then it means all the software modules are developed by two or more programmers. The matrix $(PP[i,j])$ gives an indication of the possible problematic software modules; those with only one programmer working on it as well as those with no one currently responsible. We can express the $(PP[i,j])$ matrix as follows:

$$(pp_{ij}) = \left(sma_{ij} \; iff \; \sum_{k=1}^{n} sma_{kj} < 2, \; \forall 0 \leq j \leq m \right)$$

The algorithm used to create $(PP[i,j])$ is:

```
function Cal_PairPattern(bool[1..n,1..m] SMA) {
    var bool[1..m] PPColumn
    var int moduleSum = 0
    var int columnCount = 0
    var int columnC = 0

    //Finding the columns that add up to less than 2
    for j = 1 to m
    {
      moduleSum = 0
      for k = 1 to n
        moduleSum  = moduleSum + SMA[k,j]
      if(moduleSum < 2)
```

```
      {
       PPColumn[j] = 1
       columnCount = columnCount + 1
      }
      else
       PPColumn[j] = 0
    }
    create var bool [1..n, 1..columnCount] PP

    //Assigning values to the PP matrix

    for j = 1 to m
      if(PPColumn[j] > 0)
      {
        columnC = columnC + 1
        for i = 1 to n
          PP[i, columnC] = SMA[i,j]
      }
    // Display matrix PP
    Display(PP)
    print columnCount

    return
}
```

4 Conway's Law Pattern

"Organizations which design systems (...) are constrained to produce designs which are copies of the communication structures of these organizations" [12]
This pattern states that the structure of the system mirrors the structure of the organization that designed it. The shaping forces behind this law are that, architecture shapes the communication paths in an organization and that formal organization structure shapes architecture [8,13]. Another way of looking at Conway's law is saying that dependencies between software modules cause dependencies between the developers developing them. The dependencies between the code modules maybe inheritance, aggregation or simple function calls from one module to another. These dependencies create further dependencies among the programmers who work on the particular modules [12].

We use adjacency matrices to suggest a dependency metric which can be used to measure dependencies in allocations of software modules.

4.1 Dependency Matrix Construction

For the sake of this research we consider two kinds of dependencies in the development of software:

1. Developers working on the same modules.
2. Developers working on modules which are mutually dependent themselves.

We propose an algorithm for the construction of a $n * n$ adjacency matrix representing the dependency between Software Developers based on the following:

$$(sdd_{ij}) = \begin{cases} 1 \; if \; sma_{ik} \wedge sma_{jk} = 1, & \forall \; 0 \leq k \leq m \\ 1 \; if \; sma_{ik} \wedge sma_{jl} \wedge sm_{kl} = 1, & \forall \; 0 \leq k \leq m, 0 \leq l \leq m; \\ 0 \quad otherwise \end{cases}$$

In the above construction of the software developer dependency matrix, the value of 1 is assigned whenever there are more than one developer working on the same software module. Also a 1 is assigned whenever developers work on modules which share a dependency.

The logical difference between the software developer dependency matrix ($SDD[i,j]$) and the matrix representing the work related communication network of the software developers ($SND[i,j]$) would give us a $n*n$ matrix ($DDM[i,j]$)which represents the unresolved dependencies in the existing communication network of developers. This can be represented as:

$$(ddm_{ij}) = \left(sdd_{ij} \wedge \overline{snd_{ij}}\right)$$

The summation of the logical difference between the software developer dependency matrix ($SDD[i,j]$) and the matrix representing the work related social network of the software developers ($SND[i,j]$) would give us a metric which suggests the requirement of greater communication among the software developers. As we are only considering undirected networks the matrices ($SDD[i,j]$) and ($SND[i,j]$) are symmetrical. So, the summation should be divided by 2 to get the actual number of non-existent dependencies. This can be described as a dependency metric as follows:

$$depdendency\_metric = 1/2 \sum_{i=1,j=1}^{n} (sdd_{ij} \wedge \overline{snd_{ij}})$$

Using these two matrices we construct the dependency matrix of the developers using the following algorithm:

```
function Cal_DependencyMetric(bool[1..n,1..m] SMA,
                              bool[1..m,1..m]SM)
{
    var bool[1..n,1..n] SDD
    var bool[1..n,1..n] DDM
    var int dependency_metric = 0

    //Dependencies based on people working on the same code module
```

```
    for i = 1 to n-1
     for j = i+1 to n                    //We are not interested in self dependencies
      for k = 1 to m
        SDD[i,j] = SMA[i,k] AND SMA[j,k]

    //Dependencies based on people working on dependent code
    // modules
    for i = 1 to n-1
     for j = i+1 to n
      for k = 1 to m
       for l = 1 to m
         if(!SDD[i,j])                   //We don't want to lose existing dependencies
           SDD[i,j] = SMA[i,k] AND SMA[j,l] AND SM[k,l]

    //Calculating Dependency Metric
    for i = 1 to n
     for j = 1 to n
     {
       dependency_metric = dependency_metric + 1/2 ( SDD[i,j] AND (NOT(SND[i,j])) )
       DDM[i,j] = SDD[i,j] AND (NOT(SND[i,j]))
     }
    //Display the matrix DDM
    Display(DDM)
    print dependency_metric

    return
}
```

To calculate the transitive dependencies we can use Warshall's algorithm [14] (Appendix A) for transitive closure of the matrix ($sdd_{i,j}$). This dependency metric varies with time, as do all the matrices represented. We could also plot this metric and its changes with time to get a better idea of how the project evolves.

5 Gatekeeper Pattern

5.1 Gatekeeper Pattern for co-located team

The Gatekeeper pattern (Pattern No. 23 [8]) basically says that one needs to balance communication with typically introverted engineering types. The reason being that isolationism doesn't work and information flow is important, but the problem is communication overhead goes up linearly with the number of external collaborators, hence what the team manager should strive for is moderate communication between all the employees [15]. Here, we identify the problem areas as the non -communicating introverted engineers. In order to identify what are known as isolates in social networks we have lots of clustering techniques available in social network literature.

5.2 Gatekeeper Pattern for a globally distributed team

The gatekeeper pattern for a distributed team is in slightly

6 Conclusion

In this position paper we have tried to demonstrate a technique for quantifying problems related to software processes in organizations. Once these problems have been identified, we can leave it to the discretion of the responsible manager to apply the particular process pattern related to the problem in hand. We have demonstrated this technique with two particular problems related to assignment of software modules to developers. The first problem is related to pair programming, where there might be no developer or just one developer involved in developing what might be an important software module. If this particular developer leaves the company then there could be delays and unwanted associated costs.
The second pattern deals with dependencies in software modules which create an intrinsic dependency between programmers working on it. Though past research in CSCW has focussed on these dependencies [16], they haven't described any technique to effectively identify the problem that they are trying to solve. We have described a technique to identify these dependencies, as well as a metric to calculate to measure the extent of the problem, before trying to resolve the dependencies. We are currently working on a tool which implements these algorithms described. Future work could involve quantifying other process patterns and making them more easily applicable.

References

1. Kraut, R.E., Streeter, L.A.: Coordination in software development. Commun. ACM **38**(3) (1995) 69–81
2. Kruchten, P.: The Rational Unified Process, An Introduction. Addison Wesley, Massachusetts (1998)
3. Alexander, C., Ishikawa, S., Silverstein: A Pattern Language. Oxford University Press, New York (1977)
4. Schmidt, D., Fayad, M., Johnson, R.: Software patterns. Communication of the ACM **39** (1996) 37–39
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patters Elements of Resuable Object Oriented Software. Addison Wesley, MA (1995)
6. Fowler, M.: Analysis Patterns: Reusable Object Models. Addison Wesley, Reading MA (1997)
7. Coplien, J.O., Harrison, N.B.: Organizational Patterns of Agile Software Development. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (2004)
8. Coplien, J.O., Schmidt, D.C., eds.: Pattern languages of program design. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1995)
9. Li, B., Zhou, Y., Wang, Y., Mo, J.: Matrix-based component dependence representation and its applications in software quality assurance. SIGPLAN Not. **40**(11) (2005) 29–36

10. Myers, C.R.: Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. Physical Review E (Statistical, Nonlinear, and Soft Matter Physics) **68**(4) (2003) 046116
11. Williams, L., Kessler, R.R., Cunningham, W., Jeffries, R.: Strengthening the case for pair programming. IEEE Softw. **17**(4) (2000) 19–25
12. Conway, M.: How do committees invent. Datamation **14** (1968) 28–31
13. Herbsleb, J.D., Grinter, R.E.: Splitting the organization and integrating the code: Conway's law revisited. In: ICSE '99: Proceedings of the 21st international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 85–95
14. Warshall, S.: A theorem on boolean matrices. J. ACM **9**(1) (1962) 11–12
15. Carroll, T., Burton, R.M.: Organizations and complexity: Searching for the edge of chaos. Comput. Math. Organ. Theory **6**(4) (2000) 319–337
16. Souza, C.R.B.D., Redmiles, D., Cheng, L.T., Millen, D., Patterson, J.: Sometimes you need to see through walls: a field study of application programming interfaces. In: CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work, New York, NY, USA, ACM Press (2004) 63–71

APPENDIX

A Warshall's Algorithm

Warshall's Algorithm of transitive closure. Given directed graph $G = (V, E)$, represented by an adjacency matrix $A[i, j]$, where $A[i, j] = 1$ if (i, j) is in E, compute the matrix P, where $P[i, j]$ is 1 if there is a length greater than or equal to 1 from i to j.

```
Warshall(int N, bool[1..n,1..n] A, bool[1..n,1..n] P) {
 int i,j,k;

 for(i=0; i<N;i++)
  for(j=0; j<N; j++)
   P[i,j]=A[i,j]

 for(k=0; k<N; k++)
   for(i=0;i<N; i++)
     for(j=0; j<N; j++)
       if(!P[i,j]) P[i,j]=P[i,k]&&P[k,j];
}
```