Departamento de Informática
Faculdade de Ciências e Tecnologia
UNIVERSIDADE NOVA DE LISBOA
2825 Monte Caparica - Portugal

Technical Report

DI-UNL-07/98

A Case for Components, Aspects and Reflection: The Quest for Tailorable Groupware

Jorge F. Simão
jsimao@di.fct.unl.pt
Atul Prakash
aprakash@eecs.umich.edu
José A. Legatheaux Martins
jalm@di.fct.unl.pt

Projecto DÁgora

URL Ref: http://dagora.di.fct.unl.pt/

April, 1998

A Case for Components, Aspects and Reflection: The Quest for Tailorable Groupware

Jorge Simão
FCT — New University of Lisbon
Quinta da Torre,
2825 Monte da Caparica, Portugal
jsimao@di.fct.unl.pt

Atul Prakash
University of Michigan
Ann Arbor,
MI 48109-1234 USA
aprakash@eecs.umich.edu

José Legatheaux Martins
FCT — New University of Lisbon
Quinta da Torre,
2825 Monte da Caparica, Portugal
jalm@di.fct.unl.pt

Abstract
This paper argues for the need of tailorability in groupware systems and the use of new software technologies to obtain it. It discusses issues and solutions in the design of shared workspaces, as a motivation for the tailorability requirement. It reviews three emerging software technologies: components' architectures, meta-level reflective architectures, and aspect-oriented programming. It presents a reflective components' model, and illustrates how it can be used to make tailorable groupware systems easier to build.

Introduction

A wide body of knowledge has been gathered in the last years regarding the development of groupware systems and supporting technologies. This includes both the understanding of the way people work and the way technology should support those work practices. Traditional distributed computing techniques have been adapted to the particular context of groupware. Shared workspace management issues like replication, awareness, concurrency control, and view sharing and updating, are now reasonably well understood. Although many issues are still open, suitable solutions have emerged and can be used in the design and implementation of groupware.

Unfortunately, there is no single set of design solutions that can be used in all contexts [2, 14, 6, 12]. Adequate options often depend on varying criteria: the type of the cooperative task, the current stage of the task, the nature of user's interaction, group's work practices, user's preferences, and the institutional context. This implies that generic groupware systems, both applications and platforms, must provide a wide range of services that can be easily adapted to particular scenarios. This adaptation has to be performed during the system's deployment phase, as well as at usage-time as the cooperation context changes. In short: groupware systems need to be tailorable.

Building tailorable groupware systems using traditional software technology is not an easy task. In this paper, we look inside emerging software technologies, namely, components' technology, meta-level reflective architectures, and aspect-oriented programming, and identify the relevance of them to groupware. Levering on this, we propose a reflective components' model that raises the level of abstraction at which programmers work, and makes tailorable groupware systems easier to build.

The rest of this paper is organized as follows: we start by examining the issues involved in the design of shared workspace services and discusses possible solutions; we continue by reviewing emerging software technologies; we present a reflective components' model and illustrates how it can be used to build tailorable groupware systems; and, finally, we summarize the paper.

Shared Workspace Services Design: Issues and Options

Shared workspace services are services providing mechanisms for groups of users to interact and cooperate in a common task. The technical designing of such services is a complex task, mainly because many issues related with distribution, coordination, and communication, have to be considered. This section describes some of the issues related with distribution and discusses possible solutions. The focus is on varying requirements and solution tradeoffs, and its intent is to provide concrete evidence for the need of tailorability in groupware systems. Of particular significance is the nature of user interaction, which can be tightly-coupled, loosely-coupled, or something in the between. (We use the adjectives synchronous and asynchronous, respectively, to refer to the two extremes.)

Distribution Architecture

The distributed architecture specifies the way system components are distributed: where the components are located; which components are replicated; and what replication semantics is or can be used; what types of process interactions take place. The following are the most frequent (fig. 1):

Client/Server — Each shared workspace is managed by a single server. Clients interact with each other only through server mediation. Clients usually replicate/cache (part of) the shared workspace to reduce access latency to shared artifacts. It is a well understood paradigm and is simple to program. Its major drawback is the lack of fault-tolerance to server crashes or to server unreachability (but not to client crashes). Scalability may also be a problem in some scenarios. It is often adequate for synchronous collaboration, but

may be insufficient for asynchronous collaborations where mobile and disconnected computing is required. A variation of this, is the the *centralized application – distributed user interface* approach. It suffers from the same problems as the client/server approach and it is in general less flexible. It has the advantage of allowing transparent reuse of legacy single-user applications.

Active Service Replication — Service state is actively replicated by a (small) set of servers to eliminate the fault-tolerance drawback of the simple client/server architecture. Active replication means that servers tend to maintain their state mutually consistent at all times. This can be implemented using a (reliable) group-communication subsystem that transparently replicates the service's state. Scalability improvements depend on the specifics of the service and on the usage profile. Clients interact with the service using, typically, a reliable RPC layer. The service is hard to implement if a group-communication subsystem is not already available.

(Pure) Peer — Clients interact directly without relying on servers. The coordination of all tasks is performed entirely by clients. This has the advantage of (potentially) providing low notification times of users actions, which may important in synchronous collaborations (e.g. in audio and video conferencing). Random encounters and synchronization of clients in asynchronous disconnected collaborations also fit well in this model. Scalability on the number of shared workspaces may also be improved because there is a natural distribution of load. It can be tricky to implement if certain reliability guarantees are required, e.g. atomic/reliable multicast of operations. Moreover, its efficiency depends on client resources whose availability is not as predictable as server resources.

Lazy Service Replication — A group of servers lazily replicates the service. This means that they can temporary disagree on shared workspace state. Synchronization is performed only occasionally, e.g. in pair-wise epidemic server interactions. Each client interacts with the server of its choice. It is a good model to support mobile computing, because clients can choose a server based on the criteria of (time-distance) proximity. It is not very suitable for synchronous collaboration because users require the illusion of a single physical shared workspace, and notification times are high.

Peer + External Services — An hybrid approach where clients rely on external services/servers, but still interact directly for fast dissemination of operations or for mutual synchronization of data. It may incorporate the advantages of several of the above models, but may be harder to design and implement.

Shared Artifacts and Operations

A very important feature of a shared workspace service is the way shared artifacts containing the state of the cooperative task and operations on them are modeled. Several approaches are feasible:

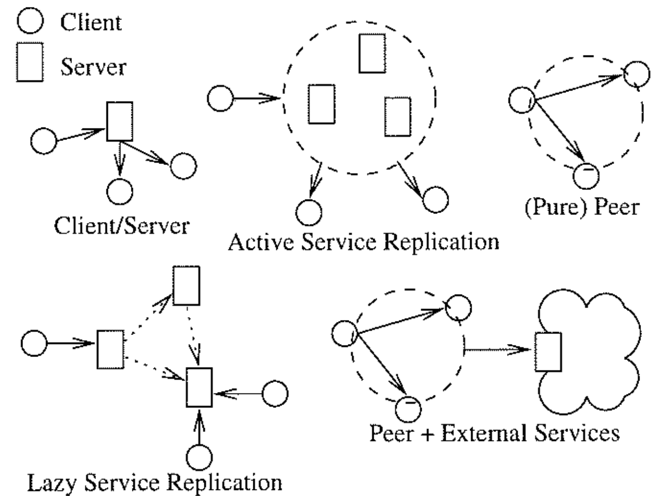

Figure 1: Distribution Architecture.

Untyped Data Items — Shared artifacts are represented as unstructured, untyped data items for which the only operations available are reads and writes. To be most useful in practice, operations should be aggregated into atomic transactions. It is easy to implement because only two types of operations are supported. It has the disadvantage of not allowing semantic knowledge to be used in the detection and resolution of conflicting concurrent operations. Traditional data base systems use this model.

Typed Objects — Shared artifacts are represented as objects that export a set of operations (defined in the objects' interface). This allows semantic knowledge to be used in the detection and resolution of concurrent conflicting operations. It is more complex to implement, and some coding is required to implement each new application type. In particular, no single (statically linked) server can be used to support all applications' types. Atomic transactions are not required, specially, if objects' structure is considered.

Hybrid — A fixed set of types is used to describe shared artifacts. Typically, this set includes a few primitive types (e.g. scalar types and row data chunks), and a few containers (e.g. lists, arrays and dictionaries). This allows semantic knowledge to be used in the detection and resolution of conflicting concurrent operations, but still makes servers independent of the particular application. Moreover, application's programmer work can be very simplified, because the supporting platform/framework is able to provide almost transparent handling of conflicting concurrent operations. The major drawback is that the set of available types may not be flexible enough. Atomic transactions should be supported, although that is not strictly required.

Operations Propagation and Awareness

Due to low response-time requirements and also to support disconnected work user's operations on shared artifacts are, typically, applied locally before they are sent to other entities (either peer-clients or servers). This raises the question of knowing when clients should propagate the operations, and, in case servers are involved, when should clients receive remote operations. When collaboration is synchronous we want both to be performed without delay, but as we loose coupling of user interactions other options may be more adequate. It may be performed on user request, as soon as is possible, or it may be decided using some heuristic based policy. A related issue is how to make users aware of each others actions. The more adequate way to choose which events are relevant to a particular user, and the way to present them, may also vary according to the work context.

Concurrency Control

A crucial issue on the management of shared workspaces is concurrency control, that is, to ensure that concurrent threads of users' activities on the shared artifact do not disrupt its consistency. Techniques to accomplish this can be characterized along two dimensions: the *consistency technique*, and the *degree of optimism* [6]. The consistency technique specifies the way access to shared artifacts is obtained and consistency is maintained. It can be based on resource allocation — *locks*, or it can be based on global *ordering of events* through the use of group ordering protocols (e.g. total ordering). The degree of optimism relates to the tradeoff between fast response-times and high availability, and the avoidance of conflicts. It can be *pessimistic* — operations are executed only when it is known that no conflicts can arise, or it can be *optimistic* — local operations are applied immediately to reduce system's response-time and increase availability. Greenberg's *et al.* study showed that no single solution is best, but rather that different contexts call for different options.

Conflicts, Dependencies and Merging

Optimistic concurrency control/replication techniques may raise conflicts between concurrent users' activities that need to be handled. What a conflict really means depends on the semantics of the operations (e.g. the pair-wise commutative and masking properties [3]), and the assumed dependencies between operations. In synchronous styles of collaboration transactions/sequences of operations tend to be small in size, while in asynchronous styles of collaboration the opposite is true. It turns out, that the adequate system's behavior regarding conflict handling is remarkable different in the two cases.

When the sequences of operations are small, the system can make assumptions that simplify conflict handling. If operations are assumed to be independent of each other then conflict handling can be made on an operation-pairs basis[1]. If operations are assumed to be independent of the particular state in which they are applied then conflict handling may be performed without user intervention. These assumptions can be made because the probability of conflicts is small, and if they do not hold the amount of "damage" done is small enough that the user can easily recover from it.

When the sequences of operations are large the above assumption may not be reasonable. For instance, the relevance of the initial work context may have to be considered and user intervention may be required. Moreover, the clustering of operations on parts of (structured) shared artifacts suggests that conflict handling may be accomplished more neatly on a per shared artifact basis, rather than on a per operation-pair basis. From a different perspective, the granularity at which conflicts should be handled must be coarse.

Whatever the particular details of the conflict handling mechanisms are, the following merging options can be envisioned for handling conflicting operations: *choose one* — one operation is considered and the other is discarded based on some criteria (e.g. precedence according to a system's imposed order; identity or role of the user; date and time); *choose both* — apply both operations in an order selected arbitrarily by the system; *transform* — apply both, but transform operations to account for the state context change (e.g. change element key in a dictionary); *merge* — merge the two operations in one that yields the desired result; *ask user* — let users decide which operation should be chosen; *make versions* — defer conflict resolution for a later time, and make separate versions of the shared artifact; *hybrid* — some configurable combination of the previous options [12].

Discussion and Directions

The above analysis on some of the issues involved in the design of shared workspace services shows that there is no single set of options which can be used in all cases. This suggests that is unlikely that a single complete service could be used to provide all the functionality required in a groupware system. The alternative of deploying a collection of (independent) services, each suitable for a different collaboration context, is also unattractive (e.g. providing independent support for synchronous and asynchronous collaboration). This is so because the work context may change frequently and the explicit switching of services may become an overhead to the user. Moreover, there is no reason to believe that the collection of services provided will be adequate for all possible contexts.

What we would like to have is a service, or a small collection of services, that can be dynamically tailored according to users needs. Furthermore, since our knowledge of the working practices is often limited, we should be able to easily prototype new services by reusing common functionality from available services. In the overall, we need an integrated system where component reuse and dynamic configuration fall within the same framework. A framework where similar

[1] If atomic transactions are used, each of these transaction can be regarded, to some extent, as a single operation for conflict handling purposes.

tailoring mechanisms can be used by platform and application programmers, system administrators, and end-users.

Unfortunately, building systems with above characteristics using basic object-oriented techniques and current software engineering practices is not an easy task. There is a need to rethink the very own foundations with which we build groupware systems. We need to look at potentially alternative technologies that will allows us raise the level of abstraction at which we work and still retain control of all aspects of the system. The second part of this paper addresses exactly this issue.

Components, Aspects and Reflection

In this section, we give a birds-eye review of emerging software technologies that promise to help in the development of tailorable groupware. We examine issues on component technology and software architectures. We describe how meta-level reflective architectures are used to create open implementations. And we describe the aspect-oriented approach to programming.

Component Technology and Software Architectures

Abstraction and modularization are the key reasoning techniques that allow us humans to think about complex problems. They allow us to reason about the all by looking individually at the parts. This *separation of concerns* naturally leads us to view software systems as a set of *components*, bound by architectural constraints, that are cooperating to perform some complex task. As systems become more, and more, complex there is a need to find new paradigms that help in the decomposition of systems' functionality into components and promote their reuse. Abstractions that proved successful in dealing with the small may not be appropriate when dealing with the large (e.g. viewing a complex system solely as a flat space of objects where the unique interaction mechanism between "components" is explicit, synchronous method invocation may not be adequate).

Research on *component technology* and *software architectures* attempts to deal with such issues of designing on the large [5, 15]. It evaluates how different architectural designs meet high-level requirements such as: understandability; easy of enhancement, by extension or modification; reuse of components, from previous or to future systems; reliability, that is, global resilience of the system to individual component's errors; and performance. This study is relevant to the groupware development to the extent that the flexibility demands on groupware systems can only be met by architectures where components can be easily added, deleted, and replaced. In a shared workspace, for example, such components would encapsulate the mechanisms and policies required to implement the service (e.g. distributed communication protocols, coordination algorithms, caching policies, objects' operations logging, etc.).

Many important architectural questions must be addressed when designing a components' infrastructure. What component interconnection topology is most adequate? Should components be allowed to interconnect freely, should they be connected in a rigid topology (e.g. in a pipe-filter style), or some combination of both? Should all interaction between components be controlled by a special *mediator* component? Should the overall system structure be flat or hierarchically organized? What should the pragmatics and semantics of the *connectors* that bind the components? Should communication be synchronous or based on event generation? Should components be explicitly addressed or should an event registration/handle model be used? What synchronization model should be used? Answering these general questions will help groupware developers to understand how flexible groupware architectures can be built.

Open Implementations and Meta-Level Reflective Architectures

Traditional software engineering is based on the concept of *black-box* abstractions. Software modules provide services with well defined interfaces, and hide the particular details of the implementation to their clients. This approach is conceptually useful because it promotes abstraction and modularization, but it presents some problems too. Very often the concrete implementation is not adequate for all service usage profiles, and clients end up performing badly. Sometimes the service provided is almost the right one, but something that the client needs is missing. *Open implementations* solve this problem by relaxing the black-box abstraction. They allow clients to modify and extend the service behavior, and influence aspects of the implementation that are crucial to them. This is done by providing services with two interfaces: the (traditional) functional service interface — the *base-interface*, and a *meta-interface* which allows clients to tune aspects of the service implementation [7, 10, 16].

In this context, object-oriented *meta-level architectures* provide a particularly elegant way of opening implementations. The designer of the service selects those parts of the system that she/he anticipates that may need to be tailored by clients, and makes them explicitly controllable through the use of *meta-objects*. Each meta-object encapsulates and controls the behavior of some aspect(s) of the system. It provides an interface, called the *meta-object protocol*, which has a dual purpose: it allows the implementation core (or run-time system) to delegate to it the responsibility of handling some aspect of the system; and allows clients to dynamically tailor the way the meta-objects control the system. Clients can also use the regular object-oriented mechanisms of inheritance and polymorphism to implement new meta-objects. This allows them to easily modify and extended the behavior of the system, and to provide an implementation strategy that is more suitable for their service usage profile. The collection of all meta-objects living in the system constitutes the *meta-level* of the system; the part of the system providing the base functionality constitutes the *base-level*. Because the resulting system has the ability to inspect, reason about, and

modify (aspects of) its behavior, we say that it is a *reflective* system.

As an example, in object-oriented languages some of the aspects a programmer might want to reflect are: the way objects are created, the way method dispatch is performed, and the way instance variables are accessed and stored. In the groupware arena, open implementations allow applications to tailor and extend the functionality of the underlying groupware platform for each specific context [2]. Reflection also allows applications to dynamically change the behavior of the groupware platform according to users' needs [13]. This can be done by using meta-objects to encapsulate the application's specific behavior, and by using the objects' meta-interface to configure it at run-time.

Aspect-Oriented Programming

Aspect-oriented programming takes *separation of concerns* further. Complete programs are written as collections of simpler programs, each dealing with a different aspect of the system. Each such *aspect program* is written in an *aspect language* that allows programmers to easily expresses the behavior of the aspect. A *weaver* tool is then used to collapse the several aspect programs into a single, fully-functional program [8]. The rationale for this is that many features of programs can not be easily isolated in clearly localized modules, that is, they are not amenable to functional decomposition. Rather, they cross cut the overall program structure and their handling intertwines in the code. This makes code writing confusing and prone to errors. As buildings architecture relies on several plans to deal with different aspects of the same building (e.g. plumbing, electricity, and ventilation), so should software engineering be able to do the same with software aspects [15].

As an example, in an object-oriented distributed programming scenario different aspect languages may be used to express different aspects of a distributed program (e.g. synchronization, persistence, and object marshaling) [11]. Likewise, the functionality of a groupware system can be seen as an amalgamation of aspects. For example, in a shared workspace management service (at least) the following aspects can be envisioned:

Replication — The mechanisms and the politics of shared artifacts replication and caching and their state transfer across address spaces.

Concurrency Control — The way concurrent operations to shared artifacts are handled.

Awareness — Shared feedback mechanisms based on event histories.

Access Control — The model used to specify access control policies.

A Reflective Components' Model

In this section, we present a model that embodies concepts and ideas from the technologies presented in the last section. It can be characterized as a *reflective component model*, where component architectures are regarded as first-class entities that are controlled by meta-level components. Components are represented hierarchically, to allow them to be manipulated at different levels of abstraction and detail. The rationale for the model is to allow meta-level components to deal with the structuring and reconfiguration aspects of a component architecture, while making the base-level components deal with the basic system's functionality. Component architectures can also be derived from other components architectures, by specifying the way in which they differ. This allows them to be easily reused. In the following, we describe the concepts, the interfaces, and the general usage of the model; we present some examples of how it can be used in practice to build tailorable groupware; we evaluate it according to the previously stated requirements; and we compare it to other models. Although our model does not prescribe an aspect-oriented language framework specific to groupware, it can be used as a substrate to which weavers generate code.

Model's Definition: Concepts and Usage

In our model, *component architectures* are explicit represented as run-time entities, which encode the structure and constituent components of the architecture. Each component has one *input port*, and zero, or more *output ports*. Input ports are used to receive service requests messages (method invocations), and output ports are used to *bind* to other components' input ports in order to delegate or continue request handling. Both input and output have (optionally) attached to them *interface specifications*, which are used for type checking purposes. A component architecture is itself a component, with an input port and output ports, rendering a hierarchical model of components. These components that define architectures are called *structured components*. *Atomic components*, on the other hand, encapsulate request handling code and constitute the building blocks for structured components (see below). We refer to the components inside a structured component, either be atomic or structured, as *sub-components*.

Each component is controlled by a *meta-level component*, that provides *introspective* methods to inquire about it and handles requests to reconfigure it. Such meta-level component (*meta-component*, for short), is the component that was first used to create the base-level component(*base-component* for short). Figure 2 illustrates the model.

Components are assumed to have global unique identifiers. In addition to that, sub-components can be (optionally) identified by *local names*, unique only within the structured component which they are part of. These names are used to associate sub-components with *roles*. Component's ports are also (optionally) identified by *port names*. Output ports have names that are specific to the context, while a component's input port is always named **in**.

A special meta-component named **meta_root** is used to bootstrap the system. It is an atomic component that defines the

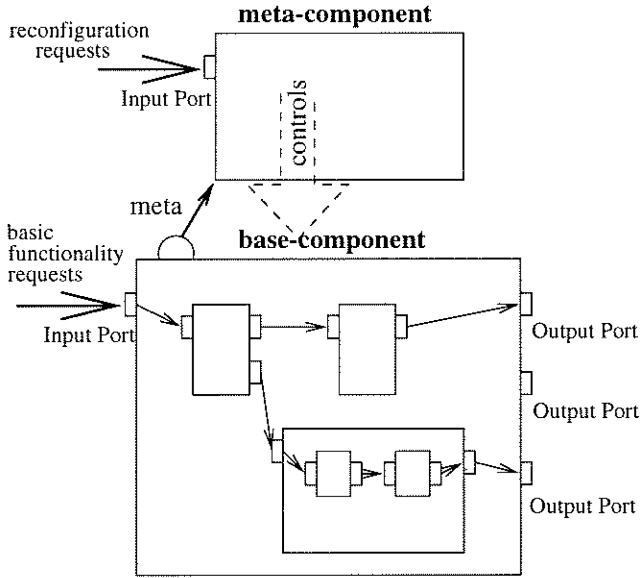

Figure 2: Components' model.

minimal (meta-)interface that all meta-components are expected to handle[2]. This interface includes methods to: add, remove, and replace sub-components from structured components; methods to add and delete output ports from atomic and structured components; methods to bind/connect input ports to output ports of sub-component; introspective functions to obtain information about sub-components, ports, and bindings; methods for component creation; a method for component destruction; a method for component activation; and a few other miscellaneous methods. Table 1 lists all this methods (the names self explain the intend). The first argument of many of the methods is a *base-component path*, that we enclose in angled parenthesis. A base-component path is either a component's global identifier, or component's global identifier followed by one or more dots and sub-components local names (e.g. A.local_name1.local_name2). This is used to easily access sub-sub-components of structured components. The **nil** identifier on arguments indicates an unspecified value. The **self** identifier is used to specify the outer component's ports.

Component creation is performed by either of three ways: by starting with an empty structured component; by creating an atomic components; or by *deriving* a new component from a pre-existing component — the *source component*. This derivation is done by specifying as argument to the method derive, a list of *derivation directives* that mimic the reconfiguration methods of table 1 (same syntax, but without specifying sub-components' identifiers). The particular semantics of the derivation is enforced by the meta-component used in the component creation. For the default case, that is, for the semantics implemented by **meta_root**, we assume it to

[2]Using the terminology of open-implementations literature, this interface might be called a meta-object protocol, although in this case the term meta-component protocol seams more appropriate.

Reconfiguration methods:
add_subcomponent<base_component_path> (local_name | **nil**, sub_component | **nil**)
remove_subcomponent<base_component_path> (local_name)
set_subcomponent<base_component_path> (local_name, sub_component | **nil**)
add_outPort<base_component_path> (out_port_name | **nil**, interface_spec)
del_outPort<base_component_path>(out_port_name)
set_port<base_component_path> (port_name, interface_spec)
add_binding<base_component_path>(local_name1 | **self**, port_name1, local_name2 | **self**, port_name2)
del_binding<base_component_path> (local_name | **self**, port_name)
Introspective methods:
get_subcomponent<base_component_path> (local_name) ⟶ sub_component
get_subcomponents<base_component_path>() ⟶ sub_component_list
get_port_spec<base_component_path> (local_name | **self**, port_name) ⟶ port_spec
get_ports<base_component_path>() ⟶ port_list
get_binding<base_component_path>((local_name | **self**, port_name) ⟶ {local_name | **self**, port_name}
get_bindings<base_component_path>() ⟶ binding_list
Component creation/destruction methods:
create_empty_component() ⟶ new_component
create_atomic_component(native_class_name) ⟶ new_component
create_derived_component<base_component_path> (derivation_directive_list) ⟶ new_component
destroy_component<base_component_path>()
Miscellaneous methods:
activate_component<base_component_path> ()
set_class<base_atomic_component_path> (native_class_name)

Table 1: Meta-components interface.

be the cloning of all components of the source component. Other meta-component may implement other "resource sharing" policies.

If the **nil** value is specified as the sub_component argument of the add_subcomponent method, that specifies an *abstract component.* An abstract component is an empty component, without a global identifier, that is used in the construction of structure components. It works as a placeholder whose intent is to be latter replaced by concrete component.

Atomic components are front-ends to native object-oriented systems. Although we could incorporate traditional object-oriented facilities in our model to program in the small (e.g. classes, objects, and inheritance), we prefer to leave this open. This allows the model to be easily incorporated with existing object-oriented systems and simplifies its definition. We assume, though, that atomic components can be *attached* to native system's classes. Such *native classes* are expected to implement an interface with a *dispatch method,* with a single argument for a request message, that allows atomic component to delegate the actual handling of requests to them. The native classes are also assumed to have a way to invoke methods on components. This allows them, among other things, to send messages through the output ports of the attached component and to invoke methods on the respective meta-component. (In a concrete implementation of the model, the available object-oriented mechanisms are probably used to represents components as objects and to make the cross model invocations with regular object method invocation.)

To simplify the creation of new meta-components, the **meta_root** component provides a special semantics for derivation when the source component is **meta_root** itself. It accepts a derivation directive that specifies a native class (set_class), and creates the derived meta-component as a structured component containing two atomic components. One atomic component, with a single output port named **out**, is attached to the native class. The other atomic component is a (logical) clone of **meta_root**. The **out** port of the first sub-component is connected to the **in** port of the second. This layout essentially allows the new meta-component to be implemented using delegation to simulate inheritance (as found in object-oriented languages). Since all meta-components are themselves components, they also need to have a (meta-)meta-component. Meta-components created this way have as (meta-)meta-component, the **meta_root** component.

Before a base-component handles any request it must be *activated.* This step allows the meta-component to perform a number of consistency checks. It can check for conformance between the component's output ports interface specification and the interface specification of the input port that they are bind to. And it can check for dangling ports, that is, ports that are not connected to any other port. As in component derivation, the particular checks performed on component activation are specific to the meta-component use. For the default case, we assume that no interface conformance checks are performed and that dangling ports are allowed. Other meta-components may implement more strict reliability policies.

To make the model easier to use, we augmented it with a *component language* that allows component to be declared at compile-time. Such declarations are mapped to run-time invocations of the component creation methods depicted in table 1. They are depicted below (when the meta-component is ommited, the meta-component of the source component or the **meta_root** is assumed as default):

Create empty component A with meta-component M:
component A [**meta** M] { derivation directives };

Create atomic component A attached to class C with meta-component M:
component A **implementation** C [**meta** M] { derivation directives };

Create component A derived from component B with meta-component M:
component A **derives** B [**meta** M] { derivation directives };

Create meta-component M extending ***meta_root*** *using class C:*
component M **derives meta_root implementation** C { derivation directives };

The Target: Tailorable Groupware

Below, we show some examples of how our model can be used in practice to build tailorable groupware. Example 1, and example 2, show how the model is used to create basic architectural styles. Example 3 leans on the first two examples, to create a more complex and realistic tailorable groupware system architecture. The common pragmatics to all examples is the use of derived meta-components to configure each specific architecture or architectural style.

Example 1: The Pipeline Architecture

A very simple, but useful way to architecturally arrange a set of components is by connect them as filters in a pipeline [1]. Whenever a component receives a request it performs a specific task, and delegates the rest of the work to the next component in the pipeline. In the context of a shared workspace service of a groupware system, components might be used to represent the several computational aspects involved in the access to shared artifacts. This might include components to deal with replication, access control, concurrency control and awareness, and the shared artifact itself. Using our model it becomes simple to realize this.

To compose components in pipeline architectures, we create a meta-component which specializes to that particular arrangement. That meta-component derives from **meta_root**, and redefines the add_subcomponent method so that it au-

tomatically bind components in a pipeline style (by inserting them at the end). This is done by using a native class that handles the method. The pseudo-code to do that is as follows:

```
component filter_meta derives meta_root
  implementation C {};
class C
{
  dispatch(message)
  {
    if (message.selector == add_subcomponent)
      add_subcomponent(message);
    else delegate message to out port;
  }
  add_subcompoent(message)
    { insert component in the pipeline }
}
```

Now we create a pipeline template that specifies the component involved in accessing the shared artifacts:

```
component shrd_artifact_templ meta filter_meta
{
  add_subcomponent(nil, replication_comp);
  add_subcomponent(nil, access_contrl_comp);
  add_subcomponent(nil, concurr_contrl_comp);
  add_subcomponent(awarn, awareness_comp);
  add_subcomponent(shrd_artifact, nil);
};
component replication_comp implementation ...;
component access_contrl_comp implementation ...;
component concurr_contrl_comp implementation ...;
component awareness_comp implementation ...;
```

Finally, we create shared artifacts by completing the above template with components that implement the specific shared artifact. This is done by replacing the abstract component locally named shrd_artifact with a concrete component. The resulting component is then ready to be activated (not shown):

```
component shrd_artifact derives shrd_artifact_templ
{ set_subcomponent(shrd_artifact, shrd_artifact_implm); }
component shrd_artifact_implm implementation ...;
```

To dynamically change the implementation of some aspect of a shared artifact, we only need to replace the respective subcomponent by another. For example, to change the awareness component we would execute at run-time:

```
meta= get_meta(shrd_artifact);
meta.set_subcomponent<shrd_artifact>
  (awarn, other_awarness_comp);
```

Example 2: The Mediator and Bus Architectures

If the interaction among components is complex enough that the filter-pipeline scheme becomes insufficient, then other architectural styles may be used instead. In the *mediator* based architectural style a distinguished component as the role of controlling the interaction between all other components — the *adaptors* [14]. In our model, this is done, again, by deriving from **meta_root** a meta-component specialized to that style. This meta component redefines the method create_empty_component to create components with a default mediator, and redefines the method add_subcomponent to bind adaptors in the appropriate places. The pseudo-code is similar to the above.

A very powerful variation of the mediator style, is the *communication bus*. In this style, a bus component is used to mediate all the communication between attached components. This allows components to cooperate without being aware of the identity of each other; any component may handle and respond to generate events. This can be implemented in our model by making components bind their input port and one of the output ports to the bus.

Example 3: A Shared Artifact Architecture

In a real system the above styles are combined to yield more complex architectures. In our model, this is best done by using hierarchical component architectures. This reduces the possibility of name clashes, and allows complex structured components to be manipulated as single entities. Figure 3 depicts a possible architecture for managing a shared artifact of a shared workspace service (meta-components not shown).

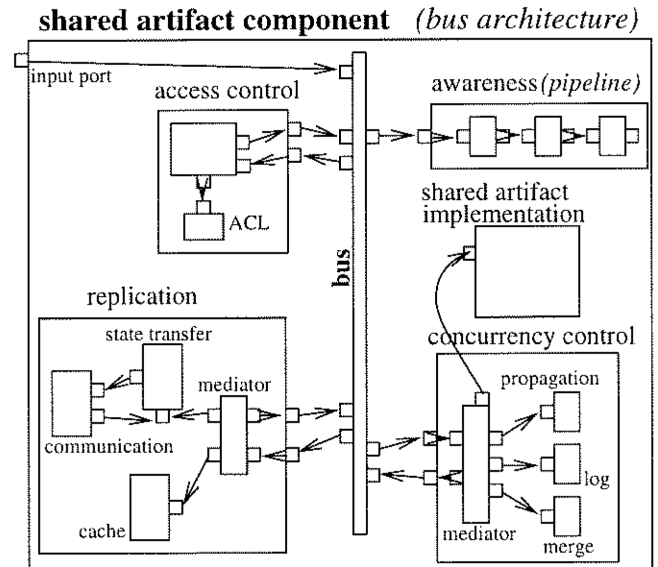

Figure 3: Shared Artifact Architecture.

In this architecture, a bus is used at the higher level of abstraction to allow the interaction between the main functionality components: replication, concurrency control, ac-

cess control, and awareness. The bus component defines the communication model that components use, and the components establish a protocol that allows them to coordinate their actions and handle operations on the shared artifact. Each main component is itself a structured component, with subcomponent for each aspect of the implementation the programmer/user may want to tailor. The awareness component is a pipeline where each sub-component is used to intercept and notify or log some of the events related to the shared artifact. The replication component and the concurrency control model use a mediator architecture to coordinate the interaction between its service building blocks. The access control component uses an ACL sub-component to specify the particular access control policy to use. The concurrency control model decides when operations are actually invoked on the shared artifact implementation. The code to build the architecture is not show, but it can be easily derived from the examples above.

To dynamically add, replace, or delete components from the bus, the meta-methods add_subcomponent, set_subcomponent, and del_subcomponent, are used as above. For example, to provide free access to the shared artifact by removing the access control component, and to change the caching policy of the replication component, the following code is used:

```
meta= get_meta(shrd_artifact);
meta.remove_subcomponent<shrd_artifact>
   (access_control);
meta.set_subcomponent<shrd_artifact.replic>
   (cache, other_cache_comp);
```

To create a shared artifact with a semantics similar to that of figure 3, but say with a different merging strategy and a replication component for a different distributed architecture, component derivation is used as follows:

```
component new_shrd_artf derives shrd_artf
{
  set_subcomponent<concurr_control>
    (merge, other_merge_comp);
  set_subcomponent(replic, other_replic_comp);
}
```

Dynamically changing a component in a complex architecture may not be always straightforward. This is so because it may require the "flushing" of ongoing activities in the replaced component and its neighbors. For components encapsulating simple policies this is hardly a problem, but for components encapsulating complex mechanisms this may be tricky (e.g. the concurrency control or replication component). In our model, that can be (partially) solved by making meta-components send special *pre-reconfiguration* messages to base-components, before the replacement is done, and *pos-reconfiguration* messages after the replacement. To synchronize the components, the base-component sends an *reconfiguration ready* message to the meta-component whenever it as completed the "flush". This would allow, for example, a distributed concurrency control algorithm to terminate gracefully and another to be started.

Model's Evaluation

Our model mets the requirements stated in the first part of the paper, to the extent that it allows simple reuse of architectural designs and implementations, and dynamic reconfiguration of groupware systems within the same conceptual framework. This is accomplished by using structural derivation as reuse mechanisms, instead of the traditional class derivation mechanism of object-oriented systems, and by dynamically reconfiguring base-components using meta-components. The explicit distinction between base-level and meta-level architectures, enforces a clear separation between the static and dynamic aspects of a tailorable groupware system. This makes the overall system's architecture more understandable and simpler to program.

Several issues remain open in our work. What concrete architectures and inter-component protocols should be used to build tailorable groupware systems? What design principles and methodologies to use? How complex is to synchronize the meta and the base level? How can existing groupware systems and services be re-factored in terms of our model to make them more tailorable? What costs are involved in the dynamic reconfiguration of a system? Our model provides a conceptual framework within which the difficult problem of building tailorable groupware systems can be addressed.

Related Models

Our model relates with reflective object-oriented models where meta-entities also control other (base-)entities (e.g. the CLOS programming language model [9], and IBM's SOM model [4]). In those models the meta-entities are meta-classes, whereas in ours they are the meta-components. The key difference is that in those models the meta-level control is performed only at the level of classes, while ours allows complex structures of interacting component to be easily manipulated. This provides the degree of flexibility required to build tailorable groupware. Our model should not be seen as a replacement for those models, but rather an extension.

The composition filter model described in [1], and the mediator-adaptor model presented in [14], also relate to ours. They both provide a way of composing functionality. The key difference is that our model does not enforce a particular architectural style; it provides only mechanisms that allow the programmer to compose functionality in these, or other ways. In this sense, our model is more general.

Paul Dourish's PROSPERO groupware platform also relies on open implementations concepts [2]. It allows the application to change the behavior of the system in important ways, but always maintaining the overall system structure. Regarding architectural flexibility, our model goes further than that. It

provides structuring abstractions that allow tailorable groupware systems to be easily built.

Summary

In this paper, we argued for the need of tailorability in groupware systems and for the reevaluation of the software technologies used to build them. We discussed the issues and solutions involved in the design of shared workspace management services, and concluded that no single, complete set of services can be used in all cases. We continued by reviewing emerging software technologies that promise to help in the design and implementation of tailorable groupware. We discussed issues involved in component architectures, the use of meta-level reflective architectures to create open implementations, and the aspect-oriented approach to programming. Finally, we have presented a reflective component model, and show how it can used to build tailorable groupware. We hope that it motivates researchers to experiment with the technologies described in this paper.

REFERENCES

1 M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In O. N. R. Guerraoui and M. Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, LNCS 791, pages 152–184. Springer-Verlag, 1994.

2 P. Dourish. Developing a reflective model of collaborative systems. *ACM Transactions on Computer-Human Interaction*, 2(1):40–63, 1995.

3 C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD RECORD*, 18(2), June 1989. Also published in/as: 19 ACM SIGMOD Conf. on the Management of Data, (Portland OR), May.-Jun.1989.

4 I. R. Forman, S. Danforth, and H. Madduri. Composition of before/after metaclasses in SOM. *ACM SIGPLAN Notices*, 29(10):427–??, 1994.

5 D. Garlan and M. Shaw. An introduction to software architecture. Technical Report CS-94-166, Carnegie Mellon University, School of Computer Science, Jan. 1994.

6 S. Greenberg and D. Marwood. Real time groupware as a distributed system : Concurrency, control and its effect on the interface. In *Proceedings of CSCW '94*, 1994.

7 G. Kiczales. Towards a new model of abstraction in software enginnering. *Proceedings of the International Workshop on New Models for Sofware Architecture — Reflection and Meta-Level Architectures, Tokyo, Japan*, 28, Nov. 1992.

8 G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es):154–154, Dec. 1996.

9 G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

10 G. Kiczales and A. Paepcke. Open implementations and metaobject protocols. Expanded tutorial notes, 1994. At http://db.stanford.edu/ paepcke/shared-documents/Tutorial.ps.

11 C. V. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010 P9710047, XEROX Corporation, Feb. 1997.

12 J. P. Munson and P. Dewan. A flexible object merging framework. In *Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work*, Technologies for Sharing I, pages 231–242, 1994.

13 O. Stiemerling. Supporting tailorability of groupware through component architectures. In *Proceedings of the Workshop on Object-Oriented Groupware Platforms, 5th ECSCW*, pages 54–60, 1997.

14 A. Syri. Tailoring cooperation support thorough mediators. In *Proceedings of Fifth ECSCW*, pages 157–172, 1997.

15 C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, Reading, 1998.

16 C. Zimmermann. *Advances in Object-Oriented MetaLevel Architectures and Reflection*. CRC Press, 1996.