An Adaptable Generation Approach to Agenda Management

Eric K. McCall, Lori A. Clarke, Leon J. Osterweil
University of Massachusetts
Amherst, MA 01003, USA
+1 413 545 2013
{mccall, clarke, ljo}@cs.umass.edu

ABSTRACT
As software engineering efforts move to more complex, distributed environments, coordinating the activities of people and tools becomes very important. While groupware systems address user level communication needs and distributed computing technologies address tool level communication needs, few attempts have been made to synthesize the common needs of both. This paper describes our attempt to do exactly that.

We describe a framework for generating an agenda management system (AMS) from a specification of the system's requirements. The framework can support a variety of AMS requirements and produces a customized AMS that is appropriate for use by both humans and software tools. The framework and generated system support evolution in several ways, allowing existing systems to be extended as requirements change. We also describe our experiences using this approach to create an AMS that supports a process programming environment.

KEYWORDS
Agenda management, process programming, cooperative work, tool integration

1 Introduction

Modern computing systems are increasingly viewed as collaborations among groups of humans and software systems whose work must be shared and coordinated. There has been a considerable amount of work on approaches for facilitating this coordination. But none of these approaches seems to provide the full range of capabilities needed to support this next generation of computing systems.

One common direction of this work has focused on how to support interoperability among systems. Such approaches have generally focused on low-level interprocess communication protocols and mechanisms. While providing a useful substrate, these mechanisms are at too low a level of abstraction to support clear exposition in application programs. We believe that higher level abstractions are needed. A higher level view of coordination and collaboration is commonly sought by the Computer Supported Cooperative Work community, but much of this work is aimed exclusively at coordination of humans.

In our work we seek to establish abstract, yet rigorously defined concepts that address coordination of both humans and computing systems. The essence of our approach is to extend and formalize the metaphor of agendas, or to-do lists. The use of agendas seems to be nearly ubiquitous, having been applied to problems in such varied domains as software engineering, the factory shop floor, and routine office work. Thus specialized instances of this metaphor have already been used to coordinate people with each other and software systems with each other. We believe that this metaphor, appropriately applied, could be effective for coordinating humans and computer systems for a wide range of applications. In this paper, we explore the problem of designing agenda management systems (AMSs) that are capable of doing just that.

To illustrate a typical context in which such an AMS might be effective, consider the following software engineering scenario for tracking and fixing software bugs. In this scenario, the bug fixing activity requires the coordination of many different people and software systems within an organization. We use this example to show that a well designed AMS can effectively coordinate the activities of programmers, testers, managers, e-mail system, configuration management software, testing software, and compilers.

> Suppose an e-mail message that contains a detailed description and stack trace information for a suspected bug in the organization's software product is sent to the organization by an off-site beta tester. Alice, the office manager, reads the message and alerts the software testers. She requests that any one of the testers verify and document the bug and, if new, send a report to the programmers so the bug may be fixed. She includes the e-mail message in this bug report. Auditing information, including the creation date and when the item was assigned to the testers, is recorded.

> Later in the day, Bob, a tester, looks at his agenda of things to do. On this agenda appears Alice's bug report. He reads Alice's instructions and the original e-mail message, and gets to work. He finds that the bug is new and is able to reproduce it. He marks the report as high priority, and forwards the report, along with current stack information, to the maintenance programmers.
>
> Catherine, a maintenance programmer, notices the arrival of Bob's annotated bug report, takes responsibility for it, and begins work on the problem it describes. She reads the email message and Bob's additional notes, and begins browsing source code. She finds that the source of the bug is a misunderstanding in the parameter orders of some routines that are similar, and decides that a standard parameter order must be agreed upon. She must now set up a meeting with Douglas, the programmer who wrote the other routines. She does this by viewing each of their agendas, finding a time when they are both free, and adding the meeting to their agendas.
>
> During the meeting, Catherine and Douglas agree on a standard parameter order, and decide to change all existing code to match their new interface specification. They will each modify the code they wrote, but Catherine will be responsible for final integration. After the meeting, Catherine requests that the configuration management (CM) tool grant her write access to all the source objects which they must change. Because one of the requested objects is already locked, the CM tool notifies Catherine that her request is pending and gives her the option of waiting for the objects to become available or retracting her request. She chooses to wait and the retract option disappears.
>
> Later, the CM tool is able to lock all of the requested source objects, so it tells Catherine the current identifiers of all the requested source objects thus granting her permission to edit them. Catherine and Douglas agree on who will modify which objects, and Catherine passes the identifiers of Douglas' objects to Douglas. They launch their editors, passing in the object names supplied by the CM tool's message.

We can assume that Catherine eventually fixes the integration problem, the CM tool moves the changes into the main source repository, and Alice is informed that the bug has been fixed.

This scenario addresses only a few activities of only a small part of the organization. Already we see that a plethora of needs, from event notification to distributed scheduling, can be met by an AMS that can effectively facilitate and coordinate interaction between the varied entitites in the organization.

Our interest in "agendas" and "agenda management" is hardly unique. Systems dealing with these ideas have appeared before. But the problems of agenda management have typically been confronted (and solved) differently for each specific system. In essence each system seems to have come up with its own, somewhat idiosyncratic, notion of agenda management and an implementation to match. In contrast, the main focus of our study is the problem of agenda management itself. We propose flexible and precise definitions of key abstractions that can then be used as the basis for the implementation of a wide variety of different systems for coordinating humans and tools in a variety of contexts.

Our approach is to provide a framework and a means of specifying and generating agenda management systems. By taking this approach we believe we can provide an important piece of communication and coordination infrastructure to system designers. Because agenda management needs change over time, generated agenda management systems must be able to evolve to meet changing system-wide requirements. Because they are used by a variety of types of users, they must be adaptable to meet differing user requirements. These issues are addressed with an iterative generation approach, flexible data structures, and decoupled user interfaces.

In this paper, we describe some typical requirements that might be encountered by agenda management system designers, and assert that a solution to the general problem of agenda management should be able to support these requirements. We then present our framework and describe its components and use. We describe a prototype implementation of the system and its use in supporting a process programming environment. Finally, we mention some related work and directions for future research.

2 Requirements

To identify the requirements for a general solution to the problem of agenda management, we informally define five entities that form the foundation of our work. These are *attribute*, *agenda item*, *agenda*, *agent*, and *view*.

An *attribute* is a <name, type, value> triple; it is the basic unit of information in an agenda management system. An *agenda item* is a collection of attributes. An *agenda* is a collection of agenda items and may also have a collection of attributes associated with it. An *agent* is a person or executable software component that can view and/or modify agendas. A *view* of agendas is a rendering of the contents of one or more agendas. Conceptually, an agenda corresponds to a set of things for an agent to do, and an agenda item is an element in that set. Clearly these concepts are broad enough to span a wide variety of specific instances of AMSs having widely varying requirements.

To illustrate the use of these terms, suppose an agent is presented with a view of all the agenda items contained in an agenda. In this view, assume the items are sorted into a list

by an associated attribute named "priority" so that agenda items containing the highest priority value appear at the top of the list. The agent may change the priority attribute of a particular agenda item, causing its position in the rendered list to change. In the scenario, when Bob sends the high-priority bug report agenda item to the programmers' agenda, it might be shown near the top of each programmer's view.

2.1 Requirements for Agenda Management Systems

To give a sense of the breadth of issues confronting an AMS designer, we now provide a list of commonly occurring requirements for AMSs, many of which might be required by any specific system, such as the one described in the bug-tracking scenario.

- **hierarchy:** support the dynamic decomposition (and composition) of agenda items into sub-items and agendas into sub-agendas
- **priority:** allow the agenda items within an agenda to be prioritized and allow priorities to be changed
- **prerequisites:** allow attributes of an agenda item to depend on the values of attributes of other agenda items or agendas.
- **alternative items:** allow an item to be composed of alternative items, meaning there is an exclusive-OR relation between them.
- **data persistence:** support varying persistence requirements, handling both transient data and data that should persist for long durations.
- **access control:** support the ability to control which agents may invoke operations on particular entities.
- **auditing:** make available a record of an entity's history (e.g., which agent created, modified, or moved an item to another agenda) and associated information (e.g., what other items depend on an item).
- **reflection:** respond to queries about the system itself.
- **a variety of views:** support the customization of views.
- **sharable:** support the sharing of entities among agents.
- **distributed, concurrent.** support distribution across different platforms and concurrent access by multiple users.
- **coherent:** guarantee coherent views of entities.
- **integration of tools and people:** support both tools and people, without making assumptions about which will be using the agenda management system.
- **evolution:** support the changing needs of its users.

In addition to usual requirements for software systems, such as scalability, openness, efficiency, and generality, this list enumerates the range of requirements that any general approach to agenda management must be prepared to address. These requirements are addressed by different parts of our approach, described in the next section.

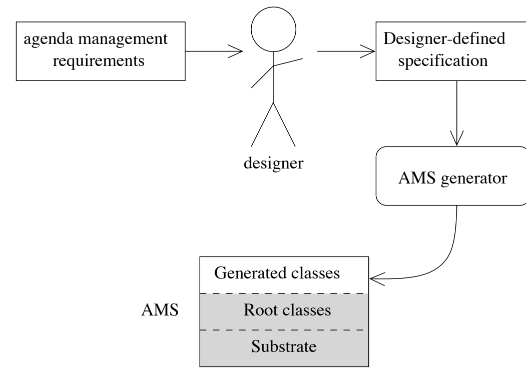

Figure 1: Agenda system creation process

3 Approach

An approach to agenda management should meet an organization's requirements while still accommodating subsequent changes in those requirements, including individual users' demands. We propose a generational approach that accomodates a range of different sorts of subsequent evolution and customization, because

- generating from a basic framework an AMS that meets an organization's requirements greatly reduces the effort required to create an agenda management system,
- ensuring that the capabilities of a generated system can be incrementally augmented (evolved) to meet additions to requirements reduces the effort required by the organization to maintain its AMS, and
- allowing users of an AMS to customize (adapt) it accommodates informal changes and specialization in the representation and use of the AMS.

Figure 1 depicts generation of a new AMS. The system designer is responsible for translating the requirements for agenda management into an appropriate specification for an AMS. The language in which the specification is written allows a designer to specify extensions to a set of basic capabilities. Once the specification is produced, it is provided to the AMS generator, represented by the rounded rectangle in figure 1. The AMS generator's function is to produce an AMS. The AMS consists of generated subclasses which extend the capabilities of several root classes, and a generic substrate. These extended capabilities meet the specification given by the designer. The extension mechanisms include an inheritance mechanism, which allows for type-strong extension of the root classes, and a policy mechanism, which provides control over the use of objects of generated classes. The root classes are the building blocks from which the more complex data structures needed by most agenda management systems are constructed. The substrate is a set of underlying facilities that support concurrency, distribution, and other fundamental needs. It may be thought of as an extended operating system library that supports agenda management. The root

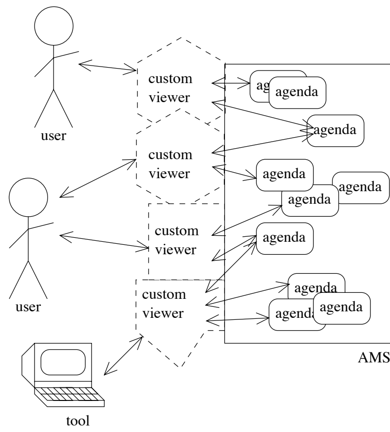

Figure 2: Agents using an adapted AMS

classes and substrate are not modified during AMS generation, but remain constant regardless of the specification. This is indicated by shading in the figure. The generated AMS is then deployed to meet the requirements originally presented to the designer. Agent specific interface code may then be linked with the application programming interface (API) to the AMS to provide agents with views of, and a means of modifying, AMS data.

For a generational approach to agenda management to be practical, it should allow the AMS to change to meet changed requirements. Our approach supports three kinds of changes: designer-specified additions to the generated AMS (evolution of the AMS), agent adaptation of the AMS, and agent view customization.

Additions to generated agenda management systems may be made by iteratively using the generation framework depicted in figure 1. To add to an existing AMS, a designer (or possibly a user acting as a designer) writes a specification that describes the additional required capabilities (in the form of additional root object subclasses) and provides this specification to the system generator. The system generator then generates additional modules that augment the AMS with new types and implementations. These modules are translated and dynamically linked with the rest of the system to provide the additional capabilities. Because the additional modules are linked dynamically, only they must be generated (hence the use of the word "incremental"), and the possibility of augmenting a running system in place exists. These kinds of changes can be characterized as global changes to the AMS because the newly generated object classes are available system-wide.

The second kind of support, agent adaptation of AMS objects, is accomplished with dynamic data structures. By providing collections, a very flexible, dynamic data structure, users have a great deal of flexibility in creating instances that are composed of existing types. An example of when this flexibility might be useful can be found in the scenario. If the AMS designer had omitted a place to store the original email report about the bug in the bug fix type of agenda item, but had created a section for arbitrary user notes, Alice could simply copy the email message into the notes field of the item. In cases where the bug was reported by phone, this "field" could be left blank with no ill effects. This kind of adaptation supports changes that require additional information for only a subset of the instances or agents associated with a particular AMS.

The third kind of support for change is a decoupling of "view" from the AMS itself: agents can be provided with a customized viewer that presents AMS information in the way each user or tool desires, as shown in figure 2. While automating this kind of adaptation is currently outside the scope of our approach, it is important to note that the approach has been designed with it in mind, and it is a logical future direction. In fact, decoupling a viewer from the AMS greatly aids efforts to treat human and software tool users of an AMS uniformly.

When a customized viewer is combined with agent-adapted AMS objects, it becomes possible for users to make local changes to the AMS that approach the power of changes made with iterative generation. If, for example, the bug testers' agenda viewers were designed to look in the user notes field of bug report items, it could find and render email messages specially, even allowing users to respond to the original email message using their favorite email systems.

The remainder of this section describes the root classes, the AMS generator, and the architecture of an instantiated AMS, which we refer to collectively as Grapevine. It also shows how these pieces work together to form an AMS that meets the designer's requirements.

3.1 Root Classes

Six root classes (attribute, agenda item, agenda, attribute collection, attribute iterator, and agenda iterator) form the basic extensible structure upon which all customized classes are built. This structure is conceptually simple, yet provides designers with enough expressive power to specify the structure of data for many different AMSs. These classes are discussed in this subsection, and the following subsections present the means for extending AMS classes from these root classes.

Figure 3 shows the root classes' fields and methods and the way in which subclasses are generated from them through the inheritance mechanism (described in the next subsection). Each root class and how it is extended is described in detail below.

An **attribute** is a class that is the fundamental building block of an AMS, forming a "field" of agenda item and agenda classes. Each attribute class consists of a name, a data type, and a value. Methods are provided to get and set each attribute's value and to get each attribute's name and type

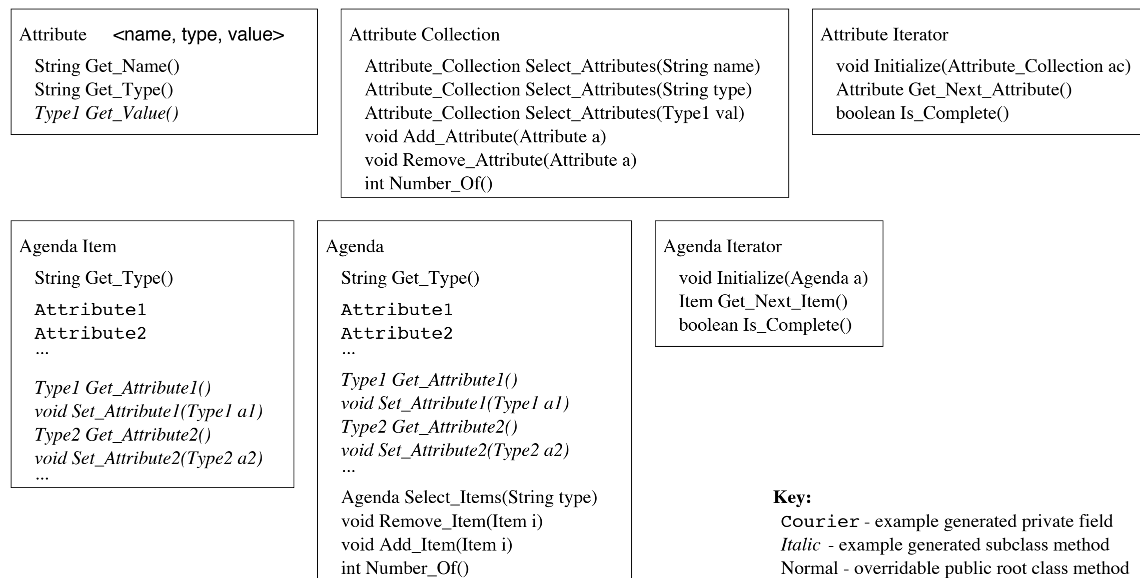

Figure 3: Root class and subclass template definition (not all methods are shown).

fields.

The type field of an attribute subclass may be any standard data type (e.g., integer, float, character, and arrays of these) plus any user-defined type, and is represented by "*Type1*" in the figure. Because the attribute root class has no type, it cannot be instantiated. Attributes are defined as root objects to provide a uniform, reflective way to store AMS data. They are uniform because they allow any type of data to be stored, and they are reflective because query methods are provided to answer queries about the type and name of that data.

An example of an attribute is the `status` of the bug report item. This attribute would have the name "status", be of enumerated type, and have some descriptive information, e.g., "Fixed" or "Aborted", as its possible value.

An **attribute collection** is a collection of attributes whose membership may change dynamically. Methods are provided to add attributes to, and remove attributes from, the collection as well as to associatively construct subcollections based on member attributes' name, type, or value, or some such combination. Attribute collections are used for grouping together related AMS entities and for dynamic adaptation of AMS data.

An **attribute iterator** is a class that is used to visit in turn each attribute in an attribute collection. Methods are provided to initialize the iterator for a particular attribute collection, to check whether any members remain to be visited, and to get the next unvisited member from the collection. Defining an attribute iterator to be a class enables multiple simultaneous iteration over any single given attribute collection. If the designer of an AMS specifies that an iterator may only iterate over attribute collections of a particular subclass, the initialization and get next attribute methods are changed accordingly.

An **agenda item** (also referred to as just "item") is a class consisting of zero or more attributes, none of which has the same name. The root agenda item class has no attributes and consequently no attribute-specific methods. If specific attributes, indicated by italics in figure 3, are added to subclasses of the root agenda item class by the designer, methods to get and set the value of each attribute are generated. An item may be queried to determine its type.

For example, the bug tracking item subclass from the scenario might have two attribute members, the `status` attribute mentioned previously, and a `name` attribute that has a name of "name," a string type, and a value that describes the agenda item, in addition to others.

Attribute collections and agenda items differ in important ways. First, an agenda item's attributes are static and determine its type, while the number and identities of an attribute collection's members may vary dynamically and don't directly determine its type. Second, an item may not contain two attributes having the same name, while an attribute collection has no such restriction.

The value of any attribute of an agenda item can be an attribute collection. This definition affords enough dynamism for items to provide required functionality, such as annotation and auditing, while still being strongly typed. For example, consider an auditing log for the bug tracking item. The log attribute's value might be an attribute collection consisting of any number of log entries. Log entries may thus be dynamically added to an item without changing its type. This is exactly the behavior we require.

An **agenda** is a class with zero or more attributes, none of which have the same name, and a collection of agenda items whose members may change dynamically. Methods that get and set the value of each attribute and methods for adding and removing items from the agenda are provided. These methods allow associative access to the items of an agenda. The agenda root class has no attributes; its only methods are the three to allow manipulation of the agenda's associated items, one to query its type, and standard create and destroy methods.

An **agenda iterator** is used to visit each item on an agenda in turn. Methods are provided to initialize the iterator for iteration over a particular agenda's item collection, to check whether any members remain to be visited, and to get the next unvisited item from the agenda. Having agenda iterators be a separate class allows multiple simultaneous iteration over an agenda's item collection. The designer of an AMS may specify an iterator subclass to only iterate over agendas of a particular class.

These six root classes, in conjunction with the extension mechanism described in the next section, allow an AMS designer to create customized classes of agenda items. A way in which subclasses of these root classes might be used to organize data in an AMS is shown in figure 4. Iterator objects are typically instantiated when needed, used, then discarded, so they do not appear in the figure.

It is important to observe that these root class definitions encourage the notion that *every collection of items is an agenda*. Conspicuously absent are the classes "Item Collection" and "Item Iterator," having been replaced with "Agenda" and "Agenda Iterator." We have taken a minimalist approach in the specification of the root agenda class to avoid specification and performance penalties in the implementation of what we believe is the most common type of agenda, namely a single collection of items (usually of the same type) with some associated attributes that describe the collection.

Allowing agendas to be sharable permits multiple ways of implementing the concept of a group of users in an organization. In our scenario, Alice needed to notify all the organization's testers of the existence of the reported bug. If each tester views an agenda shared by all the testers as well as his or her own personal agenda, Alice could simply add the bug fix item to the shared agenda. Each tester's viewer would then display the new item. Because items are also sharable, another way Alice might accomplish this is to add the item to each tester's personal agenda. In this way, each tester's viewer would also display the new item.

Having more than one way of enabling a group of agents to view an item seems necessary. One key difference between these two methods of posting the item is the degree to which the "group" is formalized. In the scenario, a shared agenda formalizes the group "testers", which is entirely appropriate given the process by which bugs are fixed in the organization. On the other hand, if Catherine needs to schedule a meeting with Douglas and their manager, she should not have to construct a group agenda for the three of them, and ensure they are all viewing it. Intuition and experience with another form of collaboration, e-mail, indicates that using more than one way of designating groups is natural. Most e-mail systems allow mail to be addressed to multiple recipients as well as to system-wide mailing lists.

As shown in figure 1, the output of the system generation phase is generated code for an AMS that meets the requirements given to the designer. To allow agents to use the AMS, agent specific interface code must be linked with the AMS. Human user interface code is included with the root type and specialized with each subtype to aid in the construction of user interfaces. Automatic user interface specialization is not discussed further because user interface issues are not the focus of this paper.

3.2 Extension Mechanisms

Inheritance and policy are the two mechanisms that allow extending root classes to create the specialized classes required by a specific agenda system. By inheritance we mean the extension of classes with additional fields and methods, as in a typical object-oriented language with single inheritance. By policy we mean the specification of higher-level constraints on the use of AMS data. Inheritance can be provided by the object-oriented inheritance found in a reasonable implementation language (e.g., Java), but code to implement policies must be synthesized during AMS generation.

3.2.1 Inheritance Mechanism Extension of the root classes by inheritance is specified through use of the keywords `extends`, `type`, `attribute`, `method`, `itemclass`, and `private`. These keywords are used to specialize the root classes for use in a particular agenda management environment. The keyword `extends` names a subclass, `type` specifies the type of the contents of a class, `attribute` specifies that an attribute should be part of an object, `method` adds a designer specified method to the class, `itemclass` specifies the type of items in an agenda's collection, and `private` is used to control which methods are visible outside of the specification, i.e., to the agents.

To give a better idea of how a specification is written, an example that might be used to create an AMS for the introductory scenario is provided below. Some attributes that are predefined have been used.

```
LogEntry extends Attribute {
    type String;
}
LogCollection extends Attribute_Collection {
    type LogEntry; // collection of LogEntry attributes
}
Log extends Attribute {
    type LogCollection;
}
```

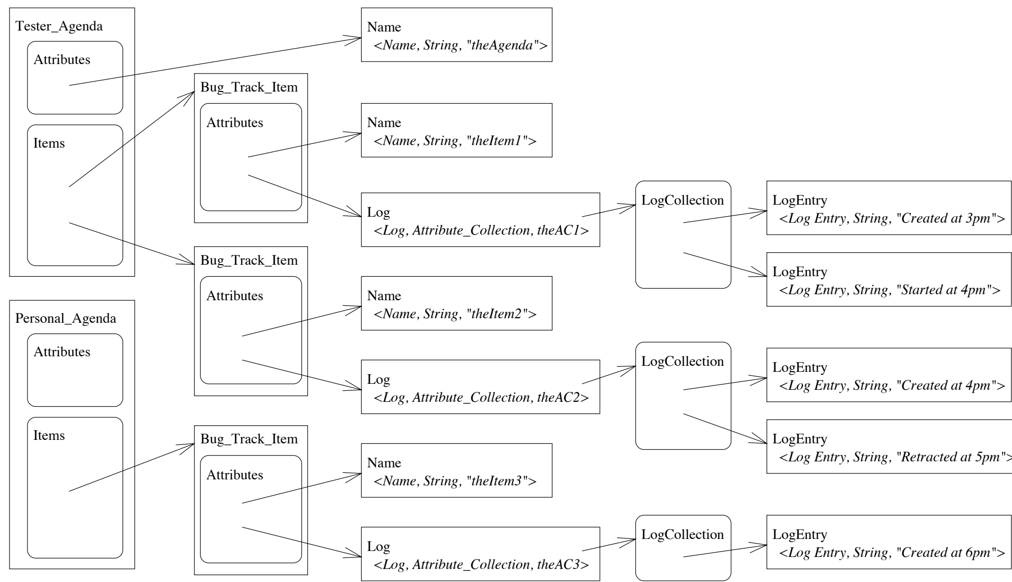

Figure 4: Example instantiated subclasses diagram.

```
Bug_Track_Item extends Agenda_Item {
    attribute Name;
    attribute Owner;
    attribute Priority;
    attribute Status;
    private attribute Log; // visible only in specification
    private attribute LogEnabled;
    method void AddNote(String Note)
        { method implementation }
    method void RemoveNote(String Note)
        { method implementation }
    method String printLog()
        { method implementation }
}
Bug_Agenda extends Agenda {
    itemclass Bug_Track_Item;
    attribute Name;
    method void MoveItem( Agenda toAgenda
            Bug_Track_Item anItem)
        { method implementation }
}
```

This specification provides simple examples of extension of the attribute, attribute collection, agenda item, and agenda root classes. Instantiation of these classes in an AMS could produce the example graph of objects shown in figure 4.

Grapevine generates code that defines the subclasses specified by the designer. The specification of attribute, attribute collection, attribute iterator, and agenda iterator subclasses serves mainly to define their type. The root class methods of these classes are overridden or replaced by methods with the correct signature for the subclass. Agenda and agenda item subclass implementations are generated by directly copying specified methods, creating private fields with the name and type of the specified attributes and generating methods to get and set the values of these fields. The names of the generated methods are synthesized by concatenating the strings "Get" and "Set" with the names of the attributes the generated methods access. The designer may also use the specification to override methods that a subclass inherits from its superclass.

The use of a generator facilitates the specification of classes and subclasses in a type strong manner. By providing the `type` and `itemclass` keyword and by generating appropriately typed subclasses, the designer may create many classes of agenda items and agendas for an AMS while ensuring that these types do not inappropriately intermix. Allowing attribute collections to be included as attributes of agendas and agenda items provides much of the adaptability that is lost in the generative approach.

3.2.2 Policy Mechanism Our approach also allows the designer to specify *policies*, or rules for the use of subclass methods. This provides the designer with a rudimentary mechanism for enforcing consistency among the objects that comprise an AMS and to discipline their use.

For example, suppose a designer wants to design a system that adds a log entry to an item every time the name of the item is changed. The designer could use the log attribute we introduced in the previous subsection to store the log, but every generated `Set_Name` method in every object would have to be rewritten to ensure that a log entry was made. The policy mechanism is the component of the extension mechanism that automates this task.

A policy is specified with a named collection of constraint, action and method name triples (similar to the ECA model presented in [5]) that is to be applied to class methods. As illustrated below, the keyword `policy` names a policy and the keywords `enforce` and `for` bind to specific meth-

ods the constraints that make up the policy. The keywords `constraint`, `action`, `check as`, `precondition`, `postcondition`, and `prepostcondition` specify the contents of a policy. The bodies of the constraints and actions making up a policy are written in the implementation language. Constraints return a boolean value, indicating whether or not the action should be taken. Actions do not return a value; they either successfully complete or interrupt execution flow by raising an exception. This mechanism allows policies to be applied to classes orthogonally to the class inheritance hierarchy. This need might also be met with multiple inheritance mechanisms, as in C++, or with subject oriented programming approaches described in the literature [8].

The following example will help make the way in which policies are specified clearer. It specifies a logging policy, such as what might be designed for the bug tracking item, with the bodies of the constraints and actions written in Java.

```
policy logging(String opname) {
   constraint IsLoggingDisabled( String s )
      { return not GetLogEnabled(); }
   action AddLogEntry( String s ) {
      LogAttrCollect ac = GetLog();
      LogEntry entry =
         new LogEntry(opname+s+"invoked");
      ac.AddAttribute(entry);
   }
   constraint IsLoggingDisabled( UID u )
      { return not GetLogEnabled(); }
   action AddLogEntry( UID u ) {
      LogAttrCollect ac = GetLog();
      LogEntry entry =
         new LogEntry(opname+u+"invoked");
      ac.AddAttribute(entry);
   }
   check as precondition;
}
```

An example of binding the above policy to an item's methods is provided below.

```
enforce logging("SetPriority") for
   Bug_Track_Item.SetPriority(String pri);
enforce logging("SetOwner") for
   Bug_Track_Item.SetOwner( UID owner ),
   Tester_Agenda.SetOwner( UID owner);
```

This example also demonstrates how parameters are used. The policy parameter (`String opname` in the example) is provided to pass to the constraints and actions information about the method on which a policy has been enforced. The names of the policy parameters may be used within the constraint and action implementations as regular identifiers. The method parameters (`String priority` and `UID owner` in the example) are used to pass to the constraints and actions information about the arguments to methods on which a policy has been enforced. Multiple constraint and action definitions may be provided in a policy

Name	Type	Name	Type
Name	String	*Description*	String
Priority	integer	*Status*	enumerated
Deadline	Date	*Subitem*	Agenda
Owner	UID	*Prerequisites*	Agenda
Alternatives	Agenda	*Reference*	URL
Log	Attribute Collection	*Notes*	Attribute Collection
LogEnabled	boolean	*LogEntry*	String

Table 1: Predefined attributes

definition. Only those constraints and actions whose parameters match the method parameters will actually be bound to the methods. If there are no matching constraints, an error will be signalled during AMS generation.

When a subclass is defined which overrides methods that have a policy enforced, the overridden methods no longer have the policy applied. The designer must explicitly re-enforce the policy for the overridden method. This allows a designer to define a subclass that has a different kind of policy (or no policy at all) enforced for the same method.

3.3 Predefined Classes and Policies

Several predefined objects and policies are provided for the AMS designer to use. By providing an ever-growing library of common high-level components, we hope to further ease the development of custom AMSs.

Predefined attributes are shown in table 1. We assume that all of these types (including UID) are available in the implementation language; if any are not, they must be defined and provided at run-time. Many of these attributes are complemented by one or more predefined policies.

Several of these attributes are self-explanatory or have been presented previously. Two new ones are the alternatives attribute, which designates items that can be worked on instead of this one, and the prerequisites attribute, which designates an agenda that holds items which must all be "completed" before this one is "started" (as reflected by the status attribute).

Several predefined policies are also provided and described in table 2. Many policies are meant to be used in conjunction with specific attributes.

3.4 Substrate

In the preceding subsections, we have described how a designer can meet some specific agenda management requirements by extending a set of root classes. The architecture of the substrate on which the root classes (and consequently the rest of the system) are built must allow a customized system to meet other requirements commonly found in an agenda management domain. The substrate must address requirements such as concurrency control, data persistence, distri-

Name	Description
logging	creates a new `LogEntry` attribute that describes the method that has been invoked and adds this to the log attribute collection.
access control	uses an external access control table and the owner attribute to look up access privileges, providing control over who may invoke a method.
alternatives	This policy, used with the alternatives and status attributes, automatically removes all alternatives to an agenda item once that item has been started.
prerequisites	This policy, used with the prerequisites and status attributes, prevents an item from having its status modified until its prerequisite items have been completed.

Table 2: Predefined policies

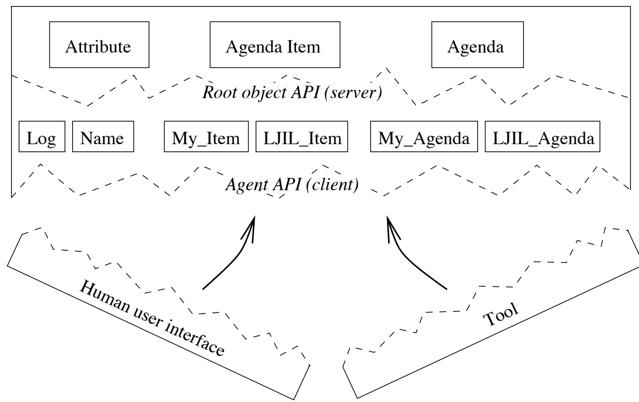

Figure 5: Interfaces of instantiated AMS.

bution, coherence, and scalability of the AMS. In this subsection, we describe how the substrate underlying any generated AMS can work with designer-specified subclasses to meet its requirements.

When an AMS is specified, the system generation mechanism must create an AMS to meet these requirements. A client-server substrate sufficies to meet the previously mentioned requirements. The server part of the substrate implements only the six root classes, and provides their methods to the clients via remote procedure calls. Thus, the root objects' interface (API) and implementation on the server are identical for any generated instance of AMS.

All designer-specified subclasses of the six root classes are implemented entirely on the client side. The designed subclass methods are implemented in terms of the small set of generic root class methods that are exported by the server. The architecture of an example instantiated system is shown in figure 5. The designer-specified subclasses appear in the client at the bottom of the figure. They subclass client-side representations of the root objects, whose methods merely make remote procedure calls to the root classes on the server that actually perform the requested function. Agent specific interfaces connect to the AMS API at the bottom of the figure. A human's user interface looks to the AMS like just another tool using the system; the special case code required by people is confined to the user interface software used by people.

Advantages of this single-server architecture include easily understandable distributed operation, straightforward synchronization of client execution, server-side persistence, and the ability to update clients with newly generated code while the server is left running. Disadvantages include security problems (because policies are implemented in client code, they can be easily circumvented), a single point of failure, lack of scalability, and other standard problems associated with centralized software architectures.

While providing only a single fixed substrate architecture means that there is no explicit control over the nature of object serverization, it also isolates designers from the details of distributed operation, simplifying system design and the instantiation process. It might be better to allow the designer to specify where the client-server split should occur, but allowing this adds complexity to the designer's task with dubious benefit.

4 Preliminary Evaluation

4.1 Prototype of Grapevine

We have partially implemented a prototype version of Grapevine to evaluate the fundamentals of this approach. We used this prototype to help generate some simple AMSs and evaluate their utility.

The prototype implementation of Grapevine's substrate is split into a client and server as described previously. On the server side, all root classes are written in Pleiades/Ada [19]. Clients communicate with the server via an interlingual RPC mechanism, called Q [17]. The substrate does not currently provide for callbacks when locally cached AMS data are changed. Thus, clients must poll for changes.

Clients are currently implemented in Java. The Java inheritance mechanism is used to support Grapevine inheritence directly. The policy mechanism is not yet implemented but we have experimented with manually doing the associated generation. All of the predefined attributes and policies are implemented. The prototype currently has a generic user interface based directly on root class implementations on the server. This interface is easily modified to use the client interface of each generated AMS.

The prototype does not support concurrency control due to performance considerations. Instead we rely on clients to avoid interfering with one another. It is not clear what sort of

concurrency control is appropriate in cooperative environments; this is an area of current research [2, 15]. Because Java classes are loaded on-demand, the prototype is able to support significant dynamism and user adaptability, though we have not yet experimented with these capabilities.

4.2 Evaluation of a Generated AMS

The Grapevine prototype was evaluated by using it to generate an AMS to support the execution of process programs written in a subset of JIL [11]. JIL programs are executed by human and software agents, and we coordinated these agents with a Grapevine-generated AMS.

The utility of the created AMS was evaluated in the context of execution of a rudimentary process program for a phase of the Booch Object Oriented design process. In this process, the AMS helps coordinate the activities of a human designer and a software client in creating a class diagram. As this process executes, the process interpreter creates agenda items and assigns them to an execution agent by posting them to the agent's agenda.

One type of agenda item and agenda were used in the prototype AMS. This item contained the predefined name, status, log, subitem, and alternative attributes, and enforced the policies for controlling alternative item choice and for adding log entries upon item creation. The process interpreter used the provided agent interface to the AMS, although the lack of a callback mechanism required implementation of a separate agenda monitoring thread. Since the Grapevine prototype is not complete, as described above, manual assistance was needed to generate this AMS. The substrate and root classes were used as provided by Grapevine. Attributes were attached to items and agendas and accessor methods were provided in a mechanical way. Policies were applied to item methods uniformly.

Overall the use of Grapevine to generate the AMS and the resulting AMS proved successful. It was interesting that preliminary discussions of the role of an AMS in process execution showed that there were as many notions of what an "agenda" is as there were people involved. Having a concrete language with well defined semantics for specifying AMS components helped the group reach consensus and proceed with design. The design went through many iterations, however, in which having the Grapevine prototype, albeit incomplete, proved extremely useful. All of this reinforced our belief in the value of a language based AMS generation approach.

5 Related Work

Research related to agenda management has been primarily in the areas of process centered environments, tool integration environments, groupware and workflow systems, and in asynchronous computer supported cooperative work. Considerable overlap between these areas exists [7]. In constrast to most related work, we have focused on a solution to the problem that is applicable to a variety of domains and is useful for both human and software tool agents. The importance of allowing agent adaptation has also been central to the development of systems such as Oval [16] and Agenda [12].

By virtue of the need to communicate with humans in a programmable way, several process execution environments have formalized the notion of agenda.

Process Weaver [6] has an integrated agenda that is used to coordinate the activities of humans by posting and delegating "work contexts." There is apparently no decoupling of "agenda" from "view", so supporting groups of users may be problematic. It is also not clear that work contexts are decomposable into subtasks. While Process Weaver is built with hooks for tools that provide process automation, it "aims at giving process support to software developer teams," and is focused on supporting human users, who may then invoke tools. Process Weaver includes a process modeling language that has notions of concurrency with semaphores, message passing, distribution, modularity. In contrast, we have decoupled process from the mechanism. An AMS is relatively passive; it serves to facilitate and control client interaction. An agent may be human or tool; this decision too is decoupled from the design of the AMS itself. Our aim has been to build a framework that could be used to construct, for instance, a Process Weaver-style AMS.

The importance of decoupling process state representation from process modelling language was noted in [9]. ProcessWall is a process state server, intended to be used to facilitate process execution, as with an AMS. Process state is represented as a DAG of task nodes, and operations are provided to create tasks, add precedence edges, and add subtasks to the DAG. The notion of task "satisfaction" is also introduced. Task nodes can be seen to roughly correspond to agenda items, and supporting these sorts of operational semantics in generated AMSs (through predefined attributes) was a primary design goal of Grapevine. In the ProcessWall there are "task parameters" which allow data to be passed between task nodes. Grapevine has no explicit data flow concept, so it is not clear whether a generated AMS could provide the functionality of the ProcessWall. In ProcessWall, clients are divided into tools, process-constructors, process-constrainers, and user-intermediaries. We have no notion of the middle two; these are considered ordinary tools. Process-constrainers seem to provide a way of "enforcing any constraints on the legal structure of the process and product state." In contrast, we might build this function into the system with our policy mechanism. It is possible that both kinds of constraint enforcement are needed; further research is needed.

The Marvel software development environment [3] has addressed the issue of assigning tasks to users, though this is tightly integrated with the process environment. The issue of

supporting cooperation in Marvel was explored in [2], however this exploration concentrated on an appropriate concurrency model for cooperative work, not the higher level issues addressed here.

In [1] the authors explore using the SPADE-1 environment to support asynchronous cooperative work. Like Grapevine, SPADE-1 is "based on the principle of separation of concerns between process model enactment and user interaction environment." The environment consists of a number of tools, among which is an Agenda tool. The Agenda uses a configuration file to define its behavior (via an event-based transition model) and the structure of tasks (the name and type of attributes of the agenda items) that will appear on the agenda, providing some tailorability. The Agenda tool is invoked by each human user and allows humans to send and receive information, however, a separate tool, SPADEShell, is used to send requests to the process execution environment.

Lotus Agenda [12] is a personal information manager characterized as a new type of database, an "item/category database." Users add items to the database and assign them to categories either by hand or automatically through selection rules. These categories may be viewed in a variety of ways. Agenda has given enormous flexibility to the end user, and in the process has lost the benefits of strong typing. In addition, Agenda is not designed to be used by tools and does not have facilities for controlling flow (as with our policies).

Lotus Notes [4] can be loosely characterized as a general replicated database system that allows programmers to create a variety of collaborative applications. While Notes might be used to provide a substrate on which an approach to agenda management may be based, no guidance to this end is provided: the concepts of agenda and agenda item are absent.

Similarly, while approaches to tool integration such as Desert [18] and distributed object computing standards such as CORBA address several of the requirements of agenda management provided by the AMS substrate, these systems provide little guidance for implementing an agenda management system and are not explicitly designed to be used by humans.

The Workflow Management Coalition's reference model[10] has many of the components of our framework and substrate, and has standardized APIs for humans and tools. But, as a reference model, it is not intended to be used to generate specific agenda management systems.

Oval [16, 13] is a tool for cooperative work that offers adaptation abilities that are well beyond those outlined in this paper. Oval is a "radically tailorable system" that allows users to create applications from semistructured objects, user customizable views, rule-based agents, and links. The objects are formed from templates of named (but not typed) fields and grouped in a type hierarchy, similar to how AMS designers create subclasses of agenda items by adding attributes. Folders exist for grouping together collections of objects; these are similar to agendas. Furthermore, a user can create rule-based agents that perform automated actions on the user's behalf; policies can be thought of as a global, procedural approximation of user agents. However, Oval is not intended to be used as a tool integration mechanism, and in ObjectLens (Oval's precursor) there is no global object space; objects are copied and sent as messages to each other.

6 Concluding Remarks, Future Work

We have proposed a framework for generating agenda management systems that

- can evolve to meet organizational changes in requirements,
- can adapt to meet differences in agent requirements, and
- are useful integration mechanisms for both humans and tools.

We described the attributes, agenda items, and agendas that comprise an AMS and described the implementation and use of AMSs.

Preliminary use of an AMS generated (with manual assistance) from a prototype version of Grapevine has been encouraging. We believe that further experimentation will show that the requirements listed in section 2 can all be met with Grapevine-generated AMSs, but more research and experience is required to validate this intuition.

While we believe that Grapevine, as described here, will prove to be very flexible and useful, we now outline some obvious ways in which its capabilities might be beneficially extended in future work.

We believe that the role and implementation of policies in AMSs warrants further investigation. Currently the policy capability is rather rudimentary, being intended to enforce well-formedness constraints on AMS structure and information. Constraints are evaluated once, and it is assumed that responses to constraint failure will either "repair" the violation or interrupt execution (e.g., by raising an exception). More complex forms of response (e.g., reevaluation of the constraint) may indeed prove useful. In addition, in more complex AMSs, we expect multiple policies to be applied to a method. Currently there are only primitive defaults for dealing with interactions among such constraints. A richer model may be needed.

Supporting adaptation and evolution in ways that are consistent with the generation approach raises a number of additional important issues. For example, we would like to allow users to make local adaptations to an AMS type system. This is not currently allowed because of concerns about how an AMS can share object instances without sharing its type hierarchies. This issue has also been problematic in systems such as Lotus Agenda [12] and Oval [16]. We expect to

explore ways in which to do this, as support for local modifications to type hierarchies can also enable AMS to AMS communication, another desirable capability.

We would also like to allow for the modification and extension of existing AMS classes. But this creates the problem of migrating existing instances of those classes, which is a research area itself [14]. The generation framework we have developed should be compatible with solutions to such problems, and we expect class evolution mechanisms to be incorporated into future generations of Grapevine.

Finally, much future work needs to address the user interface issue. It seems clear that generic user interfaces can be generated by Grapevine along with the AMS itself. But it also seems clear that much customization is desirable. Future research should address ways in which generation techniques could be used to minimize customization effort.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the work of Peri Tarr and Stan Sutton, who did important preliminary design work on this project and who provided invaluable assistance in demonstrating how Pleiades could support server side implementation. We also wish to thank Alexander Wise for lively discussions and essential help during integration of our generated AMS with the Little JIL interpretation system.

REFERENCES

[1] Sergio Bandinelli, Elisabetta Di Nitto, and Alfonso Fuggetta. Supporting Cooperation in the SPADE-1 Environment. *IEEE Transactions on Software Engineering*, 22(12):841–865, December 1996.

[2] Naser S. Barghouti. Supporting cooperation in the Marvel process-centered SDE. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software development environments*, pages 21–31, 1992.

[3] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software development environments*, pages 149–158, 1992.

[4] Lotus Development Corporation. *A Quick Tour Of Lotus Notes*. Lotus Development Corporation, 1993.

[5] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 204–214, 1990.

[6] Christer Fernström. PROCESS WEAVER: Adding Process Support to UNIX. In *Proceedings of the Second International Conference on the Software Process*, pages 12–26. IEEE Computer Society Press, 1993.

[7] D. Georgakopoulos, M. Hornick, and A. Sheth. *Distributed and Parallel Databases*, chapter An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure, pages 119–153. Kluwer Academic Publishers, 1995.

[8] William Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 411–428, 1993.

[9] Dennis Heimbigner. The ProcessWall: A Process State Server Approach to Process Programming. In *Fifth SIGSOFT Symposium on Software Development Environments*, December 1992.

[10] David Hollingsworth. The workflow reference model. Technical Report TC00-1003, Workflow Management Coalition, Nov 1994. Draft 1.0.

[11] Stanley M. Sutton Jr. and Leon J. Osterweil. The Design of a Next-Generation Process Language. To appear in the Proceedings of the Fifth annual conference on the Foundations of Software Engineering.

[12] S. Jerrold Kaplan, Mitchell D. Kapor, Edward J. Belove, Richard A. Landsman, and Todd R. Drake. Agenda: a personal information manager. *Communications of the ACM*, 33(7):105–116, Jul 1990.

[13] Kum-Yew Lai, Thomas W. Malone, and Keh-Chiang Yu. Object Lens: A "Spreadsheet" for Cooperative Work. *ACM Transactions on Office Information Systems*, 6(4):332–353, Oct 1988.

[14] Barbara Staudt Lerner. TESS: Automated Support for the Evolution of Persistent Types. In *Proceedings of the 12th Automated Software Engineering Conference*, 1997.

[15] Barbara Staudt Lerner, Arvind H. Nithrakashyap, and Lori Clarke. Cooperative concurrency control for software engineering. In *OOPSLA '97 Workshop on Collaboration in the Object Development Lifecycle*, 1997.

[16] Thomas W. Malone, Kum-Yew Lai, and Christopher Fry. Experiments with Oval: A Radically Tailorable Tool for Cooperative Work. In *CSCW '92 Proceedings*, pages 289–297, November 1992.

[17] M. Maybee and D. Heimbigner. Q: A Multi-lingual Interprocess Communications System. Technical Report CU-ARCADIA-101-93, University of Colorado, August 1993.

[18] Steven P. Reiss. Simplifying Data Integration: The Design of the Desert Software Development Environment. In *Proceedings of ICSE-18*, 1996.

[19] Peri L. Tarr and Lori A. Clarke. PLEIADES: An object management system for software engineering environments. In *ACM SIGSOFT '93 Symp. on Foundations of Software Engineering*, pages 56–70, Dec 1993.