MICRO-WORKFLOW: A WORKFLOW ARCHITECTURE SUPPORTING COMPOSITIONAL OBJECT-ORIENTED SOFTWARE DEVELOPMENT

BY

DRAGOS-ANTON MANOLESCU

Diploma de Bacalaureat, Liceul Matematică–Fizică Nr. 1 Bucureşti, 1990
Diploma de Inginer, Universitatea Politehnica Bucureşti, 1995
M.S., University of Illinois, 1997

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

MICRO-WORKFLOW: A WORKFLOW ARCHITECTURE SUPPORTING COMPOSITIONAL OBJECT-ORIENTED SOFTWARE DEVELOPMENT

Dragos-Anton Manolescu, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 2001
Ralph E. Johnson, Advisor

Workflow technology and process support lies at the center of modern information systems architectures. But despite the large number of commercial workflow systems, object-oriented developers implement their business, scientific, or manufacturing processes with home-made workflow solutions. Current workflow architectures are based on requirements and assumptions that don't hold in the context of object-oriented software development. This dissertation proposes micro-workflow, a new workflow architecture that bridges the gap between the type of functionality provided by current workflow systems and the type of workflow functionality required in object-oriented applications. Micro-workflow provides a better solution when the focus is on customizing the workflow features and integrating with other systems. In this thesis I discuss how micro-workflow leverages object technology to provide workflow functionality. As an example, I present the design of an object-oriented framework which provides a reusable micro-workflow architecture and enables developers to customize it through framework-specific reuse techniques. I show how through composition, developers extend micro-workflow to support history, persistence, monitoring, manual intervention, work-lists, and federated workflow. I evaluate this approach with three case studies that implement processes with different requirements.

To Beth Marie and Traian Paul

Acknowledgments

Many people have contributed, directly or indirectly, to the successful completion of this dissertation. Although it would take too many pages to name them all, I would like to thank the following:

Professor Ralph Johnson has been my teacher, advisor, mentor, and friend. His vision, guidance, and experience helped me learn a lot, as well as find and focus on an exciting research topic. I would also like to acknowledge the other members of my committee. Klara Nahrstedt has guided my Master's thesis and was open to accommodate my interests in objects, patterns, and business software. Although Jane Liu was not on my final committee, she has always provided lucid advice and helped me keep my confidence level high. Roy Campbell's recommendation to study the relationship between workflow technology and computer simulation helped me broaden my perspective. Upon entering the graduate program Gul Agha encouraged me to get an early start with the research. The discussions I had with him after he joined the doctoral committee helped me think of my research in the broader context of software architecture.

The work of Francis Anderson, Michel Tilman, and Jeff Oakes contributed to my starting this research. The numerous emails I've exchanged with Francis and the telecom examples he provided helped me understand how to approach workflow from an object perspective. Michel suggested the term "micro workflow" and provided insightful comments over email, as well as during an OOPSLA'99 reception. Jeff's comments provided a reality check for my ideas.

The members of Ralph Johnson's Software Architecture Group have provided a first-class environment for discussing and exploring research ideas, as well as keeping up to date with the most recent research efforts. Brian Foote's advice helped me see the path and then walk it. Don Roberts helped me clarify different Smalltalk questions and set a positive example for me. John Brant also helped with Smalltalk and GemStone, and provided constructive criticism at various points in time. Joseph Yoder gave me the chance to work with expert Smalltalkers, and helped me support myself in the PhD program.

Bosko Zivaljevic arranged for me to talk about workflow at CISCO Systems in Urbana, IL. Dirk Riehle

made possible a presentation at Skyva International in Boston, MA. Piotr Palacz and Bryce Pepper facilitated presentations at Complete Business Solutions and Transportation.com in Overland Park, KS. Finally, Clarence Ellis gave me the chance to present my work to him and his students at the University of Colorado in Boulder, CO. These talks made my defense go smoothly.

Joseph Bacanskas provided assistance with GemStone/S, and Peter Hatch helped me get started with VisualWorks 5i and Opentalk. Without them my dealing with persistence and distribution would have taken much longer. I am also indebted to the Linux community. I performed all this work (research and writing) on Intel- and Alpha-based computers running the Linux operating system. ObjectShare, Cincom, and GemStone Systems had the foresight to offer non-commercial versions of their products for Linux.

Alex Delis, Athman Bouguettaya, Clarence Ellis, Gary Nutt, Christoph Bussler, and Dimitrios Georgakopoulos helped me get acquainted with the domain. They sent me workflow bibliographies, manuscripts, and commented on my ideas and writings.

Julie Legg, Anda Ohlsson, and Bonnie Howard helped me schedule exams, reserve rooms, contact committee members, and deposit the thesis.

Don and Carolyn Mullally have been instrumental from the embryonic stages of this dissertation. None of this would have been possible without the help of Don, who planted the seed of my entering the Illinois doctoral program around 1993 and then made it happen. Carolyn and Don made my living in the flat prairie of Illinois a pleasant experience. I am grateful to both of them.

My parents and grandparents have been behind me from the beginning. All my achievements are the fruit of their patience and diligent work. My parents have also provided feedback on several draft chapters. Their input contributed to my finishing this dissertation in a timely manner.

Finally, I am grateful to my wife and son. Beth provided language and style clarifications and advice, was a patient listener, and her finishing a few months before me set an example that I've tried to follow. She has also supported financially the last stages of this work. Traian helped me remain focused and avoid procrastination. Thanks to him I was able to complete the research, write the thesis, and defend within 323 days after passing the preliminary exam.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ADF Activity Decision Flow

ACL Access Control Level

API Application Programming Interface

BOA Basic Object Adapter

BPMT Business Process Modeling Tool

BPR Business Process Reengineering

CGI Common Gateway Interface

CLOS Common Lisp Object System

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit

CSCW Computer-Supported Collaborative Work

DBMS Database Management System

DCOM Distributed Component Object Model

DLL Dynamic Link Library

ECA Event Condition Action

ERP Enterprise Resource Planning

FDL Flow Description Language

GUI Graphical User Interface

HTML Hypertext Markup Language

IDL Interface Description Language

IDPH Illinois Department of Public Health

JDBC Java Database Connectivity

LSF Load Share Facility

MOM Message Oriented Middleware

NCSA National Center for Supercomputing Applications

ODBC Open Database Connectivity

OID Object Identifier

OMG Object Management Group

OODBMS Object-Oriented Database Management System

PVM Parallel Virtual Machine

RDMBS Relational Database Management System

RMI Remote Method Invocation

RPC Remote Procedure Call

UML Unified Modeling Language

WfMC Workflow Management Coalition

WIL Worklfow Interchange Language

WPDL Work Process Description Language

WfMS Workflow Management System

XML Extended Markup Language

Chapter 1

Introduction

Simplicity does not precede complexity, but follows it.
Alan J. Perlis

Recently I studied three large object-oriented frameworks [90, 6, 24]. Each framework deals with processes from a different application domain—insurance, telecommunications billing, and school administration. In each framework the architect used workflow to allow developers to change the business processes[1] without changing the domain-specific code.

Although Sheth and colleagues [118] estimate that the number of readily-available workflow systems is between 200 and 300, none of these frameworks employed an off-the-shelf workflow product. Initially their architects considered using an existing workflow management solution. However, after studying several options, they discovered that the design of current workflow systems makes them hard to integrate in object-oriented applications. Consequently, each architect built a custom workflow solution that provided exactly the functionality required by his framework, and it integrated smoothly within the system.

The above observation points to an interesting research problem. On the one hand, software developers want to use workflow technology to implement processes within object-oriented applications. On the other hand, the workflow market offers hundreds of workflow management systems. But somehow the large supply of workflow systems doesn't quite match the demands of developers. What causes this mismatch? Perhaps developers require features that current workflow management systems don't provide? Or perhaps workflow systems make assumptions that don't hold in the context of software development? Answering

[1] I use the term "business process" to distinguish between the processes implemented with workflow and operating system processes. But workflow is applicable to many other domains besides the business domain. For simplicity, this thesis uses the term "process" instead of "business process," "administrative process," "scientific process," etc.

these questions will explain why developers are forced to craft custom workflow solutions, and will show how to avoid this problem.

This thesis starts from the observation that several object-oriented systems use hand-crafted workflow solutions instead of an existing workflow system. I've checked with other developers and they have confirmed that building workflow functionality from scratch rather than reusing it is the rule rather than the exception. This situation reveals a gap between the type of workflow products available on the market and the type of workflow object-oriented developers need. The research reported in this thesis bridges this gap.

1.1 The Problem

An increasing number of software developers use workflow technology to solve various problems. For example, Leymann and Roller [72] discuss the application of workflow to mobile computing, systems management, multi databases, the Internet, application development, object technology, operating systems, and transaction management. This wide spectrum of domains that employ workflow technology explains the large number of workflow products available on the market. However, current workflow management systems are not suitable for developers who need workflow functionality within object-oriented applications.

The mismatch between the type of functionality provided by current workflow systems and the type of workflow functionality software developers need within object-oriented applications corresponds to several research directions in workflow management:

Objects and workflow technology In his summary of trends in workflow products, standards and research C. Mohan reviews a large number of commercial products and research projects [83]. He observes that typically object-oriented technology doesn't go beyond the implementation of workflow products. Although many workflow management systems are implemented in object-oriented languages, they don't exploit object-oriented technology to its full potential. Their users can't tailor the functionality through object-oriented techniques, and have a hard time integrating current workflow systems into object-oriented applications.

New workflow architectures Current workflow systems are heavyweight, monolithic, and package a comprehensive set of features in an all-or-nothing manner. The narrow purpose design of workflow architectures limits their applicability to the types of applications for which they have been tailored. For

example, Muth and colleagues [86] observe that "most workflow management systems, both products and research prototypes, are rather monolithic and aim at providing full fledged support for the widest possible application spectrum." Additionally, current workflow systems are hard to integrate with other environments. Consequently, research projects like Mentor-lite [84] and OPERA [49] recommend that workflow researchers consider a new generation of lightweight workflow architectures that can be extended and tailored to particular problems and requirements.

Workflow for developers The current generation of workflow systems provide end-user applications aiming mainly at non-technical users. But recent research has helped software developers to obtain better insights into workflow technology, understand its interdisciplinary nature, and realize its potential for building flow-independent applications. Research efforts like Teamware [140], TOWE [99], TriGS$_{flow}$ [111], and OPENFlow [110] focus on development environments for workflow.

Flexible workflow Initial workflow systems didn't provide adequate support for dealing with unexpected situations and changing workflows at run time. This lack of flexibility disappointed workflow users, who rejected the technology. Consequently, a large number of research projects and PhD theses like MOBILE [55], Obligations [12], Endeavors [13], and CRISTAL [71] focus on leveraging interpreters and adaptive object models to provide flexible solutions.

On the software development front, the object-oriented community has developed ways of harnessing the power of objects to build *flexible*, *customizable*, and *dynamic* systems, while fostering *code* and *design reuse*. Because these objectives can reduce the costs of software development and maintenance, they have been the focus of intense research activity. But the heavyweight and monolithic workflow architectures that aim at end-users and don't fully exploit object technology are incompatible with these goals. Therefore, it is not surprising that developers find it hard to integrate existing workflow products within their applications.

The ability to integrate workflow systems within object-oriented applications has several important benefits. First, reusing an existing workflow system instead of building this functionality from scratch lowers the overall cost. Developing a piece of software for single use costs much more than buying it from someone who builds it for mass production. Second, it reduces the time to market. Developers have to deal only with the integration testing. Typically this takes less time than designing, coding, and unit testing a hand-crafted solution. Third, reusing a workflow system allows developers to focus on the application spe-

cific functionality. Their goal is to build an application that solves a particular problem (e.g., billing for telecommunication services, etc.) and not to build a workflow system. Finally, developers may be able to use other features provided by the workflow system that otherwise would be too expensive to implement from scratch. The above arguments, the shift of workflow from end-user applications to a key ingredient of the "networked economy" [118], and the prediction that "internet-mediated workflow will be the single most important technology of the early 21st century" [105] justify the study of a new workflow architecture that avoids the problems of current workflow systems in the context of software development.

1.2 The Solution

My solution to the previous problems takes a radically different approach than current workflow systems. Instead of striving to encapsulate many different features in a monolithic manner, I propose a lightweight workflow architecture that enables software developers to pick and choose the workflow features they need. I call this architecture *micro-workflow*.

Micro-workflow provides a new workflow architecture that solves the problems of traditional workflow architectures through a combination of ideas from object systems, adaptive software models, and compositional software reuse. This yields an alternative that can be integrated into object-oriented applications, can be tailored to specific domains and applications, and can grow to accommodate new features.

Micro-workflow fully embraces object-orientation. One of the key characteristics of micro-workflow is that it applies techniques typical of object systems to solve workflow management problems. Consequently, it reduces the impedance mismatch between the provider of workflow functionality and application objects. Software developers use, customize, and extend micro-workflow like any other object system. However, adopting an object-oriented architectural style requires looking at the big picture through the object lens. Clemens Szyperski observes [122]:

> Introduction of new technologies without the simultaneous introduction of adequate architectural approaches addressing all relevant levels can have disastrous effects. In the wake of early adoptions of object-oriented technology, architecture in-the-large has often been neglected. Objects were happily created and wired, all across a system. Layers were not introduced or not respected. Lines between a base and its extensions were not drawn or breached [...] The result

is that object-oriented legacy is already a problem.

Micro-workflow encapsulates workflow features in separate components. At the core of the architecture, several components provide basic workflow functionality. Additional components implement advanced workflow features. Through composition software developers extend the core only with the features they need. This design localizes the changes required to tailor a workflow feature to the component that implements it. Additionally, developers can add new features by building new components. However, this approach increases the complexity of the architecture [122]:

> The architecture of component-based systems is significantly more demanding than that of traditional monolithic integrated solutions. In the context of component software, full comprehension of established design reuse techniques is most important.

This thesis proposes the micro-workflow architecture and shows how to build it. I claim that micro-workflow provides a better way of implementing workflow functionality within object-oriented applications. I will show that starting with an architecture aimed at developers and using techniques specific to object systems and compositional software development, one can pick and choose the workflow features, customize the workflow functionality, and integrate it within applications.

1.3 The Method

Engineering disciplines have behind them large bodies of theory accumulated over long periods of time. But software engineering has a much shorter history than most engineering fields. Consequently, software engineers don't "calculate" software designs. Instead, they "follow guidelines and good examples of working designs and architectures that help to make successful decisions" [122]. Therefore in the context of software engineering, communicating experience, insight, and providing good examples are important tasks.

Software architectures strive to balance understandability, functionality, and economy. An architecture is not an end product. Rather, it provides a holistic view. Developers build systems according to invariants and prescriptions specific to a particular architecture. Consequently, they should understand the design decisions and the balance of forces corresponding to the chosen architecture.

My research provides a new way of *building* object-oriented workflow systems and flow-independent applications. Consequently I have taken an approach that is appropriate for this objective. The thesis focuses

on an object-oriented workflow architecture designed with customization and flexibility as chief considerations. It also shows how to build the architecture, extend it, and use it to implement object-oriented applications that implement processes for reviewing proposals, treating strep throat, and tracking the treatment of newborns. Building the architecture involves translating the abstractions it provides into a programming language. Using the architecture involves building components that provide advanced workflow features, as well as implementing workflows with different requirements. In effect, this approach teaches developers how to apply the research reported in this thesis in a fruitful manner.

1.4 Contributions

The micro-workflow architecture allows software developers to choose the type of workflow features they need. It enables them to tailor the existing functionality through techniques specific to object systems, and add new workflow features by composition. These characteristics represent a significant departure from traditional workflow architectures.

In this thesis I will:

- Demonstrate that object-oriented technology provides a complete architectural style for workflow management.
- Demonstrate that a lightweight workflow architecture designed with reuse and flexibility as chief considerations can be extended to provide features typical of workflow systems.
- Prove that this type of architecture can be built, and that it can implement workflows with different requirements.
- Show that micro-workflow provides a viable alternative for software developers who need customizable workflow functionality within object-oriented applications.

1.5 Thesis Organization

The thesis is structured as follows. Chapter 2 introduces workflow and discusses its characteristics. Chapter 3 introduces the micro-workflow architecture, emphasizing the differences from traditional workflow architectures. Chapters 4 and 5 discuss the design of an object-oriented framework for micro-workflow.

Chapter 6 uses three case studies with different requirements to provide a qualitative and quantitative evaluation of the architecture. Chapter 7 compares micro-workflow to other research efforts that focus on similar issues. Finally, Chapter 8 draws conclusions and discusses open issues.

Chapter 2

Workflow

Since this is a thesis about workflow management, this chapter delineates what workflow is and what it is not. Workflow shares characteristics with many other systems. This chapter also discusses the characteristics that set workflow apart from them. Understanding the differences helps developers adapt techniques developed for these systems to workflow management.

2.1 Workflow Technology

Workflow technology has been used for decades. In the 1970s, Hammer and colleagues [52], Zisman [141], and others focused on procedures for Office Information Systems. Around the same time, Ellis and Nutt [29] recognized the importance of developing models of office procedures while working on Officetalk, an experimental office information system at Xerox PARC.

Research during the 1980s placed more emphasis on process models. Winograd and Flores proposed a speech act model [135, 81] while Ellis and Nutt worked on Petri net models [30]. Workflow expanded into fields like office automation and document imaging. But the technology didn't meet the expectations of business users. Unlike the rigid workflow solutions of that time, environments for situated work offered a less restrictive alternative and captured some of the momentum.

Researchers realized that the success of workflow technology was limited by their narrow perspective. Therefore, they reconsidered workflow as a multidisciplinary endeavor, located at the intersection of different areas of computer, management and social sciences. This broad perspective contributed to the return of interest in workflow technology in the 1990s. During the last few years workflow has been the focus of intense activity in terms of products, standards and research work [83].

2.1.1 Definitions and Example

Since its early days, researchers have proposed various definitions for workflow. Many of these definitions define workflow within a single domain. Probably the most notable and widely documented domain is Business Process Reengineering (BPR) [51]. Later workflow technology was widely deployed in banking, accounting, manufacturing, brokerage, insurance, healthcare, telecommunications, customer service, and engineering, and more recently, scientific experiments. Therefore, I prefer the following definition because it doesn't depend on a particular domain:

> A **workflow** represents the operational aspects of a work procedure: the structure of tasks and the applications and humans that perform them; the order of task invocation; task synchronization and the information flow to support the tasks; and the tracking and reporting mechanisms that measure and control the tasks.

The workflow literature refers to the software that enables people to define and execute workflows as workflow management systems (WfMS). A workflow management system automates processes by managing jobs and resources. The workflow reference model [57] provides the following definition:

> **Workflow management system**: a system that completely defines, manages and executes "workflows" through the execution of software whose order of execution is driven by a computer representation of the workflow logic.

A workflow management system automates the *process logic*. Humans and software applications (processing entities) perform workflow tasks, thus implementing the *task logic*. This separation of process and task logic allows workflow users to modify one without affecting the other. It also promotes software reuse and the integration of heterogeneous software applications.

Let's consider an example that shows how workflow separates process logic from task logic. The process corresponds to a simplified procedure from the medical world—the treatment of strep throat.

> The strep throat process begins with a patient who suspects that she may have strep and goes to the doctor to seek medical attention. The doctor examines the patient and tests whether she has strep throat. If the results are positive, the doctor prescribes a treatment. Based on the patient's medical records, the doctor can treat strep throat in two different ways. If the patient

is not allergic to penicillin (an antibiotic), the doctor prescribes this treatment. Otherwise, he prescribes the sulfa drug. Next a nurse takes the prescription and instructs the patient how to about follow the treatment. If the prescription contains penicillin, she also warns the patient about the possibility of an allergic reaction to antibiotics. The patient goes home and starts taking the pills. Two days after the beginning of the treatment the nurse checks with the patient to see whether there have been any improvements. She also reminds the patient to continue taking the pills even if her condition has improved. At the end of the treatment, the nurse again checks the state of the patient.

This process involves two processing entities (sometimes referred to as "workflow actors" in the workflow literature), the doctor and the nurse. They provide medical knowledge and expertise. The doctor *examines* the patient and *prescribes* the treatment. Likewise, the nurse *administers* the prescription and *checks* the patient's condition. All of these activities correspond to the task logic and belong to the application domain. Within the workflow domain, the process logic specifies the actions performed by the processing entities and their order. It also notifies the nurse about checking the patient's condition during the treatment.

2.1.2 Workflow and Process Automation

Workflow automates processes that fit a two-tier model [45], as Figure 2.1 illustrates. In this model the *flow tier* contains the process logic, while the *work tier* corresponds to the basic process activities. The flow tier controls and automates the coordination of the work tier. Workflow management provides an environment for the definition and enactment of the process logic.

Process technology has become ubiquitous [118]. For example, people who combine activities which together realize an objective that is of value to the customer deal with *business processes* [81, 51]. People who distribute and coordinate activities among workers and information system resources deal with *information processes* [81]. People from manufacturing relate activities that are concerned with the production of physical products and deal with *material processes* [121]. Scientists focusing on activities that are part of scientific experiments deal with *scientific processes* [80].

Workflow users who design and optimize these processes work with the high-level process descriptions on the flow tier. For example, in a telecommunications billing system process designers decide how the system handles delinquent bills. Likewise, in an insurance system they set up how new applicants are

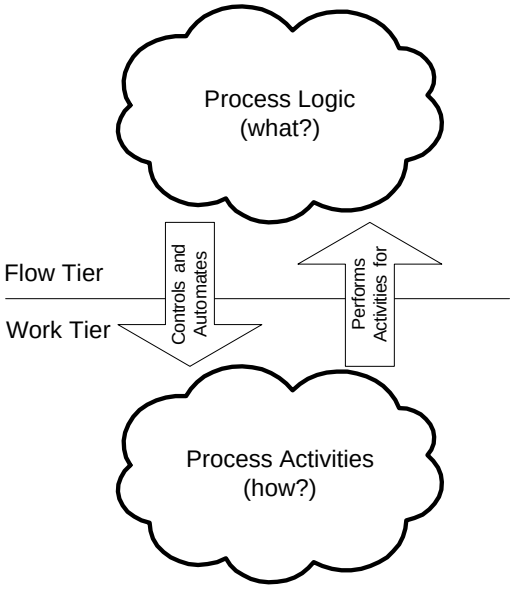


Figure 2.1: The process logic and activities are partitioned on the flow and work tiers.

evaluated. But they don't care how the workflow actions on the work tier are actually implemented in software.

2.2 Workflow Features

A recent study estimates the number of available workflow products between 200 and 300 [118]. Typically a new workflow system differentiates itself from others (with the goal of capturing a larger market segment) by offering features that are not available in other systems. But the majority of these systems share a small set of common features. Understanding these common features helps potential workflow users answer questions like "What benefits would I get from using workflow technology?" or "Could this workflow system solve my problem?"

2.2.1 Flow-Independence

There are many different concerns in a piece of software. For example, data management and user interface represent two of the aspects that many applications have to deal with. Good developers would like to write software such that every design decision is encapsulated into a component [100]. This would enable them to revisit a design decision and change it without affecting other parts of the application.

However, it takes a long time for a community to learn how to separate different concerns. When this happens, developers build a subsystem that encapsulates how the application handles a specific problem. For example, it took time to figure out how to separate the issues of low-level data management and user interfaces, but developers learned to cut down the effort. Database systems enable developers to move the knowledge about the application data from the application code into a database schema. The application manages its data through the mechanisms provided by the database system. Therefore, applications that rely on database technology for data management become *data-independent*. Likewise, user interface frameworks handle the issues of user interfaces. They enable developers to build *interface-independent* applications. Although there's no perfect way to isolate the issues of data management and user interface, to a large degree the above techniques eliminate data and user interface dependencies.

Workflow enables developers to separate the flow between an application's components/modules/objects from the application (i.e., the process). Flow-dependent software implements application-specific components and the flow between them. Most applications fall into this category since usually the underlying process emerges as the application evolves. In a more general context, Parnas and Clements observe [101]:

> ... Programmers start without a clear statement of the desired behavior and implementation constraints. They make a long sequence of design decisions with no clear statement of why they do things the way they do.

But the intertwined process logic and application code becomes a hindrance when developers make changes. Process modifications require changing the application. Likewise, changing the application's components affects modifications in the process implementation.

As databases and user interface frameworks remove data and user interface dependencies, workflow makes applications *flow-independent*. Software developers implement the process models within their applications with workflow technology. Since application components have no knowledge of the sequencing of activities and their interdependencies, changing the process doesn't affect them. Thus, workflow applications become flow-independent. Additionally, workflow technology allows developers to use workflow-specific features that otherwise would be too expensive to hand craft every time they build a new application.

Drawing an analogy between data-independence and flow-independence, Leymann and Roller predict [72]:

Just as the insight into the importance of data independence for production applications resulted into the overwhelming use of database management systems, the discovery of benefits of flow independence will foster the use of workflow management systems for building flexible applications.

2.2.2 Domain-Independence

The partitioning typical of flow-independent applications (Figure 2.1) keeps the workflow outside the application domain. Thus applying workflow to a particular application domain requires providing components that perform domain-specific work. This characteristic makes workflow technology applicable to a large number of application domains.

For example, Jackson [63] and Georgakopoulos and colleagues [43] discuss examples from the telecommunications industry. Dinkhoff and colleagues [26] apply workflow to administrative processes for property management. Vossen and Weske [130] use workflow technology for geoprocessing applications. McClatchey and colleagues [80] and Kováks [71] employ workflow in the context of the Compact Muon Solenoid high energy physics experiment. Yang and Papazoglou [138] identify workflow as part of the reference architecture for interoperable e-commerce applications. Leymann and Roller [72] discuss the application of workflow technology for software distribution, security management, and business-process-oriented systems management.

However, there are also workflow implementations that focus on vertical markets. For example, Teoss 2000 from InConcert [60] targets the telecommunications market; HP's Changengine [56] targets business administration; METEOR$_2$ [116] provides workflow solutions for the health care industry; etc. But the key point here is that a wide range of application domains can benefit from workflow technology.

2.2.3 Monitoring and History

The separation of process logic from the application components enables workflow to tap into the process level and collect information about its execution. Workflow systems provide this data at run time and after the process is complete.

The workflow system manages the runtime data corresponding to each running process. A workflow monitor enables workflow users to examine this information at run time. What workflow users can do with

this information depends on the process, as well as on the features provided by the workflow management system. The type of available actions can range from displaying process information to the early identification of out-of-line situations.

Workflow management systems also record the state of the running processes as these unfold in time. Workflow history involves a persistent store and aims at providing an audit trail after the workflow completes. This information can serve several purposes. Some workflow systems use the logged information for recovery. Workflow designers use it for process analysis, where history information forms the basis for improving the process. Workflow users can store it in a safe place for legal requirements. Health care processes keep this information since it becomes part of the medical records. Auditors examine it for auditing purposes.

Workflow monitoring and history are typical features of workflow systems. Flow-independent applications that implement their process models with workflow technology can use these features at no extra cost.

2.2.4 Manual Intervention

Workflow management systems ensure that at run time processes execute according to their definition. Under exceptional circumstances, the workflow user needs to override the process definition and manually change the course of the process. For example, she may find out that the workflow started to execute with some erroneous information. This feature enables workflow systems to handle exceptions and unique situations.

Early workflow systems didn't provide this functionality [89]. They frustrated their users, who felt that the system was merely enforcing rigid rules. Current workflow systems aim at providing various degrees of flexibility [53]. Consequently, applications that use workflow to implement processes allow their users to manually change running processes by simply leveraging this feature of the workflow management system.

2.3 Workflow and Seemingly Similar Systems

At first sight, workflow doesn't seem too different from other systems that solve similar problems but in different contexts. This section contrasts workflow with these systems and points out the characteristics that set it apart.

2.3.1 Workflow and Programming Languages

Implementing workflows amounts to writing a process specification in a workflow language. Workflow designers assemble process definitions with tools that expose only the parts of the domain that they understand. This definition specifies the sequence of activities, their interdependencies, and the data exchanged between them on the flow tier (Figure 2.1). Therefore, isn't workflow just a domain-specific programming language?

Although a process specification contains constructs similar to the ones found in programming languages, workflow management systems have many features that are unique to workflow. For example, features like monitoring, history, and manual intervention don't exist in most programming languages. Additionally, the domain-specific requirements make workflow execution different from program execution. But the similarity between programming languages and workflow enables the latter to benefit from features specific to the former. For example, Vaishnavi and colleagues [127], Meijler and colleagues [82], and Edmond and Hofstede [28] discuss reflection in the context of workflow, while Chiu and colleagues [18] propose adaptive workflow through exception handling.

Some workflow systems provide a workflow definition language which their users use to define workflows. The language reflects the structure of the underlying process model. Examples include the Flow Definition Language (FDL) [72] used by IBM's MQSeries Workflow, and the Workflow Process Definition Language (WPDL) proposed by the Workflow Management Coalition [21]. Other workflow systems provide graphical process editors. This approach allows users to build processes by dragging, connecting, and configuring process building blocks with a mouse. Internally, the process editors translate the graphical representation into an internal format/language understood by the workflow system. However, according to Leymann and Roller only definition languages can provide the "necessary precision" for nontrivial process models [72].

2.3.2 Workflow, Operating Systems and Batch Systems

There are many systems that deal with jobs and resource management and scheduling. For example, operating systems deal with processes. Likewise, load share systems manage batch jobs that execute programs without user intervention. Therefore, isn't workflow just an operating system running in user space, or a batch system enhanced with graphical editors?

Workflows run for a long time. For example, a business process for mortgage loans is active for years. In

fact, this characteristic is one of the reasons why workflow systems record process evolution into a persistent store (Section 2.2.3). Workflow users can't afford loosing the process in case of a hardware failure. In addition to slow processes, workflow also deals with slow process activities. For example, the loan officer in charge of a mortgage application might require several days to evaluate a mortgage request.

Operating systems also deal with processes management and resource scheduling. In fact, due to the similarities between operating systems and workflow systems, sometimes the latter are referred to as "application operating systems" [72]. However, operating system processes have different temporal characteristics. Compared to workflow, their lifespan is short. Their "activities" (i.e., user programs) also execute faster than workflow activities. Additionally, while operating systems typically provide information about the running processes, they don't offer other workflow-specific features like history or manual intervention.

Batch systems manage processes that execute without user input. Each batch job requires access to a set of shared resources. A load share system provides controlled access to the shared resource and ensures that each job obtains the resources that it needs to run. Typically these systems don't require user interaction and run unattended. For example, the National Center for Supercomputing Applications (NCSA) uses a Load Share Facility (LSF) to manage batch jobs on their supercomputers. Batch files contain resource specifications and job specifications. The resource specification provides the number of requested processors, and memory and running time estimates. Based on this information, the LSF system generates a schedule that attempts to maximize system utilization. For instance, two jobs requesting 4 and 8 processors respectively can run at the same time on a 16-processor host.[1] At the same time LSF attempts to minimize the turnaround time for each job.

To summarize, workflow, operating, and batch systems have different temporal characteristics. At the one end of the spectrum, the typical workflow has running times on the order of days, weeks, months, or even years. At the other end, operating systems processes usually complete in minutes. Between these extremes batch jobs have execution times on the order of hours. In addition, while workflow processes involve human workers, operating systems processes and batch jobs execute without user interaction. Figure 2.2 sketches the temporal characteristics of these systems.

[1] Assuming that the host can accommodate the memory requirements of both jobs.

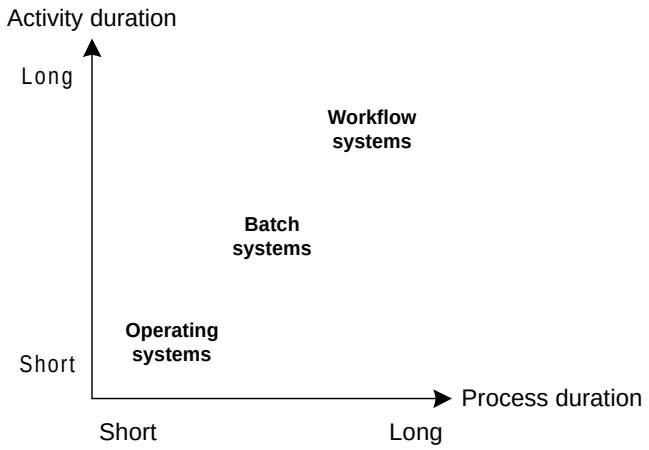


Figure 2.2: Temporal characteristics of operating, batch, and workflow management systems.

2.3.3 Workflow and Situated Work Environments

Workflow management systems coordinate process actors towards a common process goal. Typically process actors consist of workflow users and workflow-enabled applications. The field of situated work deals with providing environments where people work together on a common problem. Therefore, isn't workflow just an extended situated work environment that in addition to people also accommodates software applications?

Software that leverages computer and network services to assist a group of workers in conducting their work is referred to as groupware. While both workflow and situated work belong to groupware, they differ about how the collaborative work is performed, as Nutt [89] summarizes:

> The situated work camp advocates the use of evolutionary systems to provide increasingly sophisticated personal productivity tools, with little explicit attempt to have the system stage the work to be performed. [...] The workflow camp advocates the use of models and systems to define the way the organization performs work. [...] Whereas the situated work approach explicitly omits step definition and inter-step coordination, workflow explicitly includes it.

Therefore, the fundamental difference between workflow systems and situated work environments stems from the process model located on the flow tier (Figure 2.1). A process model explicitly specifies the sequencing of, and interdependencies between the activities performed by its actors on the work tier. At run time, the workflow management system ensures that the processes execute according to their definition.

Additionally, workflow management systems have to support interaction mechanisms specific to software applications, as well as human workers. Applications that require a mix of human-performed work and software-performed work are prime candidates for workflow systems. In contrast, environments for situated work (e.g., electronic white boards) involve only humans and therefore focus on providing a wide range of mechanisms specific to human-computer interaction.

2.3.4 Workflow and Computer Simulation

Computer simulation uses software models of real world concepts to model situations that change over time [47]. Simulators gather statistics about the models they simulate. Workflow resembles computer simulation in that the process descriptions workflow systems execute are in fact software models representing real processes. At run time, the workflow management system records data about how the process model unfolds in time. Therefore, isn't workflow just a computer simulation?

Both computer simulation and workflow use software models and collect run time information about them. In fact, this similarity prompted people to use workflow for scientific experiments. Scientific workflow systems assist scientists working with computational models. They handle activity tracking and data tagging in experiments involving large amounts of data, for example DNA fragment assembly and geoprocessing [130].

However, the fundamental difference between simulators and workflow systems lies in their purpose. Computer simulation helps people understand the simulated situation. As such, simulation is useful during the design stages, when people explore the solution space. In contrast, workflow enables its users to describe how a procedure performs work, assists in the coordination and execution of the procedure, and provides information about the procedure's run time behavior. Typically people use workflow to implement process models that they have already tested and work according to their expectations. Therefore, computer simulation complements the functionality provided by workflow systems. In fact, some workflow management systems include process simulation capabilities so that their uses can verify processes before they implement them. Klaus Hagen identifies the complementary relationship between workflow and process simulation in his PhD thesis:

> Further important tools for process development are simulation and validation environments which are necessary for the initial testing of processes before they are actually installed… In a

process support context, validation facilities like process simulation can support the prevention of faults.

For example, BPR teams [51] often use process simulation to evaluate redesigned processes. A BPR team aims at replacing business processes within the enterprise with new, better processes. The reengineering effort begins with a quick study of the existing processes. Next the team proceeds to the redesign stage and produces new process models. Once this stage is completed, they use process simulation to check the models for errors and evaluate their performance. The evaluation indicates whether the reengineering effort has reached its objective or requires further work.

Business Process Modeling Tools (BPMT) allow workflow users to analyze process definitions, simulate process enactment, and analyze the simulation results. They provide an environment that simulates the processing entities and resources involved in the workflow. Designers plug in a proceess model and the modeling tool executes it on different scenarios while varying the rates and distributions of simulation inputs, and programming various processing entities and resource parameters (e.g., the response times of people or systems performing workflow activities). At the end of the simulation BPMT provide valuable information about the process model. This includes statistics on resource utilization and queue load for evaluating the number of work items that build up at run time. Additionally, they can evaluate bottlenecks in the process definition and provide animations that show how work moves through the model.

There are several reasons why process simulation is a valuable tool. First, deploying a new process for testing is expensive, and usually is not possible since it interferes with the existing process. In contrast, simulation is cheap and remains confined to the virtual world provided by the simulator. Second, the processing entities and resources of the workflow determine how fast the process unfolds in time. In other words, they work in real (i.e., wall) time. For example, a workflow that involves lab work on a blood sample has to wait until the test results are available. In contrast, simulation is event driven and generates the test result instantly based on statistical information.

Leymann and Roller [72] suggest determining how a process model handles the work load through two types of simulation:

Analytical simulation Process modelers use the probabilities associated with the control connectors to derive the number of times each activity executes. This type of simulation doesn't account for resource limitations. It yields the probability that the process unfolds in a particular way, and lower-bound

estimates for the completion times. Analytical simulation has the advantage that it uses only basic statistical information about the process resources. Additionally, it can be performed quickly with a low amount of computational resources.

Discrete event simulation When the analytical simulation shows that the processing entities and resources can handle the workload, discrete event simulation produces more details about the process model. This type of simulation simulates the behavior of a workflow system that implements the process. Daemons representing process actors and resources generate events. Modelers specify different distribution patterns for the daemons that create processes or perform the workflow activities. Unlike analytical simulation, discrete event simulation takes into account the dynamic aspects of processes, like competing for resources. However, the additional information comes at the expense of increased computational resources.

For example, Workflow•BPR [137] is a discrete event simulator that models processes with Activity Decision Flow (ADF) diagrams. ADF diagrams enable the modeling of activity cost, execution time, and elapsed time. Calendars with the availability of resources during simulation enable Workflow•BPR to simulate real schedules.

Once the process designers implement the process in a workflow system, the history and monitoring mechanisms collect similar run time information. However, this data corresponds to actual measurements rather than simulation results. Process modelers may choose to feed it back to the process simulator and increase the accuracy of the simulation. This reason prompted Georgakopoulos and Tsalgatidou to propose a combination of BPMT and WfMS for comprehensive process lifecycle management [45].

In summary, people use computer simulation to understand problems involving processes, and workflow to solve them. Workflow systems may in fact contain simulators. Process simulation determines the short-term impact of a process model. Simulators provide process designers with information about process models. This data enables them to address practical concerns like resource utilization, queue load, bottlenecks, etc.

2.4 Workflow Standards

Despite the fact that people have used workflow technology for over two decades, a few years ago no workflow standards were available. But as workflow expanded from image and document-routing to business reengineering to mainstream middleware for process automation, interoperability issues became important. Consequently, during the last few years the workflow community has been working on standardization.

The first standardization effort dates from 1994. The Workflow Management Coalition (WfMC)—an organization of workflow product vendors, researchers, and users founded in 1993—developed the Workflow Reference Model [57]. Initially the reference model focused on defining programmatic interfaces to workflow engines, aiming at standardizing the following five interfaces (see also Figure 2.3):

Interface 1 defines a common format for the interchange of static process specifications.

Interface 2 enables workflow participants to control process execution and manipulate work items.

Interface 3 provides access to the workflow applications.

Interface 4 enables different workflow servers to interact with each other.

Interface 5 provides an entry point for administration and monitoring tools.

These standards evolved towards a set of more abstract specifications and were adapted to business objects and message-brokering environments [115]. Consequently, the focus shifted from specifying APIs towards the specification of meta-models and abstract interfaces. The recently adopted OMG Workflow Management Facility adapts the WfMC runtime standard to a business object execution environment [94]. The OMG is currently working on finalizing the standardization of the workflow facility.

However, the absence of standards encouraged vendors to build closed systems around proprietary interfaces. People use workflow systems as black boxes and have no or limited access to their internals. Sheth and colleagues [118] observe:

> . . . [T]he lack of real standards combined with a large volume of vendors has created a scattered landscape where customers are reluctant to invest in workflow products. The numerous workflow-management systems on the market today are based on different paradigms and offer contrasting functionality.

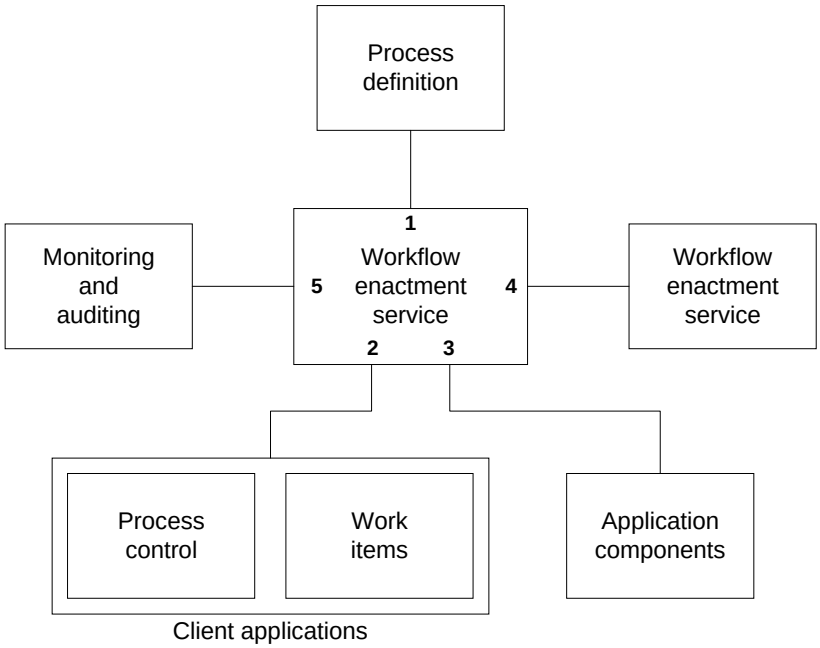


Figure 2.3: The Workflow Management Coalition's Workflow Reference Model.

Therefore, most workflow systems focus on packaging a comprehensive set of features hoping that users never need to look under the hood. This yields heavyweight systems with monolithic architectures. Additionally, it makes them hard to customize, integrate, and tailor to particular applications.

But standards are not perfect. Research efforts in the workflow domain have uncovered weaknesses in the reference models adopted by the WfMC and the OMG. For example, Paul and colleagues [103] point out that the since the WfMC Reference Model resembles the architectures of current workflow systems, *it inherits their problems*. The monolithic server of the WfMC standard impedes the flexibility and scalability of workflow systems. I have also criticized two of the proposals submitted to the OMG and discussed how their authors can improve their design methodology [73].

2.5 Workflow System Examples

People with different backgrounds regard workflow technology from different perspectives. Each perspective emphasizes things important to a particular community, leading to a different solution. This yields a wide range of approaches for building a workflow management system. To illustrate the variety of solutions, this section discusses three workflow systems implemented around active database technology, declarative

specifications, and communication infrastructures.

2.5.1 TriGS$_{flow}$

Stefan Rausch-Schott's PhD thesis proposes TriGS$_{flow}$, a flexible workflow framework supporting frequently changing requirements [111]. The TriGS$_{flow}$ architecture is based on an object-oriented database (GemStone/S), Event-Condition-Action (ECA) rules integrated into the object-oriented data model, and "roles" that support object evolution. The framework employs object-oriented database technology to implement its workflow model. TriGS (**Tri**gger system for **G**em**S**tone), an active extension for GemStone, provides the active component. Figure 2.4 shows the TriGS$_{flow}$ architecture.

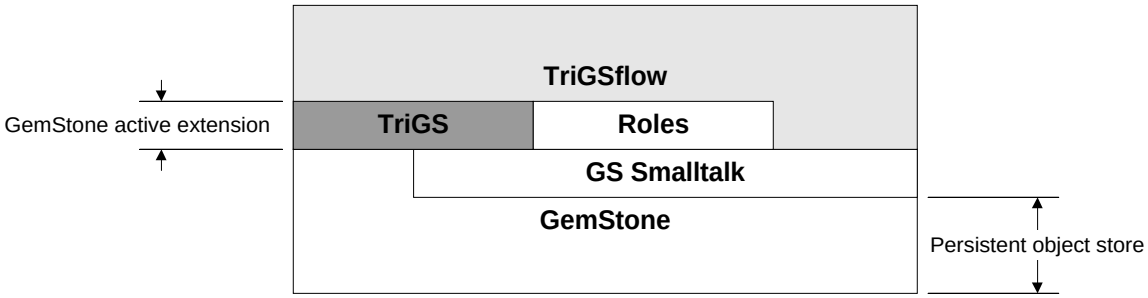


Figure 2.4: The TriGS$_{flow}$ architecture.

Rausch-Schott focuses on the transactional aspect of workflow management systems. He employs a generic activity-based workflow model based on workflow types, workflows (which he calls folders), history items, and work items. The TriGS$_{flow}$ workflow model also includes organizational, informational, and communication aspects. The organizational aspect specifies the structure of the business organization and provides the predefined classes Department, Agent, and Role. The informational aspect provides agent selection and coordination policies. And the communication aspect enables the asynchronous exchange of data between agents with the WorkList and WorklistItem classes.

The framework uses ECA rules for a wide range of functionality specific to workflow. Rules of the form

```
ON Event IF Condition DO Action
```

implement activity ordering, agent selection and synchronization, worklist management, and logging.

ECA Rules for Activity Ordering

TriGS$_{flow}$ represents the flow control with activity nets and provides control structures for sequencing, branching, and joining. The framework maps the relationships between the activities of an activity net into ECA rules. According to this mapping, a sequence maps to a single rule. OR or XOR branches map each relationship to a separate rule. A concurrent fork maps all its relationships to a single rule. An XOR join maps each relationship to a separate rule, while an OR join maps its relationship to one or more rules, depending on semantics. Finally, an AND join maps all its relationships to a single rule.

The *event* corresponds to the completion of the preceding activity (or activities in the case of an AND-Join or OR-Join). The *condition* checks whether the activity at the other end has to execute. And the *action* notifies the agent responsible for the next activity. Figure 2.5 (a) shows this mapping for a sequence with two steps. The framework fires the event as soon as Activity A completes. TriGS$_{flow}$ detects A's completion by monitoring the perform: message.[2] Next the framework evaluates the condition to decide whether to execute Activity B. If this condition is true, the framework sends the notifyAgent: message. This inserts the corresponding item into the worklist of the agent in charge of Activity B. Therefore, every rule for activity ordering contains the perform: message in the event part and the notifyAgent: message in the action part.

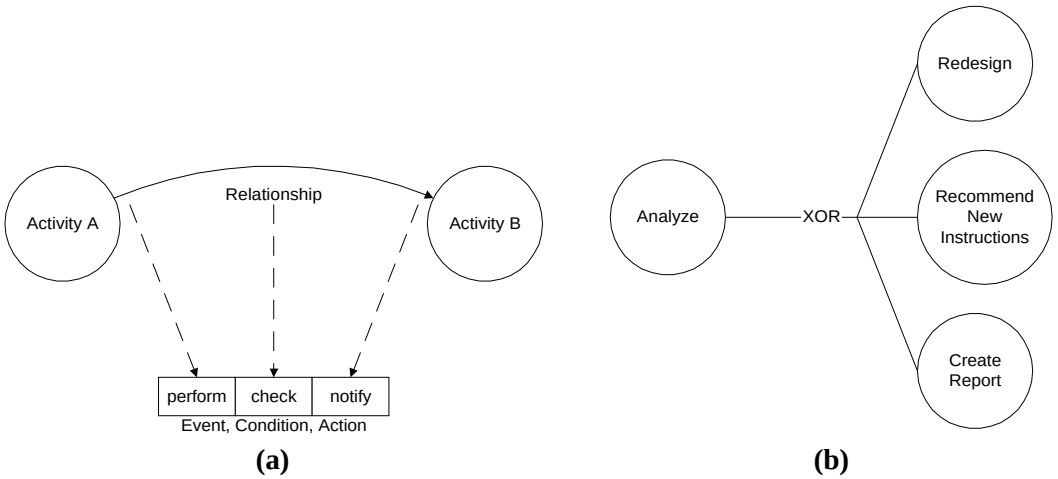


Figure 2.5: TriGS$_{flow}$ maps relationships between activities to ECA rules.

Figure 2.5 (b) shows a three-way XOR that illustrates the application of ECA rules for activity ordering. This fragment is part of a product maintenance workflow based on customer feedback. After the activity

[2]The perform: message represents the foundation of the Smalltalk message sending mechanism.

"Analyze reports" executes, the rules trigger either "Redesign," "Recommend New Instructions" or "Create Report." The XOR branch maps into three rules with complementary conditions. When the perform: message that triggered "Analyze Reports" returns, the framework fires all three rules for actAnalyzeReports, an instance of the Activity class. Then it evaluates the conditions and executes the action corresponding to the rule for which the condition evaluates to true. The conditions discriminate the amount and type of errors. For example, the "Redesign" branch corresponds to more than 10% conceptual errors and is defined as follows:

```
DEFINE RULE R_Redesign
ON POST (Activity,perform: actFolder) [trgObj==actAnalyzeReports] DO
IF ((actFolder docNamed: 'analysis report') getElems: 'subject')
  detect: [:item|(item getAttrValue: 'errortype' = 'conceptual') and: [item getAttrValue: 'percentage' > 10]] THEN
EXECUTE a_redesign notifyAgent: actFolder
END RULE R_Redesign.
```

ECA Rules for Agent Selection and Synchronization

Besides activity ordering, TriGS$_{flow}$ employs ECA rules for agent selection and synchronization. At run time, a set of rules determines agent selection based on the workflow state. This scheme enables the framework to implement different policies and switch between them dynamically.

TriGS$_{flow}$ fires the *event* right before it distributes an activity among the agents worklists. The *condition* determines the set of actual agents according to some selection policy. Finally, the *action* assigns the folder to a qualified agent. Figure 2.6 illustrates this mechanism. The condition selects the worklists corresponding to agent 1 and 2 (in the shaded box). The action assigns Activity B to Worklist 1.

Rausch-Schott's thesis provides two examples. The first example implements a minimal workload policy. Here the condition considers the set of possible agents and determines the agent with the minimum number of worklist entries. The second example takes into account the agent's qualifications. This rule considers the set of possible agents and selects one agent who is eligible to perform the activity. The definition for this rule follows. In contrast to activity ordering, the framework signals the event before it sends the notifyAgent: message.

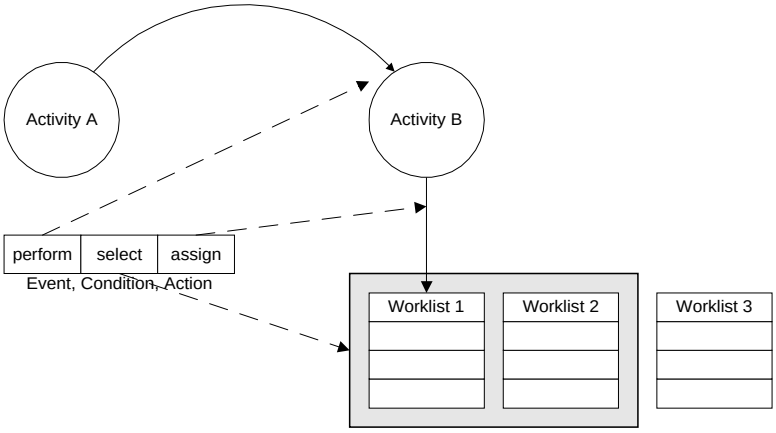


Figure 2.6: ECA rules for agent selection.

DEFINE RULE R_Qualified

ON PRE (Activity,notifyAgent: actFolder) [trgObject==actRedesign] **DO**

IF (trgObj possibleAgents) selQualifiedFor: ((actFolder docNamed: 'analysis report') getElems: 'subject') **THEN**

EXECUTE (trgObject actAgRoles) removeAll; add: conditionResult

END RULE R_Qualified.

ECA Rules for Worklist Management

ECA rules also implement worklist management. A set of rules coordinates the processing of folders queued within the worklist of a particular agent. Rausch-Schott provides examples for policies that start worklist processing whenever the framework removes or inserts a work item from or to the worklist, respectively.

Other rules implement policies for ordering and removing work items within/from work lists. An example from the former category provides two priority levels. Within a worklist, high priority items assigned to the agent jump ahead of existing items with low priority. Likewise, the framework removes work items when agent synchronization policies distribute a work item to all eligible agents and mark it as "taken care of" as soon as one agent starts processing it.

ECA Rules for Logging

Finally, TriGS$_{flow}$ uses rules to log information about workflow execution. The ECA scheme provides reduced coupling between logging and the actual execution. At run time, different rules monitor the relevant time points and situations. The framework reacts to these events by adding a new entry to the history log.

TriGS$_{flow}$ logs the time of agent notification, activity starting times, and activity termination times. A separate rule logs each kind of information. These rules don't have the condition part since the logging takes place anyway. For example, the following rule logs the agent notification time:

DEFINE RULE R_HistoryNotificationTime

ON POST (Activity,notifyAgent: aFolder) **DO**

EXECUTE aFolder addHistoryItemFor: trgObj notifiedAt: (DateTime now)

TRANSACTION MODES (A:PARALLEL)

END RULE R_HistoryNotificationTime

2.5.2 Vortex

The Vortex project at Bell Labs proposes a new programming paradigm for the specification of decision-making activities in workflows [58]. Vortex employs an artifact-based process modeling methodology and focuses on capturing how artifacts are processed within an organization. Workflows focus on the processing of attribute values for artifacts. This approach is best suited for workflows that don't have a well-determined number of steps, for example web-based store fronts.

Vortex adopts a blackboard architecture where modules compute attribute values. Modules listen for events and "enabling conditions" trigger their execution. A Vortex application consists of several Vortex programs, each of which handles a different class of events. Programs do not have an explicit representation of control flow. Rather, they infer control flow at runtime from the data flow requirements of modules and their enabling conditions. Programmers specify only the conditions under which tasks should be performed.

At the core of the Vortex execution model an interpreted engine executes specialized parallel, procedural programs. A framework that maps declarative Vortex workflows to the engine completes the execution model. The mapping framework supports several strategies that offer tradeoffs between system loads and response times. Consequently, Hull and colleagues [59] study optimization algorithms that decide which attributes can be eagerly computed and which attributes are not needed.

2.5.3 METEOR$_2$

METEOR$_2$ is a workflow research project at the University of Georgia [117]. It aims at providing a multi-paradigm transactional workflow management system capable of supporting large scale, mission critical, enterprise-wide and inter-enterprise workflow applications in heterogeneous, autonomous, and distributed environments.

METEOR$_2$ targets non-technical users and employs a graphical designer with three components. The map designer enables users to define the workflow map with the ordering of tasks and dependencies among them. Users employ a data designer to specify the data objects manipulated by tasks. Finally, a task designer enables them to define the task structure.

Once users complete the design stage, METEOR$_2$ converts the designs into workflow intermediate language (WIL) specifications. The repository service stores these definitions into a workflow repository. A runtime code generator converts the WIL specifications into code stubs during an automatic translation process. This code implements task managers, task invocation and data access routines, and recovery mechanisms for the runtime system.

The runtime architectures provide repository, enactment, monitoring, and error handling and recovery services. One of the goals of the METEOR$_2$ project is to investigate runtime architectures based on different communication infrastructures. A CORBA-based runtime employs the CORBA services for communication, distribution, transactions and fault-tolerance. A Web-based runtime targets organizations that don't have the resources to manage a CORBA product [116]. In this case, tasks consist of CGI scripts coded in C/C++ or Perl. HTML documents and forms with hidden tags implement data flow between tasks.

2.5.4 Examples Summary

The three workflow systems presented in this section have different emphases. TriGS$_{flow}$ focuses on techniques specific to database systems. Vortex accommodates declarative workflows with a process model that doesn't represent control flow. METEOR$_2$ leverages a compiler approach and code generation to accommodate different types of communication infrastructure. Unlike TriGS$_{flow}$ and Vortex, the METEOR$_2$ project emphasizes graphical editors; it includes a code-generation phase; it has a large footprint and includes additional services and facilities besides workflow.

Each of these systems provides a workflow solution based on techniques specific to the problem it solves.

As the following chapters will show, micro-workflow has different objectives and uses different techniques.

2.6 Workflow Issues Relevant to Micro-Workflow

This chapter positioned workflow at the intersection of several areas of computer science. It should be obvious that addressing all these problems is beyond the scope of a single dissertation. This thesis covers the workflow features described in Section 2.2 of this chapter. It doesn't address workflow definition languages, graphical process builders, situated work, and computer simulation. As will become clear in the next chapter, these issues are beyond the scope of micro-workflow.

Chapter 3

The Micro-Workflow Architecture

Micro-workflow is a new workflow architecture. Unlike current workflow architectures, micro-workflow targets software developers; fully embraces object technology; revolves around a lightweight core that provides basic workflow functionality; and offers advanced workflow features as components that can be added to the core. This approach avoids the problems of current workflow architectures.

3.1 From Document Routing to Middleware Services

As workflow systems evolved from office automation and image processing to business processes to middleware services, they aimed at users with different backgrounds, concerns, and requirements. How does this evolution impact the way people build workflow systems?

The first generation of workflow systems targets non-technical users. Their users regard the workflow systems as black boxes and use them as an all-in-one package. For example, people have used systems like the Coordinator or X-workflow to automate administrative processes in banks [114]. Therefore, the early workflow systems focus on being self-contained, offering a comprehensive set of features, and providing all the support tools their users would need to solve particular problems. But they don't support other applications besides the ones specifically designed to work with the system.

The second generation of workflow systems have a broader audience. They address application integration by providing various application programming interfaces (APIs). Most workflow vendors publish a set of APIs that enables third-party application developers to write workflow-enabled applications, or wrap legacy applications into workflow adapters. Products from Ultimus [126] and InConcert [60] take this approach. However, in the absence of a common standard (as discussed in Chapter 2), this is a hard prob-

lem. Proprietary interfaces make it difficult for applications to work with more than one workflow system. For example, Ultimus provides seven interfaces [125]: Workflow Objects API, Flobot API, Forms Control API, Security API, Launch Process Specifications, Server-side DLL and Database Specifications. According to Alonso and colleagues [2], this number is in the order of several hundreds in the case of InConcert. Therefore, an application that supports several workflow products would have to implement an unreasonable number of APIs. Weske and colleagues [132] have studied the problems associated with hooking up applications to a workflow management system through APIs. Their study concludes:

> In all case studies of our survey the integration of legacy systems was a critical success factor of the project. Considerable efforts were necessary to integrate the different systems into the workflow application. One of the main issues in this context was the development of interfaces between the workflow management system and the applications.

During the last few years, people have gained a better understanding of workflow and realized its impact on building flow-independent applications. The recent adoption of a workflow management facility by the OMG shows a strong interest in workflow technology from the software development community [95]. Consequently, current workflow systems are expanding from end-user tools to mainstream middleware for process automation.

The three generations of workflow systems mentioned above target very different types of users, ranging from non-technical users to application developers. The shift from an end-user application to a service/facility for developers has a strong impact on how people design and build workflow systems. Systems that target non-technical people focus on providing support tools like graphical process builders, form designers, and application wrappers. They expose only the aspects their users can understand and hide all other details. While this enables non-programmers to use workflow systems, it also yields heavyweight architectures which are hard to reuse and customize. In contrast, software developers have very different requirements. They need systems which are easy to understand, tailor, reuse, and integrate with a wide range of development environments, frameworks, development toolkits, and applications. A workflow system targeting software developers should therefore allow them to customize its functionality, and reuse it (or parts of it) in different contexts.

The research described in this thesis starts from the observation that traditional workflow architectures are based on requirements and assumptions which do not hold beyond the context of end-user applications.

Micro-workflow provides a new architecture that *targets object-oriented software developers*. It enables them to build flow-independent applications, as well as full-fledged workflow systems. Micro-workflow shifts the focus from packaging a comprehensive set of features to:

- enabling developers to pick and choose the workflow features;
- tailoring the functionality to specific application domains; and
- extending the architecture with new features.

This represents a radical departure from the all-or-nothing style of monolithic workflow architectures.

3.2 Object-Oriented Workflow Architecture

Micro-workflow regards object-orientation as an architectural style. The object paradigm requires finding abstractions for the problem domain, partitioning the functionality, and defining interfaces. These are all hard problems, but when successfully solved they could yield powerful, reusable systems [68]. Peter Deutsch observed the importance of these steps while examining design reuse in the context of the Smalltalk–80 system [23]:

> Interface design and functional factoring constitute the key intellectual content of software and are far more difficult to create or re-create than code.

Many programmers can decompose a problem into a set of classes. What differentiates experts from novices is that the former craft their objects such that they can leverage all three characteristics of object systems—encapsulation, inheritance, and polymorphism. This requires a sound understanding of the problem domain, anticipating how requirements will change, as well as mastering the techniques of good object-oriented design [39].

At first sight, the object paradigm doesn't seem appropriate for workflow management. Object-oriented systems lack a procedural representation of control flow [54]. The decomposition into classes typical of object-oriented architectures deemphasizes the control flow, distributing it among different objects. Thus, the global control flow and behavior are less visible than in procedural programs. This is contrary to the fundamental idea of workflow [83]:

> One of the chief tasks of workflow management systems is to separate process logic from task logic which is embedded in individual user applications. This separation allows the two to be independently modified and the same task logic to be reused in different processes, thereby promoting software reuse and the integration of heterogeneous and isolated applications.

Therefore, an additional challenge of building object-oriented workflow architectures lies in providing abstractions that maintain an explicit representation of the control flow without violating the principles of good object-oriented design.

An object-oriented workflow architecture must *provide abstractions* that enable software developers to define and enact how the work flows through the system. It should also allow them to *tailor the architecture* in ways specific to object systems. Object-oriented developers use several techniques to customize object systems. White-box techniques involve specializing objects through subclassing. For example, developers should be able to add a new workflow control structure by subclassing an abstract class. Black-box techniques involve tailoring the functionality through aggregation. For example, developers should be able to add a workflow-specific feature by plugging in an object providing the desired functionality. However, the use of an objects alone doesn't guarantee the applicability of these techniques. They require designs crafted with reuse and evolution as primary considerations. As Brian Foote concludes in his study on dealing with changing requirements using object-oriented programming tools and techniques, "designing to facilitate change requires a great deal of care and foresight" [33].

The object-oriented community has developed ways of leveraging the power of objects to build flexible, customizable, reusable, and dynamic systems. It has learned how to fully exploit the characteristics of object systems and reuse successful designs. For example, a first step towards an object-oriented workflow architecture would explicitly represent the aspects (i.e., features) that are likely to change with objects. Through polymorphism, developers could mix and match various features, dynamically add new features, and tailor the architecture to particular problems. Therefore, an object-oriented architecture should be able to evolve and accommodate new requirements by leveraging techniques specific to object systems.

There are many workflow systems built around objects. However, they don't fully exploit the potential of object technology. Rather, they regard it mainly as an implementation technique [83]. Object technology is an afterthought in legacy workflow architectures. For example, Leymann and Roller describe objects as merely workflow activity implementations [72]. Projects from various research groups also use object

technology. However, the projects that I've looked at focus on different research problems (e.g., transaction models, middleware services, flexible workflow models, etc.) and therefore use objects as a exploratory vehicle. For example, the TriGS$_{flow}$ system provides an object-oriented workflow model that allows its users to build workflows by specializing and instantiating predefined classes. But as I discussed in Chapter 2, TriGS$_{flow}$ focuses on ECA rules.

So why don't workflow architectures fully embrace object technology? Unfortunately, since traditional workflow architectures target end-users, they tend to focus on the number of features instead of reuse and flexibility. Moreover, most people who build workflow systems are just starting to discover the techniques that would enable them to use objects efficiently [73]. This thesis addresses these issues. The micro-workflow architecture leverages all three characteristics of object systems to foster reuse and flexibility. As Chapters 4 and 5 will show, when properly exploited the object paradigm provides an excellent foundation for workflow management.

Micro-workflow embraces object-orientation as I have set out. The next two sections discuss the structure of the micro-workflow architecture.

3.3 Basic Workflow Functionality

Current workflow architectures are heavyweight. Recent studies find that some of the functionality found in current workflow systems "might not belong to WFMSs at all" [15]. This makes them difficult to reuse, customize, and integrate with other environments. Their users have little (if any) control over the features, and have to use these systems in an all-or-nothing manner. For example, Klaus Hagen's PhD thesis [49] identifies the narrow purpose design of current workflow architectures as a liability. He observes:

> Most WfMS provide enhanced facilities for the modeling of staff hierarchies, capabilities, and responsibilities. This functionality, together with a role-based concept for work assignment, turns out to be a powerful load balancing mechanism as long as only human collaborators have to be integrated. If it comes to load balancing between execution sites for programs, the mechanisms cannot be used and there are no equivalent means to be found.

These characteristics make it hard for object-oriented software developers to use existing workflow systems whenever they require custom workflow functionality in their applications.

A first step towards addressing these problems lies in dividing the architecture into pieces (i.e., modules or components) with different responsibilities. But where should one draw the lines between components? David Parnas has discussed the criteria used to decompose systems into modules [100]. He suggests that each module should hide a design decision from others. Therefore, instead of building an architecture that lumps together functionality along many different dimensions, I break it into small components which address separate concerns. Christoph Bussler speculates that the "normalization by componentization" will cure current workflow architectures of their problems [15].

What are the minimal responsibilities of a workflow system? A workflow management system must at least enable its users to *define* and *execute* workflows. Therefore, at the focal point of the architecture the micro-workflow core provides this functionality. This minimalistic approach yields a lightweight workflow system. But as Muth and colleagues [85] observe while doing research on a new workflow architecture, "there is always a tradeoff between a simple kernel architecture with limited functionality and the benefit of sharing functionality of a rich kernel." The challenge lies in defining the kernel functionality and its interfaces, and keeping it lightweight while enabling software developers to selectively add the features typical of monolithic workflow architectures.

3.3.1 Defining Workflows

Software developers using the micro-workflow architecture define workflows with an activity-based process model. A process model consists of key process abstractions and their relationships. Activity-based process models use activity nodes and the control flow between them. They capture how to coordinate process activities, and focus on modeling the work that has to be done. Activities may be nested to obtain a hierarchical work breakdown structure. Activities at the same level in the hierarchy typically have ordering dependencies defined among them. These dependencies control the amount of concurrency among activities [19].

The activity-based process model represents workflows with directed graphs called *activity networks*. People who use these models define processes by connecting *control structures* into an activity map. In effect, this representation places activities in the network nodes and the data passed between activities on the arcs connecting these nodes, showing the *data flow* between activities [43].

The majority of existing workflow systems employ activity-based process models. A small number of workflow systems use artifact-based process models (e.g., the Vortex system discussed in Section 2.5.2),

or conversation-based process models (e.g., ActionWorkflow [81]). These process models provide different key abstractions and focus on the artifacts produced by the workflow, or on the agreements between people engaged in speech acts [135].

The key abstractions provided by activity-based process models resemble the control structures available in structured programming languages. Therefore, an activity-based process model is a good choice for a workflow architecture that targets software developers. Current workflow systems using this type of process model offer a fixed set of control structures. They provide no means for allowing their users to define new control structures. Workflow researchers argue that in many cases this limitation prevents appropriate control-flow definition, and regard it as one of the sore spots of current systems [15]. The micro-workflow architecture doesn't have this problem. Software developers can use techniques specific to object systems to *add* new control structures, as well as *tailor* how the existing control structures work.

Commercial workflow systems provide sophisticated process builders. These have graphical user interfaces which offer iconic representations for various workflow building blocks, thus *hiding* the process model. Users define their processes by dragging, connecting, and configuring icons. For example, Figure 3.1 shows the main window of Work Manager Builder. Some editors even verify the process definition for syntactic errors, and advise the user about any potential problems. Workflow vendors regard these builders as a key feature of current systems [70]. In effect, this emphasis on the view shifts the focus from the architectural issues to presentation issues. However, graphical process builders represent a support tool for non-technical users. In contrast, micro-workflow targets object-oriented developers and therefore should *provide direct access* to the process model. It should allow them to define processes in ways specific to object systems. Thus, instead of manipulating icons, developers using micro-workflow define processes through object composition. This approach requires finding abstractions that balance power and simplicity, which is not trivial. Chapter 4 focuses on the solution and discusses its consequences.

3.3.2 Executing Workflows

Traditional workflow architectures deal with processes involving people or workflow applications. In contrast, the micro-workflow core deals with processes in a pure object world—all workflow processing entities are objects. Let's consider an example that illustrates micro-workflow. This example corresponds to a simplified billing cycle closure process from the Objectiva telecommunications billing system [6]. In this

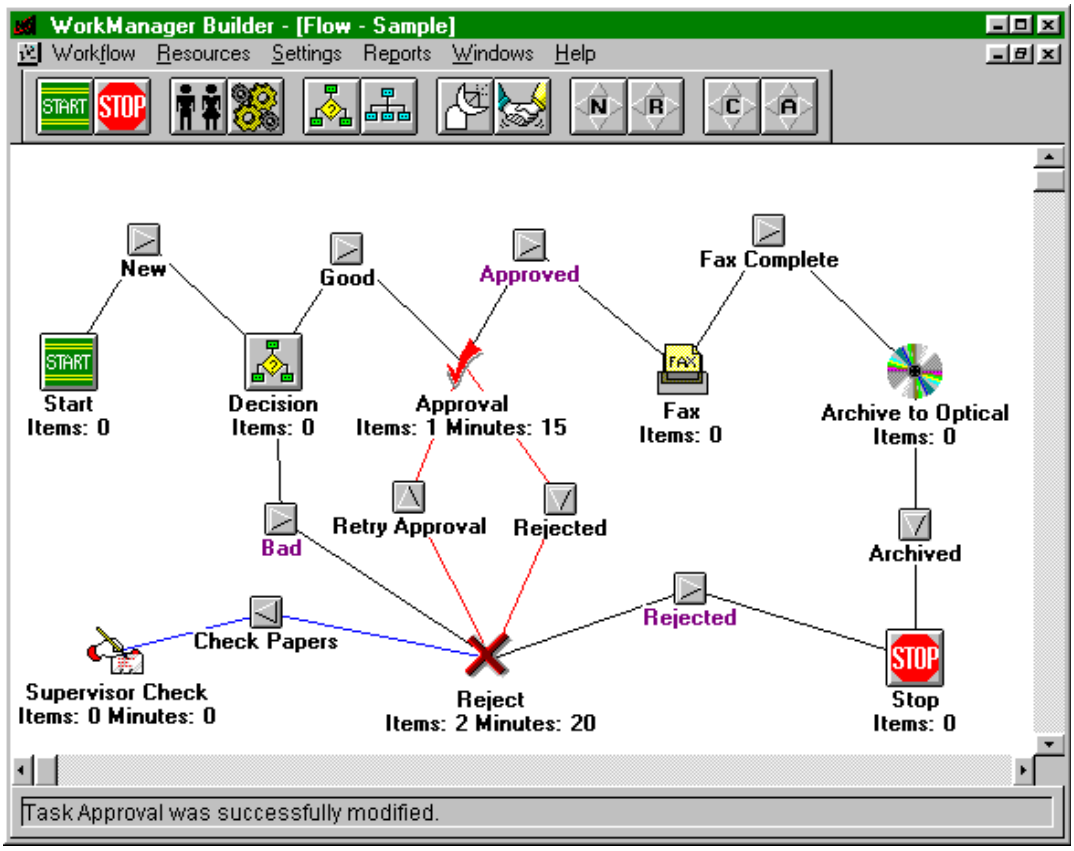


Figure 3.1: Visual process builder from Work Manager.

system, an account object holds agreements. Each agreement object represents a different product that the customer pays for.

> A customer has the following agreements with his telecommunications provider: a voice line, a data line, and a wireless phone. In addition to these services, the customer also leases a data modem. Therefore, his account object holds four agreement objects. At the end of the billing cycle, the billing system computes the balance on the account. The system iterates over the customer's agreements and asks each agreement object for its subtotal. Next the process adjusts the customer's credit and computes any additional taxes that he may be subject to. Finally, the billing system prints out a bill.

This process involves three classes of domain objects: customer, agreement and account. The agreement objects compute the *individual charges* at the end of the billing cycle. Likewise, the account object computes the *balance*, and the customer object computes the *taxes*. These activities correspond to task

logic and are provided by the customer, agreement and, account objects. The process definition specifies the actions performed by the domain objects and their order. However, it doesn't say anything about the application-specific processing that takes place within these objects, e.g., call rating, account balancing, or tax computation.

Micro-workflow takes the principles of object-orientation beyond the workflow definition stage and regards process execution as object instantiation. On the type side, the process definition corresponds to a *class*. Executing a process creates a workflow *instance* on the instance side. Process execution relies on the interplay between both sides. In the previous example, all objects comprising the definition of the call rating process reside on the type side. As the workflow unfolds in time, micro-workflow instantiates the objects corresponding to workflow execution on the instance side. This parallel between object instantiation and process execution allows for a smooth integration of the implementing technology (objects) with the implemented technology (workflow). Additionally, it facilitates the application of object-oriented techniques and methodologies to workflow management.

The micro-workflow core allows its users to define and execute processes like the billing cycle closure example. However, it doesn't provide any of the workflow-specific features discussed in Chapter 2. To allow developers to build full-fledged workflow systems, the micro-workflow architecture must provide additional functionality. For example, how would it support human workers? The message-send mechanism used by objects and supported directly by the micro-workflow core is not suitable for people. Humans have long response times and their work contributes to making workflow instances long-lived. They are also unpredictable for reasons beyond the control of the workflow system. Additionally, as far as the workflow system is concerned, humans have very limited availability. Therefore, to accommodate workflow workers, the core must support an invocation mechanism suitable for humans. Would it be possible to add this functionality only when needed, thus allowing software developers to assemble custom feature sets?

3.4 Advanced Workflow Features

Workflow products provide hundreds of features [124]. But since historically workflow management systems have targeted end users, all the features have been added in a monolithic manner, yielding heavyweight architectures. The problems with this approach are evident—monolithic systems are difficult to understand, reuse, customize, and extend.

The workflow core described in the previous section provides minimal workflow functionality. But the micro-workflow architecture allows software developers to add the workflow-specific features discussed in Section 2.2 through techniques specific to object-oriented systems. The next chapters will prove that the micro-workflow architecture provides a viable solution for developers building workflow systems or flow-independent applications. They will show that the core can be extended with the following components:

- A workflow *monitoring* component that provides information about the executing workflows.
- A *history* component that extracts the workflow information for logging purposes.
- A *persistence* component that records workflow data (e.g., the logged workflow events) to a database.
- A *worklist* component that provides the functionality required to have human workers perform workflow activities.
- A *manual intervention* component that enables workflow administrators to change a running process.
- A *federated workflow* component that provides the support required to transparently distribute workflow execution across the enterprise.

This design keeps the micro-workflow core lightweight, easy to understand and customize. A lightweight system enabling software developers to pick-and-choose the features that they need would solve the problems of monolithic, heavyweight workflow systems.

A workflow architecture that fits this description represents a radical departure from the traditional workflow architectures. But the additional flexibility has its price. First, the micro-workflow core needs support functionality to accommodate the components implementing these advanced workflow features. This functionality increases the complexity of the core without adding any new workflow features. Second, accommodating pluggable components usually involves additional overhead. This overhead occurs even when the components are not used. If these costs are too high, software developers won't use micro-workflow. This thesis will show how to build this architecture and study what impact this approach has on the core. It will demonstrate that by leveraging good object-oriented design techniques, the cost is low and worth paying for.

In summary, the micro-workflow architecture revolves around a *lightweight workflow core* (hence the prefix "micro") and focuses on enabling developers to *tailor it to particular applications* and *extend it*

through composition. In effect, this yields an open solution that allows its users to assemble custom feature sets. The challenge lies in finding the right abstractions, and designing the core and the components providing the features mentioned above such that they support this plug-and-play functionality.

3.5 Why Compositional Reuse is Hard

Compositional software reuse refers to building systems out of existing parts. Workflow relates to compositional software reuse since from a software reuse perspective, workflow acts as the "glue" that connects other systems, applications, and objects. In addition to this aspect, the micro-workflow architecture relates to compositional software reuse since software developers add workflow-specific features through composition.

The ability to produce new applications by combining existing pieces of software could have a dramatic impact on the software development process. Berlin [9] and Garlan and colleagues [40] have examined this issue. For example, Berlin's study is based on her experience with integrating five class hierarchies into an object-oriented hypertext platform comprised of over 200 CLOS classes. She found that well-designed, reusable classes are hard to integrate due to conflicting assumptions. One of the common assumptions causing mismatches is the *control model*, i.e., which components control the overall sequencing of computations.

Berlin's and Garlan's studies conclude that architectural mismatches make compositional software reuse more difficult than it seems. These mismatches center around the assumptions made by the reusable components about the structure of the application in which they are to be used. Therefore, designing a workflow architecture that can be extended through composition is a challenging endeavor. In addition to finding the right abstractions, partitioning the functionality, and defining the interfaces, the designer must craft the components avoiding any potential architectural mismatches.

3.6 When Not to Use Micro-Workflow

Micro-workflow is not an universal workflow solution, nor does this type of workflow architecture represent the best choice for all types of workflow problems. Software developers should know the types of problems suitable for the micro-workflow architecture. This boils down to evaluating whether a problem can benefit from workflow technology, and deciding whether the solution requires the features provided by micro-

workflow, or can be implemented with a traditional workflow architecture.

A wide range of problems can be regarded from a process-centric standpoint. But while some processes require the functionality and features provided by workflow management systems, others do not. For example, the definition of how a GUI works and reacts to user actions represents a process involving graphical widgets and operations on them. However, typically software developers use different metaphors and tools for GUI programming. So what types of processes do workflow systems implement? The following list shows *some* of the typical characteristics that indicate situations where workflow technology can pay off quickly.

- A set of processing entities (human workers, devices, applications, objects, etc.) carry out the specialized, application-specific actions.
- New requirements and objectives change the process definition but don't affect the domain specific processing.
- The organization that runs the process needs to collect information about how the process executes, requires information about the status of executing processes, or wants to alter process execution at run time.
- Process designers have a simulator that reads a process definition and enables them to experiment with different process parameters and evaluate what-if scenarios.

Naturally, implementing processes with a workflow tool has its costs. First, the potential users have to buy or build the workflow system. Then, they have to learn how to use it (in case they bought it), or test it (in case they built it). Finally, they implement the process. Workflow technology should be considered only when its benefits outweigh these costs.

However, not all solutions that can benefit from workflow technology require the features of micro-workflow. Before choosing micro-workflow, potential users should decide whether their problem needs micro-workflow by examining the following characteristics:

- Micro-workflow targets object-oriented software developers who build workflow systems and flow-independent applications. It involves writing code, integrating the micro-workflow components with

the application objects, and defining the process through instantiating, configuring, and composing objects. Therefore, non-programmers shouldn't use it.

- Micro-workflow allows software developers to tailor the workflow architecture to particular applications and domains. If developers can integrate another workflow product with their objects and frameworks, they probably don't need micro-workflow.

- Micro-workflow enables software developers to pick and choose the workflow features they need. Developers who use all the features of an existing workflow system and don't require any additional features probably don't need micro-workflow.

- Micro-workflow is not compatible with any of the proprietary workflow standards of office applications that have started to offer workflow functionality—e.g., Microsoft Office. People who require compatibility with families of products from different software vendors are better off using a shrink-wrapped workflow product designed for this purpose.

3.7 Thesis Contributions Revisited

This chapter has introduced the micro-workflow architecture. Most importantly, it has discussed how micro-workflow departs from the traditional workflow architectures. But in doing so it has raised several research questions. In trying to answer these questions, the next chapters will:

- Demonstrate that the object paradigm provides an excellent foundation for workflow management.

- Discuss how the micro-workflow core can be tailored through techniques specific to object systems to accommodate new requirements.

- Demonstrate that the micro-workflow architecture can be extended through composition to provide features typical of workflow systems.

- Show how to build this type of architecture, and how to implement workflows with it.

- Conclude that micro-workflow provides a better way of building workflow systems and flow-independent applications.

While examining the future of workflow management systems, Sheth and colleagues predict [118]:

> Instead of standalone workflow-management systems on which workflow applications are built, workflow capability will be built in critical enterprise systems such as ERP (Enterprise Resource Planning) and supply-chain management.

Micro-workflow, as a new architecture for building workflow systems and flow-independent applications fits this description.

Chapter 4

The Micro-Workflow Core

To test the ideas presented in Chapter 3 I have implemented the micro-workflow architecture as an object-oriented framework [23, 68] in VisualWorks Smalltalk [129]. The framework provides micro-workflow components that software developers can use right away [76]. They can change the components and tailor the framework for specific domains through techniques specific to object-oriented frameworks [112]. They can also build components that implement new features and extend the architecture through composition.

This chapter discusses the design and implementation of the framework components corresponding to the micro-workflow core. I have chosen to describe the design of the framework components using patterns [39, 14]. Patterns provide a good technique for documenting frameworks [67] because they communicate the reasons behind the design decisions. Throughout this document the pattern names appear in *slanted fonts.* Appendix A provides a brief description or each pattern.

4.1 Execution Component

A workflow definition specifies the activities that the workflow processing entities must perform to achieve a certain goal. At the core of the framework, the execution component provides the mechanism that executes workflows according to their definition, i.e, implements workflow enactment.

The key abstraction of the enactment mechanism is a *procedure*. A set of procedures define the workflow. An instance of a procedure executes according to its definition. For example, doctors follow the "strep throat treatment" procedure to treat patients with strep throat. I call an instance of this procedure, e.g., the strep throat treatment for John on November 9, 2000, a *procedure activation*.

Several procedure activations can share the same procedure. This characteristic allows the framework

to concurrently execute multiple workflows sharing the same definition. For example, Alice's strep throat treatment follows the same procedure as John's but corresponds to a different activation. Figure 4.1 sketches this situation.

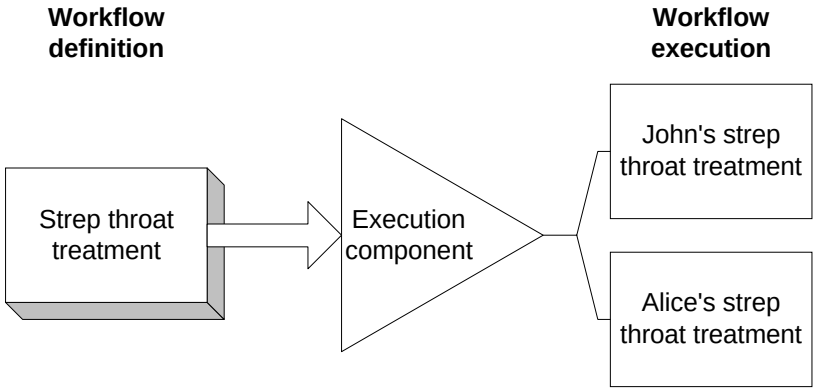


Figure 4.1: The execution component executes multiple workflows sharing the same definition.

4.1.1 Usage

The framework employs separate classes for procedure definition and activation. Procedure[1] encapsulates the behavior corresponding to the procedure definition. Instances of this class provide the rules that govern how the framework executes workflows. To enable concurrent workflows to share the same definition, Procedure should not depend on activations. The ProcedureActivation class encapsulates and manages the workflow runtime data. Instances of this class represent procedure execution events.

Procedure execution relies on the interplay between a Procedure instance and a ProcedureActivation instance. First the procedure creates an activation and provides the runtime information. In the strep throat treatment this stage corresponds to beginning a treatment for John; "John" and the data associated with him (e.g., his medical records) represent the runtime data. Then the activation executes according to the rules provided by its definition. This second stage corresponds to performing the activities required to cure John's strep throat.

This enactment mechanism is similar to an object model. The procedure definition plays the role of a class. Likewise, the procedure execution events play the role of class instances. Therefore, *workflow enactment parallels object instantiation*. Due to this similarity the framework components use techniques

[1]Throughout this document, the names that correspond to Smalltalk classes or selectors appear in sans serif fonts.

specific to object systems to solve workflow-specific problems.

There are several ways in which workflow enactment resembles an object model. First, in object systems classes define the properties and behavior of their instances. Similarly, procedures provide the definitions and rules that determine how procedure activations execute. Second, in some object-oriented languages (e.g., Smalltalk), classes create instances. The execution component implements an enactment mechanism where Procedures create ProcedureActivation instances. And finally, an object system distributes state and behavior between the instance side and the class side. Likewise, the enactment mechanism splits procedure execution between Procedure and ProcedureActivation—the former provides the rules while the latter holds the data that the rules work with.

4.1.2 Design Details

Instances of Procedure and ProcedureActivation classes play the roles of classes and instances of these classes, respectively. The Procedure class resides on the type side and the ProcedureActivation class on instance side. This arrangement (also used by Martin and Odell [78], and Fowler [36] in object-oriented analysis) corresponds to the *Type Object* design pattern [65] (see Appendix A). Activations access the "knowledge" residing on the type side by sending messages to the corresponding Procedure instance.

On the instance side, the ProcedureActivation class stores the workflow runtime information. Typically classes store the state information unique to each instance in instance variables. This solution works fine when developers know in advance the information that each instance encapsulates. But it is too restrictive for this type of framework, since it's impossible to anticipate all the instance variables developers will require to implement their workflows.

To solve this problem I store the runtime information into a context object rather than in instance variables. The context has named slots and each slot holds an object. Software developers add new state information by adding new slots to the context. This solution (of Lisp heritage) corresponds to the *Property* pattern [35] (see Appendix A). *Property* enables the context to expand at run time and doesn't require changing the code. Therefore, the context provides much more flexibility than instance variables.

The Procedure class also has a context. Procedures use this context to initialize the ProcedureActivation objects they create. In effect, the type-side context implements workflow local variables.

The Unified Modeling Language[2] (UML) [37] class diagram from Figure 4.2(a) shows the static structure of the execution component. Likewise, the instance diagram from Figure 4.2(b) shows the relationship between Procedure and ProcedureActivation objects. On the type side, the workflow definition consists of three procedures. At run time, as the workflow unfolds in time, the framework creates one instance side object for each Procedure it executes. When the framework fires off procedure1 it instantiates its corresponding workflow event activation1, and so forth. The workflow literature refers to this process as "workflow navigation" [72]—the flow of control navigates the type side process definition. Once execution completes, the instance side contains an activation object for each procedure that the framework executed on the type side.

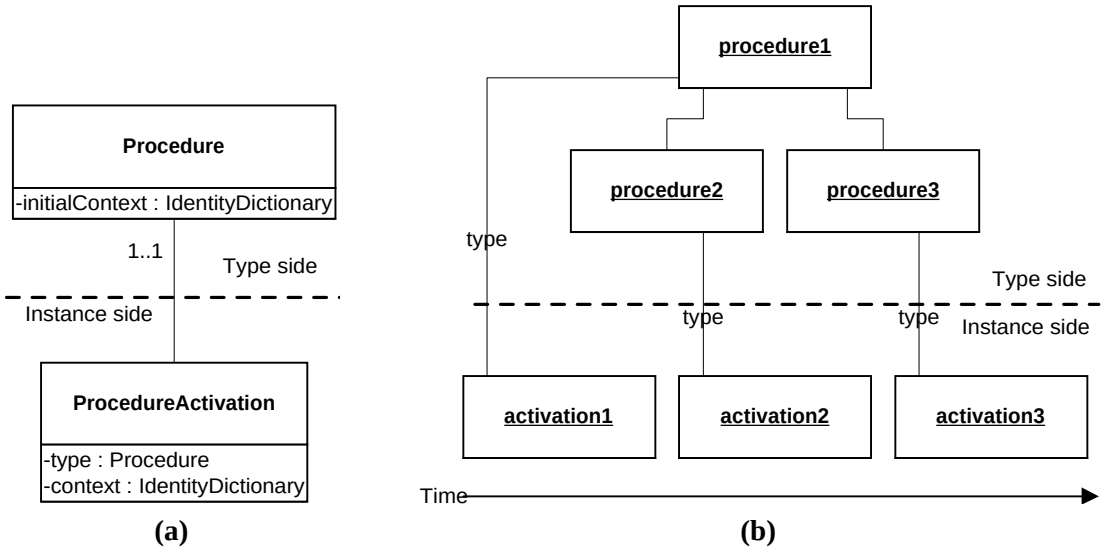


Figure 4.2: The micro-workflow execution component—class diagram (a) and instance diagram (b). The interplay between procedures and procedure activations at the core of the execution component resemble an object model

4.1.3 Discussion of the Execution Component

Most workflow management systems activate workflows by instantiating a process definition template. Then the workflow enactment service [57] executes this active instance. The micro-workflow execution component uses a mechanism that resembles an object system and revolves around the *Type Object* pattern. As the workflow executes, this mechanism creates the activations of a workflow instance and binds them to the definition.

[2]The Unified Modeling Language is an industry standard specifying diagrams and notations for object-oriented systems. I have attempted to comply with the UML standard in the class, instance, and sequence diagrams.

The difference between the template- and the *Type Object*-based execution models is that the former approach provides only a blueprint. Once instantiated, the process doesn't depend on the template any longer. The execution model based on *Type Object* relies on the interplay between procedures and activations throughout its lifespan. Here, changing the workflow definition affects all its running instances. This characteristic yields a flexible approach that can accommodate evolutionary changes [75]. Additionally, it facilitates the integration of the implementing technology (objects) with the implemented technology (workflow).

4.2 Synchronization Component

The previous section describes how the framework generates an activation when the execution component fires off a Procedure instance. The message sent to the workflow definition represents the event that triggers execution.

However, sometimes workflow enactment depends on events external to the framework. For example, let's revisit the strep throat process from Chapter 2. The doctor can't start treating John for strep throat until his lab tests confirm that John really has the illness. The treatment workflow has to wait until the lab technician completes the tests. Therefore, the execution of the strep throat workflow depends on the lab results.

The synchronization component provides a means to introduce dependencies between procedures and external events. This enables developers to synchronize workflow execution with application objects.

4.2.1 Usage

Software developers using the micro-workflow framework specify dependencies between procedures and other objects (e.g., application objects) with *preconditions*. Preconditions determine when procedures execute. A separate *manager* object handles precondition evaluation.

Let's consider the types of dependencies common to workflow systems [133]:

- A procedure that can execute only when some data item becomes available has a *data dependency*. For example, consider a workflow for the reviewing process for a conference. The workflow can't assemble the lists of accepted and rejected submissions unless all the reviewers send back their reviews.

- Likewise, a procedure that can execute only at a given time has a *notification dependency*. For instance, most billing systems generate invoices at the end of the billing cycle. In this case notifications generated by a calendar service trigger procedure execution.

The synchronization component adds a precondition to the Procedure class. At run time, the procedure blocks execution until its corresponding precondition is fulfilled. This scheme resembles the Event-Condition-Action (ECA) mechanism of active database systems [102]. ECA rules consist of three parts. The event E specifies *when* the rule should be triggered. The condition C contains a clause that determines *if* the rule is taken. Finally, the action A specifies *what* are the effects of triggering the rule. Therefore, the specification of an ECA rule looks as follows:

```
WHEN Event IF Condition DO Action
```

In the context of the synchronization component, the event corresponds to firing off a procedure, the condition to a Precondition object, and the action to continuing procedure execution. Other workflow researchers found ECA rules an "ideal paradigm" for expressing inter-task dependencies within workflows [17].

4.2.2 Design Details

The Precondition and PreconditionManager classes implement an ECA-like synchronization component.

Every Procedure instance has its own precondition object. This corresponds to the Condition part of the ECA model. In general, developers may choose to specify the Precondition's guard in a specialized language tailored to the application domain. However, for simplicity my framework uses Smalltalk blocks.

The PreconditionManager is a separate process that monitors preconditions. Procedures delegate precondition evaluation to an instance of this class. Continuing the parallel with active database systems [102], the manager plays the role of a Condition Monitor. However, I prefer the term *Manager* [120] since the object-oriented community already cataloged this solution as a software pattern.

A procedure begins executing by transferring control to its precondition. The framework initializes the precondition from the workflow runtime. Then it places the precondition into the manager's queue. In effect, this blocks procedure execution until the condition is fulfilled. In a separate thread, the PreconditionManager evaluates every queued precondition. The manager removes from its queue the preconditions that are

satisfied. In turn, this resumes the execution of the corresponding procedures.

Figure 4.3 shows how the synchronization component enhances the execution component of the micro-workflow framework. In this diagram the activation has access to the PreconditionManager object through a workflow session. Chapter 5 provides a detailed discussion of the workflow session. For now it suffices to say the workflow session contains objects that the framework uses to execute a workflow—e.g., the precondition manager.

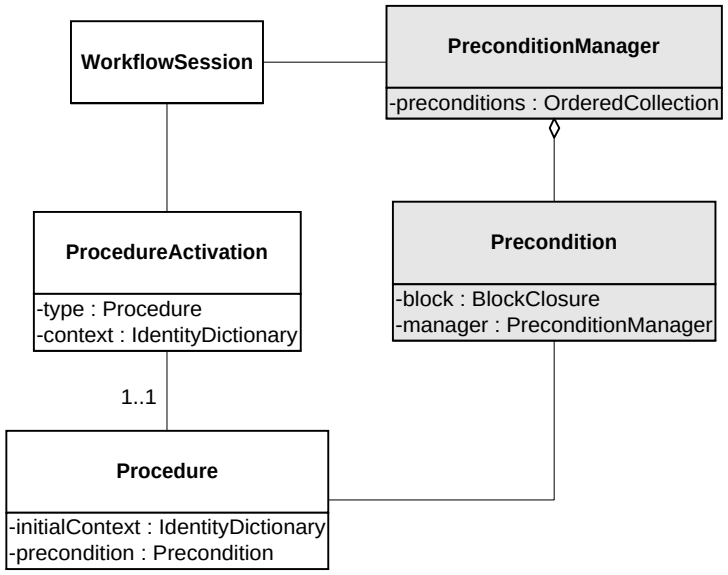


Figure 4.3: The synchronization component (grayed) enhances the execution component.

4.2.3 Discussion of the Synchronization Component

The micro-workflow component discussed in this section implements the mechanism that enables software developers to synchronize procedures. I have chosen ECA rules since they provide an abstraction that is powerful and easy to understand. Other workflow management systems and research prototypes also use this paradigm [26].

The key idea of the synchronization component is the separation of concerns. This component reifies procedure synchronization into specialized objects. In effect, it separates the “how” from the “when”: procedures deal merely with workflow enactment, while preconditions handle synchronization. Svend Frølund adopts a similar solution for coordinating distributed objects [38]. His PhD thesis proposes synchronizers (to synchronize components) and synchronization constraints (for per-object invocation constraints). Like

my solution based on preconditions and precondition manager, synchronizers are distinct entities. However, they enforce synchronization on *groups of components*, and support *incremental modification* through subclassing. Programmers describe synchronization constraints through message patterns. But micro-workflow lets developers customize its components according to their needs. Should they require these features for procedure synchronization, the architecture allows them to change the synchronization component to use synchronizers and synchronization constraints.

The solution described in this section evaluates the queued preconditions continuously, checking whether they are satisfied. This works well when the number of preconditions is small, and when their evaluation is not computationally-intensive. But it won't scale. When scalability becomes a problem, developers should choose a different solution. For example, they could change the synchronization component to use a more sophisticated approach, e.g., one that performs incremental updates [88]. This reduces the amount of computation at the cost of additional complexity.

The ability to integrate the architecture (in this case, the synchronization component) with other subsystems and customize it for particular problems represents one of the characteristics that set micro-workflow apart from current workflow architectures.

4.3 Process Component

The strep throat treatment consists of several activities. It begins with the doctor testing whether the patient has the disease or not. If the results are positive, the doctor prescribes a treatment. This depends on the patient's medical history. For example, the doctor prescribes penicillin only for patients who are not allergic to antibiotics. Next a nurse makes sure that the patient understands how to follow the treatment. Two days after the beginning of the treatment, the nurse follows up with the patient to check whether his condition is improving.

Therefore, the strep throat procedure is a sequence of activities. Several processing entities (for this example, humans) perform some of these activities, e.g., the doctor or the nurse. Other activities involve decisions: "Does John have strep throat?" or "Is John allergic to penicillin?" If the lab tests are negative, the doctor orders additional tests for diseases other than strep throat but with similar symptoms. The lab technician iterates through these tests and reports back the results.

A Procedure class that handles all these different situations would be too complex and against good soft-

ware engineering practices [108]. Instead, the framework employs several subclasses that together provide a gamut of procedure types. Sequence implements sequential activities. Primitive enables domain objects (e.g., doctor, nurse) to perform application-specific work outside the workflow domain. Conditional and Repetition provide a means to alter the control flow. Iterative works on composite objects. Finally, Fork and Join spawn and synchronize multiple threads of control in the workflow domain.

Let's see how Procedure subclasses implement the control structures.

4.3.1 Sequence

The micro-workflow framework represents the nodes of the activity map corresponding to the process definition with a set of procedures. But the Procedure class discussed in Section 4.1 doesn't offer provisions to specify sequences of activities in the workflow definition. Developers need a way to aggregate successive procedures.

SequenceProcedure is a Procedure subclass that has a number of steps, each of which is another procedure. At run time it executes all its steps sequentially. Software developers use SequenceProcedure to specify a temporal ordering between other procedures.

Usage

SequenceProcedure is a subclass of Procedure that represents a *Composite* [39], with other procedure instances as its components. The Composite pattern requires that composite objects and their components have the same interface. This property enables developers to represent the process activity map as a tree of procedure objects, as illustrated on the right side of Figure 4.2(b).

Developers (or software applications) build the sequence by adding steps in the order in which they should execute. Notice, however, that this can happen even at run time, thus allowing the framework to execute workflows that dynamically complete their own definition.

Design Details

The Procedure base class defines the run time interface and implements the ProcedureActivation instantiation mechanism described in Section 4.1. The common interface enables the framework to execute all procedures through the same protocol, regardless of their type.

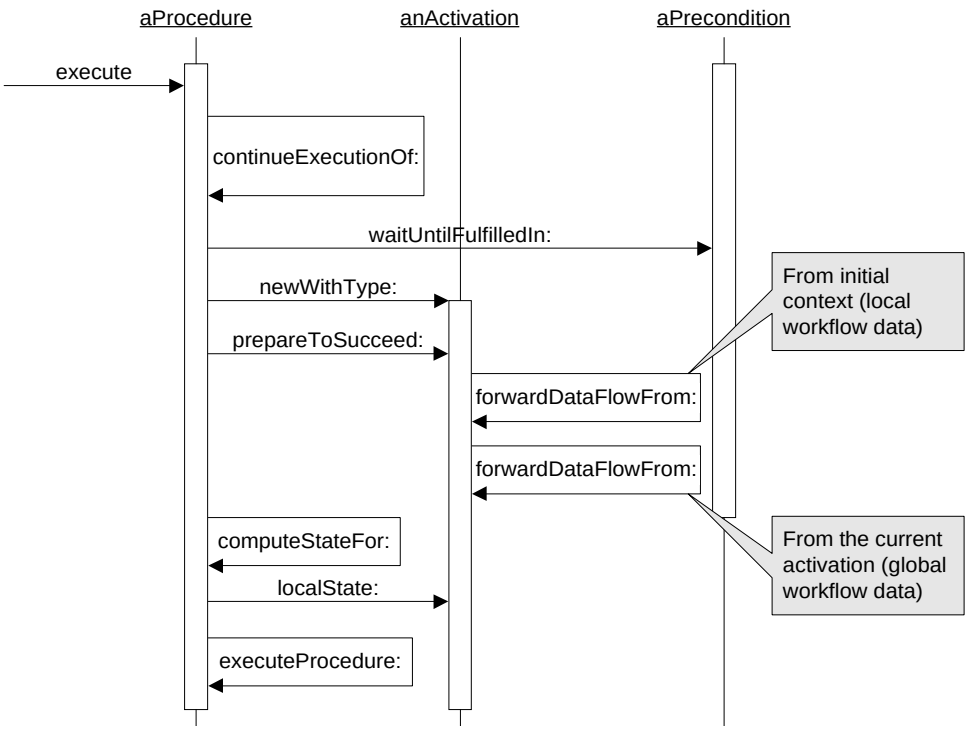


Figure 4.4: Procedure execution sequence diagram (simplified).

There are two ways to trigger the execution of a procedure object. The execute message allows clients from the *application domain* to fire off a procedure. Typically they send this message to the root node of the activity map representing the process definition. The second entry point continueExecutionOf: serves the *workflow domain*. Composite procedures send this message to execute their components.

Figure 4.4 shows how a Procedure instance responds to the execute message. The procedure sends continueExecutionOf: and the control reaches the internal entry point. Next the procedure checks its Precondition by sending the waitUntilFulfilledIn: message with the current activation as the argument. In effect, this message transfers control to the synchronization component described in Section 4.2.

The waitUntilFulfilledIn: message returns when the precondition manager determines that the precondition associated with the procedure is fulfilled. Next the procedure creates a new instance of ProcedureActivation. Then it transfers control to the new activation by sending the prepareToSucceed: message.

On the workflow instance side, the activation handles the data flow. First it initializes the local variables from the initial context of its type. The first forwardDataFlowFrom: message moves data from the procedure initial context to the activation. Then the new activation extends its context with the contents of the current

activation. Finally, it returns control to its Procedure object, on the workflow type side.

The computeStateFor: message returns additional state information required to execute the procedure. But the procedure instance can't store workflow runtime information in its instance variables. Therefore, it stores the state information in the activation, which holds the workflow instance data. At this point the procedure has all the runtime information and sends the executeProcedure: message to complete execution.

However, Procedure is an abstract class and doesn't implement computeStateFor: and executeProcedure:. Execution within the Procedure class ends here and each of its concrete subclasses implements these messages in its own way. Thus inheritance allows all procedure types to share the execution mechanism illustrated in Figure 4.4, while polymorphism enables them to augment this mechanism with the behavior specific to each type.

Let's see how SequenceProcedure responds to these messages. Procedures send computeStateFor: to obtain the run time information specific to their type. For SequenceProcedure the type side provides the information about its steps. computeStateFor: builds and returns a stream[3] with its steps. executeProcedure: iterates through the steps and executes each of them by sending the continueExecutionOf: message. SequenceProcedure carries the results of each step to the next one. This mechanism advances the workflow runtime data from one step of the process to the next. Once execution completes, SequenceProcedure returns to its caller the activation corresponding to the last step.

4.3.2 Procedure with Subject

Micro-workflow involves application objects that encapsulate domain-specific processing (task logic). The framework complements this domain-specific functionality with workflow objects that encapsulate process logic. This separation enables one to think of workflow as the glue that interconnects application objects.

Process enactment takes place within the workflow domain. To perform actions in the application domain, the framework must be able to transfer control and data not only within the workflow domain, but also across the domain boundary. ProcedureWithSubject provides the mechanism that implements this functionality.

[3]The reason why I use a stream instead of a collection will become evident in Section 5.4.

Usage

ProcedureWithSubject is an abstract class that extends Procedure to delegate processing to an object within the application domain. Delegation requires at least the application object (i.e., who?) and a selector (i.e., what?). These are sufficient to transfer control flow outside the workflow domain. Additionally, optional arguments enable data flow from the framework to the application domain.

Subclasses of ProcedureWithSubject specify how instances of this class transfers control to a domain object through two messages. sends:to: works for situations that don't require data flow from the workflow domain to the application domain. The symbol arguments of this selector specify the message to be sent to the domain object and the slot name where the domain object resides within the context. sends:with:to: adds the possibility of passing data to the application domain. An array of symbols passed as the second argument provides the slot names corresponding to the objects passed as arguments.

Design Details

Subclasses send the executeDomainOperationIn: message to request an action on the subject. ProcedureWithSubject looks up the application object and any additional data for the application domain in the workflow context. Then it delegates the requested operation to the subject, transferring control across the domain boundary. executeDomainOperationIn: returns the value passed back from the application domain.

4.3.3 Primitive

PrimitiveProcedure is an abstraction of a piece of work performed by an application object. Instances of this class perform application specific actions and add the possibility of data flow from the application domain into the workflow domain.

Usage

Primitives enable the framework to pull application-specific information into the workflow runtime. In addition to the information required to pass control across the domain boundary and described in Section 4.3.2, primitive procedures also require the slot name for the result. At run time it places the object returned from the application object into this slot.

Developers create PrimitiveProcedure instances by sending sends:to:result: or sends:with:to:result: messages to this class. These selectors add an additional argument for the result slot to the ones described in Section 4.3.2. Instances of the PrimitiveProcedure class can only be leaf nodes in the tree representation of a process.

Design Details

PrimitiveProcedure is a concrete subclass of ProcedureWithSubject, which provides the mechanism that transfers control across the domain boundary. Primitive uses it to execute operations on domain objects that contribute to the outcome of the process. A primitive procedure can also *pull* application-specific information into the workflow runtime.

Primitive procedures implement computeStateFor: but don't have any state information associated with their type side. The procedure responds to executeProcedure: by delegating the execution of the domain operation to its superclass. If the developer provides a slot name for the result, the primitive adds the object returned by executeDomainOperationIn: to its context. executeProcedure: returns the current ProcedureActivation to its caller.

Figure 4.5 shows how a primitive procedure responds to the executeProcedure: message and passes the control flow across the domain boundary.

4.3.4 Procedure with Guard

SequenceProcedure and PrimitiveProcedure let developers implement and run processes. However, control flow is limited to sequential execution. The framework needs constructs that alter the process control flow.

Different control structures change the flow of control in different ways. They determine how the control proceeds based on the evaluation of a guard condition. ProcedureWithGuard implements a guard evaluation mechanism. It is also a composite procedure with a single component (i.e., the body). Subclasses provide the logic that relates the evaluation of the guard clause to the execution of its body.

Usage

ProcedureWithGuard is an abstract subclass of ProcedureWithSubject. Subclasses have to provide the guard clause and the body procedure. The guard clause can use information from the application domain or

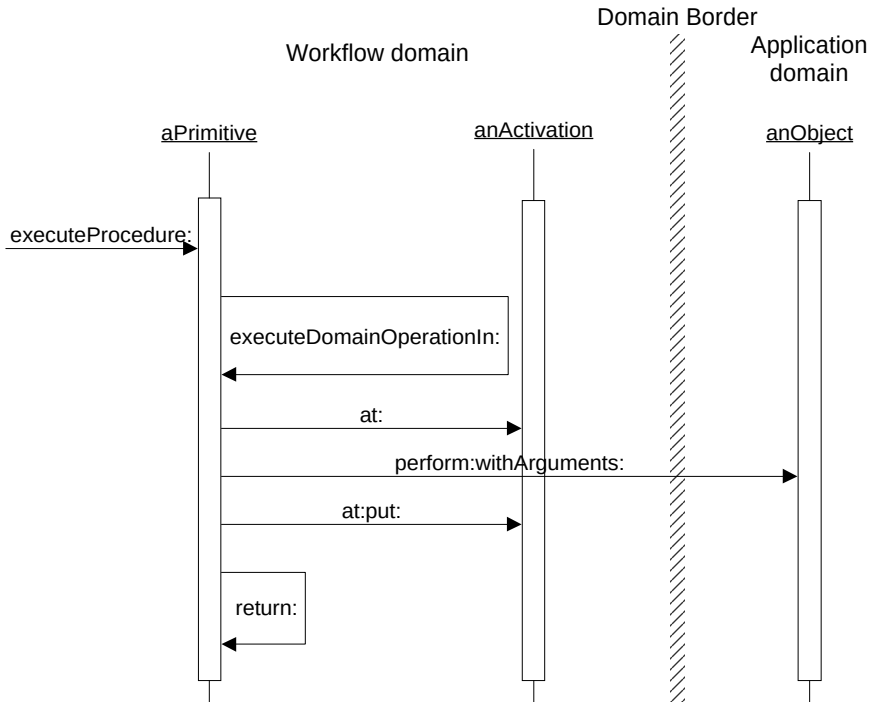


Figure 4.5: UML sequence diagram for PrimitiveProcedure.

workflow runtime data. The body: message takes a procedure instance argument and sets the procedure body.

Design Details

Developers can implement the guard clause with function objects. Function objects provide a language-independent solution and don't restrict the number of arguments. However, for my framework I chose a simplified solution which represents the guards as single-argument Smalltalk blocks.

The guard clause can obtain its arguments from the workflow runtime or from a domain object. The first possibility corresponds to a context lookup. My solution based on block closures requires a slot name. Developers specify the name of the slot through the argumentSlot: message. The second possibility uses the mechanism provided by ProcedureWithSubject and requires the information described in Section 4.3.2.

4.3.5 Conditional

ConditionalProcedure corresponds to Dijkstra's guarded command [25]. It enables developers to alter the control flow in the workflow domain. At run time, conditionals determine how the framework navigates the

activity map representing the process definition.

ConditionalProcedure is a concrete subclass of ProcedureWithGuard. The conditional implements a control structure of the type **IF** `condition` **THEN** `body`. It evaluates the guard and executes the body procedure only if the condition is satisfied.

Usage

Creating ConditionalProcedure instances involves supplying the information required by the ProcedureWithGuard superclass: the guard clause, the body procedure, and the slot name/message send data required to obtain the guard's argument. Developers send if:for:execute:, send:to:if:execute:, or send:with:to:if:execute: to the ConditionalProcedure class to create instances. Since instances of this class are composites, they can't be leaf nodes in the activity map representing the process definition.

Software developers can implement other types of conditional constructs found in programming languages with combinations of ConditionalProcedure instances. For example, an **IF THEN ELSE** construct requires two instances and the negated condition:

IF `condition` **THEN** $\texttt{body}_1$

IF `NOT condition` **THEN** $\texttt{body}_2$

Likewise, a **CASE** statement requires one ConditionalProcedure instance for each branch. For example, a 4-way **CASE** statement uses 4 ConditionalProcedure instances:

IF $\texttt{condition}_1$ **THEN** $\texttt{body}_1$

IF $\texttt{condition}_2$ **THEN** $\texttt{body}_2$

IF $\texttt{condition}_3$ **THEN** $\texttt{body}_3$

IF $\texttt{condition}_4$ **THEN** $\texttt{body}_4$

Design Details

Conditionals provide logic that connects the result of the guard clause and the execution of the body procedure into an `IF-THEN` control structure.

ConditionalProcedure implements the two messages required by the execution component (Section 4.1). computeStateFor: evaluates the guard block and returns its value. The ProcedureWithGuard superclass described in Section 4.3.4 provides this mechanism. executeProcedure: implements the logic that executes

the body procedure only when the guard block returns true. This procedure type returns to its parent the ProcedureActivation object returned by its body.

4.3.6 Repetition

Dijsktra used guarded commands and while loops to build all types of control structures. RepetitionProcedure subclasses ProcedureWithGuard and implements a **`DO`** `body` **`UNTIL`** `condition` control structure. It keeps executing the body procedure until the condition is satisfied.

Usage

Developers create RepetitionProcedure instances by sending the repeat:until:for: message. This message takes the body procedure as the first argument, the guard block as the second, and the symbol used to resolve the block's argument as the third. Since the repetition is also a composite, it can't be a leaf node in an activity map.

Design Details

RepetitionProcedure doesn't have state information on the type side. Therefore, computeStateFor: doesn't involve any computation. The executeProcedure: message provides the logic implementing the `DO-UNTIL` control structure that connects the guard condition to the body procedure. RepetitionProcedure returns to its parent the activation returned by its body.

4.3.7 Iterative

Many business processes involve composite domain objects. Sometimes a process needs to operate on each individual component of a composite object. For example, in the telecommunications billing process from Section 3.3.2 the account represents the composite and the agreements represent its components. At the end of the billing cycle, the telecommunications provider computes the total balance by adding the subtotals corresponding to each of these agreements.

Developers could use the control structures described in the previous sections to obtain all the components of a composite domain object and then execute some procedure on each of them. But since this

situation occurs so often in business applications, I decided to introduce a specialized structure. IterativeProcedure provides a means for operating on the individual components of composite objects. In effect, this demonstrates that developers can extend the micro-workflow process component with custom control structures.

Usage

IterativeProcedure is a composite with one component. It subclasses ProcedureWithSubject and transfers control to the application domain for side effects, to obtain the subject's components. At execution time it iterates through these components, places each component in a context slot and executes its body procedure. Developers send the subjectForBody: message to specify the name of the slot where the procedure puts each individual subject. Since this class is a composite, its instances can't be leaf nodes in the tree representation of a process.

Design Details

The run time information specific to an iterative procedure consists of the subject's components. Therefore, computeStateFor: builds and returns a stream containing the composite domain object components. This involves sending the subjectsIn: message to obtain them. Through the executeDomainOperationIn: mechanism, subjectsIn: transfers control to the application domain and returns a stream with the components of the domain object. executeProcedure: iterates through the subject's components. For each iteration, it places the corresponding component in the context and fires off its body procedure, carrying over the results of previous iterations. Iterative returns to its caller the activation corresponding to the last iteration.

4.3.8 Fork

The procedure types described so far allow software developers to implement processes that have a single thread of control within the workflow domain. But business processes often speed up processing by executing two or several parts of a process in parallel. For example, a travel reservation process handles the booking of flights, cars, and hotels concurrently. Therefore, the framework needs a means to support concurrent activities.

The Fork procedure spawns multiple threads of control within the workflow domain. In effect, it enables the micro-workflow framework to run concurrent procedures.

Usage

Just like Sequence, the Fork procedure is a composite. At run time it delegates processing to its components. Unlike the Sequence procedure which executes its steps sequentially, Fork fires off its branches in parallel. In effect, each branch executes in a separate thread.

Once a branch procedure completes execution, it needs to pass the result (an activation) back to the Fork procedure. But since each branch runs independently of the others, other branches that finished executing may return their result at the same time. To eliminate conflicts between concurrent procedures that return their results, the access to the return point should be controlled.

Design Details

The Fork procedure uses a Smalltalk SharedQueue to avoid concurrency conflicts between its branches. SharedQueue provides a thread-safe collection. The Fork procedure uses this collection for two purposes. First, the shared queue ensures that writes from concurrent threads don't conflict with each other. Second, it provides a synchronization point between the thread that reads it (the main thread) and the threads that write to it (the threads corresponding to each branch).

The execution component sends the computeStateFor: message right before it executes a procedure. In response to this message, the Fork procedure creates and returns a shared queue to the instance side. Next, executeProcedure: fires off each branch in a separate thread. However, instead of sending the continueExecutionOf: message discussed in Section 4.3, Fork uses executeSubProcessWith:. This message ensures that each branch returns its results to the Fork and stop unwinding the stack. Once all branches start executing, the Fork procedure returns the first activation available in the shared queue. If the queue contains no results (e.g., none of the branches has finished execution), the queue suspends the process corresponding to the main thread of control. As soon as one of the branches updates the queue with its activation, the main thread resumes and transfers control back to its caller. Figure 4.6 shows how Fork implements the executeProcedure: message.

The Fork procedure returns the first activation that becomes available. This corresponds to the dis-

```
executeProcedure: anActivation
    | stack sharedQueue |
    sharedQueue := anActivation localState.
    join sharedQueue: sharedQueue withLength: branches size.
    stack := anActivation workflowSession workflowStack.
    branches do:
            [:each |
            self
                executeBranch: each
                with: anActivation
                restoredStack: stack].
    ^self return: sharedQueue next

executeBranch: aProcedure with: anActivation restoredStack: aStack
    [| currentActivation activation sharedQueue|
    sharedQueue := anActivation localState.
    currentActivation := anActivation deepCopy.
    currentActivation workflowSession initializeStack.
    activation := aProcedure executeSubProcessWith: currentActivation.
    activation workflowSession workflowStack: aStack.
    sharedQueue nextPut: activation]
            fork
```

Figure 4.6: The Fork procedure.

junction of the activations returned by all branches. Therefore, this procedure type enables developers to implement only processes that require the output of the first completed branch. For example, the telecommunications provisioning process described by Georgakopoulos and colleagues [43], Jackson [63], and Georgakopoulos and Tsalgatidou [45] and sketched in Figure 4.7 requires an OR-join to select one out of three circuit provisioning activities.

Many activity-based process models provide fork and join constructs [111, 16]. Different workflow systems implement them in different ways. For example, in the meta model described by Leymann and Roller [72] disjunctive joins wait until all the branches complete. The authors justify this decision by providing an example which otherwise would yield a race condition—Figure 4.8. Although they don't provide details about their implementation, a stateless solution (e.g., RPC-based) would have this problem. However, waiting for all branches introduces an additional concern. Now the workflow system must discard any branches that don't execute (e.g., conditionals) and therefore it should not wait for. Leymann and Roller

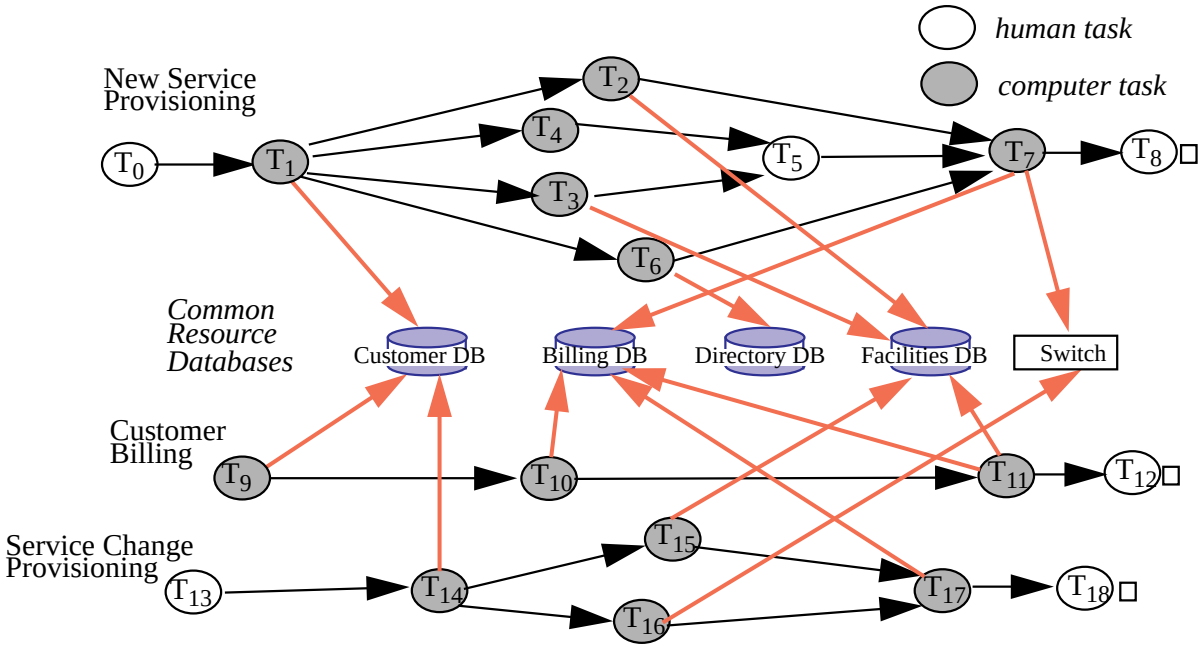


Figure 4.7: Telecommunications provisioning process. Provisioning the new service requires an OR-join (T_7) between the branches containing T_3, T_5, and T_6—diagram from Georgakopoulos and Tsalgatidou [45].

solve this problem with a dead path elimination algorithm. But computing dead paths further increases the complexity of an already complex system.

Micro-workflow represents activities with objects. These objects hold local state and therefore don't have the above problem. Consequently, the procedure can (and does) return the first activation as soon as it becomes available. Subsequent activations from the other branches don't re-trigger the procedures following the join point.

4.3.9 Join

Fork spawns multiple threads of control in the workflow domain to run concurrent branches (procedures). It passes control back to its caller as soon as one branch completes, returning the corresponding activation. However, the process may need the results of all branches of the fork procedure.

For example, Dinkhoff and colleagues [26] discuss an administrative process that creates leases for apartments. The process (depicted in Figure 4.9) begins with the creation of several documents. Three parallel activities gather the data and assemble the final documents. However, the lease process can't continue until all three documents are ready. The authors synchronize the branches with a conjunctive join node.

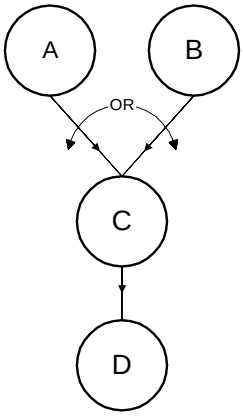


Figure 4.8: Stateless OR-join that can cause a race condition. Without dead path elimination, the activity C fires twice, once when A completes, and once when B completes.

This node ensures that the workflow system doesn't start the next activity unless the three documents are assembled.

The Join procedure provides a synchronization point for all the branches of a Fork procedure. In effect, it implements an AND-join activity type.

Usage

When software developers create a Join procedure they need to specify the matching Fork procedure. This supplies the activations returned by each branch.

Design Details

In response to the executeProcedure: message, the Join procedure obtains all the activations returned by the Fork's branches from the shared queue. Once all branches have finished, the Join procedure returns to its caller an activation whose context corresponds to the union of the activations contexts. Figure 4.10 shows how the Join procedure responds to the executeProcedure: message.

4.3.10 Discussion of the Process Component

The Sequence, Primitive, Conditional, Repetition, Iterative, Fork and Join procedures provided by the framework represent only a basic set of control structures. This set enables software developers to build quite a wide range of workflows.

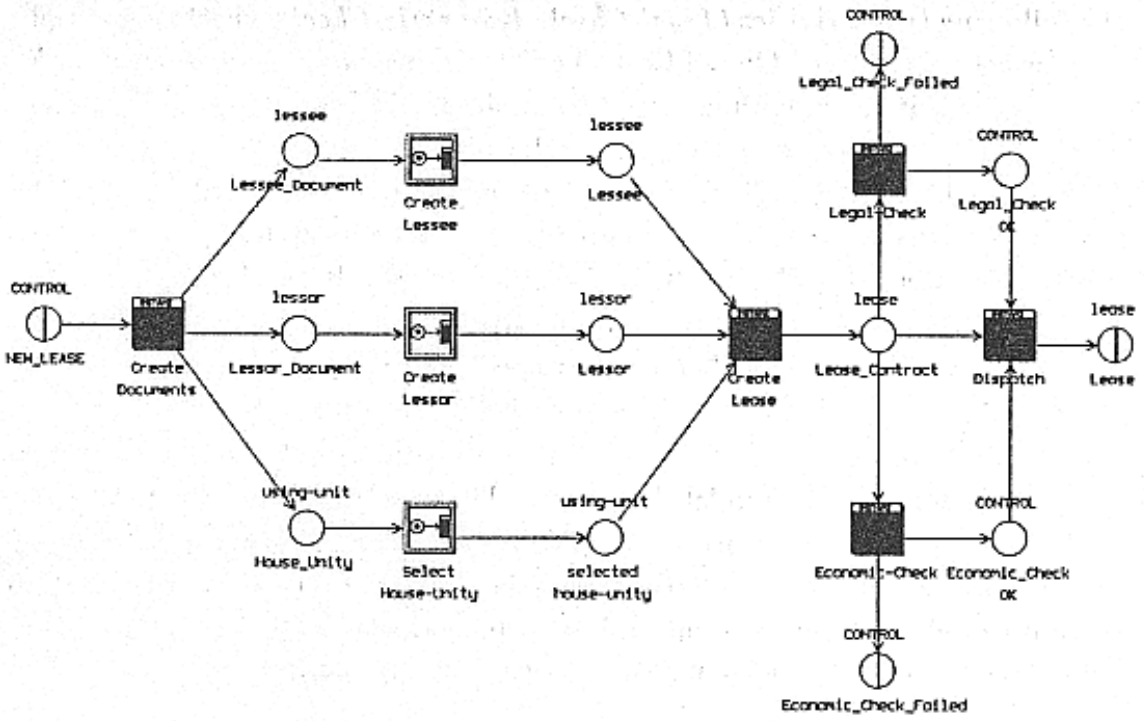


Figure 4.9: Apartment leasing process. The “Create Lease” node performs a conjunctive join of the incoming branches—diagram from Dinkhoff and colleagues [26].

```
executeProcedure: firstActivation
    | otherActivations result |
    result := firstActivation copy.
    otherActivations := OrderedCollection new.
    sharedQueue isNil
        ifTrue:
            [Dialog
                error: 'A ForkProcedure should have initialized this variable for me'].
    [numberOfBranches - 1 timesRepeat: [otherActivations add: sharedQueue next]]
        ensure: [sharedQueue := nil].
    otherActivations do: [:each | result forwardDataFlowFrom: each].
    ^self return: result
```

Figure 4.10: The Join procedure.

Douglas Bogia's PhD thesis identifies "openness" (the ability to add, modify and specialize task descriptions) as an important technical concern for computer-supported tasks [12]. Although Bogia's work focuses on collaborative processes that involve only humans, current workflow research agrees with his conclusion [15]:

> Usually, control-flow constructs (such as sequence, parallel execution, loops, and conditional branching) are sufficient to define workflows. However, in many cases these constructs prevent appropriate control-flow definition—for example, three subworkflows can be executed in any order, but only one at a time, using the above constructs. To accomplish more complex constructs, WFMSs need to allow for their definition.

The micro-workflow framework offers software developers a basic set of control structures. They can customize the existing control structures, as well as add new control structures tailored for their applications. In contrast, closed workflow systems like most commercial products limit their users to the control structures imagined and implemented by their designers.

Chapter 5

Advanced Workflow Features Through Composition

The components of the micro-workflow core discussed in Chapter 4 enable software developers to define and execute workflows. One of the key features that sets the micro-workflow architecture apart from traditional workflow architectures is that the former allows developers to add advanced workflow features by adding components.

In this chapter I show how to build and add this type of components to the micro-workflow core. I do so in order to: (i) Prove that the micro-workflow architecture can be extended through composition with features typical of workflow systems. (ii) Teach software developers how to extend the architecture—they will take similar steps to build components providing other workflow features. (iii) Demonstrate that extending the micro-workflow core with components yields a feature-rich workflow architecture.

The following sections extend the micro-workflow core with six components: a *history component* extracts workflow runtime information for logging purposes; a *monitoring component* provides information about the running workflows; a *persistence component* records workflow runtime data to a database; a *worklist component* provides support for human workers; a *manual intervention component* allows software developers to take over the sequencing of activities at run time; and a *federated workflow component* adds support for transparent workflow execution across the enterprise. These components provide a variety of features with different requirements. Some of these (e.g., history, monitoring, persistence, and worklists) features are well-established in current workflow systems. Others (e.g., manual intervention and federated workflow) are still new and have been the focus of intense research efforts during the last few years. I am confident that following the examples provided in this chapter software developers can extend the core with

new features.

5.1 History

Chapter 2 identifies history as one of the characteristics that sets workflow apart from other systems. Workflow management systems log the execution history of the workflows they execute. According to Leymann and Roller [72], a workflow management system "must provide the capability to record *process traces*." In fact, they identify history as one of the operational requirements of workflow management.

There are two reasons to collect history information. First, *workflow users* may use it after the workflow completes execution. For example, process designers use it to evaluate, improve, and even derive new process models; auditors use it for auditing purposes; etc. Second, the *workflow system* may also use the history information for recovery purposes.

Although most commercial workflow products log process execution, they provide limited access to the history mechanism. Typically their users have little control over *what* information the system records, and no control over *how* and *where* the system stores this information. This approach works well for non-technical people. End users don't care how the history mechanism works and are not interested in changing it. However, developers who use workflow to implement processes within applications want to be able to tailor the history mechanism, and customize it for particular problems. Therefore, a workflow architecture targeting software developers should provide fine grained access to and control of the history mechanism.

The components of the micro-workflow framework described in Chapter 4 don't log process execution. This section extends the framework with a separate component that provides this functionality. Therefore, unlike most workflow architectures, history is not an integral part of the system. This design lets developers add the history component *only when they need its functionality* in their applications.

5.1.1 Usage

The micro-workflow history component allows software developers to tailor the logging of workflow events by plugging in different history mechanisms. This should remain transparent for the other framework components.

Developers choose the history mechanism suitable for their application from a repertoire of logging strategies and plug it into the workflow session. Different strategies allow them to specify what type of

information they want logged, as well how and where the strategy should log this information. The execution component uses the logging strategy (i.e., the history component) to log activations. This design separates workflow enactment from the history mechanism. The UML instance diagram from Figure 5.1 sketches how plugging in the history component amounts to adding a logging strategy to the workflow session.

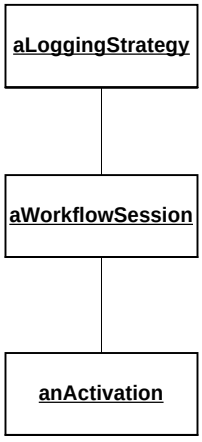


Figure 5.1: Logging strategy, instance diagram.

To demonstrate the versatility of this approach I have built logging mechanisms suitable for three different situations: no logging (discard the workflow events), memory logging, and persistent logging. Reusing one of these mechanisms illustrates how micro-workflow provides software developers with several choices. The real power of this approach is that they can easily add new logging mechanisms.

5.1.2 Design Details

Following the *Strategy* [39] pattern, the abstract class LoggingStrategy defines the interface of its concrete subclasses. The execution component interacts with the history component through the addWorkflowEvent: and the allHistory messages. addWorkflowEvent: logs the activation supplied as its argument. Concrete strategies implement this message to provide different logging mechanisms. allHistory provides access to the logged workflow events. A logging strategy object responds to this message by returning the logged workflow events. Likewise, the manual intervention component interacts with the history component through the prepareToRewindTo: and previousActivation messages. These messages support workflow backward recovery. Section 5.4 discusses this feature.

ProcedureActivation instances delegate history requests to the workflow session. This object holds the

strategy that fulfills the logging requests from all the activations executing within the workflow session. Therefore, activations and procedures don't deal directly with the historic data.

An ensemble of objects determine the state and the behavior of every activation. On the instance side the activation context holds the runtime data. On the type side the Procedure provides the rules that determine how the activation executes. Additionally, an instance of WorkflowSession (see Figure 5.1) holds: the PreconditionManager of the synchronization component (described Section 4.2); the LoggingStrategy of the history component; and a workflow stack (described later in this chapter). But some of these objects (for example, the precondition manager and the logging strategy) are transient and therefore the history mechanism should not log them. When the framework restores activations from the history (for example, while backtracking), it initializes the manager and the logging strategy from the running workflow session.

The ProcedureActivation class provides a mechanism that strips off the runtime-specific data. Each activation responds to the stripRuntimeData message by discarding the information that shouldn't be logged. stripRuntimeData sends the passivate message to its procedure. On the type side, the WorkflowSession object uses the *State* [39] pattern to accommodate workflow sessions for running or stored activations, respectively. A session for running activations has additional instance variables that hold the precondition manager and the logging strategy. In contrast, a session for stored activations doesn't have this information. Instead, it provides a means to obtain it from the runtime. Therefore, Procedure instances respond to the passivate message by changing the state of the workflow session from "active" to "passive." The instance diagrams from Figure 5.2 illustrate an activation with an active (a) and passive (b) workflow session.

The framework provides several concrete subclasses that implement different logging mechanisms: NullLogging, MemoryLogging, and GemStoneLogging. The UML class diagram from Figure 5.3 shows the structure of the history component.

Null Logging

This strategy implements the *Null Object* pattern [136] and discards the workflow events. This is the strategy for workflows that don't require historic information.

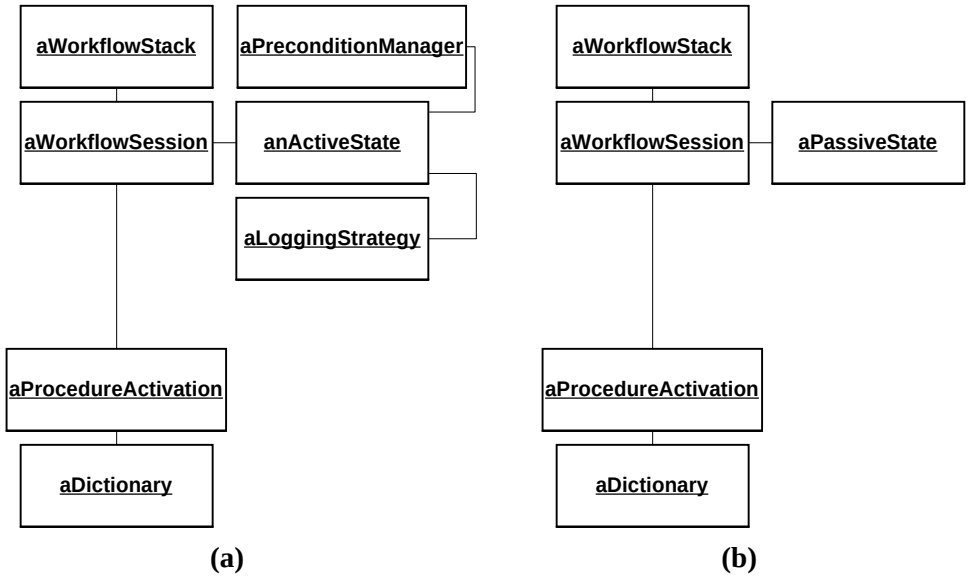


Figure 5.2: Instance diagram showing the difference between an active and a passive activation.

Memory Logging

MemoryLogging logs the workflow events into a Smalltalk OrderedCollection. This collection maintains the temporal ordering of the events. The workflow history is monotone increasing—it grows but never shrinks. As the process unfolds and the execution component fires off procedures, the collection will grow to accommodate them. Therefore, since the MemoryLogging strategy logs the workflow events in the Smalltalk image, the process history is available throughout its lifetime. Additionally, the amount of memory available to the Smalltalk image limits the number of workflow events this strategy can log.

Software developers access the workflow history through the addWorkflowEvent: and allHistory messages. addWorkflowEvent: adds the activation supplied as an argument to the workflow history. Likewise, allHistory returns the sequence of logged activations. Additionally, logging strategies support backtracking through the logged activations with two messages. The framework signals the intention to backtrack by sending the prepareToRewindTo: message. The strategy initializes a read stream with the logged workflow events, from the current activation (usually the sender of the prepareToRewindTo: message) until the activation supplied as an argument. Typically this is a subset of the entire process history. Once the initialization completes, the previousActivation message provides access to the stream's contents. The framework sends this message to access the logged workflow events, one by one. Figure 5.4 shows the implementation of the MemoryLogging history mechanism.

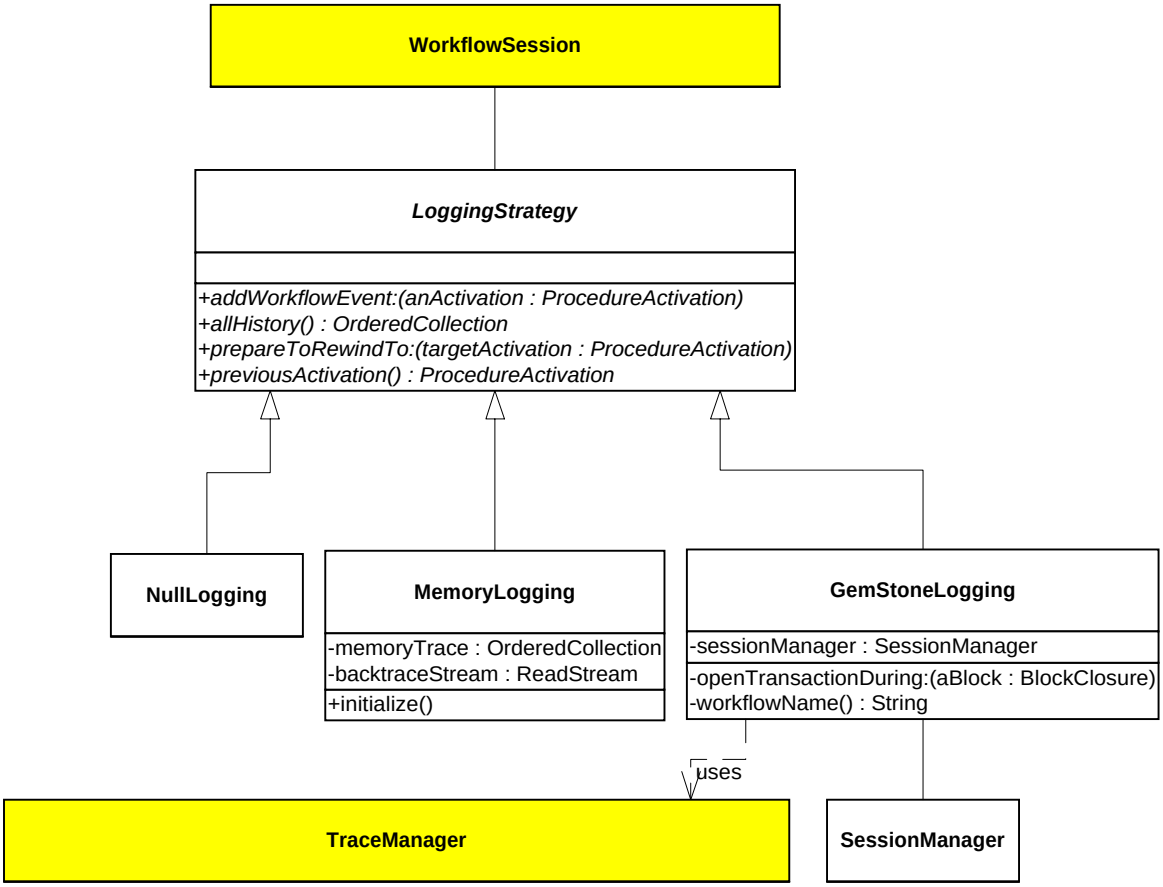


Figure 5.3: Micro-workflow history component, UML class diagram. The colored/shaded classes belong to different components.

```
LoggingStrategy subclass: #MemoryLogging
    instanceVariableNames: 'memoryTrace backtraceStream '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Workflow-History'

MemoryLogging>>initialize
    memoryTrace := OrderedCollection new

MemoryLogging>>addWorkflowEvent: aStrippedActivation
    memoryTrace add: aStrippedActivation

MemoryLogging>>allHistory
    ^memoryTrace copy

MemoryLogging>>prepareToRewindTo: anActivation
    backtraceStream := ReadStream
                on: (memoryTrace copyFrom: (memoryTrace indexOf: anActivation)
                        to: memoryTrace size) reverse

MemoryLogging>>previousActivation
    ^backtraceStream atEnd ifTrue: [nil] ifFalse: [backtraceStream next]
```

Figure 5.4: The MemoryLogging logging strategy.

GemStone Logging

GemStoneLogging uses the GemStone/S object-oriented database [42] to record activations. In this case the process history grows outside the boundaries of the Smalltalk image. Additionally, the logged workflow events exist beyond the lifespan of the workflow. GemStone requires connections between the objects that should be saved (called root objects) and its repository. Once connected, the root objects and the objects reachable from them (i.e., the transitive closure) can be saved to the persistent store. GemStone makes this process transparent.

The GemStoneLogging strategy interacts directly with the TraceManager class of the persistence component described in Section 5.2. This class provides a single point of access to the persistence component. Therefore, concurrent workflows employing the GemStoneLogging strategy share the same TraceManager instance. However, uncontrolled access to this shared access point may cause inconsistencies in the database. To avoid conflicts between concurrent accesses, this strategy should use a transaction mechanism.

There are three messages in the LoggingStrategy that update the workflow history—for this strategy, history updates correspond to writes to the database. They require the GemStoneLogging strategy to write the changes to the persistent store. But updates from different transactions can cause write/write conflicts between concurrent updates [31]. Consequently, addWorkflowEvent:, prepareToRewindTo: and previousActivation maintain a consistent view of the repository by propagating the updates within GemStone transactions. In contrast, the allHistory message performs a read-only access and operates outside a transaction.

Figure 5.5 shows the implementation of the GemStoneLogging history mechanism. The strategy implements the interface defined by its super-class by delegation to the TraceManager instance of the persistence component. This manager has an interface similar to the LoggingStrategy. However, since it provides the single point of access for all workflow sessions, it needs a means to discriminate between accesses from different workflows. Therefore, logging workflow events and backtracking require an additional argument that uniquely identifies the workflow instance. Additionally, GemStoneLogging uses the openTransactionDuring: message to wrap the three operations that change the repository within a GemStone transaction. This message implements the *Execute Around Method* pattern [8].

5.1.3 Discussion of the History Component

History is a distinguishing characteristic of workflow management systems. Monolithic workflow architectures don't provide access to the history mechanism. Their users have no choice but to use the system as a whole. When they don't require history, they *can't take it out* and therefore this feature only increases the footprint of the system. Likewise, when they require history, they *can't customize* the mechanism provided by the system designers. Several research projects attempt to address this problem. Section 2.5.1 has described how TriGS$_{flow}$ implements history with ECA rules [111]. Mentor-lite implements history as a workflow on top of a lightweight kernel [86]. Both these systems allow their users to customize the history mechanism. Micro-workflow takes workflow history one step further. The component described in this section relies on object technology and implements history as a pluggable component. This approach allows developers to add this feature to the core only when they need it, as well as customize it through techniques specific to object systems.

The micro-workflow history component provides full access to the mechanism that logs workflow activations. Software developers can change *what* type of information is recorded in the workflow history. For

```
LoggingStrategy subclass: #GemStoneLogging
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Workflow-History'

GemStoneLogging>>addWorkflowEvent: anActivation
    self openTransactionDuring:
        [TraceManager instance add: anActivation for: self workflowName]

GemStoneLogging>>allHistory
    ^TraceManager instance allHistoryFor: self workflowName

GemStoneLogging>>prepareToRewindTo: anActivation
    self openTransactionDuring:
        [TraceManager instance prepareToRewindTo: anActivation for: self workflowName]

GemStoneLogging>>previousActivation
    ^self openTransactionDuring: [TraceManager instance previousActivation]

GemStoneLogging>>openTransactionDuring: aBlock
    | session |
    session := GBSM currentSession.
    session beginTransaction.
    [aBlock value] ensure: [session commitTransaction]

GemStoneLogging>>workflowName
    ^GBSM currentSession parameters workflowName
```

Figure 5.5: The GemStoneLogging logging strategy.

example, Leymann and Roller [72] suggest recording the time it takes to execute each activity. The history component presented here doesn't record this information, but software developers can build a new strategy to log this information. This would involve sending an additional timestamp argument to the logging strategy (e.g., addWorkflowEvent:timestamp:).

The history component also enables programmers to tailor *how* and *where* the history information is recorded without changing the other framework components. Software developers can select the option that works best for their application. The example strategies presented in this section are suitable for workflow applications with three types of history requirements: no history, transient history, and—with assistance from the persistence component—persistent history. Applications that need other ways of recording the workflow events use logging strategies tailored for their requirements.

Table 5.1 shows several potential directions for customizing the micro-workflow history component.

Aspect	**Details**
Logged information	Change what data the ProcedureActivation>>stripRuntimeData message discards
How and where the framework logs the workflow events	Build custom mechanism by subclassing LoggingStrategy

Table 5.1: Customizing the history component.

5.2 Persistence

In traditional workflow architectures, history encompasses choosing the type of information the system records about workflow events, as well as specifying *where* it stores this information. Since non-technical people use workflow systems without requiring access to these mechanisms, treating them together is a valid design decision.

Workflow management systems use a database management system (DBMS) to store the workflow history. Currently most commercial products rely on relational database technology. Older systems are tailored to one database, which sometimes is part of the package. Other systems (for example Ultimus [125]) take advantage of the existing standards for database connectivity like ODBC [119] or JDBC [134]. At least in theory, their users should be able to choose any compliant database to record the logged workflow events.

The micro-workflow architecture uses the history component discussed in Section 5.1 to extract the information to record about the workflow events. However, the history component requires additional services to save this information to a persistent store. For example, the GemStoneLogging strategy discussed in Section 5.1.2 works in conjunction with the persistence component described in this section and uses the GemStone/S object-oriented database. Therefore, unlike traditional workflow architectures, micro-workflow separates *extracting* the workflow event data from *storing it into a persistent store*. This design enables object-oriented developers to customize each of these components independently. Additionally, separating the two components allows developers to add persistence *only when the workflow application requires this functionality*. Klaus Hagen's PhD thesis subscribes to this idea. He concludes that in workflow management systems "there should be the possibility to control whether a process is made persistent or not" [49].

The persistence component provides access to a persistent store, hiding the mechanism that saves and restores objects to and from the database. This solution localizes the database-dependent part within the persistence component, thus providing *database independence*. Consequently, customizing the architecture to use a different database system involves changing only this component.

5.2.1 Storing Objects in GemStone/S

In object systems, the state of an object is determined by the values of its instance variables together with the state of all the other objects it references (i.e., the transitive closure). Therefore, saving one object to a persistent store involves saving its instance variables, and then recursively traversing the relationships and saving all the other objects reached during the traversal. Restoring an object from the database involves the opposite process. The system traverses the relationship graph, restoring each object from the persistent store.

The GemStoneLogging strategy uses the framework's persistence component to save each activation supplied by the history component into a persistent store. As Section 5.1.2 has explained, the activation distributes the runtime information among several objects with separate responsibilities. Therefore, saving an activation involves saving all these objects. The instance diagram from Figure 5.6 shows the aggregation of objects that should be saved, where the objects inside the context have been omitted.

The micro-workflow architecture aims at maintaining a consistent style that revolves around objects. Consequently I chose the GemStone/S object-oriented persistent store [42] to implement the persistence

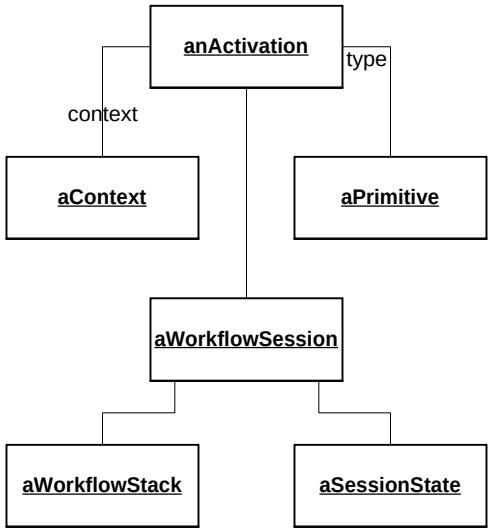


Figure 5.6: Along with the activation, the persistence component must save several other objects that hold runtime information. For simplicity, this instance diagram does not show the contents of the context.

component. GemStone handles the process of saving and restoring objects to and from the object repository transparently. Software developers only need to specify the "root object" where the save or restore should begin.

GemStone achieves persistence by attaching new objects to persistent objects. GemStone developers use this feature and organize their applications around a set of root objects. All objects that need to be saved to the repository have to be reachable from some root object. For example, Figure 5.7 shows the instance diagram with the objects that hold runtime information. In this diagram the ProcedureActivation instance represents the root object. GemStone can reach the other objects by following the context and type relationships. The association at the top of the diagram connects the activation to a persistent collection residing in the repository (shown grayed). This relationship renders the entire aggregation of objects persistent: the ProcedureActivation instance along with the other objects reachable from it—context, procedure, workflow session, workflow stack, and session state. Although for simplicity the diagram doesn't show the objects within the context, they are also reachable from the activation instance.

Once GemStone developers identify the root objects within their applications, they need to attach them to some persistent objects. GemBuilder [41], an environment for developing GemStone/S applications using VisualWorks Smalltalk, provides several types of connectors for this purpose:

- **Name connectors** link objects within the Smalltalk symbol dictionary to objects within the GemStone

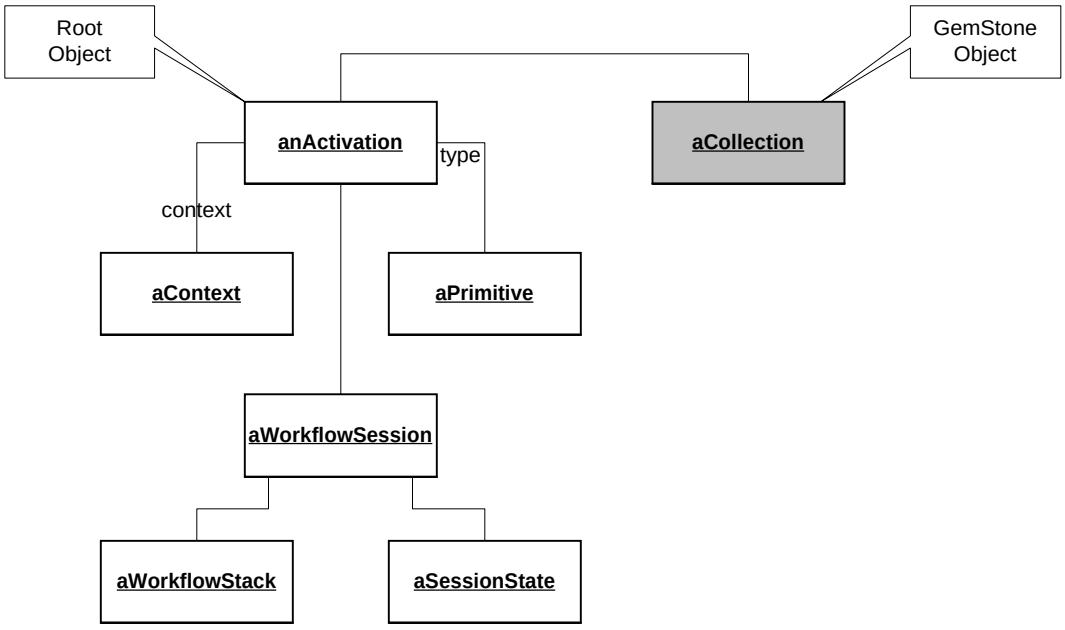


Figure 5.7: GemStone persistence amounts to connecting a client object to a persistent object. For simplicity, the objects within the context are not shown.

symbol dictionary. Since the GemStone symbol dictionary resides in the object repository, this association renders the Smalltalk object persistent. Developers use this type of connector to make global variables persistent.

- In addition to the object repository, GemStone also provides a programmable server-based system based on the Smalltalk object model. **Class connectors** establish a correspondence between Smalltalk class objects and GemStone class objects. This type of connectors allows developers to regard the data in the object repository as objects rather than plain bits.

- **Class variable connectors** and **class instance variable connectors** link variables within a Smalltalk class to variables within a GemStone class. These connectors first resolve the named objects representing the classes (i.e., connect them with class connectors) and then connect the corresponding class or class instance variables by name.

Figure 5.8 illustrates the GemStone connector browser displaying the connectors used by the persistence component.

Root objects specify which application objects are persistent, and connectors attach them to GemStone

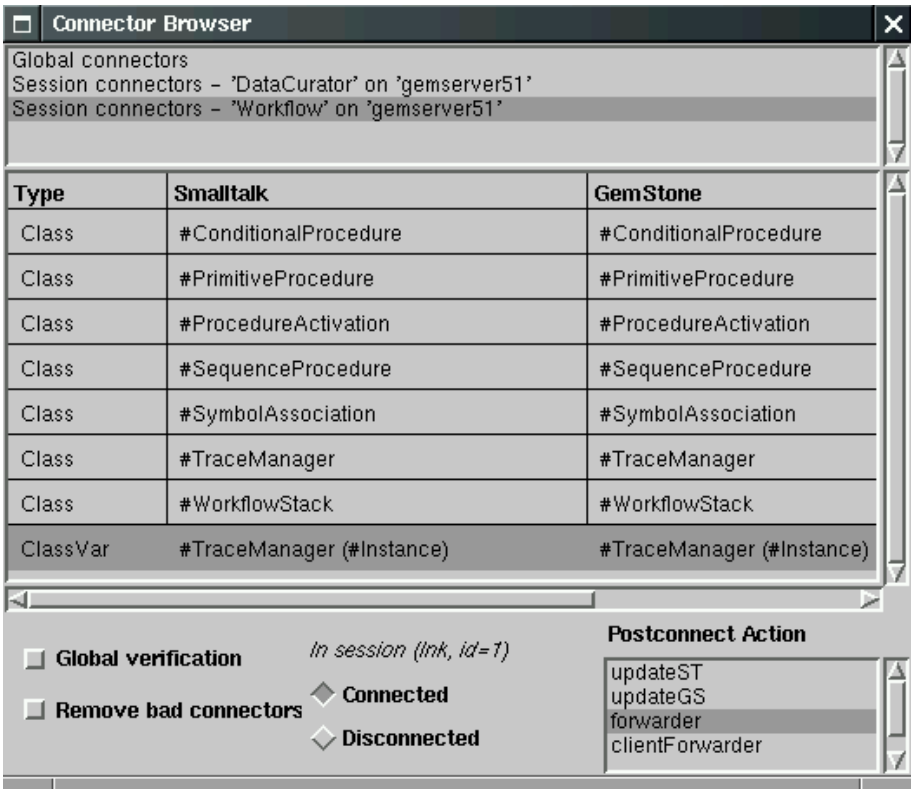


Figure 5.8: The persistence component connectors displayed in the GemStone connector browser.

objects. But once developers connect the root objects to GemStone objects, how do they specify whether to save them into GemStone or restore them from the repository? GemBuilder connectors have a postconnect action that controls the direction of data flow upon connection. Connectors can initialize objects in the repository from objects in the Smalltalk image (referred to as “updateGS”). Typically GemStone developers use this type of postconnect action when the application needs to initialize the database. Connectors can also restore objects in the Smalltalk image from persistent objects in the repository (referred to as “updateST”).

5.2.2 The Structure of GemStone Applications

Objects are characterized by both behavior and state and therefore object-oriented databases must deal with both aspects. A powerful feature of GemStone is that objects can reside *and also execute* either on the client side—in Smalltalk—or on the server side—in the Gem [41]. This is one of the characteristics that set it apart from other object-oriented databases like Objectivity [91], Versant [128] or Objectstore [92].

The possibility of running applications on the server as well as on the client enables GemStone programmers to optimize their designs by partitioning the code between the two sides. Typically the parts of an application that are data-bound, handle queries, transactions, and concurrency reside in GemStone. For example, a query that involves searching a large set should execute entirely on the server side. Once the search completes, GemStone automatically transfers only the results (usually a subset) to the client side. This lowers the traffic between the server and the client, and can have a dramatic impact on performance. In contrast, the parts of an application that require user interaction reside in the client image.

Figure 5.9 shows the typical structure of an application employing the persistence services of a GemStone/S object server.

Server-side Behavior

On the server side, GemStone/S provides a programmable object system similar to the Smalltalk object model. Class connectors and forwarders enable GemStone developers to connect client and server classes, which in fact represent the components of a distributed application.

GemStone allows a class to reside completely on the server side. In this case, a special type of connector called “forwarder” attaches the empty Smalltalk class (a proxy) to the GemStone class. The forwarder responds to all messages by passing them to the GemStone object. This mechanism enables client applications

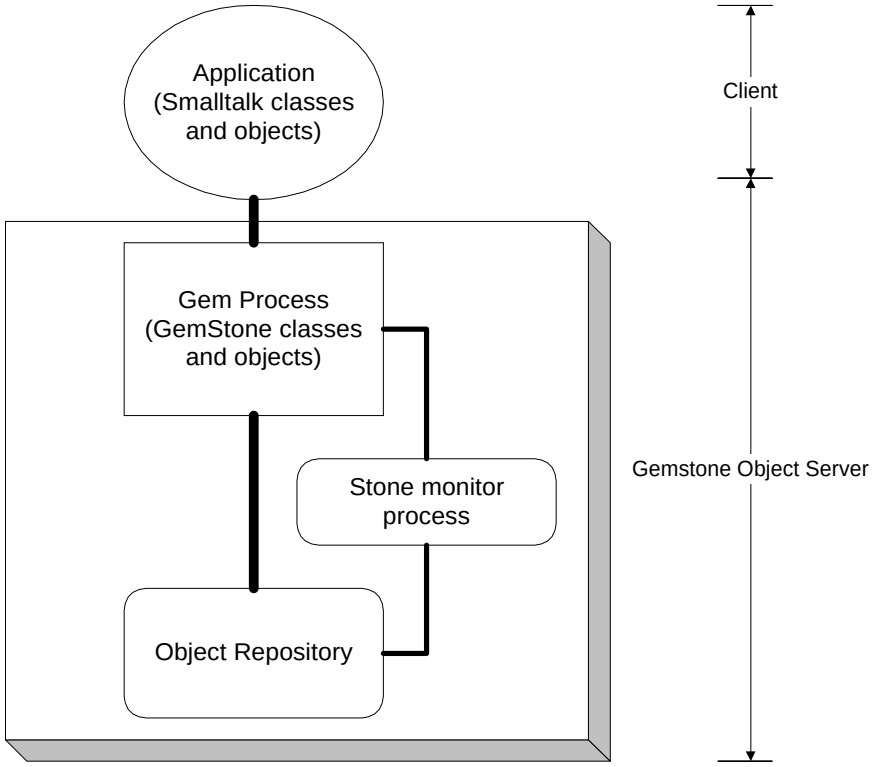


Figure 5.9: The structure of a GemStone/S object server application.

to send messages to objects that reside solely on the server side.

Sessions

All GemStone classes and methods reside in a server-based repository. GemStone developers access objects on the server by establishing a session with the repository. GemStone creates a separate Gem process (Figure 5.9) for each session. Sessions provide support for multiple users and control transactions.

GemStone sessions allow concurrent users to share objects. The server provides several mechanisms for user authentication. Once a user authenticates and obtains a session, the server controls the access to individual objects through an authorization mechanism.

Transactions enable users to commit changes to the repository. At the beginning of a transaction, the server obtains a snapshot of the repository. This snapshot provides a private view of the repository and offers a consistent image of the persistent objects during the transaction. Any changes to the objects within this view are invisible outside the transaction. A successful commit propagates the changes back to the

repository.

5.2.3 The Structure of the Persistence Component

Following the GemStone/S client-server model, the persistence component consists of two parts. On the client side, several support classes enable the Smalltalk image to connect to the server. In contrast, the classes that represent the workflow history and other classes that manage the stored workflow events reside only in server space.

5.2.4 Persistence Component, Client Side

The persistence component controls the GemStone sessions with a SessionManager. This manager implements the mechanism that establishes a GemStone session with the server. Access to the server side of the persistence component takes place within a GemStone session.

Each workflow process employing the persistence component has its own SessionManager instance. This manager has three responsibilities:

Set up the connection between the client side and the server side This consists of preparing the session by configuring several session parameters, like the Stone name and the authentication information required to connect to the server. The manager also configures the connector that links the two parts of the persistence component. A class variable connector links the class variable Instance of the GemStone TraceManager class to its Smalltalk counterpart. Figure 5.8 shows this connector highlighted in the middle pane of the connector browser. At run time the connector forwards the messages sent to the TraceManager Smalltalk object to the TraceManager GemStone object. Figure 5.10 shows the code fragment that establishes this connection.

Control the GemStone sessions The manager connects to the server by sending the login message to an instance of the GbsSessionParameters class. A successful login returns a new GemBuilder session, which is an instance of the GbsSession class. Once the workflow completes execution, the SessionManager closes the GemStone session by sending the logout message to the GbsSession instance.

Configure how the object server handles transactions GemStone offers two transaction modes. In the automatic transaction mode, committing or aborting a transaction automatically starts a new one. This

```
addDataConnectorsFor: aSessionParams
    | con |
    aSessionParams addConnector: (con := GbsClassVarConnector
                                    stName: #TraceManager
                                    gsName: #TraceManager
                                    cvarName: #Instance).
    con postConnectAction: #forwarder
```

Figure 5.10: Connecting the server and client sides of the persistence component through a class variable connector.

mode is best suited for programs that require frequent commits. In contrast, the manual transaction mode allows sessions to run outside a transaction. This mode runs with less overhead for applications that read objects from the repository and seldom require committing changes. Since the framework deals with slow running processes, I chose to have the object server operate in manual mode.

The first two responsibilities of the SessionManager are typical of all GemStone applications. Access to the object repository requires a GemStone session obtained by logging into the server. Within this session, developers save or restore objects to or from the repository.

The GemStoneLogging strategy introduced in Section 5.1.2 interacts with a single class of the persistence component, the TraceManager. But this class exists only on the server side and therefore responds to messages within the GemStone virtual machine. In reality, the logging strategy sends messages to a client side proxy. This proxy uses a class variable connector to forward messages to the server side and return results to the client side. The GemBuilder environment hides all these details, providing transparent access to GemStone objects. This requires that the objects at the two sides of the forwarding connector have the same interface. Therefore, the TraceManager class as well as the message that returns the forwarder require definitions on the client side. Figure 5.11 shows these definitions in the Smalltalk image, and Figure 5.12 shows the UML class diagram of the persistence component, client side.

5.2.5 Persistence Component, Server Side

The persistence component exploits the client-server architecture typical of GemStone applications. Besides storing the workflow events in the repository, the server side also manages the access to this information.

```
StringKeyValueDictionary variableSubclass: #TraceManager
        instanceVariableNames: ' '
        classVariableNames: 'Instance '
        poolDictionaries: ''
        category: 'Workflow-Persistence'

TraceManager class>>instance
        Instance isNil ifTrue: [Instance := self new].
        ^Instance
```

Figure 5.11: The client side definitions of the object on the server side of the connector.

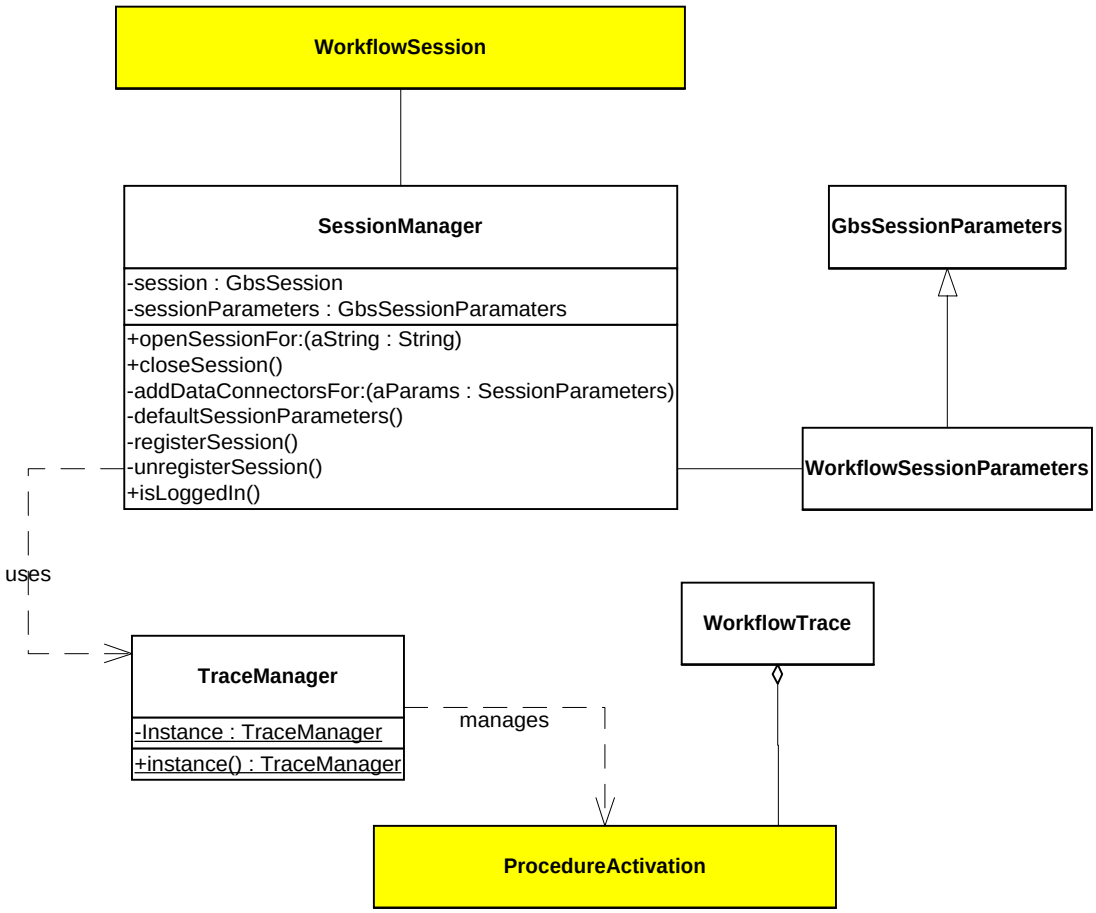


Figure 5.12: Class diagram of the persistence component, client side. These classes reside on the client, in the Smalltalk image.

Therefore, the server side of the persistence component plays two roles: the role of a persistent store and the role of a manager.

Persistent Store

The MemoryLogging strategy described in Section 5.1.2 stores the workflow events into an OrderedCollection. The logged events reside within the Smalltalk image and therefore are transient. Additionally, the memory available to the Smalltalk system limits the number of events that MemoryLogging can log.

GemStone/S provides a rich set of collection classes that mirror the client class hierarchy. However, unlike their Smalltalk counterparts, GemStone classes reside in the object repository, independently of the client image. Since GemStone accesses the repository through disk caches, it can store data beyond the capacity of the available memory.

The WorkflowTrace class provides a persistent container. Instances of this class hold workflow events. As explained in Section 5.1, the framework adds new workflow events in their order of execution. WorkflowTrace maintains this ordering by subclassing the OrderedCollection GemStone class. Since the framework and the database share the same programming language, this implementation actually resembles the solution adopted by the MemoryLogging strategy. However, instead of a Smalltalk collection, WorkflowTrace adds the workflow events to a GemStone persistent collection.

Manager

The TraceManager class provides the server-side functionality of the persistence component. Its role corresponds to the *Manager* [120] pattern. This class has three main responsibilities:

Single point of access to the persistent store The TraceManager employs the Singleton pattern [39] to provide the same instance to all clients. GemStoneLogging instances of the history component (Section 5.1) access the sole instance through a GemBuilder class variable connector. When the client side of the persistence component opens the session (Section 5.2.4), it configures this connector as a forwarder.

GemStone uses the class variable holding the Singleton instance to attach objects to the repository. In this case, the variable connects the TraceManager instance to its class. But the class resides in a GemStone symbol dictionary, within the repository. Therefore, once the persistence component

initializes this variable, subsequent commits will save the TraceManager instance to the object server. Figure 5.13 shows the corresponding instance diagram.

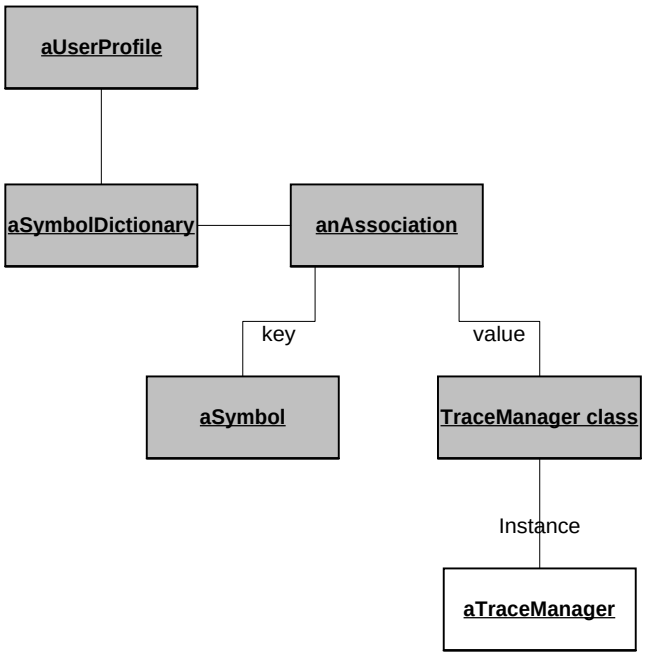


Figure 5.13: A simplified instance diagram of the trace manager. The Instance class variable connects the TraceManager instance to its class, which is persistent—along with the other shaded objects.

Access to individual workflow traces The manager maintains a separate history for every workflow that uses persistent logging. It implements this separation by subclassing the RcKeyValueDictionary GemStone class. This reduced-conflict class uses a sophisticated conflict checking algorithm to allow any number of workflow sessions to read and add workflow traces at the same time.[1] Each entry in this dictionary stores a WorkflowTrace instance. The workflow provides a unique identifier which serves as a dictionary key. Along with GemStone's multi-user capabilities, this design ensures that concurrent workflow instances can use the persistence component without interfering with each other on the server side.

History operations on workflow traces The manager implements an interface similar to LoggingStrategy. However, adding a workflow event, accessing the history, as well as preparing for rewind all require an additional argument specifying the trace.

[1] Concurrent accesses still cause conflicts if a user tries to add a key that already exists while other users add new keys, or more than a user tries to remove the same key at the same time.

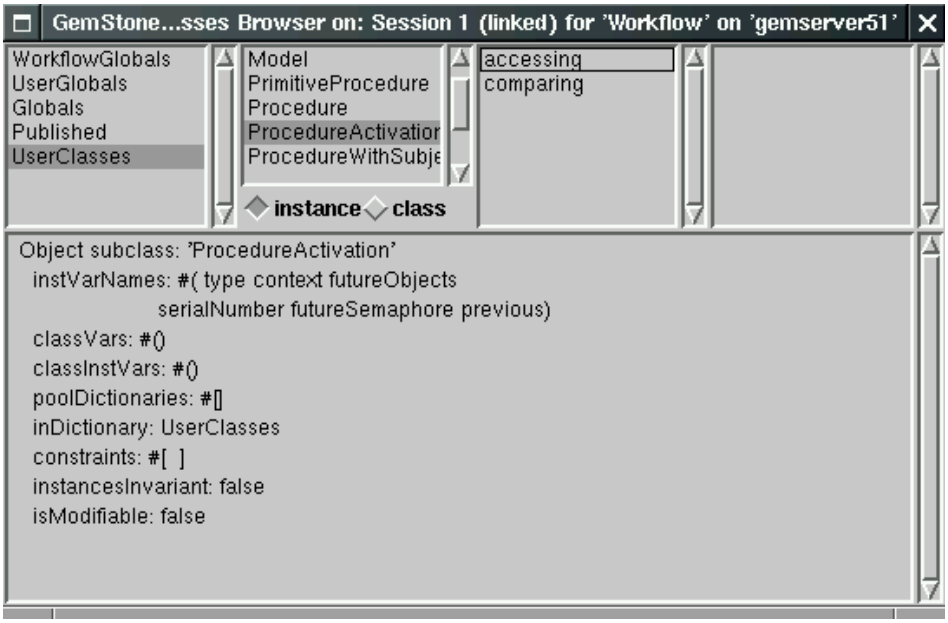


Figure 5.14: The UserClasses dictionary contains GemStone classes automatically generated by GemBuilder.

A key feature of this client-server design stems from the fact that computing the backtrace stream required by rewind operations takes place on the server side. The persistence component doesn't need to transfer the workflow history back to the Smalltalk image. Therefore, the manager can handle workflow traces larger than what can fit within the Smalltalk image.

Since the server side is in fact a Smalltalk system, it regards the data in the repository as objects. At run time, if a client Smalltalk object needs to be replicated in GemStone but it belongs to a class that doesn't already exist on the server, GemBuilder automatically generates a GemStone class with the same structure and position in the class hierarchy. However, this automatic generation involves only class structure and no behavior. GemStone developers have to use the GemBuilder programming tools to add the methods their objects require on the server side.

The GemStone system consults several symbol dictionaries to resolve the names of the objects referenced in applications. For example, the Globals dictionary contains the associations for the Kernel classes. Likewise, the UserClasses dictionary contains the class definitions automatically generated by GemBuilder. Figure 5.14 shows a snapshot of this dictionary.

The server side of the persistence component manages instances of the ProcedureActivation class. This class resides on the client and its behavior is defined within the Smalltalk image. The GemBuilder-generated definition of this class provides only class structure. Although indirectly, on the server side the TraceManager sends messages to ProcedureActivation instances—Figure 5.15. This requires the GemStone ProcedureActivation class to implement "=" and hash (required by the TraceManager), and serialNumber (required by "="). Figure 5.16 shows the UML class diagram of the persistence component, server side.

```
TraceManager>>prepareToRewindTo: anActivation for: aWorkflowName
    |history backtrace|
    history := self allHistoryFor: aWorkflowName.
    backtrace := (history copyFrom: (history indexOf: anActivation)
                to: history size) reverse.
    backtraceStream := ReadStream on: backtrace
```

Figure 5.15: Processing within GemStone involves sending messages indirectly (in indexOf:) to ProcedureActivation instances.

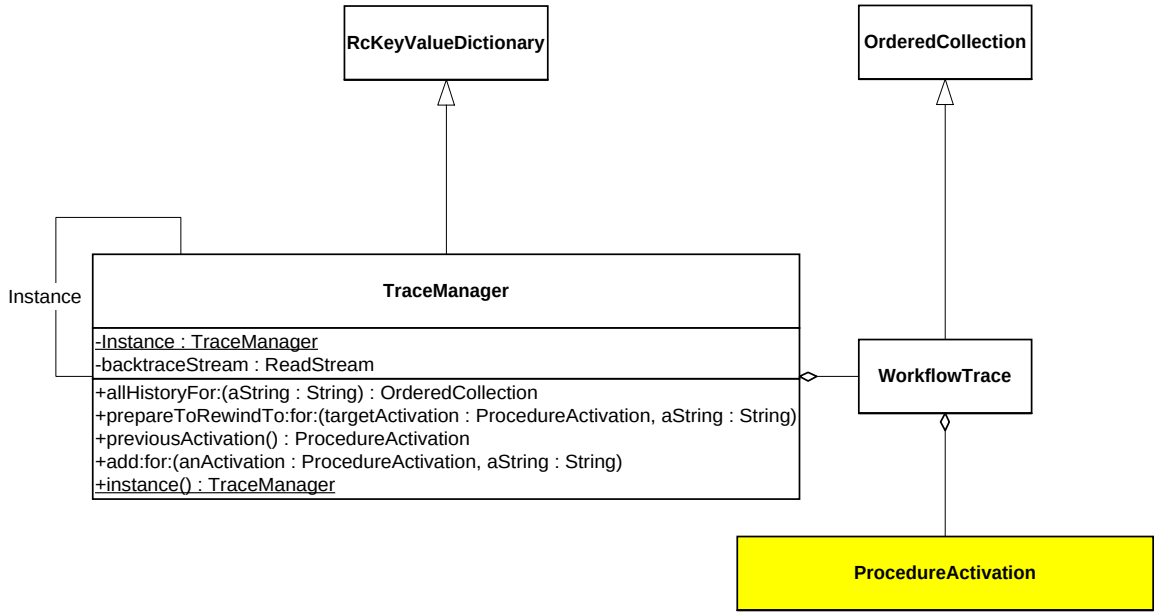


Figure 5.16: Class diagram of the persistence component, server side. These classes reside in GemStone.

5.2.6 Persistence with Relational Database Technology

Object-oriented databases provide the ideal solution for object systems that need to save and restore objects. However, sometimes object-oriented developers have to use relational databases for persistence. In fact, this situation is so common that some vendors have packages that provide standardized interfaces for reading and writing rows from and to relational databases (e.g., JDBC [134], ODBTalk [93], etc.).

Storing objects in a relational database amounts to saving their state into tables. Usually developers use an object-to-relational layer that handles the mapping between objects and tables. This layer performs automatically some of the additional work required to save and restore objects to and from tables. However, the developer has to provide "datastore maps" that describe how to map each class into one or several table(s).

For example, let's review the instance diagram from Figure 5.6. Section 5.2.1 described how GemStone handles the details of object persistence in a manner that is transparent to the programmer. The table rows at the foundation of the relational paradigm can also store object state but require additional work. The persistence layer providing the object-to-relational mapping navigates the object graph and saves or restores the objects it encounters.

There are several ways to organize the database schema. They depend on how much freedom developers have in organizing the database, and on whether the focus is on speed or size. For example, the schema can use a separate table to store the *complete* state of each class. This design has low overhead at the expense of a large footprint. The schema can also use a table for each super-class, distributing the state of each object across several tables. This solution adds the overhead of joins between rows, but requires less space in the database. Polymorphism and class hierarchies further complicate matters since they don't have direct correspondents in the relational paradigm.

In the instance diagram from Figure 5.6, saving an activation into a relational database management system (RDBMS) translates into writing a row in the table associated with the ProcedureActivation. However, this only partially records the state of the activation instance. To save all the information, the object-to-relational layer traverses the relationships and updates the tables corresponding to the other five classes that the activation references. Likewise, loading an object from a RDBMS amounts to joining table rows via their relationships rather than navigating object relationships. The activation object identifier (OID) provides the primary key in the table corresponding to the ProcedureActivation class while the tables corresponding

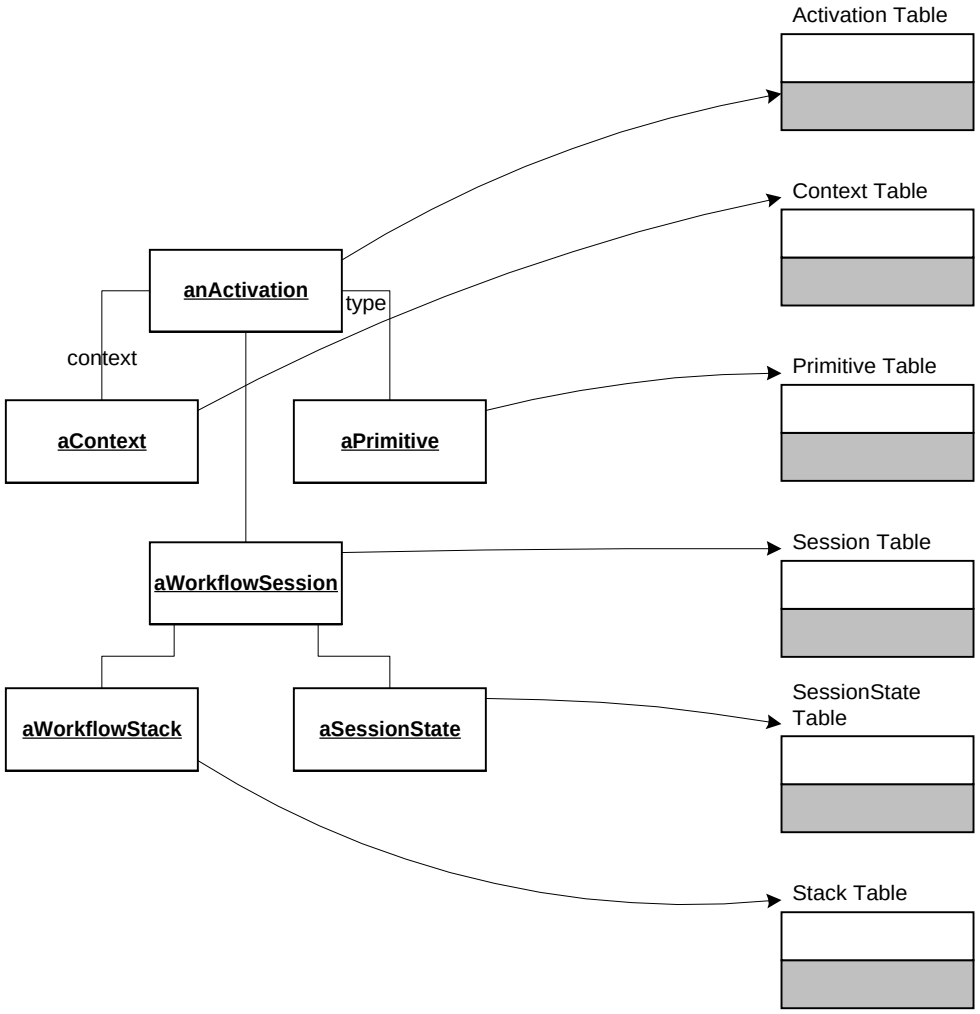


Figure 5.17: Object-to-relational mapping when each class maps into a single table.

to the related classes use it as a foreign key. Figures 5.17 and 5.18 show two possible database schemas corresponding to the instance diagram from Figure 5.6.

To summarize, on the one hand using a relational database to store objects is more complicated than using an object-oriented database. Developers have to handle the mapping of classes and class hierarchies into tables. Additionally, they have to address the issues of key generation, relationship traversal, dirty marking, and data conversion. On the other hand, mapping objects to relational databases is a well-known problem [5]. In fact, developers have solved this problem so many times that Yoder and colleagues [139] have distilled this knowledge into a pattern language. Off-the-shelf products (e.g., TopLink [123], Java Blend [64], etc.) provide ready-to-use solutions that free developers from implementing this functionality.

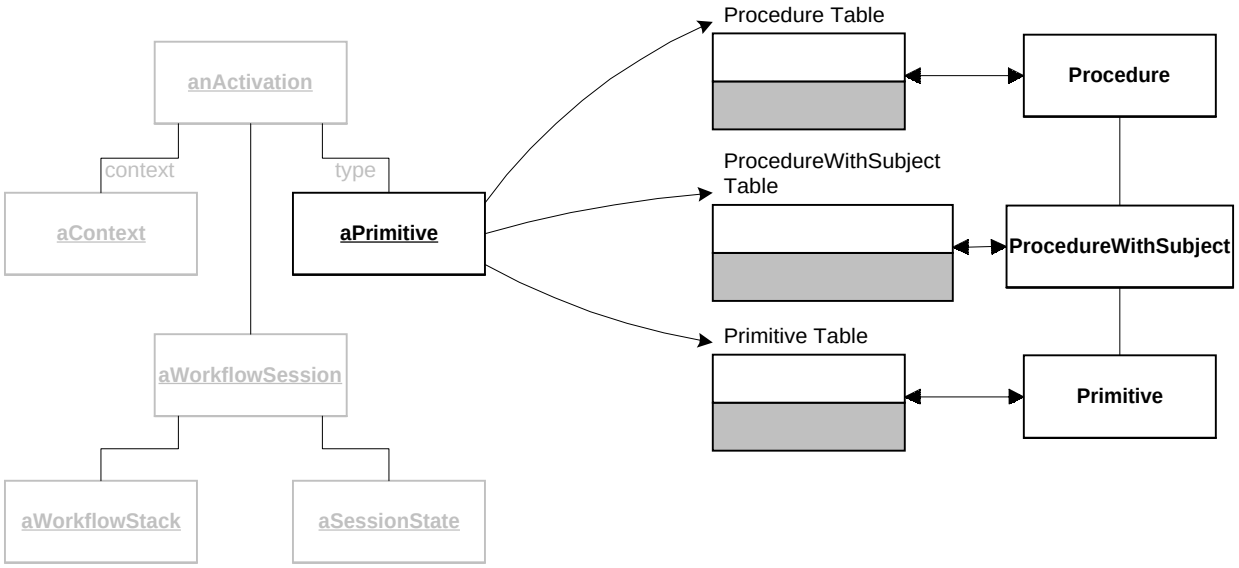


Figure 5.18: Object-to-relational mapping when each super-class maps into a separate table. This example shows only the tables corresponding to the Primitive class, which has two super-classes.

In the context of the micro-workflow persistence component, the main difference between the two types of databases lies in saving and restoring application objects. For simplicity, Figures 5.6–5.7 and 5.17–5.18 don't show the objects within the activation context. In reality, the context contains application objects that should be saved and restored along with the framework objects. A persistence component based on relational database technology would provide the datastore maps for all the framework objects associated with the workflow events. However, developers have to describe how their domain objects should map into tables. Without this information, the persistence component can't handle the contents of the activation context. In contrast, GemStone/S handles persistence in a transparent manner and therefore saves and restores application objects without additional information from the user.

5.2.7 Discussion of the Persistence Component

The persistence component enables the other components of the micro-workflow framework to use a persistent store. It hides all details of saving and restoring objects, as well as accessing the database and controlling the database sessions. To this end, the other framework components are database independent. Claus Hagen's PhD thesis identifies database independence as an important prerequisite for process support systems [49].

Notice, however, that the solution described in this section represents an extreme of a wide spectrum. In this implementation there's no impedance mismatch between Smalltalk and GemStone/S. Both are object-oriented systems and therefore the saved objects are not "flattened" inside the persistent store. Additionally, since GemStone/S provides an object server programmable in Smalltalk, the design exploits the client-server architecture by distributing behavior among the two sides. Consequently, this approach yields a thin and elegant persistence component. Relational database technology provides a solution at the other side of the spectrum. The clash between objects and the table rows paradigm requires extra work for storing objects into relational databases. In effect, this choice would significantly increase the complexity of the persistence component. Nevertheless, encapsulating persistence into a separate component allows developers to customize the framework for virtually any database system, without affecting the other components.

Table 5.2 lists some of the possible ways software developers could take to customize the persistence component.

Aspect	**Details**
Change session management	Subclass or replace the SessionManager class
Representation in persistent store	Subclass or replace the WorkflowTrace class
Workflow history management	Subclass, replace, or customize the TraceManager class
Use a different DBMS	Replace the SessionManager, WorkflowTrace and TraceManager classes

Table 5.2: Customizing the persistence component.

5.3 Workflow Monitoring

Workflow monitoring represents another feature common to many workflow management systems. A monitor provides information about the status of running workflows.

Monitoring allows workflow users to check how the process is running, and helps them identify of out-of-line situations. Sometime workflow users may be able to correct the identified problems at run time. Other times they may stop the process when no solution is available. Since typically workflows run for a long time, the possibility of obtaining runtime information in real time and the early identification of

potential problems could save time and other resources.

For example, let's review the strep throat treatment process introduced in Section 2.1.1. A workflow management system executing this process assigns patients to physicians and nurses. Therefore, at any point in time it can provide the work assignments of the hospital staff involved in the process. The workflow system records this information to the process history. *After the workflows finish execution,* historic data enables the hospital administration to derive statistical information about their operation. These numbers provide the information required for resource allocation, e.g., the number of physicians and nurses the hospital needs to handle the strep throat cases in a timely manner. They also help identify potential bottlenecks. In contrast, a workflow monitor shows the run time information in real time, *as the workflows execute*. At any given moment, workflow users can see the state of the workflows. For instance, the hospital administration may use this feature to dynamically adjust the workload of their physicians and nurses by reassigning staff, thus reducing the time their patients spend in the waiting room.

Workflow monitoring means different things in different contexts. Leymann and Roller [72] observe:

> Depending on the focus, at least three different flavors of monitors can be differentiated: the process monitor, the workload monitor, and the system monitor. [...] The *process monitor* presents the current or accumulated states of processes of a particular process model. [...] The *workload monitor* supports the monitoring of the amount of work that is carried out by the users and organizations. [...] The *system monitor* provides an overall picture of the workflow management system's operation.

Therefore it is hard to predict all types of information workflow users may want to monitor. A successful workflow system should allow them to customize workflow monitoring.

The monitoring component extends the micro-workflow core described in Chapter 4 with workflow monitoring. The compositional approach lets developers add monitoring to the framework *only when their applications need this functionality*, and localizes the changes required to customize this feature. Additionally, it facilitates application integration.

5.3.1 Usage

The application domain and requirements of each process have a strong impact on the design of a workflow monitor. For example, some applications require monitors involving GUIs that display workflow informa-

tion. Other applications need alarms whenever the workflow enters certain states. The monitoring component separates *how it tracks* workflow execution from *what it does* with this information. As an example of the latter the micro-workflow framework uses a GUI that resembles the view provided by a debugger.

The state of the workflow changes in response to the execution of process activities. Therefore, the micro-workflow monitoring component should track the execution component as it fires off procedures.

The ProcedureMonitor class implements the monitoring component. It receives notifications when the framework executes procedures. Developers plug in the monitoring component by hooking it to the execution component. This approach ensures that the execution component does not depend on the monitoring component, thus providing a great deal of flexibility. For example, developers can use several monitors to simultaneously track process evolution. Some monitors could have GUIs, and others could trigger alarms. Alternatively, developers could leave out this functionality when the application doesn't require monitoring workflow execution.

5.3.2 Design Details

The monitoring component uses the *Observer* [39] pattern to receive notifications whenever the execution component fires off procedures.

Procedure instances signal when they execute through the WorkflowSession, which maintains the observation relationship(s) with an arbitrary number of monitors. Therefore, WorkflowSession plays the role of the subject, while ProcedureMonitor instances play the role of observers. Since the WorkflowSession navigates the activity map with the flow of control, this design ensures that the subject end of the observation relationship is always attached to the executing procedure.

ProcedureMonitor maintains a queue with the most recent ProcedureActivation instances generated by the execution component. Each notification signals that a new workflow event is available. Upon receiving a notification, the monitor updates its queue. This design yields a loose coupling between the execution and monitoring components. It allows micro-workflow to have an arbitrary number of monitors, and keeps the monitor interface simple. Figure 5.19 shows the UML class diagram of the monitoring component. The code fragments from Figure 5.20 show how the Procedure sends out the notification (through signalExecutionOf:), and how the ProcedureMonitor reacts upon receiving a notification event.

The framework provides an example graphical user interface for the ProcedureMonitor. The interface

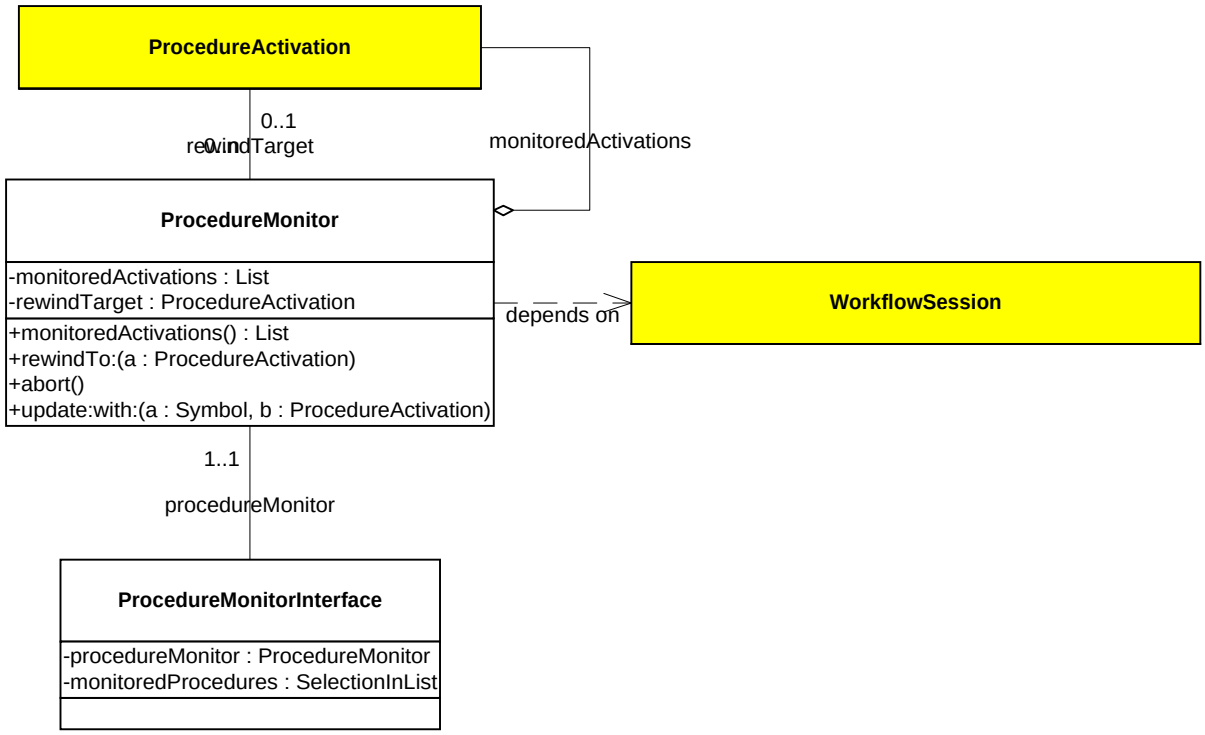


Figure 5.19: Monitoring component, UML class diagram. The colored/shaded classes belong to the execution component.

```
Procedure>>executeNewInstanceFollowing: previousActivation
    | next result |
    next := ProcedureActivation newWithType: self.
    next prepareToSucceed: previousActivation.
    self workflowSession signalExecutionOf: next.
    next logWorkflowEvent.
    next localState: (self computeStateFor: next).
    result := self executeProcedure: next.
    ^result

ProcedureMonitor>>update: aSymbol with: anActivation
    aSymbol = #instance ifTrue: [self add: anActivation].
    ^super update: aSymbol with: anActivation
```

Figure 5.20: The monitoring component uses the *Observer* pattern to hook up a workflow monitor to the execution component.

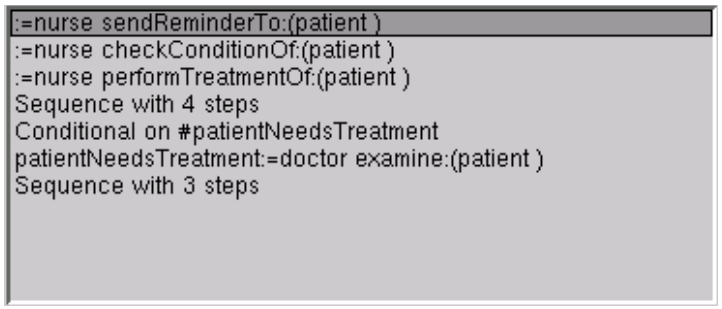


Figure 5.21: Workflow monitor graphical interface.

displays the workflow events as the process unfolds in time. The example from Figure 5.21 shows the most recent activation highlighted, at the top of the list. ProcedureMonitorInterface, a subclass of VisualWorks' ApplicationModel implements this functionality. However, the monitoring component remains consistent with the overall goal of the architecture, allowing developers to customize it according to their needs. For example, they can use the functionality provided by the monitoring component to build GUI-based monitors, alarms, or integrate it with the application's GUI.

5.3.3 Discussion of the Monitoring Component

Unlike current workflow systems, micro-workflow relies on a separate component to implement workflow monitoring. Software developers plug in this component only when they need its functionality. Additionally, this approach reduces the coupling between workflow execution and monitoring, thus allowing developers to customize one without changing the other.

The Mentor-lite project also regards workflow monitoring as a separate facility of the workflow system [85]. However, Mentor-lite implements its extensions with state and activity charts (i.e., as workflows) instead of objects.

Leymann and Roller [72] discuss a history-based workflow monitor. Their solution tracks the logged workflow events through periodic queries or database triggers. The *Observer*-based design described in this section doesn't introduce dependencies between the monitoring, history, and persistence components. Therefore this solution lets software developers mix and match freely the components providing advanced workflow features.

Table 5.3 summarizes several potential ways of customizing the manual intervention component.

Aspect	Details
Add monitors	Subclass ProcedureMonitor or register other observers that respond to the same notification protocol
Monitor other framework components	Add event notifications to other objects within the framework

Table 5.3: Customizing the monitoring component.

5.4 Manual Intervention

Workflow users may need to manually change the sequencing of process activities.

5.4.1 Context

Let's review the strep throat process discussed in Section 2.1.1. The physician decides which treatment to prescribe based on whether the patient is allergic to antibiotics. However, people who know that they are not allergic to penicillin/antibiotics can develop the allergy later on. Usually they discover that they acquired the allergy when some medicine triggers an allergic reaction.

Now let's assume that a patient has developed an allergy to antibiotics that he is not aware of.[2] Since his file doesn't mention the allergy, the physician prescribes the penicillin treatment. The patient goes home and begins following the treatment. However, shortly after he takes penicillin he starts feeling ill. He calls the doctor's office and the nurse recognizes the symptoms of an allergic reaction. She asks him to come back and updates his file. Based on the updated information, the physician prescribes the sulfa drug treatment which is suitable for people allergic to antibiotics.

5.4.2 Problem

Sometimes workflow users need to override the workflow definition and manually specify the activity node that the system should execute next. For example, the availability of new information (like in the strep throat example) might require backtracking and continuing the workflow from a previous step. How does a workflow system accommodate this situation?

[2] This is a fairly common situation.

5.4.3 Solution

Workflow management systems must allow authorized users to change the sequencing of activities while the workflow is running. (If they don't, their users perceive them as too rigid and avoid using them altogether [89].) In the case of an activity-based process model, this corresponds to manually moving the flow of control from one node of the activity map to another.

The manual intervention component adds this type of functionality to the micro-workflow core introduced in Chapter 4. It enables workflow users to change a running workflow by jumping back to an arbitrary point in the sequence of past workflow events. Users can interrupt the execution component and select a new procedure from the process history. The system backtracks and resumes execution from the selected activity.

5.4.4 Usage

The manual intervention component provides the mechanism for rewinding procedures. Instances of the Rewinder class encapsulate this functionality. To use the manual intervention component, software developers set the class implementing the rewind mechanism into the workflow session by sending the rewinderClass: message. Therefore, the micro-workflow framework enables software developers to plug in the manual intervention component *only when their processes need its functionality*.

At run time, framework users trigger a rewind by sending the rewindTo: message to the workflow's PreconditionManager (see Chapter 4). In response to this message, the manager takes the control away from the execution component and the manual intervention component takes control. Upon being activated, the manual intervention component backtracks the process to the target procedure activation, restores the workflow state, and then gives the control back to the execution component. This resumes workflow execution form the procedure specified as the rewind target.

5.4.5 Design Details

Implemented and implementing control flow

Altering process execution at run time translates into manipulating the workflow call stack. This requires separating the control flow mechanism of the *implemented* system (micro-workflow) from the control flow

mechanism of the *implementing* system (Smalltalk). The execution component accommodates this separation through the workflow session, which holds a WorkflowStack instance for calls within the workflow domain.

The Procedure class provides the mechanism that manipulates the workflow stack. From all procedures provided by the process component (sequence, primitive, conditional, repetition, iterative, fork, and join—see Section 4.3), only the composites use this mechanism. Primitives transfer control to domain objects through the Smalltalk message send mechanism.

The call:with: message enables a procedure instance to pass control to another procedure. If we picture the activity map as a tree structure with the root node on top and the leaf nodes at the bottom, a call within the workflow domain corresponds to descending one level. This message pushes on the workflow stack the parent procedure, and transfers control to the next level. SequenceProcedure sends call:with: to execute its steps. Likewise, ConditionalProcedure, RepetitionProcedure, and IterativeProcedure send it to execute their body procedure. Procedures return control to their parent/caller by sending the return: message. Upon receiving return:, a procedure obtains its caller from the workflow stack and transfers control through the return:state: message. This message restores the state of the caller and resumes its execution. Figure 5.22 shows the UML class diagram of the manual intervention component, and Figure 5.23 shows how the Procedure class responds to the call:with: and return: messages.

The Procedure class doesn't implement return:state:. The state information is type dependent and therefore each subclass provides its own implementation:

- ConditionalProcedure returns to its parent the activation obtained from the body procedure.
- SequenceProcedure and IterativeProcedure send the advanceState message, followed by executeProcedure:withState:. advanceState computes the state corresponding to the next sequence step or subject component, respectively.
- RepetitionProcedure evaluates its guard. Based on the guard's value it either returns control to its parent, or repeats executing its body procedure.
- The instances of PrimitiveProcedure can only be leaf nodes and don't call other procedures. Primitives shouldn't receive this message and therefore they don't implement it.

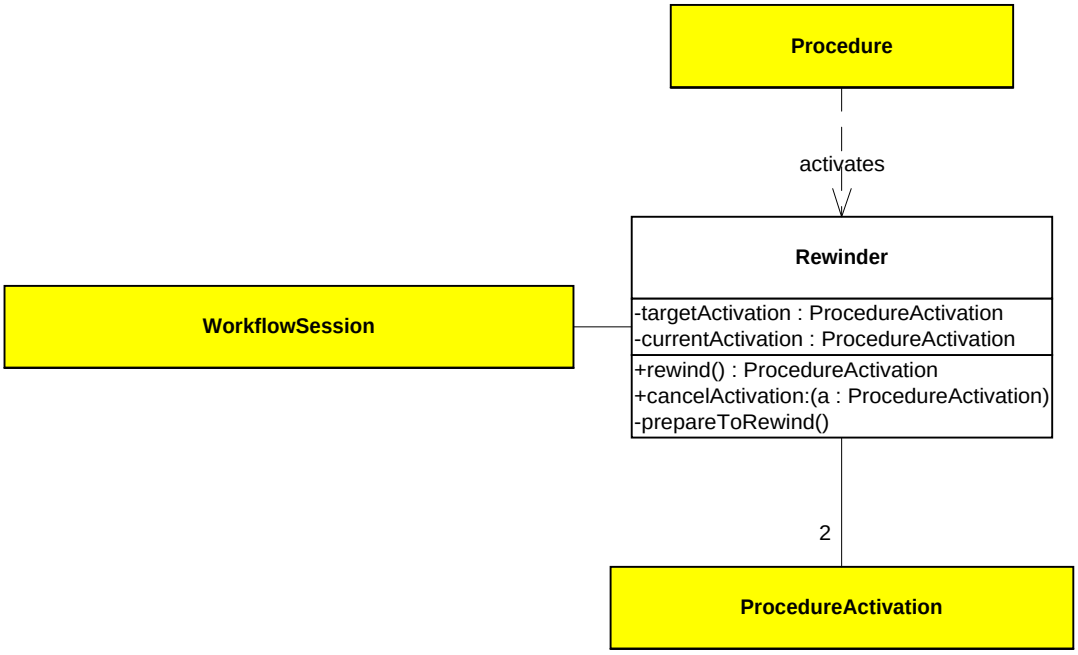


Figure 5.22: The manual intervention component, UML class diagram. The colored/shaded classes belong to the execution component.

```
Procedure>>call: aProcedure with: anActivation
    | stack |
    stack := anActivation workflowSession workflowStack.
    stack
        push: (StackRecord for: self copy withState: anActivation localState).
    anActivation workflowSession workflowStack: stack.
    ^aProcedure continueExecutionOf: anActivation

Procedure>>return: anActivation
    | stack stackRecord |
    stack := anActivation workflowSession workflowStack.
    ^stack isEmpty
        ifTrue: [anActivation]
        ifFalse:
            [| procedure |
            stackRecord := stack pop.
            procedure := stackRecord procedure.
            procedure return: anActivation state: stackRecord state]
```

Figure 5.23: The Procedure class provides its subclasses with the mechanism that enables them to transfer control to/from other procedures.

```
Rewinder>>rewind
    | savedActivation |
    self prepareToRewind.
    savedActivation := currentActivation previousActivation.
    [savedActivation = targetActivation] whileFalse:
            [self cancelActivation: savedActivation.
            savedActivation := currentActivation previousActivation].
    savedActivation activateFrom: targetActivation.
    currentActivation restoreStackFrom: savedActivation.
    ^currentActivation
```

Figure 5.24: The Rewinder walks back through the logged activations until it finds the desired activation. The cancelActivation message provides a hook for backward recovery.

Rewind to procedure

So how is this relevant to manual intervention? The workflow session of every ProcedureActivation instance contains a copy of the workflow stack. Therefore, each activation object saved in the workflow history contains the information required to restore the sequence and the state of the callers within the workflow domain.

Manual intervention requires support from the precondition manager discussed in Section 4.2. Users request a rewind by sending the rewindTo: message to the PreconditionManager instance. The manager responds by sending the forcedResumeTo: message to the waiting preconditions. This removes them from the queue regardless of whether their blocking conditions are fulfilled or not. The value returned by the waitUntilFulfilledIn: message to waiting procedures specifies the target of the rewind request, if any. When a rewind has been requested, the interrupted procedure passes control to a Rewinder instance (i.e., the manual intervention component) which backtracks the process and restores the workflow stack from the saved activation. Figure 5.24 shows how the Rewinder responds to the rewind message sent by the execution component.

This scheme allows for backward recovery in case of semantic errors (e.g., the user rewinds a workflow that started with erroneous data). Backward recovery cancels the effects of executed activations (if possible) as it rolls back to the target activation. Software developers can choose any of the cancellation mechanisms typical of long-lived activities. For example:

Undo The domain object cancels the effects of an executed action. For example, let's assume that a workflow operating on the telecommunications objects discussed in Section 3.3.2 adds an agreement to an

account object. If the workflow user rewinds the process past this addition, the account object can remove the new agreement.

Semantic compensation Sometimes the domain object can't cancel the effects of an executed action. In this case the object can execute a special compensating action (or actions) for this purpose. For instance, an account domain object can't undo the sending of a refund check since the check is already in the mail. To compensate for this action the workflow sends an explanatory letter to the customer and charges the refunded amount.

The framework allows its users to request a rewind through the procedure monitor interface. At run time, a user can select a procedure from the list with executed procedures, and trigger a rewind by clicking the corresponding button. In Figure 5.25, the ProcedureMonitorInterface displays the executed procedures (most recent on top of the list) and the controls for manual intervention.

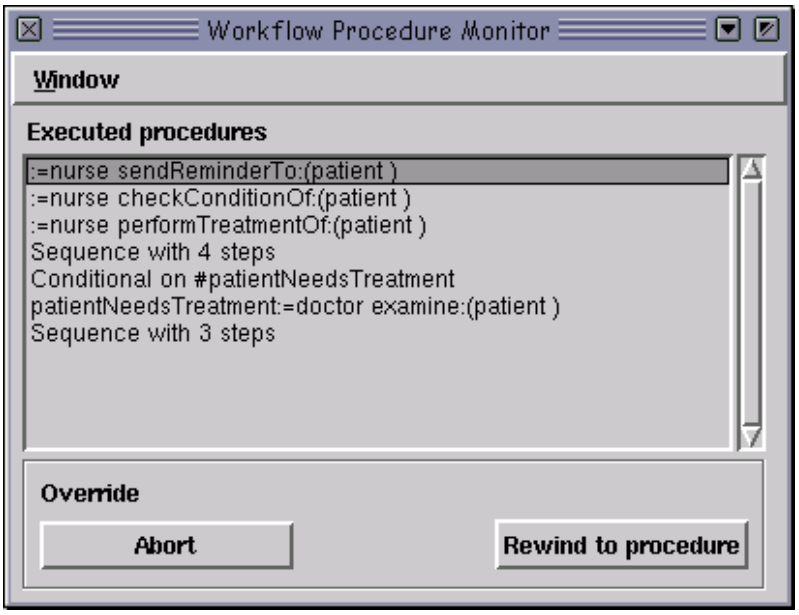


Figure 5.25: Framework users access the manual intervention component through the procedure monitor interface.

5.4.6 Discussion of the Manual Intervention Component

The manual intervention component lets workflow execution jump to any point within the process history. Therefore, it depends on the history component described in Section 5.1. Additionally, it requires support

from the synchronization component (which receives rewind requests from the GUI) and the execution component (which passes control to the manual intervention component when signaled by the synchronization component). However, since this design separates concerns, the changes have a relatively small impact on these two components. Additionally, the design facilitates the integration of the rewind triggering mechanism within applications.

I chose to have manual intervention use logged activations as rewind targets for two reasons. First, arbitrary transitions don't necessarily make sense. Typically workflow users want to change an executing process if they started it with incomplete or inaccurate data. Second, the micro-workflow activity-based process model doesn't represent state information explicitly. Altering the course of a process requires the workflow runtime (i.e., interpreter state) for the target point. The framework saves this state information when it executes a procedure but can't generate it otherwise.

Table 5.4 summarizes several potential ways of customizing the manual intervention component.

Aspect	**Details**
Recovery	Implement different backward recovery mechanisms by subclassing Rewinder.
Automatic rewind	Use monitors that track the state of the workflow and trigger rewinds

Table 5.4: Customizing the manual intervention component.

5.5 Worklists

Micro-workflow processes involve both workflow objects which encapsulate the process logic, and application objects which encapsulate the task logic. The primitive procedures located in the leaf nodes of the process activity map send messages to application objects, which perform domain-specific work. However, workflows typically involve human workers as well as application objects. In fact, as I discussed in Chapter 2, the synergy between humans and software represents one of the key features of workflow. Therefore, an object-oriented workflow management system must accommodate human workers as well as application objects.

The worklist component extends the functionality provided by the micro-workflow framework with support for human workers. This consists of an interface between the core workflow components and the people involved in the process. Additionally, the worklist component provides an asynchronous invocation mecha-

nism. Leymann and Roller identify asynchronous invocation as one of the important invocation paradigms of workflow systems [72]. Figure 5.26 sketches how the worklist component extends the framework to support human workers.

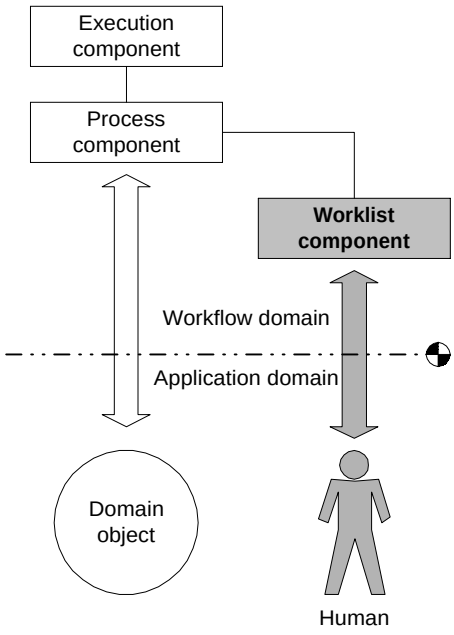


Figure 5.26: The worklist component adds to the framework the invocation mechanism and functionality required to support human workers (grayed).

Micro-workflow allows software developers to plug in the worklist component *only if their processes involve people*. This approach is consistent with one of the main goals of the architecture: developers add features to the framework by adding components on top of the workflow core.

5.5.1 Usage

The Worklist class provides an abstraction for human workers performing process activities. Primitive procedures—the control structure where the framework passes control across the domain boundary—don't differentiate between worklists and application objects. Worklist instances must be able to replace domain objects transparently. Therefore, they should respond to the messages sent from the workflow domain by PrimitiveProcedure instances (Section 4.3.3) in the same manner that the application objects they replace do.

The key difference between a domain object and a human worker stems from the fact that while objects usually process messages synchronously, human workers handle work items asynchronously. Figure 5.27

illustrates this difference.

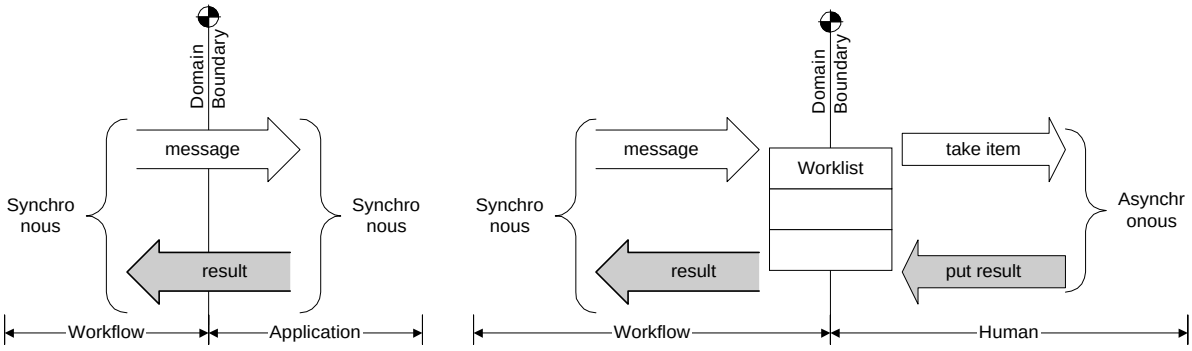


Figure 5.27: Dealing with objects (left) vs. dealing with humans (right). The open-ended arrows show synchronous messages and the close-ended arrows show asynchronous messages.

On the workflow worker side the worklist component manages worklists for human workers, and allows them to select work items and work on them asynchronously. On the workflow side this component provides an interface suitable for synchronous message sends, isolating the other framework components from the issues of human-computer interaction. The worklist component maintains the appearance of synchronous messages on one side of the domain boundary, while accommodating asynchronous operations on the other side.

Therefore, instances of the Worklist class serve a dual purpose:

- For the *user domain*, the worklist holds the work items assigned to a workflow user. When the user selects a work item to work on, the worklist removes the corresponding item. Once the user completes the work, the worklist passes the results across the domain boundary, back to the workflow domain.
- For the *application domain*, the worklist plays the role of a domain object. Developers should be able to use domain objects and worklists interchangeably.

From the perspective of the process component described in Section 4.3, replacing application objects with human workers amounts to initializing the context slot with the corresponding worklist instance instead of the domain object. Figure 5.28 shows an example from the strep throat process.

```
populateInitialContextFor: aProcedure
    | patient |
    doctorWorklist := Worklist new.
    nurseWorklist := Worklist new.
    patient := Patient newWithPenicillinAllergy: false strepThroat: true.
    aProcedure
        initialContextAt: #patient put: patient;
        initialContextAt: #doctor put: doctorWorklist;
        initialContextAt: #nurse put: nurseWorklist
```

Figure 5.28: Worklists replace application objects transparently. This code fragment initializes the context of the strep throat workflow with the worklists for the doctor and the nurse.

5.5.2 Design

Figure 5.29 shows the class diagram of the worklist component. These classes provide functionality along two orthogonal directions: human-computer interface and asynchronous invocation.

Human-Computer Interface

The worklist component translates the messages sent by primitive procedures into work items understood by human workers. This translation is application dependent. For example, in the strep throat workflow the work item in the nurse's worklist shows the case ID, the patient's name, the physician's initials, and a treatment code. Likewise, in a telecommunications provisioning workflow the work item from the technician's worklist shows the customer name, customer address, and the type of service requested.

The framework lets software developers specialize how the worklist component displays work items to human workers by defining a Workitem subclass. The default behavior (implemented by the base class Workitem) displays the corresponding Smalltalk message. User-defined subclasses allow the worklist interface to present its contents in application-specific ways.

The worklist component manages the work items (i.e., the worklist) for each workflow worker. People interact with their worklist through a GUI implemented by the WorklistGUI class. They use this graphical interface to "check out" the work item they want to work on. Software developers control how each WorklistGUI instance displays its contents by parameterizing it with a Workitem subclass. Once the human workers finish their work, an application-specific mechanism handles the return of data back into the

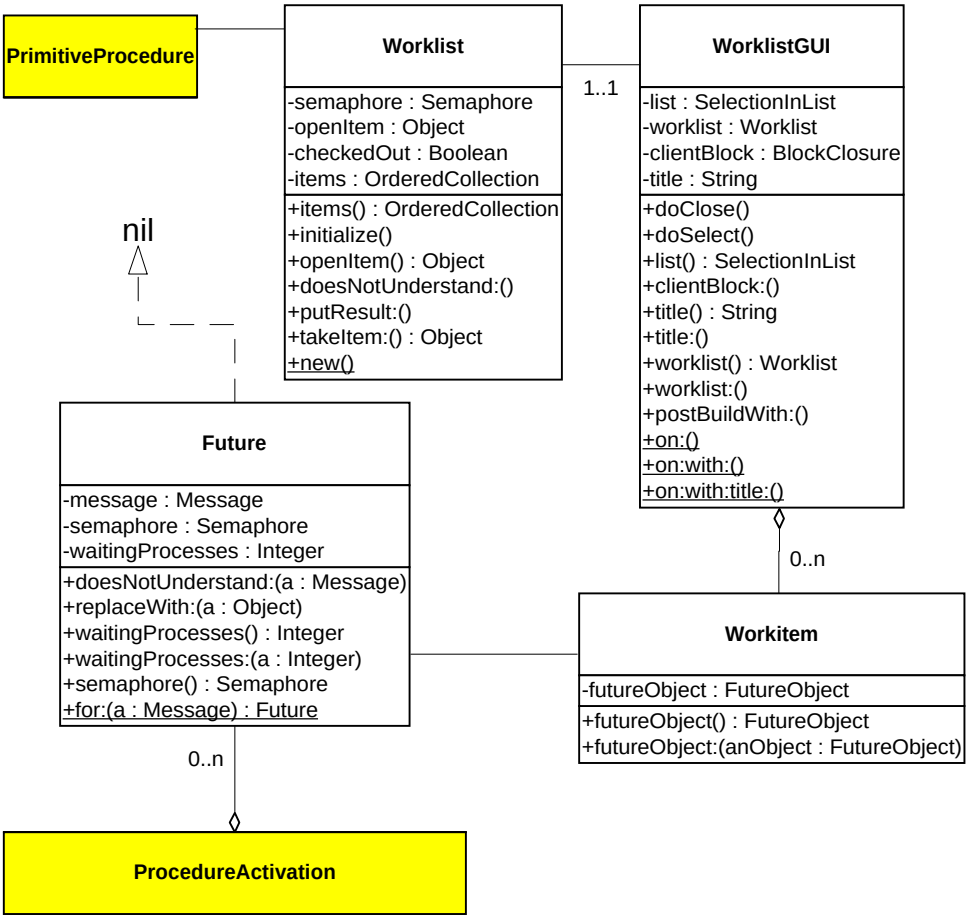


Figure 5.29: The worklist component, UML class diagram. The colored/shaded classes belong to other framework components.

workflow domain. Figure 5.30 shows the worklist of a nurse participating in the strep throat process.

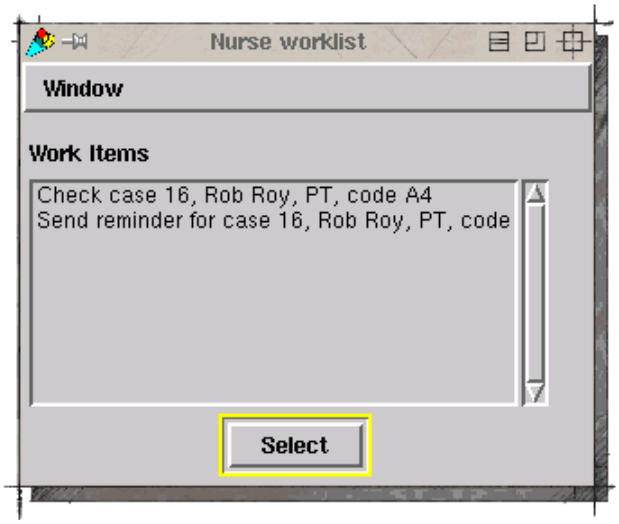


Figure 5.30: Worklist GUI.

Asynchronous Invocation Mechanism

Instances of the Worklist class replace application objects. Therefore, from a primitive procedure's perspective, the worklist and the application object should respond to the same messages (in dynamically-typed systems), or should implement the same interface (in statically-typed systems). There are several ways to achieve this plug-compatibility.

For example, workflow systems that adopt a compiler-like approach (e.g., the METEOR$_2$ system discussed in Section 2.5.3, or the Mentor system [87]) can use code generation. The workflow build time reads the application object interface and generates stubs that serve as a starting point for building worklists. Other solutions for presenting the same interface can use *Proxy* or *Facade* objects [39]. However, all these solutions involve working with (i.e., generating or writing) source code.

Building worklist objects that replace application objects transparently boils down to being able to trap the messages sent from the workflow domain to the domain objects. I chose a solution that avoids source code manipulation by leveraging the reflective facilities of Smalltalk. Worklist uses the Smalltalk doesNotUnderstand: mechanism to intercept messages sent by primitive procedures. The worklist queues the messages it receives and a human worker processes the corresponding work item asynchronously.

This scheme works fine for messages with no return values. However, it doesn't suffice if the workflow domain expects a return value since the user domain processing takes place asynchronously. The worklist solves this problem by returning a Future object.

Future objects provide placeholders for objects whose identity is determined after the future object representing them is created [50]. Other domains that deal with asynchronous components (e.g., distributed computation) have successfully used this technique [1]. The workflow framework passes around Future instances like regular application domain objects. However, as soon as the human worker finishes processing the corresponding work item, the framework substitutes the Future instance with the object returned by the worker. Another Smalltalk reflective facility provides this mechanism. The virtual machine replaces the receiver of the oneWayBecome: message with its argument such that all references to the receiver (i.e., future object) point now to the argument (i.e., user domain object). Figure 5.31 shows the key parts of the asynchronous invocation mechanism implemented by the Worklist and Future classes.

```
doesNotUnderstand: aMessage
    | future |
    future := FutureObject for: aMessage.
    items add: future.
    self changed: #items.
    ^future

Future>>replaceWith: anObject
    | sem signalsToSend |
    sem := self semaphore.
    signalsToSend := self waitingProcesses.
    self oneWayBecome: anObject.
    signalsToSend timesRepeat: [sem signal]
```

Figure 5.31: Several Smalltalk–80 reflective facilities provide the foundation of the worklist component. Worklist uses doesNotUnderstand: to trap the messages sent from the workflow domain and convert them into work items. Once the human worker processes the work item, Future sends the oneWayBecome: message to replace itself with the real result.

Johnson and Foote [34] describe a similar implementation of future objects. However, they use become: instead of oneWayBecome:. I chose a slightly different implementation because the worklist component shouldn't restrict the type of objects returned from the user domain. The mechanism described in Figure 5.31 must be able to replace future objects with any other Smalltalk object. For example, the activity assigned

to the human worker may return one of the single instance objects like true or false. Replacing one of these objects with a future object (i.e., the effect of sending become:) breaks other classes from the Smalltalk library.

The above mechanism maintains the appearance of synchronous message sends. But what happens if the workflow attempts to use a Future instance before it has been replaced with the object returned from the user? Message sends to future objects indicate that the workflow has reached a point where it involves objects which haven't been returned yet from the application domain. Therefore, in response to message sends Future instances should suspend the execution within the workflow domain until the application-domain object becomes available. I implemented this mechanism by subclassing Future from nil. In effect, this ensures that future objects don't inherit any behavior and the programmer decides all the messages they understand. Figure 5.32 shows the definition of the Future class and how it implements doesNotUnderstand:.

```
nil subclass: #Future
    instanceVariableNames: 'semaphore waitingProcesses message '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Workflow-Worklist'

Future>>doesNotUnderstand: aMessage
    self waitingProcesses: self waitingProcesses + 1.
    self semaphore wait.
    self perform: aMessage selector withArguments: aMessage arguments
```

Figure 5.32: Future doesn't inherit any behavior and therefore all messages it doesn't implement generate a doesNotUnderstand:.

Future objects deal with asynchronous processing in a manner that is transparent for the execution component. However, sometimes the process may require the completion of all work items before it can continue. This corresponds to a synchronization point between the worklist and execution components. For example, the framework shouldn't return the activation corresponding to the last procedure of a workflow unless all workers involved in the process have finished the tasks assigned to them. A simple way to check this condition is to see whether the context contains any future objects. The ProcedureActivation class of the execution component provides a message that implements this functionality. Figure 5.33 shows the implementation of this message.

```
ProcedureActivation>>waitUntilNoFutures

    context associationsDo: [:each | each value isNil]
```

Figure 5.33: Sending isNil to a Future instance suspends execution until the worklist component returns the corresponding domain object from the application domain.

5.5.3 Discussion of the Worklist Component

The micro-workflow architecture relies on a separate component to provide the human-computer interface and the asynchronous invocation mechanism required to support human workers. In my framework this functionality is implemented by the worklist component. The design decouples the workflow core from the issues of human-computer interaction, supporting the idea of extending the core functionality through composition. It also relieves the workflow core from the job of figuring out how to reach its users and send out their workitems. Several research projects adopt similar solutions. Section 2.5.1 has described how TriGS$_{flow}$ implements worklist management with ECA rules [111]. Mentor-lite implements this functionality as a workflow on top of a lightweight kernel [131]. Dayal and colleagues [22] use a separate worklist component for supporting open clients in their workflow architecture.

Micro-workflow allows developers to change the way it supports human workers. They can use a different invocation mechanism and/or user interface by plugging in a different worklist component. In their book on production workflow [72] Leymann and Roller conclude that "a workflow management system must provide a mechanism to allow a user to realize all kinds of invocation mechanisms." By implementing the functionality required to support human workers (human-computer interface and asynchronous invocation mechanism) in a separate component that developers can plug in, micro-workflow subscribes to their conclusion.

Notice that in addition to supporting human workers, the worklist mechanism allows for other workflow features that require asynchronous invocation. For example, disconnected operation lets workflow workers remove their laptop computers from the network while they process their worklists [83].

Leymann and Roller [72] discuss an implementation of worklist functionality based on Message Oriented Middleware (MOM). The MQSeries Workflow system adopts this solution and uses IBM's MQSeries MOM product. Alonso and colleagues [3] chose a similar implementation for the IBM's Exotica/FMQM

project. In contrast, my solution uses classes from the Smalltalk class library and leverages some of the reflective facilities available in Smalltalk–80 [34]. Software developers can tailor it to use persistent queues or any other mechanism appropriate to their application.

Some workflow models have an organizational dimension. This means that the part of the system implementing worklists also handles staff resolution, in addition to the invocation mechanism and interfaces for human workers. For example, the models described by Jablonski and Bussler [62], and Leymann and Roller [72] fall into this category. However, micro-workflow regards the organizational issues as beyond the responsibilities of a workflow system. Instead, this functionality should be provided by specialized directory services (e.g., Innosoft IDS and IDDS [61]), independent of the workflow system. This philosophy keeps the architecture lightweight, which is one of the main goals of micro-workflow. Consequently, the issues of staff resolution should remain outside the worklist component described in this section.

Table 5.5 summarizes several potential ways of customizing the worklist component.

Aspect	**Details**
Displaying work items	Subclass the Workitem class
Staff resolution	Extend Worklist to use a directory service

Table 5.5: Customizing the worklist component.

5.6 Federated Workflow

Some processes contain subprocesses that execute at different locations, on different workflow management systems.

5.6.1 Context

The Newborn Screening program at the Illinois Department of Public Health (IDPH) involves performing several tests on blood specimens collected from newborns. Hospitals throughout the state send dry blood samples collected on a special filter paper to a Chicago lab for testing. Public health employees at statewide locations track the test results for each baby and start a followup process whenever the lab reports an abnormal condition. The followup process requires collecting a new blood specimen and running the lab tests for a second time. If the second test confirms the problem, IDPH refers the case to a specialist. Therefore,

the followup process involves several parties: the IDPH personnel (who triggers and monitors the process), the baby's physician (who collects the blood specimen), the Chicago lab (who performs the tests), and the specialist (who treats the baby).

Several characteristics of the followup process make it a good target for workflow automation. First, it involves both humans and automated systems. Second, changes in the legislation affect the required tests, as well as how the state employees handle the followup cases. Additionally, the testing procedures used at the lab also change. For example, acquiring new test equipment requires modifying the operating procedures. The public health employees must be able to easily reflect this type of changes in the process definition. Third, the state law requires all actions to be recorded for auditing, liability, and statistical purposes. Fourth, since human life is at risk, authorized personnel monitors the process evolution. Finally, sometimes information needs to be updated after the followup starts, thus requiring the staff to alter process execution at runtime. For example, amendments of the gestational age influence the interpretation of 17–OH progesterone levels measured for the detection of congenital adrenal hyperplasia.

Can developers use the micro-workflow framework to build an application that implements the newborn followup process? A characteristic that sets this process apart from the strep throat process described in Chapter 2 is the fact that its participants are geographically distributed. The IDPH staff and physicians from across the state work together with the Chicago-based the lab facility. This means that the worklist and domain objects involved in followup processes reside at *different* locations. Distributed middleware (e.g., CORBA, DCOM, or JAVA remote method invocation) provides the infrastructure required to handle object distribution, making the physical location of objects transparent [97]. But is transparent access enough?

5.6.2 Problem

The micro-workflow components described in the previous sections assume that the entire process executes at one centralized location. Therefore, the developers would have to decide where to install the followup workflow: at the Chicago lab, or at the statewide offices?

Installing it at the lab yields a solution that *centralizes the entire processing of all followup cases to one location*. In this configuration, the domain objects representing the lab equipment and the lab staff worklists reside in the same address space as the micro-workflow. In contrast, the IDPH field offices access the workflow through distributed middleware. Figure 5.34 sketches this configuration. There are several

problems with this approach. First, a centralized solution also means a single point of failure. Field office employees can't monitor their processes if the lab site system becomes unavailable. Second, having one site deal with the cases for the entire state doesn't scale. For example, in this configuration the centralized execution site handles the preconditions associated with *all* followup processes. Third, the access to remote objects provided by the distributed middleware requires network connections with high availability. Otherwise, connectivity problems may prevent field offices to check the status of the cases they monitor. But high availability increases the cost. And finally, public health employees who need to access the workflow (e.g., to check the progress of a case) require remote access to the lab system. This complicates maintenance and may also run into scalability problems.

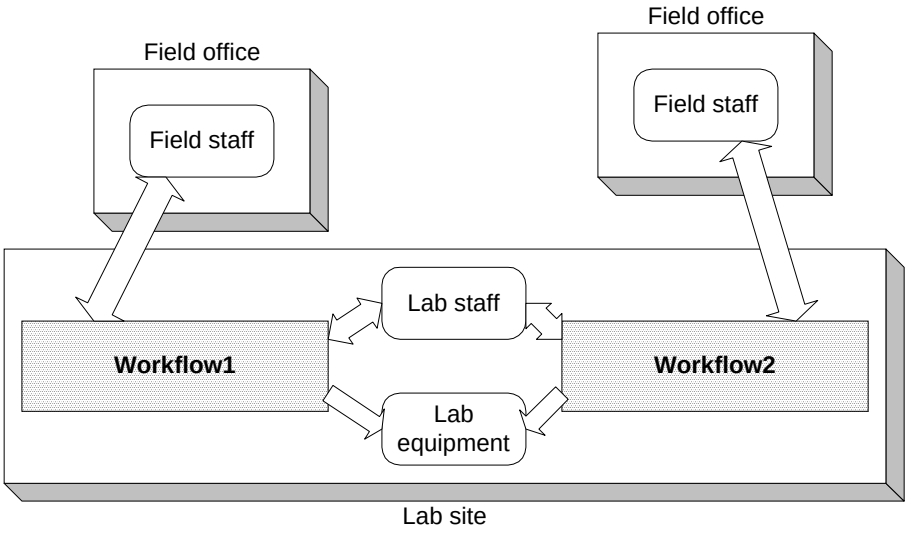


Figure 5.34: Followup workflow residing at the lab site. For clarity, this figure shows only two field offices.

Alternatively, the developers can install the workflow at the public health offices. In this case, the worklists of the public health employees reside within the same address space as the micro-workflow, while the lab worklist and domain objects communicate with the workflow through distributed middleware. The followup processing *remains centralized into several locations which handle different cases*. Figure 5.35 shows this solution. Although it distributes the processing of followup cases among field offices, this configuration still has some problems. First, running followup processes still needs a highly available network to communicate with the lab objects. Second, since each process requires remote connections while the lab performs the tests (1–2 days), the network and the distributed middleware continue to have scalability problems. Third, the lab staff and equipment can interact with the workflow only through remote systems. For

example, any updates that the lab staff might have need to go through the field office. Finally, this solution exposes the entire lab process to field staff who probably doesn't understand it. Who updates the process definition when the lab people change the testing procedure?

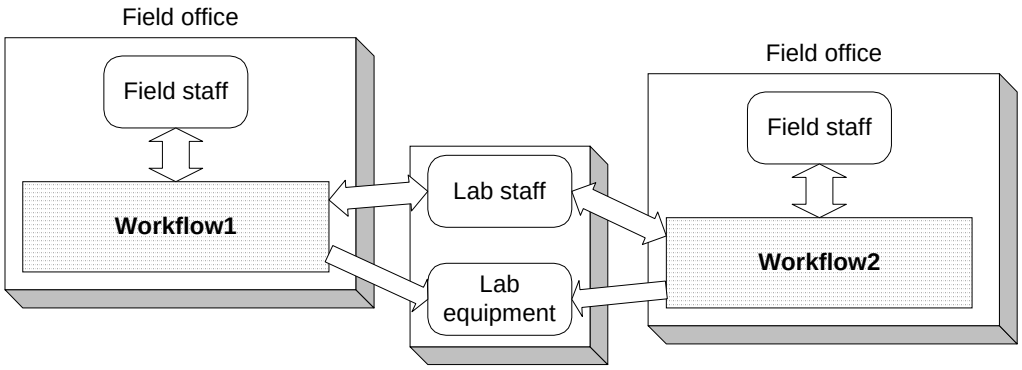


Figure 5.35: Followup workflow residing at the field offices. For clarity, this figure shows only two field offices.

The problems of the two solutions described above make them unusable in a real setting, where scalability and availability issues shouldn't be ignored. These problems stem from the fact that the workflow corresponding to each followup case *comprises the entire process* and *executes in a centralized manner*. Many researchers have identified the centralized model typical of current workflow architectures as one of their limitations. For example, in their analysis of contemporary workflow systems, Alonso and colleagues [2] conclude:

> An architecture based on a centralized server is vulnerable to server failures and offers limited scalability due to the potential performance bottlenecks caused by centralized servers.

Besides technical problems, the previous solutions also have administrative problems. Centralized workflow execution may intermix concerns that belong to different realms. In the case of the followup process, mixing lab and field office procedures causes no administrative conflicts since these sites belong to the same organization. But the intermixing becomes a thorny problem with processes distributed across organizational boundaries. As Geppert and colleagues [46] point out (and as I've also discovered while looking for examples of real processes), workflow specifications represent a strategic asset that enterprises don't disclose to outsiders.

Therefore, a workflow architecture based on centralized execution has both technical and administrative problems with distributed processes. What type of architecture can deal with this type of process?

Developers can avoid the problems of centralized execution by breaking the workflow into separate pieces, each of which executes at different locations. In effect, this corresponds to distributing *workflow execution* among several sites who work together towards a common goal. Alonso and colleagues [2] identified the distributed execution of workflow processes as a "very interesting research area" that can enhance workflow systems. Subsequent workflow research refers to this type of cooperation between different workflow systems as *federated workflow*.

Federated workflow involves the integration of several workflow management systems into a global workflow. A survey of current workflow products finds that "existing systems are almost totally incompatible" [2]. Consequently, workflow interoperability in the context of federations of heterogeneous workflow systems is a hard problem. This issue represents one of the main barriers in the way of federated workflow. Current research efforts focus on finding architectural solutions that solve this problem [46].

Federated workflow also involves sharing information among the participating workflows. In the context of multi-organizational federated workflows (which are quite common with the increasing popularity of outsourcing), the inter-workflow data flow represents an important concern. Since process data usually represents classified information, organizations share it following strict policies. Participating workflows must release and receive only the information required for their execution, as specified in the process definition [107].

The intrinsic distribution of the Newborn Screening Followup Process described in Section 5.6.1 makes it a good candidate for federated workflow. The management of the case can execute at the field office that starts the process. Likewise, the lab test subworkflow can execute at the lab site. Unlike the centralized configurations from Figures 5.34– 5.35, this arrangement decouples the processing at each site, distributing it among several workflow systems. Additionally it lowers the duration of inter-workflow communication. Figure 5.36 shows this solution.

However, federated workflow brings in additional concerns that don't exist when all objects reside within the same address space. For example, I have already mentioned the integration of potentially heterogeneous workflow systems, and the inter-organizational data sharing. So far the framework components assume that the entire workflow executes within a single address space, and that all workflow data is local. Therefore,

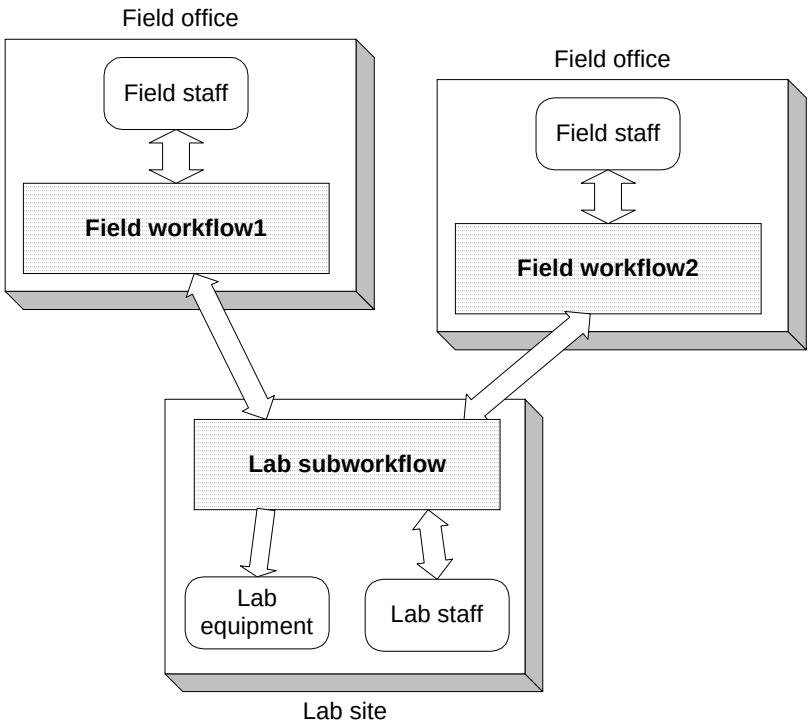


Figure 5.36: Federated followup workflow. For simplicity this diagram shows only two field offices.

they ignore workflow integration and data sharing. But implementing a distributed process with a workflow system that ignores these issues (i.e., it centralizes execution) has the types of problems described above. The reminder of this section revisits these concerns in the context of micro-workflow and extends the framework with support for federated workflow.

5.6.3 Solution

Federated workflow requires breaking up workflow execution among several workflow systems which execute parts of the process within different address spaces. Workflow decomposition into subworkflows corresponds to *hierarchical workflow*; execution within different address spaces corresponds to *distributed workflow*.

The federated workflow component extends the micro-workflow core with support for hierarchical and distributed workflow [77]. In effect, this component allows for transparent workflow distribution across the enterprise. This lets software developers depart from the centralized model of workflow execution, and implement workflows that execute parts of the process at multiple locations.

Hierarchical workflow translates into the ability to have other workflow systems implement activity nodes of the process definition (see Figure 5.37). I introduce two new abstractions for this purpose. Workflow represents a workflow system. SubworkflowProcedure is a procedure type that executes another Workflow—i.e., a subworkflow. When the execution component fires off a SubworkflowProcedure, this transfers control to its subworkflow. The transfer of control between the parent workflow and the subworkflow involves data flow. Typically the parent workflow provides data that the subworkflow executes on. For example, in the IDPH Newborn Followup Process this represents the blood sample. After the subworkflow completes execution, its results become available to the parent workflow. For example, in the lab process this data represents the test result.

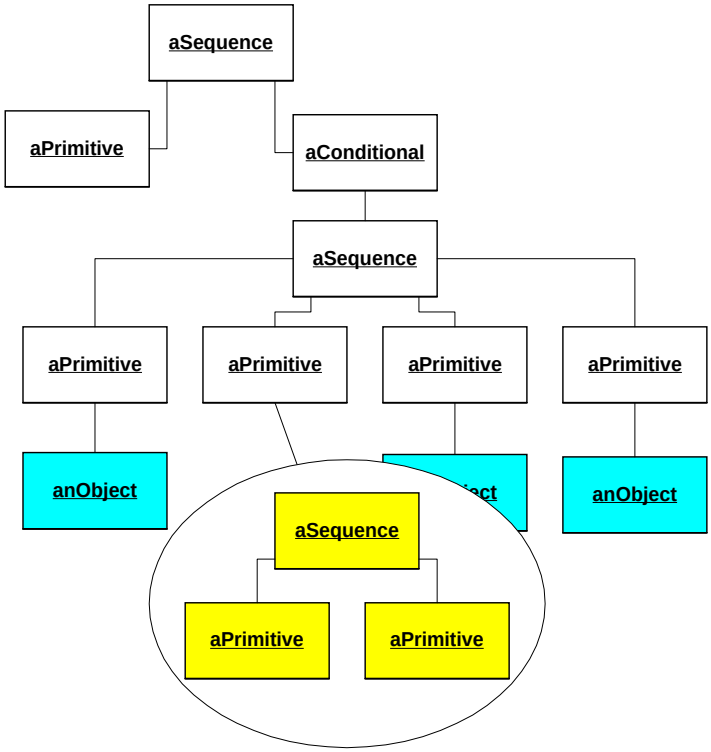


Figure 5.37: Instance diagram showing hierarchical workflow. The subworkflow is shown in yellow/light gray.

However, the Workflow and SubworkflowProcedure deal only with hierarchical workflow. They assume that the parent workflow and the subworkflow reside within the same address space. Distributed workflow adds the ability to execute workflows regardless of their location. I introduce a WorkflowFacade that represents an abstraction for remote workflow systems. The facade handles remote workflow execution and data

transfer over the network. WorkflowFacade clients interact with it through an IDL-style (i.e., message name and parameters) interface. WorkflowFacade doesn't expose any details about the workflow system behind the facade, thus facilitating the integration (through white-box techniques) of potentially heterogeneous systems. For example, the facade can encapsulate either object-oriented or legacy workflow systems (the latter wrapped to present an object-like interface). In effect, this abstraction completes the functionality required for federated workflow.

Because I couldn't obtain a commercial workflow system for this research, the prototype of the federated workflow component assumes a homogeneous federation. However, since these abstractions aim at offering a consistent view *above* the WorkflowFacade (i.e., the facade decouples clients and providers), this assumption *doesn't affect* the framework—the glue code connecting a workflow system to a facade is not reusable anyways. Additionally, having the same system implement both the workflow and the subworkflow reveals the issues that have to be dealt with at both ends.

5.6.4 Usage

The federated workflow component distributes the concerns of hierarchical workflow, inter-workflow data flow, and distributed workflow among several classes.

The Workflow class represents a workflow system. Developers provide the process definition, and the code that initializes the process context. Instances of this class hold the objects required for workflow execution (e.g., process activity map and precondition manager). But unless software developers need to customize these objects, they don't interact directly with them. The Workflow class provides a process execution template that orchestrates workflow execution. It also manages the other framework components (history, persistence) or their user interfaces (monitoring, manual intervention, and worklists). The template starts the runtime services required to execute a workflow, fires off the process, and shuts down the runtime services after the process finishes execution. Developers control what micro-workflow components they use by tailoring the process execution template.

SubworkflowProcedure provides an abstraction for a subworkflow. It enables developers to use an entire workflow (i.e., instance of Workflow) as an activity node in the process definition. SubworkflowProcedure handles the data flow between the two workflows, and the execution of the subworkflow. Data flow takes place as specified in the processes definition. The transfer of control from the workflow to the subworkflow

exposes only the objects needed by the subworkflow. Likewise, the return of control passes only the objects required by the workflow.

Usually the data flow between two different address/name spaces involves translation. For example, when both workflow systems use contexts with named slots, processing may require mapping between the different slots. Figure 5.38 illustrates the data flow between a SubworklfowProcedure and the subworkflow associated with it. The input involves mapping the contents of the workflow's slot "4" into the subworkflow's slot "foo" (since the subworkflow expects to find at "foo" the object that the workflow stores at the slot "4") along with slots "1" and "2" which don't require mapping. When the subworkflow completes, its output slots "AA" and "CC" map into the workflow's "B" and "A." Consequently, SubworkflowProcedure requires developers to program the data flow between the workflow name space and the subworkflow name space. Instances of the BidirectionalMapper class serve this purpose.

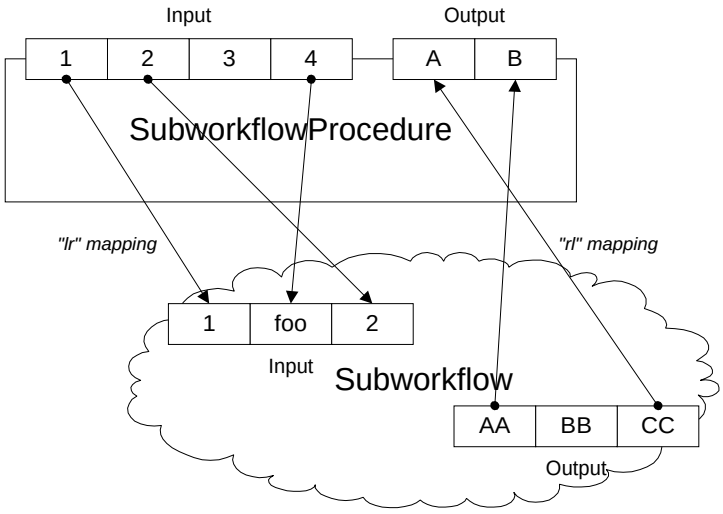


Figure 5.38: Data flow between a SubworkflowProcedure and a subworkflow.

The Workflow and SubworkflowProcedure classes add support for *hierarchical workflow*. Hierarchical workflow serves a different purpose than the hierarchical decomposition achieved through the properties of the *Composite* [39] pattern. The latter concerns process *definition* (subprocesses), while the former concerns process *execution*. Subprocesses break down a process definition into pieces that may be reused by other processes. Process designers reuse process fragments (subprocesses) from a process library—the MIT Process Handbook project proposes an on-line repository of organizational processes [106]; Steinar Carlsen also discusses the reuse of workflow model fragments in his PhD thesis [16]. However, at run

time the process and its subprocesses execute inside the *same workflow system* and therefore share the runtime mechanism (e.g., precondition manager, etc.). For example, at several stages of the followup process phone conversations are confirmed with written letters. This common sequence of actions (e.g., phone call followed by printing and sending a form letter) may be already available in a process repository. In contrast, hierarchical workflow splits execution among separate workflow systems. One workflow system relies on one or several other workflow systems to execute activities within the process definition. These activities correspond to subprocesses, but appear as atomic/primitive activities to the invoking workflow. The workflow systems involved in the execution of the entire process are independent. Therefore, they can execute in different address spaces, at different locations.

The WorkflowFacade class adds support for *distributed workflow* by leveraging the infrastructure provided by the VisualWorks Opentalk [20] distributed application environment. This enables a workflow running within one address space (i.e., the server) to execute a subworkflow within a different address space (i.e., the client):

- On the server, a WorkflowFacade instance represents a *Proxy* [39] for a remote workflow. Instances of this class replace Workflow instances transparently.
- On the client, another WorkflowFacade instance accepts requests from remote facades and executes workflows on their behalf.

Each WorkflowFacade instance uses the Opentalk naming service to obtain a reference to its peer. Therefore, prior to workflow execution the naming service should be running and have the facades registered with it.

Software developers specify that a SubworkflowProcedure triggers the execution of a remote workflow by building the procedure on a WorkflowFacade instance instead of a Workflow instance. The code fragment from Figure 5.39 illustrates the difference between building a SubworkflowProcedure that executes a local subworkflow, and building one that executes a remote workflow.

By default, Opentalk uses the CORBA 2.0 pass by reference mechanism [97] between Smalltalk images. Sending messages to objects passed by reference incurs overhead and takes longer since Opentalk transfers the messages to the virtual machine where the object resides. Sometimes applications can't afford this impedance mismatch between local and remote message sends and therefore require passing objects by value (i.e., migrate the object from an address space to another) instead of by reference. To accommodate

```
localLabScreeningSubprocess

    | mapper |
    mapper := BidirectionalMapper new.
    mapper rlMap: #testResult to: #screening2.
    ^(SubworkflowProcedure on: LabScreening new)
        mapper: mapper;
        yourself

remoteLabScreeningSubprocess

    | mapper |
    mapper := BidirectionalMapper new.
    mapper rlMap: #testResult to: #screening2.
    ^(SubworkflowProcedure on: WorkflowFacade instance)
        mapper: mapper;
        yourself
```

Figure 5.39: Building a procedure that fires off a local and a remote workflow.

this situation, Opentalk also provides a mechanism that implements pass by value.

The federated workflow component knows how to use the Opentalk pass by value mechanism. Programmers can select which workflow context objects should be passed by value. This requires creating a subclass of the Opentalk Shadow class for each class whose instances should be passed by value. Subsequently, Opentalk marshals and unmarshals instances of these Shadow subclasses. Each subclass provides the instance variables of the class it mirrors, along with accessors and mutators. The micro-workflow framework hooks workflow objects and their shadows to the Opentalk marshaling and unmarshaling mechanism through the asShadow and unshadow messages, respectively. Therefore, each class that requires its instances passed by value should implement asShadow, and its shadow class should implement unshadow. For example, Figures 5.40–5.41 show the type of code required by the federated workflow component to pass by value a blood test result object. Figure 5.40 shows the definition of the BloodTestResult class and sketches how the domain object packs its state into a BloodTestResultShadow instance in response to the asShadow message. Likewise, Figure 5.41 shows the definition of the BloodTestResultShadow class and how it reconstructs a BloodTestResult instance in response to the unshadow message.

```
Smalltalk defineClass: #BloodTestResult
	superclass: #Core.Object
	indexedType: #none
	private: false
	instanceVariableNames: 'type abnormal certification number '
	classInstanceVariableNames: ''
	imports: ''
	category: 'Workflow-Example Followup'

BloodTestResult>>asShadow
	^(BloodTestResultShadow new)
		type: type;
		abnormal: abnormal;
		certification: certification;
		number: number;
		yourself
```

Figure 5.40: Opentalk pass by value, domain object side (VisualWorks 5i-style class definition).

```
Smalltalk defineClass: #BloodTestResultShadow
	superclass: #Opentalk.Shadow
	indexedType: #none
	private: false
	instanceVariableNames: 'type abnormal certification number '
	classInstanceVariableNames: ''
	imports: ''
	category: 'Workflow-Example Followup'

BloodTestResultShadow>>unshadow
	^BloodTestResult fromShadow: self
```

Figure 5.41: Opentalk pass by value, shadow side (VisualWorks 5i-style class definition).

5.6.5 Design Details

The federated workflow component adds support for hierarchical and distributed workflow. This involves subworkflow execution, inter-workflow data flow (i.e., context mapping), and distribution. Figure 5.42 shows the UML class diagram corresponding to this component. The reminder of this section discusses these classes.

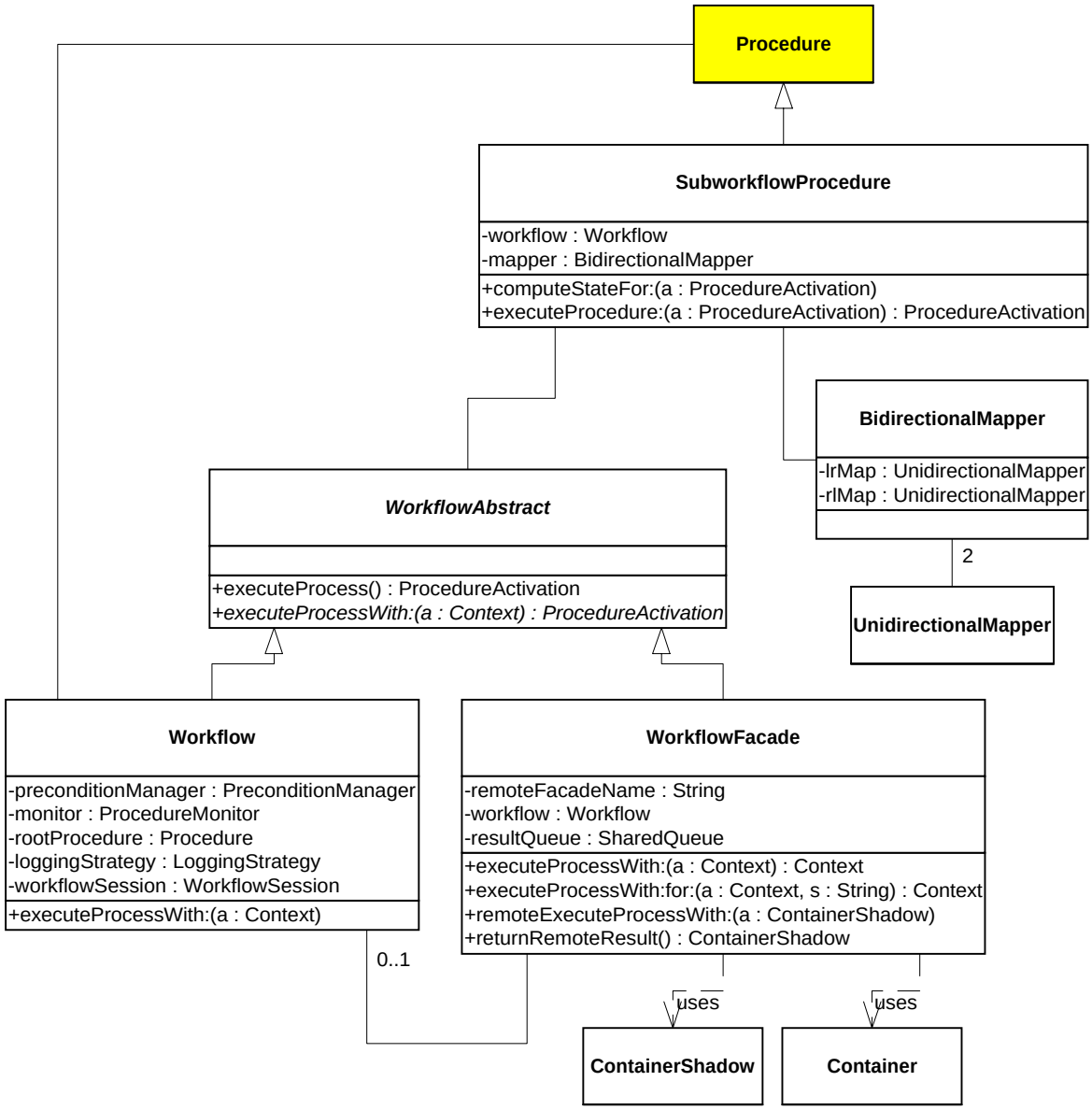


Figure 5.42: Federated workflow component, UML class diagram. The colored/shaded classes belong to other framework components.

Subworkflow Execution

SubworkflowProcedure is a concrete subclass of Procedure. It implements the execution interface defined by the Procedure abstract class and discussed in Section 4.3). Through polymorphism, these messages specify how each procedure type computes its run time information (computeStateFor:) and how it executes (executeProcedure:).

SubworkflowProcedure doesn't have local state and therefore computeStateFor: doesn't involve any computation. executeProcedure: handles the data flow between the workflow and the subworkflow. This procedure type delegates the mapping of the context slots illustrated in Figure 5.38 to a BidirectionalMapper instance. Figure 5.43 shows how SubworkflowProcedure responds to the executeProcedure: message.

```
executeProcedure: anActivation

    | inputContext outputContext activation |
    inputContext := mapper lrMapFrom: anActivation context.
    outputContext := workflow executeProcessWith: inputContext.
    activation := anActivation copy.
    activation forwardDataFlowFrom: (mapper rlMapFrom: outputContext).
    ^self return: activation
```

Figure 5.43: The subworkflow procedure.

SubworkflowProcedure executes workflows through the interface defined by the WorkflowAbstract class. WorkflowAbstract subclasses implement the executeProcessWith: message to execute the subprocess within the same Smalltalk image (Workflow), or in a remote Smalltalk image (WorkflowFacade).

Context Mapping

Two classes provide the mechanism for the inter-workflow data flow and mapping between the context of the calling workflow and the context of the called subworkflow.

UnidirectionalMapper maps into one direction, from a source context to a destination context. Developers program the mapping by sending directMap: and map:to: messages to an instance. The former specifies the same slot name in the source as well as the destination, while the latter connects slots with different names. The mapper performs the mapping in response to mapFrom:. This message takes the source context as the argument and returns the destination context. Source context slots that shouldn't appear in the destination

context don't have a mapping (i.e., through directMap: or map:to:).

BidirectionalMapper aggregates two UnidirectionalMapper instances, one for each direction. It implements the same interface as UnidirectionalMapper but each message has an additional prefix which determines the direction. Messages prefixed with "lr" (e.g., lrDirectMap:, lrMap:to:, and lrMapFrom:) affect the mapping from the workflow to the subworkflow. Likewise, messages with the "rl" prefix affect the mapping from the subworkflow back to the workflow (e.g., rlDirectMap:, rlMap:to:, and rlMapFrom:) The two mappings are annotated in Figure 5.38, and Figure 5.44 shows the Smalltalk code that configures the bidirectional mapping.

```
| mapper |
    mapper := BidirectionalMapper new.
    mapper
      rlDirectMap: #1;
      rlDirectMap: #2;
      rlMap: #4 to: #foo;
      lrMap: #AA to: #B;
      lrMap: #CC to: #A
```

Figure 5.44: Configuring the inter-workflow mapping.

Distribution

The federated workflow component uses the CORBA-based distributed application architecture provided by VisualWorks Opentalk [20]. The STST Opentalk framework enables transparent communication between Smalltalk virtual machines through Object Request Brokers (ORBs).

WorkflowFacade is a concrete subclass of WorkflowAbstract (Figure 5.42). It responds to the executeProcessWith: message (Figure 5.39) by firing off the subworkflow. This involves sending the remoteExecuteProcessWith: message to a facade residing in a different Smalltalk image. Next the local facade obtains the results of the subworkflow from a shared queue. Reading from this queue blocks execution until the remote workflow completes and sends back its output through the returnRemoteResult: message. In effect, the queue provides a synchronization point between the workflow and the subworkflow. The framework can also accommodate asynchronous subworkflow invocation using future objects, through a mechanism similar to the one used by the worklist component (Section 5.5). Figure 5.45 shows the implementation of these

messages and illustrates that the STST framework makes remote message invocation (sending remoteExecuteProcessWith:) transparent.

```
WorkflowFacade>>executeProcessWith: aContext
    | outgoingContainer containerShadow incomingContainer |

    outgoingContainer := Container fromContext: aContext.
    outgoingContainer callerName: self name.
    self remoteFacade remoteExecuteProcessWith: outgoingContainer asShadow.
    containerShadow := resultQueue next.
    incomingContainer := Container fromShadow: containerShadow.
    ^incomingContainer context

WorkflowFacade>>returnRemoteResult: aContainerShadow
    resultQueue nextPut: aContainerShadow
```

Figure 5.45: Subworkflow execution, server side.

The client facade executes the workflow in response to the remoteExecuteProcessWith: message. However, this message shouldn't wait for the return value since typically workflow execution takes much longer than message sends. Opentalk expects objects to process message sends synchronously, in a timely manner. To compensate for the impedance mismatch between workflow and object time scales, the facade executes the workflow in a separate thread, through the executeProcessWith:for: message. This mechanism allows the remoteExecuteProcessWith: message to return control immediately. Additionally, combined with the pass by value mechanism, it requires only *temporary connections* between the server and the client—once to fire off the subworkflow, and once to transfer back its results. This characteristic has several consequences. First, it enables disconnected workflow execution, an important feature that is missing from many current workflow systems [2, 103, 104, 98]. For example, the subworkflow can run on a laptop computer which is removed from the network once the subworkflow starts execution. Second, as Hagen observes in his PhD thesis [49], this type of architecture facilitates maintenance tasks. While the subworkflow executes on the client, workflow execution on the server can be suspended for maintenance and updates.

executeProcessWith:for: starts the execution of the local workflow, adding the objects passed by the server (i.e., the arguments) to its context. Once execution completes, it resolves the name of the caller and sends it the returnRemoteResult: message. This transfers back the results through the container mechanism provided by the federated workflow component. Figure 5.46 shows the implementation of these messages.

```
WorkflowFacade>>remoteExecuteProcessWith: aContainerShadow
remoteExecuteProcessWith: aContainerShadow
    | incomingContainer |
    incomingContainer := Container fromShadow: aContainerShadow.
    [self
        executeProcessWith: incomingContainer context
        for: incomingContainer callerName] fork

WorkflowFacade>>executeProcessWith: aContext for: aName
    | result outgoingContainer serverFacade |
    result := workflow executeProcessWith: aContext.
    outgoingContainer := Container fromContext: result.
    serverFacade := NameServiceRoot default resolveLeaf: aName.
    serverFacade returnRemoteResult: outgoingContainer asShadow
```

Figure 5.46: Subworkflow execution, client side.

Instances of the WorkflowFacade class use the Opentalk naming service to find their peers. At run time, the facades resolve the name of their peers with the naming service, which returns references to registered objects. For example, the server-side facade uses the name service to obtain the target for the remoteExecuteProcessWith: message (Figure 5.45). Likewise, once the subworkflow completes execution, the client-side facade resolves the receiver of the returnRemoteResult: message (Figure 5.46).

The UML sequence diagram from Figure 5.47 shows how the two WorkflowFacade instances orchestrate the remote workflow execution. The STST Opentalk framework makes sending the remoteExecuteProcess-With: and returnRemoteResult: messages across Smalltalk virtual machines transparent.

Finally, the Container and ContainerShadow classes handle the passing of objects between Smalltalk virtual machines. They use the mechanisms provided by Opentalk to pass objects either by value or by reference. Developers don't interact directly with these classes. The federated workflow component uses them internally to implement pass by value. However, for each class whose instances should be passed by value, Container and ContainerShadow require a corresponding shadow class and the asShadow message, as illustrated in Figures 5.41 and 5.40. Figure 5.48 shows how the Container class sends the asShadow and unshadow messages to domain objects and their shadows.

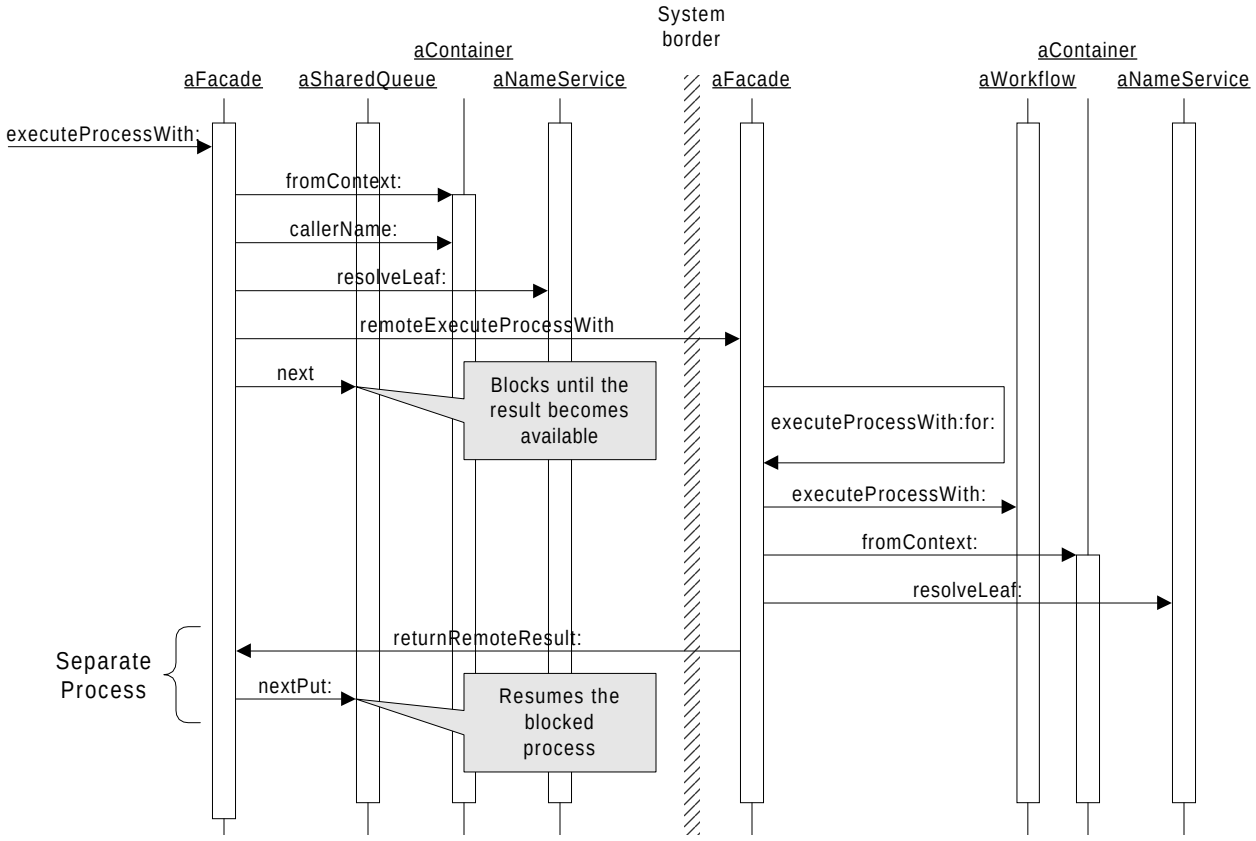


Figure 5.47: Remote workflow execution, UML sequence diagram. This figure shows synchronous workflow execution, when the parent workflow waits until the subworkflow completes execution.

```
Container>>asShadow
    | shadowContext |
    shadowContext := IdentityDictionary new.
    context keysAndValuesDo: [:key :value | shadowContext at: key put:
      ((value respondsTo: #asShadow)
        ifTrue: [value asShadow]
        ifFalse: [value])].
    ^(ContainerShadow new) context: shadowContext; yourself

Container>>initializeFromDictionary: aDictionary
    aDictionary keysAndValuesDo: [:key :value | self at: key put:
      ((value respondsTo: #unshadow)
        ifTrue: [value unshadow]
        ifFalse: [value])]
```

Figure 5.48: The Container class uses the Opentalk shadow mechanism to pass objects by value.

5.6.6 Discussion of the Federated Workflow Component

The federated workflow component extends the micro-workflow core with support for distributing workflow execution among several workflow systems. Currently most workflow systems don't provide this feature.

The Workflow and SubworkflowProcedure classes add support for hierarchical workflow, allowing developers to represent a workflow as a node of the activity map. Thus they provide the functionality of a primitive that fires off a workflow instead of sending a message to a domain object (see Figure 5.37). The BidirectionalMapper class adds the inter-workflow data flow and handles the translation (i.e., mapping) between different address spaces. The WorkflowFacade class hides the details about the remote invocation of other workflow systems behind an invariant interface, thus facilitating the integration of heterogeneous systems. As an example, the federated workflow component uses the CORBA-based Opentalk distributed application architecture to accommodate the communication of workflows running within different address spaces.

The design of the federated workflow component is consistent with the other framework components. SubworkflowProcedure binds activity nodes within the process activity map with workflows. The execution component doesn't require the binding prior to executing this procedure. Therefore, the binding can change at run time. For example, the workflow can dynamically complete its own definition.

Table 5.6 summarizes a few potential directions of customizing the federated workflow component.

Aspect	**Details**
Support a new workflow system	Subclass WorkflowFacade
Change the inter-workflow data flow mechanism	Replace the Container and ContainerShadow classes
Use RMI or DCOM	Use a CORBA-RMI or CORBA-DCOM bridge, or replace Opentalk

Table 5.6: Customizing the federated workflow component.

5.7 Putting It All Together

The micro-workflow components introduced in this chapter implement a wide variety of workflow features, ranging from history to federated workflow. This chapter has shown how the compositional approach

adopted by the micro-workflow architecture benefits software developers who need workflow functionality within their applications. First, the ability to add features by plugging in components enables them to tailor the workflow functionality to their requirements. Second, developers could customize each workflow feature individually, with a low impact on the other features. The compositional design localizes most changes to modify a feature to the component that implements it. Third, the architecture can grow and provide new features. Developers add new features through building new components. Finally, having a component encapsulate each feature facilitates application integration.

Table 5.7 summarizes how developers extend the micro-workflow core with the components discussed in this chapter. For example, to implement a workflow that requires only history and worklists, developers add the corresponding components to the micro-workflow core. Adding the history component involves plugging a logging strategy instance into the workflow session. Adding the worklist component involves initializing the workflow context with a Worklist instance for each workflow actor; provide the code that handles work item removal and returning results; and specializing Workitem to display the work items in a manner appropriate for the human workers.

Component	How to add to the core
History	Plug an instance of a concrete subclass of LoggingStrategy into the workflow session
Persistence	Select a logging strategy that uses a persistent store; plug in a SessionManager instance and execute the workflow within a database session
Monitoring	Register a ProcedureMonitor instance as a dependent of the workflow session
Manual intervention	Supply the workflow session with the class that implements the rewinding mechanism
Worklist	Replace domain objects within the workflow context with Worklist instances; provide the application-specific code that handles work items once they're taken off the worklist, and passes back the domain object once the user completes processing; subclass Workitem to control how the worklist displays each work item
Federated workflow	Build a process containing SubworkflowProcedure instances with WorkflowFacade instances; register the facades with the name servers; implement the Opentalk pass-by-value mechanism for objects whose instances should be passed by value

Table 5.7: Extending micro-workflow with advanced workflow features.

However, this chapter does not answer two important questions. First, does micro-workflow provide

a viable solution for applications that deal with real processes? Micro-workflow must be able to solve the types of problems that developers encounter in object-oriented applications. Second, what is the cost of the flexibility provided by this approach? If the cost is too high, software developers won't use the architecture. The answers to these questions determine whether micro-workflow provides a working solution for implementing workflows within object-oriented applications. The next chapter provides the answers.

Chapter 6

Evaluation of the Architecture

The micro-workflow architecture provides a set of abstractions that enable software developers to define and enact how the work flows through the system. Chapters 4 and 5 have shown how to build the components of the architecture, thus demonstrating that the architecture can be implemented. This chapter provides a qualitative and quantitative evaluation of the micro-workflow architecture. The qualitative evaluation involves studying whether object-oriented applications that contain processes corresponding to real problems can be implemented with micro-workflow. The quantitative evaluation uses the micro-framework presented in Chapters 4 and 5 to gather metrics that provide information about one of the key features of micro-workflow—the ability to add advanced workflow features through composition. The metrics measure the effort required to add the components described in Chapter 5, and measure how adding each component impacts the architecture. They also provide information about the run time cost of the additional flexibility.

The evaluation involves three case studies that each require additional workflow features. Each case study begins with an overview of the process, followed by the description of the application objects or the workflow actors (i.e., human workers), and the process definition. They are followed by a study of how the new micro-workflow components that each case study adds to the architecture impact the existing components. The study focuses on metrics that show how each component contributes to the framework, and the breakage caused to the core components. These metrics provide information about the *design cost*, and reflect the effort required to add workflow features through composition. The chapter concludes with a study of the *run time cost* which reflects the overhead required to accommodate pluggable components.

Finding well-documented case studies turned out to be challenging. Unlike other areas of computer science (e.g., operating systems, database systems, program verification, etc.), workflow management doesn't have a classic body of examples for studying and evaluating workflow systems. Therefore, I had to har-

vest real processes. But business processes are hard to get (i.e., they are classified information) since they represent key assets of the enterprise, and are on the critical path of staying ahead of the competition. Consequently, two of the following examples belong to projects that I was involved with while I worked on this thesis and that didn't require non-disclosure agreements.

The following sections discuss simplified versions of real processes for reviewing proposals, treating strep throat, and tracking the treatment of newborns. The simplifications keep the focus on the aspects relevant to micro-workflow.

6.1 Proposal Review Process

This application implements an administrative process at the National Center for Supercomputing Applications (NCSA). NCSA owns several supercomputers. Scientists and researchers from the industry and the academia use these supercomputers to solve problems with high computational demands—e.g., mechanical, chemical, or physical simulations. To request CPU time on an NCSA machine, potential users submit proposals to the NCSA Allocations Office. Several reviewers study each proposal and decide whether to grant the request or not. Sometimes the reviewers also adjust the total amount of CPU time requested. I studied the NCSA Allocations Process while I worked on NCSA's "Reengineering Allocation Processes and Organization" (ALPO) project.

6.1.1 Process Overview

When the process begins execution, the NCSA Allocations Office database contains proposals submitted by scientists requesting CPU time, and information about the NCSA reviewers who are going to examine the proposals. The Allocations Office staff assembles the initial reviewing assignments. At this stage they use the reviewers profiles to assign every proposal to as many reviewers as they determine suitable for the task.

The NCSA Allocations Office sends out the initial assignments to all reviewers. Each of them receives a list with all the proposals and their initial assignments. The Allocations Office staff handles any potential conflicts of interest. For example, if a reviewer is also the author of a proposal, she should not be able to see who is assigned to review her proposal.

Once the reviewers receive the initial assignments, they assemble their reviewing preferences. Each reviewer can organize the proposals in up to three categories. *Requested* proposals are the ones that the re-

viewer really wants to review. *Accepted* proposals correspond to the proposals suggested by the Allocations Office staff that the reviewer doesn't mind reviewing. *Rejected* proposals correspond to the submissions that the reviewer doesn't want to review at all. Once reviewers finish assembling their reviewing preferences, they submit them back to the NCSA Allocations Office.

The NCSA Allocations Office changes the initial assignments to take into account each reviewer's preferences. At the same time, they ensure that the proposals are evenly distributed among all reviewers, and that there are no conflicts of interest. For each proposal:

- the Allocations Office staff assigns any reviewer who requested to review the proposal and doesn't have a conflict of interest
- if the proposal doesn't already have 3 reviewers, the Allocations Office staff assigns any of the accepting reviewers without a conflict of interest
- if the proposal doesn't already have 3 reviewers, the Allocations Office staff assigns any of the initial reviewers that didn't reject the proposal
- if the proposal still doesn't already have 2 or 3 reviewers, the NCSA Allocations Office staff finalizes the assignment manually

At this point every proposal is assigned to at least two reviewers. For each proposal the NCSA Allocations Office notifies the assigned reviewers about the final assignment.

The reviewers start working on the proposals and send back reviews as they complete them. The NCSA Allocations Office takes each received review and stores it in the database. However, usually not all reviewers are prompt about sending back their reviews. Therefore, several days before the reviews are due the NCSA Allocations Office sends a reminder to all reviewers who haven't returned their reviews. Once all reviews have been received, the Allocations Office staff assembles them into a final report.

6.1.2 Domain Objects

The workflow corresponding to the NCSA Proposal Review Process involves the following domain objects:

Supervisor represents the Allocation Office staff. It has the following responsibilities: provide the initial assignments; record the reviewing preferences submitted by reviewers; finalize the assignments and

manage the conflicts of interest; and access the assignments for a particular reviewer in order to determine whether there are any conflicts of interest.

Reviewer corresponds to a reviewer. It has the following responsibilities: return the preferred assignments; complete the reviews; and record reminders for assigned proposals.

Repository represents the database, and holds the submitted proposals and the reviewers. Its only responsibility is to provide the stored records.

Proposal and **Review** represent additional application objects. However, the workflow doesn't interact directly with them.

6.1.3 Workflow Definition

The top level of the proposal review process consists of several steps:

1. The NCSA Allocations Office sends out the initial reviewer assignments. The workflow takes the initial assignments from the supervisor domain object, iterates over the reviewers, and sends the assignment list to each of them. Therefore, this step involves a primitive procedure followed by an iterative procedure. The primitive obtains the initial assignments from a domain object. The iterative has another primitive procedure as its body. This primitive passes the initial assignments to the domain objects representing the reviewers. Figure 6.1 shows the code corresponding to the definition of this step.

2. The reviewers assemble their reviewing preferences. Through a GUI like the one depicted in Figure 6.2, each reviewer marks submissions as "requested," "accepted," or "rejected," and sends them back to the Allocations Office. Therefore, this step involves an iterative procedure with a sequence of two primitives as its body. The first primitive obtains the reviewer preferences, and the second records them in the system. A precondition ensures that the execution component fires off the iterative procedure only after all reviewers send back their reviewing preferences. Figure 6.3 shows the code corresponding to the definition of this step.

3. The NCSA Allocations Office finalizes the assignments. This involves a primitive procedure and an iterative procedure. The primitive procedure computes the final assignments, attempting to satisfy the

```
broadcastProcess
    | obtainAssignments sendAssignments giveAssignments |
    obtainAssignments := PrimitiveProcedure
                    sends: #initialAssignments
                    to: #supervisor
                    result: #initialAssignments.
    giveAssignments := PrimitiveProcedure
                    sends: #initialAssignments:
                    with: #(#initialAssignments)
                    to: #reviewer.
    sendAssignments := IterativeProcedure
                    send: #elements
                    to: #reviewers
                    execute: giveAssignments
                    with: #reviewer.
    ^obtainAssignments , sendAssignments
```

Figure 6.1: NCSA Proposal Review Workflow—Broadcast of Initial Assignments.

reviewers' preferences and managing any potential conflicts of interest. The iterative has a sequence with two primitives as its body. The first primitive obtains the final assignments for a reviewer, and the second passes them to the domain object representing the reviewer. Figure 6.4 shows the code corresponding to the definition of this step.

4. Several days before the reviews are due, the NCSA Allocations Office sends reminders to reviewers who haven't submitted their reviews. The example application uses the GUI from Figure 6.5(a) to signal that the due date is a few days away. For each proposal that doesn't have all its reviews, the Allocations Office staff sends the late reviewers a reminder about the upcoming deadline. Upon receiving a reminder, the assigned reviews appear in the reviewer's "reminders" list—see Figure 6.5(b). This step involves an iterative procedure with a conditional as its body. The conditional tests whether a reviewer has submitted all the reviews, and sends a reminder if any reviews are still pending. Figure 6.6 shows the code corresponding to the definition of this step.

5. Once all reviews have been received, the NCSA Allocations Office assembles them into a master document to be used for the final decision. This step involves an iterative with a primitive as its body. The primitive asks the domain object representing a proposal to generate a report. A precondition

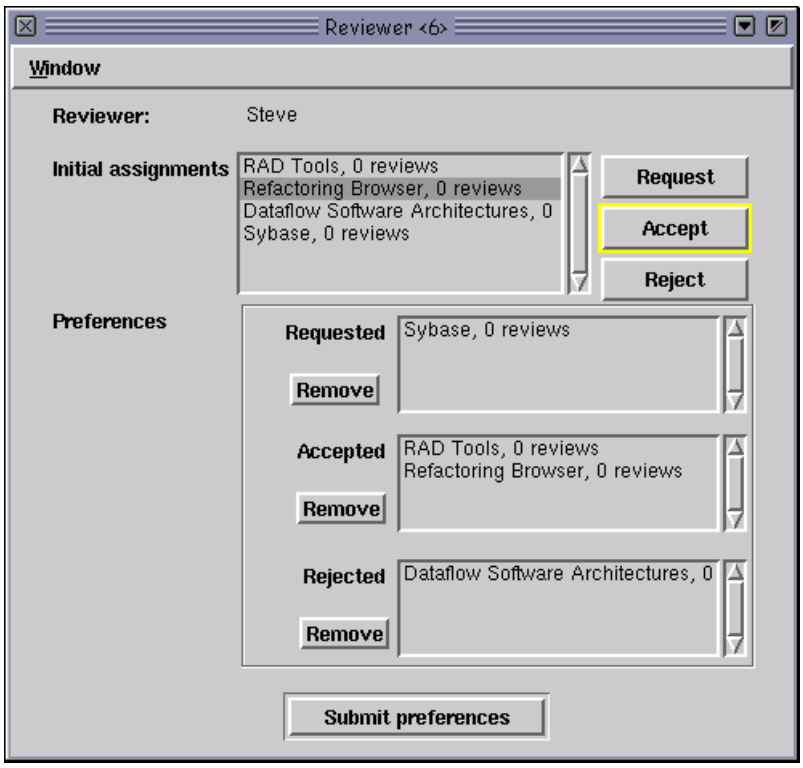


Figure 6.2: Reviewer's Review Preferences GUI.

ensures that the micro-workflow execution component executes the iterative only when all reviews are available. Figure 6.7 shows the code corresponding to the definition of this step.

Figure 6.8 shows the top-level procedure of the NCSA Proposal Review Workflow which is a SequencePro-cedure with five steps, and Figure 6.9 shows the corresponding instance diagram.

6.1.4 Discussion

An application implementing the NCSA Proposal Review Process requires basic workflow functionality, and the ability to monitor and record process execution. Additionally, the workflow functionality should integrate with application objects that (for this particular application) provide their own GUIs for reviewers and the Allocations Office supervisor, as Figures 6.2 and 6.5(a,b) show.

Micro-workflow can provide *exactly* these features. Developers extend the core with the monitoring and history components (see Table 5.7). Additionally, they can *customize* each of these components, and *integrate* them in their applications. For example, developers can change what type of run time information

```
selectionProcess
    | selectProposals recordPreferences selectionProcess precondition |
    selectProposals := PrimitiveProcedure
                    sends: #preferences
                    to: #reviewer
                    result: #preferences.
    recordPreferences := PrimitiveProcedure
                    sends: #preferencesFor:value:
                    with: #(#reviewer #preferences)
                    to: #supervisor.
    selectionProcess := IterativeProcedure
                    send: #elements
                    to: #reviewers
                    execute: selectProposals , recordPreferences
                    with: #reviewer.
    precondition := Precondition
                    withSubjectAt: #reviewers
                    block:
                        [:reviewers |
                        reviewers elements inject: true
                         into: [:haveAllPreferences :rev | rev preferences notNil and: [haveAllPreferences]]]
                    manager: preconditionManager.
    selectionProcess precondition: precondition.
    ^selectionProcess
```

Figure 6.3: NCSA Proposal Review Workflow—Send Reviewer Preferences.

the history component logs. Likewise, they can have the workflow monitor use an application's GUI to display the current state of the workflow, and/or trigger other actions.

Another characteristic that makes this case study interesting from a research viewpoint is that it requires workflow functionality that integrates with application objects which provide their own interfaces for human workers. This section has shown that micro-workflow supports the incremental integration of workflow technology within applications. Muth and colleagues [85] identify this characteristic as one of the requirements for a new generation of workflow architectures:

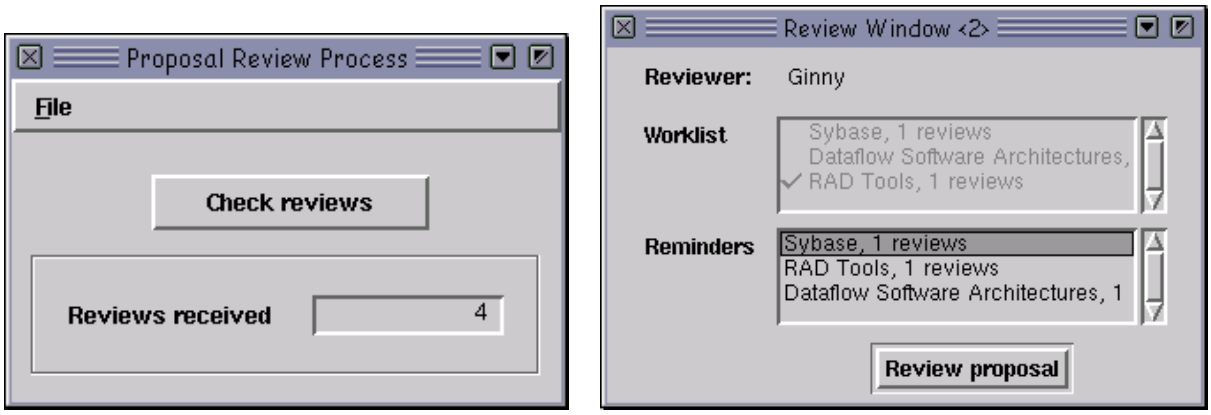
> Extending the workflow management system's functionality according to future application needs, e.g., by worklist and history management, must also be possible. In particular, at the beginning of an incremental integration process, only a limited amount of a workflow management system's functionality is actually exploited by the workflow application. Later on, as the

```
finalizeAssignmentProcess
    | assembleAssignments sendIndividualAssignments sendAllAssignments selectIndividualAssignments |
     assembleAssignments := PrimitiveProcedure
                    sends: #finalizeAssignmentsOf:for:
                    with: #(#proposals #reviewers)
                    to: #supervisor.
     selectIndividualAssignments := PrimitiveProcedure
                    sends: #finalAssignmentsFor:
                    with: #(#reviewer)
                    to: #supervisor
                    result: #individualAssignments.
     sendIndividualAssignments := PrimitiveProcedure
                    sends: #assignments:
                    with: #(#individualAssignments)
                    to: #reviewer.
     sendAllAssignments := IterativeProcedure
                    send: #elements
                    to: #reviewers
                    execute: selectIndividualAssignments , sendIndividualAssignments
                    with: #reviewer.
     ^assembleAssignments , sendAllAssignments
```

Figure 6.4: NCSA Proposal Review Workflow—Finalize Reviewer Assignments.

integration proceeds, more advanced requirements arise and demand the customization of the workflow management system to the evolving application needs.

Can current workflow systems implement this process? They can, but this is not the right question to ask. The question should rather be: can developers use the functionality provided by an existing workflow system to build an application implementing this process? Because most workflow systems focus on packaging many features, the cost of a solution based on one of these systems would be much higher. For example, many workflow management systems handle worklist management. But the Proposal Review Process doesn't need this functionality. Developers shouldn't pay for features they don't need. Nor should they have to bundle these unused features in their applications. Unfortunately, due to the monolithic nature of traditional workflow architectures, current workflow systems can only be used in an all-or-nothing manner. This approach makes it hard to reuse the functionality provided by current workflow systems within object-oriented applications. Additionally, current workflow systems don't support the incremental integration of workflow within applications, and it don't let developers customize features like workflow monitoring and



(a) **(b)**

Figure 6.5: GUIs for Allocations Staff Supervisor and Reviewer (Worklist and Reminders).

history.

6.2 Strep Throat Treatment Process

This application implements a process from the medical domain.[1] This process requires: recording the workflow history for legal reasons; allowing the physician to change a running workflow; and supporting human workers. Therefore, implementing it with micro-workflow involves extending the core with components for persistence, manual intervention, and worklists.

6.2.1 Process Overview

The strep throat treatment process begins with a patient who suspects that she may have strep throat and goes to the doctor to seek medical attention. The physician examines the patient and tests whether she has strep throat or not. If the results are positive, the physician prescribes a treatment.

Based on the patient's medical records, the doctor can treat strep throat in two different ways. If the patient is not allergic to penicillin (an antibiotic), the physician prescribes this treatment. Otherwise, he prescribes the sulfa drug. Next a nurse takes the prescription and instructs the patient about following the treatment. If the prescription contains penicillin, she also warns the patient about the possibility of an allergic reaction to antibiotics. The patient goes home and starts taking the prescribed drugs.

[1] This is the same process as the example discussed in Section 2.1.1.

```
checkAllProcess
    | remindReviewers sendReminder checkProposal checkAllProcess precondition |
    sendReminder := PrimitiveProcedure
                    sends: #notifyAbout:
                    with: #(#proposal)
                    to: #reviewer
                    result: #review.
    remindReviewers := IterativeProcedure
                    send: #lateReviewers
                    to: #proposal
                    execute: sendReminder
                    with: #reviewer.
    checkProposal := ConditionalProcedure
                    send: #lateReviewers
                    to: #proposal
                    if: [:arg | arg isEmpty not]
                    execute: remindReviewers.
    checkAllProcess := IterativeProcedure
                    send: #elements
                    to: #proposals
                    execute: checkProposal
                    with: #proposal.
    precondition := Precondition
                    withSubject: self
                    block: [:w | w hasCheckBeenRequested]
                    manager: preconditionManager.
    checkAllProcess precondition: precondition.
    ^checkAllProcess
```

Figure 6.6: NCSA Proposal Review Workflow—Check for Pending Reviews.

Two days after the beginning of the treatment the nurse checks with the patient to see whether there have been any improvements. She also reminds the patient to continue taking the prescribed drugs even if her condition has improved. At the end of the treatment, the nurse checks again the state of the patient.

Finally, the nurse updates the patient's records to reflect the completion of the treatment (if the patient was treated for strep throat), or the fact that she was tested for strep throat and the results were negative.

6.2.2 Workflow Actors

The workflow corresponding to the strep throat treatment involves the following workflow actors (i.e., humans):

```
generateAllProcess
    | generateAllProcess precondition |
    generateAllProcess := IterativeProcedure
                send: #elements
                to: #proposals
                execute: (PrimitiveProcedure sends: #createReport to: #proposal)
                with: #proposal.
    precondition := Precondition
                withSubjectAt: #proposals
                block:
                    [:proposals |
                    proposals elements inject: true
                                        into: [:haveAllReviews :proposal | proposal lateReview-
ers isEmpty and: [haveAllReviews]]]
                manager: preconditionManager.
    generateAllProcess precondition: precondition.
    ^generateAllProcess
```

Figure 6.7: NCSA Proposal Review Workflow—Generate Final Report.

```
buildWorkflow
    self
        rootProcedure: self broadcastProcess , self selectionProcess
            , self finalizeAssignmentProcess , self checkAllProcess
            , self generateAllProcess
```

Figure 6.8: Building the Root Procedure of the NCSA Proposal Review Workflow.

Doctor is the physician who examines patients and prescribes the treatment.

Nurse is the nurse. She has the following responsibilities: treat patients according to the physician's prescription; check the condition of patients under treatment; remind patients to continue the treatment; and update the records.

6.2.3 Workflow Definition

The top level of the strep throat treatment process consists of several steps:

1. The physician examines the patient and decides if she should be treated for strep throat. This step involves a primitive procedure that sends the examine: message to the domain object representing the

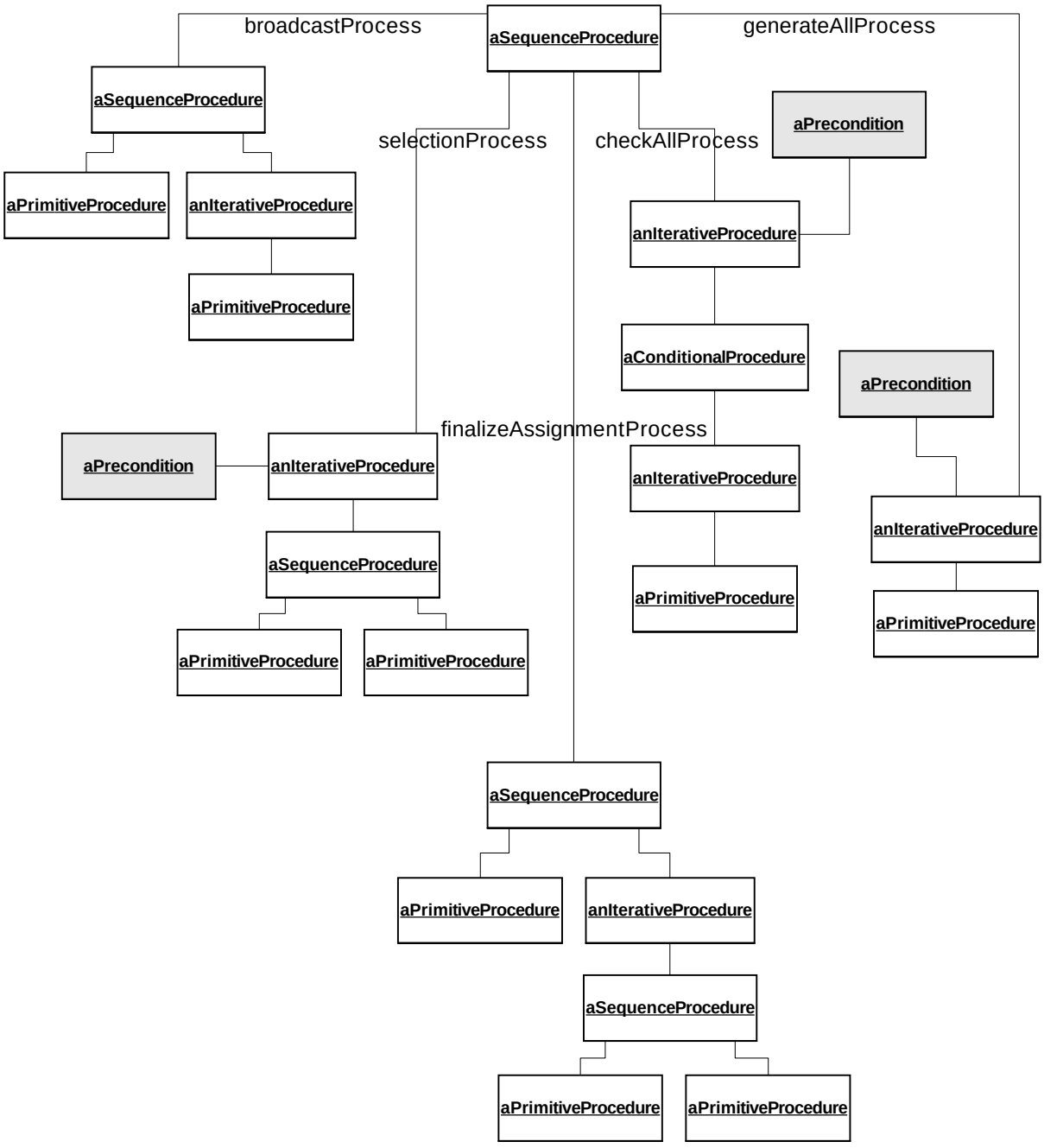


Figure 6.9: NCSA Proposal Review Workflow, instance diagram.

physician. But this workflow involves humans instead of domain objects. Therefore, the runtime has a Worklist instance at the slot corresponding to the physician. Through the mechanism described in Section 5.5, this object converts the examine: message into a human-readable work item and adds it to the physician's worklist. Figure 6.10 shows the code corresponding to this step.

```
examineProcess
    | examination |
    examination := PrimitiveProcedure
                    sends: #examine:
                    with: #(#patient)
                    to: #doctor
                    result: #patientNeedsTreatment.
    ^examination
```

Figure 6.10: Strep Throat Workflow—Examining the Patient.

2. The nurse performs the treatment. This involves a sequence with four steps which correspond to administering the prescribed drugs, checking the patient's condition on the second day of treatment, reminding the patient to continue the treatment, and checking again the condition at the end of the treatment. Preconditions synchronize the two primitives that perform checks with a calendar object. Figure 6.11 shows the definition of this step.

3. Finally, once the treatment completes, the nurse updates the patient's records. This involves another primitive procedure that sends the updateRecordsOf: message to the domain object representing the nurse. In effect, this step adds a human-readable work item to the nurse's worklist. Figure 6.12 shows the definition of this step.

Figure 6.13 shows that the top-level procedure of the Strep Throat Treatment Workflow is a SequenceProcedure with three steps. The second step—which corresponds to the treatment—executes only when the physician has determined that the patient needs to be treated for strep throat.

6.2.4 Discussion

An application implementing the Strep Throat Treatment Process requires a workflow system supporting manual intervention, persistence, and worklists.

```
treatmentProcess
    | treatment initialCheck reminder finalCheck calendar |
    treatment := PrimitiveProcedure
                    sends: #performTreatmentOf:
                    with: #(#patient)
                    to: #nurse.
    initialCheck := PrimitiveProcedure
                    sends: #checkConditionOf:
                    with: #(#patient)
                    to: #nurse.
    calendar := self calendarInterface calendar.
    initialCheck precondition: (Precondition withSubjectAt: #patient
                    block: [:aPatient | calendar currentDay >= (aPatient treatmentStartDate + 2)]).
    reminder := PrimitiveProcedure
                    sends: #sendReminderTo:
                    with: #(#patient)
                    to: #nurse.
    finalCheck := PrimitiveProcedure
                    sends: #checkConditionOf:
                    with: #(#patient)
                    to: #nurse.
    finalCheck precondition: (Precondition withSubjectAt: #patient
                    block: [:aPatient | calendar currentDay >= (aPatient treatmentStartDate + 4)]).
    ^treatment , initialCheck , reminder , finalCheck
```

Figure 6.11: Strep Throat Workflow—Performing the Treatment.

Micro-workflow enables developers to assemble *exactly* these features, and facilitates the integration with other systems. The manual intervention component provides full control over how the system handles backward recovery. Developers can use either undo or semantic compensation mechanisms, depending on the type of processing carried out by each domain object. The persistence component makes the framework database-independent. Developers can choose any type of database, or even build the application around an existing database schema. The worklist component provides full control over worklist handling and enables developers to use specialized directory services for staff resolution.

Although various studies have identified the absence of support for manual intervention as one of the shortcomings of current workflow systems [2], only few of them have this feature. The Strep Throat Treatment Process described in this section can be implemented with a workflow system only if this supports manual intervention. However, a solution built with a traditional workflow architecture doesn't have the

```
updateProcess
    | update |
    update := PrimitiveProcedure
                sends: #updateRecordsOf:
                with: #(#patient)
                to: #nurse.
    ^update
```

Figure 6.12: Strep Throat Workflow—Updating the Records.

```
buildWorkflow
    | examination test update |
    examination := self examineProcess.
    test := ConditionalProcedure
                if: [:arg | arg]
                for: #patientNeedsTreatment
                execute: self treatmentProcess.
    update := self updateProcess.
    self rootProcedure: examination , test , update
```

Figure 6.13: Building the Root Procedure of the Strep Throat Treatment Workflow.

flexibility provided by micro-workflow. Developers can't pick and choose the workflow features, nor can they choose the type of database (i.e., relational or object-oriented). Additionally, they have no (or limited) control over the backward recovery and worklist management mechanisms bundled with the system.

6.3 Newborn Followup Process

This application implements an administrative process[2] from the Newborn Screening Program at the Illinois Department of Public Health (IDPH). Hospitals throughout the state collect blood samples from newborn babies and send them to a Chicago-based laboratory for testing. The followup process is triggered only when the lab returns anomalous test results. The objective of the process is to keep track of these problems, and ensure that they are dealt with according to the state law. I studied the IDPH Newborn Followup Process while I was part of the development team in charge of building the IDPH enterprise framework and applications.

[2]This is the same process as the example discussed in Section 5.6.

6.3.1 Process Overview

The IDPH Newborn Followup Process starts when the test results indicate a potential problem. Upon receiving an abnormal test result, an IDPH employee phones the newborn's physician and asks her to obtain a new blood specimen for a second test. Once the phone call completes, IDPH also sends the physician a letter with the information communicated over the phone. The physician collects the sample on a special filter paper and sends it to the Chicago laboratory for testing.

At the Chicago lab, a clerk acknowledges the receipt of the blood specimen by scanning the barcode printed on the envelope. Next a lab technician performs the test. Once the test completes, the lab supervisor certifies the result. The supervisor may ask the technician to redo the test if the results are on the borderline, if they are completely inconsistent with the first results, or if a problem occurred during testing—e.g., two samples mixed into the same well of the batch board. Once certified, the lab releases the test result into the system. The certification improves the accuracy of the test results and ensures that the testing procedure doesn't miss any positives.

When the second test results confirm the problem, the IDPH employee assigned to the case calls the physician again and gives her a list with consultants who can handle the case. Once the phone call completes, IDPH also sends a letter with the information exchanged over the phone to the physician.

The physician refers the parents/guardian to a consultant. Seven days after the consultant reports that he is handling the case, the IDPH employee contacts him to check on the status of the case. The state employee keeps contacting the consultant every seven days, until the case is either solved, or the consultant can't contact the parents/guardian (e.g., they have moved out of state).

6.3.2 Domain Objects and Workflow Actors

Unlike the examples discussed in Sections 6.1 and 6.2, the IDPH Newborn Followup Process involves both domain objects and workflow actors (humans):

IdphSystem represents the IDPH system. It has the following responsibilities: print out and send the physician a letter requesting a second blood sample; print out and send the physician a letter with the list with consultants; and update the records.

IdphStaff (human) is the IDPH employee. It performs the following tasks: calls the physician to request a

second blood sample; calls the physician to provide the list with consultants; and calls the consultant to check the status.

LabSystem represents the lab system. It is responsible for updating the lab records.

LabStaff represents a lab clerk or a lab technician. As the lab clerk, this domain object is responsible for acknowledging the receipt of blood specimens. As the lab clerk, it is responsible for performing a test involving a dry blood sample. I have replaced the lab clerk and the lab technician with a single domain object for simplicity. Since this case study mainly illustrates federated workflow, the simplification keeps the focus on the issues relevant to hierarchical and distributed workflow.

LabSupervisor (human) is the lab supervisor in charge with test certification.

6.3.3 Workflow Definition

The top level of the IDPH Newborn Followup Process executes at the IDPH site and consists of four steps:

1. Following the notice that the first blood screening indicates an abnormal result, the IDPH employee notifies the physician over the phone. Once the phone call completes, the system automatically prints and mails a form letter. This step involves a sequence with two primitives. The first primitive queues a work item in the employee's worklist, and the second step handles the letter. Through a Future object, the object passed from the first step to the second (at the slot firstCallOk) ensures that the system sends the letter only after the employee closes the work item. Figure 6.14 shows the definition of this step.

2. The lab system receives the blood sample and runs the tests. This step involves the federated workflow component, which executes the lab workflow at the lab site. Let's begin with what happens there.

   The lab workflow consists of a SequenceProcedure with three steps. First the lab clerk acknowledges the receipt of the blood specimen. This step involves a primitive. Next the lab technician performs the test and the lab supervisor certifies the results. Since the technician and the supervisor can repeat the testing and certification several times, this step involves a repetition with a two-step sequence as its body. Finally the lab sends out the certified test result. This last step involves another primitive. Figure 6.15 shows the definition of the entire lab workflow.

```
firstScreeningProcess
    | call send |
    call := PrimitiveProcedure
                sends: #callAbnormalResult1For:
                with: #(#infant)
                to: #idphstaff
                result: #firstCallOk.
    send := PrimitiveProcedure
                sends: #sendAbnormalResult1For:callOk:
                with: #(#infant #firstCallOk)
                to: #idphsystem
                result: #firstLetterOk.
    ^SequenceProcedure with: call with: send
```

Figure 6.14: Newborn Followup Workflow—Notification of Abnormal Test Result.

The followup (i.e, parent) workflow requires the result of the lab workflow at the slot screening2. But as Figure 6.15 shows, the lab workflow puts the screening results into a slot named testResult. Thus the activity of the followup workflow that fires off the lab subworkflow configures a BidirectionalMapper instance to map the data from the slot named testResult to the one named screening2. SubworkflowProcedure intermixes transparently with the other control structures, and WorkflowFacade hides the fact that the subworkflow resides on a remote machine. These abstractions let the workflow designer focus on the process definition as a whole. Figure 6.16 shows the definition of this step.

3. The IDPH employee checks the results of the second screening. If the test results confirm the initial problem, he contacts the physician with the referral information. Once the phone call completes, the system prints and mails a letter with the list of consultants communicated over the phone. The consultant notifies IDPH when he starts handling the case. Next the IDPH employee keeps contacting the consultant every seven days until the case is closed. Therefore, this step involves a conditional (which checks the lab results) with a sequence as its body. The sequence has two primitives which handle the communication with the physician (phone and mail), and a repetition which handles contacting the consultant. A precondition ensures that the body of the repetition (another primitive) executes every seven days. Figure 6.17 shows the definition of this step.

```
buildWorkflow
    | ack test certify rep send |
    ack := PrimitiveProcedure
                sends: #acknowledgeReceipt:
                with: #(#bloodSpecimen)
                to: #labclerk
                result: #bloodSpecimen.
    test := PrimitiveProcedure
                sends: #runTest:for:
                with: #(#testType #bloodSpecimen)
                to: #labtech
                result: #uncertifiedTestResult.
    certify := PrimitiveProcedure
                sends: #certifyTest:
                with: #(#uncertifiedTestResult)
                to: #labsup
                result: #certifiedResult.
    rep := RepetitionProcedure
                repeat: test , certify
                until: [:arg | arg isCertified or: [arg number = 5]]
                for: #certifiedResult.
    send := PrimitiveProcedure
                sends: #updateRecordsFor:
                with: #(#certifiedResult)
                to: #labsystem
                result: #testResult.
    self rootProcedure: ack , rep , send
```

Figure 6.15: Newborn Followup Workflow—Lab Workflow Definition.

```
labScreeningSubprocess
    | mapper |
    mapper := BidirectionalMapper new.
    mapper rlMap: #testResult to: #screening2.
    ^(SubworkflowProcedure on: WorkflowFacade instance)
        mapper: mapper;
        yourself
```

Figure 6.16: Newborn Followup Workflow—Lab System Subworkflow.

```
testSecondScreeningProcess
	| test |
	test := ConditionalProcedure
			if: [:arg | arg isAbnormal]
			for: #screening2
			execute: self secondScreeningProcess.
	test precondition: (Precondition withSubject: screening2
			block: [:screening | screening haveResults]).
	^test

secondScreeningProcess
	| call send contact loop |
	call := PrimitiveProcedure
			sends: #callAbnormalResult2For:
			with: #(#infant)
			to: #idphstaff
			result: #secondCallOk.
	send := PrimitiveProcedure
			sends: #sendAbnormalResult2For:callOk:
			with: #(#infant #secondCallOk)
			to: #idphsystem
			result: #secondLetterOk.
	contact := PrimitiveProcedure
			sends: #contactConsultantFor:
			with: #(#infant)
			to: #idphstaff
			result: #consultantResult.
	contact precondition: (Precondition withSubjectAt: #screening2
			block: [:screening | calendar
			currentDay ~= 0 and: [calendar currentDay \\ 7 = 0]]).
	loop := RepetitionProcedure
			repeat: contact
			until: [:arg | arg]
			for: #consultantResult.
	^call , send , loop
```

Figure 6.17: Newborn Followup Workflow—Processing of Second Screening Results.

4. Finally, the IDPH employee updates the records. This involves a primitive and Figure 6.18 shows the definition of this step.

```
updateProcess
    ^PrimitiveProcedure
        sends: #updateRecordsFor:
        with: #(#infant)
        to: #idphsystem
        result: #finalresult
```

Figure 6.18: Newborn Followup Workflow—Updating the Records.

The simplified instance diagram from Figure 6.19 shows a configuration with a single lab subworkflow.

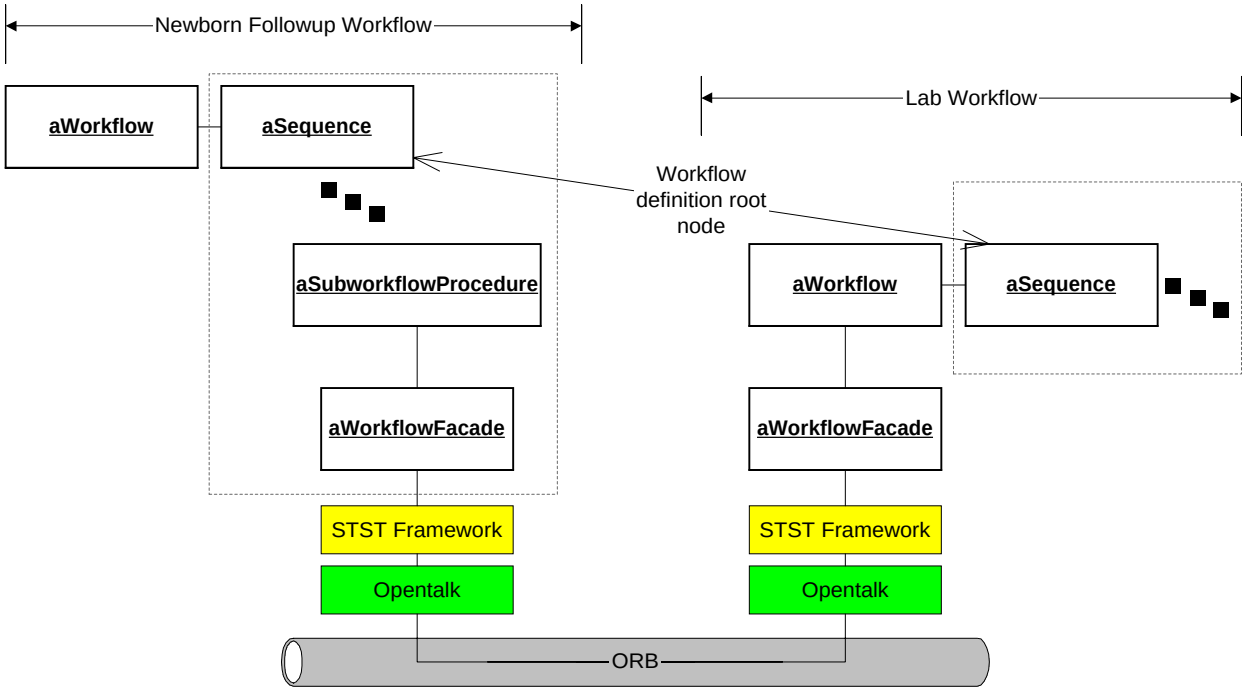


Figure 6.19: Newborn Followup Workflow, simplified instance diagram.

6.3.4 Discussion

The IDPH Newborn Followup Process requires a workflow system providing federated workflow functionality. The micro-workflow framework described in the previous chapters implements this feature. Micro-

workflow lets developers to plug in the federated workflow component *only when they need it*. This approach provides full control over federating heterogeneous workflow systems, and over the inter-workflow data flow mechanism. It also facilitates integrating the federated workflow component with other distributed application architectures.

Currently most workflow systems don't have the ability to integrate multiple workflows distributed over the network. Additionally, in the absence of a common standard, the integration of heterogeneous workflow systems requires access to the internals. But the black-box design of current workflow systems makes them hard to integrate.

6.4 Framework Changes

The micro-workflow framework discussed in Chapter 4 started with the core components. I have built the monitoring, history, persistence, manual intervention, worklist, and federated workflow components from Chapter 5 for the case studies discussed in the previous sections.

One of the key features of micro-workflow lies in the ability to extend the architecture with features implemented by separate components. Good designs strive to reduce the coupling between components. Ideally, adding new components (i.e., micro-workflow features) shouldn't require changing existing framework components. In practice, however, this is not always possible. Sometimes adding a component invalidates assumptions made by others, thus requiring code changes.

This section provides metrics that measure the impact of the components described in Chapter 5 on the core components. I collected these numbers while I implemented the case studies discussed in Sections 6.1–6.3. For each component I show the number of new classes, as well as the number of new messages added to existing classes. These two numbers show how the component contributes to the framework. I also show the percentage of core classes and the number of existing messages that were modified to accommodate the new functionality. These two numbers represent the breakage caused to the core components. The metrics reflect the amount of code required to implement a new feature, and the coupling between new and existing components. They provide a quantitative measure of the design cost of the micro-workflow framework.

While I knew from the beginning that micro-workflow will eventually have to deal with persistence, the design wasn't planned in advance for the features discussed in Chapter 5.1. Rather, I have added them

gradually, as the framework evolved and needed to accommodate different requirements. The wide range of features added to the framework show that micro-workflow can grow to accommodate features that were not part of the initial architecture.

6.4.1 Changes for the Proposal Review Process

The NCSA Proposal Review Process was the first application implemented with the micro-workflow framework. Initially the framework consisted only of the execution, process, and synchronization components. The Proposal Review Process required monitoring and history. So I extended the core with two components implementing these features.

Table 6.1 summarizes the framework changes for adding the monitoring and history components. The monitoring component adds two classes, the ProcedureMonitor and the ProcedureMonitorInterface. Likewise, the history component adds a Logger class. The rightmost two columns corresponding to breakage show that adding monitoring and history had a small impact on the core. Both components involved changing the implementation of the executeNewInstanceFollowing: message of the Procedure class (in the execution component). Therefore they affect one out of 13 core classes, which corresponds to approximately 7.6%.

Notice, however, that the first implementation of the history component logged the workflow events in memory, through a logger object Singleton [39]. I later introduced a more flexible mechanism based on strategies (see Section 5.1).

Component	New classes	New messages in existing classes	Changed core classes (%)	Changed messages
Monitoring	2	0	7.6%	1
History	1	0	7.6%	1

Table 6.1: NCSA Proposal Review Process—Summary of Framework Changes.

6.4.2 Changes for the Strep Throat Treatment Process

The physician involved in the Strep Throat Treatment Process changes a running workflow when the initial treatment doesn't work. If the antibiotic-based treatment triggers an allergic reaction, the nurse asks the

patient back to the office, and the physician prescribes a different treatment. This requires support for manual intervention. Extending the core with this functionality had the strongest impact on the other framework components.

The manual intervention component introduced the Reviewer class. Additionally, it added 4 new messages to the execution component, 3 new messages to the synchronization component, and 8 new messages to the process component. As Section 5.4 has discussed, altering the execution of a running workflow requires separating the control flow mechanism of the implemented system from the control flow mechanism of the implementing system. This involved changing one class of the execution component, and 4 classes of the process component. The call:with:, return:, and return:state: messages provide the inter-procedure control flow mechanism. The Procedure class of the execution component implements the first two, and the ConditionalProcedure, IterativeProcedure, RepetitionProcedure and SequenceProcedure of the process component implement the third.

The persistence component added 4 classes (SessionManager, TraceManager, WorkflowSessionParameters, and WorkflowTrace). It also required changing the history component to accommodate different logging mechanisms. Therefore I refactored [96] the history component to use the logging strategies (NullLogging, MemoryLogging, and GemStoneLogging) described in Section 5.1.

I have also added a workflow session object providing access to the objects shared by the procedures of an activity map (i.e., workflow definition). The WorkflowSession holds: the logging strategy of the history component, the workflow stack of the execution component, the precondition manager of the synchronization component, and the rewinder of the manual intervention component. Additionally, it sends notifications to the procedure monitor of the monitoring component.

The worklist component added 4 new classes (Future, Workitem, Worklist, and WorklistGUI). It also added the waitUntilNoFutures message (discussed in Section 5.5 and illustrated in Figure 5.33) to the ProcedureActivation class of the execution component. Its design around Smalltalk's reflective facilities [34] reduced the impact on the other framework components.

Table 6.2 summarizes the framework changes for adding the manual intervention, persistence, and worklist components required by the Strep Throat Treatment Process. The high breakage caused by the manual intervention component is due to modifying the control flow mechanism, which affected the five classes mentioned above.

Component	New classes	New messages in existing classes	Changed core classes (%)	Changed messages
History	4	0	5.8%	1
Persistence	4	0	5.8%	0
Manual intervention	2	15	33%	8
Worklist	4	1	5.5%	0

Table 6.2: Strep Throat Treatment Process—Summary of Framework Changes.

6.4.3 Changes for the Newborn Followup Process

The IDPH Newborn Followup Process involves workflows executing at different locations. This requires support for hierarchical workflow (i.e., representing the lab process as a node in the definition of the followup process) and distributed workflow (i.e., workflow execution and object transfer across the network). The federated workflow component extends the micro-workflow core with this functionality.

Eight new classes implement the functionality provided by the federated workflow component. SubworkflowProcedure adds support for hierarchical workflow, while WorkflowAbstract, Workflow, WorkflowFacade, UnidirectionalMapper, BidirectionalMapper, Container, and ContainerShadow handle distribution. In addition to these classes, federated workflow also requires adding the executeRootProcedureWith: message to the Procedure class of the execution component. The abstractions provided by the Opentalk STST framework allow this component to incur no other changes to the other framework components.

Table 6.2 summarizes the framework changes for adding the federated workflow component. The figures show that the modular design of the framework combined with powerful abstractions and good object-oriented design reduce the impact of the federated workflow component on the other framework components.

Component	New classes	New messages in existing classes	Changed core classes (%)	Changed messages
Federated workflow	8	1	5.5%	0

Table 6.3: Newborn Followup Workflow—Summary of Framework Changes.

6.5 Runtime Overhead

The ability to plug into the micro-workflow core components that implement advanced workflow features requires adding hooks to the core's components. Once added, these hooks incur runtime overhead even when the pluggable components they accommodate aren't plugged in. This overhead represents the run time cost of the flexibility achieved through composition.

This section evaluates the run time overhead in terms of the number of message sends when a pluggable component is not used. I will show that the design discussed in Chapters 4 and 5 keeps the runtime overhead low, thus yielding an affordable cost. Different designs or further refactorings of my design will change these numbers. Therefore, the measurements should be regarded as guidelines or estimates, and not as absolute values.

The history component hooks up to the execution component. The Procedure sends logWorkflowEvent to the ProcedureActivation instance. In response to this message, the activation strips off the runtime information from a copy of itself and then logs it through the workflow session. The sequence diagram from Figure 6.20 shows the four messages required to support the history component. These messages are sent even when the workflow doesn't use history—e.g., the session holds a NullLogging instance.

The Procedure class also provides the hook for the monitoring component. Once a Procedure instance initializes a new ProcedureActivation instance, it sends the signalExecutionOf: to the workflow session. The WorkflowSession instance signals the ProcedureMonitor through the changed:with: message. The sequence diagram from Figure 6.21 shows the two messages required to support the monitoring component.

The manual intervention component requires both a hook and support code. The hook involves one additional message send (in the execution component) that tests the value returned by the PreconditionManager in response to the waitUntilFulfilledIn: message. The execution and process components provide the support code that implements the workflow control flow mechanism. The Procedure class of the execution component implements the call:with: and return: messages, adding seven and eight message sends, respectively. The ConditionalProcedure, IterativeProcedure, RepetitionProcedure and SequenceProcedure classes of the process component implement the return:state: message, adding one, 4, or 5 message sends.

The persistence, worklist, and federated workflow components require no hooks and therefore don't incur any runtime overhead when they're not used.

Table 6.4 summarizes the additional message sends required to accommodate each micro-workflow

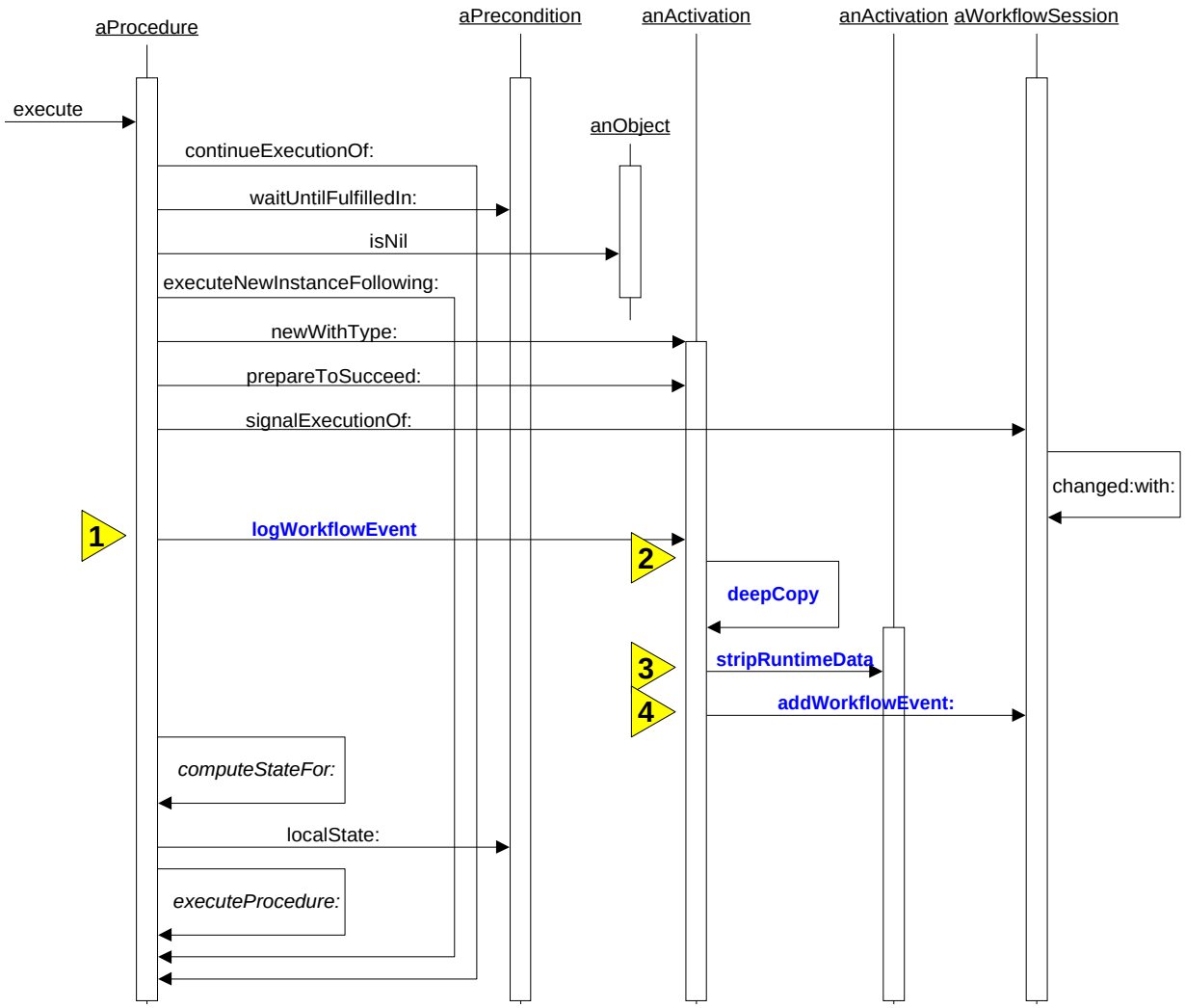


Figure 6.20: Runtime overhead added by the history component to the execution component. The additional messages are numbered on the left and are shown in blue/grayed.

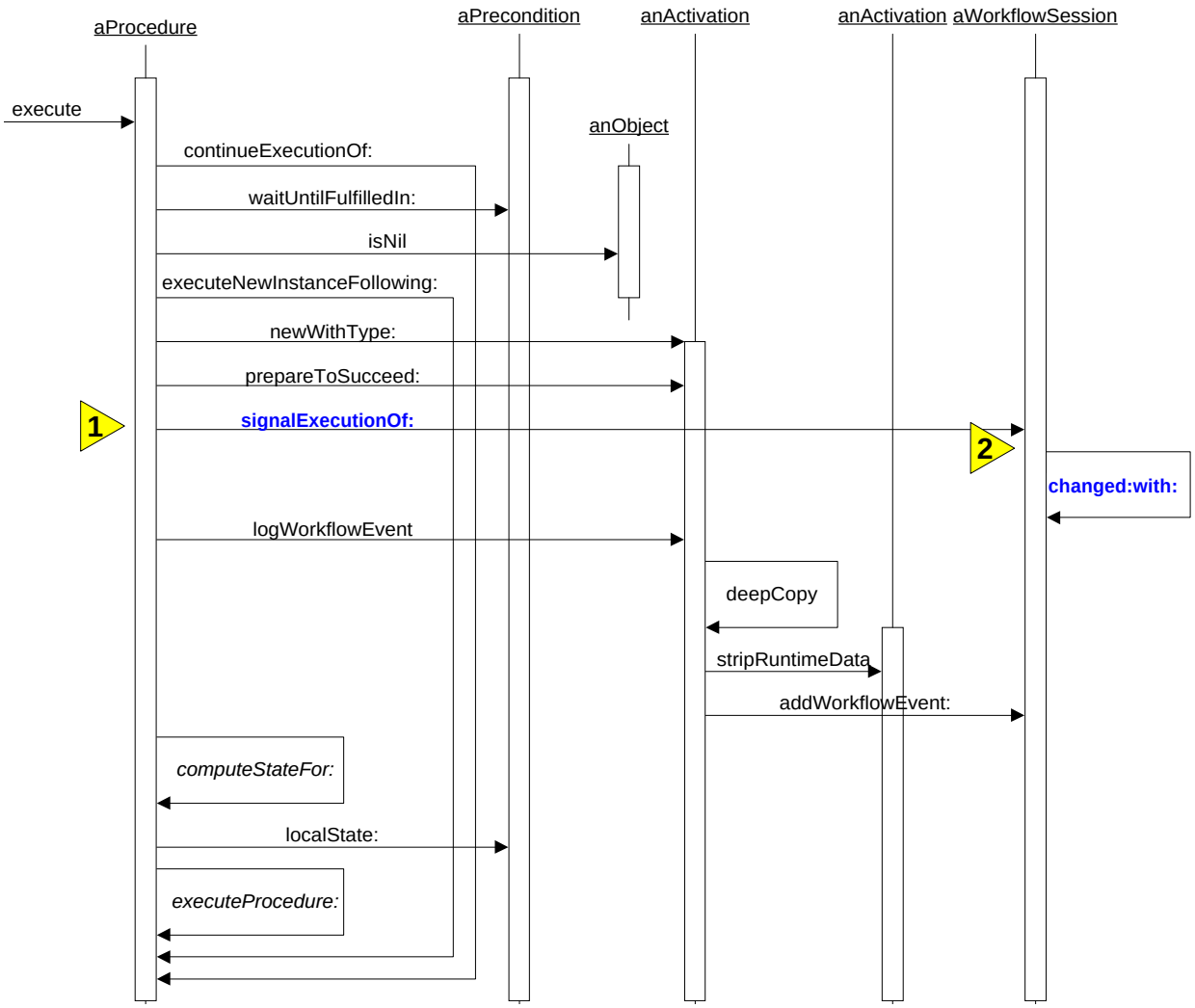


Figure 6.21: Runtime overhead added by the monitoring component to the execution component. The additional messages are numbered on the left and are shown in blue/grayed.

component. Since some schools of thought advocate *Direct Variable Access* and others *Indirect Variable Access* [8], the numbers don't take into account messages sent to self to access or change the value of instance variables.

Component	Overhead (message sends)
History	1
Persistence	0
Monitoring	2
Manual intervention	between 1 and 14
Worklist	0
Federated Workflow	0

Table 6.4: Runtime overhead (in message sends) incurred to accommodate the pluggable components.

6.6 Evaluation Summary

This chapter has presented and discussed the implementation of three applications with the micro-workflow framework. In doing so, it has answered several important questions regarding micro-workflow.

First, this chapter has shown that developers can use micro-workflow to build object-oriented applications that implement workflows corresponding to real problems. The workflows for reviewing proposals, treating strep throat, and tracking the treatment of newborns have a broad range of requirements. Each of them uses different combinations micro-workflow components.

Second, it has measured the design cost—how adding each of the six components impacts the framework. For each component, the metrics give the number of new classes and the number of new messages introduced to existing classes, which represents its contribution to the framework. They also give the breakage to the core components in terms of changed classes and messages. The metrics show a relatively low coupling between micro-workflow components.

This chapter has also measured the run time cost—the additional overhead introduced by the hooks for the pluggable components. The history, monitoring and manual intervention components add several message sends. In contrast, the persistence, worklist, and federated workflow components employ techniques that eliminate the overhead when they are not used.

Besides answering these questions, this chapter has served an additional purpose. The implementations of the three applications provide examples of how to use micro-workflow. They represent a starting point

for using micro-workflow in object-oriented applications, and help developers estimate the type of changes required to build additional micro-workflow components.

In summary, the case studies discussed in this chapter have proven that micro-workflow can implement workflows with different requirements. Additionally, the quantitative evaluation in terms of design and run time costs shows that micro-workflow provides a viable alternative for developers who need customizable workflow functionality within object-oriented applications.

Chapter 7

Related Research

The work reported in this thesis corresponds to several research directions in workflow management: workflow architectures; development environments for workflow; and flexible workflow. This chapter provides an overview of some of the research projects that focus on these issues. Section 7.4 at the end of this chapter summarizes the research projects reviewed in the following sections.

7.1 Workflow Architectures

7.1.1 Mentor-lite

The Mentor project (**M**iddleware for **Ent**erprise-wide W**or**kflow Management) is a collaboration between the University of Saarland and the Union Bank of Switzerland. Mentor models workflows with state and activity charts [87]. Initially the project focused on deriving distributed workflows starting from formal specifications. The researchers developed an algorithm that transforms a centralized state and activity chart into a provably equivalent partitioned one, suitable for distributed execution. But the work on the Mentor project helped Muth and colleagues realize the limitations of workflow architectures, and prompted them to recommend "a review of current architectures of workflow management systems" [84]. Consequently they are currently working on Mentor-lite, a second-generation Mentor system.

Mentor-lite is a lightweight workflow management system with a small footprint. It focuses on separating the workflow kernel functionality from the additional functionality typical of workflow systems, and on supporting the incremental integration of workflow technology within existing business environments. For example, the researchers observe that the administrative facilities typical of current workflow architectures don't match the requirements of applications:

> [Administrative facilities] are either not powerful enough and can not be sufficiently tailored to application needs, or they provide too much functionality which, in most cases, remains unexploited but increases the system cost, the system footprint, and affects the system performance. Based on the above observations, the need for a review of current architectures of workflow management systems is obvious. A new architecture must support an easy integration of workflow management systems into existing environments, must consider optional administrative facilities as an integral part, and must aim at minimizing the systems footprint.

Mentor-lite solves this problem by regarding workflow features as extensions implemented as workflows on top of a lightweight kernel. At the time of this writing, the project has built extensions for worklist management, history management, and monitoring [131, 86]. Figure 7.1 shows the Mentor-lite architecture.

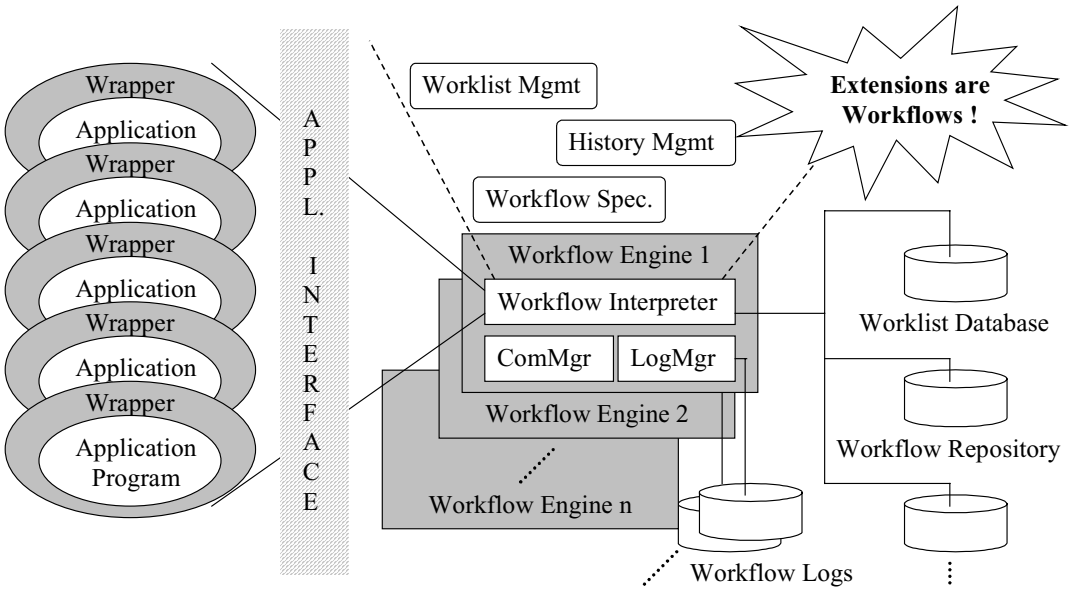


Figure 7.1: The Mentor-lite architecture. Worklist and history management are implemented as workflows on top of a lightweight workflow engine—diagram from Muth and colleagues [85].

Micro-workflow shares several characteristics with Mentor-lite. They both propose new workflow architectures centered around a lightweight kernel/core that provides basic workflow functionality. Components that implement advanced workflow features come in the form of extensions. However, two key characteristics set micro-workflow apart from Mentor-lite. First, Mentor-lite focuses on providing large scale workflow functionality, at the application level. In contrast, micro-workflow targets workflows involving objects. Sec-

ond, Mentor-lite relies on workflow technology to provide advanced features. Extensions for monitoring, history, and worklists are implemented as workflows on top of a workflow engine. In contrast, micro-workflow uses object technology. Through techniques specific to object systems, developers can customize any micro-workflow extension (i.e., component), as well as its core. Additionally, they can mix and match the workflow features and extend the core through object composition.

7.1.2 OPERA

The **O**pen **P**rocess **E**ngine for **R**eliable **A**ctivities (OPERA) project at ETH Zürich focuses on providing a kernel for process management [4]. OPERA aims at integrating middleware technology relevant to process support in distributed systems. This research stems from two observations. First, the narrow purpose design of current workflow systems limits their applicability to the domain and applications for which they have been tailored. Second, workflow architectures lack the capabilities required to adapt them to new kinds of applications. Klaus Hagen's PhD thesis [49] addresses these problems from a transactional perspective. He proposes a generic process support kernel that leverages database technology to provide support for process management and execution guarantees.

For example, Hagen and colleagues [48] observe that research projects and commercial products regard workflows as black-boxes. Processes can't exchange data while they are executing. But the coarse granularity of workflow activities requires a different degree of encapsulation in a workflow tool than in a programming language. Consequently, they argue for a workflow architecture where interprocess communication is implemented as an additional component of the execution engine.

The research reported in this thesis also aims at providing a flexible workflow architecture where components encapsulate different concerns. The OPERA kernel, the brainchild of a database research group, employs techniques specific to database systems. In contrast, micro-workflow targets object-oriented software developers and therefore concentrates on techniques specific to object systems.

7.2 Development Environments for Workflow

7.2.1 Teamware

The doctoral research of Patrick Scott Chun Young considers process specification and enactment for technical and non-technical users [140]. His prototype system Teamware provides process execution support for coordinating tasks between developers working on a software project.

Teamware employs a "category" object model. In this model, an activity category defines the behavior of instances without providing a complete data template. The activity specification complements the category and defines the state of instances. This object model targets non-technical users; it provides the behavior and allows them to reuse and customize the data through parameterization.

Teamware supports multiple stakeholders and therefore aims at a wide range of users. A layered architecture provides appropriate levels of abstraction for the three basic roles for interacting with the system. The foundation layer implements tool integration, object persistence and distribution and targets software developers. The system layer handles Teamware programs and targets process programmers. Finally, the user layer provides graphical tools for non-technical end-users. However, Teamware has several shortcomings. Integration causes problems whenever applications or tools need to call back into the system and change the process state, making bidirectional integration hard. Likewise, users can't easily change the control flow of a running process.

In contrast, micro-workflow targets only software developers. Developers build workflows by aggregating specialized workflow and application objects (black box reuse). They customize the architecture through subclassing and polymorphism (white box reuse). They add new workflow features through composition. Micro-workflow closes the gap between application objects and the processes they are part of. This facilitates bidirectional integration, one of the shortcomings of the Teamware project.

7.2.2 Transaction-Oriented Workflow Environment

One of the first research projects that attempts to provide workflow functionality for developers is the work of Papazoglou and colleagues [99]. They observed that "little attention has been given to software development environments" suitable for building workflow environments and applications. Consequently, they developed the Transaction Oriented Workflow Environment (TOWE). TOWE provides facilities for the con-

struction of network-centric workflow applications. These facilities come in the form of a class library for the Sather programming language and employ the services of the Parallel Virtual Machine (PVM) package for distributed programming and message passing. In effect, this combination yields an object-oriented language specialized for workflow applications.

TOWE combines concepts from object-oriented programming with distributed computing and open-nested transaction facilities. It focuses on the transactional aspect of workflows. Transaction classes implement operations like unsafe commit and cancel across the participating databases. TOWE strives to maintain a loose coupling between the component databases. In contrast, micro-workflow focuses on enabling developers to define and execute processes within object-oriented applications through techniques specific to object systems. Besides a collection of classes that provide workflow functionality, the framework approach also encompasses the way these classes interact, and the expectations placed on their users (i.e., programmers). In effect, this corresponds to a "skeleton" workflow system that developers tailor for their problems.

7.2.3 TriGS*flow*

TriGS*flow* (described in Section 2.5) is a PhD thesis from University of Linz. There are several differences between TriGS*flow* and micro-workflow. First, TriGS*flow* strives to provide most workflow functionality through techniques specific to reactive database systems [69]. While this may be a good choice for database people, it is not necessarily so for programmers. Micro-workflow targets object-oriented software developers and consequently focuses on techniques familiar to them. Second, TriGS*flow* still provides a monolithic workflow solution. The organizational, informational, and communication aspects are an integral part of the workflow model. Framework users have to use the whole package, including the TriGS infrastructure providing the active extension for GemStone. In contrast, the micro-workflow core provides basic workflow functionality that developers extend through composition. And finally, TriGS*flow* provides intrinsic support for human workers. Micro-workflow implements processes that involve application objects and a separate component (e.g., the worklist component discussed in Section 5.5) extends the core with the mechanisms that allow people to participate into workflows.

7.2.4 OPENFlow

OPENFlow is a joint project of the University of Newcastle upon Tyne and Nortel. The project proposes a transactional workflow system designed as a set of CORBA services [110]. OPENFlow departs from the monolithic structure adopted by workflow products. It aims at providing better support for interoperability, scalability, and flexibility. The architecture relies on the CORBA transaction service and forms the basis of Nortel's submission to the OMG workflow management facility [95].

The toolkit employs two transactional services, the workflow repository service and the workflow execution service. The repository service stores workflow schemas (workflow scripts) specified in a coordination language [109]. Schemas represent the structure of workflow applications in terms of tasks and the dependencies among them. The execution service records the inter-task dependencies of a schema in persistent objects and coordinates the execution of a workflow instance. It employs persistent shared objects and transactions to provide system-level fault tolerance. In addition to these transactional services, the toolkit also employs a graphical user interface, script servers that store the workflow specifications in text form, and workflow administration tasks for managing workflow applications.

The process model consists of task controllers and tasks as illustrated in Figure 7.2. Task controllers collectively maintain the structure of the workflow application and task status information. Tasks can be either primitive, genesis, or compound. Primitive tasks corresond to basic actions. Genesis tasks implement on-demand instantiation. Compound tasks support the recursive composition of tasks. The model requires genesis tasks since the execution service first instantiates the entire workflow schema and then executes it. A genesis task provides a means to delay instantiation until run time. The execution environment provides high level graphical tools for specifying and controlling the execution of applications.

Like the research reported in this thesis, OPENFlow identifies the monolithic structure as a problem of current workflow systems and attempts to provide a solution. One of its main objectives is to provide a composition and execution environment for long-running distributed applications using middleware services. Wheater and colleagues [133] also suggest that in OPENFlow the administrative applications can be implemented on top of the base system, as workflow applications themselves (the approach taken by the Mentor-lite system—see Section 7.1.1). But their system focuses on managing the execution of distributed applications and incorporating fault-tolerance into workflow applications through persistent objects and transactions. Micro-workflow shares the ideas of a non-monolithic workflow architecture with ad-

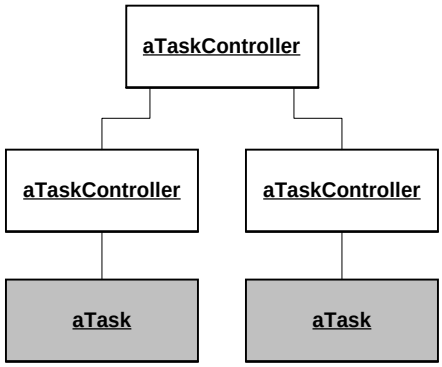


Figure 7.2: The OPENFlow task model consists of task and task controllers.

vanced features as separate components. However, it focuses on extending the core through composition, and customizing the architecture through techniques familiar to object-oriented developers. I discussed the OPENFlow/Nortel solution in more detail elsewhere [73].

7.3 Dynamic Changes

Several studies have identified the poor support for dynamic changes (also referred to as adaptive workflow, ad hoc workflow or flexible workflow) as one of the major shortcomings of current workflow products [2]. Consequently, support for dynamic changes is the subject of extensive research. While the research reported in this thesis does not specifically target this issue, it shares several characteristics with other projects that aim at supporting dynamic changes.

7.3.1 MOBILE

One of the goals of the MOBILE project at the University of Erlangen-Nürnberg is to provide a flexible workflow management system. To better understand the problem, this project proposes a classification scheme for the flexibility of workflow management applications [55]. According to this scheme, flexibility by selection enables users to choose between different execution paths. Likewise, flexibility by adaption refers to the modification of process definitions.

MOBILE employs several mechanisms to provide for both classes of flexibility. A descriptive workflow model with independent perspectives (functional, behavioral, informational, organizational and operational) [62] accommodates flexibility by selection. Likewise, a model that maintains a distinct separation

between the process modeling part (i.e., process definition) and the process execution part (i.e., process instances) accommodates flexibility by adaption. Micro-workflow facilitates both types of flexibility through late binding (e.g., between Procedure and ProcedureActivation, and between SubworkflowProcedure and the remote subworkflow) and the separation of process definition from its execution.

7.3.2 Obligations

In the related area of situated work, Douglas Bogia's PhD thesis focuses on flexible tasks within an open, active coordination environment [12]. He proposes Obligations, a CSCW environment that supports a wide range of task types, spanning from well defined to loosely defined to incomplete. An "obligation" provides a mechanism allowing an obligator to request work by one or more obligatees. Each obligation has a network of sub-obligations and stages. Sub-obligations provide a definition of the sub-activities that must be carried out to complete the obligation. Likewise, stages represent individual steps of an obligation.

Obligations place few restrictions on how users construct or alter the network. The system supports at least four styles: top-down, bottom-up, forward specification and backward specification. Users build obligations by layering templates and an obligation can "inherit" from multiple templates. Templates act as meta-classes and therefore changes made to a template affect all the obligation instances inheriting from it. A mechanism based on template versions and surrogates enables the system to support an entire continuum of modifications, from local to global. Local modifications affect a single task description, while global modifications change several instances simultaneously.

The Obligations environment targets two classes of users, end users and environment programmers. End users employ a visual programming language to create new templates and model requests and promises within and across collaborative contexts. This language is built on top of an object-oriented framework that supports the behavior of obligations. Programmers modify and extend the framework's default behavior with standard object-oriented techniques.

The technology supporting situated work helps people overcome communication problems over time and collaboration problems over distance. Bogia concentrates on supporting a gamut of collaborations between people, from free-form to well-defined. Human actors that construct, modify, and combine obligations templates are key players in the Obligations environment. But as I discussed in Section 2.3, while situated work explicitly omits step definition and inter-step coordination, workflow explicitly includes them. Micro-

workflow focuses on guidance and automation mechanisms for performing structured tasks in the workflow domain. These tasks involve application objects and, with the support of a separate component, even people. Bogia's thesis suggests software processing entities as participants alongside humans as a direction of future research.

7.3.3 Endeavors

Gregory Bolcer addresses the problem of adaptive workflow in his PhD thesis [13]. He proposes Endeavors, a workflow support system that is a successor of Teamware [140]. Endeavors builds on the insights gained from its predecessor. It leverages the Internet, the World Wide Web and the Java programming language, technologies which were not mature or even available during the Teamware project.

Endeavors focuses on providing a customizable and flexible environment for workflow definition, modeling and enactment, as illustrated in Figure 7.3. It enhances Teamware's category object model to offer better support for dynamic changes. The enhanced model provides five top-level categories that correspond to activities, artifacts, resources, logical groups of workflow objects, and dependencies. The system permits dynamic type definition and late binding of resources.

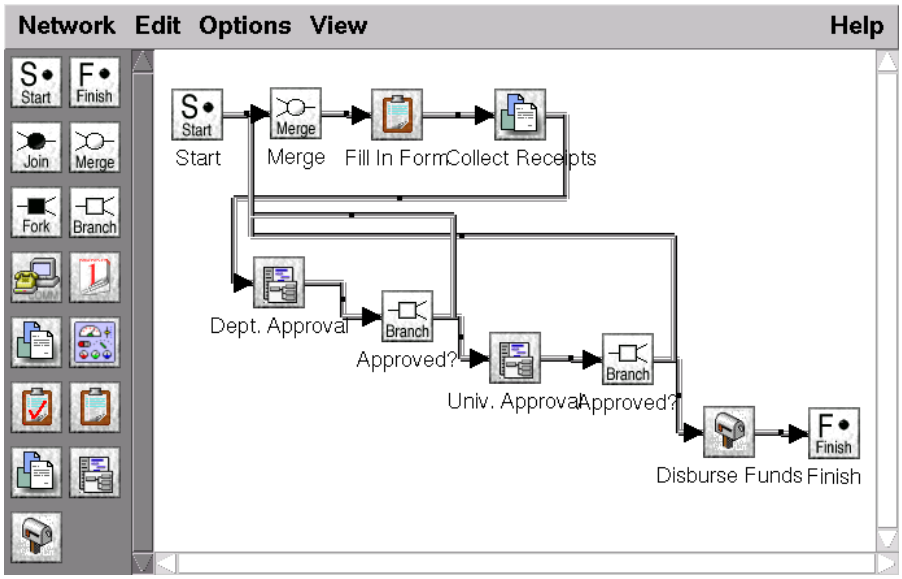


Figure 7.3: The Endeavors activity network editor.

In Endeavors, scripts (referred to as handlers) provide the behavior of workflow objects. Objects respond to events by locating, loading, and then executing the appropriate handler. Since the handlers are

bound at run time, the system may change them dynamically during the process execution. This late binding represents the crux of the Endeavors' dynamic process object model. In the object-oriented community, several studies find the dynamic object model architectural style well-suited for problems that demand a great deal of flexibility [66, 35]. The key characteristics of this architecture have been documented as software patterns—*Type Object, Property, Strategy*—and are now part of the vocabulary of object-oriented developers (Appendix A). Micro-workflow also uses this architectural style, which yields a dynamic workflow model based on proven solutions [75].

7.3.4 CRISTAL

One of the objectives of the CRISTAL (Cooperating Repositories and an Information System for Tracking Assembly Lifecycles) project is providing support for scientific workflows in the context of the Compact Muon Solenoid (CMS) experiment at CERN [80]. Workflow management in scientific and engineering applications has diferent requirements than workflow management in business applications. CRISTAL focuses on providing a flexible model that can satisfy the design constraints of time (the process spans over five years), evolution (record the physical changes to facilitate calibration) and viewpoint (calibration, maintenance, and experiment system management) of the CMS experiment.

The focus on the management and coordination of the process and the context in which scientific data is obtained makes the CRISTAL system different than other workflow systems. Therefore, Zsolt Kováks' PhD thesis [71] proposes a meta-data based design. This approach provides the flexibility required by scientific workflows and protects application software from the changes in the database schema. CRISTAL employs meta-data for workflow management as well as product data management.

Meta-data facilitates the integration of the process model with the product model, a critical issue in the context of the CMS experiment. The system stores the definitions of all parts that make up the detector; the definitions of the instruments used to produce parts or take measurements of parts; the production scheme of each part; and the descriptions of the tasks and activities performed on each part. The CMS experiment involves about 500,000 parts and the projected final amount of data is one Tera byte.

CRISTAL as well as the research reported in this thesis use a dynamic object-oriented approach. CRISTAL targets scientific workflows and focuses on accommodating the unique design constraints of the CMS experiment [7]. Micro-workflow concentrates on providing a generic process model that developers can use

and extend within their applications. I have explored the integration of process and product models elsewhere [74].

7.4 Related Research Summary

The research reported in this thesis as well as the projects reviewed in the previous sections represent a new generation of workflow systems. Several characteristics set apart this new generation from current workflow products:

- An increasing number of domains can benefit from workflow technology. This requires workflow functionality that is easy to integrate within other environments, systems, and applications. But most current products adopt a monolithic approach which provides an all-or-nothing solution. Emerging lightweight workflow systems, kernels, and facilities provide alternatives that are easier to integrate than these heavyweight systems.

- Current workflow products target mostly non-technical users. Besides workflow functionality they also provide sophisticated (graphical) tools that support their users. However, this increases complexity and makes these systems difficult to use as programming tools. Several research efforts have identified this problem and are working on providing workflow development environments for software developers.

- Many studies identify the lack of flexibility in current workflow systems as one of their major problems. These systems typically follow the compiler approach (include a code generation phase and separate the build time from the run time), or rely on a static process model. Current research on this subject focuses on providing flexible solutions by leveraging interpreters and dynamic object-oriented techniques.

Table 7.1 summarizes the main characteristics of the research projects reviewed in this chapter, as well as the key differences between these projects and micro-workflow. It is interesting to observe that at least three of the research projects reviewed in this thesis ($METEOR_2$, Endeavors, and OPENFlow) spawned commercial products.

Project	Characteristics	How is micro-workflow different
Mentor-lite	Basic workflow functionality at the application level; extensions providing worklist management, history management, and monitoring implemented as workflows.	The micro-workflow core provides basic workflow functionality inside object-oriented applications; micro-workflow core and components exploit object technology; components provide persistence, manual changes, federated workflow, as well as history, monitoring, and worklists.
OPERA	Flexible process support kernel that leverages database technology; a separate component of the execution engine implements inter-process communication.	Micro-workflow regards object-oriented technology as a complete architectural style; focuses on customization and integration; components implement all the advanced features.
Teamware	Workflow for technical and non-technical users; the category object model allows non-technical users to reuse and customize the data through parameterization.	Micro-workflow targets object-oriented developers. They use composition to assemble custom workflow functionality; black box reuse techniques to build workflows; and white box reuse techniques to tailor the framework.
TOWE	Class library for transactional workflow.	The framework approach provides a set of classes, but also specifies the way these classes interact and the expectations placed on programmers.
TriGS$_{flow}$	Focuses on active database techniques and the transactional aspect.	Micro-workflow lets developers to assemble custom workflow features through composition.
OPENFlow	Emphasis on fault-tolerance through persistent objects and transactions.	Developers extend the micro-workflow core through composition, and customize the architecture through techniques specific to object systems.
Mobile	Different types of flexibility achieved through independent perspectives and separation of definition and execution.	Micro-workflow provides a flexible model through late binding and the separation of “type” and “instance” sides.

continued on next page

Table 7.1: Summary of related research projects and prototypes.

Project	Characteristics	How is micro-workflow different
Obligations	Flexible CSCW environment that supports a continuum of collaboration models between humans.	Micro-workflow explicitly represents the task structure and task inter-coordination; workflows involve application objects and separate components add support for humans.
Endeavors	Enhanced category object model for increased flexibility.	Micro-workflow uses the dynamic object model architectural style.
CRISTAL	Dynamic object-oriented approach tailored to the unique characteristics of the CMS experiment.	Micro-workflow adopts a generic dynamic object model architecture.

Table 7.1: Summary of related research projects and prototypes (continued).

Chapter 8

Conclusion

This research started from the observation that current workflow systems do not provide the workflow functionality required in object-oriented applications, so developers are forced to build custom workflow solutions. Traditional workflow architectures are based on requirements and assumptions that don't hold in the context of contemporary object-oriented software development. This mismatch makes current workflow systems unsuitable for developers who need workflow within their applications.

My dissertation has described work leading to and including the development of micro-workflow, a novel workflow architecture that resolves this mismatch. While several research projects focus on a new generation of workflow architectures, I have taken a unique approach. Micro-workflow solves workflow problems through techniques specific to object systems and compositional software reuse. It aims at software developers and provides the type of workflow functionality they need in object-oriented applications. The components at the core of the architecture provide basic workflow functionality. Other components implement advanced workflow features. Software developers select the features they need and add the corresponding components to the core through composition.

I described why micro-workflow is different from workflow (Chapter 3). I showed how to build the core framework (Chapter 4) and components for history, persistence, monitoring, manual intervention, worklists, and federated workflow (Chapter 5).

I used three case studies to demonstrate that developers can craft customized workflow solutions with the micro-workflow framework (Chapter 6). I built applications that implement workflows for reviewing proposals (Section 6.1), treating strep throat (Section 6.2), and tracking the treatment of newborns (Section 6.3). Then I evaluated the flexibility provided by this approach in terms of the amount of effort required to add new features, the breakage of existing framework components, and the run time overhead. The metrics show

that the cost of extending the architecture with new features is not high.

Finally I compared micro-workflow with other research projects that focus on similar issues (Chapter 7). I concluded that micro-workflow belongs to a new generation of workflow systems that shifts from an end-user application to building flow-independent applications; focuses on integration with other environments; and leverages various techniques to achieve flexibility and accommodate changing requirements.

8.1 Summary of Contributions

This dissertation makes a number of primary contributions:

Demonstrates that object technology provides a complete architectural style for workflow systems The abstractions proposed in this thesis enable software developers to build and execute workflows by leveraging the three characteristics of objects—encapsulation, inheritance, and polymorphism. I have shown how developers define workflows by instantiating, configuring, and connecting objects, and how workflow enactment parallels object instantiation. Since all abstractions correspond to objects, developers customize and evolve the architecture like any other object system.

Offers an alternative to heavyweight workflow architectures Micro-workflow shifts the focus from packaging a comprehensive set of features to keeping things simple. The micro-workflow core provides basic workflow functionality, enabling software developers to define and execute workflows. Its execution, process, and synchronization components encapsulate the design decisions about workflow enactment, workflow definition, and activity synchronization. This yields a lightweight core that is easy to understand, customize, and integrate with other systems, frameworks, and applications.

Demonstrates that micro-workflow can be extended with advanced workflow features Micro-workflow allows software developers to add features typical of workflow systems by adding components to the core. This characteristic enables the architecture to grow and accommodate new functionality. I have shown how to add components that provide functionality in six different directions. Given the diversity of features provided by these components, I am confident that developers can follow the same path to add other features.

Proves that the architecture can be built The object-oriented framework described in Chapters 4 and 5

shows how to represent the abstractions provided by the micro-workflow architecture with objects. The framework implements the micro-workflow components with the Smalltalk programming language, the GemStone/S object-oriented persistent store, and the Opentalk distributed application architecture.

Shows that the architecture provides a viable solution The ability to add features by adding components incurs design and run time costs. The design cost takes into account the new classes and messages required to add a feature, and the breakage of existing classes. Likewise, the run time cost takes into account the number of additional message sends required to support pluggable components. These metrics quantify the flexibility provided by this approach. Most developers would find that the numbers reported in Sections 6.4– 6.5 (although guidelines) fit their budget.

In addition to the primary contributions discussed above, the thesis makes the following secondary contributions:

Teaches developers how to build micro-workflow components Unlike traditional workflow architectures, micro-workflow allows software developers to add new features by adding new components. A significant part of this thesis focuses on designing and implementing nine micro-workflow components with object technology. This offers a road map for customizing the existing components and crafting other components, which are two of the main reasons for using micro-workflow.

Teaches developers how to use the architecture Chapter 6 evaluates the architecture by means of three case studies. But besides providing a qualitative and quantitative evaluation, it also shows how to use micro-workflow to build applications implementing real workflows. The case studies serve as a starting point for implementing workflows with various requirements.

Provides documented workflow examples The workflow literature contains very few workflow examples. Additionally, obtaining examples is usually hard since companies don't want to reveal their internal processes. But without examples researchers can't test ideas, evaluate solutions, and choose between alternatives. The case studies discussed in Chapter 6 provide three workflow examples with different requirements.

8.2 Open Issues and Future Work

The research reported in this thesis doesn't address the following issues, which should be considered in future work:

Security The workflow run time data represents sensitive information. A workflow system should control the access to this data. Security concerns have to integrate with the security infrastructure of the enterprise. Typically the infrastructure enforces enterprise-wide security policies and provides user authentication in a uniform manner. The micro-workflow framework can be extended with a security component. Adding security involves extending the workflow session with Access Control Level (ACL) information. The security component would use ACLs to determine who can access the workflow data, who can use features like monitoring and manual intervention, etc. Federated workflow makes access control a harder problem, since workflows that span across organizational boundaries may fall under different jurisdictions.

Ad-hoc derivation during workflow execution Some research efforts study the use of workflow in uncertain environments. For example, adaptive workflow approaches based on constraint reasoning focus on carving out spaces of possible solution alternatives to process enactment through the explicit representation of constraints between tasks and roles. Others provide adaptivity through techniques from intelligent reactive control [10]. Although this thesis doesn't focus on these types of process, I believe that the architecture can accommodate them. The late binding between Procedure and ProcedureActivation instances allows the framework to start a process that builds its own definition dynamically.

Integration with process editors Developers build activity maps by instantiating and configuring the classes provided by micro-workflow components. While the micro-workflow architecture doesn't provide graphical process editors, software developers can build components for this purpose. For example, a file reader component can create the activity map (i.e., instantiate and configure the appropriate objects) starting from a process description generated with a graphical process editor. This type of component would allow developers to verify, simulate, and experiment with workflows before they implement them.

Heterogeneous federated workflow The abstractions provided by the federated workflow component hide

the workflow system executing a subworkflow behind a facade. This decreased coupling between the invoking workflow system and the invoked workflow system facilitates heterogeneous federations. I limited the discussion to homogeneous workflow systems and claimed that the abstractions provide enough separation. However, this claim is not supported by empirical evidence. Future research should pursue this direction further and test the claim.

8.3 Additional Insights

Here are some additional insights that I have gained while I worked on this research.

The Smalltalk language and development environment turned out to be an excellent choice for research and development. Several *reflective facilities* (e.g., perform:, perform:with:, doesNotUnderstand: and oneWayBecome:) enabled me to implement the PrimitiveProcedure, Worklist, and Future classes in an elegant manner. Smalltalk's *dynamic type checking* let me focus on messages instead of inheritance, thus allowing the class hierarchies to emerge naturally, as the design crystallized. The *refactoring browser* [113] and the *SUnit testing framework* were key tools for experimenting with design choices. Although the micro-workflow architecture doesn't require these features and tools, I am convinced that this research would have taken much longer without them.

The GemStone/S object-oriented persistent store takes a powerful approach. Its client-server architecture allows developers to partition their programs between the server and the client. This opens the doors to things that wouldn't be possible if all the data had to be processed on the client. Additionally, the use of the same language on both the client and the server enables the seamless relocation of functionality between them. The GemBuilder environment leverages the power of Smalltalk to hide the details of the persistence mechanism, while providing fine-grained control over features whenever developers require it. Having also used persistence solutions that store objects in relational databases, I realize that they are a far cry from what GemStone/S offers.

The federated workflow component uses the VisualWorks Opentalk distributed application architecture. I found that the STST Opentalk framework takes the right approach for building distributed applications on top of a CORBA-based architecture. The framework provides abstractions that enable objects in different Smalltalk virtual machines to send messages to each other in a transparent manner. Developers can build distributed applications without dealing with basic object adapters (BOAs), stubs, or the interface description

language (IDL), concepts that fill many pages in most books on distributed computing and CORBA.

8.4 Closing Statement

This thesis proposes the micro-workflow architecture as a better way of implementing workflow functionality within object-oriented applications. Most importantly, this thesis demonstrates that starting with a lightweight workflow architecture aimed at software developers, one can add features specific of traditional workflow systems through composition. Further, it demonstrates that object technology can provide a complete architectural style for workflow systems and flow-independent applications, allowing software developers to exploit techniques specific to object systems. I believe that the approach described in this dissertation will have broad application and could be usefully adopted by others.

Appendix A

Software Patterns

This appendix provides thumbnail descriptions for the patterns mentioned in the previous chapters. These descriptions are excerpts from the published material referenced afterwards. The order is alphabetical.

Composite composes objects intro tree structures to represent part-whole hierarchies. This pattern describes how to use recursive composition so that clients don't have to make this distinction and can treat individual objects and compositions of objects uniformly [39].

Decorator enables developers to dynamically attach additional responsibilities to an object. Decorators provides a flexible alternative to subclassing for extending functionality [39].

Execute Around Method represents pairs of actions that have to be taken together [8].

Facade minimizes the communication and dependencies between subsystems by providing a unified interface to a set of interfaces [39].

Manager encapsulates management of the instances of a class into a separate manager object. This allows for variation of management functionality independent of the class and for reuse of the manager for different object classes [120].

Null Object provides a surrogate for another object that shares the same interface but does nothing. Thus, it encapsulates the implementation decisions of how to do nothing and hides those details from its collaborators [136].

Observer defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically [39].

Property provides runtime mechanisms for accessing, altering, adding, and removing attributes at runtime [35].

Proxy provides a surrogate or placeholder for another object to control access to it [39].

Singleton ensures a class has only one instance and provides a way to access this instance [39].

Strategy defines a family of algorithms, encapsulates each one in an object, and makes them interchangeable. This solution allows the algorithm vary independently from the clients that use it [39].

Type Object decouples instances from their classes so that those classes can be implemented as instances of a class. *Type Object* allows new classes to be created dynamically at runtime, lets a system provide its own type-checking rules, and can lead to simpler and smaller systems [65].

Variable Access, Direct and Indirect Get and set an instance variable's value directly, or only through a Getting Method and a Setting Method [8].

Appendix B

The Micro-Workflow Framework

This appendix provides the UML class diagrams for the components of the micro-workflow framework discussed in Chapters 4 and 5. In each figure the colored/shaded classes belong to other framework components.

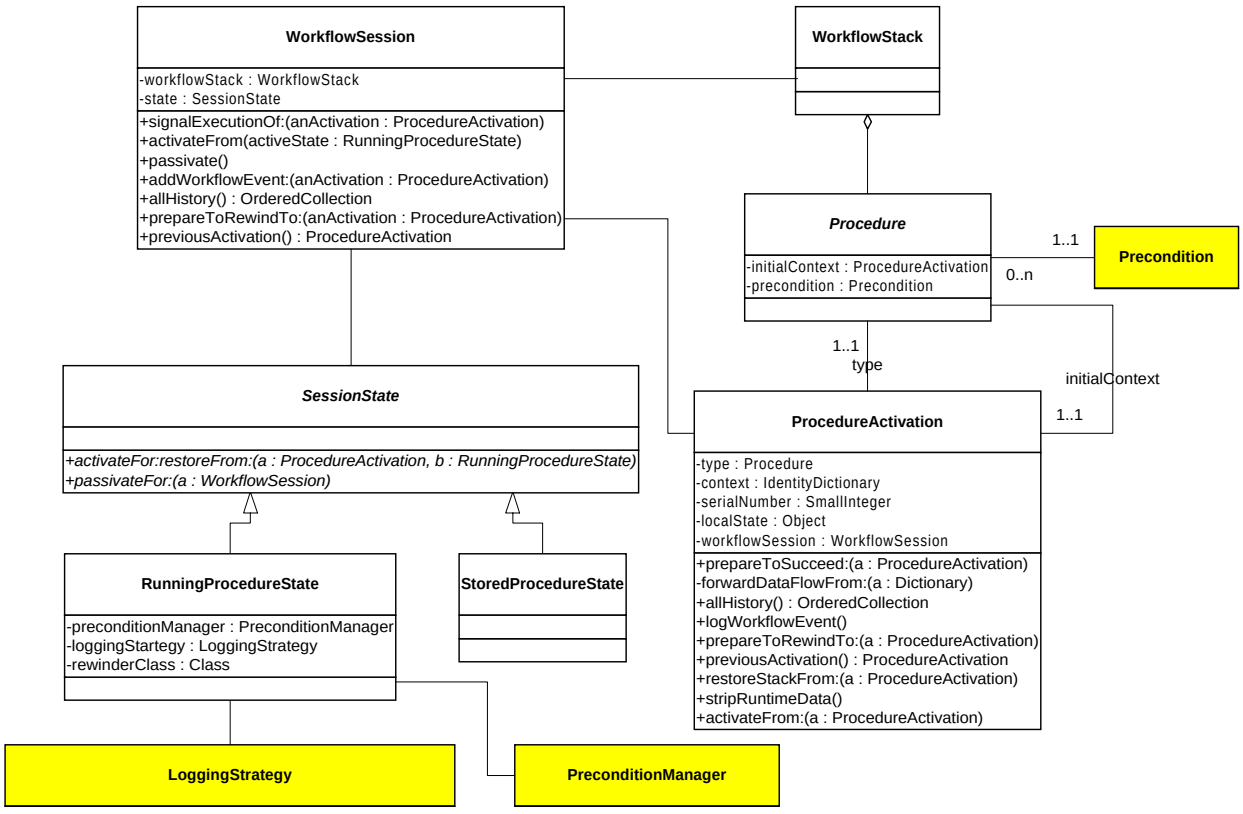


Figure B.1: The micro-workflow execution component.

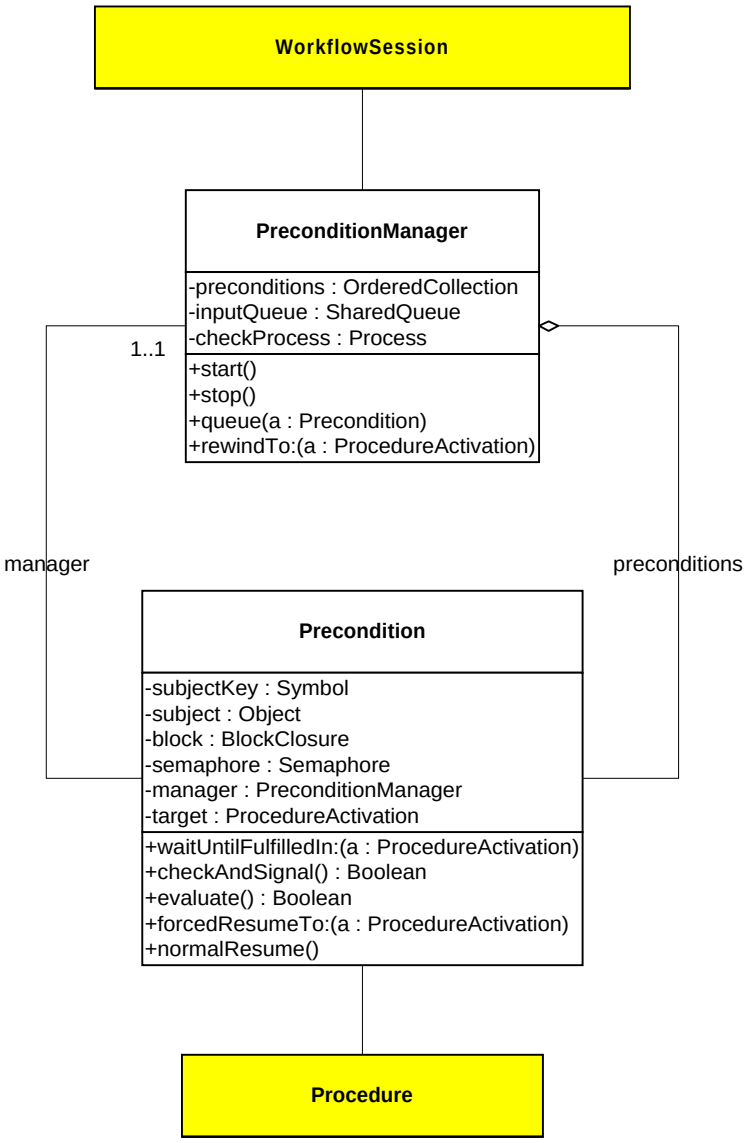


Figure B.2: The micro-workflow synchronization component.

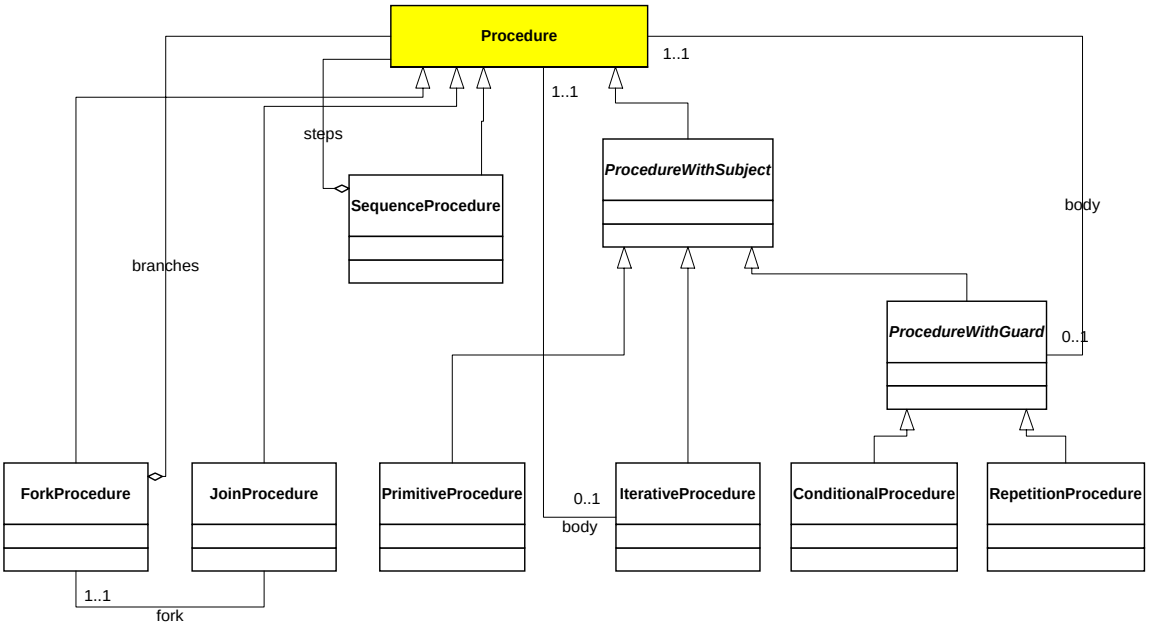


Figure B.3: The micro-workflow process component.

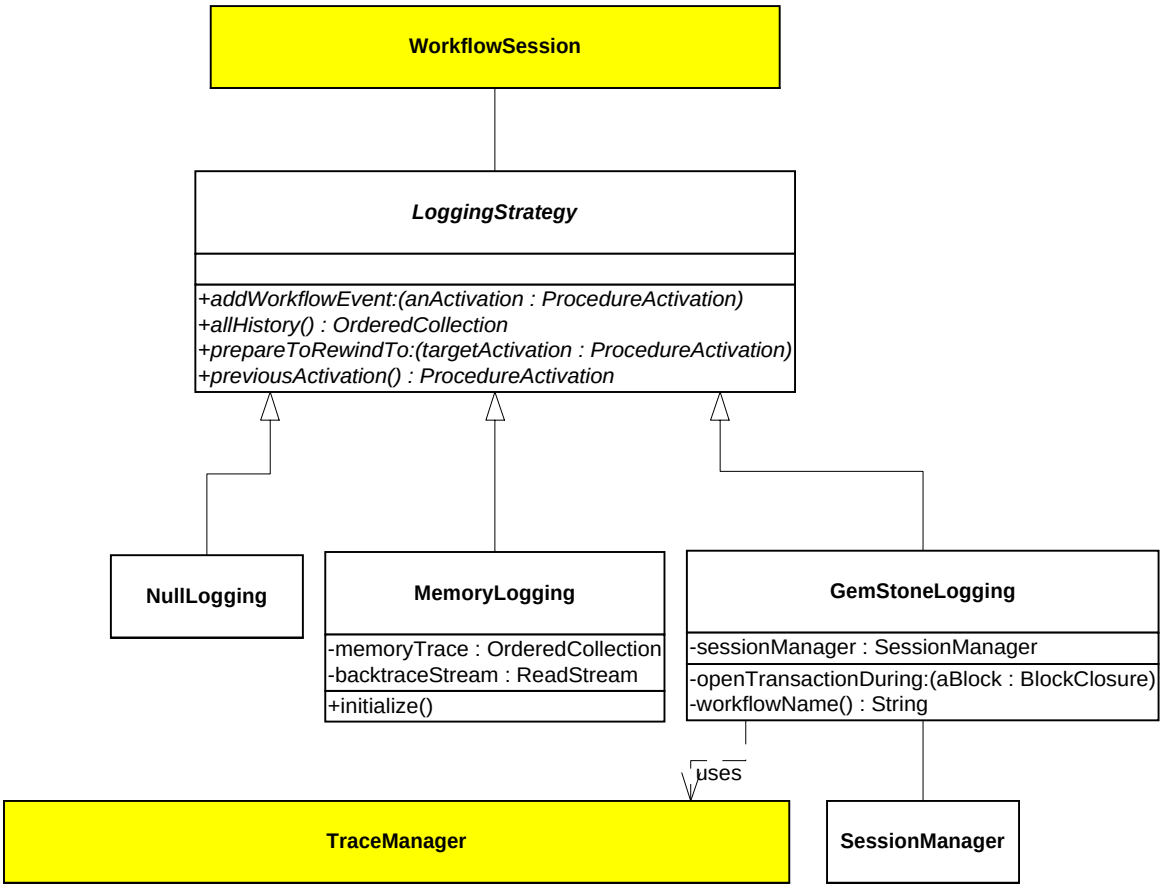


Figure B.4: The micro-workflow history component.

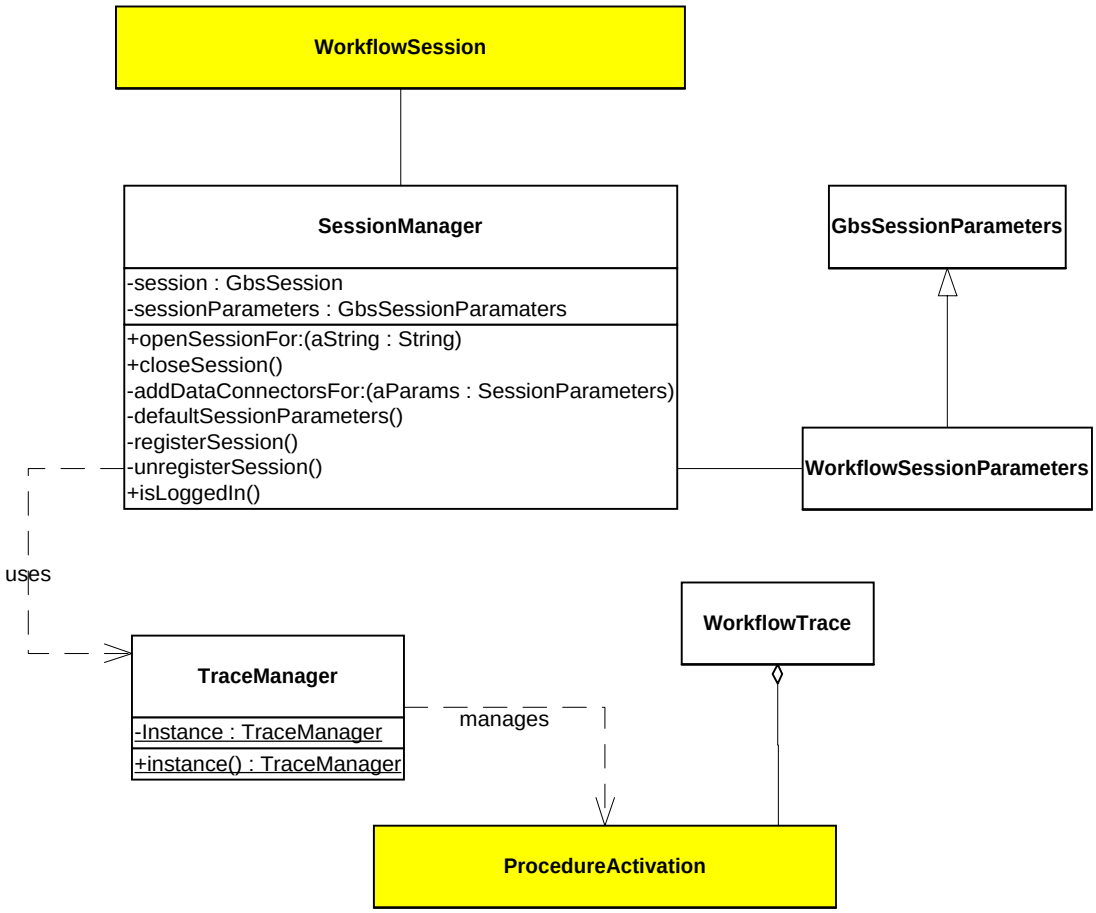


Figure B.5: The micro-workflow persistence component, client side.

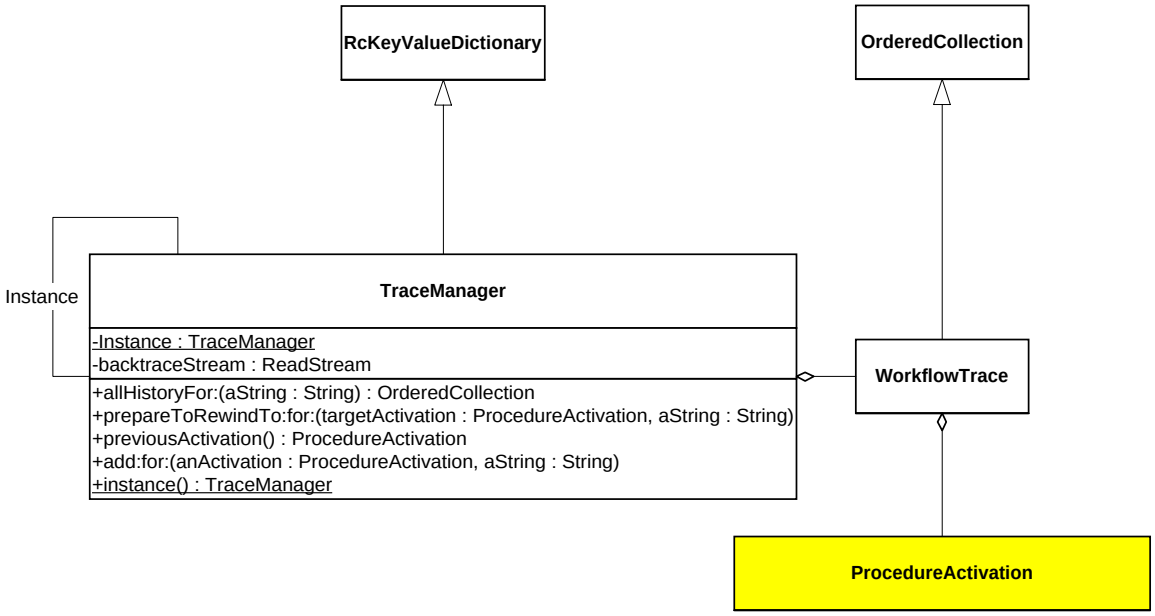


Figure B.6: The micro-workflow persistence component, server side.

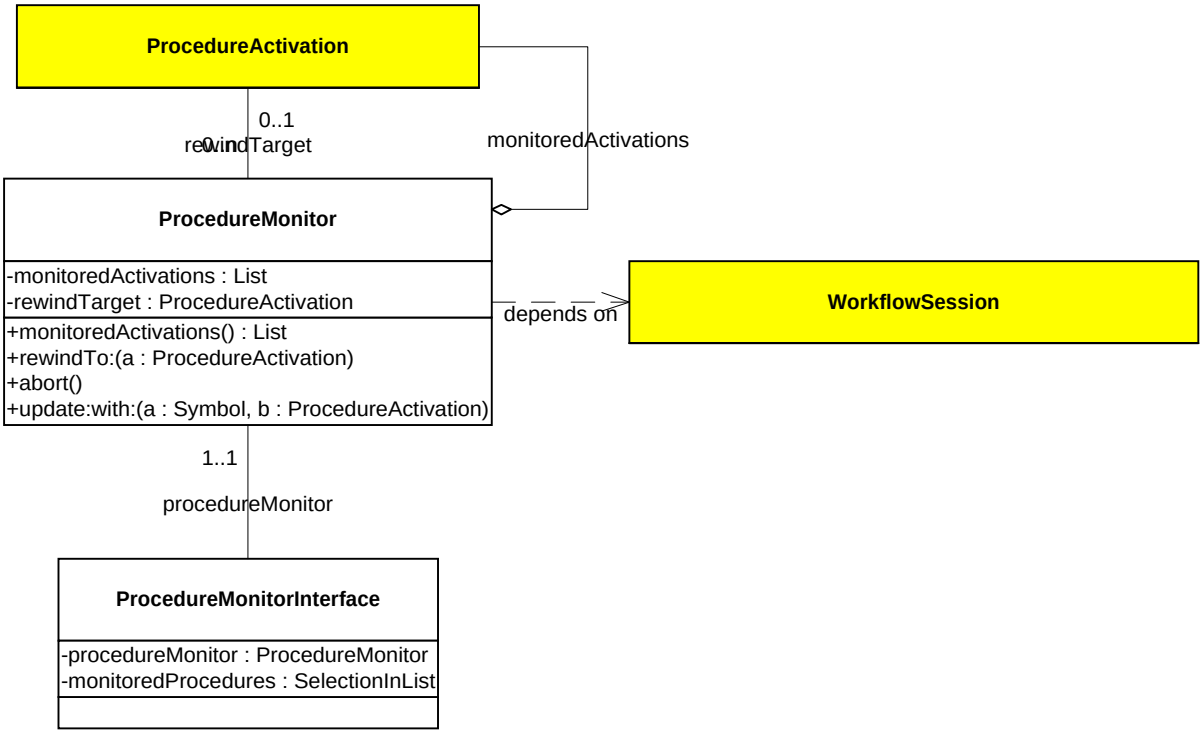


Figure B.7: The micro-workflow monitoring component.

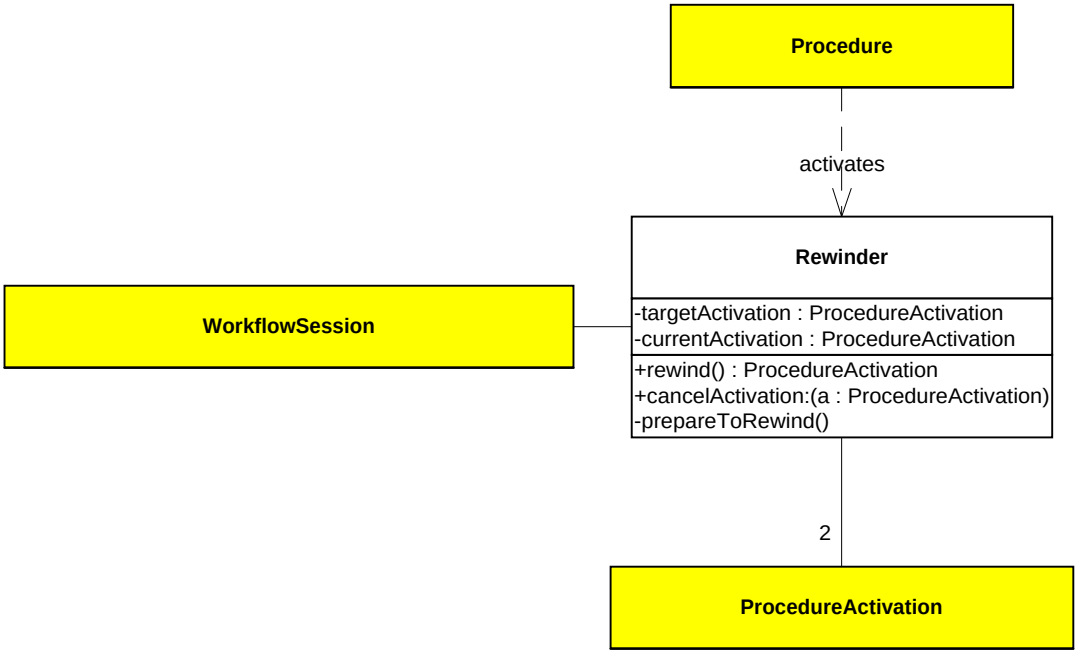


Figure B.8: The micro-workflow manual intervention component.

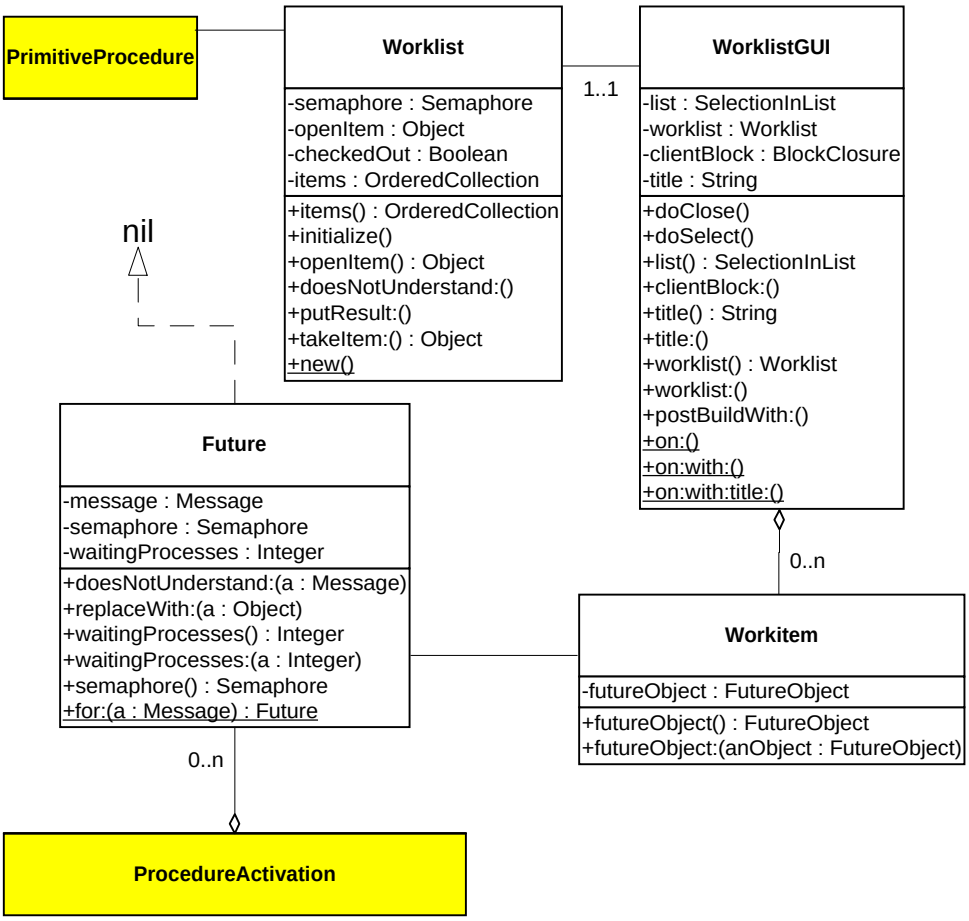


Figure B.9: The micro-workflow worklist component.

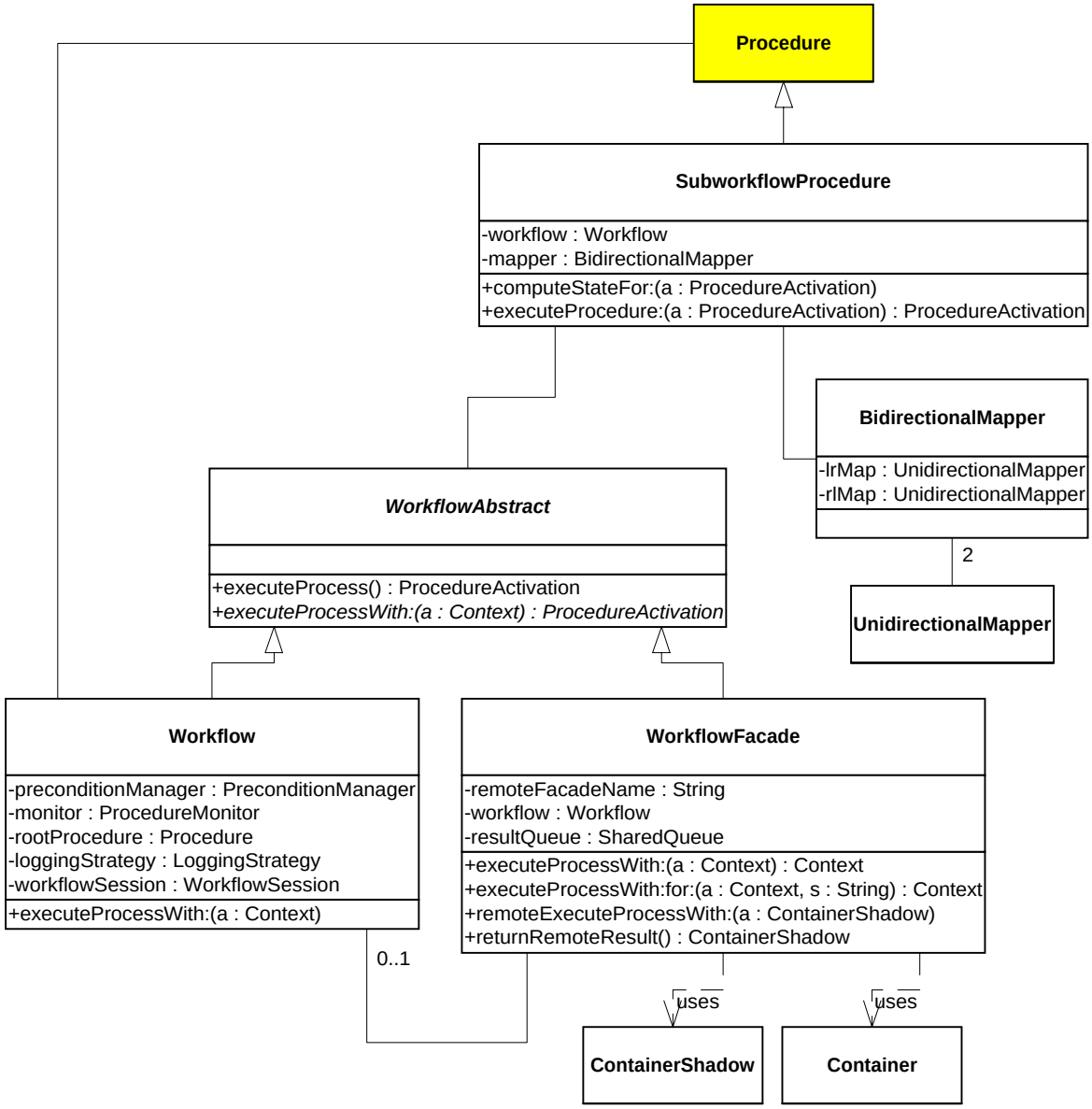


Figure B.10: The micro-workflow federated workflow component.

References

[1] Gul A. Agha. *Actors—A Model of Concurrent Computation is Distributed Systems*. The MIT Press, Cambridge, Massachusetts, 1986.

[2] G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems, 1997. Available on the Web at `http://www.almaden.ibm.com/cs/exotica/wfmsys.ps`.

[3] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, R. Günthör, and M. Kamath. EXOTICA/FMQM: A persistent message-based architecture for distributed workflow management. In *Proc. IFIP WG8.1 Working Conference on Information Systems Development for Decentralized Organizations*, Trondheim, Norway, August 1995.

[4] Gustavo Alonso, Claus Hagen, Hans-Jörg Schek, and Markus Tresch. *Towards a Platform for Distributed Application Development*, pages 195–221. Volume 164 of Dğaç et al. [27], August 1998. Available on the Web at `http://www.inf.ethz.ch/department/IS/iks/publications/ahst97b.html`.

[5] Scott W. Ambler. The design of a robust persistence layer for relational databases. White Paper, October 1999. On the Web at `http://www.ambysoft.com/persistenceLayer.html`.

[6] Francis Anderson and Ralph Johnson. The Objectiva telephone billing system. MetaData Pattern Mining Workshop, Urbana, IL, May 1998. Available on the Web at `http://www.joeyoder.com/Research/metadata/UoI98MetadataWkshop.html`.

[7] A. Barry, N. Baker, J.-M. Le Goff, R. McClatchey, and J.-P. Vialle. Meta-data based design of workflow systems. OOPSLA Meta-data workshop, Vancouver, BC, October 1998. Available on the Web at `http://www.joeyoder.com/Research/metadata/OOPSLA98MetaDataWkshop.html`.

[8] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, October 1996.

[9] Lucy Berlin. When objects collide: Experiences with reusing multiple class hierarchies. In *ECOOP/OOPSLA'90 Proceedings*, pages 181–193, October 1990.

[10] Pauline M. Berry and Karen L. Myers. Adaptive process management: An ai perspective. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[11] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability*, volume 2 of *Frontier Series*. Addison-Wesley, 1989.

[12] Douglas Paul Bogia. *Supporting Flexible, Extensible Task Descriptions in and Among Tasks*. PhD thesis, University of Illinois at Urbana-Champaign, 1995. Available on the Web from `ftp://ftp.cs.uiuc.edu/pubs`.

[13] Gregory Alan Bolcer. *Flexible and Customizable Workflow on the WWW*. PhD thesis, University Of California, Irvine, 1998.

[14] Frank Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. John Wiley & Sons, July 1996.

[15] Christoph Bussler. Enterprise-wide workflow management. *IEEE Concurrency*, pages 32–43, July–September 1999.

[16] Steinar Carlsen. *Conceptual Modeling and Composition of Flexible Workflow Models*. PhD thesis, Department of Computer and Information Science, Faculty of Physics, Informatics and Mathematics, Norwegian University of Science and Technology, 1997. Available on the Web.

[17] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Deriving active rules for workflow enactment. In *Proc. 7th International Conference on Database and Expert Systems Applications*, Lecture Notes in Computer Science, pages 94–110. Springer-Verlag, 1996.

[18] Dickson K. W. Chiu, Kamalakar Karlapalem, and Qing Li. Exception handling with workflow evolution in ADOME-WfMS: a taxonomy and resolution techniques. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[19] Andrzej Cichocki, Abdelsalam (Sumi) Helal, Marek Rusinkiewicz, and Darrell Woelk. *Workflow and Process Automation—Concepts and Technology*. Kluwer Academic Publishers, 1998.

[20] Cincom Systems, Inc. *VisualWorks Opentalk Application Developer's Guide*, 1999. Part Number P46–0131–00, Software Release 5i.1.

[21] The Workflow Management Coalition. Process definition model and interchange language, October 1999. Document WfMC-TC-1016P v1.1.

[22] Umeshwar Dayal, Quiming Chien, and Tak W. Yan. *Workflow Technologies Meet the Internet*, pages 423–438. Volume 164 of Doğaç et al. [27], August 1998.

[23] L. Peter Deutsch. *Design Reuse and Frameworks in the Smalltalk–80 System*, chapter 3, pages 57–72. Volume 2 of Biggerstaff and Perlis [11], 1989.

[24] Martine Devos and Michel Tilman. A repository-based framework for evolutionary software development. MetaData Pattern Mining Workshop, Urbana, IL, May 1998. Available on the Web at `http://www.joeyoder.com/Research/metadata/UoI98MetadataWkshop.html`.

[25] Edsger W. Dijkstra. Guarded commands, nondeterminancy, and the formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[26] Guido Dinkhoff, Volker Gruhn, Armin Saalmann, and Michael Zielonka. *Entity-Relationship Approach–ER'94, Business Modelling and Re-engineering*, chapter Business Process Modeling in the Workflow Management Environment *Leu*, pages 46–63. Number 881 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[27] Asuman Doğaç, Leonid Kalinichenko, M. Tamer Özsu, and Amit Sheth, editors. *Workflow Management Systems and Interoperability*, volume 164 of *NATO Advanced Science Institutes (ASI), Series F: Computer and Systems Sciences*. Springer-Verlag, August 1998.

[28] David Edmond and Arthur H. M. ter Hofstede. Achieving workflow adaptability by means of reflection. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[29] Clarence Ellis and Gary J. Nutt. Computer science and office information systems. *ACM Computing Surveys*, 12(1):27–60, March 1980.

[30] Clarence A. Ellis and Gary J. Nutt. *Modeling and Enactment of Workflow Systems*, pages 1–16. Invited paper.

[31] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, second edition, 1994.

[32] Mohamed Fayad, Douglas C. Schmidt, and Ralph E. Johnson, editors. *Implementing Application Frameworks: Object-Oriented Frameworks at Work*. John Wiley & Sons, 1999.

[33] Brian Foote. Designing to facilitate change with object-oriented frameworks. Master's thesis, University of Illinois at Urbana-Champaign, 1988.

[34] Brian Foote and Ralph E. Johnson. Reflective facilities in Smalltalk–80. In *Proceedings of OOPSLA'89*. ACM, 1989.

[35] Brian Foote and Joseph Yoder. Metadata and active object-models. OOPSLA Meta-data workshop, Vancouver, BC, October 1998. Available on the Web at `http://www.joeyoder.com/Research/metadata/OOPSLA98MetaDataWkshop.html`.

[36] Martin Fowler. *Analysis Patterns—Reusable Object Models*. Addison-Wesley Object-Oriented Software Engineering Series. Addison-Wesley, 1997.

[37] Martin Fowler and Kendall Scott. *UML Distilled—Applying the Standard Object Modeling Language*. Object Technology Series. Addison-Wesley, June 1997.

[38] Svend Frølund. *Coordinating Distributed Objects—An Actor-Based Approach*. The MIT Press, Cambridge, Massachusetts, 1996.

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[40] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proc. 17th International Conference on Software Engineering, Seattle, WA*, April 1995.

[41] GemStone Systems, Inc. *GemBuilder for VisualWorks*, July 1996. Version 5.0.

[42] GemStone Systems, Inc. *GemStone Programming Guide*, July 1996. Version 5.0.

[43] Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases, an International Journal*, 3:119–153, 1995. Available on the Web at `ftp://ftp.gte.com/pub/dom/reports/GEOR95a.ps`.

[44] Dimitrios Georgakopoulos, Wolfgang Prinz, and Alexander L. Wolf, editors. *Proceedings of the Joint Conference on Work Activities Coordination and Collaboration (WACC)*, volume 24 of *Software Engineering Notes*. ACM, March 1999.

[45] Dimitrios Georgakopoulos and Aphrodite Tsalgatidou. *Technology and Tools for Comprehensive Business Process Lifecycle Management*, pages 356–395. Volume 164 of Doğaç et al. [27], August 1998.

[46] Andreas Geppert, Markus Kradolfer, and Dimitrios Tombros. Federating heterogeneous workflow systems. Technical Report 05, Department of Computer Science, University of Zürich, 1998.

[47] Adele Goldberg and David Robson. *Smalltalk–80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[48] Claus Hagen and Gustavo Alonso. Beyond the black box: Event-based inter-process communication in process support systems. Technical Report 303, Swiss Federal Institute of Technology (ETH), Department of Computer Science, Zürich, Switzerland, 1999. Also in the Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS), Austin, TX, USA, June 1999. Available on the Web at `http://www.inf.ethz.ch/department/IS/iks/publications/ha99.html`.

[49] Claus Johannes Hagen. *A Generic Kernel for Reliable Process Support*. PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1999.

[50] Robert Halstead, Jr. MultiLISP: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, October 1985.

[51] Michael Hammer and James Campy. *Reengineering the Corporation—A Manifesto for Business Revolution*. Harper Business, 1993.

[52] Michael Hammer, W. Gerry Howe, Vincent J. Kruskal, and Irving Wladawsky. Very high level programming language for data processing applications. *Communications of the ACM*, 20(11):832–840, November 1977.

[53] Yanbo Han, Amit Sheth, and Christoph Bussler. A taxonomy of adaptive workflow management. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[54] Rex Hartson. User-interface management control and communication. *IEEE Software*, pages 62–70, January 1989.

[55] Petra Heinl, Stefan Horn, Stefan Jablonski, Jens Neeb, Katrin Stein, and Michael Teschke. A comprehensive approach to flexibility in workflow management systems. In Georgakopoulos et al. [44], pages 79–88.

[56] Hewlett-Packard. HP Changengine–Business Process Management for the Enterprise. Available on the Web at `http://www.ice.hp.com/cyc/af/00/index.html`.

[57] David Hollingsworth. *The Workflow Reference Model*. Workflow Management Coalition, Avenue Marcel Thiry 204, 1200 Brussels, Belgium, 1995. Available on the Web at `http://www.aiim.org/wfmc/`.

[58] Richard Hull, Francois Llirbat, Eric Simon, Jianwen Su, Guozhu Dong, Bharat Kumar, and Gang Zhou. Declarative worflows that support easy modification and dynamic browsing. In Georgakopoulos et al. [44], pages 69–78.

[59] Richard Hull, Francois Llirbat, Jianwen Su, Guozhu Dong, Bharat Kumar, and Gang Zhou. Adaptive execution of workflow: Analysis and optimization. Bell Labs working paper, October 1998. Available on the Web from `http://www-db.research.bell-labs.com/projects/vortex/`.

[60] InConcert, Inc., Cambridge, MA. *Teoss 2000 with TXM Option, Product Data Sheet*, 1999. Available on the Web from `http://www.inconcertsw.com/`.

[61] Innosoft directory services. Available on the Web from `http://www.innosoft.com/directory_solutions/`.

[62] Stefan Jablonski and Christoph Bussler. *Workflow Management—Modeling Concepts, Architecture and Implementation*. International Thomson Computer Preess, 1996.

[63] Michael Jackson and Graham Twaddle. *Business Process Implementation–Building Workflow Systems*. Addison-Wesley, 1997. ISBN 0-201-177684.

[64] Javablend. Sun Microsystems, Inc. Available on the Web from `http://www.sun.com/software/javablend/`.

[65] Ralph Johnson and Bobby Woolf. *Type Object*, chapter 4, pages 47–65. In Martin et al. [79], October 1997.

[66] Ralph E. Johnson. Dynamic object model. Work in progress; available on the Web at `http://st-www.cs.uiuc.edu/users/johnson/DOM.html`.

[67] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings of OOPSLA'92*, 1992. Available on the Web at `ftp://st.cs.uiuc.edu/pub/papers/HotDraw/documenting-frameworks.ps`.

[68] Ralph E. Johnson and Brian Foote. Designing reuseable classes. *Journal of Object-Oriented Programming*, June–July 1991.

[69] Gerti Kappel, Stefan Rausch-Schott, and Werner Retshitzegger. *A Framework for Workflow Management Systems Based on Objects, Rules and Roles*, chapter TBP. In Fayad et al. [32], 1999. Available on the Web at `ftp://ftp.ifs.uni-linz.ac.at/pub/publications/1998/1698.ps.zip`.

[70] James G. Kobielus. *Workflow Strategies*. IDG Books Worldwide, 1997.

[71] Zsolt Kováks. *The Integration of Product Data with Workflow Management Through a Common Data Model*. PhD thesis, Faculty of Computer Studies and Mathematics, University of the West of England, Bristol, April 1999.

[72] Frank Leymann and Dieter Roller. *Production Workflow—Concepts and Techniques*. Prentice-Hall, Upper Saddle River, New Jersey, 2000.

[73] Dragoş-Anton Manolescu and Ralph E. Johnson. Patterns of workflow management facility. Available on the Web at `http://www.uiuc.edu/ph/www/manolesc/Workflow/PWFMF/`.

[74] Dragoş-Anton Manolescu and Ralph E. Johnson. A proposal for a common infrastructure for process and product models. In *OOPSLA Mid-year Workshop on Applied Object Technology for Implementing Lifecycle Process and Product Models*, Denver, Colorado, July 1998. Available on the Web from `http://micro-workflow.com/`.

[75] Dragoş-Anton Manolescu and Ralph E. Johnson. Dynamic object model and adaptive workflow. OOPSLA'99 Metadata and Active Object-Model Pattern Mining Workshop, November 1999. Available on the Web from `http://micro-workflow.com/`.

[76] Dragoş-Anton Manolescu and Ralph E. Johnson. A micro workflow framework for compositional object-oriented software development. OOPSLA'99 Workshop on the Implementation and Application of Object-Oriented Workflow Management Systems II, November 1999. Available on the Web from `http://micro-workflow.com/`.

[77] Dragoş-Anton Manolescu and Ralph E. Johnson. A micro-workflow component for federated workflow. OOPSLA2000 Workshop on Implementation and Application of Object-Oriented Workflow Management Systems III, October 2000. Available on the Web from `http://micro-workflow.com/`.

[78] James Martin and James J. Odell. *Object-Oriented Methods—A Foundation*. Prentice Hall, second edition, 1998.

[79] Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors. *Pattern Languages of Program Design 3*. Software Patterns Series. Addison-Wesley, October 1997.

[80] R. McClatchey, Jean-Marie Le Goff, N. Baker, W. Harris, and Z. Kovács. *A Distributed Workflow and Product Data Management Application for the Construction of Large Scale Scientific Apparatus*, pages 18–34. Volume 164 of Doğaç et al. [27], August 1998.

[81] Raúl Medina-Mora, Terry Winograd, Rodrigo Flores, and Fernando Flores. The action workflow approach to workflow management technology. In *Proc. ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, Emerging technologies for cooperative work, pages 281–288, Toronto, Ontario, 1992. ACM Press.

[82] Theo Dirk Meijler, Han Kessels, Charles Vuijst, and Rine le Comte. Realising run-time adaptable workflow by means of reflection in the Baan workflow engine. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[83] C. Mohan. *Recent trends in workflow management products, standards and research*, pages 396–409. Volume 164 of Doğaç et al. [27], August 1998. Available on the Web at `http://www.almaden.ibm.com/cs/exotica/wfnato97.ps`.

[84] Peter Muth, Jeanine Weissenfels, Michael Gillmann, and Gerhard Weikum. Mentor-lite: Integrating light-weight workflow management systems within business environments (extended abstract), October 1998. Available on the Web from `http://www-dbs.cs.uni-sb.de/~mlite/`.

[85] Peter Muth, Jeanine Weissenfels, Michael Gillmann, and Gerhard Weikum. Integrating light-weight workflow management systems within existing business environments. In *Proc. 15th International Conference on Data Engineering*, Sydney, Australia, March 1999. Available on the Web from `http://www-dbs.cs.uni-sb.de/~mlite/`.

[86] Peter Muth, Jeanine Weissenfels, Michael Gillmann, and Gerhard Weikum. Workflow history management in virtual enterprises using a light-weight workflow management system. In *Proc. 9th International Workshop on Research Issues in Data Engineering*, Sydney, Australia, March 1999. Available on the Web from `http://www-dbs.cs.uni-sb.de/~mlite/`.

[87] Peter Muth, Dirk Wodtke, Jeanine Weissenfels, Gerhard Weikum, and Angelika Kotz Dittrich. *Enterprise-Wide Workflow Management Based on State and Activity Charts*, pages 281–303. Volume 164 of Doğaç et al. [27], August 1998. Available on the Web at `http://www-dbs.cs.uni-sb.de/public_html/papers/NATO-WF.ps.Z`.

[88] Hiroaki Nakamura and Ralph E. Johnson. Adaptive framework for the REA accounting model. OOPSLA'98 Business Object Workshop IV, October 1998. Available on the Web at `http://jeffsutherland.org/oopsla98/nakamura.html`.

[89] Gary J. Nutt. The evolution toward flexible workflow systems. *Distributed Systems Engineering*, 3(4):276–294, December 1996.

[90] Jeff Oakes and Ralph Johnson. The Hartford insurance framework. MetaData Pattern Mining Workshop, Urbana, IL, May 1998. Available on the Web at `http://www.joeyoder.com/Research/metadata/UoI98MetadataWkshop.html`.

[91] Objectivity online. Available on the Web from `http://www.objectivity.com/`.

[92] Objectstore. Available on the Web from `http://www.objectdesign.com/objectstore/`.

[93] ODBTalk smalltalk database framework for ODBC. Available on the Web from `http://www.ilap.com/lpc/html/body_odbtalk.html`.

[94] Joint workflow management facility—revised submission. OMG Document Number bom/98–06–07, 1998. Available on the Web at `ftp://ftp.omg.org/pub/docs/bom/98-06-07.pdf`.

[95] Workflow management facility specification. OMG Document Number bom/98–03–01, 1998. Available on the Web at `ftp://ftp.omg.org/pub/docs/bom/98-03-01.pdf`.

[96] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[97] Robert Orfali, Dan Harkey, and Jeri Edwards. *Instant CORBA*. John Wiley & Sons, 1997.

[98] Aris M. Ouksel and Jr. James Watson. The need for adaptive workflow and what is currently available on the market—perspectives from an ongoing industry benchmarking initiative. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[99] Mike Papazoglou, Alex Delis, Athman Bouguettaya, and Mostafa Haghjoo. Class library support for workflow environments and applications. *IEEE Transactions on Computers*, 46(6):673–686, June 1997.

[100] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[101] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–7, 1986.

[102] Norman W. Paton and Oscar Díaz. Active database systems. To be published in ACM Computing Surveys. Available on the Web at `http://www.cs.man.ac.uk/users/norm/papers/surveys.ps`.

[103] Santanu Paul, Edwin Park, and Jarir Chaar. Essential requirements for a workflow standard. OOPSLA'97 Business Object Workshop, 1997. Available on the Web from `http://jeffsutherland.org/oopsla97/`.

[104] Santanu Paul, Edwin Park, and Jarir Chaar. RainMan: A workflow system for the Internet. In USENIX, editor, *USENIX Symposium on Internet Technologies and Systems Proceedings, Monterey, California, December 8–11, 1997*, pages 159–170, Berkeley, CA, USA, 1997. USENIX.

[105] Charles Petrie and Sunil Sarin. Controlling the flow. *IEEE Internet Computing*, 4(3):34–36, May–June 2000.

[106] The MIT process handbook project. Available on the Web from `http://ccs.mit.edu/ph/`.

[107] Giacomo Piccinelli. Interaction modelling in federated process-centered environments. Technical Report HPL–98–54, HP Laboratories, Bristol, UK, March 1998.

[108] Roger S. Pressman. *Software Engineering—A Practitioner's Approach*. McGraw-Hill, New York, New York, third edition, 1992.

[109] Frédéric Ranno, Santosh K. Shrivastava, and Stuart M. Wheather. A language for specifying the composition of reliable distributed applications. In *Proc. 18th International Conference on Distributed Systems*, Amsterdam, The Netherlands, May 1998. Available on the Web from `http://arjuna.ncl.ac.uk/WorkflowSystem`.

[110] Frédéric Ranno, Santosh K. Shrivastava, and Stuart M. Wheather. A system for specifying and coordinating the execution of reliable distributed applications. Technical report, Department of Computing Science, University of Newcastle upon Tyne, 1998. Available on the Web at `http://arjuna.ncl.ac.uk/group/papers/p062.ps`.

[111] Stefan Rausch-Schott. TRIGS*flow*—*Workflow Management Based on Active Object-Oriented Database Systems and Extended Transaction Mechanisms*. PhD thesis, Institute of Applied Computer Science, Johannes Kepler University, Linz, Austria, February 1997. Published by Trauner Verlag, Linz, ISBN 3-85320-991-2.

[112] Don Roberts and Ralph Johnson. *Evolving Frameworks—A Pattern Language for Developing Object-Oriented Frameworks*, chapter 26. In Martin et al. [79], October 1997.

[113] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, April 1999. Available as Computer Science Technical Report #2092; on the Web at `ftp://ftp.cs.uiuc.edu/pub/dept/tech_reports/1999/UIUCDCS-R-99-2092.pdf.%gz`.

[114] Thomas Schäl. *Workflow Management Systems for Process Organizations*. Number 1096 in Lecture Notes in Computer Science. Springer-Verlag, 1996. ISBN 3-540-61401-X.

[115] Marc-Thomas Schmidt. The evolution of workflow standards. *IEEE Concurrency*, pages 44–52, July–September 1999.

[116] Amit Sheth, Krys Kochut, John Miller, Devashish Worah, Souvik Das, Chenye Lin, Devanand Palaniswami, John Lynch, and Ivan Shevchenko. Supporting state-wide immunization tracking using multi-paradigm workflow technology. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB96), Bombay, India*, September 1996.

[117] Amit Sheth and Krys J. Kochut. *Workflow Automations to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration systems*, pages 35–60. Volume 164 of Doğaç et al. [27], August 1998.

[118] Amit P. Sheth, Wil van der Aalst, and Ismailcem B. Arpinar. Processes driving the networked economy. *IEEE Concurrency*, pages 18–31, July–September 1999.

[119] Robert Signore, John Creamer, and Michael O. Stegman. *The ODBC Solution: Open Database Connectivity in Distributed Environments*. McGraw Hill, February 1995.

[120] Peter Sommerland. *Manager*, chapter 2, pages 19–28. In Martin et al. [79], October 1997.

[121] S. L. Stewart and James A. St. Pierre. Experiences with a manufacturing framework. In J. Sutherland, D. Patel, C. Casanave, G. Hollowell, and J. Miller, editors, *Business Object Design and Implementation—OOPSLA'95 Workshop Proceedings*, pages 135–150. Springer-Verlag, 1997. ISBN 3-540-76096-2.

[122] Clemens Szyperski. *Component Software—Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[123] TOPLink. The Object People, Inc. Available on the Web from `http://www.objectpeople.com/toplink/`.

[124] Ultimus, Inc., Raleigh, NC. *150 Essential Features of Workflow Automation*, October 1998. Available on the Web from `http://www.ultimus1.com/`.

[125] Ultimus, Inc., Raleigh, NC. *Ultimus v4: A Scalable, Open Architecture for Enterprise Workflow Automation*, July 1998. Available on the Web from `http://www.ultimus1.com/`.

[126] Ultimus, Inc., Raleigh, NC. *Ultimus Workflow Suite 4, Product Guide*, July 1998. Available on the Web from `http://www.ultimus1.com/`.

[127] Vijay Vaishnavi, Stef Joosten, and Bill Kuechler. Representing workflow management systems with smart objects. In *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems*, May 1996. Available on the Web at `http://www.cis.gsu.edu/~bkuechle/allsec3.html`.

[128] Versant ODDBMS architecture. Available on the Web from `http://www.versant.com/`.

[129] Cincom visualworks documentation. Cincom Systems, Inc. Available on the Web from `http://www.cincom.com/visualworks/documentation.html`.

[130] Gottfried Vossen and Mathias Weske. *The WASA Approach to Workflow Management for Scientific Applications*, pages 145–164. Volume 164 of Doğaç et al. [27], August 1998.

[131] Jeanine Weissenfels, Peter Muth, and Gerhard Weikum. Flexible worklist management in a light-weight workflow management system. Available on the Web from `http://www-dbs.cs.uni-sb.de/~mlite/`.

[132] Mathias Weske, Thomas Goesmann, Roland Holten, and Rüdiger Striemer. A reference model for workflow application development processes. In Georgakopoulos et al. [44], pages 1–10.

[133] Stuart M. Wheater, Santosh K. Shrivastava, and Frédéric Ranno. A CORBA compliant transactional workflow system for internet applications. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, The Lake District, England, September 15-18*, 1998. Available on the Web from `http://arjuna.ncl.ac.uk/WorkflowSystem`.

[134] Seth White, Maydene Fisher, Rick Cattell, Graham Hamilton, and Mark Hapner. *JDBC API Tutorial and Reference, Second Edition: Universal Data Access for the Java 2 Platform*. Java Series. Addison-Wesley, 2nd edition, June 1999.

[135] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition*. Addison-Wesley, 1986.

[136] Bobby Woolf. *Null Object*, chapter 1, pages 5–18. In Martin et al. [79], October 1997.

[137] Workflow•BPR. Available on the Web from `http://www.holosofx.com/`.

[138] Jian Yang and Mike P. Papazoglou. Interoperation support for electronic business. *Communications of the ACM*, 43(6):39–47, June 2000.

[139] Joseph W. Yoder, Ralph E. Johnson, and Quince D. Wilson. Connecting business objects to relational databases. In *Proc. 5th Pattern Languages of Programming*, Monticello, IL, August 1998. Available as Washington University Technical Report WUCS–98–25; on the Web from `http://jerry.cs.uiuc.edu/plop/plop98/`.

[140] Patrick Scott Chun Young. *Customizable Process Specification and Enactment for Technical and Non-Technical Users*. PhD thesis, University Of California, Irvine, 1994.

[141] M.D. Zisman. *Representation, Specification and Automation of Office Procedures*. PhD thesis, University of Pennsylvania, Warton School of Business, 1977.

Vita

Dragoş A. Manolescu was born in Bucharest, Romania, in 1971. In June 1990 he obtained a baccalaureate degree in mathematics from the Liceul Matematică–Fizică Nr. 1 in Bucharest, and in June 1995 he obtained an engineering degree in electronics from the Universitatea "Politehnica" of Bucharest. He had study grants in 1994 when he spent five months at the Inter-University Micro Electronics Centre in Leuven, Belgium, and in 1995 when he spent six months at the Institut National Politechnique de Grenoble in Grenoble, France.

He entered the graduate program at the University of Illinois at Urbana-Champaign in the Fall of 1995, where he held assistanships with the Division of Broadcasting (1995–1996), the National Center for Supercomputing Applications (1996–1998), and the Department of Computer Science (1998–1999). From 1995 through 1997 his research focused on hyper-media documents, and software patterns and architectures for multimedia. He received the Master of Science degree in Computer Science in 1997, under the technical guidance of Prof. Klara Nahrstedt.

In the Fall of 1996 he joined Prof. Ralph Johnson's Software Architecture Group. This group nurtured his interest in patterns, object-oriented frameworks and design, and Smalltalk. In 1997 he started to research object-oriented workflow architectures. Under the guidance of Prof. Ralph Johnson, he completed his doctoral research and defended his dissertation in October 2000.

His publications include:

- "*A Micro-Workflow Component for Federated Workflow*" (co-author with Ralph Johnson), OOPSLA 2000 Workshop on the Implementation and Application of Object-Oriented Workflow Management Systems III, October 2000, Minneapolis, MN, USA.

- "*A Micro Workflow Framework for Compositional Object-Oriented Software Development*" (co-author with Ralph Johnson), OOPSLA'99 Workshop on the Implementation and Application of Object-Oriented Workflow Management Systems II, November 1999, Denver, CO, USA.

- "*Dynamic Object Model and Adaptive Workflow*" (co-author with Ralph Johnson), OOPSLA'99 Metadata and Active Object-Model Pattern Mining Workshop, November 1999, Denver, CO, USA.
- "*A Proposal for a Common Infrastructure for Process and Product Models*" (co-author with Ralph Johnson), OOPSLA'98 Mid-year Workshop on Applied Object Technology for Implementing Lifecycle Process and Product Models, July 1998, Denver, CO, USA.