MATRIX BASED PROBLEM DETECTION IN THE APPLICATION OF SOFTWARE PROCESS PATTERNS

Chintan Amrit and Jos Van Hillegersberg
Department of IS & CM, University of Twente, Enschede, The Netherlands

Keywords: Software Engineering, Social Factors, Process Patterns.

Abstract: Software development is rarely an individual effort and generally involves teams of developers. Such collaborations require proper communication and regular coordination among the team members. In addition, coordination is required to sort out problems due to technical dependencies that exist when components of one part of the architecture requires services or data input from components of another part of the architecture. The dynamic allocation of the different tasks to people results in various socio-technical structure clashes (STSCs). These STSCs become more pronounced in an Agile Software Development environment and managerial intervention is constantly required to alleviate problems due to STSCs. In this paper we propose a technique based on dependency matrices that detects STSCs in the organizational process structure. We illustrate this technique using two examples from Organizational and Process Pattern literature.

1 INTRODUCTION

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 (Alexander et al., 1977). In Software Engineering patterns are attempts to describe successful solutions to common software problems (Schmidt et al., 1996). 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 reengineering 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 (Gamma et al., 1995).
- Analysis Patterns: Capture conceptual models in an application domain in order to allow reuse across applications (Fowler, 1997).
- Organizational Patterns: Describe the structure and practices of human organizations (Coplien and Harrison, 2004).
- Process Patterns: Describe the Software Design Process (Coplien and Schmidt, 1995).

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 representing patterns can be found in (Gamma et al., 1995).

Identifying the problem areas related to process patterns (Coplien and Schmidt, 1995; Coplien and Harrison, 2004) (which we would refer to as *Socio-Technical Structure Clashes* or *STSC* henceforth) can prove difficult for large distributed or collocated teams working on large software projects. The reason why the patterns are hard to implement is that the problems, or STSCs in our case addressed by the patterns are hard to detect, as purely manual techniques are labour intensive. We contend that by automating the process of STSC detection we can help monitor a large software development process. Process moni-

Amrit C. and Van Hillegersberg J. (2007).
MATRIX BASED PROBLEM DETECTION IN THE APPLICATION OF SOFTWARE PROCESS PATTERNS.
In *Proceedings of the Ninth International Conference on Enterprise Information Systems - ISAS*, pages 316-320
DOI: 10.5220/0002395503160320
Copyright © SciTePress

toring becomes essential in such an agile environment in order to keep the development process in check. Detection of such STSCs can help in reengineering the informal design process model in order to improve project planning (Fig. 1).

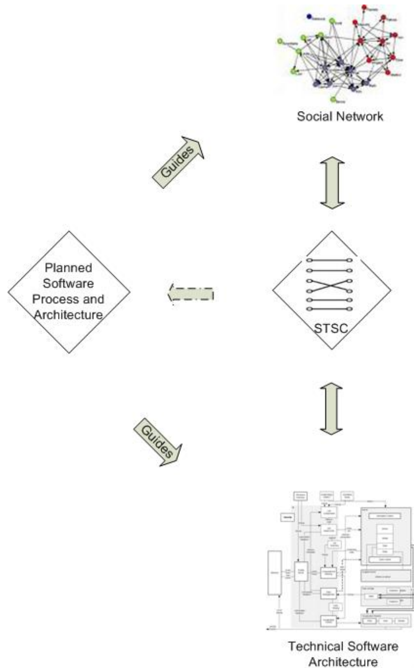

Figure 1: Planned Software Process refactoring using Socio-Technical Structure Clashes.

One needs a proper formalism of STSC 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 (STSCs).

In this paper we propose a technique that detects STSCs and a metric to gauge the extent of the STSC. In order to illustrate this technique we apply it to two patterns related to software development in distributed or collocated teams.

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 pair's pattern, Section 4 describes the matrix formulation with the Conway's law pattern and 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 (Steven et al., 1994). Recent empirical work uses DSMs to provide critical insights into the relationship between product architecture and organizational structure. For example, Sosa et al (Sosa et al., 2004) find a strong tendency for design interactions and team interactions to be aligned, and show instances of misalignment are more likely to occur across organizational and system boundaries. Li et al. (Li et al., 2005) use dependency matrices to analyze dependencies between components in a CBS (Component Based System). Sullivan et al (Sullivan et al., 2001) use DSMs to formally model (and value) the concept of information hiding, the principle proposed by Parnas to divide designs into modules (Parnas, 1972). 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. Only the modified modules are considered at any point of time, as these are the modules which require communication to resolve dependency issues, especially in an agile environment (Wagstrom and Herbsleb, 2006; Cataldo et al., 2006). 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 of the modified modules (Myers, 2003)(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:

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

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:

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

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:

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

That is each element snd_{ij} of the matrix $SND[i,j]$ has a value of 1 if the ith person is talking to the jth 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 PAIR'S PATTERN

This pattern deals with pairing compatible programmers together so that they can produce more together than they can working individually (Coplien and Schmidt, 1995). Also there has been research claiming that pair programming produces better products in less time (Williams et al., 2000). Further, it is 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 in eq (1).

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

4 CONWAYS LAW PATTERN

"Organizations which design systems (...) are constrained to produce designs which are copies of the communication structures of these organizations" (Conway, 1968)

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 (Coplien and Schmidt, 1995; Herbsleb and Grinter, 1999). 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 (Conway, 1968).

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

4.1 Conways Law 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 eq (2).

$$(sdd_{ij}) = \begin{cases} 1 \quad if \quad sma_{ik} \wedge sma_{jk} = 1, \\ \qquad\quad \forall 1 \leq k \leq m; \\ 1 \quad if \quad sma_{ik} \wedge sma_{jl} \wedge sm_{kl} = 1, \\ \qquad\quad \forall 1 \leq k \leq m, 1 \leq l \leq m; \\ 0 \qquad otherwise \end{cases} \tag{2}$$

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 in eq (3).

$$(ddm_{ij}) = \left(sdd_{ij} \wedge \overline{snd_{ij}}\right) \tag{3}$$

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 in eq (4).

$$depdendency_metric = 1/2 \sum_{i=1,j=1}^{n} (sdd_{ij} \wedge \overline{snd_{ij}}) \tag{4}$$

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

To calculate the transitive dependencies we can use Warshall's algorithm (Warshall, 1962) (see Appendix) for transitive closure of the matrix ($SDD[i,j]$).

5 SPECIFYING PATTERN SOLUTIONS

5.1 Gatekeeper Pattern

The Gatekeeper pattern (Pattern No. 23 (Coplien, 1994)) basically says that one needs to balance communication with typically introverted engineering types. This role implies that the person disseminates leading and fringe information from outside the project to project members. This role of being a Gate Keeper is similar to being a coordinating member of the team. The measure of the degree to which a team member is a Gate Keeper is similar to measuring the coordination ability of the person (Scott, 2000). Betweenness refers to the frequency with which a node falls between pairs of other nodes in the network.

6 CONCLUSION

In this paper we propose a technique for formalizing problems related to *Socio-Technical Structure Clashes* or *STSCs* in organizations. Once these STSCs 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 STSC 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 STSC deals with dependencies in software modules which create an intrinsic dependency between programmers working on it. The assumption in this STSC is that all the dependencies between software modules incur the same cost. We could also use the concept of Clustered Cost (MacCormack et al., ming) in order to attribute a different cost to the dependency between developers due to modules that are more closely clustered as compared to other modules. Though past research in CSCW has focussed on these dependencies (de Souza et al., 2004), 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. 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. We have developed a tool called PatoNet which implements some the algorithms described in this paper. Future work could involve formalizing the problem structure of other process patterns and thereby enabling an easier pattern application.

REFERENCES

Alexander, C., Ishikawa, S., and Silverstein, L. A. (1977). *A Pattern Language*. Oxford University Press, New York.

Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., and Carley, K. M. (2006). Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 353–362, New York, NY, USA. ACM Press.

Conway, M. (1968). How do committees invent. *Datamation*, 14:28–31.

Coplien, J. O. and Schmidt, D. C. (1995). *Pattern languages of program design*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Coplien, James, O. (1994). A development process generative pattern language. pages 1–33.

Coplien, James, O. and Harrison, Neil, B. (2004). *Organizational Patterns of Agile Software Development*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

de Souza, Cleidson, R. B., Redmiles, D., Cheng, L.-T., Millen, D., and Patterson, J. (2004). 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*, pages 63–71, New York, NY, USA. ACM Press.

Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Addison Wesley, Reading MA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Resuable Object Oriented Software*. Addison Wesley, MA.

Herbsleb, James, D. and Grinter, Rebecca, E. (1999). Architectures, coordination, and distance: Conway's law and beyond. *IEEE Softw.*, 16(5):63–70.

Li, B., Zhou, Y., Wang, Y., and Mo, J. (2005). Matrix-based component dependence representation and its applications in software quality assurance. *SIGPLAN Not.*, 40(11):29–36.

MacCormack, A., Rusnak, J., and Baldwin, C. (forthcoming). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*.

Myers, Christopher, R. (2003). Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 68(4):046116.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

Schmidt, D., Fayad, M., and Johnson, R. E. (1996). Software patterns. *Communication of the ACM*, 39:37–39.

Scott, J. (2000). *Social Network Analysis: a handbook*. Sage Publications Inc.

Sosa, M. E., Eppinger, S. D., and Rowles, C. M. (2004). The misalignment of product architecture and organizational structure in complex product development. *J Manage. Sci.*, 50(12):1674–1689.

Steven, D. E., Daniel, E. W., Robert, P. S., and David, A. G. (1994). A model-based method for organizing tasks in product development. *Research in Engineering Design*, V6(1):1–13. 10.1007/BF01588087.

Sullivan, K. J., Griswold, W. G., Cai, Y., and Hallen, B. (2001). The structure and value of modularity in software design. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108, New York, NY, USA. ACM Press.

Wagstrom, P. and Herbsleb, James, D. (2006). Dependency forecasting in the distributed agile organization. *Commun. ACM*, 49(10):55–56.

Warshall, S. (1962). A theorem on boolean matrices. *J. ACM*, 9(1):11–12.

Williams, L., Kessler, Robert, R., Cunningham, W., and Jeffries, R. (2000). Strengthening the case for pair programming. *IEEE Softw.*, 17(4):19–25.

APPENDIX

WARSHALL'S ALGORITHM

Warshalls 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];
}
```