Presented at the STC (Socio-Technical Congruence) Workshop at ICSE2008, Leipzig, Germany, May, 2008.

Software Development as Activities Creating and Utilizing Socio-Technical Information Spaces

Yunwen Ye[1,2]
[1]Dept. of Computer Science
University of Colorado
Boulder, CO80309, USA
yunwen@colorado.edu

Kumiyo Nakakoji[2,3]
[2]SRA Key Technology Lab
3-12 Yotsuya, Shinjuku
Tokyo, 160-0004, Japan
kumiyo@kid.rcast.u-tokyo.ac.jp

Yasuhiro Yamamoto[3]
[3]KID Laboratory, RCAST
U. of Tokyo, 4-6-1 Komaba, Meguro,
Tokyo, 153, Japan
yxy@kid.rcast.u-tokyo.ac.jp

ABSTRACT
Software development is a process of gathering and creating information; it requires programmers to uncover the various parts that are related to their current task. We propose to conceptualize a software system being developed as a socio-technical information space that has multiple layers of links that relate different units of information resources that include code, documents and programmers. This conceptualization can lead to the creation of better tools that support the exploration of various latent relations to identify relevant resources that cannot be easily achieved by technical or social analysis alone.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – *computer-aided software engineering, user interfaces.* D.2.10 [**Software Engineering**]: Design – *representation.*

General Terms
Design, Human Factors

Keywords
Socio-technical information space

1. INTRODUCTION
With the increasingly widely accepted view of software systems as evolving entities, the percentage of incremental, continuous development tasks in software development has risen quickly. Several factors fueled the rapid increase of incremental and evolutionary construction of software systems. First, the turnover rate of the software industry has skyrocketed and many projects have to hire new developers in the middle of system construction and those new hires have to work within an existing software system without knowledge of its prior history. Second, the wide spread of the agile methodology has turned many software projects into cycles of continuous development by adding features piecemeal. Third, many Web-based software systems, such as social networking systems and e-commerce systems, start with a rough understanding of its specification and have to evolve the design and development of system while such systems are being used by many users.

Because continuous development tasks add new functionality or features by changing an existing software system, programmers need to understand the existing systems and determine where changes or new development should be made. Given the size and complexity of most software systems, it is impractical and unnecessary to gain a full understanding of the whole system. The key is to gain the understanding of the *parts* of the system that bear relevance to the current task, which is called a *task context* in [5] and a *working set* in [6]. Task contexts do not exist a priori; they emerge as developers explore the software system and determine the relevancy based on their understanding of the task and the system structure. Decades of research efforts have made huge progresses in development methodologies that strive to isolate changes to local modules, but many changes are still scattered among the system. One study has shown that programmers spend 60-90% of their times to pinpoint relevant source code through reading and navigating [4].

A programming context exists in terms not only of the source code but also of related documents as well as those programmers who worked on the parts. It has been observed that much of system knowledge was retained in the head of programmers [10]. Peer programmers are also important information resources for system comprehension, and should be utilized to help pinpoint relevant source code. Treating source code, documents and programmers as equally important resources for the information needs of software development, this paper proposes to conceptualize a software system being created as an evolving *socio-technical information space* (STIS) that consists of three kinds of information resource nodes (code, documents, and programmers) that have triangulated relations. Under this conceptualization, each software development activity adds new information resource nodes to the STIS of the software system being constructed, adds new relations between information resource nodes either explicitly or implicitly, or both. At the same time, such development activities can utilize previous relations in the STIS through spread activation along the triangulated relations to explore the system and identify *latent relevance* among code parts that cannot be determined easily by either structural dependency or conceptual similarity along.

2. LAYERS OF RELATIONS AMONG INFORMATION RESOURCE NODES
The STIS of a software system consists of three kinds of information resource nodes: *Programmer*, *Documents, Code.* The entity *Programmer* refers to all developers who have participated

in the development of the system. The entity *Document* includes the traditional design documents, test plans, test cases, as well as emails in associated mailing lists for the system development, version repository commit logs (e.g. CVS) and bug report records (e.g. GNAT) that represent the domain context under which the system has evolved. The entity *Code* refers to various levels of granularity of the source code of the software system.

The STIS of a software system also includes triangulated relations among *Programmer*, *Document*, and *Code*. Such triangulated relations can help programmers identify the information resources that are relevant to their task at hand, such as semantically related code segments, design rationales that provide the context for understanding the code of interest, previous solutions to similar tasks, and other developers who might have expertise on the task.

Three basic layers of relations weave a software system into a STIS with a collection of interlinked information resource nodes: the *structural* layer, the *conceptual* layer, and the *developmental* layer (Fig 1).

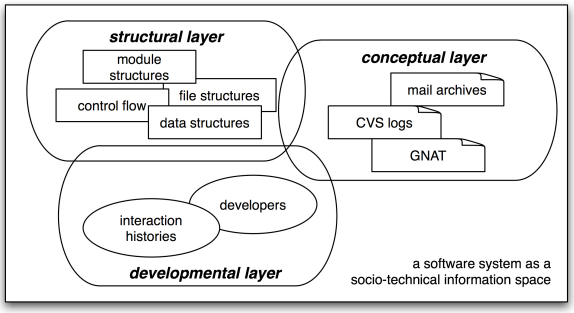

Figure 1: A Socio-Technical Information Space (STIS)

2.1 The Structural Layer

A software system can be viewed as a collection of *Code* nodes linked by the syntactic or dynamic structure of the programming language, such as data flow, control flow, or linear order.

Code nodes can have different levels of granularity. The most basic node is its physical unit, a file or a class. At this file granularity, nodes are organized and linked by their hierarchical file structure or inheritance structure. Each file node can be decomposed into language specific module-level nodes. At this module granularity, nodes are linked through their control flow and data flow that defines the order of execution. Those modules also have a linear relation as they appear sequentially in the file, which defines the order of compilation. From the perspective of socio-technical information space, the file presented in most program editors can be interpreted as a specific view of presenting the software system based on the linear link of module-level *Code* nodes. The module-level nodes can be further decomposed into nodes of statement blocks and variables, which are linked by their linear order, define and use relationships, control flow, and data flow. Links between nodes of coarse granularity can be decomposed into the links that relates nodes of finer granularity. For example, links between files can be decomposed into links that go into files or go into modules that compose the files.

2.2 The Conceptual Layer

The conceptual layer of relation links nodes based on the structure of the problem domain. Problem domains are often described in functionalities, features, concerns or performance criteria. Each functionality, feature or concern is realized in a set of code nodes. Code nodes that combine to implement a certain feature are obviously related. Functionalities and features interact with each other, and such interaction gives rise to another type of relationships among the code parts that link different parts of the code at different granularities such as files, modules and blocks. Development documents such as emails and bug management records often describe functionalities, features or concerns, and they bear relations among themselves and with code at the conceptual layer.

Ideally, the conceptual relationship among the code should be consistent with the structural relationship, and when programmers need to modify a feature, they only need to deal with changes that are localized in a specific part in the structure. As we mentioned before, the two types of relationships are unfortunately often inconsistent. An analysis of randomly chosen 20 change tasks in a well-designed software project found that 15 involves more than one classes and 5 involved more than one package [7]. The inconsistency of two kinds of relationships is one major challenge for continuous development task because developers have to crisscross the system to identify those relevant parts [6].

2.3 The Developmental Layer

Programmers do not add code to the programs linearly by one statement following another statement, or one file following the completion of another file. They add pieces to the system by realizing a certain task plan that satisfies a particular sub-goal [8]. A plan is a set of actions that, when executed in a correct order, achieve the goal of the plan, which often corresponds to a feature, or a use case. In the implementation of a plan, programmers often have to touch a set of program parts by either changing or adding new pieces at varied levels of granularity. Each development task, therefore, forms a trail that cuts through the information space of the software system by linking those relevant code parts. However, those trails become lost in the final program and cannot be easily reconstructed.

Programmers currently use a range of tools for development, communication and coordination. Those tools generate abundant information about how software systems have been developed. Viewed from the point of code, such developmental information is meta-information about the code; and although the meta-information is not essential for system execution, it provides data to reconstruct the trails previous programming activities have blazed, which can be used to help later programming activities for navigating the source code and documents.

Such meta-information about the system can be used to create another layer of links among different parts of code. Meta-information, or interaction histories of programmers revealed in communication and coordination tools (i.e. who touched what parts of the code at what time), can be used to infer the relevance of code parts. Source files that have been checked in to the code repository frequently at the same time signal a potential logical coupling [12], because programmers tend to add code one plan chunk at a time. If we dig deeper, the code fragments that have been changed at about the same time could further pinpoint the logical coupling to a finer granularity. Code parts that have been developed or changed by the same developer could also signal a potential link among them because developers are assigned to tasks along the decomposition of functionalities or features. Bug management records could help programmers gain a deep understanding of the context of certain code, and links code parts that are changed simultaneously in response to a single bug report.

3. CREATING RELATIONS IN SOCIO-TECHNICAL INFORMATION SPACES

The paramount goal of conceptualizing software systems as STISes is to provide navigational support for programmers to identify relevant information hidden in the software system for their current development task. Most development tools already have mechanisms that link code parts through structural dependency such as define-use relationship and call relationship. Using this structural dependency alone has been shown to have a high cost both in terms of task and cognitive resources [6]. Viewing system systems as STISes provides a unified conceptual framework that treats all three kinds of information resource nodes and their three layers of relationships uniformly, and guides the development of new mechanisms that allow programmers explore latent relations among different aspects of the software system. We divide relations in STISes in two types: *primary relation*s and *composite relation*s, and introduce the major categories with some examples.

3.1 Primary relations

Primary relations refers to the kinds of relations that can be directly obtained by analyzing existing data, source code, and organizational structure of the development team. Six categories of primary relations exist in the STIS of a software system:

- *code-code*: A piece of code can be related to other pieces of code in many different ways. For example, *Call(c1, c2)* defines the order of execution; they are related if one inherits from the other.
- *code-document*: A piece of code is related to a document if it implements features or functionality contained in that document based on traceability analysis [1]; or if the document, such as a CVS log, explains the development history of the code [2]; or if the code is accompanied by a reference documentation.
- *document-document*: A document is related to another document if their contents are related. For example, a bug report is related to emails that discuss the bug; an email is related to another email if it replies to the other.
- *programmer-code*: A programmer is related to a piece of code if he or she has done something with it. For example, *Modify(p, c, t)* represents that programmer *p* has modified code part *c* at time *t*. This relation could indicate that *p* might have implicit expertise about the code part *c*.
- *programmer-document*: A programmer is related to a document if he or she participated in the discussion, creation, or modification of the document. This relation may imply the programmer has contextual knowledge about the document.
- *programmer-programmer*: The relation between programmers can reflect the history of their social interactions, including who has helped whom, who prefers to work with whom, and who has sent emails to whom; or the relation between programmers can be defined by their co-location which has impacts on communication and coordination.

Examples of primary relations listed above are certainly not exhaustive. Much previous research has tried to uncover various kinds of primary relations using either the structure-based approach or the concept-based approach.

Structure-based approaches analyze the control flow and data flow of programs to generate an abstract description of the system by generating the links among code parts. Various syntactical dependency graphs are generated at varied granularites. Such graphs provide tools for thoroughly examining the impact of a file, a variable or statement to the whole system. Due to its thoroughness, such graphs are very complicated and tools unhandy to use. For a particular task, not all dependent parts are affected; sorting out the task-relevant ones is not easy in overcrowded dependency graphs. In addition to the difficulty in tool operation and the inherent complexity of dependency graphs, this approach only captures the current state of the system and does not give a historical account of its development.

The concept-based approach tries to use semantic information contained in programs and documents to pinpoint task relevant parts. This approach does not strive for the complete analysis of the whole system. Instead, it utilizes heuristics to go through the cycles of hypothesis and verification to search through the information space of the system. Many researchers have observed that programmers typically look at the comments or identifiers that reflect the concepts of the change task, assuming that code parts that contains conceptually similar or identical words are related to their current task. Conceptual similarity provides the first cut search to narrow the code parts that are to be further investigated and analyzed for verifying their true relevance. Information retrieval techniques, ranging from vector spaces and latent semantic analysis, have been used to identify relevant code based on their conceptual similarity with various degrees of successes. Hipikat [2] has gone further by locating content-similar documents to provide contextual and developmental information.

3.2 Composite relations

Composite relations further relate information resources by using the previously defined primary links through the application of the spread activation strategy.

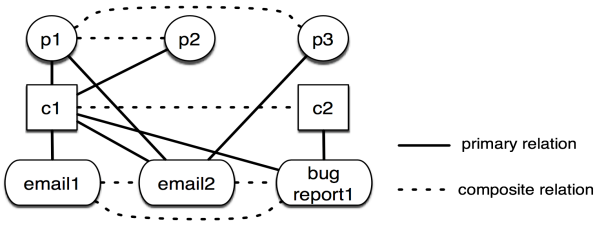

Figure 2: Composite relations in an STIS

As illustrated in Fig. 2, a programmer is linked to other programmers if they have touched the same code part *(p1, p2)* or participated in the discussion of the same topic in emails *(p1, p3)*. An email is linked to other emails if they both discuss about the same code *(email1, email2)*. Bug report records are linked to emails if they are related to the same code part *(email1, bug report1)*. Code parts are related if they are related to the same bug report *(c1, c2)*. Fig. 2 only shows the composition of two primary relations. Further composite relations can be obtained through spread activation: for example, *p1* is related to *bug report1* through the links of *(p1, c1)* and *(c1, bug report1)*.

Repeatedly applying the spread activation strategy to the links among the triangulated relations Programmer-Document-Code, programmers can identify related information resources from any entry point that they first hypothesize as relevant to their task at hand. They start with a piece of code that they know is related with their task, and locate relevant code, emails, bug report records and other knowledgeable developers. They can also start with a programmer they know that stands in relationship with

their current task to identify similar set of information resources. Or they can start with a particular email discussion or bug report.

Several researchers have attempted to use these kinds of composite relations to uncover latent relations among information resources. Ying [11] has developed a method of recommending relevant files to be changed when one file is changed by a programmer. The method mines CVS logs and deems those files that are frequently changed at the same commit transactions have logical coupling. Zimmermann et al. [12] adopts a similar approach and refines the recommendation to the granularity of code segments.

Within STIS, the above strategy can be defined as a composite relation *LogicalCoupling(c1, c2)* based on the primary relation between programmer and file *Modify(p, c, t)*:

> *LogicalCoupling(c1, c2)*: if there exists *p* and *Modify(p, c1, t1)* and *Modify(p, c2, t2)* and *t1-t2 < delta* where *delta* is the time separation used as the threshold to define co-change.

From the same primary relation *Modify(p, c, t)*, we can obtain a composite relation between developers *SocialCoupling(p1, p2)* similar to what was proposed in [9]:

> *SocialCoupling(p1, p2)*: if there exists *c* and *Modify(p1, c, t1)* and *Modify(p2, f, t2)* and *t1 – t2 < delta*.

Taking code relation into consideration, we are able to compute another composite relation between programmers described in [3]:

> *SocialDependency(p1, p2): if there exists c1 and c2, and Modify(p1, c1) and Modify(p2, c2) and Call(c1, c2)*

The above two composite relations infer that programmer *p1* and *p2* are potentially coupled in their task assignments, and one could serve as information resources for the other. Because social ties affects the effectiveness of information sharing and exchange, we have taken a step further by considering the existing social relationship among programmers and estimate the possibility that a programmer *p1* is willing and able to help programmer *p2* for a given task of dealing with code *c* [10]

> *HelpingProbablity(p1, p2, c)*: *Modify(p1, c)* and *Friendly(p1, p2)*

Or, we can loose the criterion of measuring expertise on code *c* by treating all programmers who modify code parts that affect or are affected by *c* as potential experts and get the following composite relation:

> *HelpingProbaility(p1, p2, c)*: if there exists *c1* and (*call(c1, c)* or *call(c, c1)*) and *Modify(p1, c1)* and *Friendly(p1, p2)*

The above examples are the composite relations that have been explored by existing research, demonstrating how a uniformed socio-technical information space can help us reasoning about the relations about code, documents and programmers. New composite relations can be developed by combining different types of primary relations. For example, we are currently developing a measurement *SiteCoupling(s1, s2)* that can be used to measure how two sites are coupled in a distributed software project:

> *SiteCoupling(s1, s2)*: *SUM(SocialCoupling(pi, pj)*, for all *pi* at site *s1* and *pj* at site *s2*)

This measurement could be used to predict and manage communication and coordination cost across different sites.

4. THE ROAD AHEAD

The conceptualization of software systems as socio-technical information spaces tries to capture and infer multi-layered relations among various types of information resources based on primary relations that can be obtained from source code, documents and development history. Instead of focusing on creating links based on a particular relationship, the goal of constructing social-technical information space from a software system being created is to provide programmers with multiple navigation paths along different layers of relations depending on the particular needs and background knowledge of the programmer. The goal is not to pre-compute and present all the relations, but provide means for programmers to reason and explore about relations among various information resources in a particular context in which composite relations are dynamically created depending on the interest of programmers. The major challenge lying ahead is how to develop an easy to use interface through which programmers can interact with the socio-technical information space and find relevant code and information effectively and efficiently by navigating along composite relations by combining existing primary relations as needed.

5. REFERENCES

[1] Antoniol, G., et al., Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering*, 2002. **28**(10): p. 970-983.

[2] Davor C. Cubranic and G.C. Murphy, Hipikat: Recommending Pertinent Software Development Artifacts, in *Proc. ICSE03*. 2003. p. 408-418.

[3] de Souza, C.R.B., et al. Supporting Collaborative Software Development through the Visualization of Socio-Technical Dependencies, in *Proc. GROUP07*. 2007. p.147-156.

[4] Erlikh, L., Leveraging Legacy System Dollars for E-Buisness. *IT Pro.*. **2000**(May/June): p. 17-23.

[5] Kersten, M. and G.C. Murphy, Using Task Context to Improve Programmer Productivity, in *Proc. FSE06*. 2006. p. 1-11.

[6] Ko, A.J., H.H. Aung, and B.A. Myers, Eliciting Design Requirements for Maintenance-Oriented IDES: A Detailed Study of Corrective and Perfective Maintenance Tasks, in *Proc. ICSE05*. 2005. p. 126-135.

[7] Murphy, G.C., et al., The Emergent Structure of Development Tasks, in *Proc. ECOOP05*. 2005. p. 33-48.

[8] Rist, R.S., Systems Structure and Design, in *Proc. Empirical Studies of Programmers: Sixth Workshop*, 1996, p. 163-194.

[9] Wagstrom, P. and J. Herbsleb, Dependency Forecasting in the Distributed Agile Organization. *CACM*, 2006. **49**(10): p. 55-56.

[10] Ye, Y., Y. Yamamoto, and K. Nakakoji, A Socio-Technical Framework for Supporting Programmers, in *Proc FSE07.* 2007. p. 351-360.

[11] Ying, A.T.T., *Predicting Source Code Changes by Mining Revision History*. Master's Thesis, 2003, University of British Columbia, Canada.

[12] Zimmermann, T., et al., Mining Version Histories to Guide Software Changes, in *Proc ICSE2004*. 2004. p. 563-572.