High-level Component Interfaces for Collaborative Development: A Proposal

Thomas Marlowe
Department of Mathematics and Computer Science
Seton Hall University
thomas.marlowe@shu.edu

and

Vassilka Kirova
Alcatel-Lucent
kirova@bell-labs.com

ABSTRACT

Software development has rapidly moved toward collaborative development models where multiple partners collaborate in creating and evolving software intensive systems or components of sophisticated ubiquitous socio-technical-ecosystems. In this paper we extend the concept of software interface to a flexible high-level interface as means for accommodating change and localizing, controlling and managing the exchange of knowledge and functional, behavioral, quality, project and business related information between the partners and between the developed components.

Keywords: collaborative software development, abstract component interfaces, knowledge management.

1. INTRODUCTION

Software development has rapidly moved toward collaborative development models where multiple teams from different organizations and possibly companies, distributed across the globe, work together to define, develop, deploy and often collectively maintain complex software systems [25]. The complexity of such projects and the rate of change they are exposed to are unprecedented. These include but are not limited to dynamic changes in industry trends and market landscape, in customer needs and user expectations, in user and stakeholder communities, in standards, tools and environments, and in project organization and partner polices. To effectively accommodate such change product and process means, including flexible architectures and interfaces, iterative development, and cooperative, hierarchical risk management are required [19]. In this paper, we specifically focus on high-level component interfaces and examine their role and structure as well as the benefits they can bring to large collaborative projects.

The boundary between partners in collaborative comprises not only the definition of one or more software component interfaces (to which the development efforts comply) and the exchange of project information such as modification requests and risk management plans, but also business policy and information flow, as well as knowledge and expertise exchanges. While some of the latter interactions may be generic, many depend on the nature of the components being developed and the differences between processes and policies between the partners.

A key differentiator in collaborative software development is effective cooperative knowledge management — the process of acquisition, creation, and exchange of knowledge between all participants in the development efforts. This process is facilitated by the establishment of open culture and requires infrastructure services that support it [30].

In [19], we argued for flexible software interfaces, where the flexibility entails the interfaces to not only accommodate a collection of software artifacts but also localized and specialized policies. We also presented a structure for flexible component interfaces for change-prone collaborative development, together with a change classification and a toolkit for managing change. In [17], we discuss artifact flow and dependency management, and suggest that these also naturally factor at the boundary between components. Here we consider knowledge management [1] and policy-driven [21] information management, and argue that the cross-collaborator software component interface provides a natural location for instantiating and specializing the policies and constraints of project and organizational interfaces as well as selected aspects of the software development process. We propose creating a per-component-interface business policy/information flow/knowledge management object, and suggest that the combination of this with the software interface provides a natural location for a number of highly desirable and specialized business, information and knowledge functions. (We combine those into an abstract, high-level interface for collaborative development.)

The rest of this paper is organized as follows. Section 2 briefly examines the issue of change, highlighting a set of project and product means for addressing and accommodating change in large scale collaborative projects. Section 3 discusses key properties of a high level component interface and their role in distributed and collaborative software development efforts. In Section 4 we formalize component interfaces as first-class business entities and explore their role and effects. Related work is briefly surveyed in Section 5, and Section 6 presents our conclusions and possible future directions.

2. CHANGE CLASSIFICATION AND TOOLBOX

Change is a key risk factor. The need to effectively accommodate it—whether a result of problems (corrective or preventive changes) or driven by changing needs and environments (adaptive or perfective changes)—has long been a major software engineering concern and is further exacerbated in large scale collaborative projects. Table 1 shows our change classification: *forward changes*, in which changes in requirements (introduced, respectively, by the environment, customer changes, or policy or third-party changes) drive changes in design; *revealed changes*, in which formally specifying requirements exposes difficulties in requirements, implementation, or both; and *backward changes*, where difficulties in implementation necessitate relaxation or other modification of product expectations. All stakeholders—development partners, customers, and the users they represent—jointly participate in the process of introducing changes, reacting to them, and hence bearing the economic responsibilities of the additional project costs (see Table 1).

Table 2 outlines the *change toolbox*, a set of approaches that can be combined to tame and localize change. They are classified into *process- and product-centric* means. In [19] we presented an earlier version of this toolbox, where the process-centric categories included agility, traceability, and organizational collaboration, to which we have here added artifact flow management (discussed in [3, 17]), cooperative risk management, knowledge management, and domain expert collaboration; the product-centric means included software architecture, component interfaces, interface patterns, and adaptive information, which we here extend with component interfaces. Most of these elements and practices have been used in different contexts; classifying, characterizing, and combining them, as presented here, adds new benefits and opens novel applications.

Flexible high-level component interfaces, discussed hereafter, allow for proper change localization, encapsulation and absorption of effects at component and partner boundaries.

3. FLEXIBLE SOFTWARE COMPONENT INTERFACES

The precise nature of externally visible interfaces is a serious tradeoff for collaboration. Absolutely fixed interfaces, as for contractual development, simplify project management, traceability and artifact flow, but limit the flow of information between components, and also inhibit agile development and limit flexibility in general, even within those components. In [19] we propose one possible solution, in which the interface is "structured" but not fixed, so as to accommodate maximum flexibility in development, while limiting redundant or irrelevant effort and supporting artifact flow and dependency tracking. These flexible interfaces use layered organization, design and enterprise patterns, and adaptive information to provide stable behavior while allowing the application of modern process approaches within the boundary of component development.

In order to accommodate change, provide traceability [16, 26] and support agile development [5, 18, 28] within components, the proposed interface has a tripartite structure. It consists of an invariant *kernel* describing the flow of problem information across the interface, but allowing extensions to handle newly discovered exceptions; a *shell* managing the structure and patterns of information exchange; and *auxiliary services* that allow for the exchange of adaptive information [25], specifically implementation- or execution-specific information that can be used to allow cross-tuning of components:

- Kernel interface architecture: the guaranteed core of a component's behavior and extra-functional properties. The ordinary syntax and semantics are fixed, although the exception semantics is extensible, to respond to newly found problems affecting other components. Otherwise, the kernel interface should be changed only in the most extreme circumstances. (Ordinary syntax and semantics includes but is not limited to the "happy path"—it includes all success scenarios and alternatives, plus some exception behavior. Unspecified exceptions might, for example, arise from design or implementation decisions, or from unanticipated limitations in services or platform.)

Change type	Arises from	Identified by	**Primary economic responsibility** (subject to negotiation)
Forward	Real-world changes	Customer or Developer	Joint
	Customer requests	Customer	Customer
	Policy changes	Customer, Developer or External	Varies
Revealed	Specification problems	Developer	Joint
Backward	Implementation problems	Developer	Developer

Table 1. Change types and organizational responsibilities

- Shell interface architecture: actual call-return or message structure, and additional services used in extending or evolving component behavior. Boundaries should be fixed, although details may be negotiable. The shell may include optimized or specialized calling patterns or asynchronous messages, wrapped sequences or selections of kernel calls, leverage of implementation decisions in partners, or forwarding of calls/results from clients or to services. It may also include desirable but optional features, since the nature of such late features can usually wait to be completely fixed. This is also a natural location to support changes in drivers, or more generally in platforms.
- Auxiliary services and incidental information (adaptive data/metadata): System, history, profiling, configuration, and other information developed in one component, not part of interface semantics, but usable by other components. In [19] we discuss in more detail an approach for enriching the interfaces with adaptive data to support global objectives and cross-component tuning and optimization.

Variation is supported through use of uniform software architectures [11, 24, 29], design patterns [10, 19] and enterprise patterns [12], and tools or views [16] combining traceability with software configuration management [23]. For example, agile, iterative design of a component may result in a modified interface with the outside world, but is more constrained when it affects the interactions with a foreign component. Nonetheless, many changes in the interface can be masked through use of design patterns, so that information is transmitted from one component to the other via the pre-defined *physical interface*, but each side sees its own logical version of the interface (analogous to the logical and physical views of a DBMS), using design patterns such as Façade, Adapter, and Bridge.

Another dimension of flexibility arises from moving the data boundary between components. Sometimes information produced and managed by one component can be used in another to provide or support improved performance, testing, tuning, record-keeping, implementation decisions, dynamic compilation and state-dependent conditional execution [24]. There are numerous opportunities here, for example: (1) Forwarding the identity of a service via *Proxy* can remove the need for the component to find and connect to such a service on its own; (2) Data structure state information can facilitate use of the State, Strategy, or Template Method patterns to improve performance; (3) History information may be simpler to keep on the caller side, but still be useful for dynamic specialization or optimization of component calls, and snapshots can be passed through use of *Memento*; or (4) Databases and knowledge bases shared between service and client will function more effectively if strategy and history information is allowed across the interface barrier.

Such information can also be used to simplify provision or verification of timing properties or reliability, or simplify enforcement of security and access control. While an initial description of this information may be desirable, the extent and nature of this information remains open to negotiation (although not ordinarily revocation), and its use is largely at the discretion of the component developers.

4. MAKING THE INTERFACE A FIRST-CLASS BUSINESS ENTITY

Flexible interfaces can also provide support for business and management processes [2, 21], including risk management [19], traceability and artifact flow [17], security [13], and knowledge management [1]. They provide a natural place to hook:

- Policies for accessibility, transmission, and inter-component security.
- Policies and procedures for information hiding (whether for abstraction, access control, security, protection of intellectual property and privacy, or for more efficient and effective use in risk management), handling of revealed changes, or other management contingency policies.
- Translation of artifacts (language-to-language, notation-to-notation, perhaps glossary-to-glossary [motivated by standards and regulatory compliance across jurisdictions, as well as important national/regional variations in idiom]) and generation of abstract views.
- Selective propagation plus context-sensitive versioning of change information, artifacts, and possibly policies, depending on information about the context and status of the adjoining components and the interface itself.
- Generation of some types of summary artifacts, including schedule tracking.

They also provide a natural place for aspects of knowledge management [1, 14], where various uses of information hiding also apply. Most of the above policies and actions have security [13] and knowledge management implications.

Means for dealing with change		Project Characteristics	Critical property when dealing with change
Process-centric Means	Agility	Short iterations, team collaboration, customer involvement; change tolerance and flexibility, easier evolvability	High evolvability; relatively small, hierarchical or idiomatic project; tight scheduling constraints
	Traceability (artifact dependency management)	Easy to navigate traceability matrix; well-structured artifacts required to minimize dependencies; "immediate-automated" identification of change impact; proactive monitoring of churn; tracking project progress	Large-scale projects with clearly identified requirements; known requirements paired with need for high innovation; large number of diverse dependencies or complex dependence web
	Artifact flow management	Clearly identifies artifact types, partner responsibilities, and communication; basis for controlled artifact state and transformations; positive interaction with SCM	Large and complex project with diverse set of artifacts; large number of active stakeholders; complex flow of information
	Organizational Collaboration	Open organizational collaboration, open channels of information exchange, cooperative risk management, ease of change propagation via uniform and interconnected process views	Bidirectional component interaction; end-to-end product constraints or security, integrity, reliability requirements; high demand for information processing capacity; cross-partner domain expert collaboration
	Cooperative Risk Management	Clear roles and responsibilities of stakeholders when dealing with change; change taxonomy; hierarchical risk management plans	Large and complex project with diverse set of artifacts; large number of active stakeholders; complex flow of information; high level of innovation
	Knowledge Management	Identify, structure, and specialize implicit knowledge and relationships; identify implicit patterns (data mining); handle need-to-know, security and intellectual property	Domain uses implicit knowledge and/or subjective judgment; patterns of use matter; substantial intellectual property
	Domain Expert Collaboration	Identify critical cross-component domain issues, inconsistencies, and tradeoffs	Heavy use of domain knowledge; domain models or vocabulary variable or unstable
Product-centric Means	Software Architecture	Support for "plug-and-play" modular replacement, idioms to handle complexity and problems, easier change analysis and propagation via uniform structure; approach for cross-cutting concerns (e.g., aspects)	Hierarchical application; structural complexity; evolvability of behavior; service-oriented architecture with families of services/clients; inter-component cross-cuts
	Component Interfaces	Restricted propagation of change impact across component boundaries, reduced complexity of dependence web via information hiding; design for abstractability of dependence information	Partners responsible for different subsystems; complex or heavy information flow across boundaries; intellectual property or security concerns
	Interface Patterns	Flexibility of logical interfaces with fixed physical interfaces; access to legacy or COTS/GOTS services/databases; better information hiding at component boundaries; improved agility within components; support for refactoring within components	Collaborative development; negotiable boundaries between components; agile component development
	Adaptive Information	Propagation of content, system and bookkeeping information across component boundaries; safe extensibility of information semantics and component interfaces; improved agility within components	High component coupling; component-crossing extra-functional constraints (performance, timing, etc.); negotiable information flow

Table 2. Means for dealing with change ("Change Toolbox")

In addition:

- Data mining [31, 32] of components—can use both pattern discovery in the component to find information, and filtering and abstraction, both to protect information and to hide obfuscating details—both during development and upon deployment/execution and maintenance/evolution.
- History and change information, component state, coherence of artifact configurations across components [23]
- Interesting question for discussion: ordering of filter-abstract-discover. Which leads to most patterns being discovered? Is there a chance of covert leakage?

Such interface objects will have references to the public state of the components it connects—not just object state, but process, project, product and business artifacts states.

Another advantage, specifically for emergent ubiquitous socio-technical-ecosystems, is that one or more partially instantiated first-class interfaces—software, information flow, and management policy—can be stored with a component, especially one providing fundamental business or technical support (whether for the product or the development process). When reuse of a component is considered, such interfaces can be considered simultaneously, and may greatly simplify the tasks of adaptation and integration.

5. EXTENDING THE MODEL: MULTI-PARTNER, MULTI-COMPONENT INTERACTIONS AND GLOBAL CONCERNS

Multi-dimensional interactions between partners and components require additional considerations:

- The simple model assumes that all interfaces are binary, and that there is only one interface between components developed by the same pair of partners. We expect most but not all n-ary interfaces can be factored into binary interfaces. The other issue is more interesting: there may be parallel interfaces, even potentially multiple interfaces between the same pair of components (for example, (1) from different modules inside one partner's component, or (2) requesting an intermediary for forwarding on different kinds of service).
- For any pair of partners with an interaction, there may be a single interaction object to capture legal and other details of that relationship. Appropriate component interfaces may and typically will refer to that interaction object. Likewise, since the interaction will involve all of the component interactions, the interaction object will need to be able to reference all of those interfaces.
- Nonetheless, it is possible to consider variations in information sharing or in translation (for example) among different interfaces attaching to the same relationship. Examples might include (1) two modules with different levels of internal security, or interacting with different third parties (whether partners or external entities), or (2) context-driven information hiding and translation, based on knowledge of the two communicating components and modules, and the intent of the given interface—so that technical vocabulary can be properly translated for different specialties, or details important for one module can be suppressed in another. The policies these differences generate can be separate, or simply modifications to a central policy attached to the interaction object.
- The negotiation for exceptional behavior, call and message sequences, and auxiliary services can still be separate, even if the modules on the ends of the interface are the same, through the use of interface and design patterns, provided that the interface pattern handles otherwise uncaught (and presumably uninteresting to the caller) exceptions properly.

Likewise, data mining and pattern recognition can be driven by context. Where two or more interfaces with the same partners (largely) share context, they can of course delegate to a common interface policy and knowledge object.

Finally, just as some design requirements remain global concerns, some aspects of knowledge management are likely to be better located either globally or hierarchically. The marketing plan is necessarily a global object, and the use of knowledge management for marketing will therefore be global. Identification of patterns in the change history of the project is hierarchical, since it needs both a local and a global view. In addition, cross-partner domain expert collaboration [9] is necessarily orthogonal to the component structure, although the information contributed by an individual partner's experts will be guided primarily by that partner's component (or components) and its interfaces.

6. RELATED WORK

The issues of change and its accommodation have been studied from many different perspectives: evolution and traceability [27], architectural flexibility [15], and interface design patterns [18] and enterprise patterns [12]. There is related work on dependence analysis and change impact analysis, mostly related to software configuration management and work separately on traceability [16]. For risk analysis in the context of collaboration, see [2, 20]. There is an enormous body of work on design and enterprise patterns [10, 12, 17]. For an overview of software architectures, and architecture modeling and specification, see [11, 22, 24, 28].

New architectural paradigms, such as Service Oriented Architecture (SOA) [8] define flexible architectural models that can provide basis for both the organization of the system being developed as well as the structure and operation of the collaborative environments used to develop the systems.

Inter-enterprise knowledge management is discussed in [14] in a general context; the use of knowledge management in software engineering is explored in [4, 6, 7]. Cross-component domain expert collaboration forms part of knowledge management and communication in [9]. Data mining of software artifacts is considered in [31], although largely limited to design- and implementation-level artifacts. There are numerous works and tools for distributed software configuration management [23], but nothing explicitly designed for collaborative development.

7. CONCLUSIONS AND FUTURE WORK

We propose expanding our prior work on collaborative software component interfaces to support localized treatment of change, project management information, management contingency policies and knowledge management. We have identified several advantages, and also discuss which aspects are best left centralized or delegated to a common business object.

Future work will include handling non-binary interfaces, possibly by modeling the interface as a set of specialized client-service interactions, peer-to-peer negotiation, and routing decisions, mediated by general (e.g., Adapter and Bridge) and special-purpose design patterns. We will also continue to work on codifying a set of special-purpose interface patterns and defining guidelines for their application.

REFERENCES

[1] W. Agresti: Knowledge Management, Advances in Computers, Vol. 53, 2000.

[2] H. Barki , S. Rivard , J. Talbot: An Integrative Contingency Model of Software Project Risk Management; Journal of Management Information Systems, 17 (4), 37-69, Number 4/Spring 2001.

[3] K. Bhattacharya, C. Gerede, R. Hull, R. Liu, and J. Su: Towards Formal Analysis of Artifact-Centric Business Process Models; Proc. of Business Process Management, 5th Intl. Conf. (BPM 2007), Brisbane, Australia, September 24-28, 2007, 288-304.

[4] L. C. Briand: On the many ways software engineering can benefit from knowledge engineering, Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002), 3-6, 2002.

[5] J. Coplien, N. Harrison: Organizational Patterns of Agile Software Development; Prentice Hall PTR, 2005.

[6] K. C. Desouza: Barriers to Effective Use of Knowledge Management Systems in Software Engineering. Comm. ACM, Vol. 46, No. 1, pp. 99-101, 2003.

[7] K. C. DeSouza, J.R. Evaristo: Managing Knowledge in Distributed Projects, Communications of the ACM, 47 (4), 87-91, April 2004.

[8] Th. Erl: Service-oriented Architecture: Concepts, Technology, and Design; Prentice Hall, 2005.

[9] D. Flynn, E. Brown, R. Krieg: A Method for Knowledge Management and Communication within and across Multidisciplinary Teams, 2008 Workshop on Knowledge Generation, Communication and Management (KGCM 2008), June 2008.

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

[11] D. Garlan, M. Shaw: An introduction to software architectures; Proc. of the IEEE, ICSE-15 Tutorial Notes: Architecture for Software Systems, 1993.

[12] G. Hohpe, B. Woolf: Enterprise Integration Patterns; Addison-Wesley, 2004.

[13] K.-S. Hong, Y.-P. Chi, L. R. Chao, J.-H. Tang: An Integrated System Theory of Information Security Management, Information Management and Computer Security, 11 (5), 243-248, 2003.

[14] N. Jastroch: Adaptive Interenterprise Knowledge Management Systems, 2008 Workshop on Knowledge Generation, Communication and Management (KGCM 2008), June 2008.

[15] D. Kelly: A study of design characteristics in evolving software using stability as a criterion; IEEE Transactions on Software Engineering, 32 (5), 2006, 315-329.

[16] V. Kirova, N. Kirby, D. Kothari, G. Childress: Effective Requirements Traceability: Models, Tools, and Practices; Bell Labs Technical Journal, 12 (4), Winter 2008, 143-157.

[17] V. Kirova, T. J. Marlowe: Addressing Change in Collaborative Software Development through Integrated Artifact Flow and Dependence Analysis, Proc. of 21st International Conference on Software and Systems Engineering and their Applications, December 2008 (to appear).

[18] C. Larman: Applying UML and Design Patterns; 3rd ed., Prentice Hall, 2004.

[19] T. Marlowe, V. Kirova**:** Addressing Change in Collaborative Software Development: Process and Product Agility and Automated Traceability, Proc. of the 12th World Multi-Conference on Systemics, Cybernetics and Informatics, June 2008, 209-215.

[20] M. Mohtashami, T. Marlowe, V. Kirova, F.P. Deek: Risk Management for Collaborative Software Development; Information Systems Management, 23 (4), 2006, 20-30.

[21] M. Mohtashami, V. Kirova, T. Marlowe, F. Deek: Risk-Driven Management Contingency Policies in Collaborative Software Development, Proc. of the 40th DSI Annual Meeting Conference, Nov. 2009 (http://www.decisionsciences.org/Annualmeeting/documents/DSI2009Saturday.pdf, accessed 10/19/09) .

[22] Object Management Group: Model Driven Architecture; Available at http://www.omg.org/mda/ (Accessed October 24, 2008).

[23] Open Directory Project: Software Configuration Management Tools, Available at http://www.dmoz.org/Computers/Software/Configuration_Management/Tools/, accessed 11/11/08.

[24] D. Perry, A. Wolf: Foundations for the Study of Software Architecture; Software Engineering Notes*,* 17 (4), 1992, 40-52.

[25] J. Pollock, R. Hodgson: Adaptive Information; Wiley-Interscience, 2004.

[26] B. Ramesh, M. Jarke: Toward reference models for requirements traceability; IEEE Transactions on Software Engineering, 27 (1), 2001, 58-93. R. Sangwan *et al*: Global Software Development Handbook, CRC Press, 2006.

[27] K. Schwaber, M. Beedle: Agile Software Development with Scrum; Prentice Hall, 2002.

[28] M. Shaw, D. Garlan: Software Architectures: Perspectives on an Emerging Discipline; Prentice-Hall, 1996.

[29] United Kingdom Office of Government Commerce: ITIL-Continual Service Improvement, 2007, 125-126.

[30] I. H. Witten, E. Frank: Data Mining, Morgan Kaufmann Publishers, 2000.

[31] T. Xie, J. Pei, A. E. Hassan, “Mining Software Engineering Data”, Companion to the Proceedings of the 29th International Conference on Software Engineering, 172-173, 2007.