Process Model Customization for Technical and Non-Technical Users

Technical Report UCI 94-60

March 2, 1994

Patrick S. Young and Richard N. Taylor

pyoung@ics.uci.edu taylor@ics.uci.edu

(714) 856-4101 (714) 856-6429

Department of Information and Computer Science
University of California, Irvine CA 92717-3425[1]

Process Model Customization for Technical and Non-Technical Users

Technical Report UCI 94-60

March 2, 1994

Patrick S. Young and Richard N. Taylor

pyoung@ics.uci.edu taylor@ics.uci.edu

(714) 856-4101 (714) 856-6429

Department of Information and Computer Science
University of California, Irvine CA 92717-3425[1]

This paper focuses on two important requirements which must be satisfied before widespread use of process programming in industrial settings becomes a reality. First, a process programming language must be customizable, allowing the user to fit the language into the existing environment, rather than requiring the user to change existing work environments, work procedures, or corporate culture to meet the language's worldview. Second, the process programming language should be accessible to all project personnel, both technical and non-technical. This paper presents the Teamware category object model, a new object model which has been developed to help meet these key requirements. This object model supports development of customized activity types, resource types, and artifact types to support the needs of a particular corporation or project. These new types are presented to non-technical end users as part of a "pre-existing" language. The paper shows how the model differs from traditional class systems. It also compares the customization support provided in Teamware with that of other existing process systems and shows how Teamware's category model allows definition of a higher level of abstraction.

Process programming has the potential to improve the software process, reduce product development times, and increase product quality. A number of vital research issues must be addressed, however, before process programming can deliver on these promises. This paper focuses on two of these issues: providing support for corporate- and project-specific

1. This material is based upon work sponsored by the Advanced Research Projects Agency under Grant Number MDA972-91-J-1010. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

customization and making process programming accessible to both technical and non-technical users.

The processes used in different companies and in different projects vary considerably. Work procedures may differ depending on both corporate culture and project size and complexity. Some companies, for example, may want a tightly-controlled process with management involvement at each step, while others may support a more laissez-faire approach. Development environments will differ. Some projects may require extensive interaction with a wide range of sophisticated tools, while others may work with a bare-bones compiler and file system. Moreover the amounts and types of information specified as part of the process will vary. Companies or personnel with considerable process experience may want to store extensive information including an activity's tool interactions and pre- and post-conditions. In contrast, users just beginning to work with process concepts may want to limit their process specifications to simple activity descriptions, listing only activity inputs and outputs. Because individual project needs cannot be anticipated in advance, a process programming system should be customizable. Users should be able to extend the system to support different company work procedures, including differences in reporting procedures and communications. The language should allow the user to specify as much or as little information as desired.

Software engineering projects involve both technical and non-technical workers. If we expect a process programming language and system to document and guide the development process, it must be accessible to all project personnel including managers, programmers, customer representatives, domain experts, and support staff (e.g., marketing, accounting). Many of those involved will not be able to spend the time and effort necessary to learn a complex new programming language, and some may lack the necessary background.

Unfortunately, while a wide variety of important problems are being studied by the process community (e.g., dynamism, analysis), these two issues remain largely unaddressed. However, both issues are vital to successful widespread introduction of process programming in industrial work environments.

The Teamware process programming system is designed to improve coordination and control of software engineering teams. It allows users to define a prescriptive specification of the software development process and supports process enactment. Teamware provides approximately the same level of detail as Process Weaver [1], FUNSOFT [2], and SLANG [3]. This paper describes the Teamware category object model—one component of Teamware coordination and control solution. This new object model allows development of corporate- or project-specific activity, artifact, and resource types. While a variety of complex behavior can be defined in these types, the category object model's abstraction mechanisms provide a method for hiding this complexity. This allows the category object model to support both technical and non-technical users.

The paper begins with a brief overview of the Teamware language and system. This is followed by a detailed discussion of the category object model. The discussion includes a technical description of the category object model, a comparison between the category

model and other traditional object models (e.g., classes), and a detailed example. A survey of related research follows. The paper concludes with a discussion of validation activities and a summary.

1.0 Overview of Teamware

Some understanding of Teamware is necessary in order to provide a context for discussing the category object model. This section provides a brief overview of the Teamware language, system, and users. A more detailed description can be found in [4].

1.1 Teamware Language

The Teamware *language* allows users to formally specify their software development process. It includes constructs for defining the activities, resources, and artifacts of the software development process and their inter-relationships.

Activities, Resources, and Artifacts

Activities in Teamware represent an action or set of actions which are carried out as part of the software development process. Activities described in the Teamware language include information on the artifacts used by the activity, the artifacts produced by the activity, and the resources needed to carry out the activity. As we shall see, additional information (e.g., pre- and post-conditions, tools) may be provided. Typically, the actions will be carried out by one or more team members, although, an activity may also be completely automated.

Teamware resources represent resources in the classic management sense of the word. They are corporate personnel and assets which have been assigned to the project and which in turn are assigned to carry out or enable individual Teamware activities. Teamware resources can represent consumable assets—a budget, for example—or may represent fixed or renewable assets—project personnel or meeting rooms, for example.

Artifacts represent objects provided as inputs to the project (e.g., a request for proposal) or generated by the activities in the project (e.g., design documents, code, test cases). Artifacts combine both the actual data produced and information about the data (e.g., ownership, versioning information).

Activities, resources, and artifacts in Teamware are based on the category object model which will be described in Section 2.0.

Activity Networks

Teamware defines key relationship between activities, resources, and artifacts through the use of activity networks. An activity network defines the relationship between a parent activity and its subactivities, the control relationships between the subactivities, definition

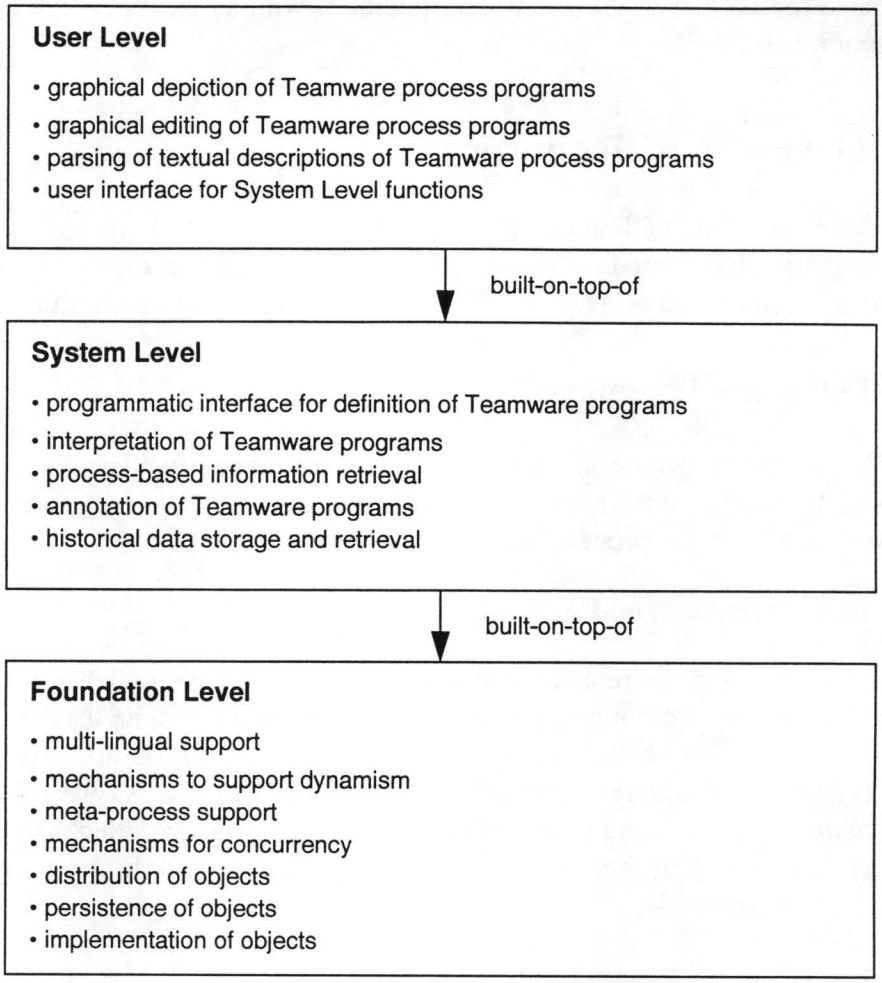

FIGURE 1 The Teamware Multi-Level Design

of the data flows between the parent activities and subactivities, data flows between the subactivities, and assignment of resources from the parent activity to subactivities.

1.2 The Teamware System

The Teamware *system* has three distinct components (see Figure 1). For the discussion in this paper, only the middle *system level* is important. The system level provides the following capabilities:

Process Definition — The system level provides a programmatic interface to allow definition and modification of Teamware programs. This interface can be accessed either by a parser, to load Teamware programs from a text file, or by a user interface, providing users with interactive program definition and modification.

Process Interpretation — The system level includes an interpreter for Teamware programs. This interpreter uses the Teamware program to coordinate execution of the actual software development process. The interpreter interacts with outside agents

(including team members) to guide them in carrying out the process as specified. It also handles routing of information and artifacts to appropriate personnel or tools. Deviations from the expected process are noted and appropriate agents (e.g., managers) are notified.

Process-Based Information Retrieval — The system level supports information retrieval based on the Teamware process programs. For example, given an activity, the system level can retrieve its associated artifacts and resources.

Annotation of Teamware Programs — In addition to retrieval of existing process-based information, the Teamware system also supports annotation of Teamware programs. This facility supports, in particular, the attachment of activity-specific computer-supported cooperative work artifacts (e.g., attachment of e-mail and computer conferences to activities).

Historical Data Storage and Retrieval — The system level supports automated archiving of process information during the interpretation of the process program and provides services for retrieving this information for use in later analysis.

1.3 Teamware Users

Three distinct groups of users are identified:

System Programmers — A number of *technical* users, called system programmers, customize the Teamware system to meet individual project and corporate needs. The system programmers integrate Teamware with the pre-existing development environment and tools. These system programmers are experts both in Teamware and in the installation-specific environment and tools with which they are integrating Teamware.

Process Programmers — Process programmers are the many users who use Teamware to specify software development processes. Process programmers may or may not have a technical background. They include managers, corporate process experts, and individual team members. Process programmers may also be system programmers.

Process Executors — Once specified, the software development process is carried out by members of the software development team. These users interact with Teamware to find out their work assignments, to find the artifacts, tools, and resources involved in each work assignment, and to understand the overall project along with their position within the process. This group, called process executors, may include members who are also process and system programmers.

The role of each type of user, and in particular the distinction between system and process programmer will become clearer as the details of the Teamware category object model are provided in the remainder of this paper.

2.0 The Category Object Model

The desire to support non-technical users while allowing corporate- and project-specific customizations is, to some extent, contradictory. Mechanisms allowing users to customize many aspects of a system are likely to add considerable complexity. Teamware's category object model is designed to address this tension.

This section begins with an overview of the category object model. This is followed by a technical description of the category model. The model is then compared to existing object models such as class and metaclass models. An extended example illustrates use of the category object model to support activities. The section concludes by briefly discussing use of categories to support resources and artifacts.

2.1 Overview of the Category Object Model

The category model can be understood by analogy with two types of abstractions found in programming. Language designers provide one set of abstractions by developing a high-level language. These abstractions simplify the task of programmers using the language. Programmers, while defining programs, develop a second set of abstractions by defining procedures (procedural abstraction) and data types (data abstraction). Teamware provides mechanisms for defining both these levels of abstractions.

The Teamware *system programmer* defines sets of activity, resource, and artifact *categories*. These categories act as corporate- or project-specific abstractions. To the non-technical *process programmer*, they appear to be elements of the process programming language. Thus, for example, a Teamware system programmer might define three separate activity categories, one for general assigned activities, one for meeting activities, and one for review activities. The meeting activity category might encapsulate interaction with a corporate meeting room scheduling program or interaction with team members' personal calendar or personal information manager (PIM) systems. The review activity might interact with the project's electronic mail system to automatically forward documents to participants for review. In each case, the behavior of the category is defined by the system programmer. From the process programmer's point-of-view, however, these three activity categories are parts of a pre-defined programming language—the implementation of their behavior is hidden from the process programmer.

Teamware provides a second set of abstractions, called specifications. This level of abstraction corresponds to the procedures and abstract data types an application programmer defines as part of a program. The Teamware *process programmer* can define an activity, resource, or artifact *specification* and use that specification in different places within the process. Thus, for example, the process programmer could define an "implement-module" activity specification. This specification would define the inputs, outputs, and resources needed to implement a module. It might also include an activity network defining the sub-activities needed to carry out the implementation. The process programmer could then use this specification in different parts of the process, just as he or she would define a procedure in a traditional programming language and then call that procedure from different parts of a program. The same "implement-module" specification

could be used as the activity definition for the implementation of a system's "database" module, the implementation of its "user interface" module, and the implementation of its "arithmetic functions" module.

2.2 Technical Description of Category Object Model

The category object model has three major components: categories, specifications, and instances. There is a one-to-many relationship between a category and a set of specifications, and a one-to-many relationship between a specification and a set of instances. A given specification is said to be *based* on a particular category. Similarly an instance is said to be *based* on a particular specification.

2.2.1 Instances

Each object instance has an associated set of data values. These values are organized into fields which are further divided into typed elements. Each element corresponds to a single data value of the specified type. Each field can contain zero or more data elements.

Object instances communicate through the use of events and requests. In Teamware, both events and requests provide point-to-point communications—each has a single sender and a single recipient. Events occur asynchronously and allow data to pass from sender to recipient. An event call can be viewed as a procedure call with input parameters only. Requests are synchronous and data can pass from recipient to sender as well as from sender to recipient. A request call can be viewed as a procedure call with both input and output parameters.

2.2.2 Categories

Each category provides zero or more field *schemas* in order to define the fields present in each corresponding instance. Field schemas can be one of three kinds:

Single-Element — A single-element field schema signifies that a field with a single element may be present in each instance. Each single-element field schema must include a field name, a field family (see below), and a field type. The schema also indicates whether the field is required (i.e., present in all instances) or optional. In addition the schema may include a default value.

Shared-Element — A shared-element field schema signifies that a single-element field may be present in each instance. However, in this case, the value of the single-element field will be shared by all instances that are based on the same *specification* (see Section 2.2.3). Each shared-element field schema must include a name, a field family, and a field type. It should indicate if the field is optional and may also provide a default value.

Multi-Element — A multi-element field schema signifies that a field with multiple elements will be present in each instance. The actual elements present in an instance is determined by the instance's specification. Each multi-element field schema must include a field name, a field family, and a field type.

Each field's *name* is used to access the field and the element or elements within the field. A field's *type* limits the type of values which can be stored in the field. The field *family* determines how the Teamware interpreter interacts with the field and also acts as a secondary means of accessing the element or elements within the field.

A category defines the event and request handling routines that its corresponding instances will use in order to respond to event and request messages. These handlers are procedures written in languages integrated with the Teamware system (Common Lisp in the current prototype). The event or request handler can perform the following actions:

- Access or modify the data element or elements in a given data field. Note, however, that the event or request handler is not familiar with individual element names, only field names. It can access the element corresponding to a single-element field, or it can iteratively perform the same action on all elements in a multi-element field. It cannot, however, selectively access or modify a particular individual element in a multi-element field.
- Access all data elements in fields based on a given field family and iteratively perform the same action on them.
- Send events or requests to other objects (assuming that the instance has access to the object).
- Explicitly call an event or request handler defined in one of the category's supercategories (see discussion on inheritance below).
- Perform typical actions in the handler's language.

Categories form a multiple-inheritance hierarchy. Each category is based on zero or more parent categories. The parent categories for a given child category are strictly ordered and form a category precedence list similar to CLOS' class precedence list [5]. A category is said to be a *subcategory* of one or more *supercategories*. The multiple-inheritance hierarchy must form a directed acyclic graph (DAG). A subcategory inherits all fields defined in its supercategories. If multiple fields are defined with the same name, the field from the highest precedence supercategory is used. A subcategory also inherits the event and request handlers of its supercategories. When an instance receives an event or request, its category is checked for an appropriate handler. If none is defined for the category itself, the category's supercategories are checked, beginning with the highest precedence category. The first handler found is used.

2.2.3 Specifications

Each *specification* is based on a category. The specification provides a field *definition* for each field *schema* in its corresponding category. For a single-element field, if the field is optional as described in the category, the specification determines whether or not the field will be present in corresponding instances. For a multi-element field, the specification defines the actual elements which will be present. The specification provides a name and a type for each element and may additionally provide a default value. The specification may also provide values for shared-element fields defined in the category.

2.3 Comparison between Categories and Other Object Models

The category object model differs considerably from existing object models.

In a traditional class model an instance's data and behavior are both determined by its corresponding class. C++ [6], Smalltalk-80 [7], and Object Pascal [8] all follow this model. In contrast, in the category model an instance's behavior is determined by its category, but much of its data is determined by its specification. A category is also different from a metaclass. In a class model with metaclasses (e.g., Smalltalk-80), each class is itself an instance of a metaclass. At first glance, it appears that Teamware categories can be mapped to metaclasses and Teamware specifications can be mapped to classes. However, the analogy breaks down. Teamware specifications are designed to provide the non-technical user with a simple model and thus do not define behavior. As a consequence, they cannot be mapped to the classes designed for use by traditional programmers.

CLOS [5] follows a different approach. In CLOS, a class defines only the data found in instances. In CLOS messages are not sent to particular recipients, instead a function is called and the response is determined by the combined classes of its arguments. Teamware behavior is organized by categories. Thus Teamware's behavioral model is much more similar to the message/recipient model found in traditional classes.

Prototype-based models provide an alternative to the class approach. In a prototype language (e.g., Self [9], Garnet [10]), all data and behavioral information is stored in instances. No classes or metaclasses are provided. The category object model is much closer to class models as it includes categories and specifications which combine to provide the definition for a set of instances.

2.4 Extended Example

The category object model will now be illustrated with an extended example. Section 2.4.1 defines an activity category, Section 2.4.2 presents a specification based on the category, and Section 2.4.3 describes an instance based on the specification.

2.4.1 Time-Limited Activity Category

The `time-limited-activity` category supports activities which must be completed within a preset time limit. If the activities are not completed within the prescribed time limit, the manager responsible for the activity is notified. The category's fields and events and event handlers are summarized in Figure 2—this category has no requests or request handlers.

Due to space restrictions, in this example, the `time-limited-activity` category is defined as a stand alone category. In practice, the `time-limited-activity` category would most likely be a subcategory of a more general `assigned-activity` category. The `time-limited-activity` category would inherit most of its fields and event handlers from the `assigned-activity` category, adding only timing specific information and actions.

Field Definition

The category uses a number of pre-defined field families (among several defined for all activity categories):

Input-Family — Elements in `input-family` fields represent artifacts developed outside of an activity which are needed to carry out the activity.

Output-Family — Elements in `output-family` fields represent artifacts produced (or modified) by an activity.

Resource-Family — Elements in `resource-family` fields represent resources assigned to the activity which are used to carry out the activity.

Other-Family — Elements in `other-family` fields are not directly accessed or modified by the Teamware interpreter, but can be used by a category's event and request handlers. However, as we shall see below, these handlers can explicitly pass the field's contents to the interpreter.

Activity networks define the data flow relationships between an activity's inputs (as determined by its fields based on `input-family`) and another activity's outputs (as determined by its fields based on `output-family`). Activity networks also define the relationship between the resources assigned to a parent activity (as determined by its fields based on `resource-family`) and those assigned to its subactivities. During process execution the Teamware interpreter ensures that each activity instance receives the proper data and is assigned the appropriate resources as determined by the activity network.

In addition to the pre-defined families above, a fifth family, the `manager-family`, is defined for use in this example. This new family is added by the system programmer using Teamware's family definition mechanisms (see [11] for more information on these mechanisms). The `manager-family` inherits much of the characteristics of the `resource-family`—the activity network can define the relationship between the managers assigned to a parent activity and the managers responsible for each subactivity. However fields based on the `manager-family` are distinct from fields based on the `resource-family`; while a manager is responsible for an activity, he or she does not necessarily participate in the activity itself.

`Time-limited-activity` category's fields can now be defined. Two specific data elements must be present in every activity based on the `time-limited-activity` category, they are (1) the manager of the activity and (2) the time limit of the activity. Because these specific elements must be present, they will be represented by single-element fields. The `manager` field will be of `manager-family` field family and type `team-member`. The `time-limit` will be of `other-family` field family and type `integer`. Both fields are required.

In addition to these specific data elements, activities based on the `time-limited-activity` category will also have inputs, outputs, and personnel. However, in contrast to the manager and time limit, the actual input, output, and personnel elements will vary from activity specification to activity specification. Therefore, rather than representing them as single-element fields, they will be represented by multi-element fields. The `inputs`,

Category: Limited-Time-Activity

based on: None

Fields

Field Name	Family	Single/ Multi	Type of Element	Comment
Manager	Manager	Single	Team-Member	Manager responsible for activity.
Time-Limit	Other	Single	Integer	Time alloted before manager informed of potential problem.
Inputs	Input	Multi	Artifact	List of non-specialized input artifacts.
Outputs	Output	Multi	Artifact	List of non-specialized output artifacts.
Personnel	Resource	Multi	Team-Member	List of team members.
Subnetwork	Other	Shared (optional)	Activity-Network	Activity network subdefinition for activity.

Event Handlers

Event Name	Actions
Enabled	(1) Send Register-Alarm event to alarm system. (2) Send self Execute event.
Execute	(1) Send resources Handle-Executing event. (2) Check for subnetwork. If network exists, execute.
Completed	(1) Send resources Handle-Completing event. (2) Send Deactivate-Alarm event to alarm system.
Alarm-Triggered	Send e-mail message to Manager.

FIGURE 2 Limited-Time-Activity Category

`outputs`, and `resources` fields are of `input-family`, `output-family`, and `resource-family` and types `artifact`, `artifact`, and `team-member` respectively.

Activities based on the `time-limited-activity` category may be further subdefined with an activity network. In this simple example, the network used will be the same for all activities based on the same specification. The network is therefore represented as a shared-element field. The `subnetwork` field is of the `other-family` field family and type `activity-network`. Because not all activity specifications will have a subdefinition (some may be simple enough to be carried out directly) the field is optional. Note that the one subnetwork per specification limitation is the choice of the category writer and is not built into Teamware. The category could be defined, for example, to allow multiple subnetworks for each specification with the actual network used chosen at run-time depending on the activity's inputs.

Event and Event Handler Definition

Teamware interpreter supports a number of pre-defined events. The following three are relevant for this example:

Enabled — Receipt of this event indicates that the activity *may* begin execution. Depending on the event handler defined, activities which receive an `enabled` event may begin execution or may perform some other action (informing the manager or checking pre-conditions, for example).

Execute — Receipt of this event indicates that the activity *should* begin execution. The event handler defined should take whatever actions are necessary to ensure that the activity is properly executed. The `execute` event handler might, for example, inform team members that they should begin an activity or it might, for a fully automatable activity, carry out the execution.

Completed — The activity receives this event when the activity has been completed. The `completed` event handler should perform cleanup and handle any other actions needed to conclude the activity (for example, releasing allocated resources and, if desired, informing managers that the activity has been completed). This event may be generated by a team member, signifying completion of an assigned activity, or by an external process responsible for carrying out a fully automated activity.

In addition to the support provided by Teamware itself, the `Time-Limited-Activity` category depends on the existence of two external systems. First, it requires the services of an electronic mail system.[2] For the purposes of this example, we assume that a system programmer has integrated the company's e-mail system with Teamware, and that a Teamware event can be used to send e-mail to a given team member. Second, because Teamware does not include intrinsic timing constructs, an alarm system must be defined. This example assumes that the system programmer has built a simple alarm system. This system accepts a `register-alarm` message which includes parameters for the length of time before an alarm should be sounded and the recipient of the alarm. In addition, the alarm system accepts a `deactivate-alarm` message which removes a previously registered alarm. When an alarm is triggered, the alarm system sends an `alarm-triggered` event to the designated recipient. Standard operating system services such as UNIX's cron can be used to implement these services.

The `Time-Limited-Activity` category's event handlers can now be defined. The `enabled` event handler sends a `register-alarm` message to the alarm system with the value of the instance's single-element `time-limit` field as the length of time and designating the activity instance itself as the recipient of the `alarm-triggered` message. The event handler then sends the activity itself an `execute` event. Upon receiving an `execute` event, the activity's `execute` event handler sends each element in the `resources` multi-element field a `handle-executing` event. The resources' response to this event will vary from resource type to resource type (i.e., a team member resource will respond differently from a budget resource) with the actual response determined by the resource's category. The `execute` event handler then checks if the activity has a `subnetwork` shared-element field. If it does, the event handler explicitly passes the subnetwork to the Teamware interpreter for execution, otherwise the resources (e.g., team members) are solely responsible for carrying out the activity. When the activity receives a `completed` event, it notifies each resource in the `resource` multi-element field and then sends the alarm system a `deactivate-alarm` event. Finally, if the activity receives an

2. Different organizations will use different e-mail systems. Rather than forcing a specific mail system on an organization, Teamware allows system programmers to integrate Teamware with a company's existing e-mail system. The e-mail system is therefore considered external to Teamware.

Specification: Implement-Module

of Category: Time-Limited-Activity

Field Name	Element Name	Element Type
Manager	Manager	Team-Member
Time-Limit	Time-Limit	Integer
Inputs	Requirements	Text File
Outputs	Code	Text File
	Test-Results	Text File
Personnel	Chief-Programmer	Team-Member
	Asst-Programmer	Team-Member
	QA-Programmer	Team-Member
Subnetwork	Subnetwork	Activity-Network

FIGURE 3 Implement-Module Activity Specification

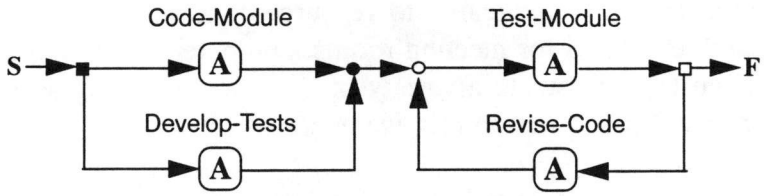

FIGURE 4 Graphical Representation of Implement-Module Activity Network

`alarm-triggered` event, it accesses the value of the `manager` single-element field and sends an e-mail message noting the passage of the allotted time.

2.4.2 Implement-Module Specification

The `Implement-Module` specification defines how modules will be implemented in a given project (see Figure 3). It is based on the `Time-Limited-Activity` category. The process programmer fills in the elements of each multi-element field. Activity instances based on the specification will have a single input (`Requirements`), two outputs (`Code` and `Test-Results`), and three personnel resources (`Chief-Programmer`, `Asst-Programmer`, and `QA-Programmer`). In addition the `Implement-Module` specification defines the value of the `Subnetwork` shared-element field. This network is graphically depicted, sans data flow and resource assignments, in Figure 4. Space does not allow discussion of issues involved or techniques used in Teamware which enable non-technical users to effectively build a subnetwork (see [4]).

2.4.3 Database Module Implementation Instance

Figure 5 shows an instance based on the `Implement-Module` specification. The instance has no name because instances are anonymous unless the category explicitly includes a name field. The instance defines the actual manager, time limit, input artifacts, output artifacts, and personnel associated with a module implementation activity.

Instance of Implement-Module

Field Name	Element Name	Element Value
Manager	Manager	Mike-Brookings
Time-Limit	Time-Limit	14 days
Inputs	Requirements	system/reqs/data.doc
Outputs	Code	system/code/data.ada
	Test-Results	system/tests/data.test
Personnel	Chief-Programmer	Chloe-Szkrybalo
	Asst-Programmer	Maria-Hernandez
	QA-Programmer	George-Sommerfield

FIGURE 5 An Activity based on the Implement-Module Activity Specification

2.5 Resource and Artifact Categories

While the previous example illustrated use of the category object model for an activity, the category model is also applicable to resources and artifacts. Resource categories, for example, can be defined for meeting rooms, computers, and budgets. Artifact categories can be defined to encapsulate a variety of behavior including security access (e.g., top-secret, secret) and configuration management.

3.0 Comparison with Other Process Systems

In the previous section, we have seen how the Teamware category object model supports definition of corporate- and project-specific activity, resource, and artifact types and how these types are made accessible to non-technical users. In this section the support provided by Teamware is compared to that found in other similar process systems. The section begins with an example highlighting the difference between Teamware's category support for customization and the more limited support found in some graph-based process languages. This is followed by a set of brief comparisons with specific systems.

3.1 System Comparison Example

The difference between Teamware's approach to customization and that found in other systems can best be seen through an example. Figure 6 (taken from [12]) shows part of a SLANG network defining a module coding activity. This

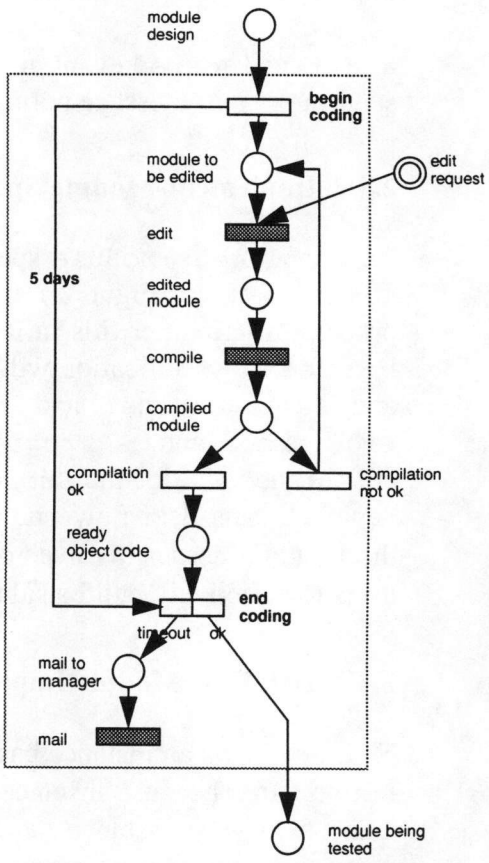

FIGURE 6 SLANG Network

network provides a good example of the kinds of customization supported by Process Weaver, SLANG, or FUNSOFT. The SLANG network represents a process to implement a module. If the module implementation has not been completed in 5 days, the manager will be sent e-mail. The network has a number of major limitations which are addressed by the Teamware category object model.

The category object model provides better support for process reuse. For example, the "limited time activity followed by e-mail to manager" behavior may be needed in several different places in a process. In Process Weaver, SLANG, and FUNSOFT, repeating this behavior in a different context requires the process specifier to duplicate the network in an editor, and then to modify the copied network to fit the new situation. This procedure must be repeated for every network inheriting the timing and e-mail behavior. Teamware category mechanism, however, provides a method for encapsulating the behavior. A single category can be defined and then reused in multiple contexts.

In both SLANG and FUNSOFT, the process specifier must be familiar with an traditional programming language (e.g., C) in order to define the shaded transitions which invoke the editor and compiler and to send e-mail. In Teamware, the tool invocation and e-mail access can be encapsulated in an activity category (see [4] for an example of generic tool invocation). The process specifier can simply specify the tools to execute and the team member to which the e-mail should be sent, rather than worrying about the details of tool invocation and e-mail system integration.

Process Weaver provides a number of pre-defined constructs including tool invocation and interaction using a superset of the HP Softbench protocols. However, Process Weaver does not provide a means of defining additional constructs at the Process Weaver language level. Additional constructs can only be defined and accessed using the Co-Shell language—a language comparable to UNIX shell languages in complexity. A timing system, for example, is not provided in Process Weaver. An external system (similar to that described in Section 2.4.1) can be defined in order handle this behavior. However, interaction with this system will require the process specifier to directly access the timing system via Co-Shell code. As we have seen in the `limited-time-activity` category example, Teamware's category mechanism provides a level of abstraction which supports time-limited behavior while hiding the timing system from the process programmer.

3.2 Additional Comparisons

This section describes additional similarities and differences between Teamware and three graph-based process programming systems. In addition, Teamware is compared to PML, an object-oriented process programming system.

Process Weaver

Process Weaver's default templates for terminal activities (see [13]) provide a few of the capabilities of Teamware categories. Process Weaver, however, only supports the definition of a single template, while Teamware provides for a potentially infinite number

of templates. Moreover, Process Weaver's default template only supports terminal activities while Teamware categories can be used at any level in the activity hierarchy.

Process Weaver resources are limited to human team member; no support is provided for defining new resource types. Process Weaver does not provide facilities for defining artifact behavior comparable to Teamware artifact categories.

FUNSOFT

FUNSOFT's separation of jobs from agencies provides a first step towards supporting abstraction on Teamware's level. However, as with Process Weaver, this only occurs at the terminal activity level (as refined agencies do not have corresponding jobs). Moreover, FUNSOFT jobs combine the properties of Teamware's categories and specifications into a single entity. This means that either the manager works with a completely pre-defined set of jobs, or the manager must be capable of defining new jobs. Unfortunately, the latter option, as mentioned in Section 3.1, requires managers to be conversant in C. In contrast, Teamware's activity specifications allow managers to define new "jobs" as long as their behavior fits into that of a pre-existing category.

FUNSOFT only supports human resources. These resources are represented as roles, which are essentially placeholders with no associated data or behavior beyond activity assignment. FUNSOFT artifact types specify data only, no provisions are made for defining custom behavior.

SLANG

SLANG does not explicitly represent resources (including team members). SLANG artifacts are based on an object-oriented model and thus can include definition of data and behavior. SLANG's object model, however, is based on traditional classes. No provisions are made for shielding non-technical users from the complexities of behavioral definition.

PML

In contrast to the previously mentioned systems, PML [14] is object-oriented, rather than graph-based. PML does not have activities in the same sense as the previously mentioned systems. Instead, each resource (role, in PML terminology), has a set number of predefined activities (called actions) which it can perform. Unfortunately, definition of each role requires the process specifier to understand a complex language—thus making PML inappropriate for non-technical process specifiers. PML roles, can however, support definition of both human and non-human resources. PML artifacts (called entities) appear to support definition of data, but not behavior.

4.0 Validation Exercises

In order to validate the category object model, a number of typical usage scenarios were identified and categories were then developed or modified to address the needs of each scenario.

Teamware's ability to automate parts of the process, for example, was demonstrated in the development of a meeting activity category and meeting room resource category which handle automatic scheduling of meetings, a review activity category which automatically forwards review documents to team members, and an activity category which supports fully automated activities. Teamware's ability to support a variety of organizations, ranging from egoless groups to tightly-managed teams, was explored through development of a variety of categories. These activity categories exhibit different reporting relationships, depending on group structure. Categories supporting tool invocation show how the category model can be used to hide details of the work environment from the process programmer. Activity categories supporting pre-conditions demonstrate Teamware's ability to provide different amounts of process specification sophistication.

These categories have been implemented in a prototype of the Teamware system built on top of Common Lisp.

5.0 Conclusions

The Teamware category object model provides a means of defining corporate- or project-specific custom behavior. The category abstraction hides implementation details of complex behavior from non-technical process specifiers. This level of abstraction is not found in existing process support systems. In addition, Teamware extends the work in existing graph-based process system by allowing the user to define new resources and artifact behaviors.

[1] C. Fernström. Process Weaver: Adding Process Support to UNIX. In *Proceedings of the Second International Conference on the Software Process*, 1993.

[2] Volker Gruhn. *Validation and Verification of Software Process Models.* Ph.D. Thesis, University of Dortmund, 1991.

[3] S. Bandinelli, A. Fuggetta, and S. Grigolli. Process Modeling in-the-large with SLANG. In *Proceedings of the Second International Conference on the Software Process*, 1993.

[4] P.S. Young. *Customizable Process Specification and Enactment for Technical and Non-Technical Users.* Ph.D. Thesis, Department of Information and Computer Science, University of California, Irvine, 1994.

[5] G.L. Steele, Jr. *Common Lisp: The Language, Second Edition.* Digital Press, 1990.

[6] B. Stroustrup. *The C++ Programming Language, Second Edition.* Addison-Wesley, 1991.

[7] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison Wesley, 1983.

[8] *Think Pascal Object-Oriented Programming Manual.* Symantec Corporation 1991.

[9] D. Ungar and R.B. Smith. Self: The Power of Simplicity. In *OOPSLA '87 Proceedings*, 1987.

[10] B.A. Myers, D.A. Giuse, and B.V. Zanden. Declarative Programming in a Prototype-Instance System: Object-Oriented Programming without Writing Methods. In

OOPSLA '92 Proceedings, 1992.

[11] P.S. Young. *The Teamware Language Reference Manual, Version 1*, 1994.

[12] S. Bandinelli, A. Fuggeta, C. Ghezzi, and S. Grigolli. Process Enactment in SPADE. In *Proccedings of the 2nd European Workshop on Software Process Technology*, 1992.

[13] *Process Weaver: General Information Manual*, Cap Gemini Innovation, 1992.

[14] R.F. Bruynooghe, J.M. Parker, and J.S. Rowles. PSS: A System for Process Enactment. In *Proceedings of the First International Conference on the Software Process*, 1991.