Addressing Change in Collaborative Software Development: Process and Product Agility and Automated Traceability

Thomas MARLOWE
Mathematics & Computer Science, Seton Hall University
South Orange, NJ 07079 USA

and

Vassilka Kirova
Computing and Software Principles Research, Bell Labs, Alcatel-Lucent
Murray Hill, NJ 07974 USA

ABSTRACT
This paper discusses process and product-centric means for accommodating change in large scale collaborative software development projects. We examine traceability, agile development methods and architectural flexibility for their roles and impact in such projects. And while these practices are well known and used in a variety of contexts, in this paper we focus on their cooperative use and codify an overall approach for dealing with change in large scale collaborative efforts, from both management and technical perspectives.

Keywords: Traceability, Agility, Software Architectures, Dependence Analysis, Change Impact Analysis, Collaboration, Interfaces, Design Patterns.

1. INTRODUCTION
Today’s software projects are large in size, rich in content, and complex, with multiple dependences, rapid technology innovation, team diversity and last but not least dynamic changes in requirements, technical and business environments, and markets. A release in the telecom, banking or other sectors can deliver tens to hundreds of features with functionality captured in thousands of requirements. The designs contain hundreds to thousands of classes, and depend on dozens of libraries. Project teams for such large scale development efforts are often distributed across continents and belong to different organizations. Multiple external suppliers also deliver features, components and services. The individual project teams are bound through processes as well as cooperative, contractual and organizational agreements. Changes impacting releases and related teams are dynamic and significant, with requirements percentage churn in double digits driven by changing customer needs, business constraints and technology evolution. Handling change effectively requires a shared understanding of the nature of changes and control of their “footprint”. Current software engineering offers multiple approaches for dealing with change, including agile methods, traceability, risk management, rigorous regression testing, comprehensive change management, and flexible software architectures.

Agile methods rely on close team and customer collaboration, gradual discovery and refinement of requirements, and short iterative cycles. However, the lack of focus on specification and intermediate project artifacts reduces the benefits for large-scale complex projects, especially for distributed and collaborative development. By comparison, traceability becomes critical when a large number of artifacts and components are created in response to a large number of changing requirements. Requirements traceability is intended to ensure continued alignment between stakeholder requirements and the outputs of the system development process [22]. Traceability maintains a “web” of dependences among project artifacts, and allows for easy determination of the impact of changes, among other benefits. Good risk management adds benefits by identifying and monitoring the potential sources of change. These process-centric approaches must be co-applied with good product definition and architectural practices to result in open, scalable, modifiable architectures and interfaces that can accommodate change.

One serious challenge in collaborative development is that the need for modular development with fixed interfaces limits both traceability and process agility. Changes can be traced from and back to interfaces, but not directly past the boundary, limiting the use of the dependence web. Likewise, interfaces—whether server-side, client-side, or peer-to-peer—freeze requirements and limit the value of agile, iterative development.

While these practices and others are being used alone and in combination, and have been examined in a variety of contexts, in this paper we focus on formal integration of a subset of them—agility, traceability and architectural properties—into a methodological framework for collaborative software development, structuring and codifying an overall approach and architectural guidelines for dealing with change in large scale collaborative efforts. We emphasize that only the structure and in some cases the implementation of the discussed practices are new, while the basic components are known and typically well-understood. The rest of this paper is organized as follows. Section 2 outlines critical attributes of collaborative software development projects. Section 3 examines traceability and agility as tools for accommodating change. Section 4 focuses on architectural concerns related to change. Section 5 presents a summary, Section 6 briefly discusses related work, and Section 7 provides conclusions and suggests future directions.

2. CHANGE AND COLLABORATIVE PROJECTS
Traditionally, software engineering practices have been focused on teams employed by a single organization, under a single central authority, and subcontractors, if any, used to develop relatively independent and fully specified components, services or clients, rather than sharing in the main development effort. However, software development practice has moved beyond this model toward widely distributed inter-organizational development:

- In distributed software development, teams work at different, broadly distributed sites, often with different
expertise and background; face-to-face communication is rare and often completely replaced by electronic means, such as teleconferencing.

- In inter-organizational development, teams work for different organizations. The model can either be contractual, with one central authority, and teams working in specific functional areas, or on selected components with carefully delineated pre-defined interfaces and behavior, or cooperative, where teams from partner alliances [17] work on major sub-systems and components with somewhat flexible interfaces and iteratively/evolutionary specified behavior, often without a clear, universally accepted central authority for resolving differences and conflicts.

Moreover, collaboration between, for example, university/corporate research and corporate R&D, or in the development of software tools expected to work together in collaborative environments, often results in partners working on strongly interacting and interdependent components, each depending not only on external functionality and interfaces, but on the a priori unpredictable quality of the information provided or performance of the application. When projects aim at solving novel problems, or otherwise entail substantial innovation, or need to respond to rapidly changing technical, business, and social environments, or have severe extra-functional (e.g., security, reliability, availability, timeliness, or safety) supplementary constraints and objectives, even the functionality and behavior of the system may not be easy to fully delimit a priori, while satisfaction of end-to-end constraints can only be determined by integrating the implementations, so that the nature of a component and its interfaces may not be fully specifiable in advance. The collaborative software development efforts considered in this paper are distributed, inter-organizational and cooperative, and must adjust simultaneously to each of these pressures. Further, we focus on large scale, innovative or otherwise complex development efforts (for project complexity, see [21]).

Nonetheless, while the trend for large in-house projects has very much moved toward object-oriented software development, increasingly (but slowly) with agile methods and iterative development, the relationship between developer and subcontractor continues to follow a more traditional model, for legal and economic reasons, and in accord with standard practice. The boundaries, functionality and interfaces of subcontractor components are typically frozen, constraining both subcontractor and primary developer. While a reasonable model for contractual development, it is a poor fit for more collaborative models of software development, or for high-innovation collaborative projects.

### 3. TRACEABILITY AND AGILITY: A PROCESS-CENTRIC “TOOL BOX” FOR DEALING WITH CHANGE

#### Change classification and dependence analysis

Software projects, as the systems being developed, are increasingly large, interconnected, complex, and dynamic. They involve substantial subcontracting and/or collaboration, and have multiple stakeholders [17].

The need to effectively accommodate change—whether a result of problems (corrective and preventive changes) or driven by changing needs and environments (adaptive and perfective changes)—has long been a major software engineering concern and is further exacerbated in such projects. Change is a key risk factor—motivating design and management both to anticipate future changes (in requirements, technology, staffing, resources, plans, etc.), and to facilitate changes necessitated by feasibility issues or by problems and flaws uncovered during development or subsequently during use [17].

Handling change requires an understanding of the nature of changes and their “footprint” on the project. Changes in software design resulting from changes in requirements due to real-world environment changes or customer (client) requests constitute forward changes. Backward changes, in contrast, occur when platform or development problems, or real-world constraints, result in modifying requirements. Revealed changes, finally, reflect refinements occurring as result of difficulties arising during analysis, design, coding, testing, or use. The development organization and the customer jointly participate in the process of introducing changes and bear the economic responsibilities of the additional project costs (see Table 1, below).

<table>
<thead>
<tr>
<th>Change type</th>
<th>Arises from</th>
<th>Identified by</th>
<th>Economic responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Real-world changes</td>
<td>Either</td>
<td>Customer</td>
</tr>
<tr>
<td></td>
<td>Customer requests</td>
<td>Customer</td>
<td>Customer</td>
</tr>
<tr>
<td>Revealed</td>
<td>Specification problems</td>
<td>Developer</td>
<td>Joint</td>
</tr>
<tr>
<td>Backward</td>
<td>Implementation problems</td>
<td>Development Organization</td>
<td>Development Organization</td>
</tr>
</tbody>
</table>

The footprint of a change, and thus the part of the project that must be examined for impact and perhaps modified, will depend on the form of the change and on the dependence relationships between project artifacts. Artifacts include not only code and configuration information, but also requirements and architecture specifications, design artifacts, test plans and test suites, deployment artifacts such as user manuals, and the change history documenting changes and decisions, among other elements.

The key dependences include implementation (or derivation) relationships, e.g., architectural designs implement the requirements (also called “vertical,” see [5]) and reference (or horizontal) relationships that represent a multitude of relationship types, for example, functional (e.g., feature dependences), structural (platform-application feature relationships, feature–sub-feature, or between different views) and contractual (e.g., between different components or services).

Other dependences may include cause-effect or source-manifestation linkages (e.g., test criteria and test suites), data and control (and perhaps temporal) dependences between code modules, or between interfaces and implementations, as well as reason-decision links. The dependence web for a project should include vertical, horizontal, and other dependences based on project-specific cost/benefits considerations. The actual set of elements included and maintained in a project’s dependence web is a function of the formality of the project, the available level of automation (tools, environments and integration) and the software engineering methodology used.

#### Strategies for dealing with change and complexity

Traceability and agile methods provide a process-centric “mitigation” approach to deal with change. (While each offers other benefits, we will only focus on this facet here.) Traceability is an established software engineering practice, traditionally used in artifact-centric (document-driven) software development models. It requires the creation and maintenance of bidirectional links (traces) between requirements and all other related artifacts in the lifecycle at different level of granularity.
derivation process—low-level system requirements are
product lifecycle. Thus, traceability is a result of the
relationships or dependences is created starting with
impact and handled accordingly. The traceability web of
change—forward, backward or revealed—can be evaluat ed for its
decisions) to manifestation (in design or program artifacts), any
set of artifacts is/can be traced from source (in requirements or
set.
Traditional software engineering embraces a strong version of
traceability, emphasizing early elicitation, development, and
analysis of a (nearly) complete and robust set of requirements, and
then, through the process of development, requires major
decisions in later phases to be traced back through earlier phases
to specific requirements, to development or deployment
constraints, or (in some cases) to choices given tradeoffs among
equally valid alternatives. The difficulty, however, is that the
initial requirements description for large complex projects is
often highly incomplete, imprecise, and modification-prone. On
one hand this makes trace information highly valuable tool in
identifying the root cause of changes and their consequences: first, the
cost of manually creating and maintaining large number of dynamically changing traces can
overwhelm the benefit of good dependence information, and,
second, “freezing” requirements early can coerce development
into blind alleys and less than useful approaches. There can also be a reluctance to consult the client (or by the client, to be consulted) after development of a high-level but highly detailed
specification, potentially increasing the magnitude and cost of
problems while “non-requirements-modifying” solutions are
sought.

Nonetheless, traceability remains useful even in a highly
dynamic project. If creation and maintenance of the web has been largely automated (as in model-driven development with
good requirements management, artifact generation and
software configuration management), there are immediate benefits. However, as some annotations and analysis are
required and the cost/benefit ratio can be high, the traceability
model (e.g., levels of granularity and types of supported
relationships) and implementation tradeoffs need to be
considered carefully [13]. When the dependence web is well-
maintained, traceability has two major benefits: (1) “immediate”
impact analysis of most changes (by forward or backward walks
through the web, with a bit of semantic analysis); and (2) a view of
coverage and state (including completeness, project progress,
etc.). In large and complex projects it has a major additional
benefit of consolidating the fractured requirements space
distributed between multiple organizations and development
teams and thus allowing for consistency and completeness. The
cost lies primarily in maintaining dependences and updating
artifacts for consistency, particularly non-code artifacts from
early in the development lifecycle. Therefore automating
traceability is critical for achieving effectiveness and
quantifiable benefits.

We have successfully used automated traceability to deal with
change, and to provide project monitoring and control in large
collaborative software projects. Our traceability model mirrors
the systematic artifact derivation process and allows for step-by-
step creation of a multi-dimensional traceability matrix—a web
of traces reflecting the relationships or dependences between
process artifacts at different levels of granularity (the lowest
being individual requirements objects). The model relies on
feature-based (functional) decomposition to control the scope of
the traces, which also reflects the structure of the sub-projects,
e.g., a feature team is responsible for the feature’s artifacts and
their traces. Features are also the contractual unit with 3rd
party developers, suppliers, or company-in-tial but I saw
organizational collaborations “black-box” traceability is applied, supported by
memorandum of agreement [5] with the changes tracing to the
agreements. To support the model we have implemented an
automated traceability environment which integrates multiple
artifact repositories, auto-generates a significant portion of
artifact mappings, and supports instant impact analysis and
dynamic project monitoring [14].

Agile software development views the software process as
iterative co-refinement of requirements and application, starting
with a set of “stories” that describe expected behavior, patterns
of interaction, and problems to be avoided [2, 3]. There may also be a supplementary specification of cross-cutting extra-
functional requirements such as performance, security, and
availability, which—depending on domain—may be equally
flexible or substantially more firm. A key feature is that,
regardless of the completeness or precision of the initial
requirements or specification document, the structure of, the
boundaries between components, and the placement of
responsibilities within components, remain open for
modification and negotiation, limited only by a small set of
stable architectural decisions that assure that the system is sound,
scalable, maintainable and within the boundaries of the agreed-
upon system concept.

In this model, the client (or customer) is an active partner.
Changes in requirements are natural and relatively easily incorpo rated. Incremental forward changes are readily
accommodated by iterative development, as are incremental
revealed changes. Backward changes must be negotiated with
the client, particularly where changes affect the value or utility
of the product. But these changes still have a much smaller
process footprint than if development had assumed a complete
initial set of requirements.

The agile methods approach also has its costs—effort can be
wasted when requirements could and should have been
formulated in advance. Also, for most projects, formal
documentation of specification and design will be needed at
some point, if only for archival and system maintenance and
evolution support purposes, and will be more difficult to extract
from project history than if a traceable set of artifacts had been
maintained. Further, coordination and information exchange in
large scale projects also becomes more challenging due to the
lack of sharable artifacts and the need to dynamically propagate
decisions, new functionality requests and changes.

Agile software development typically occurs in an iterative
object-oriented software engineering paradigm, which, for large
projects is currently usually based on some light version of the
Rational Unified Process [4, 25]. At the extreme, both stories
and supplementary specification should be as minimal as
possible, and application functionality and behavior should
emerge through team discussion and negotiation, client
consultation, and pair programming (incorporating comments as
There are four additional problems with this straw-man version of agile programming: first, clients are often interested in more complete initial requirements for legal and economic reasons; second, significant extra-functional supplementary requirements such as timeliness or security must be incorporated throughout development, rather than emerging or being added on as an afterthought; and third, interfaces between components cannot be left completely or even largely unspecified when many teams are involved in the development process. Finally, there appears to be an issue of diminishing returns [1]: the extra programmer and team effort seems to result in higher quality and more robust software as well as in greater customer satisfaction and team understanding, but not as effectively as one might hope.

Traceability and agility are often viewed as incompatible approaches to managing change. Strong traceability entails full requirements descriptions, maintenance of all process artifacts as formal components in a repository, and upon a change, update of all impacted artifacts, and of the dependence web, so that the repositories have a complete and consistent snapshot of the project state whenever a change is committed. Strong agility, on the other hand, relies on minimal requirements, embraces change, often creates only informal artifacts between formal milestones, and identifies only the dependence information needed to advance the project.

Yet almost all realistic software development has to integrate elements of both approaches. Even though these two approaches seem to be in conflict, many large projects will use both. Long-lived projects cannot ignore—must be willing to modify—existing requirements and specification, and must update those artifacts to support future development, coherent planning, and product usability, so that the dependence web must be maintained. On the other hand, long-lived projects must often make changes in response to changes in user or management expectation (forward changes) or to adjust to rapid advances in technology or standards and those are often best addressed by planning and managing the development in iterations or increments as well as keeping the channels of collaboration in the project as open as possible.

Further it is also important that the software architecture and the component interfaces in particular are flexible enough to accommodate change and support iterative development. The interfaces in collaborative software development are often treated as requirements with subsequent traceability obligations. There are also advantages in employing agile methods to develop components, which may result in requests for changes in the components’ boundary or the details of their interface.

Agile development in this form of collaboration is strongly affected by the specification of component boundaries, functionality, and interfaces between partners. To support process agility, we need to provide at least some level of flexibility in component specification. This entails both determining when changes affect the boundaries, and reacting to changes from other partners. Such changes will affect the specification and eventually the requirements both for the components, and in some cases for the entire system. To deal with this, some level of traceability will be beneficial. Thus we identify a “balanced sweet spot” with a project and domain-specific level of traceability, providing a map for easily identifying the footprint of a change and a level of agility that encourages iterative development and allows flexibility, modifiability and late decisions with regard to the interface specification (see Figure 1).

4. CHANGE AND ARCHITECTURAL CONCERNS – A PRODUCT-CENTRIC SOLUTION

Collaborative design of large, complex projects must also consider the structure of the product solution, which for simplicity we assume to be an object-oriented design and implementation (compare Figure 1). Here, also, there are several dimensions in better change accommodation and support:

- The software architecture and its specification, while based for example on an object-oriented modeling notation such as UML plus design patterns and perhaps aspects, should include both guidelines and structures to accommodate change while restricting its footprint or impact, and allowing for “deep” traceability (reachability) between interfaces and component artifacts.
- Highly collaborative environments targeted at highly innovative projects also require inter-component interfaces defined with care early in development, but such interfaces should allow some level of flexibility.
- Often, intended interfaces may not correspond to the exact way incoming information is used, or outgoing information is generated, especially where an agile process based on ongoing requirements discovery is used in developing the component. Generic [10] and special-purpose design patterns, following the principle of Protected Variation [12], can be used to provide a high degree of freedom in the internal component structure while substantially reducing actual interface changes.
- There will also be situations in which system or history information generated in one component may be useful or even necessary to assure required extra-functional behavior, or to dynamically select among method implementations or services in another. State or Strategy patterns can be adapted for these scenarios.

We explore these approaches in the remainder of this section—and summarize the results in Table 1.

Architecture

If agility and traceability represent process approaches to the management of change and complexity, software architectures represent a product approach, based on informal hierarchical reuse of idioms, guidelines, and components. Software architectures are used at both macro [11, 24] and micro levels, with the modern emphasis on analysis, design, and other patterns—with [10, 16] providing examples of the latter.
For collaborative software development, we are most interested in considering the design patterns intended to address variability, or to limit the footprint and impact of change. Adapter, which can be used to manage variability in services or their interfaces, and Bridge, which can be used to manage variability in implementation on different platforms, are examples of the former; while Decorator, which can be used to add functionality to a class, can be seen as an example of the latter, since use of Decorator can localize problems that would otherwise arise in a large class hierarchy. One can also consider uniform use of patterns [15] for cross-cutting concerns (e.g., access control) that persist across components during execution, both to facilitate fault detection and change, and to simplify provision and use of extra-component system and history information.

And while in our future work, we intend to consider additional design patterns (including domain-specific patterns and analysis patterns), here we focus primarily on patterns that can provide broad logical interfaces.

**Interface Structure**

While determining a good uniform system architecture and development strategy must be an initial concern, the immediate and critical next step must be to specify a very-high-level component design and component interfaces. Ordinarily, the division of both data and functionality between partner components is fixed, in contrast to the division within a partner component, where during responsibility-driven design the location of data, functionality, and interaction with services may be assigned or reassigned late in the design process. We would like to be able to do (or at least mimic) a limited amount of this in the collaborative environment as well. One fairly obvious case is a need for access to a common external service from multiple components. If its functionality requires that all partners see the results of a single call to the service, then we have to assign (or reassign) the responsibility for accessing that service to one partner, and add service calls from the other partners who require the information.

To accommodate change and agility, the interface should be provided with a maximum of flexibility. This entails a tripartite interface definition: kernel, shell, and services.

- **Kernel interface architecture:** the core of the component’s behavior and extra-functional properties, on which other components must rely. The ordinary semantics and syntax will be fixed, although the exception semantics should be extensible to respond to newly found problems. Otherwise, the kernel interface should be changed only in the most extreme circumstances. It may, however, provide some built-in flexibility, such as indicating some leeway in its timing constraints.

- **Shell interface architecture:** actual call-return or message structure corresponding to the kernel and additional services that may be used in extending or evolving component behavior. The boundaries should be fixed, although details may be negotiable. The shell interface may for example 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. The shell may also include desirable but optional features to be implemented if time and resources allow, since the nature of such late features can usually wait to be completely fixed. The shell layer is also a natural location to support adaptation for changes in

<table>
<thead>
<tr>
<th>Means for dealing with change</th>
<th>Critical property when dealing with change</th>
<th>Project Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process-centric Solutions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traceability</td>
<td>Easy to navigate traceability matrix; well-structured artifacts to minimize dependencies; “immediate-automated” identification of change impact; proactive monitoring of churn</td>
<td>Complex or tight extra-functional constraints (timing, security, etc.); high innovation paired with known requirements</td>
</tr>
<tr>
<td>Agility</td>
<td>Short iterations, team collaboration, customer involvement; change tolerance and flexibility, easier evolvability</td>
<td>Tight performance constraints; high evolvability; structural complexity</td>
</tr>
<tr>
<td>Organizational Collaboration</td>
<td>Open channels of information exchange, cooperative risk management, ease of change propagation via process uniformity</td>
<td>Bidirectional component interaction; security, integrity, reliability constraints</td>
</tr>
<tr>
<td><strong>Product-centric Solutions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Architecture</td>
<td>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 (aspects)</td>
<td>Hierarchical application; structural complexity; evolvability of behavior; families of services/clients; inter-component cross-cuts</td>
</tr>
<tr>
<td>Component Interfaces</td>
<td>Restricted propagation of change impact across component boundaries, reduced complexity of dependence web via information hiding</td>
<td>Largely independent modules; families of services/clients</td>
</tr>
<tr>
<td>Interface patterns</td>
<td>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</td>
<td>Collaborative development; negotiable boundaries between components; agile component development</td>
</tr>
<tr>
<td>Adaptive Information</td>
<td>Propagation of content, system and bookkeeping information across component boundaries; safe extensibility of information semantics and component interfaces; improved agility within components</td>
<td>High component coupling; component-crossing extra-functional constraints (performance, timing, etc.)</td>
</tr>
</tbody>
</table>

**Table 1: Means for dealing with change (“Change Toolbox”)**
drivers, and more generally in platforms, which may require different implementations of operations, and/or introduce different stresses and tradeoffs on memory, performance, and integrity.

- Auxiliary services and incidental information (adaptive data): System, history, profiling, configuration, and other information developed in one component, not part of interface semantics, but usable by other components. We discuss the nature and use of this information in below.

### Patterns and Logical Interfaces

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).

Useful patterns include:

- **Facade**, on the service side, to receive and respond to all interface messages, hiding the actual component entry points, and *Mediator*, to decompose Facade calls and to reconstruct return values. This approach allows the service developer to create a reusable component which can be sliced depending on the client’s desired entry points, and so to develop a flexible resource which is a significant intellectual property asset for this and other applications.

- **Adapter**, on the client side, to modify calling sequences and add required adaptive information, and, with *Proxy*, to allow the use of alternative services developed in a component. (Adapter is commonly used in federated databases for a uniform data interface, particularly when legacy data is involved.) In addition to supporting adaptive data, Adapter allows a single client component to interact with multiple, independently constructed service components, constraining their developers to a logical interface, but not necessarily a uniform implementation. Proxy can then be used to add access control, remote access, and a variety of other desired behaviors.

- **Bridge**, on the service side, to decouple logical server components from their implementation, so that all interface exchanges are with the logical view of the service component. *Builder* may then be used on the client side to paste together and wrap a number of service components available through 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, and so on [20]. There are numerous opportunities here, for example: (1) Forwardsing 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.

### 5. A STRATEGY FOR MANAGING CHANGE IN COOPERATIVE CSD

To summarize and codify the above discussion and our experience, we suggest the following set of high-level guidelines for managing change in large scale cooperative CSD projects:

1. Select an appropriate and uniform software architecture meeting the needs of the application and/or business domain. Determine required cross-component cross-cutting services.

2. Using the tripartite decomposition in Section 4.2, specify (to the extent possible) the flexibility in boundaries and interfaces. Create buffered interfaces as described in Section 4.3.

3. Use cost-benefit analysis to select the right level of traceability; define the traceability model [13].

4. Create/generate and maintain traceability relative to interface requirements and specification. Maintain bidirectional local dependence web to and from local interfaces to local and available/visible global requirements and to any other artifacts that are part of the traceability model and are in project’s (local scope). Notify collaborators of any new interface exceptions and of any faults, whose impact traced to an interface.

5. Maintain agility within components. Permit “under-specification” at interfaces insofar as it falls within the permitted flexibility boundaries, or, even better, can be masked in the buffered logical interface.

6. Determine adaptive data that will assist in perfecting the design or implementation of the component, and request that it be added to the interface. Provide requested adaptive data where feasible.

7. If aspects are to be used, provide interfaces to the uniform aspects implementing the cross-cutting concerns identified above. If at all possible, avoid separate implementation of common aspects in separately developed components.

8. Develop and register all required project and product artifacts as required for global version control.

9. Interact throughout development with risk management and with project management in general [17].

### 6. RELATED WORK

Due to space constraints, we can only give pointers to the literature. The issues of change and its accommodation have been studied extensively from many different perspectives: evolution and traceability [5, 6, 22]; architectural flexibility [12]; and interface design patterns [16]. There is related work on dependence analysis and change impact analysis, mostly related to software configuration management and work separately on agile methods [2, 3, 7] and traceability [21]. The concept of balancing agility and discipline and comparing flexible versus sequential process models has been addressed in detail by Barry Boehm and others, [4, 18]. For risk analysis in the context of collaboration, see [17]. There is an enormous body of work on design patterns [10, 16] and aspects [15]. For an overview of software architectures, see [11, 19, 24].

Software methods and tools are becoming available to support flexible development—for example, Eclipse [8], Scrum [23], and shared UML space tools such as Canyon Blue’s Konesa (which integrates with Eclipse)—although these are at present largely used for moderate-scale intra-organizational
collaboration, or for limited, typically research-oriented, inter-institutional collaboration, without high levels of management or legal concerns. Further, new architectural paradigms, such as Service Oriented Architecture (SOA) [9] 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.

Finally, in complex collaborative development efforts as the ones discussed here, legislation, regulations, standards and case law are also significant. Intellectual Property Agreements, for example, should prescribe ownership and use rights of the artifacts developed and limit the use of proprietary or confidential information by the other collaborating parties. Issues remain that will have to be resolved for full inter-organizational collaboration to be realized.

7. CONCLUSIONS AND FUTURE WORK

We have reviewed the challenges in using standard traceability or agility in collaborative software development, and then indicated a set of guidelines for accommodating both in large, complex, highly distributed collaborative projects, while integrating other standard practices for taming change. We have sketched an approach for providing stable guarantees of behavior and functionality at component interfaces, while allowing enough flexibility to support agile development. Our approach synthesizes a cross-section of existing techniques, correlating process- and product dimensions, and outlines guidelines for complex, innovative collaborative software development projects. We also discuss an approach for enriching the interfaces with adaptive data to support global objectives and cross-component tuning and optimization.

Future work will look in more detail at wrapping interfaces in standard patterns, and explore other general and special-purpose patterns to support flexible component development. Patterns for other aspects of the development process (analysis and testing patterns) should also be integrated with this approach; likewise, we will consider adapting organizational and communication patterns for collaborative development. Risk management activities will be addressed to see which can effectively be accommodated and improved within a flexible interface framework, and a taxonomy and approach will be developed for adaptive information and its use. More work also remains to be done on incorporating or accommodating tool support, and on addressing a broad range of managerial and legal issues and concerns.

Our work aims at encapsulating experiences with large scale collaborative development efforts and building a “toolbox” of process and product related guidelines and practices for dealing with change.

8. REFERENCES


